UNIVERSITY OF PORTO · FACULTY OF ENGINEERING

Adaptive Object-Modeling
Patterns, Tools and Applications

HUGO JOSÉ SERENO LOPES FERREIRA

December 2010

Scientific Supervision by

Ademar Aguiar, Assistant Professor
Department of Informatics Engineering

In partial fulfillment of requirements for the degree of
Doctor of Philosophy in Computer Science
by the
MAPi Joint Doctoral Programme

Contact Information:

Hugo Sereno Ferreira
Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Rua Dr. Roberto Frias, s/n
4200-465 Porto
Portugal

Tel.: +351 22 508 1400
Fax.: +351 22 508 1440
Email: hugo.sereno@fe.up.pt
URL: http://www.fe.up.pt/ hugosf

This thesis was typeset on an Apple® MacBook® Pro running Mac OS® X 10.6 using the free LaTeX typesetting system, originally developed by Leslie Lamport based on TeX created by Donald Knuth. The body text is set in Minion Pro, a late Renaissance-era type originally designed by Robert Slimbach. Other fonts include Sans and Typewriter from the Computer Modern family, and Courier, a monospaced font originally designed by Howard Kettler at IBM and later redrawn by Adrian Frutiger. Typographical decisions were based on the recommendations given in *The Elements of Typographic Style* by Robert Bringhurst (2004), and graphical guidelines were inspired in *The Visual Display of Quantitative Information* by Edward Tufte (2001). This colophon has exactly one hundred and twenty-two (122) words, excluding all numbers and symbols.

"Adaptive Object-Modeling: Patterns, Tools and Applications"

This work was partially funded by FCT and PARADIGMAXIS, S.A. as a doctoral scholarship, ref. SFRH/BDE/33298/2008.

…to Helena

This page was intentionally left mostly blank.

Abstract

The field of Software Engineering (SE) could be regarded as *in crisis* since the early days of its birth. Project overruns and failures have hitherto characterized the *norm*, blamed upon the *unreasonable expectations* set by the stakeholders and the *constant change* of the requirements. While some development processes struggle to systematically eradicate these *barriers* from software development, other methodologies, specially those coined *agile*, promote a continuous *embrace of change* as a first-class property of this activity. Even so, it is a yet to be disputed contingency that there is no *silver bullet*; no single, unifying methodology capable of consistently improving the efficiency of software development by even an order of magnitude.

Nonetheless, the agglomeration of empirical evidence is now starting to suggest that *change*, more than a mere *cause*, may be a *symptom* of a deeper nature, directly related to inherent properties of some of the domains software tries to address — software may need to be under constant change because any modeling of those domains is *incomplete* since the day it starts.

Therefore, if such solutions need to be constantly *evolving* and *adapted* to new realities, hence regarded as *incomplete by design*, shouldn't they be actively *designed for incompleteness*? If agile methodologies shifted the way teams and individuals plan development to *embrace change*, how should one deliberately design to cope with *continuous change*?

Even if the answer to this question could be promptly given, the goal of pragmatically contributing to the body of knowledge in the SE field rises a problem of its own. As knowledge grows in a specific area, solutions are captured and documented by experts in books, research papers, concrete implementations, webpages, etc. While one may intuitively think that this growth implies better design, it has been shown that the way (and the amount) this knowledge is captured raises an important issue *per-se*. Design, as an intentional act of choice, is constantly overwhelmed by the sheer quantity of available information.

In this dissertation, the author starts by presenting arguments pointing to how these *incomplete by design* systems are a common issue in nowadays practice of software engineering. During that overview, one must inevitably delve into the forces that lead architects, designers and developers to recognize these kind of systems. The target thus become readers who have recognized (or are interested) in the set of systems that show a high degree of variability and incompleteness, and which goal is to reduce the overall *effort* — monetary cost, available time and resources, required skills, resulting complexity, etc. — of coping with continuous change made to the do-

main model. Mainly, this dissertation aims to answer *what form should this type of systems take, and which kind of tools and infrastructures should be available to support the development of such software?*

The way this problem is here coped, with a permanent focus on the development of prescriptive contributions to the SE body of knowledge, is as follows. First, it is presented as a formalization of a unified conceptual *pattern language*, following the theories first laid by Alexander *et al.*, for systems which domain model is target of continuous change during runtime. The underlying meta-architecture of these systems is known as *Adaptive Object-models*, and this pattern language aims to answers questions such as when do these type of systems occur, their advantages and disadvantages, their underlying requirements, the set of available techniques and solutions and their main benefits and liabilities.

The author then proceeds to specify a reference architecture of an object-oriented framework capable of dealing with such systems. Questions here addressed include the type and form of needed infrastructures, what abstractions should be made and supported, the generic functionalities it should provide, its default behavior and points of extension. The author takes a further step in providing and detailing an industrial-level implementation of such framework, along with the specific design issues that arise only when pursuing such concrete artifacts.

Finally, the author's main thesis is validated by providing empirical evidence of the framework benefits through industrial use-case applications and controlled academic quasi-experiments. Two commercial software systems built on top of that framework are used as case studies, reflecting their own specific context, their requirements, their particular problems, and the way the framework — and the underlying theories here built — were used to address them, the outcomes, and the lessons learned. Inherent threats to this type of validation are further dismissed by presenting the results of a (quasi-)experiment within a controlled academic environment, where the results of studying groups of undergraduate students following different treatments are shown to be consistent with those earlier presented.

Resumo

A Engenharia de Software (ES), enquanto área de profissionalização, debate-se numa profunda *crise* que remonta aos primeiros dias do seu nascimento, e se reflecte nos constantes atrasos e insucessos dos seus projectos, ao ponto dessas situações definirem a *norma* na área. No topo da "lista" das principais causas desta situação encontram-se as expectativas irrealistas dos intervenientes e a mudança contínua dos requisitos. Em resposta, têm surgido alguns processos de desenvolvimento que defendem a erradicação sistemática destas barreiras do acto de criação de software; mas outros, especialmente aqueles auto-denominadas *ágeis*, sublinham a necessidade de *abraçar a mudança* como uma propriedade essencial da própria actividade. Apesar de tudo, permanece por refutar a hipótese de não existir uma *bala de prata*, uma única metodologia unificadora capaz de melhorar consistentemente a eficiência do desenvolvimento de software, pelo menos, numa ordem de grandeza.

A observação empírica começa, no entanto, a sugerir que a mudança, mais do que uma simples *causa*, apresenta-se como um *sintoma* resultante da natureza das propriedades intrínsecas a alguns domínios modelados pelo software — a respectiva necessidade de ser continuamente modificado existe porque qualquer tentativa de modelar de forma definitiva esses domínios está condenada a ser *incompleta* à partida.

Mas se tais soluções são um alvo inevitável de evoluções e adaptações constantes a novas realidades — *incompletas por natureza* — não deveriam então ser activamente desenhadas para serem *naturalmente incompletas*? Se as metodologias ágeis modificam a forma como as equipas e os indivíduos planeiam e executam o processo de desenvolvimento para *abraçar a mudança*, como deveremos desenhar software para deliberadamente suportar uma *mudança contínua*?

Mesmo que uma resposta estivesse facilmente ao nosso alcance, o objectivo de contribuir pragmaticamente para o corpo de conhecimento na área da ES levanta um novo problema. Por norma, à medida que o conhecimento aumenta numa determinada área, as novas soluções são capturadas e documentadas por especialistas em livros, artigos de investigação, implementações, páginas de internet, etc. Embora pareça intuitivo que este aumento de conhecimento implique uma melhoria nas escolhas de desenho, tem sido demonstrado que a forma como este é capturado é problemática. O desenho, como um acto deliberado de escolha, é sistematicamente subjugado pela quantidade de informação disponível.

Nesta dissertação começa-se por apresentar os argumentos que tornam os sistemas *incom-*

pletos por natureza uma contingência da ES. Para isso, será importante perceber as forças que levam arquitectos, *designers* e programadores a identificarem este tipo de sistemas. O público alvo é assim os leitores que se identificam com estas forças, ou que estão interessados no estudo de sistemas caracterizados por um elevado grau de variabilidade e incompletude, e que tenham como objectivo o de reduzir o esforço — em termos de custo monetário, tempo, recursos, complexidade resultante, etc. — para lidar com a *mudança*, a evolução constante do modelo de domínio. Principalmente, esta dissertação tem como objectivo responder à pergunta *qual a forma — no sentido de aparência, estrutura — que estes sistemas tomam, e que tipo de ferramentas e infra-estruturas são necessárias para suportar o desenvolvimento desse tipo de software?*

Em seguida descreve-se a metodologia utilizada, com um foco permanente no desenvolvimento de contribuições prescritivas para o corpo de conhecimento da área. Seguindo as teorias seminalmente estabelecidas por Alexander, é apresentada a formalização conceptual de uma linguagem de padrões de sistemas cujo modelo de domínio é objecto de evoluções contínuas. A meta-arquitectura subjacente é conhecida como "Modelos de Objectos Adaptativos", e a linguagem de padrões tem como objectivo responder a questões relacionadas com a ocorrência deste tipo de sistemas, as suas vantagens e desvantagens, os seus requisitos básicos, as técnicas e as soluções disponíveis.

Segue-se a especificação de uma arquitectura de referência de uma *framework* orientada a objectos, capaz de lidar com tais sistemas. Aqui são abordadas questões tais como o tipo e forma de tais infra-estruturas, que abstracções se devem providenciar, que funcionalidades genéricas devem ser incorporadas e quais os comportamentos base e pontos de extensão adequados. É paralelamente desenvolvida e detalhada uma implementação de referência de nível industrial, na qual se aprofundam as idiossincrasias do seu desenho e as peculiaridades dos problemas intrínsecos à tentativa de atingir artefactos para produção.

Por fim procede-se à validação da tese principal, demostrando-se a evidência dos benefícios da *framework* e, consequentemente, das teorias subjacentes. Começa-se por considerar métodos de "estudo de caso" sobre dois sistemas comerciais desenvolvidos, onde se apresentam tanto as respectivas especificidades dos seus contextos, requisitos, e problemas relevantes, como a forma em que as teorias aqui desenvolvidas foram neles utilizadas. Posteriormente são descartadas as ameaças de validação inerentes a tal forma de estudo através da realização de quasi-experiências conduzidas em ambiente controlado, através do estudo de grupos de alunos de mestrado sujeitos a diferentes tratamentos.

Résumé

Le domaine du Génie Logiciel (GL) peut être vu comme en étant en crise depuis les premiers jours de sa naissance. Les échecs et overruns de projets ont depuis été considérés comme la norme, attribués aux attentes déraisonnables des acteurs concernés et au changement constant des exigences. Alors que certains processus de développement luttent pour éradiquer systématiquement ces barrières au développement de logiciels, d'autres méthodologies, spécialement les coined agile, continuellement embrassent le changement comme une propriété de première importance de cette activité. Quand même, il y toujours l'éventualité qu'il n'y ait pas de recette miracle, pas d'unique méthodologie unificatrice capable d'améliorer d'une façon constante l'efficience du développement de logiciels, même par seulement un ordre de grandeur.

De plus, si de telles solutions ont besoin d'évoluer et de s'adapter constamment à de nouvelles réalités, et par conséquent regardées comme incomplètes dans leur modèle, ne devraient-elles pas être activement modelées pour cette incomplétude? Si d'agiles méthodologies changent la manière des équipes et des individus de planifier le développement pour embrasser le changement, comment devrions-nous délibérément modeler pour s'ajuster au changement continuel?

Même si la réponse à cette question pourrait être donnée promptement, l'objectif de contribuer d'une manière pragmatique au corps de connaissance dans le domaine du GL soulève un problème par lui-même. Comme les connaissances s'accroissent dans un certain domaine, les solutions sont capturées et documentées par les experts dans les livres, publications scientifiques, réalisations concrètes, sites Internet, etc. Alors que nous pouvons penser intuitivement que cet accroissement implique de meilleurs choix de modélisation, il a été démontré que la manière (et la quantité) que ces connaissances sont capturées soulève un problème important en soi. Modélisation, comme un choix actif intentionnel, est constamment dépassé par la quantité absolue d'information disponible.

Dans cette dissertation, l'auteur commence par présenter les arguments démontrant comment les systèmes incomplets dans leur modélisation sont un problème commun dans la pratique courante du Génie Logiciel. Pendant ce survol, il est inévitable de fouiller dans les forces qui mènent les architectes, les modeleurs, et les développeurs à reconnaître ce genre de systèmes. L'audience devient donc le public qui s'est reconnu (ou qui est intéressé) dans l'ensemble de systèmes qui démontrent un niveau élevé de variabilité et d'incomplétude, et donc l'objectif est de réduire l'effort global (i.e. les coûts monétaires, la disponibilité du temps et des ressources, les

talents requis, la complexité résultante, etc.) de s'ajuster au changement continuel que subit le domain model. Surtout, cette dissertation vise à répondre à quelle forme devrait prendre ce genre de système, et quel genre d'outils et d'infrastructures devraient être disponible pour supporter le développement de tels logiciels?

La manière dont ce problème est ici traité, avec un focus constant sur le développement des contributions prescriptives au corps de connaissance du GL, va comme suit. Tout s'abord, une formalisation d'un langage conceptuel de pattern unifié est présentée, suivie des théories initialement présentées par Alexander et coll., pour des systèmes dont le domain model est la cible de changement continuel durant l'exécution. La méta-architecture sous-jacente à ces systèmes est connue sous Adaptive Object-models, et le langage de pattern vise à répondre aux questions telles que quand ces genres de systèmes surviennent, leurs avantages et désavantages, leurs exigences sous-jacentes, l'ensemble de techniques et de solutions disponibles et leur principaux bénéfices et responsabilités.

L'auteur procède ensuite à spécifier les reference architecture d'un cadre de références orienté sur l'objet capable de gérer de tels systèmes. Les questions telles que le genre et la forme des infrastructures requises, quelles abstractions devraient être faites et supportées, les fonctionnalités génériques qu'ils devraient fournir, leur comportement par défaut et leurs points of extension, devraient ici être adressées. L'auteur fait un pas de plus en fournissant et détaillant un cadre de références pour implémentation en industrie, en même temps que les problèmes spécifiques de modélisation qui surviennent uniquement en pourchassant de tels concrets artéfacts.

Finalement, la principale hypothèse de l'auteur est ensuite validée en fournissant les évidences des bénéfices du cadre de références, tirés à la fois des applications en industrie et des quasi-expériences contrôlées en milieu académique. Deux logiciels système commerciaux basés sur ce cadre de références sont utilisés comme études de cas, reflétant leur contexte spécifique, leurs exigences, leurs problèmes particuliers, and la manière dont le cadre de références (et les théories sous-jacentes ici élaborées) a été utilisé pour les adresser, les résultats, et les leçons tirées. Les menaces inhérentes de ce type de validation sont davantage éconduites en présentant les résultats d'une quasi-expérience dans un environnement académique contrôlé, où l'étude de groupes d'étudiants suivant différents traitements s'est avérée consistante avec les études précédemment présentées.

Contents

Abstract i

Resumo iii

Résumé v

Foreword by Joseph Yoder xvii

Preface xix

1 Introduction 1

1.1 Software Crisis . . . 2
1.2 Motivational Example . . . 2
1.3 Incomplete by Design . . . 4
1.4 Accidental Complexity . . . 6
1.5 Designing for Incompleteness . . . 7
1.6 Software Design . . . 8
1.7 Patterns . . . 9
1.8 Research Goals . . . 10
1.9 Epistemological Stance . . . 10
1.10 Research Methods . . . 11
1.11 Main Goals . . . 12
1.12 How to Read this Dissertation . . . 13

2 Background 15

2.1 Fundamentals . . . 17
2.1.1 Variability and Commonality . . . 17
2.1.2 Adaptability, Adaptivity and Self-Adaptation . . . 17
2.1.3 Reflection . . . 18
2.2 Key Abstractions . . . 18
2.2.1 Metaprogramming . . . 20

2.2.2 Generative Programming . . . 21
2.2.3 Metamodeling . . . 22
2.2.4 Aspect-Oriented Programming . . . 23
2.2.5 Domain Specific Languages . . . 24
2.2.6 Meta-Architectures . . . 25
2.3 Approaches . . . 26
2.3.1 Software Product Lines . . . 26
2.3.2 Naked Objects . . . 26
2.3.3 Domain-Driven Design . . . 26
2.3.4 Model-Driven Engineering . . . 27
2.3.5 Model-Driven Architecture . . . 28
2.4 Application Frameworks . . . 29
2.4.1 Building Frameworks . . . 30
2.4.2 Documenting Frameworks . . . 31
2.5 Relevant Tools . . . 31
2.5.1 Wikis . . . 32
2.5.2 Smalltalk . . . 32
2.5.3 Ruby on Rails . . . 33
2.6 Conclusion . . . 34

3 State of the Art in Adaptive Object-Models 35
3.1 The AOM Architectural Pattern . . . 35
3.1.1 Why is it a Pattern? . . . 36
3.1.2 Towards a Pattern Language . . . 36
3.2 Formalized Patterns for AOM . . . 39
3.2.1 Type-Object Pattern . . . 40
3.2.2 Property Pattern . . . 41
3.2.3 Type-Square Pattern . . . 42
3.2.4 Accountability Pattern . . . 42
3.2.5 Composite Pattern . . . 42
3.2.6 Strategy and Rule Object Patterns . . . 43
3.2.7 Interpreter Pattern . . . 44
3.2.8 Composing the Patterns . . . 44
3.3 Differences from other Approaches . . . 45
3.4 Conclusion . . . 46

4 Research Problem 47
4.1 Epistemological Stance . . . 48
4.2 Fundamental Challenges . . . 48

4.2.1 Viewpoints . . . 50
4.3 Thesis Statement . . . 50
4.4 Specific Research Topics . . . 53
4.4.1 Specific Challenges . . . 53
4.4.2 Thesis Decomposition . . . 54
4.4.3 Thesis Goals . . . 55
4.5 Validation Methodology . . . 56
4.6 Conclusion . . . 56

5 Pattern Language 59
5.1 On Patterns and Pattern Languages . . . 59
5.1.1 What is a Pattern Language? . . . 60
5.1.2 Form . . . 60
5.2 General Context . . . 61
5.3 Technical Description . . . 63
5.4 General Forces . . . 64
5.5 Core Patterns . . . 67
5.5.1 Everything is a Thing Pattern . . . 67
5.5.2 Closing the Roof Pattern . . . 71
5.5.3 Bootstrapping Pattern . . . 72
5.5.4 Lazy Semantics Pattern . . . 74
5.6 Evolution Patterns . . . 78
5.6.1 History of Operations Pattern . . . 79
5.6.2 System Memento Pattern . . . 83
5.6.3 Migration Pattern . . . 87
5.6.4 Resulting Context . . . 89
5.7 Composing the Patterns . . . 90
5.8 Conclusion . . . 91

6 Reference Architecture & Implementation 93
6.1 Overview . . . 93
6.1.1 General Principles and Guidelines . . . 94
6.1.2 High-level Architecture . . . 95
6.1.3 Key Components . . . 96
6.1.4 Component Composition . . . 96
6.2 Thread of Control . . . 97
6.2.1 Bootstrapping . . . 98
6.2.2 Main Process . . . 99
6.2.3 Query Events . . . 99

6.2.4 Commit Events . . . 100
6.3 Architectural Decomposition . . . 100
6.3.1 Structural Core . . . 100
6.3.2 Behavioral Core . . . 103
6.3.3 Warehousing and Persistency . . . 105
6.3.4 Communications . . . 107
6.4 Crosscutting Concerns . . . 108
6.4.1 Serialization . . . 108
6.4.2 Integrity . . . 110
6.4.3 (Co-)Evolution . . . 110
6.5 User Interface . . . 112
6.6 Development Effort . . . 113
6.7 Conclusion . . . 114

7 Industrial Case-Studies 115
7.1 Research Design . . . 115
7.2 Complexity Analysis . . . 116
7.3 Baseline . . . 117
7.4 Locvs . . . 117
7.4.1 Core Concepts . . . 118
7.4.2 Time-Series Analysis . . . 119
7.4.3 Conclusion . . . 119
7.5 Zephyr . . . 121
7.5.1 Core Concepts . . . 121
7.5.2 Time-Series Analysis . . . 121
7.5.3 Conclusion . . . 121
7.6 Survey . . . 122
7.6.1 Background . . . 123
7.6.2 Overall Satisfaction . . . 123
7.6.3 Development Style & Process . . . 123
7.6.4 Future Expectations . . . 124
7.7 Lessons Learned . . . 125
7.8 Validation Threats . . . 126
7.9 Conclusion . . . 127

8 Academic Quasi-Experiment 129
8.1 Research Design . . . 129
8.1.1 Treatments . . . 130
8.1.2 Pre-test Evaluation . . . 130

8.1.3 Process . . . 131
8.1.4 Post-Test Questionnaire . . . 132
8.1.5 Independent Validation . . . 132
8.2 Experiment Description . . . 132
8.2.1 First Phase: Construction . . . 132
8.2.2 Second Phase: Evolution . . . 135
8.3 Data Analysis . . . 137
8.3.1 Background . . . 138
8.3.2 External Factors . . . 141
8.3.3 Overall Satisfaction . . . 142
8.3.4 Development Process . . . 144
8.4 Objective Measurement . . . 147
8.5 Validation Threats . . . 147
8.6 Conclusion . . . 149

9 Conclusions 151
9.1 Summary of Hypotheses . . . 151
9.2 Main Results . . . 152
9.3 Future Work . . . 153
9.3.1 Evolving Oghma . . . 153
9.3.2 Web-based Adaptive User Interfaces . . . 154
9.3.3 Improving Usability of Automatically Generated GUI . . . 155
9.3.4 Continuing the Pattern Language . . . 155
9.3.5 Self-Hosting . . . 155
9.4 Epilogue . . . 156

Appendices 158

A Pre-Experiment Data 161

B Post-Experiment Questionnaire 163

C Post-Experiment Questionnaire Results 169

D Experimental Group Documentation 171

E Industrial Survey 179

Nomenclature 181

References 185

List of Figures

1.1 Example diagram of a Medical Center IS . . . 3
1.2 Waterfall software development lifecycle model . . . 4
1.3 The Scrum software development lifecycle model . . . 5
1.4 Software as the crystallization of an abstraction . . . 5
1.5 Refactored solution of a Medical Center IS . . . 6
1.6 The four recognized forces of project management . . . 8

2.1 Concept map on related tools, approaches and abstractions. . . . 16
2.2 The four layers of abstraction in UML/MOF. . . . 29
2.3 Patterns for Evolving Frameworks . . . 31

3.1 A pattern language for Adaptive-Object Models . . . 37
3.2 The meta layers of an AOM . . . 40
3.3 A class diagram of the Type-Object pattern . . . 41
3.4 A class diagram of the Property pattern . . . 41
3.5 A class and objects diagram of the Type-Square pattern . . . 42
3.6 A class and objects diagram of the Accountability pattern . . . 43
3.7 A class and objects diagram of the Composite pattern . . . 43
3.8 A class diagram of the Strategy and Rule Object patterns . . . 44
3.9 A class diagram of the Interpreter pattern . . . 44
3.10 AOM core architecture and design . . . 45

5.1 The reflective tower of a video renting system . . . 64
5.2 The relationship among forces of object-oriented meta-architectures. . . . 65
5.3 Pattern map for core patterns of meta-architectures . . . 67
5.4 Class diagram of the Everything is a Thing pattern . . . 69
5.5 Data and metadata evolution patterns. . . . 78
5.6 Applying Everything is a Thing for evolution patterns . . . 79
5.7 History of Operations class diagram . . . 81
5.8 System Memento class diagram . . . 85
5.9 Literal instantiation of the System Memento pattern . . . 86

5.10 Delta instantiation of the System Memento pattern . . . 86
5.11 Class diagram of the Migration pattern . . . 88
5.12 A pattern language for Adaptive-Object Models . . . 92

6.1 High-level architecture of the Oghma framework . . . 95
6.2 Architecture of a client-server setup . . . 97
6.3 Architecture of a distributed setup . . . 97
6.4 Activity diagram of the framework initialization . . . 98
6.5 Activity diagram of the main loop . . . 99
6.6 Activity diagram of query events . . . 99
6.7 Activity diagram of commit events . . . 100
6.8 Core design of the structural meta-model. . . . 101
6.9 An extension of the Type-Square pattern. . . . 101
6.10 Implementing the Everything is a Thing pattern . . . 102
6.11 An example of two different states of the same entity. . . . 103
6.12 Implementation model of the structural meta-model . . . 103
6.13 A class diagram of the Strategy and Rule Object patterns . . . 104
6.14 Dynamic core of the Oghma framework . . . 104
6.15 Class diagram of the warehousing design . . . 106
6.16 Class diagram of the persistency design . . . 107
6.17 Data and meta-data are manipulated through operations . . . 110
6.18 Merging mechanism used to validate and apply operations . . . 111
6.19 Example of a simple model evolution . . . 112
6.20 Oghma code-base complexity . . . 113

7.1 GISA evolution . . . 117
7.2 SMQVU evolution . . . 117
7.3 Locvs model evolution . . . 119
7.4 Locvs model complexity . . . 120
7.5 Zephyr model complexity . . . 122

8.1 Experiment design . . . 130
8.2 Task 1 . . . 133
8.3 Task 2 . . . 134
8.4 Task 3 . . . 135
8.5 Changes for Task 1 . . . 136
8.6 Changes for Task 2 . . . 136
8.7 Changes for Task 3 . . . 137

9.1 Evolution of the Oghma implementation . . . 154

List of Tables

3.1 A pattern catalog for Adaptive Object Models 38

6.1 Wiki design principles 94
6.2 Design principles for incomplete by design systems 95
6.3 Code metrics for the current implementation of Oghma 114

7.1 Locvs Chronogram 119
7.2 Background results of industrial survey 123
7.3 Overall satisfaction results of industrial survey 124
7.4 Development style results of industrial survey 124
7.5 Development process results of industrial survey 125
7.6 Future expectations results of industrial survey 125

8.1 Student grades group statistics 130
8.2 Independent Samples Test 131
8.3 Background Results 139
8.4 External Factors 141
8.5 Overall Satisfaction 142
8.6 Development Process 145
8.7 Implemented requirements 147

9.1 New features results of industrial survey 153

A.1 Student grades for the Experimental group 161
A.2 Student grades for the Baseline group 161

C.1 Post-experiment questionnaire results 169

E.1 Industrial survey results 180

Foreword by Joseph Yoder

Over the last decade it has become extremely important for systems to be able to adapt quickly to changing business requirements. Because of this, there has been an evolution of the way we develop and think about building systems. For example, Agile processes have evolved as a means to build systems more quickly. But when working on Agile projects, developers can become overwhelmed with the rate at which requirements change. The ease with which requirements can change encourages users to overwhelm developers with feature requests. The result: *Featuritis*, which promotes hasty construction of poorly designed software to support those features. The design of an expressive domain model might get lost in the rush to write working code. Adaptive Object-Models (AOM) support changeable domain modules by casting business rules as interpreted data and representing objects, properties and relationships in external declarations. At first glance, AOM systems seem to contradict Agile values. Yet we find that under the right conditions, an AOM architecture has made our users happier and has given them the ability to control the pace of change. It can be the ultimate in agility!

This thesis has taken the next step in state of art research on best practices for building these types of dynamic systems. This work illustrates the necessity of software to adapt and change at sometimes rapid speeds. Hugo's addition of seven key patterns brings together years of study and produces formalized patterns as an important contribution to the AOM community. The thesis also brings real world problems to light and centers much of the discussion on the inherent flaws of much of today's software design practices.

The work reveals much about the current state of Adaptive Object Model practices in industry. I found the thesis to be a quality entry into the current AOM research and very enlightening for those in the software community building these types of architectures. The review of AOM architectural style, the patterns catalogue, the outline of the reference architecture for AOM along with an example implementation makes the work very relevant and informative, as well as a pleasure to read. Additionally the case studies highlight the common issues and problems addressed by AOM along with proving that AOM exist as a common solution for these types of systems.

I look forward to future collaboration with Hugo on this topic area and it is my pleasure to strongly recommend this work to architects and developers that are building these types of dynamic architectures.

Preface

The journey of a thousand miles begins with one step.

Lao Tzu

My earliest memories as an "engineer" date back to childhood and Christmas, to LEGO and robots, to dismantling things without quite knowing how to assemble them back. It was a fun hobby for me, and a nuisance — if not dangerous activity — for my parents. I recall that once I had the "brilliant" idea of connecting a portable playing device — a *Game&Watch* — to the AC mains when I ran out of batteries. To my understanding, they both provided energy, so it should have worked. I was five... About one year later, my neighbor got one thing called *computer*, which when connected to a television allowed us to control what happened there: the *Atari 2600*. Soon my parents offered me a MSX as a Christmas gift. My father, not very fond of technology, bought it second-handed believing it was an entertainment device, but missed the fact he had to actually *buy games* separately. I usually tell this story about why I claim to have learned programming at the age of six: a kid with a new toy, no games, and a book called *Programmer's Manual*; the outcome seems almost inevitable among those born in late '70s and early '80s.

I like to think that such an early background gave me time to *ruminate* over the nature of software, specifically what separates *good and elegant* from *bad and sloppy* programs, before one starts facing programming as a professional need. Maybe the *aesthetic* influences from my art-oriented friends played a key role on that. Programming, from my point of view, was no more of a *science* than of an *art*. But it was after finishing my degree, when I became acquainted with *design patterns* and *meta-programming*, that I've realized I wasn't alone in this pursue for *elegance* in a seemingly mechanical world:

> *What is the Chartres of programming? What task is at a high enough level to inspire people writing programs, to reach for the stars? Can you write a computer program on the same level as Fermat's last theorem?…Can you write a program which overcomes the gulf between technical culture of our civilization, and which inserts itself into our human life as deeply as Eliot's poems of the wasteland or Virginia Woolf's The Waves?* [Gab96]

Experiencing at first-hand the problems of *bad design* in the industrial world, by spending too much nights with too little sleep, tight time constraints persuaded to make things *as automatic*

as possible — climbing the hill of abstraction becomes a *necessary fun*. Doing requirements engineering also made us painfully aware of an important pattern: most customers hardly know what they *want*, and very seldom what they *need*. Undoubtedly all this influenced the course of my Ph.D. even before it started. But, I was still an engineer...

Gopalakrishnan, in his book COMPUTATION ENGINEERING [Gopo6], differentiates *"people who seek an in-depth understanding of the phenomenon of computation to be computer scientists"* from *"those people who seek to efficiently apply computer systems to solve real-world problems to be computation engineers"*. And alas, something inside me is deeply disturbed every time someone disregards as *"just a matter of implementation"*. Just?... But implementation changes everything! It is the act of crafting, of making a theory materialize, the ultimate aspiration of creating something out of nothing — or from scarce resources — that shapes the *motivational forces* behind an engineer.

Take the Church-Turing thesis; it is a deep, beautiful hypothesis that provides an insight on the meaning of *computability* — and a bridge between symbols and machines —, which is nothing less than an amazing display of the motivation that drive scientists to pursue mathematical truth. Personally, I am amazed by their efforts to find the underlying building blocks of construction; an "herculean" task that establishes the boundaries and foundations of what *can* and *cannot* be done. But engineers regard their art as something different: it is not the *why* so much as the *how*. While a scientist may regard two programs that are proven to be functionally equivalent to be one and the same, an engineer will rush to shout a torrent of *quality attributes* such as *usability*, *maintainability*, *performance*, *modularity*, *affordability*, among others, that somehow would render the two programs *completely different*.

And yet, here lies the nature of *design*, the Alexandrian *"Quality Without A Name"*, the forces that shape up a pattern, that yield an optimal solution for a good problem. The scientific treatment of the contributions in this dissertation may stand on the pillars of the scientific method, but, ultimately, they are the outcome of an engineering *necessity*.

That said, from childhood up until now, willingly or not, many people had defined what I am and the few things I was able to accomplish. Chronologically, my parents, who though not *technology-savvy*, traced my future the moment they handed me a computer. My *fiancée* Helena, who encouraged me to pursue my goals knowing their impact on both of our lives. My main supervisor Ademar Aguiar, who played a tremendous role in my studies and always ensured my research would be possible to pursue. My industrial co-supervisor Alexandre Sousa, who has given me the support and opportunity to enroll in the Ph.D. while still working for ParadigmaXis, S.A. My scientific co-supervisor João Pascoal Faria, whose work preceded mine in a very similar area. My external supervisor Joseph Yoder, one of the first (and current) researchers on Adaptive Object-Models, whose work inspired and lead me to this point. My fellow researchers Filipe Correia and Nuno Flores, currently pursuing their Ph.D in software engineering, with whom I shared hours of work and friendship. And to Ana Paiva, with whom I have had the pleasure of

lecturing formal methods in software engineering for the past three editions.

My gratitude also goes to the Department of Informatics Engineering (DEI) where I have been lecturing in the past two years, in the person of the head of department, Prof. Raul Vidal, for the invaluable support over my academic and professional life. And to INESC-P, for partially supporting my research.

Last, but not the least*, my acknowledgments to all my friends, colleagues, fellow researchers, working partners, students, and overall good chums, in alphabetical order: Alexandre Silva, Ana Mota, Andreia Teixeira, António Rito Silva, Aurélio Pires, Bruno Matos, David Pereira, Diogo Lapa, Diogo Romero, Fátima Pires, Gabriela Soares, Henrique Dias, Hugo Silva, João Ferreira, João Gradim, Joel Varanda, Jorge Pinheiro, José Pedro Oliveira, José Porto, José Vilaça, León Welicki, Paulo Cunha, Pedro Abreu, Rosaldo Rossetti, Rui Maranhão, Sérgio Mesquita, and Tiago Cunha. Finally, my gratitude to Cinthia Pagé and Paulo Romero, for helping in the french translation of the abstract.

Porto, 21 December 2010

* It seems my gratitude wouldn't be complete if I didn't acknowledged $C_8H_{10}N_4O_2$ and $C_9H_8O_4$, commonly known as *caffeine* and *aspirin*, which are known to be avid supporters of this research.

Chapter 1

Introduction

1.1 Software Crisis . . . 2
1.2 Motivational Example . . . 2
1.3 Incomplete by Design . . . 4
1.4 Accidental Complexity . . . 6
1.5 Designing for Incompleteness . . . 7
1.6 Software Design . . . 8
1.7 Patterns . . . 9
1.8 Research Goals . . . 10
1.9 Epistemological Stance . . . 10
1.10 Research Methods . . . 11
1.11 Main Goals . . . 12
1.12 How to Read this Dissertation . . . 13

The practice of software engineering is regarded as being in crisis since the early days of its existence, constantly plagued by project overruns and failures. Once *expectations* and *change* were identified as major contributing factors, most development processes urged their systematic eradication from software development. Other methodologies, especially those coined *agile*, took an opposite direction by rethinking the way software is built in order to *"embrace change"*. Either way — and since there is no *"silver bullet"* — evidence suggests that change may not be just a *cause*, but a *symptom* of a deeper nature. In fact, it has been observed that some software needs to change because the domain it is modeling is *incomplete* even before it starts to be developed. Such software has to be constantly *evolving* and *adapted* to a new reality from the moment it is conceived. Yet, if these systems are accepted and regarded as being *incomplete by design*, shouldn't they be actively *designed for incompleteness*? If there was the need to shift the way software is developed to *embrace change*, how should one deliberately design to cope with *continuous change*? What *extent* should it consider?

1.1 SOFTWARE CRISIS

Software — and the corresponding area of software engineering — seems to be in crisis since the early days of computing science. The rapid growth in both computer power and problem complexity has been resulting in a corresponding increase in the difficulty of writing *correct*, *understandable* and *verifiable* computer programs. This was the state of affairs presented in a NATO sponsored conference on software engineering circa 1968, where *unreasonable expectations* and *change* were underlined as major contributing factors [BBH69]. Edsger Dijkstra commented this same issue few years later, during his 1972 ACM Turing Award lecture [Dij72]:

> *The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

On the technology side, a similar concern was being shared by Alan Kay during the design of *Smalltalk* and *object-oriented programming* in the dawn of personal computers [Kay93]:

> *... there would be millions of personal machines and users, mostly outside of direct institutional control. Where would the applications and training come from? Why should we expect an applications programmer to anticipate the specific needs of a particular one of the millions of potential users? An extensional system seemed to be called for in which the end-users would do most of the tailoring (and even some of the direct constructions) of their tools* [*sic*].

And yet, despite the natural reaction on creating better processes, tools, and methodologies for software development, a 1987 *gödelian* exposure by Fred Brooks brings a general consensus among the software engineering community that there is no *"silver bullet"* — no single approach will prevent project overruns and failures in all possible cases [Bro87]. A conjecture somewhat disturbing in light of consecutive CHAOS reports [Sta94], where the success rate of software projects was estimated to be only 16%, with challenged projects accounting for 53%, and impaired (cancelled) for 31%  in 1994, and a success of 32%, 44% challenged, and 24% failed  in 2009[1].

1.2 MOTIVATIONAL EXAMPLE

In order to exemplify and further illustrate some of the central ideas and concepts of this dissertation, let us consider a real-world information system for a medical healthcare center, partially

[1] Although the exact nature of these figures has been target of recent criticism in [EV10], it seems that either their results are heavily biased, or even a moderate change in the accuracy of the success ratio, e.g. from 32% to 50%, would probably still render the field as *in crisis*.

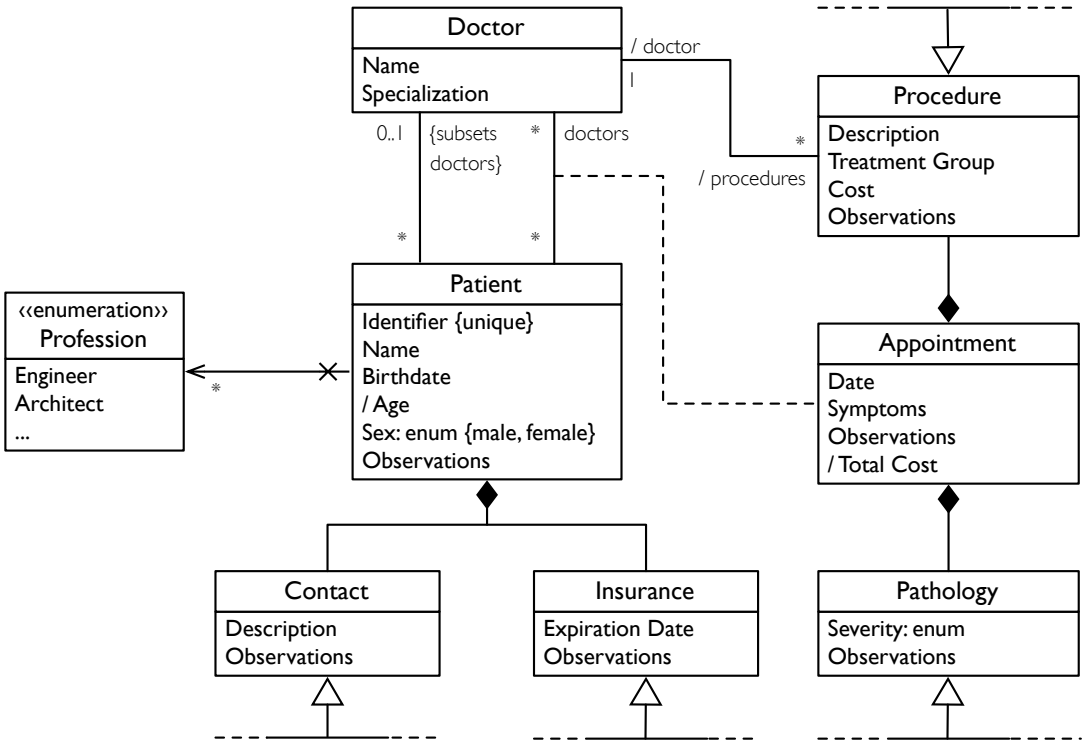


Figure 1.1: Example of a domain diagram of an information system for a medical center. The horizontal dashed lines denote open (incomplete) inheritances. The dots inside the enumeration also denotes incomplete knowledge which should be editable by the end-user.

depicted as a class diagram in Figure 1.1. Shortly, the medical center has Patients and specialized Doctors. Information about a patient, such as her personal information, Contacts and Insurances, are required to be stored. Patients go to the center to have Appointments with several Doctors, though they are usually followed by a main one. During an appointment, several Pathologies may be identified, which are often addressed through the execution of medical Procedures.

In this example, the reader should be able to observe a key property in this kind of information systems: *incompleteness*. For example, procedures, insurances, pathologies and contacts are specified as having open-hierarchies (where each specialization may require different fields). Patients may not have all the relevant information recorded (e.g., critical health conditions) and foreseeing those missed formalizations, either the designer or the customer make extensive usage of an observations field. The system may also be missing some domain concepts, such that of auxiliary personnel, which would require the introduction of a new class. The relevance of storing personal information about doctors may also have been overlooked; actually, in the presence of these new requirements, a designer would likely make Patients, Doctors and Auxiliary Personnel inherit from a single abstraction, e.g., Persons. The healthcare center likely has policies against doctors performing procedures for which they are not qualified, which would introduce specific constraints based on their specialization in the model. In fact, it now seems evident that a doctor may have multiple specializations.

These are examples of requirements that could easily elude developers and stakeholders during the analysis process. What may seem a reasonable, realistic and useful system at some point, may quickly evolve beyond the original expectations, unfortunately after analysis is considered finished (see Figure 1.2).

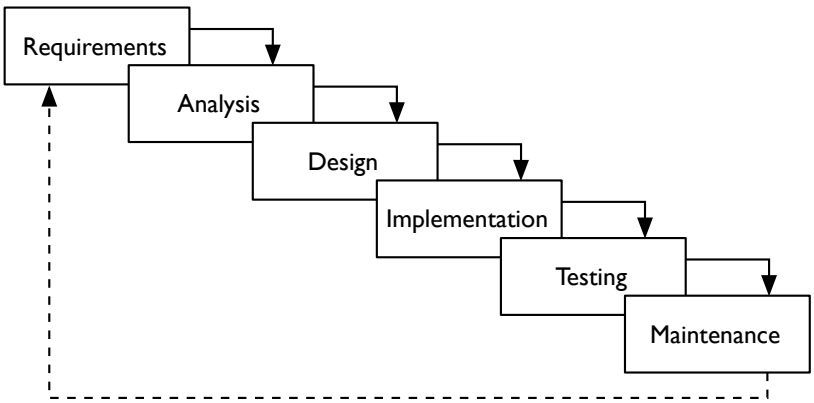


Figure 1.2: Waterfall software development lifecycle model as closely described in the work of Winston Royce [Roy87]. Some modified models introduce feedback loops (pictured as a dashed arrow), either connecting the end-points or between different phases.

1.3 INCOMPLETE BY DESIGN

The previous example represents a recurrent problem in software development: the difficulty of acquiring, inferring, capturing and formalizing requirements, particularly when designing systems where the process is highly coupled with the stakeholders' perspective and the requirements often change faster than the implementation. This reality, well known in industrial environments, is mostly blamed upon issues related to the stakeholders' ability to correctly elicit their needs [PT07], for the sole reason that maintaining and evolving software is a knowledge intensive task accounting for a significant amount of effort [AOSD07]. Consequently, once the analysis phase is finished and the implementation progresses, strong resistance to further change emerges, due to the mismatch between specification and implementation artifacts. Notwithstanding, from the stakeholder's perspective, some domains do rely on constant adaptation of their processes to an evolving reality, not to mention that new knowledge is continuously acquired, which leads to new insights of their own business and what support they expect from software.

Confronted with the above issues, some development methodologies (particularly those coined *agile*) have intensified their focus on a highly iterative and incremental approach [WC03]. From the subtitle *"embrace change"* in Kent Beck's EXTREME PROGRAMMING EXPLAINED book [BA04], to the motto *"responding to change over following a plan"* from the Agile Manifesto, these methodologies accept that *change* is, in fact, an invariant of software development.

This stance is in clear contrast with other practices that focus on *a priori*, time-consuming, rigorous design, considering continuous change as a luxurious (and somewhat dangerous) asset for the productivity and quality of software development (see Figure 1.3).

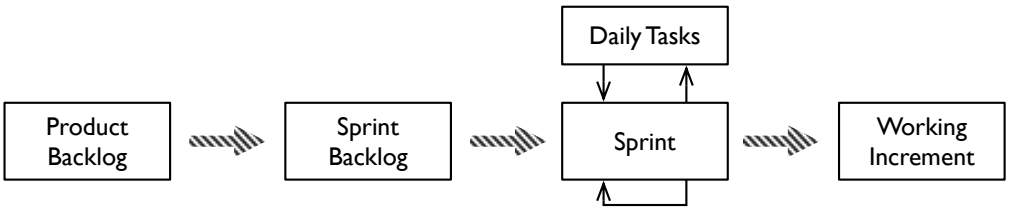


Figure 1.3: The Scrum software development lifecycle model, where product requirements reside on the backlog, and from where they are chosen at the beginning of every sprint. The sprint then follows a 2/4 weeks cycle, with daily tasks taken from the sprint backlog. Thus, the product is synthesized as a series of discrete increments. [TN86, DS90]

Although the benefits of an up-front, correct and validated specification are undeniable — and have been praised by formal methods of development, particularly when coping with critical systems — their approach is often recognized as impractical in a large majority of software projects [Hei98], particularly in environments characterized by continuous change. At the other end of the spectrum, the way many developers cope with change easily result in a Big Ball of Mud[2] [FY97] where systems are permanently *in the verge* of needing a total reconstruction, but either way with obvious impacts in the project's cost. For that reason, software that is target of continuous change should be regarded as *incomplete by design*, i.e., it needs to be constantly evolving and adapting itself to a new reality from the moment it is conceived, and most attempts to freeze its requirements are out-of-sync with its own condition — see Figure 1.4.

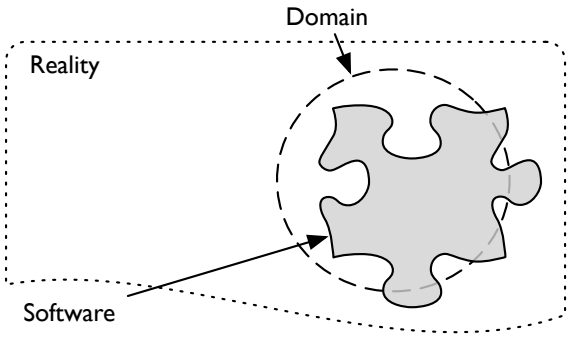


Figure 1.4: Software may be regarded as the crystallization of an abstraction that models a specific domain. Ideally, it should match the exact limits of that domain. But in practice (i) those limits are fuzzy, (ii) software often imposes an artificial, unnaturally rigid structure, and (iii) reality itself keeps changing.

[2] A software architectural pattern described by its authors as *"a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle"*.

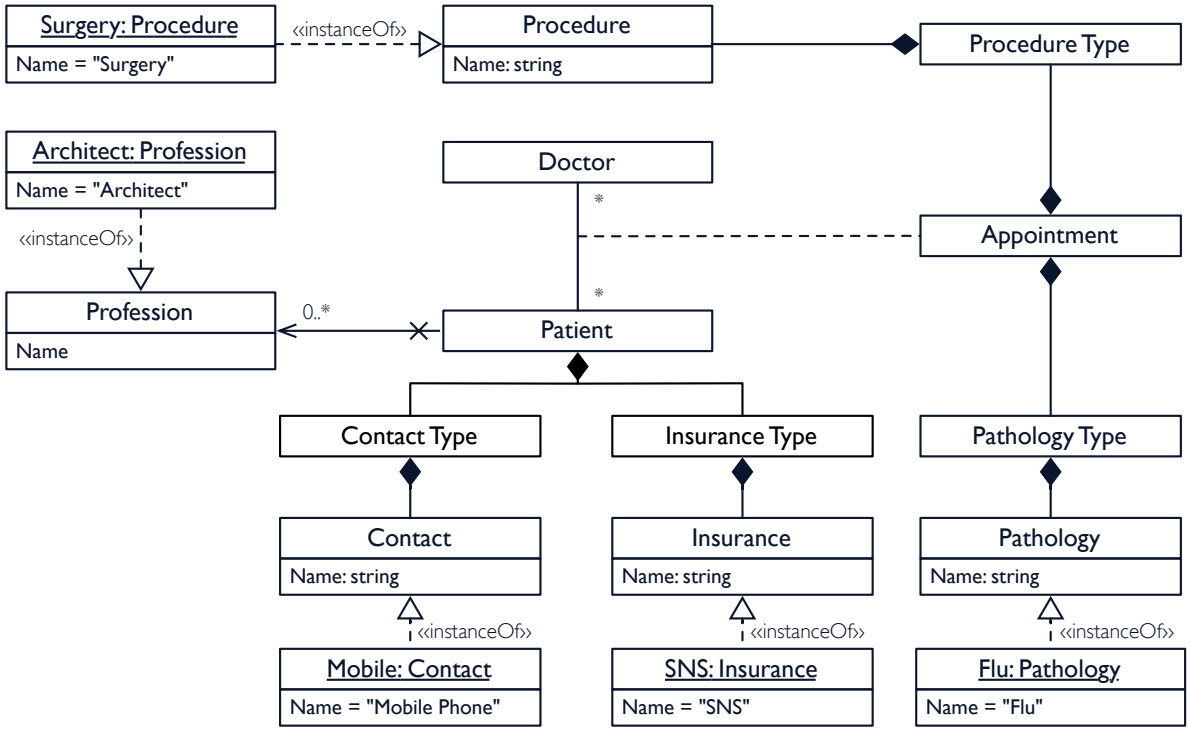


Figure 1.5: A *refactored* solution for the diagram in Figure 1.1 (p. 3), mainly depicting the elements that were changed/added for providing a mechanism to cope with open inheritance and enumerations. This example makes extensive use of the TYPE-OBJECT pattern to solve open inheritances and enumerations.

1.4 ACCIDENTAL COMPLEXITY

Should the system be required to cope with the kind of incompleteness displayed in § 1.2 (p. 2), the designer would have to deliberately make it *extensible* in appropriate points. A potential solution is depicted in Figure 1.5, comprising only the *refactored* elements intended to address the issues of *open inheritances* and *enumerations*.

Compared to the initial design seen in Figure 1.1 (p. 3), this one reveals itself as a much larger model. In fact, it is now more difficult to distinguish between elements that are *essential* to the specification of the domain, from those whose role is instrumental — in this case, that provide extensibility to the system. The result is an increase of what is defined as *accidental complexity* — complexity that arises in computer artifacts, or their development process, which is *non-essential* to the problem being solved. In contrast, the first model was much closer to that of *essential complexity* — inherent and unavoidable. This increase in accidental complexity was only caused by the specific approach chosen to solve the problem — in this case, recurrent usage of the TYPE-OBJECT pattern § 3.2.1 (p. 40), applied to solve open inheritances and enumerations.

While sometimes accidental complexity can be due to mistakes such as ineffective planning, or low priority placed on a project, some accidental complexity always occurs as the side effect of solving any problem. For example, mechanisms to deal with out-of-memory errors are part of the accidental complexity of most implementations, although they occur just because one has

decided to use a computer to solve the problem.

Notwithstanding, and to a great extent, most accidental complexity is introduced by the inappropriate actions used to cope with the evolution of software artifacts, hence resulting in a BIG BALL OF MUD. Because the minimization of accidental complexity is considered a *good* practice to any architecture, design, and implementation, excessive accidental complexity is a clear example of a *bad* practice.

1.5 DESIGNING FOR INCOMPLETENESS

While newer software engineering methodologies struggle to increase the ability to adjust easily to changes of both the process and the development team, generally they seem to maintain a certain agnosticism regarding the form of the produced software artifacts[3]. This doesn't mean they are not aware of this "need to change". In fact, iterative means several cycles going from analysis to development, and back again. Some are also aware of the BIG BALL OF MUD pattern (or anti-pattern, depending on the perspective); the practice of *refactoring* after each iteration in order to cope with *design debt* is specifically included to address that [NL06]. But the problem seems to remain in the sense that the outcome, w.r.t. *form*, of each iteration is mostly synthesized as if it would be the last one, albeit deliberatively recognizing it is not.

Yet, if these systems are accepted and regarded as being *incomplete by design*, it seems reasonable to assume they should be actively *designed for incompleteness*. If we shift the way we develop software to *embrace change*, it seems a natural decision to deliberately design that same software to best cope with *continuous change*. Citing the work of Garud et al. [GJT07]:

> *The traditional scientific approach to design extols the virtues of completeness. However, in environments characterized by continual change* [*new solutions*] *highlight a pragmatic approach to design in which incompleteness is harnessed in a generative manner. This suggests a change in the meaning of the word* design *itself — from one that separates the process of design from its outcome, to one that considers design as both the medium and outcome of action.*

This is in particular dissonance with most of the current approaches to software engineering, where processes attempt to establish a clear line between designing and developing, specifying and implementing. Though it seems that, should we wish to harness *continual change*, that distinction no longer suits our purposes: *design should become both the medium and outcome of action*. Ergo, we are looking forward not just for a process to be effective and agile, but to what *form* should agile software take.

[3] Probably an over-simplification since agile methodologies prescribe the *simplest design that works*, which somewhat addresses (without prescribing any specific) form.

1.6 SOFTWARE DESIGN

So far, the word *design* has been loosely used. But, what exactly *is* design? What makes the difference between *good* and *bad* design? How do we *measure* it? Why is it becoming a key factor in the development of sophisticated computer systems?

The Webster's dictionary defines it as the act of (i) *"working out the form of something"*, (ii) *"sketching a plan for something"*, (iii) *"creating something in the mind"*, or (iv) *"doing something for a specific role or purpose or effect"* [Web10]. Software engineering has also been faced with the task of sketching, planning, and creating suitable products for specific purposes. For that purpose, any specific solution drafted during the design of a software artifact typically considers forces such as (i) the experience of the team, (ii) the available budget, (iii) the exact functional and non-functional requirements, etc. More generally, software projects are recognized as having four major forces through which any particular balance (or imbalance) of them directly influences the viability of a specific solution, as depicted in Figure 1.6.

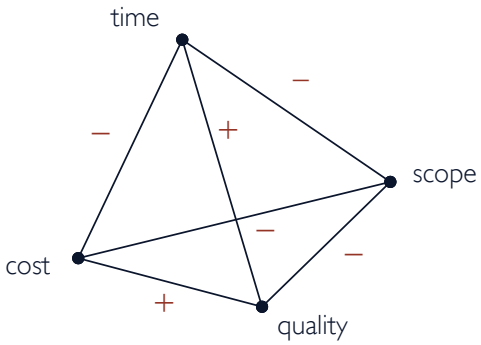


Figure 1.6: The four recognized forces of project management w.r.t. software engineering: time refers to the amount of time available to complete a project; cost refers to the budgeted amount available; scope refers the project's domain limit and detail. Each force mutually influences (either positively or negatively) every other. Quality is thus a result function of how the other forces are balanced.

But even taking into consideration these four major forces, the ever increasing complexity (both inherent and accidental) of building and managing software artifacts are pushing the limits of creation beyond the ability of a single entity [Ale64]. And similarly to every other areas of engineering, it is not uncommon for a single problem to have multiple ways to be coped with. So, in the end, how does an engineer choose the *best design* for a *specific function*?

As knowledge grows in a specific area, solutions are captured and documented by experts in books, research papers, concrete implementations, web pages, and a myriad of other types of communication media. While we may intuitively think that any growth in the body of knowledge implies better design choices, it seems that the way (and the amount) this knowledge is being captured raises an important issue *per-se*.

In THE PARADOX OF CHOICE: WHY MORE IS LESS, Schwartz points that the overabundance of choice that our modern society has poses a cause for *psychological distress*, where the *information overflow* ultimately results on an over-simplification of criteria [Sch05]. Preceding Schwartz in four decades, Alexander claimed, in his Ph.D. thesis NOTES ON THE SYNTHESIS OF FORM, that most information on any specific body of knowledge is *"hard to handle, widespread, diffuse, unorganized, and ever-changing"*. Moreover, this *"amount of information is increasingly growing beyond the reach of single designers"*. He concluded that *"although a specific solution should reflect all the known facts relevant to a specific design* (...) *the average designer scans whatever information he happens on and introduces it randomly"* [Ale64]. Design, as an intentional act of choice, is constantly overwhelmed by the sheer quantity of available information.

