Aspect-Oriented Thinking

An approach to bridging the disciplinary divides

A thesis submitted for the degree of
Doctor of Philosophy
of the
Australian National University

Shayne Russell Flint
July 2006

Typeset by TEX and LATEX

I declare that the work in this thesis is entirely my own and that to the best of my knowledge it does not contain any materials previously published or written by another person except where otherwise indicated.

Shayne Russell Flint
22 July 2006

Acknowledgements

I would like to start by thanking Clive Boughton, my supervisor, for his continuous friendship, guidance, support and interest throughout the long process of producing this thesis. He is the one that made it possible for me to satisfy a long held interest in pursuing an academic career. I am also grateful for the constructive feedback that Brian Molinari and Ramesh Sankaranarayana have provided throughout the process and particularly following their review of the draft thesis. These experienced *eyes* have helped me produce a greatly improved thesis.

During my candidature, many people have provided me with great ideas and problems to think about. I want to express particular thanks to Annette Vincent and Malte Stien for their technical interest in my work, and to Stephen Mellor for the books that changed the way I conceptualised software development, and for the few, but extremely valuable, conversations we have had about software development and my work.

During the early stages of this research Jennie Clothier and Marilyn Cross gave me the opportunity to work as part of an interdisciplinary team of psychologists, sociologists and computer scientists at the Australian Defence Science and Technology Organisation. This was a turning point in my research. Conversations with Jennie, Marilyn, Nicole Zambelli, Toby Keene and Derek Bopping enabled me to fully understand the vital need for engineering to be a truly interdisciplinary activity. Thanks for opening my eyes.

Special thanks also goes to Barry Newell, who has helped me develop a great appreciation for the way engineering should be practised and the responsibility we have in protecting the future of our planet. Without Barry's influence, I would still be 'looking in'. I hope that I will get the chance to work with Barry in the future and to learn as much as I can from him and his world.

My transition from industry to the academic environment has been difficult, daunting and stressful. I am therefore grateful for the research and teaching opportunities I have been given at the Australian National University. I am particularly thankful, again to my supervisor Clive, and to Chris Johnson for demonstrating confidence in my ability by employing me as a full time academic in the Department of Computer Science. I would also like to acknowledge the support and assistance I have received from other members of staff in the department. Lynette Johns-Boast with her knowledge, experience and energy, has enabled me to complete my teaching responsibilities while having enough time to also complete this thesis. Clem Baker-Finch, Henry Gardner, Malcolm Newey, Margaret Rossiter and Ramesh Sankaranarayana have all put up with my rantings and have always

provided invaluable advice and help whenever asked. For all of this support and friendship, I am truly thankful. Thanks to them all, I am now starting to feel at home in academia.

Closer to home, I would like to acknowledge the friendship that has formed between my family and that of Alex Zelinsky. Our wives are great mates, and this has allowed Alex and me to spend many hours discussing academic life, research, business, and all of the world's problems. He has shown me that an exciting and rewarding career can be had in research and has offerred me many pieces of valuable advice to that end.

Finally, I wish to thank my family for all of their support and tolerance during this most challenging period of my life. My parents have always been there to help me through tough times and to provide a place I can go to get away from the rat race. While my two children, Andy and Sara, are too young to fully understand why daddy hasn't been able to attend all of their concerts and sporting events or why I don't always go on holiday with them, they have provided me with a great deal of pleasure and cause for much reflection. I will now have much more time to be involved in these two lives. Underpinning all of this is the bedrock provided by my wife Sanae. She has always done whatever she can to ensure that I had the time and freedom to complete this thesis. Without her support, I would never have been able to complete this work. I love you all.

Abstract

Engineering is often described as the application of scientific and technical knowledge to solve problems. In this thesis, I support a more general view that engineering should be treated as a continuous process of learning and action that aims to make well understood improvements within dynamically complex environments of co-evolving social, man-made and natural systems. I argue that this can only be achieved by adopting an approach that systematically develops, manages and integrates the knowledge and expertise of many disciplines to conceive, develop, modify, operate and retire systems. A novel implementation of such an approach, called *Aspect-Oriented Thinking*, is presented.

Aspect-Oriented Thinking begins with the development and verification of a set of *Domain Models*. Each *Domain Model* represents knowledge about a separate, autonomous and possibly discipline specific concern or view within a given context. *Domain models* are developed by engineers, scientists, sociologists, psychologists, lawyers, philosophers, economists and others, using languages and techniques with which they are familiar. Knowledge captured in a set of *Domain Models* is then *woven* together, in accordance with a set of separately developed patterns and rules, to construct, modify, operate and retire systems, including models, hardware, software, processes and simulations. This is a continuous process which, in the first instance, involves those systems used to learn about a given context and to make decisions regarding required changes. Later, the process involves those systems used to implement and evaluate the impact of these decisions.

The significance of *Aspect-Oriented Thinking* lies in its broad applicability to any situation in which the expertise and knowledge of diverse disciplines is required to understand and make improvements within complex multifaceted environments such as those that involve sustainable development and national security.

A proof-of-concept within the context of software engineering is provided to demonstrate the mechanics and viability of *Aspect-Oriented Thinking*. The results of this demonstration are used to support an argument for future experimentation aimed at evaluating the effectiveness of *Aspect-Oriented Thinking* in a more general interdisciplinary environment.

Contents

Acknowledgements **v**

Abstract **vii**

Preface **xxi**

I Introduction **1**

1 Overview **3**
1.1 Introduction 4
1.2 Initial motivation and research aim 4
1.3 Thesis scope 5
1.4 Thesis structure 5
1.5 Publications 7
1.6 Summary of contribution 8

2 Background **11**
2.1 Introduction 13
2.2 The original research proposal 13
2.3 Preliminary research 15
2.4 Understanding the bigger picture 17
2.5 Capability Dynamics 20
2.6 eXecutable and Translatable UML 24
2.7 Aspect-Oriented Systems Engineering (AOSE) 27
2.8 Emergence of Aspect-Oriented Thinking 28
2.9 Summary of influences 29
2.10 Conclusion and conjecture 31

II Contribution **33**

3 Research design **35**
3.1 Introduction 36
3.2 A conceptual model of software engineering research 36
3.3 The Aspect-Oriented Thinking research approach 44
3.4 Conclusion 48

4 Aspect-Oriented Thinking Concepts **49**
4.1 Introduction . . . 51
4.2 Problem situations . . . 51
4.3 Systems . . . 52
4.4 Domains . . . 53
4.5 Domain Models . . . 61
4.6 Aspect-Oriented Specifications . . . 62
4.7 Aspect-Oriented Specification Archetypes . . . 67
4.8 Implementation Models . . . 69
4.9 Conclusion . . . 72

5 Aspect-Oriented Thinking Process **73**
5.1 Introduction . . . 74
5.2 Problem situation analysis . . . 75
5.3 Knowledge domain identification and analysis . . . 76
5.4 Aspect-Oriented Specification Archetype development . . . 79
5.5 Aspect-Oriented Specification development . . . 79
5.6 Implementation modelling . . . 80
5.7 Aspect-Oriented Specification implementation . . . 80
5.8 Continuous learning and improvement . . . 80
5.9 Comparison with traditional life-cycle models . . . 81
5.10 Documentation . . . 83
5.11 Conclusion . . . 84

III Proof of Concept and Conclusion **85**

6 Proof-of-Concept **87**
6.1 Introduction . . . 88
6.2 Problem situation analysis . . . 88
6.3 Domain identification and analysis . . . 92
6.4 Aspect-Oriented Specification Archetype development . . . 95
6.5 Aspect-Oriented Specification development . . . 96
6.6 Implementation modelling . . . 98
6.7 Implementation . . . 99
6.8 Conclusion . . . 99

7 Summary and Conclusion **101**
7.1 Introduction . . . 102
7.2 Summary of contribution . . . 102
7.3 Limitations of contribution . . . 104
7.4 Overall conclusion . . . 105

7.5 Recommendations for future work 106
7.6 Closing remarks 109

IV Appendices **111**

A Aspect-Oriented Specification notation **113**
A.1 Introduction 114
A.2 Domain model notation 114
A.3 Individual requirement notation 114
A.4 Requirement set notation 114
A.5 Domain model set notation 116
A.6 Conclusion 117

B Proof of concept - *Knowledge Domain Models* **119**
B.1 Introduction 120
B.2 *Contact Management* domain model 120
B.3 *Product Management* domain model 123
B.4 *Security* domain model 125
B.5 *User Interface* domain model 128
B.6 *LAMP Framework* domain model 134
B.7 $\frac{X}{T}$*UML* domain model 136
B.8 *Linux Operating System* domain model 136
B.9 *PHP Web Programming Language* domain model 136
B.10 *Structured Query Language* domain model 136
B.11 *APACHE Web Server* domain model 136
B.12 *MySQL Relational Database* domain model 136
B.13 Domain model verification 137
B.14 Conclusion 137

C Proof of concept - *LAMP Aspect-Oriented Specification Archetype* **139**
C.1 Introduction 141
C.2 User interface related requirement archetypes 142
C.3 Security related requirement archetypes 151
C.4 Database related requirement archetypes 152
C.5 Implementation specific requirement archetypes 153
C.6 *Security Mechanisms* domain model 155
C.7 Conclusion 155

D Proof of concept - *Product Management System Aspect-Oriented Specification* **157**
D.1 Introduction 159

D.2 PMS persistence requirements . . . 161
D.3 *PMS Security* domain model . . . 164
D.4 *PMS Security* requirements . . . 165
D.5 *PMS User Interface* domain model . . . 166
D.6 *PMS User Interface* requirements . . . 174
D.7 Conclusion . . . 186

E Proof of concept - *LAMP Implementation model* **187**
E.1 Introduction . . . 189
E.2 General implementation rules . . . 190
E.3 User interface implementation rules . . . 190
E.4 Database implementation rules . . . 226
E.5 Security implementation rules . . . 237
E.6 Conclusion . . . 239

F Proof of concept - *Annotated Source Code* **241**
F.1 Introduction . . . 244
F.2 Application configuration . . . 244
F.3 Database . . . 247
F.4 Application startup . . . 249
F.5 Authentication . . . 249
F.6 Main menu . . . 253
F.7 Product management . . . 255
F.8 Product category management . . . 268
F.9 Contact management . . . 274
F.10 User management . . . 293
F.11 Conclusion . . . 299

G Proof of concept - *LAMP Mechanisms* **301**
G.1 Introduction . . . 302
G.2 *Application Mechanisms* domain model . . . 302
G.3 *Database Mechanisms* domain model . . . 303
G.4 *User Interface Mechanisms* domain model . . . 304
G.5 *Security Mechanisms* domain model . . . 312

H Reproduced paper: *Capability Dynamics: An approach to Capability Planning and Development in Large Organisations* **315**
H.1 Introduction . . . 317
H.2 Definitions . . . 318
H.3 Capability dynamics models . . . 321
H.4 Distributed capability dynamics modelling tool . . . 327
H.5 The decision control process . . . 327

H.6 Proposed Experimentation 330
H.7 Conclusions 333

I Reproduced paper: *Simplified Development Of Web Applications With Armidale* **335**
I.1 Background 338
I.2 Requirements, Constraints and Design Approaches 340
I.3 Implementation 349
I.4 Functional Testing 349
I.5 Effectiveness 350
I.6 Some Design Details 351
I.7 Summary 355

J Reproduced paper: *eXecutable and Translatable UML and Systems Engineering* **357**
J.1 Background 359
J.2 Executable and Translatable UML: Key Concepts 362
J.3 ${}^{X}_{T}UML$ and Systems Engineering 368
J.4 The benefits of ${}^{X}_{T}UML$ 371
J.5 Further Work 373
J.6 Conclusions 373

Bibliography **375**

Proof-of-Concept Index **385**

Author Index **387**

List of Figures

1.1 Activity diagram depicting the structure and the flow of ideas and results throughout this thesis. 6

2.1 An activity diagram depicting the organisation of this chapter and the flow of ideas that led to the notion of *Aspect-Oriented Thinking*. . 14
2.2 An example of the way in which systems are considered by *Capability Dynamics*. 22
2.3 The *Aspect-Oriented Thinking* concept. 30

3.1 Class diagram depicting the proposed conceptual model of software engineering research 37
3.2 Activity diagram depicting the idealistic life-cycle common to all of the research methods described in Section 3.2.1. 39
3.3 Activity diagram depicting the 'Analytical Advocacy Research' life-cycle [*Fenton et al.*, 1994]. 40
3.4 Activity diagram depicting the 'Research-then-Transfer' life-cycle [*Potts*, 1993]. 41
3.5 Activity diagram depicting the 'Industry-as-Laboratory' life-cycle *Potts* [1993]. 43
3.6 Activity Diagram depicting the *Aspect-Oriented Thinking* research approach. 45

4.1 Class diagram depicting the major concepts involved in *Aspect-Oriented Thinking*. More detailed concepts are depicted in other class diagrams throughout this chapter. 52
4.2 *Aspect-Oriented Thinking* domains are organised into the domain categories depicted in this class diagram. 54
4.3 Class diagram depicting the concepts involved in *Aspect-Oriented Specifications*. 63
4.4 Example of a *Requirement* that references a data store within a *Data Flow Diagram* being used as a *Domain Model*, and the *Relational Database Management System* domain model. 64
4.5 Examples of *Requirements* represented using the notation described in Appendix A. 65
4.6 Example of a *Requirement* that references a system in a *Capability Dynamics* model and an *Aspect-Oriented Specification*. 66
4.7 Class diagram depicting the concepts involved in *Aspect-Oriented Specification Archetypes*. 68

4.8 The *Requirement* depicted in Figure 4.4 and the *Requirement Archetype* with which it conforms. 70
4.9 Class diagram depicting the concepts involved in *Implementation Models*. 71

5.1 An activity diagram depicting the process of *Aspect-Oriented Thinking*. 74
5.2 An activity diagram depicting the process of *Aspect-Oriented Thinking* and how the activities involved correspond to traditional systems engineering life-cycle activities of *Requirements Analysis, Architectural Design, Implementation, Verification* & *Validation*. 82

6.1 The Block Diagram Meta-Model. 94
6.2 Package diagram depicting the *Aspect-Oriented Specification* for the *Product Management System*. 97

A.1 Examples of the simplified notation that can be used to depict individual *Aspect-Oriented Specification Requirements*. 115
A.2 Examples of the notation that can be used to represent *Requirement Sets*. 116
A.3 An *Aspect-Oriented Specification* depicting *Requirement Sets* involved in a simple software application. 116
A.4 The *Aspect-Oriented Specification* depicted in Figure A.3, simplified using the domain set notation. 117

B.1 Class diagram depicting concepts involved in the *Contact Management* domain. 120
B.2 Class diagram depicting concepts involved in the *Product Management* domain. 123
B.3 Class diagram depicting concepts involved in the *Security* domain. . 126
B.4 Class diagram depicting concepts involved in the *User Interface* domain. 129
B.5 Block diagram depicting elements of the *LAMP Framework* domain. 135

C.1 Application of the *raLinkAvailableWithRole* requirement archetype. *Rq01* is a requirement for *theLink* to be only visible if the current user has *theRole*. 143
C.2 Application of the *raClassList* requirement archetype. *Rq01* is a requirement for instances of the *Employee* class (employees) to be listed on *theCrudList*. 145

C.3 Application of the *raGroupClassList* requirement archetype. *Rq01* is a requirement for instances of the *Employee* class (employees) to be listed on *theCrudList*. *Rq02* is a requirement for the listed employees to be grouped by the name of the company to which they belong. . . . 146
C.4 Application of the *raTextInputUpdatesAttribute* requirement archetype. *Rq01* is a requirement for instances of the *Employee* class (employees) to be listed on *theCrudList*. *Requirement Rq02* specifies a requirement for the *Employee.name* attribute to be updated from data entered into *theTextInput* on *Employee Edit Page*. 148
C.5 Application of the *raSelectItemCreatesLink* requirement archetype. *Rq01* is a requirement for instances of the *Employee* class (employees) to be listed on *theCrudList*. *Requirement Rq02* specifies a requirement for the creation of a link between an employee and a company selected using *theSelectItem* on the *Employee Edit Page*. . . 150

D.1 Package diagram depicting the *Aspect-Oriented Specification* of the *Product Management System*. 160
D.2 Page layout notation. See Section B.5 for descriptions of each user interface page element 167
D.3 The *loginPage* layout. 168
D.4 The *registerNewUserPage* layout. 168
D.5 The *registrationSuccessPage* layout. 169
D.6 The *mainMenuPage* layout. 169
D.7 The *productListPage* layout. 170
D.8 The *editProductPage* layout. 171
D.9 The *editProductVariantPage* layout. 171
D.10 The *productCategoryListPage* layout. 171
D.11 The *editProductCategoryPage* layout. 172
D.12 The *contactListPage* layout. 172
D.13 The *editContactPage* layout. 173
D.14 The *editContactRelationshipPage* layout. 174
D.15 The *editNotePage* layout. 174
D.16 The *registeredUserListPage* layout. 175
D.17 The *editRegisteredUserPage* layout. 175
D.18 The *errorPage* layout. 175
D.19 The *fatalErrorPage* layout. 176

E.1 Block diagram depicting the organisation of PHP modules required to implement Forms. 202
E.2 Block diagram depicting the organisation of PHP modules required to implement Class Lists. 207

H.1 A report writing system. 320
H.2 Capability Development. 323
H.3 Composite Systems 323
H.4 Non-Hierarchical Composite Systems 324
H.5 Decision control process 328
H.6 Experimental software 331

I.1 The Armidale conceptual design 341
I.2 Some Armidale applications running on a KDE desktop 343
I.3 The Armidale Launcher 344
I.4 Generation of Armidale GUI element source code 346
I.5 An Armidale application running in a stand-alone context 352
I.6 An Armidale application running in a client-server context 353

J.1 Primary MDA models and transformation 360
J.2 An Annotated Software Domain Chart 365
J.3 An Annotated Application Domain Chart 369
J.4 An Annotated Hardware Domain Chart 369
J.5 An Annotated Process, Training and Human Resources Domain Chart 370

List of Tables

6.1 *Proof-of-Concept* knowledge domains and meta-models 95

D.1 *Product Management System* security specification 164

I.1 Armidale Test Configurations 350

Preface

The research reported in this thesis started when, after 15 years experience in the software and systems engineering industry, circumstances allowed me to take an extended break from work. With support from Dr Clive Boughton, my current supervisor, I decided to use this break to undertake a post-graduate research project that would aim to improve, in some way, software and systems engineering practice.

After enrolling in a PhD program at the Australian National University I was able to embark on a long and broad exploration of what could be done to improve the ability of software-intensive systems projects to deliver the capabilities expected by stakeholders. My investigations started with the widely held view that problems related to requirements are a major cause of project failure. I studied the requirements engineering literature covering topics such as specification, modelling, measurement and capability maturity, and concluded that requirements engineering process improvement may provide a way to improve software and systems engineering practice. While considering what it meant to *improve* a process, I explored systems theory, system dynamics, decision making, organisational learning and change, socio-technical systems and ethics. I also revisited my early career as a junior engineer and military officer to reflect on success and failure within the complex socio-technical environment of modern defence.

From these explorations it became clear that industrial practice could be improved by developing and using software and systems engineering approaches that help ensure the development of capability that has well understood impacts (both positive and negative) throughout a defined environment and over time.

As these ideas formed, I was offered and accepted a position at the Australian Defence Science and Technology Organisation (DSTO). Whilst at DSTO I worked in a multi-disciplinary group headed by a linguist and comprising social scientists, psychologists, computer scientists and myself - an engineer. From this experience I realised that a multi-disciplinary team was required to fully understand the broader impact of existing, new and modified systems. Any software or systems engineering approach that focused on the development of capability with known impacts in time and space would therefore need to be interdisciplinary in nature.

It was from this milieu of exploration and experience that *Aspect-Oriented Thinking* emerged.

Shayne R. Flint
Canberra, Australia

Part

I

Introduction

Chapter 1

Overview

> One wonders sometimes if science will not grind to a stop in an assemblage of walled-in hermits, each mumbling to himself words in a private language that only he can understand. The spread of specialized deafness means that someone who ought to know something that someone else knows isn't able to find it out for lack of generalized ears.
>
> *Boulding* [1956]

Contents

1.1 Introduction **4**

1.2 Initial motivation and research aim **4**

1.3 Thesis scope **5**

1.4 Thesis structure **5**

1.4.1 Part I - Introduction 5

1.4.2 Part II - Contribution 7

1.4.3 Part III - Discussion and Conclusion 7

1.5 Publications **7**

1.6 Summary of contribution **8**

1.1 Introduction

This thesis is submitted for the degree of Doctor of Philosophy of the Australian National University. In it I describe exploratory research that has resulted in the development and demonstration of *Aspect-Oriented Thinking*, a novel interdisciplinary approach to software engineering which has applicability to engineering in general.

In this introductory chapter, I provide an overview of the research along with information intended to help the reader navigate this thesis. I conclude the chapter with a summary of work published during the project and contributions made to engineering research and practice.

1.2 Initial motivation and research aim

The initial motivation for conducting the research reported in this thesis can be summed up by Glass' Law '*Requirements deficiencies are the prime cause of project failures*' [*Endres and Rombach*, 2003, Law L1, pp16-17]. This led to an initial aim to find ways of improving software requirements development and management practices.

Preliminary research reported in Chapter 2 has shown that requirements practices can be improved by focussing on the development and effectiveness of systems that can be used to learn about and improve *Problem Situations* which emerge within a dynamically complex environment of interacting social, man-made and natural systems. Further research reported in Chapter 2 has shown that this would be best achieved using an interdisciplinary approach to system development and operation, and that the development of such an approach, based on *aspect-oriented modelling* concepts, is feasible. The preliminary research concludes by presenting the concept of *Aspect-Oriented Thinking*, an approach that can be used by engineers, scientists, sociologists, psychologists, lawyers, philosophers, economists and others to 'bridge the disciplinary divides' [*Grafton et al.*, 2005] and work together to build, operate and evolve the systems needed to both understand and improve *Problem Situations* of all kinds.

This preliminary work has resulted in a research aim *to develop, model, demonstrate and evaluate an aspect-oriented interdisciplinary approach to engineering that can be used to build and operate systems required to understand and improve Problem Situations of all kinds*.

1.3 Thesis scope

The scope of this thesis is the development, modelling and demonstration of *Aspect-Oriented Thinking* within the context of software engineering. While the broader applicability of *Aspect-Oriented Thinking* is discussed, evaluation of the approach within a more general context is beyond the scope of this thesis.

Experimentation aimed at understanding the value of *Aspect-Oriented Thinking* in an industrial context is discussed in Sections 3.3.2.2 and 7.5.4. The conduct of such experimentation is beyond the scope of this thesis.

While some ideas regarding appropriate *Aspect-Oriented Thinking* tool support are introduced in Section 7.5.2, and an implementation of an appropriate architectural framework is described in *Flint and Boughton* [2002, reproduced in Appendix I], the full analysis, design and implementation of a complete *Aspect-Oriented Thinking* toolset is beyond the scope of this thesis.

1.4 Thesis structure

The structure of this thesis is depicted in the Unified Modelling Language (UML) activity diagram [*Object Management Group*, 2005, pp 285-406] shown in Figure 1.1. Each activity (grey rounded box) represents a chapter of this thesis, and directed arrows between them represent the flow of ideas and research results. Objects on the diagram (white square boxes) represent key contributions made by the research.

This thesis is organised in three parts comprising two or more chapters as represented by the vertical partitions of the activity diagram shown in Figure 1.1. The contents of each part and chapter of this thesis are outlined below. A more detailed overview of the research can be obtained by reading the preface, Chapter 2, the introduction and conclusion of Chapters 3 to 6, and the overall conclusion provided in Chapter 7. The impatient reader can gain an overview of the *Aspect-Oriented Thinking* approach itself by reading Chapter 5.

1.4.1 Part I - Introduction

Part I of this thesis comprises 2 chapters. Chapter 1 is the current chapter in which I state the motivation and aim for the research reported in this thesis. Within Chapter 2, I describe the context within which *Aspect-Oriented Thinking* emerged. This includes a description of preliminary research which began with an

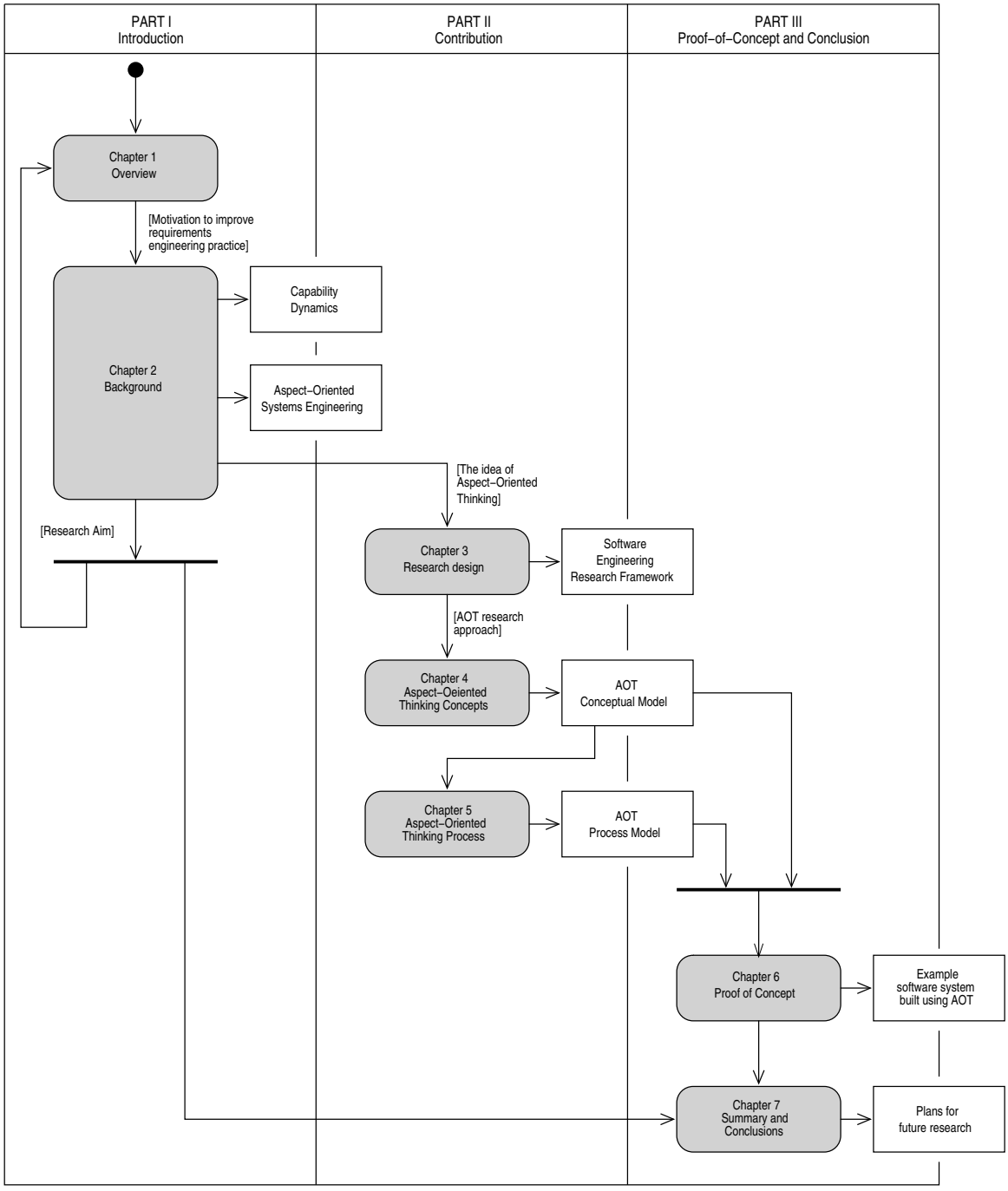

Figure 1.1: Activity diagram depicting the structure and the flow of ideas and results throughout this thesis.

exploration of requirements engineering, model-based process improvement and agile development in the hope of finding approaches and practices that could be used to deal with the original software requirements problem stated in Section 1.2. This research led to the development of *Capability Dynamics* [*Flint*, 2001]

and *Aspect-Oriented Systems Engineering* [*Flint and Boughton*, 2003] as depicted in Figure 1.1. It was from this work and reflection upon the notions of *systems thinking*, *problem situations* [*Checkland*, 1981] and *wicked problems* [*Rittel and Webber*, 1973], that the idea of *Aspect-Oriented Thinking* emerged.

1.4.2 Part II - Contribution

Part II of this thesis comprises three chapters. Within Chapter 3, I propose a novel conceptual model of software engineering research. I then use this model to describe and discuss existing approaches to software engineering research and, to propose a research approach for the development and evaluation of *Aspect-Oriented Thinking*. This proposed approach has been designed to reduce stakeholder risk by only entering *evaluative* phases of research after the potential viability of *Aspect-Oriented Thinking* has been demonstrated within a research context.

Within Chapters 4 and 5, I describe the *Aspect-Oriented Thinking* approach itself. Chapter 4 contains a model of the concepts involved in *Aspect-Oriented Thinking* and the artifacts that are developed and manipulated to construct, modify, operate and retire systems. Examples from the software engineering domain are included to help explain the concepts involved. Within Chapter 5, I describe the continuous process of *Aspect-Oriented Thinking* and how it can be used to learn about and improve *Problem Situations*.

1.4.3 Part III - Discussion and Conclusion

Part III of this thesis comprises two chapters. Within Chapter 6 and associated Appendices B to G, I present a detailed *proof-of-concept* to demonstrate the application and viability of *Aspect-Oriented Thinking*. This *proof-of-concept* comprises the development of a complete software application from an initial set of user requirements through to working software.

Within Chapter 7, I summarise the work presented in this thesis and discuss its limitations. I then offer a final conclusion and proposals for further research, development and evaluation of *Aspect-Oriented Thinking*.

1.5 Publications

The contributions made by this thesis are based on the results of preliminary research presented in Chapter 2 and published in the following refereed conference papers.

- Flint, S. (1999), *A model which helps control change in dynamically complex purposeful systems*, Proceedings of the 1999 Systems Engineering, Test & Evaluation Conference (SETE99), Adelaide, Australia, October 1999.
- Flint, S. (2001), *Capability Dynamics: An Approach to Capability Planning and Development in Large Organisations*, proceedings of the 2001 International Council on Systems Engineering (INCOSE) symposium, Melbourne, Australia, July 2001. Reproduced in Appendix H of this thesis.
- Flint, S. and Boughton, C. (2002), *Simplified Development of Web Applications using Armidale*, proceedings of the 2002 AusWeb conference, Sunshine Coast, Australia, July 2002. Also available from `http://ausweb.scu.edu.au`. Reproduced in Appendix I of this thesis.
- Flint, S. and Boughton, C. (2003), *Executable/Translatable UML and Systems Engineering*, proceedings of the 2003 Systems Engineering, Test & Evaluation Conference (SETE2003), Canberra, Australia. Reproduced in Appendix J of this thesis.

1.6 Summary of contribution

The research reported in this thesis has resulted in the following contributions to existing knowledge required to understand and make improvements within dynamically complex *Problem Situations*.

- **Aspect-Oriented Thinking.** *Aspect-Oriented Thinking* is a novel interdisciplinary engineering approach that can be used to build and operate systems that have a well understood impact on their environment over time. The approach can be used within the context of software development and systems engineering as well as broader interdisciplinary areas such as sustainable development, the environment and society.
- **Capability Dynamics.** *Capability Dynamics* [*Flint*, 2001, reproduced in Appendix H] is a novel approach to understanding where improvements can be made in dynamically complex environments and the effectiveness over time of any changes that aim to make such improvements. *Capability Dynamics*, along with other approaches such as *System Dynamics* [*Forrester*, 1961] and *Soft Systems Methodology* [*Checkland*, 1981] can be used within the *Aspect-Oriented Thinking* framework.

Other contributions made by this thesis include:

- **Conceptual model of software engineering research.** A conceptual model of software engineering research is proposed. The model can be used to describe approaches discussed in the literature as well as those designed to satisfy the needs of specific research programs. The research approach adopted in this thesis has been described using the proposed model.

- **Implementation of advanced $\frac{X}{T}$UML concepts.** Key concepts developed for *Aspect-Oriented Thinking* go some way to filling a gap in knowledge surrounding the use of eXecutable and Translatable UML ($\frac{X}{T}$UML) [*Mellor and Balcer*, 2002]. For example, it is not clear from the literature how the concept of an *Implicit Bridge* between domains can be implemented. It is also unclear how the more recent concepts of model transformation described by *Mellor et al.* [2004] can be implemented to build operational software from a diverse range of domains covering subjects such as user interfaces, security, persistence and communication.

  Aspect-Oriented Thinking represents one way of implementing these advanced concepts of $\frac{X}{T}$UML and model-driven development.

- **Armidale application development framework.** Armidale [*Flint and Boughton*, 2002] is a software development framework that radically simplifies the development of web applications by abstracting away (separating) all concerns regarding the internet. This framework was developed to simplify the development of distributed *Aspect-Oriented Thinking* tools as part of future research.

Chapter 2

Background

When solving problems, dig at the roots instead of just hacking at the leaves.

Anthony J. D'Angelo

Contents

2.1 Introduction 13
2.2 The original research proposal 13
2.3 Preliminary research 15
2.3.1 Requirements engineering 15
2.3.2 Software process improvement 16
2.3.3 Agile software development 17
2.4 Understanding the bigger picture 17
2.4.1 General systems 19
2.4.2 Problem situations 19
2.4.3 Systems Thinking 20
2.5 Capability Dynamics 20
2.5.1 Capability Dynamics Models 22
2.5.2 Limitations of Capability Dynamics 23
2.6 eXecutable and Translatable UML 24
2.6.1 Complexity management 25
2.6.2 Waste reduction 26
2.6.3 Productivity improvement 26
2.7 Aspect-Oriented Systems Engineering (AOSE) 27
2.8 Emergence of Aspect-Oriented Thinking 28
2.9 Summary of influences 29

2.10 Conclusion and conjecture 31

2.1 Introduction

The *Aspect-Oriented Thinking* approach emerged within a context of study, practice and reflection across a wide range of subjects including software and systems engineering, systems thinking, soft systems, organisational change and learning, system dynamics, simulation and advanced approaches to software development. In this chapter I discuss the path I took through this milieu and how it led to the notion of *Aspect-Oriented Thinking* and development of preliminary approaches.

The organisation of this chapter is summarised in the UML activity diagram shown in Figure 2.1. The partition on the left of the diagram represents preliminary research undertaken during early stages of the project. The centre partition represents the reporting of preliminary research results including the two conference papers reproduced in Appendices H and I. The final partition covers the emergence of *Aspect-Oriented Thinking* and a conjecture that provides a foundation for the remainder of this thesis.

2.2 The original research proposal

> Requirements deficiencies are the prime cause of project failures.
>
> Glass' Law [*Endres and Rombach*, 2003, Law L1, pp16-17]

The original motivation for the research reported in this thesis was personal industrial experience with the negative impact that poor requirements practices have on the success of software and systems engineering projects. In addition to this personal experience, much had been published at the time (late 1990's) to show that poor requirements practices play a key role in software project failure [*The Standish Group*, 1994, 1995; *Jones*, 1996].

In his influential 'No Silver Bullet' article *Brooks* [1987] sums up the requirements problem with his statement that:

> 'The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.'

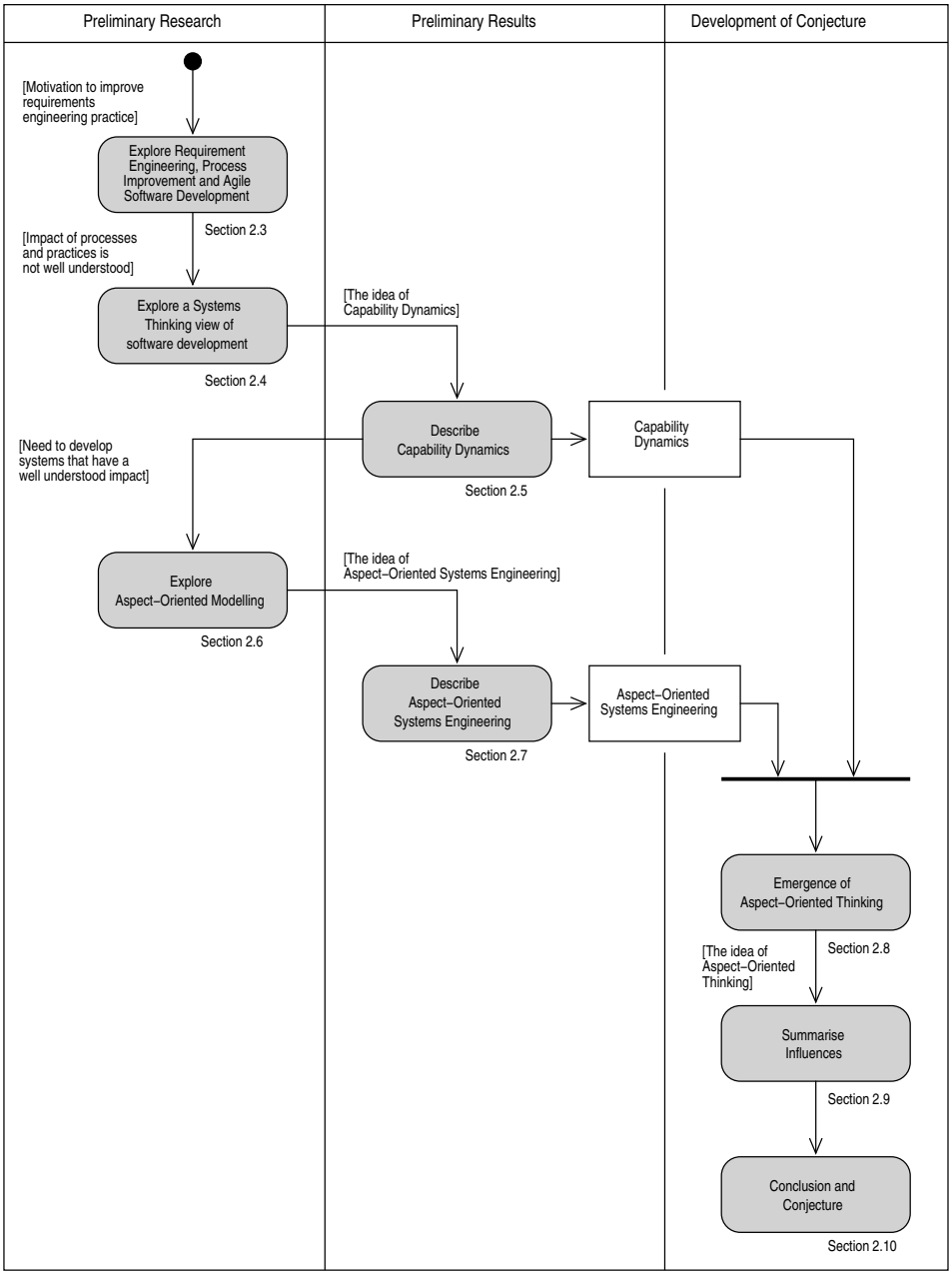

Figure 2.1: An activity diagram depicting the organisation of this chapter and the flow of ideas that led to the notion of *Aspect-Oriented Thinking*.

More recent work by *The British Computer Society and Royal Academy of Engineering* [2004] confirms this view by reporting that poor requirements practices remain a key cause of software project failure today.

Based on this experience and published work, an initial research proposal was made to study the way in which the effectiveness of requirements practices could be measured and continuously improved.

The remainder of this chapter describes the path that led from this initial proposal to the development of *Aspect-Oriented Thinking*, an approach that can be used to improve requirements practices as well as address more general and complex problems such as those facing society and the environment.

2.3 Preliminary research

In order to understand the state of requirements research and practice, the active subject areas of Requirements Engineering and Model-Based Software Process Improvement were investigated. More recently, the subject of Agile Software Development was investigated.

2.3.1 Requirements engineering

> Requirements engineering is a relatively new term which has been invented to cover all of the activities involved in discovering, documenting, and maintaining a set of requirements for a computer based system. The use of the term 'engineering' implies that systematic and repeatable techniques should be used to ensure that system requirements are complete, consistent, relevant etc.
>
> *Kotonya and Sommerville* [1998, p8]

In response to problems caused by poor requirements practices, industry and researchers have adopted an engineering approach to the elicitation, analysis, specification, validation and management of requirements. The field of *Requirements Engineering* has therefore become an active area of research that aims to improve software and systems engineering project success by developing improved requirements development and management practices.

The results of this effort are wide ranging and many of the developed practices have been incorporated into various *practice guides* [*Sommerville and Sawyer*, 1997; *Kotonya and Sommerville*, 1998; *Young*, 2001; *Wiegers*, 2003]. However, despite the progress being made by this work, many of the practices are presented with little evidence as to their effectiveness or sensitivity to context. This has been expounded on most recently by *Davis and Zowghi* [2006].

It was the observation regarding a lack of evidence that led me to explore Process Improvement in the hope of finding techniques that might be used to understand and improve the effectiveness of requirements engineering practices.

2.3.2 Software process improvement

> The software process is the set of tools, methods, and practices we use to produce a software product. The objectives of software process management are to produce a software product according to plan while simultaneously improving the organisation's capability to produce better products.
>
> *Humphrey* [1989]

Software process improvement is a very active research area and is practiced widely in industry, particularly by those involved in the development of critical and large scale *software-intensive systems*. The underlying belief behind such approaches is that organisations can improve the quality of their products by assessing and improving the processes they use to develop and support those products.

Most process improvement activities centre around the various *Capability Maturity Models* developed by the Software Engineering Institute at Carnegie Mellon University [*Humphrey*, 1989; *Paulk et al.*, 1993; *CMMI Product Development Team*, 2000; *Ahern et al.*, 2004] and the *Software Process Improvement and Capability dEtermination* (SPICE) framework and associated standards [*International Organization for Standardization*, 1997, 2002, 2005a].

While there is extensive evidence that the adoption of process improvement strategies can lead to reductions in defects, improved schedule and cost control, as well as increased productivity [*Diaz and Sligo*, 1997; *Ferguson et al.*, 1997, 1999; *Young*, 2001; *Capell*, 2004; *Subramanyam et al.*, 2004; *Nichols and Connaughton*, 2005], there is little reporting of process improvement failures. In addition, there is no real understanding of how particular processes or variations in process performance impact project success, or even whether the schedules and budgets met by a *successful* project are optimal. It might, for example, be the use of *good* management and people that has led to the reported project success. There is some evidence to support this hypothesis in that project management has been identified as a key factor in the success of software projects by both *The Standish Group* [2001] and *The British Computer Society and Royal Academy of Engineering* [2004].

2.3.3 Agile software development

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - **Individuals and interactions** over processes and tools
> - **Working software** over comprehensive documentation
> - **Customer collaboration** over contract negotiation
> - **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.
>
> The Agile Manifesto [*Agile Alliance*, 2001]

Agile approaches have been widely adopted throughout the software development industry and represent an active area of research. They offer an alternative to model-based process improvement by advocating the use of practices that respond to change, deliver functioning software using rapid iterative development lifecycles and promote close customer involvement, rather than advocating the definition, execution and continuous improvement of a set of processes identified by an external authority.

Many agile methods have emerged [*Beck*, 1999; *Highsmith*, 1999; *Cockburn*, 2002; *Schwaber and Beedle*, 2002] and there is evidence that their use can result in improved productivity, reduced cost and schedule, as well as reduced defect rates [*Reifer*, 2002]. However, like model-based software process improvement, there is no real understanding of how particular agile practices impact project success. Again, it might be the use of *good* people that has led to project success. Agile project *success* might also be due to a form of *Stockholm Syndrome* in which client representatives bond and sympathise with the developers more than with their organisation and its real needs. This could slowly lead an entire project off track while all involved believe valuable software is being produced.

2.4 Understanding the bigger picture

> When the performances of the parts of a system, considered separately, are improved, the performance of the whole may not be (and usually is not) improved.
>
> *Ackoff* [1999, p9]

Evidence available to support the adoption of either model-based process improvement or agile practices (Sections 2.3.2 and 2.3.3) relates to their ability to help organisations deliver software on time, within budget and with low defect densities. In some cases *customer satisfaction* is also considered using techniques such as *The Balanced Scorecard* [*Kaplan and Norton*, 1996]. However, as *Ferguson et al.* [1999] show, customers too are sometimes only interested in cost, delivery time and delivered defects, rather than any value they derive from using the delivered software. None of this is surprising given the widespread acceptance of such measures of project success [*The Standish Group*, 1999; *Sauer and Cuthbertson*, 2003; *The British Computer Society and Royal Academy of Engineering*, 2004].

I contend that this singular focus on cost, schedule and defect density is often inappropriate as it only deals with a narrow set of stakeholder concerns. In order to form a complete picture of project success or failure, it is necessary to consider the evolving concerns of all stakeholders. Of particular importance are the concerns of end users and the benefits they are able to derive from using software and other artifacts produced by a project. The importance of taking such an approach is highlighted by *Boehm and Turner* [2004, p8] who note that 'the initially successful Chrysler Comprehensive Compensation (C3) project that gave birth to XP as a remedy for failed traditional methods was eventually cancelled' and that 'Cleanroom, one of the most successful and empirically studied of the formal, process-based methods, has never really crossed Moore's chasm into the mainstream'. That is, neither of these projects produced artifacts that were of any value to end users and organisations despite being *successful* according to popular measures of cost, schedule and defect density, and despite their adoption of particular practices.

At the other extreme, the Windows® series of operating systems [*Microsoft*, 2006c] has been a clear success for Microsoft despite frequent schedule overruns during their development and a less than perfect record in relation to defects and security.

So, while it is appropriate to do whatever we can to reduce the cost, schedule and defect density associated with the development of all software systems, we should only do so *after* we have done what we can to ensure that the likely impacts of developing, maintaining and using such systems are well understood and considered *beneficial* by all stakeholders involved. To achieve this, it will be necessary to understand the wider context within which systems are developed and operated.

2.4.1 General systems

> No part of a system should be changed without understanding its effect on the whole and determining that this effect is beneficial.
>
> *Ackoff* [1999, p9]

While the above discussion centres around the development of software systems, it equally applies to systems of other kinds. For example, the development, modification, operation and retirement of hardware, processes, laws and organisational structures should be conducted with a full understanding of the dynamic impact that such activities are likely to have on the systems of which they are a part.

It was at this point that the scope of research reported in this thesis was increased to deal with the development of *all* kinds of system.

What is a system?

This thesis adopts Jordan's view that 'the only things that need be common to all systems are identifiable entities and identifiable connections between them. In all other ways systems can vary unlimitedly. The quest for a more detailed, specific definition for "system" is chimerical' [*Jordan*, 1968, pp 64-65]. Accordingly, the following definition of *system* is used in this thesis:

A system is a set of identifiable entities and identifiable connections between them.

2.4.2 Problem situations

> **Problem Situation:** A nexus of real-world events and ideas which at least one person perceives as problematic: for him other possibilities concerning the situation are worth investigating.
>
> *Checkland* [1981, p316]

In order to fully understand the likely impact of a new system, it is first necessary to accept that the development, modification, operation and retirement of systems is conducted within an environment of co-evolving human, man-made and natural systems. Changes made in one part of such an environment will usually lead to further changes which are often uncoordinated and may occur at

different times and places. This results in a dynamically complex [*Casti*, 1979] environment in which the impact of a change is difficult to predict and therefore control.

For the purposes of this thesis, I will refer to such environments as *Problem Situations* [*Checkland*, 1981, 1989] which comprise a set of systems that behave and interact in ways that are considered unsatisfactory by at least one of the humans involved. Other terms commonly used to refer to such environments include *Messes* [*Ackoff*, 1999, p13] and *Wicked Problems* [*Rittel and Webber*, 1973].

Because of their complex nature, *Problem Situations* are difficult to eliminate. Instead they are best dealt with using a continuous process of learning and change aimed at improving acceptance of the situation by the humans involved.

2.4.3 Systems Thinking

Systems thinking [*Churchman*, 1968b; *Kramer*, 1977; *Checkland*, 1981; *Senge*, 1992] refers to a set of approaches that can be used to learn about and make decisions regarding changes to dynamically complex systems, such as *Problem Situations*. They are distinguished from other approaches by their focus on the whole and the study of interactions among the parts of a system, rather than the parts themselves. It is by studying these interactions that we can understand emergent properties of an entire system including the impact of a change on the whole.

If we are to make and implement appropriate decisions regarding the improvement of *Problem Situations*, it will be necessary to adopt a *Systems Thinking* approach that helps people *learn* about the systems involved, how they interact, and how changes are likely to impact the situation as a whole and over time.

2.5 Capability Dynamics

> The performance of any system has two dimensions: the *efficiency* with which it does whatever it does (doing things right) and the *effectiveness* of what it does (doing the right thing, its value). These things should be taken together because *The righter we do the wrong thing, the wronger we become*.
>
> *Ackoff* [1999, p10]

Capability Dynamics is a proposed systems thinking approach which can

help people learn about a *Problem Situation* and make decisions regarding the development, operation and retirement of systems to improve it.

An early paper presenting the initial idea of *Capability Dynamics* [*Flint*, 2001] is reproduced at Appendix H. While the idea, as presented in that paper, needs further research, particularly in relation to demonstrating the effectiveness of the approach, the concept is to consider a *Problem Situation* as a set of interacting systems that each develop a set of *capabilities*. These capabilities are intended to satisfy the evolving *requirements* of other systems so that they can, in turn, develop and deliver their own capabilities. The internal operation of each system is not considered within the *Capability Dynamics* approach.

A *Problem Situation* exists, at a given point in time, when some of these *Requirements* are not properly satisfied. As systems are developed, modified, operated and retired, and capabilities develop, *Problem Situations* will, themselves, evolve.

Figure 2.2 shows an example which depicts the way in which a *Problem Situation* is considered in *Capability Dynamics*. The icons with titles such as *Reporting Process* represent systems. The arrows between systems represent the development and delivery of capability by one system for another. Labels on the arrows indicate the nature of the capability provided. Capabilities flowing into a system are used by that system to develop and deliver new capabilities. The *Reporter* system, for example, makes use of capabilities from several systems including the *Reporting Process* and *Computing System* to deliver the ability to produce reports in response to requirements of the *Manager*.

The ability of each system to satisfy the requirements of other systems is evaluated against Measures of Effectiveness (MOEs) [*Sproles*, 2000]. That is, the *Capability Dynamics* approach focuses on the ability of systems to *do the right thing* [*Ackoff*, 1999, p10] with respect to their environment rather than their individual performance.

Note that this arrangement forms a network of dependencies and that decisions are being made and implemented throughout the *Problem Situation* resulting in continuously changing requirements, capability and measures of effectiveness. These changes are often uncoordinated and occur at different tempos resulting in a dynamically complex [*Casti*, 1979] system in which the impact of a change is difficult to predict and therefore control.

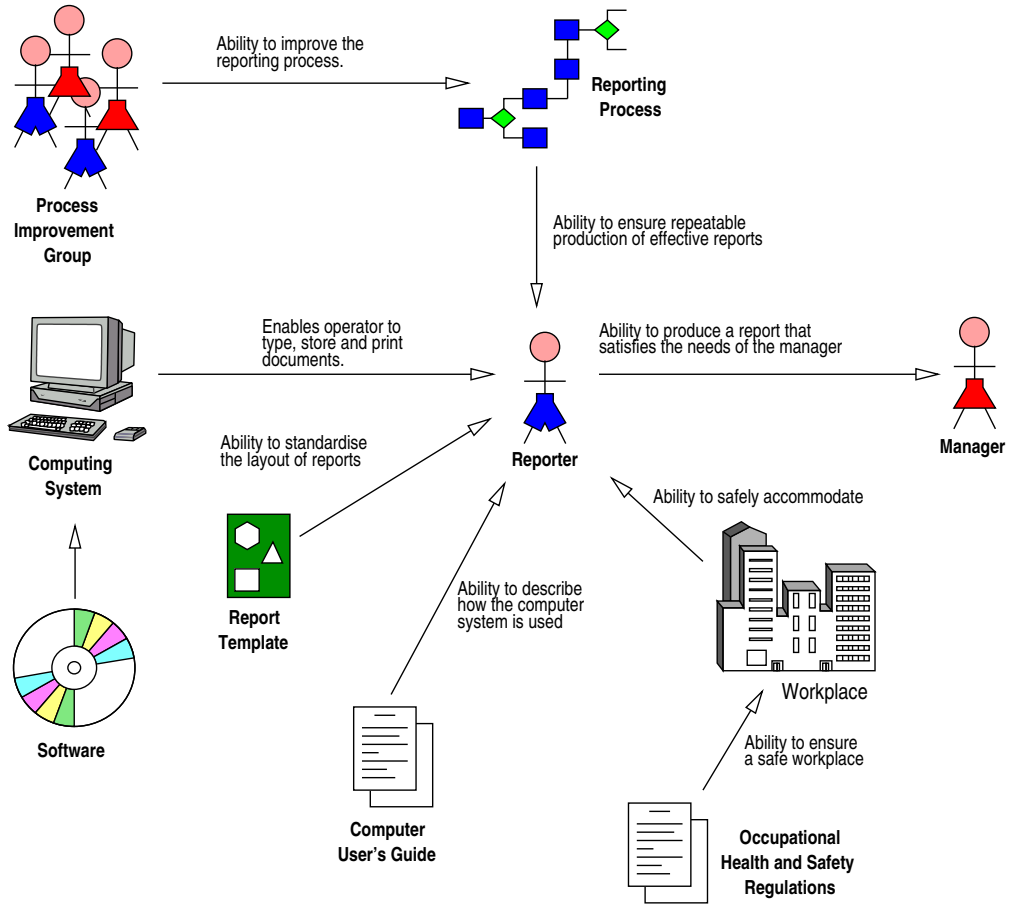

Figure 2.2: An example of the way in which systems are considered by *Capability Dynamics*.

2.5.1 Capability Dynamics Models

> A complex system is no more than an interlocking structure of feedback loops. This loop structure surrounds all decisions, public or private, conscious or unconscious. The process of man and nature, of psychology and physics, of medicine and engineering all fall within this structure.
>
> *Forrester* [1969]

A popular approach to understanding dynamically complex systems involves the use of agent-based simulation technologies such as SWARM [*Swarm Development Group*, 2006]. These tools can be used to build agent-based simulations in which each agent represents a system element that behaves in response to information from its environment (including other agents) and in accordance with some specified policy.

The concepts of *Capability Dynamics* are applied by using *Capability Dynamics*

*Model*s which are agent-based simulations created as follows:

1. **Systems.** Identify the systems involved, including systems that are being built, modified or evaluated, along with all those systems with which they interact such as people, organisations, regulatory authorities, hardware, software and processes.

2. **Requirements.** Specify the requirements that each system involved in a given *Capability Dynamics Model* expects other systems to satisfy. This will often include conflicting requirements.

3. **Measures of Effectiveness.** Define *Measures-of-Effectiveness (MOEs)* that can be used to evaluate the ability of a system to satisfy each requirement. The *Capability Dynamics* approach views this effectiveness as the impact that one system has on another.

4. **Agents.** Model each system as a separate agent that generates data indicating the effectiveness with which it satisfies the requirements of dependent systems. This data is usually a function of the effectiveness of other systems (i.e., data provided by other agents), along with timing and other information that drives variation in system performance. For example, an agent that represents the *Reporting Process* depicted in Figure 2.2, would produce a value of capability derived, in part, from values of capability provided by an agent that represents the *Process Improvement Group*.

By constructing and executing *Capability Dynamics Model*s based on current, planned, proposed and experimental configurations of systems, requirements, capabilities and measures of effectiveness, the decision maker can learn about the likely impacts of various changes within a *Problem Situation*. This information can then be used to make decisions that have an increased likelihood of improving a given *Problem Situation*.

Once decisions are implemented, they are reflected in applicable *Capability Dynamics Models* and the above process is repeated. That is, *Capability Dynamics* is intended to provide a foundation for continuous learning and improvement within *Problem Situations*.

2.5.2 Limitations of Capability Dynamics

The scope of *Capability Dynamics* does not include the complete specification, design, implementation, maintenance, operation and retirement of the systems that agents represent. While it is made clear in the original paper at Appendix

H that this work is left to other research and practice, it has become evident that little research has been conducted to create and/or evaluate approaches that can be used to develop and operate systems that have *a well understood impact on their environment* over time.

2.6 eXecutable and Translatable UML

> Building a system involves understanding many different subject matters and gluing them together to make a coherent whole.
>
> *Mellor and Balcer* [2002, p29]

It was at this point in my research effort that I began to fully understand the significance of Sally Shlaer and Stephen Mellor's work with domain modelling [*Shlaer and Mellor*, 1992] and Mellor's later work with Balcer on the development of *eXecutable and Translatable UML* ($^{X}_{T}$*UML)* [*Mellor and Balcer*, 2002]. The significance of their concept of autonomous domain modelling and bridging [*Mellor and Balcer*, 2002] had escaped me.

In summary, $^{X}_{T}$*UML* is a translative approach to software development [*Bell*, 1998] in which a set of autonomous domain models are developed to specify functional and data requirements, interfaces, architecture, implementation and other aspects of a software system. These domain models are developed, verified and maintained independently of each other.

When complete and verified, domain models that represent aspects of a software system are *translated* by a *model compiler* into a conventional programming language such as Java. This is achieved using various processes of model refinement, migration, merging and weaving [*Mellor et al.*, 2004, pp51-59]. The resulting source code is then compiled into operational software by conventional compilers.

The significance of building software in this way is discussed in Sections 2.6.1 to 2.6.3.

2.6.1 Complexity management

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained –on the contrary!– by tackling these various aspects simultaneously. It is what I sometimes have called 'the separation of concerns', which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by *focussing one's attention upon some aspect*: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.
>
> *Dijkstra* [1982]

$\frac{X}{T}$*UML* supports separation of concerns in ways which have a direct impact on complexity management. Firstly, the approach separates concerns related to life-cycle activities. This reduces complexity by allowing the analyst, for example, to focus on the problem at hand without concern for architectural, design and implementation issues. The architect can in turn, focus on developing an architecture to satisfy specified performance and other non-functional requirements without any concern for functional requirements or implementation.

Secondly, and more importantly, the approach supports the separation of concerns along multiple dimensions *within* each life-cycle activity. During architectural design, for example, concerns related to distribution can be modelled without concern for issues related to persistence or security. During user interface design, concerns related to the layout of user interfaces can be modelled without any concern for the presentation language that will be used.

All of this means that $\frac{X}{T}$*UML* can be used to help manage the complexity inherent in software development.

2.6.2 Waste reduction

> Errors are most frequent during the requirements and design activities and are the most expensive the later they are removed.
>
> Boehm's First Law [*Endres and Rombach*, 2003, Law L2, pp17-19]

Irrespective of adopted life-cycle model, processes, or practices, it can be claimed that the cost of removing requirements defects increases with the time they remain undetected. The reason for this lies in the amount of effort required to rework architecture, design, implementation and integration artifacts that may have been produced to satisfy defective requirements. The longer a requirements defect remains undetected, the more rework will be necessary.

The added cost of removing any defect constitutes *waste*. The levels of such waste can be very high in software development projects. *Boehm and Basili* [2001], for example, claim that 'current software projects spend about 40 to 50 percent of their effort in avoidable rework'.

Much of this unnecessary rework can be avoided when using $\frac{X}{T}$*UML* because domain models that deal with functional requirements, for example, can be deemed correct early in the software development lifecycle before any architectural, design or implementation work has been completed or even started. Similarly, architectural domains can be developed and deemed correct before implementation is started and independently of any other domain models, including those dealing with functional requirements. This is achieved by using interactive verification tools such as those available from *Mentor Graphics* [2006] and *Kennedy Carter* [2006]. These tools can be used to execute and debug domain models to ensure that requirements, architecture and implementation errors are identified and removed as early as possible in the development process.

2.6.3 Productivity improvement

There is general agreement amongst researchers and users familiar with $\frac{X}{T}$*UML* that its use has potential to improve the productivity of software development teams. While there is some evidence to support this belief [*Stien*, 2003; *Boughton*, 2006; *Stien*, 2006], it is mostly based on intuition stemming from the following features of the approach.

- **Full code generation.** Models developed using $\frac{X}{T}$*UML* can be automatically translated into source code [*Bell*, 1998; *Mentor Graphics*, 2006; *Kennedy Carter*, 2006]. While such automation eliminates most implementation

activities, the approach requires additional architectural design work and development of appropriate model compilers should existing commercial products be unsuitable.

- **Faster change cycles.** When using the $\frac{X}{T}$*UML* approach, changes to functional and data requirements are made to appropriate domain models rather than the source code of an application. Similarly, changes related to non-functional requirements such as performance are made to appropriate architectural and implementation domain models. That is, 'the models are the code' [*Starr*, 2001].

  Once changes have been made, the domain models can be verified as correct and then automatically translated to generate new source code. This process results in a rapid change cycle. Some evidence to support this claim is reported by *Boughton* [2006] who shows that more than ten functional changes to an application per day were possible in the later stages of an *eXecutable and Translatable UML* project.

- **Large-scale reuse.** Because of their autonomous nature, $\frac{X}{T}$*UML* models represent bodies of knowledge that can be reused across a set of applications. For example, domain models that deal with subjects such as an organisation's business processes will be a valuable intellectual asset that is independent of changing implementation technology. Because of their autonomous nature, these domain models can be reused in the construction of any system that involves the organisation's business processes. Similarly, a set of models that represent aspects of a specific architecture, can be reused to construct any application that is required to run on that architecture. That is, domain models, when properly constructed, become *corporate assets* rather than expenses [*Mellor et al.*, 2004, pp10-11].

2.7 Aspect-Oriented Systems Engineering (AOSE)

In March 2003, the *Object Management Group (OMG)* released a Request For Proposal (RFP) [*Object Management Group*, 2003c] for customisation of the Unified Modelling Language (UML) to support the modelling of a wide range of systems including software, hardware, people, procedures and facilities within the framework of the OMG's *Model Driven Architecture (MDA)* [*Object Management Group*, 2003b]. Much of the response to this RFP focused on how the *UML* notation might be extended rather than how UML and MDA could be used effectively during the conduct of systems engineering lifecycle process activities.

In response to this and the possibility of generalising the applicability of ${}^{X}_{T}UML$, I, together with my supervisor, wrote a paper [*Flint and Boughton*, 2003, reproduced in Appendix J] describing the important concepts underlying ${}^{X}_{T}UML$ and how they and more traditional modelling concepts can be used to support the objectives of MDA within the Systems Engineering context.

This work, along with a full understanding of the ${}^{X}_{T}UML$ domain modelling concept, suggested the intriguing possibility of being able to form a general systems engineering approach that could be used to help people from multiple disciplines work together to build well understood systems of all kinds. These systems would include the hardware, software, organisational, legal, and social systems required to implement decisions made using techniques such as *Capability Dynamics* (Section 2.5), *System Dynamics* [*Forrester*, 1961] and the *Soft Systems Methodology* [*Checkland*, 1981; *Checkland and Scholes*, 1999].

2.8 Emergence of Aspect-Oriented Thinking

> The ideal engineer is a composite ... He is not a scientist, he is not a mathematician, he is not a sociologist or a writer; but he may use the knowledge and techniques of any or all of these disciplines in solving engineering problems.
>
> *N. W. Dougherty, 1955*

> One of the greatest disservices of formal education lies in the fact that students are induced to believe that every problem can be placed in an academic discipline such as physical, chemical, biological, psychological, sociological, political and ethical. In business schools, problems are placed in such categories as financial, personnel, public relations, production, marketing, distribution and purchasing. However, the world is not organised like universities, colleges and schools, by disciplines.
>
> *Ackoff* [1999, p15]

The preliminary research reported in this chapter was conducted over an extended period of time. It was after working through and reflecting upon this research in the later stages of the project that I came to realise that engineering projects should always be undertaken to make well understood and agreed changes within *Problem Situations*. In addition, I developed the view that a true measure of engineering success can only be found by understanding the impact of such changes on a *Problem Situation* over time. In order to fully understand such

impact, the knowledge and expertise of many disciplines will often be required [*Grigg et al.*, 2003]. For example, understanding the full impact of a system such as a dam, that involves large scale restructuring of natural systems, would require collaboration across a wide range of disciplines including civil, mechanical, electrical and software engineering, geology, biology, economics and philosophy. At present such collaboration is achieved using mostly ad hoc processes and practices in response to specific problems as they arise.

The real question was now one of how interdisciplinary collaboration could be achieved in a more systematic and effective manner. It seemed that the aspect-oriented concept of separating concerns into autonomous models that can be later woven together to form complete systems, might allow people from multiple disciplines to work together to improve *Problem Situations* of all kinds. This was the genesis of *Aspect-Oriented Thinking*.

In summary, the idea of *Aspect-Oriented Thinking* is to have engineers, scientists, lawyers, economists, philosophers and other discipline experts develop autonomous models that capture their discipline specific knowledge and perspectives on a particular *Problem Situation* as depicted in Figure 2.3. *Aspect-Oriented Specifications* are then formed to specify the way in which selected information from these models is to be used to construct, modify, operate and retire systems involved in the *Problem Situation*. Each of these specifications is formed in accordance with a particular *Aspect-Oriented Specification Archetype* which captures reusable system architecture and implementation knowledge. Finally, each specification is implemented in accordance with a separately developed and reusable *Implementation Model*.

The systems manipulated in this way might include *System Dynamics*, *Capability Dynamics* and other models that can be used to better understand a *Problem Situation*, as well as those systems involved in improving it. These systems, by way of their construction, will incorporate the multi-disciplinary knowledge required to fully understand a *Problem Situation* and make changes that have a well understood impact.

2.9 Summary of influences

As far as I can determine, there is no prior work that applies aspect-oriented concepts to the improvement of *Problem Situations* arising from the interaction of social, natural and man-made systems. However, the formulation of *Aspect-Oriented Thinking* concepts has been influenced by the work of many researchers and practitioners. While much of their work is cited throughout this thesis, its

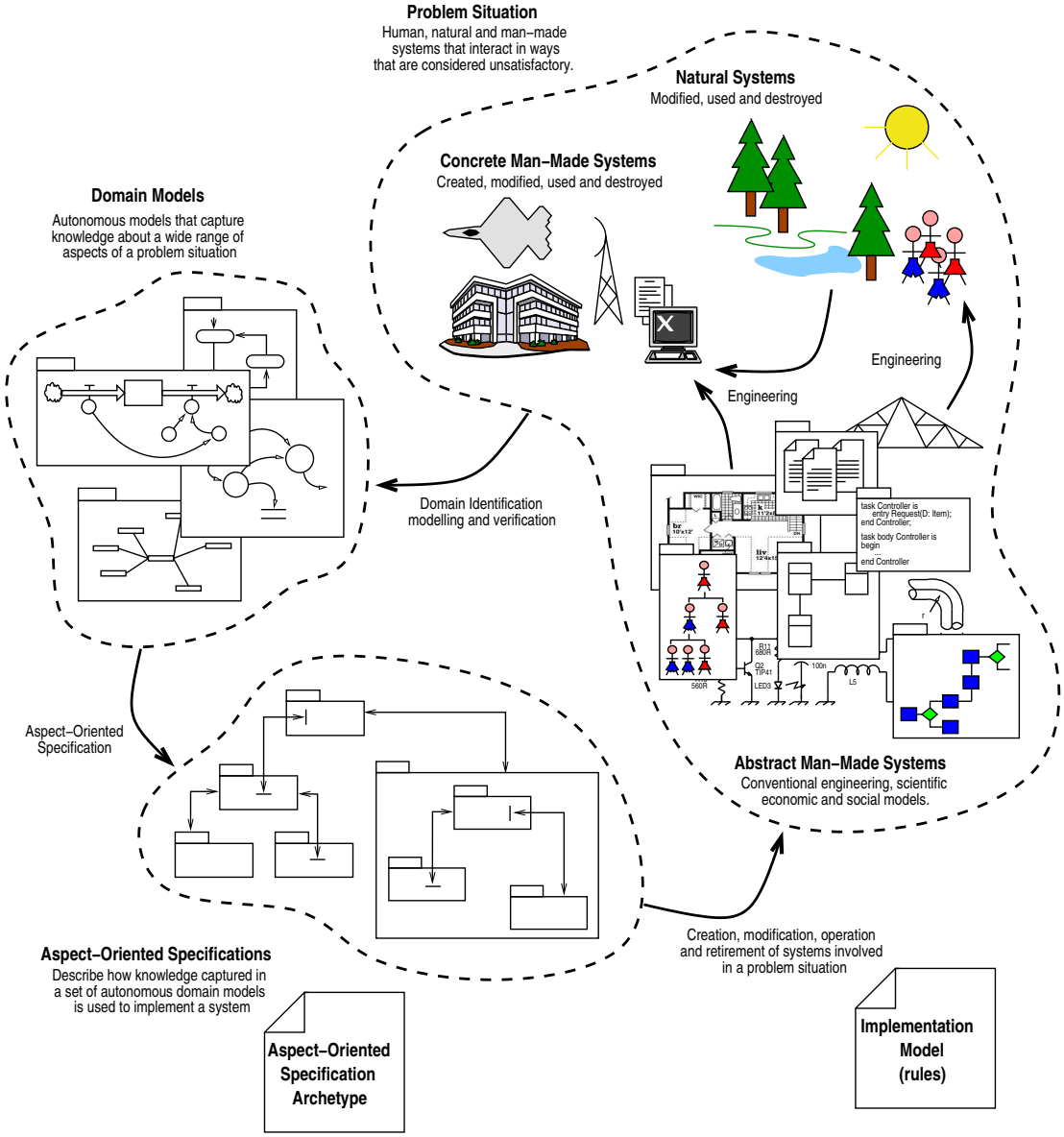

Figure 2.3: The *Aspect-Oriented Thinking* concept.

impact on the development of *Aspect-Oriented Thinking* can be summarised as follows.

My research began with a professional interest in requirements engineering as presented by *Sommerville and Sawyer* [1997]. Within a short period of time I was introduced to *Checkland* [1981] and the concept of soft-systems thinking. This in turn sparked an interest in all forms of systems thinking. Of particular interest to me is the early work of *Boulding* [1956], *Bertalanffy* [1962], *Churchman* [1968a], *Ackoff* [1971], *Jordan* [1968] and *Kramer* [1977]. This work, along with the generalised *System Dynamics* work of *Forrester* [1961], more specialist work on

software process dynamics by *Abdel-Hamid and Madnick* [1991], and the *Observe-Orient-Decide-Act (OODA)* loop concepts and inspiration provided by John Boyd [*Boyd*, 1986; *Coram*, 2002], have been influential in the development of *Capability Dynamics* discussed in Section 2.5.

The notion of *Aspect-Oriented Thinking* emerged after in-depth study, practice and teaching of the Shlaer/Mellor method [*Shlaer and Mellor*, 1992] and *eXecutable and Translatable UML ($^{X}_{T}$UML)* [*Mellor and Balcer*, 2002; *Starr*, 2002] within the context of a broad and deep interest in *Systems Thinking* and concepts involved in *Capability Dynamics*. It is the concept that the various concerns involved in a complex system can be separated and modelled in many dimensions, that most influenced the formation of *Aspect-Oriented Thinking*.

More recently, much has been written about aspect-oriented approaches to software development. While little of this has had a direct impact on the development of *Aspect-Oriented Thinking*, the aspect-oriented software development landscape as presented by *Filman et al.* [2005], the work of *Clarke and Baniassad* [2005] and *Jacobson and Ng* [2005] along with a special issue of *IEEE Software* [*Murphy and Schwanninger*, 2006] on *Aspect-Oriented Computing*, confirm that I have not reinvented the wheel.

Finally, the interdisciplinary and general nature of *Aspect-Oriented Thinking* has been influenced by many personal conversations as well as the work of *Hawken et al.* [1999] and ideas like those expressed by *Offen* [2002] and *Greenwood* [2003].

2.10 Conclusion and conjecture

In this chapter, I have argued that the success of engineering activities needs to be measured in terms of their impact over time on the human, natural and man-made systems throughout the environment within which they are conducted. I also noted that a *Systems Thinking* approach that includes a process of *continuous learning* is required to understand such impact and that decisions made to control it can only be implemented by creating, modifying, operating and retiring *Systems* of various kinds. I then proposed *Capability Dynamics*, a systems thinking approach based on *System Dynamics* which can help people understand the dynamic impact of change by modelling co-evolving systems in terms of their changing *Capabilities* and the effectiveness with which they satisfy the changing *Requirements* of dependent systems. As such, the *Capability Dynamics* approach can be used to help people make informed decisions regarding the development, modification, operation and retirement of systems that impact the satisfaction of stakeholder needs throughout a dynamically complex environment.

A limitation of the *Capability Dynamics* approach is that, while it can help people make appropriate decisions, it offers no approach to implementing them. I addressed this limitation by first showing how the *Aspect-Oriented* concepts that underpin the work of *Shlaer and Mellor* [1992] and *Mellor and Balcer* [2002] can be used within a systems engineering context. I then raised the possibility of adapting the techniques for use during the creation, modification, operation and retirement of all kinds of systems within a more general systems thinking and continuous learning context. An important implication of this idea is that it might provide a foundation for an effective interdisciplinary approach to improving large problem situations including those surrounding environmental change and national security.

In conclusion, it might be possible to blend the strong technical concepts of aspect-orientation and formal modelling with elements of systems thinking to bring together the multi-disciplinary expertise and knowledge required to understand and improve dynamically complex problem situations. From this, I offer the following conjecture that will be explored in the remainder of this thesis.

> *An approach (Aspect-Oriented Thinking), based on aspect-oriented modelling techniques, can be developed and used to help multi-disciplinary teams build systems that can be used to learn about and improve their environment over time.*

Conjecture

Part

II

Contribution

Chapter 3

Research design

> Unfortunately, software engineering research is often more solution-driven than problem-driven.
>
> *Potts* [1993]

Contents

3.1 Introduction 36
3.2 A conceptual model of software engineering research 36
3.2.1 Research methods 36
3.2.2 Research activities 38
3.2.3 Research life-cycles 39
3.3 The Aspect-Oriented Thinking research approach 44
3.3.1 Research method 44
3.3.2 Research life-cycle 44
3.4 Conclusion 48

3.1 Introduction

A key motivation for undertaking the research documented in this thesis is a need to improve industrial software engineering practice. In order to address this need, it is necessary to adopt an approach to research that responds to industry problems and facilitates the orderly evaluation, adoption and continuous improvement of research outputs.

In this chapter, I propose a novel conceptual model which can be used to describe, discuss and compare a broad range of research approaches. I validate this model by using it to discuss prior contributions in the area of software engineering research approaches. I then use this information to develop a two-phase approach to *Aspect-Oriented Thinking* research. The first phase comprises the theoretical work and proof-of-concept presented in this thesis. The second phase constitutes future research including industrial evaluation, adoption and continuous improvement of the *Aspect-Oriented Thinking* approach.

3.2 A conceptual model of software engineering research

The subject of this thesis was initially one of software engineering. The research approach adopted was therefore derived from ideas presented in the software engineering research literature.

In order to compare and evaluate prior contributions regarding software engineering research approaches and to describe new ones, a conceptual model of research is proposed. The model is loosely based on the structure (not the content) of ISO/IEC 12207 *Software Lifecycle Processes* [*International Organization for Standardization*, 1997] and ISO/IEC 15288 *Systems Enginering - System life cycle processes* [*International Organization for Standardization*, 2002] and is applicable in any research domain.

The model, represented as the UML class diagram shown in Figure 3.1, considers software engineering research approaches in terms of three orthogonal concerns: Research method, activities and life-cycles.

3.2.1 Research methods

Research methods describe the philosophical basis of a research approach. No concern is given here to the specific activities involved or the precise order in which

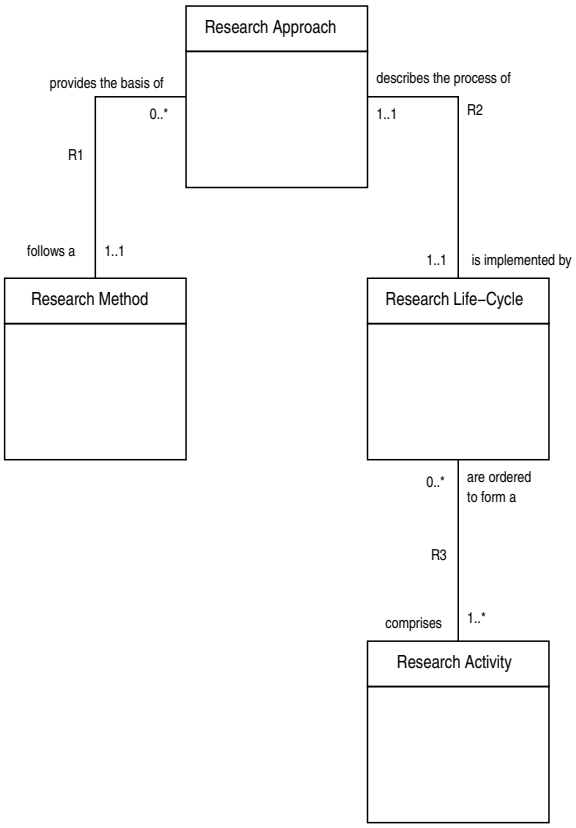

Figure 3.1: Class diagram depicting the proposed conceptual model of software engineering research

they are conducted.

During the 1990's a series of papers were published dealing with software engineering research. One of the early and often cited papers by *Adrion* [1993] presents the following software engineering research methods:

- '**Scientific Method:** observe the world, propose a model or theory of behaviour, measure and analyse, validate hypothesis of the model or theory, and, if possible, repeat.'
- '**Engineering Method:** observe existing solutions, propose better solutions, build or develop, measure and analyse, and repeat until no further improvements are possible.'
- '**Empirical Method:** propose a model, develop statistical or other methods, apply to case studies, measure and analyse, validate the model, and repeat.'
- '**Analytical Method:** propose a formal theory or set of axioms, develop a theory, derive results, and, if possible, compare with empirical observations.'

These methods and their descriptions are adopted *as-is* from *Adrion* [1993] as part of the proposed conceptual model of software engineering research.

