

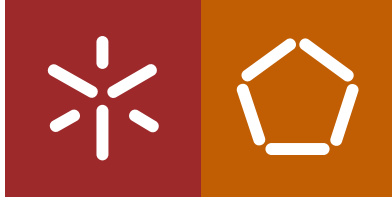


Universidade do Minho
Escola de Engenharia

Rui Miguel Silva Couto | **Pattern Based Software Development**

Rui Miguel Silva Couto

Pattern Based Software Development



Universidade do Minho
Escola de Engenharia

Rui Miguel Silva Couto

Pattern Based Software Development

Tese de Doutoramento em Informática

Trabalho realizado sob a orientação do
Professor António Nestor Ribeiro
e do
Professor José Francisco Creissac Campos

DECLARAÇÃO

Nome

Rui Miguel Silva Couto

Endereço electrónico: rmscouto@gmail.com

Número do Bilhete de Identidade: 13425192

Título tese:

Pattern Based Software Development

Orientadores:

Professor António Nestor Ribeiro

Professor José Francisco Creissac Campos

Ano de conclusão: 2016

Designação do Doutoramento:

Programa Doutoral em Informática

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, 1/07/2017

Assinatura: Rui Miguel Silva Couto

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration. I further declare that I have full acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho: 1 de julho de 2017

Full Name: Rui Miguel Silva Couto

Signature: Rui Miguel Silva Couto

Acknowledgments

This work would not have been possible without the collaboration of many people, to whom I owe a lot.

First, and foremost I thank my supervisors António Nestor Ribeiro e José Creissac Campos, not only for providing me this opportunity, their support and encouragement, but also for their friendship. They provided guidance throughout this project, with numerous suggestions, revisions, and always had the availability to help me.

To my wife, Isabel, I thank with all my heart for always being there for me (even when I was not!), for all the love, patience and forgiveness for my absence. She always encouraged me to keep going even in the hardest moments.

To my family I thank for everything that I will never be able to give back. In particular, to my parents, my brother, my sister in law, my nephews I thank for their love and care, and always remembering me that they were there for me. They always supported me in this project, and forgave me even when I was not present for them.

To all my colleagues I thank for the help, encouragement and funny moments they provided. My special thank to André, Bernardo, Cláudio, Georges, Hugo, Jorge, Marco, Miguel, Paulo, Rui, and I wish them the best success.

Finally, I thank everyone not mentioned here that contributed directly or indirectly to this project.

This project was supported by the University of Minho and INESC TEC/HASLab.

Abstract

Several types of approaches support the software development process. Special interest should be paid to model driven development methodologies, of which Model Driven Architecture (MDA) is a main example. The usage of software models in these methodologies improves the quality of the produced solutions. On the one hand, models are formal artifacts to represent the software to develop. On the other hand, models represented in computable formats are amenable to the application of systematic transformation techniques, in order to produce other models or source code as output.

The architectural models used in the MDA are derived from requirement specifications, and are achieved through manual processes. The negative effects of manual transformation steps are well known, since they are susceptible to interpretation errors and subjectivity. Errors resulting from this process are propagated through all of the development process, and reflected in the produced solutions. Since requirement models specify the system to be developed, naturally, they should not be disconnected from the development process itself.

Formalizing requirement specifications in computable formats would enable their operationalization. Such would provide the possibility to analyze and manipulate them, and also to perform a requirement patterns inference process. Requirement patterns represent well known solutions for recurring problems, and their nature provides architectural hints. If software patterns can be derived from the requirement patterns, then through the composition of the resulting software patterns, architectural models can be achieved. As a result, requirements models will be better integrated into the MDA chain, thus extending the advantages of the MDA to requirement models, providing a software development process which starts from requirements and through rigorous transformations results in software solutions.

This work presents an approach that aims to provide such an integration of requirements models into the MDA. The approach starts with the formalization of software requirements in a controlled natural language. The requirements are then transformed into an intermediary representation (namely, an ontology), with support for information extraction. Such makes it possible to perform requirement pattern inference, in order to understand, at a higher level of abstraction, the features required in the software solution. Associating the requirement patterns with software patterns, makes it possible to instantiate and compose such patterns, in order to produce architectural

artifacts as output. The presented approach is supported by a tool, designed to support the several steps of the approach. Furthermore, the tool provides the required automation level to produce the architectural models. Two validation studies and a case study in the eCommerce domain are also presented, in order to illustrate the viability of both the tool and the approach.

Resumo

Diferentes tipos de abordagens suportam o processo de desenvolvimento de software. Especial interesse deve ser dado às metodologias baseadas em modelos, das quais a *Model Driven Architecture* (MDA) é um exemplo relevante. O uso de modelos de software nestas metodologias melhora a qualidade das soluções obtidas. Por um lado, os modelos são artefactos formais para representar o software a ser desenvolvido. Por outro lado, os modelos representados em formatos computáveis podem ser manipulados utilizando técnicas de transformação sistemáticas, de modo a obter como resultado outros modelos, ou código fonte.