1.7 PATTERNS

As Alexander struggled with this scenario, he came up with the concept of *pattern*: a recurrent *solution* for a specific *problem*, that is able to achieve an *optimal balance* among a set of *forces* in a specific *context*. Further on his research, he extended this notion beyond the triplet $\langle problem, forces, solution \rangle$, to that of a *pattern language*, which also encompasses the relationship between each pattern in a specific domain. This effort resulted in the first pattern language on civil architecture[4] to be compiled and released as a book [AIS77].

Almost two decades later, both Gamma et al. [GHJV94] and Buschman et al. [BMR+96] borrowed these concepts into the fields of computer science and software engineering, publishing the first books on software *design* and *architectural* patterns. These two types of patterns, along with a third called *idioms*, can be defined as follows [BMR+96]:

- **Architectural patterns** express fundamental structural organization schemes for software systems, decomposing them into *subsystems*, along with their responsibilities and interrelations.

- **Design patterns** are medium-scale tactical patterns, specified in terms of interactions between elements of object-oriented design, such as *classes*, *relations* and *objects*, providing generic, prescriptive templates to be instantiated in concrete situations. They do not influence overall system structure, but instead define micro-architectures of subsystems and components.

- **Idioms** (sometimes also called coding patterns) are low-level patterns that describe how to implement particular aspects of components or relationships using the features of a specific programming language.

[4] It should be noted that Alexander was a civil architect, and thus his theories emerged from the observation of cities and buildings, although he had a M.Sc. in Mathematics.

Built on top of Alexander's theories, these resulting pattern languages were synthesized by systematic analysis and documentation of scattered empirical knowledge, and had hitherto a profound impact in the way architects and designers build and manage software artifacts today.

1.8 RESEARCH GOALS

Up until this point, the author has presented arguments pointing to how *incomplete by design* systems are a common issue in nowadays practice of software engineering. However, it is not part of this study to prove where — nor why — these kind of systems emerge. Likewise, this dissertation will just slightly delve into the forces that may lead an architect to recognize these kind of systems. Instead, it is assumed the following set of premises, which upon this work is built: (i) the system we are working with exhibits an high degree of variability, (ii) its specifications have shown a high degree of incompleteness, and (iii) it is desirable to reduce the effort[5] of coping with changes made to the domain model. Should one agree with these premises, the author's fundamental research question may be stated as ***what form should this type of systems take, and which kind of tools and infrastructures should be available to support the development of such software systems***?

Consequently, the main goal is to research this *form*, here to be understood as the *architecture* and *design* of such software systems, along with the specification and construction of the appropriate tools and infrastructure.

1.9 EPISTEMOLOGICAL STANCE

To understand the way software engineers build and maintain complex and evolving software systems, research needs to focus beyond the tools and methodologies. Researchers need to delve into the social and their surrounding cognitive processes *vis-a-vis* individuals, teams, and organizations. Therefore, research in software engineering may be regarded as inherently coupled with *human activity*, where the value of generated knowledge is directly dependent on the methods by which it was obtained.

Because the application of reductionism to assess the practice of software engineering, particularly in field research, is a complex — perhaps unsuitable — activity, this dissertation is aligned with a *pragmatistic* view of acquisition of knowledge, valuing acquired practical assets [Gouo8]. In other words, the author chooses to use whatever methods seem to be more appropriate to prove — or at least improve knowledge about — the questions raised, provided their scientific significance.

[5] The intended meaning of *effort* should be loosely interpreted as monetary cost, available time and resources, required skills, resulting complexity, etc.

Mostly, though, this perspective makes extended use of mixed methods, such as (i) systematization of widespread, diffuse, and unorganized empirical best practices through *observational* and *historical* methods, (ii) laboratorial (quasi-)experiments, usually suitable for academic environments, and (iii) industrial case-studies, as both a conduit to harvest practical requirements, as to provide a tight feedback and application over the conducted investigation.

1.10 RESEARCH METHODS

A categorization proposed at Dagstuhl workshop [THP93], groups research methods in four general categories, quoted from Zelkowitz and Wallace [ZW98]:

- **Scientific method.** "Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis."
- **Engineering method.** "Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement."
- **Empirical method.** "A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis."
- **Analytical method.** "A formal theory is developed, and results derived from that theory can be compared with empirical observations."

These categories apply to science in general. Effective experimentation in software engineering requires more specific approaches. As discussed in § 1.9 (p. 10), software engineering research comprises computer science issues, human issues and organizational issues. It is thus often convenient to use combinations of research approaches both from computer science and social sciences. The taxonomy described by Zelkowitz and Wallace [ZW98] identifies twelve different types of experimental approaches for software engineering, grouped into three broad categories:

- **Observational methods.** "An observational method collects relevant data as a project develops. There is relatively little control over the development process other than through using the new technology that is being studied". There are four types: project monitoring, case study, assertion, and field study.
- **Historical methods.** "A historical method collects data from projects that have already been completed. The data already exist; it is only necessary to analyze what has already been collected". There are four methods: literature search, legacy data, lessons learned, and static analysis.

- **Controlled methods.** "A controlled method provides multiple instances of an observation for statistical validity of the results. This method is the classical method of experimental design in other scientific disciplines". There are four types of controlled methods: replicated experiment, synthetic environment experiment, dynamic analysis, and simulation.

There are, however, particular research approaches, methods and outcomes regarding the study of software architectures, which can be grouped into five types of *products*, quoted from Shaw [Sha01]:

- **Qualitative or descriptive model.** "Organize and report interesting observations about the world. Create and defend generalizations from real examples. Structure a problem area; formulate the right questions. Do a careful analysis of a system or its development". Examples: early architectural models, and architectural patterns.
- **Technique.** "Invent new ways to do some tasks, including procedures and implementation techniques. Develop a technique to choose among alternatives". Examples: product line and domain-specific software architectures, and UML to support object-oriented design.
- **System.** "Embody result in a system, using the system development as both source of insight and carrier of results". Examples: architecture description languages.
- **Empirical predictive model.** "Develop predictive models from observed data".
- **Analytic model.** "Develop structural (quantitative or symbolic) models that permit formal analysis". Examples: HLA specification, and COM inconsistency analysis.

The best combination of methods to use in a concrete research approach is strongly dependent on the specific characteristics of the research study to perform, viz. its purpose, environment and resources. Hereafter, the research methods referred will use this terminology. Further description of each method can be found in [ZW98], and a detailed rational on the methods chosen in Chapter 4 (p. 47).

1.11 MAIN GOALS

The primary outcomes of this thesis encompasses the following aimed contributions to the body of knowledge in software engineering:

1. **The formalization of a pattern language for systems which domain model is target of continuous change during runtime.** When do these type of systems occur? What are the advantages? What are their underlying requirements? How do we cope with each of them? What are the benefits and liabilities of each specific solution? This contribution expands

an unified conceptual pattern language, which allows architects and designers to recognize and adequately use some of the best practices in software engineering that cope with this type of systems. Further details are presented in Chapter 5 (p. 59).

2. **The specification of a reference architecture for adaptive object-model frameworks.** What kind of infrastructures are needed? What form should they take? What type of abstractions should be made and supported? What are the generic functionalities it should provide? What should be its default behavior? How can it be extended? This contribution addresses several issues concerning framework design for Adaptive Object-Models, and presents a solution through the composition of architectural and design elements. More details can be found in Chapter 6 (p. 93).

3. **A reference implementation of such framework.** From theory to practice, Chapter 6 (p. 93) also details a concrete implementation of a framework based on the proposed reference architecture, codenamed *Oghma*. The goal of attaining an industrial-level implementation of such framework, along with the research of specific design issues that arise only when pursuing such concrete implementations, allowed to further pursue the research using the chosen validation methodologies.

4. **Evidence of the framework benefits through industrial use-case applications.** A framework should emerge from reiterated design in real-world scenarios. As such, the implementation shown in Chapter 6 (p. 93) is mainly the result of an incremental engineered solution for specific industrial applications. This contribution presents software systems built on top of that framework, their context, their requirements, their particular problems, the way the framework was used to address them, the outcomes, and the lessons learned in Chapter 7 (p. 115).

5. **Evidence of the framework properties through controlled academic quasi-experiments.** Although the industrial usage of the framework provides pragmatic evidence of its benefits, there are some threats that are inherent to that type of validation. These shortcomings are addressed in Chapter 8 (p. 129), by conducting a (quasi-)experiment within a controlled academic experimental environment, where the study of groups of undergraduate students interacting with the framework has shown the results to be consistent with those presented in Chapter 7 (p. 115).

1.12 HOW TO READ THIS DISSERTATION

The remaining of this dissertation is logically organized into three parts, with the following overall structure:

Part 1: Background & State of the Art. The first part reviews the most important concepts and issues relevant to the thesis, namely:

- Chapter 2, "Background" (p. 15), provides an extensive literature survey on software engineering techniques, tools, and practices able to cope with *incomplete by design* systems.
- Chapter 3, "State of the Art in Adaptive Object-Models" (p. 35), provides a state-of-the-art overview mostly focused on Adaptive Object-Models.

Part 2: Problem & Solution. The second part states the problem researched and the proposed solution:

- Chapter 4, "Research Problem" (p. 47), lays both the fundamental and specific research questions in scope for this thesis.
- Chapter 5, "Pattern Language" (p. 59), presents an unified conceptual pattern language for Adaptive Object-Models.
- Chapter 6, "Reference Architecture & Implementation" (p. 93), composes several architectural and design elements into a framework based on the pattern language, codenamed *Oghma*, and describes its current implementation.

Part 3: Validation & Conclusions. The third part presents case studies and one (quasi-)experiment for the validation of the thesis, and presents the conclusions of the dissertation:

- Chapter 7, "Industrial Case-Studies" (p. 115), presents a detailed analysis on using this framework in industrial settings.
- Chapter 8, "Academic Quasi-Experiment" (p. 129), addresses validation issues through controlled experimental environments.
- Chapter 9, "Conclusions" (p. 151), drafts the main conclusions of this dissertation, and points to further work.

For a comprehensive understanding of this dissertation[6], all the parts should be read in the same order they are presented. Those already familiar with meta-architectures and adaptive object-models, who only want to get a fast but detailed impression of the work, may skip the first part, and go directly to Chapter 4 (p. 47).

[6] Some typographical conventions are used to improve the readability of this document. Pattern names always appear in SmallCase style. Whenever referring to programming or modeling elements, such as classes or property names, they are printed using fixed-width characters. Relevant concepts are usually introduced in *italics*. Book titles and acronyms are type-faced in ALLCAPS. References and citations appear inside [square brackets] and in highlight color — when viewing this document in a computer, these will also act as *hyperlinks*. If not otherwise specified, the graphical notation used complies to the latest versions of UML [OMG10d] and OCL [OMG10c] available at the date of publication (v.2.3).

Chapter 2

Background

2.1 Fundamentals 17
2.2 Key Abstractions 18
2.3 Approaches 26
2.4 Application Frameworks 29
2.5 Relevant Tools 31
2.6 Conclusion 34

The current demand for industrialization of software development is having a profound impact in the growth of software complexity and time-to-market. Still, most effort in the development of software is repeatedly applied to the same tasks. Despite all the effort in research for more effective *reuse* techniques and practices, which although remain a promising approach to the efficient development of high quality software, *reuse* has not yet fulfilled its true potential [HC01, Szy02, Gou08]. As in other areas of scientific research, the reaction has been to hide the inherent complexities of technological concerns by creating increasingly higher levels (and layers) of abstractions with the goal of helping reasoning, albeit often at the cost of widening the already existing gap between specification and implementation artifacts [FR07].

To make these abstractions useful beyond documentation, analytical and reasoning purposes [AEQ99, KOS07], higher-level models must be made executable, by systematic transformation (or interpretation) of problem-level abstractions into software implementations [RFBLO01, Völ03]. The primary focus of model-driven engineering (MDE) is to find ways of automatically animating such models, promoting them from auxiliary, document-level diagrams, to first-class software artifacts able to drive complex systems at multiple levels of abstraction and perspectives [FR07].

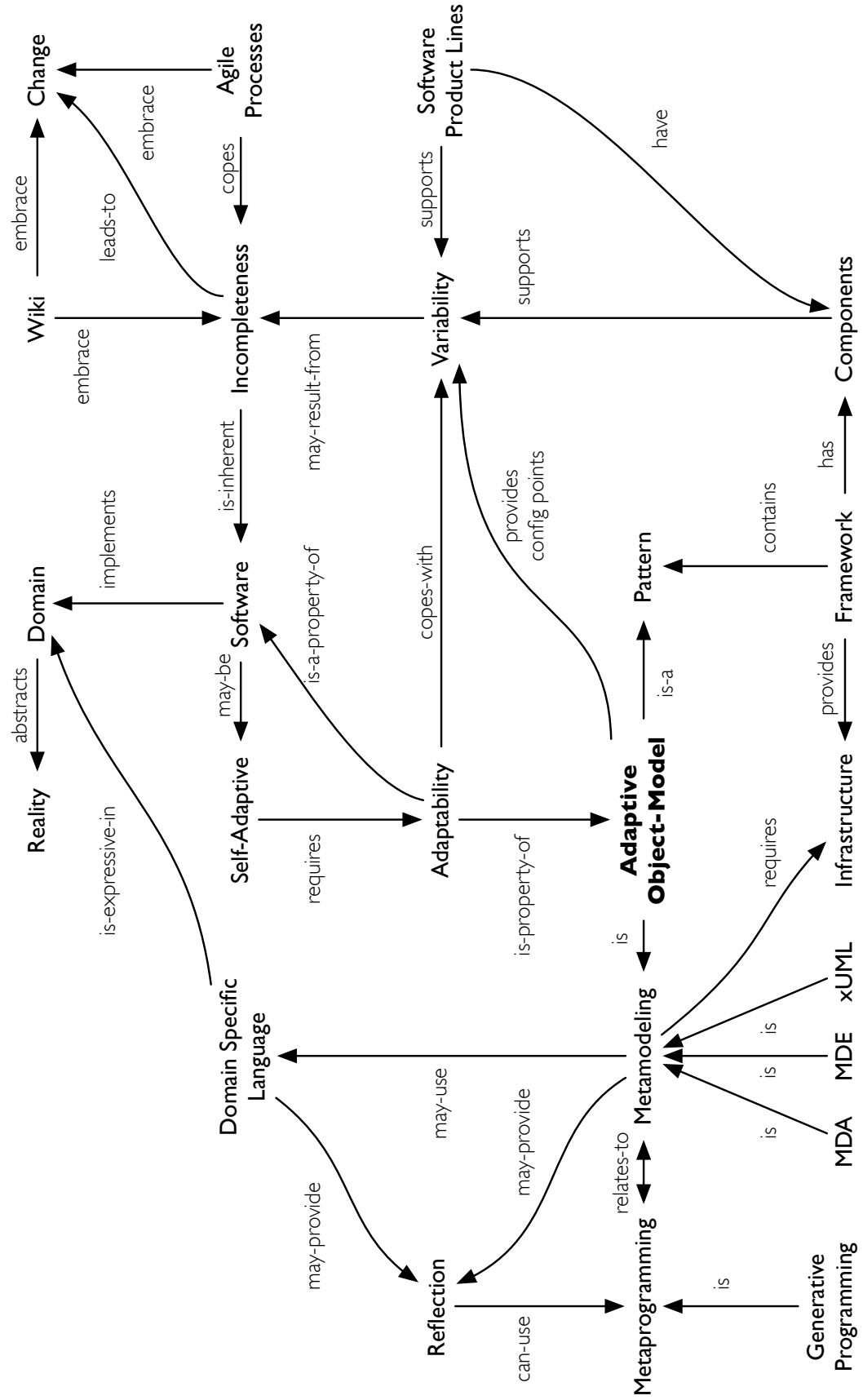


Figure 2.1: Concept map representing the relationship between several areas, concepts and techniques presented in this chapter.

2.1 FUNDAMENTALS

One way to design software capable to cope with incompleteness and change is to encode the system's concepts into higher-level representations, which could then be systematically synthesized (by interpretation or transformation) into executable artifacts, thus reducing the overall effort (and complexity) of changing it. An overview of the several concepts that will be approached in this section is shown in Figure 2.1 (p. 16). In summary, *variability* and *adaptability* are two key principles, usually related to *self-adaptive* software and *software product lines*. These principles are highly related to the inherent *incompleteness* of some software *domains*. *Wikis* and *Agile* processes, for examples, are known to *embrace change* and thus coping with incompleteness. On the technology side, one can find (i) *metaprogramming*, which is usually related to *generative programming* and *reflection*, but whose artifacts are close to programming languages, and (ii) *metamodeling*, usually related to *Model Driven Engineering*, *Model Driven Architectures* and *Executable UML*, whose artifacts are more close to the domain. Both, but especially the latter, require the appropriate *infrastructure* to be used. *Frameworks* provide such infrastructures by assembling *components* and *patterns*. A very close concept is the *domain specific languages*, which is an abstraction especially tailored to be highly expressive in a particular domain.

2.1.1 Variability and Commonality

Software *variability* represents the need of a software system or artifact to be *changed*, *customized* or *configured* for use in different contexts [JVBS01]. *High variability* means that the software may be used in a broader range of contexts, i.e., the software is more reusable. The degree of variability of a particular system is given by its *Variation Points*, or roughly the parts that support (re)configuration and consequently *tailoring* of the product for different contexts or for different purposes. On the other hand, the identification of what is *subject to change* is intimately related to that of what *does not change*, i.e., *commonality*. Variability and commonality are base concepts in *Software Product Lines*, which will be covered in § 2.3.1 (p. 26).

2.1.2 Adaptability, Adaptivity and Self-Adaptation

While variability is given by context, i.e. it is the target applicability of a software system, the property of such systems to be efficiently reconfigured due to changed circumstances is called *adaptability*. Im other words, a piece of software may be reconfigured by developers, to be deployed and used in different contexts (e.g., by recompiling with an additional component or through a component-based architecture), and this need is called variability as discussed in § 2.1.1. Adaptability, on the other hand, is the property that allow software to change its behavior by end-users with limited, or non-existent, programming skills.

The literature also mentions *adaptivity*, which is a broader property where a computer system is able to adapt itself to changes in internal, user-related, or environmental characteristics. The difference to *adaptability*[1] is that it may be more pro-active in the suggestion — or execution — of a particular change, hence not strictly dependent on human intervention. A more specific case of adaptivity is *self-adaptation*, when systems are empowered with the ability to adjust *their own* behavior in response to their perceptions of *self*, or the environment, without further human intervention [CLG+09].

All these properties, or capabilities, usually rely on a certain degree of *introspection* and *intercession* – two aspects of *reflection* — which will be further discussed in § 2.1.3. The work by Andresen *et al.* [AG05] identifies some enabling criteria for adaptability in enterprise systems. Further work by Meso *et al.* [MJ06] provides an insight into how agile software development practices can be used to improve adaptability.

2.1.3 Reflection

Reflection is the property of a system that allows to observe and alter its own structure or behavior during its own execution. This is normally achieved through the usage and manipulation of (meta-)data representing the state of the program/system. There are two aspects of such manipulation: (i) *introspection*, i.e., to observe and reason about its own state, and (ii) *intercession*, i.e., to modify its own execution state (structure) or alter its own interpretation or meaning (semantics) [Caz98]. A simple example using both introspection (to find a class by name) and intercession (by creating a new instance of a class) can be observed in Source 2.1. These capabilities makes reflection a key property for metaprogramming, as further exposed in § 2.2.1 (p. 20).

```
1 Foo.new.hello                                  # without reflection
2 Object.const_get(:Foo).send(:new).send(:hello) # with reflection
```

Source 2.1: Two ways of invoking a method from a newly created instance in Ruby.

2.2 KEY ABSTRACTIONS

Abstraction is a very broad concept, which definition depends on the concrete area of application. Quoting several articles from [Wik10a]:

- **Conceptually,** "it is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose".

[1] This difference is not always evident, due to a relaxed usage of both terms.

- **In Mathematics,** "it is the process of extracting the underlying essence of a mathematical concept, removing any dependence on real world objects with which it might originally have been connected, and generalizing it so that it has wider applications or matching among other abstract descriptions of equivalent phenomena".
- **In Computer Science,** "it is the mechanism and practice of reducing and factoring out details so that one can focus on a few concepts at a time".
- **In Software Engineering,** "it is a principle that aims to reduce duplication of information in a program (usually with emphasis on code duplication) whenever practical by making use of abstractions provided by the programming language[2]".

All these definitions share one factor in common — that *abstraction* involves relinquishing some property, such as *detail*, to gain or increase another property, like *simplicity*. For example, a common known use of the word *abstraction* is to classify the level of a programming language: Assembly is often regarded as *low-level* because it exposes the underlying mechanisms of the machine with an high degree of fidelity; Haskell, on the other hand, is a *high-level* language, struggling to hide as much as possible the underlying details of its execution. The latter trades execution *performance* in favor of *cross-platform* and *domain expressiveness*, among others. And yet, raising the level of abstraction seems to be the future of programming languages, as exposed by C#'s lead designer Anders Hejlsberg [BW09]:

> *…the ongoing trend in language evolution has always been this constant increase of abstraction level. If you go all the way back to plugboards and machine code and then symbolic assemblers and then C and then C++ and now managed execution environments and so forth, each step we moved a level of abstraction up. The key challenge always is to look for the next level of abstraction.*

However, abstractions should hardly be considered *win-win* solutions. Joel Spolsky observes a recurrent phenomena in technological abstractions called *Leaky Abstractions*, which occurs when one technological abstraction tries to completely hide the concepts of another, lower-level technology, and sometimes the underlying concepts *"leak through"* those supposed invisible layers, rendering them visible [Spo02]. For example, an high-level language may try to hide the fact that the program is being executed at all by a *von-neumann* machine. But, although two programs may be functionally equivalent, memory consumption and processor cycles may eventually draw a clear and pragmatic separation between them. The programmer may thus need to learn about the middle and lower-level components (i.e., processor, memory, compiler, etc.) for the purpose of *designing a program that executes in a reasonable time*, thus breaking the abstraction layer. Spolsky goes as far as conjecturing that *"all non-trivial abstractions, to some degree, are*

[2] Also known as the *"don't repeat yourself"* principle, or eXtreme Programming's *"once and only once"*.

leaky". Hence, good abstractions are specifically designed to express the exactly intended details in a specific context, while relinquishing what is considered unimportant, quoting [AS96]:

> *We must constantly turn to new languages in order to express our ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand.*

Despite one's tendency to interpret "new languages" as programming languages, that is not necessary the case. In fact, a pattern language *"enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand"* § 1.7 (p. 9).

2.2.1 Metaprogramming

Metaprogramming[3] consists on writing programs that generate or manipulate either other programs, or themselves, by treating *code* as *data*. Quoting [CI84]:

> *A metaprogramming system is a programming facility (subprogramming system or language) whose basic data objects include the programs and program fragments of some particular programming language, known as the target language of the system. Such systems are designed to facilitate the writing of metaprograms, that is, programs about programs. Metaprograms take as input programs and fragments in the target language, perform various operations on them, and possibly, generate modified target-language programs as output.*

Historically, metaprogramming systems were divided into two separate languages: (i) the meta-language, in which the meta-program is written, and (ii) the object language which the metaprogram produces or manipulates. Nowadays, most programming languages use the same language for the two functions [Caz98], either (i) by being *homoiconic* such as LISP, (ii) being *dynamic* such as Ruby and Python, or (iii) by exposing the internals of the runtime engine through APIs, such as Java and Microsoft .NET.

One of the most common applications of metaprogramming is in the translation of high-level descriptions into low-level implementations by means of *application generators*, as further discussed in § 2.2.2 (p. 21). Such techniques allows developers to focus on specifications based on tested standards rather than in implementation details, making tasks like system maintenance and evolution more easy and affordable [Bas87, Cle88]. Claims about the economic benefits in terms of development and adaptability have been studied and published for more than twenty

[3] Or *meta-programming*; the usage of the *hyphen* after the prefix *meta* varies among authors. The same applies for *meta-modeling*, *meta-program*, *meta-data*…

years [Lev86]. However, the focus of these techniques is mostly directed to the code level rather than high-level domain concepts.

A simple example of using metaprogramming to implement a factorial function whose value is pre-calculated at compile-time can be found in Source 2.2. This example makes usage of C++ techniques such as *template metaprogramming* and *template specialization*.

```
template <int N>
struct Factorial { enum { value = N * Factorial<N - 1>::value }; };

template <>
struct Factorial<0> { enum { value = 1 }; };

void foo() {
    int x = Factorial<4>::value; // == 24
    int y = Factorial<0>::value; // == 1
}
```

Source 2.2: Calculating a factorial function at compile-time by using C++ templates.

An amusing and somewhat *mind-twisting* example is that of a program which produces a copy of its own source code as its only output; these *metaprograms* are known as *quines*[4], and a quine written by John McCarthy in LISP can be found in Source 2.3.

```
((lambda (x)
   (list x (list (quote quote) x)))
  (quote
     (lambda (x)
       (list x (list (quote quote) x)))))
```

Source 2.3: An example of a LISP program which produces a copy of its own source code.

2.2.2 Generative Programming

One common approach to address variability and adaptability is the use of Generative Programming (GP) methods, which transform a description of a system (model) based in primitive structures [RKS98], into either executable code or code skeleton. This code can then be further modified and/or refined and linked to other components [Cza04]. Generative Programming deals with a wide range of possibilities including those from Aspect Oriented Programming [KH01, DYBJ08], § 2.2.4 (p. 23), and Intentional Programming [CE00].

[4] A term popularized by Douglas Hofstadter in his book GÖDEL, ESCHER, BACH: AN ETERNAL GOLDEN BRAID [Hof79], named after philosopher Willard Van Orman Quine due to his paradox-producing expression: "*Yields falsehood when preceded by its quotation*" *yields falsehood when preceded by its quotation*.

Because generative approaches focus on the automatic generation of systems from high-level (or, at least, higher-level) descriptions, it is arguable whether those act like a meta-model of the generated system. Still, the resulting functionality is not directly produced by programmers but specified using domain-related artifacts. In summary, GP works as an off-line code-producer and not as a run-time adaptive system [YBJ01a].

This technique typically assumes that (i) changes are always introduced by developers (change agents), (ii) within the development environment, and (iii) that a full transformation (and most likely, compilation) cycle is affordable (i.e., no run-time reconfiguration). When these premises fail to hold, generative approaches are either unsuitable, or requires additional infrastructures such as *incremental compilation* [RFBLO01].

Perhaps one of the biggest drawbacks in GP is on round-trip engineering (RTE): the synchronization among multiple artifacts that represent the same information, e.g. a model and the generated code. When the information present in multiple artifacts is open to change, then inconsistencies may occur if those artifacts are not consistently updated to reflect any introduced change, due to their *causal connections*[5]. A common example is in maintaining a piece of code and its UML representation synchronized.

2.2.3 Metamodeling

Metamodeling can be defined as the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems [SV06]; in other words, it is the practice of constructing models that specify other models. Metamodelling is closely related to, and to some extent overlapping, ontology analysis and conceptual modeling, since both are often used to describe and analyze relations between concepts [SAJ+02].

A model is said to conform to its metamodel in the way that a particular computer program conforms to the grammar of the programming language in which it is written. Quoting [GPHS08]:

> *In epistemology and other branches of philosophy, the prefix "meta-" is often used to mean "about its own category"; in* [*metamodeling,*] *"meta-" means that the model that we are building represents other models i.e. a metamodel is a model of models. This relationship is often described as paralleling the Type–Instance relationship. In other words, the metamodel (Type) and each of the models (Instances) are of "different stuff" and are described using different languages (natural or software engineering languages). Such type models therefore use classification abstraction — which leads to a metamodelling hierarchy.*

[5] Consider a common GP system that transforms UML into code; in total, there's three potential issues to manage, viz. (i) *forward engineering*, where the model changes so the code needs to be re-generated, (ii) *reverse engineering*, where the code was changed so the model needs to be re-fitted, and (iii) *round-trip engineering*, where both the model and the code has been changed, so they need to be reconciled.

Common uses for metamodels include (i) schemas for semantic data that needs to be exchanged or stored, (ii) languages that supports a particular method or process, (iii) languages that express additional semantics of existing information [Wik10c]. The usage of metamodeling techniques during runtime provides an answer to *high-variability* systems, where the large semantic mismatch between specification and implementation artifacts in traditional systems can be reduced by the use of models, metamodels, and metadata in general [RFBLO01].

2.2.4 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm that isolates secondary or auxiliary behaviors from the implementation of the main business logic. It has been proposed as a viable implementation of modular crosscutting concerns. Since crosscutting concerns cannot be properly modularized within object-oriented programming, they are expressed as aspects and are composed, or *woven*, with traditionally encapsulated functionality referred to as components [KH01, KM05]. This paradigm has been gaining some momentum particularly because even conceptually simple crosscutting concerns such as tracing during debugging and synchronization, lead to tangling, in which code statements addressing the crosscutting concern become interlaced with those addressing other concerns within the application [LC03].

Many AOP implementations, such as those related to Python, work by automatically injecting new code into existing classes (a process which is commonly called decorating a class), effectively adding new functionalities that promote extensive code reuse [MS03]. An example of such can be found in Source 2.4, using the Aspyct library [ddB10].

```
1 class MyAspect(Aspect):
2     def atCall(self, cd):
3         print("atCall occurs now")
4
5 @MyAspect()
6 def test():
7     print("Hello World!")
```

Source 2.4: An example using Aspect Oriented Programming in Python for tracing purposes.

Despite the growing popularity of AOP, much of its success seems to be based on the conception that it improves both modularity and the structure of code, while in fact, it has been observed to work against the primary purposes of the two, namely independent development and understandability of programs, up to the point of rendering it as almost paradoxical [Ste06].

2.2.5 Domain Specific Languages

A domain-specific language (DSL) is a programming or specification language specifically designed to suit a particular problem domain, representation technique, and/or solution technique [Par07]. They can be either (i) visual diagramming languages, such as UML, (ii) programatic abstractions, such as EMF, or (iii) textual languages, such as SQL.

The benefits of creating a DSL (along with the necessary infrastructure to support its interpretation or execution) may reveal considerable whenever the language allows a more clear expression of a particular type of problems or solutions than pre-existing languages would, and the type of problem in question reappears sufficiently often (i.e., recurrent, either in a specific project, like extensive usage of mathematical formulae, or global-wise, such as database querying). But despite its benefits, DSL development is hard, requiring both domain knowledge and language development expertise. Hence, the decision to develop a DSL is often postponed indefinitely, if considered at all, and most DSLs never get beyond the application library stage [MHS05].

Recently, the term DSL has also been used to coin a particular type of syntactic construction within a general purpose language which tends to more naturally resemble a particular problem domain, but without actually extending or creating a new grammar. The Ruby community, for example, has been enthusiastic in applying this term to such syntactic sugar. This has divided DSLs into two different categories:

- **External,** are built more likely a generic-purpose language. They have its own custom syntax and are developed with the support of tools such as ANTLR [PQ95] or YACC [Joh78], which take a formalized grammar (e.g., defined in a meta-syntax such as BNF), and generate parsers in a supported target language, such as C. More recent systems, like JetBrain's MPS [Jet10], provide *MetaIDE*s that allows one to design new languages — and extend existing ones — along with modern instrumental tools such as an editor, code completion, find usages, debugger, etc.

- **Internal,** also known as *embedded* DSL, is a technique where one uses a host language by strictly complying with its grammar, but in a way that it *feels* it is other language. As said, this specific approach has been recently popularized by the Ruby community, although it is possible to find earlier examples in LISP and Smalltalk [Sch07].

Domain-specific languages share common design goals that contrast with those of general-purpose languages, in the sense they are (i) less comprehensive, (ii) more expressive in their domain, and (iii) exhibit minimum redundancy. Language Oriented Programming [War94] considers the creation of special-purpose languages for expressing problems as a standard methodology of the problem solving process.

2.2.6 Meta-Architectures

We have already seen several techniques used to address systems with high-variability needs. There is, nonetheless, differences between them. For example, some do not parse or interpret the system definition (meta-data) while it is running: Generative Programming and Metamodeling rely on code generation done at compile time. Reflection is more of a property than a technique by itself, and the level at which it is typically available (i.e., programming language) is inconvenient to deal with domain-level changes. Domain Specific Languages are programming (or specification) languages created for a specific purpose. They are not generally tailored to deal with change (though they could), and they do require a specific infrastructure in order to be executed.

Meta-architectures are one of those *Humpty-Dumpty*[6] words. Yoder *et al.* [YBJ01b] defined it as architectures that can dynamically adapt at runtime to new user requirement, also known as "reflective architectures". Ferreira *et al.* [FAF09] defined it as an "architecture of architectures, i.e., an abstraction over a class of systems which usually rely on reflective mechanisms". This seemingly disagreement is due to the *meta*[7] prefix, which can be understood as being applied to the word architecture (i.e., an architecture of architectures), or as a subset categorization (i.e., those architectures that rely on *meta* mechanisms).

For the sake of simplicity, we will always refer to the latter meaning henceforth. Thus, meta-architectures are architectures that strongly rely on reflective properties, and usually have the ability to dynamically adapt to new user requirements during runtime (through intercession). Both pure object-oriented environments, and MOF-based systems are examples of such architectures, as they make use of meta-data to create different levels that sequentially comply to each other. But, in these cases, the line that separates what is data and is a systems definition — i.e., loosely called it's model — is blurred, so we need to establish further vocabulary conventions. Whenever we talk about data (or instances) we are drafting a parallel with M_0 — bare information that doesn't provide structure. By model we are referring to M_1 — its structure gives meaning to data. By meta-model we are referring to M_2 — a model to define models. The "top-most" level doesn't have a name, though; in MOF is called meta-meta-model, due to it being the third model layer. This building up of levels (or layers), where each is directly accountable for providing structure and meaning to the layer below, resembles a tower, and so it is also commonly called *Reflective Tower*, as depicted in Figure 2.2 (p. 29).

Nonetheless, it should be noted that the property of being *reflective* may actually be regarded as the key principle of meta-architectures, since systems that adapt their behavior do not necessarily rely of *meta* mechanisms.

[6] From the Lewis Carroll's DOWN THE RABBIT HOLE, where Humpty-Dumpty says to Alice: *"When I use a word, it means just what I choose it to mean, neither more nor less"*.

[7] From Oxford's Dictionary *"(of a creative work) referring to itself or to the conventions of its genre; self-referential"*.

2.3 APPROACHES

Several approaches and techniques exist to deal with systems with high-variability needs. The following sections provides an overview of those most commonly found.

2.3.1 Software Product Lines

A software product line (SPL) is a set of software systems which share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [CN01]. Software product line development, encompasses software engineering methods, tools and techniques for supporting such approach. A characteristic that distinguishes SPL from previous efforts is predictive versus opportunistic software reuse. Rather than put general software components into a library in the hope that opportunities for reuse will arise, software product lines only call for software artifacts to be created when reuse is predicted in one or more products in a well defined product line.

2.3.2 Naked Objects

Naked Objects takes the automatic generation of graphical user interfaces for domain models to the extreme. This software architectural pattern, first described in Richard Pawson's Ph.D. thesis [Paw04] which work[8] includes a thorough investigation on prior known uses and variants of this pattern, is defined by three principles:

1. All business logic should be encapsulated onto the domain objects, which directly reflect the principle of *encapsulation* common to object-oriented design.

2. The user interface should be a direct representation of the domain objects, where all user actions are essentially creation, retrieval and message send (or method invocation) of domain objects. It has been argued that this principle is a particular interpretation of an object-oriented user-interface (OOUI).

3. The user interface should be completely generated solely from the definition of the domain objects, by using several different technologies (e.g., source code generation or reflection).

2.3.3 Domain-Driven Design

Domain-driven design (DDD) was coined by Eric Evans in his books of the same title [Eva03]. It is an approach[9] to developing software for complex needs by deeply connecting the implementation

[8] Pawson's thesis also contains a controversial foreword by Trygve Reenskaug, who first formulated the model-view-controller (MVC) pattern, suggesting that Naked Objects is closer to the original MVC intent than many subsequent interpretations and implementations.

[9] In the sense that it is neither a technology nor a methodology.

to an evolving model of the core business concepts. DDD encompasses a set of practices and terminology for making design decisions that focus and accelerate software projects dealing with complicated domains, based on the following premises:

1. Placing the project's primary focus on the core domain and domain logic;
2. Basing complex designs on a model;
3. Initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem.

2.3.4 Model-Driven Engineering

The term Model-Driven Engineering (MDE) has roots that go back as 2001 [MM03] as an answer to the growing complexity of system architectures. This growing complexity and the lack of an integrated view "forced many developers to implement suboptimal solutions that unnecessarily duplicate code, violate key architectural principles, and complicate system evolution and quality assurance" [Sch06].

To address these issues, Model-Driven engineering combines domain-specific modeling languages (DSMLs) with transformation engines and generators in order to generate various types of artifacts, such as source code or alternative model definitions. The usage of DSMLs ensures that the domain model is perfectly captured in terms of syntax and semantics. This guarantees a flatter learning curve as the concepts present in the language are already known by the domain experts. This also helps a broader range of experts, such as system engineers and experienced software architects, ensure that software systems meet user needs.

Object-oriented based MDE are typically metamodeling techniques that strives to close the gap between specification artifacts which are based upon high-level models, and concrete implementations. A conservative statement claims that MDE tries to reduce the effort of shortening (not completely closing) that gap by generating code, producing artifacts or otherwise interpreting models such that they become executable [FR07]. Proponents of MDE claim several advantages over traditional approaches [RFBLO01]:

1. **Shorter time-to-market.** Since users model the domain instead of implementing it, focusing on analysis instead of implementation details;
2. **Increased reuse.** Because the inner workings are hidden from the user, avoiding to deal with the intricate details of frameworks or system components;
3. **Fewer bugs.** Because once one is detected and corrected, it immediately affects all the system leading to increased coherence;

4. **Easier-to-understand systems and up-to-date documentation.** Since the design is the implementation, not only they never fall out of sync, but they are both the medium and outcome of design.

One can argue if these advantages are exclusive of MDE or just a consequence of "raising the level of abstraction" § 2.2.5 (p. 24). Downsides in typical generative MDE approaches include the delay between model change and model instance execution due to code generation, debugging difficulties, compilation, system restart, installation and configuration of the new system, which can take a substantial time and must take place within the development environment [RFBLO01]. Once again, it doesn't seem a particular downside of MDE, but a general property of normal deployment and evolution of typical systems.

Further issues related to current MDE adoption will be addressed in Chapter 4 (p. 47). It seems worthwhile to note that the prefix *Model-driven* seems to be currently serving as a kind of umbrella definition for several techniques.

2.3.5 Model-Driven Architecture

Model-driven architecture (MDA) in an approach to MDE proposed by the OMG, for the development of software systems [MM03, OMG10b]. It provides a set of guidelines to the conception and use of model-based specifications and may be regarded as a type of domain engineering. It bases itself on the Meta Object Facility (MOF) [OMG10a], which main purpose is to define a strict, closed metamodeling architecture for MOF-based systems and UML itself. It provides four modeling levels (M_3, M_2, M_1 and M_0), each conforming to the upper one (M_3 is in conformance to itself).

Like most of the MDE practices, MDA thrives within a complex ecosystem with specialized tools for performing specific actions. Moreover, MDA is typically oriented for generative approaches, using systematic offline transformation of high-level models into executable artifacts. For example, trying to answer MDA's goal of covering the entire gap between specification and implementation, XUML was developed [MB02]. It is a UML profile which allows the graphical representation of a system, and takes an approach in which platform specific implementations are created automatically from platform-independent models, with the use of model compilers.

While some complex parts of MDA allow runtime adaptivity, developers seldom acquire enough knowledge and proficiency of the overall technologies to make them cost (and time) effective in medium-sized applications. Runtime adaptivity may be approached in different ways, including the use of reflection, and the interpretation of models at runtime [RFBLO01], covering the concept of Adaptive Object-Model — see Chapter 3 (p. 35).

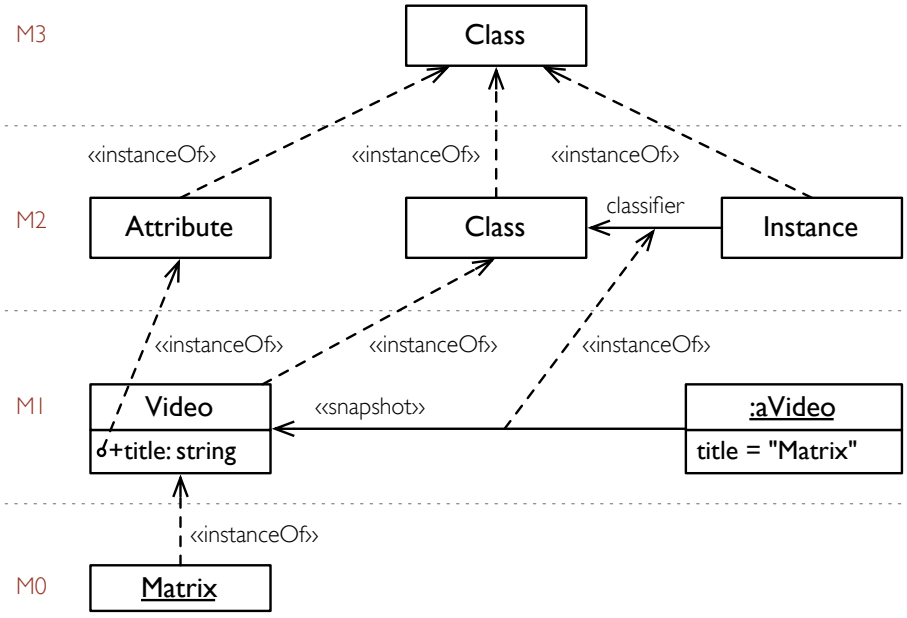


Figure 2.2: The four layers of abstraction in UML/MOF.

2.4 APPLICATION FRAMEWORKS

Object-oriented frameworks are both reusable designs and implementations, that orchestrate the collaboration between core entities of a system [RJ96]. While they establish part of the system's behavior, they are deliberately open to specialization by providing hooks and specialization points. A framework usually makes use of the *Hollywood Principle*[10], which is known to promote *high cohesion* and *low coupling* in object-oriented designs. The framework dictates the architecture of the underlying system, defining its overall structure, key responsibilities of each component (which together form a COMPONENT LIBRARY), and the main THREAD OF CONTROL. It captures design decisions common to its application domain, thus emphasizing design reuse over code reuse. Moreover, it is well known that mature frameworks can exhibit a reduction of up to 90% of source code, when compared to software written with the support of a conventional function library [WGM89, FSJ99a, FJ00, FSJ99b].

Some authors though argue that the development effort required to produce a framework (as thus its cost) is extraordinarily higher than the costs of developing a specific application [FPR00]. And when similar applications already exist, they have to be studied carefully to come up with an appropriate framework for that particular domain. Adaptations of the resulting framework lead to an iterative redesign. So frameworks represent a long-term investment that pays off only if similar applications are developed again and again in a given domain.

[10] A cliché response given to amateurs auditioning in Hollywood: *"Don't call us, we'll call you"*.

2.4.1 Building Frameworks

Despite the considerable number of successful frameworks developed during the last decades, designing a high-quality framework is a difficult task. One of the most common observations about successful framework development is that it requires iteration: "Good frameworks are usually the result of many design iterations and a lot of hard work" [WBJ90].

The precise need for iteration in framework development (though a common practice nowadays for the development of most software artifacts) is mainly due to three reasons. The first one is that finding the correct abstractions in immature domains is very hard. This difficulty often results in mistakes being discovered only when the framework is used, which then leads to refinements and iteration. Another reason for iterating is that the flexibility provided by a framework is difficult to fine-tune a priori, and its evaluation usually requires to experiment it in concrete situations. A third reason is that frameworks are high-level abstractions, and thus their generality and reusability is highly dependent on the original examples used during the abstraction process.

Despite the importance of iteration in framework refinement, it would be beneficial to have processes able to reduce the number of design iterations and the effort spent on framework refinement. The lack of mature processes for developing frameworks is an issue. While framework development processes get more and more mature, framework development is considered (by some) as an art, requiring both experience and experimentation [JF88].

Several methods have been proposed in the literature to support framework development in order to reduce the refinement effort and make it more predictable. Some examples of methods include (i) a pattern language for evolving frameworks [RJ96], (ii) systematic framework design by generalization [Sch97], (iii) hot-spot-driven development [Pre99], (iv) the catalysis approach [DW99], (v) role-modeling approach for framework design [RG98], and (vi) the approach for deriving frameworks from domain knowledge [ATMB00].

All these approaches define similar activities for developing frameworks but differ on the emphasis put on some activities and artifacts used as starting points. For example, [RJ96] suggests starting from concrete applications and abstract the framework using iterative refinements, thus emphasizing a bottom-up approach; [ATMB00], on the other hand, suggests starting not from concrete applications, but from domain knowledge, thus emphasizing a top-down approach.

Roberts and Johnson proposes a pattern language to evolve a framework [RJ96]. The method assumes that primary abstractions are very hard to find, and therefore considers that several development efforts are required to achieve a successful framework. The method suggests to start from concrete applications and iteratively abstract and refine the framework from those applications. The patterns can be found in Figure 2.3 (p. 31).

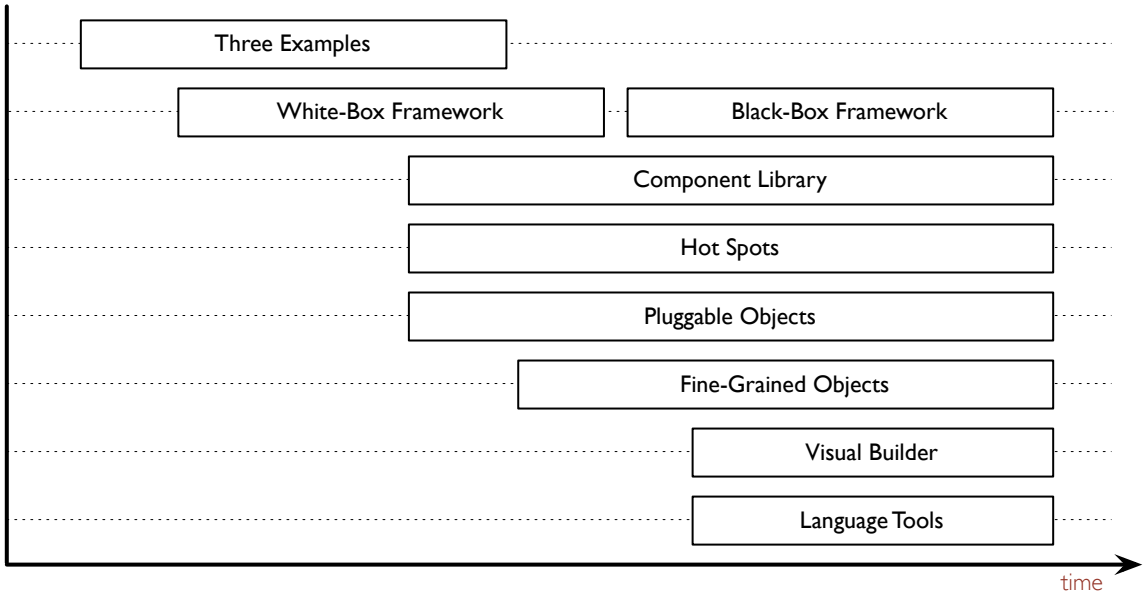


Figure 2.3: Patterns for Evolving Frameworks [RJ96].

2.4.2 Documenting Frameworks

Aguiar proposed a minimalist approach to framework documentation, which covered the overall documentation process, from the creation and integration of contents till the publishing and presentation, by focusing on minimizing the obtrusiveness to the learner of training material, offering the minimal amount of information that the learner needs to get the task done. It encompasses (i) a documentation model to organize the contents required to produce a minimalist framework manual along a virtual layered documentation space, (ii) a process which defines roles, techniques and activities involved in the production of minimalist framework manuals, and (iii) a specific set of tools, convenient to be adopted in a wide range of software development environments [Agu03].

In terms of notation, the UML profile for framework architectures provides an enriched UML subset, with some UML-compliant extensions, to allow the annotation of such artifacts. This profile is called UML-F and is targeted to framework technology [FPR00].

2.5 RELEVANT TOOLS

From full programming languages, to approaches and frameworks, several tools to support them already exist, which are somewhat capable to address *incomplete by design* systems. The list that follows is not meant to be *exhaustive*, but rather *representative* of existing technology, either due to relevant particularities or just by sheer popularity in this context.

2.5.1 Wikis

Earlier in 1995, Cunningham wrote a set of scripts that allowed collaborative editing of webpages inside the very same browser used to view them [Cun95]. He named this system *WikiWikiWeb*, due to the analogy between the meaning of the word *wiki* — quick — and the underlying philosophy of its creation: a *quick-web*. Since then, *wikis*[11] have gradually become a popular tool on several domains, including that of software development, e.g., to assist the creation of lightweight software documentation [Agu03]. They ease collaboration, provide generalized availability of information, and allow the combination of different types of content, e.g., text, images, models, and code, in a common substrate [AD05].

One may similarly regard modern development and usage of software systems as increasingly *collaborative*. The underlying design and the knowledge that lead to it (e.g., requirements, use-cases, models) is mostly devised and shared between both developers and stakeholders. Data is collaboratively viewed and edited among users. But usually these circles — developers and users — seem to be regarded as disjoint, where the emergent knowledge from this collaboration *begin* and *end* towards the synthesis of the artifacts themselves, despite the fact they have — and are — built incrementally.

Reflecting on the overall success of wikis, it seems to be related to a set of design principles drafted by Cunningham [Cun03]. Specifically, wikis share a common set of properties, viz.: (i) open, (ii) incremental, (iii) organic, (iv) universal, (v) overt, (vi) tolerant, (vii) observable, and (viii) convergent. The underlying rational behind these principles exhibit a constructionist attitude towards *change* and *incompleteness* — instead of regarding the incompleteness of information as a *defect*, they embrace it as the *means* through which the system evolves, in a *continuous* fulfillment of its function. A transposition of this central notion to *collaborative epistemology* can be found in Surowiecki's "The Wisdom of Crowds" [Sur05], and once the fundamental rational is set, the same underlying principles can be applied to other computational systems as well. This attitude towards incompleteness will be further explored in § 6.1.1 (p. 94).

2.5.2 Smalltalk

Smalltalk is a pure[12] object-oriented, dynamically typed, reflective programming language, designed and created for "constructionist learning", led by Alan Kay and Dan Ingalls at the Xerox Palo Alto Research Center (PARC) [Kay93].

[11] After the creation of the *WikiWikiWeb*, several new sites and systems — or engines — emerged based on the same underlying principles, and are generally called *wikis*. Among them is the well-known *Wikipedia* [Wik10d], based on the *MediaWiki* [Med07] engine.

[12] Pure object-oriented languages differ from *"hybrids"*, like Java and C++, in the sense there is no fundamental difference between objects and primitive types. Unlike Smalltalk, primitive types in Java were either stored directly in fields (for objects) or on the stack (for methods) rather than on the heap, prior to v.5.0. More recently, primitive types are automatically *boxed* — a technique that allows primitives to be treated as instances of their wrapper class.

One key property to Smalltalk is being both structurally and computationally reflective, up to the point that both the compiler and virtual machine (VM) are accessible to the application [GR83]. For example, whenever the user enter textual source code, this is typically compiled into method objects that are instances of CompiledMethod, and subsequently added to the target class' method dictionary. New class hierarchies are created by sending a message to the parent class. Even the Integrated Development Environment (IDE) is running from within the Smalltalk VM, making extensive usage of *introspection* to find implemented methods, classes, fields, etc. and assist the user with *code-completion* and *unit-testing*, among others. Unlike Java or C#, programmers usually don't organize their code in textual files and proceed with a *full* compilation. Instead, they incrementally modify and extend the system during run-time, storing the entire VM state in an image file [NDP09].

```
1 doesNotUnderstand: aMessage
2     ^target
3         perform: aMessage selector
4         withArguments: aMessage arguments
```

Source 2.5: Example of an implementation of the Proxy pattern in Smalltalk.

Another relevant property of Smalltalk w.r.t. *incompleteness* is the way methods are (not) invoked — rather a message is *sent* to the target object. If the target object doesn't implement the expected method, the full message is reified as an argument, and sent as a doesNotUnderstand: message. In a Smalltalk IDE, the default implementation is to open an error window, where the user could *inspect* and *redefine* the offending code, and simply continue from that point onward [NDP09]. Alternatively, one can create an object redefining the doesNotUnderstand: default behavior, effectively implementing the PROXY pattern [GHJV94], as seen in Source 2.5.

2.5.3 Ruby on Rails

Ruby on Rails (ROR) is a full-stack web framework, initially developed by David Hansson in 2003, based on the MVC pattern. According to Hansson, ROR's goal is *"for programmers happiness and sustainable productivity [letting] you write beautiful code by favoring convention over configuration"* [Han]. ROR takes a lot of its properties from Ruby's metaprogramming capabilities, exemplified in Source 2.6 (p. 34), heavily inspired in LISP and Smalltalk [THB+06]. Despite that, ROR's current version takes an approach much closer to generative programming than intercession — see § 2.1.3 (p. 18) and § 2.2.2 (p. 21).

Regarding code generation, the ROR framework includes a series of mechanisms for system artifacts generation, viz. models, controllers and views. It does so by analyzing the underlying relational database model and deriving the object-oriented specifications from available meta-information (e.g., column's type, name, foreign keys, etc.). Instead of generating a static model

```ruby
class Greeting
  def initialize(text)
    @text = text
  end
  def welcome
    @text
  end
end

my_object = Greeting.new("Hello")
my_object.class                          # ≡ Greeting
my_object.class.instance_methods(false) # ≡ [:welcome]
my_object.instance_variables             # ≡ [:@text]
```

Source 2.6: Example of an implementation of the Proxy pattern in Ruby.

definition, ROR deduces the necessary information whenever the system is loaded, and generates an adequate code skeleton for basic CRUD operations in views[13], greatly accelerating the development process by providing the developers with a basic blueprint of a fully functional system that can be subsequently refined and tailored for specific needs [THB+06].

2.6 CONCLUSION

This chapter has mainly focused on related work to this dissertation, summarizing the state-of-the-art in the background themes. One way to design software capable to cope with incompleteness and change is to encode the system's concepts into higher-level representations, which could then be systematically synthesized into executable artifacts. To correctly capture and express these high-level representations, one must correctly identify the commonalities and variability of the system, and support the correct degree of adaptivity and/or adaptability. Techniques such as metaprogramming, metamodelling, generative programming, and domain specific languages, among others, were described and their potential interest in this dissertation analyzed. Existing tools and infrastructures, such as wikis and dynamic languages, also served as examples for the study of the architecture and design of these kind of systems. The next chapter will now focus specifically on the state-of-the-art of Adaptive Object-Models.

[13] A process known as *scaffolding*.

Chapter 3

State of the Art in Adaptive Object-Models

3.1 The AOM Architectural Pattern 35
3.2 Formalized Patterns for AOM 39
3.3 Differences from other Approaches 45
3.4 Conclusion 46

In search for flexibility and run-time adaptability, many developers had systematically applied code and design reuse of particular domains, effectively constructing higher-level representations (or abstractions). For example, some implementations have their data structure and domain rules extracted from the code and stored externally as modifiable parameters of well-known structures, or statements of a DSL. This strategy gradually transforms some components of the underlying system into an interpreter or virtual machine whose run-time behavior is defined by the particular instantiation of the defined model. Changing this "model description" results on the system following a different business domain model. Whenever we apply these techniques to object-oriented design and principles, we usually converge to an architectural style known as the Adaptive Object-Model (AOM) [YBJ01b].

3.1 THE AOM ARCHITECTURAL PATTERN

An Adaptive Object-Model may be defined as (i) a class of systems' architectures, i.e., an architectural style, (ii) based on metamodeling and object-oriented design, (iii) often relying in Domain Specific Languages (DSL), (iv) typically having the property of being reflective, and (v) specially tailored to expose its domain model to the end-user. Because it is an abstraction over a set of systems and techniques, resulting into a common abstract architecture, it is called an "architectural pattern".

Some literature also classifies the AOM as a meta-architecture, due to its core usage of meta-programming and metamodeling [YBJ01b]. In fact, the evolution of the core aspects of a AOM (and its associated vocabulary) can be observed by the broad nomenclature used in the literature in the past, e.g., *Type Instance* [GHV95], *User Defined Product Architecture* [JO98], *Active Object-Models* [YFRT98] and *Dynamic Object Models* [RD98]. As theories and technology crystallize, the nomenclature begins to solidify, but it is an empirical observation of the current literature that that phase has not yet been achieved.