3.2.2 Research activities

Research activities are sets of tasks that may be carried out to implement part of a research approach. The purpose here is to identify and describe a generic set of activities that can later be arranged in various ways to form a specific approach to software engineering research. No concern is given to whether all activities are used in a particular approach, the sequence in which they are conducted or how they are conducted.

Glass [1995], when considering each of the research methods described by *Adrion* [1993], concluded that they all, in general, involve the following phases.

- '**The Informational Phase:** gathering or aggregating information via reflection, literature survey, people/organisational survey, or poll (e.g. Delphi approaches).'
- '**The Propositional Phase:** proposing and/or formulating a hypothesis, method or algorithm, model, theory, or solution.'
- '**The Analytical Phase:** analysing and exploring a proposition, leading to a demonstration and/or formulation of a principle or theory.'
- '**The Evaluative Phase:** evaluating a proposition or analytic finding by means of experimentation (controlled) or observation (uncontrolled, such as a case study or protocol analysis), perhaps leading to a substantiated model, principle, or theory.'

Because, as *Glass* [1995] points out, not all of these phases are used in each research method and because some research approaches comprise iterative and/or repetitive applications of the phases, the proposed model recasts these *phases* as *activities* that should be considered when forming a research approach tailored to meet the needs of a specific research program.

In addition to activities derived from *Glass* [1995], the proposed model includes the following activities:

- **Practice.** Used to describe industrial practice that is an integral part of a research program.

- **Development.** Used to describe the development and maintenance of software and other tools that support a research program or use of research outputs.

The selection and sequencing of research activities to form a specific research approach is the subject of research Life-Cycles described below.

3.2.3 Research life-cycles

Research life-cycles identify and specify the order in which research activities need to be executed to conduct a research project using a particular research method.

3.2.3.1 Idealistic life-cycles

Each of the research methods described in Section 3.2.1 can be conducted using an idealistic lifecycle comprising all of the activities described in Section 3.2.2 ordered as depicted in the UML activity diagram shown in Figure 3.2. Some iteration may be present between the *Evaluation* activity and previous activities as depicted in Figure 3.2.

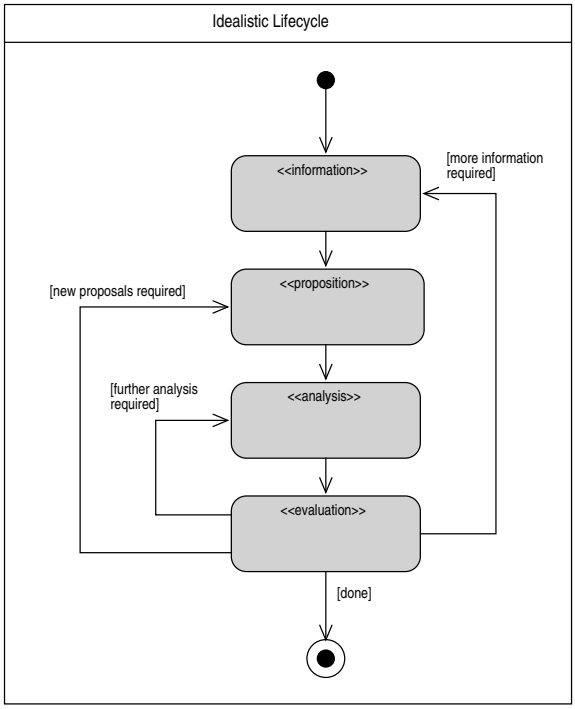

Figure 3.2: Activity diagram depicting the idealistic life-cycle common to all of the research methods described in Section 3.2.1.

In practice, *Potts* [1993] and *Fenton et al.* [1994] found that most software engineering research programs follow life-cycles that are modifications of the idealistic one depicted in Figure 3.2. In the following sections a number of such life-cycles, derived from the literature, are described using the conceptual model presented in Section 3.2.

3.2.3.2 The 'Analytical Advocacy Research' life-cycle

Fenton et al. [1994] show that many software engineering research findings can be categorised as 'Analytical Advocacy research' in which authors develop and analyse the benefits of a new approach and then, based on this analysis, advocate its use in industrial settings.

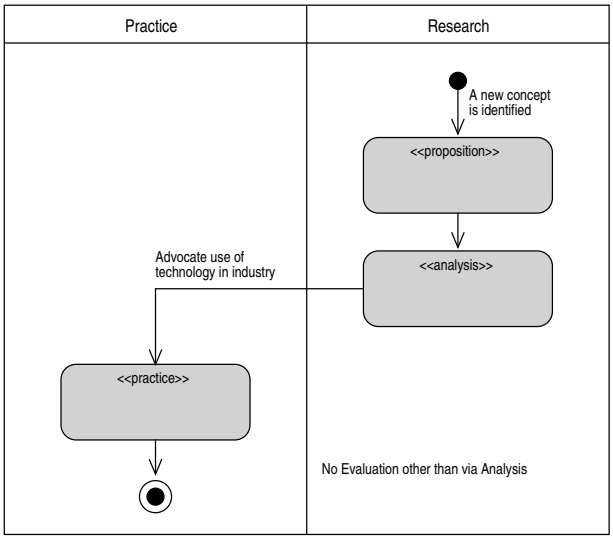

Figure 3.3: Activity diagram depicting the 'Analytical Advocacy Research' life-cycle [*Fenton et al.*, 1994].

The life-cycle of such an approach can be formed from the activities described in Section 3.2.2 and represented using the UML activity diagram shown in Figure 3.3. Stereotypes are used to indicate the type of each activity involved. The two partitions depicted ('Industry' and 'Research') are used to indicate the domain in which each research activity is conducted.

Because this approach is conducted in isolation from industrial practice, relevance of the research and its likely uptake by industry cannot be easily understood. As such, the 'Analytical Advocacy research' approach is very risky and likely to generate research outputs of very little or no industrial value.

3.2.3.3 The 'Research-then-Transfer' life-cycle

Potts [1993] describes the dominant approach to software engineering research as 'Research-then-transfer'. Despite its apparent popularity, this life-cycle is little more than a sophisticated version of the often ineffective 'Analytical Advocacy Research' approach. That is, research proceeds separately from any industrial application of its outputs.

The 'Research-then-Transfer' approach depicted in Figure 3.4, starts with motivation to use an idea or specific technology to solve a perceived industrial problem. The research then proceeds with little or no involvement with industry. At some point, when the research is considered 'ready for transfer', it is introduced to industry. Because industrial practice is likely to have evolved during the period of research, the original problem may have been solved or become irrelevant. New problems may have emerged and technological, social, organisational and commercial changes may have occurred. The result is that research conducted with such an approach may be inapplicable by the time it is put to use.

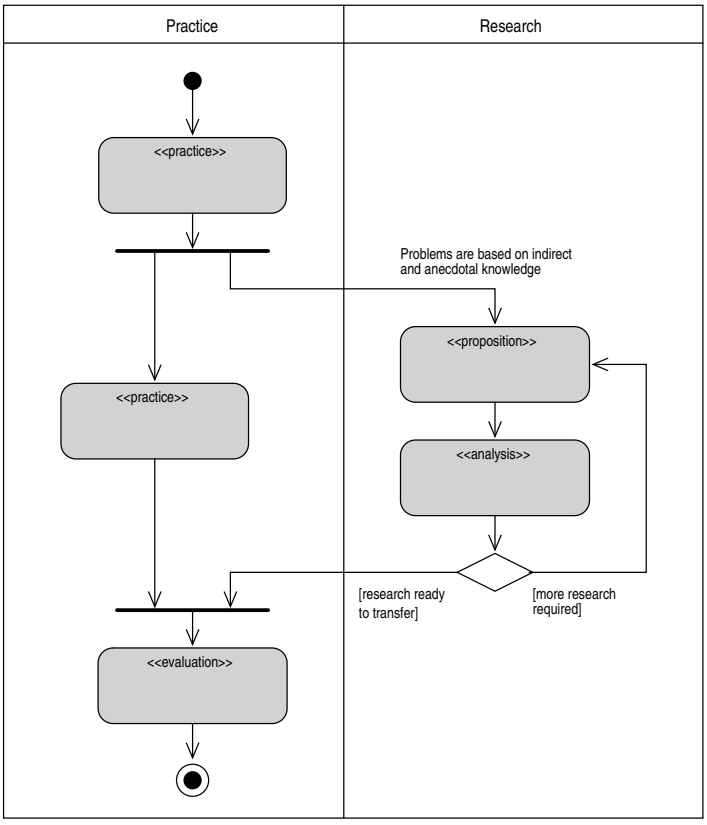

Figure 3.4: Activity diagram depicting the 'Research-then-Transfer' life-cycle [*Potts*, 1993].

Interestingly, this scenario is similar to those software development ap-

proaches which begin with the elicitation and specification of stakeholder requirements and are completed by applying a sequence of activities to develop the software with no further input from stakeholders. At the end of the process, the software is delivered to bewildered users who now require software different to what was originally specified. A common solution to this problem in software development is to introduce a more incremental approach in which there is frequent interaction between the software developers and stakeholders. *Potts* [1993] proposes a similar approach to software engineering research. He refers to this improved approach as 'Industry-as-Laboratory'.

3.2.3.4 The 'Industry-as-Laboratory' life-cycle

> It is important that software engineering problems be identified from an industrial basis rather than an academic perception.
>
> *Potts* [1993]

The UML activity diagram shown in Figure 3.5 depicts the 'Industry-as-Laboratory' approach described by *Potts* [1993]. Using this approach, researchers identify a problem through close involvement with industry and with an open mind regarding potential solutions.

Ideas, technology and approaches emerge from interactions between the research and practice domains. Results are always evaluated in an industrial setting. An important consequence of this interaction is that industrial projects not only demonstrate the effectiveness of research results, but also increase the understanding of characteristics that certain approaches and technology exhibit in specific industrial contexts. Such interaction may also lead to the discovery of real-world problems that could not have been imagined in an isolated research environment.

3.2.3.5 Research life-cycles and the OODA loop

Important insights related to research life-cycles can be drawn from concepts that underpin the use of Boyd's *Observe-Orient-Decide-Act (OODA)* decision loop [*Boyd*, 1986] in competitive military and business environments. The idea is that decision makers who cycle through their OODA loop more rapidly than others, will have a competitive advantage. The reason for this is that slower decision makers tend to make decisions based on observations and orientation that are invalid by the time decision are implemented. As a result, their actions are often inappropriate and costly.

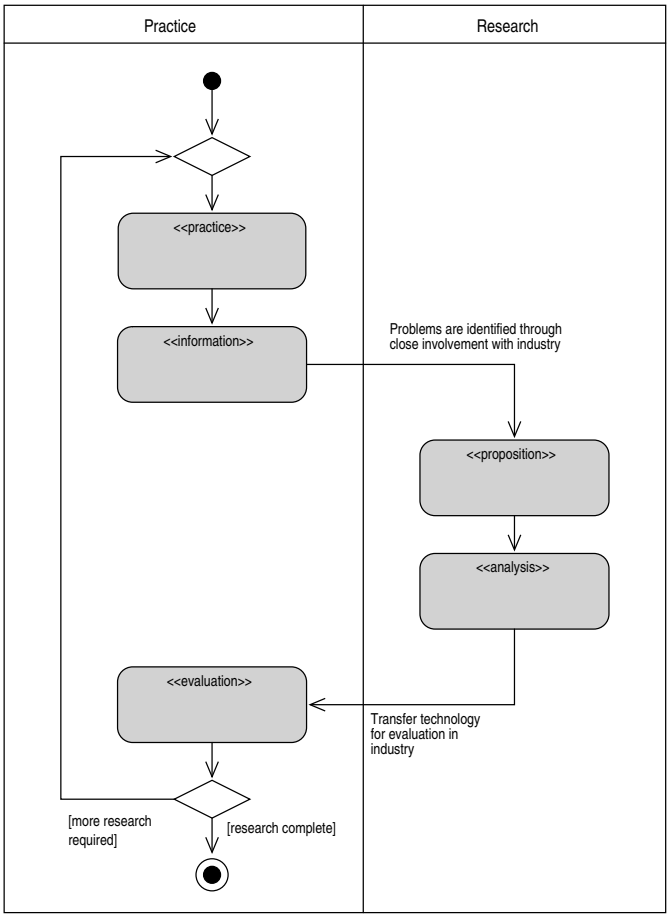

Figure 3.5: Activity diagram depicting the 'Industry-as-Laboratory' life-cycle *Potts* [1993].

The same idea can be applied to research. Research programs are likely to be more successful if they are able to react quickly to evolving industrial problems. Those that do not, like those that follow the 'Research-then-Transfer' approach or slowly cycle through the 'Industry-as-Laboratory' approach, are more likely to be unsuccessful. They may perceive changes in industrial practice as chaotic and will often research insignificant, old or non-existent problems that are mis-aligned with industrial needs. Even worse, they might involve industry in work that is of little value.

The significance of this insight is that adoption of the 'industry-as-laboratory' life-cycle might not be enough to ensure the success of research programs which aim to benefit industry. It will also be necessary to react quickly to changes in industrial practice.

3.3 The Aspect-Oriented Thinking research approach

Within this section I use the conceptual model introduced in Section 3.2 to describe the research approach that has been adopted to develop, demonstrate, evaluate and continuously improve *Aspect-Oriented Thinking*.

3.3.1 Research method

As stated in Section 1.2, the aim of this thesis is *to develop, model, demonstrate and evaluate an aspect-oriented interdisciplinary approach to engineering that can be used to build and operate systems required to understand and improve Problem Situations of all kinds*.

To achieve this aim, the *Engineering Method* of research has been adopted. The research has been conducted by observing the use of current approaches to system development and operation (Chapter 2), and then proposing and evaluating an improved approach in the form of *Aspect-Oriented Thinking* (Chapters 4 to 6).

3.3.2 Research life-cycle

Potts [1993] identifies a number of issues that need to be dealt with before 'wholesale' adoption of the 'Industry-as-Laboratory' approach. Of significance is his view that the approach can lead to an over-emphasis on the development of short-term solutions at the expense of more general solutions, and that it leads to evolutionary change rather than revolutionary change.

While *Aspect-Oriented Thinking* cannot (yet) be classified as a revolutionary approach, it is one that may have broad applicability to *Problem Situations* of all kinds. In order to ensure that this general applicability is fully explored, the two phase research life-cycle depicted in Figure 3.6 has been adopted.

3.3.2.1 Phase One - Aspect-Oriented Thinking development

Phase One of the research has been completed and documented within this thesis.

The objective of *Phase One* was to satisfy the first part of the research aim to *develop and demonstrate* an aspect-oriented interdisciplinary approach to engineering (Section 1.2). It comprised the following activities depicted in the upper horizontal partition of the activity diagram shown in Figure 3.6.

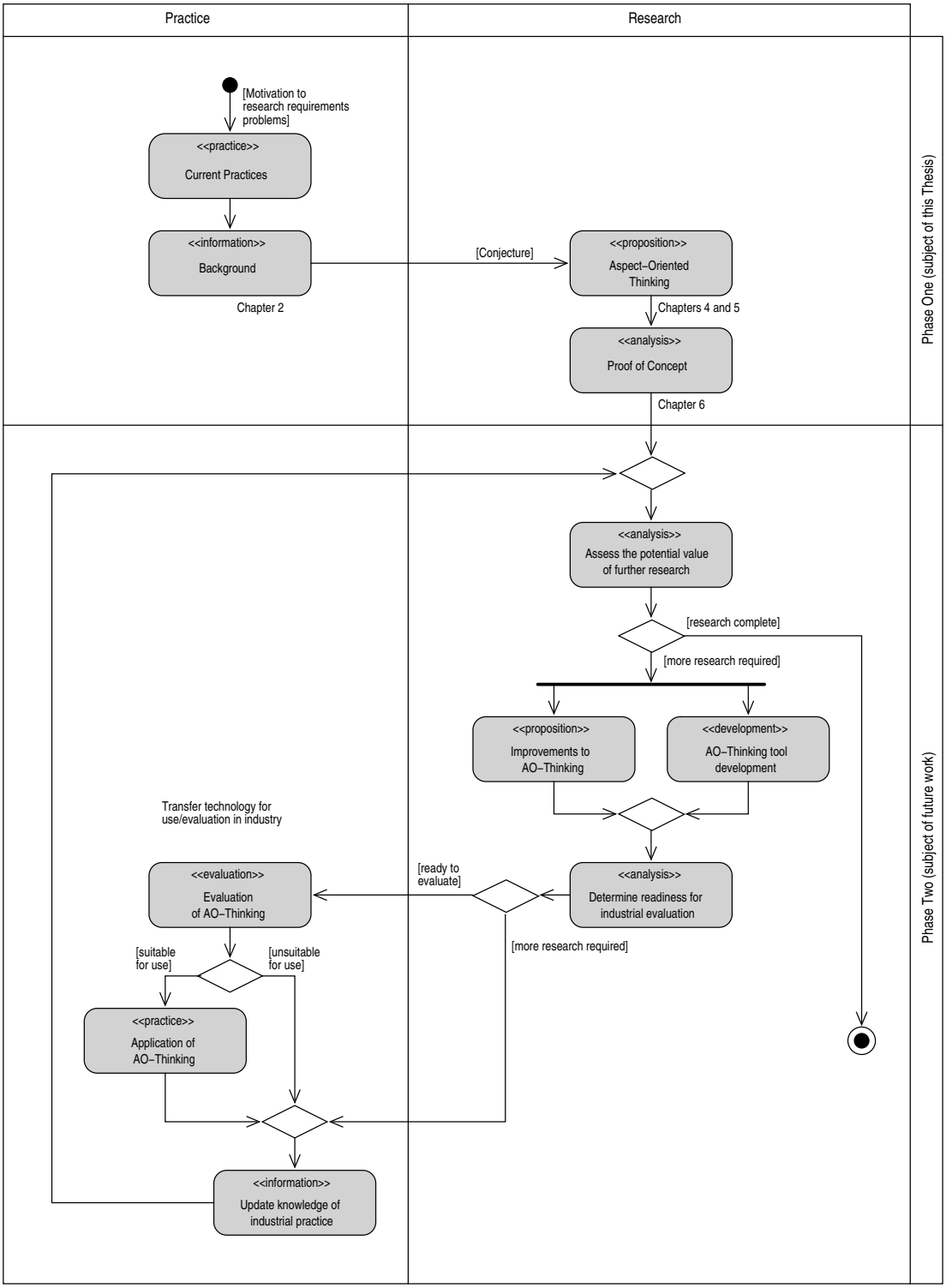

Figure 3.6: Activity Diagram depicting the *Aspect-Oriented Thinking* research approach.

- **Information - Background.** This activity established the context within which *Aspect-Oriented Thinking* was developed. This included the development of *Capability Dynamics*, an approach that can be used to understand *Problem Situations*, and *Aspect-Oriented Systems Engineering*, a preliminary aspect-oriented approach to engineering.

  The results of this activity have been reported in Chapter 2.

- **Proposition - Aspect-Oriented Thinking.** This activity proposed the idea of *Aspect-Oriented Thinking* as an interdisciplinary approach to improving *Problem Situations* (Section 2.8). The proposed approach has been fully described in the form of a *conceptual model* (Chapter 4) and a *process model* (Chapter 5).

- **Analysis - Proof-of-Concept.** *Tichy* [1998] says 'that demonstrations can provide a proof-of-concept or incentives to study a question further'. The purpose of this activity was to demonstrate the potential value of *Aspect-Oriented Thinking* as an approach to engineering any kind of man-made system. This has been achieved by developing and documenting a complete software system using the *Aspect-Oriented Thinking* approach.

  The results of this activity have been documented in Chapter 6 and Appendices B to G, and will be used to justify further development and evaluation of the approach during *Phase Two* of the project.

It could be argued that this initial research phase shows signs of the deficient 'Research-then-Transfer' lifecycle [*Potts*, 1993]. However, the research built upon existing research results ($\frac{X}{T}$UML and System Dynamics) that have been developed, evaluated and improved through close collaboration with industry over many years. The research has generalised the concepts that underpin $\frac{X}{T}$UML to produce an approach that may help solve very long standing problems associated with the improvement of dynamically complex *Problem Situations*.

In order to ensure that *Aspect-Oriented Thinking* remained industrially relevent during its development, and to improve the likelihood of industry based evaluation, ideas that emerged during the development of *Aspect-Oriented Thinking* were discussed with industry based practitioners on a regular basis.

This approach was necessary because continuous close collaboration with industry would require a significant methodological shift for most companies and their employees. This would have been a very expensive and risky proposition for most potential partners. It was felt that *Aspect-Oriented Thinking* needed to be complete, well documented and of demonstrated potential value before industry could be expected to risk an investment in effective evaluation of the approach.

3.3.2.2 Phase Two - Ongoing development and evaluation

The objective of *Phase Two* is to satisfy the second part of the research aim to *evaluate* the approach developed during *Phase One*. In addition, *Phase Two* aims to further develop *Aspect-Oriented Thinking* in collaboration with industry and as part of the interdisciplinary research community. It is an iterative process, based on the 'Industry-as-Laboratory' research life-cycle described in Section 3.2.3.4, which will continue until it is determined that *Aspect-Oriented Thinking* is of no further value to industry or it has been refined to a point where additional research would be ineffective.

Phase Two will comprise the following activities depicted in the lower horizontal partition of the activity diagram shown in Figure 3.6.

- **Analysis - Assess the potential value of further research.** The purpose of this activity is to determine if further research is likely to be of value. If so, the subsequent activities of *Phase Two* will be undertaken. If not, the research program will end.
- **Proposition - Improvements to Aspect-Oriented Thinking.** In this activity, improvements to the *Aspect-Oriented Thinking* approach will be proposed. These improvements will be modelled and described as refinements and additions to the conceptual and process models presented in Chapters 4 and 5.
- **Development - Aspect-Oriented Thinking tool development.** In order to make effective use of *Aspect-Oriented Thinking*, it will be necessary to develop and use software tools that allow users to maintain a repository of *Aspect-Oriented Thinking* models. More advanced tools will be required to automate the execution of *Implementation Rules*. High level requirements for proposed tool support are discussed in Section 7.5.2.
- **Analysis - Determine readiness for industrial evaluation.** Before industrial evaluation, the improved *Aspect-Oriented Thinking* approach and tool support will be analysed to demonstrate their potential value to industry. The purpose of this is to minimise budgetary, schedule and technical risks associated with industrial evaluation and adoption of the improved approach.

  If evaluation of the proposed improvements is justified, then the process moves on to the *Evaluation* activities described below. If not, the process moves directly to the *Information* activity where further investigations are made to determine if further improvements and changes can be made.

- **Evaluation - Evaluation of Aspect-Oriented Thinking.** The aim of this activity is to plan and conduct industry-based experimentation aimed at evaluating the effectiveness of *Aspect-Oriented Thinking* and shaping ongoing development of the approach and associated tool support. Initial evaluations will focus on the application of *Aspect-Oriented Thinking* within the context of software and systems engineering. As the research evolves, more diverse disciplines will be introduced until *Aspect-Oriented Thinking* can be evaluated as an interdisciplinary approach to improving problem situations of all kinds.
- **Practice - Application of Aspect-Oriented Thinking.** If industry evaluations indicate that *Aspect-Oriented Thinking* is suitable for use in practice, then it will be applied on a real project.
- **Information - Update knowledge of industrial practice.** After each evaluation and application of *Aspect-Oriented Thinking*, the overall state of industrial practice and problems will be re-assessed to determine how they have evolved. This information, along with the results of evaluation and application, will then be used to justify further research if required.

The conduct of *Phase Two* is beyond the scope of this thesis. The adoption and application of *Aspect-Oriented Thinking* in industry will require a significant shift in existing practice and a clear demonstration of the approach's real and potential value. This thesis is dedicated to demonstrating the potential value of *Aspect-Oriented Thinking* and as such provides justification for the conduct of *Phase Two*.

3.4 Conclusion

In this chapter I have proposed a novel conceptual model for describing research approaches. I have used the model to compare existing approaches to software engineering research and to describe the two-phase research approach that has been adopted to develop, evaluate and continuously improve *Aspect-Oriented Thinking*.

The aim of *Phase One* was to develop a solid foundation for *Aspect-Oriented Thinking* and to demonstrate its potential value to industry without requiring industry to make significant investments or accept any risk. This thesis reports the results of *Phase One* and will be used to support an argument for entering *Phase Two* which will comprise a continuous process of evaluation, adoption and improvement of *Aspect-Oriented Thinking* within an industrial context.

Chapter 4

Aspect-Oriented Thinking Concepts

> Building a system involves understanding many different subject matters and gluing them together to make a coherent whole
>
> [*Mellor and Balcer*, 2002]

Contents

4.1 Introduction 51
4.2 Problem situations 51
4.3 Systems 52
4.4 Domains 53
4.4.1 Domain categories 53
4.4.2 Knowledge domains 54
4.4.3 Specification domains 61
4.5 Domain Models 61
4.5.1 Meta-Models, Models and Instances 62
4.6 Aspect-Oriented Specifications 62
4.6.1 Specification Domain Models 63
4.6.2 Requirements 63
4.6.3 Requirement examples 64
4.6.4 Requirements that reference Aspect-Oriented Specifications 67
4.6.5 Requirement Sets 67
4.7 Aspect-Oriented Specification Archetypes 67
4.7.1 Requirement Archetypes 68
4.8 Implementation Models 69

4.8.1 Implementation rules 69
4.8.2 The form of implementation rules 71
4.8.3 Execution of implementation rules 72
4.9 Conclusion 72

4.1 Introduction

In order to facilitate effective communication among people involved in the ongoing development, evaluation and improvement of *Aspect-Oriented Thinking*, it is necessary to describe the meaning of all concepts involved with some degree of formality. It is also necessary to provide natural language descriptions of each concept and examples that help readers develop a solid understanding of the approach. To achieve this, *Aspect-Oriented Thinking* concepts have been modelled using an *eXecutable and Translatable UML ($^{X}_{T}$UML) Class Model* presented in this chapter as a set of *Class Diagrams* and associated natural language descriptions. Simple examples are provided to help readers understand the meaning of each concept.

A complete understanding of *Aspect-Oriented Thinking* can be gained by reading this chapter along with Chapter 5 which describes the process of *Aspect-Oriented Thinking*. Chapter 6 contains a *Proof-of-Concept* that demonstrates the use of *Aspect-Oriented Thinking* to build a complete software system.

Typographical and other conventions

Within the structure of this chapter, each key concept is described in a separate section. Within the text of this chapter the names of key concepts are printed in *italic* font. When introduced for the first time, these names are presented with a reference to a figure which shows a class diagram that depicts the concept and how it relates to other concepts. For example, '*Problem Situation* (Figure 4.1)' refers to a key concept depicted in the class diagram shown in Figure 4.1.

Classes shaded grey within a class diagram are defined in another class diagram. Such classes are annotated with the number of the figure in which they are defined.

4.2 Problem situations

The overall objective of *Aspect-Oriented Thinking* is to make improvements within *Problem Situations* (Figure 4.1).

For the purpose of this thesis, a *Problem Situation* is an environment in which social, natural and man-made systems interact in ways that are considered unsatisfactory by one or more of the people involved.

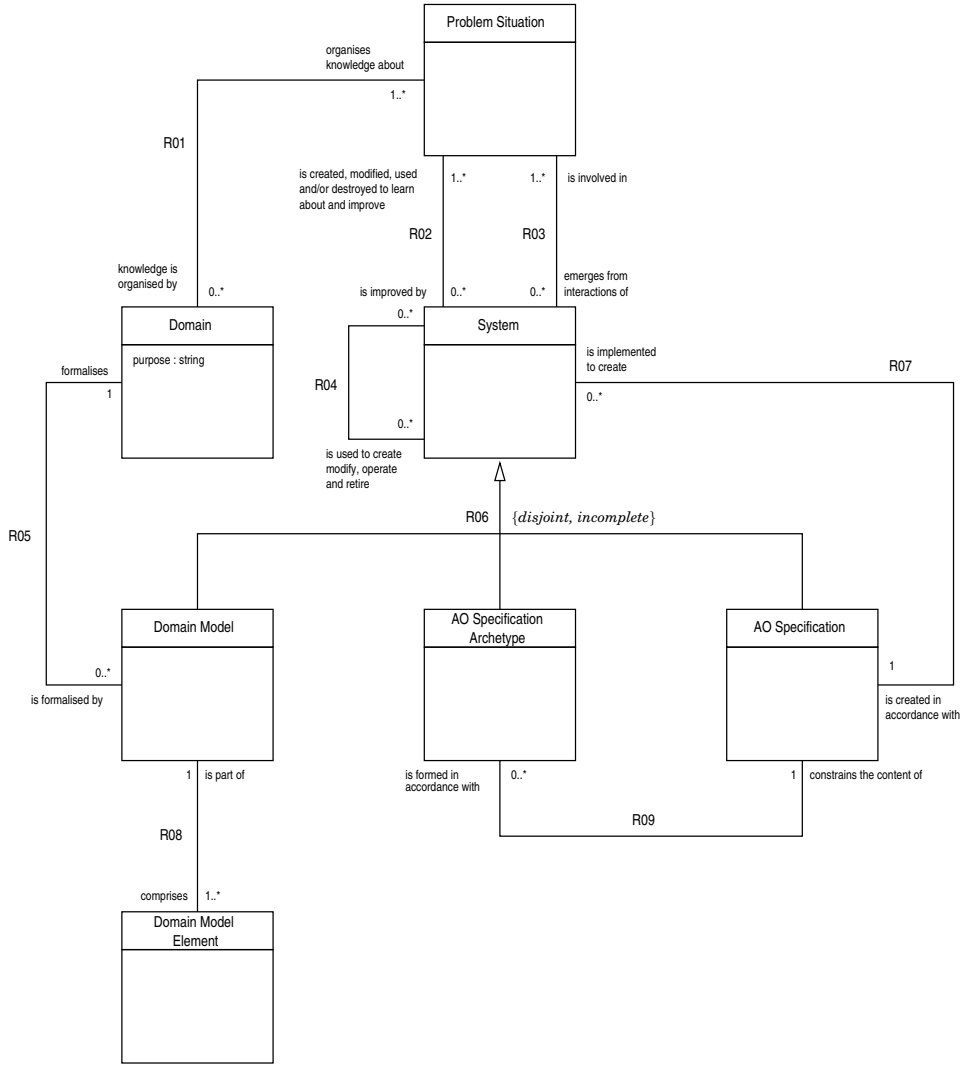

Figure 4.1: Class diagram depicting the major concepts involved in *Aspect-Oriented Thinking*. More detailed concepts are depicted in other class diagrams throughout this chapter.

4.3 Systems

Systems (Figure 4.1) are viewed from two perspectives in *Aspect-Oriented Thinking*: the *emergence* and *improvement* of *Problem Situations*.

Problem Situations emerge from the interaction of social, man-made and natural systems. Social systems include humans as individuals and organised into groupings such as teams and families. Natural systems include biological, chemical, geological and atmospheric structures. Man-made systems include concrete systems such as mechanical, electronic and urban systems as well as abstract systems such as plans, requirements and architectures.

Systems are created, modified, operated and/or retired in order to learn about and improve a *Problem Situation*. This is achieved by applying concepts described in the remainder of this chapter in accordance with the *Aspect-Oriented Thinking* process described in Chapter 5.

4.4 Domains

> A Domain is an autonomous, real, hypothetical, or abstract world inhabited by a set of conceptual entities that behave according to characteristic rules and policies.
>
> [*Mellor and Balcer*, 2002]

Existing approaches to *Problem Situation* improvement such as the *Soft Systems Methodology* [*Checkland*, 1981] focus, at all stages, on systems. While the development and use of systems is necessary to study and improve a *Problem Situation*, the *Aspect-Oriented Thinking* approach starts by applying the principle of separating concerns [*Dijkstra*, 1982] to organise the diverse subject matters (knowledge) involved in a *Problem Situation* into a set of autonomous *Domains* (Figure 4.1).

These *Domains* are broad ranging and can include education, finance, health, defence and telecommunications. They can also include design subjects such as human-machine interfaces, mechanical structures and aesthetics, as well as system architecture and implementation subjects such as topology, persistence, distribution, programming languages, structures and materials. The subject matter of project, risk and change management as well as estimation, training, quality assurance, ethics and governance can also be organised as separate domains.

4.4.1 Domain categories

While all domains are autonomous, they can be categorised in order to simplify discussion and help in their management. There is general consensus that $\frac{X}{T}$UML domains can be categorised as *application, service, architectural* or *implementation* domains [*Shlaer and Mellor*, 1992; *Raistrick et al.*, 2004]. Because *Aspect-Oriented Thinking* is a more general approach, it adopts a variation of these categories as depicted in Figure 4.2 and described below. In particular, two high-level categories are introduced to distinguish between those *Domains* of a general reusable nature

(*Knowledge Domains*) and those that relate to the specification of particular systems within a *Problem Situation* (*Specification Domains*).

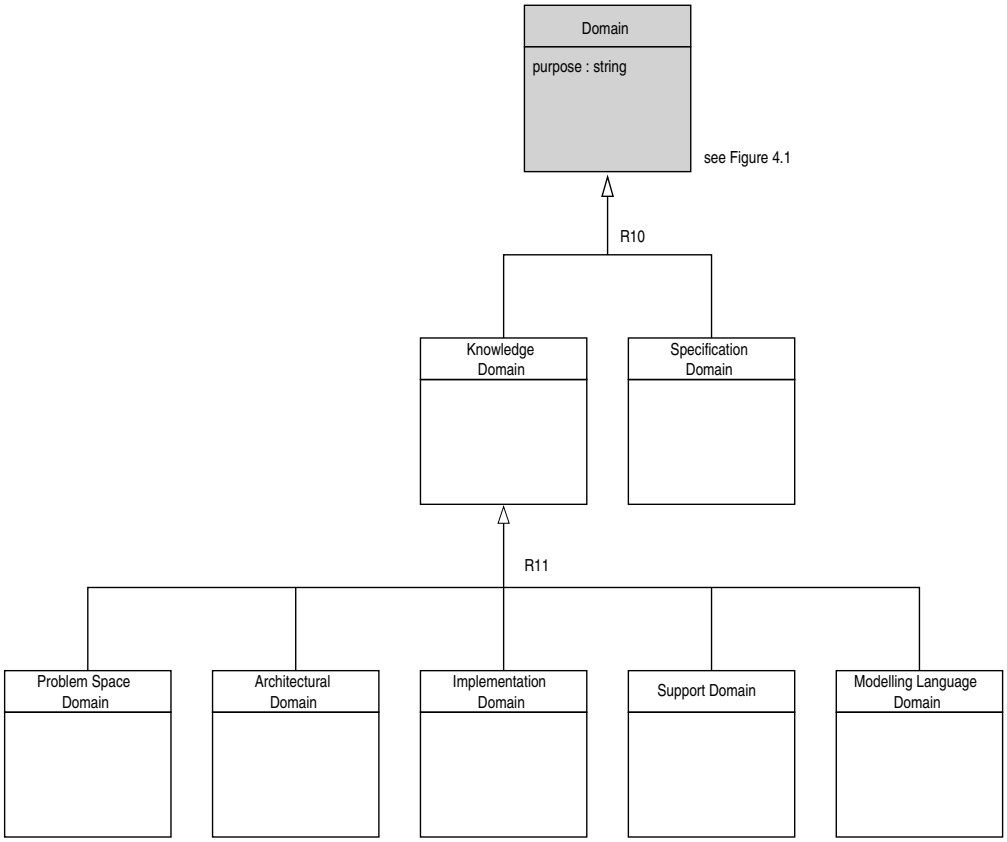

Figure 4.2: *Aspect-Oriented Thinking* domains are organised into the domain categories depicted in this class diagram.

4.4.2 Knowledge domains

Knowledge Domains represent reusable knowledge that is applicable to the creation, modification, operation and/or retirement of more than one system within a *Problem Situation*.

Note that *Knowledge Domains* on their own do not specify requirements for any particular system in that they only represent knowledge about a given subject matter. However, this knowledge can be *used* in the specification of any system to which it applies. For example, the specification of an accounting system might make use of a *Knowledge Domain* that represents knowledge about the taxation system. This *Taxation Domain*, if properly constructed, might be reused in the specifications of other systems including those related to training and fraud detection.

For the purpose of *Aspect-Oriented Thinking*, *Knowledge Domains* are organised into the categories depicted in Figure 4.2 and discussed in Sections 4.4.2.1 to 4.4.2.5 below.

4.4.2.1 Problem space domains

Problem Space Domains (Figure 4.2) deal with the core subject matters of a *Problem Situation* and ignore concerns such as those related to the architecture, design and implementation of any system involved.

Examples of *Problem Space Domains* include:

- **Warehousing.** A warehousing domain would describe entities such as stock items, locations and storeman. It would describe the relationships between each of these entities and the way in which orders, invoices and other information is processed. The domain would not concern itself with the way in which users interact with entities involved in the domain or how they are implemented.
- **Railway Signaling.** A railway signaling domain would describe how entities such as tracks, signals, points and trains are organised and the way in which they are controlled.
- **Banking.** A banking domain might include entities such as accounts, interest rates, transactions, customers, managers and regulations. Behavior of the domain might include the opening and closing of accounts, transactions and audits. Such a domain would make no reference to system architecture or implementation.
- **Production-Distribution.** *Forrester* [1961] describes a *System Dynamics* model of an industrial production and distribution system. The model deals with entities such as factories, warehouses, distributors, inventory and orders. The dynamics of the model involve factory lead times, decision times and shipping delays. The model ignores all architectural and implementation issues and focuses on the problem at hand, as perceived by users of the system. The subject of this model is an example of a *Problem Space Domain*.

It is often the case that specific *Problem Space Domains* revolve around more generic subject matter. For example, the warehousing domain described above revolves around the subject matter of storage, control and movement of items. The banking domain revolves around the subject matter of accounting and security. Within the *Aspect-Oriented Thinking* approach (as in $^{X}_{T}$UML), these concerns can,

themselves, be treated as autonomous *Domains* and separated from the more specific *Problem Space Domains*. *Mellor and Balcer* [2002, §17.1.2] provides an example of this in the form of an *Inventory* domain that deals with subject matter such as inventory items, orders and suppliers. This domain is used with Mellor's Bookshop system, but can also be used in other systems that involve an 'inventory' aspect.

Another example is a generic security domain which could include entities such as protected resource, users, roles and privileges. It would describe how these entities are related to each other and how they behave in response to certain stimuli. Behavior might include registration of new users, authentication and managing access to protected resources. As a separate generic concern, the security domain would make absolutely no reference to any particular application, architectural or implementation domain. For example, it would not make any assumptions about the way in which authentication is managed as this is a separate concern and the subject of another domain. Nor would it refer to any application specific roles such as *administrator* or *user*.

4.4.2.2 Architectural domains

The subject matter of *Architectural Domains* (Figure 4.2) is used to understand, specify and manage the structure of systems involved in a *Problem Situation*. In simple cases, a single *Architectural Domain* could deal with an entire architecture. A single domain that deals with the subject matter of the *armidale* architecture [*Flint and Boughton*, 2002], for example, could be developed. For more complex architectures, or when a more detailed model is required, it may be better to separate architectural concerns into a set of separate domains. Examples of such domains include:

- **Architectural style.** System architecture is often based on one or more architectural styles such as pipes and filters, hierarchical layers, repository, distributed and client-server [*Bass et al.*, 2003]. Within the *Aspect-Oriented Thinking* approach these styles could be modelled in separate *Architectural Domains* that can be used as required to describe the architectural style of systems involved in a *Problem Situation*.
- **Persistence.** Many software architectures are required to provide ways of maintaining persistent data. One implementation choice is to use a relational database. A domain that deals with the subject matter of persistence could take this approach by modelling elements such as databases, tables, rows and

queries. The domain could also deal with the subject matter of backups and recovery.

An alternative persistence domain could be modelled around the concepts of serialisation. A given system architecture could then make use of the persistence *Domain Model* that best satisfies non-functional requirements such as performance, maintainability and scaleability.

- **Human-Computer interfaces.** While it is often the case that user interfaces are developed within the context of requirements, the architecture of each class of human-computer interfaces should be modelled in an architectural domain. For example, the subject matter of web-based user interfaces could be modelled as a separate domain that can be used within any architecture that requires a web-based interface to satisfy applicable design constraints and other non-functional requirements.

  Similarly, the principles involved in designing interfaces for aircraft instrumentation could be modelled in a separate domain.

- **Distribution.** The way in which elements of a software system are distributed across a network is an architectural concern that can be modelled separately. The domain subject matter would include topology, communications, redundancy and network management.

- **Organisational structure.** Various forms of organisational structure such as hierarchical and matrix arrangements can be modelled as architectural domains that can be instantiated to form structures for specific organisations within a *Problem Situation*.

4.4.2.3 Implementation domains

The subject matter of *Implementation Domains* (Figure 4.2) is used to construct the systems involved in a *Problem Situation*. Examples include:

- **Programming languages.** Programming languages used to build software systems would be represented by programming language domains. Each one would describe the elements that comprise a particular programming language (e.g., classes, modules, packages, variables, tasks) and how they are organised and related.

- **Mechanical structures.** Mechanical systems are often constructed from standardised components or concepts. Beams, columns and panels for example are often used to construct buildings. Shafts, bearings and gears

are often used to construct rotating machinery. Within the *Aspect-Oriented Thinking* approach, the subject of mechanical structures could be treated as a separate *Implementation Domain* that can be used to help understand and construct the systems involved in a *Problem Situation*.

- **Transistor-Transistor Logic (TTL).** TTL is a design concept that underpins the 7400 series integrated circuits (IC). These ICs are widely used in the construction of digital systems. Within the *Aspect-Oriented Thinking* approach, the subject of TTL along with the specifications of available TTL devices could be considered as a separate implementation domain that can be used to help understand and construct the systems involved in a *Problem Situation*.

4.4.2.4 Support domains

Support Domains (Figure 4.2) can be used to deal with the environment within which other domains and systems involved in a *Problem Situation* are developed, modified and used. Examples include:

- **Project Planning.** A project planning domain might deal with resources, tasks, schedules, and reviews. Such a domain would not deal with a particular project plan, but rather the concepts from which actual plans could be formed.
- **Quality Improvement.** A support domain that deals with quality assurance systems and quality improvement could include entities such as products, customers, standards, metrics and processes (possibly described in terms of the subject matter of a separate *process* domain).

Domains that deal with the subject matter of safety, ethics, risk management, process improvement, organisational learning, knowledge management, training, human resource management, public relations, financial control and documentation could also be developed to support the lifecycle of systems.

4.4.2.5 Modelling language domains

Many systems thinking and engineering approaches advocate, and are often based on, the use of a single language to model all aspects of a problem or problem situation. For example, the $\frac{X}{T}$UML approach uses a subset of the UML to model aspects of a software system and *System Dynamics* advocates the use of a single language to model a wide variety of dynamic systems.

Common languages such as the UML may not always be suitable for modelling all domains involved in a given problem situation. For example, domains dealing with mechanical and electronic design should be modelled using languages of the disciplines involved. In fact, software may sometimes be more effectively modelled using techniques such as Data Flow Diagrams [*Demarco*, 1979; *Hatley and Pirbhai*, 1987; *Ward and Mellor*, 1985] and Entity Relationship Diagrams [*Chen*, 1977] despite the widely held view amongst software developers that these techniques are *old fashioned* and have been superceded by newer object-oriented approaches.

The subject matter of *Aspect-Oriented Thinking* domains will vary greatly. They will be modelled by people and organisations from a wide range of disciplines, backgrounds and cultures, each likely to have their own languages and ways of doing business. Because of this, the *Aspect-Oriented Thinking* approach has been constructed to allow the systematic use of multiple languages to construct *Domain Models* (Figure 4.1).

This multi-language capability allows users to develop *Domain Models* in languages with which they are familiar and are appropriate to the subject matter at hand. The approach also allows a single domain to be modelled using different languages in order to more clearly represent different aspects of the subject matter of concern.

$\frac{X}{T}$UML Realised Domains

Within the $\frac{X}{T}$UML approach, a *realised* domain is one that exists as software and does not require further modelling using the $\frac{X}{T}$UML language [*Mellor and Balcer*, 2002, §17.1.3, p280]. Within the *Aspect-Oriented Thinking* approach there are no realised domains because the approach works with *Domain Models* developed using any modelling language, including English!. For example, the syntax and descriptions of the realised domain examples quoted by *Mellor and Balcer* [2002] (HTML, Java and XML) are viewed as *Aspect-Oriented Thinking* domain models that may have been developed using languages such a *Backus-Naur Form* (BNF) and *English*. That is, there is no need to distinguish existing models from those developed within the context of *Aspect-Oriented Thinking*.

The following are common modelling languages likely to be used to develop *Aspect-Oriented Thinking Domain Models*.

- **eXecutable and Translatable UML.** $\frac{X}{T}$UML is likely to be the dominant language for modelling software aspects of a problem situation. It is a well defined subset of the Unified Modelling Language (UML) and includes an

implementation of the UML action semantics [*Object Management Group*, 2003a, Chapter 11].

An important characteristic of $^{X}_{T}$UML, as opposed to the full UML, is that it is an *executable* language. This allows $^{X}_{T}$UML models to be executed within tools such as those available from *Mentor Graphics* [2006] and *Kennedy Carter* [2006]. This facilitates improved learning and a more complete treatment of behaviour within a given domain. In addition, *Specification Domain Models* (Section 4.4.3) developed using $^{X}_{T}$*UML*, can be tested and deemed correct early in the system life-cycle. This has the potential to significantly reduce later rework (Section 2.6.2).

- **Executable UML.** When using the $^{X}_{T}$UML language as defined by *Mellor and Balcer* [2002], all behaviour is defined within procedures associated with the states of active classes. In practice, some domain behavior is independent of state. Modelling such behavior within class state machines can sometimes lead to complex and convoluted state models. For this reason, some domains may be modelled using Executable UML (eUML) [*Raistrick et al.*, 2004], an executable modelling language which is similar to $^{X}_{T}$UML but includes *operations* that can be used to model state-independent behaviour.

- **Data Flow Diagrams.** Data flow diagrams [*Demarco*, 1979] can be used to model *Domains* that involve the flow of information through various processes including those related to software applications. The approach provides a way to partition a *Domain* into processes or tasks and is effective in modelling the subject matter of information systems such as invoicing, insurance claims processing and payrolls.

- **Flow Charts.** While considered dated by many software developers, Flow Charts [*IBM*, 1969] provide an effective way to describe a specific sequence of actions that may involve repetition and conditional branching. Note that UML activity diagrams [*Object Management Group*, 2005, section 12.5] can, in many cases, be used as a replacements for Flow Charts.

- **System Dynamics.** Dynamically complex *Domains* are often best modelled using the language of *System Dynamics* [*Forrester*, 1961]. Examples include population growth, the impact of pollutants on the marine environment, economics and health care.

  Like $^{X}_{T}$UML, *System Dynamics* is an *executable* language. This allows *System Dynamics* models to be executed within tools such as those available from *ISEE Systems* [2006].

- **Mind Maps.** *Domains* that deal with ideas and concepts may be best modelled, at least initially, using Mind Maps [*Buzan*, 2003] or similar techniques.

These can be used to develop a shared, preliminary, understanding of a subject matter prior to more formal modelling using another language such as $\frac{X}{T}$*UML* or *System Dynamics*.

- **Capability Dynamics.** *Capability Dynamics* is a novel approach to understanding where improvements can be made in *Problem Situations* and the effectiveness of any changes that aim to make such improvements. A summary description of the approach together with references can be found in Section 2.5.

4.4.3 Specification domains

Unlike *Knowledge Domains*, which deal with subject matter that may be applicable to many systems throughout one or more *Problem Situations*, *Specification Domains* deal with subject matter that is unique to the creation, modification, operation and retirement of a particular system. For example, a *Specification Domain* might deal with the specific user interface for a new software system or a specific security policy for a new building.

Each *Specification Domain* will normally be modelled as an instance of an appropriate *Knowledge Domain Model*. For example, a *Specification Domain* that deals with the user interface for a new software system might be modelled as an instance of a *Knowledge Domain Model* that deals with the subject matter of HTML Web-based interfaces. Examples of *Specification Domains* and their corresponding models can be found in Sections D.3 and D.5.

4.5 Domain Models

> The value of a model depends on the view taken, but none is best for all purposes
>
> Davis' Law [*Endres and Rombach*, 2003, Law L4, p22]

The subject matter of each *Domain* is organised and communicated using one or more *Domain Models* (Figure 4.1). Each *Domain Model* represents a particular autonomous view of the subject matter involved in a *Domain*. For example, several *Domain Models* could be developed for a single *Graphical User Interface* (GUI) *Domain*. One could describe the elements involved and how they can be arranged, while others could describe usability issues, or look and feel policies. It is also possible to use separate *Domain Models* to represent alternative, and possibly conflicting, views of a particular *Domain*.

Depending on the nature of a *Domain*, *Domain Models* will be formed using the knowledge and expertise of many disciplines including engineering, management, business, economics, medicine, physics, mathematics, chemistry, biology, humanities, social science, the law and the environment.

4.5.1 Meta-Models, Models and Instances

Within the *Aspect-Oriented Thinking* approach no real significance is attached to whether a *Domain Model* is an instance model, model, meta-model or meta-meta-model corresponding to levels M0, M1, M2 and M3 of the MOF Meta-Model Architecture [*Object Management Group*, 2003e]. All that is important is that *Aspect-Oriented Thinking Domain Models* are usually instances of some other *Domain Model*.

For example, generalised *Problem Space, Architectural* and *Support Domains* would normally be instances of a *Modelling Language Domain* such as $\frac{X}{T}$*UML*. Instances of these *Architectural* and *Support Domains* might then be formed to specify requirements for specific systems. For example, a *Domain Model* that describes the concepts involved in building web-based user interfaces might be formed as an instance of the $\frac{X}{T}$*UML Modelling Language Domain*. This model would then be instantiated to form a set of *Domain Models* that specify the user interfaces for particular software systems.

4.6 Aspect-Oriented Specifications

Verified and well understood *Domain Models* represent valuable intellectual assets that capture knowledge and expertise involved in separate aspects of a *Problem Situation*. However, in order to learn about and improve a *Problem Situation*, it is necessary to create, modify, operate and retire *Systems*. Within the *Aspect-Oriented Thinking* approach, this is achieved by forming and then implementing *Aspect-Oriented Specifications* (Figures 4.1 and 4.3). Each *Aspect-Oriented Specification* describes how subject matter defined in a set of *Domain Models* is used to create a specific *System* (the *target* system). Some of these *Systems* will be used to learn about, and improve a *Problem Situation*. Others will be used to support the development, modification, operation and/or retirement of other *Systems*.

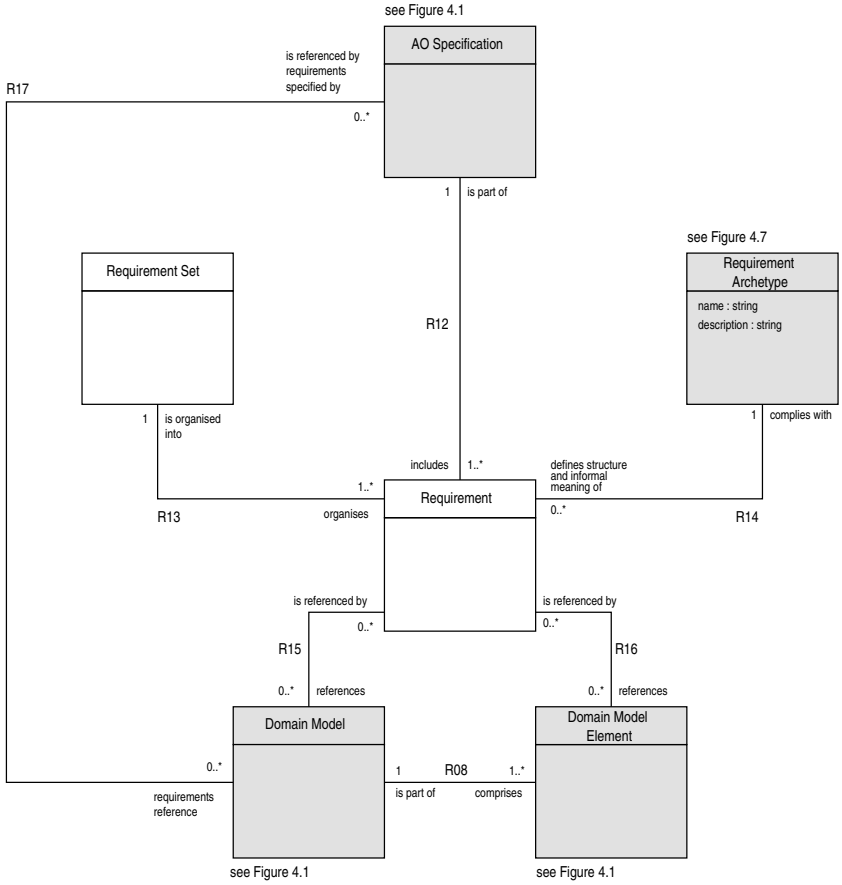

Figure 4.3: Class diagram depicting the concepts involved in *Aspect-Oriented Specifications*.

4.6.1 Specification Domain Models

Each *Aspect-Oriented Specification* may include a number of *Specification Domain Models* (Section 4.4.3) which describe subject matter that is unique to the target system. For example, the user interface for a software system would be included within an *Aspect-Oriented Specification* as a *Specification Domain Model*.

4.6.2 Requirements

Each *Aspect-Oriented Specification* includes a set of *Requirements* (Figure 4.3). Each *Requirement* specifies a property of the target system in terms of specific subject matter captured in applicable *Specification* and *Knowledge Domain Models*.

Each *Requirement* is formed in accordance with a *Requirement Archetype* (Section 4.7.1). Each archetype specifies the *Domain Model* subject matter that compliant *Requirements* must reference, and provides a natural language

statement of what the *Requirements* mean. All *Requirements* that conform to a given *Requirement Archetype* will have the same meaning, but in relation to different subject matter. For example, a set of *Requirements* that each reference a different $\frac{X}{T}$UML attribute, might comform to a single *Requirement Archetype* which states that the referenced attributes are to be stored in a non-volatile memory. That is, a single *Requirement Archetype* has provided the meaning for a set of *Requirements* that each reference a different $\frac{X}{T}$*UML* attribute.

4.6.3 Requirement examples

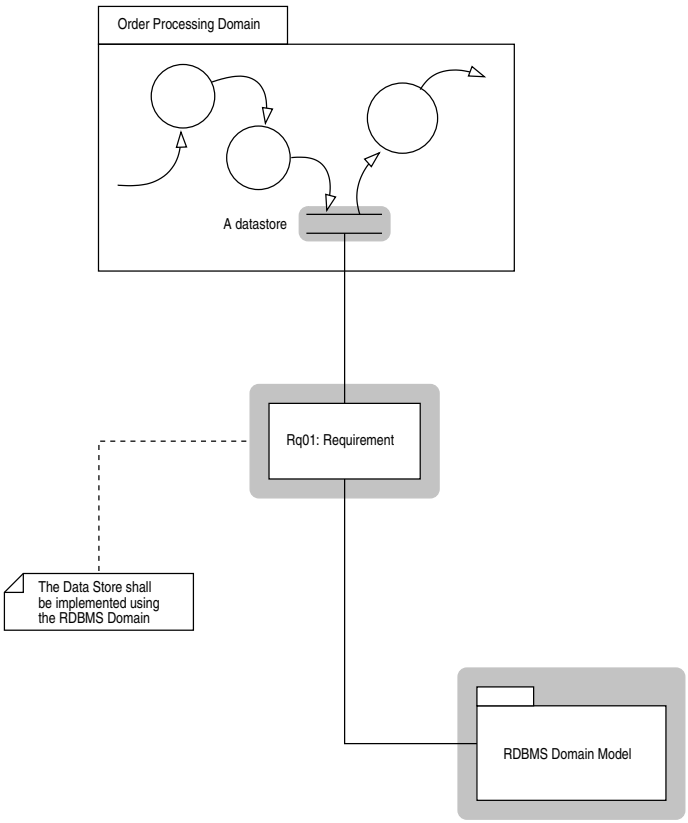

Figure 4.4: Example of a *Requirement* that references a data store within a *Data Flow Diagram* being used as a *Domain Model*, and the *Relational Database Management System* domain model.

Figure 4.4 depicts a simple example of a *Requirement* that references a *Data Store* within a data flow diagram being used as a *Domain Model* and a *Relational Database Management System (RDBMS) Domain Model*. In this example, a natural language description of the *Requirement*, as provided by an associated *Requirement Archetype*, could be something like 'The Data Store shall be implemented using the RDBMS domain model'.

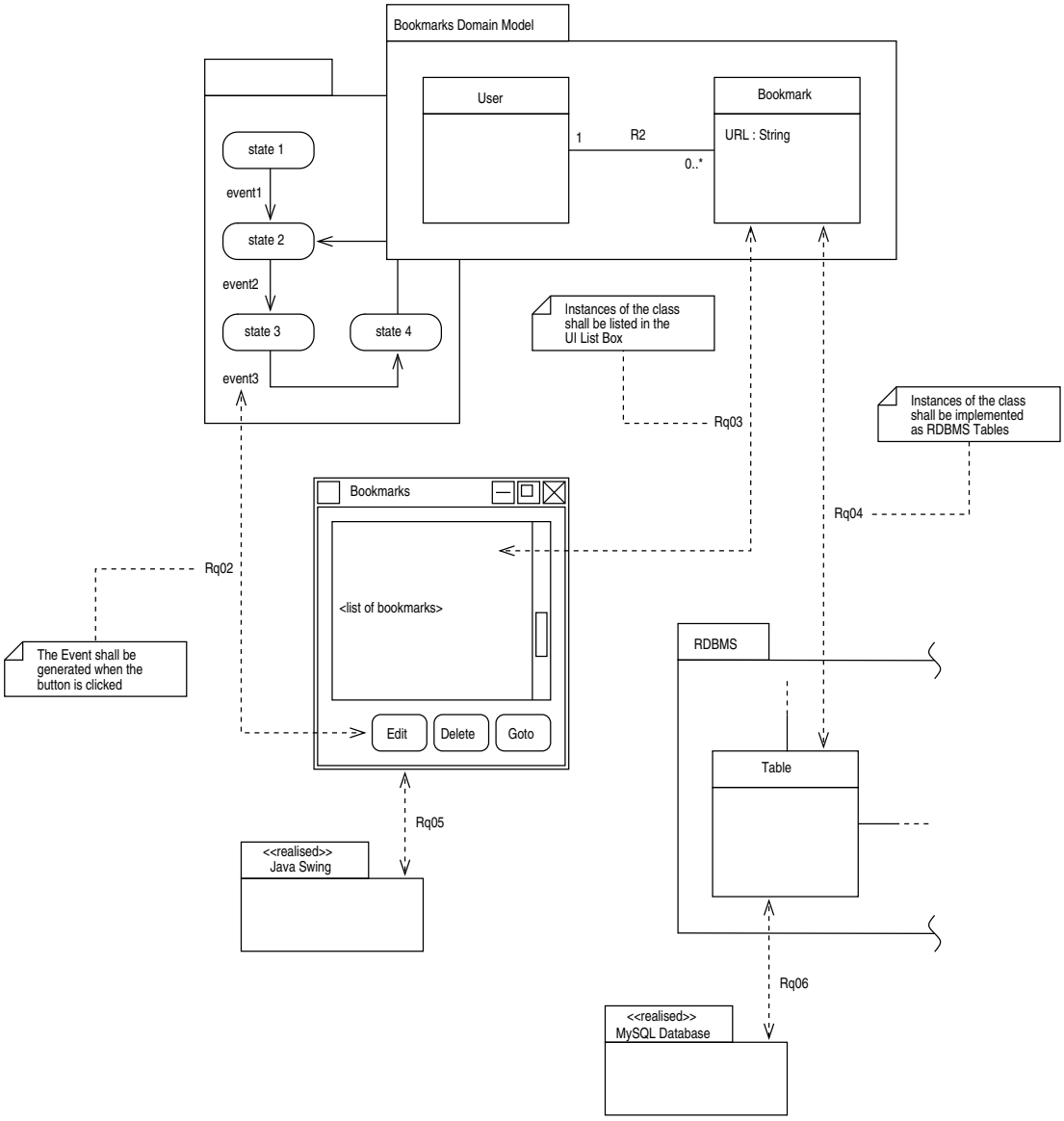

Figure 4.5: Examples of *Requirements* represented using the notation described in Appendix A.

Figure 4.5 depicts use of the simplified notation described in Appendix A to show *Requirement Rq02* which references an *event* in a problem domain modelled using $\frac{X}{T}$UML and a *Button* in a graphical user interface domain modelled using screen layouts. The meaning of this *Requirement* could be informally described by an associated *Requirement Archetype* as a requirement for the event in the $\frac{X}{T}$UML domain to be generated when the button, however it is implemented, is clicked.

Figure 4.5 also depicts *Requirements Rq03* and *Rq04* which reference an $\frac{X}{T}$UML class, a list box on a GUI and a the *Table* class in the *RDBMS Domain Model*. The meaning of *Rq03* could be described by an associated *Requirement*

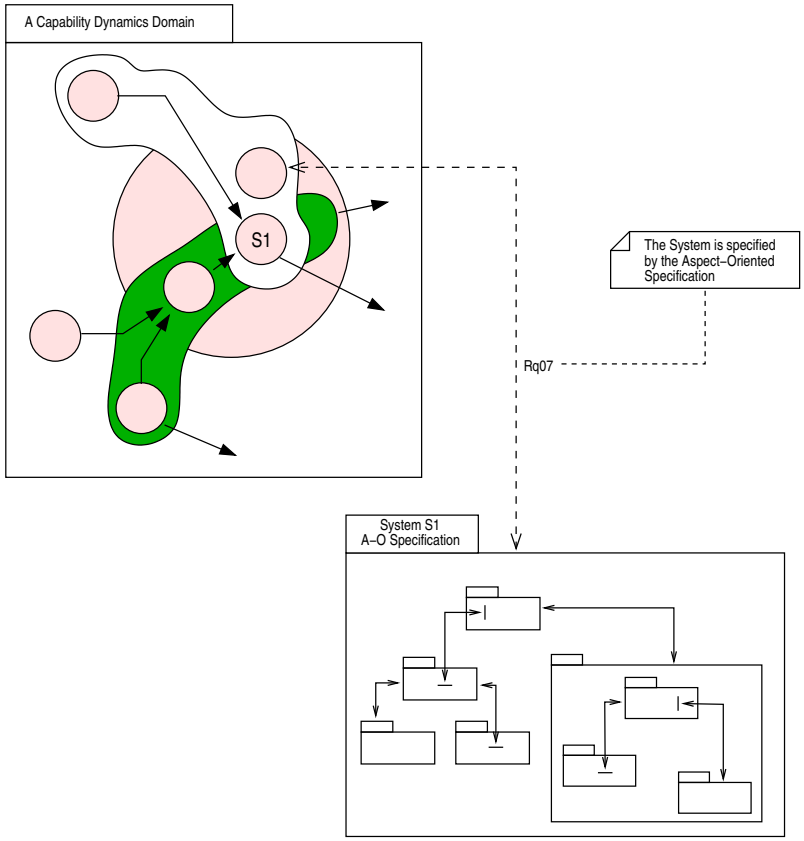

Figure 4.6: Example of a *Requirement* that references a system in a *Capability Dynamics* model and an *Aspect-Oriented Specification*.

Archetype as a requirement for instances of the referenced *Bookmark* class to be listed in the referenced GUI *List Box*. Similarly, the meaning of *Rq04* could be described by an associated *Requirement Archetype* as a requirement for the *Bookmark* class to be implemented as a relational database table.

Other examples of *Requirements* include:

- The specification of a data flow from an external entity in one *Domain* modelled using *Data Flow* techniques may be referenced by a *Requirement* that also references a specific sensor in a *Domain* that deals with environmental sensors that are capable of providing environmental data. The meaning of such a *Requirement* could be described by an associated *Requirement Archetype* as a requirement for the sensor to generate the data flow specified in the data flow domain. The first domain makes no assumptions about *how* the flow is generated and the sensor domain has no knowledge of what its sensors are used for. It is the *Requirement* and *Requirement Archetype*, which are separate from both *Domain Models*, that provide this information.

- A *Requirement* for all classes defined in an $\frac{X}{T}$UML domain to be documented using a standardised documentation format could reference the $\frac{X}{T}$UML meta-model *Domain* and a LaTeX template *Domain Model* that is capable of describing documentation formats. Such a *Requirement* could be described by an associated *Requirement Archetype* as a requirement for all $\frac{X}{T}$UML classes to be documented in accordance with the referenced template format.

4.6.4 Requirements that reference Aspect-Oriented Specifications

Aspect-Oriented Specifications can themselves participate, as *Domain Models*, in larger *Aspect-Oriented Specifications*. To use an *Aspect-Oriented Specification* as part of a larger model, *Requirements* that reference entire *Aspect-Oriented Specifications* or *Domain Models* and *Requirements* within an *Aspect-Oriented Specification* are established. For example, Figure 4.6 depicts *Requirement Rq07* which references a *System* in a *Capability Dynamics* model and an entire *Aspect-Oriented Specification*. The meaning of *Rq07* might be described by an associated *Requirement Archetype* as a requirement for the referenced *System* to be implemented in accordance with the referenced *Aspect-Oriented Specification*.

4.6.5 Requirement Sets

Requirements can be organised into *Requirement Sets* (Figure 4.3) which can be used to simplify the representation of *Aspect-Oriented Specifications* using the notation described in Appendix A.

4.7 Aspect-Oriented Specification Archetypes

Each *Aspect-Oriented Specification* comprises a set of *Requirements* that comply with appropriate *Requirement Archetypes* defined within a single *Aspect-Oriented Specification Archetype*. That is, an *Aspect-Oriented Specification Archetype* constrains the contents of an *Aspect-Oriented Specification*.

In most cases, an *Aspect-Oriented Specification Archetype* will define a set of *Requirement Archetypes* that cover all of the *Requirements* necessary to fully specify an entire *System*. That is, a single *Aspect-Oriented Specification Archetype* describes how knowledge from a wide variety of *Domain Models* can be brought together to fully specify the functional, architectural and implementation requirements for a complete *System*.

In effect, *Aspect-Oriented Specification Archetypes* capture architectural decisions in a form that can be exploited during the specification of many systems that share the same architecture. For example, an *Aspect-Oriented Specification Archetype* might be developed for the specification of applications that run on a particular mobile phone. This archetype could then be instantiated to form *Aspect-Oriented Specifications* for a range of different applications that will run on the same type of phone.

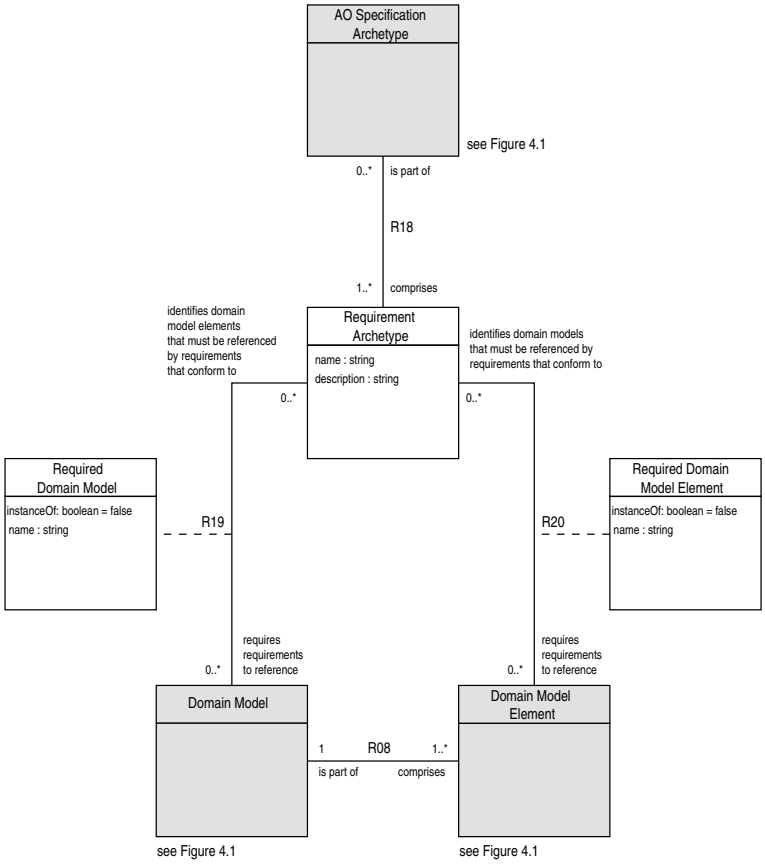

Figure 4.7: Class diagram depicting the concepts involved in *Aspect-Oriented Specification Archetypes*.

4.7.1 Requirement Archetypes

Within the *Aspect-Oriented Thinking* approach, the meaning of each *Aspect-Oriented Specification Requirement* is provided, informally in the first instance, by an associated *Requirement Archetype* (Figures 4.1 and 4.7) defined in an *Aspect-Oriented Specification Archetype*. Formal meaning is provided by *Implementation Rules* described in Section 4.8.

Requirement Archetypes also describe the form that *Requirements* must take.

The *Domain Models* that must be referenced by a specific *Requirement* are identified by *Required Domain Models* (Figure 4.7). Each *Required Domain Model* formalises a link between a *Requirement Archetype* and a specific *Domain Model*. The *Required Domain Model* association class includes an *instanceOf* attribute which, if *True*, indicates that a compliant *Requirement* must reference an instance of the linked *Domain Model*. If the *instanceOf* attribute is *False*, then a compliant *Requirement* must reference the linked *Domain Model* itself.

Similarly, *Required Domain Model Elements* formalise links between *Requirement Archetypes* and a specific *Domain Model Element*. They include an *instanceOf* attribute which, if *True*, indicates that a compliant *Requirement* must reference an instance of the linked *Domain Model Element*. If the *instanceOf* attribute is *False*, then a compliant *Requirement* must reference the linked *Domain Model Element* itself.

For example, Figure 4.8 depicts the *Requirement Archetype* for *Requirement Rq01* depicted in Figure 4.4. It requires compliant *Requirements* to reference an instance of the *Data Store* class in the *Data Flow Meta-Model* and the *RDBMS* domain model.

4.8 Implementation Models

An *Implementation Model* (Figure 4.9) comprises a set of *Implementation Rules* (Figure 4.9) which describe how an *Aspect-Oriented Specification*, formed in accordance with a particular *Aspect-Oriented Specification Archetype*, is to be implemented.

4.8.1 Implementation rules

Implementation Rules describe how part of a target system is to be created. For example, an *Implementation Rule* might be formed to generate a fragment of software source code or part of a mechanical structure.

In most cases, an *Implementation Rule* will be associated with a single *Requirement Archetype* (see association *R23* depicted in Figure 4.9). In such cases, the *Implementation Rule* will be executed with respect to every *Requirement* that is specified in accordance with the archetype. For example, an *Implementation Rule* associated with *Requirement Archetype RA01* depicted in Figure 4.8, would be applied to every data store referenced by a *Requirement* that conforms to the archetype.

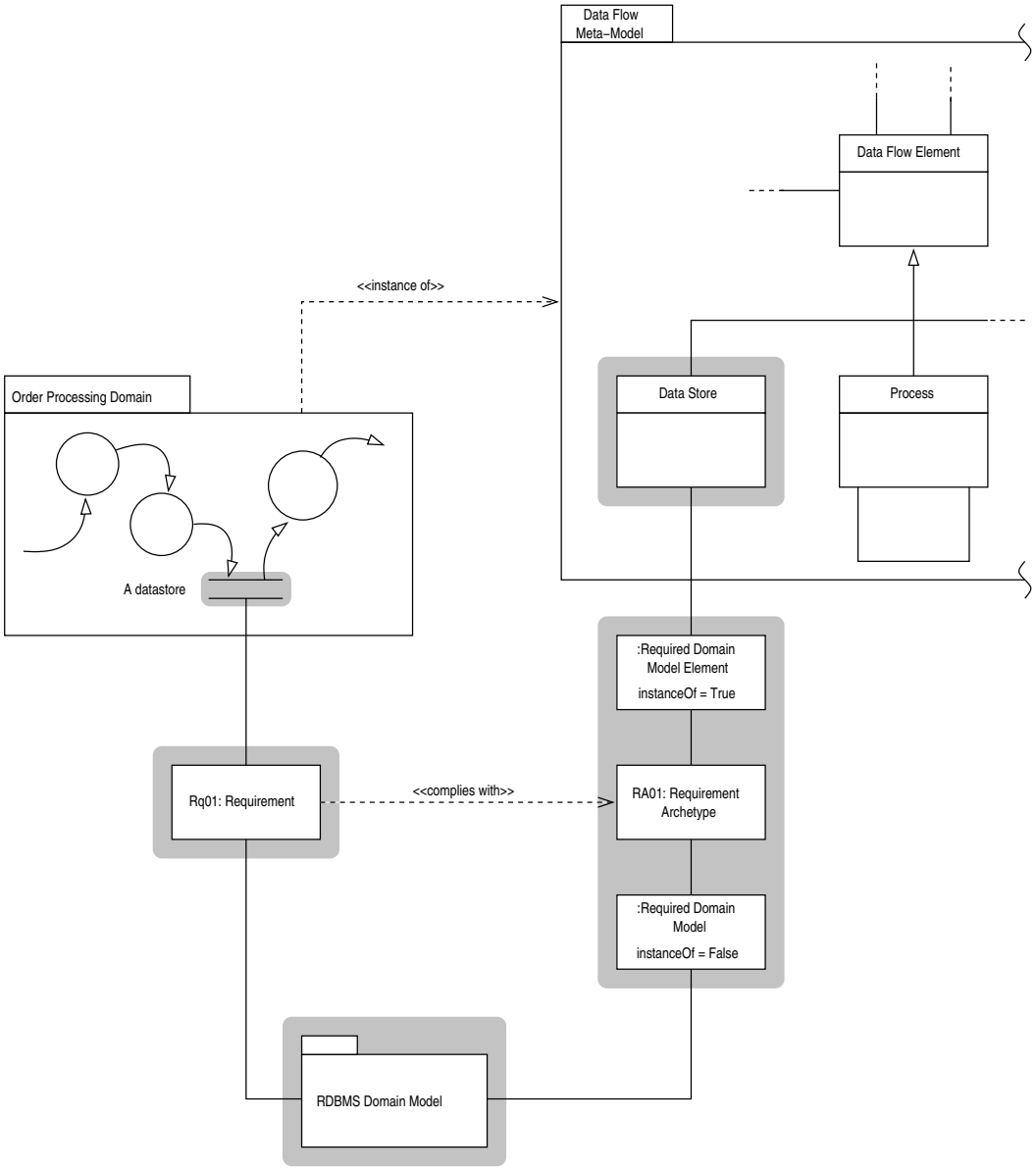

Figure 4.8: The *Requirement* depicted in Figure 4.4 and the *Requirement Archetype* with which it conforms.

In other cases, an *Implementation Rule* will act independently of any *Requirements* stated in an *Aspect-Oriented Specification*. For example, the directory structure used to hold the source code for a software system might be described in an *Implementation Rule* that does not relate to any specific *Requirement Archetype*.

It is also possible for an *Implementation Rule* to be associated with two or more *Requirement Archetypes* that are related in some way. This is often the case when it is necessary to *weave* together subject matter from several *Domain Models*. For example, a *Requirement* for an event in an $\frac{X}{T}$*UML Domain Model* to be generated

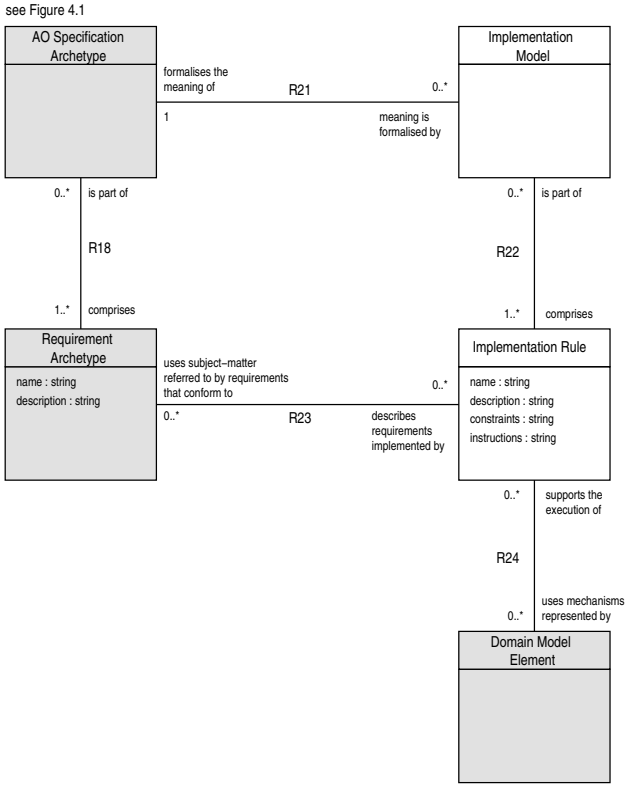

Figure 4.9: Class diagram depicting the concepts involved in *Implementation Models*.

when a *Button* on a user interface is *clicked*, might be implemented along with a further *Requirement* for the same *Button* to be visible only to users with specific privileges defined in a separate *Security Domain Model*. Implementation of this woven set of requirements will require the use of subject matter from several *Domain Models*.

Finally, it is possible for several different *Implementation Rules* to be associated with the same *Requirement Archetype*. All this means is that each rule is used to implement a different part of the target system using information referenced by the same *Requirement*.

4.8.2 The form of implementation rules

At this stage of the research, *Implementation Rules* are not formally modelled. Instead, they are described using natural language instructions which will normally involve manual tasks to be undertaken by a human using simple tools and techniques. Some instructions might involve the use of conventional engineering, scientific, management and social methodologies, tools and processes. They might

also involve the use of tools designed to automate the implementation of an *Aspect-Oriented Specification*.