3.1.1 Why is it a Pattern?

When building systems, there are recurrent problems that have proven, recurrent solutions, and as such are known as *patterns* [AIS77]. These patterns are presented as a three-part rule, that expresses a relation between a certain context, a problem, and a solution. A software design pattern addresses specific design problems specified in terms of interactions between elements of object-oriented design, such as classes, relations and objects [GHJV94]. They aren't meant to be applied *as-is*; rather, they provide a generic template to be instantiated in a concrete situation, as discussed in § 1.7 (p. 9).

The concept of Adaptive Object-Model is inherently coupled with that of an architectural pattern [BMR+96], as it is an effective, documented, and prescriptive solution to the following recurrent problem: *how to allow the end-user to modify the domain model of the system while it is running?* It should therefore be noted that most AOM emerge from a bottom-up process [YBJ01b], resulting in systems that will likely use a subset of its concepts and properties, only when and where they are needed. This is in absolute contrast with top-down efforts of specific meta-modeling techniques (e.g., Model-Driven Architecture) where the whole infrastructure is specifically designed to be as generic as possible (if possible, to completely abstract the underlying level).

3.1.2 Towards a Pattern Language

The growing collection of AOM-related patterns [WYWB09, FCW08, WYWB07] which is forming a domain pattern language [WYWBJ07], is currently divided into six categories, viz. (a) Core, which defines the basis of a system, (b) Creational, used for creating runtime instances, (c) Behavioral, (d) GUI, for providing human-machine interaction, (e) Process, to assist the creation of a AOM, and (f) Instrumental, which helps the instrumentation. The first draft of this pattern language was done by Welicki *et al.* [WYWBJ07], and is summarized in Figure 3.1 (p. 37) and Table 3.1 (p. 38).

Core AOM Patterns

The following patterns define the structural and behavioral basis of any AOM system:

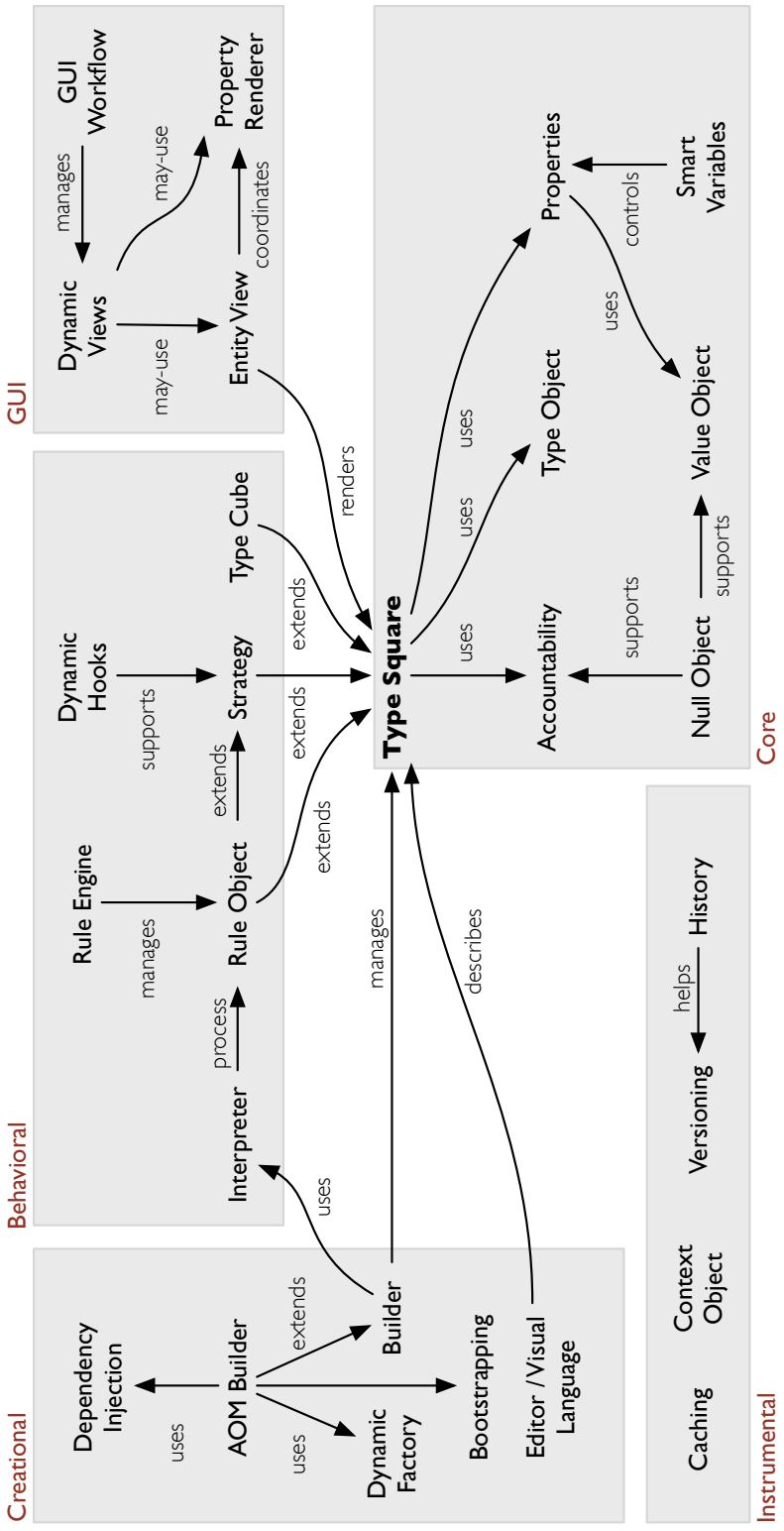


Figure 3.1: Pattern map of design patterns and concepts related to Adaptive Object-Models. Adapted from [WYWBJ07].

Category	Patterns
Core	Type Square, Type Object, Properties, Accountability, Value Object, Null Object, and Smart Variables
Creational	Builder, AOM Builder, Dynamic Factory, Bootstrapping, Dependency Injection, and Editor / Visual Language
Behavioral	Dynamic Hooks, Strategy, Rule Object, Rule Engine, Type Cube, and Interpreter
GUI	Property Renderer, Entity View, Dynamic View, and GUI Workflow
Process	Domain Specific Abstraction, Simple System, Three Examples, White Box Framework, Black Box Framework, Component Library, Hot Spots, Pluggable Objects, Fine-Grained Objects, Visual Builder, and Language Tools
Instrumental	Context Object, Versioning, History, and Caching

Table 3.1: A pattern catalog for Adaptive Object Models, adapted from [WYWBJ07].

- **Type Object.** As described by Johnson *et al.* [JW97] and Yoder *et al.* [YBJ01b], a TypeObject decouples instances from their classes so that those classes can be implemented as instances of a class. TypeObject allows new "classes" to be created dynamically at runtime, lets a system provide its own type-checking rules, and can lead to simpler, smaller systems.

- **Property.** The Property pattern gives a different solution to class attributes. Instead of being directly created as several class variables, attributes are kept in a collection, and stored as a single class variable. This makes it possible for different instances, of the same class, to have different attributes [Fow96].

- **Type Square.** The combined application of the TypeObject and Property patterns result in the TypeSquare pattern [Fow96]. Its name comes from the resulting layout when represented in class diagram, with the classes Entity, EntityType, Attribute and AttributeType.

- **Accountability.** Is used to represent different relations between parties using an AccountabilityType to distinguish between different kinds of relation [Fow96, Ars00].

- **Composite.** This pattern consists of a way of representing part-hole hierarchies with the Rule and CompositeRule classes [JW97].

- **Strategy.** Strategies are a way to encapsulate behavior, so that it is independent of the client that uses it. Rules are Strategies, as they define behavior that can be attached to a given EntityType [JW97].

- **Rule Object.** This pattern results from the application of the COMPOSITE and STRATEGY patterns, for the representation of business rules by combining simpler elementary constraints [WYWBJ07].

- **Interpreter.** An AOM consists of a runtime interpretation of a model. The INTERPRETER pattern is used to extract meaning from a previously defined user representation of the model [JW97].

- **Builder.** A model used to feed a AOM-based system is interpreted from its user representation and a runtime representation of it is created. The BUILDER pattern is used to separate a model's interpretation from its runtime representation construction [JW97].

Patterns of Graphical User Interface

The patterns in [WYWB07] focus specifically on User Interface (UI) generation issues when dealing with AOM. In traditional systems, data presented in user interfaces is usually obtained from business domain objects, which are thus mapped to UI elements in some way. In a AOM business objects exist under an additional level of indirection, which has to be considered. In fact, it can be taken into our advantage, as the existing meta-information, used to achieve adaptivity, can be used for the same purpose regarding user interfaces. User interfaces can this way be adaptive to the domain model in use.

- **Property Renderer.** Describes the handling of user-interface rendering for different types of properties.

- **Entity View.** Explains how to deal with the rendering of EntityTypes, and how PROPERTYRENDERERS can be coordinated for that purpose.

- **Dynamic View.** Approaches the rendering of a set of entities considering layout issues and the possibility of coordinating ENTITYVIEWS and PROPERTYRENDERERS in that regard.

3.2 FORMALIZED PATTERNS FOR AOM

The basic architecture of a AOM may be divided into three parts or levels that roughly correspond to the levels presented by MOF: M_0 is the operational level, where system data is stored, M_1 is the knowledge level where information that defines the system (i.e., the model) is stored, and M_2 is the design of our supporting infrastructure. M_0 and M_1 are variants of our system. M_2 is an invariant, i.e., the meta-model — should it need to change, we would have to transform M_0 and M_1 to be compliant with the new definition. See Figure 2.2 (p. 29) and Figure 3.2 (p. 40).

We may say that the key to good software design is two-fold: (a) recognize what changes, and (b) recognize what doesn't change; it is the search for *patterns* and *invariants*. The following

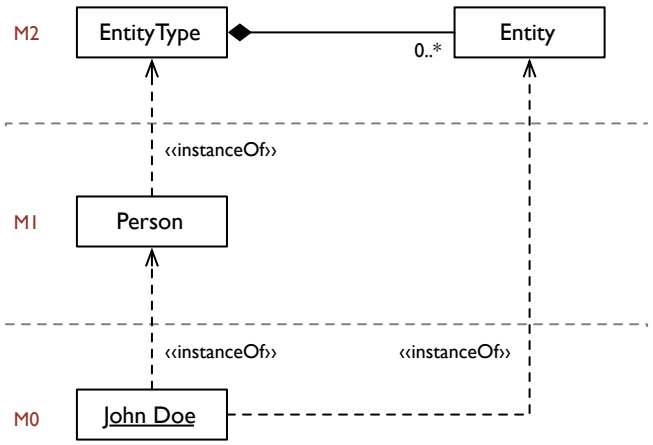


Figure 3.2: The meta layers of an AOM, when compared to MOF.

patterns try to anticipate what will be the target of continuous change, and encapsulate it accordingly. The first patterns, viz. TypeObject, Property, TypeSquare, and Accountability capture the structural core of an AOM.

Additional, some structural rules such as (i) legal subtypes, (ii) legal types of relationships between entities, (iii) cardinality of relationships and, and (iv) mandatory properties, can be captured through simple extensions of these patterns, as will be discussed in Chapter 5 (p. 59) and Chapter 6 (p. 93). More complex rules will require the usage of other patterns, viz. Composite, RuleObject, Strategy and Interpreter.

3.2.1 Type-Object Pattern

In the context of object-oriented programming and analysis, the type of every object is defined as a class, and its instance as an object which conforms to its class. A typical implementation of a software system would hardcode into the program structure, i.e., its source-code, every modeled entity, such as the concept of a *patient*. Whenever the system needs to be changed, e.g., to support a new entity, the source-code has to be modified.

However, if one anticipate this change, objects can be generalized and their variation described as parameters, just like one can make a function that sums any two given numbers regardless of their actual value. The TypeObject pattern, depicted in Figure 3.3 (p. 41), identifies that solution, which involves splitting the class in two: a type named Entity Type, and its instances, named Entity [JW97].

Using this pattern, patients now become instances of Entity Types — they become *meta-data*. The actual system *data*, such as the patient *John Doe*, is now represented as instances of Entity. Because both *data* and *meta-data* are defined and stored independently of the program structure, they are allowed to change during run-time.

There are some variations to this solution; for example, an optional relation between Entity-

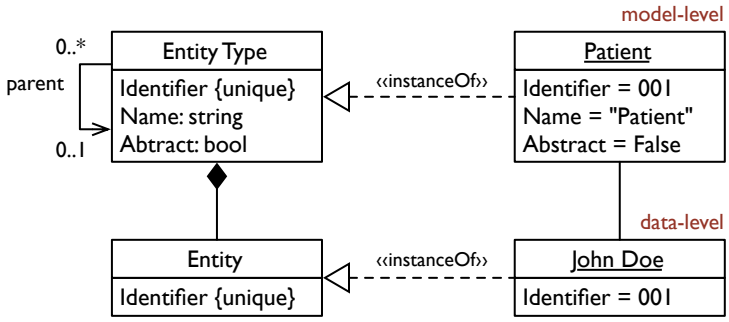


Figure 3.3: A class diagram of the TYPE-OBJECT pattern.

Types can support the notion of inheritance, which will incidentally be a solution to the problem of open-inheritance, as first discussed in Chapter 1 (p. 1). Provided sufficient mechanisms exists to allow the end-user customization of the model, new specializations can be added without modifying the source-code.

3.2.2 Property Pattern

We face a very similar problem to the TYPEOBJECT, when dealing with the attributes of an object, such as the name and age of a patient. Once again, these are usually classified as fields of a class, and its values stored in the object. The anticipation of change leads to the PROPERTY pattern; a similar bi-section between the definition of a property and its corresponding value is depicted in Figure 3.4.

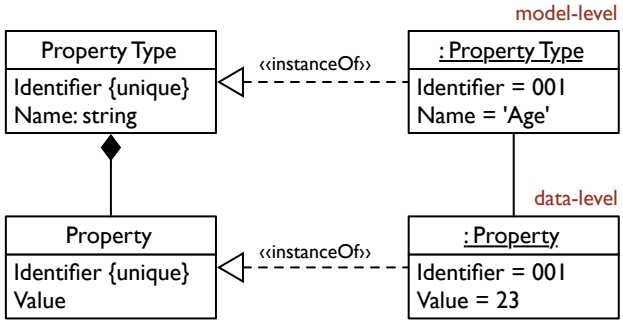


Figure 3.4: A class diagram of the PROPERTY pattern.

Using this pattern, the several attributes of the domain's entities become instances of Property Types, and their particular values instances of Property. Again, this technique solves the problem of adding (or removing) more information to existing entities beyond those originally designed, without touching the source-code.

3.2.3 Type-Square Pattern

The two previous patterns, TypeObject and Property, are usually used in conjunction, resulting in what is known as the Type-Square pattern — the very structural core of a AOM, and depicted in Figure 3.5. It is important to note that the diagram has instance elements representing both the systems' data and model, while the classes represent the static, abstract infrastructure.

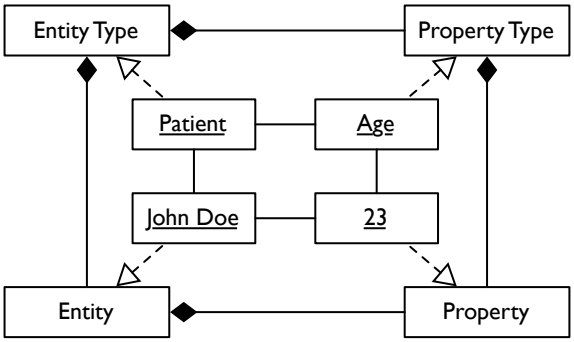


Figure 3.5: A class and objects diagram of the Type-Square pattern. The outer four elements are classes representing the system's infrastructure. The inner four elements are objects representing an example instantiation of a particular domain.

3.2.4 Accountability Pattern

Attributes are properties that refer to primitive data types like numbers or strings. Other possible type of properties are associations between two (or more) entities. For example, if Sue is the mother of John, then Sue e John are related through the parenthood relationship. There are several ways to represent a relationship using the Property pattern. One way is to inherit from property and make the separation from attributes and relations. Another way is to duplicate the Property pattern, using it once for attributes and once for associations. Finally, the distinction can be made by inspection of the value type: primitive data types will represent attributes, and model types would represent relations [Yodo2].

The most common pattern in the context of AOM is the duplication of the Property pattern, by modifying it to become the Accountability pattern [Fow96, Arsoo]. This pattern is usually implemented as two classes, the AccountabilityType, which represents the type of the relation (e.g. "parenthood") and is part of the EntityType, and the Accountability, which holds the values pointing to the Entities that participate in the relation. Figure Figure 3.6 (p. 43) depicts the application of this pattern to AOM, although there are slight variations, including the Party pattern [Fow97].

3.2.5 Composite Pattern

Consider the following rule: "the value of a specific property (i) of an object must be greater than 0 and less than 10 ($0 < i < 10$)". We may express this rule as a conjunction of two simpler

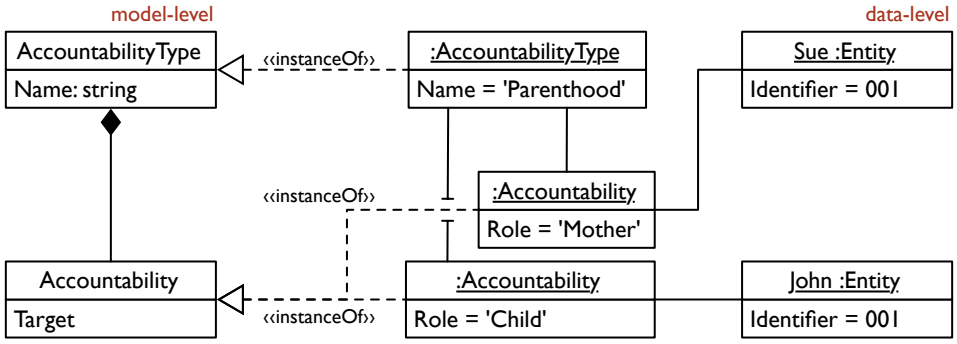


Figure 3.6: A class and objects diagram of the ACCOUNTABILITY pattern, forming the structure of the relationship between two entities.

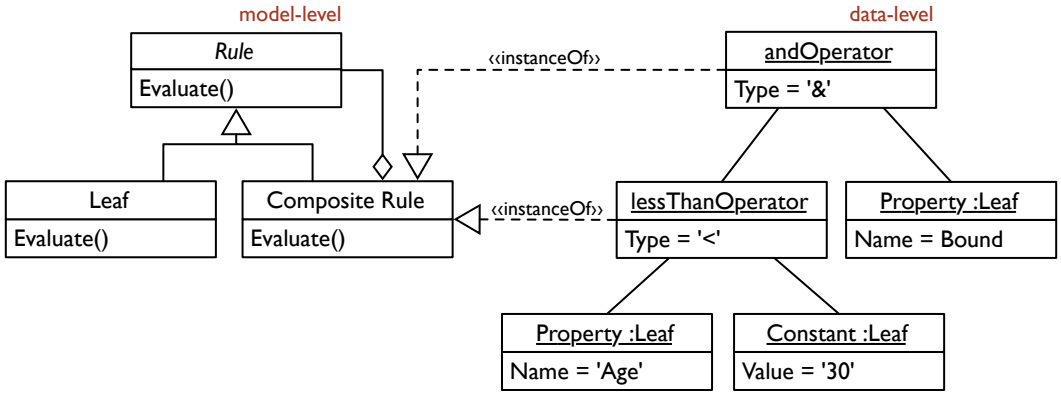


Figure 3.7: A class and objects diagram of the COMPOSITE pattern, forming the structure of the part-whole hierarchy of a AST for the expression $\mathrm{Age} < 30 \wedge \mathrm{Bound}$.

rules, namely that $(0 < \mathrm{i} \wedge \mathrm{i} < 10)$. Usually, this type of expressions are parsed into an Abstract Syntax Tree (AST), with operators as nodes, and effective values as leafs. Although each node may be recursively evaluated, the precise semantics given to this evaluation belongs to the node itself. From the client point-of-view (i.e., the user of the expression), mostly it wants to treat individual objects and compositions of objects uniformly. Composite can thus be used to this propose, as seen in Figure 3.7, explicitly ignoring the differences between compositions of objects and individual objects [GHJV94]. This pattern is used to support the INTERPRETER pattern as seen in § 3.2.7 (p. 44).

3.2.6 Strategy and Rule Object Patterns

Consider the following rule: "the value of a specific property is mandatory". There are two common ways to support this rule in a AOM, namely (i) to add a boolean field to the PROPERTY pattern, specifying if that particular property is mandatory, and (ii) to define a family of algorithms and encapsulate each one as a rule (e.g., MandatoryProperty), and call them whenever an Entity is validated. Because rules are encapsulated into objects, it makes them interchangeable. The latter

form the basis of the STRATEGY [GHJV94] a RULE OBJECT [Ars00] when applying to AOM, and an example can be seen in Figure 3.8.

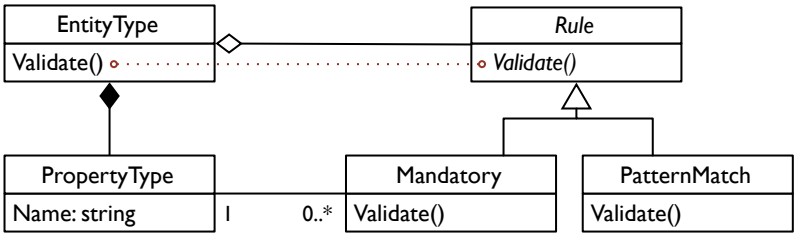


Figure 3.8: A class diagram of the STRATEGY and RULE OBJECT patterns, forming the structure of the family of algorithms for enforcing rules. The dashed line represents an invocation call.

3.2.7 Interpreter Pattern

As seen in § 3.2.5 (p. 42), the rule "the value of a specific property of an object (i) must be greater than 0 and less than 10 ($0 < i < 10$)", may be specified using an expression defined as a simple language, which will then use (or be transformed into) RULE OBJECTS as described in § 3.2.6 (p. 43). For that, a representation of the grammar's language along with the interpreter that uses the representation to interpret sentences in that same language, can be structured as a COMPOSITE of objects, resulting in the INTERPRETER pattern [GHJV94]. See both Figure 3.7 (p. 43) and Figure 3.9

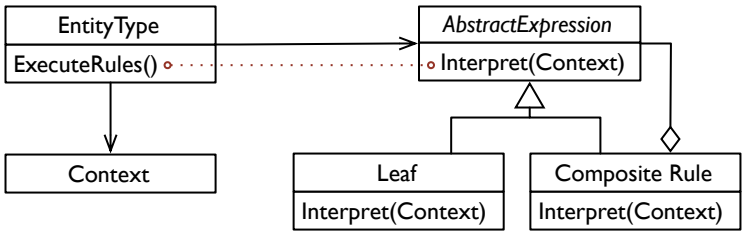


Figure 3.9: A class diagram of the INTERPRETER pattern, forming the structure of an expression language.

3.2.8 Composing the Patterns

The usual architecture of an Adaptive Object-Model is usually the product of applying one or more of the previously mentioned patterns in conjunction with other object-oriented design patterns, such as the FACTORY METHOD and BUILDER [GHJV94, Yod02]. One such typical architecture can be seen in Figure 3.10 (p. 45).

Nonetheless, mixing up patterns do not provide any concrete implementation of an AOM, and neither it points to what a framework for AOM would look like (in terms of its form). Yoder *et al.* pointed to the fact that *"every Adaptive Object-Model is a framework of a sort but there is*

currently no generic framework for building them" [Yod02]. In Chapter 6 (p. 93), the architecture and design of such framework will be proposed, and an implementation will be detailed.

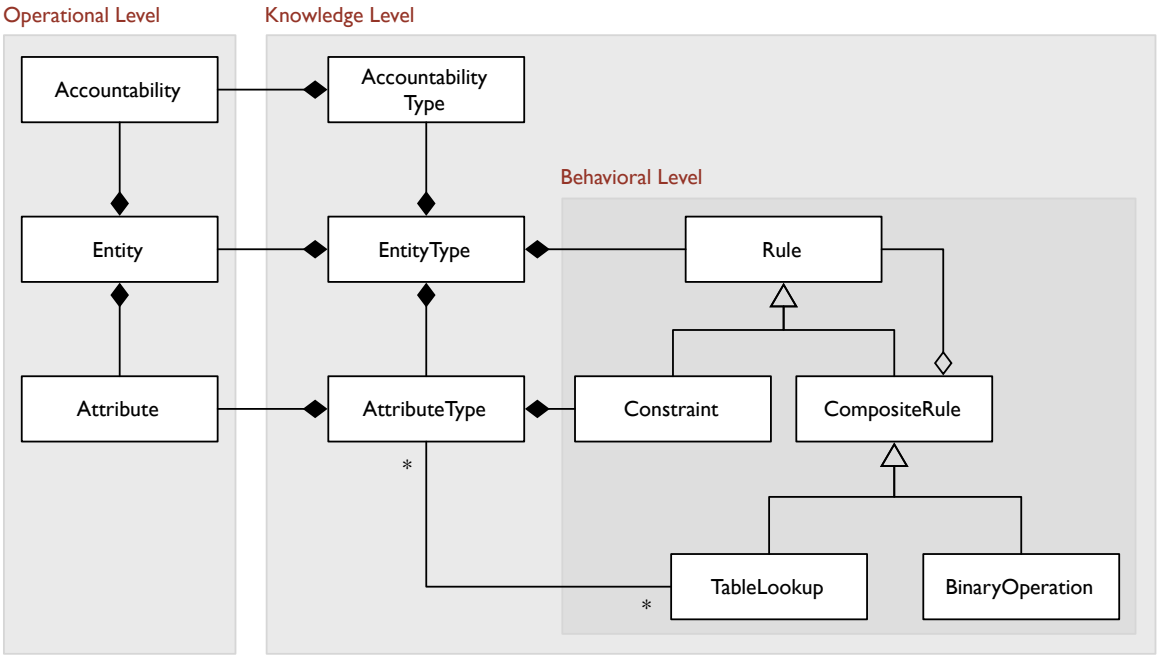


Figure 3.10: AOM core architecture and design, adapted from [YBJ01b].

3.3 DIFFERENCES FROM OTHER APPROACHES

The concepts of *end-user programming* and *confined variability* — the capability of allowing the system's users to introduce changes and thus control either part of, or the entire, system's behavior — are significant consequences of the AOM architecture which are not easily reconcilable with other techniques such as *Generative Programming*.

In previous sections, both *Generative Programming* § 2.2.2 (p. 21) and *Metamodeling* § 2.2.3 (p. 22) techniques were mentioned and briefly described as ways to address systems with high-variability needs. In this context, we may now begin to answer in what ways they differ from Adaptive Object-Models.

Perhaps the most important difference is when — i.e., at what time — they parse or interpret the system definition. Both Generative Programming and Metamodeling typically rely on code generated at compile time. On the other hand, the AOM interpret the underlying system definition during runtime. This difference allows a AOM to adapt to new, or changed, user requirements without having to turn down the system because of recompilation [YBJ01a]. This difference may also distinguish the AOM architectural style from typical CASE tools. It also leads to what is here defined as "confined variability", or *the ability of allowing the end-user, typically referred as change agent, to introduce controlled changes in specific points of the application*. Sys-

tems based on the AOM architecture elegantly support this type of requirements since the ability to interpret changes is built into the system.

Another, though definitely more open to argumentation, is that the design of Adaptive Object-Models is done from bottom-up [YBJ01b]. This bottom-up approach carries the promise of being able to incrementally evolve a standard rigid design towards a more flexible one a step-by-step fashion. As a result, the resulting system tends to only use such concepts and techniques where (and when) it is most needed, thus slowly transforming rigid code into adaptable components. When compared to the top-down approach of other metamodeling techniques where the whole system is thought to be as generic as possible, this property can reduce the overall complexity needed to achieve adaptability, and therefore increase its adoption in typical development cycles.

3.4 CONCLUSION

The Adaptive Object-Model meta-architectural style can be defined as a class of system's architectures based on metamodeling, meta-programming and object-oriented design that, relying on several techniques, have the property of being reflective and specially tailored to allow the end-user to make changes to the domain model. The theories and techniques behind the AOM style have been captured as patterns, and recently a pattern language started to be drafted. In that proposal, almost thirty patterns were identified and divided into six categories. So far, less than a dozen of them were properly formalized in the context of AOM.

In this chapter we have reviewed the current state-of-the-art regarding adaptive object-models, and some of the patterns that have already been formalized. The typical architecture of a AOM is the result of applying several of these patterns in conjunction with other object-oriented design principles. Yet, at least two issues remain open, viz. (i) most of the patterns regarding the AOM remain to be formalized, and perhaps not all of them were identified, (ii) there is currently no generic framework for building AOM-based systems and (iii) no rigorous empirical study presenting evidence of any specific benefits of AOM exists in the literature. These issues will be addressed in Chapter 5 (p. 59) and Chapter 6 (p. 93).

Chapter 4

Research Problem

4.1 Epistemological Stance 48
4.2 Fundamental Challenges 48
4.3 Thesis Statement 50
4.4 Specific Research Topics 53
4.5 Validation Methodology 56
4.6 Conclusion 56

The Adaptive Object-Model and its ecosystem is composed of architectural and design patterns that provide domain adaptability to object-oriented based systems. As patterns, they have been recurrently observed and systematically documented [Yodo2]. A pattern language has been proposed, identifying almost thirty patterns, divided into six different categories [WYWBJo7]. However, less than a dozen have yet been formalized, and there is strong evidence that not all of them have been identified. Likewise, no generic framework[1] intended for the construction of AOM-based systems have been found in the literature so far [Yodo2]. Further, no rigorous empirical study supporting claims about any kind of benefits or liabilities of AOM exists in the literature, despite the fact that it is a pattern. In this chapter, we will raise several research questions about the benefits of MDE in general, and AOM in particular, argue what kind of validation should be used to support common claims, point to what should be the baseline for pursuing empirical studies, and underline the need to design controlled experiments as repeatable packages for independent validation.

[1] This may sound as a contradiction, as frameworks are usually domain-specific. Therefore, a AOM-framework used to implement a complete AOM-oriented system would call for a top-down approach, which was pointed as a disadvantage in the previous chapter. We will see ahead that such consideration will shape the framework design.

4.1 EPISTEMOLOGICAL STANCE

In order to understand the way software engineers build and maintain complex and evolving software systems, researchers need to focus beyond the tools and methodologies; they need to delve into the social and their surrounding cognitive processes *vis-a-vis* individuals, teams, and organizations. In this sense, research in software engineering is regarded as inherently coupled with human activity, where the value of generated knowledge is directly linked to the methods by which it was obtained.

Because the application of reductionism to assess the practice of software engineering, particularly in field research, is very complex (if not unsuitable), the author claims that the presented research is to be aligned with a pragmatist view of truth, valuing acquired practical knowledge. In other words, the author choose to use whatever methods seemed more appropriate to prove — or at least improve our knowledge about — the questions here raised. Formally, a systematic scientific approach based on this epistemological stance requires the use of mixed methods, among which are (a) (Quasi-)Experiments, used to primarily assess exploratory or very confined questions, and are suitable for an academic environment and (b) industrial Case-Studies, as both a conduit to harvest practical requirements, as to provide a tight feedback and real-world application over the conducted investigation.

4.2 FUNDAMENTAL CHALLENGES

The fundamental research questions directly inherited from the current research trends and challenges in the area of Model-Driven Engineering (MDE), can be found in a recent research roadmap published by France *et al.* [FR07], which states the following three driving issues:

1. **What forms should runtime models take?** Specifically in the context of this dissertation, what form should *incomplete by design* software systems take? What type models do they require? What are the forces that shape those models?

2. **How can the fidelity of the models be maintained?** Particularly, how can the fidelity of models be maintained during *evolution*? What models are adequate to *observe* the system? What models are needed to *modify* the system?

3. **What role should the models play in validation of the runtime behavior?** What validation provides a domain model that is a first-class artifact of the software system? What if the outcome of design is the same model the system is running?

Another survey by Teppola *et al.* [TPT09] synthesizes several obstacles related to wide adoption of MDE:

1. **Understanding and managing the interrelations among artifacts.** Multiple artifacts such as models, code and documentation, as well as multiple types of the same artifact (e.g., class, activity, state diagrams) are often used to represent different views or different levels of abstraction. Subsets of these overlap in the sense that they represent the same concepts. Often because they are manually created and maintained without any kind of causal connection, evolution raises the problem of maintaining consistency.

2. **Evolving, comparing and merging different versions of models.** The tools we currently have to visualize differences among code artifacts are suitable because they essentially deal with text. Models do not necessary have a textual representation, and when they do, it may not be the most appropriate to understand its evolution and to make decisions, particularly if these are to be carried by the end-user.

3. **Model transformations and causal connections.** Models are often used to either (i) reflect a particular system, or (ii) dictate the system's behavior. The relationships between the system and its model, or between different models that represent different views of the same system, are called *causal connections*. Maintaining their consistency when artifacts evolve is a complex issue, often carried manually.

4. **Model-level debugging.** If the model is being used to dictate a system's behavior, enough causal connections must be kept in order to understand and debug a running application at the model-level.

5. **Combination of graphical and forms-based syntaxes with text views.** Developers and end-users have different preferences concerning textual syntaxes and graphical editors to view and edit models. To this extent, a complete correspondence between each strategy is currently not well supported.

6. **Moving complexity rather than reducing it.** Model-Driven Engineering is not a "silver-bullet" [Bro87] and as such its benefits must be carefully weighted in context to assess whether the approach will actually reduce complexity, or simply move it elsewhere in the development process.

7. **Level of expertise required.** It is not clear if the interrelationships among multiple artifacts (which may have different formalisms), combined with the necessary (multiple) levels of abstraction to express a system's behavior actually eases the task of any given stakeholder to understand the impact and carry out a particular change, and to which extent current training in computer science and software engineering courses is adequate.

4.2.1 Viewpoints

Systems based on Adaptive Object-Models, have two distinct viewpoints from where its benefits can be measured: (i) the developer viewpoint, who is actively trying to build a system for a specific domain, and (ii) the end-user, who, when provided, will be evolving the system once delivered. The existence of an end-user as a change-agent, although always cited as a benefit of the use of AOM, represents a new amount of additional concerns. Any technique that may be regarded as a good way to improve the application adaptability to a well-trained developer, may be revealed as an encumbrance to the end-user, and a particular nuisance to the user-interface design.

Therefore, there are some questions regarding end-user development should be either specifically researched in the area of AOM, or borrowed from other fields of research, namely:

1. **End-user perception of the model.** The way end-users see their systems is different from the abstraction the developer are used to. Understanding the differences between these two perspectives is essential to provide mechanisms in the user-interface that are suitable, and avoids an higher-level BIG BALL OF MUD [FY97].

2. **Visual metaphors.** We shouldn't expect the common end-user to actually type in a textual DSL to express some new rules they want to insert in the system. Other kinds of visual metaphors should be considered as a proxy for the underlying rule engine. A more detailed discussion can be found in [Nar93].

3. **Evolving the model.** A tentative, failed, evolution may be disastrous regarding the meaning of data. Mechanisms to recover from mistakes, though already useful to the developer, are paramount to the end-user.

4.3 THESIS STATEMENT

It is the author belief that most current problems developers face when developing software with high-variability and runtime adaptability needs, could be efficiently coped with the Adaptive Object-Model architectural style. Furthermore, the author believes that where several other model-driven approaches have failed to be generally accepted, perhaps mostly because of differences presented in § 3.3 (p. 45), the AOM, within its bottom-up approach, may reveal to have the properties needed to successfully give answer to a set of architectural problems.

Notwithstanding the issues presented in Chapter 1 (p. 1), the author does not consider part of this study to prove where – and why – these kind of systems emerge, nor the specific reasons that may lead an architect to recognize these kind of systems. Instead, the author assumes that: (i) the system we are working with exhibit an high degree of variability, (ii) its specifications have

shown a high degree of incompleteness, and (iii) it is desirable to reduce the effort[2] of coping with changes made to the domain model.

Should one agree with the aforementioned premises, the author's fundamental research question may be stated as:

> *What form should this type of systems take, and which kind of tools and infrastructures should be available to support the development of such software systems?*

Consequently, the main goal is to research this *form*, here to be understood as the *architecture* and *design* of such software systems, along with the specification and construction of the appropriate tools and infrastructure. More specifically, the author claims the following hypothesis:

> *When developers are provided with a reusable and extensible infrastructure to build systems based upon the Adaptive Object-Model as the main architectural style, either in the form of a pattern language, or through concrete software components, they will (i) significantly increase their efficiency to construct and evolve systems with high-variability needs when compared to traditional approaches, and (ii) will be able to empower end-users (as the domain experts) to conduct their own (confined) evolution during run-time. Such an infrastructure would maximize architectural and design reuse, and leverage both developers and end-users' capability to efficiently adapt to frequent domain changes.*

This statement uses terms whose meaning may not be consensual, and therefore lead to questions that deserve further discussion:

1. **What should be understood by a *pattern language*?**

   The concept of pattern language and its importance for the development of high-quality systems has already been discussed in Chapter 1 (p. 1) and Chapter 3 (p. 35). The pattern language here mentioned is based on the theories proposed by Christopher Alexander [Ale64], and later adopted to software by Gamma *et al.* [GHJV94] and Buschmann *et al.* [BMR+96].

2. **What should be understood by an *infrastructure*?**

   An infrastructure is here intended as a object-oriented *framework*, in the sense of both reusable designs and implementations that orchestrate the collaboration between core entities of a system [RJ96], built upon the *design and architectural patterns* presented in the pattern language for adaptive object-models.

3. **Who *benefits* from this infrastructure?**

[2] The intended meaning of *effort* should be loosely interpreted as monetary cost, available time and resources, required skills, resulting complexity, etc.

There are three resulting outcomes from this work, where the target audience sometimes overlap. Regarding the pattern language that will be presented in Chapter 5 (p. 59), the target audience are architects and framework developers building or trying to understand the inner workings of meta-architectures, among which may be (i) those whose interest is in the design of object-oriented programming or specification languages, and (ii) others that aim to improve their systems' adaptivity.

Concerning both the reference architecture and framework implementation proposed in Chapter 6 (p. 93), any team composed of framework developers, application developers, and framework selectors, who might be developing a product with high-adaptability needs, may directly benefit by either reusing or adapting the artifacts here presented.

4. **How is *reusability* and *extensibility* measured?**

   A framework's *reusability* and *extensibility* is measured by the given HOOKS and extension of the COMPONENT LIBRARY. However, because any measure is always relative to a given purpose[3], their existence will be primarily driven by the requirements established in the case studies described in Chapter 7 (p. 115), thus forming a feedback loop closely similar to that of Action Research [Lew46].

5. **How is *efficiency* to be measured?**

   Efficiency is to be measured by the quantity of *work* produced to achieve a desired effect. For measuring efficiency in large timespan, such as in a use-case analysis, it will be considered both the velocity (i.e., the amount of work per person per unit of time), and the amount of changes needed to achieve the necessary effect.

6. **Which are the *traditional approaches*?**

   When referring to traditional approaches, one is mentioning those that are capable of delivering adaptable systems, i.e., those where the end-user can introduce changes. This way, most Rapid Application Development (RAD) tools should not be considered, and as such it is to be chosen any particular setup suitable to provide such systems.

7. **Who are the *end-users*?**

   From the point of view of a framework, both *application developers* and *domain experts* are to be considered end-users. Therefore, end-users are defined as the group of persons who will ultimately operate the domain application built on top of this infrastructure.

8. **What is *confined evolution*?**

   Confined evolution refers to a situation where an adaptable system provide the means to restrict the possible changes conducted by the end-user in the underlying domain model,

[3] For example, does it make sense to ask how many HOOKS should a good framework provide?

despite the infrastructure supporting the creation of a new domain model from scratch. Therefore, the system is deliberately limited to a certain (confined) scope which will be open to end-user intervention.

9. **What is understood by frequent domain changes?**

   Frequent domain changes are to be viewed in perspective with the arguments presented in § 1.2 (p. 2) and § 1.3 (p. 4), where the application domain we are dealing with is characterized by continuous change.

4.4 SPECIFIC RESEARCH TOPICS

Despite the diversity of questions currently requiring research effort, this dissertation will focus more on those capable of supporting a MDE approach for the construction of AOM-based systems. In particular, the following concerns will be the main drive for the research:

1. **Patterns.** While several (approximately thirty) AOM-related patterns have been identified, most of them remain unstudied, undocumented and scattered, thus in need to be brought comprehensively into the literature [WYWBJ07]. Moreover, as the research deepens, more patterns and variations may be identified.

2. **Tools.** Concerning tools to support AOM based systems, several questions arise: (i) what kind of support these systems require from runtime infrastructures? (ii) Is it feasible to build a generic AOM runtime infrastructure? [Yod02] (iii) which tools are needed to enable application developers to build and maintain AOM-based applications? Frameworks, as reusable designs of software systems compromising a set of interrelated and extensible components, have been shown to reduce the cost of developing application by an order of magnitude [RJ96]. Hence, what *form* should a framework for developing AOM systems take?

3. **Applications.** The thesis statement assumes that systems which have highly-variable domains and runtime adaptability needs are good candidates for using the Adaptive Object-Model architectural style. This assertion will be addressed during the study of industrial use-cases of systems that have used the theoretical results and implementation artifacts of this dissertation.

4.4.1 Specific Challenges

Although the research in Adaptive Object-Models may be regarded as a subset of the research in MDE, we think the following questions, when carefully assessed, would pragmatically contribute to the current body of knowledge, particularly when choosing to use (or not) this pattern.

Though the belief that AOM are able to efficiently cope with several of the stated issues in software development, and this belief has been substantiated both by research on the wider area of MDE, as well as through the studies by the patterns community, carefully designed controlled experiments should provide statistical evidence that an AOM is not an anti-pattern (i.e., an obvious, apparently good solution, but with unforeseen and eventually disastrous consequences):

1. **Fitness for purpose.** When is an AOM adequate to use? When should the use of an AOM be considered *over-engineering*. What metrics should we base our judgment for applicability?

2. **Target audience.** What type of developers are best suited for AOM? Are current developers lacking in specific formation that hinders the usage and construction of AOM? What about end-users? Are there specific profiles that could point to a more suitable audience?

3. **Development speed indicator(s).** What is the impact on the usage of AOM during the several phases of the process? Do developers increase their ability to produce systems? How long is their *start-up* time?

4. **Extensibility indicator(s).** How easy is to extend a AOM-based system? Is it possible to HOOK a particular customization into the base architecture?

5. **Quality indicator(s).** What is the impact on software quality metrics when using AOM? How does it affect performance? How does it ensure correctness? Is consistency a major factor? What about the usability of automatically-generated interfaces? How can we improve them?

4.4.2 Thesis Decomposition

From the original thesis statement, it follows the following hypothesis that are believed to hold when comparing to traditional systems:

- **H1:** The framework provides a more suitable infrastructure for developing *incomplete by design* systems.

- **H2:** Developers focus more on domain objects than implementation artifacts.

- **H3:** It is easier to translate conceptual specifications into final artifacts.

- **H4:** The effort of adding or changing existing requirements is considerably lower than the alternatives.

- **H5:** Prototyped applications could be immediately (or with little changes) used in production-level environments.

- **H6:** This style of development is more in line with the agile principles.
- **H7:** This style of development could be used to develop face-to-face with the client.

4.4.3 Thesis Goals

The primary outcomes of this thesis encompasses the following aimed contributions to the body of knowledge in software engineering:

1. **The formalization of a pattern language for systems whose domain model is subject to continuous change during runtime.** When do this type of systems occur? What are the advantages? What are their underlying requirements? How do we cope with each of them? What are the benefits and liabilities of each specific solution? This contribution expands an unified conceptual pattern language, which allows architects and designers to recognize and adequately use some of the best practices in software engineering that cope with this type of systems. Further details are presented in Chapter 5 (p. 59).

2. **The specification of a reference architecture for adaptive object-model frameworks.** What kind of infrastructures are needed? What form should they take? What type of abstractions should be made and supported? What are the generic functionalities it should provide? What should be its default behavior? How can it be extended? This contribution addresses several issues concerning framework design for Adaptive Object-Models, and presents a solution through the composition of architectural and design elements. More details can be found in Chapter 6 (p. 93).

3. **A reference implementation of such framework.** From theory to practice, Chapter 6 (p. 93) also details a concrete implementation of a framework based on the proposed reference architecture, codenamed *Oghma*. The goal of attaining an industrial-level implementation of such framework, along with the research of specific design issues that arise only when pursuing such concrete implementations, allowed to further pursue the research using the chosen validation methodologies.

4. **Evidence of the framework benefits through industrial use-case applications.** A framework should emerge from reiterated design in real-world scenarios. As such, the implementation shown in Chapter 6 (p. 93) is mainly the result of an incremental engineered solution for specific industrial applications. This contribution presents software systems built on top of that framework, their context, their requirements, their particular problems, the way the framework was used to address them, the outcomes, and the lessons learned in Chapter 7 (p. 115).

5. **Evidence of the framework properties through controlled academic quasi-experiments.** Although the industrial usage of the framework provides pragmatic evidence of its benefits,

there are some threats that are inherent to that type of validation. These shortcomings are addressed in Chapter 8 (p. 129), by conducting a (quasi-)experiment within a controlled academic experimental environment, where the study of groups of undergraduate students interacting with the framework has shown the results to be consistent with those presented in Chapter 7 (p. 115).

4.5 VALIDATION METHODOLOGY

In order to pursue a scientific validation of the aforementioned thesis, it is necessary to adequately define the experimental protocols which assess these claims in a rigorous and sound way. This includes the precise definition of the processes followed in industrial case studies (*cf.* Chapter 7, p. 115), as well as in the quasi-experiments (*cf.* Chapter 8, p. 129) performed in academic contexts. The design of experimental protocols for the industrial case studies attempt to cover the whole experimental process, i.e, from the requirements definition for each experiment, planning, data collection and analysis, to the results packaging. Discussion on guidelines for performing and reporting empirical studies have been recently going by the works of Shull *et al.* [SSS07] and Kitchenham *et al.* [KAKB+08]. The typical tasks and deliverables of a common experimental software engineering process can be found in [GA07].

As such, in the author understanding, the best way of validating the thesis would be relying on empirical studies and controlled (quasi-)experiments to compare with other approaches, as defined in this dissertation. Due to the effort required, and the operational difficulties of conducting such experiments in the field of software engineering, it was decided to use a case study based approach for evaluating the applicability and efficiency on the usage of AOM and AOM frameworks, and to conduct a small experiment for discarding validation threats that should persist. The research strategy for each of them is detailed in Chapter 7 (p. 115) and Chapter 8 (p. 129).

The independent experimental validation of claims is not as common in Software Engineering as in other, more mature sciences. Hence, the author stresses the need to build reusable experimental packages that support the experimental validation of each claim by independent groups. Therefore, the (quasi-)experiment detailed in Chapter 8 (p. 129) was designed as an *experimental package*, to be performed in different locations, and lead by different researchers, in order to enhance the ability to integrate the results obtained and allow further meta-analysis on them.

4.6 CONCLUSION

There are fundamental research questions directly related to the current research trends and challenges in the area of Model-Driven Engineering, viz. (i) what forms should runtime models

take, (ii) how can the fidelity of the models be maintained, and (iii) what role should the models play in validation of the runtime behavior? Some properties of AOM systems are known to be particularly suited to solve systems with high-variability needs — the so called *incomplete by design* systems. In what concerns the current research on AOM, three main goals/questions were identified: (i) the study of undocumented and scattered patterns, and the identification of new patterns for the construction of a pattern language for AOM, (ii) the construction of a generic framework for building AOM-based systems, and (iii) the study on the benefits and liabilities of using such technology. Aligned to a pragmatist view of truth, valuing acquired practical knowledge, the author proposes to answer the above goals/questions, and validate them through the usage of mixed methods, among which are (i) *observational* and *historical* methods for the contributions regarding the pattern language, (ii) industrial case-studies, and (iii) (quasi-)experiments performed in academic contexts.

Chapter 5

Pattern Language

5.1 On Patterns and Pattern Languages 59
5.2 General Context 61
5.3 Technical Description 63
5.4 General Forces 64
5.5 Core Patterns 67
5.6 Evolution Patterns 78
5.7 Composing the Patterns 90
5.8 Conclusion 91

This chapter focus on Goal 1 of this thesis, namely **to formalize the underlying patterns of systems which domain model need to change frequently during runtime.** We discuss when do these type of systems occur, what are their advantages, and what are their underlying requirements. We further delve into the existing recurrent solutions that cope with each of them. Each pattern also exposes the benefits and liabilities of each specific solution. This main outcome of this goal is the establishment of an unified conceptual pattern language, which allows architects and designers to recognize and adequately use some of the best practices in software engineering that cope with these type of systems.

5.1 ON PATTERNS AND PATTERN LANGUAGES

The opening of third book on Pattern Languages of Program Design [MRB97] starts with the following sentence: *"What's new here is that there's nothing new here"*. This single assertion characterizes the epistemological nature of patterns, in what concerns its methodology and goals; patterns result from the observation, analysis and formalization of empirical knowledge in search for stronger invariants, allowing rational design choices and uncovering newer abstractions. A pattern should not report on surface properties but rather *capture hidden structure* at a *suitably*

general level. A comprehensive discussion on the epistemology of patterns and pattern languages can be found in a recent work by Kohls and Panke [KP09]:

> *The argument that there is "nothing new" in a pattern must be rejected; otherwise there would be nothing new to physics either, since physical objects and the laws of physics have been around before, just as design objects or programming styles have been around before somebody sets up a pattern language or collection. The discovery and description of a new species is without question considered scientific progress. Of course, the animals are not new – they lived there before – but they are newly discovered. In the same way, "the most important patterns capture important structures, practices, and techniques that are key competencies in a given field, but which are not yet widely known"* [Cop96].

5.1.1 What is a Pattern Language?

As discussed in § 1.7 (p. 9), a pattern may be regarded as a recurrent *solution* for a specific *problem*, that is able to achieve an *optimal balance* among a set of *forces* in a specific *context*. We could succinctly define each pattern as a triplet $p = \langle problem, forces, solution \rangle$. A pattern catalog would thus be defined as a non-empty finite set of patterns on a specific domain $PC = \{p_1, p_2, ..., p_n\}$.

A pattern language (PL) is an extension of a pattern catalogue in the sense that it has a set of patterns; but it extends on the notion of catalogue by also incorporating the relationships between its elements. Each pattern may be related to other patterns by different relationships, so $r = \langle p_a \rightarrow p_b, \mathtt{description} \rangle$ where $(r \in PL)$. The concept is close to that of an *ontology*, since the language could be synthesized as a formal representation of knowledge (in a specific domain), and the relationships between its concepts. Yet, while an ontology aims to be *descriptive*, a pattern language is *prescriptive* — it provides normative instructions intended to have an impact on shaping the act of design.

5.1.2 Form

The patterns community have been experimenting with several structures of the pattern description. There is the original structure that has been defined by Alexander *et al.* in their book A PATTERN LANGUAGE: TOWNS, BUILDINGS, CONSTRUCTION [AIS77], and is commonly known as the *Alexander's Pattern Language* format (APL). Then, there is the seminal work of Gamma *et al.* [GHJV94] where a different format was used specifically tailored for the area of software engineering, commonly known as the *Gang of Four* format (GOF). Both have benefits and liabilities: the APL is implicitly structured, and results in a fluid, narrative-like text, persuading the reader to identify herself with the pattern; the GOF form poses a more methodological partitioning with several explicit subsections.

This work will use elements of both the APL and GOF forms, by explicitly structuring the pattern into the following subsections when needed:

1. **Summary.** An introductory paragraph, which sets the intent of the pattern, and possibly links to other patterns.

2. **Context.** Usually a scenario in which the problem occurs and where this pattern can be applied.

3. **Problem.** This section starts with an *emphasized* headline, which gives the essence of the problem in one or two sentences. After the headline comes the body of the problem, describing the empirical background of the pattern, and the range of different ways the pattern can be manifested.

4. **Solution.** This section also starts with an *emphasized* headline, which describes the concrete action necessary to solve the stated problem.

5. **Example.** A situation where the pattern has been applied, with a possible graphical representation, listing the classes and objects used in the pattern and their roles in the design.

6. **Consequences.** Applying a pattern generates a resulting context, where the resolution of the forces now pose benefits and liabilities.

7. **Implementation notes.** Some additional concerns on effectively applying the pattern and dealing with lower-level issues, such as *performance* and *memory consumption*.

8. **Known Uses.** A pattern is a pattern because there is empirical evidence for its validity. This section points to systems where the pattern has been previously observed.

9. **Related Patterns.** The relationship to other patterns of the same language, or more other domains, thus forming the basis of a pattern language.

The rest of this chapter is organized as follows. A general context is established in § 5.2, by presenting a narrative of a real-world scenario. It is followed by an overview in § 5.3 (p. 63) of the technical background needed to apply these patterns. From the general context, § 5.4 (p. 64) drafts some common design forces shared by all the patterns here formalized. Then, § 5.5 (p. 67) and § 5.6 (p. 78) will describe the seven patterns central to this chapter. Finally, all the patterns are unified into a proposed evolution of the pattern language in § 5.7 (p. 90), first drafted by Welicki *et al.* [WYWBJ07].

5.2 GENERAL CONTEXT

In 2005 the author was leading a software project consisting on the construction of a geographical information system which would help a department of architectural and archaeological heritage

to manage both their inventory and business processes[1]. Although our development methodology was a slight variant of *eXtreme Programming* [BA04], we were considerably restricted in applying some of the practices for this particular project: (i) it was bided, so the cost was fixed, (ii) we could not reduce the scope, although it was systematically enlarged, (iii) we could not have an on-site costumer, and (iv) somehow, the result should be considered a success at all cost.

Our problems begun in the very first official meeting we had. Because the bid was made years before the official start of the project, the stakeholders' understanding had evolved since then. Therefore, the initial requirements were no longer a reflection of their current manual processes. Our contract enforced the deliver of a *requirement analysis* document which had to undergo validation before starting the development. And so we begun the task of collecting requirements... for two years.

At first, this seems a good example of how it should not be done; two years collecting requirements smells like a good old *waterfall*. However, our mere presence was directly contributing to this status. We started to question things they had for granted, and in the process of formalizing their practices, we uncovered inconsistencies which could not be solved promptly. This resulted in a series of analysis iterations, where the stakeholders' had to re-think their goals, their processes, and their resulting artifacts, before we could synthesize a coherent domain model.

At the end of those two years, we were strongly convinced of one thing: no matter how many time we invested in analysis, the resulting system would hardly be considered finished. As an example, consider the following requirement: they needed to collect the physical properties of archaeological artifacts found in excavations. At first, *length*, *width* and *height* seemed a good measure. But some artifacts are highly irregular, like a three thousand year old *jar*. For these, *weight* and *material composition* are greatly more useful. Other artifacts, like *coins*, are very regular and rely on different properties, like *radius* and *thickness*. Then we have things in-between, such as *plates*. The more we categorized, the more complex and longer the hierarchy become, without any confidence we would be able to cover all the exceptions. Our model was being haunted by *accidental complexity*, § 1.4 (p. 6), and a simple solution urged objects to be characterized by the end-user according to a pre-defined set of properties (which were not pre-defined at all). Of course users are no programmers, so they needed to add new properties and create new hierarchies *on-the-fly* from inside the application, without being explicitly aware of the underlying model.

As previously discussed in Chapter 1 (p. 1), this is a clear example of an *incomplete by design* system [GJT07]. The author will make usage of this story henceforth to illustrate parts and pieces of the patterns.

[1] This concrete example was also used as a case-study for this dissertation, and a detailed description can be found in Chapter 7 (p. 115).

5.3 TECHNICAL DESCRIPTION

The separating line between data and model is blurred when speaking about *meta-data*, in the sense that everything is, ultimately, data; only its purpose is different. For example, the information that some particular *Video* in our system is named *The Matrix*, and another one named *Lord of the Rings*, is called data for the purpose of using it as an information system for video renting. We could hypothetically draw a line encompassing all the objects that account for *data* (normally called *instances*) and name it the *meta-level-zero* (M_0) of our system.

The way we would typically model such a simple system in an object-oriented language would be to create a class named *Video*, along with an attribute named *Title*. But what may be considered the model in one context, may be seen as data in another, e.g., the compiler. As such, this information is meta-data in the sense that it is data about data itself: in fact, it conveys a very crucial information, which is the data's structure (and meaning), for the purpose of specifying an executable program. Once again, we could draw a line around these things that represent information about other things — classes, properties, etc. — and call them *meta-level-one* (M_1), or simply *model*.