4.8.3 Execution of implementation rules

By automating the execution of *Implementation Rules* defined in an *Implementation Model*, it is possible to automate the implementation of associated *Aspect-Oriented Specifications*. The implication of this possibility is that productivity improvements similar to those described in Section 2.6.3 can be gained during the development of all kinds of systems. This will, in turn, improve the ability of interdisciplinary teams to effectively respond to evolving *Problem Situations*.

A limitation of an automated approach is that a separate automated *Implementation Engine* will be required for each *Implementation Model*. The cost of developing such tools, can however, be minimised by applying the *Aspect-Oriented Thinking* approach itself to their development. For example, aspects of existing *Implementation Models* could be separated into autonomous *Domain Models*. A reusable *Aspect-Oriented Specification Archetype* for *Implementation Engines* could then be formed and used to create *Aspect-Oriented Specifications* for various specialised *Implementation Engines*. These specifications could then be implemented to create automation tools. The implementation of these tools could also, of course, be automated using the same approach.

4.9 Conclusion

In this chapter I have merged *Systems Thinking* concepts with those of *Aspect Orientation* to form a novel approach to learning about and improving *Problem Situations*. The approach supports the use of *Systems Thinking* techniques such as *System Dynamics* and *Soft Systems Methodology* within an engineering framework of system creation, modification, operation and retirement. Most significantly, *Aspect-Oriented Thinking* has the potential of becoming an effective interdisciplinary approach to understanding and improving *Problem Situations* of all kinds. This is possible because of the way in which *Aspect-Oriented Thinking* is able to integrate autonomous *Domain Models* that may be developed by people from different disciplines using different languages.

Chapter 5

Aspect-Oriented Thinking Process

Contents

5.1 Introduction 74
5.2 Problem situation analysis 75
5.3 Knowledge domain identification and analysis 76
5.3.1 Domain identification 76
5.3.2 Language modelling 77
5.3.3 Domain modelling 77
5.3.4 Domain model verification and exploration 78
5.4 Aspect-Oriented Specification Archetype development . . . 79
5.4.1 Requirement archetype development 79
5.5 Aspect-Oriented Specification development 79
5.6 Implementation modelling 80
5.7 Aspect-Oriented Specification implementation 80
5.8 Continuous learning and improvement 80
5.9 Comparison with traditional life-cycle models 81
5.10 Documentation 83
5.11 Conclusion 84

5.1 Introduction

In order to make use of the *Aspect-Oriented Thinking* concepts presented in Chapter 4, it is necessary to develop, apply and continuously improve a well defined process.

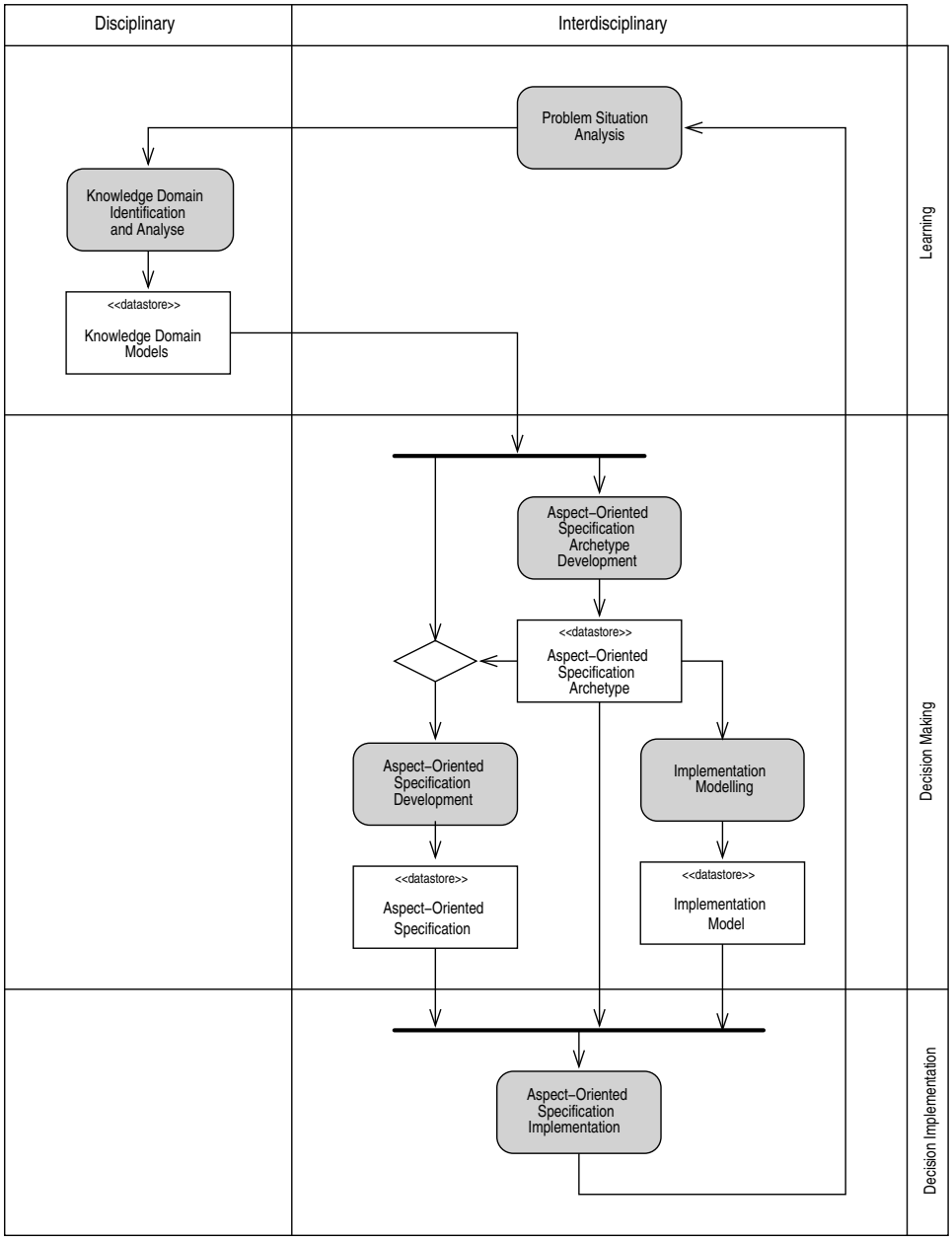

Figure 5.1: An activity diagram depicting the process of *Aspect-Oriented Thinking*.

In this chapter, I describe the process of *Aspect-Oriented Thinking* with the aid of the UML activity diagram shown in Figure 5.1. This diagram depicts the activities involved in *Aspect-Oriented Thinking*, the order in which they

are conducted, and the data items they consume and produce. The activity diagram also makes use of vertical partitions to identify those activities conducted within the context of a single discipline and those that are conducted within a interdisciplinary context. More importantly, horizontal partitions are used to identify those activities which are involved in learning about a *Problem Situation*, making decisions that aim to improve a *Problem Situation* and those involved in implementing such decisions.

5.2 Problem situation analysis

The *Aspect-Oriented Thinking* process begins with *Problem Situation Analysis*. The aim of this activity is to fully understand a *Problem Situation* and to make decisions regarding the creation, modification, operation and/or retirement of *Systems* to improve it.

Problem Situations can be explored using existing approaches such as *Soft Systems Methodology* [*Checkland*, 1981; *Checkland and Scholes*, 1999], *System Dynamics* [*Forrester*, 1961] or *Capability Dynamics* [*Flint*, 2001] (Section 2.5). Simulations and other tools used to support these approaches can be constructed using the *Aspect-Oriented Thinking* approach itself. For example, a *Domain Model* developed using *Capability Dynamics* could be used to describe the effectiveness of interactions between the systems involved in a *Problem Situation*. This *Domain Model* could then be woven together with *Domain Models* that deal with agent-based simulation technology to form an *Aspect-Oriented Specification* for a *Capability Dynamics* simulation. This specification could then be implemented to form a simulation that can be used to learn about a *Problem Situation*, identify system interactions that need to be improved and to evaluate the impact of various options to bring about such improvements.

Domain Models developed using *System Dynamics* could be combined with *Domain Models* that represent off-the-shelf simulation tools such as *iThink* [*ISEE Systems*, 2006] to form *Aspect-Oriented Specifications* for simulations that can be used to identify and understand required improvements. In this case, the *Implementation Model* associated with the *Aspect-Oriented Specification* would comprise a set of instructions that humans could follow to use *iThink*. In fact, the *Implementation Model* may simply refer to existing *iThink* documentation[1].

[1]This example highlights the very general applicability of the *Aspect-Oriented Thinking* approach. In this simple case the approach aligns with existing practices for using established modelling languages and off-the-shelf tools. As the approach is used for larger and more complex systems, the number of *Domain Models* involved will increase and the *Implementation Model* will most probably become more sophisticated and possibly automated.

More complex *Aspect-Oriented Specifications* could also be formed by integrating simulations of several separate domains. Environmental simulations dealing with separate concerns such as climate, epidemiology and population could be integrated using *Aspect-Oriented Thinking* to produce a single, possibly distributed, simulation (an *aspect-oriented simulation!*).

5.3 Knowledge domain identification and analysis

The purpose of this activity is to develop and verify *Knowledge Domain Models* (Section 4.4.2) that each deal with a specific aspect of a *Problem Situation*. *Knowledge Domain Models* are used to improve understanding of a specific subject matter and to help promote consensus and communication amongst the various relevent stakeholders, while avoiding confusion with unrelated concerns such as volatile technology.

The *Knowledge Domain Identification and Analysis* activity comprises the sub-activities described in the remainder of this section. These activities will normally be conducted by people with relevent, often discipline based, domain knowledge.

5.3.1 Domain identification

The aim of this sub-activity is to identify the separate concerns, or subject matters, involved in a *Problem Situation*. In many cases, *Domains* can be identified by considering the various disciplines required to fully understand a *Problem Situation*. For example, *Domains* that deal with the environment, civil engineering, organisational structure, politics, medicine, sociology and the law might be identified when dealing with water shortages and the possibility of constructing new dams or recycling waste water.

Domains might also align with life-cycle activities such as requirements, architecture and implementation, or with concerns that cut-across a number of other domains, such as project, risk and configuration management, security, testing and maintenance.

That is, *Domains* are used to separate the concerns involved in a *Problem Situation* in multiple dimensions.

Note that the development of preliminary *Aspect-Oriented Specifications* is also often helpful when identifying domains. They help Aspect-Oriented thinkers understand how the subject matter of each domain might be used.

5.3.2 Language modelling

Most *Aspect-Oriented Thinking Domain Models* are developed using a language defined in a *modelling Language Domain Model* (a *Meta-model*). For example, *Domain Models* developed using $\frac{X}{T}$UML are instances of a separate *Domain Model* which describes the $\frac{X}{T}$UML language.

The purpose of the *Language modelling* sub-activity is to develop models of the languages used to form *Aspect-Oriented Thinking Domain Models*. This is typically a once-off activity conducted whenever a modelling language, for which there is no existing meta-model, is required.

Note, however, that it may not necessary to have a complete and formal meta-model for every *Domain Model*. All that is required is that every *Domain Model Element* referenced by a *Requirement Archetype* or *Requirement*, be clearly identifiable. Indeed, the only formal meaning of such elements that is of interest, is that provided by *ImplementationRules* (Section 4.8.1).

5.3.3 Domain modelling

The aim of this sub-activity is to develop models of *Domains* involved in a *Problem Situation*. The purpose of modelling *Domains* is to develop a clear understanding of (potentially) complete subject matters in isolation of other subject matters. These models can then be used to communicate concepts within a team or organisation and, more importantly, across traditional interdisciplinary boundaries. Ultimately, they are used in the formation of *Aspect-Oriented Specifications*.

Domain Models are usually developed by domain experts using domain specific languages and techniques. Significantly, no expertise in *Aspect-Oriented Thinking* is required at this stage other than a clear understanding that each *Domain Model* should be autonomous. That is, each *Domain Model* should only deal with a specific aspect of a *Problem Situation* and should not be cluttered with unrelated concerns.

Note that *Knowledge Domain Models* and *Specification Domain Models* will often be developed in different contexts and at different stages of the *Aspect-Oriented Thinking* process. For example, many of the reusable *Knowledge Domain Models* that deal with subject matters such as system architecture and implementation will be developed in the context of organisational support for the life-cycle of systems throughout an enterprise and over an extended period of time. As specific *Problem Situations* emerge, *Problem Space Knowledge Domain Models* (Section 4.4.2.1) will be developed and verified in order to learn about and improve situations of immediate concern. *Specification Domain Models*, on the other hand,

will only be developed in the context of *Aspect-Oriented Specifications* related to the creation, modification, operation and retirement of specific systems.

5.3.4 Domain model verification and exploration

Because of their autonomous nature, the correct structure and behavior of each *Domain Model* can be verified independently of other models, and before they are used in the context of an *Aspect-Oriented Specification*. For example, *Domain Models* that deal with system requirements can be verified independently of, and possibly before considering any system architectural, design or implementation concerns. This has advantages in terms of cost and schedule because the possibility of architectural, design and implementation rework is reduced as discussed in Section 2.6.2.

5.3.4.1 Static verification

Most *Aspect-Oriented Thinking Domains* are modelled using a language defined in a specific meta-model (see Section 5.3.2). The aim of *static verification* is to ensure that each *Domain Model* is properly constructed in accordance with its meta-model. The process is not dissimilar to the way in which programming language compilers check software source code for syntax errors.

While *static verification* can be conducted using manual processes, it is often automated within modelling tools. For example, *Bridgepoint* [*Mentor Graphics*, 2006] and *iUML* [*Kennedy Carter*, 2006] can statically verify $\frac{X}{T}$UML models as they are created. Similarly, *System Dynamics* tools such as *iThink* [*ISEE Systems*, 2006] can statically verify *System Dynamics* models as they are created.

5.3.4.2 Dynamic verification

The behaviour of *Domain Models* developed using executable languages such as $\frac{X}{T}$UML and *System Dynamics* can be verified using simulation. This involves execution of *Domain Models* in an interactive execution/simulation environment such as the *Bridgepoint Verifier* [*Mentor Graphics*, 2006] for $\frac{X}{T}$UML models and *iThink* [*ISEE Systems*, 2006] for *System Dynamics* models. These simulations can be used to verify and explore the behavior of a *Domain Model*. Information gathered from *dynamic verification* can be used to learn about a *Problem Situation*, and to make decisions regarding its improvement.

5.4 Aspect-Oriented Specification Archetype development

The purpose of this activity is to develop *Aspect-Oriented Specification Archetypes* (Section 4.7) that can be used to form *Aspect-Oriented Specifications* for systems required to implement decisions made during the *Problem Situation Analysis* activity. It will usually be conducted by interdisciplinary teams with expertise in *Aspect-Oriented Thinking* as well as architectural and implementation aspects of target *Systems*.

5.4.1 Requirement archetype development

The purpose of this sub-activity is to develop the *Requirement Archetypes* for a given *Aspect-Oriented Specification Archetype* and to provide a natural language description of their meaning. Note that these meanings are informal at this stage in the process of improving a *Problem Situation*. It is the development of *Implementation Rules* that will provide formal meaning to *Requirement Archetypes*.

5.5 Aspect-Oriented Specification development

The purpose of this activity is to form *Aspect-Oriented Specifications* for systems required to implement decisions made during the *Problem Situation Analysis* activity. This involves the development of *Specification Domain Models* applicable to the specification of a new *Systems*, followed by the actual *Requirements*.

The development of *Aspect-Oriented Specifications* will need to be conducted by individuals or teams that have some expertise in each of the domains involved, as well as *Aspect-Oriented Thinking*. For example, the development of an *Aspect-Oriented Specification* for a new aircraft will require the expertise and knowledge of a wide range of disciplines. While most of this expertise and knowledge will be developed within the context of each separate discipline, it must be woven together in various ways to specify particular aspects of the new aircraft. This will involve the development of *Specification Domain Models* that are unique to the new aircraft, along with a set of requirements that specify how these and applicable *Knowledge Domain Models* are to be used to build the aircraft. While this may seem onerous, it should be noted that most modelling effort will relate to the development and maintenance of the separate, and autonomous, *Knowledge Domain Models* involved. This effort can be provided by domain experts who have little or no expertise in *Aspect-Oriented Thinking*.

5.6 Implementation modelling

The purpose of this activity is to develop an *Implementation Model* (Section 4.8) which comprises an integrated set of *Implementation Rules* that can be used to construct a new system in accordance with an *Aspect-Oriented Specification*.

The development of *Implementation Models* will require expertise in applicable implementation methods, practices and technology. For example, *Implementation Rules* that involve the development of complex artifacts, may rely on the principles of *Systems Engineering* and as such, will need to be developed by people familiar with the *Systems Engineering* approach as well as the associated standards and tools. Similarly, *Implementation Rules* that involve the generation of software source code will need to be developed by people with detailed knowledge of software architecture, frameworks and programming languages.

5.7 Aspect-Oriented Specification implementation

The purpose of this activity is to implement an *Aspect-Oriented Specification*. This is achieved by executing the *Implementation Rules* of an *Implementation Model* associated with the *Aspect-Oriented Specification Archetype* to which the *Aspect-Oriented Specification* complies.

Implementation Rules can invoke manual operations or the use of tools such as programming language compilers, power-tools and bulldozers. They can also invoke the use of *Model Compilers* which process *Domain Models*, *Aspect-Oriented Specifications* and *Aspect-Oriented Specification Archetypes* to automatically generate systems as described in Section 4.8.3.

5.8 Continuous learning and improvement

Following the implementation of an *Aspect-Oriented Specification*, the process of *Aspect-Oriented Thinking* returns to the *Problem Situation Analysis* activity described in Section 5.2. This forms a continuous process in which each iteration will relate to the development and use of *Systems* to:

- learn about a *Problem Situation*,
- improve a *Problem Situation*, or to
- create, modify, operate and retire other *Systems*.

The *Aspect-Oriented Thinking* process depicted in Figure 5.1 therefore forms a continuous process of learning and improvement within a *Problem Situation*.

5.9 Comparison with traditional life-cycle models

Organisations that adopt the *Aspect-Oriented Thinking* approach will need to change the way they conceptualise the life-cycle of software and other systems. For example, traditional life-cycle models such as the *Waterfall* [*Royce*, 1970] and *Spiral* [*Boehm*, 1988] models as well as the widely practiced *agile* approaches, rely on separating concerns in a single dominant dimension. Concerns such as *Requirements, Architecture, Design, Implementation* and *Testing* are often considered within discrete, often repeated, phases of a project that produce *concern-specific* artifacts such as *Requirements Specifications*, *Design Descriptions* and software source code. Ordering amongst these phases is used to ensure that implementations comply with designs and that designs satisfy requirements.

Concerns related to the process of *Aspect-Oriented Thinking*, on the other hand, are separated into multiple dimensions, each appropriate to a particular perspective or context. That is, the concept of traditional life-cycle phases such as requirements, architectural design and implementation, is just one possible dimension to separating concerns within the *Aspect-Oriented Thinking* approach. For example, Figures 5.1 and 5.2 show the following four dimensions of separate concerns related to the *Aspect-Oriented Thinking* process.

- **Aspect-Oriented Thinking activity.** The activities depicted in Figure 5.1 represent the most important dimension to separating concerns within the *Aspect-Oriented Thinking* process. They define what needs to be done in order to apply the approach.
- **Learning and decision making.** In some situations it will be beneficial to consider and discuss *Aspect-Oriented Thinking* as a continuous process of learning and change. Horizontal partitioning of the activities depicted in Figure 5.1 shows how the process of *Aspect-Oriented Thinking* can be considered in terms of *learning, decision making* and *decision implementation*.
- **Disciplinarity.** It might also be appropriate to discuss the *Aspect-Oriented Thinking* process in terms of the disciplines involved. Vertical partitioning of activities depicted in Figure 5.1 shows how the process of *Aspect-Oriented Thinking* can be considered in terms of the disciplines involved in using the approach.

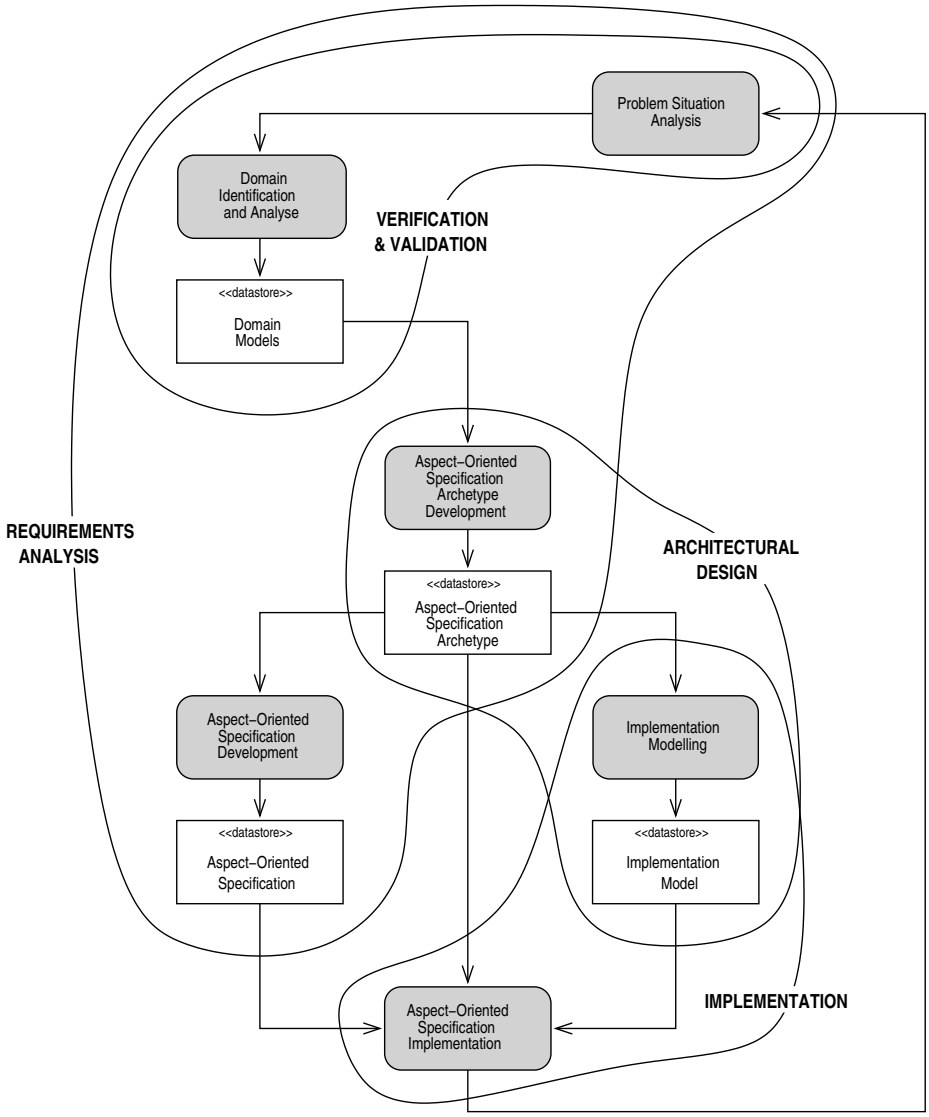

Figure 5.2: An activity diagram depicting the process of *Aspect-Oriented Thinking* and how the activities involved correspond to traditional systems engineering life-cycle activities of *Requirements Analysis, Architectural Design, Implementation, Verification* & *Validation*.

- **Traditional life-cycle processes.** Figure 5.2 depicts a fourth dimension to separating concerns based on the traditional life-cycle processes described in ISO/IEC 15288 [*International Organization for Standardization*, 2002]. For example, the *Implementation modelling* and *Aspect-Oriented Specification Implementation* activities could be considered as part of a traditional *Implementation* process. Similarly, the *Aspect-Oriented Specification Archetype Development* and *Implementation modelling* activities could be considered as part of a traditional *Architectural Design* process.

In summary the traditional system engineering life-cycle activities of requirements analysis, architectural design, implementation and others do not dominate the *Aspect-Oriented Thinking* process. They represent just one of a number of dimensions that can be used to consider the process.

5.10 Documentation

Traditional documents such as *Requirements Specifications* and *Design Descriptions* are not appropriate when using the *Aspect-Oriented Thinking* approach to system development. Because the approach centres around the development and use of artifacts depicted in Figures 5.1 and 5.2, the following documents are suggested:

- **Domain Model Description (DMD).** A DMD will contain a detailed description of an individual *Domain Model*. Because *Domain Models* are autonomous, each DMD can stand-alone and will represent a reusable asset. For example, each *Domain Model* presented in Appendix B could form the basis of a separate DMD.
- **Specification Archetype Description (SAD).** An SAD will contain a detailed description of a particular *Aspect-Oriented Specification Archetype*. For example, the *LAMP Aspect-Oriented Specification Archetype* presented in Appendix C could form the basis of an SAD.
- **Implementation Model Description (IMD).** An IMD will contain a detailed description of a specific *Implementation Model* and will be associated with a particular *Aspect-Oriented Specification Archetype*. For example, the *LAMP Implementation Model* presented in Appendix E could form the basis of an IMD.
- **Aspect-Oriented Specification (AOS).** An AOS will contain a detailed description of an *Aspect-Oriented Specification* for a particular system. Each AOS will reference a number of separate DMDs and a specific SAD. For example, the *Product Management System Aspect-Oriented Specification* presented in Appendix D could form the basis of an AOS that references DMDs that contain descriptions of the *Domain Models* presented in Appendix B and a SAD that contains a description of the *LAMP Aspect-Oriented Specification Archetype* described in Appendix C.

  Note that an AOS is a complete specification for a new system and will not only specify functional and performance requirements, but also requirements

regarding architecture and implementation. It is also possible to form several AOSs for a new system. For example an AOS might be formed to specify the development of a new system, while others might be formed to specify the development of maintenance and operating procedures for the new system.

Note that the emphasis here is not on documentation for its own sake, but on the formalisation of reusable intellectual assets. The DMDs, SADs and IMDs, in particular, will be relatively stable and highly reusable artifacts and would therefore lend themselves to thorough formalisation and management. This may take the form of traditional printed documents or, more likely, an on-line repository managed using an appropriate toolset. This subject is taken up in Section 7.5.2.

5.11 Conclusion

In this chapter I have described the process of *Aspect-Oriented Thinking* in terms of necessary activities. The process, in conjunction with the *Conceptual Model* described in Chapter 4, forms an approach that can be used, in a continuous fashion, to develop systems needed to learn about and improve a *Problem Situation*.

Part

III

Proof of Concept and Conclusion

Chapter 6

Proof-of-Concept

Contents

6.1 Introduction 88
6.2 Problem situation analysis 88
6.2.1 Contact management user requirements 89
6.2.2 Product management user requirements 90
6.2.3 Security user requirements 91
6.2.4 Constraints 92
6.3 Domain identification and analysis 92
6.3.1 Knowledge domain identification 92
6.3.2 Language modelling 94
6.3.3 Knowledge domain modelling 94
6.4 Aspect-Oriented Specification Archetype development . . . 95
6.5 Aspect-Oriented Specification development 96
6.5.1 Specification domain models 96
6.6 Implementation modelling 98
6.7 Implementation 99
6.8 Conclusion 99

6.1 Introduction

In Chapters 4 and 5 I presented the conceptual and process models of *Aspect-Oriented Thinking*. As discussed in Section 3.3.2.1, the next step in the planned research is to demonstrate the mechanics and viability of *Aspect-Oriented Thinking* in order to support a case for its future evaluation and subsequent adoption within an industrial context.

In this chapter and the supporting Appendices B to G, I demonstrate the mechanics and viability of *Aspect-Oriented Thinking* by using it to develop a complete software application. The decision to use a software case study was based on the diversity of concerns involved and the consequential ability to demonstrate important properties of *Aspect-Oriented Thinking* throughout the entire system life-cycle, from *concept* through to *implementation* and *operation*. In particular, a software case study will demonstrate the effectiveness of separating concerns into autonomous domains, the use of appropriate languages to model each domain, and the development of valuable reusable intellectual assets.

Note that Appendices B to G contain a series of very detailed models which form an integral part of the *Aspect-Oriented Thinking Proof-of-Concept*. In most cases, they do not need to be read in their entirety. Rather, they should be treated as a reference that readers can *drill into*, if required, to gain a thourough understanding of a particular facet of the *Aspect-Oriented Thinking* approach. It should also be noted that, because the models were developed without any tool support, they may contain minor errors. However, this should not detract from the effectiveness and instructive nature of the Proof-of-Concept.

Cross-References

Extensive use of cross-referencing is made throughout the presentation of the *Product Management System proof-of-concept*. In addition, a *Proof-of-Concept Index* of elements involved is provided on page 385.

6.2 Problem situation analysis

The *Problem Situation* I am dealing with here is that industry partners are unlikely to become involved in the future development and evaluation of *Aspect-Oriented Thinking* unless they develop some degree of confidence in the viability of the approach.

It has been decided that this *Problem Situation* can be improved, though not

eliminated, by using *Aspect-Oriented Thinking* to develop a complete software application. The results of this demonstration will be used to support proposals for future industry-based evaluation and development of *Aspect-Oriented Thinking*.

The chosen software application is the *Product Management System*, a web-based system which will allow employees of small stores to manage a database of products they sell. The application will also allow customers to view the products available for sale. User requirements[1] for the *Product Management System (PMS)* are enumerated below. Note that these requirements are intentionally informal, incomplete and inconsistent to reflect the kinds of requirements that can be expected from an end user of a software system.

6.2.1 Contact management user requirements

(a) The *Product Management System* shall record the following details about people and organisations with which the business interacts. These are collectively referred to as *contacts*.

- Name
- Address
- Telephone number
- Mobile Phone number
- Email address

(b) Users of the *Product Management System* shall be able to:

- create contact records
- edit contact records
- delete contact records

(c) The *Product Management System* shall record the following details about notes that can be attached to contact records:

- Date and Time the note was created
- Text message

(d) Users of the *Product Management System* shall be able to:

- create and attach notes to contact records

[1] *Wiegers* [2003, page 490] defines user requirements as 'User goals or tasks that users must be able to perform with a system, or statements of the user's expectations of system quality.'

- edit note records
- delete note records

(e) The *Product Management System* shall record relationships between contacts of the following kinds:

- <contact A> *owns* <contact B>
- <contact A> *manages the finances of* <contact B>
- <contact A> *manages shipping for* <contact B>
- <contact A> *manages human resources for* <contact B>
- <contact A> *handles product returns for* <contact B>

(f) Users of the *Product Management System* shall be able to:

- create relationships between any two contacts
- change the type of a relationship
- delete relationships

6.2.2 Product management user requirements

(a) The *Product Management System* shall record the following details about each product sold by the business:

- Name
- Description
- Manufacturer
- Category

(b) Users of the *Product Management System* shall be able to:

- create product records
- edit product records
- delete product records

(c) The *Product Management System* shall record the following details about each category of product sold by the business:

- Name
- Notes

(d) Users of the *Product Management System* shall be able to:

- create product category records
- edit product category records
- delete product category records

(e) Products are often available in a number of variations such as color or material. Each variant will often be differently priced. The *Product Management System* shall record the following information about variants of each product:

- Description
- Price

(f) Users of the *Product Management System* shall be able to:

- create product variant records
- edit product variant records
- delete product variant records

6.2.3 Security user requirements

(a) Users shall be authenticated before they are permitted to use the *Product Management System*.

(b) Users must be registered with the *Product Management System* before they can be authenticated.

(c) Users shall register with the *Product Management System* by providing a user name and password.

(d) The user names of each registered user shall be unique.

(e) Each user shall be assigned one or more of the following roles:

- Staff
- Customer

(f) Users assigned the *Staff* role shall be able to create, edit and delete any information recorded by the *Product Management System*.

(g) Users assigned the *Customer* role shall only be able to read product information.

6.2.4 Constraints

(a) Users shall interact with the software via a standard web browser such as Firefox.

(b) The software shall be implemented to run on the *Linux, Apache, MySQL* and *PHP* (LAMP) web-based client-server architectural framework.

The LAMP framework has been chose because, despite its popularity, it supports few of the well established principles of software engineering. This provides scope, within this proof-of-concept, to demonstrate the way in which *Aspect-Oriented Thinking* can facilitate appropriate 'engineering' of a software system despite poorly engineered implementation technologies.

6.3 Domain identification and analysis

The purpose of domain identification and analysis is to develop and capture the knowledge required to understand and improve a *Problem Situation*. This is achieved by identifying, modelling and verifying aspects of a given *Problem Situation*. The following sections contain the results of domain analysis for the *Product Management System*.

6.3.1 Knowledge domain identification

Domain identification is a difficult task. The domains involved in a *Problem Situation* are likely to evolve as the situation itself evolves and understanding of it matures. In the case of software and systems development, an initial set of *Knowledge Domains* can be identified by considering the categories of available user requirements.

The following initial set of *Knowledge Domains* can be identified based on the user requirements presented in Section 6.2. *Specification Domains* will emerge during the development of the *Product Management System Aspect-Oriented Specification*.

- **Contact Management Domain.** The *Contact Management* problem space domain will deal with the subject matter of *Contacts*, *Notes* and *Relationships*. It will ignore all other concerns, by only modelling the functional requirements stated in Section 6.2.1.

- **Product Management Domain.** The *Product Management* problem space domain will deal with the subject matter of *Products*, *Product Categories*, *Product Variants* and *Manufacturers*. It will ignore all other concerns, by only modelling the functional requirements stated in Section 6.2.2.
- **Security Domain.** The *Security Support* domain will deal with the general subject matter of *Security*. It will not deal with specific security requirements for the *Product Management System*. Instead, it will describe a reusable body of knowledge that can be used to specify application specific security policies including the one required for the *Product Management System* (Section 6.2.3). Like all *Domains*, it will be completely autonomous and will not refer to the subject matter of any other *Domain*, including that of the *Contact* and *Product Management Domains*.
- **User Interface (UI) Domain.** The *UI Support Domain* will deal with the subject matter of user interfaces. Like the *Security Domain*, the *UI Domain* will not deal with specific user interface requirements for the *Product Management System*. Instead, it will describe a reusable body of knowledge that can be used to specify any user interface, including one for the *Product Management System*.

The following initial set of *Architectural* and *Implementation* domains can be identified from the constraints stated in Section 6.2.4.

- **LAMP Domain.** The LAMP *architectural* domain will describe the high level organisation of an application developed using the *LAMP* framework. It will include elements such as *Web Browser, Server* and *Database*.
- **Linux Domain.** The *Linux Implementation Domain* will represent the subject matter of the Linux open-source operating system. Note that the Linux domain will not be modelled because it is well understood and documented in many academic, industrial and user documents. In addition, the Linux domain is not a software system that is called or executed in any way. Rather, it represents knowledge about the Linux operating system that will be used during construction of the *Product Management System* application.
- **Apache Domain.** The *Apache Implementation Domain* will represent knowledge about the Apache web server. The subject matter of this domain will be used to implement the server component of the *Product Management System* architecture.
- **Structured Query Language (SQL).** The *SQL Implementation Domain* will represent knowledge about standard SQL. All database interaction re-

quired by the *Product Management System* will be conducted using concepts captured by the SQL domain.

- **MySQL Domain.** The *MySQL Implementation Domain* will be used to implement database concepts represented by the *SQL Domain*. Note that the autonomous nature of domains means that the *MySQL Relational Database* domain model can be replaced by one that represents another database such as *PostgreSQL*.
- **PHP Domain.** The *PHP Implementation Domain* will represent knowledge about the *PHP5* programing language. This knowledge will be used to implement parts of the *Contact Management, Product Management, Security* and *User Interface Domains*.

6.3.2 Language modelling

In most cases, *Aspect-Oriented Thinking* knowledge domains will be modelled as instances of the common meta-models identified in Section 4.4.2.5. Modelling of domains for the *proof-of-concept* is no exception. Table 6.1 lists each of the *proof-of-concept Knowledge Domain Models*, their meta-models and references.

Note that the *Informal Block Diagram* meta-model is not one of the common models identified in Section 4.4.2.5. As such, it needs to be developed before the *LAMP Framework* domain model can be developed. Figure 6.1 shows a class diagram that depicts the simple Block Diagram meta-model used in this *proof-of-concept*.

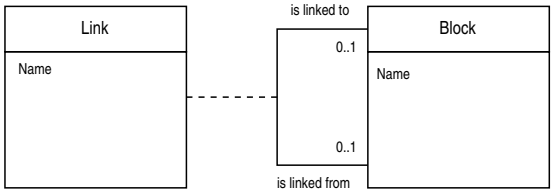

Figure 6.1: The Block Diagram Meta-Model.

6.3.3 Knowledge domain modelling

As mentioned in Section 5.3.3, the purpose of domain modelling is to develop a clear understanding of the subject matter of a specific *Domain*. Appendix B contains the *Knowledge Domain Models* which will be used in the formation of the *Product Management System Aspect-Oriented Specification*. These models relate to the *Domains* identified above.

Table 6.1: *Proof-of-Concept* knowledge domains and meta-models

Domain	**Meta-Model**	**Reference**
Contact Management	$^{X}_{T}$UML	Section B.2
Product Management	$^{X}_{T}$UML	Section B.3
Security	$^{X}_{T}$UML	Section B.4
User Interface	$^{X}_{T}$UML	Section B.5
LAMP	Informal Block Diagram (Figure 6.1)	Section B.6
Linux	English	*Kernel.Org Organization, Inc.* [2006]
Apache	English	*Apache Software Foundation* [2006]
Structured Query Language	English	*International Organization for Standardization* [2005b]
MySQL	English	*MySQL AB* [2006]
PHP	English	*The PHP Group* [2006]

6.4 Aspect-Oriented Specification Archetype development

Aspect-Oriented Specification Archetypes describe how subject matter captured in *Knowledge Domain Models* can be used to form *Aspect-Oriented Specifications* for systems that share a common architecture.

For the purposes of this proof-of-concept, the *Aspect-Oriented Specification Archetype* contained in Appendix C has been developed using the *Knowledge Domain Models* contained in Appendix B. This archetype comprises a set of *Requirement Archetypes* that can be used to form *Aspect-Oriented Specifications* for software systems that share an architecture based on the *Linux, Apache, MySQL, and PHP* (LAMP) framework.

6.5 Aspect-Oriented Specification development

As mentioned in Section 4.6, the purpose of developing an *Aspect-Oriented Specification* is to state requirements for a new *System*. For the purposes of this *proof-of-concept*, an *Aspect-Oriented Specification* for the *Product Management System* has been formed in accordance with the *Aspect-Oriented Specification Archetype* contained in Appendix C.

The completed *Product Management System Aspect-Oriented Specification* is contained in Appendix D. Figure 6.2 shows a diagram which depicts the contents of the specification using the notation described in Appendix A.

6.5.1 Specification domain models

The *LAMP Aspect-Oriented Specification Archetype* requires conformant *Aspect-Oriented Specifications* to include instances of the $\frac{X}{T}$*UML* domain model to specify data requirements, an instance of the *User Interface* domain model to specify user interface requirements, and an instance of the *Security* domain model to specify security requirements. For the purposes of this proof-of-concept, the following *Domain Models* have been formed or reused to satisfy these needs.

- ***Product Management*** **domain model (Section B.3) and** ***Contact Management*** **domain model (Section B.2).** These *Knowledge Domain Models* are instances of the $\frac{X}{T}$*UML* domain model and have been used to specify data requirements for the *Product Management System*. The data to be managed by the *Product Management System* has been specified using *Requirements* represented by *Requirement Set RS01* and *Requirement Set RS02* depicted in Figure 6.2. The details of these *Requirements* are contained in Section D.2.

- ***PMS User Interface*** **domain model (Section D.5).** User interface requirements for the PMS were not provided in the original user requirements listed in Section 6.2. It was left, at that time, to Human-Computer Interaction (HCI) and other user interface *domain experts* to elicit user interface requirements and to capture them in the form of user interface *Domain Models*.

  The *PMS User Interface* domain model is a *Specification Domain Model* that has been formed to specify user interface requirements for the *Product Management System*. It is an instance of the *User Interface* domain model and has been depicted as a series of *page layouts* shown in Figures D.3 to D.19.

  The way in which the user interface controls the *Product Management System* and displays its data is specified by *Requirements* that reference subject

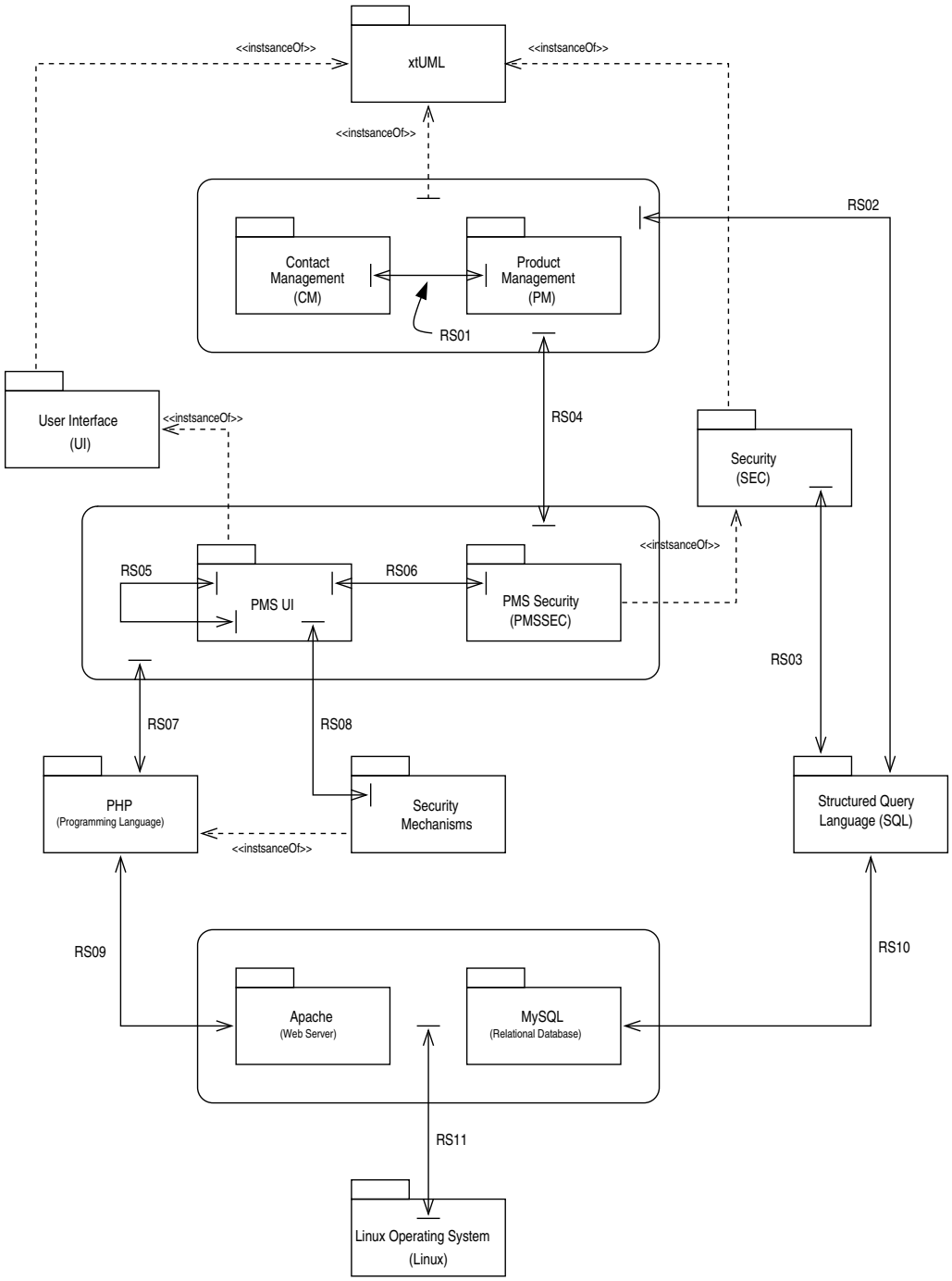

Figure 6.2: Package diagram depicting the *Aspect-Oriented Specification* for the *Product Management System*.

matter within the *PMS User Interface* domain model, *Contact Management* domain model and *Product Management* domain model as indicated by *Requirement Set RS04*. Interaction among the pages is, likewise, specified by *Requirements* as indicated by *Requirement Set RS05*. The details of these *Requirements* are contained in Section D.6.

- ***PMS Security* domain model (Section D.3).** This is a *Specification Domain Model* that has been formed to specify the *Roles* and *Permissions* that are applicable to the *Product Management System*. It is an instance of the *Security* domain model and formalises the user requirements stated in Section 6.2.3. *Requirements* that reference subject matter from the *PMS Security* domain model, *Contact Management* domain model and *Product Management* domain model are represented by *Requirement Set RS04* and are used to identify those data items that are subject to a specific security policy. The details of these *Requirements* are contained in Section D.4.

6.6 Implementation modelling

Implementation modelling involves the development of *Implementation Rules* which describe how an appropriately formed *Aspect-Oriented Specification* can be implemented to create a new *System*.

Appendix E contains an *Implementation Model* that can be used to generate a complete LAMP application from any *Aspect-Oriented Specification* that conforms to the *LAMP Aspect-Oriented Specification Archetype* described in Appendix C. As such, this and other *Implementation Models* are highly reusable.

Note that $\frac{X}{T}$*UML Domain Models* referenced by the *Product Management System Aspect-Oriented Specification* do include state models. As such, no action language is used. This is possible because the *Product Management System* is a simple *Create, Read, Update, Delete* (CRUD) application with no business rules other than those captured in associations among domain model elements. As a result, the *LAMP Implementation Model* described in Appendix E is limited to the generation of CRUD type applications. More complex applications involving the use of the *Object Constraint Language* [*Object Management Group*, 2006] and *State Models* will require extentions to the *LAMP Implementation Model*.

6.7 Implementation

The implementation of an *Aspect-Oriented Specification* requires the systematic execution of all *Implementation Rules* associated with an appropriate *Implementation Model*. In the case of this proof-of-concept, this means the execution of *Implementation Rules* contained in the *Implementation Model* described in Appendix E against *Requirements* stated in the *Product Management System Aspect-Oriented Specification*.

Note that while the *Implementation Model* described in Appendix E contains *Implementation Rules* that are intended to be executed manually (i.e., by programmers), it is possible to automate all of the rules. In practice, however, it may only be cost effective to automate repetitive or tedious parts of the *Implementation Model*.

The results of implementing the *Product Management System Aspect-Oriented Specification* are presented in Appendices F and G

6.8 Conclusion

In this chapter and Appendices B to G, I have presented a demonstration of *Aspect-Oriented Thinking* by using the approach to develop a complete software system. The case study has demonstrated how several autonomous domain models can be developed to understand a *Problem Situation*, and to then specify and implement a software system that aims to improve the situation. The case study has clearly demonstrated the significance of separating concerns into autonomous *Domain Models*. In particular it has demonstrated the ability to produce well-engineered specifications for a complete software application, despite the adoption of implementation technology that may not be well-engineered or based on well established principles of software and systems engineering.

In conclusion, *Aspect-Oriented Thinking* is certainly a viable approach to the development of (at least) web-based database software systems. It will be the subject of future research and argument to demonstrate broader applicability of the approach.

Chapter 7

Summary and Conclusion

Contents

7.1 Introduction 102
7.2 Summary of contribution 102
7.3 Limitations of contribution 104
7.4 Overall conclusion 105
7.5 Recommendations for future work 106
7.5.1 Bootstrapping the approach 106
7.5.2 Tool support 107
7.5.3 Additional software system examples 107
7.5.4 Industrial-scale evaluation 108
7.5.5 Research in a interdisciplinary context 109
7.6 Closing remarks 109

7.1 Introduction

The aim of this thesis was '*to develop, model, demonstrate and evaluate an aspect-oriented interdisciplinary approach to engineering that can be used to build and operate systems required to understand and improve Problem Situations of all kinds*'. This aim has been achieved through the development and demonstration of *Aspect-Oriented Thinking*.

In this chapter, I summarise the contribution made by this thesis including key elements of *Aspect-Oriented Thinking*, results of the *Proof-of-Concept* reported in Chapter 6, and current limitations of the approach. I then draw final conclusions and make recommendations regarding future research and development of the approach.

7.2 Summary of contribution

In Chapter 2, I argued that the success of software engineering and engineering in general can only be understood by considering all of the stakeholders involved, their separate needs and how they evolve with time. This view stands in contrast to common definitions of project success that centre around the satisfaction of budgetary and schedule constraints. It is also a view that led to the development of *Capability Dynamics* (Section 2.5), *Aspect-Oriented Systems Engineering* (Section 2.7) and ultimately the idea of *Aspect-Oriented Thinking*.

In Chapter 3, I proposed a plan for *Aspect-Oriented Thinking* research and development. This was achieved by first developing and using a novel conceptual model of software engineering research to describe and compare various research approaches described in the software engineering literature. The model was then used to help describe the *Aspect-Oriented Thinking* research approach adopted in this thesis. The approach comprises two phases. The first has been completed and its results have been reported in this thesis. The second phase constitutes future research including industrial evaluation and development of *Aspect-Oriented Thinking* as discussed in Section 7.5 below.

In Chapter 4, I described the language of *Aspect-Oriented Thinking* using a UML class model. The key concepts involved are:

- **Problem Situations.** A problem situation can be defined as an environment of interacting systems in which at least one person perceives a need for change. In Chapter 4, I recast Software and systems engineering as a process

of bringing about changes which improve *Problem Situations* rather than simply satisfying a set of stated requirements and constraints.

- **Systems.** Systems are either human, man-made or natural. Man-made systems are created, modified, operated and retired in order to learn about *Problem Situations* and to realise changes that aim to improve *Problem Situations*.

- **Domains.** *Problem Situations* are considered as a set of autonomous subject matters called *Domains*. Each *Domain* represents a separate view or perspective of a *Problem Situation* or the *Systems* involved. Examples within the context of software-intensive systems development include human-machine interfaces, security, safety, architecture, functional requirements, management and quality assurance. Within a broader context of the environment, for example, domains might include various aspects of climate, pollution, economics, biology and social science.

- **Domain Models.** *Domain Models* are formed, using appropriate languages, to develop and capture knowledge about *Domains*. These models can be used to learn about a subject matter of concern within a *Problem Situation* or in the implementation of changes using *Aspect-Oriented Specifications, Aspect-Oriented Specification Archetypes* and *Implementation Models*.

- **Aspect-Oriented Specifications.** *Aspect-Oriented Specifications* are formed to specify systems. These systems include those required to learn about and improve a *Problem Situation*, as well as those required to create, modify, operate and retire other systems.

  Each *Aspect-Oriented Specification* comprises a set of *Specification Domain Models* and a set of *Requirements*. *Specification Domain Models* capture information such as interface specifications, security policies and performance constraints for a required system.

  Requirements make use of subject-matter captured in *Knowledge* and *Specification Domain Models* to specify some property of a required *System*. They are formed in accordance with *Requirement Archetypes* defined in an *Aspect-Oriented Specification Archetype*.

- **Aspect-Oriented Specification Archetypes.** An *Aspect-Oriented Specification Archetype* comprises an integrated set of *Requirement Archetypes*. These archetypes describe the way in which subject matter captured in a set of autonomous *Knowledge Domain Models* can be used to form an *Aspect-Oriented Specification* which fully specifies the functional, performance, architectural, implementation and other requirements for a required *System*.

Because they cover all aspects of a *System*, including architectural and implementation concerns, each *Aspect-Oriented Specification Archetype* can be reused in the formation of *Aspect-Oriented Specifications* for many systems that share the same architecture.

- **Implementation Models.** A given *Aspect-Oriented Specification Archetype* will be associated with one or more *Implementation Models*. These models comprise a set of *Implementation Rules* that can be followed by people or automated tools to implement any *Aspect-Oriented Specification* formed in accordance with the given *Aspect-Oriented Specification Archetype*.

In Chapter 5, I described a continuous process that makes use of the above concepts to learn about and improve *Problem Situations*. In Chapter 6, along with associated Appendices B to G, I demonstrated the use of this process to develop a complete software application using the *Aspect-Oriented Thinking* approach.

7.3 Limitations of contribution

Limitations of the contribution made by the research presented in this thesis, relate mainly to the *Proof-of-Concept* presented in Chapter 6. While the *Proof-of-Concept* demonstrates the mechanics of *Aspect-Oriented Thinking* and the viability of the approach for developing software systems, it has the following limitations:

- The *Proof-of-Concept* involved the development of a functionally simple, small and well defined software application within a controlled environment. An appropriate evaluation of *Aspect-Oriented Thinking* will need to involve the development of a large software-intensive system within a real world *Problem Situation* of co-evolving man-made, natural and social systems.
- The results produced so far provide limited evidence that the *Aspect-Oriented Thinking* approach can lead to improved productivity along with reduced defect rates and rework. However, results reported by *Stien* [2003, 2006] and *Boughton* [2006] in relation to the application of $\frac{X}{T}$*UML*, indicate that such improvements are possible within an industrial context.
- The results do not demonstrate reuse of *Domain Models*, *Aspect-Oriented Specification Archetypes* and *Implementation Models*. For example, it should be possible to reuse many of the *Domain Models* described in Chapter 6 to form *Product Management System Aspect-Oriented Specifications* in accordance with archetypes other than the *LAMP* archetype presented in Appendix C. These other archetypes could, for example, make use of

architectural domains such as the *Java Platform, Enterprise Edition* [*Sun Microsystems*, 2006a] or *Asynchronous JavaScript and XML (AJAX)* [*Garrett*, 2006].

Similarly, it should be possible to reuse the *LAMP Aspect-Oriented Specification Archetype* to form *Aspect-Oriented Specifications* for systems other than the *Product Management System*.

- The detail involved in the *Proof-of-Concept* presented in Chapter 6 and Appendices B to G highlights some of the tediousness of applying the *Aspect-Oriented Thinking* approach without software tool support. In any real-world context, it will be necessary to make use of tools that support the development of *Aspect-Oriented Thinking* artifacts and to manage the *Aspect-Oriented Thinking* process.

More generally, the *Proof-of-Concept* described in Chapter 6 does not demonstrate applicability of *Aspect-Oriented Thinking* in *Problem Situations* that require the expertise of a broad range of disciplines to build systems other than software. For example, *Problem Situations* involving the environment, will require the expertise of many disciplines including engineering, chemistry, biology, economics, the law and social science to understand, build, modify, operate and/or retire a broad range of man-made and natural systems.

A summary of plans for future work to address the above limitations is presented in Section 7.5.

7.4 Overall conclusion

The overall conclusion that can be drawn from the research presented in this thesis is that *Aspect-Oriented* software development techniques can be combined with *Systems Thinking* ideas to form a novel approach that has the potential to help people from multiple disciplines work together to learn about and improve a broad range of *Problem Situations*.

It is the form of *Aspect-Oriented Specifications* and the way in which they are developed that differentiates *Aspect-Oriented Thinking* from other interdisciplinary approaches to *Problem Situation* improvement. While other approaches such as *System Dynamics* and *Soft Systems Methodology* can be used to learn about a *Problem Situation*, they rely on the use of a single approach across all aspects of a *Problem Situation* and offer little in the way of an approach that can be used to act on any decisions made.

Aspect-Oriented Thinking, on the other hand, provides a framework within which these and other existing approaches to the development of knowledge and expertise, can be used to specify, build and operate the systems necessary to learn about and improve entire *Problem Situations*.

Because *Aspect-Oriented Thinking* is based on the notion of initially separating each aspect of a *Problem Situation* into autonomous and often discipline based domains, it might provide a way to 'Bridging the disciplinary divides' [*Grafton et al.*, 2005].

7.5 Recommendations for future work

This thesis contains the results of research conducted in *Phase One* of the research approach presented in Chapter 3. That is, this thesis contains the theoretical foundations of *Aspect-Oriented Thinking* and a demonstration of its potential value. The aim of *Phase Two* of the research approach is to the continue the development of *Aspect-Oriented Thinking* and to support its adoption by industry.

In this section, I outline a series of specific activities for future research and development of *Aspect-Oriented Thinking* that might be undertaken within the framework established in Chapter 3 (see Figure 3.6 on page 45). The aim of these activities is to address the limitations identified in Section 7.3.

7.5.1 Bootstrapping the approach

Because *Aspect-Oriented Thinking* is a general approach to improving *Problem Situations*, it should be possible to improve *Aspect-Oriented Thinking* itself using the approach. This could be achieved by developing a set of *Knowledge Domain Models* based on aspects of the conceptual and process models presented Chapters 4 and 5. Various *Aspect-Oriented Specification Archetypes* could then be formed to enable users to specify and implement various systems to support the development, practice and evaluation of *Aspect-Oriented Thinking*.

Within this context, this thesis can be viewed as part of a *bootstrapping* process that aims to establish a situation in which future *Aspect-Oriented Thinking* research, development and evaluation is conducted using the *Aspect-Oriented Thinking* approach itself.

7.5.2 Tool support

Software tools will need to be developed to support the application of *Aspect-Oriented Thinking*. At this early stage, three phases of development are envisaged. The first phase will aim to develop a database system to store artifacts produced during the process of *Aspect-Oriented Thinking*. This phase could be developed using the *LAMP Aspect-Oriented Specification Archetype* presented in Appendix C and application specific *Domain Models* including one based on the *Aspect-Oriented Thinking Conceptual Model* presented in Chapter 4.

In phase two, a set of graphical model editors will be incorporated for important modelling languages including *eXecutable and Translatable UML ($^{X}_{T}$UML)* and *System Dynamics*.

Phase three will aim to provide a framework that can be used to automate the execution of *Implementation Models*. This framework will, itself, be a collection of *Aspect-Oriented Thinking* artifacts including a set of *Domain Models* that deal with repository navigation, code generation and reporting. An *Aspect-Oriented Specification Archetype* will be developed to describe how these models can be assembled to form *Aspect-Oriented Specifications* for applications that automate the execution of *Implementation Models*.

7.5.3 Additional software system examples

In order to demonstrate the reusable nature of *Knowledge Domain Models* and *Aspect-Oriented Specification Archetypes*, it will be necessary to develop a second *Proof-of-Concept*. This second *Proof-of-Concept* will comprise a set of new *Application Domain Models* and a new *Aspect-Oriented Specification* formed in accordance with the existing *LAMP Aspect-Oriented Specification Archetype* presented in Appendix C. The new specification will be implemented using the existing *LAMP Implementation Model* presented in Appendix E.

In addition to the above, a new set of architectural and implementation *Domain Models*, along with a new *Aspect-Oriented Specification Archetype* and *Implementation Model*, will be developed to capture knowledge required to build applications using the *armidale* architecture presented in *Flint and Boughton* [2002, Reproduced in Appendix I]. These artifacts will be used to form a new *Aspect-Oriented Specification* for implementation of the *Product Management System* using the new architecture. This will demonstrate the reuse of *Application Domain Models* with different architectures.

7.5.4 Industrial-scale evaluation

The initial plan for future evaluation of *Aspect-Oriented Thinking* involved the development of a software system within an industrial context (Section 3.3.2.2). During the later stages of the research reported in this thesis I concluded that this might not be possible given the novel nature of the approach. It would be difficult to convince potential industry collaborators to participate in a live evaluation of *Aspect-Oriented Thinking* on anything other than a small, non-critical project.

Recent discussions with colleagues have suggested the possibility of demonstrating and evaluating *Aspect-Oriented Thinking* by using the approach to redevelop a large-scale real-world government information system within a research context.

The proposal is to gain sponsorship from an organisation to form a development team to undertake the re-development of an existing project within a research context using leading-edge research ideas, including *Aspect-Oriented Thinking*. The team is likely to be significantly smaller than the commercial equivalent and would comprise researchers with significant industrial experience and understanding of *Aspect-Oriented Thinking*, post-graduate students, other researchers and representatives from the sponsoring organisation (possibly as full-time post-graduate research students). The team would need to have access to documentation including plans, specifications and studies related to the real project, as well as the real-world users, sponsors and other stakeholders.

The aim of such a project would be to answer the following questions by comparing, where appropriate, results of the research based project with those of the corresponding commercial project.

- Can *Aspect-Oriented Thinking* help people manage complexity?
- Can the benefits reported by *Stien* [2003, 2006] and *Boughton* [2006] in relation to the application of $\frac{X}{T}$*UML*, be realised with respect to the use of *Aspect-Oriented Thinking* within the broader context of a large software-intensive system development project?
- Can *Aspect-Oriented Thinking* help people from multiple disciplines work together?
- Can *Aspect-Oriented Thinking* lead to the development and use of system capabilities that are better aligned with stakeholder needs?
- Can *Aspect-Oriented Thinking* improve an organisation's ability to respond to evolving stakeholder needs?

- Can effective tool support, including automated implementation of *Aspect-Oriented Specifications*, be developed and used? Techniques such as the *Technology Acceptance Model* [*Davis*, 1989] could be used to help answer this question.

Note that an industrial-scale project, such as that described above, will also facilitate the conduct of other research related to the development of large software-intensive systems. This research could include that related to project management, process improvement, measurement, architecture, human-computer interfaces, data mining and high-performance computing. It will also provide an important case study for teaching and small group projects within a university context.

7.5.5 Research in a interdisciplinary context

By including people from non-computing disciplines within the above project team, it will be possible to evaluate the effectiveness of *Aspect-Oriented Thinking* as an interdisciplinary approach to understanding and improving *Problem Situations*. Other disciplines that might be involved include sociology, psychology, teaching, economics, accounting, the law and philosophy.

7.6 Closing remarks

With further research and development I believe that *Aspect-Oriented Thinking* will prove itself to be of broad applicability and a viable interdisciplinary approach to learning about, and improving complex *Problem Situations* of all kinds.

It is my intention to continue with this work for as long as I am able and so long as the approach, and any derivatives from it, remain relevent.

The End

Part

IV

Appendices

Appendix

A

Aspect-Oriented Specification notation

Contents

A.1 Introduction 114
A.2 Domain model notation 114
A.3 Individual requirement notation 114
A.4 Requirement set notation 114
A.5 Domain model set notation 116
A.6 Conclusion 117

A.1 Introduction

Within this appendix, I describe a notation that can be used to simplify the representation of *Aspect-Oriented Specifications*. The notation has been derived from elements of the *HiGraph notation* [*Harel*, 1988] and the *Unified Modelling Language (UML)* [*Object Management Group*, 2005].

A.2 Domain model notation

Each *Domain Model* involved in an *Aspect-Oriented Specification* is represented using a UML *Package* icon. For example, the packages depicted in Figure A.2 represent *Domain Models* named *Domain A*, *Domain B* and *Domain C*

A.3 Individual requirement notation

The notation adopted for representing *Aspect-Oriented Specifications* allows individual *Requirements* to be represented using *dashed arrows*. For example, the representation of *Requirement Rq01*, depicted in Figure 4.4 on page 64, can be simplified as depicted in Figure A.1. In this diagram, a *dashed line* is used to represent *Requirement Rq01*. Arrow heads at each end of the line point to elements referenced by the *Requirement* (a specific *Data Store* within a data flow model and the *RDBMS Domain Model* in this case).

Figure A.1 also depicts *Requirement Rq02* which references three *Domain Model Elements*: a *Concept* within a concept map, a *Process* within a data flow model, and a *Class* within an $\frac{X}{T}$UML model.

A.4 Requirement set notation

The representation of complex *Aspect-Oriented Specifications* can be simplified by using *solid double-ended arrows* to represent *Requirement Sets* (Section 4.6.5) as depicted in Figure A.2.

For example, *Requirement Set RS01* depicted in Figure A.2 represents all *Requirements* that reference elements in both *Domain Model A* and *Domain Model B*. The *bars* at each end of *Requirement Set RS01*, indicate that the *Requirements* represented by *Requirement Set RS01* involve elements *within* the *Domain Models* rather than the models themselves.

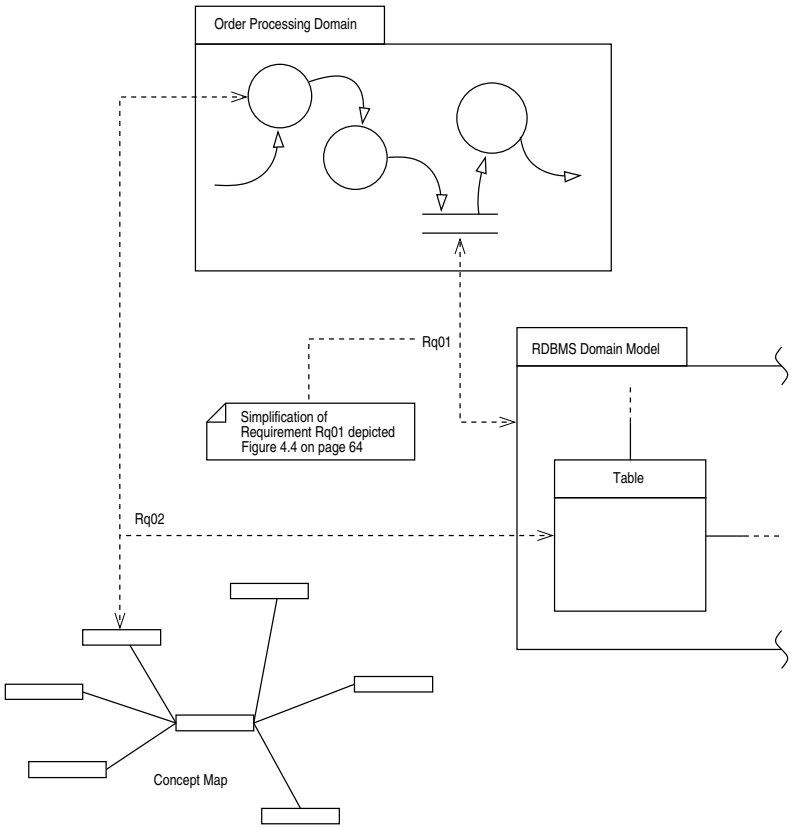

Figure A.1: Examples of the simplified notation that can be used to depict individual *Aspect-Oriented Specification Requirements*.

Requirement Set RS02, respresents *Requirements* that reference *Domain Model B* (in its entirety) and elements of *Domain Model C*. It is the way in which one end of *Requirement Set RS02* is connected the the border of the package that represents *Domain Model B*, that indicates involvement of the entire *Domain Model*. The *bar* at the other end indicates the involvement of elements within *Domain Model C*.

Note that *Requirement Rq01*, depicted in Figure A.2, represents a single *Requirement* that references both *Domain Model C* and *Domain Model D* in their entirety. It is depicted as a *dashed double-ended arrow*.

Figure A.3 shows a more complete example of an *Aspect-Oriented Specification* depicted using the *Requirement Set* notation.

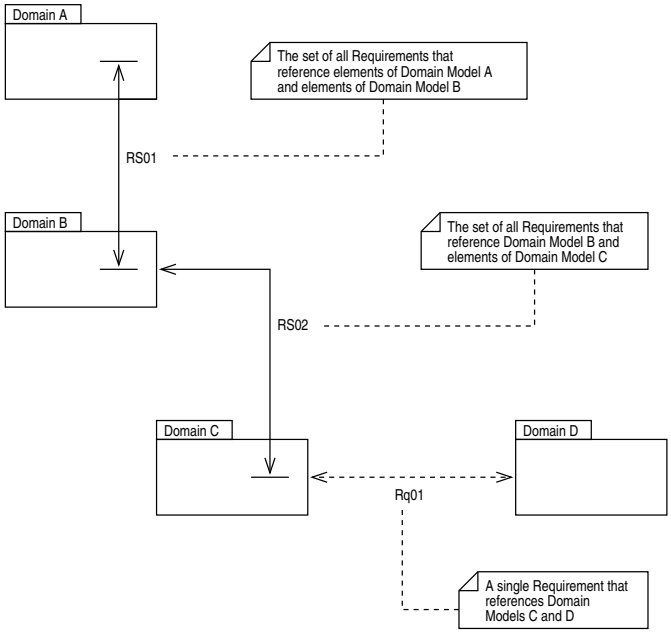

Figure A.2: Examples of the notation that can be used to represent *Requirement Sets*.

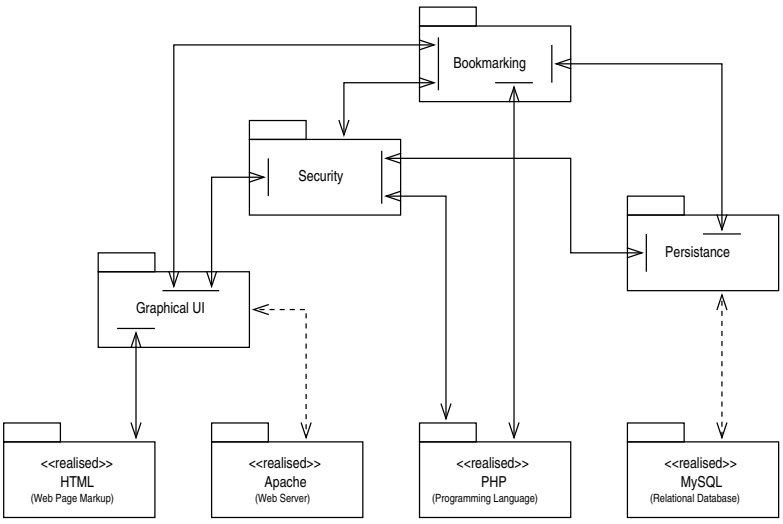

Figure A.3: An *Aspect-Oriented Specification* depicting *Requirement Sets* involved in a simple software application.

A.5 Domain model set notation

Because of the large number of *Domains* likely to be involved in a significant *Aspect-Oriented Specification*, and because *Requirements* often involve subject matter captured in more than one *Domain Model*, the *Requirement Set* notation described above is extended along the lines of Harel's HiGraph notation [*Harel*,

1988]. The HiGraph notation is a similar to Venn Diagrams and uses rounded boxes to enclose elements of a set. In the case of *Aspect-Oriented Thinking*, set elements are *Domain Models* as depicted in Figure A.4. Arrows between packages that represent *Domain Models* now depict *Requirement Sets* between *sets of Domains* as well as individual *Domains*.

Figure A.4 shows the diagram shown in Figure A.3 redrawn and simplified using the *Domain Set* notation. The *bar* at the top end of *Requirement Set RS01* is used to indicate that the *Requirement Set* comprises *Requirements* that reference elements of *Domain Models* within the rounded box. That is, *Requirement Set RS01* comprises *Requirements* that reference elements of the *Bookmarking* and/or *Security Domain Models* as well as elements of the *Graphical UI Domain Model*. Similarly, *Requirement Set RS03* comprises *Requirements* that reference elements of the *Bookmarks* and/or *Security Domain Models* as well elements of the *Persistence Domain Model*.

Requirement Set RS04 comprises *Requirements* that reference elements of the *Bookmarks* and/or *Security Domain Models* as well and the entire *PHP Domain Model*.

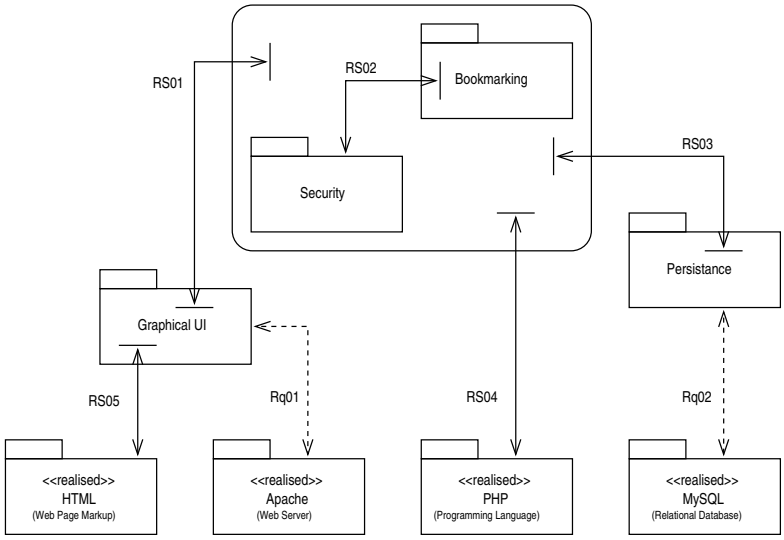

Figure A.4: The *Aspect-Oriented Specification* depicted in Figure A.3, simplified using the domain set notation.

A.6 Conclusion

Within this appendix, I have described a notation that can be used to represent *Aspect-Oriented Specifications*. Examples of its use can be found on Chapters 4

and 6.

Appendix

B

Proof of concept - *Knowledge Domain Models*

Contents

B.1 Introduction 120
B.2 *Contact Management* domain model 120
B.2.1 Domain specific types 120
B.2.2 Domain classes 121
B.3 *Product Management* domain model 123
B.3.1 Domain specific types 123
B.3.2 Domain classes 124
B.4 *Security* domain model 125
B.4.1 Domain specific types 125
B.4.2 Domain classes 126
B.5 *User Interface* domain model 128
B.5.1 Domain specific types 128
B.5.2 Domain classes 130
B.6 *LAMP Framework* domain model 134
B.7 $\frac{X}{T}$*UML* domain model 136
B.8 *Linux Operating System* domain model 136
B.9 *PHP Web Programming Language* domain model 136
B.10 *Structured Query Language* domain model 136
B.11 *APACHE Web Server* domain model 136
B.12 *MySQL Relational Database* domain model 136
B.13 Domain model verification 137
B.14 Conclusion 137

B.1 Introduction

The first set of artifacts to be produced when using the *Aspect-Oriented Thinking* approach are *Knowledge Domain Models* (Section 4.4.2). Each of these models capture stakeholder understanding of a specific subject matter applicable to the *Problem Situation* at hand. In this appendix, I present models which have been developed for the *Knowledge Domains* identified in Section 6.3.1. These models will later be used to form an *Aspect-Oriented Specification* for the *Product Management System*.

B.2 *Contact Management* domain model

The *Contact Management* domain model deals with data aspects of the user requirements stated in Section 6.2.1. Figure B.1 shows an $\frac{X}{T}$*UML* class diagram that depicts entities involved in the *Contact Management* domain and relationships among them.

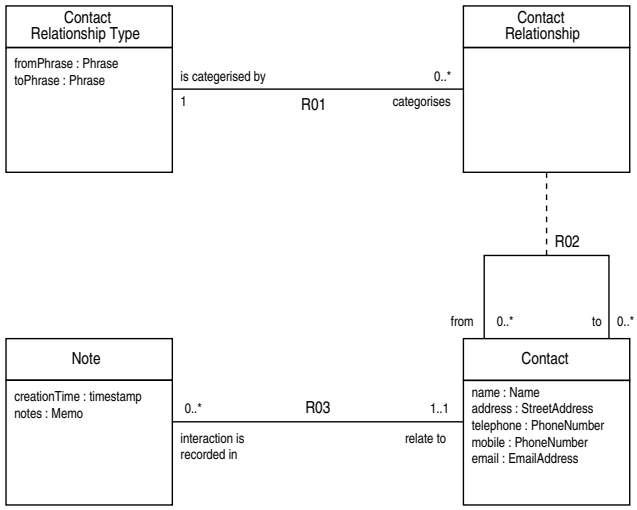

Figure B.1: Class diagram depicting concepts involved in the *Contact Management* domain.

B.2.1 Domain specific types

The following domain specific types are used throughout the *Contact Management* domain model.

type **Phrase**

A 30 character string used to describe the nature of relationships between contacts.

type **Memo**
A 200 character string used to store a block of text.

type **Name**
A 100 character string used to store the names of various data items such as *Contacts*.

type **StreetAddress**
A 200 character string used to store the complete street address of a *Contact*. Note that addresses are not decomposed into elements such as street, suburb and post code. It is up to the user to establish formatting conventions.

type **PhoneNumber**
A 50 character string comprising digits, '-' or '+' characters used to record telephone and mobile numbers. It is up to the user to establish formatting conventions such as the inclusion or otherwise of country codes.

type **EmailAddress**
A 100 character string representation of an email address of the form *name@domain*.

B.2.2 Domain classes

The *Contact Management* domain model comprises the following classes.

class **Contact**
Contacts are people and organisations. They are identified by name and can be contacted via a physical *Street Address* and various electronic means.

The *Contact* class has the following attributes.

attribute **name : Name**
The name of the *Contact*. No attempt is made to separate first and last names for individuals, or individuals from company names. It is up to the user to adopt a standardised naming convention such as '*lastName, firstName*'.

attribute **address : StreetAddress**
The physical street address of the *Contact*. No attempt is made to decompose the address. It is treated as a simple text string.

attribute **telephone : PhoneNumber**
The land line telephone number of the *Contact*.

attribute **mobile : PhoneNumber**
The mobile telephone number of the *Contact*.

attribute **email : EmailAdress**
The email address of the *Contact*.

class **Contact Relationship**
A *Contact Relationship* is used to record connections between contacts. For example, the people working for a company would be identified by creating *Contact Relationships* between each employee and the company.

The *Contact Relationship* class has no attributes.

class **Contact Relationship Type**
Each relationship between a pair of *Contacts* is characterised by a *Contact Relationship Type* which describes the relationship from the perspective of each *Contact* involved. The relationship between a parent and a child for example, would be described from the parent's perspective as *isParentOf*. From the child's perspective the same relationship could be described as *isChildOf*.

The *Relationship Type* class has the following attributes.

attribute **fromPhrase : Phrase**
The *fromPhrase* attribute is used to describe relationships of particular type from the perspective of the contact at the *from* end of *association* R02 (Figure B.1).

attribute **toPhrase : Phrase**
The *toPhrase* attribute is used to describe relationships of particular type from the perspective of the contact at the *to* end of *association* R02 (Figure B.1).

class **Note**
Notes are used to record interactions between a user and *Contacts* maintained by the *Product Management System*. These interactions deal with any aspect of the users business including sales, supply, product information and shipping.

The *Note* class has the following attributes.

attribute **creationTime: timestamp**
Records the date and time that the *Note* is created.

attribute **notes : Memo**
Record the text of a *Note*.

B.3 *Product Management* domain model

The *Product Management* domain model deals with data aspects of the user requirements stated in Section 6.2.2. Figure B.2 shows an $\frac{X}{T}$*UML* class diagram that depicts entities involved in the *Product Management* domain and relationships among them.

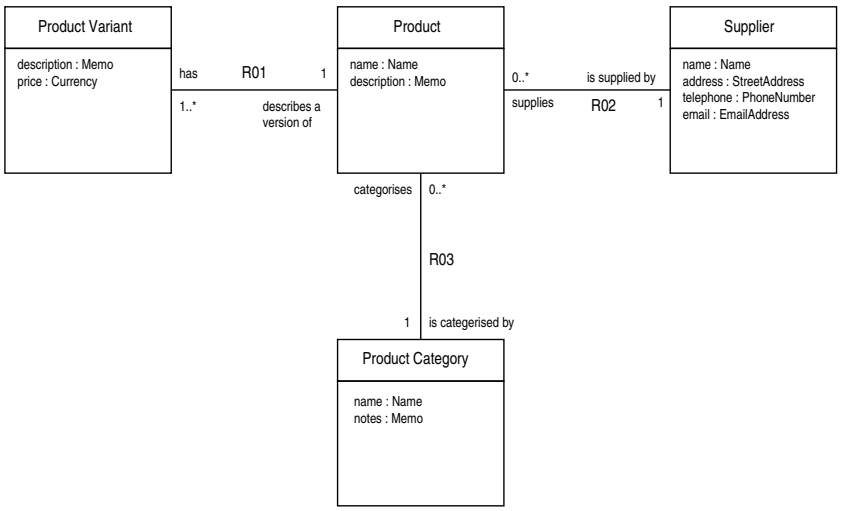

Figure B.2: Class diagram depicting concepts involved in the *Product Management* domain.

B.3.1 Domain specific types

The following domain specific types are used throughout the *Product Management* domain model.

type **Name**
A 100 character string used to identify various data items.

type **Memo**
A 200 character string used to store a block of text.

type **StreetAddress**
A 200 character string used to store the complete street address of a *Supplier*. Note that addresses are not decomposed into elements such as street, suburb and post code.

type **PhoneNumber**
A 50 character string comprising digits, '-' or '+' characters used to record

telephone and mobile numbers. It is up to the user to establish formatting conventions such as the inclusion or otherwise of country codes.

type **EmailAddress**
A 100 character string representation of an email address of the form *name@domain*.

type **Currency**
Used to record the cost of an item. This is a fixed point number that records dollars and cents between $0.00 and $1,000,000.00

B.3.2 Domain classes

The *Product Management* domain model comprises the following classes.

class **Product**
Products are the things sold by a business. The *Product* class is used to record basic information about a product. The *product Variant* class, described below, is used to record more detailed information including prices.

The *Product* class has the following attributes.

attribute **name : Name**
The name of the product. No attempt is made to structure product names. It is up to the user to adopt a standardised naming convention.

attribute **description : Memo**
This is a more detailed description of the product. It does not deal with product variants as these are handled by the *ProductVariant* class described below.

class **Product Variant**
Products sold by a business are often available in different versions. For example, an office chair might be available in different colours or fabrics. Each variant will often have a different price. The purpose of the *ProductVariant* class is to record details of each product variant along with their prices.

The *Product Variant* class has the following attributes.

attribute **description : Memo**
Describes a particular variant of a product.

attribute **price : Currency**
The price of the product variant.

class **Supplier**
Products are supplied to a business by *Suppliers* which can be either companies or individuals.

The *Supplier* class has the following attributes.

attribute **name : Name**
The name of the supplier.

attribute **address : StreetAddress**
The physical street address of the *Supplier*.

attribute **telephone : PhoneNumber**
The land line telephone number of the *Supplier*.

attribute **email : EmailAdress**
The email address of the *Supplier*

class **Product Category**
Products sold by a business are grouped into categories. For example, all of the different tyres sold by an auto repair company would be categorised as *Tyres*. The *ProductCategory* class is used to identify and describe such categories.

The *Product Category* class has the following attributes.

attribute **name : Name**
The name of the *Product Category*.

attribute **notes : Memo**
A more complete description of the *Product Category*. This may include general conditions of sale that apply to an entire category of products.

B.4 *Security* domain model

The *Security* domain model defines various classes which can be instantiated to specify security policies for a wide range of software applications including the *Product Management System*. Figure B.3 shows an $\frac{X}{T}$*UML* class diagram that depicts entities involved in the *Security* domain and relationships among them.

B.4.1 Domain specific types

The following domain specific types are used throughout the *Security* domain model.

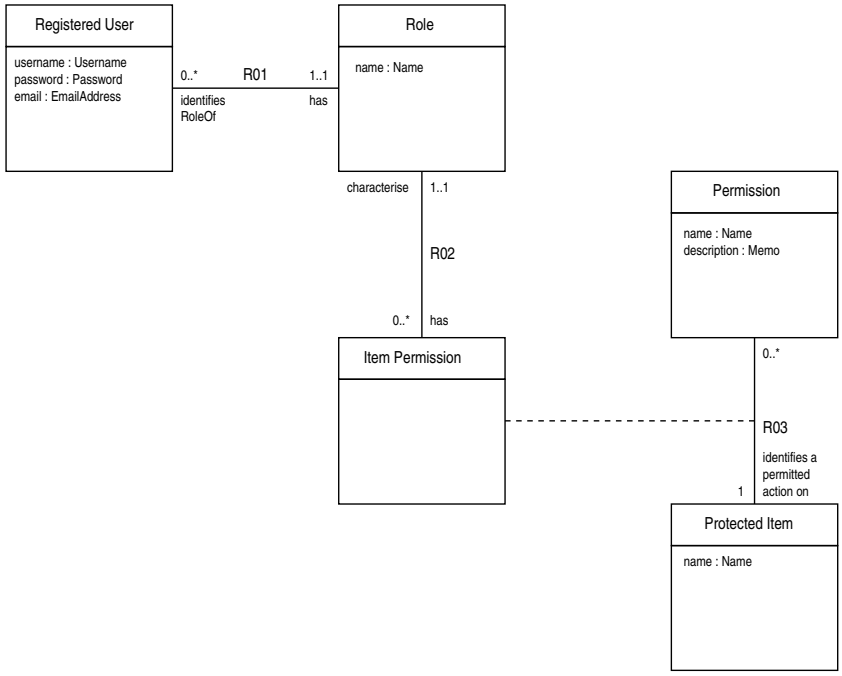

Figure B.3: Class diagram depicting concepts involved in the *Security* domain.

type **Username**
A 30 character string used to name registered users.

type **Password**
A 100 character string used to record user passwords.

type **Name**
A 30 character string used to identify various data items such a security *Roles* and *Permissions*.

type **Memo**
A 300 character string used to store a block of text.

B.4.2 Domain classes

The *Security* domain model comprises the following classes.

class **Registered User**
A *Registered User* class is a person who is able to manipulate *Protected Items* within an application subject to a particular security policy. These policies are defined in terms of *Roles* and *Permissions* described below.

The *Registered User* class has the following attributes.

attribute **username : Username**
The name of the *Registered User* used for authentication purposes.

attribute **password : Password**
Authentication is achieved by checking a *Registered User's* username and password against a list of registered users. Note that the actual authentication mechanism used is not dealt with by the *Security* domain. It is an implementation concern left to another domain.

class **Protected Item**
A *Protected Item* is something that is subject to protection by policies established using the *Security* domain. For example, *Products* in the *Product Management* domain might be protected such that they can only be deleted by users acting in a specific *Role* such as '*administrator*'.

The *Protected Item* class has the following attributes.

attribute **name : Name**
The name of the *Protected Item*.

class **Permission**
Instances of the *Permission* class are used to enumerate the permissions that *Registered Users* of a specific application might have. They will vary from application to application. For example, the *Permissions* applicable to the *Product Management System* are defined as part of the *PMS Security* domain model (Section D.3).

The *Permission* class has the following attributes.

attribute **name : Name**
The name of the *Permission*.

attribute **description : Memo**
A description of the *Permission* for explanatory purposes.

class **Item Permission**
An *Item Permission* is used to specify a *Permission* that a given *Role* has with respect to a *Protected Item*. Each *Role* will usually be linked with a set of *Item Permissions*.

The *Item Permission* class has no attributes.

class **Role**
A *Role* identifies the set of *Permissions* a *Registered User* has with respect to *Protected Items*. For example, the permissions that are assigned to users engaged in the role of *administrator* in a university will be very different to those assigned to users engaged in the role of *student*. Instances of the

Role class are used to enumerate the roles applicable to a specific application. Like, the *Permissions* class described above, they will vary from application to application. The roles to be used in the *Product Management System* are defined as part of the *Product Management System Security domain model* (Section D.3).

The *Role* class has the following attributes.

attribute **name : Name**
The name of the *Role*. Examples might include 'Administrator' and 'User'.

B.5 *User Interface* domain model

The *User Interface* domain model describes elements that can be used to specify user interfaces. Figure B.4 shows an $\frac{X}{T}$*UML* class diagram that depicts entities involved in the *User Interface* domain and relationships amongst them.

Note that each instance of the *User Interface* domain model represent the specification of a single user interface page (or form). A complete user interface specification will comprises a set of *Pages* linked together using *Requirements* (Section 4.6.2) that reference elements of such pages. For example, a *Requirement* may reference a *Button* on one *Page* and another complete *Page* to specify a requirement for the second page to be displayed when the *Button* is *clicked*.

B.5.1 Domain specific types

The following domain specific types are used throughout the *User Interface* domain model.

type **DisplayText**
A 100 character string used to specify static text that is displayed on a user interface page.

type **Name**
A 50 character string used to identify various data items such a *Page*.

type **InputText**
Strings, up to 100 characters in length, that are entered by users.

type **Memo**
Blocks of text, up to 300 characters in length, that are entered by users.

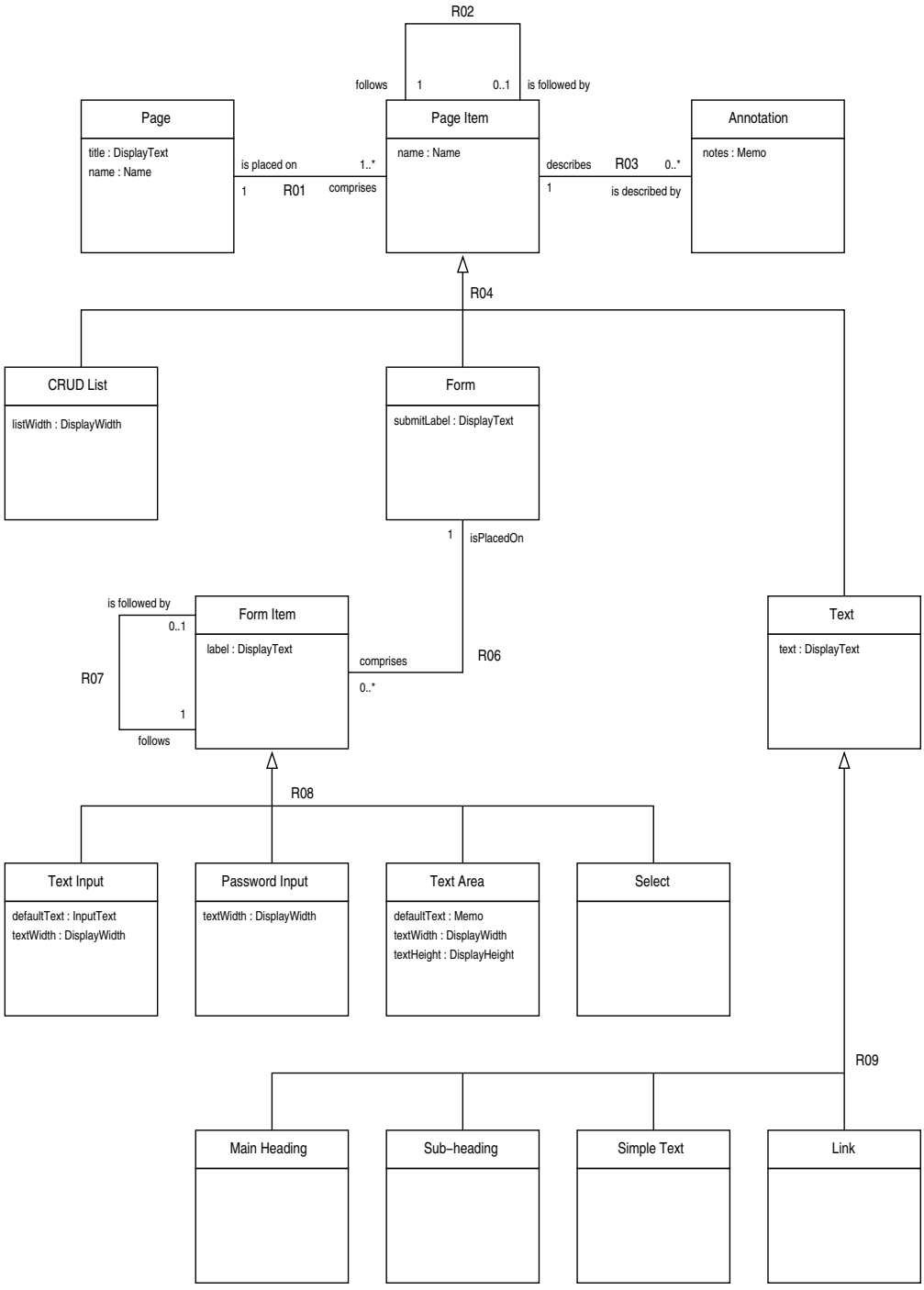

Figure B.4: Class diagram depicting concepts involved in the *User Interface* domain.

type **DisplayWidth**
An natural number used to specify the width of user interface elements.

type **DisplayHeight**

An natural number used to specify the height of user interface elements.

B.5.2 Domain classes

The *User Interface* domain model comprises the following classes.

class **Page**
Complete user interfaces comprise one or more *Pages* that each contain a set of user interface elements such as buttons, links and text.

The *Page* class has the following attributes.

attribute **title : DisplayText**
The text displayed as a title for the page. In the case of a web implementation, for example, this text might be displayed in the browser title bar. In the case of a Java Swing implementation, this text might be displayed in a window title bar.

attribute **name : Name**
A short page name used for referencing purposes.

class **Page Item**
Each user interface page comprises a set of *Page Items* such as text and buttons.

Note that it is the policy of the *User Interface* domain defined here, that individual *Page Items* cannot be placed at specific locations within a *Page*. Instead, each *Page Item* is placed on a page in a vertical sequence starting at the top of the page and arranged down the page in order of creation (association *R02* in Figure B.4).

The *Page Item* class has the following attributes.

attribute **name : Name**
A short name used for referencing purposes.

class **Annotation**
Annotations are used to informally describe the purpose or content of a *Page Item*. Note that a more formal meaning for each *Page Item* will be provided when *Requirements* (Section 4.6.2) that reference such pages and elements of other *Domain Models* are specified.

Note that this incremental approach to formalising the meaning of each user interface element, allows developers to manipulate the structure of a user interface without having to be, initially, overly formal or concerned with the detailed modelling of data to be displayed or actions to be invoked.

These things are the concern of other separate and autonomous domains and *Requirements* among them.

The *Annotation* class has the following attributes.

attribute **notes : Memo**
An informal description of the linked *Page Element*. This might, for example, include a description of data displayed in a *CRUD List* or actions to be taken when a *Link* is *clicked*.

class **Text**
The *Text* class is a generalisation of a set of user interface *Page Items* that comprise a single piece of text.

The *Text* class has the following attributes.

attribute **text : DisplayText**
The string displayed in the *Text* item.

The *Text* class has the following sub-classes.

class **Main Heading**
A text heading displayed in a large font.

The *Main Heading* class has no attributes apart from those inherited from the *Text* class.

class **Sub-Heading**
A text Heading displayed using a font smaller than that used to display *Main Headings*.

The *Sub-Heading* class has no attributes apart from those inherited from the *Text* class.

class **Simple Text**
A piece of text displayed using a font smaller than that used to display *Sub-Headings*.

The *Simple Text* class has no attributes apart from those inherited from the *Text* class.

class **Link**
A piece of text, displayed using the font used to display *Simple Text* items. *Links* can be *clicked* by the user to invoke some action. *Links* are usually referenced by *Requirements* that specify the actions to be performed when the text is *clicked*.

The *Link* class has no attributes apart from those inherited from the *Text* class.

class **CRUD List**
CRUD List items allow users to *Create, Read, Update* and *Delete* (CRUD) items displayed in a list. They will comprise a list of items and associated *Links* or buttons that can be used to create, update, read and delete any of the listed items.

Note that within the *User Interface* domain model, it is only the above *concept* of a *CRUD List* that is defined. Details of what items are displayed, how they are displayed and how each of the *Create, Read, Update* and *Delete* actions are implemented is not defined by this domain. It is left to appropriate *Requirement Archetypes* and other *Domain Models* to define such things.

See the *raClassList* requirement archetype (Section C.2.6) for an example of how *CRUD Lists* are used in the *Product Management System*.

The *CRUD List* class has the following attributes.

> *attribute* **listWidth : DisplayText**
> The width of the *Crud List*.

class **Form**
Forms comprise a set of *Form Items* (text input boxes and select lists) which accept data input from users. Each *Form* also includes a *Submit Button* which is *clicked* when a user has finished entering data. The input data is then processed in some way.

Note that within the *User Interface* domain model, it is only the above *concept* of a *Form* that is defined. Details of how *Forms* are implemented and how data entered by the user is processed is not defined by this domain. It is left to appropriate *Requirement Archetypes* and other *Domain Models* to define such things.

See the *raFormHandledByPhpFunction* requirement archetype (Section C.2.5) for an example of how *Forms* are used in the *Product Management System*.

The *Form* class has the following attributes.

> *attribute* **submitLabel : DisplayText**
> Each *Form* has a submit button that is labeled with a short piece of text. This attribute is used to set the *Submit Button* label.

class **Form Item**
Form Item is an abstract class that generalises a set of user interface items that can be used on a form to accept input from users.

Note that it is the policy of the *User Interface* domain defined here, that individual *Form Items* cannot be placed at specific locations within a *Form*.

Instead, each *Form Item* is placed on a form in a vertical sequence starting at the top of the form and arranged down the form in order of creation (association *R07* in Figure B.4).

The *Form Item* class has the following attributes.

attribute **label : DisplayText**
A label that describes the *Form Item*. This label is displayed on a *Form*, next to the *Form Item* itself. For example, a *Text Input* item for a person's name may be labeled with the string 'Name'.

The *Form Item* class has the following sub-classes.

class **Text Input**
The *Text Input* class allows a user to enter and edit a single line of text.

The *Text Input* class has the following attributes.

attribute **defaultText : InputText**
The default text value. This string is automatically placed into the input field when a *Text Input* item is first displayed. It can then be edited by the user. Note that the *defaultText* attribute can be an empty string.

attribute **textWidth : DisplayText**
The width of the text input field.

class **Password Input**
The *Password Input* class allows a user to enter and edit a single line of text. It is similar to the *Text Input* class described above except the text item is displayed as a sequence of asterisks such as '*****'. That is, the text entered by a user cannot be read by the user or other observers.

The *Password Input* class has the following attributes.

attribute **textWidth : DisplayWidth**
The width of the password input field.

class **Text Area**
The *Text Area* class allows a user to enter and edit multiple lines of text.

The *Text Area* class has the following attributes.

attribute **defaultText : Memo**
The default text value. This string is automatically placed into the input field when a *Text Area* item is first displayed. It can then be edited by the user. Note that the *defaultText* attribute can be an empty string.

attribute **textWidth : DisplayWidth**
The width of the text input field.

attribute **textHeight : DisplayWidth**
The height of the text input field.

class **Select**
The *Select* class allows the user to select an item from a list of items. It will usually be implemented as a pull-down list in which each item is a text string.
Note that within the *User Interface* domain model, it is only the above *concept* of a *Select* that is defined. Details of what items are listed, the default item and what actions are executed when an item is selected are all defined by appropriate *Requirement Archetypes* and other *Domain Models*.
See the *raSelectItemCreatesLink* requirement archetype (Section C.2.11) for an example of how a *Select* is used in the *Product Management System*.
The *Select* class has no attributes.

B.6 *LAMP Framework* domain model

Figure B.5 shows a block diagram which depicts a high-level *Domain Model* of the reusable *Linux, Apache, MySQL* and *PHP* (LAMP) architectural framework [*O'Reilly*, 2006]. This *Domain Model* will be used to form specifications that satisfy the *Product Management System* architectural constraints stated in Section 6.2.4.

The purpose of this model is two-fold. Firstly, it provides an overview of the structure of software built on the LAMP platform. Secondly, it can be referenced by *Requirements* that specify where and how software modules are installed and executed in a LAMP application.

Elements of the LAMP framework are briefly described below. Full details can be found at the *onLamp* web site [*O'Reilly*, 2006].

Web Browser
A conventional web browser, such as *Mozilla.org* [2006], that implements the World Wide Web Consortium HTML 4.01 Specification [*World Wide Web Consortium*, 1999a]. The LAMP framework does not specify the operating system on which the browser might run.

Wide Area Network
The internet which allows communication between browsers and the Apache web server using the Hyper-Text Transfer Protocol (HTTP) protocol [*World Wide Web Consortium*, 1999b].

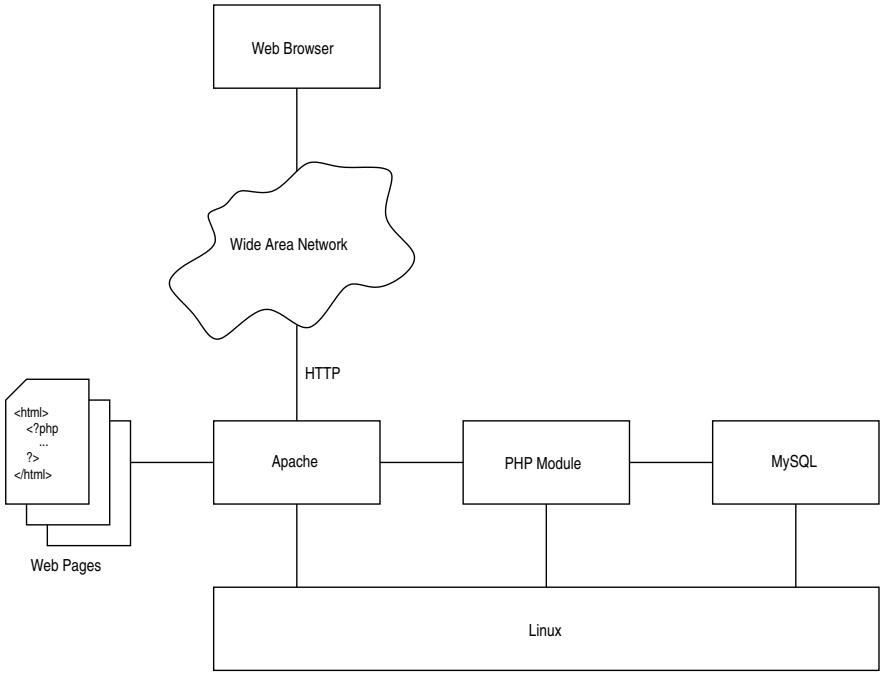

Figure B.5: Block diagram depicting elements of the *LAMP Framework* domain.

Apache
Apache [*Apache Software Foundation*, 2006] is a web server that processes HTTP requests from web browsers. Within the LAMP framework used here, the Apache browser will respond to *GET* and *PUT* by processing requested web pages using the PHP module. This module executes any PHP scripts embedded in the requested web page before it is sent, if required, to the browser.

Web Pages
HTML, HTML with embedded PHP and PHP files that are used by the APACHE component to build and serve HTML pages in response to HTTP requests from browsers.

PHP Module
The PHP Module [*The PHP Group*, 2006] is used by Apache to execute PHP scripts embedded in web pages that are requested by web browsers.

MySQL
An SQL 2000 compliant relational database system [*MySQL AB*, 2006]. PHP scripts embedded in web pages are able to execute SQL queries using the MySQL component of the LAMP framework.

Linux
An open-source UNIX like operating system [*Kernel.Org Organization, Inc.*,

2006] upon which Apache, PHP and MYSQL run.

B.7 $\frac{X}{T}$*UML* domain model

The model of $\frac{X}{T}$UML described by *Mellor and Balcer* [2002] has been adopted for the purposes of the *LAMP Aspect-Oriented Specification Archetype*.

B.8 *Linux Operating System* domain model

The model of *Linux* described by the *Kernel.Org Organization, Inc.* [2006] has been adopted for the purposes of the *LAMP Aspect-Oriented Specification Archetype*.

B.9 *PHP Web Programming Language* domain model

The model of *PHP* described by *The PHP Group* [2006] has been adopted for the purposes of the *LAMP Aspect-Oriented Specification Archetype*.

B.10 *Structured Query Language* domain model

The model of *SQL* described by the *International Organization for Standardization* [2005b] has been adopted for the purposes of the *LAMP Aspect-Oriented Specification Archetype*.

B.11 *APACHE Web Server* domain model

The model of *Apache* described by the *Apache Software Foundation* [2006] has been adopted for the purposes of the *LAMP Aspect-Oriented Specification Archetype*.

B.12 *MySQL Relational Database* domain model

The model of *MySQL* described by *MySQL AB* [2006] has been adopted for the purposes of the *LAMP Aspect-Oriented Specification Archetype*.

B.13 Domain model verification

In this *Product Management System* proof-of-concept example, static verification was conducted using model inspections. That is, no tools support was used. While dynamic verification of those *Domains Models* developed using $\frac{X}{T}$UML is possible using tools such as BridgePoint [*Mentor Graphics*, 2006], no such verification was conducted.

B.14 Conclusion

In this appendix, I have demonstrated the use of *Aspect-Oriented Thinking* concepts to develop a set of autonomous *Knowledge Domain Models*. These models are applicable to development of the *Product Management System* introduced in Chapter 6.

Appendix

C

Proof of concept - *LAMP Aspect-Oriented Specification Archetype*

Contents

C.1 Introduction 141
C.1.1 Notation and Terminology 141
C.2 User interface related requirement archetypes 142
C.2.1 Requirement Archetype ‘raPageImplementedInPhp’ 142
C.2.2 Requirement Archetype ‘raLinkOpensPage’ 142
C.2.3 Requirement Archetype ‘raLinkAvailableWithRole’ 142
C.2.4 Requirement Archetype ‘raLinkCallsPhpFunction’ 143
C.2.5 Requirement Archetype ‘raFormHandledByPhpFunction’ . . 144
C.2.6 Requirement Archetype ‘raClassList’ 144
C.2.7 Requirement Archetype ‘raGroupClassList’ 146
C.2.8 Requirement Archetype ‘raTextInputUpdatesAttribute’ . . . 147
C.2.9 Requirement Archetype ‘raPasswordInputUpdatesAttribute’ 148
C.2.10 Requirement Archetype ‘raTextAreaUpdatesAttribute’ . . . 149
C.2.11 Requirement Archetype ‘raSelectItemCreatesLink’ 149
C.3 Security related requirement archetypes 151
C.3.1 Requirement Archetype ‘raSecurityPhpImplementation’ . . 151
C.3.2 Requirement Archetype ‘raSecureClass’ 151
C.4 Database related requirement archetypes 152
C.4.1 Requirement Archetype ‘raMysqlImplementation’ 152
C.4.2 Requirement Archetype ‘raPersistentClass’ 152
C.4.3 Requirement Archetype ‘raPersistentAssociation’ 152

C.4.4 Requirement Archetype 'raCorrespondingClasses' 153
C.5 Implementation specific requirement archetypes 153
C.5.1 Requirement Archetype 'raUiImplementedInPhp' 153
C.5.2 Requirement Archetype 'raPhpImplementedOnApache' . . . 154
C.5.3 Requirement Archetype 'raApacheImplementedOnLinux' . . 154
C.5.4 Requirement Archetype 'raMysqlImplementedOnLinux' . . 155
C.6 *Security Mechanisms* domain model 155
C.7 Conclusion 155

C.1 Introduction

Once a set of *Knowledge Domain Models* have been developed, they can be used to form *Aspect-Oriented Specifications* for systems that can be implemented and used to learn about and improve a *Problem Situation*. As discussed in Section 4.7 on page 67, *Aspect-Oriented Specifications* are formed in accordance with a *Aspect-Oriented Specification Archetypes* which describes the ways in which specific *Knowledge Domain Models* can be used to specify systems that share a particular architecture.

In this appendix, I present a reusable *Aspect-Oriented Specification Archetype* that has been developed using the *Knowledge Domain Models* contained in Appendix B. This archetype can be used to form *Aspect-Oriented Specifications* for database applications that share a common architecture based on the *Linux, Apache, MySQL* and *PHP (LAMP)* framework.

C.1.1 Notation and Terminology

In this appendix the following notation is used to identify *Domain Models* and *Domain Model Elements* that must be referenced by a *Requirement* that conforms to a given *Requirement Archetype*.

- <**referenceName**> **:** <**domainName[.elementName]**>

  This notation is used to identify a named reference to an *instance of* a *Domain Model* or *Domain Model Element*.

- <**referenceName**> **=** <**domainName[.elementName]**>

  This notation is used to identify a named reference to a specific *Domain Model* or *Domain Model Element*.

Requirements will often be identified using the form '<*requirement archetype name*> (Section X.Y) requirement'. For example, '*raClassList* (Section C.2.6) requirement' refers to a requirement that conforms to the *raClassList* requirement archetype.

C.2 User interface related requirement archetypes

C.2.1 Requirement Archetype 'raPageImplementedInPhp'

Description. The *raPageImplementedInPhp* requirement archetype is used to specify the implementation of a *UI.Page* using PHP.

References. Instances of the *raPageImplementedInPhp* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***thePage* : UI.Page** *(Section B.5)*

  The *Page* to be implemented in PHP.

- ***thePhpDomain* = PHP** *(Section B.9)*

  The *Domain Model* that represents knowledge about the PHP programming language.

C.2.2 Requirement Archetype 'raLinkOpensPage'

Description. The *raLinkOpensPage* requirement archetype is used to specify the use of a user interface *Link* to open a user interface *Page*.

References. Instances of the *raLinkOpensPage* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theLink* : UI.Link** *(Section B.5)*

  A *Link* that appears on a user interface *Page*.

- ***thePage* : UI.Page** *(Section B.5)*

  The *Page* that is to be opened when *theLink* is clicked.

C.2.3 Requirement Archetype 'raLinkAvailableWithRole'

Description. The *raLinkAvailableWithRole* requirement archetype is used to specify that a *Link* (Section B.5) shall only be visible to users who have a designated security *Role* (Section B.4). *Requirement Rq01*, depicted in Figure C.1, summarises the application of this *Requirement Archetype*.

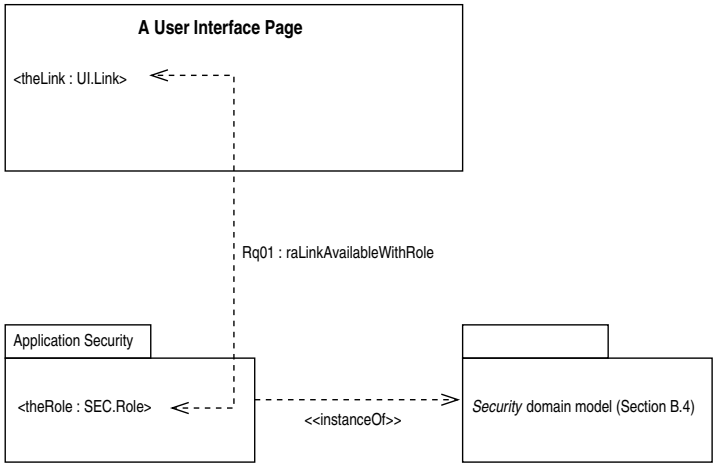

Figure C.1: Application of the *raLinkAvailableWithRole* requirement archetype. *Rq01* is a requirement for *theLink* to be only visible if the current user has *theRole*.

References. Instances of the *raLinkAvailableWithRole* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theLink* : UI.Link** *(Section B.5)*

  A *Link* that appears on a user interface *Page*.

- ***theRole* : SEC.Role** *(Section B.4)*

  A security *Role* that must be held by the user if *theLink* is to be visible to that user.

C.2.4 Requirement Archetype 'raLinkCallsPhpFunction'

Description. The *raLinkCallsPhpFunction* requirement archetype is used to specify the execution of a *PHP function* when a *Link* (Section B.5) is clicked on a user interface *Page*. If the function executes without raising an exception, a specified user interface *Page* will be displayed. If the function raises an exception, a standardised error page will be displayed.

References. Instances of the *raLinkCallsPhpFunction* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theLink* : UI.Link** *(Section B.5)*

  A *Link* that appears on a user interface *Page*.

- ***theFunction* : PHP.Function** *(Section B.9)*

  The *PHP function* that will be called when *theLink* is clicked.

- ***theSuccessPage* : UI.Page** *(Section B.5)*

  If *theFunction* executes without raising any exceptions, this user interface *Page* will be displayed to the user.

C.2.5 Requirement Archetype 'raFormHandledByPhpFunction'

Description. The *raFormHandledByPhpFunction* requirement archetype is used to specify the execution of a *PHP function* to process data entered on a form when its *Submit Button* (Section B.5) is clicked. If the function executes without raising an exception, a specified user interface *Page* will be displayed. If the function raises an exception, a standardised error page will be displayed.

References. Instances of the *raFormHandledByPhpFunction* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theForm* : UI.Form** *(Section B.5)*

  A *Form* that appears on a user interface *Page*.

- ***theFunction* : PHP.Function** *(Section B.9)*

  The *PHP function* that will be called when *theForm's Submit Button* is clicked. Note that, at this point, it is only the notion that a particular PHP function will be invoked when the submit button is clicked. The way in which form data is passed to the PHP function is specified as part of appropriate *Implementation Rules* (Appendix E).

- ***theSuccessPage* : UI.Page** *(Section B.5)*

  This user interface *Page* will be displayed to the user if *theFunction* executes without raising any exceptions.

C.2.6 Requirement Archetype 'raClassList'

Description. The *raClassList* requirement archetype is used to specify the display of $\frac{X}{T}$*UML Class* instances on a *CRUD list* (Section B.5).

For example, *Requirement Rq01*, depicted in Figure C.2, is a requirement for instances of the *Employee* class to be listed on a *CrudList*.

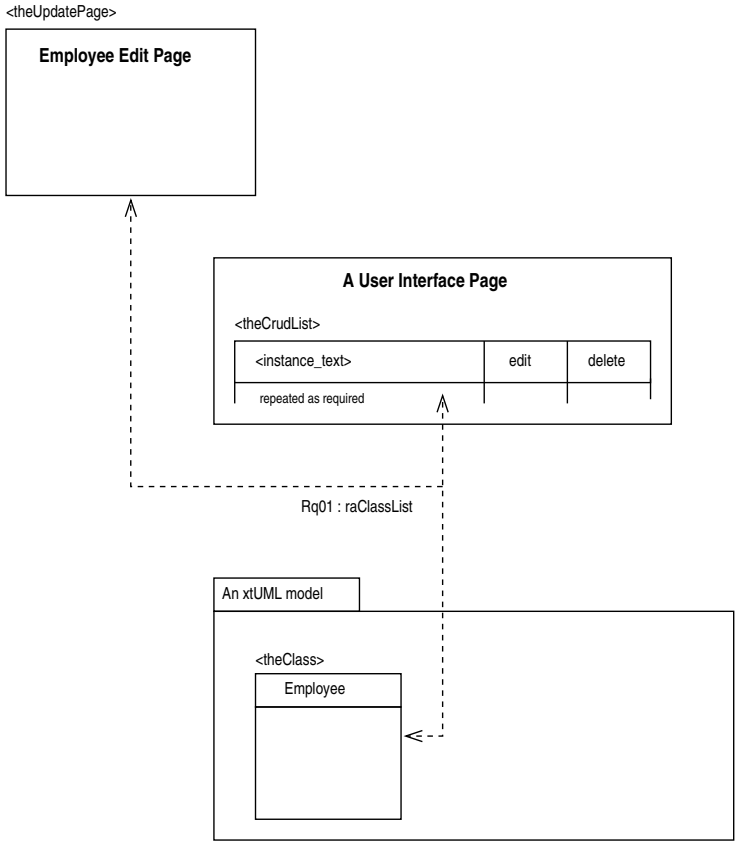

Figure C.2: Application of the *raClassList* requirement archetype. *Rq01* is a requirement for instances of the *Employee* class (employees) to be listed on *theCrudList*.

References. Instances of the *raClassList* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theCrudList* : UI.CrudList** *(Section B.5)*

  A *CrudList* that appears on a user interface *Page*.

- ***theClass* : xtUML.Class** *(Section B.7)*

  The $\frac{X}{T}$*UML Class* of objects to be displayed in *theCrudList*.

- ***theEditPage* : UI.Page** *(Section B.5)*

  CrudLists allow the user to *edit* and *view* listed items. This reference identifies a user interface *Page* that will used to edit instances listed in *theCrudList*. These *Pages* are specified in the same way as any other *Page* and will normally include a *Form* that allows users to view and update the attributes of an object.

C.2.7 Requirement Archetype 'raGroupClassList'

Description. The *raGroupClassList* requirement archetype is used in conjunction with the *raClassList* requirement archetype described above, to specify a requirement for instances of an $\frac{X}{T}$*UML Class* displayed in a *CRUD list* (Section B.5) to be organised into groups.

For example, *Requirement Rq01*, depicted in Figure C.2, is a requirement for instances of the *Employee* class to be listed on a *CrudList*. *Requirement Rq02* extends this requirement by requiring employees to be grouped by the name of the company to which they belong.

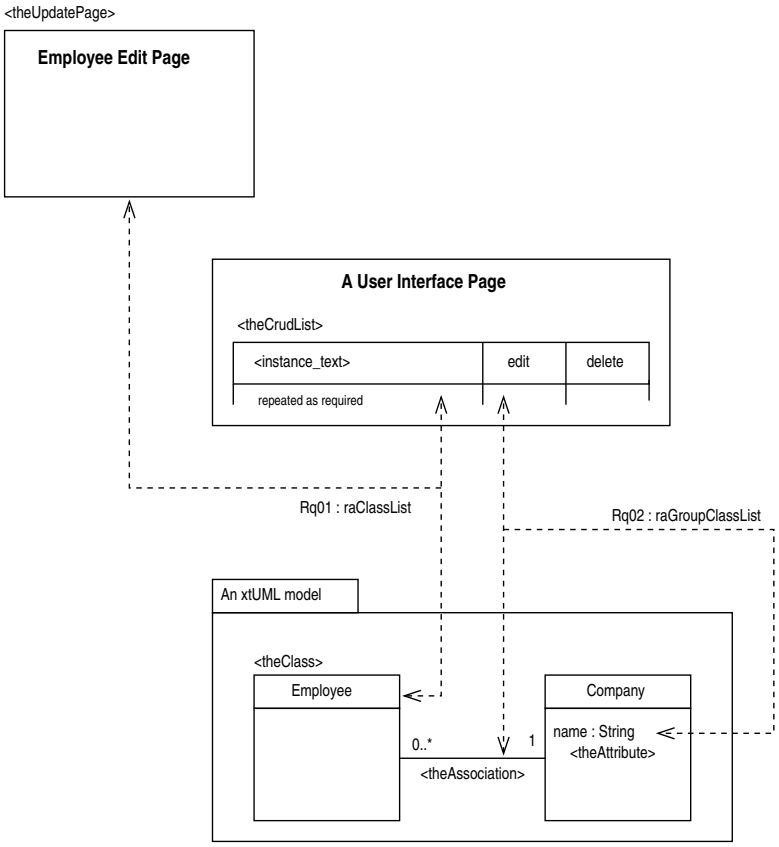

Figure C.3: Application of the *raGroupClassList* requirement archetype. *Rq01* is a requirement for instances of the *Employee* class (employees) to be listed on *theCrudList*. *Rq02* is a requirement for the listed employees to be grouped by the name of the company to which they belong.

References. Instances of the *raGroupClassList* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theCrudList* : UI.CrudList** *(Section B.5)*

  A *CrudList* that appears on a user interface *Page* and is referenced by an *raClassList* (Section C.2.6) requirement.

- ***theAssociation* : xtUML.Association** *(Section B.7)*

  An *Association* between the *Class* referenced by the above mentioned *Requirement* and a *Grouping Class* that will used to identify the groups by which items in the *CrudList* will be organised.

- ***theGroupingAttribute* : xtUML.Attribute** *(Section B.7)*

  An *Attribute* of the above mentioned *Grouping Class* that will be used to group instances listed in *theCrudList*.

C.2.8 Requirement Archetype 'raTextInputUpdatesAttribute'

Description. The *raTextInputUpdatesAttribute* requirement archetype is used to specify *Text Input Items* (Section B.5) that are used to update the value of an attribute of a *class* referenced by an *raClassList* (Section C.2.6) requirement. The *Text Input Item* must appear on *theEditPage* referenced by the *raClassList* (Section C.2.6) requirement.

For example, *Requirement Rq01*, depicted in Figure C.4, is a requirement for instances of the *Employee* class to be listed on a *CrudList*. *Requirement Rq02* extends this requirement by requiring the *Employee.name* attribute to be updated from data entered into a *Text Input Item* on the *Employee Edit Page*.

References. Instances of the *raTextInputUpdatesAttribute* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theTextInput* : UI.TextInput** *(Section B.5)*

  A *Text Input Item* that appears on *theEditPage* referenced by an *raClassList* (Section C.2.6) requirement.

- ***theAttribute* : xtUML.Attribute** *(Section B.7)*

  An *Attribute* of the $\frac{X}{T}$*UML Class* referenced by the above mentioned *raClassList* (Section C.2.6) requirement.

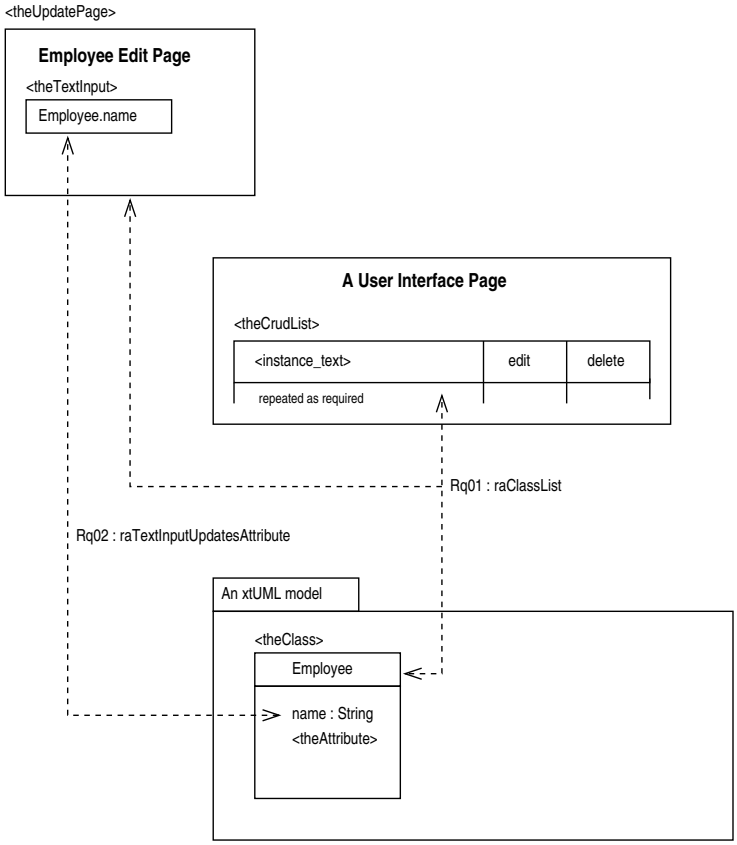

Figure C.4: Application of the *raTextInputUpdatesAttribute* requirement archetype. *Rq01* is a requirement for instances of the *Employee* class (employees) to be listed on *theCrudList*. *Requirement Rq02* specifies a requirement for the *Employee.name* attribute to be updated from data entered into *theTextInput* on *Employee Edit Page*.

C.2.9 Requirement Archetype 'raPasswordInputUpdatesAttribute'

Description. The *raPasswordInputUpdatesAttribute* requirement archetype is used to specify *Password Input Items* (Section B.5) that are used to update the value of an attribute of a *class* referenced by an *raClassList* (Section C.2.6) requirement. The *Password Input Item* must appear on *theEditPage* referenced by the *raClassList* (Section C.2.6) requirement.

The operation of this archetype is similar to that of the *raTextInputUpdatesAttribute* requirement archetype (Section C.2.8).

References. Instances of the *raPasswordInputUpdatesAttribute* requirement archetype must reference the following *Domain Models* and *Domain Model Ele-*

ments.

- ***thePasswordInput* : UI.PasswordInput** *(Section B.5)*

  A *Password Input Item* that appears on *theEditPage* referenced by an *raClassList* (Section C.2.6) requirement.

- ***theAttribute* : xtUML.Attribute** *(Section B.7)*

  An *Attribute* of the $^{X}_{T}$*UML Class* referenced by the above mentioned *raClassList* (Section C.2.6) requirement.

C.2.10 Requirement Archetype 'raTextAreaUpdatesAttribute'

Description. The *raTextAreaUpdatesAttribute* requirement archetype is used to specify *Text Area Items* (Section B.5) that are used to update the value of an attribute of a *class* referenced by an *raClassList* (Section C.2.6) requirement. The *Text Area Item* must appear on *theEditPage* referenced by the *raClassList* (Section C.2.6) requirement.

The operation of this archetype is similar to that of the *raTextInputUpdatesAttribute* requirement archetype (Section C.2.8).

References. Instances of the *raTextAreaUpdatesAttribute* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theTextArea* : UI.TextArea** *(Section B.5)*

  A *Text Area Item* that appears on *theEditPage* referenced by an *raClassList* (Section C.2.6) requirement.

- ***theAttribute* : xtUML.Attribute** *(Section B.7)*

  An *Attribute* of the $^{X}_{T}$*UML Class* referenced by the above mentioned *raClassList* (Section C.2.6) requirement.

C.2.11 Requirement Archetype 'raSelectItemCreatesLink'

Description. The *raSelectItemCreatesLink* requirement archetype is used to specify *Select Items* that are used to create *Links* between instances of a specified $^{X}_{T}$*UML Class* and instances of a *Class* referenced by an *raClassList* (Section C.2.6) requirement.

For example, *Requirement Rq01*, depicted in Figure C.5, is a requirement for instances of the *Employee* class to be listed on a *CrudList*. *Requirement Rq02* extends this requirement by requiring the creation of a link between an employee and a company selected using a *Select Item* on the *Employee Edit Page*.

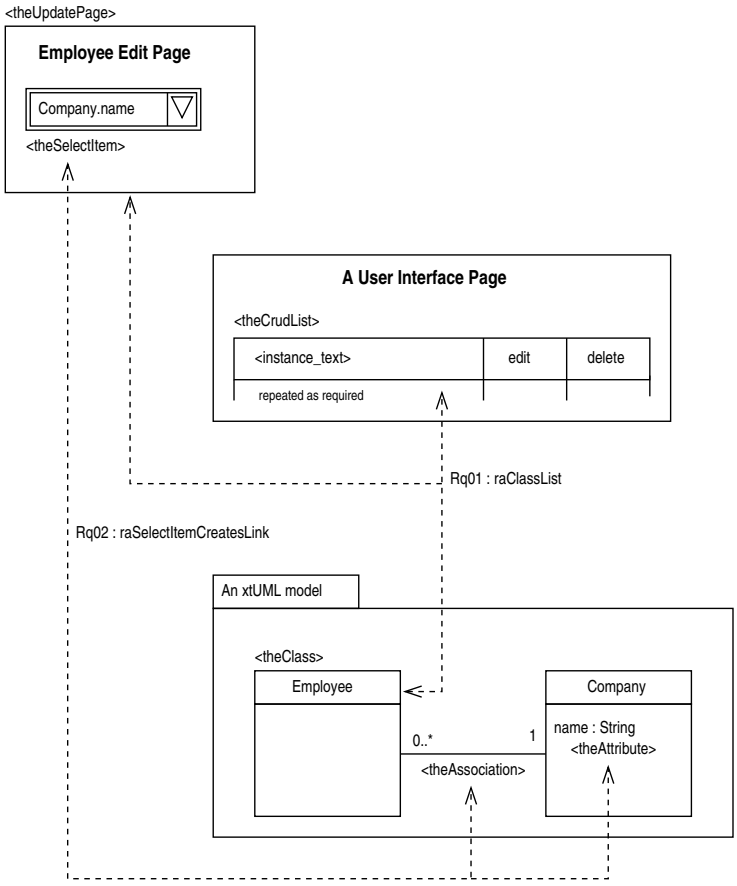

Figure C.5: Application of the *raSelectItemCreatesLink* requirement archetype. *Rq01* is a requirement for instances of the *Employee* class (employees) to be listed on *theCrudList*. *Requirement Rq02* specifies a requirement for the creation of a link between an employee and a company selected using *theSelectItem* on the *Employee Edit Page*.

References. Instances of the *raSelectItemCreatesLink* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theSelectItem* : UI.Select** *(Section B.5)*

  A *Select Item* that appears on *theEditPage* referenced by an *raClassList* (Section C.2.6) requirement.

- ***theAssociation* : xtUML.Association** *(Section B.7)*

An *Association* between the *Class* referenced by the above mentioned *raClassList* requirement and the *Class* of objects to be linked.

- ***theAttribute*** **: xtUML.Attribute** *(Section B.7)*

  An *Attribute* of the *Class* of objects to be linked. Values of this attribute will be listed in the *Select Item* to identify the instance to be linked.

C.3 Security related requirement archetypes

C.3.1 Requirement Archetype 'raSecurityPhpImplementation'

Description. The *raSecurityPhpImplementation* requirement archetype is used to specify implementation of an instance of the *Security* domain model using PHP.

References. Instances of the *raSecurityPhpImplementation* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theSecurityDomain*** **: SEC** *(Section B.4)*

  An instance of the *Security* domain model that specifies the security policy applicable to the application being built.

- ***thePhpDomain*** **= PHP** *(Section B.9)*

  The *PHP Web Programming Language* domain model.

C.3.2 Requirement Archetype 'raSecureClass'

Description. The *raSecureClass* requirement archetype is used to specify that instances of an $\frac{X}{T}$UML *Class* be managed as *Protected Items* (Section B.4).

References. Instances of the *raSecureClass* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theClass*** **: xtUML.Class** *(Section B.7)*

  Any $\frac{X}{T}$UML class.

- ***theProtectedItem*** **: SEC.ProtectedItem** *(Section B.4)*

  A *Protected Item* declared within an instance of the *Security* domain model.

C.4 Database related requirement archetypes

C.4.1 Requirement Archetype 'raMysqlImplementation'

Description. The *raMysqlImplementation* requirement archetype is used to specify implementation of the *Structured Query Language* domain model using MySQL.

References. Instances of the *raMysqlImplementation* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theSqlDomain* = SQL** *(Section B.10)*

  The *Structured Query Language* domain model.

- ***theMysqlDomain* = MySQL** *(Section B.12)*

  The *MySQL Relational Database* domain model.

C.4.2 Requirement Archetype 'raPersistentClass'

Description. The *raPersistentClass* requirement archetype is used to specify implementation of an $\frac{X}{T}$UML class using the *Structured Query Language* domain model.

References. Instances of the *raPersistentClass* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theClass* : xtUML.Class** *(Section B.7)*

  Any $\frac{X}{T}$UML class.

- ***theSqlDomain* = SQL** *(Section B.10)*

  The *Structured Query Language* domain model.

C.4.3 Requirement Archetype 'raPersistentAssociation'

Description. The *raPersistentAssociation* requirement archetype is used to specify implementation of an $\frac{X}{T}$UML *Association* using the *Structured Query Language* domain model.

References. Instances of the *raPersistentAssociation* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theAssociation* : xtUML.Association** *(Section B.7)*

  Any $\frac{X}{T}$UML *Association* between two *Classes* that are each referenced by separate *raPersistentClass* (Section C.4.2) requirements.

- ***theSqlDomain* = SQL** *(Section B.10)*

  The *Structured Query Language* domain model.

C.4.4 Requirement Archetype 'raCorrespondingClasses'

Description. The *raCorrespondingClasses* requirement archetype is used to specify implementation of an $\frac{X}{T}$UML *Class* in one *Domain Model* as a specialisation of an $\frac{X}{T}$UML *Class* in another *Domain Model*.

References. Instances of the *raCorrespondingClasses* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theParentClass* : xtUML.Class** *(Section B.7)*

  A *Class* defined in one *Domain Model* that is to be implemented as a generalisation of a *Class* defined in another *Domain Model*.

- ***theSpecialisedClass* : xtUML.Class** *(Section B.7)*

  The *Class* that is to be implemented as a specialisation of *theParentClass*.

C.5 Implementation specific requirement archetypes

C.5.1 Requirement Archetype 'raUiImplementedInPhp'

Description. The *raUiImplementedInPhp* requirement archetype is used to specify the implementation of a user interface using PHP.

References. Instances of the *raUiImplementedInPhp* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theUI* : UI** *(Section B.5)*

  The *User Interface* to be implemented entirely in PHP.

- ***thePhpDomain* = PHP** *(Section B.9)*

  The *Domain Model* that represents knowledge about the PHP programming language.

C.5.2 Requirement Archetype 'raPhpImplementedOnApache'

Description. The *raPhpImplementedOnApache* requirement archetype is used to specify implementation of the *PHP Web Programming Language* domain model using the *Apache Web Server* and associated modules.

References. Instances of the *raPhpImplementedOnApache* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***thePhpDomain* = PHP** *(Section B.9)*

  The *PHP Web Programming Language* domain model.

- ***theApacheDomain* = Apache** *(Section B.11)*

  The *APACHE Web Server* domain model.

- ***theStartPage* : UI.Page** *(Section B.5)*

  The *Page* that is to be displayed when the application is started.

C.5.3 Requirement Archetype 'raApacheImplementedOnLinux'

Description. The *raApacheImplementedOnLinux* requirement archetype is used to specify implementation of the *APACHE Web Server* domain model on the *Linux* operating system.

References. Instances of the *raApacheImplementedOnLinux* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theApacheDomain* = Apache** *(Section B.11)*

  The *APACHE Web Server* domain model.

- ***theLinuxDomain* = Linux** *(Section B.8)*

  The *Linux Operating System* domain model.

C.5.4 Requirement Archetype 'raMysqlImplementedOnLinux'

Description. The *raMysqlImplementedOnLinux* requirement archetype is used to specify implementation of the *MySQL Relational Database* domain model on the *Linux* operating system.

References. Instances of the *raMysqlImplementedOnLinux* requirement archetype must reference the following *Domain Models* and *Domain Model Elements*.

- ***theMySqlDomain*** **= MySQL** *(Section B.12)*

  The *MySQL Relational Database* domain model.

- ***theLinuxDomain*** **= Linux** *(Section B.8)*

  The *Linux Operating System* domain model.

C.6 *Security Mechanisms* domain model

The *Security Mechanisms* domain model (Section G.5) comprises a collection of *PHP* functions that implement various security related operations including *registerUser, login* and *logoff*. These functions can be referenced by *raFormHandledByPhpFunction* (Section C.2.5) requirements to form user interfaces for managing the authentication of application users. For this reason, the *Security Mechanisms* domain model is considered to be part of the *LAMP Aspect-Oriented Specification Archetype* as well as the *LAMP Implementation Model* described in Appendix E.

C.7 Conclusion

In this appendix, I have presented an *Aspect-Oriented Specification Archetype* which demonstrates how *Aspect-Oriented Thinking* concepts can be used to capture system architectural decisions in a form that can be used to specify many different systems that share the same architecture. The *Aspect-Oriented Specification Archetype* presented in this appendix can be used to form *Aspect-Oriented Specifications* for software systems that share a common architecture based on the LAMP framework.

Appendix

D

Proof of concept - *Product Management System Aspect-Oriented Specification*

Contents

D.1 Introduction 159
D.1.1 Domain models 159
D.1.2 Requirements Notation 159
D.2 PMS persistence requirements 161
D.2.1 *Product Management* persistence 161
D.2.2 *Contact Management* persistence 162
D.2.3 *Security* persistence 163
D.2.4 PMS persistence implementation 164
D.3 *PMS Security* domain model 164
D.4 *PMS Security* requirements 165
D.4.1 *Product Management* security 165
D.4.2 *Contact Management* security 166
D.4.3 *PMS Security* implementation 166
D.5 *PMS User Interface* domain model 166
D.5.1 User Interface Notation 167
D.5.2 Authentication interface 168
D.5.3 The main menu interface 169
D.5.4 Product management interface 170
D.5.5 Contacts management interface 170
D.5.6 Registered User management interface 172
D.5.7 Error reporting 173

D.6 *PMS User Interface* requirements 174
D.6.1 Login Page 174
D.6.2 Register New User Page 176
D.6.3 Registration Success Page 177
D.6.4 Main Menu Page 177
D.6.5 Product List Page 178
D.6.6 Edit Product Page 179
D.6.7 Edit Product Variant Page 180
D.6.8 Product Category List Page 180
D.6.9 Edit Product Category Page 181
D.6.10 Contact List Page 181
D.6.11 Edit Contact Page 182
D.6.12 Edit Contact Relationship Page 183
D.6.13 Edit Note Page 184
D.6.14 Registered User List Page 184
D.6.15 Edit Registered User Page 184
D.6.16 *PMS User Interface* implementation 185
D.7 Conclusion 186

D.1 Introduction

The *Aspect-Oriented Specification Archetype* presented in Appendix C can be used to specify software systems that share a common architecture based on the LAMP framework. In this appendix, I present an *Aspect-Oriented Specification* that has been developed for the entire *Product Management System* (Chapter 6) in accordance with the *LAMP Aspect-Oriented Specification Archetype*.

D.1.1 Domain models

Figure D.1 shows the *Product Management System Aspect-Oriented Specification* documented in this appendix. It comprises the *PMS User Interface* domain model and *PMS Security* domain model, several *Requirement Sets* and the following reusable *Knowledge Domain Models*.

- *Contact Management* domain model (Section B.2)
- *Product Management* domain model (Section B.3)
- *Security* domain model (Section B.4)
- *User Interface* domain model (Section B.5)
- *LAMP Framework* domain model (Section B.6)
- $\frac{X}{T}$*UML* domain model (Section B.7)
- *Linux Operating System* domain model (Section B.8)
- *PHP Web Programming Language* domain model (Section B.9)
- *Structured Query Language* domain model (Section B.10)
- *APACHE Web Server* domain model (Section B.11)
- *MySQL Relational Database* domain model (Section B.12)

D.1.2 Requirements Notation

Within this appendix, *Requirements* are presented using the following notation.

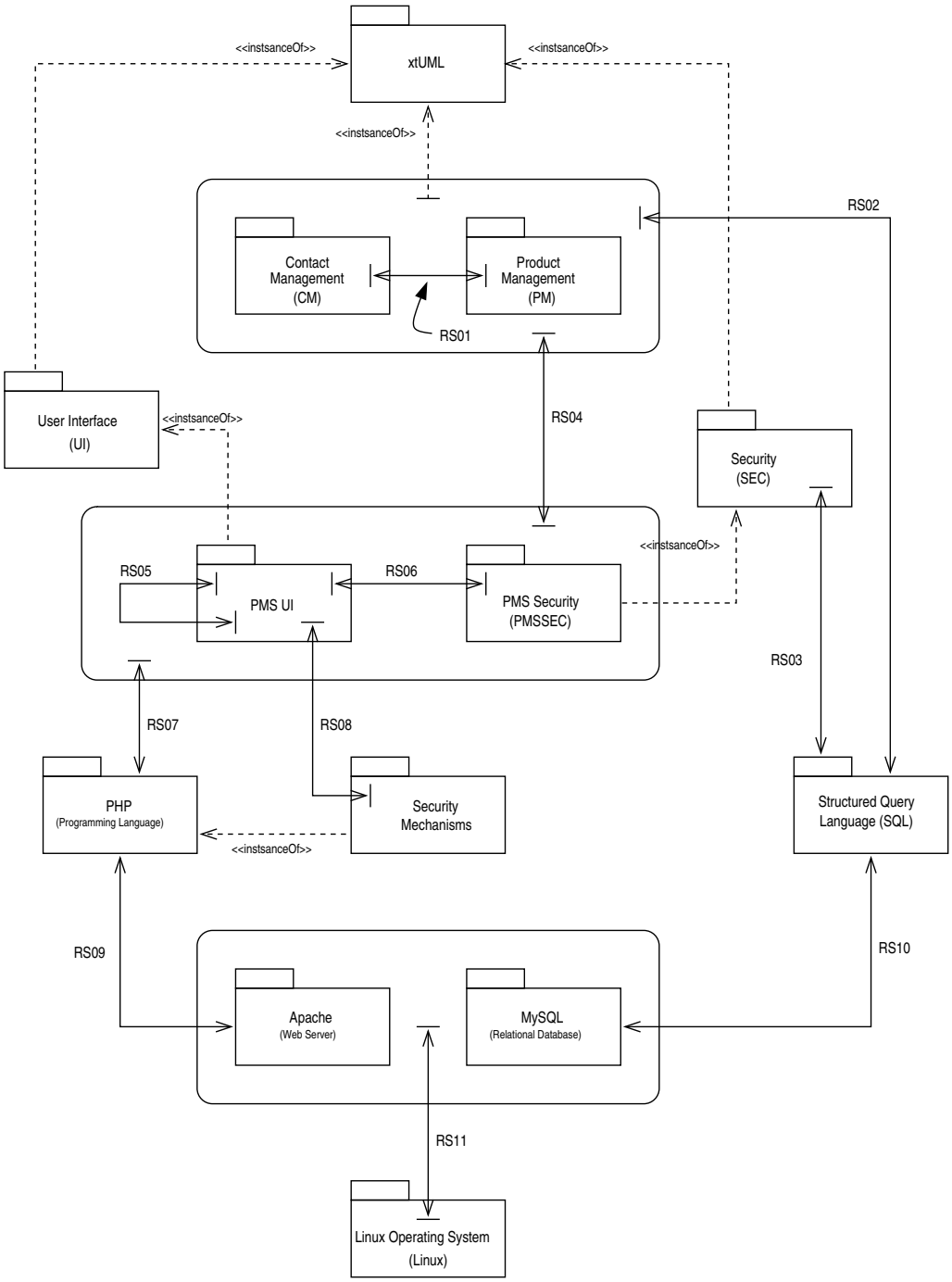

Figure D.1: Package diagram depicting the *Aspect-Oriented Specification* of the *Product Management System*.

RS<**rsNum**>**.R**<**rNum**> : <raName> (<raRef>)

- <refName> => <domain>[.<element>] (<domainRef>)
- ...

Where:

- **rsNum**: the *Requirement Set* number.
- **rNum**: the number of the *Requirement* within its enclosing set.
- **raName**: then name of the *Requirement Archetype*
- **raRef**: *RequirementArchetype* cross-reference.
- **refName**: the name of a referenced *Domain Model* or *Domain Model Element* defined with the above *Requirement Archetype*.
- **domain**: the name of a *Domain Model*.
- **element**: if applicable, the name of an *Element* defined within the above *Domain Model*
- **domainRef**: *Domain Model* cross-reference

D.2 PMS persistence requirements

The *Requirements* specified in this section relate to the persistence of *Product Management System* objects and links amongst them.

D.2.1 *Product Management* persistence

RS02.R1 : raPersistentClass *(Section C.4.2)*

- theClass => PM.Product *(Section B.3)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R2 : raPersistentClass *(Section C.4.2)*

- theClass => PM.ProductVariant *(Section B.3)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R3 : raPersistentClass *(Section C.4.2)*

- theClass => PM.ProductCategory *(Section B.3)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R4 : raPersistentAssociation *(Section C.4.3)*

- theAssociation => PM.R01 *(Section B.3)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R5 : raPersistentAssociation *(Section C.4.3)*

- theAssociation => PM.R02 *(Section B.3)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R6 : raPersistentAssociation *(Section C.4.3)*

- theAssociation => PM.R03 *(Section B.3)*
- theSqlDomain => SQL *(Section B.10)*

D.2.2 *Contact Management* persistence

RS01.R1 : raCorrespondingClasses *(Section C.4.4)*

- theParentClass => CM.Contact *(Section B.2)*
- theSpecialisedClass => PM.Supplier *(Section B.3)*

RS02.R7 : raPersistentClass *(Section C.4.2)*

- theClass => CM.Contact *(Section B.2)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R8 : raPersistentClass *(Section C.4.2)*

- theClass => CM.ContactRelationship *(Section B.2)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R9 : raPersistentClass *(Section C.4.2)*

- theClass => CM.RelationshipType *(Section B.2)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R10 : raPersistentClass *(Section C.4.2)*

- theClass => CM.Note *(Section B.2)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R11 : raPersistentAssociation *(Section C.4.3)*

- theAssociation => CM.R01 *(Section B.2)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R12 : raPersistentAssociation *(Section C.4.3)*

- theAssociation => CM.R02 *(Section B.2)*
- theSqlDomain => SQL *(Section B.10)*

RS02.R13 : raPersistentAssociation *(Section C.4.3)*

- theAssociation => CM.R03 *(Section B.2)*
- theSqlDomain => SQL *(Section B.10)*

D.2.3 *Security* persistence

RS03.R1 : raPersistentClass *(Section C.4.2)*

- theClass => SEC.RegisteredUser *(Section B.4)*
- theSqlDomain => SQL *(Section B.10)*

RS03.R2 : raPersistentClass *(Section C.4.2)*

- theClass => SEC.Role *(Section B.4)*
- theSqlDomain => SQL *(Section B.10)*

D.2.4 PMS persistence implementation

RS10.R1 : raMysqlImplementation *(Section C.4.1)*

- theSqlDomain => SQL *(Section B.10)*
- theMysqlDomain => MySQL *(Section B.12)*

RS11.R1 : raMysqlImplementedOnLinux *(Section C.5.4)*

- theMySqlDomain => MySQL *(Section B.12)*
- theLinuxDomain => Linux *(Section B.8)*

D.3 *PMS Security* domain model

This section contains the *PMS Security* domain model. This model formally specifies the *Product Management System* security requirements stated in Section 6.2.3 using an instance of the *Security* domain model (Section B.4).

Table D.1: *Product Management System* security specification

	Role	
Protected Item	staffRole	customerRole
Product	C,R,U and D	R
ProductVariant	C,R,U and D	R
ProductCategory	C,R,U and D	R
Contact	C,R,U and D	
ContactRelationship	C,R,U and D	
Note	C,R,U and D	

Table D.1 identifies the *Roles*, *Permissions* and *Protected Items* involved in *Product Management System* security. Each column of the table represents an instance of the *SEC.Role* class. Rows represent instances of the *SEC.ProtectedItem* class. Each cell represents a set of *Item Permissions* which each identify a *Permission* a particular *Role* has with respect to a *Protected Item*. Each *Permission* is identified by one of the following letters.

- **C:** A *Permission* to **create** an instance of a *Protected Item*.

- **R:** A *Permission* to **read** any instance of a *Protected Item*.
- **U:** A *Permission* to **update** any instance of a *Protected Item*.
- **D:** A *Permission* to **delete** any instance of a *Protected Item*.

For example, Table D.1 shows that *customerRole* is linked to three *Item Permissions*. The first represents a permission to read *Products*. The second and third represent permissions to read *Product Variants* and *Product Categories*. The *staffRole*, on the other hand, is linked to *Item Permissions* that represent permissions to *Create, Read, Update* and *Delete* any of the *Protected Items*.

Note that instances of *Protected Item* identified in Table D.1 are not the same as classes of the same name identified in the *Contact Management* domain model (Section B.2) and *Product Management* domain model (Section B.3). The protected items will, however, be referenced by *raSecureClass* (Section C.3.2) requirements as shown in the Sections D.4.1 and D.4.2.

D.4 *PMS Security* requirements

The *Requirements* specified in this section relate to the security of *Product Management System* data.

D.4.1 *Product Management* security

RS04.R1 : raSecureClass *(Section C.3.2)*

- theClass => PM.Product *(Section B.3)*
- theProtectedItem => PmsSEC.Product *(Section D.3)*

RS04.R2 : raSecureClass *(Section C.3.2)*

- theClass => PM.ProductVariant *(Section B.3)*
- theProtectedItem => PmsSEC.ProductVariant *(Section D.3)*

RS04.R3 : raSecureClass *(Section C.3.2)*

- theClass => PM.ProductCategory *(Section B.3)*
- theProtectedItem => PmsSEC.ProductCategory *(Section D.3)*

D.4.2 *Contact Management* security

RS04.R4 : raSecureClass *(Section C.3.2)*

- theClass => CM.Contact *(Section B.2)*
- theProtectedItem => PmsSEC.Contact *(Section D.3)*

RS04.R5 : raSecureClass *(Section C.3.2)*

- theClass => CM.ContactRelationship *(Section B.2)*
- theProtectedItem => PmsSEC.ContactRelationship *(Section D.3)*

RS04.R6 : raSecureClass *(Section C.3.2)*

- theClass => CM.Note *(Section B.2)*
- theProtectedItem => PmsSEC.Note *(Section D.3)*

D.4.3 *PMS Security* implementation

RS07.R1 : raSecurityPhpImplementation *(Section C.3.1)*

- securityDomain => PmsSEC *(Section D.3)*
- thePhpDomain => PHP *(Section B.9)*

D.5 *PMS User Interface* domain model

This section contains the *PMS User Interface* domain model. It is an instance of the *User Interface* domain model (Section B.5) and comprises a set of *Pages*. Each of these *Pages* is represented in this section as a screen layout diagram. Informal *Annotations* (Figure B.4) describe the purpose of each user interface page element.

At this point, it is only the user interface page layouts and informal descriptions of each user interface element that are provided. The way in which these elements interact among themselves and with other parts of the *Product Management System*, is specified later using the *separate* set of *Requirements* presented in Section D.6. Complete formality is finally provided as part of the *LAMP Implementation Model* (Section E.3).

This separation of concerns along the dimension of formality, enables the development of user interfaces to proceed in an agile manner, while maintaining

some degree of precise specification.

Note that some design decisions have been made during the modelling of user interface pages and the specification of user interface *Requirements* in Section D.6. These not only include decisions about look and feel, but also decisions about the way in which pages interact and how the user is provided access to the various functions of the *Product Management System*. None of this was specified in the original user requirements provided in Section 6.2.4.

D.5.1 User Interface Notation

Figure D.2 depicts the notation used in this thesis to represent instances of the *User Interface* domain model. Note that the UML *Note* symbol is used to represent *Annotations* (Figure B.4) which informally describe elements on a particular page.

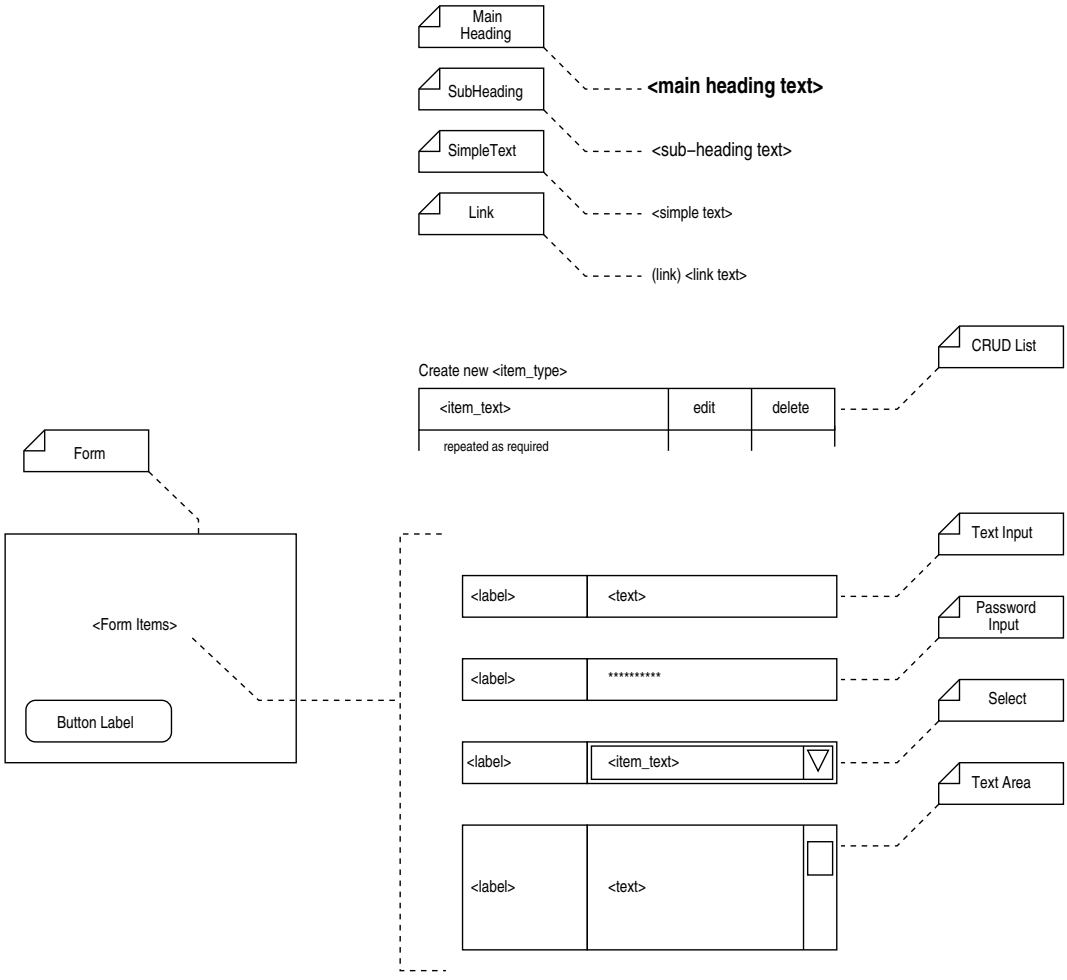

Figure D.2: Page layout notation. See Section B.5 for descriptions of each user interface page element

D.5.2 Authentication interface

Figures D.3 to D.5 depict user interface pages involved in security aspects of the *Product Management System*.

The *loginPage* depicted in Figure D.3 shall be the first page displayed to users and allows the user to gain access to the *Product Management System* by providing their username and password. The page also provides a link that allows new users to register for later access to the system.

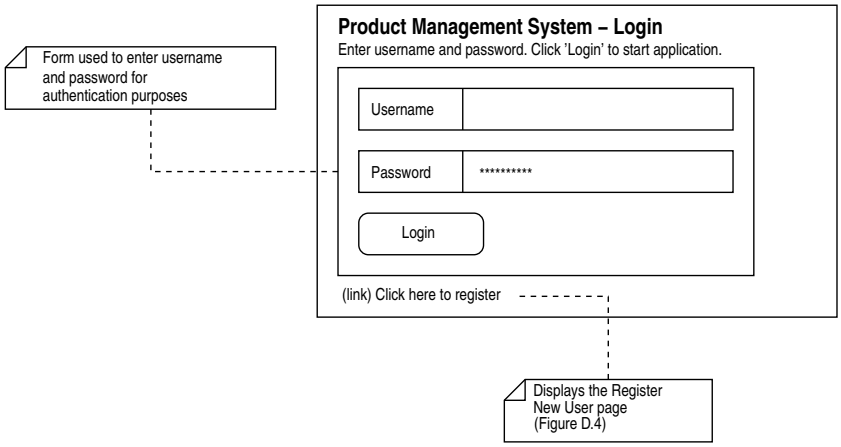

Figure D.3: The *loginPage* layout.

The *registerNewUserPage* depicted in Figure D.4 allows new users to register with the *Product Management System* by providing a new username and password. If registration is successful, the *registrationSuccessPage* depicted in Figure D.5 shall be displayed.

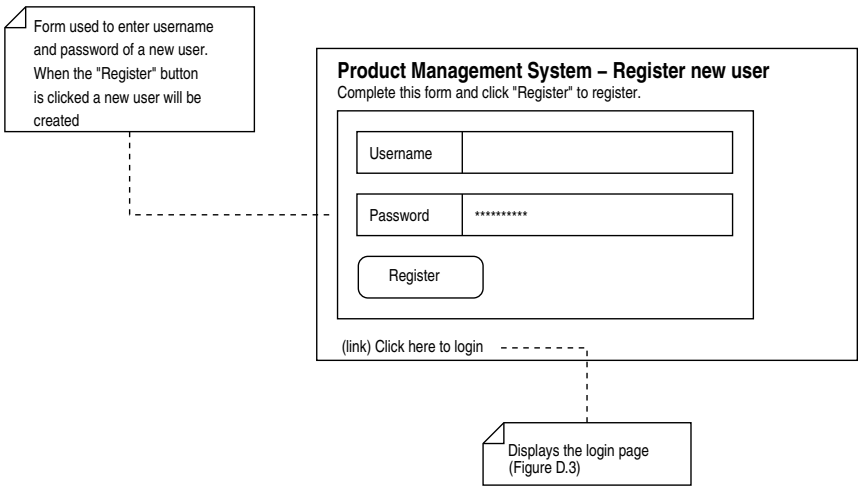

Figure D.4: The *registerNewUserPage* layout.

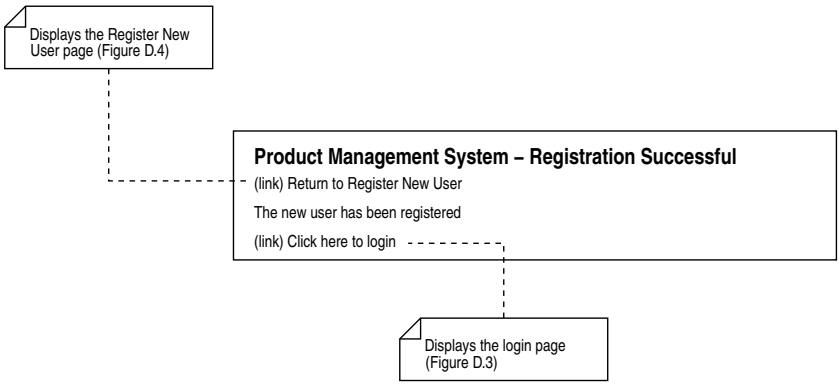

Figure D.5: The *registrationSuccessPage* layout.