But, what exactly is a class, or a property? What is the meaning of calling a method, or storing a value? As the reader might have guessed, once again, there is structure behind structure itself — an *infrastructure* — and the collection of such things is called *meta-level-two* (M_2), or *meta-model* for short (i.e., a model that defines models), which is composed of *meta-classes*, *class factories*, and other similar artifacts.

Hence, when we talk about data (or instances) we are referring to M_0 — bare information that doesn't provide structure. By model we are referring to M_1 — its information gives structure to data. By meta-model we are referring to M_2 — information used to define the infrastructure. And so on...

Ultimately, depending on the system's purpose, we will reach a level which has no layer above. This "top-most" level doesn't (yet) have a name; in MOF [OMG10a] it is called a *meta-meta-model*, due to being the third model layer[2]. This building up of levels (or layers), where each one is directly accountable for providing structure and meaning to the layer below is known as the *Reflective Tower*, a visual metaphor that can be observed in Figure 5.1 (p. 64).

All this would not be very useful if it did not had a purpose. We already mentioned the compiler, whose task is to read a particular kind of information (known as *source code*) and translate it into a set of structures and instructions (known as a *program*), which would later be executed by a computer — a process known as *compilation*. The compiler acts as a processing machine: the input goes into one side, and the outcome comes from the other. Once the compiler has done its job, it is no longer required, and so it does not *observe* nor *interact* with the final program. Should we wish to modify the final program, we would need to change the source

[2] Would it be the sixth, we seriously doubt anyone would apply the same prefix five times.

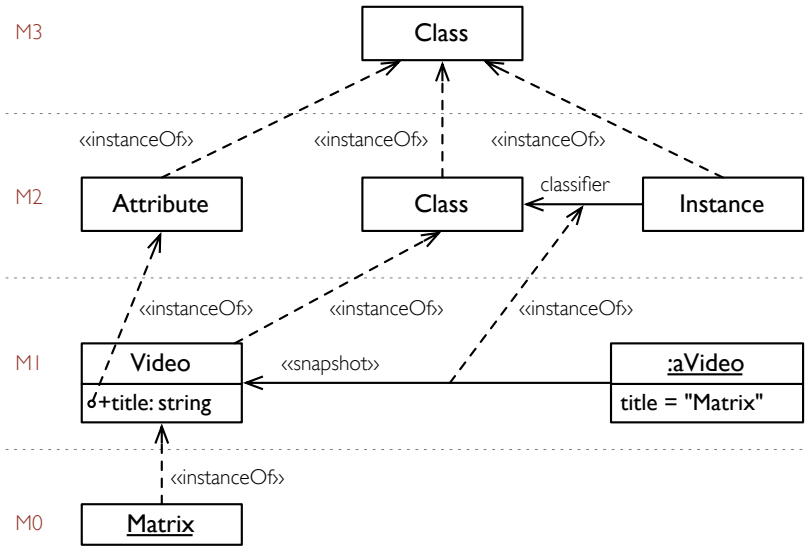


Figure 5.1: The *Reflective Tower* of a video renting system, showing four layers of data.

code and handle it again to the compiler.

Now let us suppose we wanted to add a new property to a *Video*, like the name of its *Director*, or create new sub-types of videos as needed, like *Documentary* or *TV Series*, each one with different properties and relations? In other words, what if we need to *adapt* the program as it is running? For that, we would need both to observe and interact with our running application, modifying its structure on-the-fly (the technical term is during *run-time*). The property of systems that allow such thing to be performed is called *Reflection*, i.e., the ability of a program to manipulate as data something representing the state of the program during its own execution. The two mentioned aspects of such manipulation, observation and interaction, are respectively known as *introspection*, i.e., to observe and reason about its own state, and *intercession*, i.e., to modify its own execution state (*structure*) or alter its own interpretation or meaning (*semantics*).

The technique of using programs to manipulate other programs, or the running program itself, is known as *meta-programming*, and the high-level design of such system is called a *meta-architecture*. The debate on the exact meaning of this word seemingly due to the *meta* prefix which can be understood as being applied to the word architecture (i.e., an architecture of architectures), or as a subset categorization, was already mentioned in § 2.2.6 (p. 25). For the purpose of this work, we will stand with the latter, i.e., *a meta-architecture is a software system architecture that relies on reflective mechanisms*.

5.4 GENERAL FORCES

Each pattern has a set of forces, things that should be weighted in order to achieve a good solution. Because the patterns here formalized are deeply connected, most of them share a good amount of forces in common. Figure 5.2 (p. 65) shows a schematic relationship among some of the following

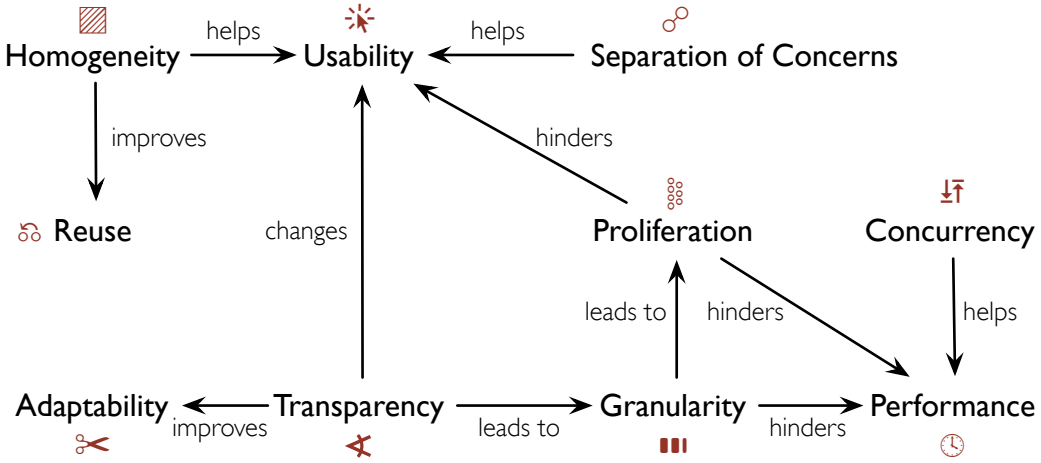


Figure 5.2: The relationship among forces of object-oriented meta-architectures.

forces:

1. **Transparency.** How much of the underlying system is available through reflection? In other words, to which degree does the infrastructure expose its own mechanisms for observation and manipulation? We may regard a system which is more transparent to improve usability in the sense that adds more power to it (hence, the user is able to do *more*). On the other side, a lot of transparency exposes details that can hinder its understandability, and consequently, its usability.

2. **Usability.** This is defined as *"the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"* [ISO98]. In this sense, meta-architectures, particularly Adaptive Object-Models, have two type of target users: (i) those which develop and evolve the infrastructure, and (ii) those who use the public facilities of the infrastructure to develop domain specific systems. Design choices may have different influences on the usability of different target users. This force is actually a product of several other forces.

3. **Separation of Concerns.** This is a general design force the establishes the fact that a particular functionality of a systems should be the concern of a different component – in this case, a different level of the reflective tower. For example, M_1 should be reserved to only express domain-level concerns, but most systems regard it as immutable during runtime. Thus, accidental complexity arises when this level is tweaked by non-domain concerns which should belong to M_2.

4. **Concurrency.** Is a general counter-force to reflective meta-architectures, mainly due to integration mismatch (i.e., tight interconnection among different level artifacts, causal connection among entities to provide consistency in the meta-representation of the system,

information flux among levels, etc.). Concurrency is mainly relevant due to performance and distributivity concerns, and has been a common issue in database design.

5. **Granularity.** Represents the smallest aspect of the base-entities of a computation system that are represented by different meta-entities, depending on the reflectivity scope – structural and/or behavioral. Typical granularity levels are classes, objects, properties, methods and method calls. The particular choice of the level of granularity is driven by its transparency, and has consequences on the resulting systems' object proliferation and performance.

6. **Proliferation.** Increasing the reflectivity granularity, e.g., by representing method calls as objects, leads to object proliferation, in the sense that more elements exist to represent the system's state. Likewise, more elements typically means more communication among them, increasing information flux and likely hindering overall performance.

7. **Information Flux.** Measures the amount of information that is exchanged between elements of a system to perform a desired computation. Depending on the meta-architecture design, instances typically exchange information with its class, classes with their meta-classes, and so on...

8. **Lifecycle.** The period of the system execution in which a specific meta-entity has to exist. For example, structural meta-entities that define an information system may be considered as persistent (or having a long life-cycle). On the other hand, introspective aspects have a shorter life-cycle (typically, only during the execution of the application). This has a correlation with granularity.

9. **Performance.** This is also a general engineering force that may mean short response time, high throughput, low utilization of computing resources, etc.

10. **Adaptability.** Characterizes a system that empowers end-users without or with limited programming skills to customize or tailor it according to their individual or environment-specific requirements.

11. **Reuse.** Is the ability of using existing artifacts, or knowledge, to build or synthesize new solutions, or to apply existing solutions to different artifacts. For example, one can reuse the persistency engine, typically tailored to persist data, to also persist model and meta-model elements. Reusing generally leads to a reduce of overall systems complexity and improves usability.

5.5 CORE PATTERNS

The relationship between the following patterns, and the way they extend the pattern language for adaptive object-models by coupling to the TypeSquare pattern, is presented in Figure 5.3:

1. **Everything is a Thing.** Which addresses the problem of having multiple representations of the same underlying concept.

2. **Closing the Roof.** A pattern that encloses the structure and meaning of a meta-architecture by stopping the seemingly infinite escalation of meta-levels.

3. **Bootstrapping.** Which solves the fact that any enclosed structure able to define itself relies on a (small) set of basic definitions, upon which it can build more complex structures.

4. **Lazy Semantics.** A pattern that defers the meaning of a definition until it is absolutely needed, essential for bootstrapping to occur without the danger of infinite recursions.

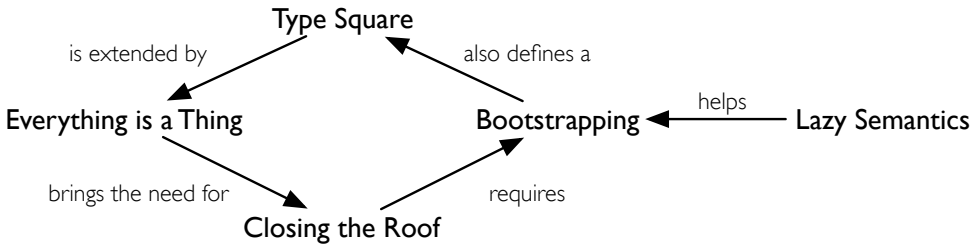


Figure 5.3: Pattern map for core patterns of meta-architectures, extending the pattern language by linking to TypeSquare.

5.5.1 Everything is a Thing Pattern

The system, with its several types of composing parts, needs to be adapted. Meta-architectures make use of elements available at runtime (i.e., models and meta-models) to specify the system's behavior. The system's data is observed and manipulated according to such elements, addressing concerns such as persistency, behavioral rules, graphical user-interfaces, and communications, among others.

Context

Back to the story started in § 5.2 (p. 61), our system began as a simple variant of the TypeSquare pattern that included attributes, relations, compositions, etc. In fact, it was heavily inspired in UML class diagrams and we were trying to infer as most functionality as possible only based on it. At first it was sufficient to store the model description (i.e., EntityTypes, AttributeTypes,

etc.) in a separate XML file, and distribute it over client applications and load it at start-up. Truth be told, to modify the domain model we had to modify the XML file, so there was not that much run-time "adaptivity". There was also a "mapper", with the purpose of interpreting the XML file into runtime elements. Then we had a GUI engine which followed a set of heuristic rules and was able to automatically create a user interface by, like everything else, observing the system's definition.

The lack of **homogeneity** was a problem first spotted with the need to actually manipulate the domain model at runtime. Although easy to deal with, the round-trip to XML was ugly. Also, changing the name of an AttributeType or of an EntityType required specialized operations, such as ChangeAttributeTypeName or ChangeEntityTypeName, that established the degree of **transparency** at the model level, but what was really bugging us was the logic being **duplicated** all around. The GUI engine inferred the user interface to manipulate the data level, but we were implementing by hand most of the same rules to manipulate the model level. Then we realized that relying on XML to persist the model would not work well in a **concurrent** environment. We asked ourselves: two types of representation for the entities of our system? If we have the infrastructure to manipulate data, why don't we **reuse** it to manipulate meta-data?

Problem

How to represent all that needs to be reflected upon? Clearly, we were lacking a fundamental, **unifying** principle. We have a system that observes and manipulates data, but it cannot do the same for meta-data? Why may we use a certain operation to change the attribute value of any instance, like setting the name of a person to John, but a different operation is required to change the name of a model element? Why could the data be stored in a warehouse, but needed the XML to store the model? We had **decoupled** the system from the domain, but we were coupled to what we believed to be a fixed structure; a false belief, since it soon needed to evolve. Our implementation pointed to a system that would need a large number of specific operations and components to manipulate the meta-level, and that number would increase in direct proportion with the system's **transparency**. What was so different between elements of M_0 and elements of M_1? The solution was right in front of us: the system knew how to manipulate *instances*, so we needed to make the elements of our model to be also *instances*.

Solution

Make all system's elements specializations of a single concept, regardless of their model level. These highly generic concepts are *Things* (or *Instances*, or *Objects*...). They are a single, unifying, primitive structure, as seen in Figure 5.6 (p. 79). To manipulate data or model elements, the system always relies on the manipulation of *Things*, that have a common set of basic capabilities for their own observation and manipulation. By **homogenizing** these concepts, the mechanisms

that deal with such generalizations don't need to be specific to every kind of entity. For example, setting the name of a type is performed as setting the attribute called *name* of that instance. Consequently, this increases the degree of reflection **transparency** of the system.

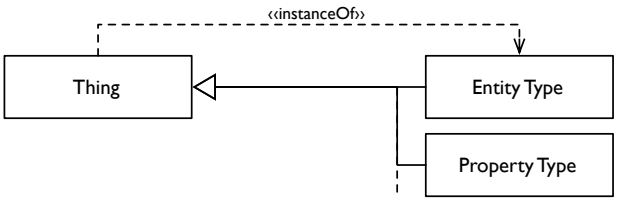


Figure 5.4: Class diagram of the EVERYTHING IS A THING pattern, which extends the TYPESQUARE by making EntityType a specialization of Entity (here represented as Thing).

Lets suppose that the persistency mechanism focuses on loading/saving states of *Things*; then the same mechanism can be reused for both levels (whether they are base-level objects, or types). This is also valid for graphical user-interfaces (GUI) and other features relying on the system's reflective properties. A known use is the *Oghma* framework, Chapter 6 (p. 93), which is able to render a GUI for editing meta-levels. This GUI is dynamically generated using the same rules as those used for the base-level. For example, every enumeration is rendered as either a combo-box (if the property has an upper-bound cardinality of 1), or a check-list (for more than 1). Because the concept of enumeration is equal both in the user defined model (e.g., the gender of a person) and in the system's meta-model (e.g., the rule of an association), both are rendered in the same way.

Example

A snippet of a C# unit-test, asserting several properties that hold after implementing this pattern, can be found in Source 5.1. Line 1 creates a new container loaded with the system's basic infrastructure. Line 2 finds and strongly types the Thing with an Identifier named Entity. Line 4 and 6 are sanity checks. Line 5 states that everything that is defined inside the container derives from Thing. Line 7 checks that if the model says the meta of a Thing is an Entity, then the infrastructure ensures it is typed as one.

```
1 var m = new ThingContainer();
2 var entity = m.OfType<Entity>().ByIdOrDefault("entity");
3
4 Expect(entity, Is.Not.Null);
5 Expect(m.ToList(), Is.All.AssignableFrom<Thing>());
6 Expect(m.ToList().All(t ⇝ t.Meta.Identity ≡ entity.Identity), Is.False);
7 Expect(m.Where(t ⇝ t.Is(entity)), Is.All.AssignableFrom<Entity>());
```

Source 5.1: Instantiating a self-compliant meta-model and asserting key properties.

Another snippet listed in Source 5.2 shows the usage of these features. Lines 9, 11 creates a new entity car and instantiates it. Line 13 verifies there are no violations for that instance. Lines 14 – 17 creates a new entity vehicle, with a mandatory attribute. Line 19 changes the inheritance of car, and 22 checks that there is now a reported violation due to the mandatory parent entity attribute.

```
1  var car = entity.New<Entity>();
2  var attr = m1.Get<Entity>("attributetype");
3  var c1 = car.New<Thing>();
4  Expect(c1.Violations, Is.Empty);
5  var vehicle = entity.New<Entity>();
6  var p = attr.New<AttributeType>(m1, "name");
7  p.Owner = vehicle; p.lowerBound = 1;
8  car.ParentEntity = vehicle;
9  Expect(c1.Violations, Is.Not.Empty);
```

Source 5.2: Creating and modifying model definitions in runtime.

Consequences

There are some **liabilities** to this pattern, which are direct consequence of the level of transparency. The model can be changed in many more ways than if we don't have specialized mechanisms to manipulate it. This results in a higher **coupling** between meta-levels, mainly due to an increase of **information flux**. For example, instead of a type having a specialized field to hold its name, it would have to rely on holding it in a separate object (attribute), which is defined by its meta-type. The type would thus need to exchange information with the meta-type to access its own name. Considering that the model may change anytime, the same thing is even more evident with base-level objects. The fact that more objects are needed to hold basic properties of a system leads to what is known as **object proliferation**. Both information flux and object proliferation may contribute to a decrease in the **overall performance** of the system.

Known Uses

In set theory, everything is a set. In LISP, everything is a list. In the object-oriented world, everything is an object. Well, not quite everything – there are binary relations, function applications, and message passing. But the principle still applies, in the sense that there is a single, unifying, primitive aspect (set, list, object…) defining the fundamental underlying structure.

Known uses of this pattern include the Meta Object Facility (MOF) and pure object-oriented languages like Smalltalk. Both the MEMENTO and HISTORY OF OPERATIONS patterns can be used for storing the states of Things, as seen in § 5.6.1 (p. 79) and § 5.6.2 (p. 83).

5.5.2 Closing the Roof Pattern

Also known as *Self-Compliance*, *Rooftop*, *Idempotence*.

Context

By seeing the model as data, one can use Everything is a Thing to manipulate the several model levels using the same mechanisms. But, whenever we raise up a level, we find ourselves needing another (probably more abstract) level to describe it.

Problem

How can we ensure there won't be an infinite number of model levels? Every time we need to observe or change a particular level we would need an higher-level model that describes it. In other words, we want to provide enough *transparency* between levels so that they are both *observable* and *changeable*. But this points towards a potentially unbounded number of levels (since each level requires an higher — more or equally abstract — level to describe it), thus resulting in a seemingly infinite escalation. Otherwise, we would be *dependent* on an external interpretation of our system (e.g., what is the name of a type? it's the field of this particular implementation class), thus reducing overall *homogeneity*.

Solution

Choose a top-most model level and make it compliant to itself. Make the top-most level as simple as possible, with the bare information needed to specify more extensive (and eventually more complex) models. The metamodel should be *expressive* enough to define its own structure (and possibly its own semantics). Once you achieve it, you would not need to go up another level. This particular solution is also known as a *meta-circular* model (i.e., the primary representation of the model is a primitive model element in the model itself, a property known as *homoiconicity*). One primary advantage is that we no longer require (or depend on) an *external representation* of our system.

Consequences

One main liability of this pattern is the potential to be trapped into infinite loops, since interpreting the topmost level will require that same level to be introspected. This may pose threats on the decidability of the meta-model, particularly when semantics-level reflection is provided. Bootstrapping and Lazy Semantics may be used to solve such infinite dependencies.

Known Uses

There are several known uses for this pattern, from model-driven (e.g. MOF) to grammar definitions (BNF) and XML. For example, XSLT is a meta-circular interpreter, since XSLT programs are written in XML, thus allowing XSLT files to be written that manipulate other XSLT files. Another example of XML being self-describing is on the usage of XSD.

5.5.3 Bootstrapping Pattern

Context

Closing the Roof, requires your model to provide enough expressiveness to describe itself. However, a first instantiation of the model is still needed to allow its own instantiation, and the instantiation of lower-levels.

Problem

How do we start defining a model whose definition depends on itself? This often leads to a chicken-and-the-egg problem, where you may find yourself in need of definitions which may not yet have been defined, and in turn those definitions need whatever you are now defining. This is similarly to what happens when writing a dictionary, how do you write the meaning of a word if the words you'll use also require a meaning? Normally, you start with a small set of existing resources and then proceed to create something more complex and effective.

Solution

Provide a minimalistic core of well-known elements from where you can build more complex constructions. The smaller and simpler it is, the less the system will be bound to specific model elements, and the less likely the top-most level will need to change in the future. A thorough formalization of the core will benefit the system, as it will serve as a foundation for all the other levels. In order to help solving cyclic dependencies, Lazy Semantics may be used. Although maintaining the core small and simple, boot-strapping a system also requires a substantial degree of expressiveness, which will eventually result in a considerably powerful infrastructure.

The term bootstrap seems to have its roots on a metaphor derived from pull straps sewn onto the backs of leather boots with which a person could pull on their own boots (without outside help). The term was heavily to refer to the seemingly paradoxical fact that a computer cannot run without first loading its basic software, but to do so it needed to be running.

Example

The excerpt of code listed in Source 5.3 (p. 73) shows a small part of the infrastructure bootstrapping of *Oghma*, where one can observe the structural core being defined, namely the interface,

entity and the inheritance relationship.

```
<entity id="interface" name="Interface" inherits="objecttype" isinstanceof=
    "Oghma.Core.Structural.Interface">
  <list columns="{_identity}|{_name}" />
</entity>

<entity id="entity" name="Entity" inherits="objecttype" isinstanceof="Oghma
    .Core.Structural.Entity">
  <list columns="{_identity}|{_name}|{_isAbstract}" />
  <attr id="_strPattern" name="ToString" domain="string" cardinality="0..1"
      />
  <attr id="_rowPattern" name="ToRow"    domain="string" cardinality="0..1"
      />
  <attr id="_isAbstract" name="Abstract" domain="boolean" cardinality="1"
      default="false" />
</entity>

<relationship id="rel_entity_parententity">
  <node entity="entity" id="_parent" name="Parent" cardinality="0..*"
      navigable="true" />
  <node entity="entity" id="_child"  name="Child"  cardinality="0..1"
      navigable="true" />
</relationship>
```

Source 5.3: A small excerpt from the Oghma infrastructure bootstrapping.

Known Uses

The most common known uses of this pattern are programming languages and their compilers (e.g. Smalltalk and LISP). The advantages of starting with a small self-describing core to define the whole system are very patent in the following war story from Alan Kay:

> *It was easy to stay motivated, because the virtual machine. running inside Apple Smalltalk, was actually simulating the byte codes of the transformed image just five weeks into the project, A week later, we could type 3 + 4 on the screen, compile it, and print the result, and the week after that the entire user interface was working, albeit in slow motion. We were writing the C translator in parallel on a commercial Smalltalk, and by the eighth week, the first translated interpreter displayed a window on the screen. Ten weeks into the project, we* crossed the bridge *and were able to use Squeak to evolve itself, no longer needing to port images forward from Apple Smalltalk. About six weeks later, Squeak's performance had improved to the point that it could simulate its own interpreter and run the C translator, and Squeak became entirely self-supporting."*

For model languages, the most well-known use is probably MOF, which was first defined from a small subset of UML structural diagrams, and eventually evolved to become the core meta-metamodel of UML.

5.5.4 Lazy Semantics Pattern

The pattern arrises when there is the need to cope with expressions or model constructions where its meaning is initially undefined or subject to change.

Context

When developing systems based upon meta-architectures, developers may find themselves in need of definitions which may not yet have been defined, and in turn those definitions need whatever they are defining. Or the definitions may change in time. This may very easily happen when BOOTSTRAPPING and when CLOSING THE ROOF [**?**].

Let us consider three different examples where this pattern may occur, viz. (i) in dynamic language constructs, (ii) in meta-programming, and (iii) in meta-modeling.

Example A. Consider the following Smalltalk code:

```
1 Vehicle wheels
```

What is the meaning of this code, i.e., its semantics? From a language point of view, we know it is sending the message wheels to the object Vehicle. We would expect that such evaluation would produce some behavior. But what if we cannot define the meaning of wheels until we evaluate the running state of the system? Or what if the messages semantics depend on a specific execution context?

Example B. Consider the following C# code:

```
1 public class Customer {
2   public string CustomerID;
3   public string City;
4
5   public static IEnumerable<Customer> GetCustomersInLondon();
6 }
```

How would we implement the GetCustomersInLondon() method? If all the objects were stored in memory, we could do something like this:

```
1 foraech(var c in Customers) if (c.City == "London") yield return c;
```

But, would Customer be mapped into a relational database, we would probably want to do an SQL query like this:

```
1 select * from Customers where city = "London"
```

The problem is, how do we abstract GetCustomersInLondon() so that it could work with both (or probably more) modes? How could we adapt its semantics to the executing context?

Example C. Consider we are bootstrapping a meta-modelling based system (for example, a self-contained AOM, which is using the CLOSING THE ROOF and BOOTSTRAPPING patterns).

We may start be defining what is a Class, by saying that there's a class named Class, with a property Name that will hold the name of the class. But for that we first need to define what a property is. We could backtrack and start by defining that first: a property is a Class named Property. Wait!... We haven't defined what a class is yet.

This situation is usually known as the chicken-and-the-egg problem. Which came first? The class or the property? Sometimes we may need some definitions that have not yet been defined, and in turn those definitions need whatever we are now defining.

Problem

How can we make the system able to handle different meanings depending on its current known state? The system is expected to be consistent, that is, the semantics that each model-level establishes are expected to be enforced over its instances (i.e., the elements of the lower model-level). Enforcing *consistency* at all times may be hard, if at all possible, as it requires to find a sequence of actions that accounts for all dependencies. For example, there may be cyclic dependencies when BOOTSTRAPPING. If the system is made more *tolerant* to undefined constructions, consistency may not be assessed at all times. The following forces are to be considered: (i) transparency, in what concerns the degree of the system that is to be exposed by reflection; even if the system doesn't rely on CLOSING THE ROOF, the problem may occur in different circumstances, (ii) granularity, since by defining a string as an thing, we are homogenizing the infrastructure and thus we are lead to (iii) reuse, due to the preservation of existing mechanisms to deal with different meta-levels, (iv) convergence, in the sense that we may consider the system to be eventually consistent and (v) complexity, which is an overall concern to be reduced.

Solution

Enforce meaning only when absolutely required. Pertain your system doesn't enforce any meaning up until it is absolutely necessary (e.g., your cyclic dependencies are resolved, or you have enough information to bind to a specific meaning). Change the execution from *call-by-value* or *call-by-reference* to *call-by-need*.

Example A. Referring back to this example, let us imagine that the start message is not defined,

i.e., the message could not be dispatched to a specific method. When this happens in smalltalk (or very similarly in Ruby), the interpreter reifies the wheels message into an object, and pass it as an argument of the doesNotUnderstand: message. The later could inspect the message and decide what to do with it. For example, it could decide that the method is a proxy to a relational table, and thus appropriately generate and query an SQL engine. In fact, while it could simply return the appropriate collection of wheels, it could also dispatch side-effects, like creating a new method that will specifically answer the wheels message.

In other words, the exact meaning (the semantics) of the wheels message is determined when it is needed. Moreover, since the precise meaning of the wheels message can change over time, it should only be determined when it is absolutely necessary. This is one of the reasons why dynamically typed languages are generally slower that strongly typed languages, since message passing cannot be transformed into a pointer derreferentiation.

Example B. Consider the following C# code, written in the LINQ DSL:

```
var q = from c in Customers
        where c.City == "London"
        select c;
```

The above expression is quite peculiar. Its type is IEnumerable<T>, and it will not be executed until it is absolutely needed, such as in the following snippet:

```
foreach (var c in q) Console.WriteLine("id={0}, City={1}", c.Id, c.City);
```

Here, the system will have to enumerate q. This will trigger some evaluation routines which will check the Customers collection. Let's consider that it was defined as:

```
[Table(Name="Customers")]
public class Customer {
  [Column(IsPrimaryKey=true)] public string CustomerID;
  [Column] public string City;
}

DataContext db = new DataContext("database.mdf");
Table<Customer> Customers = db.GetTable<Customer>();
```

In this case, the system will generate the appropriate SQL code by inspecting the LINQ expression, and pass it through the DataContext. It would then convert each row into a Customer instance, and yield it as items of the IEnumerable<Customer>. But, if the Customers was defined as a simple List<Customer>:

```
var Customers = new List<Customer> {
  new Customer() { Id = 1, City = "Paris" },
  new Customer() { Id = 2, City = "London" }
}
```

Then the DSL would be converted into a simple expression that would filter the collection of objects based on the given criteria:

```
var q = Customers.Where(c ⇝ c.city ≡ "London")
```

Once again, the precise meaning of the LINQ expression cannot be known *a priori*. The system's state when the expression is executed will determine the exact semantics of that expression.

Example C. One solution to the scenario described in this example is to relax on the mandatory definitions. Instead of requiring every class to have names, we would first create the class Class without handing it any name. Next, we would create the class Property. We proceed to say that a class has a property, and so on. Sometimes infinite cycles can be worked around by carefully relaxing on the structure and choosing the appropriate sequence.

Another solution would be to relax on the meaning of properties. Instead of a class having a property named Name, it could have a slot accessed through a string identifier, that would hold its name. After defining what a property is, we could then resort to define that getting the values of a property consists on searching the slots for the property name. Hence, we have two different semantics for accessing a property: (i) a property is nothing but it is name looked up in the slots, or (ii) a property is a known-object that maps to values. The precise meaning of property changes over time, and depends on the state of the system.

Related Patterns

One very similar pattern is LAZY EVALUATION, although it focus on a different aspect. LAZY EVALUATION may be resumed as delaying any kind of computation to the point where it is absolutely needed. For example, in the programming language Haskell, everything is lazily evaluated, allowing one to inductively define sets and only compute needed elements. For example, the following function defines an infinite list containing the Fibonacci series:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

In languages without lazy evaluation the computer would enter an infinite cycle trying to produce all the values of the list. Lazy evaluation doesn't require a flexible semantics. The above expression never changes its meaning (in fact, Haskell is statically typed).

The framework described in Chapter 6 (p. 93) also makes use of LAZY SEMANTICS to solve the cyclic dependencies that arise from the auto-compliance of the upper-most model level, achieved by CLOSING THE ROOF.

5.6 EVOLUTION PATTERNS

One of the key aspects of Adaptive Object-Models is their ability to allow changes to the model even at run-time. Model evolution is thus a recurrent problem that developers adopting this architecture face, since it may introduce inconsistency in its structure. This problem can be split into three complementary issues:

Track. How to keep track of the operations performed for evolving the system?

Time Travel. How to access specific key states of the system at any particular point of its evolution?

Evolution. How to introduce changes into the system while preserving its integrity?

We now present three domain specific design patterns that have risen from the experience implementing Adaptive Object-Models, and researching how other systems, particularly Object-Oriented Databases and Version Control Systems, deal with these problems [RL04, WE00, BEK+04]. These patterns contributed to the on-going effort on defining a pattern language for AOMs [WYWBJ07, WYWB07, WYWB09] and are:

History of Operations. Addresses the problem of maintaining a history of operations that were taken upon a set of objects.

System Memento. Deals with preserving the several states the system has achieved upon its evolution.

Migration. Addresses the concern of performing evolution upon the system while maintaining its structural integrity.

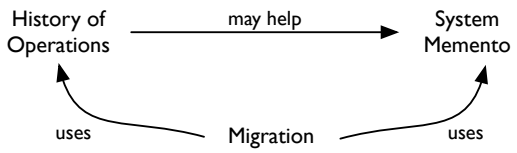


Figure 5.5: Data and metadata evolution patterns.

All patterns further presented are closely related, as depicted in Figure 5.5). MIGRATION depends upon the concepts of HISTORY OF OPERATIONS and SYSTEM MEMENTO (which in turn may

be helped by HISTORY OF OPERATIONS). MIGRATION orchestrates the coordination between the other two patterns, so that enough semantics is gathered to fulfill its intended purpose.

Patterns under the name of HISTORY and VERSIONING have been foreseen as part of the original pattern language for AOM proposed by Welicki *et. al.* [WYWBJ07], though they are here redefined as HISTORY OF OPERATIONS and SYSTEM MEMENTO respectively. The MIGRATION pattern did not belong to the original pattern language.

While the traditional AOM architecture only considers the M_0 and M_1 levels, nothing keeps system developers from defining higher-level models. Because the patterns that follow are intended to be applied regardless of the model level, the TYPE-SQUARE pattern is extended by using EVERYTHING IS A THING § 5.5.1 (p. 67) as explained in Figure 5.6.

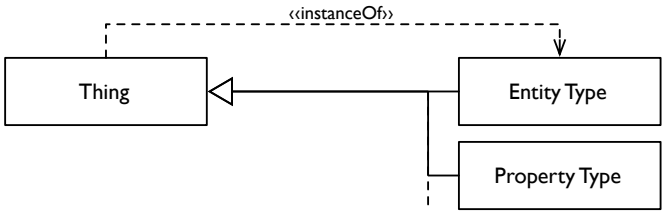


Figure 5.6: Applying EVERYTHING IS A THING for evolution patterns. Every model-level element inherits from a data-level element, and every data-level element is an instance of a model-level element. In closed meta-architectures, the top-most model-level element is an instance of itself.

The concept of Thing is here defined as representing both data (i.e. Entities and Properties), and metadata (i.e. EntityTypes and PropertyTypes). Any object of type Thing is actually an instantiation of an EntityType, thus allowing an unbounded definition of meta-levels. Eventually, the upper-bound may be delimited when a defined Thing is regarded as an instantiation of itself (or simply not defined), as discussed in § 5.5.2 (p. 71). Because every class in the model and metamodel derives from Thing, this extension allows one to explicitly state the IDENTITY [Fow02] of an object as will be described in § 5.6.2 (p. 83).

5.6.1 History of Operations Pattern

Addresses the problem of maintaining a history of operations that were taken upon a set of objects. An application based on the Adaptive Object-Model as the main architectural style is being developed, and there is the need to track the system's usage by end-users, including changes to both the knowledge and operational levels.

Context

Imagine an insurance company who's users keep changing the system's information at a fast pace. There is the need to keep track of *what*, *how* and probably *when* and by *whom* it has been changed. For this example system, meta-information is as important as the information itself.

Keeping track of the operations' history can go beyond auditing purposes, like performing statistical analysis (e.g. number of created instances per user), controlling user behavior (i.e. without recurring to explicit user access control), automating activities (e.g. finding systematic modifications to the information) or recovering past states of the system.

In an AOM based application, there are different levels at which Things can change (data, model...). For example, consider an EntityType named *Person* and a particular Entity named *John*. The kind of actions we may perform can be as simple as CRUD-like operations (e.g. deleting the Entity, or changing its name), or model operations (e.g. adding the attribute *Number of Children*, or moving it to the superclass).

The history of operations must be made available in the application, since it will be used by end-users. However, simply storing messages in a file or database makes the mapping between them and the operations over things, a complex (if at all possible) activity. Furthermore, as the underlying AOM interpreter evolves, the type of operations that should be recorded may also evolve. This can result in an set of messages to parse, with obsolete syntactic details that may no longer be directly mappable.

Problem

Given a set of Things, how do we keep track of the history of operations that were performed upon them, without knowing the specific details of each operation? We should take into consideration the following set of forces: (i) Encapsulation, since we do not want to pollute the system with logging structures wherever they are needed, (ii) Extendability, since we may want to add additional information to the history (e.g. *previous state*, *modifying user*, *last modification time*), (iv) Operations' Semantics, i.e., each operations should have enough semantics to allow automatization, (v) Simplicity, as occurred operations should be easy to store and retrieve, (vi) Modifiability, to allow operations to be expanded and evolved, (vii) Performance, to reduce the impact of overheads on the system, (viii) Reusability, to provide the same mechanism regardless of the model-level, (ix) Consistency, since operations should comply to semantic constraints assuring system's integrity, (x) Reproducibility, since operations should be able to be re-executed and achieve the same result (e.g. deterministic), and (xi) Resource Consumption, since additional manipulated and stored information should be carefully minimized.

Solution

Encapsulate the allowed operations in a set of commands that operate over Things. A sequence of invoked commands constitutes the History of Operations. Create Operations, using the Command pattern [GHJV94], with the responsibility of defining and encapsulating the types of modifications allowed (i.e. Evolution Primitives [RL04]). These may be elemental — Con-

crete Operations —, or grouped in a sequence — Macros – through the use of COMPOSITE pattern [GHJV94].

Every action taken by the application must occur by instantiating and executing a defined Operation. Creating an History object is as simple as storing the sequence of the invoked Operations. Each Operation will retain enough information in order to be mappable to the Things it operates over (*cf.* Figure 5.7). However, note that if operations are not versioned, they should be made either static or semantically equivalent upon evolution, otherwise the history may become unusable.

By using the HISTORY OF OPERATIONS pattern, developers can factor the responsibility of creating and storing modifications in a semantically rich way. This will allow an easier evolution of the underlying interpreter and other automatizations.

Example

Consider five employees from the automobile insurance department, working as a team. During a week, they create and alter information on the system, either from external demands (e.g. clients) or from the rest of the company. Namely, they subscribe clients to policies, answer to the events of new occurrences, and redefine conditions for upcoming policies.

On this particular week, the same client record happened to be edited by three different users. Yet, there was an incorrectly registered occurrence for that client, and it's important to understand why it happened in order to prevent future mistakes. The history of operations registered throughout the week allows users to find out exactly what happened: the occurrence was registered by one particular employee on Tuesday, because the client was wrongly chosen to begin with, since it was selected by searching his name, instead of his client-number. By Friday, the department's director wants to know how the week went, before the weekly meeting with his staff. He uses the system's functionality that collects several statistics from that week's history of operations, and realizes it was in fact a particularly busy week.

Consequences

This pattern results in the following benefits. Because every modification is abstracted into an evolution primitive (as an Operation), the history is made simply by storing the sequence of performed commands — *encapsulation* – which also *simplifies* the control of semantic/constraints

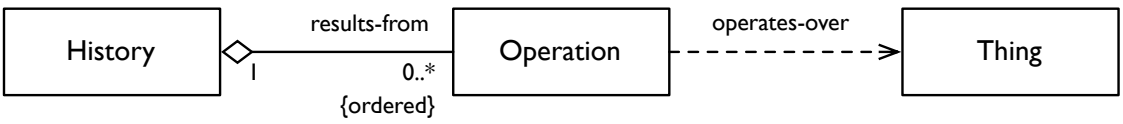


Figure 5.7: Class diagram of the HISTORY OF OPERATIONS pattern. A History results from a set of Operations done over Things.

checking, *auditing* and *security* issues. A side-effect of mapping the allowed operations to COMMANDS is the further promotion of *reuse*, *easier maintenance* and *consistency*. If enough information is stored with each evolution primitive — *semantics* — it becomes possible to playback the executed operations, although it requires that the behavior of all business objects is *deterministic*. The use of the COMPOSITE pattern to create *macros* of operations also addresses the issue of atomicity, when the fine-grained operations are not enough. Since an up-to-date data set can be rebuilt by applying all the history of commands to it, this design improves *Fault tolerance*. *Consistency* is also assured as each command represents an atomic operation – no change can be made to the data without using such a transaction, and transactions are applied sequentially. Finally, persistence using a this pattern is *transparent*, as operations are applied directly to the business objects with no need to use third-party mechanisms such as direct SQL access or object-relational mappings.

Nonetheless, the usage of this pattern incurs the following liabilities. The quantity — *space consumption* — of additionally stored meta-information may be considerable, as it will always grow with time, despite the size of the current valid objects and meta-objects. However, the use of compression techniques and the external archiving of unnecessary history may lessen the impact of this liability. Hence, *performance* may be affected because of the quantity of instantiated objects and the necessary pointer dereferencing/set joins associated with particular implementations. Overall, the final implementation may result in a *complex* solution to maintain.

Implementation Notes

The semantic consistency of Things can be kept by enforcing constraints defined at an upper abstraction level (i.e. operational-level constraints are defined at the knowledge-level). One way to enforce these constraints is to use *pre* and *post* operation conditions. Keeping operations as general as possible will leverage their reusability and maintainability, but leads to operations of low granularity. The use of CRUD-like operations is a good example, as they focus on very straightforward tasks, and cover a wide scope of use cases when combined.

However, some sequences of operations may be impossible to carry out while ensuring consistency at the end of each of them, although information would be in a consistent state upon completion of the entire sequence. Consider two classes, A and B, with a mandatory one to one relation between them, and two particular instances of these classes, a0 and b0, thus connected through that same relation. Suppose we replace b0 by a new instance b1, as the other end of the relation. If we consider only CRUD-like operations, three different operations would be needed: (i) the deletion of the relation between a0 and b0, (ii) the creation of a new relation between a0 and b1, and (iii) the deletion of b0. By the end of these operations, information would be in a consistent state, but that would not be the case just after each individual operation completes, since mandatory relations would not yet exist. As described, through the use of the COMPOSITE pattern, Operations can be grouped in sequences, or Macros. These macros are a means to

the reuse of operations, but may also be used to establish consistency-checking frames. Instead of checking the consistency of information after each individual elemental operation, it may be checked only at the end of the macro in which they are enclosed. This notion is akin to the concept of TRANSACTIONS in database systems.

Known Uses

Operations are structured using the COMMAND pattern [GHJV94]. The hierarchy of operations are also related to the COMPOSITE pattern [GHJV94]. The storage of information may be done similarly to the AUDITLOG pattern [Fow10a], though with more semantics to increase *traceability* and *automation*.

This pattern is common in *Object-oriented Database Management Systems* and *Data Warehouses* [RL04, WE00, BEK+04]. The *Prevayler framework* [Ope10] and the *COPE tool* [HBJ09] are also known to use this pattern, as well as the work presented in [AKK].

5.6.2 System Memento Pattern

Deals with preserving the several states the system has achieved upon its evolution. An application based on the AOM architectural style is being developed, and there is the need to access the state of the system at any point (present or past) of its evolution.

Context

Lets consider an heritage research center where its users keep collecting information as they perform their regular activities. Due to the nature of the research, uncertainty of the information is common, leading to several changes over time. While the pace of collected information may not be high, any change in the system is critical since it should prevent lost of historical (i.e., previous) information; even if deleted at one point, it should be easily recoverable if the need arrises.

For example, suppose we have an EntityType named *Archeological Survey* and a particular Entity called *Survey of the Coliseum*. At a certain point in time, the *Coliseum* could have been dated as 100AC, but more recent research could cast doubt on that date, and thus make it oscillate between 200BC and 500AC.

One can also consider a case in which this system has been running for some amount of time, possibly accumulating considerable information, e.g., several thousand *Archeological Surveys* could have been registered. In these types of systems there are several levels at which we want to persist the state of the objects as they are evolved (data, model, meta-model...). For example, through acquired experience, users may have found the need to additionally register the leader of each archeological expedition. As such, an evolution would need to take place at the model level, to accommodate a new property of the *Archeological Survey*'s EntityType.

Problem

How can we access the state of the whole system at any particular point of its evolution? We should take into consideration the following set of forces: (i) Reusability, since we want to use the same versioning mechanism regardless of model-level (i.e. data and metadata), (ii) Encapsulation, to prevent polluting the system with versioning logic everywhere it is needed, (iii) Identity, to be possible to reference either an object or one of its states, independently of each other, (iv) System-Level Semantics, since versions should represent the evolution of the system, and not of a particular object, (v) Time Independence, though evolution usually occurs with the passage of time, the system should not need to be aware of it (the concern is the sequence of changes), (vi) Accessibility, to be possible to access the system at any arbitrary point of its evolution, (vii) Space Consumption, since the data-set at hand should be kept to a manageable size, (viii) Concurrency, which can be addressed by allowing branching of information, but introduced the added complexity of integrating merging mechanisms, and (ix) Consistency, where any particular state of the system complies to integrity constraints (e.g. an M_0 object must be compliant to its M_1 definition).

Solution

Separate the identity of a Thing from its properties such that, by aggregating a particular State of Things, one can capture the global state of the system at any particular point of its evolution. Applying this pattern usually starts by decoupling Things from their States [Fow02]. While Things represent the identity of an object, States represent its content, which will evolve over the use of the system's information, as depicted in Figure 5.8 (p. 85)). A Version thus captures the global state of the system, by referencing all the valid States at some point of the system's lifetime. Each Version maintains references to those that gave origin to it (previous), and to those that originated from it (subsequent). Usually, however, each Version is based on a single previous Version, and will give origin to a single other Version, thus resulting in a linear timeline. However, more than one previous and/or next Versions may be considered, specially in concurrent usage environments, for purposes of data reconciliation.

Each individual Version may accommodate both instance and model-level Things. This results in a particularly useful design, since a change at the model-level can often lead to changes at the instance-level. In order to aggregate a consistent group of States, every Version need to be able to reference States from both levels. In fact, this is an essential issue for the MIGRATION pattern, since changes to the model usually require changes to the data.

Example

Consider the aforementioned *Survey of the Coliseum*. Over the last year new information about the Coliseum was acquired, through the study of newly found manuscripts. Users updated the

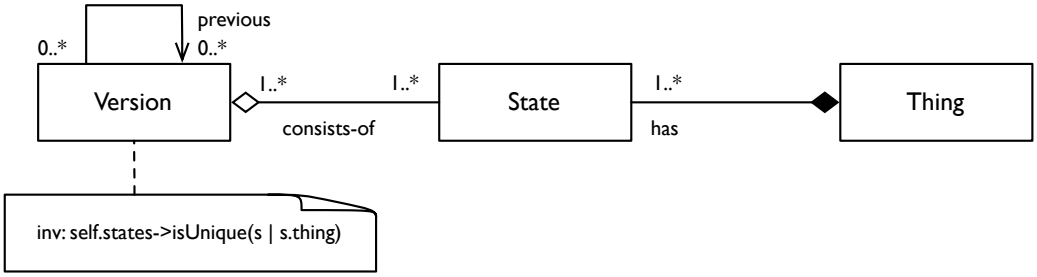


Figure 5.8: Class diagram of the System Memento pattern. A Version is a collection of States, one per Thing (i.e. there cannot be multiple states of the same thing in the same version).

information on the system, such that it would reflect their best knowledge at each phase of the research. Therefore, the description of this monument evolved over time. As such, several Versions may have been created, each representing a consistent point on the evolution of the available information. Thus, it becomes possible to access, and even recover, previous states of the system.

Eventually, the model may also need to evolve. As described in the example, a new AttributeType may be added to accommodate the name of the leader of each archeological expedition. Since an AttributeType is a Thing, a new Version will be created that references model-level States and Things.

Consequences

This pattern results in the following benefits. It is now possible to use the same versioning mechanism regardless of the model-level — **reusability**. By **decoupling** the state from the object, we are able to isolate the object's **identity**. Because the concept of version is now at system level — **system-level semantics** — instead of object-level, we are now able to address **consistency**. If multiple evolution branches are used, **concurrency** may be coped with more easily.

This pattern has the following liabilities. The quantity of stored information may be larger than affordable — **space consumption**. The choice of appropriate persistency strategies may reduce this issue. The branching of versions will require additional merging mechanisms. **Performance** may be affected by the overhead introduced while changing, storing and accessing information. It may increase the systems' **complexity** due to additional object dereferenciation.

Implementation Notes

It should be noted that a literal implementation of this approach may lead to an unnecessary use of space as the system evolves. Versioning systems typically deal with this issue by partially inferring, instead of explicitly storing, the complete set of states that define a particular version (i.e. by just keeping the *deltas*). Because this issue can determine the feasibility of a system, we present some notes overviewing one possible solution.

Consider the following sets of operations (i) create *carA*, (ii) create *wheelA*, *wheelB* and *carB*, (iii) modify *wheelA*, and (iv) modify *carA* and delete *wheelA*. The resulting set of versions can be observed as an object diagram in Figure 5.9.

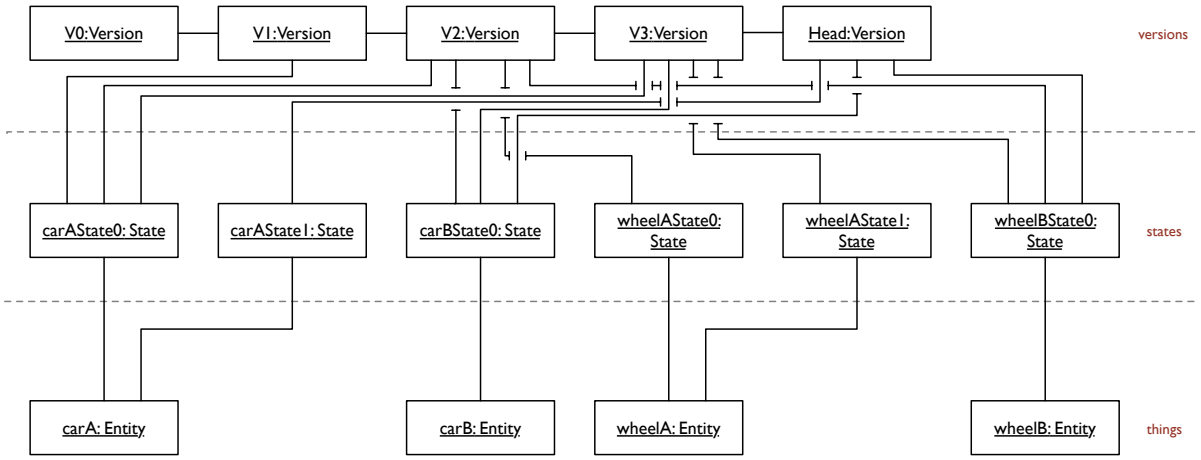


Figure 5.9: Object diagram for an example instantiation of the SYSTEM MEMENTO pattern.

Any Thing that doesn't change its State in any subsequent version, would have its State replicated across those versions. Using a strategy where only changes to states are stored, thus inferring (instead of storing) the complete set of states for any version, the stated example would become as observed in Figure 5.10.

In english, the inference rules can be summarized as: *if a state belongs to a delta, then it also belongs to the corresponding and subsequent versions, until a new state is defined or the null state is reached (i.e. when an object is deleted).*

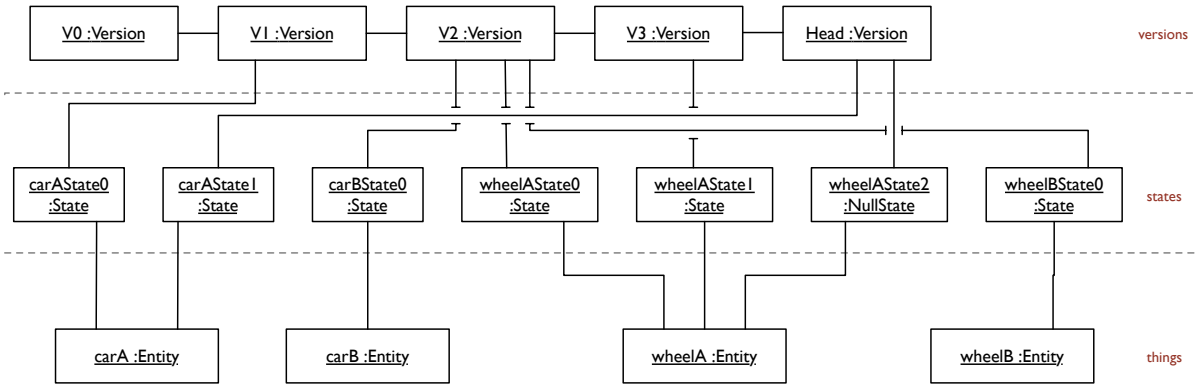


Figure 5.10: Object diagram for the *delta* strategy instantiation of the SYSTEM MEMENTO pattern.

Related Patterns

The patterns TEMPORAL PROPERTY [Fow10d], EFFECTIVITY [Fow10b], MEMENTO [GHJV94], TEMPORAL OBJECT [Fow10c], SNAPSHOT [CEF98], and several others [And, ABBL05], are di-

rectly related to the problem of storing the changing values of an object. However, none of them explicitly addresses the concerns of *system-level semantics* (i.e. they focus on the change of a single object instead of the whole system) and *Meta-modeling* (i.e. the change of an object's specification).

The MIGRATION Pattern, described in this work, uses SYSTEM MEMENTO to allow arbitrary evolution of the system between any two versions. The IDENTITY pattern [Fow02] is also used to decouple a Thing from its State.

Known Uses

This pattern is common on Wikis and Version Control Systems. The work presented in [ABBL05] also details the implementation of several versioning techniques in object-oriented design. Several *Object-oriented Database Management Systems* and *Data Warehouses* [RL04, WE00, BEK+04], as well as the *Prevayler framework* [Ope10] and the *AMOR system* [AKK], are known uses of this pattern.

5.6.3 Migration Pattern

Addresses the concern of performing evolution upon the system, while maintaing its structural integrity.

Context

An application based on the AOM architectural style is being developed, and it will be necessary to evolve model and data definition (assuring consistency) after system's deployment.

Example

Consider an insurance company where several domain rules and structure, due to the nature of the business, keep changing to fulfill market needs. One example is the insurance payback for any particular kind of incident, which is based on a complex formula that takes into account several factors. Not only the formula changes as the system evolves, but also the factors taken into account change, thus needing new information to be either collected or inferred (e.g. the number of children of an individual while calculating his life insurance payment).

However, even simple evolutions of the structure or behavior, like removal of information, can have a significant impact in the system. Valid objects may depend on the information being changed, thus leading to inconsistency. These issues need to be addressed upon each evolution step, to guarantee that the integrity of the system holds to the specification.

Another typical concern is maintaining legacy interfaces. If the system must inter-operate with third-party components, once the model definition evolves, the interface can become in-

valid. In this case, it may be necessary to provide a layer of data transformation, thus maintaining legacy interfaces over previous versions of the system. This approach may increase the complexity of the underlying architecture.

Problem

How do we support the evolution of a system while maintaining its integrity? Since this pattern uses the HISTORY OF OPERATIONS and SYSTEM MEMENTO patterns, is is also subject to the same forces. Additionally, we should consider: (i) Automation of the evolution, instead of relying on monolithic, custom made scripts, (ii) Integrity, since applying a migration should result in a consistent state of the system, (iii) Control, to restrict the kind of evolutions allowed upon the system, and (iv) Interoperability with third-party systems not aware of the model evolution.

Solution

Use the History of Operations to support the Versioning of Things. Achieving a target Version is the result of applying the sequence of Operations defined between two Versions.

As described in the HISTORY OF OPERATIONS and SYSTEM MEMENTO patterns, first start by decoupling the State of a Thing from its identity (see the IDENTITY pattern [Fow02]). Also, every Operation over a Thing should be structured as a COMMAND [GHJV94]. Instead of operating over Things, operations should occur over (or generate new) States. Considering there is a one-to-one relationship between the History and Version classes (see HISTORY OF OPERATIONS and SYSTEM MEMENTO patterns), the later can fulfill both roles (*cf.* Figure 5.11).

An Operation can be specialized into either Concrete Operations, or Macros that establish a sequenced group of other Operations, through the use of the COMPOSITE pattern [GHJV94].

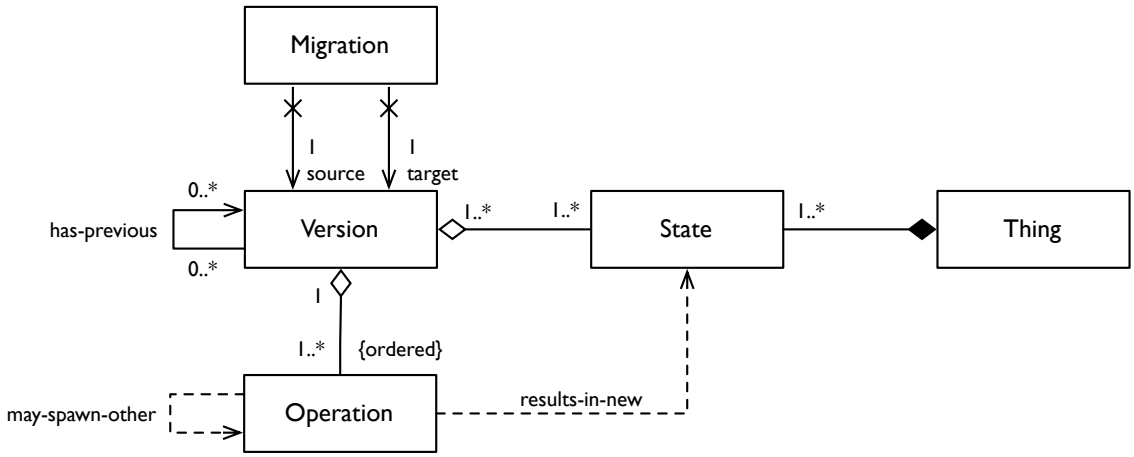


Figure 5.11: Class diagram of the MIGRATION pattern. A Migration between any two Versions consists on applying, in the correct order, all the histories of Operations that were executed between those versions. A Version may not refer more than one State from the same Thing.