D.5.3 The main menu interface

After a user logs in, he shall be presented with the *mainMenuPage* depicted in Figure D.6. This page includes links to the various functions provided by the *Product Management System* system. Note that the actual functions available to a particular user will be determined by security policies specified in Section D.4 and that the operation of the functions will be specified by establishing appropriate *Requirements* that reference elements of the *PMS User Interface* and other *Domain Models*.

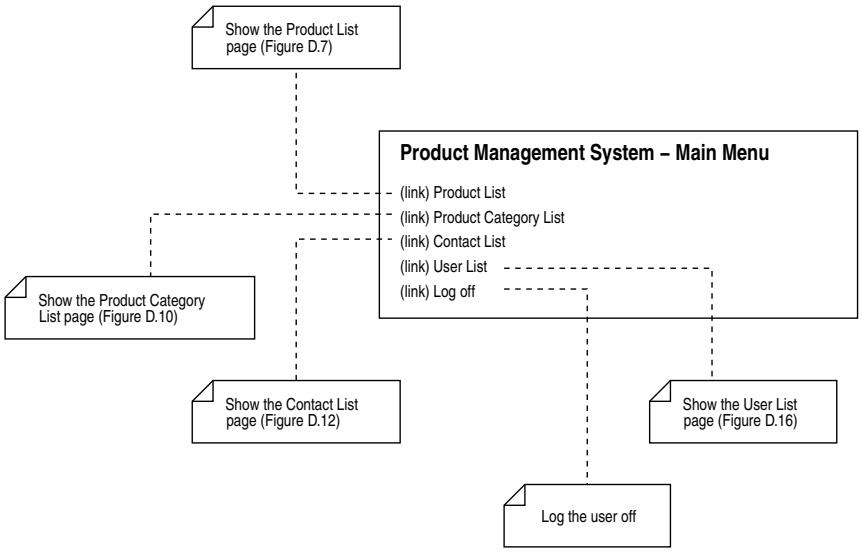

Figure D.6: The *mainMenuPage* layout.

D.5.4 Product management interface

Figures D.7 to D.11 depict user interface pages involved in the management of *Product* (Figure B.2) information.

Figure D.7 depicts the *productListPage*. This page displays a list of all *Products* stored by the *Product Management System* and allows users to create new products as well as update, view and delete existing products. A link is also provided to allow the user to return to the *mainMenuPage*.

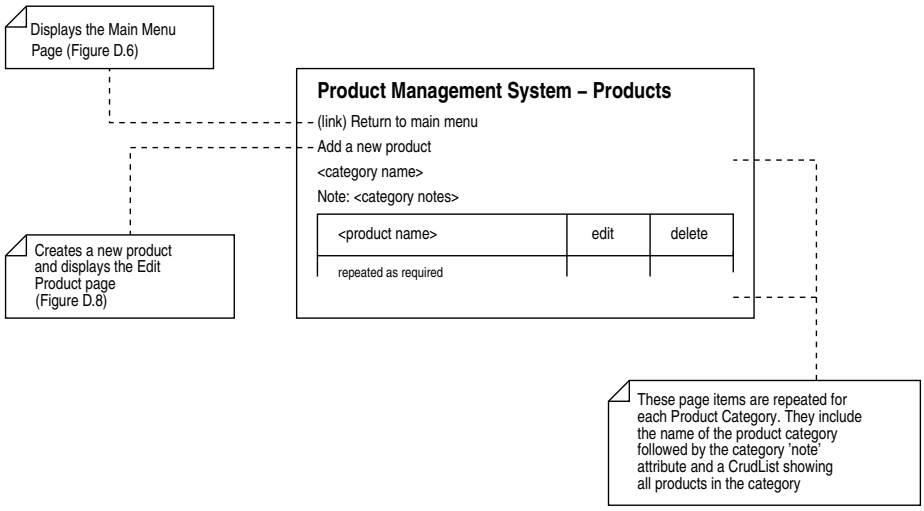

Figure D.7: The *productListPage* layout.

Figure D.8 depicts the *editProductPage* which is used to view, edit and create new *Products*. The page includes user interface elements for each attribute of the *Product* class as well as a *CRUD List* of associated *Product Variants* (Figure B.2).

Figure D.9 depicts the *editProductVariantPage* which is used to view, edit and create new *Product Variants*. The page includes user interface elements for each attribute of the *Product Variant* class and a link that allows the user to return to the *editProductPage* of the *Product* with which the variant is associated.

Figures D.10 and D.11 depict the *productCategoryListPage* and *editProduct-CategoryPage* which are used to manage *Product Categories* (Figure B.2). They operate in a similar way to those used to manipulate *Products* and *Product Variants*.

D.5.5 Contacts management interface

Figures D.12 to D.15 depict user interface pages involved in the management of *Contact* (Figure B.1) information. The *contactListPage* depicted in Figure D.12,

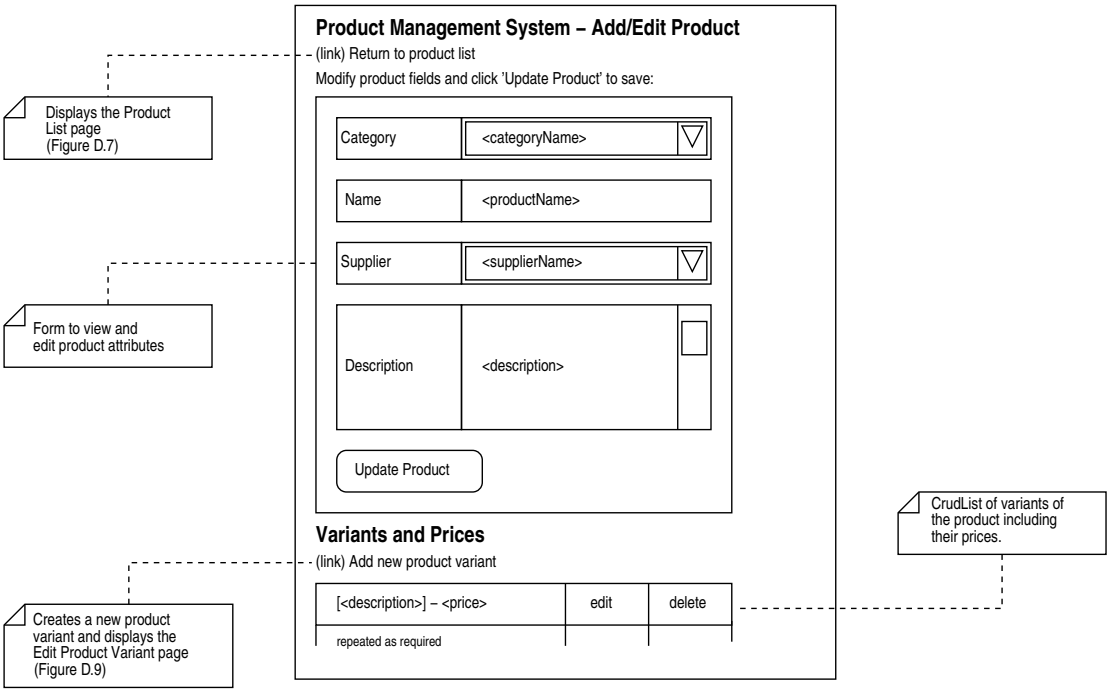

Figure D.8: The *editProductPage* layout.

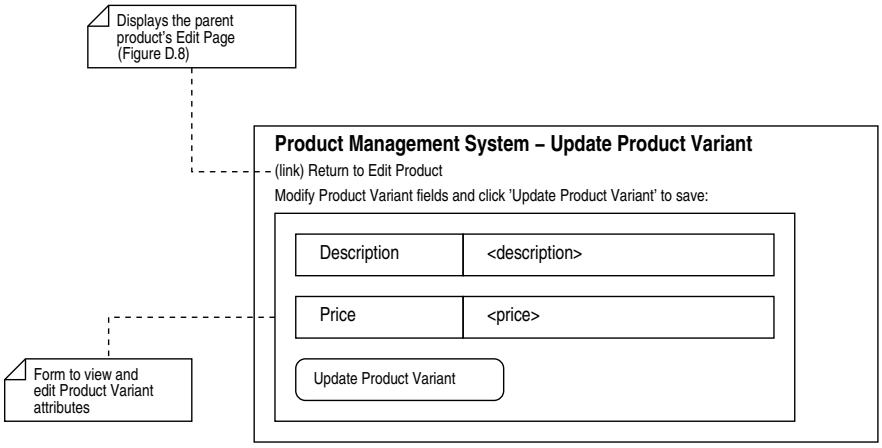

Figure D.9: The *editProductVariantPage* layout.

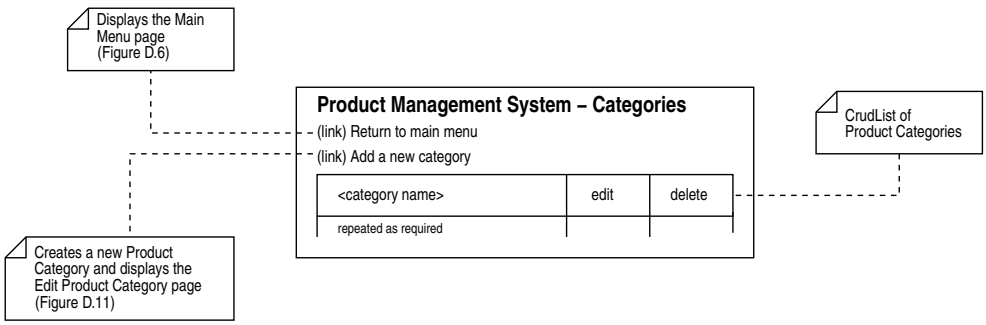

Figure D.10: The *productCategoryListPage* layout.

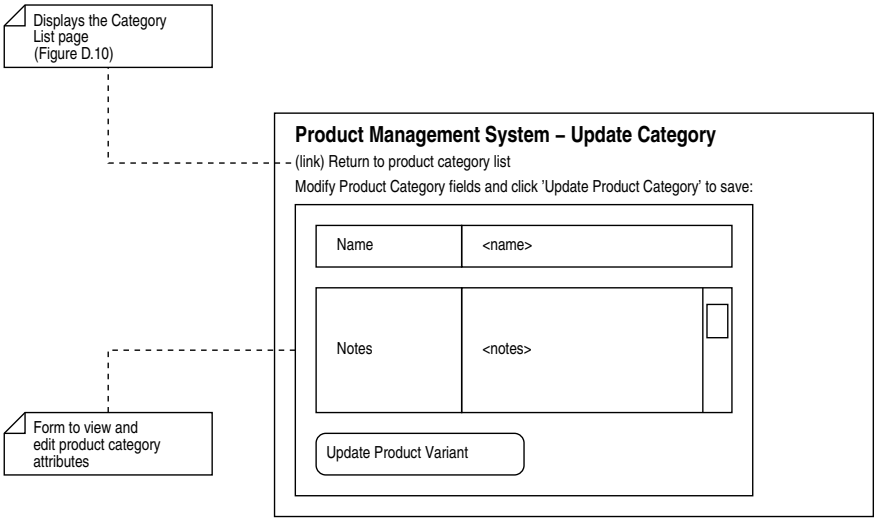

Figure D.11: The *editProductCategoryPage* layout.

along with the *editContactPage* depicted in Figure D.13, operate in a similar way to the interfaces for *Products* described above. The *editContactRelationshipPage* depicted in Figure D.14 and the *editNotePage* depicted in Figure D.15 are user interfaces used to manage *Contact Relationships* and *Notes* associated with a specific *Contact*.

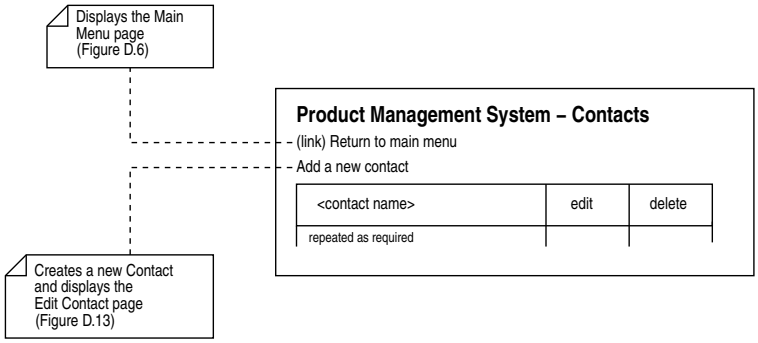

Figure D.12: The *contactListPage* layout.

D.5.6 Registered User management interface

Figures D.16 and D.17 depict user interface pages associated with the management of *Registered Users* (Figure B.3). The *registeredUserListPage* depicted in Figure D.16 along with the *editRegisteredUserPage* depicted in Figure D.17 operate in a similar way to the interfaces for *Products* and *Contacts* described above.

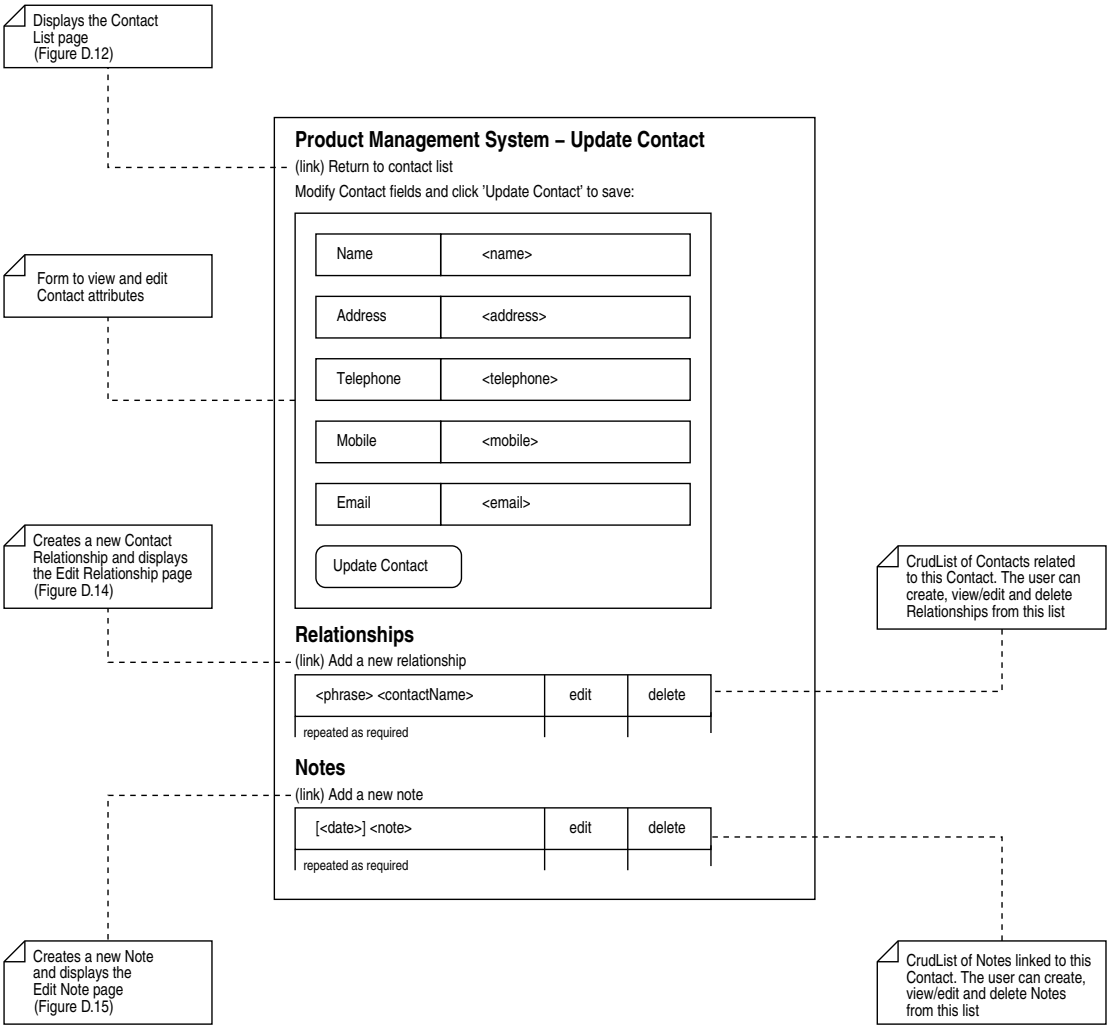

Figure D.13: The *editContactPage* layout.

D.5.7 Error reporting

When an error occurs during operation of the *Product Management System*, an error message shall be displayed using one of the *Error* pages depicted in Figures D.18 and D.19.

In the case of recoverable errors, such as the detection of invalid data entry, the *errorPage* depicted in Figure D.18 shall be displayed. This page shows the error message and provides a link that allows the user to continue using the system by returning to the last page displayed before the error occurred.

In the case of unrecoverable errors, such as complete failure of the database system, the *fatalErrorPage* depicted in Figure D.19 shall be displayed. This page does not provide any links to continue - it is the *end of the line*.

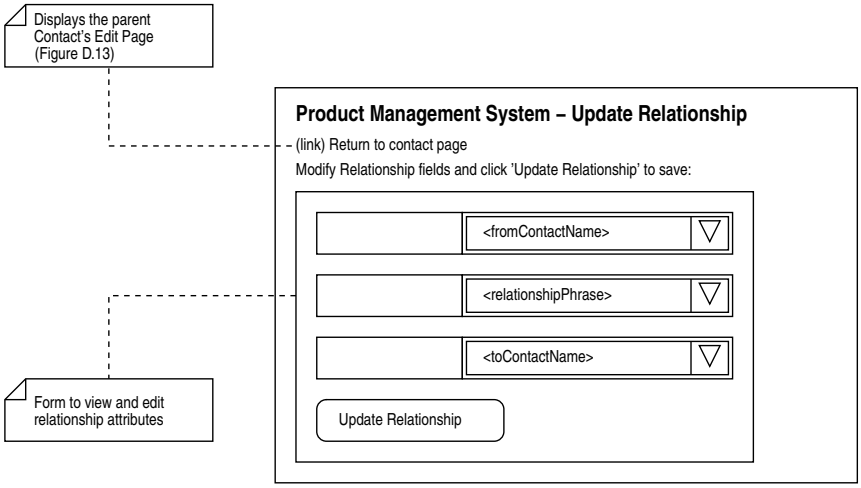

Figure D.14: The *editContactRelationshipPage* layout.

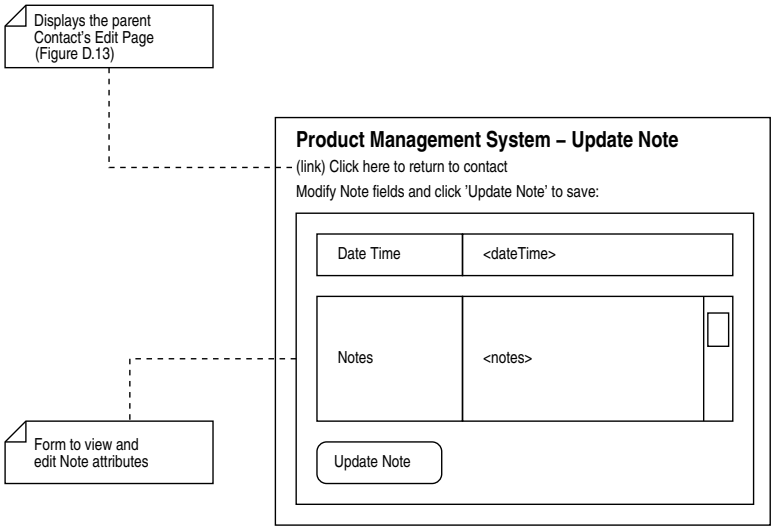

Figure D.15: The *editNotePage* layout.

D.6 *PMS User Interface* requirements

The *Requirements* specified in this section relate to the *Product Management System* user interface.

D.6.1 Login Page

The *loginPage* (Figure D.3) shall be implemented in accordance with the following requirements.

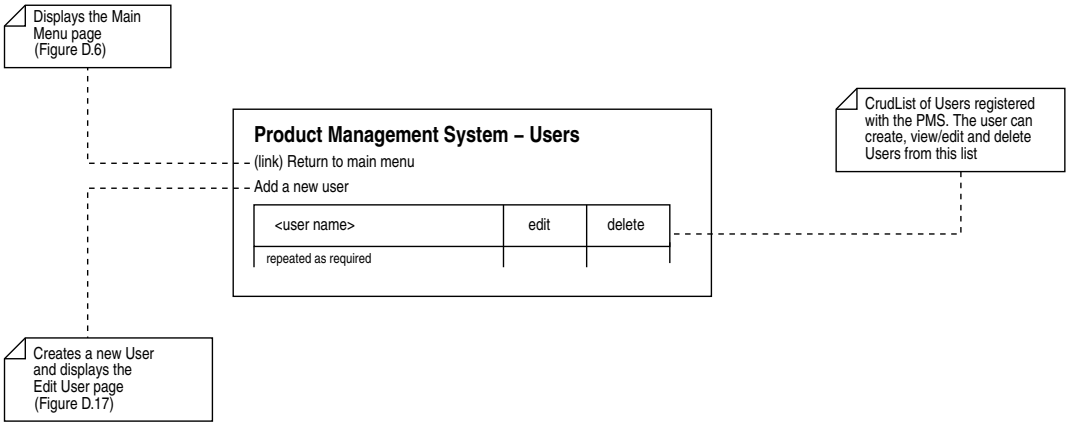

Figure D.16: The *registeredUserListPage* layout.

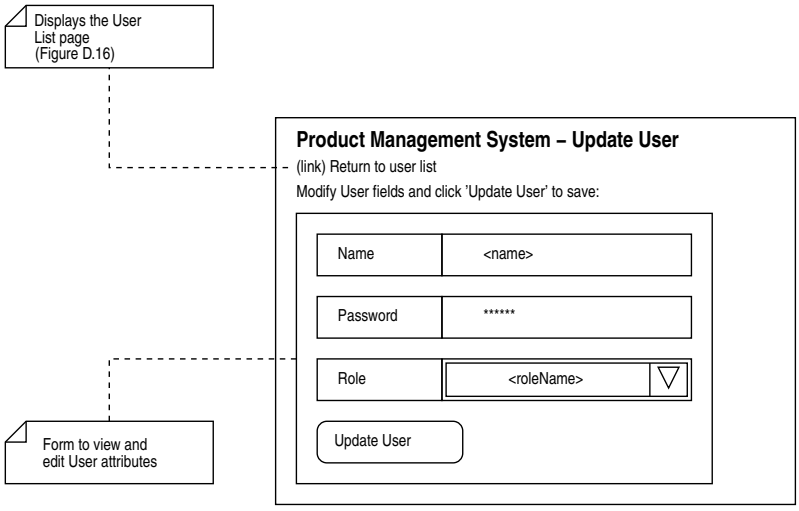

Figure D.17: The *editRegisteredUserPage* layout.

RS07.R2 : raPageImplementedInPhp *(Section C.2.1)*

- thePage => PmsUI.loginPage *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

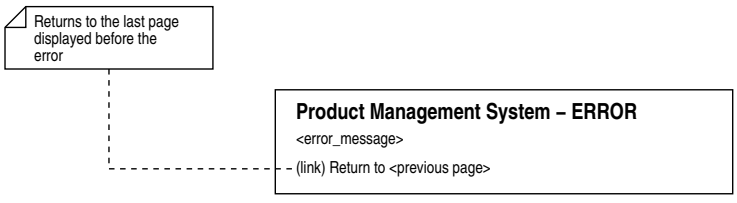

Figure D.18: The *errorPage* layout.

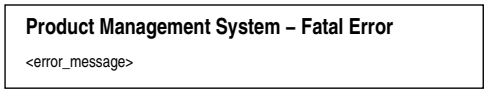

Figure D.19: The *fatalErrorPage* layout.

RS05.R1 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.loginPage.registerUserLink *(Section D.5)*
- thePage => PmsUI.registerUserPage *(Section D.5)*

RS08.R1 : raFormHandledByPhpFunction *(Section C.2.5)*

- theForm => PmsUI.loginPage.form *(Section D.5)*
- theFunction => SecurityMechanisms.login *(Section E.5.1)*
- theSuccessPage => PmsUI.mainMenuPage *(Section D.5)*

D.6.2 Register New User Page

The *registerNewUserPage* (Figure D.4) shall be implemented in accordance with the following requirements.

RS07.R3 : raPageImplementedInPhp *(Section C.2.1)*

- thePage => PmsUI.registerUserPage *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

RS05.R2 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.registerUserPage.loginLink *(Section D.5)*
- thePage => PmsUI.loginPage *(Section D.5)*

RS08.R2 : raFormHandledByPhpFunction *(Section C.2.5)*

- theForm => PmsUI.registerUserPage.form *(Section D.5)*
- theFunction => SecurityMechanisms.registerUser *(Section E.5.1)*
- theSuccessPage => PmsUI.registrationSuccessPage *(Section D.5)*

D.6.3 Registration Success Page

The *registrationSuccessPage* (Figure D.5) shall be implemented in accordance with the following requirements.

RS07.R4 : raPageImplementedInPhp *(Section C.2.1)*

- thePage => PmsUI.registrationSuccessPage *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

RS05.R3 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.registrationSuccessPage.registrationLink *(Section D.5)*
- thePage => PmsUI.registerNewUserPage *(Section D.5)*

RS05.R4 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.registrationSuccessPage.loginLink *(Section D.5)*
- thePage => PmsUI.loginPage *(Section D.5)*

D.6.4 Main Menu Page

The *mainMenuPage* (Figure D.6) shall be implemented in accordance with the following requirements.

RS07.R5 : raPageImplementedInPhp *(Section C.2.1)*

- thePage => PmsUI.mainMenuPage *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

RS05.R5 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.mainMenuPage.productListLink *(Section D.5)*
- thePage => PmsUI.productListPage *(Section D.5)*

RS05.R6 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.mainMenuPage.categoryListLink *(Section D.5)*
- thePage => PmsUI.categoryListPage *(Section D.5)*

RS05.R7 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.mainMenuPage.contactListLink *(Section D.5)*
- thePage => PmsUI.contactListPage *(Section D.5)*

RS05.R8 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.mainMenuPage.userListLink *(Section D.5)*
- thePage => PmsUI.registeredUserListPage *(Section D.5)*

RS08.R3 : raLinkAvailableWithRole *(Section C.2.3)*

- theLink => PmsUI.mainMenuPage.categoryListLink *(Section D.5)*
- theRole => PmsSEC.staffRole *(Section D.3)*

RS08.R4 : raLinkAvailableWithRole *(Section C.2.3)*

- theLink => PmsUI.mainMenuPage.contactListLink *(Section D.5)*
- theRole => PmsSEC.staffRole *(Section D.3)*

RS08.R5 : raLinkAvailableWithRole *(Section C.2.3)*

- theLink => PmsUI.mainMenuPage.userListLink *(Section D.5)*
- theRole => PmsSEC.staffRole *(Section D.3)*

RS08.R6 : raLinkCallsPhpFunction *(Section C.2.4)*

- theLink => PmsUI.mainMenuPage.logoffLink *(Section D.5)*
- theFunction => SecurityMechanisms.logoff *(Section E.5.1)*
- theSuccessPage => PmsUI.loginPage *(Section D.5)*

D.6.5 Product List Page

The *productListPage* (Figure D.7) shall be implemented in accordance with the following requirements.

RS07.R6 : raPageImplementedInPhp *(Section C.2.1)*

- thePage => PmsUI.productListPage *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

RS04.R7 : raClassList *(Section C.2.6)*

- theCrudList => PmsUI.productListPage.list *(Section D.5)*
- theClass => PM.Product *(Section B.3)*
- theEditPage => PmsUI.editProductPage *(Section D.5)*

RS04.R8 : raGroupClassList *(Section C.2.7)*

- theCrudList => PmsUI.productListPage.list *(Section D.5)*
- theAssociation => PM.R03 *(Section B.3)*
- theGroupingAttribute => PM.ProductCategory.name *(Section B.3)*

RS05.R9 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.productListPage.mainMenuLink *(Section D.5)*
- thePage => PmsUI.mainMenuPage *(Section D.5)*

D.6.6 Edit Product Page

The *editProductPage* (Figure D.8) shall be implemented in accordance with the following requirements.

RS04.R9 : raSelectItemCreatesLink *(Section C.2.11)*

- theSelectItem => PmsUI.editProductPage.category *(Section D.5)*
- theAssociation => PM.R03 *(Section B.3)*
- theAttribute => PM.ProductCategory.name *(Section B.3)*

RS04.R10 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editProductPage.productName *(Section D.5)*
- theAttribute => PM.Product.name *(Section B.3)*

RS04.R11 : raSelectItemCreatesLink *(Section C.2.11)*

- theSelectItem => PmsUI.editProductPage.supplier *(Section D.5)*
- theAssociation => PM.R02 *(Section B.3)*
- theAttribute => PM.Supplier.name *(Section B.3)*

RS04.R12 : raTextAreaUpdatesAttribute *(Section C.2.10)*

- theTextArea => PmsUI.editProductPage.description *(Section D.5)*
- theAttribute => PM.Product.description *(Section B.3)*

RS04.R13 : raClassList *(Section C.2.6)*

- theCrudList => PmsUI.editProductPage.productVariantList *(Section D.5)*
- theClass => PM.ProductVariant *(Section B.3)*
- theEditPage => PmsUI.editProductVariantPage *(Section D.5)*

D.6.7 Edit Product Variant Page

The *editProductVariantPage* (Figure D.9) shall be implemented in accordance with the following requirements.

RS04.R14 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editProductVariantPage.description *(Section D.5)*
- theAttribute => PM.ProductVariant.description *(Section B.3)*

RS04.R15 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editProductVariantPage.price *(Section D.5)*
- theAttribute => PM.ProductVariant.price *(Section B.3)*

D.6.8 Product Category List Page

The *productCategoryListPage* (Figure D.10) shall be implemented in accordance with the following requirements.

RS07.R7 : raPageImplementedInPhp *(Section C.2.1)*

- thePage => PmsUI.productCategoryListPage *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

RS04.R16 : raClassList *(Section C.2.6)*

- theCrudList => PmsUI.productCategoryListPage.list *(Section D.5)*
- theClass => PM.ProductCategory *(Section B.3)*
- theEditPage => PmsUI.editProductCategoryPage *(Section D.5)*

RS05.R10 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.productCategoryListPage.mainMenuLink *(Section D.5)*
- thePage => PmsUI.mainMenuPage *(Section D.5)*

D.6.9 Edit Product Category Page

The *editProductCategoryPage* (Figure D.11) shall be implemented in accordance with the following requirements.

RS04.R17 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editProductCategoryPage.name *(Section D.5)*
- theAttribute => PM.ProductCategory.name *(Section B.3)*

RS04.R18 : raTextAreaUpdatesAttribute *(Section C.2.10)*

- theTextArea => PmsUI.editProductCategoryPage.notes *(Section D.5)*
- theAttribute => PM.ProductCategory.notes *(Section B.3)*

D.6.10 Contact List Page

The *contactListPage* (Figure D.12) shall be implemented in accordance with the following requirements.

RS07.R8 : raPageImplementedInPhp *(Section C.2.1)*

- thePage => PmsUI.contactListPage *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

RS04.R19 : raClassList *(Section C.2.6)*

- theCrudList => PmsUI.contactListPage.list *(Section D.5)*
- theClass => CM.Contact *(Section B.2)*
- theEditPage => PmsUI.editContactPage *(Section D.5)*

RS05.R11 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.contactListPage.mainMenuLink *(Section D.5)*
- thePage => PmsUI.mainMenuPage *(Section D.5)*

D.6.11 Edit Contact Page

The *editContactPage* (Figure D.13) shall be implemented in accordance with the following requirements.

RS04.R20 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editContactPage.name *(Section D.5)*
- theAttribute => CM.Contact.name *(Section B.2)*

RS04.R21 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editContactPage.address *(Section D.5)*
- theAttribute => CM.Contact.address *(Section B.2)*

RS04.R22 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editContactPage.telephone *(Section D.5)*
- theAttribute => CM.Contact.telephone *(Section B.2)*

RS04.R23 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editContactPage.mobile *(Section D.5)*
- theAttribute => CM.Contact.mobile *(Section B.2)*

RS04.R24 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editContactPage.email *(Section D.5)*
- theAttribute => CM.Contact.email *(Section B.2)*

RS04.R25 : raClassList *(Section C.2.6)*

- theCrudList => PmsUI.editContactPage.relationshipList *(Section D.5)*
- theClass => CM.ContactRelationsip *(Section B.2)*
- theEditPage => PmsUI.editContactRelationshipPage *(Section D.5)*

RS04.R26 : raClassList *(Section C.2.6)*

- theCrudList => PmsUI.editContactPage.noteList *(Section D.5)*
- theClass => CM.Note *(Section B.2)*
- theEditPage => PmsUI.editNotePage *(Section D.5)*

D.6.12 Edit Contact Relationship Page

The *editContactRelationshipPage* (Figure D.14) shall be implemented in accordance with the following requirements.

RS04.R27 : raSelectItemCreatesLink *(Section C.2.11)*

- theSelectItem => PmsUI.editRelationshipPage.fromContact *(Section D.5)*
- theAssociation => CM.R02'from' *(Section B.2)*
- theAttribute => CM.Contact.name *(Section B.2)*

RS04.R28 : raSelectItemCreatesLink *(Section C.2.11)*

- theSelectItem => PmsUI.editRelationshipPage.name *(Section D.5)*
- theAssociation => CM.R01 *(Section B.2)*
- theAttribute => CM.RelationshipType.fromName *(Section B.2)*

RS04.R29 : raSelectItemCreatesLink *(Section C.2.11)*

- theSelectItem => PmsUI.editRelationshipPage.toContact *(Section D.5)*
- theAssociation => CM.R02'to' *(Section B.2)*
- theAttribute => CM.Contact.name *(Section B.2)*

D.6.13 Edit Note Page

The *editNotePage* (Figure D.15) shall be implemented in accordance with the following requirements.

RS04.R30 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editNotePage.dateTime *(Section D.5)*
- theAttribute => CM.Note.creationTime *(Section B.2)*

RS04.R31 : raTextAreaUpdatesAttribute *(Section C.2.10)*

- theTextArea => PmsUI.editNotePage.note *(Section D.5)*
- theAttribute => CM.Note.notes *(Section B.2)*

D.6.14 Registered User List Page

The *registeredUserListPage* (Figure D.16) shall be implemented in accordance with the following requirements.

RS07.R9 : raPageImplementedInPhp *(Section C.2.1)*

- thePage => PmsUI.registeredUserListPage *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

RS04.R32 : raClassList *(Section C.2.6)*

- theCrudList => PmsUI.registeredUserListPage.userList *(Section D.5)*
- theClass => SEC.RegisteredUser *(Section B.4)*
- theEditPage => PmsUI.editRegisteredUserPage *(Section D.5)*

RS05.R12 : raLinkOpensPage *(Section C.2.2)*

- theLink => PmsUI.registeredUserListPage.mainMenuLink *(Section D.5)*
- thePage => PmsUI.mainMenuPage *(Section D.5)*

D.6.15 Edit Registered User Page

The *editRegistereduserPage* (Figure D.17) shall be implemented in accordance with the following requirements.

RS04.R33 : raTextInputUpdatesAttribute *(Section C.2.8)*

- theTextInput => PmsUI.editUserPage.username *(Section D.5)*
- theAttribute => SEC.username *(Section B.4)*

RS04.R34 : raPasswordInputUpdatesAttribute *(Section C.2.9)*

- theTextInput => PmsUI.editUserPage.password *(Section D.5)*
- theAttribute => SEC.password *(Section B.4)*

RS04.R35 : raSelectItemCreatesLink *(Section C.2.11)*

- theSelectItem => PmsUI.editUserPage.role *(Section D.5)*
- theAssociation => SEC.R01 *(Section B.4)*
- theAttribute => SEC.Role.name *(Section B.4)*

D.6.16 *PMS User Interface* implementation

The *PMS User Interface* shall be implemented in accordance with the following requirements.

RS07.R10 : raUiImplementedInPhp *(Section C.5.1)*

- theUI => PmsUI *(Section D.5)*
- thePhpDomain => PHP *(Section B.9)*

RS09.R1 : raPhpImplementedOnApache *(Section C.5.2)*

- thePhpDomain => PHP *(Section B.9)*
- theApacheDomain => Apache *(Section B.11)*
- theStartPage => PmsUI.loginPage *(Section D.5)*

RS11.R2 : raApacheImplementedOnLinux *(Section C.5.3)*

- theApacheDomain => Apache *(Section B.11)*
- theLinuxDomain => Linux *(Section B.8)*

D.7 Conclusion

In this appendix, I have presented a demonstration of how an *Aspect-Oriented Specification* can be formed in accordance with an *Aspect-Oriented Specification Archetype*. In particular, I presented an *Aspect-Oriented Specification* for the *Product Management System* introduced in Chapter 6. This specification was formed in accordance with the *LAMP Aspect-Oriented Specification Archetype* presented on Appendix C.

The significance of this demonstration is that the *LAMP Aspect-Oriented Specification Archetype* contains nothing whatsoever about the *Product Management System*, yet it describes an architecture that can be used to create a complete *Aspect-Oriented Specification* for the *Product Management System*. It is this autonomous nature of *Aspect-Oriented Specification Archetypes* that allows them to be reused to specify any number of different *LAMP* database applications.

Appendix

E

Proof of concept - *LAMP Implementation model*

Contents

E.1 Introduction 189
E.1.1 Organisation of this appendix 189
E.2 General implementation rules 190
E.2.1 *Application Mechanisms* domain model 190
E.2.2 Implementation Rule 'irFileAndDirectoryStructure' 190
E.3 User interface implementation rules 190
E.3.1 *User Interface Mechanisms* domain model 190
E.3.2 Implementation Rule 'irStartPage' 191
E.3.3 Implementation Rule 'irUiConfig' 192
E.3.4 Implementation Rule 'irPage' 193
E.3.5 Implementation Rule 'irStaticPageItems' 195
E.3.6 Implementation Rule 'irFunctionLink' 196
E.3.7 Implementation Rule 'irPageLink' 198
E.3.8 Implementation Rule 'irRolePageLink' 199
E.3.9 Implementation Rule 'irForm' 201
E.3.10 Implementation Rule 'irClassList' 206
E.3.11 Implementation Rule 'irGroupedClassList' 212
E.3.12 Implementation Rule 'irEditClassPage' 215
E.3.13 Implementation Rule 'irEditClassForm' 218
E.3.14 Implementation Rule 'irAttributeTextInput' 220
E.3.15 Implementation Rule 'irAttributePasswordInput' 222
E.3.16 Implementation Rule 'irAttributeTextArea' 223
E.3.17 Implementation Rule 'irLinkObjectSelect' 224

E.4 Database implementation rules 226
E.4.1 *Database Mechanisms* domain model 226
E.4.2 Implementation Rule 'irDatabaseConfig' 226
E.4.3 Implementation Rule 'irDatabaseScripts' 227
E.4.4 Implementation Rule 'irPersistentClass' 230
E.4.5 Implementation Rule 'irCorrespondingClasses' 237
E.5 Security implementation rules 237
E.5.1 *Security Mechanisms* domain model 237
E.5.2 Implementation Rule 'irSecurityConfig' 238
E.6 Conclusion 239

E.1 Introduction

In this appendix, I present an *Implementation Model* (Section 4.8) that has been developed to implement any *Aspect-Oriented Specification* that conforms to the *LAMP Aspect-Oriented Specification Archetype* contained in Appendix C. In particular, it can be used to implement the *Product Management System Aspect-Oriented Specification* presented in Appendix D.

E.1.1 Organisation of this appendix

Within this appendix, each *Implementation Rule* (Figure 4.9 on page 71) is presented in the following parts.

- The *Requirement Archetypes* of *Requirements* to which the rule applies (association *R25* depicted in the class diagram shown in Figure 4.9).
- Any constraints that apply to the application of the rule. For example, some rules operate on sets of related requirements. Constraints might be used to describe how such requirements are related.
- A reference to *Product Management System* code that shows how the rule has been applied.
- A textual description of the rule (the *ImplementationRule.description* attribute depicted in Figure 4.9 on page 71).
- The mechanisms, if any, that are used by the rule (association *R24* depicted in the class diagram shown in Figure 4.9).
- A description of the actual implementation of the rule (the *ImplementationRule.implementation* attribute depicted in Figure 4.9). Each description will normally include code fragments that are used to implement parts of the target application. These code fragments often include tags of the form <EXAMPLE-TAG> which act as place-holders for information (text) provided by the implementer (the programmer in this case).

Note that *Domain Models* and *Domain Model Elements* (subject matter) are referenced throughout this appendix using the *Reference Names* associated with *Requirement Archetypes* specified in the *LAMP Aspect-Oriented Specification Archetype* presented in Appendix C. For example, references made by the *raLinkOpensPage* requirement archetype (Section C.2.2) description include *theLink* and *thePage*. Components of a referenced element are referred to using

the conventional *dot* notation. For example, the *name* attribute of an $\frac{X}{T}$*UML class* referred to by *thePage* would be referred to as *thePage.name*.

E.2 General implementation rules

E.2.1 *Application Mechanisms* domain model

The web interface employed by the LAMP architecture is essentially *stateless*. That is, the server has no history of interactions between a specific user and the application. Each request from a web browser is independent of all others. PHP works-round this problem by implementing the concept of a *session*. The session mechanism allows the application to record data during the processing of a request and retrieval of that information during the processing of a later request from the same user.

The *Application Mechanisms* domain model comprises a collection of *PHP* functions that implement various *Session* related functions. This *Domain Model* is an instance of the *PHP Web Programming Language* domain model (Section B.9) and is contained in Section G.2 on page 302.

E.2.2 Implementation Rule 'irFileAndDirectoryStructure'

Description. The *irFileAndDirectoryStructure* implementation rule describes the required organisation of implementation files and directories.

Directory structure. All PHP source files developed within the rules of the LAMP architecture shall be copied to a subdirectory under the Apache *document root* (usually `/var/www` on Linux). The name of the subdirectory shall reflect the name of the application and be referred to as the *Application Directory*. For example, the user interface pages created for the *Product Management System* would be stored in the `/var/www/pms` directory.

E.3 User interface implementation rules

E.3.1 *User Interface Mechanisms* domain model

The *User Interface Mechanisms* domain model comprises a collection of *PHP* functions that implement various user interface related functions including those

required to systematically construct HTML pages. This *Domain Model* is an instance of the *PHP Web Programming Language* domain model and comprises the PHP source code listed in Section G.4 on page 304.

E.3.2 Implementation Rule 'irStartPage'

Implemented Requirements. The *irStartPage* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raPhpImplementedOnApache* requirement archetype (Section C.5.2)
  - *thePhpDomain* = PHP.
  - *theApacheDomain* = Apache.
  - *theStartPage* : UI.Page

Description. The *irStartPage* implementation rule describes the generation of a start-up web page for an application.

Example. See the `index.html` source file (Section F.4.1 on page 249) for an example of code generated using this rule.

Start page. An application is started by directing a web browser to the *Application Directory* on the server hosting the application. For example, the *Product Management System* hosted at `softeng.anu.edu.au` would be started by directing a browser to `http://softeng.anu.edu.au/pms`. The Apache web server will respond by serving a HTML page. This will normally be the page defined in a file named `index.html`.

For the purposes of the LAMP architecture, an `index.html` file with the following contents shall be created and stored in the *Application Directory*. This file tells the *Apache web-server* to return a specified page rather than the `index.html` page itself.

```
1 <html>
2 <head>
3 <meta http-equiv="REFRESH"
4       content="0; URL=<START-PAGE-NAME>Page.php">
5 </head>
6 </html>
```

Where:

<**START-PAGE-NAME**> *theStartPage.name* attribute value.

E.3.3 Implementation Rule 'irUiConfig'

Implemented Requirements. The *irUiConfig* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raUiImplementedInPhp* requirement archetype (Section C.5.1)
  - *theUI* : UI.
  - *thePhpDomain* = PHP.

Description. The *irUiConfig* implementation rule is used to implement a user interface configuration file for an application developed using the LAMP architecture.

Example. See the `userInterfaceConfiguration.php` source file (Section F.2.1 on page 244) for an example of code generated using this rule.

User interface configuration file. There shall be a PHP file named as follows and stored in the *Application Directory*.

```
1 userInterfaceConfiguration.php
```

The file shall have the following contents:

```
1 <?php
2
3 session_start();
4
5 define("APPLICATION_NAME",  "<APPLICATION-NAME>");
6
7 define("PAGE_FOOTER",       "<FOOTER-TEXT>");
8
9 ?>
```

Where:

<**APPLICATION-NAME**> The name of the application. This text will be displayed at the top of each user interface page.

<**FOOTER-TEXT**> Text that will be displayed in a small font at the botton of each user interface page.

E.3.4 Implementation Rule 'irPage'

Implemented Requirements. The *irPage* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raPageImplementedInPhp* requirement archetype (Section C.2.1)
  - *thePage* : UI.Page
  - *thePhpDomain* = PHP.

Description. The *irPage* implementation rule is used to implement the structure of *thePage*.

Example. See the `loginPage.php` source file (Section F.5.1 on page 249) for an example of code generated using this rule.

Page source file. For *thePage* there shall be a PHP file named as follows and stored in the *Application Directory*.

```
1 <NAME>.php
```

Where:

<**NAME**> *thePage.name* attribute value.

The file shall have the following contents:

```
1 <?php
2
3 require_once('userInterfaceMechanisms.php');
4
5 session_start();
6
```

```
7  show<NAME>();
8
9  function show<NAME>() {
10   uiStartPage('<TITLE>');
11   uiSetCurrentUrl('<FORMATTED-NAME>', '<NAME>Page.php');
12   <UI-PAGE-ITEMS>
13   uiEndPage();
14 }
15
16 ?>
```

Where:

<**NAME**> *thePage.name* attribute value.

<**TITLE**> *thePage.title* attribute value.

<**FORMATTED-NAME**> *thePage.title* attribute value formatted to include appropriate capitalisation and spaces (e.g., 'mainMenu' could be formatted as 'Main Menu').

<**UI-PAGE-ITEMS**> Implementation of the following *UI.PageItems* as applicable and described in the identified *Implementation Rules*.

- Main Heading
  - *irStaticPageItems* implementation rule (Section E.3.5)
- Sub-heading
  - *irStaticPageItems* implementation rule (Section E.3.5)
- Simple Text
  - *irStaticPageItems* implementation rule (Section E.3.5)
- Link
  - *irPageLink* implementation rule (Section E.3.7)
  - *irRolePageLink* implementation rule (Section E.3.8)
- Form
  - *irForm* implementation rule (Section E.3.9)
- CRUD List
  - *irClassList* implementation rule (Section E.3.10)

E.3.5 Implementation Rule 'irStaticPageItems'

Implemented Requirements. The *irStaticPageItems* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raPageImplementedInPhp* requirement archetype (Section C.2.1)
  - *thePage* : UI.Page
  - *thePhpDomain* = PHP.

Description. The *irStaticPageItems* implementation rule is used to implement all *Main Heading*, *Sub-heading* and *Simple Text* items (Section B.5.2) displayed on *thePage*.

Example. See lines 12-14 of the `loginPage.php` source file (Section F.5.1 on page 249) for an example of code generated using this rule.

Mechanisms. The *irStaticPageItems* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiHeading** *(Section G.4)*: Displays a *Main Heading* on a *UI.Page*.
- **User Interface Mechanisms.uiSubHeading** *(Section G.4)*: Displays a *Sub-heading* on a *UI.Page*.
- **User Interface Mechanisms.uiText** *(Section G.4)*: Displays a *Simple Text* item on a *UI.Page*.

***Main Heading* items.** *UI.MainHeadings* shall be implemented using the following PHP code.

```
1 uiHeading('<HEADING-TEXT>');
```

Where:

<**HEADING-TEXT**> The *MainHeading.text* attribute value.

***Sub-Heading* items.** *UI.SubHeadings* shall be implemented using the following PHP code.

```
1 uiSubHeading('<HEADING-TEXT>');
```

Where:

<**HEADING-TEXT**> The *SubHeading.text* attribute value.

***Simple Text* items.** *UI.SimpleText* items shall be implemented using the following PHP code.

```
1 uiText('<TEXT>');
```

Where:

<**TEXT**> The *SimpleText.text* attribute value.

E.3.6 Implementation Rule 'irFunctionLink'

Implemented Requirements. The *irFunctionLink* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raLinkCallsPhpFunction* requirement archetype (Section C.2.4)
  - *theLink* : UI.Link
  - *theFunction* : PHP.Function
  - *theSuccessPage* : UI.Page

Constraints. The *irFunctionLink* implementation rule shall be used to implement each *UI.Link* referenced by an *raLinkCallsPhpFunction* (Section C.2.4) requirement and not by an *raLinkAvailableWithRole* (Section C.2.3) requirement.

Description. The *irFunctionLink* implementation rule is used to implement *theLink*. When *theLink* is clicked by a user, *theFunction* will be invoked.

Example. See line 19 of the `mainMenuPage.php` source file (Section F.6.1 on page 253) for an example of code generated using this rule.

Mechanisms. The *irFunctionLink* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiLink** *(Section G.4)*: Displays a user clickable link on a page

Instructions. Function links are implemented using the following PHP code.

```
1  uiLink('<PHP-FUNCTION-NAME>Bridge.php', '<TEXT>');
```

Where:

<**PHP-FUNCTION-NAME**> The name of *theFunction*.

<**TEXT**> *theLink.text* attribute value.

Bridge to PHP function. For each function link, a bridge function shall be implemented in a file with the following name. This PHP file will be called when the link is clicked.

```
1  <PHP-FUNCTION-NAME>Bridge.php
```

Where:

<**PHP-FUNCTION-NAME**> The name of *theFunction*.

The contents of each Link Bridge file shall be as follows. For example, see the `logoffBridge.php` source file (Section F.6.2 on page 254).

```
1  <?php
2
3  require_once('<MODULE-NAME>');
4  require_once('userInterfaceMechanisms.php');
5
6  session_start();
7
8  try {
9    <FUNCTION-CALL>
10   uiShowUrl('<SUCCESS-PAGE-URL>');
11 } catch (Exception $e) {
```

```
12    uiErrorPage($e);
13  }
14
15  ?>
```

Where:

<**MODULE-NAME**> This is the name of the PHP module in which the PHP function that implements the joined operation is declared.

<**FUNCTION-CALL**> A call to *theFunction*.

<**SUCCESS-PAGE-URL**> The name of the file that implements *theSuccessPage*.

E.3.7 Implementation Rule 'irPageLink'

Implemented Requirements. The *irPageLink* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raLinkOpensPage* requirement archetype (Section C.2.2)
  - *theLink* : UI.Link
  - *thePage* : UI.Page

Constraints. The *irPageLink* implementation rule shall be used to implement each *UI.Link* referenced by an *raLinkCallsPhpFunction* (Section C.2.4) requirement and not by an *raLinkAvailableWithRole* (Section C.2.3) requirement.

Description. The *irPageLink* implementation rule is used to implement *theLink*. When *theLink* is clicked by a user, *thePage* will be displayed.

Example. See line 19 of the `registerUserPage.php` source file (Section F.5.3 on page 251) for an example of code generated using this rule.

Mechanisms. The *irPageLink* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiLink** *(Section G.4)*: Adds a *UI.Link* to a *UI.Page*

Instructions. *theLink* is implemented using the following PHP code.

```
1  uiLink('<PAGE-NAME>.php', '<TEXT>');
```

Where:

<**PAGE-NAME**> *thePage.name* attribute value.

<**TEXT**> *theLink.text* attribute value.

E.3.8 Implementation Rule 'irRolePageLink'

Implemented Requirements. The *irRolePageLink* implementation rule implements a set of requirements that conform to the following related *Requirement Archetypes*.

- *raLinkOpensPage* requirement archetype (Section C.2.2)
  - *theLink* : UI.Link
  - *thePage* : UI.Page
- *raLinkAvailableWithRole* requirement archetype (Section C.2.3)
  - *theLink* : UI.Link
  - *theRole* : SEC.Role

Constraints. The *irRolePageLink* implementation rule shall be used to implement all *UI.Links* that are referenced by an *raLinkOpensPage* (Section C.2.2) requirement as well as an *raLinkAvailableWithRole* (Section C.2.3) requirement.

Implementing related requirements

The *irRolePageLink* implementation rule is an example of a rule that implements a set of related *Requirements* as discussed in Section 4.8.1 on page 69. In such cases, the rule shall be applied to a set of requirements that reference, directly or indirectly, a common *Domain Model* or *Domain Model Element*. For example, the *irRolePageLink* implementation rule described in this section will be applied to an *raLinkOpensPage* (Section C.2.2) requirement and *raLinkAvailableWithRole* (Section C.2.3) requirement that both reference the same *UI.Link* via the *theLink* reference in each archetype.

Description. The *irRolePageLink* implementation rule is used to implement a *theLink* which is only displayed if the current user has *theRole*.

Example. See lines 14-18 of the `mainMenuPage.php` source file (Section F.6.1 on page 253) for an example of code generated using this rule.

Mechanisms. The *irRolePageLink* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiLink** *(Section G.4)*: Adds a *UI.Link* to a *UI.Page*
- **Security Mechanisms.userHasRole** *(Section G.5)*: Returns true if the current logged-in user has a specified *Security.Role*.

Weaving in the security policy.

Security policy specified in an *Aspect-Oriented Specification* that complies with the *LAMP Aspect-Oriented Specification Archetype*, is implemented by weaving security code throughout the *User Interface Mechanisms* (Section G.4.1 on page 304) and code used to implement pages. For example, user interface mechanisms that involve the creation, modification and deletion of $\frac{X}{T}$*UML* objects, check that the current user has appropriate permissions.

Instructions. *theLink* is implemented using the following PHP code.

```
1  if (userHasRole('<ROLE-NAME>')) {
2    uiLink('<PAGE-NAME>.php', '<TEXT>');
3  }
```

Where:

<**ROLE-NAME**> *theRole.name* attribute value.

<**PAGE-NAME**> *thePage.name* attribute value.

<**TEXT**> *theLink.text* attribute value.

Optimisation. If there is more than one *UI.Link* positioned together on a page and subject to the same security *Requirement*, the following optimisation can be applied.

```
1  if (userHasRole('<ROLE-NAME>')) {
2    uiLink('<PAGE-NAME>.php', '<TEXT>'); // first link
3    ...
4    uiLink('<PAGE-NAME>.php', '<TEXT>'); // nth link
5  }
```

E.3.9 Implementation Rule 'irForm'

Implemented Requirements. The *irForm* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raFormHandledByPhpFunction* requirement archetype (Section C.2.5)
  - *theForm* : UI.Form
  - *theFunction* : PHP.Function
  - *theSuccessPage* : UI.Page

Description. The *irForm* implementation rule is used to implement simple *UI.Forms* that collect user input that is to be processed by *theFunction*.

Note that the *irEditClassForm* implementation rule (Section E.3.13) should be used to implement *UI.Forms* that manipulate instances of $\frac{X}{T}$UML classes.

Figure E.1 shows a block diagram depicting the elements involved in the implementation of each *UI.Form*. They are constructed using *UI.FormItems* such as *UI.TextInput* and *UI.TextArea* items which collect data entered by users. When *theForm's UI.SubmitButton* is clicked by a user, the *<phpFunction>Bridge* page is requested. This page collects data from the form's *UI.FormItems* and passes them to *theFunction* declared in a PHP module (an instance of the *PHP Web Programming Language* domain model). If *theFunction* executes without error, the *UI.Page* containing *theForm* will be redisplayed. If, on the other hand, *theFunction* raises an exception, the generic *errorPage* (Figure D.18) will be displayed. The '*Return to <Previous Page>*' link on this *errorPage* will be linked back to the *UI.Page* containing *theForm*.

Example. See lines 15-21 of the `loginPage.php` source file (Section F.5.1 on page 249) for an example of code generated using this rule.

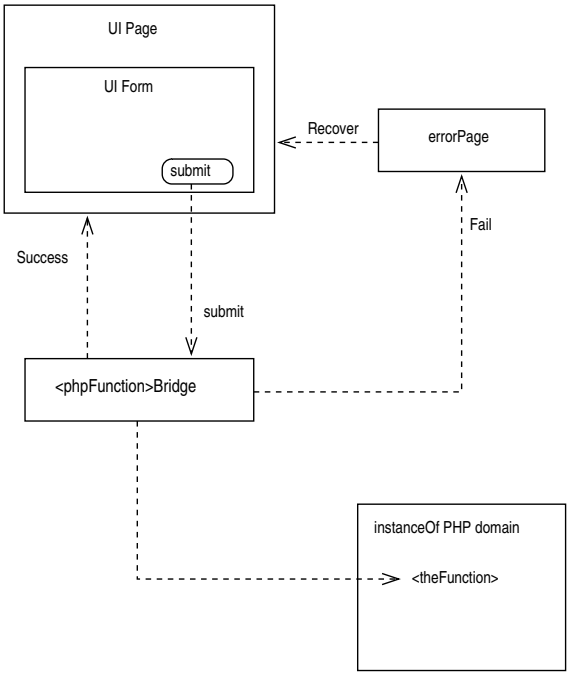

Figure E.1: Block diagram depicting the organisation of PHP modules required to implement Forms.

Mechanisms. The *irForm* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiStartForm** *(Section G.4)*: Starts a *HTML Form*.
- **User Interface Mechanisms.uiEndForm** *(Section G.4)*: Ends a *HTML Form*.
- **User Interface Mechanisms.uiStartTable** *(Section G.4)*: Starts a *HTML Table*.
- **User Interface Mechanisms.uiEndTable** *(Section G.4)*: Ends a *HTML Table*.
- **User Interface Mechanisms.uiSubmitButton** *(Section G.4)*: Implements a *HTML Form submit button*.
- **User Interface Mechanisms.uiTextInput** *(Section G.4)*: Displays a *UI.TextInput* item on a *UI.Page*.
- **User Interface Mechanisms.uiPasswordInput** *(Section G.4)*: Displays a *UI.PasswordInput* item on a *UI.Page*.

- **User Interface Mechanisms.uiTextArea** *(Section G.4)*: Displays a *UI.TextArea* item on a *UI.Page*.

Form structure. Simple *UI.Forms* shall be implemented using the following PHP code.

```
uiStartForm('<PHP-FUNCTION-NAME>Bridge.php>');
uiStartTable(<TABLE-WIDTH>);
<FORM-ITEMS>
uiEndTable();
uiSubmitButton('<SUBMIT-LABEL>');
uiEndForm();
```

Where:

<**PHP-FUNCTION-NAME**> The name of *theFunction*.

<**TABLE-WIDTH**> *UI.FormItems* are arranged using an HTML table. This parameter specifies the width of this table and hence the width of the form.

<**FORM-ITEMS**> Implementation of the following *UI.PageItems* as applicable. Rules for implementing these items follow.

- Text Input items
- Password Input items
- Text Area items

<**SUBMIT-LABEL**> *theForm.submitLabel* attribute value.

Text Input items. *UI.TextInput* items are labeled input boxes that allow the user to view, edit and enter single lines of text. They shall be implemented using the following code.

```
uiTextInput
  (<LABEL>, <NAME>, <TEXT>, <WIDTH>);
```

Where:

<**LABEL**> *UI.TextInput.label* attribute value.

<**NAME**> Form items, including *UI.TextInput* items, are referred to in PHP functions using *names*. This parameter is the *UI.TextInput.name* attribute value.

<**TEXT**> The *UI.TextInput.defaultText* attribute value.

<**WIDTH**> The *UI.TextInput.textWidth* attribute value.

Password Input items. *UI.PasswordInput* items are labeled text input boxes that allow the user to enter single lines of text that are echoed as a series of asterisks. They shall be implemented using the following code.

```
uiPasswordInput
  (<LABEL>, <NAME>, <WIDTH>);
```

Where:

<**LABEL**> The *UI.PasswordInput.Label* attribute value.

<**NAME**> Form items, including *UI.PasswordInput* items, are referred to in PHP functions using *names*. This parameter is the *UI.PasswordInput.name* attribute value.

<**WIDTH**> The *UI.PasswordInput.textWidth* attribute value.

Text Area items. *UI.TextArea* items are labeled text input boxes that allow the user to view, edit and enter multiple lines of text. They shall be implemented using the following code.

```
uiTextArea
  (<LABEL>, <NAME>, <TEXT>, <WIDTH>, <HEIGHT>);
```

Where:

<**LABEL**> The *UI.TextArea.label* attribute value.

<**NAME**> Form items, including *UI.TextArea* items, are referred to in PHP functions using *names*. This parameter is the *UI.TextArea.name* attribute value.

<**WIDTH**> The *UI.TextArea.textWidth* attribute value.

<**HEIGHT**> The *UI.TextArea.textHeight* attribute value.

Bridge to PHP function. A bridge function shall be implemented in a file with the following name. This PHP file will be called when *theForm's Submit Button* on a form is *clicked*. Its purpose is to collect data entered on the form and pass it to *theFunction* for processing.

```
<PHP-FUNCTION-NAME>Bridge.php
```

Where:

<**PHP-FUNCTION-NAME**> The name of *theFunction*.

The contents of each Form Bridge file shall be as follows. For example, see the `loginBridge.php` source file (Section F.5.2 on page 250).

```
<?php

require_once('userInterfaceMechanisms.php');
require_once('<MODULE-NAME>');
session_start();

<FORM-DATA-RETRIEVAL>

try {
  <FUNCTION-CALL>
  uiShowUrl('<SUCCESS-PAGE-URL>');
} catch (Exception $e) {
  uiErrorPage($e);
}

?>
```

Where:

<**MODULE-NAME**> This is the name of the PHP module in which *theFunction* is declared.

<**FORM-DATA-RETRIEVAL**> Each form data item is retrieved for processing using the following PHP code.

```
$<PARAMETER-NAME> = $HTTP_POST_VARS['<FORM-ITEM-NAME>'];
```

Where:

<**PARAMETER-NAME**> The name of the parameter to be retrieved.

<**FORM-ITEM-NAME**> The *UI.FormItem.name* attribute value of the item from which the parameter value is to be read.

<**FUNCTION-CALL**> A call to *theFunction*. Parameters retrieved using the statements described above can be passed to the function as required.

<**SUCCESS-PAGE-URL**> The URL of *theSuccessPage*.

E.3.10 Implementation Rule 'irClassList'

Implemented Requirements. The *irClassList* implementation rule implements a set of requirements that conform to the following related *Requirement Archetypes*.

- *raClassList* requirement archetype (Section C.2.6)
  - *theCrudList* : UI.CrudList
  - *theClass* : xtUML.Class
  - *theEditPage* : UI.Page
- *raSecureClass* requirement archetype (Section C.3.2)
  - *theClass* : xtUML.Class
  - *theProtectedItem* : SEC.ProtectedItem

Constraints. *theClass* referenced by the *raClassList* (Section C.2.6) requirement must be the same as that referenced by the *raSecureClass* (Section C.3.2) requirement.

Description. The *irClassList* implementation rule is used to implement *UI.CrudLists* that allow users to create, view, edit and delete instances of *theClass* subject to applicable security permissions.

Note that the *irGroupedClassList* implementation rule (Section E.3.11) should be used if *theCrudList* is referenced by an *raGroupClassList* (Section C.2.7) requirement.

Figure E.1 shows a block diagram depicting the elements involved in the implementation of *Class Lists*. At the centre is *theCrudList* which lists all instances of *theClass*. When the '*Edit*' link next to an instance of *theClass* is clicked, the '*edit*<*class*>*Page*', implemented using the *irEditClassPage* implementation

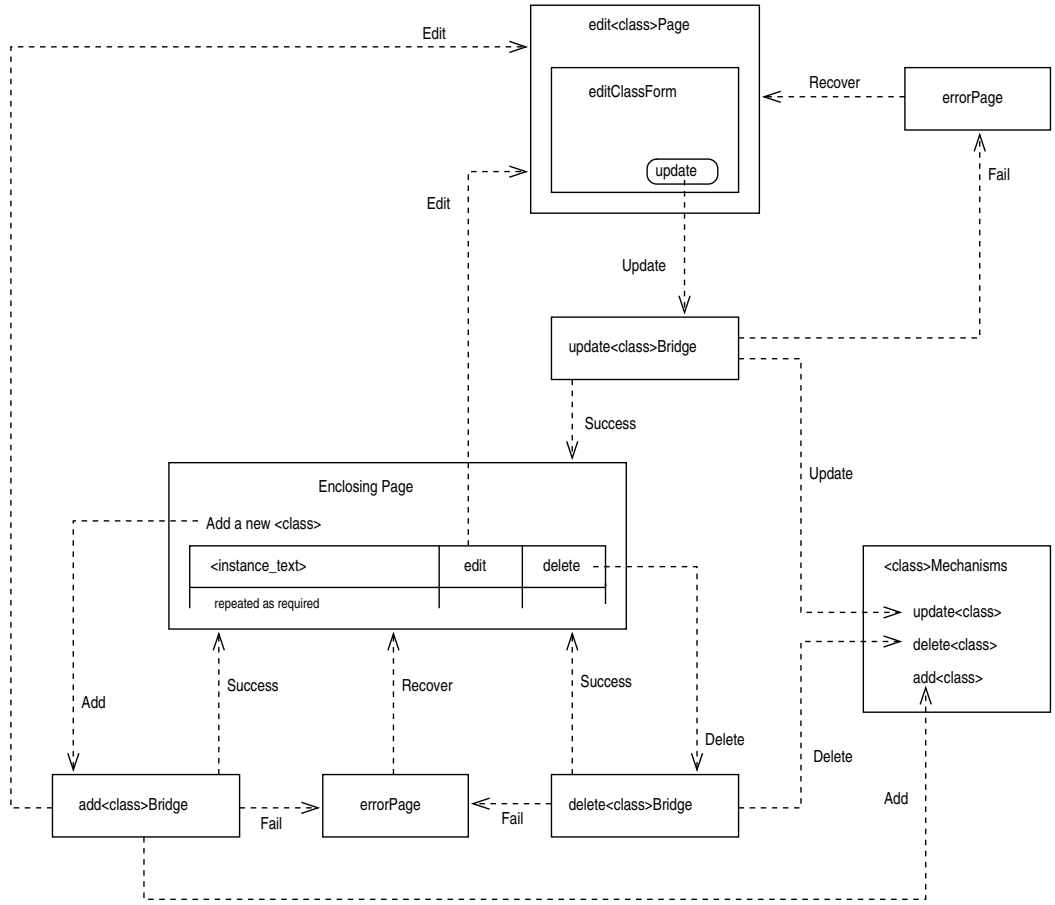

Figure E.2: Block diagram depicting the organisation of PHP modules required to implement Class Lists.

rule (Section E.3.12), is displayed. This page can be used to modify attributes of the instance. When updates are complete, the user will click the '*Update*' button. This will request the '*update<class>Bridge*' page which will invoke the '*update<class>*' PHP function in the '*<class>Mechanisms.php*' package. This function updates a record in the database table that implements *theClass*. If this function executes without error, the page enclosing the *theCrudList* is redisplayed. If, on the other hand, the function raises an exception, the generic *errorPage* (Section D.5.7) will be displayed.

When the '*Add new <class>*' link is clicked the '*add<class>Bridge*' page is requested. This page invokes the '*add<class>*' PHP function defined in '*<class>Mechanisms.php*' package. This function adds a new record in the database table that implements *theClass*. If this function executes without error, the '*edit<class>Page*' is displayed as described above. If, on the other hand, the function raises an exception, the generic *errorPage* will be displayed.

When the '*Delete*' link is clicked the '*delete<class>Bridge*' page is requested. This page invokes the '*delete<class>*' PHP function in the '*<class>Mechanisms.php*' package. This function deletes a record in the database table that implements *theClass*. If this function executes without error, the '*edit<class>Page*' is displayed as described above.If this function executes without error, *theCrudList* is redisplayed. If, on the other hand, the function raises an exception, the generic *errorPage* will be displayed.

Example. See lines 17-36 of the `contactListPage.php` source file (Section F.9.1 on page 274) for an example of code generated using this rule.

Mechanisms. The *irClassList* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiSecureEditDeleteRow** *(Section G.4)*: Used to display text that identifies an instance of a class along with links that can be used to view, edit and delete the instance subject to security permissions.
- <**CLASS-NAME**>**Mechanisms.new**<**CLASS-NAME**> Creates a new instance of *theClass*. See the *irPersistentClass* implementation rule (Section E.4.4).
- <**CLASS-NAME**>**Mechanisms.delete**<**CLASS-NAME**> Deletes an instance of *theClass*. See the *irPersistentClass* implementation rule (Section E.4.4).

Implementation. *Class Lists* comprise a set of HTML table rows that each display information about an instance of *theClass*. The following pattern shall be used to implement a *Class Lists*.

```
1  if (userHasPriv('<PROTECTED-ITEM-NAME>', 'create')) {
2    uiLink
3      ('add<CLASS-NAME>Bridge.php<ID-PARAMETER>',
4       'Add a new <CLASS-NAME>');
5  }
6  $set = mysql_query( "SELECT * FROM <CLASS-NAME>" );
7  if (!$set) {
8    uiFatalErrorPage
9      ("Unable to query <CLASS-NAME> - ".mysql_error());
10 }
```

```
11 uiStartTable(<WIDTH>);
12 $count = mysql_num_rows($set);
13 for ($i=0; $i<$count; $i++) {
14   $instance       = mysql_fetch_array($set);
15   $<CLASS-NAME>Id = $instance['id'];
16   $name           = <NAME-EXPRESSION>;
17   uiSecureEditDeleteRow
18     (<PROTECTED-ITEM-NAME>,
19      $name,
20      "edit<CLASS-NAME>Page.php?id=$<CLASS-NAME>Id",
21      "delete<CLASS-NAME>Bridge.php?id=$<CLASS-NAME>Id");
22 }
23 uiEndTable();
```

Where:

<**PROTECTED-ITEM-NAME**> *theProtectedItem.name* attribute value.

<**CLASS-NAME**> The name of *theClass*.

<**ID-PARAMETER**> If new instances of *theClass* must be linked to an instance of some other class, then this tag shall be replaced with the following code.

```
1 ?<ASSOCIATED-CLASS-NAME>Id = <ID-EXPRESSION>
```

Where:

<**ASSOCIATED-CLASS-NAME**> The name of the class with which objects on the list must be associated.

<**ID-EXPRESSION**> An expression that returns the ID of an instance of the associated class to which a new instance of the listed class must be linked. For example, see lines 50-54 of the `editProductPage.php` source file (Section F.7.4 on page 258).

<**WIDTH**> *theCrudList.listWidth* attribute value.

<**NAME-EXPRESSION**> An expression that returns the text to be displayed for a given instance in the list. This is often an attribute read of the following form:

```
1 $name = $instance['productName'];
```

Edit Page. An *Edit Page* and associated *Update Bridge* shall be implemented for *theClass* using the *irEditClassPage* implementation rule (Section E.3.12).

Add instance bridge. A bridge to the `new<CLASS-NAME>` function declared in *theClass* mechanisms file described by the *raPersistentClass* implementation rule (Section C.4.2) shall be implemented in a file named as follows.

```
1 add<CLASS-NAME>Bridge.php
```

If instances of *theClass* do not have to be linked with instances of some other class, then the bridge file shall contain the following PHP code. For example, see the `addContactBridge.php` source file (Section F.9.2 on page 275).

```
1  <?php
2
3  require_once('<CLASS-NAME>Mechanisms.php');
4  require_once('userInterfaceMechanisms.php');
5  require_once('databaseMechanisms.php');
6
7  session_start();
8
9  try {
10
11   $id = new<CLASS-NAME>();
12   uiShowUrl("edit<CLASS-NAME>Page.php?id=$id");
13
14 } catch (Exception $e) {
15   uiErrorPage($e);
16 }
17
18 ?>
```

Where:

<**CLASS-NAME**> The name of *theClass*.

If instances of *theClass* must be linked with instances of some other class, then the bridge file shall contain the following PHP code. For example, see the `addProductVariantBridge.php` source file (Section F.7.6 on page 261).

```
<?php

require_once('<CLASS-NAME>Mechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

$<ASSOCIATED-CLASS-NAME>Id
  = $HTTP_GET_VARS['<ASSOCIATED-CLASS-NAME>Id'];

try {

  $id = new<CLASS-NAME>($<ASSOCIATED-CLASS-NAME>Id);
  uiShowUrl("edit<CLASS-NAME>Page.php?id=$id");

} catch (Exception $e) {
  uiErrorPage($e);
}

?>
```

Where:

<**CLASS-NAME**> The name of *theClass*.

<**ASSOCIATED-CLASS-NAME**> The name of the class with which the above class is associated.

Delete instance bridge. A bridge to the `delete<CLASS-NAME>` function declared in *theClass* mechanisms file described by the *raPersistentClass* implementation rule (Section C.4.2) shall be implemented in a file named as follows.

```
delete<CLASS-NAME>Bridge.php
```

The bridge file shall contain the following PHP code. For example, see the `deleteProductBridge.php` source file (Section F.7.3 on page 257).

```
<?php

require_once('<CLASS-NAME>Mechanisms.php');
```

```
4  require_once('userInterfaceMechanisms.php');
5  require_once('databaseMechanisms.php');
6
7  session_start();
8
9  $id = $HTTP_GET_VARS['id'];
10
11 try {
12
13   delete<CLASS-NAME>($id);
14   uiShowTopUrl();
15
16 } catch (Exception $e) {
17   uiErrorPage($e);
18 }
19
20 ?>
```

Where:

<**CLASS-NAME**> The name of *theClass*.

E.3.11 Implementation Rule 'irGroupedClassList'

Implemented Requirements. The *irGroupedClassList* implementation rule implements a set of requirements that conform to the following related *Requirement Archetypes*.

- *raClassList* requirement archetype (Section C.2.6)
  - *theCrudList* : UI.CrudList
  - *theClass* : xtUML.Class
  - *theEditPage* : UI.Page
- *raGroupClassList* requirement archetype (Section C.2.7)
  - *theCrudList* : UI.CrudList
  - *theAssociation* : xtUML.Association
  - *theGroupingAttribute* : xtUML.Attribute
- *raSecureClass* requirement archetype (Section C.3.2)
  - *theClass* : xtUML.Class
  - *theProtectedItem* : SEC.ProtectedItem

Constraints. The *irGroupedClassList* implementation rule shall be used to implement an *raClassList* (Section C.2.6) requirement that references *theClass* that is also referenced by an *raSecureClass* (Section C.3.2) requirement. In addition, *theCrudList* referenced by the *raClassList* (Section C.2.6) requirement must also be referenced by an *raGroupClassList* (Section C.2.7) requirement.

Description. The *irGroupedClassList* implementation rule is a variation of the *irClassList* implementation rule (Section E.3.10) described above in which instances of a class are displayed in groups as described in the *raGroupClassList* requirement archetype (Section C.2.7).

Example. See lines 17-58 of the `productListPage.php` source file (Section F.7.1 on page 255) for an example of code generated using this rule.

Mechanisms. See the *irClassList* implementation rule (Section E.3.10).

Implementation. The following pattern shall be used to implement grouped class lists.

```
1  if (userHasPriv('<PROTECTED-ITEM-NAME>', 'create')) {
2    uiLink
3      ('add<CLASS-NAME>Bridge.php<ID-PARAMETER>',
4       'Add a new <CLASS-NAME>');
5  }
6
7  $groupSet = mysql_query( "SELECT * FROM <GROUPING-CLASS-NAME>" );
8  if (!$groupSet) {
9    uiFatalErrorPage
10     ("Unable to query <GROUPING-CLASS-NAME> - ".mysql_error());
11 }
12 $groupCount = mysql_num_rows($groupSet);
13 for ($i=0; $i<$groupCount; $i++) {
14   $groupInstance  = mysql_fetch_array($groupSet);
15   $groupId        = $groupInstance['id'];
16
17   $groupName = <GROUP-NAME-EXPRESSION>;
18   uiSubHeading($groupName);
19
20   <OTHER-GROUP-TEXT>
21
22   $set = mysql_query
```

```
23            ( "SELECT * FROM <CLASS-NAME>
24               WHERE <GROUPING-CLASS-NAME>Id='$groupId'" );
25    if (!$set) {
26      uiFatalErrorPage
27        ("Unable to query <CLASS-NAME> - ".mysql_error());
28    }
29
30    uiStartTable(<WIDTH>);
31    $count = mysql_num_rows($set);
32    for ($i=0; $i<$count; $i++) {
33      $instance       = mysql_fetch_array($set);
34      $<CLASS-NAME>Id = $instance['id'];
35      $name           = <NAME-EXPRESSION>;
36      uiSecureEditDeleteRow
37        (<PROTECTED-ITEM-NAME>,
38         $name,
39         "edit<CLASS-NAME>Page.php?id=$<CLASS-NAME>Id",
40         "delete<CLASS-NAME>Bridge.php?id=$<CLASS-NAME>Id");
41    }
42    uiEndTable();
43  }
```

Where:

<**PROTECTED-ITEM-NAME**> *theProtectedItem.name* attribute value.

<**ID-PARAMETER**> If new instances of *theClass* must be linked to an instance of some other class, then this tag shall be replaced with the following code.

```
1  ?<ASSOCIATED-CLASS-NAME>Id = <ID-EXPRESSION>
```

Where:

<**ASSOCIATED-CLASS-NAME**> The name of the class with which objects on the list must be associated.

<**ID-EXPRESSION**> An expression that returns the ID of an instance of the associated class to which a new instance of the listed class must be linked.

<**CLASS-NAME**> The name of *theClass*.

<**GROUPING-CLASS-NAME**> The name of the class to which *theGroupingAttribute* belongs.

<**GROUP-NAME-EXPRESSION**> An expression that returns the text to be displayed as a heading for each group. It normally refers to an attribute of the above class. For example, see line 31 of the `productListPage.php` source file (Section F.7.1 on page 255).

<**OTHER-GROUP-TEXT**> A sequence of statements that display additional information about each group. The following code for example can be used to display the value of a grouping class attribute.

```
1 $notes = $instance['notes'];
2 uiText("Note: <i>".$notes."</i>");
```

<**WIDTH**> *theCrudList.listWidth* attribute value.

<**NAME-EXPRESSION**> An expression that returns the text to be displayed for a given instance in the list. This is often an attribute read of the following form:

```
1 $name = $instance['productName'];
```

Edit page. An *Edit Page* and associated *Update Bridge* shall be implemented for *theClass* using the *irEditClassPage* implementation rule (Section E.3.12).

Add and Delete Bridges. Add and Delete bridges shall be implemented as detailed in the *irClassList* implementation rule (Section E.3.10).

E.3.12 Implementation Rule 'irEditClassPage'

Implemented Requirements. The *irEditClassPage* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raClassList* requirement archetype (Section C.2.6)
  - *theCrudList* : UI.CrudList
  - *theClass* : xtUML.Class
  - *theEditPage* : UI.Page

Description. The *irEditClassPage* implementation rule is used to implement *theEditPage* that is used to edit objects listed in *theCrudList*.

Example. See the `editProductPage.php` source file (Section F.7.4 on page 258) for an example of code generated using this rule.

Page source file. *theEditPage* shall be implemented in a PHP file named as follows and stored in the *Application Directory*.

```
edit<CLASS-NAME>Page.php
```

Where:

<**CLASS-NAME**> The name of *theClass*.

The file shall have the following contents:

```
<?php

require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');

session_start();

$id = $HTTP_GET_VARS['id'];

showEdit<CLASS-NAME>Page($id);

function showEdit<CLASS-NAME>Page($id) {
  connectToDatabase();
  uiStartPage('Edit <FORMATTED-CLASS-NAME>');

  uiSetCurrentUrl
    ("<FORMATTED-CLASS-NAME>",
     "edit<CLASS-NAME>Page.php?id=$id");
  uiLastUrlLink();

  uiText
    ("Modify <CLASS-NAME> fields and click "
        ."'Update <CLASS-NAME>' to save:");

  $set = mysql_query("SELECT * FROM <CLASS-NAME> WHERE id=$id");
  if (!$set) {
    uiFatalErrorPage
      ("Unable to query <CLASS-NAME>s - ".mysql_error());
```

```
30    }
31    $the<CLASS-NAME>   = mysql_fetch_array($set);
32    <GET-ATTRIBUTE-VALUES>
33
34    <UI-PAGE-ITEMS>
35    <EDIT-CLASS-FORM>
36    <UI-PAGE-ITEMS>
37    uiEndPage();
38  }
39
40  ?>
```

Where:

<**CLASS-NAME**> The name of *theClass*.

<**FORMATTED-CLASS-NAME**> The name of *theClass* formatted to include appropriate capitalisation and spaces (e.g., 'productCategory' could be formatted as 'Product Category').

<**GET-ATTRIBUTE-VALUES**> The following PHP code shall be implemented for each *Attribute* of *theClass*:

```
1  $<ATTRIBUTE-NAME> = $the<CLASS-NAME>['<ATTRIBUTE-NAME>'];
```

Where:

<**ATTRIBUTE-NAME**> The name of an *Attribute* of *theClass*.

<**UI-PAGE-ITEMS**> Implementation of the following *UI.PageItems* as applicable and described in the identified *Implementation Rules*.

- Main Heading
  - *irStaticPageItems* implementation rule (Section E.3.5)
- Sub-heading
  - *irStaticPageItems* implementation rule (Section E.3.5)
- Simple Text
  - *irStaticPageItems* implementation rule (Section E.3.5)
- Link
  - *irPageLink* implementation rule (Section E.3.7)
  - *irRolePageLink* implementation rule (Section E.3.8)

<**EDIT-CLASS-FORM**> This is the implementation of a *UI.Form* used to enter and modify the *Attributes* of *theClass*. This *UI.Form* shall be implemented using the *irEditClassForm* implementation rule (Section E.3.13)

E.3.13 Implementation Rule 'irEditClassForm'

Implemented Requirements. The *irEditClassForm* implementation rule implements a set of requirements that conform to the following related *Requirement Archetypes*.

- *raClassList* requirement archetype (Section C.2.6)
  - *theCrudList* : UI.CrudList
  - *theClass* : xtUML.Class
  - *theEditPage* : UI.Page
- *raSecureClass* requirement archetype (Section C.3.2)
  - *theClass* : xtUML.Class
  - *theProtectedItem* : SEC.ProtectedItem

Description. The *irEditClassForm* implementation rule is used to implement *UI.Forms* that appear on *theEditPage*.

Example. See lines 32-48 of the `editProductPage.php` source file (Section F.7.4 on page 258) for an example of code generated using this rule.

Form structure. The following PHP code shall be used to implement the *UI.Form* contained on *theEditPage*.

```
1  uiStartForm('update<CLASS-NAME>Bridge.php');
2  uiStartTable(300);
3  uiHiddenField('id', $id);
4  <HIDDEN-LINKED-OBJECT-ID>
5  <FORM-ITEMS>
6  uiEndTable();
7  uiSecureSubmitButton
8    ('<PROTECTED-ITEM-NAME>', 'Update <CLASS-NAME>');
9  uiEndForm();
```

Where:

<**CLASS-NAME**> The name of *theClass*.

<**HIDDEN-LINKED-OBJECT-ID**> If instances of *theClass* must be linked with instances of another class, then this tag shall be replaced with the following code. For example, see line 40 of the `editProductVariantPage.php` source file (Section F.7.8 on page 262).

```
1 uiHiddenField
2   ('<ASSOCIATED-CLASS-NAME>Id', $<ASSOCIATED-CLASS-NAME>Id);
```

Where:

<**ASSOCIATED-CLASS-NAME**> The name of the associated class.

<**TABLE-WIDTH**> *UI.FormItems* are arranged using an HTML table. This parameter specifies the width of this table and hence the width of the form.

<**FORM-ITEMS**> Implementation of the following *UI.FormItems* as applicable and described in the identified *Implementation Rules*.

- Attribute Text Input
  - *irAttributeTextInput* implementation rule (Section E.3.14)
- Attribute Password Input
  - *irAttributePasswordInput* implementation rule (Section E.3.15)
- Attribute Text Area
  - *irAttributeTextArea* implementation rule (Section E.3.16)
- Attribute Select
  - *irLinkObjectSelect* implementation rule (Section E.3.17)

<**PROTECTED-ITEM-NAME**> *theProtectedItem.name* attribute value.

Update instance bridge. A bridge to the `update<CLASS-NAME>` function declared in *theClass* mechanisms file described by the *raPersistentClass* implementation rule (Section C.4.2) shall be implemented in a file named as follows.

```
1 update<CLASS-NAME>Bridge.php
```

The bridge file shall contain the following PHP code. For example, see the `updateProductBridge.php` source file (Section F.7.5 on page 260).

```
<?php

require_once('<CLASS-NAME>Mechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

$id = $HTTP_POST_VARS['id'];
<GET-ATTRIBUTE-VALUES>

try {

  update<CLASS-NAME>($id, <UPDATE-PARAMETER-LIST>);
  uiShowTopUrl();

} catch (Exception $e) {
  uiErrorPage($e);
}

?>
```

Where:

<**CLASS-NAME**> The name of *theClass*.

<**GET-ATTRIBUTE-VALUES**> For each *Attribute* of *theClass*, the following code shall be inserted into the above file.

```
$<PARAMETER-NAME>
  = $HTTP_POST_VARS['<PARAMETER-NAME>'];
```

Where:

<**PARAMETER-NAME**> The *name* of the *UI.FormItem* to retrieve from the form.

<**UPDATE-PARAMETER-LIST**> A comma delimited list of the parameter names extracted above.

E.3.14 Implementation Rule 'irAttributeTextInput'

Implemented Requirements. The *irAttributeTextInput* implementation rule implements a set of requirements that conform to the following related *Require-*

ment Archetypes.

- *raTextInputUpdatesAttribute* requirement archetype (Section C.2.8)
  - *theTextInput* : UI.TextInput
  - *theAttribute* : xtUML.Attribute
- *raSecureClass* requirement archetype (Section C.3.2)
  - *theClass* : xtUML.Class
  - *theProtectedItem* : SEC.ProtectedItem

Constraints. *theAttribute* must be an attribute of *theClass*.

Example. See line 36 of the `editUserPage.php` source file (Section F.10.4 on page 295) for an example of code generated using this rule.

Description. This *irAttributeTextInput* implementation rule is used to implement *UI.TextInput* items that appear on*UI.Forms* implemented using the *irEditClassForm* implementation rule (Section E.3.13).

Mechanisms. The *irAttributeTextInput* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiSecureTextInput** *(Section G.4)*: Displays a *UI.TextInput* item that allows the user, subject to security permissions, to view and edit the value of a specified class attribute.

Instructions. *UI.TextInput* items shall be implemented using the following PHP code.

```
1 uiSecureTextInput
2   (<PROTECTED-ITEM-NAME>, <LABEL>,
3    <ATTRIBUTE-NAME>, $<ATTRIBUTE-NAME>, <WIDTH>);
```

Where:

<**PROTECTED-ITEM-NAME**> *theProtectedItem.name* attribute value.

<**LABEL**> The name of *theAttribute* formatted to include appropriate capitalisation and spaces (e.g., 'streetAddress' could be formatted as 'Street Address').

<**ATTRIBUTE-NAME**> The name of *theAttribute*.

<**WIDTH**> The *theTextInput.textWidth* attribute value.

E.3.15 Implementation Rule 'irAttributePasswordInput'

Implemented Requirements. The *irAttributePasswordInput* implementation rule implements a set of requirements that conform to the following related *Requirement Archetypes*.

- *raPasswordInputUpdatesAttribute* requirement archetype (Section C.2.9)
  - *thePasswordInput* : UI.PasswordInput
  - *theAttribute* : xtUML.Attribute
- *raSecureClass* requirement archetype (Section C.3.2)
  - *theClass* : xtUML.Class
  - *theProtectedItem* : SEC.ProtectedItem

Constraints. *theAttribute* must be an attribute of *theClass*.

Example. See line 37 of the `editUserPage.php` source file (Section F.10.4 on page 295) for an example of code generated using this rule.

Description. This *irAttributePasswordInput* implementation rule is used to implement *UI.PasswordInput* items that appear on*UI.Forms* implemented using the *irEditClassForm* implementation rule (Section E.3.13).

Mechanisms. The *irAttributePasswordInput* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiSecurePasswordInput** *(Section G.4)*: Displays a *UI.PasswordInput* item that allows the user, subject to security permissions, to edit the value of a specified class attribute.

Instructions. *UI.PasswordInput* items shall be implemented using the following PHP code.

```
uiSecurePasswordInput
  (<PROTECTED-ITEM-NAME>, <LABEL>,
   <ATTRIBUTE-NAME>, <WIDTH>);
```

Where:

<**PROTECTED-ITEM-NAME**> *theProtectedItem.name* attribute value.

<**LABEL**> The name of *theAttribute* formatted to include appropriate capitalisation and spaces (e.g., 'streetAddress' could be formatted as 'Street Address').

<**ATTRIBUTE-NAME**> The name of *theAttribute*.

<**WIDTH**> The *thePasswordInput.textWidth* attribute value.

E.3.16 Implementation Rule 'irAttributeTextArea'

Implemented Requirements. The *irAttributeTextArea* implementation rule implements a set of requirements that conform to the following related *Requirement Archetypes*.

- *raTextAreaUpdatesAttribute* requirement archetype (Section C.2.10)
  - *theTextArea* : UI.TextArea
  - *theAttribute* : xtUML.Attribute
- *raSecureClass* requirement archetype (Section C.3.2)
  - *theClass* : xtUML.Class
  - *theProtectedItem* : SEC.ProtectedItem

Constraints. *theAttribute* must be an attribute of *theClass*.

Example. See line 37 of the `editNotePage.php` source file (Section F.9.12 on page 286) for an example of code generated using this rule.

Description. This *irAttributeTextArea* implementation rule is used to implement *UI.TextArea* items that appear on *UI.Forms* implemented using the *irEditClassForm* implementation rule (Section E.3.13).

Mechanisms. The *irAttributeTextArea* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiSecureTextArea** *(Section G.4)*: Displays a *UI.TextArea* item that allows the user, subject to security permissions, to view and edit the value of a specified class attribute.

Instructions. *UI.TextArea* items shall be implemented using the following PHP code.

```
1 uiSecureTextArea
2   (<PROTECTED-ITEM-NAME>, <LABEL>,
3    <ATTRIBUTE-NAME>, $<ATTRIBUTE-NAME>, <WIDTH>, <HEIGHT>);
```

Where:

<**PROTECTED-ITEM-NAME**> *theProtectedItem.name* attribute value.

<**LABEL**> The name of *theAttribute* formatted to include appropriate capitalisation and spaces (e.g., 'streetAddress' could be formatted as 'Street Address').

<**ATTRIBUTE-NAME**> The name of *theAttribute*.

<**WIDTH**> The *theTextArea.textWidth* attribute value.

<**HEIGHT**> The *theTextArea.textHeight* attribute value.

E.3.17 Implementation Rule 'irLinkObjectSelect'

Implemented Requirements. The *irLinkObjectSelect* implementation rule implements a set of requirements that conform to the following related *Requirement Archetypes*.

- *raSelectItemCreatesLink* requirement archetype (Section C.2.11)
  - *theSelectItem* : UI.Select

  - *theAssociation* : xtUML.Association
  - *theAttribute* : xtUML.Attribute

- *raSecureClass* requirement archetype (Section C.3.2)
  - *theClass* : xtUML.Class
  - *theProtectedItem* : SEC.ProtectedItem

Constraints. *theClass* must be the same as the class of which *theAttribute* is a member.

Example. See lines 40-42 of the `editProductPage.php` source file (Section F.7.4 on page 258) for an example of code generated using this rule.

Description. This *irLinkObjectSelect* implementation rule is used to implement *UI.Select* items that appear on*UI.Forms* implemented using the *irEditClassForm* implementation rule (Section E.3.13).

Mechanisms. The *irLinkObjectSelect* implementation rule relies on the following mechanisms.

- **User Interface Mechanisms.uiSecureSelect** *(Section G.4)*: Displays a *UI.Select* item that allows the user, subject to security permissions, to view and select an instance of a specified $\frac{X}{T}$UML *Class*.

Instructions. *UI.Select* items shall be implemented using the following PHP code.

```
1 uiSecureSelect
2   (<PROTECTED-ITEM-NAME>, <LABEL>,
3    '<SELECT-CLASS-NAME>Id',
4    '<SELECT-CLASS-NAME>', '<SELECT-ATTRIBUTE-NAME>',
5    $<CLASS-NAME>Id);
```

Where:

<**PROTECTED-ITEM-NAME**> *theProtectedItem.name* attribute value.

<**LABEL**> The name of *theAttribute* formatted to include appropriate capitalisation and spaces (e.g., 'streetAddress' could be formatted as 'Street Address').

<**SELECT-CLASS-NAME**> The name of the *theClass* to which *theAttribute* belongs.

<**SELECT-ATTRIBUTE-NAME**> The name of *theAttribute*.

E.4 Database implementation rules

E.4.1 *Database Mechanisms* domain model

The *Database Mechanisms* domain model comprises a collection of *PHP* functions that implement various database related functions. This *Domain Model* is an instance of the *PHP Web Programming Language* domain model and comprises the PHP source code listed in Section G.3 of Appendix G.

E.4.2 Implementation Rule 'irDatabaseConfig'

Implemented Requirements. The *irDatabaseConfig* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raMysqlImplementation* requirement archetype (Section C.4.1)
  - *theSqlDomain* = SQL.
  - *theMysqlDomain* = MySQL.

Description. The *irDatabaseConfig* implementation rule is used to implement a database configuration file for use with the application.

Example. See the `databaseConfiguration.php` source file (Section F.2.2 on page 245) for an example of code generated using this rule.

Database configuration file. A PHP file, named as follows, shall be created and stored in the *Application Directory*.

```
1 databaseConfiguration.php
```

The file shall have the following contents:

```
<?php

session_start();

define("DATABASE_HOST",     "<HOST>");
define("DATABASE_NAME",     "<DATABASE-NAME");
define("DATABASE_USERNAME", "<DATABASE-USERNAME>");
define("DATABASE_PASSWORD", "<DATABASE-PASSWORD>");

?>
```

Where:

<**HOST**> The domain name or IP address of a server running the MySQL database.

<**DATABASE-NAME**> The name of the application, formatted to be compatible with SQL naming conventions.

<**DATABASE-USERNAME**> The username of the account used to create the application database.

<**DATABASE-PASSWORD**> The password for the above account.

E.4.3 Implementation Rule 'irDatabaseScripts'

Implemented Requirements. The *irDatabaseScripts* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raMysqlImplementation* requirement archetype (Section C.4.1)
  - *theSqlDomain* = SQL.
  - *theMysqlDomain* = MySQL.

Description. The *irDatabaseScripts* implementation rule is used to create the scripts necessary to create and manage the database tables required to implement $^{X}_{T}$UML *Class* and *Association* persistence.

Note that the specific contents of these scripts are created in accordance with later *Implementation Rules*.

Example. See the `pmsCreateTables.sql` (Section F.3.1 on page 247), `pmsDropDatabase.sql` (Section F.3.2) and `pmsPopulateTables.sql` (Section F.3.3) source files for an example of code generated using this rule.

Table creation script. This purpose of this script is to create all of the MySQL database tables required by an application. The *Table Creation* scripts shall be named as follows:

```
1 <APPLICATION-NAME>CreateTables.sql
```

Where:

<**APPLICATION-NAME**> The name of the application being developed.

The *Table Creation* script shall have the following contents.

```
1 create database <DATABASE-NAME>;
2 use <DATABASE-NAME>;
3
4 <TABLE-CREATE-STATEMENTS>
```

Where:

<**DATABASE-NAME**> The name of the database specified in the `databaseConfiguration.php` file generated using the *irDatabaseConfig* implementation rule (Section E.4.2).

<**TABLE-CREATE-STATEMENTS**> A set of SQL statements that create the tables required by the application. The generation of these statements is the subject of the *irPersistentClass* implementation rule (Section E.4.4).

Running the table creation script. The *Table Creation* script can be run using the following command line. This command will ask for the MySQL root password. It will then create the required tables. Note that errors will occur if the tables already exist.

```
1 $ mysql -u root -p < <APPLICATION-NAME>CreateTables.sql;
2 $
```

Drop Database script. This purpose of this script is to delete an application's database. The *Drop Database* script shall be named as follows:

```
1 <APPLICATION-NAME>DropDatabase.sql
```

Where:

<**APPLICATION-NAME**> The name of the application being developed.

The *Drop Database* script shall have the following contents.

```
1 drop database <DATABASE-NAME>;
```

Where:

<**DATABASE-NAME**> The name of the database specified in the `databaseConfiguration.php` file generated using the *irDatabaseConfig* implementation rule (Section E.4.2).

Running the drop database script. The *Drop database* script can be run using the following command line. This command will ask for the MySQL root password. It will then delete the database.

```
1 $ mysql -u root -p < <APPLICATION-NAME>DropDatabase.sql;
2 $
```

Table population script. It is sometimes necessary to populate parts of a database before an application can be run correctly. If such population is required, a script with the following name shall be created.

```
1 <APPLICATION-NAME>PopulateTables.sql
```

Where:

<**APPLICATION-NAME**> The name of the application being developed.

The *Table Population* script shall have the following contents.

```
use <DATABASE-NAME>;

<DATA-INSERT_STATEMENTS>
```

Where:

<**DATABASE-NAME**> The name of the database specified in the `databaseConfiguration.php` file generated using the *irDatabaseConfig* implementation rule (Section E.4.2).

<**DATA-INSERT-STATEMENTS**> A set of SQL statements that insert rows of data into tables. The generation of these statements is the subject of the *irPersistentClass* implementation rule (Section E.4.4).

Running the table population script. The *Table Population* script can be run using the following command line. This command will ask for the MySQL root password. It will then populate the applicable tables. Note that errors will occur of the tables do not exist.

```
$ mysql -u root -p < <APPLICATION-NAME>PopulateTables.sql;
$
```

E.4.4 Implementation Rule 'irPersistentClass'

Implemented Requirements. The *irPersistentClass* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raPersistentClass* requirement archetype (Section C.4.2)
  - *theClass* : xtUML.Class
  - *theSqlDomain* = SQL.

Description. The *irPersistentClass* implementation rule is used to implement $\frac{X}{T}$UML *Classes* that are to be made persistent using the *Structured Query Language* domain model (Section B.10).

Table creation. *theClass* shall be implemented as a MySQL database table. The name of the table shall be the same as *theClass* name. Class attributes shall be mapped to table columns.

The following SQL statements shall be added to the *Table Creation Script* described in the *irDatabaseScripts* implementation rule (Section E.4.3). For example, see lines 4-52 of the `pmsCreateTables.sql` source file (Section F.3.1 on page 247).

```
1 create table <TABLE-NAME>
2   (id int unsigned not null auto_increment primary key,
3    <COLUMN-SPECS>);
4
5 describe <TABLE-NAME>;
```

Where:

<**TABLE-NAME**> The table name which is the same as the *theClass* name.

<**COLUMN-SPECS**> A comma delimited list of table column specifications. A column specification shall be included for each attribute of *theClass*. The form of a column specification shall be as follows:

```
1 <ATTRIBUTE-NAME> <DATA-TYPE>
```

Where:

<**ATTRIBUTE-NAME**> The database column name shall be the same as *theClass* attribute name.

<**DATA-TYPE**> The data type of each column shall be determined using the *Data Types* rule described below.

Data types. A mapping shall be established between *Domain Specific Types* defined in $\frac{X}{T}$UML domains and MySQL data types. This mapping will be used when generating table creation statements as described above.

Associative classes. If *theClass* is an associative class between two classes (*class 1* and *Class 2*), then the following additional column specifications shall be included in the <COLUMN-SPECS> substitution described above. For example, see lines 22-25 of the `pmsCreateTables.sql` source file (Section F.3.1 on page 247).

```
1 <VERB-PHRASE-1><CLASS-1-NAME>Id int,
2 <VERB-PHRASE-2><CLASS-2-NAME>Id int
```

Where:

<**VERB-PHRASE-1**> The verb phrase at the *Class 1* end of the association.

<**CLASS-1-NAME**> The name of *Class 1*

<**VERB-PHRASE-1**> The verb phrase at the *Class 2* end of the association.

<**CLASS-2-NAME**> The name of *Class 2*.

Associations with persistent classes. If *theClass* is associated with other classes with a multiplicity of 0..1 or 1..1, then the following column definition shall be included in class table for each associated class. For example, see line 35 of the `pmsCreateTables.sql` source file (Section F.3.1 on page 247).

```
1 <CLASS-NAME>Id int
```

Where:

<**CLASS-NAME**> The name of the associated class.

Instance population. In some applications it may be necessary for certain instances of persistent classes to exist before an application starts.

If required, the following SQL statements shall be added to the *Table Population Script* described in the *irDatabaseScripts* implementation rule (Section E.4.3). For example, see lines 3-17 of the `pmsPopulateTables.sql` source file (Section F.3.3 on page 248).

```
1 insert into <TABLE-NAME> values
2   (NULL, <COLUMN-VALUES>),
3   (NULL, <COLUMN-VALUES>),
4   ...
5   (NULL, <COLUMN-VALUES>);
```

Where:

<**TABLE-NAME**> The table name is the same as *theClass* name.

<**COLUMN-VALUES**> A comma delimited list of table column values. Note that, if the column value is to be encrypted (e.g., the column is a password), then the column value will take the following form

```
password([COLUMN-VALUE])
```

Example:

```
insert into contacts values
  (NULL, 'Simpson, Bart',
   '742 Evergreen Terrace, Springfield, USA',
   '61234567', '0410123456', 'bart@springfield.net'),
  (NULL, 'Flanders, Ned',
   '740 Evergreen Terrace, Springfield, USA',
   '61234568', '0410123457', 'ned@springfield.net'));
```

Generation of mechanisms. For each persistent *Class*, there shall be a set of mechanisms declared in a file named as follows.

```
<CLASS-NAME>Mechanisms.php
```

The mechanisms file shall contain the following PHP code. For example, see the `productMechanisms.php` source file (Section F.7.10 on page 265).

```
<?php
session_start();

require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');
<ASSOCIATED-CLASS-MECHANISMS>

connectToDatabase();


function new<CLASS-NAME>(<ASSOCIATED-OBJECT-ID-PARAMETER>) {

  $result = mysql_query("INSERT INTO <CLASS-NAME> SET
              <NEW-ATTRIBUTE-ASSIGNMENTS>");

  if (!$result) {
```

```
17      throw new Exception
18                  ("Unable to add <CLASS-NAME> - ".mysql_error());
19    }
20
21    $id = mysql_insert_id();
22
23    <SET-ASSOCIATED-OBJECT-ID>
24
25    return $id;
26  }
27
28
29  function update<CLASS-NAME>($id, <UPDATE-PARAMETER-LIST>) {
30    connectToDatabase();
31
32    $result = mysql_query("UPDATE <CLASS-NAME> SET
33                <UPDATE-ATTRIBUTE-ASSIGNMENTS>
34                WHERE id=$id");
35    if (!$result) {
36      throw new Exception
37                  ("Unable to update <CLASS-NAME> - ".mysql_error());
38    }
39  }
40
41
42  function delete<CLASS-NAME>($id) {
43    connectToDatabase();
44
45    $result = mysql_query("DELETE FROM <CLASS-NAME> WHERE id=$id");
46    if (!$result) {
47      throw new Exception
48                  ("Unable to delete <CLASS-NAME> - ".mysql_error());
49    }
50    <DELETE-LINKED-INSTANCES>
51  }
52
53  ?>
```

Where:

<**ASSOCIATED-CLASS-MECHANISMS**> If instances of *theClass* must be linked with an instance of some other class, then this tag shall be replaced with the following PHP code. For example, see line 6 of the `productMechanisms.php` source file (Section F.7.10 on page 265).

```
1 require_once('<ASSOCIATED-CLASS-NAME>Mechanisms.php');
```

Where:

<**ASSOCIATED-CLASS-NAME**> The name of the *Class* with which new instances must be associated.

<**CLASS-NAME**> The name of *theClass*.

<**ASSOCIATED-OBJECT-ID-PARAMETER**> If new instances of *theClass* must be linked to an instance of some other class, then the following parameter shall be added to the `new<CLASS-NAME>()` method. For example, see line 11 of the `productVariantMechanisms.php` source file (Section F.7.11 on page 266).

```
1 $<ASSOCIATED-CLASS-NAME>Id
```

<**NEW-ATTRIBUTE-ASSIGNMENTS**> A comma delimited set of attribute assignments of the following form:

```
1 <ATTRIBUTE-NAME> = <DEFAULT-VALUE>
```

Where:

<**ATTRIBUTE-NAME**> The name of an attribute of *theClass*.

<**DEFAULT-VALUE**> An appropriate default value for the above attribute.

<**SET-ASSOCIATED-OBJECT-ID**> If new instances must be linked to an instance of some other class, then the following code shall be included to form the appropriate link. For example, see lines 25-32 of the `productVariantMechanisms.php` source file (Section F.7.11 on page 266).

```
1  $result = mysql_query
2    ("UPDATE <CLASS-NAME> SET
3         <ASSOCIATED-CLASS-NAME>Id='$<ASSOCIATED-CLASS-NAME>Id'
4         WHERE id=$id");
5
6  if (!$result) {
7    throw new Exception
8     ("Unable to link <ASSOCIATED-CLASS-NAME> to <CLASS-NAME> - "
9      .mysql_error());
10 }
```

<**UPDATE-PARAMETER-LIST**> A comma delimited list of names of attributes of *theClass*. The order shall be the same as they are declared in the $\frac{X}{T}$UML *Class* definition. The form shall be as follows. For example, see line 26 of the `productMechanisms.php` source file (Section F.7.10 on page 265).

```
1 $<ATTRIBUTE-NAME>, $ATTRIBUTE-NAME>, ...
```

<**UPDATE-ATTRIBUTE-ASSIGNMENTS**> A comma delimited set of attribute assignments of the following form. There shall be an assignment corresponding to each parameter declared using the above rule.

```
1 <ATTRIBUTE-NAME> = <ATTRIBUTE-VALUE>
```

Where:

<**ATTRIBUTE-NAME**> The name of an attribute of *theClass*.

<**ATTRIBUTE-VALUE**> The new value of the attribute. For normal attributes, this will take the form `'$<ATTRIBUTE-NAME>'`. If the attribute is to be encrypted, this this field will take the form `password('$<ATTRIBUTE-NAME>')`.

<**DELETE-LINKED-INSTANCES**> If each instance of *theClass*is linked with 0..* instances of another class, then the following code shall be used to delete those instances. This implements the rules of $\frac{X}{T}$UML documents in *Mellor and Balcer* [2002]. For example, see lines 51-62 of the `productMechanisms.php` source file (Section F.7.10 on page 265).

```
1  $set = mysql_query("SELECT * FROM <LINKED-CLASS-NAME>
2                      WHERE <CLASS-NAME>Id=$id");
3  if (!$set) {
4    throw new Exception(
5      "Unable to select <LINKED-CLASS-NAME> - ".mysql_error());
6  }
7  $count = mysql_num_rows($set);
8  for ($i=0; $i<$count; $i++) {
9    $instance   = mysql_fetch_array($set);
10   $<LINKED-CLASS-NAME>Id  = $instance['id'];
11   delete<LINKED-CLASS-NAME>($<LINKED-CLASS-NAME>Id);
12 }
```

Where:

<**LINKED-CLASS-NAME**> The name of the linked class from which instances are to be deleted.

<**CLASS-NAME**> The name of *theClass*.

E.4.5 Implementation Rule 'irCorrespondingClasses'

Implemented Requirements. The *irCorrespondingClasses* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raCorrespondingClasses* requirement archetype (Section C.4.4)
  - *theParentClass* : xtUML.Class
  - *theSpecialisedClass* : xtUML.Class

Description. The *irCorrespondingClasses* implementation rule is used to implement situations in which instances of *Classes* in two separate domains represent the same thing. For example, a *Manager* class in a banking domain may correspond to a *AuthenticatedUser* in a security domain. Instances of the *Manager* class are individuals who are also represented by instances of the *AuthenticatedUser* class.

Example. See line 36 of the `pmsCreateTables.sql` source file (Section F.3.1 on page 247) for an example of code generated using this rule.

Instructions. Each of the classes are implemented as described in the *irPersistentClass* implementation rule (Section E.4.4) with the following additional attribute.

```
1 <OTHER-CLASS-NAME>Id int
```

Where:

<**OTHER-CLASS-NAME**> The name of the class with which this class corresponds.

E.5 Security implementation rules

E.5.1 *Security Mechanisms* domain model

The *Security Mechanisms* domain model comprises a collection of *PHP* functions that implement various security related functions including user authentication.

This *Domain Model* is an instance of the *PHP Web Programming Language* domain model and comprises the PHP source code listed in Section G.5 of Appendix G.

E.5.2 Implementation Rule 'irSecurityConfig'

Implemented Requirements. The *irSecurityConfig* implementation rule implements requirements that conform to the following *Requirement Archetype*.

- *raSecurityPhpImplementation* requirement archetype (Section C.3.1)
  - *theSecurityDomain* : SEC.
  - *thePhpDomain* = PHP.

Description. The *irSecurityConfig* implementation rule is used to implement a security configuration file for use with the application.

Example. See the `securityConfiguration.php` source file (Section F.2.3 on page 245) for an example of code generated using this rule.

Security configuration file. A PHP file, named as follows, shall be implemented and stored in the *Application Directory*.

```
1 securityConfiguration.php
```

The file shall have the following contents:

```
1 <?php
2
3 session_start();
4
5 define ('DEFAULT_ROLE',  <DEFAULT-ROLE-NAME>);
6
7 $securityPermissions = <PERMISSIONS-ARRAY>;
8
9 ?>
```

Where:

<**DEFAULT-ROLE**> The name of the default *Security.Role* for new users.

<**PERMISSIONS-ARRAY**> A three dimensional array that specifies the *Security.Permissions* associated with each *Security.Role* used by the application. The first dimension identifies the *Security.Roles* defined within *theSecurityDomain*. The second identifies the *Security.ProtectedItems* identified within *theSecurityDomain*, and the third dimension identifies each of the *Security.Permissions* applicable to each *ProtectedItem*. The value of each array element is a boolean that is *true* when the permission is granted.

E.6 Conclusion

In this appendix, I have demonstrated how an *Implementation Model* can be developed for a given *Aspect-Oriented Specification Archetype*. In particular, I presented an *Implementation Model* that can be used to implement any *Aspect-Oriented Specification* that has been formed in accordance with the LAMP *Aspect-Oriented Specification Archetype* presented in Appendix C.

Appendix

F

Proof of concept - *Annotated Source Code*

Contents

F.1 Introduction 244
F.2 Application configuration 244
F.2.1 userInterfaceConfiguration.php 244
F.2.2 databaseConfiguration.php 245
F.2.3 securityConfiguration.php 245
F.3 Database 247
F.3.1 pmsCreateTables.sql 247
F.3.2 pmsDropDatabase.sql 248
F.3.3 pmsPopulateTables.sql 248
F.4 Application startup 249
F.4.1 index.html 249
F.5 Authentication 249
F.5.1 loginPage.php 249
F.5.2 loginBridge.php 250
F.5.3 registerUserPage.php 251
F.5.4 registerUserBridge.php 252
F.5.5 registrationSuccessPage.php 253
F.6 Main menu 253
F.6.1 mainMenuPage.php 253
F.6.2 logoffBridge.php 254
F.7 Product management 255
F.7.1 productListPage.php 255
F.7.2 addProductBridge.php 257

F.7.3 deleteProductBridge.php . . . 257
F.7.4 editProductPage.php . . . 258
F.7.5 updateProductBridge.php . . . 260
F.7.6 addProductVariantBridge.php . . . 261
F.7.7 deleteProductVariantBridge.php . . . 262
F.7.8 editProductVariantPage.php . . . 262
F.7.9 updateProductVariantBridge.php . . . 264
F.7.10 productMechanisms.php . . . 265
F.7.11 productVariantMechanisms.php . . . 266
F.8 Product category management . . . 268
F.8.1 productCategoryListPage.php . . . 268
F.8.2 addProductCategoryBridge.php . . . 269
F.8.3 deleteProductCategoryBridge.php . . . 270
F.8.4 editProductCategoryPage.php . . . 270
F.8.5 updateProductCategoryBridge.php . . . 272
F.8.6 productCategoryMechanisms.php . . . 272
F.9 Contact management . . . 274
F.9.1 contactListPage.php . . . 274
F.9.2 addContactBridge.php . . . 275
F.9.3 deleteContactBridge.php . . . 276
F.9.4 editContactPage.php . . . 276
F.9.5 updateContactBridge.php . . . 280
F.9.6 addContactRelationshipBridge.php . . . 281
F.9.7 deleteContactRelationshipBridge.php . . . 282
F.9.8 editContactRelationshipPage.php . . . 282
F.9.9 updateContactRelationshipBridge.php . . . 284
F.9.10 addNoteBridge.php . . . 285
F.9.11 deleteNoteBridge.php . . . 285
F.9.12 editNotePage.php . . . 286
F.9.13 updateNoteBridge.php . . . 287
F.9.14 contactMechanisms.php . . . 288
F.9.15 contactRelationshipMechanisms.php . . . 290
F.9.16 noteMechanisms.php . . . 291
F.10 User management . . . 293
F.10.1 userListPage.php . . . 293
F.10.2 addUserBridge.php . . . 294
F.10.3 deleteUserBridge.php . . . 295
F.10.4 editUserPage.php . . . 295

F.10.5 updateUserBridge.php 297
F.10.6 userMechanisms.php 298
F.11 Conclusion 299

F.1 Introduction

This appendix, along with Appendix G, contains all of the software source code required to build and operate the *Product Management System*. All of the software listed in this appendix was generated by manually translating the *Product Management System Aspect-Oriented Specification* (Appendix D) in accordance with the *LAMP Implementation Model* (Appendix E).

This appendix is organised by grouping software modules along functional lines. Within each functional grouping, modules are listed along with annotations which identify the *Implementation Rules* used to generate specific lines of code within each module.

Note that the text of this appendix was automatically generated from the *Product Management System* source code and therefore reflects the actual *Product Management System* code. A CDROM containing the *Product Management System* source code is also available from the author at the following address:

Shayne Flint
College of Engineering and Computer Science
Australian National University
ACTON ACT 0200
Australia

Email: shayne.flint@anu.edu.au

F.2 Application configuration

F.2.1 userInterfaceConfiguration.php

generated by *irUiConfig* implementation rule (Section E.3.3):

```
1  <?php
2
3  session_start();
4
5  define("APPLICATION_NAME", "Product Management System");
6
7  define("PAGE_FOOTER", "Aspect-Oriented Thinking proof-of-concept");
8
```

F.2.2 databaseConfiguration.php

generated by *irDatabaseConfig* implementation rule (Section E.4.2):

```
<?php

session_start();

define("DATABASE_HOST",     "127.0.0.1");
define("DATABASE_NAME",     "shayne_phd");
define("DATABASE_USERNAME", "root");
define("DATABASE_PASSWORD", "password");

```

F.2.3 securityConfiguration.php

generated by *irSecurityConfig* implementation rule (Section E.5.2):

```
<?php

session_start();

define ('DEFAULT_ROLE',  'customer');

$securityPermissions = array
  ( 'staff'    =>
       array ( 'products'          => array ( 'view'      => True,
                                              'edit'      => True,
                                              'create'    => True,
                                              'delete'    => True
                                            ),
               'productVariants'   => array ( 'view'      => True,
                                              'edit'      => True,
                                              'create'    => True,
                                              'delete'    => True
                                            ),
               'productCategories' => array ( 'view'      => True,
                                              'edit'      => True,
                                              'create'    => True,
                                              'delete'    => True
                                            ),
               'contacts'          => array ( 'view'      => True,
                                              'edit'      => True,
                                              'create'    => True,
```

```
                                          'delete'    => True
                                        ),
            'relationships'     => array ( 'view'      => True,
                                          'edit'      => True,
                                          'create'    => True,
                                          'delete'    => True
                                        ),
            'notes'             => array ( 'view'      => True,
                                          'edit'      => True,
                                          'create'    => True,
                                          'delete'    => True
                                        ),
            'users'             => array ( 'view'      => True,
                                          'edit'      => True,
                                          'create'    => True,
                                          'delete'    => True
                                        )
         ),
   'customer' =>
      array ( 'products'          => array ( 'view'      => True,
                                          'edit'      => False,
                                          'create'    => False,
                                          'delete'    => False
                                        ),
            'productVariants'   => array ( 'view'      => True,
                                          'edit'      => False,
                                          'create'    => False,
                                          'delete'    => False
                                        ),
            'productCategories' => array ( 'view'      => True,
                                          'edit'      => False,
                                          'create'    => False,
                                          'delete'    => False
                                        ),
            'contacts'          => array ( 'view'      => False,
                                          'edit'      => False,
                                          'create'    => False,
                                          'delete'    => False
                                        ),
            'relationships'     => array ( 'view'      => False,
                                          'edit'      => False,
                                          'create'    => False,
                                          'delete'    => False
                                        ),
            'notes'             => array ( 'view'      => False,
                                          'edit'      => False,
```

```
                                                'create'    => False,
                                                'delete'    => False
                                            ),
                'users'             => array ( 'view'      => False,
                                                'edit'      => False,
                                                'create'    => False,
                                                'delete'    => False
                                            )
            )
  );


?>
```

F.3 Database

F.3.1 pmsCreateTables.sql

generated by *irDatabaseScripts* implementation rule (Section E.4.3):

```
create database shayne_phd;
use shayne_phd;

```

generated by *irPersistentClass* implementation rule (Section E.4.4):

```
create table users
  (id int unsigned not null auto_increment primary key,
   userName char(30), password char(100), roleId int);
describe users;

create table roles
  (id int unsigned not null auto_increment primary key,
   name char(30));
describe roles;



create table contacts
  (id int unsigned not null auto_increment primary key,
   name char(100), address char(200), telephone char(50),
   mobile char(50), email char(100) );
```

```
describe contacts;

create table contactRelationships
  (id int unsigned not null auto_increment primary key,
   fromContactId int, toContactId int, relationshipTypeId int);
describe contactRelationships;

create table relationshipTypes
  (id int unsigned not null auto_increment primary key,
   toName char(30), fromName char(30));
describe relationshipTypes;


create table products
  (id int unsigned not null auto_increment primary key,
   productCategoryId int, name char(100), description text,
   supplierId int );
describe products;

create table productCategories
  (id int  unsigned not null auto_increment primary key,
   name char(100), notes text);
describe productCategories;

create table productVariants
  (id int unsigned not null auto_increment primary key,
   productId int, description text, price decimal(7,2));
describe productVariants;

create table notes
  (id int unsigned not null auto_increment primary key,
   contactId int, userId int, date datetime, notes char(200));
describe notes;
```

F.3.2 pmsDropDatabase.sql

generated by *irDatabaseScripts* implementation rule (Section E.4.3):

```
drop database shayne_phd;
```

F.3.3 pmsPopulateTables.sql

generated by *irDatabaseScripts* implementation rule (Section E.4.3):

```
use shayne_phd;

insert into roles values
  (NULL, 'staff'),
  (NULL, 'customer');

insert into users values
  (NULL, 'teststaff',    password('password'), 1),
  (NULL, 'testcustomer', password('password'), 2);

insert into relationshipTypes values
  (NULL, 'is owner of', 'is owned by'),
  (NULL, 'manages finances for', 'finances are managed by'),
  (NULL, 'manages shipping for', 'shipping is managed by'),
  (NULL, 'manages HR for', 'HR is managed by'),
  (NULL, 'manages returns for', 'returns are managed by');

```

F.4 Application startup

F.4.1 index.html

generated by *irStartPage* implementation rule (Section E.3.2):

```
<html>
<head>
<meta http-equiv="REFRESH"
      content="0; URL=loginPage.php">
</head>
</html>
```

F.5 Authentication

F.5.1 loginPage.php

generated by *irPage* implementation rule (Section E.3.4):

```
<?php
require_once('userInterfaceMechanisms.php');

```

```
4  session_start();
5
6  showLoginPage();
7
8  function showLoginPage() {
9    uiInit();
10   uiStartPage('Login');
11   uiSetCurrentUrl("Login Page", "loginPage.php");
```

generated by *irStaticPageItems* implementation rule (Section E.3.5):

```
12   uiText
13    ("Enter username and password. "
14       ."Click 'Login' to start application:");
```

generated by *irForm* implementation rule (Section E.3.9):

```
15   uiStartForm('loginBridge.php');
16   uiStartTable(300);
17   uiTextInput('Username', 'username', '', 20);
18   uiPasswordInput('Password', 'password', 20) ;
19   uiEndTable();
20   uiSubmitButton('Login');
21   uiEndForm();
```

generated by *irPageLink* implementation rule (Section E.3.7):

```
22   uiLink('registerUserPage.php', 'Click here to register');
```

generated by *irPage* implementation rule (Section E.3.4):

```
23   uiEndPage();
24 }
25
26 ?>
27
```

F.5.2 loginBridge.php

generated by *irForm* implementation rule (Section E.3.9):

```
<?php
require_once('securityMechanisms.php');
require_once('userInterfaceMechanisms.php');

session_start();

$username = $HTTP_POST_VARS['username'];
$password = $HTTP_POST_VARS['password'];

try {
  login($username, $password);
  uiShowUrl("mainMenuPage.php");
} catch (Exception $e) {
  uiErrorPage($e);
}

?>

```

F.5.3 registerUserPage.php

generated by *irPage* implementation rule (Section E.3.4):

```
<?php
require_once('userInterfaceMechanisms.php');

session_start();

showRegisterUserPage();

function showRegisterUserPage() {
  uiSetCurrentUrl("Register New User", "registerUserPage.php");
  uiStartPage("Register New User");
```

generated by *irStaticPageItems* implementation rule (Section E.3.5):

```
  uiText('Complete this form and click "Register" to register:');
```

generated by *irForm* implementation rule (Section E.3.9):

```
  uiStartForm('registerUserBridge.php');
  uiStartTable(300);
```

```
14    uiTextInput('Username', 'username', '', 20);
15    uiPasswordInput('Password', 'password', 20);
16    uiEndTable();
17    uiSubmitButton('Register');
18    uiEndForm();
```

generated by *irPageLink* implementation rule (Section E.3.7):

```
19    uiLink("loginPage.php", "Click here to login");
```

generated by *irPage* implementation rule (Section E.3.4):

```
20    uiEndPage();
21  }
22
23  ?>
24
```

F.5.4 registerUserBridge.php

generated by *irForm* implementation rule (Section E.3.9):

```
1   <?php
2
3   require_once('securityMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5
6   session_start();
7
8   $username = $HTTP_POST_VARS['username'];
9   $password = $HTTP_POST_VARS['password'];
10
11  try {
12    registerUser($username, $password);
13    uiShowUrl("registrationSuccessPage.php");
14  } catch (Exception $e) {
15    uiErrorPage($e);
16  }
17  ?>
18
```

F.5.5 registrationSuccessPage.php

generated by *irPage* implementation rule (Section E.3.4):

```
<?php
require_once('userInterfaceMechanisms.php');

session_start();

showRegistrationSuccessPage();

function showRegistrationSuccessPage() {
  uiStartPage("Registration Success");
  uiSetCurrentUrl
    ("Registration Success", "registrationSuccessPage.php");
  uiLastUrlLink();
```

generated by *irStaticPageItems* implementation rule (Section E.3.5):

```
  uiText("The new username has been registered.");
```

generated by *irPageLink* implementation rule (Section E.3.7):

```
  uiLink('loginPage.php', 'Click here to login');
```

generated by *irPage* implementation rule (Section E.3.4):

```
  uiEndPage();
}

?>

```

F.6 Main menu

F.6.1 mainMenuPage.php

generated by *irPage* implementation rule (Section E.3.4):

```
1  <?php
2
3  require_once('userInterfaceMechanisms.php');
4  require_once('securityMechanisms.php');
5
6  session_start();
7
8  showMainMenuPage();
9
10 function showMainMenuPage() {
11   uiStartPage("Main Menu");
12   uiSetCurrentUrl("Main Menu", "mainMenuPage.php");
```

generated by *irPageLink* implementation rule (Section E.3.7):

```
13   uiLink('productListPage.php', 'Product List');
```

generated by *irRolePageLink* implementation rule (Section E.3.8):

```
14   if (userHasRole('staff')) {
15     uiLink('productCategoryListPage.php', 'Product Category List');
16     uiLink('contactListPage.php', 'Contact List');
17     uiLink('userListPage.php', 'User List');
18   }
```

generated by *irFunctionLink* implementation rule (Section E.3.6):

```
19   uiLink('logoffBridge.php', 'Click here to logoff');
```

generated by *irPage* implementation rule (Section E.3.4):

```
20   uiEndPage();
21 }
22
```

F.6.2 logoffBridge.php

generated by *irFunctionLink* implementation rule (Section E.3.6):

```
<?php
require_once('securityMechanisms.php');
require_once('userInterfaceMechanisms.php');

session_start();

try {
  logoff();
  uiShowUrl("loginPage.php");
} catch (Exception $e) {
  uiErrorPage($e);
}


?>

```

F.7 Product management

F.7.1 productListPage.php

generated by *irPage* implementation rule (Section E.3.4):

```
<?php

require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');

session_start();

showProductListPage();

function showProductListPage() {
  connectToDatabase();
  uiStartPage("Products");

  uiSetCurrentUrl("Product List", "productListPage.php");
  uiLastUrlLink();
```

generated by *irGroupedClassList* implementation rule (Section E.3.11):

```
17  if (userHasPermission('products', 'create')) {
18    uiLink('addProductBridge.php', 'Add a new product');
19  }
20
21  $groupSet = mysql_query( "SELECT * FROM productCategories" );
22  if (!$groupSet) {
23    uiFatalErrorPage
24      ("Unable to query productCategories - ".mysql_error());
25  }
26  $groupCount = mysql_num_rows($groupSet);
27  for ($i=0; $i<$groupCount; $i++) {
28    $groupInstance = mysql_fetch_array($groupSet);
29    $groupId       = $groupInstance['id'];
30
31    $groupName = $groupInstance['name'];
32    uiSubHeading($groupName);
33
34    $notes = $groupInstance['notes'];
35    uiText("Note: <i>".$notes."</i>");
36
37    $set = mysql_query
38      ("SELECT * FROM products
39          WHERE productCategoryId='$groupId'");
40    if (!$set) {
41      uiFatalErrorPage
42        ("Unable to query products - ".mysql_error());
43    }
44
45    uiStartTable(800);
46    $count = mysql_num_rows($set);
47    for ($j=0; $j<$count; $j++) {
48      $instance          = mysql_fetch_array($set);
49      $productCategoryId = $instance['id'];
50      $name              = $instance['name'];
51
52      uiSecureEditDeleteRow
53        ("products", $name,
54         "editProductPage.php?id=$productCategoryId",
55         "deleteProductBridge.php?id=$productCategoryId");
56    }
57    uiEndTable();
58  }
```

generated by *irPage* implementation rule (Section E.3.4):

```
59    uiEndPage();
60  }
61
```

F.7.2 addProductBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
1   <?php
2
3   require_once('productMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   try {
10
11    $id = newProduct();
12    uiShowUrl("editProductPage.php?id=$id");
13
14  } catch (Exception $e) {
15    uiErrorPage($e);
16  }
17
18  ?>
```

F.7.3 deleteProductBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
1   <?php
2
3   require_once('productMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   $id = $HTTP_GET_VARS['id'];
10
```

```
try {

  deleteProduct($id);
  uiShowTopUrl();

} catch (Exception $e) {
  uiErrorPage($e);
}

?>

```

F.7.4 editProductPage.php

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
<?php

require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');

session_start();

$id = $HTTP_GET_VARS['id'];

showEditProductPage($id);

function showEditProductPage($id) {
  connectToDatabase();
  uiStartPage("Edit Product");

  uiSetCurrentUrl("Edit Product", "editProductPage.php?id=$id");
  uiLastUrlLink();

  uiText
    ("Modify Product fields and click 'Update product' to save:");

  $set = mysql_query("SELECT * FROM products WHERE id=$id");
  if (!$set) {
    uiFatalErrorPage("Unable to query products - ".mysql_error());
  }
  $theProduct        = mysql_fetch_array($set);
  $productCategoryId = $theProduct['productCategoryId'];
```

```
29  $name              = $theProduct['name'];
30  $description       = $theProduct['description'];
31  $supplierId        = $theProduct['supplierId'];
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
32  uiStartForm('updateProductBridge.php');
33  uiStartTable(300);
34  uiHiddenField('id', $id);
```

generated by *irLinkObjectSelect* implementation rule (Section E.3.17):

```
35  uiSecureSelect
36    ('products', 'Category', 'productCategoryId',
37     'productCategories', 'name', $productCategoryId);
```

generated by *irAttributeTextInput* implementation rule (Section E.3.14):

```
38  uiSecureTextInput
39    ('products', 'Name', 'name', $name, 60);
```

generated by *irLinkObjectSelect* implementation rule (Section E.3.17):

```
40  uiSecureSelect
41    ('products', 'Supplier', 'supplierId', 'contacts',
42     'name', $supplierId);
```

generated by *irAttributeTextArea* implementation rule (Section E.3.16):

```
43  uiSecureTextArea
44    ('products', 'Description', 'description',
45     $description, 60, 6);
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
46  uiEndTable();
47  uiSecureSubmitButton('products', 'Update product');
48  uiEndForm();
```

generated by *irStaticPageItems* implementation rule (Section E.3.5):

```
49   uiHeading('Variants and Prices');
```

generated by *irClassList* implementation rule (Section E.3.10):

```
50   if (userHasPermission('productVariants', 'create')) {
51     uiLink
52       ("addProductVariantBridge.php?productId=$id",
53        "Add a new product variant");
54   }
55   $set = mysql_query
56     ( "SELECT * FROM productVariants WHERE productId=$id" );
57   if (!$set) {
58     uiFatalErrorPage
59       ("Unable to query productVariants product - ".mysql_error());
60   }
61   uiStartTable(400);
62   $count = mysql_num_rows($set);
63   for ($j=0; $j<$count; $j++) {
64     $instance         = mysql_fetch_array($set);
65     $productVariantId = $instance['id'];
66
67     $description      = $instance['description'];
68     $price            = $instance['price'];
69     $name = $description." - $".$price;
70
71     uiSecureEditDeleteRow
72       ('productVariants', $name,
73        "editProductVariantPage.php?id=$productVariantId",
74        "deleteProductVariantBridge.php?id=$productVariantId");
75   }
76   uiEndTable();
```

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
77   uiEndPage();
78 }
79
80
```

F.7.5 updateProductBridge.php

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
<?php

require_once('productMechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

$id                = $HTTP_POST_VARS['id'];
$productCategoryId = $HTTP_POST_VARS['productCategoryId'];
$name              = $HTTP_POST_VARS['name'];
$description       = $HTTP_POST_VARS['description'];
$supplierId        = $HTTP_POST_VARS['supplierId'];

try {

  updateProduct
    ($id, $productCategoryId, $name, $description, $supplierId);
  uiShowTopUrl();

} catch (Exception $e) {
  uiErrorPage($e);
}
?>

```

F.7.6 addProductVariantBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
<?php

require_once('productVariantMechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

$productId = $HTTP_GET_VARS['productId'];

try {

  $id = newProductVariant($productId);
```

```
14    uiShowUrl("editProductVariantPage.php?id=$id");
15
16  } catch (Exception $e) {
17    uiErrorPage($e);
18  }
19  ?>
```

F.7.7 deleteProductVariantBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
1   <?php
2
3   require_once('productVariantMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   $id        = $HTTP_GET_VARS['id'];
10
11  try {
12
13    deleteProductVariant($id);
14    uiShowTopUrl();
15
16  } catch (Exception $e) {
17    uiErrorPage($e);
18  }
19
20  ?>
21
```

F.7.8 editProductVariantPage.php

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
1   <?php
2
3   require_once('userInterfaceMechanisms.php');
4   require_once('databaseMechanisms.php');
5   require_once('securityMechanisms.php');
```

```
6  require_once('applicationMechanisms.php');
7
8  session_start();
9
10 $id = $HTTP_GET_VARS['id'];
11
12 showEditProductVariantPage($id);
13
14
15 function showEditProductVariantPage($id) {
16   connectToDatabase();
17   uiStartPage("Update Product Variant");
18
19   uiSetCurrentUrl
20     ("Edit Product Variant", "editProductVariantPage.php?id=$id");
21   uiLastUrlLink();
22
23   uiText
24     ("Modify Product Variant fields and click"
25        ." 'Update Product Variant' to save:");
26
27   $set = mysql_query
28     ("SELECT * FROM productVariants WHERE id=$id");
29   if (!$set) {
30     uiFatalErrorPage
31       ("Cannot query productVariants - ".mysql_error());
32   }
33   $theRecord   = mysql_fetch_array($set);
34   $description = $theRecord['description'];
35   $price       = $theRecord['price'];
36   $productId   = $theRecord['productId'];
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
37   uiStartForm('updateProductVariantBridge.php');
38   uiStartTable(300);
39   uiHiddenField('id', $id);
40   uiHiddenField('productId', $productId);
```

generated by *irAttributeTextInput* implementation rule (Section E.3.14):

```
41   uiSecureTextInput
42     ('productVariants', 'Description', 'description',
43      $description, 60);
```

```
44    uiSecureTextInput
45      ('productVariants', 'Price', 'price', $price, 60);
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
46    uiEndTable();
47    uiSecureSubmitButton
48      ('productVariants', 'Update Product Variant');
49    uiEndForm();
```

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
50    uiEndPage();
51  }
52
```

F.7.9 updateProductVariantBridge.php

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
1   <?php
2
3   require_once('productVariantMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   $id          = $HTTP_POST_VARS['id'];
10  $description = $HTTP_POST_VARS['description'];
11  $price       = $HTTP_POST_VARS['price'];
12  $productId   = $HTTP_POST_VARS['productId'];
13
14  try {
15
16    updateProductVariant($id, $description, $price);
17    uiShowTopUrl();
18
19  } catch (Exception $e) {
20    uiErrorPage($e);
21  }
22  ?>
```

F.7.10 productMechanisms.php

generated by *irPersistentClass* implementation rule (Section E.4.4):

```
<?php
session_start();

require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');
require_once('productVariantMechanisms.php');

connectToDatabase();


function newProduct() {
  $result = mysql_query("INSERT INTO products SET
              productCategoryId=0,
              name='',
              description='',
              supplierId=0");
  if (!$result) {
    throw new Exception("Unable to add product - ".mysql_error());
  }
  $id = mysql_insert_id();
  return $id;
}


function updateProduct
  ($id, $productCategoryId, $name, $description, $supplierId) {
  connectToDatabase();

  $result = mysql_query("UPDATE products SET
              productCategoryId=$productCategoryId,
              name='$name',
              description='$description',
              supplierId=$supplierId
              WHERE id=$id");
  if (!$result) {
    throw new Exception
      ("Unable to update product - ".mysql_error());
  }
}


```

```
function deleteProduct($id) {
  connectToDatabase();

  $result = mysql_query("DELETE FROM products WHERE id=$id");
  if (!$result) {
    throw new Exception
      ("Unable to delete product - ".mysql_error());
  }

  $set = mysql_query
    ("SELECT * FROM productVariants WHERE productId=$id");
  if (!$set) {
    throw new Exception
      ("Unable to select productVariants - ".mysql_error());
  }
  $count = mysql_num_rows($set);
  for ($i=0; $i<$count; $i++) {
    $instance   = mysql_fetch_array($set);
    $productVariantId  = $instance['id'];
    deleteProductVariant($productVariantId);
  }
}


```

F.7.11 productVariantMechanisms.php

generated by *irPersistentClass* implementation rule (Section E.4.4):

```
<?php
session_start();

require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');

connectToDatabase();




function newProductVariant($productId) {
  connectToDatabase();

  $result = mysql_query("INSERT INTO productVariants SET
              productId=0,
```

```
                description='',
                price=0.0");

  if (!$result) {
    throw new Exception("Unable to add product - ".mysql_error());
  }

  $id = mysql_insert_id();

  $result = mysql_query("UPDATE productVariants SET
                productId='$productId' WHERE id=$id");

  if (!$result) {
    throw new Exception
      ("Unable to link productVariant to product - "
          .mysql_error());
  }

  return $id;
}


function updateProductVariant($id, $description, $price) {
  connectToDatabase();

  $result = mysql_query
     ("UPDATE productVariants SET
               description='$description',
               price='$price'
               WHERE id=$id");
  if (!$result) {
    throw new Exception
      ("Unable to update productVariant - ".mysql_error());
  }
}


function deleteProductVariant($id) {
  connectToDatabase();

  $result = mysql_query
    ("DELETE FROM productVariants WHERE id=$id");
  if (!$result) {
    throw new Exception
      ("Unable to delete productVariant - ".mysql_error());
  }
```

```
62 }
63
64
65
66 ?>
```

F.8 Product category management

F.8.1 productCategoryListPage.php

generated by *irPage* implementation rule (Section E.3.4):

```
1  <?php
2
3  require_once('userInterfaceMechanisms.php');
4  require_once('databaseMechanisms.php');
5  require_once('securityMechanisms.php');
6
7  session_start();
8
9  showCategoryListPage();
10
11 function showCategoryListPage() {
12   connectToDatabase();
13   uiStartPage("Categories");
14
15   uiSetCurrentUrl
16    ("Product Category List", "productCategoryListPage.php");
17   uiLastUrlLink();
```

generated by *irClassList* implementation rule (Section E.3.10):

```
18   if (userHasPermission('productCategories', 'create')) {
19     uiLink('addProductCategoryBridge.php', 'Add a new category');
20   }
21
22   $set = mysql_query( "SELECT * FROM productCategories" );
23   if (!$set) {
24     uiFatalErrorPage
25       ("Unable to query productCategories - ".mysql_error());
26   }
27   uiStartTable(400);
```

```
  $count = mysql_num_rows($set);
  for ($i=0; $i<$count; $i++) {
    $instance          = mysql_fetch_array($set);
    $productCategoryId = $instance['id'];
    $name              = $instance['name'];
    uiSecureEditDeleteRow
      ("productCategories", $name,
       "editProductCategoryPage.php?id=$productCategoryId",
       "deleteProductCategoryBridge.php?id=$productCategoryId");
  }
  uiEndTable();
```

generated by *irPage* implementation rule (Section E.3.4):

```
  uiEndPage();
}

```

F.8.2 addProductCategoryBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
<?php

require_once('productCategoryMechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

try {

  $id = newProductCategory();
  uiShowUrl("editProductCategoryPage.php?id=$id");

} catch (Exception $e) {
  uiErrorPage($e);
}

```

F.8.3 deleteProductCategoryBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
<?php

require_once('productCategoryMechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

$id = $HTTP_GET_VARS['id'];

try {

  deleteProductCategory($id);
  uiShowTopUrl();

} catch (Exception $e) {
  uiErrorPage($e);
}

?>

```

F.8.4 editProductCategoryPage.php

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
<?php

require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');
require_once('applicationMechanisms.php');

session_start();

$id = $HTTP_GET_VARS['id'];

showEditProductCategoryPage($id);

function showEditProductCategoryPage($id) {
```

```
15  connectToDatabase();
16  uiStartPage("Update Category");
17
18  uiSetCurrentUrl
19    ("Edit Product Category",
20     "editProductCategoryPage.php?id=$id");
21  uiLastUrlLink();
22
23  uiText
24    ("Modify Category fields and click"
25       ." 'Update category' to save:");
26
27  $set = mysql_query
28    ("SELECT * FROM productCategories WHERE id=$id");
29  if (!$set) {
30    uiFatalErrorPage("Unable to edit category - ".mysql_error());
31  }
32  $theCategory   = mysql_fetch_array($set);
33  $name   = $theCategory['name'];
34  $notes  = $theCategory['notes'];
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
35  uiStartForm('updateProductCategoryBridge.php');
36
37  uiStartTable(300);
38  uiHiddenField('id', $id);
```

generated by *irAttributeTextInput* implementation rule (Section E.3.14):

```
39  uiSecureTextInput
40    ('productCategories', 'Name', 'name', $name, 60);
```

generated by *irAttributeTextArea* implementation rule (Section E.3.16):

```
41  uiSecureTextArea
42    ('productCategories', 'Notes', 'notes', $notes, 60, 6);
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
43  uiEndTable();
44  uiSecureSubmitButton('productCategories', 'Update category');
45  uiEndForm();
```

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
46    uiEndPage();
47  }
48
```

F.8.5 updateProductCategoryBridge.php

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
1   <?php
2
3   require_once('productCategoryMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   $id          = $HTTP_POST_VARS['id'];
10  $name        = $HTTP_POST_VARS['name'];
11  $notes       = $HTTP_POST_VARS['notes'];
12
13  try {
14
15    updateProductCategory($id, $name, $notes);
16    uiShowTopUrl();
17
18  } catch (Exception $e) {
19    uiErrorPage($e);
20  }
21  ?>
```

F.8.6 productCategoryMechanisms.php

generated by *irPersistentClass* implementation rule (Section E.4.4):

```
1   <?php
2   session_start();
3
4   require_once('databaseMechanisms.php');
5   require_once('securityMechanisms.php');
6   require_once('productMechanisms.php');
```

```
7
8  connectToDatabase();
9
10
11 function newProductCategory() {
12   connectToDatabase();
13
14   $result = mysql_query("INSERT INTO productCategories SET
15                                      name='',
16                                      notes=''");
17   if (!$result) {
18     throw new Exception
19       ("Unable to add productCategory - ".mysql_error());
20   }
21   $id = mysql_insert_id();
22   return $id;
23 }
24
25
26
27 function updateProductCategory($id, $name, $notes) {
28   connectToDatabase();
29
30   $result = mysql_query("UPDATE productCategories SET
31                                  name='$name',
32                                  notes='$notes'
33                                  WHERE id=$id");
34   if (!$result) {
35     throw new Exception
36       ("Unable to update productCategory - ".mysql_error());
37   }
38 }
39
40
41 function deleteProductCategory($id) {
42   connectToDatabase();
43
44   $result = mysql_query
45     ("DELETE FROM productCategories WHERE id=$id");
46   if (!$result) {
47     throw new Exception
48       ("Unable to delete productCategory - ".mysql_error());
49   }
50
51   $result = mysql_query
52     ("SELECT * FROM products WHERE productCategoryId=$id");
```

```
53    if (!$result) {
54      throw new Exception
55        ("Unable to query products - ".mysql_error());
56    }
57    $count = mysql_num_rows($result);
58    for ($i=0; $i<$count; $i++) {
59      $instance = mysql_fetch_array($result);
60      $productId = $instance['id'];
61      deleteProduct($productId);
62    }
63  }
64
```

F.9 Contact management

F.9.1 contactListPage.php

generated by *irPage* implementation rule (Section E.3.4):

```
1   <?php
2
3   require_once('userInterfaceMechanisms.php');
4   require_once('databaseMechanisms.php');
5   require_once('securityMechanisms.php');
6
7   session_start();
8
9   showContactListPage();
10
11  function showContactListPage() {
12    connectToDatabase();
13    uiStartPage("Contacts");
14
15    uiSetCurrentUrl("Contact List", "contactListPage.php");
16    uiLastUrlLink();
```

generated by *irClassList* implementation rule (Section E.3.10):

```
17    if (userHasPermission('contacts', 'create')) {
18      uiLink('addContactBridge.php', 'Add a new contact');
19    }
20
```

```
21    $set = mysql_query( "SELECT * FROM contacts" );
22    if (!$set) {
23      uiFatalErrorPage("Unable to query contacts - ".mysql_error());
24    }
25    uiStartTable(400);
26    $count = mysql_num_rows($set);
27    for ($i=0; $i<$count; $i++) {
28      $instance   = mysql_fetch_array($set);
29      $contactId  = $instance['id'];
30      $name       = $instance['name'];
31      uiSecureEditDeleteRow
32          ("contacts", $name,
33           "editContactPage.php?id=$contactId",
34           "deleteContactBridge.php?id=$contactId");
35    }
36    uiEndTable();
```

generated by *irPage* implementation rule (Section E.3.4):

```
37    uiEndPage();
38  }
39
```

F.9.2 addContactBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
1   <?php
2
3   require_once('contactMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   try {
10
11    $id = newContact();
12    uiShowUrl("editContactPage.php?id=$id");
13
14  } catch (Exception $e) {
15    uiErrorPage($e);
16  }
```

```
17
18 ?>
```

F.9.3 deleteContactBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
1  <?php
2
3  require_once('contactMechanisms.php');
4  require_once('userInterfaceMechanisms.php');
5  require_once('databaseMechanisms.php');
6
7  session_start();
8
9  $id = $HTTP_GET_VARS['id'];
10
11 try {
12
13   deleteContact($id);
14   uiShowTopUrl();
15
16 } catch (Exception $e) {
17   uiErrorPage($e);
18 }
19
20 ?>
```

F.9.4 editContactPage.php

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
1  <?php
2
3  require_once('userInterfaceMechanisms.php');
4  require_once('databaseMechanisms.php');
5  require_once('securityMechanisms.php');
6  require_once('applicationMechanisms.php');
7
8  session_start();
9
10 $id = $HTTP_GET_VARS['id'];
```

```
11
12 showEditContactPage($id);
13
14 function showEditContactPage($id) {
15   connectToDatabase();
16   uiStartPage("Update Contact");
17
18   uiSetCurrentUrl("Edit Contact", "editContactPage.php?id=$id");
19   uiLastUrlLink();
20
21   uiText
22     ('Modify Contact fields and click "Update Contact" to save:');
23
24   $set = mysql_query("SELECT * FROM contacts WHERE id=$id");
25   if (!$set) {
26     uiFatalErrorPage("Unable to query contacts - ".mysql_error());
27   }
28   $theContact = mysql_fetch_array($set);
29   $name       = $theContact['name'];
30   $address    = $theContact['address'];
31   $telephone  = $theContact['telephone'];
32   $mobile     = $theContact['mobile'];
33   $email      = $theContact['email'];
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
34   uiStartForm('updateContactBridge.php');
35   uiStartTable(300);
36   uiHiddenField('id', $id);
```

generated by *irAttributeTextInput* implementation rule (Section E.3.14):

```
37   uiSecureTextInput
38     ('contacts', 'Name', 'name', $name, 60);
```

generated by *irAttributeTextInput* implementation rule (Section E.3.14):

```
39   uiSecureTextInput
40     ('contacts', 'Address', 'address', $address, 60);
41   uiSecureTextInput
42     ('contacts', 'Telephone', 'telephone', $telephone, 60);
43   uiSecureTextInput
44     ('contacts', 'Mobile', 'mobile', $mobile, 60);
```

```
45  uiSecureTextInput
46    ('contacts', 'Email', 'email', $email, 60);
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
47  uiEndTable();
48  uiSecureSubmitButton('contacts', 'Update Contact');
49  uiEndForm();
```

generated by *irStaticPageItems* implementation rule (Section E.3.5):

```
50  uiHeading('Relationships');
```

generated by *irClassList* implementation rule (Section E.3.10):

```
51  if (userHasPermission('relationships', 'create')) {
52    uiLink
53     ("addContactRelationshipBridge.php?contactId=$id",
54      "Add a new relationship");
55  }
56
57  $set = mysql_query
58    ("SELECT * FROM contactRelationships WHERE fromContactId=$id");
59  if (!$set) {
60    uiFatalErrorPage
61      ("Unable to query contactRelationships - ".mysql_error());
62  }
63  uiStartTable(510);
64  $count = mysql_num_rows($set);
65  for ($i=0; $i<$count; $i++) {
66    $instance              = mysql_fetch_array($set);
67    $contactRelationshipId = $instance['id'];
68    $toContactId           = $instance['toContactId'];
69    $relationshipTypeId    = $instance['relationshipTypeId'];
70
71    $contactSet = mysql_query
72       ( "SELECT * FROM contacts WHERE id=$toContactId" );
73    if (!$contactSet) {
74      uiFatalErrorPage
75        ("Unable to query contacts - ".mysql_error());
76    }
77    $theContact = mysql_fetch_array($contactSet);
78    $name       = $theContact['name'];
```

```
79
80      $relationshipSet = mysql_query
81        ("SELECT * FROM relationshipTypes"
82           ." WHERE id=$relationshipTypeId");
83      $theRelationship  = mysql_fetch_array($relationshipSet);
84      $relationshipName = $theRelationship['fromName'];
85
86      uiSecureEditDeleteRow
87        ('relationships', "<i>$relationshipName</i> $name",
88         "editContactRelationshipPage.php?id=$contactRelationshipId",
89         "deleteContactRelationshipBridge.php"
90            ."?id=$contactRelationshipId&contactId=$id");
91    }
92
93    $set = mysql_query
94      ( "SELECT * FROM contactRelationships WHERE toContactId=$id" );
95    if (!$set) {
96      uiFatalErrorPage
97        ("Unable to query contactRelationships - ".mysql_error());
98    }
99    uiStartTable(510);
100   $count = mysql_num_rows($set);
101   for ($i=0; $i<$count; $i++) {
102     $instance              = mysql_fetch_array($set);
103     $contactRelationshipId = $instance['id'];
104     $fromContactId         = $instance['fromContactId'];
105     $relationshipTypeId    = $instance['relationshipTypeId'];
106
107     $contactSet = mysql_query
108       ("SELECT * FROM contacts WHERE id=$fromContactId");
109     if (!$set) {
110       uiFatalErrorPage
111         ("Unable to query contacts - ".mysql_error());
112     }
113     $theContact = mysql_fetch_array($contactSet);
114     $name       = $theContact['name'];
115
116     $relationshipSet  = mysql_query
117       ("SELECT * FROM relationshipTypes"
118          ." WHERE id=$relationshipTypeId");
119     $theRelationship  = mysql_fetch_array($relationshipSet);
120     $relationshipName = $theRelationship['toName'];
121
122     uiSecureEditDeleteRow
123       ('relationships', "<i>$relationshipName</i> $name",
124        "editContactRelationshipPage.php?id=$contactRelationshipId",
```

```
125         "deleteContactRelationshipBridge.php"
126           ."?id=$contactRelationshipId&contactId=$id");
127     }
128     uiEndTable();
```

generated by *irStaticPageItems* implementation rule (Section E.3.5):

```
129     uiHeading('Notes');
```

generated by *irClassList* implementation rule (Section E.3.10):

```
130     if (userHasPermission('notes', 'create')) {
131       uiLink("addNoteBridge.php?contactId=$id", "Add a new note");
132     }
133
134     $set = mysql_query("SELECT * FROM notes WHERE contactId=$id");
135     if (!$set) {
136       uiFatalErrorPage("Unable to query notes - ".mysql_error());
137     }
138     uiStartTable(510);
139     $count = mysql_num_rows($set);
140     for ($i=0; $i<$count; $i++) {
141       $instance = mysql_fetch_array($set);
142       $noteId   = $instance['id'];
143       $date     = $instance['date'];
144       $notes    = $instance['notes'];
145       uiSecureEditDeleteRow
146          ('notes', "[$date] $notes", "editNotePage.php?id=$noteId",
147           "deleteNoteBridge.php?id=$noteId&contactId=$id");
148     }
149     uiEndTable();
```

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
150     uiEndPage();
151 }
152
```

F.9.5 updateContactBridge.php

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
<?php

require_once('contactMechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

$id         = $HTTP_POST_VARS['id'];
$name       = $HTTP_POST_VARS['name'];
$address    = $HTTP_POST_VARS['address'];
$telephone  = $HTTP_POST_VARS['telephone'];
$mobile     = $HTTP_POST_VARS['mobile'];
$email      = $HTTP_POST_VARS['email'];

try {

  updateContact($id, $name, $address, $telephone, $mobile, $email);
  uiShowTopUrl();

} catch (Exception $e) {
  uiErrorPage($e);
}

?>
```

F.9.6 addContactRelationshipBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
<?php

require_once('contactRelationshipMechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();


$contactId = $HTTP_GET_VARS['contactId'];

try {

```

```
14    $id = newContactRelationship($contactId);
15    uiShowUrl("editContactRelationshipPage.php?id=$id");
16
17  } catch (Exception $e) {
18    uiErrorPage($e);
19  }
20  ?>
```

F.9.7 deleteContactRelationshipBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
1   <?php
2
3   require_once('contactRelationshipMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   $id = $HTTP_GET_VARS['id'];
10
11  try {
12
13    deleteContactRelationship($id);
14    uiShowTopUrl();
15
16  } catch (Exception $e) {
17    uiErrorPage($e);
18  }
19
```

F.9.8 editContactRelationshipPage.php

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
1   <?php
2
3   require_once('userInterfaceMechanisms.php');
4   require_once('databaseMechanisms.php');
5   require_once('securityMechanisms.php');
6   require_once('applicationMechanisms.php');
```

```
7
8  session_start();
9
10 $id = $HTTP_GET_VARS['id'];
11
12 showEditRelationshipPage($id);
13
14
15 function showEditRelationshipPage($id) {
16   connectToDatabase();
17   uiStartPage("Update Relationship");
18
19   uiSetCurrentUrl
20     ("Edit Contact Relationship",
21      "editContactRelationshipPage.php?id=$id");
22   uiLastUrlLink();
23
24   uiText
25     ("Modify Relationship fields and click"
26        ." 'Update relationship' to save:");
27
28   $set = mysql_query
29     ("SELECT * FROM contactRelationships WHERE id=$id");
30   if (!$set) {
31     uiFatalErrorPage
32       ("Cannot query contactRelationships - ".mysql_error());
33   }
34   $theRelationship    = mysql_fetch_array($set);
35   $fromContactId      = $theRelationship['fromContactId'];
36   $toContactId        = $theRelationship['toContactId'];
37   $relationshipTypeId = $theRelationship['relationshipTypeId'];
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
38   uiStartForm('updateContactRelationshipBridge.php');
39   uiStartTable(300);
40   uiHiddenField('id', $id);
```

generated by *irLinkObjectSelect* implementation rule (Section E.3.17):

```
41   uiSecureSelect
42     ('relationships', "", 'fromContactId',
43      'contacts', 'name', $fromContactId);
44   uiSecureSelect
```

```
45      ('relationships', "", 'relationshipTypeId',
46       'relationshipTypes', 'fromName', $relationshipTypeId);
47    uiSecureSelect
48      ('relationships', "", 'toContactId',
49       'contacts', 'name', $toContactId);
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
50    uiEndTable();
51    uiSecureSubmitButton('relationships', 'Update relationship');
52    uiEndForm();
```

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
53    uiEndPage();
54  }
```

F.9.9 updateContactRelationshipBridge.php

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
1   <?php
2
3   require_once('contactRelationshipMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   $id                   = $HTTP_POST_VARS['id'];
10  $fromContactId        = $HTTP_POST_VARS['fromContactId'];
11  $toContactId          = $HTTP_POST_VARS['toContactId'];
12  $relationshipTypeId   = $HTTP_POST_VARS['relationshipTypeId'];
13
14  try {
15
16    updateContactRelationship
17      ($id, $fromContactId, $toContactId, $relationshipTypeId);
18    uiShowTopUrl();
19
20  } catch (Exception $e) {
21    uiErrorPage($e);
```

```
}

?>
```

F.9.10 addNoteBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
<?php

require_once('noteMechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

$contactId = $HTTP_GET_VARS['contactId'];

try {

  $id = newNote($contactId);
  uiShowUrl("editNotePage.php?id=$id");

} catch (Exception $e) {
  uiErrorPage($e);
}
?>
```

F.9.11 deleteNoteBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
<?php

require_once('noteMechanisms.php');
require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');

session_start();

$id = $HTTP_GET_VARS['id'];

```

```
try {

  deleteNote($id);
  uiShowTopUrl();