The ability of an Operation to spawn other Operations, allows changes on the knowledge-level to be reflected upon the operational-level, whose purpose is to maintain the consistency of the system. For example, a *Move Attribute to Superclass* at the knowledge-level may generate several Operations at the operational-level, since data may also need to be moved.

The MIGRATION acts as an interpreter, or patch engine, which, given a Version and a set of Operations, achieves a target Version.

Example Resolved

One of the most complex examples this pattern support is the ability to evolve what is normally called the schema (in this case, the word *model* is more appropriate) and to immediately affect data at lower levels.

Let us consider the aforementioned example. The introduction of new laws will require the creation of new fields in existing entities (e.g. *number of dependent children*). Others, previously belonging to a particular sub-class, will now be moved into the super-class (e.g. *number of days overseas per year*).

Consider this evolution will occur from version V_1 to version V_2. Two M_1 (knowledge-level) operations are issued: (a) *Create Attribute* and (b) *Move Attribute to Superclass*. While the former doesn't need to spawn any M_0 Operations, the later should be defined as a Macro, mixing sequential M_0 and M_1 operations (e.g. *Create Attribute* at M_1, *Duplicate Data to Attribute* at M_0, *Delete Attribute* at M_1 and *Dispose Data* at M_0). Each Operation will act upon a specific given State of a set of Things to generate new States. This sequence of commands, interweaving different level operations, may be stored in the new Version (V_2).

In summary, *a migration between any two versions consist on applying, in the correct order, all the histories of operations that were executed between those versions.*

5.6.4 Resulting Context

The resulting context of applying this pattern is the combined resulting contexts of the HISTORY OF OPERATIONS and SYSTEM MEMENTO patterns. The following benefits are particular to this pattern:

- We are now able to **automatically** evolve between any two versions of the system, provided that we issue a semantically correct sequence of operations.
- Consistency of the system is dependent on the consistency of the operations. This functional decomposition may help achieving higher confidence in the model **integrity** after a migration procedure.

This pattern has the following liabilities:

- If the system does not provide enough operations to perform complex tasks, it can be difficult (or even impossible) to express the intended semantics of the evolution.
- The migration mechanism, along with all the additional information that it requires, adds **complexity** to the system.

Implementation Notes

Refactorings as Evolution Primitives. In object-oriented programming, behavior-preserving source-to-source transformations are known as refactorings [Fow99]. The concept of refactoring applied to models [CW04, MFJ05, HBJ09] has already been pointed out as a way to cope with system evolution. This notion may be applied when designing Operations, such that they represent a set of refactorings specifically designed for evolving Things. Each refactoring should assure system integrity upon its completion.

Related Patterns

All related patterns to History of Operations and System Memento apply.

Known Uses

Several *Object-oriented Database Management Systems* and *Data Warehouses* [RL04, WE00, BEK+04], as well as the *Prevayler framework* [Ope10], the *COPE tool* [HBJ09], and the *AMOR system* [AKK], are known uses of this pattern.

The *Ruby on Rails (RoR)* framework uses a variation [Und08, Usi08] of Migration, but expresses operations within relational models, since it's based upon the Active Record Pattern [Fow02].

5.7 COMPOSING THE PATTERNS

There is a growing collection of AOM-related patterns currently contributing to the domain pattern language proposed by Welicki *et al.* [WYWBJ07], including those here presented. Originally, the pattern language was divided into six categories, viz. (i) Core, (ii) Creational, (iii) Behavioral, (iv) GUI, (v) Process, and (vi) Instrumental. In this dissertation, after formalizing the seven patterns presented in this chapter, we now propose a new map and categorization depicted in Figure 5.12 (p. 92), divided into eight categories, viz. (i) Structural, which provides the basic object model, (ii) Behavioral, for rules and system dynamics, (iii) Architectural, which defines the infrastructure, (iv) Interaction, focusing on user-computer interfaces, (v) Creational, reporting to patterns that allow specific domains to be instantiated, (vi) Evolution, focusing on supporting changes to the data and meta-data, (vii) Construction, which exposes to the end-user

the (re-)configuration of the system, and (viii) Support, for miscellaneous patterns that address cross-cutting concerns.

5.8 CONCLUSION

In this chapter we have formalized seven new patterns related to systems which domain model needs to be changed frequently during runtime. We have discussed some scenarios were they occur, as well as their advantages and liabilities. These patterns directly contribute to the ongoing effort of building a pattern language for Adaptive Object-Models, to which we have also proposed a new categorization.

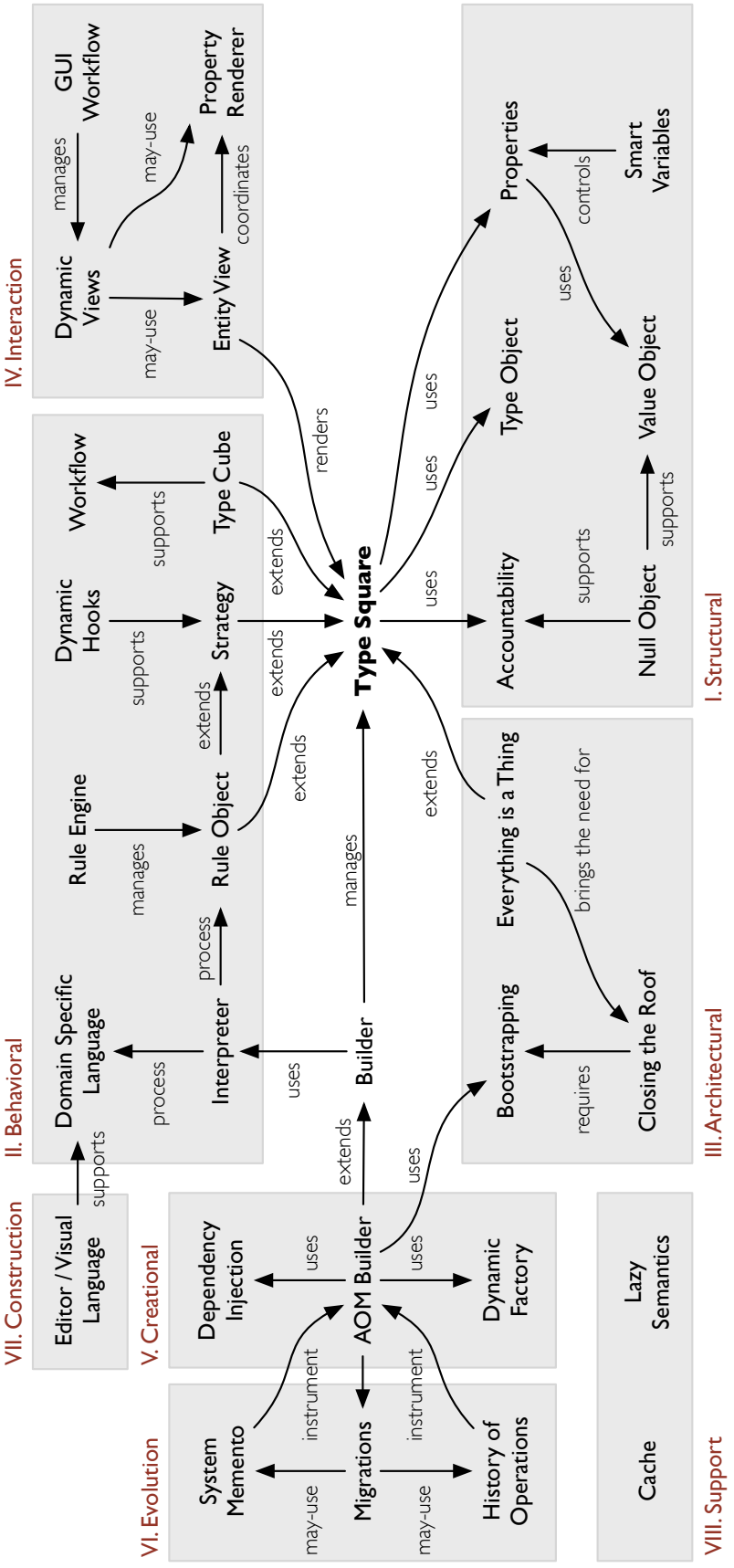


Figure 5.12: Proposed evolution of the pattern language for Adaptive Object-Models.

Chapter 6

Reference Architecture & Implementation

6.1 Overview 93
6.2 Thread of Control 97
6.3 Architectural Decomposition 100
6.4 Crosscutting Concerns 108
6.5 User Interface 112
6.6 Development Effort 113
6.7 Conclusion 114

This chapter focus on Goals 2 and 3 of this thesis, namely **to specify a reference architecture for adaptive object-model frameworks**, and **to provide an industrial-level implementation of such framework**, codenamed *Oghma*. We start by identifying the underlying design principles and guidelines — i.e., the requirements — of *incomplete by design* systems. We then provide the high-level architectural view of such framework, and decompose it into the key components involved. In due course, each component is showed how it addresses the proposed guidelines, providing implementation details where necessary.

6.1 OVERVIEW

Oghma is an object-oriented framework [RJ96, ATMB00, FSJ99b], targeted to the development of information systems whose structural requirements can be best described as *incomplete by design* [GJT07]. Applications making usage of this framework can easily implement the Adaptive Object-Model meta-architectural pattern [YBJ01b] to allow run-time domain evolution by the end-user. It is structured as a collection of layered component library [BMR+96], allowing their high-level composition to achieve different functional architectures, e.g. client-server v.s. single-process. It makes extensive usage of meta-programming § 2.2.1 (p. 20) and

meta-modeling § 2.2.3 (p. 22) following the general guidelines of the NAKED OBJECTS architectural pattern [Paw04] to provide runtime adaptation of the domain model, and automatic generation of graphical user-interfaces and persistency. It extends the common meta-object/meta-class model design [Caz98] by raising object versioning to a first-class concept of the system. It also adds the ability to specify a wide class of behavioral rules, such as derived properties, class invariants, and data views, by using the INTERPRETER pattern [GHJV94], and presenting them to the end-user as a domain specific language, either through a classic textual syntax or graphical representation.

6.1.1 General Principles and Guidelines

Probably the most well known system specifically built to be *incomplete by design* is the *WikiWikiWeb*, created by Ward Cunningham in 1995 [Cun95], and which is nowadays the base[1] for a class of systems commonly called *wikis*. At the core of a wiki lies the set of principles and guidelines shown in Table 6.1.

PRINCIPLE	DESCRIPTION
Simple	Easier to use than abuse. A wiki that reinvents HTML markup ([b]bold[/b], for example) has lost the path!
Open	Should a page be found to be incomplete or poorly organized, any reader can edit it as they see fit.
Incremental	Pages can cite other pages, including pages that have not been written yet.
Organic	The structure and text content of the site are open to editing and evolution.
Mundane	A small number of (irregular) text conventions will provide access to the most useful page markup.
Universal	The mechanisms of editing and organizing are the same as those of writing, so that any writer is automatically an editor and organizer.
Overt	The formatted (and printed) output will suggest the input required to reproduce it.
Unified	Page names will be drawn from a flat space so that no additional context is required to interpret them.
Precise	Pages will be titled with sufficient precision to avoid most name clashes, typically by forming noun phrases.
Tolerant	Interpretable (even if undesirable) behavior is preferred to error messages.
Observable	Activity within the site can be watched and reviewed by any other visitor to the site.
Convergent	Duplication can be discouraged or removed by finding and citing similar or related content.

Table 6.1: Wiki design principles, quoted from [Cun03].

Inspired by the above principles, we argue that any system[2] deliberately designed to cope with *incompleteness* should provide similar guidelines as those present in Table 6.2 (p. 95).

[1] Not strictly the code-base and neither the requirements, but instead the core functionalities.
[2] For the purposes of this dissertation, we will always consider information systems.

Principle	Description
Open	Should a resource be found to be incomplete or poorly organized, the end-user can evolve it as they see fit.
Incremental	Resources can link to other resources, including those who have not yet been brought into existence.
Organic	Structure and content are open to editing and evolution. Evolution may be made more difficult if it is mandatory for information to strictly conform to a pre-established model.
Universal	The same (or very similar) mechanisms for modifying data and model should be exposed by the system with no apparent distinction.
Overt	End-user evolution should be made by non-programmers. The introduction of linguistic constructions (such as textual syntax) is usually required in order to provide formalization. However, such constructions may reveal unnatural, intrusive and complex to the end-user, thus model edition should be made as readily apparent (and transparent) as possible.
Tolerant	Interpretable behavior is preferred to system halt.
Observable	Activity should be exposed and reviewed by end-users, fomenting social collaboration.
Convergent	Duplication is discouraged and removed by incremental restructuring and linking to similar or related content.

Table 6.2: Design principles for *incomplete by design* systems.

6.1.2 High-level Architecture

The high-level architecture of the *Oghma* framework is depicted in Figure 6.1. It allows the development of AOM-based information systems by providing a Component Library and a default Thread of Control. The main components are highly modular, deliberately designed to cope with a range of different needs and deployment scenarios, such as (i) client-server, where several processes are orchestrated by a central service, (ii) single-process, when one can disregard concurrency issues and choose a monolithic setup, (iii) web-based, whereas a single process is running but receiving multiple requests, and (iv) distributed, which takes advantage of automatic data-replication mechanisms provided by underlying persistency engines to propagate changes across multiple services. More details will be given in § 6.1.4 (p. 96).

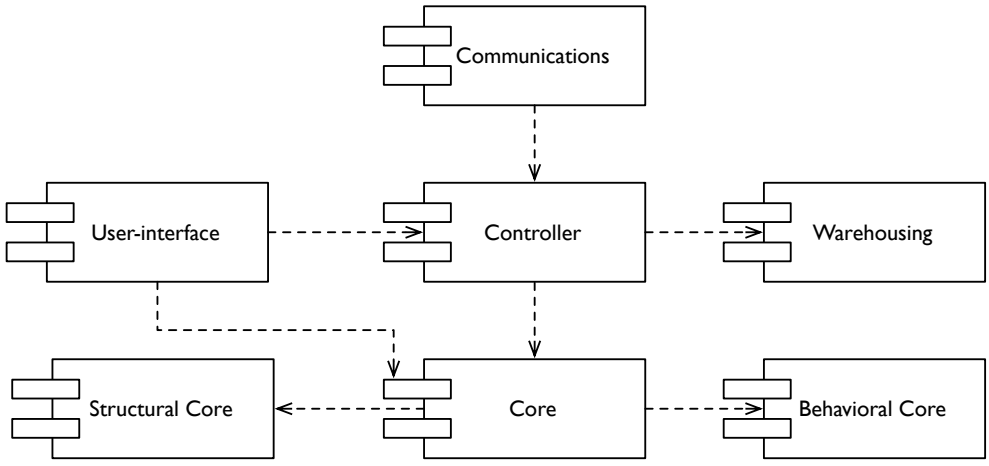


Figure 6.1: High-level architecture of the *Oghma* framework.

Oghma supports a custom object model closely resembling class and object diagrams from UML and MOF, and it is aimed to cover the entire cycle of software development, from design to deployment § 1.3 (p. 4). The introduction of changes to the model, either originating in the development team or in the end-user, can be performed during runtime. Furthermore, the framework leverages the infrastructure to raise the overall awareness of the system's evolution, providing functionalities such as *auditing* and *time-traveling*, in accordance to principles drafted in § 6.1.1 (p. 94).

6.1.3 Key Components

The proposed architecture identifies the following components, partitioned according to the best-practices of object-oriented design, namely *low-coupling* and *high-cohesion*:

- **Core.** This component provides the model object supported by the framework; it can be further divided into the structural and behavioral cores.
- **Structural Core.** This component describes the basic structure of the object model, e.g., the concepts of an object, a class, and a primitive type, and the way they are related.
- **Behavioral Core.** This component addresses the dynamic concerns, e.g., class invariants and derivation rules.
- **Controller.** This component orchestrates the other components in the system, by holding the responsibility of the THREAD OF CONTROL
- **Warehousing.** This component provides mechanisms for storing data and metadata.
- **Communications.** When several applications are running (e.g., in a client-server architecture), a channel must be established between them.
- **User-interface.** Eventually, the end-user must interact with the application through non-programmatic artifacts.

6.1.4 Component Composition

Each component may be specialized into different components, thus providing a modular composable architecture. For example, the architecture of a client-server application, with a rich GUI and persisting data into a relational database is depicted in Figure 6.2 (p. 97). Here, the controller was further specialized into a client and server controller, the former relying in the communications component. The warehousing relies on a persistency module that deals with the relational database.

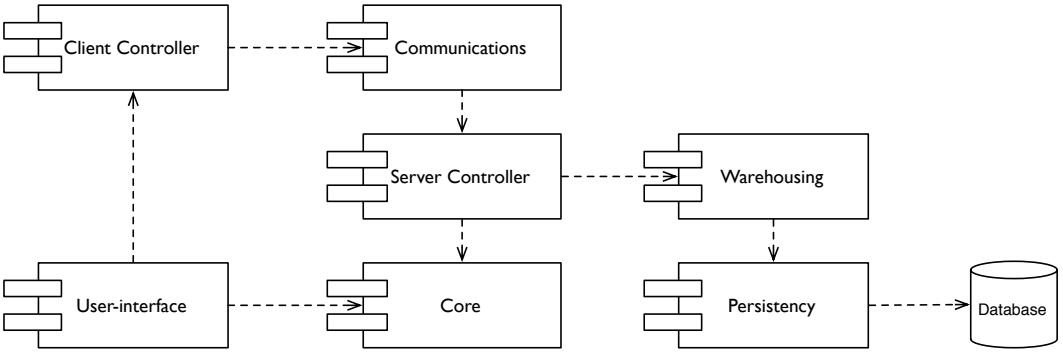


Figure 6.2: Architecture of a client-server setup of the *Oghma* framework.

A different composition may be observed in Figure 6.3, taking advantages of a distributed database to eliminate the communications component. We will call any particular composition of the components the application *stack*, and the way it is achieved will be described in due course.

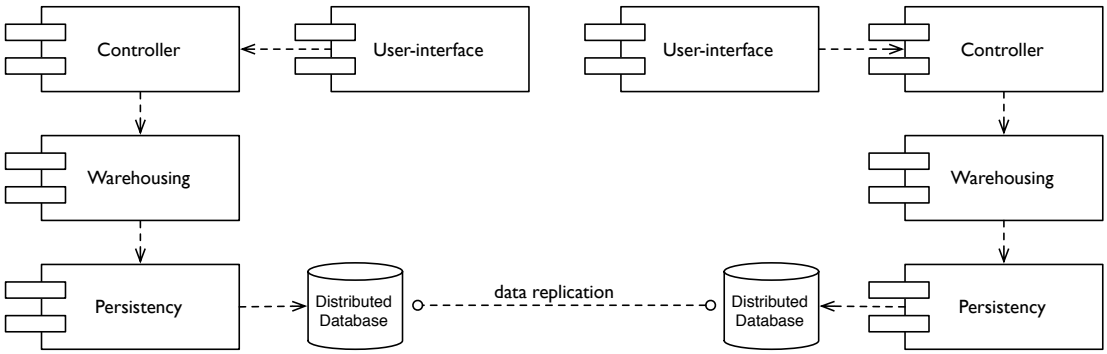


Figure 6.3: Architecture of a distributed setup of the *Oghma* framework.

6.2 THREAD OF CONTROL

As seen in Figure 6.1 (p. 95), the controller serves as an entry point for both the user-interface and communications component, and its key responsibility is to orchestrate the several other components in the framework by establishing a thread of control. This thread of control may be divided into three stages, viz. (i) the initialization process, also known as bootstrapping, (ii) the main process, which receives external events for data querying and manipulation, and interacts with the warehousing, and (iii) the shutdown process.

The controller also holds the responsibility of providing *Hooks* to the framework through the Chains of Responsibility and Plugins patterns, for extensibility reasons (e.g. interoperability with third-party systems by allowing subscribers to intercept requests).

6.2.1 Bootstrapping

At the application startup, the framework has to execute two types of bootstrapping, viz. (i) infrastructure bootstrapping, where the description of the structural core is loaded, and (ii) application bootstrapping, where the first version of the model is loaded. The latter only occurs if it is the first time the application is started, since subsequent startups rely on the latest model information stored in the warehouse. This activity can be observed in Figure 6.4.

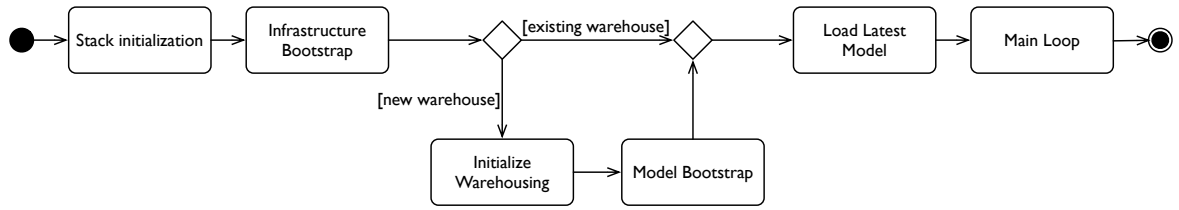


Figure 6.4: Activity diagram of the framework initialization.

The excerpt of code listed in Source 6.1 shows a small part of the infrastructure bootstrapping, where one can observe the structural core being defined, namely the Thing, Named Element, Owned Element and Enum Literal.

```
1  <entity id="metathing" name="MetaThing" abstract="true" tostring="{
       _identity}">
2    <attr id="_identity" name="Identifier" domain="identity" cardinality="1"
       readonly="true" />
3    <attr id="_nativetype" name="Native Type" domain="string" cardinality="
       0..1" />
4  </entity>
5
6  <entity id="namedelement" name="Named Element" abstract="true" inherits="
       metathing" tostring="{_name}">
7    <list columns="{_identity}|{_name}" />
8    <attr id="_name" name="Name" domain="string" cardinality="1" />
9    <attr id="_displayname" name="Display Name" domain="string" cardinality="
       1" />
10 </entity>
11
12 <entity id="ownedelement" name="Owned Element" abstract="true" inherits="
       namedelement" />
13 <entity id="enumliteral" name="Literal" tostring="{_displayname}" inherits=
       "namedelement" isinstanceof="Oghma.Core.Structural.EnumLiteral" />
```

Source 6.1: A small excerpt from the infrastructure bootstrapping.

6.2.2 Main Process

After the bootstrapping, the framework enters the main loop waiting for events to occur. There are two main type of events, viz. (i) query events, which do not modify the current state, and (ii) commit events, which may introduce new data or meta-data. This activity is depicted in Figure 6.5.

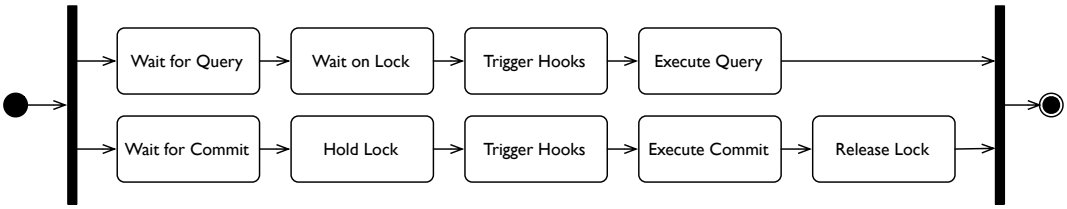


Figure 6.5: Activity diagram of the main loop.

Query and commit events may be received by different threads, but only query events can be processed simultaneously. In order to ensure consistency, particularly due to modified causal connections when model changes occur, commit events lock the application. This is a design decision that may be subject to change, since it may introduce *performance* issues in write-intensive, highly concurrent systems. Both query and commit events trigger before, during, and after Hooks for extension purposes.

6.2.3 Query Events

The majority of events during a normal session are queries, which include: (i) requesting a list of instances based on the type, (ii) requesting a list of instances based on conditionals, (iii) requesting a particular instance based on its identifier. Due to the non-differentiation between data and meta-data, some possible events such as (iv) requesting a list of types based on the meta-type, are variants of requesting instances. Other query events may include versioning concerns, such as (v) requesting the history of transactions, or (vi) requesting instances in a given (previous) version. This activity can be observed in Figure 6.6.

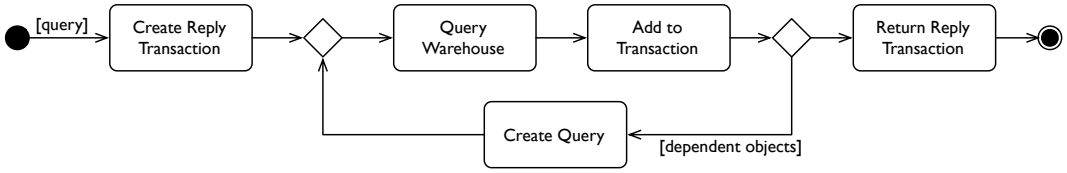


Figure 6.6: Activity diagram of query events.

6.2.4 Commit Events

Commit events modify the current state of the application, by creating or updating its data and metadata, as seen in Figure 6.7. It includes (i) creating a new instance, (ii) updating an attribute, (iii) establishing/deleting a link, (iv) deleting an instance. Again, due to the non-differentiation between data and meta-data, some possible events such as (v) creating a new type, are variants of those over instances. This activity is slightly more complex than the query activity; first, there are two ways of performing a commit event: (i) state-based, where one provides the final state of the object(s), and (ii) operation based, where it is sequence of operations performed upon an object(s) that are provided. If the controller receives the former, then it infers a possible sequence of operations, by using semantic heuristics. Second, before applying the operations, a merged container is created that provides a safe view over the existing data; operations become pure, i.e., they generate new objects. In the end, all validation rules (e.g., class invariants) are verified, and side-effects due to triggered rules, e.g., auto-numbers, are replied in a new transaction.

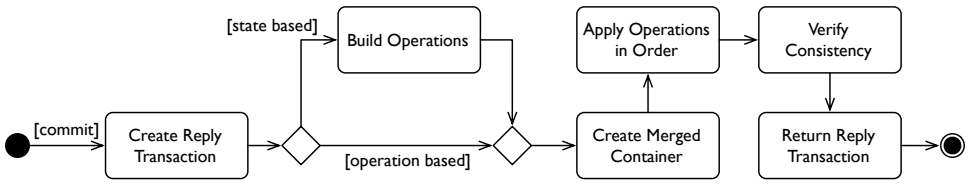


Figure 6.7: Activity diagram of commit events.

6.3 ARCHITECTURAL DECOMPOSITION

We now proceed to decompose the high-level architecture into components, and each component into a conceptual structure (design).

6.3.1 Structural Core

The design of the structural core of *Oghma*, built on top of the TypeSquare § 3.2.3 (p. 42) pattern, can be observed in Figure 6.8 (p. 101). The ObjectType is refined into Entities (that represent classes) and Interfaces. An Instance, which must comply to a given Entity, holds a collection of Properties, which in turn complies to its PropertyType. PropertyTypes are refined into RelationNodes and Attribute-Types. A Relation-Node specifies cardinality, navigability and role, by connecting to another node. Two connected nodes establish a RelationType. For properties belonging to relations (similar to the *Associative Class* in UML), the RelationType may refer another Entity. Object-oriented inheritance and polymorphism is established by linking Entities with other Entities and/or multiple Interfaces.

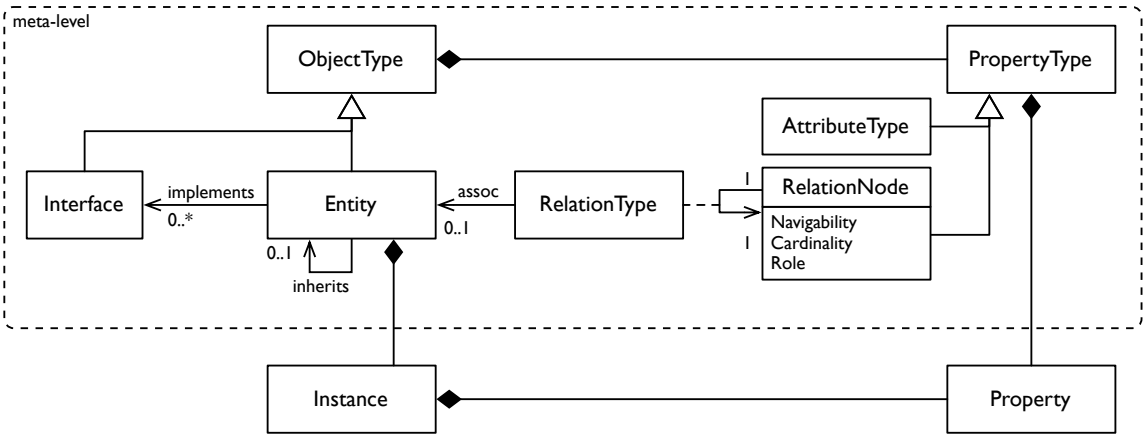


Figure 6.8: Core design of the structural meta-model.

Extending the Property Pattern

The usual form to capture relationships between different objects is by using the ACCOUNTABILITY pattern, as previously discussed in Chapter 3 (p. 35). But we should ask what exactly is the difference between a field and a relation? Object fields, in OOP, are used to store either values of native types (such as an int or a float in Java) or references to other objects. They can also be either scalars or collections. Some pure OO languages (e.g., Smalltalk) treat everything as an object, and as such do not make any difference from native types to references[3]. Some also discard scalar values and instead use singleton sets. We may borrow these notions to extend the PROPERTY pattern in order to support associations between entities, as seen in Figure 6.9, provided we are able to state properties such as cardinality, navigability, role, etc. The actual difference between what is a scalar property and a relation becomes a runtime detail resolution.

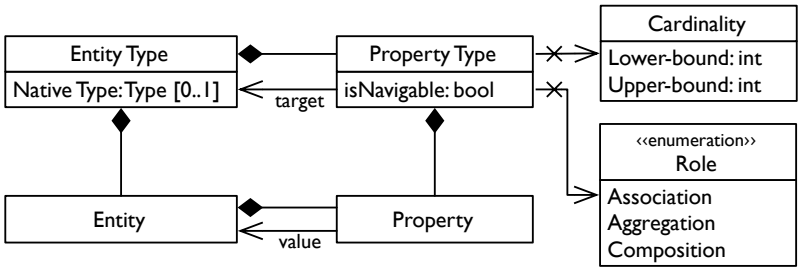


Figure 6.9: An extension of the TYPE-SQUARE pattern.

One HOOK introduced between the framework and the host language is the use of the Native Type property in Entity Type, to allow any custom Entity Type to be directly mapped into a native type (such as integers and strings). There are also several constraints not depicted in the diagram. For example, the lower and upper bound in cardinality should restrain the number of associations from a single Property to Entities. Likewise, Properties should only link to

[3] Another blatant simplification, since the smalltalk virtual machine do treat primitive types differently; they are just exposed as if they were another object.

Entities which are of the same Entity Type as that defined in Property Type. The complete formalization of the semantics of the presented models is outside of the scope of this dissertation, but part of it may be found in the test suits of the framework.

Self-Compliance

The aforementioned description represents a very simplified[4] view of the entire set of implementation concerns. The bootstrapping of the infrastructure requires that the meta-model be defined in terms of itself, due to reasons previously discussed in § 5.5.3 (p. 72). MOF is an example of self-compliance, by making the M_3 layer self-describing. There are several reasons to make that design choice: (i) it makes a strict meta-modeling architecture, since every model element on every level is strictly in correspondence with a model element of above level; and (ii) the same mechanisms used to view, modify and evolve data can be reused for meta-data.

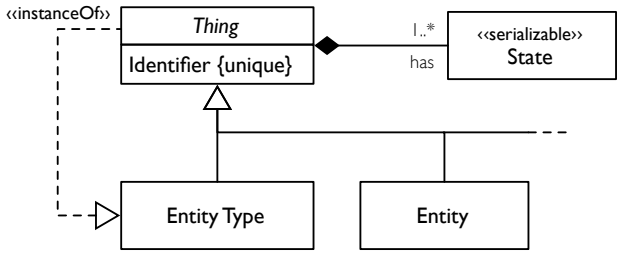


Figure 6.10: Implementing the EVERYTHING IS A THING pattern, and decoupling the IDENTITY of an object from its state.

One way to accomplish self-compliance relies on using the EVERYTHING IS A THING pattern with CLOSING THE ROOF. Basically, it consists on abstracting the elements of every level into a single, unifying concept; the Thing. A Thing is thus an instance of another Thing, including of itself, such as depicted in Figure 6.10. This implies that during bootstrapping, the system would first need to load its own meta-model, as discussed in § 6.2.1 (p. 98). Therefore, Entity Types are actually M_2 Entities, thus relying on context for proper semantics.

Versioning

As seen in Figure 6.10, the identity of each instance is maintained as Things, while the respective details are kept as States. Therefore, all the previous States of a Thing become reachable to the application, as seen in Figure 6.11 (p. 103), where two distinct values exist for the same Property Type, corresponding to two different changes to the same instance over time. This design choice allows to leverage of the *Observable* guideline § 6.1.1 (p. 94), e.g., to provide auditability and time-traveling mechanisms.

[4] Almost *naïve*.

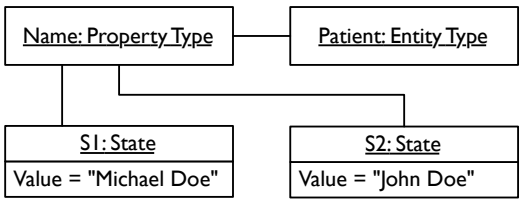


Figure 6.11: An example of two different states of the same entity.

Implementation Model

By also taking into consideration versioning concerns, a design more close to the implementation may be observed in Figure 6.12.

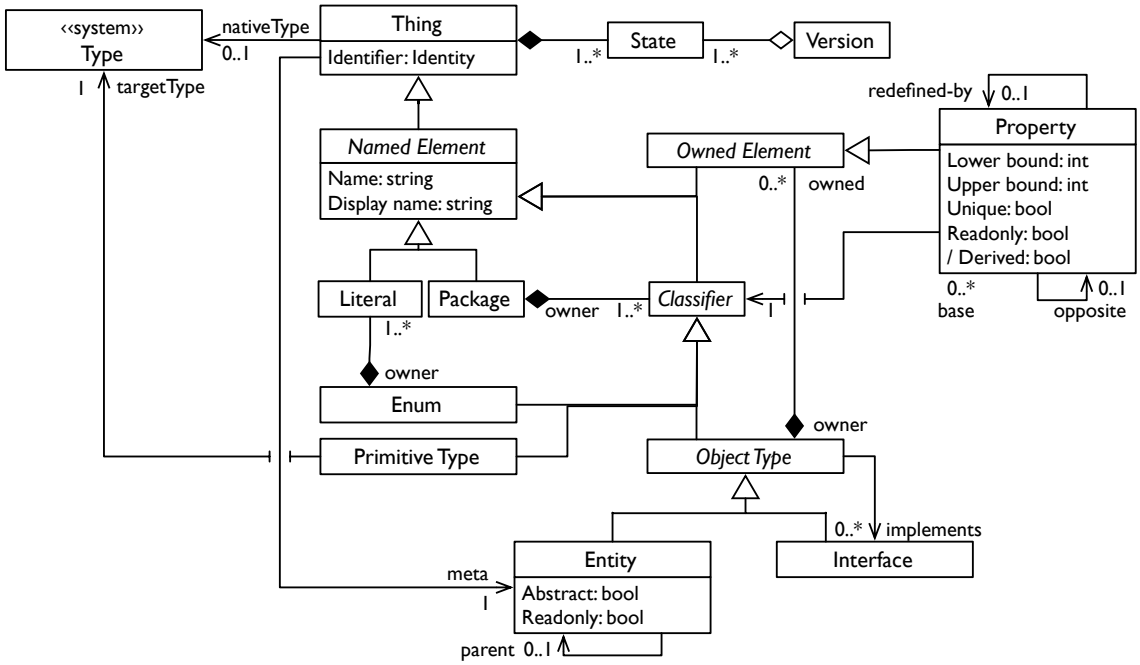


Figure 6.12: Implementation model of the structural meta-model.

6.3.2 Behavioral Core

In addition to structure, such as entities, properties and associations, a system possesses dynamics; its data and metadata evolves[5] over time. It also relies on the ability to support rules and automatic behavior. Some examples of these include, but are not limited to (i) constraints, such as relationship cardinality, navigability, type redefinition, default values, pre-conditions, etc. (ii) functional rules, which include reactive logic such as triggers, events and actions, and (iii) workflow logic.

[5] Adding data to the system should also be considered evolution.

Strategy and Rule Objects

Even during the bootstrapping of the infrastructure, after all types of objects and their respective attributes are created, there are some rules that must be considered. For example, the cardinality of an association should be preserved; this is the same as saying that the target collection's size should be between the lower and upper bounds defined by its type. To support such common rules, relatively immutable or otherwise parameterized, one could resort to using the STRATEGY and RULE OBJECT pattern, as previously discussed in § 3.2.6 (p. 43) and seen in Figure 6.13.

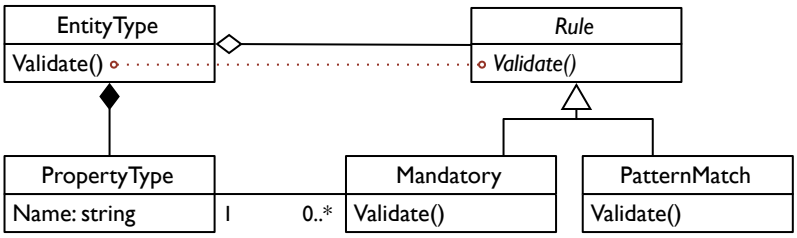


Figure 6.13: A class diagram of the STRATEGY and RULE OBJECT patterns, forming the structure of the family of algorithms for enforcing rules. The dashed line represents an invocation call.

An extension of the RULE OBJECT is depicted in Figure 6.14, which allows the definition of: (i) Entity Type invariants, which are predicates that must always hold, (b) derivation rules for Property Types and Views, (c) the body of Methods, (d) guard-conditions of Operations, etc. The structural constraints above mentioned, such as the cardinality and uniqueness of a Property Type, are specializations of conditions. Methods may be invoked manually and/or triggered by events, thus providing enough expressivity to specify STATE MACHINES and WORKFLOWS. Patterns such as SMART VARIABLES and TABLE LOOKUP also provide an example of common rules implemented as strategies [YFRT98].

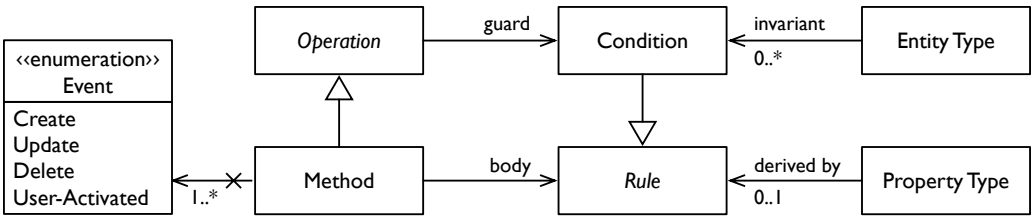


Figure 6.14: Dynamic core of the *Oghma* framework.

Interpreter

When the desired behavior reaches a certain level of complexity and dynamism, then a domain specific language should be considered to extend RULE OBJECTS. After parsing, the language can be converted into an AST and further processed using an INTERPRETER [GHJV94], where primitive rules are defined and composed together with logical objects mimicking the tree structure to

be interpreted during runtime, or by a VIRTUAL MACHINE. They are widely used to define: (i) ObjectType invariants, (ii) derivation rules in PropertyTypes and Views, (iii) body of Methods, (iv) guard-conditions of Operations, etc. As such, they also play an important role in assuring semantic integrity during model evolution, as will be discussed in § 6.4.3 (p. 110). Source 6.2 shows a model excerpt stating a behavioral rule ensuring that a person cannot be its own parent, using the custom-developed DSL.

```
<entity id='person' name='Person'>
  <inv name='notmyancestor'>not self.In(Walk(pai))</inv>
  <attr id='parent' name='Parent' domain='person' cardinality='0..1' />
</entity>
```

Source 6.2: Example of a model defining behavioral rules.

A common problem that arises with the abstraction of rules, is that the developer may fall in the trap of creating (and then maintaining) a general-purpose programming language, which may result in increased complexity regarding the implementation and maintenance of the target application, far beyond what would be expected if those rules were hardcoded [YBJ01a].

Views

Consider the following requirement: "list all doctors in a particular medical center, along with the number of high-risk procedures they have performed and its total cost." In the design presented so far, there would be no mechanism to support such query. We thus must define the concept of a *view*, as an Entity Type having a derivation rule that returns a collection, and whose properties also depend on derivation rules (often manipulating each item of that collection). This allows the existence of "virtual" entities, whose information is not stored, but automatically derived[6].

The requirement could then be fulfilled by a new type of entity, e.g. "High-Risk Treatments Income by Doctor", that iterates over doctors and their procedures, aggregating the income and so on.

6.3.3 Warehousing and Persistency

Because of the evolutive nature of the model, it is very complex and inefficient to map and maintain data and meta-data in a classic relational database, even with the help of common technologies such as Object-Relational Mapping (ORM), or other techniques such as model-to-model transformations. An alternative would be to make use of object-oriented database management systems (OODBMS).

[6] Akin to the querying mechanisms (SQL) present in relational databases.

Still, persistency based on static-scheming, such as automatic generation of DDL code for specifying relational databases schema and subsequent DML code for manipulating data, which attempts a semantic correspondence between both models, significantly increases the implementation complexity, particularly when dealing with model co-evolution. This has been long referred to as object-relational *impedance mismatch* [Amb03], and evidence of such issues may be observed in the way ORM frameworks attempt to deal with them, often requiring knowledge of both representations and manual specification of their correspondence (e.g., the migration mechanism in RoR). Therefore, using the filesystem for storing serialized objects, or maintaining objects BLOBS in key-valued or even relational databases, without trying to achieve a semantic correspondence, has revealed to be a better choice.

Warehouse Design

Despite all the above concerns, warehousing encapsulates the details of maintaining objects and persisting its information, from the rest of the system, by exposing and consuming data and meta-data (i.e., Things) and managing versioning (i.e., through Versions and States). Its most abstract concept is that of a Container, a collection of possibly interrelated objects, as depicted in Figure 6.15.

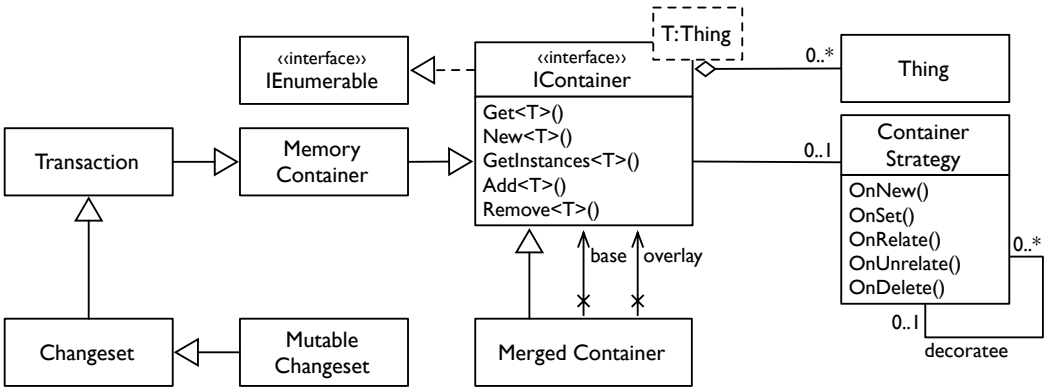


Figure 6.15: Class diagram of the warehousing design.

Its behavior can be extended and modified through inheritance and composition by using the DECORATOR, CHAIN OF RESPONSIBILITY and STRATEGY patterns [GHJV94], with Things regarded as opaque, key-valued objects. Transient memory-only, direct data-base access, lazy and journaling strategies (e.g., using CACHES) are just a few examples of existing (and sometimes simultaneous) configurations.

Persistency Design

Storing and fetching information from an external datasource, such as a database, is accomplished by specializing a container (typically MemoryContainer) and interfacing with the per-

sistency components mainly through the IProvider interface, as sketched in Figure 6.16. By further specialization, GreedyProvider and LazyProvider implement different access strategies, respectively: (i) by fetching all objects to memory, and doing a *deep-write*[7] whenever a commit occurs; (ii) by keeping a CACHE of latest requested/modified objects. Further details are given in § 6.4.1 (p. 108).

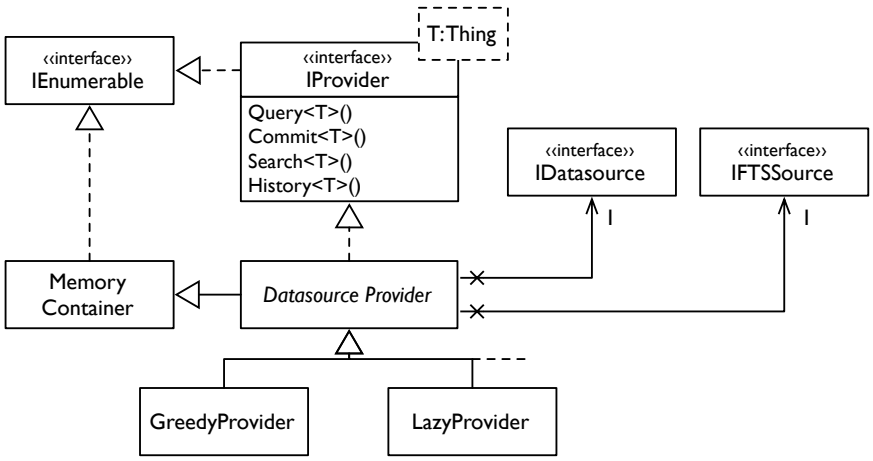


Figure 6.16: Class diagram of the persistency design.

The Datasource Provider requires an implementation of IDatasource and IFTSSource. The first one will ultimately be responsible for connecting to a third-party system. Currently, *Oghma* provides four different datasources, viz. (i) SQLite, (ii) Oracle, and (iii) MongoDB. Extending it to other technologies is just a matter of appropriately reimplementing the IDatasource interface. The second requirement is a full text-search provider (FTS). Currently *Oghma* only has an implementation over FTS3 provided by SQLite (since it can be used independently), but some preliminary tests have been made with Apache Lucene.

6.3.4 Communications

When components are concurrently running, a channel must be established between them to exchange all the necessary information. Due to the modularity of the presented architecture, it is possible to achieve different configurations. Those involving concurrent requests are (i) client-server, where several processes are orchestrated by a central service, (ii) web-based, whereas a single process is running but receiving multiple requests, and (iii) distributed, where data-replication mechanisms provided by underlying persistency engines are used to propagate changes across multiple services. For more information see § 6.1.2 (p. 95) and § 6.4.1 (p. 108).

[7] A deep-write is a process that ensures information is immediately stored, disregarding caching mechanisms.

Main Concerns

Establishing a channel of communication across different services or components, raises a number of practical concerns that may affect the architecture and design of our system. Mainly, we should be worried with (i) minimization bandwidth, so that we only communicate what is needed, (ii) maximization of performance, (iii) maintenance of consistency, so that the system does not suddenly become incorrect, and (iv) supporting concurrency, so that multiple services or components can interact simultaneously.

HTTP-based Communications

Configuring *Oghma* for client-server or web-based architectures will trigger a HTTP-based strategy of communication, where the current implementation provides a REST API for communication between the Server Controller and Client Controller through a pair of HTTP Bridge and Server Dispatcher acting as PROXIES [GHJV94]. Every Thing is addressable by its unique identifier as a resource. The contents of States and Changesets are serialized in XML. Simple queries can be expressed directly in the URL; those more elaborate require POST methods.

By specialization of the communication layer, other types of technology can be used for bridging the controllers (e.g .NET Remoting). For example, in the case of a single-user stack, the Client Controller would interact directly with the Server Controller. A different approach from the client-server architecture is to use distributed key-value databases (e.g. CouchDB), to handle both persistency and communication. Here, every application would assume direct access to the controller, delegating the responsibility of disseminating contents to the underlying data warehouse.

6.4 CROSSCUTTING CONCERNS

The following sections refer to concerns that are not particular to any component, but instead are pervasive in the overall solution.

6.4.1 Serialization

To support the exchange of information, objects (i.e., data and meta-data) need to be "dematerialized", propagated, and again "materialized" across different *environments*. The same applies for requests and replies. This means that they must be *serializable*, i.e., the information must be convertible into an encapsulated format that can be stored (or transmitted) and later recovered in the same or another computer environment. This is also required for persistency, as discussed in § 116 (p. 106).

```
<oghma version='127'>
  <data>
    <house href='5b6d2926-4c86-47c8-b1ec-a9db737b94e3'>
      <name>Home sweet home</name>
      <street>
        <street href='46a49183-50aa-4f4f-9437-c3d1b66b4afb' assoc='45f32dce
            -5438-4b75-8dce-aabfb03da19b' />
      </street>
    </house>
    <street href='46a49183-50aa-4f4f-9437-c3d1b66b4afb'>
      <name>Piccadely Avenue</name>
    </street>
    <streetnumber href='45f32dce-5438-4b75-8dce-aabfb03da19b'>
      <number>35</number>
    </streetnumber>
  </data>
</oghma>
```

Source 6.3: Example of a serialized state-based commit.

The proposed architecture deals with two different ways of achieving this, viz. (i) state-based, and (ii) operation-based serialization. Using state-based, the object is serialized *declaratively*, i.e., the value of each property is described *as is*. An example of a possible serialization in XML using this strategy is shown in Source 6.3.

In operation-based serialization, objects are described *constructively*, i.e., it is given the necessary steps to reconstruct the intended state, based upon primitive *operations*. The equivalent of Source 6.3 can be seen in Source 6.4

```
<oghma version='127'>
 <ops>
   <create id="5b6d2926-4c86-47c8-b1ec-a9db737b94e3" type="house" />
   <update id="5b6d2926-4c86-47c8-b1ec-a9db737b94e3" attr="house::name">
       Home sweet home</update>
   <create id="46a49183-50aa-4f4f-9437-c3d1b66b4afb" type="street" />
   <update id="46a49183-50aa-4f4f-9437-c3d1b66b4afb" attr="street::name">
       Piccadely Avenue</update>
   <create id="45f32dce-5438-4b75-8dce-aabfb03da19b" type="street" />
   <update id="45f32dce-5438-4b75-8dce-aabfb03da19b" attr="street::number">
       35</update>
   <relate id="5b6d2926-4c86-47c8-b1ec-a9db737b94e3" node="house::street"
       target="46a49183-50aa-4f4f-9437-c3d1b66b4afb" assoc="45f32dce-5438-4
       b75-8dce-aabfb03da19b" />
 </ops>
</oghma>
```

Source 6.4: Example of a serialized operation-based commit.

This strategy requires a set of primitive operators to be known and precisely defined by all

involved parties. Figure 6.17 shows the most basic operations here considered, viz. (i) create instance, (ii) update attribute, (iii) create relation, (iv) remove relation and (v) delete instance. Higher-level operations (i.e., those that manipulate meta-data) are variants of these, and will be discussed in § 6.4.3.

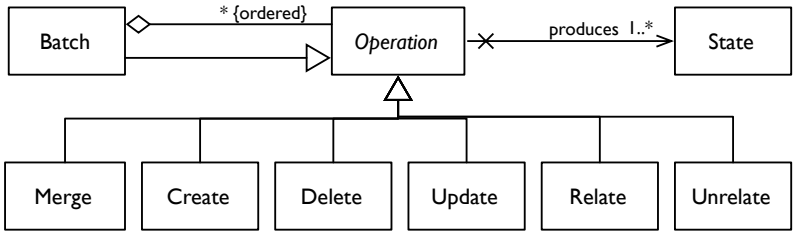


Figure 6.17: Data and meta-data are manipulated through operations, similarly to the COMMAND pattern. Each operations applies (or creates) Things to produce new States.

Object serialization is also used for bootstrapping the system, since it is easier to maintain both the infrastructure description and the first version of the model in a XML file. For more information, see § 6.2.1 (p. 98).

6.4.2 Integrity

The tolerant guideline states that *interpretable behavior is preferred to system halt* § 6.1.1 (p. 94). However, due to causal connections between the meta-data and data, the system must ensure that a correct interpretation is possible, and if not, to allow corrections to be provided. Integrity can thus be divided into two separate concerns, viz. (i) structural integrity, discussed in § 134 (p. 111), and (ii) semantic integrity, detailed in § 134 (p. 111).

6.4.3 (Co-)Evolution

Allowing collaborative co-evolution of model and data by the end-user introduces a new set of concerns not usually found in classic systems. They are (a) how to preserve model and data integrity, (b) how to reproduce previously introduced changes, (c) how to access the state of the system at any arbitrary point in the past, and (d) how to allow concurrent changes. These concerns can be found in software quality attributes such as traceability, reproducibility, auditability, disagreement and safety, and are commonly found on version-control systems.

Typically, *evolution* is understood as the introduction of changes to the model. Yet, the presented design does not establish a difference between changing data or meta-data; both are regarded as the evolution of Things, expressed as Operations over States, and performed by the same underlying mechanisms as previously discussed in § 6.4.1 (p. 108). To provide enough expressivity such that semantic integrity can be preserved during co-evolution, model-level Batches operate simultaneously over data and meta-data.

Sequences of Operations are encapsulated as ChangeSets, following the HISTORY OF OPERATIONS pattern § 5.6.1 (p. 79), along with meta-information such as user, date and time, base version, textual observations, and data hashes. Whenever the framework validates or commits a ChangeSet, the controller uses the merge mechanism depicted in Figure 6.18, inspired by the SYSTEM MEMENTO pattern § 5.6.2 (p. 83). This dynamically overlays the modifications onto the base version by orderly applying each Operation, allowing for behavioral rules to be evaluated, and finally resulting in a new version.

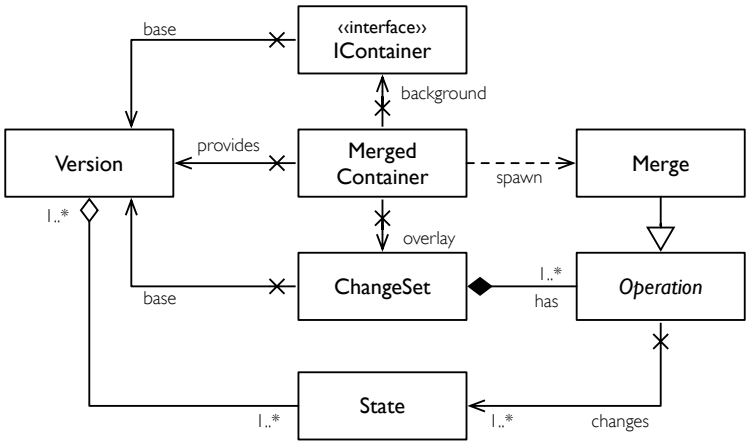


Figure 6.18: Merging mechanism used to validate and apply operations, which are stored in a Changeset.

Structural Integrity

The structural integrity of the system is asserted through rules stated in the meta-model. For example, Instances should conform to their specified Entity (e.g. they should only hold Properties which PropertyType belong to its Entity). Nonetheless, evolving the model may corrupt structural integrity, such as when moving a mandatory PropertyType to its superclass (e.g. if it doesn't have a default value, it can render some Instances non-compliant). Some model evolutions can be solved by foreseeing integrity violations and applying prior steps to avoid them (e.g. one could first introduce a default value before moving the PropertyType to its superclass).

Another issue arrises when parts of a composite evolution violate model integrity, although the global result would be valid. For example if a PropertyType is mandatory, one cannot delete its Properties without deleting itself and vice-versa. This problem is solved by the use of TRANSACTIONS or Changesets, and by suspending integrity check until the end.

Semantic Integrity

Semantic integrity, on the other hand, is much harder to ensure since it is not encoded as rules. The system simply cannot look to the results of an arbitrary evolution and infer the steps that lead

to it. Consider the scenario depicted in Figure 6.19, where an AttributeType age is renamed to date-of-birth, recalculated according to the current date, and moved to its superclass Person.