} catch (Exception $e) {
  uiErrorPage($e);
}

?>
```

F.9.12 editNotePage.php

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
<?php

require_once('userInterfaceMechanisms.php');
require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');
require_once('applicationMechanisms.php');

session_start();

$id = $HTTP_GET_VARS['id'];

showEditNotePage($id);


function showEditNotePage($id) {
  connectToDatabase();
  uiStartPage("Update Note");

  uiSetCurrentUrl("Edit Note", "editNotePage.php?id=$id");
  uiLastUrlLink();

  uiText("Modify Note fields and click 'Update Note' to save:");

  $set = mysql_query("SELECT * FROM notes WHERE id=$id");
  if (!$set) {
    uiFatalErrorPage("Cannot query notes - ".mysql_error());
  }
  $theRecord  = mysql_fetch_array($set);
  $contactId  = $theRecord['contactId'];
```

```
30    $date       = $theRecord['date'];
31    $notes      = $theRecord['notes'];
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
32    uiStartForm('updateNoteBridge.php');
33    uiStartTable(300);
34    uiHiddenField('id', $id);
35    uiHiddenField('contactId', $contactId);
```

generated by *irAttributeTextInput* implementation rule (Section E.3.14):

```
36    uiSecureTextInput('notes', 'Date Time', 'date', $date, 60);
```

generated by *irAttributeTextArea* implementation rule (Section E.3.16):

```
37    uiSecureTextArea('notes', 'Notes', 'notes', $notes, 60, 6);
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
38    uiEndTable();
39    uiSecureSubmitButton('notes', 'Update Note');
40    uiEndForm();
```

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
41    uiEndPage();
42  }
43
```

F.9.13 updateNoteBridge.php

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
1   <?php
2
3   require_once('noteMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
```

```
6
7  session_start();
8
9  $id        = $HTTP_POST_VARS['id'];
10 $date      = $HTTP_POST_VARS['date'];
11 $notes     = $HTTP_POST_VARS['notes'];
12 $contactId = $HTTP_POST_VARS['contactId'];
13
14 try {
15
16   updateNote($id, $date, $notes);
17   uiShowTopUrl();
18
19 } catch (Exception $e) {
20   uiErrorPage($e);
21 }
22
23 ?>
```

F.9.14 contactMechanisms.php

generated by *irPersistentClass* implementation rule (Section E.4.4):

```
1  <?php
2  session_start();
3
4  require_once('databaseMechanisms.php');
5  require_once('securityMechanisms.php');
6
7  connectToDatabase();
8
9
10 function newContact() {
11   $result = mysql_query("INSERT INTO contacts SET
12                name='',
13                address='',
14                telephone='',
15                mobile='',
16                email=''");
17   if (!$result) {
18     throw new Exception("Unable to add contact - ".mysql_error());
19   }
20   $id = mysql_insert_id();
21   return $id;
```

```
}


function updateContact
  ($id, $name, $address, $telephone, $mobile, $email) {
  connectToDatabase();

  $result = mysql_query("UPDATE contacts SET
               name='$name',
               address='$address',
               telephone='$telephone',
               mobile='$mobile',
               email='$email'
               WHERE id=$id");
  if (!$result) {
    throw new Exception
                ("Unable to update contact - ".mysql_error());
  }
}


function deleteContact($id) {
  connectToDatabase();

  $result = mysql_query("DELETE FROM contacts WHERE id=$id");
  if (!$result) {
    throw new Exception
      ("Unable to delete contact - ".mysql_error());
  }
  $result = mysql_query
    ("DELETE FROM notes WHERE contactId=$id");
  if (!$result) {
    throw new Exception
      ("Unable to delete notes - ".mysql_error());
  }
  $result = mysql_query
    ("DELETE FROM contactRelationships"
       ." WHERE toContactId='$contactId'");
  if (!$result) {
    throw new Exception
       ("Unable to delete contactRelationship - ".mysql_error());
  }
  $result = mysql_query
    ("DELETE FROM contactRelationships"
       ." WHERE fromContactId='$contactId'");
  if (!$result) {
```

```
68      throw new Exception
69        ("Unable to delete contactRelationship - ".mysql_error());
70    }
71  }
72
73
74  ?>
```

F.9.15 contactRelationshipMechanisms.php

generated by *irPersistentClass* implementation rule (Section E.4.4):

```
1   <?php
2   session_start();
3
4   require_once('databaseMechanisms.php');
5   require_once('securityMechanisms.php');
6   require_once('productMechanisms.php');
7
8   connectToDatabase();
9
10
11  function newContactRelationship($contactId) {
12
13    connectToDatabase();
14
15    $result = mysql_query("INSERT INTO contactRelationships SET
16                           fromContactId=0,
17                           toContactId=0,
18                           relationshipTypeId=0");
19
20    if (!$result) {
21      throw new Exception
22        ("Unable to add contactRelationship - ".mysql_error());
23    }
24
25    $id = mysql_insert_id();
26
27    $result = mysql_query("UPDATE contactRelationships SET
28                 fromContactId='$contactId' WHERE id=$id");
29
30    if (!$result) {
31      throw new Exception
32        ("Unable to link contactRelationships to product - "
```

```
33          .mysql_error());
34    }
35    return $id;
36  }
37
38
39  function updateContactRelationship
40    ($id, $fromContactId, $toContactId, $relationshipTypeId) {
41    connectToDatabase();
42
43    if ($fromContactId == $toContactId) {
44      $result = mysql_query
45        ("DELETE FROM contactRelationships WHERE id=$id");
46      throw new Exception
47        ('fromContact cannot equal toContact');
48    }
49
50    $result = mysql_query("UPDATE contactRelationships SET
51                fromContactId=$fromContactId,
52                toContactId=$toContactId,
53                relationshipTypeId=$relationshipTypeId
54                WHERE id=$id");
55    if (!$result) {
56      throw new Exception
57        ("Unable to update relationship - ".mysql_error());
58    }
59  }
60
61
62  function deleteContactRelationship($id) {
63    connectToDatabase();
64
65    $result = mysql_query
66      ("DELETE FROM contactRelationships WHERE id=$id");
67    if (!$result) {
68      throw new Exception
69        ("Unable to delete contactRelationship - ".mysql_error());
70    }
71  }
72
```

F.9.16 noteMechanisms.php

generated by *irPersistentClass* implementation rule (Section E.4.4):

```
<?php
session_start();

require_once('databaseMechanisms.php');
require_once('securityMechanisms.php');

connectToDatabase();



function newNote($contactId) {

  connectToDatabase();

  $date = date('Y-m-d H:i:s');

  $result = mysql_query("INSERT INTO notes SET
              contactId='$contactId',
              date='$date'");

  if (!$result) {
    throw new Exception("Unable to add note - ".mysql_error());
  }

  $id = mysql_insert_id();

  $result = mysql_query("UPDATE notes SET
              contactId='$contactId' WHERE id=$id");

  if (!$result) {
    throw new Exception
      ("Unable to link note to product - ".mysql_error());
  }

  return $id;
}


function updateNote($id, $date, $notes) {
  connectToDatabase();

  $result = mysql_query("UPDATE notes SET
              date='$date',
              notes='$notes'
              WHERE id='$id'");
```

```
46    if (!$result) {
47      throw new Exception
48        ("Unable to update note - ".mysql_error());
49    }
50  }
51
52
53  function deleteNote($id) {
54    connectToDatabase();
55
56    $result = mysql_query("DELETE FROM notes WHERE id=$id");
57    if (!$result) {
58      throw new Exception
59        ("Unable to delete note - ".mysql_error());
60    }
61  }
62
63  ?>
```

F.10 User management

F.10.1 userListPage.php

generated by *irPage* implementation rule (Section E.3.4):

```
1   <?php
2
3   require_once('userInterfaceMechanisms.php');
4   require_once('databaseMechanisms.php');
5   require_once('securityMechanisms.php');
6
7   session_start();
8
9   showUserListPage();
10
11  function showUserListPage() {
12    connectToDatabase();
13    uiStartPage("users");
14
15    uiSetCurrentUrl("User List", "userListPage.php");
16    uiLastUrlLink();
```

generated by *irClassList* implementation rule (Section E.3.10):

```
17    if (userHasPermission('users', 'create')) {
18      uiLink('addUserBridge.php', 'Add a new user');
19    }
20
21    $set = mysql_query( "SELECT * FROM users" );
22    if (!$set) {
23      uiFatalErrorPage("Unable to query users - ".mysql_error());
24    }
25    uiStartTable(400);
26    $count = mysql_num_rows($set);
27    for ($i=0; $i<$count; $i++) {
28      $instance   = mysql_fetch_array($set);
29      $userId     = $instance['id'];
30      $name       = $instance['userName'];
31      uiSecureEditDeleteRow
32          ("users", $name,
33           "editUserPage.php?id=$userId",
34           "deleteUserBridge.php?id=$userId");
35    }
36    uiEndTable();
```

generated by *irPage* implementation rule (Section E.3.4):

```
37    uiEndPage();
38  }
39
```

F.10.2 addUserBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
1   <?php
2
3   require_once('userMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   try {
10
11    $id = newUser();
```

```
12    uiShowUrl("editUserPage.php?id=$id");
13
14  } catch (Exception $e) {
15    uiErrorPage($e);
16  }
17
18  ?>
```

F.10.3 deleteUserBridge.php

generated by *irClassList* implementation rule (Section E.3.10):

```
1   <?php
2
3   require_once('userMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   $id = $HTTP_GET_VARS['id'];
10
11  try {
12
13    deleteUser($id);
14    uiShowTopUrl();
15
16  } catch (Exception $e) {
17    uiErrorPage($e);
18  }
19
20  ?>
```

F.10.4 editUserPage.php

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
1   <?php
2
3   require_once('userInterfaceMechanisms.php');
4   require_once('databaseMechanisms.php');
5   require_once('securityMechanisms.php');
```

```
6  require_once('applicationMechanisms.php');
7
8  session_start();
9
10 $id = $HTTP_GET_VARS['id'];
11
12 showEditUserPage($id);
13
14
15 function showEditUserPage($id) {
16   connectToDatabase();
17   uiStartPage("Edit User");
18
19   uiSetCurrentUrl("Edit User", "editUserPage.php?id=$id");
20   uiLastUrlLink();
21
22   uiText
23     ("Modify User fields and click 'Update user' to save:");
24
25   $set = mysql_query("SELECT * FROM users WHERE id=$id");
26   if (!$set) {
27     uiFatalErrorPage("Cannot query users - ".mysql_error());
28   }
29
30   $theUser    = mysql_fetch_array($set);
31   $userName   = $theUser['userName'];
32   $roleId     = $theUser['roleId'];
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
33   uiStartForm('updateUserBridge.php');
34   uiStartTable(300);
35   uiHiddenField('id', $id);
```

generated by *irAttributeTextInput* implementation rule (Section E.3.14):

```
36   uiSecureTextInput('users', 'User', 'userName', $userName, 60);
```

generated by *irAttributePasswordInput* implementation rule (Section E.3.15):

```
37   uiSecurePasswordInput('users', 'Password', 'password', 60);
```

generated by *irLinkObjectSelect* implementation rule (Section E.3.17):

```
38    uiSecureSelect
39      ('users', "Role", 'roleId',
40       'roles', 'name', $roleId);
```

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
41    uiEndTable();
42    uiSecureSubmitButton('users', 'Update User');
43    uiEndForm();
```

generated by *irEditClassPage* implementation rule (Section E.3.12):

```
44    uiEndPage();
45  }
46
```

F.10.5 updateUserBridge.php

generated by *irEditClassForm* implementation rule (Section E.3.13):

```
1   <?php
2
3   require_once('userMechanisms.php');
4   require_once('userInterfaceMechanisms.php');
5   require_once('databaseMechanisms.php');
6
7   session_start();
8
9   $id        = $HTTP_POST_VARS['id'];
10  $userName  = $HTTP_POST_VARS['userName'];
11  $password  = $HTTP_POST_VARS['password'];
12  $roleId    = $HTTP_POST_VARS['roleId'];
13
14  try {
15
16    updateUser($id, $userName, $password, $roleId);
17    uiShowTopUrl();
18
19  } catch (Exception $e) {
20    uiErrorPage($e);
21  }
```

```
22
23 ?>
```

F.10.6 userMechanisms.php

generated by *irPersistentClass* implementation rule (Section E.4.4):

```
1  <?php
2  session_start();
3
4  require_once('databaseMechanisms.php');
5  require_once('securityMechanisms.php');
6
7  connectToDatabase();
8
9
10 function newUser() {
11   connectToDatabase();
12
13   $result = mysql_query("INSERT INTO users SET
14                          userName='',
15                          password='',
16                          roleId=''");
17   if (!$result) {
18     throw new Exception
19       ("Unable to add user - ".mysql_error());
20   }
21   $id = mysql_insert_id();
22   return $id;
23 }
24
25
26
27 function updateUser($id, $userName, $password, $roleId) {
28   connectToDatabase();
29
30   $result = mysql_query("UPDATE users SET
31                                userName='$userName',
32                                password=password('$password'),
33                                roleId='$roleId'
34                                WHERE id=$id");
35   if (!$result) {
36     throw new Exception
37       ("Unable to update user - ".mysql_error());
```

```
38    }
39  }
40
41
42  function deleteUser($id) {
43    connectToDatabase();
44
45    $result = mysql_query
46      ("DELETE FROM users WHERE id=$id");
47    if (!$result) {
48      throw new Exception
49        ("Unable to delete user - ".mysql_error());
50    }
51  }
52
```

F.11 Conclusion

In this appendix, along with Appendix G, I have demonstrated how an *Implementation Model* can be used to fully implement an *Aspect-Oriented Specification*. In particular, I have presented the entire source code for the *Product Management System* which was generated by implementing the *Aspect-Oriented Specification* presented in Appendix D using the *Implementation Model* presented in Appendix E.

Appendix

G

Proof of concept - *LAMP Mechanisms*

Contents

G.1 Introduction 302

G.2 *Application Mechanisms* domain model 302

G.2.1 applicationMechanisms.php 302

G.3 *Database Mechanisms* domain model 303

G.3.1 databaseMechanisms.php 304

G.4 *User Interface Mechanisms* domain model 304

G.4.1 userInterfaceMechanisms.php 304

G.5 *Security Mechanisms* domain model 312

G.5.1 securityMechanisms.php 312

G.1 Introduction

This appendix contains the mechanisms used by the *Implementation Model* documented in Appendix E. These *Mechanisms* are simply *Domain Models* that are instances of the *PHP Domain Model*.

G.2 *Application Mechanisms* domain model

The *Application Mechanisms* used by the *Implementation Model* documented in Appendix E are declared in the `applicationechanisms.php` source file presented below.

G.2.1 applicationMechanisms.php

```
1  <?php
2
3  session_start();
4
5
6  function setContext($name, $value) {
7    $_SESSION[$name] = $value;
8  }
9
10 function getContext($name) {
11   return $_SESSION[$name];
12 }
13
14 function contextSet($name) {
15   return $_SESSION[$name] != NULL;
16 }
17
18 function clearContextStack($name) {
19   $_SESSION[$name."_TOP"] = 0;
20 }
21
22 function sizeContextStack($name) {
23   return $_SESSION[$name."_TOP"];
24 }
25
26 function pushContextStack($name, $value) {
27   $_SESSION[$name."_TOP"] = $_SESSION[$name."_TOP"] + 1;
28   $stack = $_SESSION[$name];
```

```
29    $stack[$_SESSION[$name."_TOP"]] = $value;
30    $_SESSION[$name] = $stack;
31  }
32
33  function popContextStack($name) {
34    $stack = $_SESSION[$name];
35    $value = $stack[$_SESSION[$name."_TOP"]];
36    $_SESSION[$name."_TOP"] = $_SESSION[$name."_TOP"] - 1;
37    return $value;
38  }
39
40  function topContextStack($name) {
41    $stack = $_SESSION[$name];
42    $value = $stack[$_SESSION[$name."_TOP"]];
43    return $value;
44  }
45
46  function getContextStack($name, $offset) {
47    $stack = $_SESSION[$name];
48    $value = $stack[$_SESSION[$name."_TOP"]+$offset];
49    return $value;
50  }
51
52  function printContextStack($name) {
53    $stack = $_SESSION[$name];
54    $top   = $_SESSION[$name."_TOP"];
55    print "Context Stack '".$name."' top='".$top."'<br>";
56    for ($p=$top; $p>0; $p--){
57      print "[".$p."] = ".$stack[$p]."<br>";
58    }
59  }
60
61  ?>
```

G.3 *Database Mechanisms* domain model

The *Database Mechanisms* used by the *Implementation Model* documented in Appendix E are declared in the `databaseMechanisms.php` source file presented below.

G.3.1 databaseMechanisms.php

```
<?php

require_once('databaseConfiguration.php');

session_start();

function connectToDatabase() {
  $db = mysql_pconnect
    (DATABASE_HOST, DATABASE_USERNAME, DATABASE_PASSWORD);
  if (!$db) {
    die("Couldn't connect to MySQL");
  }
  if (!mysql_select_db(DATABASE_NAME))
    die("Couldn't select ".DATABASE_NAME." database");
  return $db;
}


?>
```

G.4 *User Interface Mechanisms* domain model

The *User Interface Mechanisms* used by the *Implementation Model* documented in Appendix E are declared in the `userInterfaceMechanisms.php` source file presented below.

G.4.1 userInterfaceMechanisms.php

```
<?php

require_once('userInterfaceConfiguration.php');
require_once('applicationMechanisms.php');

session_start();

$uiPageStr = '';


function uiInit() {
  clearContextStack("RETURN_URL");
```

```
13    clearContextStack("RETURN_NAME");
14  }
15
16
17
18  function uiShowUrl($url) {
19    header('location:'.$url);
20  }
21
22  function uiSetCurrentUrl($name, $url) {
23    if (topContextStack("RETURN_URL")!=$url) {
24      pushContextStack("RETURN_URL", $url);
25      pushContextStack("RETURN_NAME", $name);
26    }
27  }
28
29  function uiShowLastUrl() {
30    $url  = popContextStack("RETURN_URL");
31    $name = popContextStack("RETURN_NAME");
32    $url  = topContextStack("RETURN_URL");
33    if ($url=="") {
34      uiShowUrl("index.html");
35    } else {
36      uiShowUrl($url);
37    }
38  }
39
40  function uiShowTopUrl() {
41    $url  = topContextStack("RETURN_URL");
42    if ($url=="") {
43      uiShowUrl("index.html");
44    } else {
45      uiShowUrl($url);
46    }
47  }
48
49  function uiLastUrlLink() {
50    $name = getContextStack("RETURN_NAME", -1);
51    uiLink("returnBridge.php", "Return to " .$name);
52  }
53
54
55  function uiEcho($str) {
56    global $uiPageStr;
57    $uiPageStr = $uiPageStr.$str;
58  }
```

```
59
60
61  function bgColor() {
62    return "ccccff";
63  }
64
65  function headingColor() {
66    return "cc0099";
67  }
68
69
70  function uiStartPage($title) {
71    global $uiPageStr;
72    $uiPageStr = "";
73    uiEcho('
74      <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
75      <html lang="en" dir="ltr">
76        <head>
77           <title>'.APPLICATION_NAME.'-'.$title.'</title>
78        </head>
79        <body>');
80    uiHeading(APPLICATION_NAME." - ".$title);
81    uiEcho("<p>");
82  }
83
84
85  function uiEndPage() {
86    global $uiPageStr;
87    uiEcho("
88      <p><FONT size='1'>".PAGE_FOOTER." (PHP ".phpversion().")</FONT>
89      </body>
90    </html>");
91    echo $uiPageStr;
92  }
93
94
95  function uiHeading($title) {
96    uiEcho("<FONT size='+2' color=".headingColor().">
97           <b>$title</b></FONT> <br/>");
98  }
99
100 function uiSubHeading($title) {
101   uiEcho("<FONT size='+1' color=".headingColor().">
102          <b>$title</b></FONT> <br/>");
103 }
104
```

```
105
106 function uiText($text) {
107   uiEcho($text."<p/>");
108 }
109
110
111 function uiLink($url, $text) {
112   uiEcho("
113     <a href='$url'>$text</a><p/>");
114 }
115
116
117 function uiStartForm($action) {
118   uiEcho("
119     <form action='$action' method='post'>");
120 }
121
122 function uiEndForm() {
123   uiEcho("
124     </form>
125     <p/>");
126 }
127
128
129 function uiSubmitButton($label) {
130   $name = $label."Btn";
131   uiEcho("
132     <input type='submit' name='$name' value='$label'>");
133 }
134
135 function uiSecureSubmitButton($protectedItem, $label) {
136   $name = $label."Btn";
137   uiEcho("
138     <input type='submit' name='$name' value='$label'");
139   if (userHasPermission($protectedItem, 'edit')) {
140     uiEcho(">");
141   } else {
142     uiEcho(" DISABLED>");
143   }
144 }
145
146
147 function uiStartTable($width) {
148   uiEcho("
149     <table width='$width' border=0>");
150 }
```

```
151
152 function uiEndTable() {
153   uiEcho("
154     </table>
155     <p/>");
156 }
157
158 function uiHiddenField($name, $value) {
159   uiEcho("
160     <input type='hidden' name='$name' value='$value'>");
161 }
162
163 function uiTextInput($label, $name, $value, $size) {
164   uiEcho("
165     <TR>
166       <TD bgcolor=".bgColor()." align=right>$label</TD>
167       <TD bgcolor=".bgColor()." align=left>
168           <input type='text' name='$name'
169                  value='$value' size='$size'></TD>
170     </TR>");
171 }
172
173 function uiPasswordInput($label, $name, $size) {
174   uiEcho("
175     <TR>
176       <TD bgcolor=".bgColor()." align=right>$label</TD>
177       <TD bgcolor=".bgColor()."> 
178           <input type='password' name='$name' size='$size'></TD>
179     </TR>");
180 }
181
182 function uiTextArea($label, $name, $value, $cols, $rows) {
183   uiEcho("
184     <TR>
185       <TD bgcolor=".bgColor()." align=right>$label</TD>
186       <TD bgcolor=".bgColor()."> 
187           <textarea cols='$cols' rows='$rows' name='$name'></TD>
188     </TR>");
189 }
190
191
192
193
194 function uiSecureTextInput
195   ($protectedItem, $label, $name, $value, $size) {
196
```

```
197   if (userHasPermission($protectedItem, 'view')) {
198     uiEcho("
199       <TR>
200         <TD bgcolor=".bgColor()." align=right>$label</TD>
201         <TD bgcolor=".bgColor()." align=left>
202             <input type='text' name='$name'
203                    value='$value' size='$size'");
204     if (userHasPermission($protectedItem, 'edit')) {
205       uiEcho("></TD>");
206     } else {
207       uiEcho(" DISABLED></TD>");
208     }
209     uiEcho("</TR>");
210   }
211 }
212
213
214 function uiSecurePasswordInput
215   ($protectedItem, $label, $name, $size) {
216
217   if (userHasPermission($protectedItem, 'view')) {
218     uiEcho("
219       <TR>
220         <TD bgcolor=".bgColor()." align=right>$label</TD>
221         <TD bgcolor=".bgColor()." align=left>
222             <input type='password' name='$name'
223                    size='$size'");
224     if (userHasPermission($protectedItem, 'edit')) {
225       uiEcho("></TD>");
226     } else {
227       uiEcho(" DISABLED></TD>");
228     }
229     uiEcho("</TR>");
230   }
231 }
232
233
234
235 function uiSecureTextArea
236   ($protectedItem, $label, $name, $value, $cols, $rows) {
237
238   if (userHasPermission($protectedItem, 'view')) {
239     uiEcho("
240       <TR>
241         <TD bgcolor=".bgColor()." align=right>$label</TD>
242         <TD bgcolor=".bgColor().">
```

```
243         <textarea cols='$cols' rows='$rows' name='$name'");
244       if (userHasPermission($protectedItem, 'edit')) {
245         uiEcho(">");
246       } else {
247         uiEcho(" DISABLED>");
248       }
249       uiEcho("
250         $value</textarea></TD>
251         </TR>");
252     }
253 }
254
255
256 function uiSecureSelect
257     ($protectedItem, $label, $name,
258      $selectClass, $selectAttribute, $selectDefault) {
259
260     if (userHasPermission($protectedItem, 'view')) {
261       uiEcho("
262         <TR>
263           <TD bgcolor=".bgColor()." align=right>$label</TD>
264           <TD bgcolor=".bgColor()." align=left>");
265
266       connectToDatabase();
267       $set = mysql_query("SELECT * FROM $selectClass");
268       uiEcho("
269         <SELECT NAME='$name'");
270       if (!userHasPermission($protectedItem, 'edit')) {
271         uiEcho(" DISABLED>");
272       } else {
273         uiEcho(">");
274       }
275       $count = mysql_num_rows($set);
276       for ($i=0; $i<$count; $i++) {
277         $selectInstance = mysql_fetch_array($set);
278         $data   = $selectInstance[$selectAttribute];
279         $select = $selectInstance['id'];
280         uiEcho("
281           <OPTION VALUE='$select'");
282         if ($select == $selectDefault) {
283           uiEcho(" SELECTED");
284         }
285         uiEcho(">$data</OPTION>");
286       }
287       uiEcho("
288           </SELECT>
```

```
289         </TD>
290       </TR>");
291   }
292 }
293
294
295
296 function uiSecureEditDeleteRow
297   ($protectedItem, $label, $editLink, $deleteLink) {
298
299   if (userHasPermission($protectedItem, 'view')) {
300     uiEcho("
301       <tr>
302         <td bgcolor=".bgColor().">$label</td>
303         <td bgcolor=".bgColor()."><a href='$editLink'>");
304     if (userHasPermission($protectedItem, 'edit')) {
305       uiEcho("edit</td>");
306     } elseif (userHasPermission($protectedItem, 'view')) {
307       uiEcho("view</td>");
308     }
309     if (userHasPermission($protectedItem, 'delete')) {
310       uiEcho("
311         <td bgcolor=".bgColor()."><
312         <a href='$deleteLink'>delete</a></td>");
313     }
314     uiEcho("</tr>");
315   }
316 }
317
318
319
320
321 function uiErrorPage($exception) {
322   uiStartPage("ERROR");
323
324   uiText($exception->getMessage());
325   uiText($php_errormsg);
326
327   uiSetCurrentUrl("Error Page", "");
328   uiLastUrlLink();
329
330   uiEndPage();
331 }
332
333
334 function uiFatalErrorPage($message) {
```

```
335   uiStartPage("FATAL ERROR");
336
337   uiText($message);
338   uiText($php_errormsg);
339   uiLink('logoffBridge.php', 'Click here to log off');
340
341   uiEndPage();
342   exit;
343 }
344
345
346
347 ?>
```

G.5 *Security Mechanisms* domain model

The *Security Mechanisms* used by the *Implementation Model* documented in Appendix E are declared in the `securityMechanisms.php` source file presented below.

G.5.1 securityMechanisms.php

```
1  <?php
2  session_start();
3
4  require_once('databaseMechanisms.php');
5  require_once('applicationMechanisms.php');
6  require_once('securityConfiguration.php');
7
8
9  function registerUser($username, $password) {
10   connectToDatabase();
11   $set = mysql_query
12     ("SELECT * FROM users WHERE username='$username'");
13   $theUser = mysql_fetch_array($set);
14   if ($theUser) {
15     throw new Exception('User already exists');
16   } else {
17     $theRole = DEFAULT_ROLE;
18     $result = mysql_query("INSERT INTO users SET
19                            username='$username',
20                            password=PASSWORD('$password'),
```

```
                                roleName='$theRole'");
    if (!$result) {
      uiFatalErrorPage
        ("Unable to add new user - ".mysql_error());
    }
  }
}

function login($username, $password) {
  connectToDatabase();
  $set = mysql_query
    ("SELECT * FROM users
        WHERE username='$username'
        AND   password = PASSWORD('$password')");
  $theUser = mysql_fetch_array($set);
  if (!$theUser) {
    throw new Exception('invalid user name or incorrect password');
  } else {
    $userId = $theUser['id'];
    $userRoleId = $theUser['roleId'];
    $roleSet = mysql_query
      ("SELECT * FROM roles
          WHERE id='$userRoleId'");
    $theRole = mysql_fetch_array($roleSet);
    if (!$theRole) {
      throw new Exception('invalid role in user record');
    } else {
      $userRole = $theRole['name'];
    }
    setContext('userId', $userId);
    setContext('userName', $username);
    setContext('userRoleName', $userRole);
  }
  return false;
}

function logoff() {
  session_destroy();
}


function userHasRole($roleName) {
  return (strcmp(getContext('userRoleName'), $roleName)==0);
}

```

```
67 function userHasPermission($protectedItem, $permission) {
68   global $securityPermissions;
69
70   $role = getContext('userRoleName');
71
72   return $securityPermissions[$role][$protectedItem][$permission];
73 }
74
75 ?>
```

Appendix

H

Reproduced paper: *Capability Dynamics: An approach to Capability Planning and Development in Large Organisations*

Contents

H.1 Introduction 317
H.1.1 Overview of the Approach 317
H.1.2 Proposed Experimentation 318
H.2 Definitions 318
H.3 Capability dynamics models 321
H.3.1 Primitive Systems and Capability Development 322
H.3.2 Composite Systems, Synthesis and Decomposition 322
H.3.3 Composite Systems and Non-Hierarchical Structure 324
H.3.4 Composite Systems and Change 324
H.3.5 Composite Systems and Abstraction 325
H.3.6 Stakeholder Perspective 325
H.3.7 Scenarios and Preparedness 325
H.3.8 Scalability 326
H.3.9 Emergent Capabilities 326
H.4 Distributed capability dynamics modelling tool 327
H.4.1 Tool Support 327
H.4.2 Simulation 327

H.5 The decision control process 327
H.5.1 Iteration 330
H.6 Proposed Experimentation 330
H.6.1 Experimental Distributed Capability Dynamics Modelling Tool 330
H.6.2 Experiment One: Internet Based Virtual organisation . . . 331
H.6.3 Experiment Two: Case Study 332
H.7 Conclusions 333

Capability Dynamics: An approach to Capability Planning and Development in Large Organisations

Shayne R. Flint
Department of Computer Science
Australian National University
Tel: 02-6125-8183
Email: shayne.flint@anu.edu.au

Clive V. Boughton
Department of Computer Science
Australian National University
Tel: 02-6125-5689
Email: clive.boughton@anu.edu.au

This paper was published in
Proceedings of the 2001 International Council on Systems Engineering Symposium
Melbourne, Australia.

Abstract. This paper describes *Capability Dynamics*, a synthesis of system dynamics, control, agent based simulation and distributed systems techniques, which aims to increase the likelihood that decisions made throughout and at all levels of an organisation contribute in a coordinated way to satisfying the organisation's requirements and those of its stakeholders.

The focus of *Capability Dynamics* is on the identification of systems that need to be developed, modified and retired within a whole-of-capability, and whole-of-life context of required capabilities and evaluation criteria. It is left to other research and existing systems engineering practice to improve individual systems within this context.

H.1 Introduction

H.1.1 Overview of the Approach

The *Capability Dynamics* approach comprises the following integrated elements, each of which will be described fully in this paper:

1. A Decision Control Process which is used throughout an organisation to

ensure that all decisions contribute to the satisfaction of identified requirements.

2. A *Capability Dynamics Model* [*Flint*, 1999] which is developed and maintained by the organisation to provide decision-makers with an organisation wide, shared and evolving understanding of the systems comprising the organisation and its environment, how the capabilities of these systems interact to satisfy stakeholder requirements, and how all these things are likely to evolve with time and in response to change.

3. A computer based Distributed *Capability Dynamics Modelling Tool* which is used by people who work within an organisation as well as other stakeholders to collaborate in the development and use of *Capability Dynamics Models*.

H.1.2 Proposed Experimentation

Experimentation is proposed to evaluate the effectiveness of *Capability Dynamics* by testing the following conjectures:

1. That the effectiveness of an organisation and its internal systems (as measured by the satisfaction of stakeholder requirements) can be improved using the proposed Decision Control Process.

2. That *Capability Dynamics Models* can be used as a framework for capturing information needed by the proposed decision control process.

3. That the proposed Distributed *Capability Dynamics Modelling Tool* enables people within an organisation to collaborate in building, maintaining and using effective *Capability Dynamics Models*.

H.2 Definitions

The following terms are used throughout this paper.

Entity. [*Kramer*, 1977] defines an entity as *something that has objective or physical reality and distinction of being and character*.

Capability. A capability is defined as an ability of an entity to pursue an objective or goal. Examples of capabilities include:

1. The ability to prevent force being used against Australia.

2. The ability to defend Australia against armed attack.
3. The ability of an aircraft to transport a specified load under certain conditions.
4. The ability of a word processor to print documents.
5. The ability of an individual to satisfy a performance agreement.
6. The ability of an organisation to maintain staff morale.

System. A system is defined as a set of interrelated entities, of which no subset is unrelated to any other subset, organised to develop and deliver capabilities by using capabilities of other systems.

Capability Dynamics takes a very broad view of *system* which includes the more obvious entities such as equipment, people, and organisations, but also includes entities such as procedures, practices, processes, the law, and regulations as well as more abstract entities such as plans and architectures.

This definition of system also includes less formal entities such as *Communities of Practice* [*Stewart*, 1996] which are informal social groupings (or systems) that are not specifically developed by organisations, but nevertheless rely on capability from other systems in order to deliver real capability to an organisation.

Figure H.1 shows an example of a system which includes several sub-systems that fall within this broader definition along with an indication of the type of capabilities they may produce.

The icons with titles such as *Reporting process* represent systems. The arrows between systems represent the development and delivery of capability by one system for another. Labels on the arrows indicate the nature of the capability provided. Capabilities flowing into a system are used by that system to develop and deliver new capabilities. For example the *reporter* system makes use of capabilities from several systems including the *Reporting process* and *Computing System* to deliver the ability to produce reports in response to *Manager* system requirements.

Requirement. A requirement is defined as a demand on a system for a specific capability. The *Capability Dynamics* approach asserts that all systems develop and deliver capability in response to requirements and that a set of evaluation criteria is associated with each requirement. These criteria are used as the basis upon which capability (or satisfaction of a requirement) is measured. In general, they equate to what are often called *Measures of Effectiveness*.

Most requirements, apart from physical constraints, result from interactions between people and organisations. These interacting parties will often have

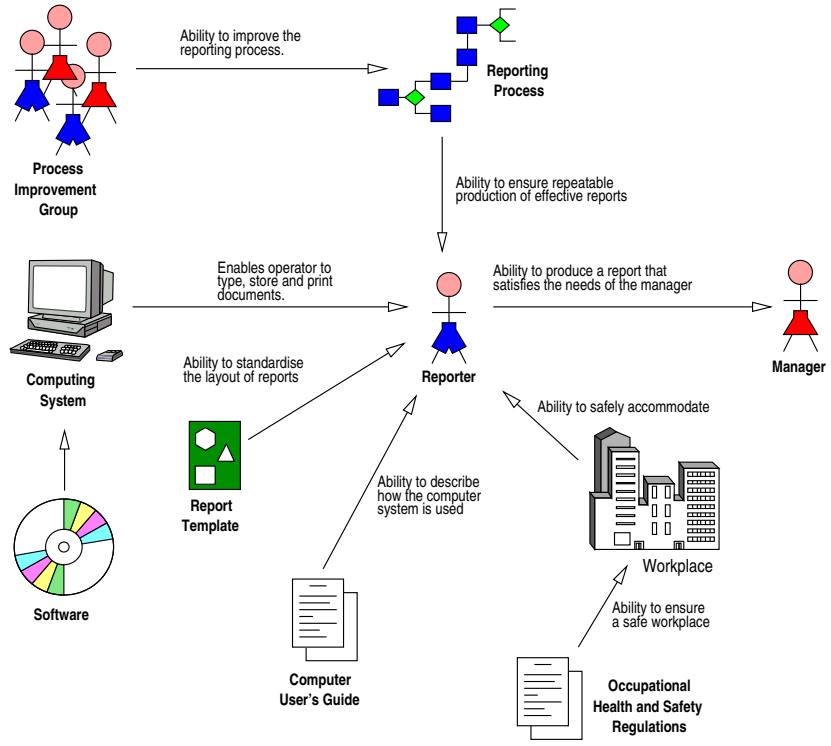

Figure H.1: A report writing system.

different views, motivations and commitments that may not be easily resolved or even explained. This means that requirements, by their very nature, will not always be well defined or understood. In the Australian Defence Organisation (ADO) context, for example, the most fundamental requirements are those set by government policy which is usually developed by defence white paper processes and public consultation activities. These political/people-oriented processes will often result in an eclectic set of requirements for ADO capabilities.

Because of this reality, the *Capability Dynamics* approach assumes no constraints on the nature of requirements. In fact, requirements are considered in a more general, possibly abstract, way than is traditionally the case. For example, evolution can be considered a change in capability resulting from a requirement to survive changes in the environment.

Capability Dynamics therefore assumes the existence of vague and poorly defined requirements and that such requirements cannot be ignored.

Stakeholder. A stakeholder in a given system is defined as any other system (including people) which uses any of the capabilities of, or delivers some capability to that system.

H.3 Capability dynamics models

The Capability Dynamics approach includes a Decision Control Process (described later in this paper) that relies on the following information to support the forming of decisions aimed at satisfying stakeholder requirements.

1. The stakeholder requirements that need to be satisfied as a function of time.
2. How satisfaction of these requirements can be measured.
3. How well capabilities satisfy these requirements over time.
4. The set of sub-systems (if any) involved in satisfying the requirements and how this set changes with time.
5. The dependencies between these systems in terms of requirements and capabilities, including circularities (feedback).
6. The likely impact in time and space of changes to any of the above.

In order for the proposed Decision Control Process to operate within a whole-of-capability and whole-of-life context, the above information needs to be provided by a single integrated model of an entire organisation. The development of such models presents the following problems:

1. Organisation wide models are likely to be very large[1].
2. Such models will be dynamically complex in nature [*Casti*, 1979] because of the dependencies, possible feedback and inherent delays in capability development between systems.
3. An organisation's environment (i.e., stakeholders) will be in a constant state of change. In particular, their requirements on the organisation and their ability to deliver required capabilities will change.
4. Internal stakeholders will have differing, evolving, and possibly conflicting views.
5. Decisions made at different places and times throughout an organisation are likely to interact in dynamically complex ways.

[1]Because CDMs deal only with requirements and capability, they are somewhat simpler and more stable than traditional organisational models that attempt to model organisations in terms of their implementation i.e., Work flows, processes, and social networks etc.

6. The implementation of decisions (use of resources, timing etc.) will interact with other implementation efforts as well as other decision processes in dynamically complex ways.

Because of this complexity, systems are modelled within the *Capability Dynamics* approach using *Capability Dynamics Models* which are Agent Based Models that apply the conceptual foundations of Systems Thinking [*Kramer*, 1977] and Systems Dynamics [*Forrester*, 1961] at the level of requirements and capability.

Capability Dynamics Models describe organisations and other dynamically complex entities as collections of systems which interact to develop, deliver and make use of capability that is intended to satisfy stakeholder requirements - indicated earlier in Figure 1. They do not attempt to model any implementation aspects of systems such as workflow, process and communications. This kind of information is not considered appropriate to the problem of controlling the satisfaction of requirements.

H.3.1 Primitive Systems and Capability Development

Primitive Systems are systems within a *Capability Dynamics Model* that are not decomposed into or synthesised from other systems.

Primitive systems are represented as agents that generate a measure of capability in terms of evaluation criteria associated with the corresponding requirements. These measures are described by Capability Development Models which are functions of time as depicted in Figure H.2.

The *Capability Dynamics* approach is not specific about the nature of Capability Development Models. They can be based on many things including:

1. Mathematical functions of time and measures of the capabilities used from other systems as depicted in Figure H.2.
2. The actual capability developed by real systems as measured by instrumentation, observation, surveys and questionnaires etc.
3. Current plans for capability development.

H.3.2 Composite Systems, Synthesis and Decomposition

Composite Systems are systems within a *Capability Dynamics Model* that are decomposed into, or synthesised from other composite or primitive systems.

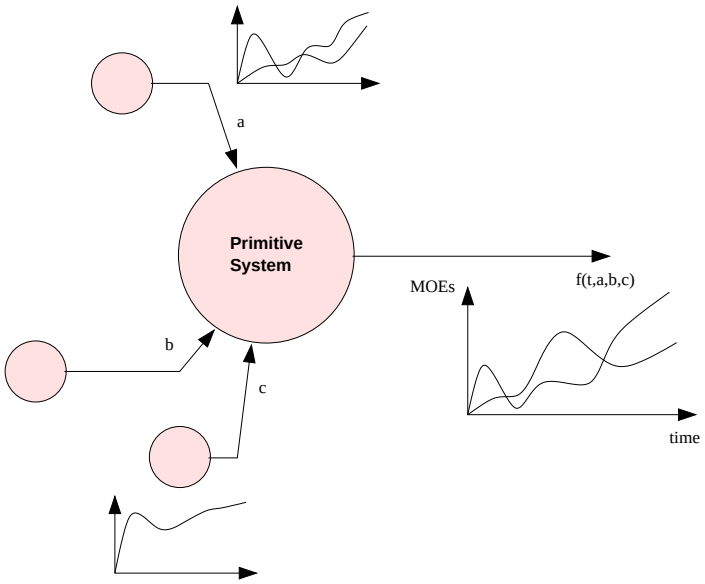

Figure H.2: Capability Development.

Composite Systems, like all systems, make use of capability from other systems to develop and deliver new capability in response to requirements. Unlike primitive systems, the capability delivered by a composite system is defined by the collective capabilities that its sub-systems deliver to external systems (i.e., stakeholders in the composite system). Similarly, the capabilities used by a composite system are those that sub-systems receive from external systems. This view of composite system structure is depicted in Figure H.3.

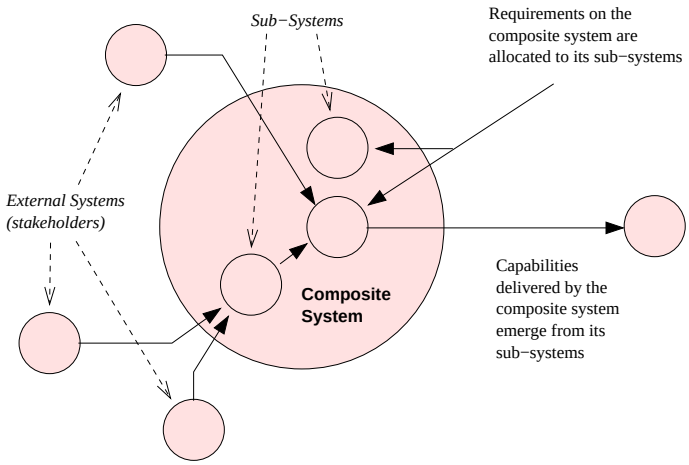

Figure H.3: Composite Systems

H.3.3 Composite Systems and Non-Hierarchical Structure

Composite Systems support more realistic modelling of large organisations than would be the case if a strict hierarchical structure were followed. In particular, *Capability Dynamics Models* are able to capture the fact that a given system may be part of many different composite systems as depicted in Figure H.4. People, for example, often have multiple hats; that is, they are part of more than one composite system. Many generic capabilities such as general maintenance, computing services, communications etc. may also be elements of many composite systems.

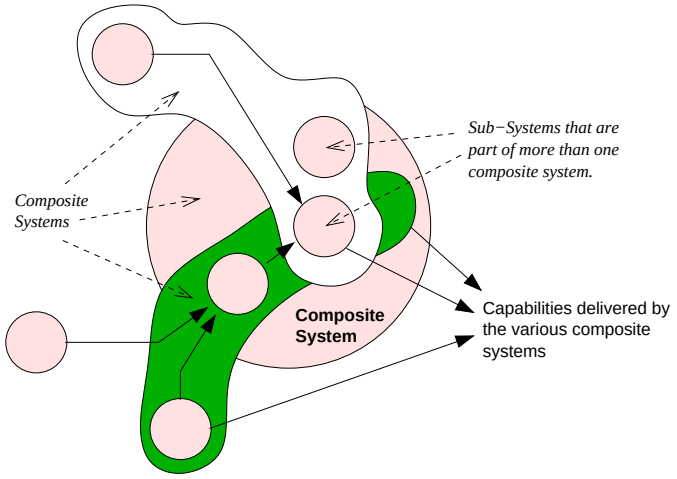

Figure H.4: Non-Hierarchical Composite Systems

H.3.4 Composite Systems and Change

The systems that comprise a composite system will almost certainly change with time due to many factors including the following.

1. Requirements on a composite system may change or emerge over time. Sub-Systems will need to be created, modified and/or assembled to satisfy these new and changing requirements.
2. The capabilities upon which a composite system depends may change.
3. Sub-systems may become unmaintainable, unreliable, ineffective or unsafe with time. Such systems will need to be replaced or modified.
4. New technology and processes may suggest ways to improve the satisfaction of existing requirements.

Whatever the cause, *Capability Dynamics Models* capture changes in subsystems of a composite system as a function of time.

H.3.5 Composite Systems and Abstraction

Composite systems provide an abstraction mechanism which can be used to consider collections of systems as single entities. For example, a detailed *Capability Dynamics Model* of the ADO may include low level details about individuals, units, facilities and equipment. Aggregates of these entities may be used to form Composite Systems representing the Army, Navy and Air Force for the purposes of evaluating and reporting their capability against government requirements.

H.3.6 Stakeholder Perspective

Various stakeholders may consider a given system very differently. Some may consider a system to be a primitive system, but may have different views as to the associated Capability Development Model. Other stakeholders may consider a system to be a composite system, but may have different views as to its composition. Because of this, *Capability Dynamics* does not assume a single view, but rather supports a range of views which can be explored, compared, developed and possibly merged using the Distributed *Capability Dynamics Model*ling Tool described later in this paper.

H.3.7 Scenarios and Preparedness

Most organisations need to plan for a number of possible future scenarios. The ADO, for example, may need to plan for war, peacekeeping and emergency relief. The *Capability Dynamics* approach considers such scenarios as different sets of requirements that may need to be satisfied at some future time. In order to be prepared for such scenarios an organisation may need to develop, maintain, and exercise capabilities that satisfy these scenario-based requirements in advance of a scenario emerging.

Capability Dynamics Models will provide important information regarding the development of these capabilities. For example, appropriate Capability Development Models for systems that could be assembled for a given scenario may provide important information about the time it might take to satisfy the requirements of a given scenario once it emerges. In addition, such models will provide a basis for assessing various options that may exist for dealing with a given scenario including the assessment of risk and the impact a given option may have

on the rest of the organisation and its stakeholders. This kind of information may be important when making decisions about what capabilities need to be developed in advance and to what level (i.e., preparedness requirements).

H.3.8 Scalability

Capability Dynamics is a highly scalable approach because of the way in which the notions of Composite Systems and Capability Development Models of primitive systems work together.

For example, very large organisations such as the ADO could be modelled as a single primitive system that delivers capability to its stakeholders, including the government. Such a model could be used to describe Australia's Strategic Policy [*Australian Department of Defence*, 1997]. At a slightly lower level of detail, the ADO could be modelled as a composite system comprising Defence Operations, Governance and Support sub-systems [*Australian Department of Defence*, 2000]. Models at this level could be used to describe the ADO in terms of requirements laid out in Defence Portfolio Budget Papers and the measurement of delivered capability reported in Defence Annual Reports.

This decomposition can continue to the very lowest levels of the ADO including individuals delivering capability aimed at satisfying requirements laid out in individual performance agreements.

H.3.9 Emergent Capabilities

When working with systems theory, it is common to consider those properties of a system that emerge as a result of complex interactions between system components over time.

In the case of *Capability Dynamics*, there is no interest in emergent properties per se. Instead, they are considered part of system implementation contributing along with other system properties and behavior to the delivery of capability that is measured against stakeholder evaluation criteria.

However, when it comes to improving the satisfaction of stakeholder needs (i.e. making a change), an understanding of the emergent capabilities of systems may suggest changes that would not otherwise be considered. For example, the emergence of new technology may facilitate new systems that can deliver improved capability against existing requirements.

H.4 Distributed capability dynamics modelling tool

The development of *Capability Dynamics Models* is likely to require significant effort. In addition, the authors believe that a sense of ownership in their development and use will be important to the success of *Capability Dynamics*. We therefore propose that *Capability Dynamics Models* be developed by collaboration amongst all stakeholders within an organisation, researchers and consultants.

H.4.1 Tool Support

In order to support the collaborative development of *Capability Dynamics Models*, the authors are developing a computer based Distributed *Capability Dynamics Modelling Tool* which is intended for day-to-day use throughout and at all levels of an organisation. The tool has been designed to allow everyone within an organisation to capture their own view of those sections of the organisation with which they interact or are a part.

The proposed tool will provide facilities to allow stakeholders to discuss differences in opinion, identify those differences that are important and, if required, resolve them. It is hoped that this collaborative approach and use of computer based tool support will result in the emergence of a *Capability Dynamics Model* of an entire organisation from the knowledge and individual contributions of people throughout the organisation.

H.4.2 Simulation

The proposed support tool is designed to allow people throughout an organisation to run agent based simulations of *Capability Dynamics Models* to describe the way in which capability develops over time and in response to actual, proposed and planned changes.

This simulation function plays a critical role in the Decision Control Process described below.

H.5 The decision control process

When a problem situation[2] exists in which requirements are not being satisfied, the *Capability Dynamics* approach requires disciplined application of the Decision

[2][*Checkland*, 1981] defines a *systems approach* as *an approach to a problem which takes a broad view, which tries to take all aspects into account, which concentrates on interaction between different*

Control Process depicted in Figure H.5. The proposed process comprises the following steps which have been designed to ensure that all decisions made throughout an organisation contribute to the satisfaction of identified requirements.

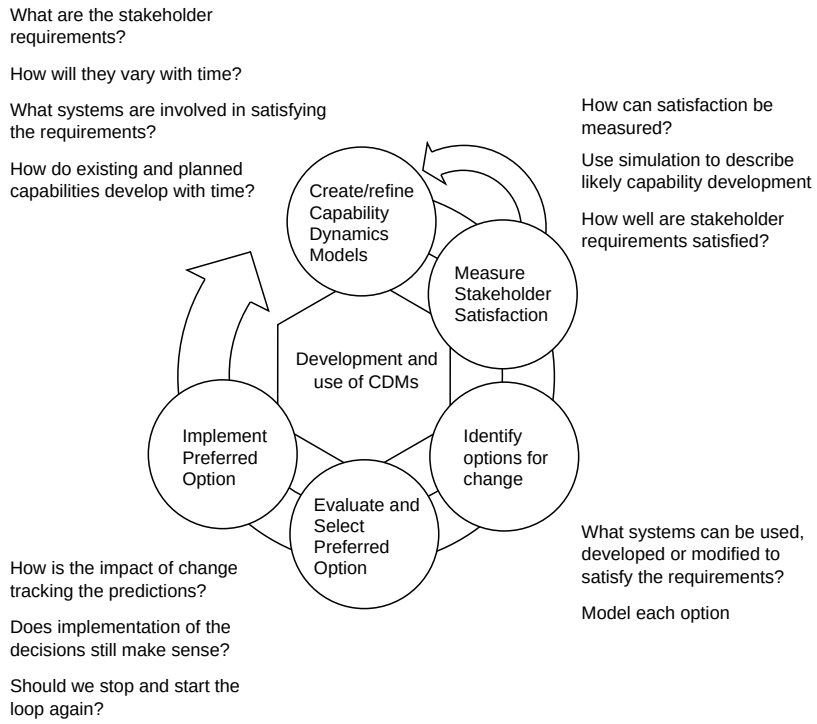

Figure H.5: Decision control process

STEP 1: Create or Refine *Capability Dynamics Models*. The first step in the process is to create or refine those parts of the organisation's *Capability Dynamics Model* which describe the problem situation and how it is likely to change with time. This will be a collaborative process aimed at establishing a common view of the following:

1. The requirements that need to be satisfied along with associated evaluation criteria.
2. How these requirements are likely to evolve with time.
3. The systems involved in satisfying the requirements.
4. The definition of Capability Development Models for each of the primitive systems involved including the modelling of planned changes such as system modification, acquisition and retirement.

parts of the problem.

STEP 2: Measure Stakeholder Satisfaction. The measurement of stakeholder satisfaction comprises the following activities:

1. Simulation of the *Capability Dynamics Model* is used to describe how capability has developed to date and how it is likely to develop in the future.
2. The satisfaction of stakeholder requirements and how it is likely to develop with time is determined by comparing the requirements developed in Step 1 with the capability profiles produced by the simulation described above.
3. If stakeholder requirements are satisfied, Step 1 is repeated. If not the process moves on to Step 3.

Note that if scenarios are being considered, then the above analysis will need to be conducted within the context of selected scenarios emerging in various combinations and at various times.

STEP 3: Identify Options for Change. The aim of this step is to identify options that can be used to improve the satisfaction of requirements and will include the development of capability through the acquisition, modification and/or assembly of systems. The techniques used to identify such options are beyond the scope of *Capability Dynamics*, but will include many traditional systems and requirements engineering approaches, system dynamics, work flow analysis, softsystems methods and so on.

Details of each option, including the proposed evolution of systems, requirements and capabilities, are described within the organisation's *Capability Dynamics Model*.

STEP 4: Evaluate and Select Preferred Option. The aim of this step is to select a preferred option from those identified in Step 3 by conducting the following activities.

1. A simulation of the organisation's *Capability Dynamics Model* is run for each of the options identified in Step 3. The simulations will describe the behavior of the entire organistion in response to each option.
2. The results of these simulations are used to analyse the impact and risk associated with each option within a *whole of organisation, capability and life* context.
3. Based on the above analysis, consensus is developed amongst all stakeholders as to a preferred option.

4. Details of the preferred option are entered into the organisation's *Capability Dynamics Model* as a plan to be implemented during Step 5. This plan can then be taken into account by Decision Control Processes operating in other parts of the organisation.

If scenarios are being considered, then the options for dealing with each scenario will need to be investigated in the context of selected scenarios emerging in various combinations and at various times.

STEP 5: Implement Preferred Option. The final step of the proposed process is to implement the preferred option. This may be carried out by applying the entire Decision Control Process to each of the systems that comprise the preferred option or, when appropriate, by the application of traditional systems engineering approaches.

During implementation of the preferred option, the Capability Development Models of each of its primitive sub-systems are updated to reflect actual development of capability. This information can then be taken into account by Decision Control Processes operating in other parts of t he organisation.

H.5.1 Iteration

While the above describes the proposed Decision Control Process as a set of sequential steps, the authors expect that in practice the process will be more flexible involving a great deal of iteration amongst the various steps.

H.6 Proposed Experimentation

The proposed experimentation comprises the development of an experimental Distributed *Capability Dynamics Modelling Tool* and its use in two experiments.

H.6.1 Experimental Distributed Capability Dynamics Modelling Tool

A software tool is being developed by the authors to support the construction, maintenance and use of *Capability Dynamics Models*. The tool is based on *Armidale* technology [*Flint and Boughton*, 2002] and comprises a centralised model repository, an application server and lightweight client *application browsers* as depicted in Figure H.3.

The client application browsers currently run on a variety of platforms, including standard Windows and Linux PCs, and will allow the user to access, via the internet or an intranet, various applications including those related to the development and use of a centralised *Capability Dynamics Model* repository.

The major functionality of the tool has been described earlier in this paper. In addition, the tool will include instrumentation to provide data in support of the two experiments described below.

It is intended that the proposed tool be used throughout and at all levels of an organisation in much the same way as a word processor and email. That is, it is hoped that the tool will become a part of everyday life in the work place.

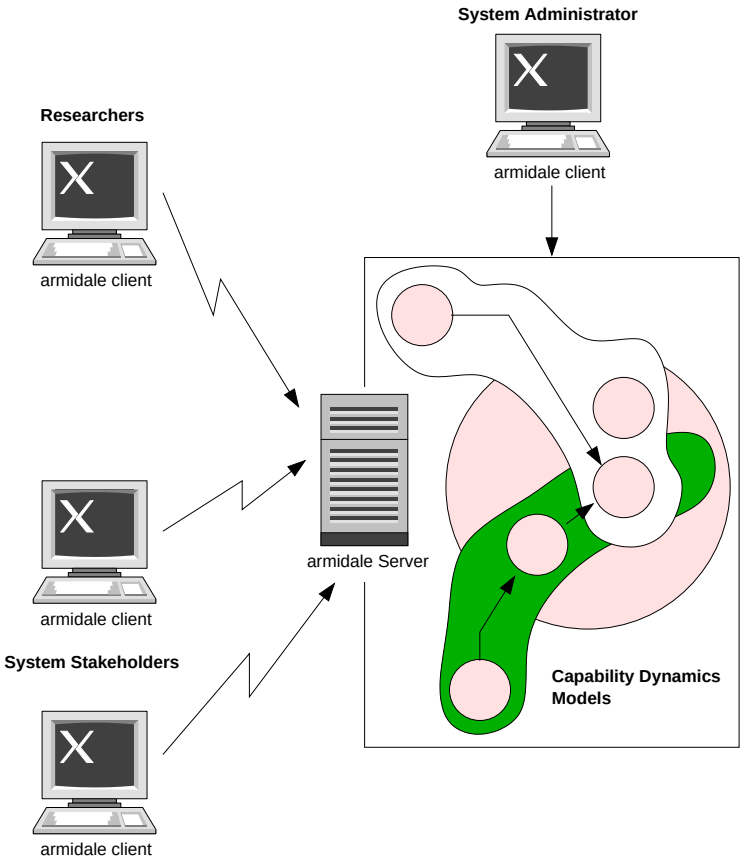

Figure H.6: Experimental software

H.6.2 Experiment One: Internet Based Virtual organisation

The first experiment is intended to show that the approach presented in this paper is practical and effective.

The basic idea is to use the proposed software tool to build a 'virtual'

organisation that undertakes Systems Engineering projects. Practices and other information detailed in the Capability Maturity Model - Integrated for Systems Engineering/Software Engineering (CMMI-SE/SW) [*CMMI Product Development Team*, 2000] will be used as a basis for identifying the systems comprising such an organisation, the requirements placed on them (e.g., requirement to manage configuration), and how they interact to develop systems.

Once an initial model is built, we plan to make it available on a server which can be accessed from anywhere around the world using a armidale client downloadable via the internet. We will encourage interested parties around the world to act as stakeholders in various aspects of the virtual organisation and to make decisions in response to changes elsewhere in the organisation and its environment.

This experiment will:

1. Indicate the effectiveness and practicality (or not) of the *Capability Dynamics* approach.
2. Provide a platform for testing and further development of the proposed software tool.
3. Identify any problems that could be corrected before deploying the approach in a real case study.

H.6.3 Experiment Two: Case Study

The aim of the second experiment is to evaluate the effectiveness and practicality of *Capability Dynamics* within a real organisation. While we do not have any particular organisation in mind, it will likely be a software development or acquisition organisation.

The basic approach will be to apply the Technology Acceptance Model (TAM) [*Davis*, 1989] to assess the acceptance of approaches currently used by an organisation to build or acquire software. *Capability Dynamics* will then be introduced to the organisation along with some training. The organisation will be encouraged to use the approach for some time before the TAM is again used to assess the acceptance of *Capability Dynamics* through the indicators of perceived usefulness and perceived ease of use.

H.7 Conclusions

This paper has described how ideas and techniques associated with system dynamics, agent based simulation, control systems and distributed systems can be integrated to form an approach that may improve the likelihood that decisions made within an organisation provide the most effective contribution possible to the satisfaction of organisational and stakeholder requirements within a whole-of-capability, whole-of-life context.

The authors believe that systems thinking at the level of requirements and capability along with the ability to put sophisticated collaborative modelling and simulation tools on the desktop are key enablers of *Capability Dynamics*.

The approach may be considered ambitious and could prove difficult to implement and take sometime before conclusive results emerge. It is however considered worth pursuing and evaluating because of the potential for substantial gain. If the approach works, large organisations will be able to make more effective decisions with a whole of capability, and life perspective leading to efficiencies and cost savings not previously realised.

Appendix

I

Reproduced paper: *Simplified Development Of Web Applications With Armidale*

Contents

I.1 Background 338
I.2 Requirements, Constraints and Design Approaches 340
I.2.1 Simplified Application Development 340
I.2.2 Rich Graphical User Interfaces 342
I.2.3 Starting Applications 342
I.2.4 Platform Independence 344
I.2.5 Access to third party technologies. 345
I.2.6 Extensibility 345
I.2.7 Armidale Application Launcher Platforms 347
I.2.8 Use of Network Bandwidth 347
I.2.9 Open Source 349
I.3 Implementation 349
I.4 Functional Testing 349
I.5 Effectiveness 350
I.6 Some Design Details 351
I.6.1 Operation in the Stand-Alone Context 351
I.6.2 Operation in the Client-Server Context 352
I.6.3 Making the connection 352
I.6.4 Object Creation 354
I.6.5 Changing Object Attributes 354
I.6.6 Callbacks and Events 354

I.7 Summary 355

Simplified Development Of Web Applications With Armidale

Shayne R. Flint
Department of Computer Science
Australian National University
Tel: 02-6125-8183
Email: shayne.flint@anu.edu.au

Clive V. Boughton
Department of Computer Science
Australian National University
Tel: 02-6125-5689
Email: clive.boughton@anu.edu.au

This paper was published in
Proceedings of the 2002 AusWeb conference
Sunshine Coast, Australia.

The World Wide Web is increasingly being used, across all sectors of the Information Technology industry, to host distributed interactive applications. Many of these applications are built using a combination of different, and sometimes inappropriate, technologies. Often the final products are complex, and the associated likelihood of increased development, integration, deployment and maintenance problems lead to high overall lifecycle costs.

This paper outlines the requirements, design, implementation and testing of armidale, a set of open source programs and libraries designed to radically simplify the development, deployment and use of web applications that have rich graphical user interfaces. Armidale applications are developed, using conventional programming techniques and the armidale API, as if they were stand-alone programs. These programs can then be run on stand-alone computers or on an armidale server. When running on a server, armidale applications display their GUI on client computers across the internet (or intranet). A high level, light weight message protocol between the server and its clients ensures that armidale applications respond well to user interaction.

I.1 Background

Capability Dynamics. The work reported in this paper is part of a larger research effort to develop a technique which aims to increase the likelihood that decisions made throughout, and at all levels of, an organisation contribute in a coordinated way to satisfying the organisation's requirements and those of its stakeholders [*Flint*, 2001]. The approach, called Capability Dynamics, is based on the use of Capability Dynamics Models [*Flint*, 1999] which are developed and maintained by the organisation to provide decision-makers with an organisation wide, shared and evolving understanding of the systems comprising the organisation and its environment, how the capabilities of these systems interact to satisfy stakeholder requirements, and how all these things are likely to evolve with time and in response to change.

The development of these Capability Dynamics Models is likely to require significant effort. In addition, the authors believe that a sense of ownership in their development and use will be important to the success of Capability Dynamics. It was therefore proposed [*Flint*, 2001] that Capability Dynamics Models be developed by collaboration amongst all stakeholders within an organisation, as well as researchers and consultants.

Tool Support. In order to support the collaborative development of Capability Dynamics Models, the authors are developing a computer based Distributed Capability Dynamics Modelling Tool which is intended for day-to-day use throughout and at all levels of an organisation. The tool has been designed to allow everyone within an organisation to capture their own view of those systems with which they interact or are a part, and to run agent based simulations to describe the way in which capability develops over time, and in response to actual, proposed and planned changes. In addition, the proposed tool will provide facilities that allow stakeholders to discuss differences in opinion, identify those differences that are important and, if required, resolve them.

It is hoped that this collaborative approach, supported by computer based tools, will result in the emergence of Capability Dynamics Models for entire organisations formed from the knowledge and individual contributions of their people.

An Internet/Intranet based solution. The distributed and collaborative nature of the tool support discussed above suggests that a web based implementation of the Distributed Capability Dynamics Modelling Tool would be appropriate. This

provided the motivation that led the authors to look at current web application development approaches and to then develop armidale.

Existing approaches to web development. Popular approaches to the development of web applications use various combinations of well established technologies including HTTP, HTML, client side scripts such as JavaScript [*Mozilla.org*, 2004a] and server side scripts such as PHP [*The PHP Group*, 2006]. It is also common to use server side technologies such as *Java Server Pages* [*Sun Microsystems*, 2006b] and *Active Server pages* [*Microsoft*, 2006a] to generate HTML and client side scripts on-the-fly in response to HTTP requests from clients.

More recently, a number of newer technologies have emerged, including the *XML-based User Interface Language* [*Mozilla.org*, 2004b], which aims to simplify the development of client side GUIs, and *Microsoft .NET* [*Microsoft*, 2006b] which supports the development of web services using the *Simple Object Access Protocol* [*World Wide Web Consortium*, 2004]. As is often the case, these technologies build upon the growing mountain of HTTP, HTML and scripting technologies described above.

Difficulties with Existing approaches to web development. In order to develop effective web applications, developers need to master many of the above technologies within the context of individual projects. In addition, developers need to deal with complex issues of compatibility between different versions of the technologies, and in particular, the compatibility of different web browsers with the various scripting languages and variations of HTML used throughout the internet. To a lessor extent, some of these issues also need to be understood by end users.

The authors believe that the need to master this ever-increasing set of technologies and their interaction is likely to increase development, integration, deployment and maintenance problems associated with the life cycle of web applications. In addition, the authors find it strange that the use of such inherently complex and poorly engineered approaches is rarely questioned and that 'new' ideas and approaches often add even more complexity to what appears to be an unsustainable foundation for the development of increasingly complex web applications.

The need for a simple open source solution. In light of the above issues, the authors believe that development of the Distributed Capability Dynamics Modelling Tool could benefit from a radically simplified web application development approach. The only real ongoing efforts to offer such an approach are those by

Bullant [*Gravana Pty Ltd*, 2005] and Droplets [*Droplets, Inc.*, 2005]. Bullant is an Australian company that has developed a set of proprietary technologies which enable the development of sophisticated and highly scalable web applications by eliminating what they call the 'infrastructure zoo' of HTML and associated add-ons. The *Droplets* technology, which was recently introduced to the authors, is another commercial web application development technology that eliminates the use of HTML and associated web technologies.

Because of their commercial nature, Bullant and Droplets are unattractive to our ongoing Capability Dynamics research. This view has strongly influenced the design, implementation and testing of the open source armidale system described in the remainder of this paper. It is hoped that other projects can also make use of the armidale technology.

I.2 Requirements, Constraints and Design Approaches

Initial requirements for Armidale emerged from the wider research desrcibed in this thesis and evolved in an iterative fashion as the system has been developed and used. While additional requirements are likely to emerge during the open source development process proposed for the future, the requirements, as they were at the time of writing, are outlined below along with details of the design approaches used to satisfy them.

I.2.1 Simplified Application Development

Requirement. Armidale shall provide a simple programming model to support the development of interactive applications that are able to run on stand-alone computers or on a server. When running on stand-alone computers, application GUIs shall be displayed on the host computer. Armidale applications installed on a server shall be started by clients connected to the server via the internet or an intranet. In this mode of operation, application GUIs shall be displayed on the connected client computer.

Design. The development of Armidale applications does not involve the use of HTTP, HTML or scripting languages of any kind. Instead, Armidale applications are written in Java, using conventional programming techniques and the Armidale API, as if they were stand-alone programs running in isolation from the internet. That is, the Armidale API abstracts the complexities of the internet and platform dependencies away from the concerns of developers as depicted in Figure I.1.

Developers using Armidale need only learn one programming language (Java), and a simple user interface API. There is no need to learn and become experienced in multiple Web technologies such as HTML, PHP and JavaScript etc. There is no need to understand the way in which these technologies interact and perform on various platforms and in various combinations of versions.

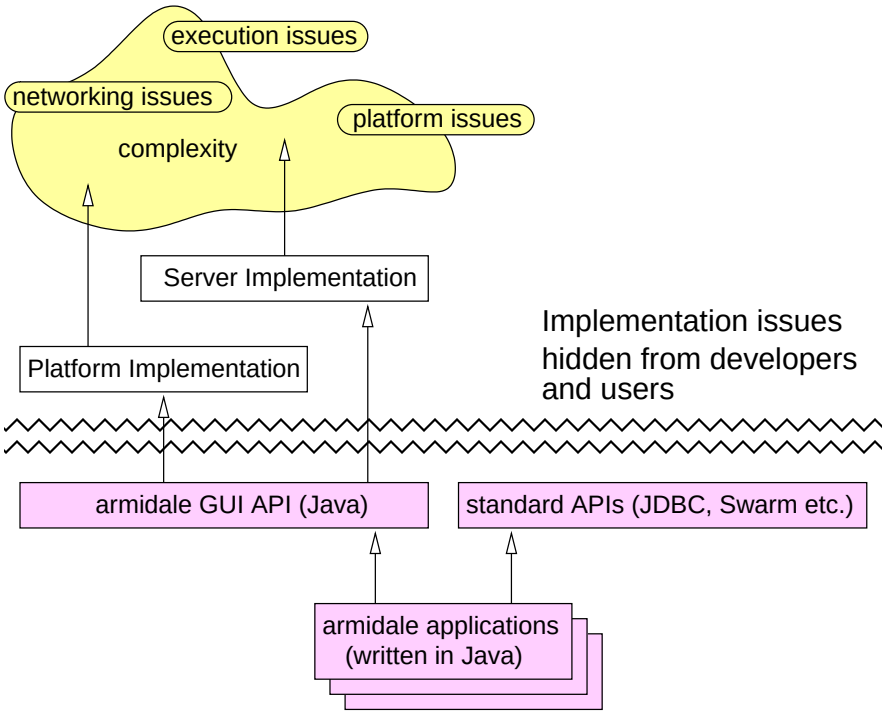

Figure I.1: The Armidale conceptual design

When an Armidale application is started, it is given a context. This context is used by Armidale API factories [*Gamma et al.*, 1995] to determine which implementation of the API should be used for the current execution environment. When an application is started within a platform (stand-alone) context, the factories will return implementations that use a platform API such as Java Swing [*Sun Microsystems*, 2006c] to create and manipulate GUI objects on the host computer. When an application is started within a client-server context, the factories will return implementations that communicate with connected clients via the Open Binary Message Protocol (OBMP) described elsewhere in this paper. On the client, platform context implementations are used to display GUI objects so that applications running in a client-server context have the same look and feel as they have when running in a stand-alone context. Appendix A provides more details on the operation of Armidale applications in stand-alone and client-server contexts.

I.2.2 Rich Graphical User Interfaces

Requirement. Application developers shall be able to use Armidale to create interactive applications with rich Graphical User Interfaces (GUIs) comprising elements such as buttons, images, lists, tree views, tables etc. Such applications shall have the same look and feel as conventional applications available on user computers.

Design. The Armidale API provides mechanisms to create and manipulate the GUI objects listed in Appendix B. In general, the behavior of these objects follows that of corresponding Swing widgets [*Sun Microsystems*, 2006c]. Platform context implementations of the Armidale API make use of platform specific APIs such as Java Swing. The look and feel of GUI objects, is therefore similar to that of applications native to the host platform. Because Platform Context implementations are also used to display GUIs on clients connected to Armidale applications running on servers, the look and feel of Armidale applications is independent of where the application is actually running. Figure I.2 shows a number of Armidale test applications running on a KDE desktop. The applications shown could be running locally or on remote servers over the internet as it makes no difference to their look and feel on the desktop.

I.2.3 Starting Applications

Requirement. Users shall be able to start Armidale applications in the same way as native applications available on the user's computer. In addition, users shall be able to start an Armidale application from a launcher by entering its URL or by referring to a list of bookmarked applications.

Design. Armidale applications, whether installed locally or on remote servers, can be started using any of following methods, all of which create an appropriate context (local platform or server) and then run the application within that context. As far as the user is concerned, Armidale applications start, look and feel like any other application. The user receives no visible indication of where applications are actually running.

One way to start an application is to use the Armidale application launcher. The launcher, which is itself an Armidale application, works in a similar way to a normal web browser. The URL of an application is entered into a text box and a button or menu item is used to start the named application. The format of an Armidale URL allows the specification of local or remote applications and a

Figure I.2: Some Armidale applications running on a KDE desktop

bookmarking facility is provided as depicted in Figure I.3. More details can be found at the Armidale web site [*Flint*, 2006].

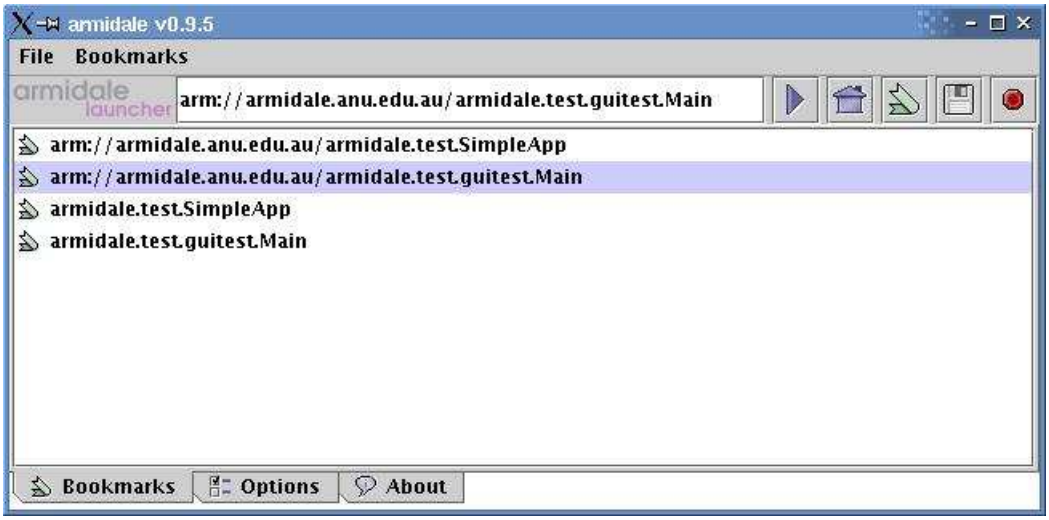

Figure I.3: The Armidale Launcher

Armidale applications can be started using a command line. Locally installed applications are started in the same way as any other Java program. Remote applications can be started by running the armidale.api.RemoteApplication class and passing a URL to it as a parameter. Desktop icons that reference one of the command lines described above can be used to start local or remote Armidale applications. Finally, Armidale applications can be started programmatically by using the armidale.api.RemoteApplication and armidale.api.LocalApplication classes in any Java application. By using these classes to start applications in response to button, menu and image action (i.e. click) events, a set of hyperlinked applications that behave in a similar way to hyperlinked web pages can be established. Users could then browse or 'surf' such applications around the internet.

I.2.4 Platform Independence

Design Constraint. Armidale applications shall be capable of running stand-alone and as server applications on computers running Linux (Intel), Solaris (Sparc), Mac OS X (Power PC) and Windows 2000 (Intel).

Design. The Armidale system has been written in standard Java 2 using the Sun Microsystems JDK 1.4.0 (standard edition) and has been designed to run on any JDK 1.3 or later platform. Successful testing has been conducted on Linux (SuSE

and RedHat), Solaris 8, Mac OS X and Windows 2000.

I.2.5 Access to third party technologies.

Requirement. Armidale shall not restrict the use, by application developers, of third party technologies such as SQL databases and the SWARM agent based simulation system [*Swarm Development Group*, 2006].

Design. Armidale applications are developed using standard Java and as such can make use of any available Java API. This includes the JDBC API which can be used to access SQL databases, and the Swarm Java API. The only exception is that user interface APIs such as AWT and Swing should not be used because Armidale provides its own user interface API which makes it possible to run Armidale applications locally or over the internet or an intranet.

I.2.6 Extensibility

Requirement. Armidale application developers shall be able to add new GUI objects to the Armidale system. The implementation of this requirement shall ensure that objects added by one developer are uniquely identified and that they do not interfere with objects created by other developers.

Design. The standard Armidale API includes mechanisms to create and manipulate many standard GUI objects such as buttons, lists, tables and icons etc. Each of these GUI objects are implemented by a number Java classes and interfaces. For example the PushButton object is implemented by the following Java interface and classes:

```
interface armidale.api.gui.PushButton
class     armidale.api.gui.PushButtonFactory
class     armidale.api.gui.impl
            .clientserver.PushButtonClientImpl
class     armidale.api.gui.impl
            .clientserver.PushButtonServerImpl
class     armidale.api.gui.impl.platform
            .<PLATFORM>.PushButtonImpl