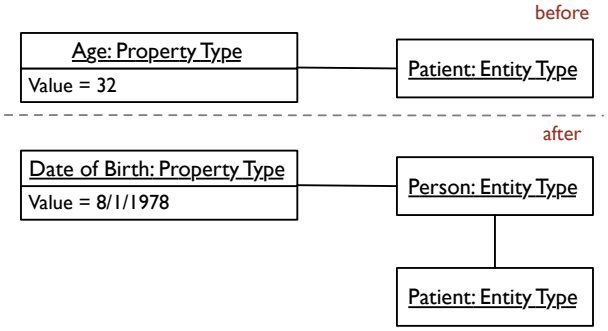


Figure 6.19: Example of a simple model evolution.

Would we rely on the direct comparison of the initial and final models, a possible solution would be to delete the attribute age in Employee and create the attribute date-of-birth in Person. However, the original meaning of the intended evolution (e.g. that we wanted to store birth-dates instead of ages) would be missed. To solve this problem, *Oghma* makes use of MIGRATIONS § 5.6.3 (p. 87), providing and storing sequences of model-level operations that cascade into instance-level changes.

End-user Evolution

If all changes to the data and model are preserved, one can easily recover past information. This not only solves the aforementioned issues, but it also brings to information systems the same notions of versioning inherent to collaboration in *wikis* and software development. In order for the end-user to perform arbitrary model evolutions, a sufficiently large library of (composable) operations should be provided. This also opens way to solve concurrent changes to the model, by allowing the existence of multiple branches of evolution, and provide disagreement and reconciliation mechanisms.

6.5 USER INTERFACE

Adaptive object-models require the graphical user-interfaces to also automatically adapt, through inspection and interpretation of the model, and by using a set of predefined heuristics and patterns [WYWB07]. A minimalistic workflow for an automated GUI can be used based on: (i) a set of grouped entry-points declared in the model, and further presented to the user grouped by packages, (ii) a list of the instances by Entity Type or View, which show several details in distinct columns, inferred from special annotations made in the model, (iii) pre-defined automated views inferred by model inspection (edition and visualization) based on heuristics that

consider the cardinality, navigability and role of properties, (iv) generic search mechanisms, (v) generic undo and redo mechanisms, (vi) support of custom panels for special types (e.g., dates) or model-chunks (e.g., user administration), through PLUGINS, etc.

The reactive user interface provided by *Oghma* also resembles a type of *offline mode*, similar to using version-control systems. User changes, instead of being immediately applied, are stored into the user Changeset, and sent to the main controller to subsequently assert the resulting integrity of applying changes, and provide feedback on behavioral rules. The user can *commit* its work to the system when she wants to save it, review the list of operations she has made, and additionally submit a descriptive text about her work.

Awareness of the system's evolution is achieved through the usage of several feedback techniques such as (i) charts showing the history of changes, (ii) alerts for simultaneous pendent changes in the same subjects from other users, (iii) reconciliation wizards whenever conflicts are detected due to concurrent changes, etc.

6.6 DEVELOPMENT EFFORT

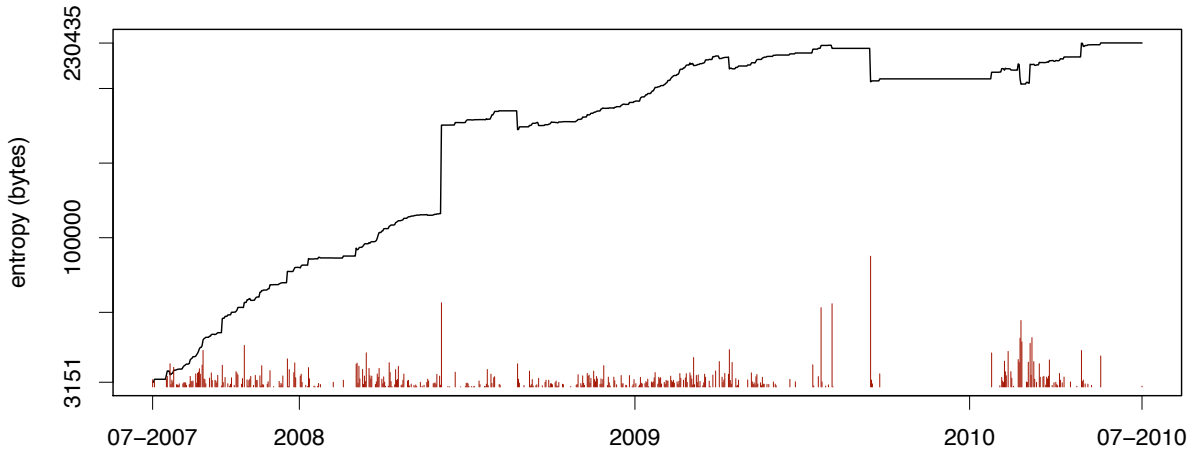


Figure 6.20: Oghma code-base complexity. This chart shows the evolution of code complexity between July 2007 and July 2010, measured in compressed bytes. The vertical lines shows the cumulative size in bytes of the compressed differences per day.

Oghma has been under development since July 2007 as seen in Figure 6.20, and is being used for production-level applications, which will be detailed in Chapter 7 (p. 115). The figure shows the evolution measured in *entropy*, of a codebase with an average of three contributors. Table 6.3 (p. 114) shows some base metrics of the code-base, which allows to draft some learned lessons.

Subjectively, the majority of the development effort was perceived as centered around the structural and behavioral core, and main controller. But, objectively, it is the domain specific language and user-interface that have the largest code-base measured in LOC. This somewhat

Package	Cyclometric Complexity	Real LOC
User Interface	2743	14052
Core	1742	11034
Core.Behavioral	223	867
Core.Behavioral.Grammar	159	5168
Core.Structural	498	1923
Core.Views	211	781
Core.Controller	168	573
Core.Warehousing	151	447
Core.Communications	195	854
Web Server	142	953
SQLite Datasource	28	351
Oracle Datasource	24	325
Tests	114	2179

Table 6.3: Code metrics for the current implementation of *Oghma*.

explains the fact that most of the advanced features supported by the infrastructure, such as divergence and reconciliation mechanisms by object-versioning, are not yet presented in the user-interface.

6.7 CONCLUSION

In this chapter we specified a reference architecture for adaptive object-model frameworks, and provided some implementation details on *Oghma*, an industrial-level implementation of a framework to build AOM-based systems. We have identified the underlying design principles of *incomplete by design* systems by taking inspiration from the guidelines of *wikis*. An high-level architectural view of *Oghma* was followed, and further decomposed into its key components. Then, each component's design was detailed, showing how its role in the whole system is orchestrated. Finally, a general overview of the development of *Oghma* was provided, leaving the details of its usage for Chapter 7 (p. 115).

Chapter 7

Industrial Case-Studies

7.1 Research Design 115
7.2 Complexity Analysis 116
7.3 Baseline 117
7.4 Locvs 117
7.5 Zephyr 121
7.6 Survey 122
7.7 Lessons Learned 125
7.8 Validation Threats 126
7.9 Conclusion 127

The architecture and framework proposed in this dissertation was developed and validated with the help of two case studies and one experiment that evaluated how the results work in practice. This case studies were mainly used for validating the framework during its development phase, and they provided a tight feedback mechanism for new requirements. The (quasi-)experiment was subsequently performed to harness some validation threats inherent to the use of case-studies in Chapter 8 (p. 129). This chapter presents how the case studies were run to assess the usefulness of the primary outcomes, and it ends with the analysis of a questionnaire handed to participants to evaluate their professional opinion on the framework.

7.1 RESEARCH DESIGN

The research conducted so far is as follows: (i) empirical observation is used to detect, acquire and capture requirements and patterns, as described in Chapter 5 (p. 59); (ii) the resulting requirements and patterns help in deriving theoretical models to be incrementally refined into specification and design artifacts, (iii) the combination of these artifacts drive the implementation of a framework, thus producing a set of reusable components, detailed in Chapter 6 (p. 93),

and further serving as basis to (iv) enterprise-level commercial use-cases, which in turn provide feedback for tuning the framework regarding HOT SPOTS, the COMPONENT LIBRARY, and other infrastructure capabilities. This chapter is then focused on those use-cases to assess the impact in productivity, maintainability, developer customization and overall customer success of the conducted research.

Two use-cases were studied, viz. (i) *Locvs*, a medium-sized information system for architectural and archeological heritage and (ii) *Zephyr*, a small information system for document records management.

7.2 COMPLEXITY ANALYSIS

The assessment of the impact on the use cases will be based on complexity analysis of the resulting artifacts. In algorithmic information theory, the Kolmogorov complexity of an object, such as a piece of text or code, is a measure of the computational resources needed to specify that same object. More formally, the complexity of a string is the length of the string's shortest description in some fixed universal description language. It can be shown that the Kolmogorov complexity of any string cannot be more than a few bytes larger than the length of the string itself. Strings whose Kolmogorov complexity is small relative to the string's size are not considered to be complex [Wik10b, LV08].

In order to approximate the upper bounds of the Kolmogorov complexity $K(s)$, of a string s, one can simply compress that string with some method, implement the corresponding decompressor in the chosen language, concatenate the decompressor to the compressed string, and measure the resulting string's length. For this purpose, the author used the bzip2 [Ope08] application, setting its compression level to the supported maximum (-9). If one is only interested in a time-series analysis of a sequence of strings, calculating the compressed *deltas* is enough to present a relative measure of complexity (or entropy) between any two strings [CV05]. For that, the author used the diff [GNU10] unix tool, with the –minimal parameter, again followed by a bzip2 compression, with the aforementioned settings. The results are shown representing the cumulative compressed size in bytes, and the vertical lines the cumulative size of the compressed differences per day. Note that it is the relative, not the absolute number of bytes, that is relevant, i.e., the shape of the curve and the overall growth.

Using an entropy measurement to assess the complexity evolution of a software artifact has some benefits over measuring the number of lines of code (LOC) or size in bytes. For example, the impact of superfluous usage of blank space, long strings, or duplicated code is considerably attenuated after undergoing compression, as observed in Figure 7.4 (p. 120) when compared to Figure 7.3 (p. 119). This technique also seems to be able to show fluctuations in entropy even when the cumulative number of LOC does not change.

7.3 BASELINE

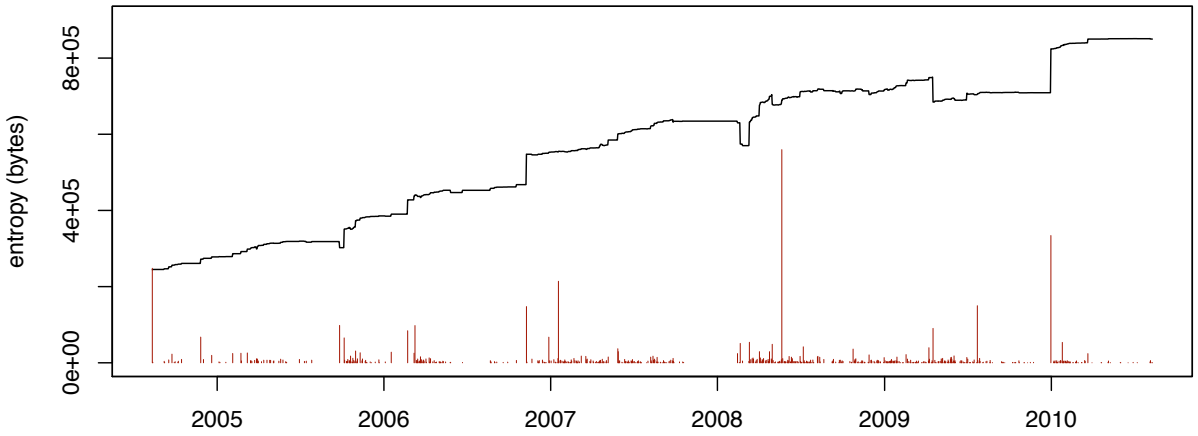


Figure 7.1: This chart shows the GISA evolution between 2004 and 2010 (≃ 6 years).

To establish the baseline for comparison, other projects that met the following criteria were analyzed using the same techniques: (i) projects should be from the same company, and (ii) developer's expertise should be approximately the same[1]. Two sample projects are shown in Figure 7.1 and Figure 7.2. For purposes that will be explained later, one should observe that the evolution in mostly sub-linear, approaching logarithmic best-fits at the end of iterations.

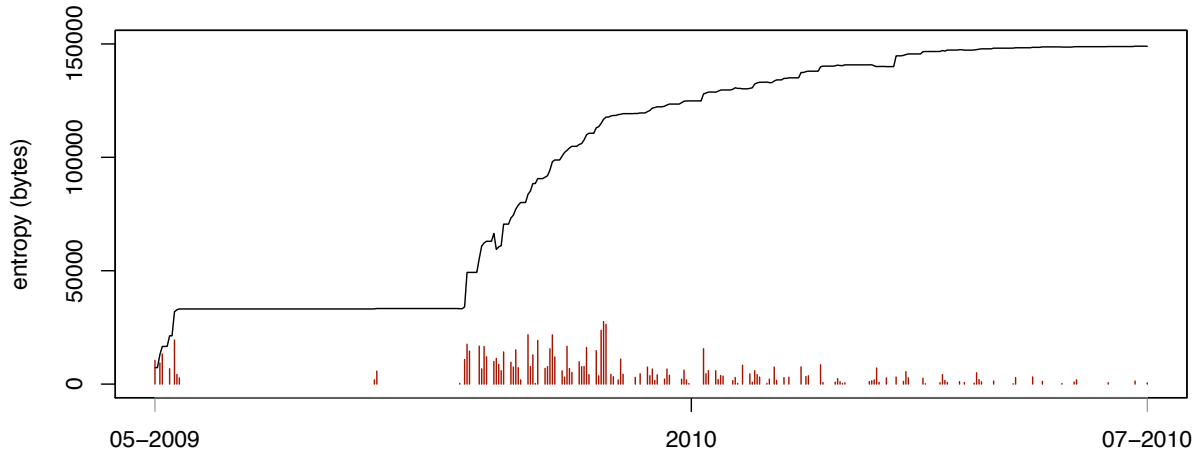


Figure 7.2: This chart shows the SMQVU evolution between May 2009 and July 2010 (≃ 1 year).

7.4 LOCVS

The municipal city hall of Porto established the goal of developing an information system to manage the collections, premises and places in the city, with relevant interest regarding architec-

[1] If possible, the same developers should have worked on all projects.

tural, urbanism, landscaping and archeological traits. Due to the complexity and scope of this topic, the fulfillment of this goal had a first phase consisting on the elicitation of sectorial information based on the Inventory of Architectural Heritage of Porto (IPAP), and in the production of archeological records regarding interventions and materials.

In a subsequent phase, the objective was to develop and deploy a software solution called *Locvs - Management Architectural and Archeological Heritage of the City of Porto*, compromising two main sub-goals (i) to allow the iterative and incremental structuring of available information, and foster its exploration, and also (ii) to support the workflow of managing and evolving this system of sectorial information [PFSP10].

In this section, we briefly document the development of the software system components of *Locvs*, briefly describing its idiosyncratic traits, main occurrences, and pursued activities *w.r.t.* its relevant phases for this dissertation, namely: (i) requirement analysis, (ii) application development, and (iii) software evolution.

7.4.1 Core Concepts

The conceptual structure of information regarding *architectural heritage* is based on a group of analysis units, namely (i) *public spaces*, (ii) *architectural collections*, (iii) *buildings and private spaces*, and (iv) *construction and ornamental materials*. Each unit of analysis aggregates a set of forms which fundamentally contains the most relevant information for the characterization and evaluation of the reality under analysis, and in some cases, complementary information of photographic, cartographic, bibliographical and otherwise archival nature.

Architectonic heritage, either under protection (i.e., according to the law in vigor) or in the process of becoming protected (and thus under internal regulations), is defined as those existences produced at an architectural scale, namely (i) constructions of monumental nature or relevant interest, and (ii) collections of building, streets, constructions or spaces which recognition as heritage may not be straightforward, and which evaluation require them to undergo a process of rigorous analysis of all the known elements and their relationship with the surrounding environment.

Archeological heritage refers to (i) all the archeological interventions that take place within the city boundaries, (ii) all the records and estate directly resultant from them, and (iii) to singular findings or donations. The system also encompasses the process of appreciation for urbanization licensing, with respect to (i) the delimitation of areas with high archeological potential, (ii) the establishment of areas of such potential, and (iii) the establishment of several other restrictions resultant from the "Plano Director Municipal" [Min99, Cam04]. It also includes the management of information, in diverse media supports (not always standard), directly related to the activity of managing the archeological heritage of the city.

7.4.2 Time-Series Analysis

The development of *Locvs* started in the 17th July 2007, simultaneously with the *Oghma* framework, with the first beta deployment occurring two months later in the 19th of September. The allocated team had a maximum size of three members during this period. As seen in Table 7.1, the first final deployment occurred in 10th March 2008, and the initial implementation was considered finished one month later, in 28th April. The customer then initiated a 10½ months revision, until the official acceptance in 7th August 2009. The project was then assigned a maintenance period of one year.

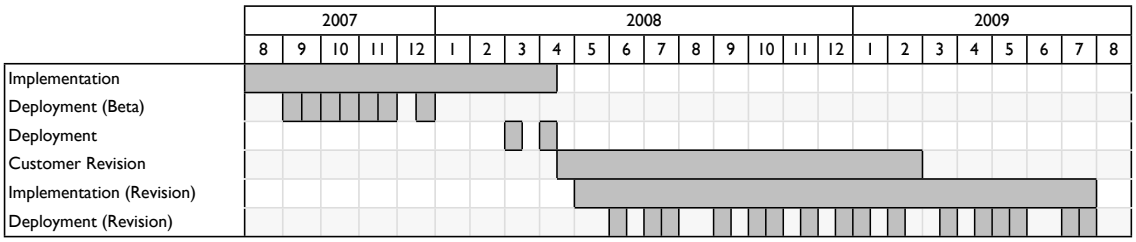


Table 7.1: Chronogram of the development of *Locvs*.

Because the model definition could be serialized as a XML file, it was relatively easy to measure its evolution. The chart presented in Figure 7.3 shows the cumulative number of lines of XML code, along with the number of changes.

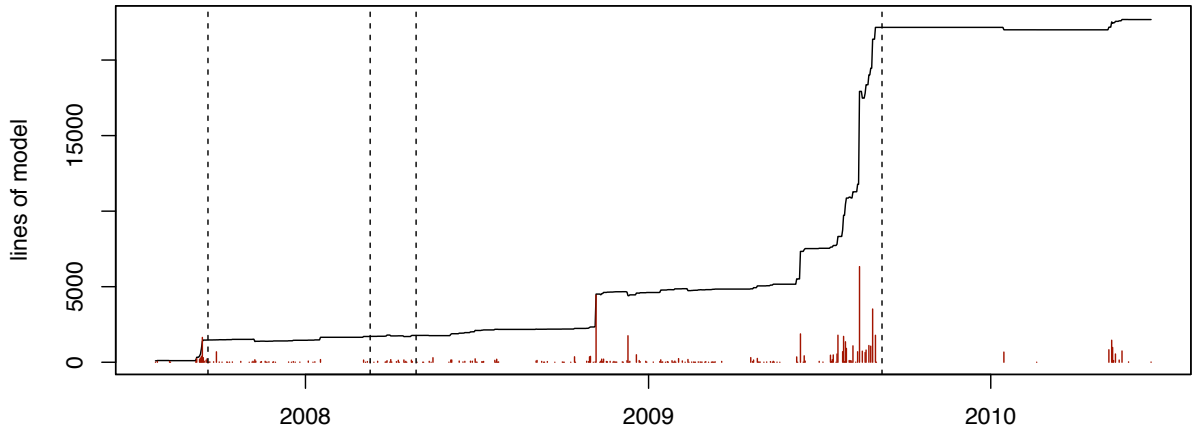


Figure 7.3: This chart shows the *Locvs* model evolution between July 2007 and June 2010 (3 years), measured in XML-serialized lines of code. The vertical lines are the number of changes. Dashed vertical lines mark major milestones seen in § 7.4.2.

7.4.3 Conclusion

From the observation of Figure 7.3, it seems obvious that the project has undergone three different levels of evolution. The first one, between the first beta deployment in September 2007 and

the last stable deployment in April 2008, corresponds to 1400–2000 lines of model description. Because of the closely resemblance between the two artifacts (i.e., the DSL used in Oghma and UML), it was the analysts opinion that this first description of the model represented an accurate translation of the analysis artifacts, namely UML class diagrams and use-case stories. This phase ended around model revision 750, after 14 iterations (i.e., deployments) with an average time of two-weeks per iteration.

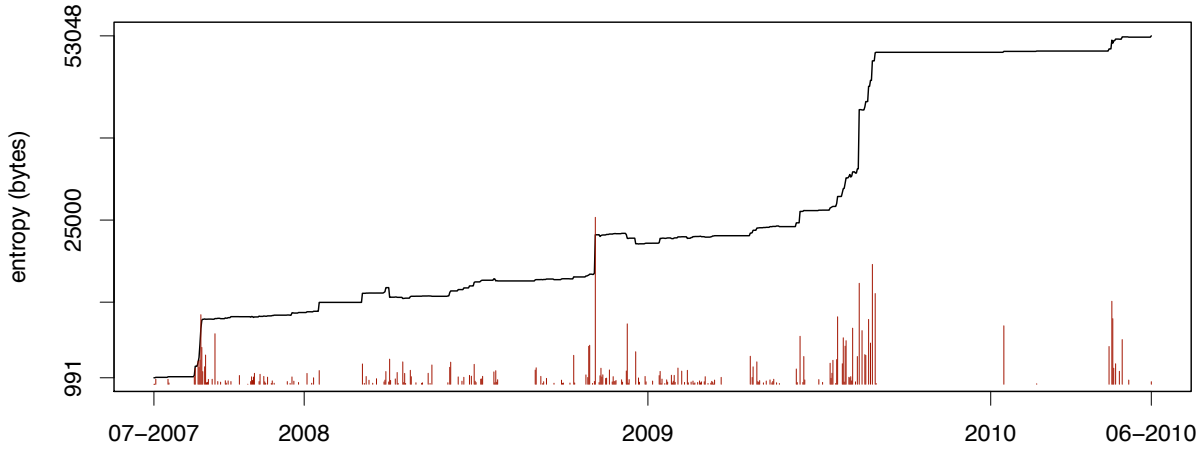


Figure 7.4: Locvs model complexity. This chart shows the evolution of model complexity between July 2007 and June 2010, measured in compressed bytes. The vertical lines display the cumulative size in bytes of the compressed differences per day.

The second leap in complexity occurred during the 10½ months length revision, with a drastic change of information near the end of 2008. The fact that Figure 7.3 (p. 119) shows a two-fold increase in LOC (to roughly 5000), supported by a more striking increase in entropy shown by Figure 7.4, provides evidence that the revision process introduced more structural elements (and hence new requirements) than those elicited during analysis.

The final leap in complexity occurred near the official acceptance of the final application, around August 2009 (revisions 1551–1589), which boosted the LOC to roughly 22162 (a four-fold increase when compared to the previous phase), but without an observable impact in the total number of structural elements after manual observation of the model. These results are consistent with the massive increase of non-structural elements (e.g., invariants, derived rules, views, etc.), which definition became possible (and a priority) as soon as the core structure of the application stabilized. Nonetheless, it is still notable that they alone account for an entropy increase of more than 100%.

In summary, from the first deployment to the official acceptance (16 months), one can observe a ten-fold increase in the model description (measured in LOC), a five-fold increase in the model complexity (measured in entropy), and a two-fold increase in basic structural elements (measured in structural elements), supporting the claim that traditional development approaches would not had efficiently coped with the nature of this project in the same time-span here ob-

served.

7.5 ZEPHYR

With the advent of mass digitalization of documents, which until now required physical storage, the need of ensuring its correct maintenance and long-term preservation is becoming paramount. As such, the municipal city hall of Porto has the goal to implement a central unit of digitalization in charge of this process.

7.5.1 Core Concepts

The dematerialization of the physical processes in digital objects becomes indissociable of certain premisses, such as (i) the correct identification of the digitized documents, and (ii) the maintenance of all inherent structure to each of these documents. It is thus necessary to validate and attach to each document every information that needs to be maintained about it, to ease subsequent access to the information and data preservation. The very act of storing this information is a process that can involve a multitude of participants along the several phases a digital object flows, mainly because during the lifespan of a document, its registration (and metadata) may still being altered while not yet associated with the digital object it represents.

The *Zephyr* application, presented as an extension module to GISA [CPP10], is a tool that allows to collect metadata about digital objects. Users rely on *Zephyr* to create records for (i) digital compound objects, such as a "Processo de Obra", and (ii) simple digital objects, such as "Actas", "Requerimentos", etc. After capturing metadata for each of these units, *Zephyr* transforms this information, creating *ingestion packets* in FOXML [Fed10], and ingesting them in a FedoraCommons [Sta03] repository.

7.5.2 Time-Series Analysis

The chart presented in Figure 7.5 (p. 122) shows the model entropy, along with the differences per day. Contrary to *Locvs*, this project revealed very stable, with an actual decrease of entropy around September 2009[2].

7.5.3 Conclusion

Zephyr was a single-person project, with minimal changes to the model, and very quick application deployment. Its primary usage as a use-case[3] was to discard three concerns raised by the

[2] When the developer was asked for the reason of this decrease, it was shown that the initial model contained some assumptions that revealed unnecessary after the application was first deployed.

[3] If one choses to ignore the commercial reasoning that *Oghma* was used because development was already expected to be faster.

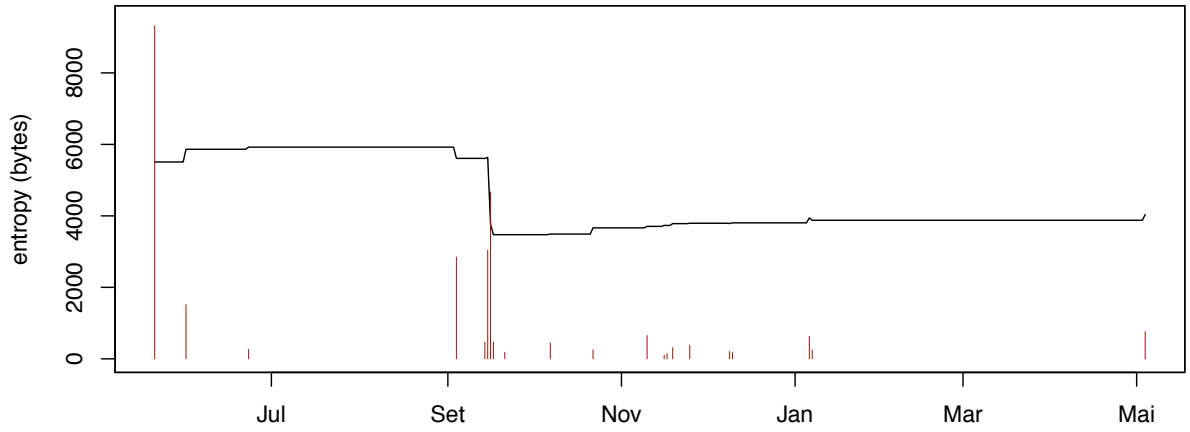


Figure 7.5: Zephyr model complexity. This chart shows the evolution of model complexity between June 2009 and May 2010, measured in compressed bytes. The vertical lines display the cumulative size in bytes of the compressed differences per day.

on-going experience with *Locvs*, viz. (i) was the framework sufficiently decoupled from its initial application to be used in other products, (ii) would a non-framework developer have time to adapt to *Oghma* and still deliver in an acceptable timespan, and (iii) is the framework only suitable for systems with unstable requirements?

Obviously, there is a strong interpretation to data presented in Figure 7.5; the application was not *incomplete by design*. This was known and expected, but the success in deploying *Zephyr* shows that the framework's, albeit not intentionally, can easily act as a *Rapid Application Development* tool. Still, it helps to support hypothesis H2, H3, H5-H7 — see Chapter 4 (p. 47).

7.6 SURVEY

A questionnaire was handed to professionals that had significant contact with the *Oghma* framework. This questionnaire was designed as a set of five-point Likert-scale [Lik32] items, or statements, which the respondent is asked to evaluate according to any kind of subjective or objective criteria, thus measuring either positive or negative response to the statement. The scale was set as: (1) strongly disagree, (2) somewhat disagree, (3) neither agree nor disagree, (4) somewhat agree, and (5) strongly agree. These items where divided into the following groups: (i) Background, (ii) Overall satisfaction, (iii) Development style and process, (iv) Future expectations, and (v) New features.

The subjective volatility is based on judgment of the subjects. All chosen subjects were directly involved in the host projects as developers, analysts, domain experts and/or internal users. They also had the task of reviewing requirements. Therefore, we believe that they were capable of accurately and reliably completing the survey. The raw data can be found in Table E.1 (p. 180) for $n = 7$.

7.6.1 Background

The results from the background of each participant can be found in Table 7.2. Participants of the study formed a relatively heterogeneous group (as evidenced by some values of σ), but with considerable experience with the object-oriented paradigm, UML, and the C# programming language. Most were used to develop industrial-level applications, by doing requirements engineering and analysis and specification of information systems. Regarding development style, participants were mainly used to classic and agile development methodologies.

I have considerable experience…	12345	$\bar{x}$	σ
…with object-oriented frameworks		3.57	1.27
…with the Oghma framework		3.43	1.27
…with the C# programming language		4.00	1.00
…doing requirements engineering		3.57	0.98
…developing industrial-level applications		4.00	1.53
…analyzing and specifying information systems		3.86	1.07
…testing and ensuring product quality		3.43	1.13
…with object-oriented architecture, design and implementation		4.00	0.58
…with agile development methodologies		3.14	0.90
…with classical development methodologies		3.29	0.76
…with formal development methodologies		1.86	1.07
…with UML		3.14	1.21

Table 7.2: Background results of industrial survey, each line representing the data of a single question, with corresponding means and standard deviation values.

7.6.2 Overall Satisfaction

The results for the overall satisfaction can be found in Table 7.3 (p. 124). Participants' answers supports the hypothesis that the framework provides a suitable infrastructure for developing *incomplete by design* systems. The answers also provide evidence that the development focus more on domain objects than implementation artifacts, easing the translation of conceptual specifications and the construction of user-interfaces. The effort of adding or changing existing requirements was strongly considered lower than any other conventional approach, and the usage of resulting artifacts in production-level requirements was significantly supported.

7.6.3 Development Style & Process

The results for the development style can be found in Table 7.4 (p. 124). As expected, the answers support the hypothesis that using such framework, and generally AOM systems, would be more in line with the agile principles in what concerns *embracing change*, and having constant feedback over the project (which would be very high when developing face-to-face with the client).

Overall, I…	12345	$\bar{x}$	σ
…found Oghma suitable for solving most tasks I needed		3.86	0.38
…thought more in terms of domain objects than implementation artifacts		4.00	0.82
…found it difficult to directly translate specifications into final artifacts		2.14	0.69
…was able to create and evolve the object model at least as rapidly as I could create a specification		4.00	0.82
…was able to create and evolve the user interface at least as rapidly as it could normally have been prototyped		4.14	0.90
…felt that any additional requirements represented considerable added effort compared to conventional approaches		1.86	0.90
…felt that any change of requirements represented considerable added effort compared to conventional approaches		1.86	0.90
…would use the resulting application in production-level environments with minimal or no change		4.29	0.76

Table 7.3: Overall satisfaction results of industrial survey, each line representing the data of a single question, with corresponding means and standard deviation values.

I found the development style of this setup suitable for…	12345	$\bar{x}$	σ
…using in the context of agile methodologies		4.43	0.98
…using in the context of classic methodologies		3.71	0.76
…using in the context of formal methodologies		3.29	0.76
…developing face to face with the client		3.86	0.90

Table 7.4: Development style results of industrial survey, each line representing the data of a single question, with corresponding means and standard deviation values.

The results for the development process can be found in Table 7.5 (p. 125). As expected, most of the difficulties where centered around understanding and extending the framework. Learning to use the framework's capabilities of persistency, user-interface and rules languages didn't pose a problem.

7.6.4 Future Expectations

The results for the development process can be found in Table 7.6 (p. 125). These results clearly support the hypothesis that development would be faster, less expensive, and easily maintainable than conventional approaches. Participants also agree that the final application would be more comprehensively tested, although not as strongly. This may be due to the switch of focus in quality; deployed applications would be as thoroughly tested as the framework is, but still dependent on the correct interpretation of the requirements. It is interesting to observe that there were no negative answers in these items, and that the values of σ are relatively low.

Most of my difficulties during development where…	12345	$\bar{x}$	σ
…dealing with persistency		2.14	0.69
…building the graphical user interface		2.43	0.98
…implementing the core business logic		3.43	0.98
…learning the domain specific language		2.71	0.76
…understanding the infrastructure		3.29	0.76
…extending the framework		3.57	0.98

Table 7.5: Development process results of industrial survey, each line representing the data of a single question, with corresponding means and standard deviation values.

Given that the basic infrastructure now exists, and with suitable modifications to the development process, my expectations are that subsequent systems developed with it, when compared to a more conventional approach, would be…	12345	$\bar{x}$	σ
…developed faster		4.57	0.79
…less expensive		4.29	0.76
…more comprehensively tested		3.71	0.49
…easier to maintain		4.29	0.76

Table 7.6: Future expectations results of industrial survey, each line representing the data of a single question, with corresponding means and standard deviation values.

7.7 LESSONS LEARNED

The development and usage of *Oghma* targeting adaptive applications allows us to elicit some lessons:

- **Skilled developers.** The skills needed to develop and maintain this type of architecture (at the infrastructure level) are not trivial to find, and developers are not necessarily at ease to work at these levels of abstraction. On the other hand, the skills needed to work on top of the developer are roughly at the same level as with traditional programming languages.

- **Domain specific v.s. general purpose.** From a framework standpoint, there is also a thin balance between a framework that makes the creation of new systems a quick and easy process, and one that is flexible enough to cover a wide scope of systems. Because it is very tempting to make the framework address all use-cases using an adaptive and model-driven approach, there is a risk of the final models becoming as elaborate and complex as a full-blown programming language. In this sense, HOOKS are a key issue, as they are not always easy to foresee, but they establish the border line between what should be regarded as part of the framework and what is particular behavior of a specific instantiation.

- **Quick prototyping and agile development.** Applying AOM-based systems in industry pre-

sented evidence of how easy it is to quickly build a functional prototype that can be shown to the customer, thus providing very early feedback before refining it into a production-level application. Not only the customer involvement in this process will increase due to the high number of iterations that become possible, reducing the burden of up-front design by allowing an incremental approach to formalization of the underlying business model, but end-user development capabilities offered by the framework could also ease the dependency on the developers. It is now the team belief that, now that the infrastructure is ready, the two-year analysis prior to the development could have been greatly reduced, by early providing functional prototypes.

7.8 VALIDATION THREATS

One issue that could affect conclusion validity is the number of case-studies (two) and involved software architects, engineers, and analysts (eight), despite the sheer dimension of the *Locvs* case-study, which collected data spawn throughout three whole years. In the context of software engineering research conducted in industrial settings, one should be aware that the company is taking a high risk employing novel techniques, with unpredictable results, to their normal workflow. Nonetheless, the success of the *Oghma* framework, and the conclusions here presented, were found sufficient to at least provide the basis for further usage in subsequent projects, which will allow the harvest of more data for further analysis.

The participants of the study also formed an heterogeneous group. Mainly, we had four different roles dealing with the *Oghma* framework: the analysts, the domain experts, the framework users, and the framework developers. Although some roles were overlapped by the same people, we believe that having chosen a homogeneous group would lead to the disadvantages of decreasing the number of subjects and affecting external validity. As was also mentioned, the subjective volatility is based on judgement of the subjects. Because all chosen subjects had direct contact with the host projects as developers, analysts, domain experts or internal users, we believe that they were capable of accurately and reliably completing the survey.

Finally, the most relevant threat to validation comes from the nature of use-case analysis; there is simply too much variables out of control. A more solid validation would require more than one team allocated to the same project, using different technologies, as well as tighter control over the development lifecycle to allow a rigorous measurement of time and effort spent by teams. Since this issue cannot be pursued within our industrial environment, it will be addressed in the next chapter.

7.9 CONCLUSION

This chapter presented two case studies built on top of the primary outcomes from this dissertation, as well as a questionnaire intended to evaluate the professional opinion of the participants. We based most of the time-series analysis on information theory, specifically by measuring Kolmogorov complexity and considering the project's lifecycles, which allowed to make comparisons among projects using different technologies. Most hypothesis presented in Chapter 4 (p. 47) were supported by evidence, although some validation threats remain due to the nature of use-case analysis. Those issues will be addressed in the next chapter as a quasi-experiment performed in a controlled environment.

Chapter 8

Academic Quasi-Experiment

8.1 Research Design 129
8.2 Experiment Description 132
8.3 Data Analysis 137
8.4 Objective Measurement 147
8.5 Validation Threats 147
8.6 Conclusion 149

This chapter details a quasi-experiment conducted within a controlled experimental environment using the *Oghma* framework. Although the industrial usage of the framework provides pragmatic evidence of its benefits, as detailed in Chapter 7 (p. 115), several of the threats inherent to that type of validation are here discarded, by performing a study on groups of MSc students building an information system from scratch, and applying two different treatments, viz. (i) the baseline, using Java, MySQL and Eclipse, and (ii) the experimental, using *Oghma* and Microsoft Visual Studio. Pre-test evaluation and post-test questionnaires are used to assess the outcomes of each treatment, and the final results reveal consistent with those presented in the previous chapter.

8.1 RESEARCH DESIGN

(Quasi-)experiments conducted in an academic context should be randomized, multiple-group, comparison designs, which may be implemented as part of graduate student teams lab work [CJMS10]. For this experiment, 18 MSc students from the Integrated Masters in Informatics and Computing Engineering, lectured at the University of Porto, Faculty of Engineering, were recruited. They all were 4th year students which chose to take an optional course on "Architecture of Software Systems", and hence were interested and motivated in the design and construction of complex (medium to large-scale) systems.

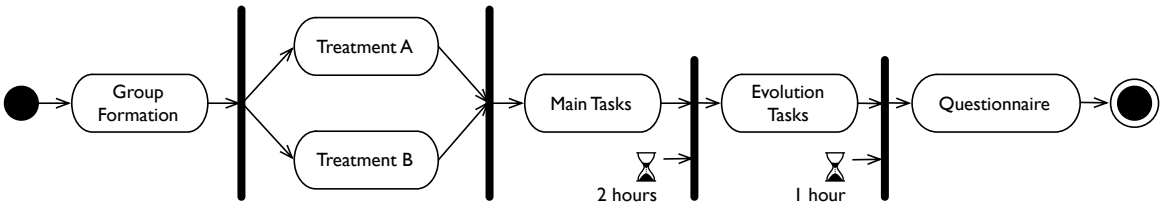


Figure 8.1: Experiment design. Each group received one of two possible treatments. All then proceeded for the main tasks. After two hours, the evolution tasks were handed. At the end of the 3rd hour, the subjects fully stopped their tasks, and proceeded to answer the questionnaire.

8.1.1 Treatments

To perform the experiment, two different treatments were applied:

- **Baseline treatment.** The establishment of the baseline followed traditional development methodologies and tools, assumed as familiar to the subjects, to construct and evolve the given system. These consisted in Java for programming, supported by MySQL as a database backend, and Eclipse as an IDE.
- **Experimental treatment.** The system under test, *Oghma*, was handed to the subjects that received the experimental treatment.

8.1.2 Pre-test Evaluation

It is important to ensure that the base skills for every member does not pose any significant statistical deviation. To ensure that, a selection of a subset of courses based on their academic track was made, which could influence the experiment outcome, namely: (i) Programming Fundamentals, (ii) Programming, (iii) Algorithms and Data Structures, (iv) Algorithm Design and Analysis, (v) Software Engineering, (vi) Software Development Laboratory, and (vii) Information Systems. The grades of each subject that participated in the experiment can be found in Table A.1 and Table A.2 (p. 161). An independent-samples t-test was conducted to compare the average students' grades for experimental and baseline groups. As shown in Table 8.1 and Table 8.2 (p. 131), there was **no significant** difference in the scores for the experimental ($M = 14.20$, $SD = 1.27$) and baseline ($M = 13.87$, $SD = 0.77$) conditions; $t(13) = 0.572$, $\rho = 0.577$, within a 95% confidence interval.

Group	N	Mean	Std. Deviation	Std. Error Mean
1	9	14.200	1.2738	0.4246
2	6	13.867	0.7659	0.3127

Table 8.1: Student grades group statistics.

	F	Sig.	t	df	Sig. (2-tailed)
Eq. Var. Assumed	4.329	0.058	0.572	13.000	0.577
Eq. Var. Not Assumed			0.632	12.941	0.538

Table 8.2: Independent Samples Test. The first two columns are the Levene's Test for Equality of Variances, showing a significance greater than 0.05. The other three columns are the t-test for Equality of Means. Since we can assume equal variances, the 2-tailed value of 0.577 allow us to conclude that there is no statistically significant difference between the two conditions.

There is also the need to ensure that all subjects share common skills with respect to metamodeling, architectural and design patterns, adaptive object-models and its ecosystem. Although both GoF [GHJV94] and POSA [BMR+96] patterns are part of the curricula, it was required to add two lectures about AOMs prior to performing the experiment. In these lectures, both groups learned the core patterns of AOMs, and performed a simple exercise of implementing the Type-Object [JW97] pattern in Java.

Nevertheless, it should be noted that the subjects under the experimental treatment never had any contact with the framework prior to the experiment. All the available information was handed at the beginning of the experiment, as a small digital document, which is here reproduced in Appendix D. This documentation was deliberately incomplete, containing slight omissions and inconsistencies for the purpose of simulating a real-world documentation and avoid any bias towards the tasks subjects were meant to perform.

8.1.3 Process

This (quasi-)experiment was intended to assess several distinct claims, which were matched into two different phases: development and maintenance. The two treatment groups were further split in groups of three subjects. Each group would enter a restrict laboratory environment, with (i) a single computer available with internet access, and (ii) a whiteboard.

Data Collection

The experiment had two live video feeds being recorded: (i) a standard camera, pointing directly at both the subjects and the whiteboard, and (ii) a *screencast* of the computer the subjects were using. This was used to measure time, correctness and complexity of the produced artifacts non-intrusively. The analysis of this data is made in § 8.4 (p. 147).

First Phase: Construction

In the first phase, a small "Requirements Specification Report" where handed in a closed envelope, which included brief user stories and UML class diagrams semi-formalizing a small system

for managing scientific conferences. Their objective would be to implement a full system using their given technique and restrictions. While the time available for pursuing the implementation could be based in effort estimations made by software-engineer professionals, there was severe time constraints from the availability of the subjects. As such, the overall time limit was set to 3 hours, thus reserving 2 hours for this phase. The exact details of each task can be found in § 8.2.1.

Second Phase: Evolution

The subjects were not aware of a second phase, which goal was to assess the efficiency in adapting to changing requirements. As such, 1 hour before the time limit, the subjects were handed a second envelope as detailed in § 8.2.2 (p. 135).

8.1.4 Post-Test Questionnaire

At the end of the experiment, i.e., after the three hours time limit, the questionnaire in Appendix B was handed to the subjects. This questionnaire was designed using a Likert scale [Lik32]. This psychometric bipolar scaling method contains a set of Likert items, or statements, which the respondent is asked to evaluate according to any kind of subjective or objective criteria, thus measuring either positive or negative response to the statement. In this experiment, we used 54 five-point Likert items with the following format: (1) strongly disagree, (2) somewhat disagree, (3) neither agree nor disagree, (4) somewhat agree, and (5) strongly agree. The answers from all the participants are detailed in Appendix C, and further analyzed in § 8.3 (p. 137).

8.1.5 Independent Validation

The independent experimental validation of claims is not as common in Software Engineering as in other, more mature sciences. Therefore, the (quasi-)experiment here detailed was designed as an *experimental package*, to be performed in different locations, and lead by different researchers, in order to enhance the ability to integrate the results obtained and allow further meta-analysis on them.

8.2 EXPERIMENT DESCRIPTION

The following sections describe every task given to the participants, quoting the presented text.

8.2.1 First Phase: Construction

During the construction phase the tasks were mainly focused on assessing how efficiently the participants could incrementally build an information system, with three iterations (or releases):

You have been given the job of implementing an information system for managing scientific conferences. After a careful requirements analysis, the engineers have concluded that the system should be implemented in three distinct iterations. For each of these iterations – one for each task – you'll find a detailed UML *class diagram. Due to the fact that the client wants to validate your system at the end of each iteration, you'll have to produce three intermediate releases. Each release should have a working Graphical User Interface and Persistency Engine. The user should be able to create, read, update and delete the modeled concepts.*

Description of Task 1

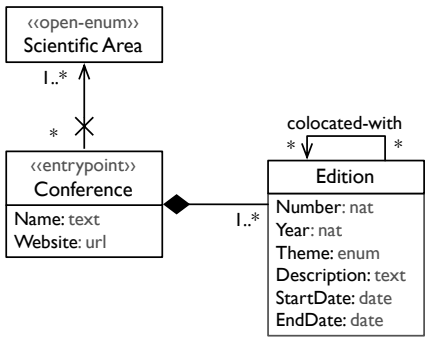


Figure 8.2: Task 1

The first task was designed to yield a very simple system, where only a single screen would be needed to view and edit the information. There was no polymorphism, shared aggregations or any type of conditional rules, and the whole system could be roughly stored in two database tables:

In this task, you'll implement the basic concepts of a scientific conference. A conference is typically related to a specific Scientific Area (e.g., Computer Science or Software Engineering), but is it not uncommon to find conferences related to several areas. Each conference has several editions, normally once per year (e.g., Pattern Languages Of Programs 2008). Due to several factors, it is also common for conferences to be co-located with others. For example, the 2008 editions of "Pattern Languages of Programs" and "Object-Oriented Programming, Systems, Languages & Applications" were co-located. In the end of this iteration, the user should be able to manage Conferences and Editions. Model elements tagged with the stereotype entrypoint *represent main entry points to the system (e.g., the user should be able to invoke a list of Scientific Areas from the applications main menu or similar mechanism).*

Description of Task 2

The second task was designed to introduce the need for more screens (due to indexation), and a "false" sense of polymorphism in the session type. The number of database tables could grew from two to four, but with minimal modification to exiting artifacts:

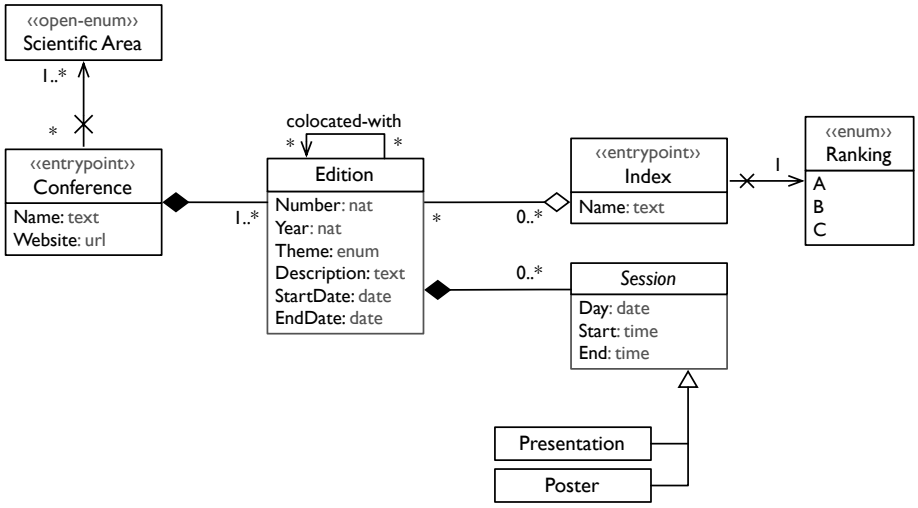


Figure 8.3: Task 2

In this task, you'll extend your system with two extra features. The first one – Indexes – classifies conferences according to a rating system per year. For example, the index "ISI Web of Knowledge" rates thousands of conferences every year. The rating is given to a particular edition of a conference, so ISI could rate "Pattern Languages of Programs" as an A in 2015, and as a B in 2016. The second one - Sessions - allows the user to manage the program (contents) of a conference. There are two types of sessions: (a) Presentations, and (b) Poster Sessions. For example, the 2008 edition of "Object-Oriented Programming, Systems, Languages & Applications" had one (1) poster session, and twelve (12) presentations.

Description of Task 3

The third task was deliberately designed to require more complex user-interaction and conditional rules, involving shared many-to-many relations and relation properties:

In this task, you'll provide your system by adding some remaining core concepts of scientific conferences. People that participate in conferences as authors have to submit at least one paper, either alone or with colleagues. In addition, they may be Chairs, Organizers or simple Participants. Different editions normally have different chairs and organizers.

Description of Task 4

The purpose of the fourth task was to break any "abstractions" that could have emerged during the previous iterations:

In this final task, for reasons of practicability, the client has requested that whenever a paper is submitted to a conference, an email should be sent for the Chairs with the Submission and Authors relevant information.

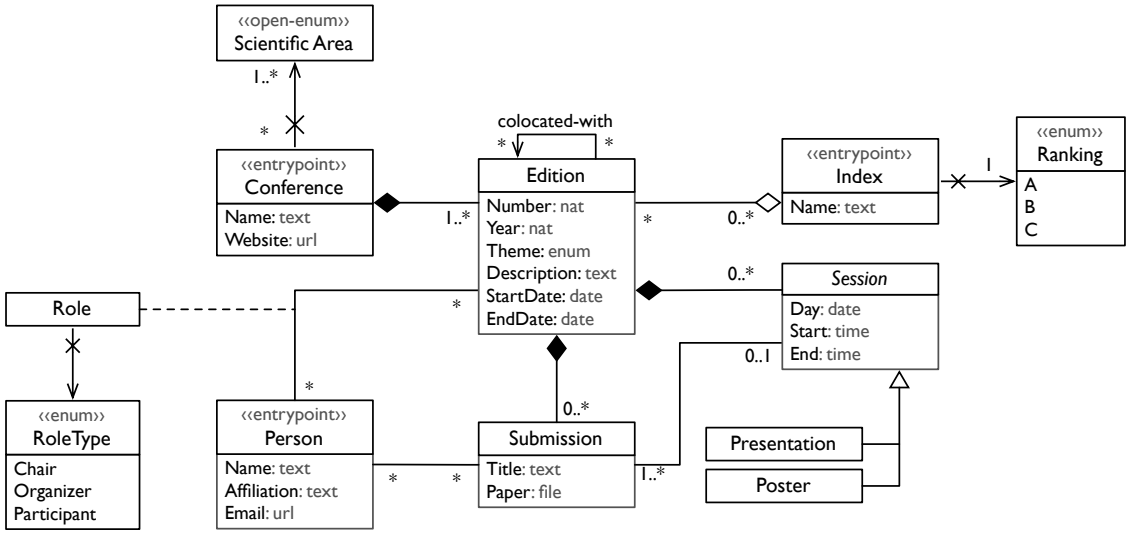


Figure 8.4: Task 3

8.2.2 Second Phase: Evolution

During the evolution phase the tasks were focused on assessing of efficiently the participants could handle change in their artifacts:

> *During one of the validation sessions with the users, both of them realized that some core aspects of the system were neglected, and, as such, there would be slight changes in the requirements in order to improve the overall value of the application.*

Description of Requirements Change 1

This task was designed with the intention of changing some basic assumptions that would naturally emerge in Task 1. The Scientific Area enumeration was promoted to an *entrypoint*. It also introduced "true" polymorphism in Event, due to the relation between Conference and Workshop:

> *It is imprecise to say that every scientific meeting is a conference. In fact, it is quite common for conferences and workshops to occur simultaneously. Usually, workshops take place as events within a conference. This way, Conference should now be considered a specialization of Event. the same applies to Workshop. In addition, a workshop can only occur in the context of a single Conference. Additionally, it is now relevant to promote Scientific Area to an entry point in the system.*

Description of Requirements Change 2

This task was very small, but it could easily disrupt previous assumptions. The "false" polymorphism in the type of Session (which could have been implemented using a simple type-

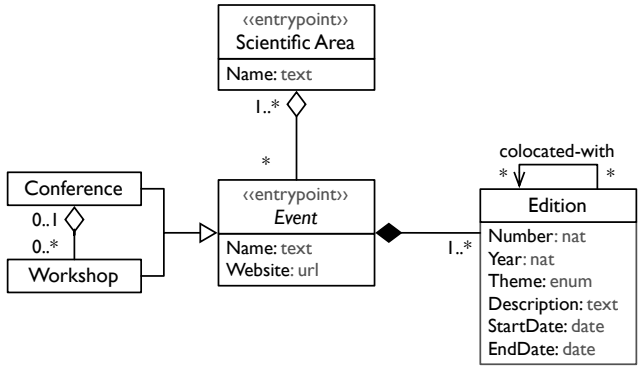


Figure 8.5: Changes for Task 1

discriminator), now lead to the need of inheritance, by extending the allowed sub-types and introducing a property that only makes sense in Keynote:

> *A new session type is added – the Keynote – for which is necessary to store its synopsis.*

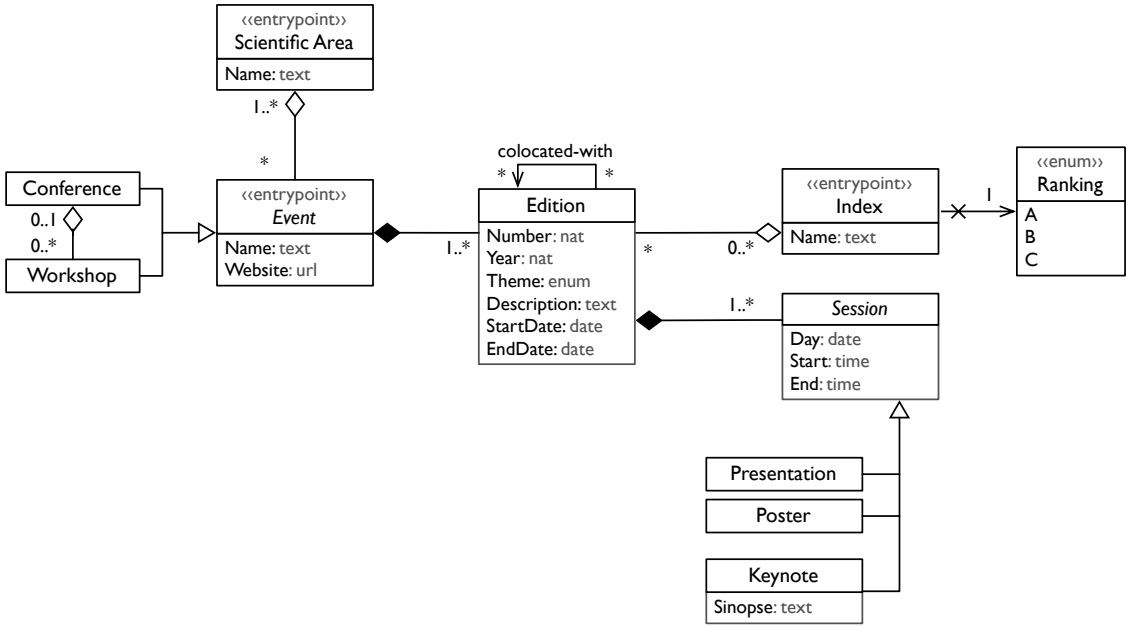


Figure 8.6: Changes for Task 2

Description of Requirements Change 3

This final task bring several rules to the system: the new relations between Submission, Presentation and Poster override the previous relation between Submission and Session, specifying different cardinalities in relation ends. Role now also carries an additional property:

> *The users realized that, although Presentation and Poster sessions are always related to Submission (via Session), a presentation is always about a single submission, while posters sessions*

normally occur in parallel (i.e., all submitted posters presented at the same time). Keynotes, on the other hand, have nothing to do with submissions, and are presented by a high-profile invited researcher in a particular field. A subtle detail is that the relationship between Person and any particular Conference should allow users to store some Notes over the registration (e.g., some guests don't pay fee, other have special requests like vegetarian food, etc.)