(where PLATFORM is a platform name such as swing)
```

All of these interfaces and classes, with the exception of the platform implementation classes such as `armidale.api.gui.impl.platform.swing.PushButtonImpl`, are completely generated by the armidale.utilities.makeclass.Main program based on specifications provided in XML files as depicted in Figure I.4. An example of an XML GUI object specification is provided at Appendix D to this paper.

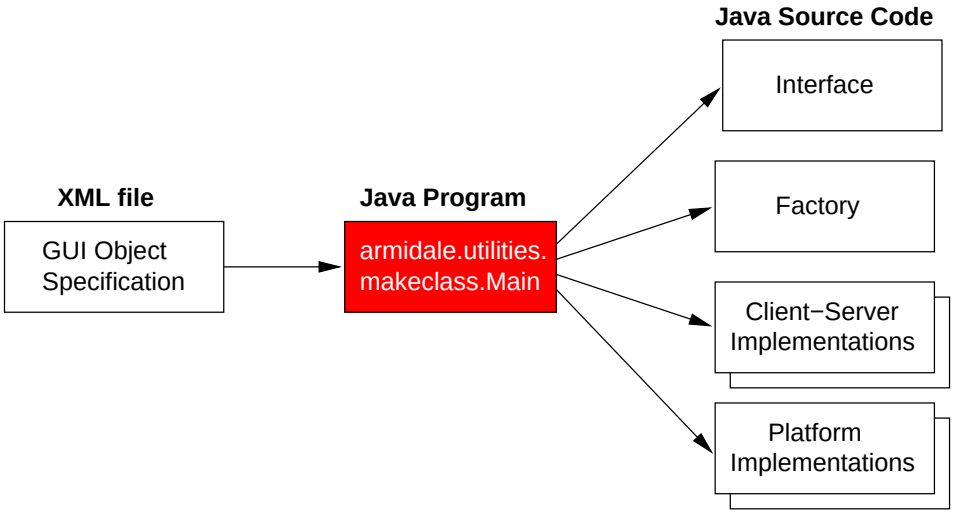

Figure I.4: Generation of Armidale GUI element source code

The platform implementation classes for each GUI object, such as `armidale.api.gui.impl.platform.swing.PushButtonImpl`, are generated by the armidale.utilities.makeclass.Main program with empty method bodies. It is up to the GUI object developer to implement the bodies of these methods (e.g. the swing implementation of the object). This often involves no more than making simple calls to platform objects such as those provided by Swing. An important feature of Armidale is that the above process can be used by application developers to add new GUI Objects to the system. All that is required is an XML specification of the new GUI object and implementations of the platform specific methods described above. Once a new GUI object has been developed, it can be installed on the client and server computers by copying its class files, or a JAR file containing the class files, to a directory on the user's Java class path. The new GUI objects can then be used by application developers in the same way as the standard objects provided in the Armidale distribution. Note that the Armidale API provides a mechanism for applications to check that any non-standard GUI objects required by the application are correctly installed on a client computer before the application starts (see the description at Appendix A). In addition, all GUI objects have a unique Class ID. These Class IDs are 32 bit numbers and will be allocated by the authors in blocks of 100 to interested developers.

I.2.7 Armidale Application Launcher Platforms

Design Constraint. The design of Armidale shall make provision for the use of devices such as mobile phones and PDAs as Armidale clients.

Design. The armidale.utilities.makeclass.Main program described above is able to generate platform implementation classes (with empty method bodies) for any number of client platforms. At present, only the Swing platform has been fully implemented, but all of the infrastructure is in place to implement other platforms such as Espial. Once a platform API has been implemented for a new device such as a phone or PDA, the Armidale application launcher can be used on the new device to start Armidale applications running on Armidale servers.

I.2.8 Use of Network Bandwidth

Performance Requirement. The Armidale system shall minimise the use of network bandwidth. An important motivation for this requirement is that devices such as phones and PDAs may, in future, be used as Armidale clients connected to servers via low bandwidth wireless links.

Design. The design of Armidale makes use of the following techniques to minimise the use of network bandwidth.

I.2.8.1 The Open Binary Message Protocol

The Armidale system uses TCP/IP to pass messages between clients and servers. In order to minimise bandwidth usage, Armidale messages use a simple binary format called the Open Binary Message Protocol (OBMP). Each OBMP message comprises a 32 bit message length and a set of data items including primitive type values such as int, byte, float, double etc. , and more complex types like String, Color and Font. Armidale includes an API for constructing and interpreting OBMP messages. Apart from the minimal use of bandwidth, there are two other reasons for choosing a binary protocol rather than one of the more popular XML protocols such as SOAP [*World Wide Web Consortium*, 2004]. Firstly, our messages are abstracted away from any programmer or user involvement because they are created, transmitted and interpreted in code generated by the armidale.utilities.makeclass.Main program from high level specifications. So the only real requirements on the message protocol are that it be able to transmit the required information and that it be as efficient as possible. Secondly, despite

a common belief to the contrary, XML is nothing more than a human readable format for describing data. If data contained in any kind of message is to be of value, the same meaning must be attached to it by its creator and interpreter. This applies to XML messages as much as it does to any other form of message including binary. XML does not magically add meaning to messages and does not change the fundamentals of communications. So, because Armidale messages are unlikely ever to be created or read by humans there is little reason to use XML.

I.2.8.2 Image handling

When an application creates an Image object it initially has no associated image data and is displayed as a ”broken” picture icon. The application can then set the image data by calling the setImageData() or setFile() methods. When the setImageData() method is called, image data is passed as a parameter and used to create a platform specific image. The implication of this is that in a client-server context, the image data is sent over the network each time the setImageData() method is called. A more efficient approach is to use the setFile() method which takes the name of a file in the Armidale filesystem. This filesystem is simply a directory within the user's home directory called .Armidale/ filesystem and is created during the installation process. When the setFile() method is used in a stand-alone context, the contents of the file are used to create a platform specific image. In a client-server context, the name of the file and its date-time stamp are sent to the client. When the client receives the message, it attempts to open a file with the same name in its local Armidale filesystem. If the file exists on the client and has the same date-time stamp, a platform specific image is created from the contents of the local file. If the required file does not exist in the client's filesystem, or the date-time stamp is incorrect, then the client will request a copy of the file from the server. The server will then send the file to the client, and it will be saved on the client's Armidale filesystem. The contents of the file are then used to create a platform specific image. The above approach implements a simple, but effective image cache and can be applied to other media types in future.

I.2.8.3 Handing large data structures

Armidale is able to efficiently display information from very large data structures or models in GUI lists and tables on the client. The technique ensures that only those items that are visible and are needed for smooth scrolling of lists and tables are passed from the server to the client. As a list or table is scrolled by the user, items that are not yet available on the client are displayed as ”please wait...”. When the user stops scrolling a request is sent to the server for any items that need to

be displayed. When the items arrive at the client the ”please wait...” messages are replaced with the real data.

I.2.9 Open Source

Design Constraint. Armidale shall be developed as an open source project.

Design. Early versions of Armidale were designed and implemented by the authors with the view of making the project open source. The entire system was developed using technologies that would allow Armidale to become an open source project. Armidale has since been released as an open source project. The project is hosted by SourceForge.org, a web site that provides a suite of tools which coordinate the development, distribution, maintenance and use of open source projects, including file transfer services, problem reporting and tracking, support request tracking, and discussion forums etc. It is hoped that this open source approach will encourage wider use of Armidale as well as its continued development and support.

I.3 Implementation

The implementation of Armidale has been uneventful. The tools used included SuSE Linux, the Sun Microsystems Java Development Kit Standard Edition (version 1.3 and 1.4), the Apache Ant java build tool, the JEdit integrated development environment and various Linux tools such as xfig and Gimp. There were no project delays caused by limitations or problems with the tools used.

I.4 Functional Testing

Functionality of the Armidale system was tested using two test applications. The first, SimpleApp, was intended only to test the basic Armidale infrastructure. This covered the Armidale API architecture, the Armidale application launcher, the ability to run applications in different contexts, and the ability to manipulate GUI objects and to process their events correctly. The annotated source code for SimpleApp is at Appendix C to this paper. The second test application, GuiTest, was designed to test each of the Armidale GUI objects. As GUI objects were implemented, they were added to this test application and as more are developed in the future, they too will be added to GuiTest. A design feature of GuiTest is

Table I.1: Armidale Test Configurations

Hardware	OS	OS Version	Java version
Intel PC	Linux	SuSE 7.3	1.3, 1.4
Intel PC	Linux	RedHat 7.2	1.3, 1.4
Sun Sparc	Solaris	8	1.3
Macintosh G4 Titanium Notebook	MacOS X	10.1.3	1.3
Intel PC	Windows	2000	1.3, 1.4

that individual GUI object tests are implemented in separate classes which can be executed in isolation from the main GuiTest application. The screen shot depicted in Figure 1 shows the SimpleApp and GuiTest applications along with a number of individual widget test programs. Table I.1 shows the platforms, operating systems and Java versions on which the above applications were tested. While Java 1.4 is the preferred platform, Armidale should run on any Java 1.3 or 1.4 platform.

I.5 Effectiveness

While the design and conduct of credible software engineering experimentation is considered important (Fenton 1994), none has been conducted to test the view that Armidale simplifies web application development. Such experimentation was beyond the scope and resources of the Armidale project at the time of writing. Nonetheless, a number of characteristics support the view that Armidale simplifies the development and use of web applications. The most important of these is the way in which all aspects of the internet, hardware platform, and operating system are abstracted away from both the application developer and user. When using more traditional web application development approaches, the developer will usually deal concurrently with low level details of the internet including HTML, an array of client side and server side scripting languages, browsers and the many compatibility issues that go with them. Likewise, users of traditional web applications also need to deal with a wide range of compatibility issues and limited capabilities of the web browser model. All of these issues are completely irrelevant to the developer and user of Armidale applications, or are abstracted away from the developer behind the Armidale API. When we also consider the use of a single traditional object oriented programming language, and the ease with which Armidale can be extended, the claim that Armidale enables simplified development of web applications would appear reasonably sound. From a qualitative perspective, the authors experience in developing demonstration and test programs would also support the view that Armidale does indeed simplify the whole process of

developing, testing, deploying and using web applications. The authors believe that the reader will develop a similar view by walking-through the sample application code at Appendix C, installing the Armidale development system available from SourceForge.org and going through the process of compiling, deploying and using the various demonstration programs included in the distribution.

I.6 Some Design Details

Armidale applications are based on Java classes that extend the armidale.api.Application class and provide an implementation for the inherited init() method. This init() method is effectively the ”main” method of an Armidale application. User interfaces are created programmatically using the armidale.api.gui. API. This API provides a set of GUI object factories (Gamma 1995) that create instances of classes which implement specific GUI object interfaces. For example, the armidale.api.gui.PushButtonFactory creates instances of classes that implement the armidale.api.gui.PushButton interface. When an Armidale application is started, it is given a context. This context is used by the Armidale API factories to determine which implementation of the API should be used for the current execution environment. When an application is started within a platform context, the factories will return implementations that use a platform API such as Java Swing [*Sun Microsystems*, 2006c] to create and manipulate GUI elements on the host computer. When an application is started within a server context, the factories will return implementations that communicate with connected clients via the Open Binary Message Protocol (OBMP) described elsewhere in this paper. On the client, platform context implementations of GUI objects are used so that applications running in a client-server context have the same look and feel they have when running in a stand-alone context. The following sections provide more detail about the operation of Armidale applications in stand-alone and client-server contexts. The information provided is best read in conjunction with the annotated source code for the SimpleApp test program at Appendix C.

I.6.1 Operation in the Stand-Alone Context

If an Armidale application is started in a stand-alone environment (i.e. no internet or intranet involved) a Platform Context such as SwingContext will be created and used as depicted in Figure I.5 (also refer to the main() method in the sample code at Appendix C). This context will cause the Armidale API factories to use Platform Implementations of the API to create objects that translate Armidale API method calls directly into corresponding calls to a Platform API (e.g. Swing) thus causing

the GUI of the application to be displayed on the stand-alone device.

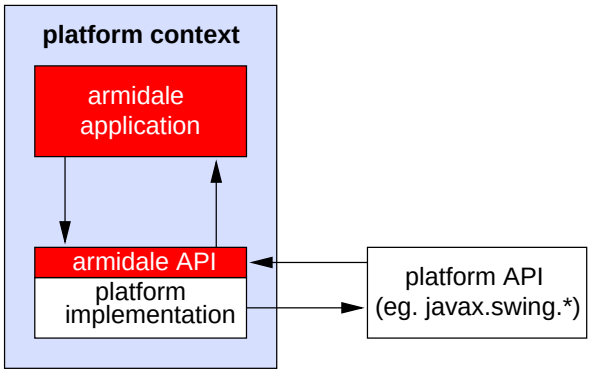

Figure I.5: An Armidale application running in a stand-alone context

I.6.2 Operation in the Client-Server Context

The following paragraphs and the block diagram at Figure I.6 describe how Armidale applications run in the server context. More details, along with the source code for Armidale can be found at SourceForge.org.

I.6.3 Making the connection

The Armidale server (armidale.api.Server) is a very simple Java program. It listens on a specified port for TCP/IP connections from clients. All clients, including the armidale application launcher, make use of the armidale.api.RemoteApplication class to start applications on remote Armidale servers. This class is instantiated with the URL of a remote application and any required arguments. When an instance of this class is created, it sends a CONNECT message to the server containing the name of the application the client wants to run and a list of arguments. This application name is in fact the name of a Java class that extends armidale.api.Application, is installed on the server computer, and is visible to the Armidale server (i.e. on its class path). When a server receives a CONNECT message from a client, it creates a Server Context containing a Server Transport connected to the requesting client. This context is then used to create an instance of the requested application class. If everything is OK, a CLASS_INFORMATION message is sent back to the client specifying any special requirements the application may have (such as non- standard widgets). If a problem occurs on the server (such as a missing application class), an ERROR message is returned to the client. When the client receives the CLASS_INFORMATION message it checks that it is able to satisfy the needs of the remote application (e.g. availability of any special

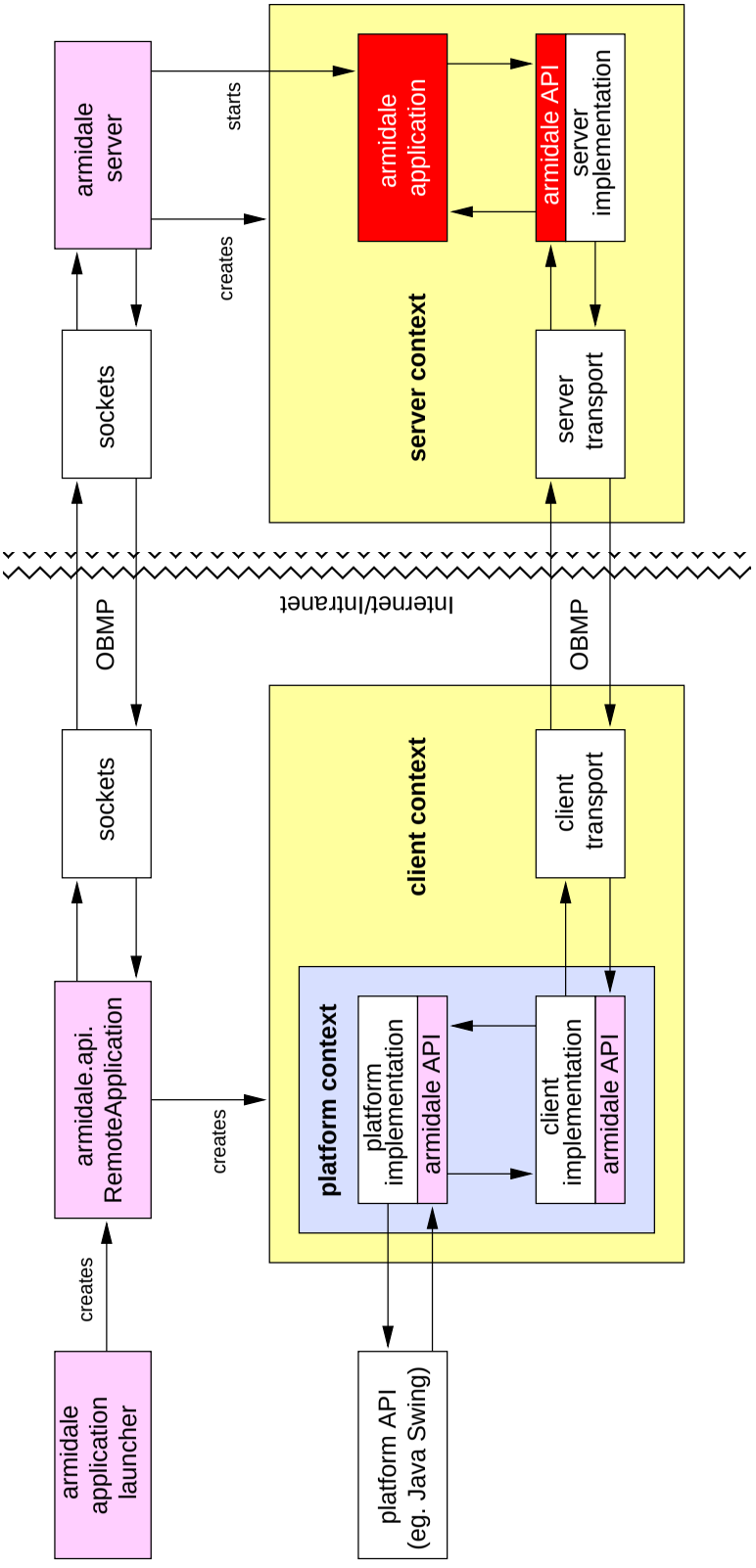

Figure I.6: An Armidale application running in a client-server context

widgets). If all is well, the client will send a CLIENT_OK message to the server and will create a Client Context containing a Platform Context (used to display the GUI on the client) and a Client Transport which waits for and process messages from the connected server. When the server receives the CLIENT_OK message, it calls the start() method of the requested application instance (which is a Java Thread). When the application is started, the init() method is called. The init() method, defined by the application developer, will usually create a user interface (which will be displayed on the client) and set up callbacks. The application's Server Transport will then wait for and processes EVENT messages from the client.

I.6.4 Object Creation

When an application creates a new GUI object such as a PushButton (using the PushButtonFactory), the server implementation of the object sends a CONSTRUCT message to the client. When the client receives this message it uses the applicable factory to create a instance of the required class within the Client Context. These client implementations contain an associated implementation of the Gui object class within a Platform Context such as Swing. These Platform Context objects display the GUI widgets on the client device.

I.6.5 Changing Object Attributes

When an application modifies an attribute of a GUI object, such as the title of a PushButton, the server implementation of the object sends a SET_ATTRIBUTE message to the client. When the client transport receives this message, it dispatches it to the applicable client implementation which in turn calls the appropriate methods in the associated platform implementation.

I.6.6 Callbacks and Events

Many of the GUI objects supported by Armidale are capable of generating events. PushButtons, for example, are able to generate ACTION events. Events are handled in Armidale by callbacks which are implementations of callback interfaces that define methods called when an event is generated by a GUI object. These callback interfaces are implemented by the application programmer to define the required response to events. Instances of these classes are registered with the GUI Object of concern. When a client event occurs (e.g. the user clicks a PushButton), an EVENT message is sent to the server via the Client Transport. When the corresponding Server Transport receives an EVENT message it is dispatched to the

subject object which in turn calls the applicable method of each callback registered with it. Event messages are only sent from the client to the server if one or more callbacks are registered with the subject GUI object.

I.7 Summary

This paper has described the requirements, design and implementation of software used to develop interactive internet based applications in support of wider research being undertaken by the authors. Because the resulting software appears to have wide general purpose applicability, it has been released to the public as an open source project in the hope that others may find it useful and that it may be enhanced and matured by a group of interested open source developers.

Appendix

J

Reproduced paper: *eXecutable and Translatable UML and Systems Engineering*

Contents

J.1 Background 359
J.1.1 The Unified modelling Language 360
J.1.2 The Model Driven Architecture (MDA) 360
J.1.3 UML, MDA and Systems Engineering 361
J.2 Executable and Translatable UML: Key Concepts 362
J.2.1 $^{X}_{T}UML$ is a translative method of software development . . 362
J.2.2 Domains 363
J.2.3 Executable domain models 364
J.2.4 Bridges 365
J.2.5 $^{X}_{T}UML$ and the software development lifecycle 368
J.3 $^{X}_{T}UML$ and Systems Engineering 368
J.3.1 Systems Engineering Domain Models 368
J.3.2 Bridges between system domains 371
J.4 The benefits of $^{X}_{T}UML$ 371
J.4.1 Integration of disciplines 371
J.4.2 Intellectual Property Management 372
J.4.3 The adoption of new technology 372
J.5 Further Work 373
J.5.1 Translative approaches to enterprise architecture 373
J.5.2 Capability Dynamics 373
J.6 Conclusions 373

Executable/Translatable UML and Systems Engineering

Shayne R. Flint
Department of Computer Science
Australian National University
Tel: 02-6125-8183
Email: shayne.flint@anu.edu.au

Clive V. Boughton
Department of Computer Science
Australian National University
Tel: 02-6125-5689
Email: clive.boughton@anu.edu.au

This paper was published in
Proceedings of the 2003 Systems Engineering Test and Evaluation conference
Canberra, Australia.

In March 2003 the *Object Management Group* (*OMG*) released an RFP for customisation of the *Unified modelling Language* (UML) to support the modelling of a wide range of systems including software, hardware, people, procedures and facilities within the framework of *OMG*'s *Model Driven Architecture* (MDA). Much of the response to this RFP has focused on how the UML notation might be extended rather than how UML and MDA could be used during the conduct of systems engineering lifecycle process activities. In this paper we describe important concepts underlying the Executable/Translatable UML and how they and more traditional modelling concepts can be used to support the objectives of MDA in the Systems Engineering context.

J.1 Background

Efforts are currently underway within the *Object Management Group* (*OMG*) and associated organisations to enhance the *Unified modelling Language* (UML) and *Model Driven Architecture* (MDA) for use in systems engineering. Much of this effort has been directed towards the UML notation rather than how UML and MDA can be used in support of systems engineering lifecycle processes.

In this paper we provide an overview of UML and MDA followed by a description of the Executable/Translatable UML ($\frac{X}{T}UML$) approach to software

development and how it supports the objectives of MDA. It is then shown how concepts that underpin ${}^{X}_{T}UML$ might support the use of UML and MDA in the systems engineering context. We conclude with a summary of the benefits that ${}^{X}_{T}UML$ may bring to systems engineering, and plans for future work.

J.1.1 The Unified modelling Language

The UML [*Object Management Group*, 2003a] is a notation for specifying, visualising, and documenting models of software systems. It comprises a set of diagrams that can be used to represent models of software structure and behavior. The UML does not prescribe a method for developing software. It is only a notation for representing the models produced by many of the software development methods in use today.

J.1.2 The Model Driven Architecture (MDA)

The *Model Driven Architecture* (MDA) is an *OMG* initiative that aims to improve the productivity of software development, as well as the portability and maintainability of software systems, by exploiting the well established principle of separating the specification of required software operation from its design and implementation. MDA makes use of *Platform Independent Models* (PIM), which capture software requirements completely free of any design or implementation concerns, and *Platform Specific Models* (PSM) which represent software design and implementation. PIMs are transformed into one or more PSMs according to well defined transformation rules. The PSMs produced in this way may then undergo further transformations to more specific PSMs and eventually to source code. Figure J.1 depicts these primary MDA models and associated transformations.

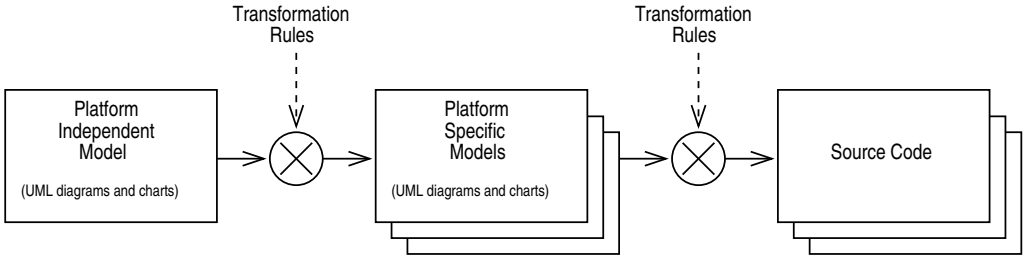

Figure J.1: Primary MDA models and transformation

Productivity is expected to improve by automating the transformation of PIMs to PSMs and then to final code. It is anticipated that portability will be enhanced because PIMs remain unchanged in the face of changing technology. As new

platform technologies emerge they will be modelled as new PSMs and associated transformation rules. Existing, technology neutral, PIMs can then be transformed into these new PSMs leading to the possibility of large scale reuse of corporate intellectual property captured in the PIMs. Maintenance could also be improved because of the clear separation of functional requirements modelled in the PIMs and non-functional requirements modelled in the PSMs, and because PIMs may have extended lifecycles through a series of emerging technologies.

The MDA Guide [*Object Management Group*, 2003b] and [*Frankel*, 2003] provide more complete descriptions of MDA.

J.1.3 UML, MDA and Systems Engineering

In March 2003 the *OMG* released a Request For Proposal [*Object Management Group*, 2003c] for customisation of UML to support the modelling of a wide range of systems including software, hardware, people, procedures and facilities within the framework of *OMG*'s MDA. This effort to bridge the software-hardware-people gap is being conducted by the *Systems Engineering Domain Special Interest Group* [*Object Management Group*, 2003d] of the *OMG* and is supported by a several organisations including the *International Council on Systems Engineering* [2006b].

While a number of organisations are working on the UML specification itself, few are working on how the UML and MDA should be used to support the disciplined conduct of systems engineering lifecycle process activities. As a result, there is a risk of repeating events in the software industry which have led to the widely and falsely held view that UML is much more than just a notation and that its use alone somehow constitutes a disciplined approach to software specification and design.

In order to maximise the benefits of using UML and MDA in the systems engineering context, we believe that methods for using the technology need to be developed along side development of the UML and MDA standards. The Executable/Translatable UML ($\overset{X}{T}UML$), described in the next section, is a well established method for the development of software and while it facilitates an MDA approach to software engineering, it is based on principles more advanced than those underpinning MDA. It is these more advanced principles that may support the practical application of MDA in a systems engineering context.

J.2 Executable and Translatable UML: Key Concepts

In the following subsections we describe key concepts that set ${}^{X}_{T}UML$ apart from other approaches to implementing MDA. In particular, ${}^{X}_{T}UML$ is a translative method of software development which supports the separation of concerns beyond those represented by MDA (separation of PIMs from PSMs). We also show how ${}^{X}_{T}UML$ provides a framework for the systematic development and verification of requirements specifications and design descriptions, and their subsequent translation into deployable software.

J.2.1 ${}^{X}_{T}UML$ is a translative method of software development

Object oriented software development methods can generally be classified as either *elaborative* or *translative*.

J.2.1.1 Elaborative methods

Elaborative methods of software development are most popular and are based on the belief that object orientation can be used to smooth the transition from analysis to design and implementation. An *elaborative* approach begins during analysis by creating object oriented models of software at a high level of abstraction, omitting design and implementation details. During design phases these models are *elaborated* (or refined) to include design and implementation information thus blurring the distinction between requirements specification and design descriptions. Eventually the models become specifications for code.

Elaborative techniques, by definition, do not support complete translation limiting the capabilities and promise of MDA. Most of the object oriented methods and tools available support elaboration but not translation.

J.2.1.2 Translative methods

Translative methods of software development are based on the belief that maintaining a clear separation of concerns throughout the entire software lifecycle improves the understandability, verifiability, portability, large scale reuse and productivity associated with software engineering artifacts. These separated concerns include aspects associated with requirements specifications, software architecture, and implementation.

Models produced during analysis activities specify the required data, state and behavior of separate aspects of a software system independent of implementation. Information from these models is then *translated* and *woven* together with details from separately developed software architecture and implementation models to form code and other software engineering artifacts. These concepts of separating concerns and weaving constitute an important generalisation of the more specific MDA concept of developing a PIM and transforming it into one or more PSMs.

The translative approach to software development was pioneered by Sally Shlaer and Stephen Mellor in the 80's leading to the development of the *Shlaer/Mellor method* [*Shlaer and Mellor*, 1992]. During recent years and with the adoption of some translative ideas by the *OMG* in MDA, the Shlaer/Mellor method has been refined and updated to make use of a well defined subset of UML for representing models which can be executed for verification and simulation purposes. This method is now called Executable/Translatable UML ($\overset{X}{T}UML$) and is described in a number of recent books [*Mellor and Balcer*, 2002; *Starr*, 2002, 2001]. The method is also supported by at least two commercial software engineering tools [*Mentor Graphics*, 2006; *Kennedy Carter*, 2006].

J.2.2 Domains

$\overset{X}{T}UML$ extends the simple concept of PIMs and PSMs by supporting the separate modelling of the various subject matters that comprise a software system. An $\overset{X}{T}UML$ model comprises a set of *domain* models that each capture the concepts of an autonomous subject or aspect of a system such as the application itself, user and other interfaces, databases, security and networking. While some domains may appear to be subsystems, most are not. Most domains specify a particular aspect of a system which may touch all parts of the final software. An example of such a domain is security. A security domain may specify security related concepts such as users, roles, passwords, and privileges, how the concepts relate to each other, and how they respond to various stimuli. Security may affect many parts of an operational software system but nonetheless is modelled and verified independently of any other subject matter. During model translation, subject matter of the application domain, security domain and other domains will be systematically *translated* and *woven* together to form a working software system.

Domains can be categorised as *application*, *intermediate abstractions*, or *implementation*. Application domains deal with software requirements and contain no design or implementation concerns. Implementation domains, on the other hand, deal with design and technology concerns such as software architecture, programming languages and communications. A working software system will

be constructed from the concepts defined in implementation domains. Intermediate abstractions are domains used to decouple an application requiring generic services, such as a database, from the specific technology that provides the service. Some of these domains may be well understood or they may already exist. These *realised* domains are not modelled using $\frac{X}{T}UML$ and usually represent implementation technologies and external systems with which the subject software must interact, including off-the-shelf or legacy software.

Figure J.2 depicts a domain chart showing the software aspects of a simple system. The implementation domains represent various aspects of the popular open source LAMP architecture - Linux (Operating system), Apache (web server), PHP (programming language) and MySQL (database). The Persistence Domain is an intermediate abstraction that decouples application domains from implementation technology (MySQL). The Warehouse and Security domains model application requirements.

J.2.3 Executable domain models

An important feature of $\frac{X}{T}UML$ is that domain models are complete executable models of a given subject matter. This means that domains dealing with requirements are executable specifications which allow for verification of required functionality early in the software lifecycle before any design or implementation technologies are even considered. This emphasis on a set of independently executable domain models is a cornerstone of the $\frac{X}{T}UML$ approach and sets it well apart from the other approaches to MDA.

To facilitate the construction of executable models, $\frac{X}{T}UML$ domains are represented using a well defined and integrated subset of UML comprising class diagrams (no operations), state charts, collaboration diagrams and sequence diagrams together with well defined semantics. Such a subset is necessary because UML, as it is defined in the current UML specification [*Object Management Group*, 2003a], is unnecessarily large and devoid of the precisely defined semantics necessary to produce executable models.

Note that while $\frac{X}{T}UML$ domain models are represented using a small subset of UML and include an asynchronous view of behavior, they place no restrictions on how domain subject matter is translated. For example, domain models can be translated into object-oriented or structured designs, or directly into object oriented, structured or unstructured programming languages. An asynchronous specification can be implemented as a synchronous design.

J.2.4 Bridges

Application domains are usually modelled independently of issues such as security, user interaction and persistence. They rely on *bridges* to other domains to deal with such issues. These bridges are shown as dashed arrows on the domain chart and represent the assumptions made by a client domain and a set of corresponding requirements that are placed on some other server domain.

For example, the arrow between the Warehouse and Graphical User Interface (GUI) domains depicted in Figure J.2 indicates that the Warehouse domain assumes that some other (anonymous) domain will provide a user interface and that in this case the bridge has placed corresponding requirements on the GUI domain.

The consequences of this bridging concept include the ability to replace or supplement server domains without changing any client domains. For example, the GUI domain depicted in Figure J.2 could be replaced, at the level of specification rather than design or implementation, with a domain that models another form of user interface such as could be provided by a mobile phone. Replacing a domain in this way would require changes to bridges from client domains, but not the domains themselves.

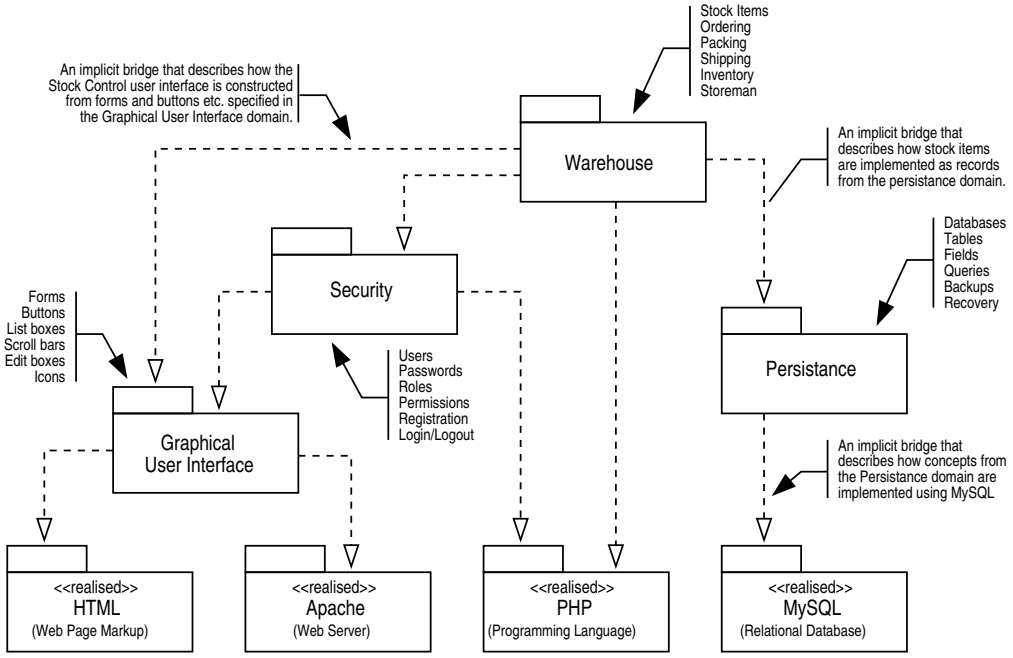

Figure J.2: An Annotated Software Domain Chart

J.2.4.1 Explicit Bridges

Explicit Bridges represent assumptions within the model of one domain concerning the existence of another domain that are explicitly represented as signals to and from external entities and the invocation of operations on such entities. In effect, the model of one domain assumes that some other anonymous domain, linked via an explicit bridge, will generate required signals, consume and act appropriately on signals it is sent and implement correctly any operations invoked on it.

Like domain models, the implementation of links between domains is not constrained by modelling them in terms of signals and operations. Signals, for example, could be implemented as asynchronous events or method calls within a single processor, by passing messages between computers connected via the internet or by any other means that satisfy requirements placed on the domains involved. The choice of implementation approach will often depend on design and architectural constraints.

Explicit bridges are usually used to link application and intermediate abstraction domains to realised domains that represent external systems and legacy software.

J.2.4.2 Implicit Bridges

By far the most common and important bridges in an $\frac{X}{T}UML$ model are *implicit bridges*. Unfortunately they are also the most difficult to understand for those new to $\frac{X}{T}UML$.

In contrast to explicit bridges, *implicit bridges* represent assumptions within the model of one domain concerning the existence of another domain as a set of rules that direct the use of subject matter in the bridged domains to form requirements specifications, design descriptions, implementation and other software engineering artefacts.

Implicit bridging rules can be categorised as *mapping*, *transformation*, or *weaving*. *Mapping* rules are used to state the correspondence between the subject matter of one domain and that of another domain. For example, stock items in the Warehouse domain depicted in Figure J.2 may correspond to text items in a scrollable list box in the Graphical User Interface (GUI) domain. In such a case, the implicit bridge may model a complete user interface for warehouse operations based on concepts specified in the GUI domain. These models may take the form of screen shots or some other diagrammatic representation of the required user interface. The implication of this is that the GUI domain, which only defines

the concepts of a GUI and how the concepts relate and behave, is reusable in other applications that require a GUI. It is the implicit bridge to the GUI domain that specifies how the concepts will be used to satisfy the needs of the Warehouse domain.

Transformation rules involve the *transforming* of subject matter of one domain into subject matter of another domain. For example, stock items in the Warehouse domain may be transformed into records in a relational database table specified in the Persistence domain. Other data from the Warehouse domain could be implemented in other database tables. Relationships between warehouse data would be implemented as foreign keys in the appropriate relational database tables. Note that we have described two different translations of stock items; one as a *mapping* to items in a GUI scrollable list box and another as a *transformation* into records of a database table. In effect the two implicit bridges require the concept of a stock item to end up as a record in a database table, and that information from this database table record be displayed in a GUI list box.

Weaving rules are more complex and involve the principles of aspect orientation [*Elrad et al.*, 2002] where the subject matter of one domain is woven throughout the subject matter of another domain. For example, concepts such as registered users, access rights, roles, passwords and encryption specified in the Security domain depicted in Figure J.2 may be woven throughout the Warehouse domain. The implicit bridge between the two domains is used to specify the actual access rights and roles required by the warehouse and how such details impact its operation.

J.2.4.3 modelling Implicit Bridges

There are no concrete rules in $\frac{X}{T}UML$ regarding the modelling of implicit bridge rules. In fact, it is often assumed that the operation of implicit bridges is embedded within automated translation tools such as those included with commercial $\frac{X}{T}UML$ modelling environments [*Mentor Graphics*, 2006; *Kennedy Carter*, 2006]. However, in practice, automated translation tools are limited to a small range of target platforms and are not particularly good at dealing with implicit bridging rules other than the transformational kind. It is therefore necessary to consider approaches to modelling implicit bridges so that translations can be accurately performed by software engineers.

As a general rule, it has been found that implicit bridges should be modelled using approaches appropriate to the nature of the bridge involved. For example, bridges from application domains to GUI domains might be modelled using pictures of the required user interface screens. Bridges from an application domain

to a database domain may use a table to describe the mapping of data types in an application domain to field types supported by an SQL database. Bridges from an application domain to a programming language domain are likely to make use of design patterns [*Gamma et al.*, 1995] to describe the translation of application subject matter into source code.

J.2.5 $^{X}_{T}UML$ and the software development lifecycle

$^{X}_{T}UML$ domains are identified, modelled and used throughout the software development lifecycle. During analysis, application domains, applicable intermediate abstraction domains and associated bridges are identified and modelled. During design, any other required intermediate abstraction domains are modelled along with all chosen implementation domains and associated bridges. During implementation, domains are translated into deployable software in accordance with rules associated with implicit bridges connecting the domains.

During maintenance, domains and bridges may be added, removed and/or modified to correct defects, add new functionality and to take advantage of new technology.

Implicit bridges should be created and maintained separately from models of the domains involved. That is, domain models capture intellectual property, while the creation and maintenance of bridges puts intellectual property to work.

J.3 $^{X}_{T}UML$ and Systems Engineering

While $^{X}_{T}UML$ is primarily used to develop software systems, the concepts it employs may have wider applicability in areas such as systems engineering and the development of intelligent enterprises [*International Council on Systems Engineering*, 2006a]. So, how can $^{X}_{T}UML$ concepts be used to model systems comprising hardware, software and people? We believe the answer may lie in the systematic development of autonomous domain models and the use of implicit bridging between them.

J.3.1 Systems Engineering Domain Models

The purpose of $^{X}_{T}UML$ domain modelling is to separate specific autonomous subject matter. While this approach has been mainly used in the software development context, we believe it may have applicability within the wider context of systems engineering. For example, domains associated with hardware, maintenance,

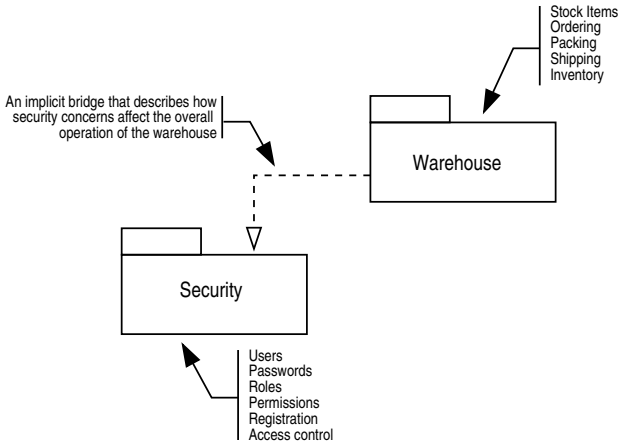

Figure J.3: An Annotated Application Domain Chart

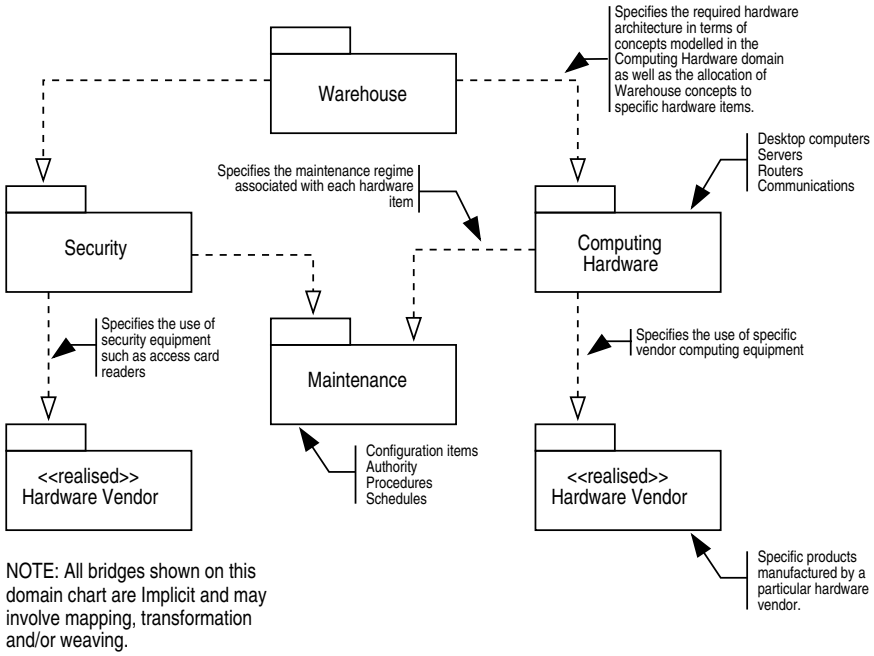

Figure J.4: An Annotated Hardware Domain Chart

process and training and even project management, safety, risk management, test and evaluation could be used.

Because of the large number of domains likely to be involved in a systems engineering effort, we have found that a series of domain chart views, each showing those domains associated with a particular viewpoint, leads to improved clarity and scaleability over the single domain chart used by the $\frac{X}{T}UML$ method. A number of such domain views are depicted in Figures J.2, J.3, J.4 and J.5. Annotations on these figures indicate the nature of each domain and the implicit bridges between them.

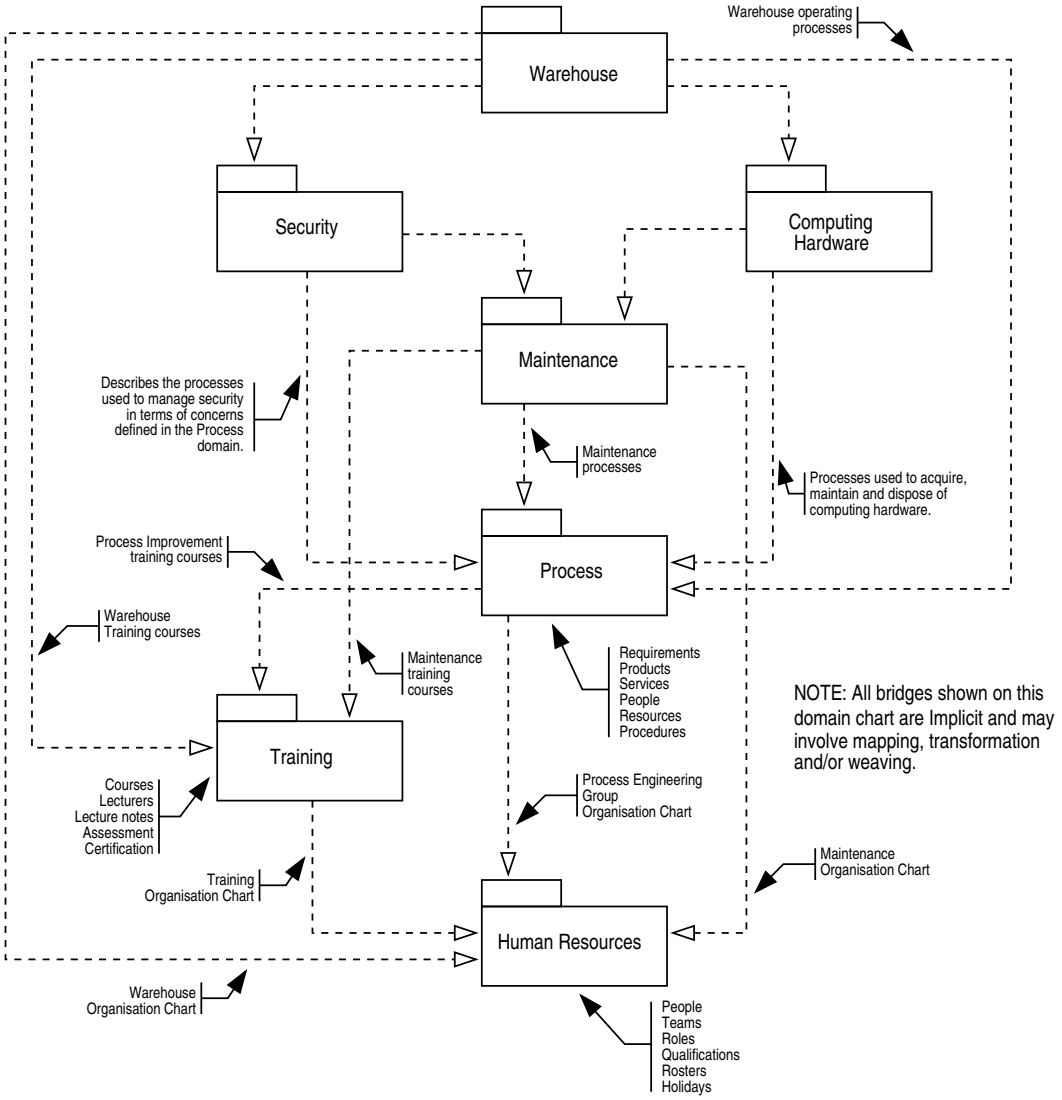

Figure J.5: An Annotated Process, Training and Human Resources Domain Chart

The kind of models developed for each domain need to be considered carefully. In order to ensure the executability of domain models, the $^{X}_{T}UML$ subset of UML could be used. In some cases, however, UML may not be appropriate. For example domains dealing with mechanical and electronic design should be modelled using traditional approaches. In fact, software may sometimes be more effectively modelled using techniques such as Data Flow Diagrams, Entity Relationship Diagrams and Process models.

Obviously, the basic concepts underpinning $^{X}_{T}UML$ and MDA do not require the use of UML, and so the universal use of UML as proposed by the *OMG* may not be required to implement an MDA approach to systems engineering or indeed software engineering.

The key to understanding how this might work lies in the systematic use of implicit bridges between domains.

J.3.2 Bridges between system domains

Bridges between domains in a systems engineering context serve the same purpose as those in a software only context. That is, they comprise assumptions made by a client domain and a set of corresponding requirements that are placed on some other server domain.

In the software context *explicit* bridges are specified in terms of signals and operations which can be implemented in a variety of ways. In the systems engineering context, some explicit bridges will need to be specified in new ways including mechanical interfaces and procedures that can be followed by humans.

Similarly, implicit bridges may need to be specified in new ways including those depicted in Figures J.4 and J.5. For example, the bridge from the Warehouse domain to the Computing Hardware domain depicted in Figure J.4 might use a network model to describe the generic equipment and communications required to host software produced to support warehouse operations. The bridge from Computing Hardware to the Hardware Vendors domain might use a table to identify specific pieces of vendor supplied equipment required to populate the above network of equipment and communications. The bridge from Maintenance to Human Resources depicted in Figure J.5 might use organisation charts to describe the organisation of people, teams and roles required to maintain equipment described by the above mentioned bridges. The bridge from Maintenance to Process might use work flow diagrams and the like to describe maintenance processes.

J.4 The benefits of $\frac{X}{T}UML$

The real value of the $\frac{X}{T}UML$ approach is not so much its support for MDA, but rather it's more general applicability in supporting the integration of disciplines practised in systems engineering, the systematic creation and profitable exploitation of intellectual property as well as the simplified adoption of new technology.

J.4.1 Integration of disciplines

The modelling of autonomous subject matter in separate domains, together with the use of implicit bridges between them, supports the conduct of all systems engineering lifecycle activities within a single framework. Such a framework

facilitates the independent work of domain experts and supports the systematic weaving of their output to construct, maintain and operate a system or indeed a large system of systems. This approach directly supports the *OMG*'s *Systems Engineering Domain Special Interest Group* [*Object Management Group*, 2003c] objectives to bridge the gap between software and hardware engineers as well as integration of the work products of soft sciences such as human resource management, psychology and sociology.

J.4.2 Intellectual Property Management

$\frac{X}{T}UML$ provides direct support for the disciplined and systematic creation, maintenance and exploitation of intellectual property. Domain modelling can be viewed as the creation, verification and maintenance of intellectual property related to a particular subject matter. Domain models will often represent significant corporate knowledge and investment that is independent of changing technology. Intellectual property created in this way can be exploited by assembling a set of domains and associated bridges to form a complete system model which can be transformed into an operational system.

The intellectual property captured in domain models, can be reused to specify and form other systems. For example, a company could develop domain models that deal with the subjects of military situation display, tracking, mission planning, messaging, security and graphical user interfaces. These separately developed domain models could then be bridged implicitly in various configurations and with various implementation domain models (platforms) to form a range (product line) of military command and control systems.

J.4.3 The adoption of new technology

The concept of domains and implicit bridging simplifies the adoption of new technology in existing and new projects. As new technologies emerge, the value of intellectual property captured in many existing domain models is maintained. To use new technology, these existing domains are simply bridged to domains that model such technology.

For example, the *realised* domains depicted in Figure J.2 could be replaced with a completely different software implementation without any need to change the application and intermediate abstraction domains. The MySQL database could be replaced with PostgreSQL and the PHP domain could be replaced with Java without affecting the content of other domains. Similarly, the Computer

and Vendor Hardware domains depicted in Figure J.4 could be replaced with new technology without needing to change the warehouse domain.

Of course, any change in technology is likely to have an impact on requirements in that more or less may be acheivable with new technology. In such cases, application domain models may need to be changed to reflect changing requirements, but would still remain independent of any technology, new or otherwise.

J.5 Further Work

J.5.1 Translative approaches to enterprise architecture

The $\frac{X}{T}UML$ principles described in this paper and the way in which they could be applied to systems engineering, may also be applicable to the design and maintenance of enterprise architectures. This will be the subject of further work by the authors.

J.5.2 Capability Dynamics

[*Flint*, 2001] describes research which has led to the development of *Capability Dynamics*, a novel approach to modelling the dynamics of requirements and capabilities developed and used throughout complex systems and organisations. This ongoing research is now exploring the use of domains and implicit bridging to systematically translate *Capability Dynamics* models into operational systems and organisations that easily adapt to changing environments, objectives and measures of effectiveness.

J.6 Conclusions

In this paper we have outlined moves by the *OMG* to extend UML and MDA for use in the systems engineering context. We noted that this work is mainly concerned with development of the UML and MDA standards rather than how these standards could be applied to systems engineering. Concepts that underpin $\frac{X}{T}UML$ were described and considered in the context of systems engineering. We concluded that the $\frac{X}{T}UML$ concepts of domain modelling and implicit bridging could support an MDA approach to systems engineering without further development of the UML and MDA specifications.

Bibliography

Abdel-Hamid, T., and S. Madnick, *Software Project Dynamics: An Integrated Approach*, Prentice-Hall Software Series, Englewood Cliffs, NJ., 1991.

Ackoff, R., Towards a system of systems concepts, *Management Science*, *17*(11), 661–671, 1971.

Ackoff, R., *Re-Creating the Corporation, A Design of Organisations for the 21st Century*, Oxford University Press, Oxford, 1999.

Adrion, W., Research methodology in software engineering, *Software Engineering Notes*, *18*(1), 36, 1993.

Agile Alliance, Manifesto for agile software development, viewed 30 June 2006, ⟨`http://www.agilealliance.org`⟩, 2001.

Ahern, D., A. Clouse, and R. Turner, *CMMI Distilled*, Addison-Wesley, Boston, MA, 2004.

Apache Software Foundation, The Apache HTTP server project, viewed 30 June 2006, ⟨`http://httpd.apache.org`⟩, 2006.

Australian Department of Defence, *Australia's Strategic Policy*, Australian Government, Canberra, 1997.

Australian Department of Defence, Press release 146/00: Good governance to underpin defence renewal, viewed 30 June 2006, ⟨`http://www.defence.gov.au/media`⟩, 2000.

Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, Pearson Education, Boston, MA, 2003.

Beck, K., *Extreme Programming Explained: Embracing Change*, Addison-Wesley, reading, MA, 1999.

Bell, R., Code generation from object models, *Embedded Systems Programming*, *11*(3), 74–82, 1998.

Bertalanffy, L., General systems theory - a critical review, *General Systems*, *7*, 1–20, 1962.

Boehm, B., A spiral model of software development and enhancement, *Computer*, *21*(5), 61–72, 1988.

Boehm, B., and V. Basili, Software defect reduction top 10 list, *Computer*, *34*(1), 135–137, 2001.

Boehm, B., and R. Turner, *Balancing Agility and Discipline, A guide for the perplexed*, Pearson Education, Boston, MA, 2004.

Boughton, C., Real Agility: Lessons learnt from implementing MDA, in *Proceedings of the Software Education Conference*, Sydney, Australia, 2006.

Boulding, K., General systems theory - the skeleton of science, *Management Science*, *2*(3), 197–208, 1956.

Boyd, J., Patterns of conflict (unpublished), viewed 30 June 2006, ⟨`http://www.d-n-i.net/boyd/pdf/poc.pdf`⟩, 1986.

Brooks, F., No silver bullet: Essence and accidents of software engineering, *Computer*, *20*(4), 10–19, 1987.

Buzan, T., *The Mind Map Book*, BBC Worldwide, London, UK, 2003.

Capell, P., Benefits of improvement efforts, *Tech. Rep. CMU/SEI-2004-SR-010*, Software Engineering Institute, Pittsburgh, PA, 2004.

Casti, J., *Connectivity, complexity, and catastrophe*, J. Wiley, Chichester, England, 1979.

Checkland, P., *Systems Thinking, Systems Practice*, J. Wiley, New York, 1981.

Checkland, P., Soft systems methodology, in *Rational Analysis for a Problematic World*, edited by J. Rosenhead, pp. 71–100, John Wiley & Sons Ltd, Chichester, England, 1989.

Checkland, P., and J. Scholes, *Soft Systems Methodology in Action*, J. Wiley, New York, 1999.

Chen, P., *The entity-relationship approach to logical data base design*, Q.E.D. Information Sciences, Wellesley, MA, 1977.

Churchman, C. W., *Challenge to Reason*, McGraw-Hill, New York, 1968a.

Churchman, C. W., *The Systems Approach*, Dell Publishing Inc., New York, 1968b.

Clarke, S., and E. Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison-Wesley, Upper Saddle River, NJ, 2005.

CMMI Product Development Team, Capability maturity model for systems engineering/software engineering, *Tech. Rep. CMU/SEI-2000-TR-019*, Software Engineering Institute, Pittsburgh, PA, viewed 30 June 2006, ⟨`http://www.sei.cmu.edu/publications/documents/00.reports/00tr019.html`⟩, 2000.

Cockburn, A., *Agile Software Development*, Addison-Wesley, Boston, MA, 2002.

Coram, R., *BOYD: The Fighter Pilot Who Changed The Art of War*, Back Bay Books / Little, Brown and Company, New York, NY, 2002.

Davis, A., and D. Zowghi, Good requirements practices are neither necessary or sufficient, *Requirements Engineering*, *11*(1), 1–3, 2006.

Davis, F., Perceived usefulness, perceived ease of use, and user acceptance of information technology, *MIS Quarterly*, *13*(3), 319–340, 1989.

Demarco, T., *Structured analysis and system specification*, Yourdon Press, Englewood Cliffs, NJ, 1979.

Diaz, M., and J. Sligo, How software process improvement helped Motorola, *IEEE Software*, *14*(5), 75–81, 1997.

Dijkstra, W., E, Ewd 447: On the role of scientific thought, in *Selected writings on Computing: A Personal Perspective*, pp. 60–66, Springer-Verlag, 1982.

Droplets, Inc., Droplets web site, viewed 30 June 2006, ⟨`http://www.droplets.com`⟩, 2005.

Elrad, T., R. Filman, and A. Bader, Aspect-oriented programming: Introduction, *Communications of the ACM*, *44*(10), 29–32, 2002.

Endres, A., and D. Rombach, *A Handbook of software and systems engineering - Empirical observations, laws and theories*, Addison-Wesley, Harlow, Essex, England, 2003.

Fenton, N., S. Pfleeger, and R. Glass, Science and substance: A challenge to software engineers, *IEEE Software*, *11*(4), 86–95, 1994.

Ferguson, P., W. Humphrey, S. Khajenoori, S. Macke, and A. Matvya, Results of applying the personal software process, *Computer*, *30*(5), 24–31, 1997.

Ferguson, P., G. Leman, P. Perini, S. Renner, and G. Seshagiri, Software process improvement works!, *Tech. Rep. CMU/SEI-99-TR-027*, Software Engineering Institute, Pittsburgh, PA, 1999.

Filman, R., T. Elrad, S. Clarke, and M. Aksit, *Aspect-Oriented Software Development*, Pearson Education, Inc., Boston, MA, 2005.

Flint, S., A model which helps control change in dynamically complex purposeful systems, in *Proceedings of the 1999 Systems Engineering Test and Evaluation conference (SETE 1999)*, Adelaide, Australia, 1999.

Flint, S., Capability dynamics - an approach to capability planning in large organisations, in *Proceedings of the 2001 International Council on Systems Engineering Symposium (INCOSE 2001)*, Melbourne, Australia, 2001.

Flint, S., Armidale web site, viewed 30 June 2006, `http://softeng.anu.edu.au/armidale`, 2006.

Flint, S., and C. Boughton, Simplified development of web applications with armidale, in *Proceedings of the Eighth Australian World Wide Web Conference (AUSWEB02)*, Sunshine Coast, Australia, 2002.

Flint, S., and C. Boughton, Executable/Translatable UML and Systems Engineering, in *Proceedings of the 2003 Systems Engineering Test and Evaluation conference (SETE2003)*, Canberra, Australia, 2003.

Forrester, J., *Industrial Dynamics*, The MIT Press, Cambridge, MA, 1961.

Forrester, J., *Urban Dynamics*, Pegasus Communications, Waltham, MA, 1969.

Frankel, D., *Model Driven Architecture, Applying MDA to Enterprise Computing*, Wiley Publishing, Indianapolis, IN, 2003.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

Garrett, J., Ajax: A new approach to web applications, viewed 30 June 2006, ⟨`http://www.adaptivepath.com/publications/essays/archives/000385.php`⟩, 2006.

Glass, R., A structure-based critique of contemporary computing research, *Journal of Systems Software*, *28*(1), 3–7, 1995.

Grafton, Q., L. Robin, and R. Wasson, *Understanding the Environment: Bridging the disciplinary divides*, University of New South Wales Press, Sydney, Australia, 2005.

Gravana Pty Ltd, Bullant web site, viewed 30 June 2006, ⟨`http://www.bullant.com`⟩, 2005.

Greenwood, P., From the president - whither engineering?, *Engineers Australia Magazine*, *75*(4), 3, 2003.

Grigg, L., R. Johnson, and N. Milsom, Emerging issues for cross-disciplinary research: Conceptual and empirical dimensions, *Tech. rep.*, Commonwealth Department of Education, Science and Training, Canberra, Australia, 2003.

Harel, D., On visual formalisms, *Communications of the ACM*, *31*(3), 514–530, 1988.

Hatley, D., and I. Pirbhai, *Strategies for real-time system specification*, Dorset House, New York, NY, 1987.

Hawken, P., A. Lovins, and H. Lovins, L, *Natural Capitalism: The Next Industrial Revolution*, Earthscan Publication Ltd, London, UK, 1999.

Highsmith, J., *Adaptive Software Development: A collaborative approach to manageing complex systems*, Dorset House Publishing, New York, NY, 1999.

Humphrey, W., *Managing the software process*, Addison-Wesley Publishing Company Inc., USA, 1989.

IBM, IBM Flowcharting Techniques, *Tech. Rep. C20-8152-1*, International Business Machines, White Plains, NY, 1969.

International Council on Systems Engineering, Intelligent Enterprise Working Group, viewed 30 June 2006, ⟨`http://www.incose.org/practice/techactivities/wg/intelent`⟩, 2006a.

International Council on Systems Engineering, INCOSE home page, viewed 30 June 2006, ⟨`http://www.incose.org`⟩, 2006b.

International Organization for Standardization, *ISO/IEC 12207:1995/Amendment 1:2002 Software Lifecycle Processes*, International Standards Organisation, 1997.

International Organization for Standardization, *ISO/IEC 15228 Systems Enginering - System life cycle processes*, International Standards Organisation, 2002.

International Organization for Standardization, *ISO/IEC 15504:2005 Information technology - Process assessment*, International Standards Organisation, 2005a.

International Organization for Standardization, *ISO/IEC 9075:2005 Information technology - Database Languages - SQL*, International Standards Organisation, 2005b.

ISEE Systems, Stella and iThink system dynamics modeling tools, viewed 30 June 2006, ⟨`http://www.iseesystems.com`⟩, 2006.

Jacobson, I., and P. Ng, *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, Upper Saddle River, NJ, 2005.

Jones, C., *Applied Software Measurement*, McGraw Hill, New York, 1996.

Jordan, N., *Themes in Speculative Psychology*, Tavistock Publications Limited, London, England, 1968.

Kaplan, R., and D. Norton, *The Balanced Scorecard: Translating Strategy into Action*, Harvard Business School Press, Cambridge, MA, 1996.

Kennedy Carter, iUML executable UML modeling tool, viewed 30 July 2006, ⟨`http://www.kc.com`⟩, 2006.

Kernel.Org Organization, Inc., The Linux kernel, viewed 30 June 2006, ⟨`http://www.kernel.org`⟩, 2006.

Kotonya, G., and I. Sommerville, *Requirements Engineering - processes and techniques*, John Wiley & Sons, New York, NY, 1998.

Kramer, N., *Systems thinking: concepts and notions*, Martinus Nijhoff, Leiden, 1977.

Mellor, S., and M. Balcer, *Executable UML, A foundation for Model-Driven Architecture*, Addison-Wesley, Indianapolis, IN, 2002.

Mellor, S., K. Scott, A. Uhl, and D. Weise, *MDA Distilled, Principles of Model-Driven Architecture*, Addison-Wesley, Boston, MA, 2004.

Mentor Graphics, Bridgepoint development suite, viewed 30 June 2006, ⟨`http://www.mentor.com`⟩, 2006.

Microsoft, Active Server Pages, viewed 30 June 2006, ⟨`http://www.microsoft.com/asp`⟩, 2006a.

Microsoft, Microsoft .net, viewed 30 June 2006, ⟨`http://www.microsoft.com/net`⟩, 2006b.

Microsoft, Microsoft windows, viewed 30 June 2006, ⟨`http://www.microsoft.com/windows`⟩, 2006c.

Mozilla.org, Javascript, viewed 30 June 2006, ⟨`http://www.mozilla.org/js`⟩, 2004a.

Mozilla.org, XML-based User Interface Language, viewed 30 June 2006, ⟨`http://www.mozilla.org/xpfe`⟩, 2004b.

Mozilla.org, Firefox web browser, viewed 30 June 2006, ⟨`http://www.mozilla.com/firefox`⟩, 2006.

Murphy, G., and C. Schwanninger, Special issue on aspect-oriented computing, *IEEE Software*, *23*(1), 20–23, 2006.

MySQL AB, MySQL open source database, viewed 30 June 2006, ⟨http://www.mysql.com⟩, 2006.

Nichols, R., and C. Connaughton, Software process improvement journey: IBM Australia Application Management Services, *Tech. Rep. CMU/SEI-2005-TR-002*, Software Engineering Institute, Pittsburgh, PA, 2005.

Object Management Group, *Unified Modeling Language Specification, version 1.5*, Object Management Group, Needham, MA, viewed 30 June 2006, ⟨http://www.omg.org⟩, 2003a.

Object Management Group, *Model Driven Architecture Guide, version 1.0*, Object Management Group, Needham, MA, viewed 30 June 2006, ⟨http://www.omg.org⟩, 2003b.

Object Management Group, *UML for Systems Engineering, Request for Proposal*, Object Management Group, Needham, MA, viewed 30 June 2006, ⟨http://www.omg.org⟩, 2003c.

Object Management Group, *OMG Systems Engineering Domain Special Interest Group*, Object Management Group, Needham, MA, viewed 30 June 2006, ⟨http://www.omg.org⟩, 2003d.

Object Management Group, *OMG Meta Object Facility (MOF) Specification*, Object Management Group, Needham, MA, viewed 30 June 2006, ⟨http://www.omg.org⟩, 2003e.

Object Management Group, *Unified Modeling Language: Superstructure, version 2.0*, Object Management Group, Needham, MA, viewed 30 June 2006, ⟨http://www.omg.org⟩, 2005.

Object Management Group, *Object Constraint Language, version 2.0*, Object Management Group, Needham, MA, viewed 30 June 2006, ⟨http://www.omg.org⟩, 2006.

Offen, R., Domain understanding is the key to successful system development, *Requirements Engineering*, *7*(3), 172–175, 2002.

O'Reilly, LAMP: The open source web platform, viewed 30 June 2006, ⟨http://www.onlamp.com⟩, 2006.

Paulk, M., B. Curtis, C. M.B., and C. Weber, Capability maturity model for software, version 1.1, *Tech. Rep. CMU/SEI-93-TR-024*, Software Engineering Institute, Pittsburgh, PA, viewed 30 June 2006, ⟨http://www.sei.cmu.edu/publications/documents/93.reports/93.tr.024.html⟩, 1993.

Potts, C., Software engineering research revisited, *IEEE Software*, *10*(5), 19–28, 1993.

Raistrick, C., P. Francis, J. Wright, C. Carter, and I. Wilkie, *Model Driven Architecture with Executable UML*, Cambridge University Press, Cambridge, UK, 2004.

Reifer, D., How good are agile methods?, *IEEE Software*, *19*(4), 16–18, 2002.

Rittel, H., and M. Webber, Dilemmas in a general theory of planning, *Policy Sciences*, *4*(2), 155–169, 1973.

Royce, W. W., Managing the development of large software systems, in *IEEE WESCON*, pp. 1–9, IEEE, 1970.

Sauer, C., and C. Cuthbertson, The state of IT project management in the UK, *Tech. rep.*, Templeton College, Oxford, viewed 30 June 2006, ⟨`http://www.computerweeklyms.com/pmsurveyresults/index.asp`⟩, 2003.

Schwaber, K., and M. Beedle, *Agile Software Development with SCRUM*, Prentice Hall, Upper Saddle River, NJ, 2002.

Senge, P., *The Fifth Discipline*, Random House Australia, Milsons Point NSW, 1992.

Shlaer, S., and S. Mellor, *Object Lifecycles, modeling the world in states*, Prentice-Hall, Upper Saddle River, NJ, 1992.

Sommerville, I., and P. Sawyer, *Requirements Engineering - A good practice guide*, John Wiley & Sons, Chichester, England, 1997.

Sproles, N., Coming to grips with measures of effectiveness, *Systems Engineering*, *3*(1), 50–58, 2000.

Starr, L., *Executable UML, A case study*, Model Integration, LLC., San Francisco, CA, 2001.

Starr, L., *Executable UML, How to build class models*, Prentice-Hall, Upper Saddle River, NJ, 2002.

Stewart, T., The invisible key to success, *Fortune Magazine*, 1996.

Stien, M., Redeveloping eVACS using xtUML and BridgePoint, viewed 30 June 2006, ⟨`http://www.softimp.com.au/softeng/white_papers.html`⟩, 2003.

Stien, M., Developing Enterprise Applications with Executable Translatable UML, presentation to the Australian Computer Society (ACT) Software Quality Association technical session, 2006.

Subramanyam, V., S. Deb, P. Krishnaswamy, and R. Ghosh, An integrated approach to software process improvement at Wipro Technologies: veloci-Q, *Tech. Rep. CMU/SEI-2004-TR-006*, Software Engineering Institute, Pittsburgh, PA, 2004.

Sun Microsystems, Java platform, enterprise edition, viewed 30 June 2006, ⟨`http://java.sun.com/javaee`⟩, 2006a.

Sun Microsystems, Java Server Pages, viewed 30 June 2006, ⟨`http://java.sun.com/products/jsp`⟩, 2006b.

Sun Microsystems, Java foundation classes, viewed 30 June 2006, ⟨`http://java.sun.com/products/jfc`⟩, 2006c.

Swarm Development Group, SwarmWiki, viewed 30 June 2006, ⟨`http://www.swarm.org`⟩, 2006.

The British Computer Society and Royal Academy of Engineering, The challenges of complex IT projects, *Tech. rep.*, The Royal Academy of Engineering, Westminster, London, 2004.

The PHP Group, PHP: Hypertext preprocessor, viewed 30 June 2006, ⟨`http://www.php.net`⟩, 2006.

The Standish Group, Chaos, *Tech. Rep. T23E-T10E*, The Standish Group, West Yarmouth, MA, viewed 30 June 2006, ⟨`http://www.standishgroup.com/sample_research`⟩, 1994.

The Standish Group, Unfinished voyages, *Tech. rep.*, The Standish Group, West Yarmouth, MA, viewed 30 June 2006, ⟨`http://www.standishgroup.com/sample_research`⟩, 1995.

The Standish Group, Chaos: A recipe for success, *Tech. rep.*, The Standish Group, West Yarmouth, MA, viewed 30 June 2006, ⟨`http://www.standishgroup.com/sample_research`⟩, 1999.

The Standish Group, Extreme chaos, *Tech. rep.*, The Standish Group, West Yarmouth, MA, viewed 30 June 2006, ⟨`http://www.standishgroup.com/sample_research`⟩, 2001.

Tichy, W., Should computer scientists experiment more?, *Computer*, *31*(5), 32–40, 1998.

Ward, P., and S. Mellor, *Structured development for real-time systems*, vol. 1, Yourdon Press, Englewood Cliffs, NJ, 1985.

Wiegers, K., *Software Requirements*, Microsoft Press, Redmond, Washington, USA, 2003.

World Wide Web Consortium, HyperText Markup Language, viewed 30 June 2006, ⟨`http://www.w3.org/TR/1999/REC-html401-19991224`⟩, 1999a.

World Wide Web Consortium, HyperText Transfer Protocol 1.1, viewed 30 June 2006, ⟨`http://www.w3.org/Protocols/rfc2616/rfc2616.html`⟩, 1999b.

World Wide Web Consortium, Simple Object Access Protocol, viewed 30 June 2006, ⟨`http://www.w3c.org/TR/Soap`⟩, 2004.

Young, R., *Effective Requirements Practices*, Addison-Wesley, Upper Saddle River, NJ, USA, 2001.

Proof-of-Concept Index

Classes
- Annotation, 130
- Contact, 121
- Contact Relationship, 122
- Contact Relationship Type, 122
- CRUD List, 132
- Form, 132
- Form Item, 132
- Item Permission, 127
- Link, 131
- Main Heading, 131
- Note, 122
- Page, 130
- Page Item, 130
- Password Input, 133
- Permission, 127
- Product, 124
- Product Category, 125
- Product Variant, 124
- Protected Item, 127
- Registered User, 126
- Role, 127
- Select, 134
- Simple Text, 131
- Sub-Heading, 131
- Supplier, 125
- Text, 131
- Text Area, 133
- Text Input, 133

Domain Models
- PMS Security, 164
- PMS User Interface, 166
- $\frac{X}{T}$UML, 136
- APACHE Web Server, 136
- Application Mechanisms, 302
- Contact Management, 120
- Database Mechanisms, 303
- LAMP Framework, 134
- Linux Operating System, 136
- MySQL Relational Database, 136
- PHP Web Programming Language, 136
- Product Management, 123
- Security, 125
- Security Mechanisms, 312
- Structured Query Language, 136
- User Interface, 128
- User Interface Mechanisms, 304

Domain Specific Types
- Currency, 124
- DisplayHeight, 129
- DisplayText, 128
- DisplayWidth, 129
- EmailAddress, 121, 124
- InputText, 128
- Memo, 121, 123, 126, 128
- Name, 121, 123, 126, 128
- Password, 126
- PhoneNumber, 121, 123
- Phrase, 120
- StreetAddress, 121, 123
- Username, 126

Implementation Rules
- irAttributePasswordInput, 222
- irAttributeTextArea, 223
- irAttributeTextInput, 220
- irClassList, 206
- irCorrespondingClasses, 237
- irDatabaseConfig, 226
- irDatabaseScripts, 227
- irEditClassForm, 218
- irEditClassPage, 215
- irFileAndDirectoryStructure, 190
- irForm, 201

irFunctionLink, 196
irGroupedClassList, 212
irLinkObjectSelect, 224
irPage, 193
irPageLink, 198
irPersistentClass, 230
irRolePageLink, 199
irSecurityConfig, 238
irStartPage, 191
irStaticPageItems, 195
irUiConfig, 192

Requirement Archetypes
raApacheImplementedOnLinux, 154
raClassList, 144
raCorrespondingClasses, 153
raFormHandledByPhpFunction, 144
raGroupClassList, 146
raLinkAvailableWithRole, 142
raLinkCallsPhpFunction, 143
raLinkOpensPage, 142
raMysqlImplementation, 152
raMysqlImplementedOnLinux, 155
raPageImplementedInPhp, 142
raPasswordInputUpdatesAttribute, 148
raPersistentAssociation, 152
raPersistentClass, 152
raPhpImplementedOnApache, 154
raSecureClass, 151
raSecurityPhpImplementation, 151
raSelectItemCreatesLink, 149
raTextAreaUpdatesAttribute, 149
raTextInputUpdatesAttribute, 147
raUiImplementedInPhp, 153

Author Index

Abdel-Hamid and Madnick [1991], 31
Ackoff [1971], 30
Ackoff [1999], 17, 19–21, 28
Adrion [1993], 37, 38
Ahern et al. [2004], 16
Bass et al. [2003], 56
Beck [1999], 17
Bell [1998], 24, 26
Bertalanffy [1962], 30
Boehm and Basili [2001], 26
Boehm and Turner [2004], 18
Boehm [1988], 81
Boughton [2006], 26, 27, 104, 108
Boulding [1956], 3, 30
Boyd [1986], 31, 42
Brooks [1987], 13
Buzan [2003], 60
Capell [2004], 16
Casti [1979], 20, 21, 321
Checkland and Scholes [1999], 28, 75
Checkland [1981], 7, 8, 19, 20, 28, 30, 53, 75, 327
Checkland [1989], 20
Chen [1977], 59
Churchman [1968a], 30
Churchman [1968b], 20
Clarke and Baniassad [2005], 31
Cockburn [2002], 17
Coram [2002], 31
Davis and Zowghi [2006], 15
Davis [1989], 109, 332
Demarco [1979], 59, 60
Diaz and Sligo [1997], 16
Dijkstra [1982], 25, 53
Elrad et al. [2002], 367
Endres and Rombach [2003], 4, 13, 26, 61
Fenton et al. [1994], 40
Ferguson et al. [1997], 16
Ferguson et al. [1999], 16, 18
Filman et al. [2005], 31
Flint and Boughton [2002], 5, 9, 56, 107, 330
Flint and Boughton [2003], 7, 28
Flint [1999], 318, 338
Flint [2001], 6, 8, 21, 75, 338, 373
Flint [2006], 344
Forrester [1961], 8, 28, 30, 55, 60, 75, 322
Forrester [1969], 22
Frankel [2003], 361
Gamma et al. [1995], 341, 368
Garrett [2006], 105
Glass [1995], 38
Grafton et al. [2005], 4, 106
Greenwood [2003], 31
Grigg et al. [2003], 29
Harel [1988], 114, 116
Hatley and Pirbhai [1987], 59
Hawken et al. [1999], 31
Highsmith [1999], 17
Humphrey [1989], 16
IBM [1969], 60
Jacobson and Ng [2005], 31
Jones [1996], 13
Jordan [1968], 19, 30
Kaplan and Norton [1996], 18
Kotonya and Sommerville [1998], 15
Kramer [1977], 20, 30, 318, 322
Mellor and Balcer [2002], 9, 24, 31, 32, 49, 53, 56, 59, 60, 136, 236, 363
Mellor et al. [2004], 9, 24, 27
Microsoft [2006a], 339
Microsoft [2006b], 339
Microsoft [2006c], 18

Murphy and Schwanninger [2006], 31
Nichols and Connaughton [2005], 16
Offen [2002], 31
Paulk et al. [1993], 16
Potts [1993], 35, 40–44, 46
Raistrick et al. [2004], 53, 60
Reifer [2002], 17
Rittel and Webber [1973], 7, 20
Royce [1970], 81
Sauer and Cuthbertson [2003], 18
Schwaber and Beedle [2002], 17
Senge [1992], 20
Shlaer and Mellor [1992], 24, 31, 32, 53, 363
Sommerville and Sawyer [1997], 15, 30
Sproles [2000], 21
Starr [2001], 27, 363
Starr [2002], 31, 363
Stewart [1996], 319
Stien [2003], 26, 104, 108
Stien [2006], 26, 104, 108
Subramanyam et al. [2004], 16
Tichy [1998], 46
Ward and Mellor [1985], 59
Wiegers [2003], 15, 89
Young [2001], 15, 16
Agile Alliance [2001], 17
Apache Software Foundation [2006], 95, 135, 136
Australian Department of Defence [1997], 326
Australian Department of Defence [2000], 326
CMMI Product Development Team [2000], 16, 332
Droplets, Inc. [2005], 340
Gravana Pty Ltd [2005], 340
ISEE Systems [2006], 60, 75, 78
International Council on Systems Engineering [2006a], 368
International Council on Systems Engineering [2006b], 361
International Organization for Standardization [1997], 16, 36
International Organization for Standardization [2002], 16, 36, 82
International Organization for Standardization [2005a], 16
International Organization for Standardization [2005b], 95, 136
Kennedy Carter [2006], 26, 60, 78, 363, 367
Kernel.Org Organization, Inc. [2006], 95, 135, 136
Mentor Graphics [2006], 26, 60, 78, 137, 363, 367
Mozilla.org [2004a], 339
Mozilla.org [2004b], 339
Mozilla.org [2006], 134
MySQL AB [2006], 95, 135, 136
O'Reilly [2006], 134
Object Management Group [2003a], 60, 360, 364
Object Management Group [2003b], 27, 361
Object Management Group [2003c], 27, 361, 372
Object Management Group [2003d], 361
Object Management Group [2003e], 62
Object Management Group [2005], 5, 60, 114
Object Management Group [2006], 98
Sun Microsystems [2006a], 105
Sun Microsystems [2006b], 339
Sun Microsystems [2006c], 341, 342, 351

Swarm Development Group [2006], 22, 345
The British Computer Society and Royal Academy of Engineering [2004], 14, 16, 18
The PHP Group [2006], 95, 135, 136, 339
The Standish Group [1994], 13
The Standish Group [1995], 13
The Standish Group [1999], 18
The Standish Group [2001], 16
World Wide Web Consortium [1999a], 134
World Wide Web Consortium [1999b], 134
World Wide Web Consortium [2004], 339, 347