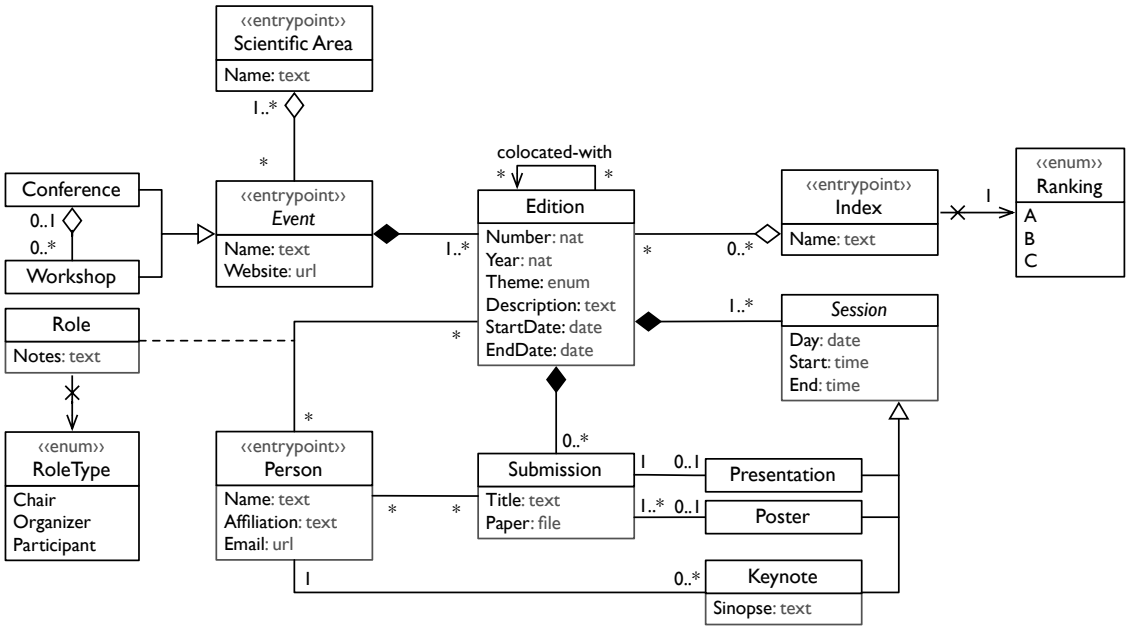


Figure 8.7: Changes for Task 3

8.3 DATA ANALYSIS

The post-test questionnaire handed to the subjects was designed using a Likert scale — see Appendixes B and C. This experimental design used a 54 five-point Likert items with the following format: (1) strongly disagree, (2) somewhat disagree, (3) neither agree nor disagree, (4) somewhat agree, and (5) strongly agree. The 54 items where divided into the following relevant groups:

1. **Background.** Although an objective comparison between the background of each group was already conducted using the subjects average grades in key courses § 8.1.2 (p. 130), it is important to assert that there is no subjective difference among the participants with respect to basic skills, with the exception of BG1.2, since no group had any prior contact with the *Oghma* framework.

2. **External Factors.** Analysis of external factors allowed us to remove any validation threats concerning the experimental conditions, particularly the fact that subjects were being video-taped.

3. **Overall Satisfaction.** This was the main group that provided subjective validation to the thesis, by questioning subjects about their performance, effectiveness, correctness, and reaction to the introduced requirements change in Phase 2.

4. **Development Process.** This group of questions intended to put the experiment in perspective with both previous subject experiences, and the main difficulties they had encountered.

We will now present a detailed analysis for the most relevant items in the questionnaire. Let the null hypothesis be denoted as H_0, the alternative hypothesis as H_1, the baseline group as G_b, the experimental group as G_e, and ρ the probability estimator of wrongly rejecting the null hypothesis. Then, the alternative hypothesis are either: (i) $H_1 : G_e \neq G_b$, the experimental group differs from the baseline, (ii) $H_1 : G_e < G_b$, the measure in the experimental group is lower than the baseline, or (iii) $H_1 : G_e > G_b$, the measure in the experimental group is greater than the baseline. The outcomes of the two treatments were compared for every answer using the non-parametric, two-sample, rank-sum Wilcoxon-Mann-Whitney [HW99] test, with $n_1 = n_2 = 9$. The significance level for all tests was set to 5%. Probability values of $\rho \leqslant 0.05$ are considered *significant*, and $\rho \leqslant 0.01$ considered *highly significant*. The corresponding alternative hypothesis are further detailed for each question, and the raw data can be found in Table C.1 (p. 169).

8.3.1 Background

Although an objective comparison between the background of each group was already conducted using the subjects average grades in key courses § 8.1.2 (p. 130), this section rejects any subjective difference among the participants with respect to their basic skills, despite a deliberate disadvantage in the experimental group *w.r.t.* the programming language and environment, as observed in the results of items BG1.2 and BG1.3.

BG1.1 I have considerable experience using frameworks

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental ($\bar{x} = 3.44$, $\sigma = 0.53$) and baseline ($\bar{x} = 3.44$, $\sigma = 0.88$) conditions, as seen in Table 8.3 (p. 139). Students revealed a very modest knowledge on using frameworks, consistent with their academic track.

BG1.2 I have considerable experience with this particular framework

Let $H_1 : G_e < G_b$, there was **highly significant** difference ($\rho = 0.001$) in the scores for the experimental ($\bar{x} = 1.11$, $\sigma = 0.33$) and baseline ($\bar{x} = 3.11$, $\sigma = 1.27$) conditions, as seen

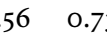
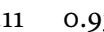
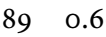
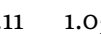
	EXPERIMENTAL			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
BG1.1		3.44	0.53		3.44	0.88	$\neq$	41.0	1.000
BG1.2		1.11	0.33		3.11	1.27	$<$	06.0	<u>0.001</u>
BG1.3		3.33	1.12		4.33	0.50	$\neq$	19.5	<u>0.045</u>
BG1.4		2.56	0.73		2.78	0.97	$\neq$	36.5	0.731
BG1.5		3.11	0.93		3.00	1.00	$\neq$	39.5	0.962
BG1.6		4.33	0.71		3.44	0.88	$\neq$	62.5	<u>0.036</u>
BG1.7		3.67	0.71		2.89	0.93	$\neq$	62.0	<u>0.043</u>
BG1.8		3.89	0.60		3.67	0.71	$\neq$	48.5	0.457
BG1.9		2.78	0.83		2.67	0.71	$\neq$	43.0	0.848
BG1.10		4.11	1.05		3.44	0.73	$\neq$	58.5	0.102

Table 8.3: Summary of Background results, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test. Items with significant statistical difference in their score have the value of the probability estimator underlined.

in Table 8.3. This result is in accordance to the fact that no subject had any prior contact with the *Oghma* framework.

BG1.3 I have considerable experience with this particular programming language

Let $H_1 : G_e \neq G_b$, there was **significant** difference ($\rho = 0.045$) in the scores for the experimental ($\bar{x} = 3.33$, $\sigma = 1.12$) and baseline ($\bar{x} = 4.33$, $\sigma = 0.50$) conditions, as seen in Table 8.3. This result was also expected, since Java plays a significant role in the students curricula, with C# only being marginally mentioned, if at all.

BG1.4 I have considerable experience developing industrial-level applications

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 0.731$) in the scores for the experimental ($\bar{x} = 2.56$, $\sigma = 0.73$) and baseline ($\bar{x} = 2.78$, $\sigma = 0.97$) conditions, as seen in Table 8.3. As expected from 4th year students, their subjective opinion shows that they were not very experienced in developing industrial-level applications.

BG1.5 I have considerable experience analyzing and specifying information systems

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 0.962$) in the scores for the experimental ($\bar{x} = 3.11$, $\sigma = 0.93$) and baseline ($\bar{x} = 3.00$, $\sigma = 1.00$) conditions, as seen in Table 8.3. Similarly to item BG1.4, their subjective opinion is that they possess moderate experience in information system analysis and specification.

BG1.6 I have considerable experience with object-oriented architecture design and implementation

Let $H_1 : G_e \neq G_b$, there was **significant** difference ($\rho = 0.036$) in the scores for the experimental ($\bar{x} = 4.33$, $\sigma = 0.71$) and baseline ($\bar{x} = 3.44$, $\sigma = 0.88$) conditions, as seen in Table 8.3 (p. 139). These results are odd, since their academic background was the same, and the pre-test evaluation § 8.1.2 (p. 130) showed no statistical deviation on their grades. We assume this item was probably influenced by the overall success using the *Oghma* framework, which changed their subjective understanding on how they perceive object-oriented architectures.

BG1.7 I have considerable experience with agile development methodologies

Let $H_1 : G_e \neq G_b$, there was **significant** difference ($\rho = 0.043$) in the scores for the experimental ($\bar{x} = 3.67$, $\sigma = 0.71$) and baseline ($\bar{x} = 2.89$, $\sigma = 0.93$) conditions, as seen in Table 8.3 (p. 139). Similarly to item BG1.6, we also assume some subjective understanding on their perception after the experiment was done. Nevertheless, the scores are lower than item BG1.8, which is consistent with their academic background.

BG1.8 I have considerable experience with classical development methodologies

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 0.457$) in the scores for the experimental ($\bar{x} = 3.89$, $\sigma = 0.60$) and baseline ($\bar{x} = 3.67$, $\sigma = 0.71$) conditions, as seen in Table 8.3 (p. 139). As expected, both scores are consistently higher than the two other methodologies in items BG1.7 and BG1.9, which is consistent with their academic background.

BG1.9 I have considerable experience with formal development methodologies

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 0.848$) in the scores for the experimental ($\bar{x} = 2.78$, $\sigma = 0.83$) and baseline ($\bar{x} = 2.67$, $\sigma = 0.71$) conditions, as seen in Table 8.3 (p. 139). Also as expected, both scores are consistently lower than the two other methodologies in items BG1.8 and BG1.9, which is consistent with their academic background.

BG1.10 I have considerable experience with UML class diagrams

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 0.102$) in the scores for the experimental ($\bar{x} = 4.11$, $\sigma = 1.05$) and baseline ($\bar{x} = 3.44$, $\sigma = 0.73$) conditions, as seen in Table 8.3 (p. 139). Both groups exhibited positive responses to this item (which is also consistent with their academic background), and thus not posing a validation threat to the usage of UML class diagrams in their tasks.

8.3.2 External Factors

In the design of this experiment, data gathering was done using video-cameras, microphones, and software that captured screencasts. From the experimental point of view, it is important to discard if these devices, and the fact that participants knew they were being observed, posed a threat to validation. Additionally, each treatment were applied to groups of three participants, so it is also important to analyze if they worked well together and didn't reacted negatively to the experiment. This is a two-fold analysis, both focusing on the participants as a whole, and in group comparison.

	EXPERIMENTAL			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
EF1		1.89	0.78		1.78	1.39	$\neq$	50.0	0.389
EF2		4.44	0.73		3.33	1.12	$\neq$	63.5	0.037
EF3		4.67	0.50		3.89	0.78	$\neq$	63.0	0.035

Table 8.4: Summary of External Factors results, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

EF1 I felt disturbed and observed by the cameras

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 0.389$) in the scores for the experimental ($\bar{x} = 1.89$, $\sigma = 0.78$) and baseline ($\bar{x} = 1.78$, $\sigma = 1.39$) conditions, as seen in Table 8.4. Moreover, 94.4% of the total participants felt indifferent to the fact they were being observed by the cameras, so this factor can be discarded as a threat.

EF2 I enjoyed programming in the experiment

Let $H_1 : G_e \neq G_b$, there was **significant** difference ($\rho = 0.037$) in the scores for the experimental ($\bar{x} = 4.44$, $\sigma = 0.73$) and baseline ($\bar{x} = 3.33$, $\sigma = 1.12$) conditions, as seen in Table 8.4. In other words, this item was measuring the *fun factor* or the *novel factor*. Although both groups reacted positively to the experience, 89% of the participants in the experimental group agreed with the assertion, among which 63% strongly agreed with it. Despite the several interpretations to this results, it is believed that the general success of the experimental group, as well as the feeling of working with "something new", contributed to the statistical difference observed. Nonetheless, only two of the total participants had a mildly negative feeling towards the experiment, so this factor can also be discarded as a threat to the whole experiment.

EF3 I would work with my partners again

Let $H_1 : G_e \neq G_b$, there was **significant** difference ($\rho = 0.035$) in the scores for the experimental ($\bar{x} = 4.67$, $\sigma = 0.50$) and baseline ($\bar{x} = 3.89$, $\sigma = 0.78$) conditions, as seen in Table 8.4 (p. 141). Similarly to item EF2, the experimental group showed a slight evidence in working with the same partners again. Nonetheless, the fact that there were no negative answers further allow us to discard this factor as a threat to the whole experiment.

8.3.3 Overall Satisfaction

This was the main group that provided subjective validation to the thesis, by questioning subjects about their performance, effectiveness, correctness, and participant's reaction to the introduced requirements change in Phase 2.

	EXPERIMENTAL			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
OVS1		3.67	0.50		2.44	0.73	$>$	72.0	0.002
OVS2		3.00	0.87		3.67	0.50	$>$	21.0	0.977
OVS3		2.89	0.93		2.89	0.78	$<$	42.5	0.594
OVS4		2.22	0.83		2.89	0.78	$<$	24.5	0.072
OVS5		3.44	0.73		3.22	0.67	$>$	48.5	0.233
OVS6		3.67	1.50		1.33	0.50	$>$	73.5	0.001
OVS7		1.67	1.12		2.78	0.83	$<$	15.5	0.012
OVS8		1.89	1.05		3.00	0.87	$<$	16.5	0.016
OVS9		2.44	0.88		1.56	0.88	$>$	60.5	0.029

Table 8.5: Summary of Overall Satisfaction results, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

OVS1 Overall this particular setup was suitable for solving every task presented

Let $H_1 : G_e > G_b$, there was **highly significant** difference ($\rho = 0.002$) in the scores for the experimental ($\bar{x} = 3.67$, $\sigma = 0.50$) and baseline ($\bar{x} = 2.44$, $\sigma = 0.73$) conditions, as seen in Table 8.5. The results of this answer support the hypothesis that the *Oghma* framework is more suitable to develop and evolve the type of information system proposed in this experiment than a traditional approach of using a general purpose language and a relational database.

OVS2 I found myself thinking more about the end system purely in terms of the structure of domain objects

Let $H_1 : G_e > G_b$, there was **no significant** difference ($\rho = 0.977$) in the scores for the experimental ($\bar{x} = 3.00$, $\sigma = 0.87$) and baseline ($\bar{x} = 3.67$, $\sigma = 0.50$) conditions, as seen in Table 8.5.

The original hypothesis postulated that participants in the experimental group would be motivated to abstract from implementation details and think more in terms of domain objects. Surprisingly, it was the baseline group that showed an higher tendency to agree with this assertion, although not statistically significantly.

OVS3 I found myself thinking more about the end system more in terms of the structure of the database and user interaction

Let $H_1 : G_e < G_b$, there was **no significant** difference ($\rho = 0.594$) in the scores for the experimental ($\bar{x} = 2.89$, $\sigma = 0.93$) and baseline ($\bar{x} = 2.89$, $\sigma = 0.78$) conditions, as seen in Table 8.5 (p. 142). Similar to OVS2, the original hypothesis postulated that participants in the experimental group would be discouraged from considering details such as database structure and user-interaction. However, both groups displayed similar answers to this item and as such we cannot reject the null hypothesis.

OVS4 I found very difficult to directly translate specifications into final artifacts

Let $H_1 : G_e < G_b$, there was **no significant** difference ($\rho = 0.072$) in the scores for the experimental ($\bar{x} = 2.22$, $\sigma = 0.83$) and baseline ($\bar{x} = 2.89$, $\sigma = 0.78$) conditions, as seen in Table 8.5 (p. 142). Despite no single participant in the experimental group agreed with this assertion, there isn't statistical significance to support the hypothesis of a difference in complexity when translating conceptual specifications into final artifacts.

OVS5 I was able to create the underlying object model at least as rapidly as I could normally have created a specification

Let $H_1 : G_e > G_b$, there was **no significant** difference ($\rho = 0.233$) in the scores for the experimental ($\bar{x} = 3.44$, $\sigma = 0.73$) and baseline ($\bar{x} = 3.22$, $\sigma = 0.67$) conditions, as seen in Table 8.5 (p. 142). The results for this item do not support the hypothesis that creating the object model in the experimental group would be much quick that in the baseline group, when subjectively comparing to the effort of creating a specification. As such, the subjective notion that the effort of producing a specification would be as quick as declaring the object model using the framework is not supported by the results. The inability to reject the null hypothesis may be related to the fact that the experimental group was having its first contact with the framework, and as such the timespan of the experiment wasn't enough to reveal those benefits.

OVS6 I was able to create the user interface at least as rapidly as it could normally have been prototyped

Let $H_1 : G_e > G_b$, there was **highly significant** difference ($\rho = 0.001$) in the scores for the experimental ($\bar{x} = 3.67$, $\sigma = 1.50$) and baseline ($\bar{x} = 1.33$, $\sigma = 0.50$) conditions, as seen in Ta-

ble 8.5 (p. 142). As expected, the baseline group evidenced the complexity and time-consuming activity inherent to building an maintaining a functional user-interface, compared to the generative approach in the experimental treatment.

OVS7 Additional requirements represented considerable added effort

Let $H_1 : G_e < G_b$, there was **significant** difference ($\rho = 0.012$) in the scores for the experimental ($\bar{x} = 1.67$, $\sigma = 1.12$) and baseline ($\bar{x} = 2.78$, $\sigma = 0.83$) conditions, as seen in Table 8.5 (p. 142). These results support the hypothesis that the effort of adding new requirements was lower in the experimental group.

OVS8 Change of existing requirements represented considerable added effort

Let $H_1 : G_e < G_b$, there was **significant** difference ($\rho = 0.016$) in the scores for the experimental ($\bar{x} = 1.89$, $\sigma = 1.05$) and baseline ($\bar{x} = 3.00$, $\sigma = 0.87$) conditions, as seen in Table 8.5 (p. 142). Similarly to OVS7, these results support the hypothesis that the effort of changing and evolving existing requirements is lower in the experimental group.

OVS9 I found that the resulting application could be used in production-level environments with minimal or no change

Let $H_1 : G_e > G_b$, there was **significant** difference ($\rho = 0.029$) in the scores for the experimental ($\bar{x} = 2.44$, $\sigma = 0.88$) and baseline ($\bar{x} = 1.56$, $\sigma = 0.88$) conditions, as seen in Table 8.5 (p. 142). Despite no group had time to finish all tasks presented, this item measures the confidence that each group had to deliver the product *as-is*, or with minimal change. 67% of the participants in the baseline group strongly rejected the idea of deploying their implementation, compared to 22% in the experimental group. Still, even in the experimental group, no single participant agreed with this assertion, which could be explained by the experimental timespan available and overall experience with the framework.

8.3.4 Development Process

This group of questions intended to put the experiment in perspective with both previous subject experiences, and the main difficulties they had encountered.

DVP1.1 I found the development style of this setup suitable for using in the context of agile methodologies

Let $H_1 : G_e \neq G_b$, there was **significant** difference ($\rho = 0.022$) in the scores for the experimental ($\bar{x} = 4.33$, $\sigma = 0.71$) and baseline ($\bar{x} = 3.33$, $\sigma = 0.87$) conditions, as seen in Table 8.6 (p. 145). This item should be analyzed as potentially correlated with BG1.7, where a sig-

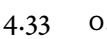
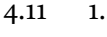
	EXPERIMENTAL			BASELINE			STATISTICS		
	12345	$\bar{x}$	σ	12345	$\bar{x}$	σ	H_1	W	ρ
DVP1.1		4.33	0.71		3.33	0.87	$\neq$	65.0	0.022
DVP1.2		2.89	0.78		2.89	0.60	$\neq$	40.0	1.000
DVP1.3		2.44	1.01		2.44	0.53	$\neq$	42.0	0.924
DVP1.4		4.11	1.27		2.78	1.39	$>$	63.0	0.022
DVP2.1		2.33	1.66		3.56	1.59	$<$	22.0	0.050
DVP2.2		2.00	1.22		4.22	0.67	$<$	05.0	0.001
DVP2.3		3.33	1.22		1.89	0.93	$<$	66.0	0.991

Table 8.6: Summary of Development Process results, including the values of the non-parametric significance Mann-Whitney-Wilcoxon test.

nificant statistical deviation among the two different treatments was observed. According to the results, the experimental group had a stronger background in agile methodologies than the baseline group. Similarly, in this item, the experimental group also had a stronger positive opinion regarding the use of this technology in agile setups. Because of this correlation, a conservative interpretation to the hypothesis that the experimental treatment is suitable for agile environments should be inconclusive, until further studies, and a possible meta-analysis, discard the correlation.

DVP1.2 I found the development style of this setup suitable for using in the context of classic methodologies

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 1.000$) in the scores for the experimental ($\bar{x} = 2.89$, $\sigma = 0.78$) and baseline ($\bar{x} = 2.89$, $\sigma = 0.60$) conditions, as seen in Table 8.6. Since there was also no statistical differences observed in item BG1.8, one cannot reject the null hypothesis that the two treatments are indifferent to the context of classic methodologies.

DVP1.3 I found the development style of this setup suitable for using in the context of formal methodologies

Let $H_1 : G_e \neq G_b$, there was **no significant** difference ($\rho = 0.924$) in the scores for the experimental ($\bar{x} = 2.44$, $\sigma = 1.01$) and baseline ($\bar{x} = 2.44$, $\sigma = 0.53$) conditions, as seen in Table 8.6. Since there was also no statistical differences observed in item BG1.9, one cannot reject the null hypothesis that the two treatments are indifferent to the context of formal methodologies. Still, when compared to items DVP1.1 and DVP1.2, the scores are slightly lower, as expected.

DVP1.4 I found the development style of this setup suitable for developing face to face with the client

Let $H_1 : G_e > G_b$, there was **significant** difference ($\rho = 0.022$) in the scores for the experimental ($\bar{x} = 4.11$, $\sigma = 1.27$) and baseline ($\bar{x} = 2.78$, $\sigma = 1.39$) conditions, as seen in Table 8.6 (p. 145). The score of this item support the hypothesis that the experimental treatment is more suitable for quick prototyping and addressing *real-time* requirements.

DVP2.1 Concerning this particular setup most of my difficulties were dealing with the Persistency Engine

Let $H_1 : G_e < G_b$, there was **significant** difference ($\rho = 0.050$) in the scores for the experimental ($\bar{x} = 2.33$, $\sigma = 1.66$) and baseline ($\bar{x} = 3.56$, $\sigma = 1.59$) conditions, as seen in Table 8.6 (p. 145). The score of this item support the hypothesis that the effort to deal with data persistency when requirements change would be significantly lower in the experimental setup.

DVP2.2 Concerning this particular setup most of my difficulties were building a Graphical User Interface

Let $H_1 : G_e < G_b$, there was **highly significant** difference ($\rho = 0.001$) in the scores for the experimental ($\bar{x} = 2.00$, $\sigma = 1.22$) and baseline ($\bar{x} = 4.22$, $\sigma = 0.67$) conditions, as seen in Table 8.6 (p. 145). Similarly to item DVP2.1, the score of this item support the hypothesis that the effort to deal with the graphical user-interface when requirements change would be significantly lower in the experimental setup.

DVP2.3 Concerning this particular setup most of my difficulties were implementing the core Business Logic

Let $H_1 : G_e < G_b$, there was **no significant** difference ($\rho = 0.991$) in the scores for the experimental ($\bar{x} = 3.33$, $\sigma = 1.22$) and baseline ($\bar{x} = 1.89$, $\sigma = 0.93$) conditions, as seen in Table 8.6 (p. 145). Contrary to expected, this scores do not display a significant statistic difference between the two groups when considering that the experimental group should be lower than the baseline group (it shows a statistical difference in the exact opposite direction). Probably this is due to correlation with items DVP2.1 and DVP2.2, where participants answered "proportionally" to those items (i.e., because the experimental group had low difficulties in the GUI and persistency, then, by elimination of choices, any difficulties they had would fall into the Business Logic). Hence, not only this score doesn't allow the rejection of the null hypothesis, it also points to a potential issue when designing Likert-scale questionnaires.

8.4 OBJECTIVE MEASUREMENT

The experiment had two live video feeds being recorded: (i) a standard camera pointing directly to both the participants and the whiteboard, and (ii) a *screencast* of the computer the participants were using. The initial intent was to measure time, correctness and complexity of the produced artifact in a non-intrusive manner. However, due to a design mistake, the participants were not properly informed they should strictly finalize each task independently, and as such they paralleled the implementation. This prevented an objective measurement of time expended for each task, since they tended to overlap.

Nevertheless, it is still possible to measure the final artifacts in terms of implemented requirements. Each task may be partitioned into three major concerns: (i) domain structure, (ii) persistency, and (iii) user-interface. A summary of the final artifacts is provided in Table 8.7, using the following grades: (A) perfect implementation, with user-interface, persistency and rules, (B) mostly implemented, with minimal issues, (C) poorly implemented, with incomplete user-interface, persistency and missing rules, (D) hardly implemented, with minimal or no user-interface and persistency, and (-) Not addressed.

	Construction				Evolution		
	T1	T2	T3	T4	E1	E2	E3
Baseline Group I	C	-	-	-	-	-	-
Baseline Group II	C	-	-	-	D	C	D
Baseline Group III	C	C	D	-	D	C	D
Experimental Group I	A	B	B	-	B	A	B
Experimental Group II	A	A	-	-	B	A	B
Experimental Group III	B	B	-	-	-	-	-

Table 8.7: Implemented requirements.

No group in the baseline treatment was able to produce an usable (even if minimal) user-interface, although they mostly tried to design it using the graphical tools provided by Eclipse. The same applies for persistency. All groups in both the baseline and experimental treatments ignored task 4. Most groups were also planning for the whole 3h to produce tasks 1 – 4, and thus the development was in very early stages when the evolution tasks were handed. Most groups never tried to re-address missing tasks once confronted with those related to the system's evolution.

8.5 VALIDATION THREATS

The outcome of validation is to gather enough scientific evidence to provide a sound interpretation of the scores. Validation threats are issues and scenarios that may distort that evidence and

thus incorrectly support (or discard) expected results. Each validation threat should be expected and addressed *a priori* in order to yield unbiased results:

- **Misunderstanding the given tasks.** Because the tasks relied on specifications given in textual form and supported by UML diagrams, it is necessary to ensure that the participants correctly interpret them. This threat is discarded by both the pre-test evaluation and the results from item BG1.10.
- **Unsuitable base skills to perform the tasks.** The tasks required participants to have the necessary skills to build and evolve information systems, namely knowing how to work with the given programming language, integrated development environment and database engine. Once again, this threat is discarded by both the pre-test evaluation and the results from items BG1.3, BG1.5, and BG1.6.
- **Overhead of necessary tools to perform the tasks.** Because the experiment was timed, it is necessary to ensure that participants focus on the tasks at hand. In order to discard this threat, the necessary setup was conducted before the experiment by the researchers.
- **Proficiency with the experimental treatment.** Although it was required that the baseline group had previous contact with the needed theory and tools, a conservative approach to the experimental treatment where no participant had previous contact with the framework (as seen in item BG1.2) allows to completely discard this threat.
- **Disturbance due to observation procedures.** Due to the nature of data gathering devices (i.e., video-cameras, microphones, and screencast software), participant's performance could be hindered by the feeling of being "judged" and observed. The results of item EF1 allow this threat to be discarded.
- **Disturbance due to social factors.** The fact that groups were formed randomly could lead to situations were some individuals would't work well collaboratively. This threat is discarded by the results of item EF3, although the experimental group exhibited higher scores.
- **Disturbance due to lack of motivation.** Due to the length of the tasks (3h in total), and the fact that there was no compensation to individuals participating in the experiment, the lack of motivation could hinder the outcome. This threat is discarded by the results of item EF2, although the experimental group also exhibited higher scores, consistent with item EF3.

The following threats were not completely discarded, and should be the focus of future studies:

- **Different skills regarding agile methodologies.** The potential correlation between items BG1.7 and DVP1.1, may suggest that because the experimental group had higher skills in agile methodologies, somehow it influenced the experimental results. While it is hard to see why such skills would make a significant difference in a setup were the specifications were not formalized by the participants, but given at the beginning of the experiment, one could argue that those with an agile background were skilled in the production of artifacts more suitable to change. Would this premise be accepted, then a new study where the tendency is changed (i.e., non-agile practitioners belonging to the experimental group), would provide more data to correctly interpret the observations. However, because no participant knew that the requirements would be the target of change, one can discard this threat with moderate confidence.

- **Biased responses in the background section due to post-test subjective perception.** The items measuring the background expertise were part of the post-test questionnaire. Although pre-test evaluation discard this item as an experimental threat, it doesn't eliminate possible correlations between the background and other scores, due to a possible biased subjective perception gained *a posteriori w.r.t.* the achieved results. In order to eliminate this threat, a new study should move the background section to a pre-test questionnaire.

- **Correlation among different items due to no alternate choice.** When participants are faced with a set of items that may be interpreted as a choice partition (e.g. items DVP2.1, DVP2.2, and DVP2.3), those items may become correlated or "relativized". A stronger issue can arise if the items, though disjoint, are not complete (i.e., there could be more possible choices), and would force a participant to choose among one of the alternatives. In future studies, adding an extra item that would "catch" all other unforeseen choices could diminish this potential threat.

The power of this study could also be improved by (i) increasing the number of participants, and (ii) switching the participant roles, where all individuals in the experimental group would be observed under the baseline treatment, and *vice-versa*.

8.6 CONCLUSION

This chapter detailed a quasi-experiment conducted within a controlled experimental environment using the framework developed in previous chapters. One of the goals was to provide evidence in areas that are open to validation threats inherent to case-studies analysis, as performed in chapter Chapter 7 (p. 115). The pre-test evaluation guaranteed no statistical deviation among the two treatments *w.r.t.* their background and basic skills.

The post-test questionnaire was used to assess the outcomes of each treatment. The final results supports the hypothesis that developing using the *Oghma* framework is more efficient,

both in terms of producing a system from scratch, as well as dealing with changing requirements, when compared to a traditional approach of using a general purpose language, and a relational database management system.

Some threats to this validation were identified and further discarded by analyzing the scores in the post-test questionnaire and due to the nature of the experimental setup. Not all original hypothesis were supported, though, and some borderline threats which emerged after the experiment can help to refine new studies.

Chapter 9

Conclusions

9.1 Summary of Hypotheses 151
9.2 Main Results 152
9.3 Future Work 153
9.4 Epilogue 156

In this dissertation, we started by looking at the recurrent issues of the software engineering field, specifically on two contingent factors: (i) that the field is in *crisis*, in terms of the ratio between successful and challenged/canceled projects, and (ii) that no *"silver bullet"* has yet been found. Behind this *status quo*, unreasonable expectations from the stakeholders and constant change of the requirements are to be blamed. And, as such, the mainstream focus has mainly been on coping with these issues through new software development methodologies. Yet, we decided to tackle the problem from a different angle. Inspired by the way agile methodologies looked upon the development process to *embrace change*, we hypothesized how software specifically synthesized to cope with *continuous change* would look like. And in search for this *form*, the architecture and design behind such systems, we have found the *adaptive object-model* architectural pattern.

9.1 SUMMARY OF HYPOTHESES

The author's fundamental research question was stated as:

> *What form should this type of systems take, and which kind of tools and infrastructures should be available to support the development of such software systems?*

And further decomposed into the following hypotheses, always in comparison with a traditional approach of using a generic purpose language:

- **H1:** A AOM framework would provide a more suitable infrastructure for developing *incomplete by design* systems.
- **H2:** Developers' focus would shift from implementation artifacts to domain objects.
- **H3:** Translation of conceptual specifications into final applications would be more straightforward.
- **H4:** The effort of adding or changing existing requirements would be lower.
- **H5:** Prototyped applications could be immediately, or with little changes, used in production-level environments.
- **H6:** The style of development would promote agile principles.
- **H7:** The style of development would be suitable to develop face-to-face with the client.

9.2 MAIN RESULTS

The following items summarize the main results obtained:

- **Contributions to the formalization of a pattern language for AOM.** Seven new patterns were formalized in Chapter 5 (p. 59), viz. (i) EVERYTHING IS A THING, which solves the problem of finding a unified representation for observing and manipulating data and metadata, (ii) CLOSING THE ROOF, which addresses the issue of having a possible unbound number of (meta-)meta-models, (iii) BOOTSTRAPPING, which is used to cope with definitions that depend on themselves, (iv) LAZY SEMANTICS, which is able to handle transiently inconsistent/undefined system states, (v) HISTORY OF OPERATIONS, intended to keep track of operations performed upon objects, without knowing their specific details, (vi) SYSTEM MEMENTO, which preserves the notion of system-wide evolution, and allows access to any arbitrary previous state of the system, and (vii) MIGRATION, which supports the evolution of a system while maintaining its integrity, through the composition of evolution rules.
- **The specification of a reference architecture for AOM frameworks.** A high-level architecture of a AOM framework was defined, identifying the main components and their relationships. Through the composition of patterns previously identified, it proposes solutions for design concerns, namely: (i) the necessary abstractions, (ii) the generic functionalities it should provide, (iii) a default, but modifiable, behavior, and (iv) the points of extension.
- **A reference implementation of such framework.** The implementation of an object-oriented framework targeted to the development of information systems whose structural requirements could be best described as *incomplete by design*, built upon the other contributions made in this dissertation.

Additionally, the results of validation procedures also add to the body of knowledge:

- **Study on the usage of AOM in industrial applications.** The framework proposed in this dissertation was validated using a time-series analysis performed during the SDLC of two case studies, viz. (i) *Locvs*, a medium-sized information system for architectural and archaeological heritage, and (ii) *Zephyr*, a small information system for document records management. The experience gained in building and deploying industrial-level applications based on the AOM meta-architecture, as well as construction of a AOM framework, is documented in Chapter 7 (p. 115) and represents *per-se* a contribution not yet found in the literature. This study provided evidence that all the above mentioned hypotheses hold.
- **Study on key benefits of AOM through (quasi-)experiments.** A quasi-experiment specifically designed to discard some validation threats inherent to the use of case-studies, also gathers empirical evidence of several claims on benefits and liabilities of using AOM. This experiment provided evidence that the hypothesis H1-H5 and H7 hold.

9.3 FUTURE WORK

In the questionnaire handed to the professionals that had contact with the framework, we asked about what would they value as new features. The summary of the answers is presented in Table 9.1. At the top of the most-wanted features is a graphical model editor — understandable since the auto-generated GUI currently only allows evolution to a certain extent, and the generated GUI for editing the model based on the meta-model is not very friendly. The next two most-wanted features is a HTML-based interface, and support for workflows and business process modeling.

Concerning new framework features, I would value…	12345	$\bar{x}$	σ
…an web-powered (HTML5) user-interface		4.29	0.76
…workflows and business process management modeling		4.29	0.95
…more hooks		3.71	0.95
…graphical model editor		4.57	0.79
…interoperability with third-party model tools		3.43	0.79
…extensive support for model refactoring		4.14	0.90

Table 9.1: New features results of industrial survey, each line representing the data of a single question, with corresponding means and standard deviation values, with $n = 7$.

The following items pose further research that could be pursued in line with this dissertation:

9.3.1 Evolving Oghma

Perhaps one of the greatest problems in pursuing empirical validation on software engineering artifacts is that they must *exist*. In this case, as pointed by Yoder *et al.* [Yod02], no known frame-

work for building AOM-based systems was published up until this point. If the original thesis statement in this dissertation was to prove an existential, i.e., $\exists x \forall y : \mathrm{IsFramework}(x) \wedge \mathrm{Build}(x, y) \wedge \mathrm{IsAOM}(y)$, then one could simply argue about the (im)possibility of existing such framework. But merely pointing out to its possible existence, or even drafting how it would look like, wouldn't allow empirical experimentation to be carried upon — one is required to *have* it. As such, a considerable amount of *time* and *effort* was dedicated to actually *build* such framework, up to a point where it could be used in industrial-level applications[1]. Right now, the framework slightly exceeds 50k LOC, developed in a timespan of almost three years, as seen in Figure 9.1.

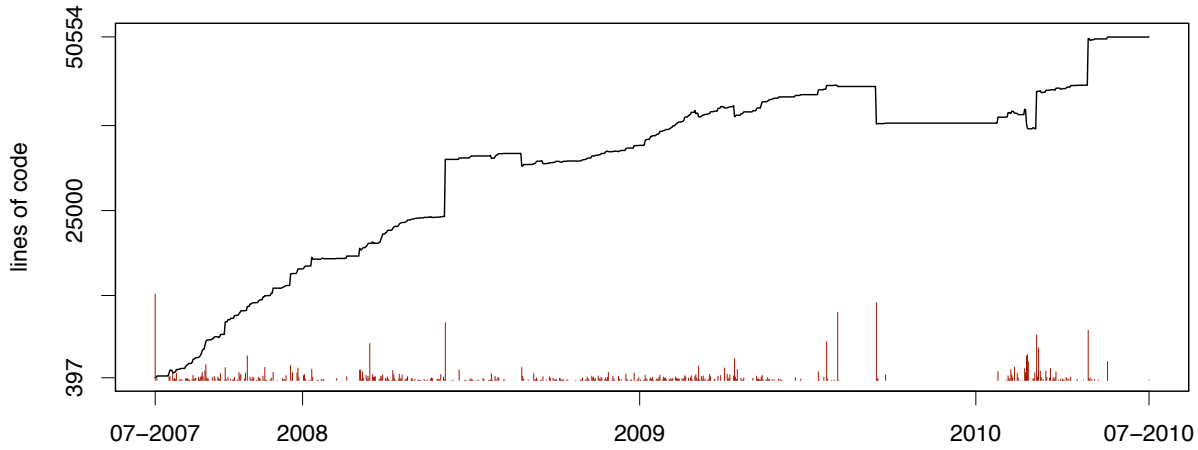


Figure 9.1: Evolution of the *Oghma* implementation of, showing the project code-base between July 2007 and July 2010, measured in lines of code. The vertical lines display the cumulative size of the differences per day.

It shouldn't be surprising that the architecture and design presented in Chapter 6 (p. 93) already points to solutions that are not yet present in the current implementation of the framework, e.g., branching and reconciliation of data and meta-data by the end-user. At the time of this dissertation, the latest development builds of the framework have been used as a starting project for 4th year students to learn, explore and develop on top of large code bases on the course *Software Development Laboratory*[2] [FCR10].

9.3.2 Web-based Adaptive User Interfaces

Up until now, only a small amount of work regarding adaptive systems on the web can be found, though several websites offer varying degrees of *customization*, such as repositioning content on the webpage, adding new content[3], or changing the overall look of the application[4] [Goo, Pag].

[1] Which poses another problem *per-se*, since there is a plethora of concerns that must be addressed and implemented, and which does not contribute to the state-of-the-art.
[2] Master in Informatics and Computing Engineering, from the University of Porto, Faculty of Engineering.
[3] Usually small applications known as *widgets*.
[4] Commonly known as *skinning* or *theming*.

There are also some examples of *adaptability* to each individual user. This is accomplished by mining information relative to where the user comes from and its past actions in order to cater to the perceived results the user expected [GGGR09]. As already mentioned in this dissertation, the fundamental property of adaptive software is to allow end-users to introduce changes in its underlying structure and behavior. However, these websites do not allow their end-users to modify the underlying model of the system; only the overall look&feel and, to some extent, what information is displayed on the page — thus still relying on developers to deal with structural and behavioral rules. As such, these systems are very different from those discussed in this work.

Nevertheless, such websites possess valuable information regarding the most common user-interface mechanisms. Drag-and-drop is used extensively to reorganize information and perform basic tasks such as adding and removing basic container blocks. These mechanisms and interaction paradigms can be used to modify any kind of system from an end-user perspective, be it a AOM system or not. This study is currently being pursued by João Gradim as his master's thesis [Gra10].

9.3.3 Improving Usability of Automatically Generated GUI

While automatic run-time generated user-interfaces may not be on pair with custom-made ones regarding usability, they seem to be consistent and based on a strict set of metaphors, supporting a quick learning process by users. Nonetheless, which mechanisms should the framework provide to improve usability and customization of GUI, while retaining the capability of automatically generating them? This study is currently being pursued by André Carmo as his master's thesis.

9.3.4 Continuing the Pattern Language

The study on automatically generating GUI, and allowing non-programmer end-users to efficiently cope with domain changes may provide new patterns to add to the pattern language. And despite the seven patterns here presented, the pattern language as a whole is not yet finished, probably requiring a multi-year, multi-person effort, with constant feedback from the patterns community.

9.3.5 Self-Hosting

Self-hosting is the capability of a programming language to be built on top of itself. Few programming languages actually exploit this characteristic, with LISP probably being the most well known [HL95]. In object-oriented programming, since the creation of Smalltalk [Kay93], most modern languages are not designed with self-hosting in mind; instead, this feature is delegated to non-mainstream projects, such as PyPy [Pyp] for Python and Rubinius [Rub] for Ruby.

Likewise, *Oghma* is currently not self-hosted (though it is self-compliant). It would be interesting to see the advantages and disadvantages[5] of attaining such goal.

9.4 EPILOGUE

At the end of a period of three years, it is inevitable to take a look back and ponder. When we start a Ph.D., we are told that we must have novel ideas, that we must publish good results, and that we must conduct science. Perhaps the biggest misunderstanding comes from what it is actually meant for *novel*, *results* and *science*.

At the beginning, it is usual to try to imagine something that no-one has ever done before. Something we believe could solve most of the problems in our area. Then realize that most ideas we have had already been thought of…probably years ago. This is why one is required to do an extensive review of the literature: to learn about all those previous ideas[6] and try focus on the specific problems yet to be solved. But how exactly are we supposed to build on top of those ideas? Perhaps in a branch such as mathematics[7], one could point to other's proofs and theories and then move to our own. But in engineering, as the branch of science concerned with the design, building, and usage of structures, one must *have* those previous structures. And when they aren't readily available, we are faced with a dilemma: either we restrict to build prototypes — proof of concepts — and leave the burden of integration for someone else, or we must get our hands "dirty" and start from scratch. Looking back, I cannot be absolutely sure if the former was the right choice, given the available timespan, but it surely was the best way to make this work usable in settings beyond the academy. *Novel* should thus be regarded as *advancing* the state-of-the-art, taking care to *stand on the shoulders of giants*.

Then comes the problem of publishing results. Much has been said about the reason and importance of publishing, and from which I refrain to make comments (considering my short experience as a researcher). But perhaps the intended message may be found by observing the innovative revision process of the conferences on PATTERN LANGUAGES OF PROGRAMS (PLOP): each paper has a shepherd — a non-anonymous reviewer — that leads the author in a series of iterations to improve the paper's *content*, before being accepted or rejected. Even if accepted, the content is further discussed during the conference by a group of peers in a writer's workshop. Only after that unusually long cycle of revisions is the paper considered ready for digital publication. *Iterative* and *constant feedback* are important keywords here, not surprisingly mimicking (or being mimicked by) the agile principles. Presenting results becomes the product of an incremental process of research; snapshots in time. The same can be said about the artifacts

[5] The turing tarpit [Per82] being the most likely one.
[6] Ideally, both good and bad ideas, though researchers are pressed to only publish the good ones, and thus making everyone else recurrently fall into the bad ones.
[7] Not to disregard mathematicians, to which I have the utmost respect.

here described; for a dissertation inspired on the idea of *incompleteness*, it is, by itself, a mere *photography* of an ever-evolving structure.

And finally, there is science. It took me a short amount of time to realize that works relying on mathematics were more inclined to be coined as *scientific*, but a long time to understand that it is neither about the numbers nor the formulas. The key to make science is to understand what epistemology is, and to make the same questions that epistemology does: what is knowledge? how is knowledge acquired? how do we know what we know? Once we understand that by *science* we mean the process of acquiring and testing new knowledge, then, in the words of Richard Feynman, *"the first principle is that you must not fool yourself, and you are the easiest person to fool."* ▪

Appendices

Appendix A

Pre-Experiment Data

	I	II	III	IV	V	VI	VII	$\bar{x}$	σ
Subject A	10	12	12	13	13	16	17	13.3	2.4
Subject B	15	16	12	12	11	18	19	14.7	3.1
Subject C	14	12	13	11	12	16	-	13.0	1.8
Subject D	15	16	16	16	16	17	18	16.3	1.0
Subject E	17	13	16	18	13	16	17	15.7	2.0
Subject F	10	17	13	16	15	17	18	15.1	2.8
Subject G	15	12	11	12	10	16	16	13.1	2.5
Subject H	12	13	11	13	12	14	16	13.0	1.6
Subject I	11	13	10	10	10	18	16	12.6	3.3

Table A.1: Student grades for the Experimental group. Each column represents the following courses: (I) Programming Fundamentals, (II) Programming, (III) Algorithms and Data Structures, (IV) Algorithm Design and Analysis, (V) Software Engineering, (VI) Software Development Laboratory, and (VII) Information Systems.

	I	II	III	IV	V	VI	VII	$\bar{x}$	σ
Subject J	14	10	14	15	16	15	17	14.4	2.2
Subject K	10	11	11	16	12	16	17	13.3	2.9
Subject L	13	10	13	12	12	16	17	13.3	2.4
Subject M	16	16	13	12	12	17	18	14.9	2.5
Subject N	10	10	13	14	12	15	17	13.0	2.6
Subject O	11	12	13	15	15	16	18	14.3	2.4
Subject P	-	-	-	-	-	-	-	-	-
Subject Q	-	-	-	-	-	-	-	-	-
Subject R	-	-	-	-	-	-	-	-	-

Table A.2: Student grades for the Baseline group. Each column represents the same courses in Table A.1. The last three subjects did not had this information publicly available at the time of the experiment.

Appendix B

Post-Experiment Questionnaire

The following is a copy of the anonymous questionnaire handed to the subjects after the end of the experiment.

Empirical Studies in Software Engineering

TENFOGS01

May - June 2010

Post-test Questionnaire

Thank you for participating in this experiment. We now ask you to take a deep breath, have a coke, and try to answer this brief questionary that won't take you more than 5 minutes.

Each question relates to issues regarding your profile as a developer and your perception about the experiment. The questionary is divided into sections with questions. Each question has an identifier (for easy processing later on) and may have either a single answer, or a list of possible answers. Each answer should be rated as follows: **1 (Strongly Disagree), 2 (Somewhat Disagree), 3 (Neither Agree nor Disagree), 4 (Somewhat Agree), 5 (Strongly Agree)**. Your should rate with an '**X**' every answer as best it resembles your opinion as possible.

Questionnaire

Background

BG1. I have considerable experience...

	1	2	3	4	5
...using frameworks.					
...with this particular framework.					
...with this particular programming language.					
...developing industrial-level applications.					
...analyzing and specifying information systems.					
...with object-oriented architecture, design and implementation.					
...with agile development methodologies.					
...with classical development methodologies.					
...with formal development methodologies.					
...with UML class diagrams.					

External Factors

	1	2	3	4	5
EF1. I felt disturbed and observed by the cameras.					
EF2. I enjoyed programming in the experiment.					
EF3. I would work with my partners again.					

Overall Satisfaction

	1	2	3	4	5
OVS1. Overall, this particular setup was suitable for solving every task presented.					
OVS2. I found myself thinking more about the end system purely in terms of the structure of domain objects.					
OVS3. I found myself thinking more about the end system more in terms of the structure of the database and user interaction.					
OVS4. I found very difficult to directly translate specifications into final artifacts.					
OVS5. I was able to create the underlying object model at least as rapidly as we could normally have created a specification.					
OVS6. I was able to create the user interface at least as rapidly as it could normally have been prototyped.					
OVS7. Additional requirements represented considerable added effort.					
OVS8. Change of existing requirements represented considerable added effort.					
OVS9. I found that the resulting application could be used in production-level environments with minimal or no change.					

Development Process

DVP1. I found the development style of this setup suitable for...

	1	2	3	4	5
... using in the context of agile methodologies.					
... using in the context of classic methodologies.					
... using in the context of formal methodologies.					
... developing face to face with the client.					

DVP2. Concerning this particular setup, most of your difficulties where...

	1	2	3	4	5
... dealing with the Persistency Engine.					
... building a Graphical User Interface.					
... implementing the core Business Logic.					

The Future

FT1. Given that the basic infrastructure now exists, and with suitable modifications to the development process, my expectations are that subsequent systems developed with it would be...

	1	2	3	4	5
Developed faster when compared to a conventional approach.					
Less expensive than using a more conventional approach.					
More comprehensively tested than would using a more conventional approach.					
More easy to maintain overall consistency.					

Learning Support

LS1. Consider all the steps you took to solve a particular task. You have a couple of minutes to quickly share your knowledge of how to solve that problem to other developers, or even to yourself if, later, you need to come back to it. You would...

	1	2	3	4	5
Point out the correct sequence of steps that lead to the solution.					
Point out the superfluous and "dead-end" paths to warn of the hazard.					
Tell them what you thought the solution could be.					
Tell them only the classes they need to look at. The rest is up to them.					
The starting point you took. They eventually will dig out the rest.					

LS2. If you were asking for help on how to reach a solution to a particular problem, and the developer assisting you only had 30 seconds to answer (very busy person), what would you like to hear from him/her...

	1	2	3	4	5
The minimal steps he/she took to reach the solution.					
All the steps (including the "dead-ends") he/she took to reach the solution.					
The starting-point and a direction would be sufficient.					
The fundamental step that makes you "click" (Ahá!! Got it!!).					
What the previous developer thought of when trying to solve the problem.					
The final class diagram.					

LS3. During the process of solving the tasks, when do you think an expert's hint would benefit you the most (imagine you only have one shot at the expert)?

	1	2	3	4	5
Right at the start.					
Figuring out how concepts (classes) relate.					
When writing code.					
Testing.					

	1	2	3	4	5
LS4. I would be more confident using solving tips coming from other developers than from the inline help system.					
LS5. I would be more productive if the system pointed to potential errors instead of having to rely on the documentation.					

LS6. If you had to continue developing with this setup, you would value...

	1	2	3	4	5
... having extensive reference documentation.					
... having written walk-throughs and tutorials.					
... having screencasts/videos.					
... having more example implementations.					

If you wish to leave any further comments, please use the following space:

Thank you for your time.

Appendix C

Post-Experiment Questionnaire Results

	Experimental								$\bar{x}$	σ	Baseline								$\bar{x}$	σ	H_1	t-test	W	$\rho \neq$	$\rho <$	$\rho >$
BG1.1	4	3	3	3	3	3	4	4	4	3.44	0.53	4	3	4	3	3	2	4	3	5	3.44	0.88	$\neq$	0.500	41.0	1.000	0.539	0.500
BG1.2	1	1	2	1	1	1	1	1	1	1.11	0.33	4	4	4	1	2	2	5	3	3	3.11	1.27	$<$	0.000	06.0	0.001	0.001	1.000
BG1.3	5	3	4	4	4	2	2	2	4	3.33	1.12	5	4	4	5	4	4	4	4	5	4.33	0.50	$\neq$	0.013	19.5	0.045	0.022	0.982
BG1.4	3	4	3	2	3	2	2	2	2	2.56	0.73	2	2	3	2	2	2	4	4	4	2.78	0.97	$\neq$	0.295	36.5	0.731	0.366	0.671
BG1.5	2	5	4	2	3	3	3	3	3	3.11	0.93	3	3	3	4	1	4	3	2	4	3.00	1.00	$\neq$	0.405	39.5	0.962	0.481	0.557
BG1.6	4	5	4	4	3	4	5	5	5	4.33	0.71	2	4	2	4	3	4	4	4	4	3.44	0.88	$\neq$	0.016	62.5	0.036	0.986	0.018
BG1.7	4	4	4	4	2	4	4	4	3	3.67	0.71	3	3	4	4	3	2	3	3	1	2.89	0.93	$\neq$	0.031	62.0	0.043	0.983	0.021
BG1.8	4	5	4	3	3	4	4	4	4	3.89	0.60	3	3	3	4	4	4	4	3	5	3.67	0.71	$\neq$	0.241	48.5	0.457	0.800	0.229
BG1.9	3	4	4	3	2	3	2	2	2	2.78	0.83	2	3	2	3	2	2	3	3	4	2.67	0.71	$\neq$	0.382	43.0	0.848	0.613	0.424
BG1.10	2	5	4	4	3	4	5	5	5	4.11	1.05	3	3	4	4	4	4	3	2	4	3.44	0.73	$\neq$	0.069	58.5	0.102	0.958	0.051
EF1	1	1	1	2	2	2	3	3	2	1.89	0.78	1	1	1	2	1	1	3	1	5	1.78	1.39	$\neq$	0.419	50.0	0.389	0.831	0.195
EF2	5	5	5	3	4	4	5	4	5	4.44	0.73	3	2	5	3	5	2	3	4	3	3.33	1.12	$\neq$	0.012	63.5	0.037	0.985	0.019
EF3	5	5	4	5	4	4	5	5	5	4.67	0.50	5	3	4	3	5	4	4	4	3	3.89	0.78	$\neq$	0.011	63.0	0.035	0.986	0.018
OVS1	3	4	3	4	4	4	3	4	4	3.67	0.50	2	2	3	2	3	2	2	4	2	2.44	0.73	$>$	0.000	72.0	0.004	0.999	0.002
OVS2	1	3	3	3	4	3	4	3	3	3.00	0.87	4	4	4	3	4	3	4	3	4	3.67	0.50	$>$	0.031	21.0	0.059	0.030	0.977
OVS3	1	2	3	4	3	4	3	3	3	2.89	0.93	4	3	3	4	3	3	2	2	2	2.89	0.78	$<$	0.500	42.5	0.886	0.594	0.443
OVS4	3	2	3	3	3	2	1	1	2	2.22	0.83	3	3	4	2	3	2	3	2	4	2.89	0.78	$<$	0.050	24.5	0.143	0.072	0.940
OVS5	4	3	2	4	4	4	3	3	4	3.44	0.73	2	3	3	4	4	3	3	4	3	3.22	0.67	$>$	0.254	48.5	0.466	0.796	0.233
OVS6	1	3	3	4	2	5	5	5	5	3.67	1.50	1	1	1	2	1	2	1	2	1	1.33	0.50	$>$	0.000	73.5	0.003	0.999	0.001
OVS7	1	1	4	1	3	2	1	1	1	1.67	1.12	2	2	2	3	4	2	3	4	3	2.78	0.83	$<$	0.015	15.5	0.025	0.012	0.990
OVS8	1	1	4	2	3	2	1	1	2	1.89	1.05	2	3	2	4	4	2	4	3	3	3.00	0.87	$<$	0.013	16.5	0.032	0.016	0.987
OVS9	3	3	3	2	1	1	3	3	3	2.44	0.88	1	1	1	1	2	3	1	3	1	1.56	0.88	$>$	0.024	60.5	0.058	0.977	0.029
DVP1.1	5	5	4	4	4	3	5	4	5	4.33	0.71	3	4	3	4	4	2	2	4	4	3.33	0.87	$\neq$	0.008	65.0	0.022	0.991	0.011
DVP1.2	3	3	2	3	4	4	2	3	2	2.89	0.78	3	3	3	3	2	3	4	2	3	2.89	0.60	$\neq$	0.500	40.0	1.000	0.500	0.539
DVP1.3	1	3	3	3	3	4	1	2	2	2.44	1.01	3	3	3	2	2	2	3	2	2	2.44	0.53	$\neq$	0.500	42.0	0.924	0.576	0.462
DVP1.4	1	4	5	5	4	4	5	4	5	4.11	1.27	3	5	3	4	2	2	1	4	1	2.78	1.39	$>$	0.025	63.0	0.045	0.982	0.022
DVP2.1	1	5	4	1	4	3	1	1	1	2.33	1.66	3	5	5	2	1	2	5	4	5	3.56	1.59	$<$	0.065	22.0	0.101	0.050	0.959
DVP2.2	3	3	4	1	1	3	1	1	1	2.00	1.22	5	4	4	4	5	4	3	4	5	4.22	0.67	$<$	0.000	05.0	0.001	0.001	1.000
DVP2.3	2	2	3	5	4	4	5	3	2	3.33	1.22	3	2	1	1	3	1	1	2	3	1.89	0.93	$<$	0.006	66.0	0.023	0.991	0.012
FT1.1	5	5	4	4	4	5	5	5	4	4.56	0.53	5	5	5	3	3	4	4	4	1	3.78	1.30	$>$	0.058	55.5	0.166	0.931	0.083
FT1.2	5	5	4	4	4	4	4	4	3	4.11	0.60	5	3	4	3	3	3	4	4	1	3.33	1.12	$>$	0.042	59.0	0.085	0.966	0.042
FT1.3	3	4	4	3	3	1	3	2	2	2.78	0.97	5	3	3	4	4	3	4	3	1	3.33	1.12	$>$	0.139	27.5	0.242	0.121	0.897
FT1.4	3	5	4	3	3	3	3	3	3	3.33	0.71	5	4	4	4	2	2	4	3	1	3.22	1.30	$>$	0.412	39.0	0.926	0.463	0.574
LS1.1	5	4	4	4	3	3	1	3	3	3.33	1.12	4	4	3	3	2	2	3	4	3	3.11	0.78	$\neq$	0.316	47.5	0.540	0.760	0.270
LS1.2	5	3	4	3	4	4	5	4	4	4.00	0.71	3	5	3	3	4	3	4	4	3	3.56	0.73	$\neq$	0.103	54.5	0.196	0.917	0.098
LS1.3	3	3	4	4	4	4	3	3	3	3.44	0.53	4	3	4	5	2	3	3	3	4	3.44	0.88	$\neq$	0.500	41.0	1.000	0.539	0.500
LS1.4	1	4	4	2	2	2	3	4	2	2.67	1.12	3	1	3	2	2	2	2	2	3	2.22	0.67	$\neq$	0.160	49.0	0.446	0.804	0.223
LS1.5	5	4	4	3	3	2	3	3	5	3.56	1.01	4	2	3	4	3	3	3	2	4	3.11	0.78	$\neq$	0.157	50.0	0.399	0.826	0.200
LS2.1	5	3	4	2	4	4	5	4	4	3.89	0.93	5	4	5	4	4	2	3	2	3	3.56	1.13	$\neq$	0.252	47.5	0.544	0.758	0.272
LS2.2	1	2	3	2	4	1	1	2	3	2.11	1.05	4	4	4	3	3	3	2	1	3	3.00	1.00	$\neq$	0.043	21.5	0.091	0.045	0.963
LS2.3	2	4	3	4	2	4	3	5	5	3.56	1.13	3	3	4	5	3	3	4	4	3	3.56	0.73	$\neq$	0.500	41.5	0.963	0.556	0.481
LS2.4	5	5	4	5	4	4	5	4	1	4.11	1.27	5	4	5	4	2	4	4	4	5	4.11	0.93	$\neq$	0.500	44.0	0.770	0.652	0.385
LS2.5	1	2	3	3	2	4	1	3	2	2.33	1.00	2	5	1	2	4	3	2	1	1	2.33	1.41	$\neq$	0.500	43.5	0.819	0.626	0.410
LS2.6	3	5	4	4	2	3	3	2	2	3.11	1.05	1	4	4	3	3	4	3	3	4	3.22	0.97	$\neq$	0.410	35.0	0.642	0.321	0.712
LS3.1	3	5	4	3	2	5	4	5	5	4.00	1.12	2	5	4	4	4	2	4	3	4	3.56	1.01	$\neq$	0.195	51.0	0.356	0.845	0.178
LS3.2	4	3	5	4	2	3	5	4	3	3.67	1.00	3	5	4	2	3	2	3	4	5	3.44	1.13	$\neq$	0.332	45.5	0.680	0.693	0.340
LS3.3	5	2	3	2	4	3	3	3	1	2.89	1.17	4	2	1	3	4	4	3	2	2	2.78	1.09	$\neq$	0.419	42.0	0.927	0.573	0.464
LS3.4	1	2	3	1	4	4	1	3	2	2.33	1.22	3	2	1	4	5	4	3	2	4	3.11	1.27	$\neq$	0.102	26.5	0.221	0.111	0.906
LS4	5	4	3	4	4	4	5	5	4	4.22	0.67	4	3	4	3	3	3	3	4	3	3.33	0.50	$\neq$	0.003	67.5	0.011	0.996	0.006
LS5	5	3	3	4	4	5	5	4	4	4.11	0.78	5	2	3	4	4	3	3	3	3	3.33	0.87	$\neq$	0.031	60.5	0.069	0.972	0.034
LS6.1	4	2	4	4	4	4	5	4	5	4.00	0.87	2	3	4	4	4	2	2	2	3	2.89	0.93	$\neq$	0.009	65.0	0.022	0.992	0.011
LS6.2	2	4	4	5	4	4	3	4	3	3.67	0.87	2	4	4	5	2	3	2	2	5	3.22	1.30	$\neq$	0.203	49.0	0.460	0.797	0.230
LS6.3	5	3	4	3	3	4	1	3	2	3.11	1.17	2	4	1	3	2	2	4	4	5	3.00	1.32	$\neq$	0.426	42.5	0.892	0.590	0.446
LS6.4	5	5	4	4	4	4	5	4	4	4.33	0.50	5	3	5	4	4	4	5	4	3	4.11	0.78	$\neq$	0.241	46.5	0.585	0.740	0.293

Table C.1: Post-experiment questionnaire results with corresponding means, standard deviation, t-test probability values and the non-parametric significance Mann-Whitney-Wilcoxon test; see § 8.3 (p. 137).

Appendix D

Experimental Group Documentation

The following is a copy of the available documentation handled to the experimental group during the course of the experiment. It should not be regarded as the complete documentation of the framework, since it contains deliberate omissions and inconsistencies for the purpose of the experiment. For a comprehensive description see Chapter 6 (p. 93).

Oghma Documentation

v2.0 alpha

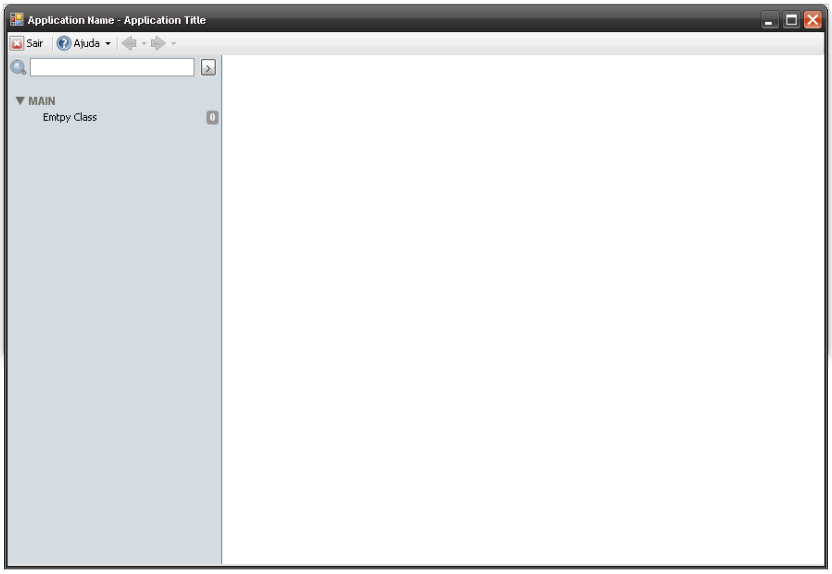
Introduction

This document provides the basic documentation for the Oghma framework.

Getting the project to compile

The following steps should get an empty Oghma project to compile with Visual Studio 2008:

- Download and unpack the source code;
- Open the main solution by double-clicking on oghma.sln inside your uncompressed folder;
- In the Solution Explorer, right-click on the project 'Template' and set as the 'StartUp Project';
- In the Build menu, select Rebuild Solution. Use F5 to run (with the debugger attached);
- You should now be able to see the application running, with a single package (Main) and an Empty Class.



Exploring the Model of a Running Example

Inside the main solution, you'll find MedSystem, a real-world use-case of Oghma. This project uses the filename medsystem.xml for its initial system definition. These configurations can be found in 'program.cs'. Due to a bug in Visual Studio, you've to right-click in 'program.cs' and choose 'View Code'

The Anatomy of OghmaML

OghmaML is a XML vocabulary to define the initial system. Its concepts are close to those of a class diagram. The basic structure of an OghmaML file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<model>
  <data>
    <!-- Packages -->
    <!-- Enums -->
    <!-- Entities -->
    <!-- Relationships -->
  </data>
  <views>
    <!-- Views -->
  </views>
</model>
```

Packages

A package represents an aggregation of concepts (namely, entities), and is also used to define which entities are considered entry points. Example:

```
<package id="main" name="Clinic">
  <entity id="patient" entrypoint="true" />
  <entity id="appointment" entrypoint="true" />
</package>
```

Packages and entry points are shown on the left menu bar of the application:

Entities

An entity is one of the main objects in the system. Its definition is close to that of a *Class*. It has properties – attributes and relationships. The following example shows the skeleton of an entity:

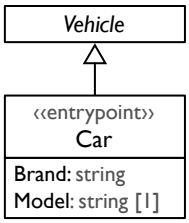
```
<entity id="" name="" inherits="" abstract="" tostring="">
  <list columns="" />
  <attr />
  ...
</entity>
```

The Entity tag has several attributes:

- `id`: Unique identifier for the entity. Mandatory, and should not contain numerals.
- `name`: `String` defining the label text (e.g. display name) for this object.
- `inherits`: indicates the ID of a class from which this class is derived.
- `abstract`: Flagged with `true` or `false`, it indicates if this entity is abstract (and hence not able to be instantiated). Optional field, `false` by default.
- `tostring`: Expression that determines how this entity is shown textually. The attributes of the entity and their ancestors can be part of the expression, and are identified by their `id` and the use of brackets (e.g. `'My name is {name} and I'm {age} yrs old'`)

The contents of an Entity tag can only have one List declaration and may enclose several Attribute tags (see Attributes).

Examples:




```
<entity id="car" name="Car" inherits="vehicle" tostring="{brand}">
  <list columns="{brand}|{name}" />
  <attr id="carbrand" name="Brand" domain="string" />
  <attr id="carmodel" name="Model" domain="string" cardinality="1"/>
</entity>
```

Base Domains

The following domains can be used in attributes:

- string: is a sequence of characters and is drawn as a text-field.
- text: is a (big) sequence of characters and is drawn as a text-area.
- float: is a numeral that accepts decimals and is drawn as a text-field.
- integer: is a natural number and is drawn as a text-field.
- boolean: is a boolean (true/false) and is drawn as a checkbox.

Attributes

An entity is typically composed by several attributes:

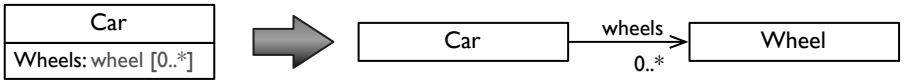
```
<attr id="" name="" domain="" cardinality="" role="" tostring="" isreadonly="" />
```

- `id`: Unique identifier for the attribute. Mandatory, and should not contain numerals.
- `name`: `String` defining the label text (e.g. display name) for this object.
- `domain`: Field defining a data type associated with this field. Different data types have different visual representations in Oghma (see Data Types).

- `cardinality`: Defines the cardinality of this side of the relation. Cardinality is indicated by a lower and upper bound (`lower..upper`). Examples: `0..*`,`1..*`, `1` or `0..2`.
- `role`: Optional field specifying if this attribute is either a `composer` or an `agreggator`.
- `tostring`: Expression that determines how this attribute is shown textually. The attribute itself can be part of the expression, identified by its `id` and the use of brackets.
- `isReadOnly`: Optional field that specifies if this attribute should be read-only. Accepts `true` or `false`. If an attribute is read-only, that means this attribute can only be one instance that has already been created.

See Entity for examples.

NOTE: When the domain of an attribute is another entity (instead of a base-type), oghma automatically transforms the attribute into a relationship. Example:



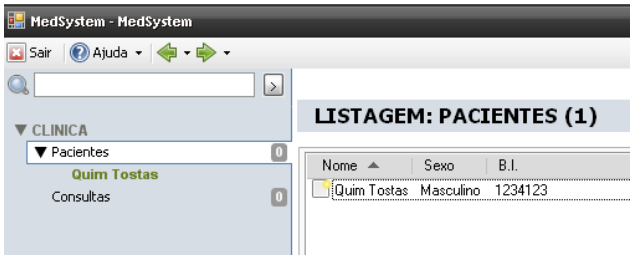
Listing

Listing refers to how several instances of an entity are presented on a table or list. It is an optional element within the `entity` element:

```
<list columns="" />
```

- `columns`: Defines which columns should appear when representing an entity in a list or table. The attributes to show can be identified by its unique `id` between brackets (e.g. `{name}`). Columns should be separated by using a pipe character (e.g. `{name}|{age}`).

See Entity for examples.



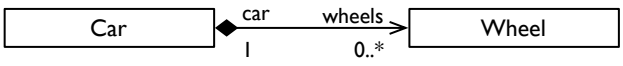
Relationships

Defines a relation between two entities. A relation contains two `node` elements (one for each end of the relation) that aim to define its scope.

```
<relationship id="" entity="" >
   <node entity="" id="" name="" cardinality="" navigable="" role="" />
   <node entity="" id="" name="" cardinality="" navigable="" role="" />
</relationship>
```

- `id`: Unique identifier for the node. Mandatory, and should not contain numerals.
- `entity`: Optional field with the ID of the associative entity for this relationship.
- `name`: `String` defining the label text for this object.
- `cardinality`: Defines the cardinality of this side of the relation. Cardinality is indicated by a lower and upper bound (`lower..upper`). Examples: `0..*`,`1..*`, `1` or `0..2`.
- `navigable`: Indicates navigability to this side of the relation. Accepted values are `true`, `false` and `unspecified`. `true` means that the node can be navigated to; `false` means the relation cannot be accessed thru this node. `unspecified` does not imply any type of restrictions to navigability and assumes the default value.

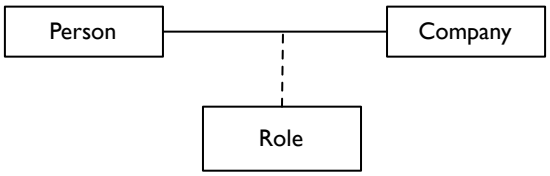
Examples:




```
<relationship id="car_wheel">
   <node entity="car" id="wheel" name="Wheels" cardinality="1" />
   <node entity="wheel" id="car" name="Car" cardinality="0..*" />
</relationship>
```

NOTE: Cardinality should be expressed in reverse!!! This is a known issue.

For associative entities, use the entity attribute in relationship:

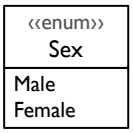



```
<entity id="role_id" name="Role" ...>
  ...
</entity>

<relationship id="person_company" entity=”role_id”>
   <node entity="person" ... />
   <node entity="company" ... />
</relationship>
```

Enumerations

An enumeration is simply an entity that inherits from 'enum'. Defining the initial values of that enumeration is currently not supported, though they can be edited in runtime.




```
<entity id="sex" name="Sex" inherits="enum" />
```

An enumeration is displayed as a combo-box when the upper-bound of the attribute 1, and as a check-list if the upper-bound greater than 1 (e.g. 0..*)

Views

If an entity doesn't have a view specified in the XML file, the system will automatically generate a default one. For examples on usage of custom views, see 'medsystem.xml'.

When everything goes wrong

Don't panic... This is an alpha version, and as such it has some bugs. Probably, the most problematic issue is that if the XML file is ill-defined, the debugger will stop in the thrown exception instead of showing an error message. When this happens, double-check your initial model definition for errors, like mismatching identifiers, redundant XML tags, etc.

If data becomes inconsistent, you can always delete both the database and the full-text search files. These are normally stored as 'database.db' and 'fts.db' when using the SQLiteDataSource provider. When running from Visual Studio, these are present in the debug folders '/bin/debug' or '/x86/bin/debug' of the running project (e.g. '/examples/medsystem/medsystem').

Appendix E

Industrial Survey

	ANSWERS							12345	$\bar{x}$	σ
BG1	4	3	4	4	5	4	1		3.57	1.27
BG2	3	4	3	5	4	4	1		3.43	1.27
BG3	5	4	4	4	5	4	2		4.00	1.00
BG4	3	3	4	4	4	2	5		3.57	0.98
BG5	5	3	5	4	5	1	5		4.00	1.53
BG6	3	4	4	4	5	2	5		3.86	1.07
BG7	2	4	4	3	4	2	5		3.43	1.13
BG8	4	4	4	4	5	3	4		4.00	0.58
BG9	2	4	4	3	4	3	2		3.14	0.90
BG10	4	4	3	3	4	3	2		3.29	0.76
BG11	4	2	1	1	2	1	2		1.86	1.07
BG12	1	3	4	4	4	2	4		3.14	1.21
OVS1	3	4	4	4	4	4	4		3.86	0.38
OVS2	3	4	4	5	5	4	3		4.00	0.82
OVS3	3	3	2	2	2	1	2		2.14	0.69
OVS4	3	5	4	4	4	5	3		4.00	0.82
OVS5	3	5	4	5	4	5	3		4.14	0.90
OVS6	3	2	3	2	1	1	1		1.86	0.90
OVS7	3	2	3	1	1	1	2		1.86	0.90
OVS8	3	4	4	5	4	5	5		4.29	0.76
DVP1.1	3	5	5	5	5	5	3		4.43	0.98
DVP1.2	3	4	5	4	4	3	3		3.71	0.76
DVP1.3	3	3	5	3	3	3	3		3.29	0.76
DVP1.4	3	3	4	5	4	5	3		3.86	0.90
DVP2.1	3	2	2	1	2	2	3		2.14	0.69
DVP2.2	3	2	4	1	2	2	3		2.43	0.98
DVP2.3	3	5	3	2	4	4	3		3.43	0.98
DVP2.4	3	2	3	2	4	2	3		2.71	0.76
DVP2.5	3	4	4	2	4	3	3		3.29	0.76
DVP2.6	3	4	5	2	4	4	3		3.57	0.98
FT1	3	5	4	5	5	5	5		4.57	0.79
FT2	3	4	4	5	4	5	5		4.29	0.76
FT3	3	3	4	4	4	4	4		3.71	0.49
FT4	3	5	4	4	5	5	4		4.29	0.76
LS1	3	4	5	5	5	4	3		4.14	0.90
LS2	3	5	5	5	5	3	3		4.14	1.07
LS3	3	5	5	5	4	5	3		4.29	0.95
LS4	3	3	4	4	3	4	3		3.43	0.53
LS5	3	4	5	5	4	3	3		3.86	0.90
W1	3	5	5	5	4	4	4		4.29	0.76
W2	3	4	5	5	3	5	5		4.29	0.95
W3	3	4	5	4	2	4	4		3.71	0.95
W4	3	5	4	5	5	5	5		4.57	0.79
W5	3	3	3	5	4	3	3		3.43	0.79
W6	3	4	5	5	5	4	3		4.14	0.90

Table E.1: Industrial survey results, each line representing the data of a single question, with corresponding means and standard deviation values.

Nomenclature

Abstraction The process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose [Wik10a].

Accidental Complexity Complexity that arises in computer artifacts, or their development process, which is non-essential to the problem to be solved.

Actor In UML, it specifies a role played by a user or any other system that interacts with the subject [OMG10d].

Adaptability Characteristic of a system that empowers end-users without or with limited programming skills to customize or tailor it according to their individual or environment-specific requirements.

Agile Agile software development refers to a collection of development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

AOM Acronym for **A**daptive **O**bject-**M**odel [YBJ01b].

API Acronym for **A**pplication **P**rogramming **I**nterface.

BNF Acronym for **B**ackus-**N**aur **F**orm. A meta-syntax for context-free grammars such as those defining programming languages.

Case Study Research methodology based on an in-depth investigation of a single individual, group, or event to explore causation in order to find underlying principles.

Concurrency A property of systems in which several computations are executing simultaneously, and potentially interacting with each other.

CRUD Acronym for **C**reate, **R**ead, **U**pdate, and **D**elete.

DBMS Acronym for **D**atabase **M**anagement **S**ystem.

DDD Acronym for **D**omain-**D**riven **D**esign [Eva03].

DDL Acronym for **D**ata **D**efinition **L**anguage.

DML Acronym for **D**ata **M**anipulation **L**anguage.

DSL Acronym for **D**omain **S**pecific **L**anguage.

DSML Acronym for **D**omain **S**pecific **M**odeling **L**anguage.

EMF Acronym for **E**clipse **M**odeling **F**ramework.

Encapsulation The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation [BME+07].

End-user In software engineering, it refers to an abstraction of the group of persons who will ultimately operate a piece of software, i.e., the expected user or target-user.

Epistemology The branch of philosophy concerned with the nature and scope (limitations) of knowledge.

Essential Complexity In contrast to Accidental Complexity, it is that considered inherent and unavoidable to perform a desired computation or express a software artifact.

Extensibility It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension.

FR Acronym for **F**unctional **R**equirement. A function of a software system or its component, described as a set of inputs, expected behavior, and consequent outputs.

Framework In Object-Oriented Programming, frameworks are reusable designs of all or part of a software system described by a set of abstract artifacts and the way they collaborate [RJ96].

Generative Programming Systematic transformation of an (high) level description of a system (model) into executable code or code skeleton [RKS98].

Granularity In the context of reflective systems, represents the smallest aspect of the base-entities of a computation system that are represented by different meta-entities.

GUI Acronym for **G**raphical **U**ser **I**nterface.

Hollywood Principle A framework design principle based on the cliché response given to amateurs auditioning in Hollywood, *"Don't call us, we'll call you"*.

IDE Acronym for **I**ntegrated **D**evelopment **E**nvironment.

Information Flux Measures the amount of information that is exchanged between elements of a system to perform a desired computation.

kLOC Acronym for **k**ilo **L**ines **O**f **C**ode — effectively thousands of LOC.

Lifecycle In the context of reflective systems, it is the period of the system execution in which a specific meta-entity has to exist.

LOC Acronym for **L**ines **O**f **C**ode.

MDA Acronym for **M**odel-**D**riven **A**rchitecture [OMG10b].

MDE Acronym for **M**odel-**D**riven **E**ngineering [Sch06].

Meta-Architecture Architectures that can dynamically adapt at runtime to new user requirements.

Metamodelling In software engineering, it is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems.

Metaprogramming The process of writing programs that generate or manipulate either other programs, or themselves, by treating code as data [CI84].

MOF Acronym for **M**eta-**O**bject **F**acility [OMG10a].

MVC Acronym for **M**odel-**V**iew-**C**ontroller [BMR+96].

Naked Objects A software architectural pattern that heavily emphasizes the automatic generation of graphical user interfaces from a very straightforward interpretation of the domain models [Paw04].

NFR Acronym for **N**on-**F**unctional **R**equirement. Specifies criteria that can be used to judge the operation of a system, rather than specific behaviors, i.e., while FRs define what a system is supposed to do (function), NFRs define how a system is supposed to be (form).

OO Acronym for Object-Oriented.

OOP Acronym for Object-Oriented Programming.

OOUI Acronym for Object-Oriented User Interface.

ORM Acronym for Object-Relational Mapping.

Pattern In software, it is a recurrent, recognized good solution for a recurrent architectural, design or implementation problem [AIS77, GHJV94].

Performance General measure that may mean short response time, high throughput, low utilization of computing resources, etc.

PIM Acronym for Platform-Independent Model.

Proliferation In the context of reflective systems, it is a measure of the quantity of objects necessary to perform, or represent, a given meta-computation.

PSM Acronym for Platform-Specific Model.

Requirement It is a statement that identifies a necessary attribute, capability, characteristic, or quality of a system in order for it to have value and utility to a user.

Reuse The ability of using existing artifacts, or knowledge, to build or synthesize new solutions, or to apply existing solutions to different artifacts.

SDLC Acronym for Software Development Lifecycle.

Separation of Concerns A concept that establishes the fact that a particular functionality of a systems should be the concern of different, specialized, components.

Silver Bullet In software engineering, it refers to the claim that there is no single development, in either technology or management technique, which by itself promises even one order of magnitude (tenfold) improvement within a decade in productivity, reliability, and/or simplicity [Bro87].

Software Product Line A set of software systems which share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [CN01].

Transparency In reflective systems, it is a measure of how much of the underlying system is available through reflection.

UI Acronym for User Interface.

UML Acronym for Unified Modeling Language [OMG10d].

Usability The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use [ISO98].

Use Case In software engineering, it is a description of an intended system's behavior as the respond to an outside request or interaction.

Variability The need of a software system or artifact to be changed, customized or configured for use in different contexts [JVBS01].

VM Acronym for Virtual Machine.

XML Acronym for eXtensible Markup Language. An open standard which specifies a set of rules for encoding documents in both human and machine-readable form.

XP Acronym for eXtreme Programming. An agile software development methodology which is intended to improve software quality and responsiveness to changing customer requirements [BA04].

XSLT Acronym for e**X**tensible **S**tylesheet **L**anguage **T**ransformations. A declarative, XML-based language used for the transformation of XML documents into other XML (or textual) documents.

References

[ABBL05] M Arnoldi, K Beck, M Bieri, and M Lange, *Time travel: A pattern language for values that change*. Cited on pp. 86 and 87.

[AD05] Ademar Aguiar and Gabriel David, *Wikiwiki weaving heterogeneous software artifacts*, WikiSym '05: Proceedings of the 2005 international symposium on Wikis (New York, NY, USA), ACM, 2005, pp. 67–74. Cited on p. 32.

[AEQ99] Jim Arlow, Wolfgang Emmerich, and John Quinn, *Literate modelling — capturing business knowledge with the uml*, UML'98: Selected papers from the First International Workshop on The Unified Modeling Language (London, UK), Springer-Verlag, 1999, pp. 189–199. Cited on p. 15.

[AG05] Katja Andresen and Norbert Gronau, *An approach to increase adaptability in erp systems*, Proceedings of the 2005 Information Recources Management Association International Conference, Idea Group Publishing, May 2005, pp. 883–885. Cited on p. 18.

[Agu03] Ademar Aguiar, *A minimalist approach to framework documentation*, Ph.D. thesis, Faculdade de Engenharia da Universidade do Porto, September 2003. Cited on pp. 31 and 32.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein, *A pattern language: Towns, buildings, construction*, Oxford University Press, October 1977. Cited on pp. 9, 36, 60, and 183.

[AKK] K Altmanninger, G Kappel, and A Kusel, *Amor–towards adaptable model versioning*, info.fundp.ac.be. Cited on pp. 83, 87, and 90.

[Ale64] Christopher Alexander, *Notes on the synthesis of form*, Harvard University Press, October 1964. Cited on pp. 8, 9, and 51.

[Amb03] Scott Ambler, *Agile database techniques: Effective strategies for the agile software developer*, John Wiley & Sons, Inc., New York, NY, USA, 2003. Cited on p. 106.

[And] F Anderson, *A collection of history patterns*, Collected papers from the PLoP'98 and EuroPLoP'98 Conference. Cited on p. 86.

[AOSD07] Nicolas Anquetil, Káthia M. Oliveira, Kleiber D. Sousa, and Márcio G. Batista Dias, *Software maintenance seen as a knowledge management issue*, Inf. Softw. Technol. **49** (2007), no. 5, 515–529. Cited on p. 4.

[Ars00] A Arsanjani, *Rule object: A pattern language for adaptive and scalable business rule construction*, Proceeding of PLoP (2000). Cited on pp. 38, 42, and 44.

[AS96] Harold Abelson and Gerald J. Sussman, *Structure and interpretation of computer programs*, MIT Press, Cambridge, MA, USA, 1996. Cited on p. 20.

[ATMB00] Mehmet Aksit, Bedir Tekinerdogan, Francesco Marcelloni, and Lodewijk Bergmans, *Deriving object-oriented frameworks from domain knowledge*, Building Application Frameworks: Object-Oriented Foundations of Framework Design (Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, eds.), John Wiley & Sons Inc., New York, USA, 2000, pp. 169–198. Cited on pp. 30 and 93.

[BA04] Kent Beck and Cynthia Andres, *Extreme programming explained: Embrace change*, 2nd ed., Addison-Wesley Professional, 2004. Cited on pp. 4, 62, and 183.

[Bas87] P.G. Bassett, *Frame-based software engineering*, IEEE Software **4** (1987), 9–16. Cited on p. 20.

[BBH69] F. L. Bauer, L. Bolliet, and Dr. H. J. Helms, *Software engineering*, Report on a conference sponsored by the NATO SCIENCE COMMITTEE (Peter Naur and Brian Randell, eds.), Scientific Affairs Division, NATO, 1969, p. 136. Cited on p. 2.

[BEK+04] Bartosz B□bel, Johann Eder, Christian Koncilia, Tadeusz Morzy, and Robert Wrembel, *Creation and management of versions in multiversion data warehouse*, SAC '04: Proceedings of the 2004 ACM symposium on Applied computing (New York, NY, USA), ACM, 2004, pp. 717–723. Cited on pp. 78, 83, 87, and 90.

[BME+07] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston, *Object-oriented analysis and design with applications*, Addison-Wesley Professional, 2007. Cited on p. 181.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-oriented software architecture volume 1: A system of patterns*, Wiley, August 1996. Cited on pp. 9, 36, 51, 93, 131, and 182.

[Bro87] Fred P. Brooks, *No silver bullet — essence and accidents of software engineering*, IEEE Computer **20** (1987), 10–19. Cited on pp. 2, 49, and 183.

[BW09] Federico Biancuzzi and Shane Warden, *Masterminds of programming: Conversations with the creators of major programming languages*, O'Reilly Media, Inc., 2009. Cited on p. 19.

[Cam04] Camara Municipal do Porto, *Revisão do Plano Director Municipal do Porto – Regulamento*, Maio 2004. Cited on p. 118.

[Caz98] Walter Cazzola, *Evaluation of object-oriented reflective models*, Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), in 12th European Conference on Object-Oriented Programming (ECOOP'98) (1998). Cited on pp. 18, 20, and 94.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker, *Generative programming: Methods, tools, and applications*, Addison-Wesley Professional, June 2000. Cited on p. 21.

[CEF98] Andy Carlson, Sharon Estepp, and Martin Fowler, *Temporal patterns*, PLoP '98: Proceedings of the 5th Conference on Pattern Languages of Programs, August 1998. Cited on p. 86.

[CI84] Robert D. Cameron and M. Robert Ito, *Grammar-based definition of metaprogramming systems*, ACM Transactions on Programming Languages and Systems **6** (1984), no. 1, 20–54. Cited on pp. 20 and 182.

[CJMS10] Jeffrey C. Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull, *A checklist for integrating student empirical studies with research and teaching goals*, Empirical Softw. Eng. **15** (2010), no. 1, 35–59. Cited on p. 129.

[Cle88] J. Craig Cleaveland, *Building application generators*, IEEE Softw. **5** (1988), no. 4, 25–33. Cited on p. 20.

[CLG+09] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee (eds.), *Software engineering for self-adaptive systems*, Springer-Verlag, Berlin, Heidelberg, 2009. Cited on p. 18.

[CN01] Paul Clements and Linda Northrop, *Software product lines: practices and patterns*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. Cited on pp. 26 and 183.

[Cop96] James O. Coplien, *Software patterns*, 1996. Cited on p. 60.

[CPP10] Filipe Correia, Fátima Pires, and Aurélio Pires, Tech. report, ParadigmaXis, S.A., 2010, http://gisa.paradigmaxis.pt/. Cited on p. 121.

[Cun95] Ward Cunningham, *WikiWikiWeb*, 1995, http://c2.com/cgi/wiki. Cited on pp. 32 and 94.

[Cun03] _____, *Wiki design principles*, 2003, http://c2.com/cgi/wiki$?WikiDesignPrinciples. Cited on pp. 32 and 94.

[CV05] Rudi Cilibrasi and Paul Vitanyi, *Clustering by compression*, 2005, pp. 1523–1545. Cited on p. 116.

[CW04] A Correa and C Werner, *Applying refactoring techniques to uml/ocl models*, UML (2004). Cited on p. 90.

[Cza04] Krzysztof Czarnecki, *Overview of generative software development*, Unconventional Programming Paradigms (UPP), Springer-Verlag, September 2004, pp. 313–328. Cited on p. 21.

[ddB10] Antoine d'Otreppe de Bouvette, *Aspyct*, 2010, [Online; Accessed 28-August-2010]. Cited on p. 23.

[Dij72] Edsger Wybe Dijkstra, *The Humble Programmer*, Communications of the ACM **15** (1972), no. 10, 859–866. Cited on p. 2.

[DS90] Peter DeGrace and Leslie Hulet Stahl, *Wicked problems, righteous solutions*, Yourdon Press, Upper Saddle River, NJ, USA, 1990. Cited on p. 5.

[DW99] Desmond F. D'Souza and Alan Cameron Wills, *Objects, components, and frameworks with uml: the catalysis approach*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. Cited on p. 30.

[DYBJ08] Ayla Dantas, Joseph Yoder, Paulo Borba, and Ralph Johnson, *Using aspects to make adaptive object-models adaptable*, Research Reports on Mathematical and Computing Sciences (2008). Cited on p. 21.

[EV10] J. Laurenz Eveleens and Chris Verhoef, *The rise and fall of the chaos report figures*, IEEE Software **27** (2010), 30–36. Cited on p. 2.

[Eva03] Eric Evans, *Domain-driven design: Tackling complexity in the heart of software*, Addison-Wesley Professional, August 2003. Cited on pp. 26 and 181.

[FAF09] Hugo Sereno Ferreira, Ademar Aguiar, and João Pascoal Faria, *Adaptive object-modelling: Patterns, tools and applications*, International Conference on Software Engineering Advances (Los Alamitos, CA, USA), IEEE Computer Society, 2009, pp. 530–535. Cited on p. 25.

[FCR10] Nuno Flores, Filipe Correia, and Rosaldo Rossetti, 2010, https://www.fe.up.pt/si/disciplinas_geral.FormView?P_CAD_CODIGO=EIC0086&P_ANO_LECTIVO=2010/2011&P_PERIODO=1S [Online; accessed 16-December-2010]. Cited on p. 154.

[FCW08] Hugo Sereno Ferreira, Filipe Figueiredo Correia, and León Welicki, *Patterns for data and metadata evolution in adaptive object-models*, PLoP '08: Proceedings of the 15th Conference on Pattern Languages of Programs (New York, NY, USA), ACM, 2008, pp. 1–9. Cited on p. 36.

[Fed10] Fedora Commons, *Introduction to Fedora Object XML (FOXML)*, 2010, http://www.fedora-commons.org/download/2.0/userdocs/digitalobjects/introFOXML.html [Online; accessed 31-July-2010]. Cited on p. 121.

[FJ00] Mohamed E. Fayad and Ralph E. Johnson, *Domain-specific application frameworks: framework experience by industry*, John Wiley & Sons, Inc., New York, NY, USA, 2000. Cited on p. 29.

[Fow96] Martin Fowler, *Analysis patterns: Reusable object models*, Addison-Wesley Professional, October 1996. Cited on pp. 38 and 42.

[Fow97] _____, *Analysis patterns: reusable objects models*, Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1997. Cited on p. 42.

[Fow99] _____, *Refactoring: Improving the design of existing code*, Addison-Wesley, Boston, MA, USA, 1999. Cited on p. 90.

[Fow02] _____, *Patterns of enterprise application architecture*, Addison-Wesley Professional, November 2002. Cited on pp. 79, 84, 87, 88, and 90.

[Fow10a] _____, *Analysis patterns: Audit log*, 2010, http://www.martinfowler.com/ap2/auditLog.html [Online; accessed 5-July-2010]. Cited on p. 83.

[Fow10b] _____, *Analysis patterns: Effectivity*, 2010, http://www.martinfowler.com/ap2/effectivity.html [Online; accessed 5-July-2010]. Cited on p. 86.

[Fow10c] _____, *Analysis patterns: Temporal object*, 2010, http://www.martinfowler.com/ap2/temporalObject.html [Online; accessed 5-July-2010]. Cited on p. 86.

[Fow10d] _____, *Analysis patterns: Temporal property*, 2010, http://www.martinfowler.com/ap2/temporalProperty.html [Online; accessed 5-July-2010]. Cited on p. 86.

[FPR00] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe, *The UML Profile for Framework Architectures*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. Cited on pp. 29 and 31.

[FR07] Robert France and Bernhard Rumpe, *Model-driven development of complex software: A research roadmap*, FOSE '07: 2007 Future of Software Engineering (Washington, DC, USA), IEEE Computer Society, 2007, pp. 37–54. Cited on pp. 15, 27, and 48.

[FSJ99a] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, *Building application frameworks: object-oriented foundations of framework design*, John Wiley & Sons, Inc., New York, NY, USA, 1999. Cited on p. 29.

[FSJ99b] _____, *Implementing application frameworks: object-oriented frameworks at work*, John Wiley & Sons, Inc., New York, NY, USA, 1999. Cited on pp. 29 and 93.

[FY97] Brian Foote and Joseph Yoder, *Big ball of mud*, Pattern Languages of Program Design, Addison-Wesley, 1997, pp. 653–692. Cited on pp. 5 and 50.

[GA07] Miguel Goulao and Fernando Brito Abreu, *Modeling the experimental software engineering process*, QUATIC '07: Proceedings of the 6th International Conference on Quality of Information and Communications Technology (Washington, DC, USA), IEEE Computer Society, 2007, pp. 77–90. Cited on p. 56.

[Gab96] Richard P. Gabriel, *Patterns of software: tales from the software community*, Oxford University Press, Inc. New York, NY, USA, 1996. Cited on p. xix.

[GGGR09] Poonam Goyal, Navneet Goyal, Ashish Gupta, and T. S. Rahul, *Designing self-adaptive websites using online hotlink assignment algorithm*, Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia, 2009, pp. 579–583. Cited on p. 155.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley Professional, November 1994. Cited on pp. 9, 33, 36, 43, 44, 51, 60, 80, 81, 83, 86, 88, 94, 104, 106, 108, 131, and 183.

[GHV95] E. Gamma, R. Helm, and J. Vlissides, *Design patterns applied*, 1995. Cited on p. 36.

[GJT07] Raghu Garud, Sanjay Jain, and Philipp Tuertscher, *Incomplete by design and designing for incompleteness*, Organization studies as a science of design (Marianne and Georges Romme, eds.), 2007. Cited on pp. 7, 62, and 93.

[GNU10] GNU, *GNU Diffutils*, 2010, http://www.gnu.org/software/diffutils/ [Online; accessed 5-July-2010]. Cited on p. 116.

[Goo] Google, *iGoogle*, http://google.com/ig [Online; accessed 16-July-2010]. Cited on p. 154.

[Gopo6] Ganesh Gopalakrishnan, *Computation engineering: Applied automata theory and logic*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. Cited on p. xx.

[Gou08] Miguel Goulao, *Component-based software engineering: a quantitative approach*, Ph.D. thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2008. Cited on pp. 10 and 15.

[GPHS08] Cesar Gonzalez-Perez and Brian Henderson-Sellers, *Metamodelling for software engineering*, Wiley Publishing, 2008. Cited on p. 22.

[GR83] Adele Goldberg and David Robson, *Smalltalk-80: the language and its implementation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. Cited on p. 33.

[Gra10] Joao Gradim, 2010, http://paginas.fe.up.pt/ei05030/thesis/ [Online; accessed 16-December-2010]. Cited on p. 155.

[Han] David Heinemeier Hansson, *Ruby on rails*, [Online; Accessed 28-August-2010]. Cited on p. 33.

[HBJ09] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens, *Cope - automating coupled evolution of metamodels and models*, Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 52–76. Cited on pp. 83 and 90.

[HC01] George T. Heineman and William T. Councill (eds.), *Component-based software engineering: putting the pieces together*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. Cited on p. 15.

[Hei98] Constance L. Heitmeyer, *On the need for practical formal methods*, FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (London, UK), Springer-Verlag, 1998, pp. 18–26. Cited on p. 5.

[HL95] Tim Hart and Mike Levin, *Ai memo 39-the new compiler*. Cited on p. 155.

[Hof79] Douglas R. Hofstadter, *Godel, escher, bach: An eternal golden braid*, Basic Books, Inc., New York, NY, USA, 1979. Cited on p. 21.

[HW99] Myles Hollander and Douglas A. Wolfe, *Nonparametric statistical methods*, 2nd ed., Wiley-Interscience, January 1999. Cited on p. 138.

[ISO98] ISO 9241-11:1998, *Ergonomic requirements for office work with visual display terminals (vdts) – part 11: Guidance on usability*, ISO, Geneva, Switzerland, 1998. Cited on pp. 65 and 183.

[Jet10] JetBrains, *Meta Programming System*, 2010, [Online; Accessed 28-August-2010]. Cited on p. 24.

[JF88] Ralph E. Johnson and Brian Foote, *Designing reusable classes*, Journal of Object-Oriented Programming 1 (1988), no. 2, 22–35. Cited on p. 30.

[JO98] Ralph Johnson and Jeff Oakes, *The user-defined product framework*, 1998. Cited on p. 36.

[Joh78] Stephen C. Johnson, *Yacc: Yet another compiler-compiler*, Tech. report, Bell Laboratories, 1978. Cited on p. 24.

[JVBS01] Gurp Jilles Van, Jan Bosch, and Mikael Svahnberg, *On the notion of variability in software product lines*, WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (Washington, DC, USA), IEEE Computer Society, 2001, p. 45. Cited on pp. 17 and 183.

[JW97] Ralph Johnson and Bobby Woolf, *The type object pattern*, Pattern Languages of Program Design 3, Addison-Wesley, 1997, pp. 47–65. Cited on pp. 38, 39, 40, and 131.

[KAKB+08] Barbara Kitchenham, Hiyam Al-Khilidar, Muhammed Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu, *Evaluating guidelines for reporting empirical software engineering studies*, Empirical Softw. Eng. 13 (2008), no. 1, 97–121. Cited on p. 56.

[Kay93] Alan C. Kay, *The early history of smalltalk*, HOPL-II: The second ACM SIGPLAN conference on History of programming languages (New York, NY, USA), ACM, 1993, pp. 69–95. Cited on pp. 2, 32, and 155.

[KH01] Gregor Kiczales and Erik Hilsdale, *Aspect-oriented programming*, ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (New York, NY, USA), ACM, 2001, p. 313. Cited on pp. 21 and 23.

[KM05] Gregor Kiczales and Mira Mezini, *Aspect-oriented programming and modular reasoning*, ICSE '05: Proceedings of the 27th international conference on Software engineering (New York, NY, USA), ACM, 2005, pp. 49–58. Cited on p. 23.

[KOS07] John Krogstie, Andreas L. Opdahl, and Guttorm Sindre (eds.), *Advanced information systems engineering, 19th international conference, caise 2007, trondheim, norway, june 11-15, 2007, proceedings*, Lecture Notes in Computer Science, vol. 4495, Springer, 2007. Cited on p. 15.

[KP09] Christian Kohls and Stefanie Panke, *Is that true...? – thoughts on the epistemology of patterns*, PLoP '09: Proceedings of the 16th Conference on Pattern Languages of Programs, 2009. Cited on p. 60.

[LC03] Donal Lafferty and Vinny Cahill, *Language-independent aspect-oriented programming*, OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (New York, NY, USA), ACM, 2003, pp. 1–12. Cited on p. 23.

[Lev86] Leon S. Levy, *A metaprogramming method and its economic justification*, IEEE Trans. Softw. Eng. **12** (1986), no. 2, 272–277. Cited on p. 21.

[Lew46] *Action research and minority problems*, Journal of Social Issues **2** (1946), no. 4, 34–46. Cited on p. 52.

[Lik32] Rensis Likert, *A technique for the measurement of attitudes*, Archives of Psychology **22** (1932), no. 140, 1–55. Cited on pp. 122 and 132.

[LV08] Ming Li and Paul M.B. Vitnyi, *An introduction to kolmogorov complexity and its applications*, Springer Publishing Company, Incorporated, 2008. Cited on p. 116.

[MB02] Stephen J. Mellor and Marc Balcer, *Executable uml: A foundation for model-driven architectures*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Cited on p. 28.

[Med07] MediaWiki, *MediaWiki — MediaWiki, The Free Wiki Engine*, 2007, [Online; accessed 20-August-2010]. Cited on p. 32.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel, *Weaving executability into object-oriented meta-languages*, Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems, October 2005. Cited on p. 90.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane, *When and how to develop domain-specific languages*, ACM Comput. Surv. **37** (2005), no. 4, 316–344. Cited on p. 24.

[Min99] Ministério do Equipamento, do Planeamento e da Administração do Território, *Decreto-Lei n. 380/99*, Setembro 1999. Cited on p. 118.

[MJ06] Peter Meso and Radhika Jain, *Agile software development: Adaptive systems principles and best practices*, Information Systems Management **23** (2006), no. 3, 19–30. Cited on p. 18.

[MM03] J. Miller and J. Mukerji, *Mda guide version 1.0.1*, Tech. report, Object Management Group (OMG), 2003. Cited on pp. 27 and 28.

[MRB97] Robert Martin, Dirk Riehle, and Frank Buschmann (eds.), *Pattern languages of program design 3*, Addison-Wesley, 1997. Cited on p. 59.

[MS03] David Mertz and Michele Simionato, *Metaclass programming in Python*, 2003, [Online; Accessed 16-July-2010]. Cited on p. 23.

[Nar93] Bonnie A. Nardi, *A small matter of programming: Perspectives on end user computing*, MIT Press, Cambridge, MA, USA, 1993. Cited on p. 50.

[NDP09] Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet, *Squeak by example*, Square Bracket Associates, October 2009. Cited on p. 33.

[NL06] Colin J. Neill and Philip A. Laplante, *Paying down design debt with strategic refactoring*, Computer **39** (2006), 131–134. Cited on p. 7.

[OMG10a] OMG, *MetaObject Facility (MOF)*, 2010, http://www.omg.org/mof/ [Online; accessed 5-July-2010]. Cited on pp. 28, 63, and 182.

[OMG10b] _____, *Model Driven Architecture (MDA)*, 2010, http://www.omg.org/mda/ [Online; accessed 5-July-2010]. Cited on pp. 28 and 182.

[OMG10c] _____, *Object Constraint Language (OCL)*, 2010, http://www.omg.org/spec/OCL/ [Online; accessed 5-July-2010]. Cited on p. 14.

[OMG10d] _____, *Unified Modelling Language (UML)*, 2010, http://www.uml.org/ [Online; accessed 5-July-2010]. Cited on pp. 14, 181, and 183.

[Ope08] Open Source, *bzip2*, 2008, http://www.bzip.org/ [Online; accessed 5-July-2010]. Cited on p. 116.

[Ope10] _____, *Prevayler — the open source prevalence layer*, 2010, http://www.prevayler.org [Online; accessed 5-July-2010]. Cited on pp. 83, 87, and 90.

[Pag] Pageflakes, http://pageflakes.com/ [Online; accessed 16-July-2010]. Cited on p. 154.

[Par07] Terence Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*, Pragmatic Bookshelf, 2007. Cited on p. 24.

[Paw04] Richard Pawson, *Naked objects*, Ph.D. thesis, University of Dublin, Trinity College, June 2004. Cited on pp. 26, 94, and 182.

[Per82] Alan J. Perlis, *Special feature: Epigrams on programming*, SIGPLAN Not. **17** (1982), 7–13. Cited on p. 156.

[PFSP10] Aurélio Pires, Hugo Sereno Ferreira, Hugo Silva, and José Porto, *Locvs – gestão do património arquitectónico e arqueológico da cidade do porto*, Tech. report, ParadigmaXis, S.A., 2010, Produced for Câmara Municipal do Porto. Cited on p. 118.

[PQ95] Terence Parr and Russell Quong, *ANTLR: A Predicated-LL(k) parser generator*, Journal of Software Practice and Experience, **25** (1995), no. 7, 789–810. Cited on p. 24.

[Pre99] *Hot-spot-driven development*, John Wiley and Sons., 1999. Cited on p. 30.

[PT07] Carla Pacheco and Edmundo Tovar, *Stakeholder identification as an issue in the improvement of software requirements quality*, CAiSE'07: Proceedings of the 19th international conference on Advanced information systems engineering (Berlin, Heidelberg), Springer-Verlag, 2007, pp. 370–380. Cited on p. 4.

[Pyp] Pypy, http://codespeak.net/pypy/dist/pypy/doc/ [Online; accessed 16-July-2010]. Cited on p. 155.

[RD98] D Riehle and E Dubach, *Why a bank needs dynamic object models*, OOPSLA Workshop on Metadata and Active Object Models (1998). Cited on p. 36.

[RFBLO01] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe, *The architecture of a uml virtual machine*, OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), ACM, 2001, pp. 327–341. Cited on pp. 15, 22, 23, 27, and 28.

[RG98] Dirk Riehle and Thomas Gross, *Role model based framework design and integration*, SIGPLAN Not. 33 (1998), no. 10, 117–133. Cited on p. 30.

[RJ96] Don Roberts and Ralph Johnson, *Evolving frameworks: A pattern language for developing object-oriented frameworks*, Proceedings of the Third Conference on Pattern Languages and Programming, vol. 3, 1996. Cited on pp. 29, 30, 31, 51, 53, 93, and 182.

[RKS98] G Roy, J Kelso, and C Standing, *Towards a visual programming environment for software development*, Proceedings on Software Engineering: Education & Practice (1998). Cited on pp. 21 and 182.

[RL04] Awais Rashid and Nicholas Leidenfrost, *Supporting flexible object database evolution with aspects*, Springer, 2004, pp. 75–94. Cited on pp. 78, 80, 83, 87, and 90.

[Roy87] Winston W. Royce, *Managing the development of large software systems: concepts and techniques*, Proceedings of the 9th international conference on Software Engineering (Los Alamitos, CA, USA), ICSE '87, IEEE Computer Society Press, 1987, pp. 328–338. Cited on p. 4.

[Rub] Rubinus, http://rubini.us/ [Online; accessed 16-July-2010]. Cited on p. 155.

[SAJ+02] Eva Söderström, Birger Andersson, Paul Johannesson, Erik Perjons, and Benkt Wangler, *Towards a framework for comparing process modelling languages*, CAiSE '02: Proceedings of the 14th International Conference on Advanced Information Systems Engineering (London, UK), Springer-Verlag, 2002, pp. 600–611. Cited on p. 22.

[Sch97] Han Albrecht Schmid, *Systematic framework design by generalization*, Commun. ACM 40 (1997), no. 10, 48–51. Cited on p. 30.

[Sch05] Barry Schwartz, *The paradox of choice: Why more is less*, Harper Perennial, January 2005. Cited on p. 9.

[Sch06] D.C. Schmidt, *Model-driven engineering*, IEEE Computer (2006), no. 39. Cited on pp. 27 and 182.

[Sch07] Werner Schuster, *What's a ruby dsl and what isn't?*, June 2007, http://www.infoq.com/news/2007/06/dsl-or-not. Cited on p. 24.

[Sha01] Mary Shaw, *The coming-of-age of software architecture research*, ICSE '01: Proceedings of the 23rd International Conference on Software Engineering (Washington, DC, USA), IEEE Computer Society, 2001, p. 656. Cited on p. 12.

[Spo02] Joel Spolsky, *The law of leaky abstractions*, 2002, http://www.joelonsoftware.com/articles/LeakyAbstractions.html [Online; accessed 5-July-2010]. Cited on p. 19.

[SSS07] Forrest Shull, Janice Singer, and Dag I.K. Sjøberg, *Guide to advanced empirical software engineering*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. Cited on p. 56.

[Sta94] Standish Group International, *The chaos report*, Tech. report, 1994. Cited on p. 2.

[Sta03] Thornton Staples, *An open-source digital object repository management system*, 2003. Cited on p. 121.

[Ste06] Friedrich Steimann, *The paradoxical success of aspect-oriented programming*, OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (New York, NY, USA), ACM, 2006, pp. 481–497. Cited on p. 23.

[Suro5] James Surowiecki, *The wisdom of crowds*, Anchor, 2005. Cited on p. 32.

[SV06] Thomas Stahl and Markus Völter, *Model-driven software development: Technology, engineering, management*, Wiley, May 2006. Cited on p. 22.

[Szy02] Clemens Szyperski, *Component software: Beyond object-oriented programming*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Cited on p. 15.

[THB+06] Dave Thomas, David Hansson, Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz, *Agile web development with rails*, Pragmatic Bookshelf, 2006. Cited on pp. 33 and 34.

[THP93] Walter F. Tichy, Nico Habermann, and Lutz Prechelt, *Summary of the dagstuhl workshop on future directions in software engineering: February 17–21, 1992, schloßdagstuhl*, SIGSOFT Softw. Eng. Notes **18** (1993), no. 1, 35–48. Cited on p. 11.

[TN86] Hirotaka Takeuchi and Ikujiro Nonaka, *The new new product development game*, Harvard Business Review (1986). Cited on p. 5.

[TPT09] Susanna Teppola, Paivi Parviainen, and Juha Takalo, *Challenges in deployment of model driven development*, International Conference on Software Engineering Advances (2009), 15–20. Cited on p. 48.

[Und08] *Understanding migrations in ruby on rails*, 2008, http://wiki.rubyonrails.org/rails/pages/understandingmigrations [Online; accessed 8-August-2008]. Cited on p. 90.

[Usi08] *Using migrations in ruby on rails*, 2008, http://wiki.rubyonrails.org/rails/pages/UsingMigrations [Online; accessed 8-August-2008]. Cited on p. 90.

[Völ03] Markus Völter, *A catalog of patterns for program generation*, Proceedings of the Eighth European Conference on Pattern Languages of Programs, June 2003. Cited on p. 15.

[War94] Martin P. Ward, *Language-oriented programming*, Software — Concepts and Tools **15** (1994), no. 4, 147–161. Cited on p. 24.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson, *Surveying current research in object-oriented design*, Commun. ACM **33** (1990), no. 9, 104–124. Cited on p. 30.

[WC03] Laurie Williams and Alistair Cockburn, *Guest editors' introduction: Agile software development: It's about feedback and change*, Computer **36** (2003), 39–43. Cited on p. 4.

[WE00] Han-Chieh Wei and Ramez Elmasri, *Schema versioning and database conversion techniques for bi-temporal databases*, Annals of Mathematics and Artificial Intelligence **30** (2000), no. 1-4, 23–52. Cited on pp. 78, 83, 87, and 90.

[Web10] Webster's Online Dictionary, *Design*, 2010, http://www.websters-online-dictionary.org/ [Online; accessed 5-July-2010]. Cited on p. 8.

[WGM89] André Weinand, Erich Gamma, and Rudolf Marty, *Design and implementation of ET++, a seamless object-oriented application framework*, Structured Programming (1989), 63–87. Cited on p. 29.

[Wik10a] Wikipedia, *Abstraction — wikipedia, the free encyclopedia*, 2010, http://en.wikipedia.org/w/index.php?title=Abstraction&oldid=373722967 [Online; accessed 9-July-2010]. Cited on pp. 18 and 181.

[Wik10b] _____, *Kolmogorov complexity — wikipedia, the free encyclopedia*, 2010, http://en.wikipedia.org/w/index.php?title=Kolmogorov_complexity&oldid=375473546 [Online; accessed 2-August-2010]. Cited on p. 116.

[Wik10c] _____, *Metamodeling — Wikipedia, The Free Encyclopedia*, 2010, [Online; accessed 4-September-2010]. Cited on p. 23.

[Wik10d] _____, *Wikipedia, The Free Encyclopedia*, 2010, [Online; accessed 20-August-2010]. Cited on p. 32.

[WYWB07] León Welicki, Joseph Yoder, and Rebecca Wirfs-Brock, *A pattern language for adaptive object models: Part i-rendering patterns*, PLoP '07: Proceedings of the 14th Conference on Pattern Languages of Programs, 2007. Cited on pp. 36, 39, 78, and 112.

[WYWB09] León Welicki, Joseph Yoder, and Rebecca Wirfs-Brock, *Adaptive object-model builder*, PLoP '09: Proceedings of the 16th Conference on Pattern Languages of Programs, 2009. Cited on pp. 36 and 78.

[WYWBJ07] León Welicki, Joseph Yoder, Rebecca Wirfs-Brock, and Ralph E. Johnson, *Towards a pattern language for adaptive object models*, OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (New York, NY, USA), ACM, 2007, pp. 787–788. Cited on pp. 36, 37, 38, 39, 47, 53, 61, 78, 79, and 90.

[YBJ01a] Joseph Yoder, Federico Balaguer, and Ralph Johnson, *Adaptive object-models for implementing business rules*, Urbana (2001). Cited on pp. 22, 45, and 105.

[YBJ01b] _____, *Architecture and design of adaptive object-models*, ACM SIG-PLAN Notices **36** (2001), 50–60. Cited on pp. 25, 35, 36, 38, 45, 46, 93, and 181.

[YFRT98] Joseph Yoder, Brian Foote, Dirk Riehle, and Michel Tilman, *Metadata and active object-models*, OOPSLA '98 Addendum: Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum) (New York, NY, USA), ACM, 1998. Cited on pp. 36 and 104.

[Yod02] Joseph Yoder, *The adaptive object model architectural style*, Software Architecture: System Design (2002). Cited on pp. 42, 44, 45, 47, 53, and 153.

[ZW98] Marvin V. Zelkowitz and Dolores R. Wallace, *Experimental models for validating technology*, Computer **31** (1998), 23–31. Cited on pp. 11 and 12.