Os modelos arquiteturais usados na MDA derivam das especificações de requisitos, sendo obtidos através de processos manuais. O impacto negativo da aplicação de transformações manuais é bem conhecido, uma vez que estas são susceptíveis a erros de interpretação e subjectividade. Os erros resultantes deste processo são propagados através do processo de desenvolvimento, e reflectem-se nas soluções produzidas. Uma vez que os modelos de requisitos especificam os sistemas a desenvolver, naturalmente, estes não devem estar desligados do processo de desenvolvimento.

A formalização dos modelos de requisitos em formatos computáveis possibilitaria a sua operacionalização. Tal forneceria a capacidade de analisar e manipular os modelos, e também suportaria a inferência de padrões de requisitos. Padrões de requisitos representam soluções bem conhecidas, para problemas recorrentes, e a sua natureza fornece indicações arquiteturais. Se for possível obter padrões de software, através de padrões de requisitos, então através de um processo de composição de padrões de software, é possível obter modelos arquiteturais. Como resultado, os padrões de requisitos podem ser integrados na cadeia MDA, estendendo assim as vantagens desse processo aos modelos de requisitos, e obtendo um processo de desenvolvimento que inicia nos requisitos, e fornece transformações rigorosas que levam a soluções de software.

Este trabalho apresenta uma abordagem que tem como objectivo fornecer tal integração de modelos de requisitos, na MDA. A abordagem inicia com a formalização de requisitos de software numa linguagem natural controlada. Os requisitos são então transformados numa representação intermédia (nomeadamente, uma ontologia), com suporte para extração de informação. Tal fornece a possibilidade de efectuar inferência de padrões de requisitos, de modo a perceber, a um alto nível de abstracção, as funcionalidades necessárias nas soluções de software. Associando os padrões de requisitos com padrões de software, é possível instanciar e compor esses padrões,

de modo a obter artefactos arquitecturais. A abordagem apresentada é suportada por uma ferramenta, desenhada para suportar os diferentes passos da abordagem. Para além disso, a ferramenta fornece a automação necessária para produzir os modelos arquitecturais. São também apresentados dois estudos de validação e um caso de estudo na área de *eCommerce*, de modo a ilustrar a viabilidade da abordagem e da ferramenta.

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem	3
1.3	Proposal	4
1.3.1	Thesis	5
1.3.2	Research Questions	6
1.3.3	Objectives	6
1.4	Contributions	7
1.5	Document Structure	9
2	Background	11
2.1	The Model Driven Architecture Framework	11
2.1.1	The MDA process	13
2.1.2	Discussion	14
2.2	Simplified Languages With Support for Automation	14
2.2.1	Use Cases	14
2.2.2	Natural Language Processing	15
2.2.3	Computable Formats	16
2.2.4	Intermediary Languages	16
2.2.5	Annotation	18
2.2.6	Operationalization Approaches	18
2.2.7	Discussion	20
2.3	Representation of Requirements on Knowledge Bases	21
2.3.1	Ontologies	21
2.3.2	Ontology Languages	22
2.3.3	Knowledge Base Analysis	22
2.3.4	Web Ontology Language	23
2.3.5	Discussion	24
2.4	Knowledge Inference Mechanism	25
2.4.1	Inference in OWL	25

2.4.2	Available OWL Tools	26
2.4.3	Discussion	26
2.5	Software Patterns	27
2.5.1	Patterns	27
2.5.2	Requirement Patterns	27
2.5.3	Patterns Categorization	29
2.5.4	Pattern Inference	30
2.5.5	Software Pattern Instantiation	32
2.5.6	Discussion	32
2.6	Software Pattern Composition Process	33
2.6.1	Composition Techniques	33
2.6.2	Discussion	33
2.7	From Requirement Patterns to Software Patterns	34
2.7.1	Selecting Patterns for Requirements	34
2.7.2	Relating Requirements	35
2.7.3	Forces	36
2.7.4	Discussion	36
2.8	Similar Approaches and Supporting Tools	37
2.8.1	Requirements Based Approaches	37
2.8.2	Supporting Tools	37
2.8.3	Discussion	37
2.9	Summary	38
3	The SCARP Approach	41
3.1	Automation of a Model based Process	42
3.1.1	MDA and the Proposed Approach	42
3.1.2	The Entities and Workflows Framework	42
3.2	SCARP Overview	43
3.3	SCARP Parametrization	44
3.3.1	Domain Model Specification	44
3.3.2	Patterns catalogs	47
3.4	Use Cases Specification	47
3.4.1	RUS	49
3.4.2	RUST	50
3.5	OWL Generation	51
3.6	Requirements Patterns	53
3.6.1	Pattern Inference	56
3.7	Software Patterns	57
3.7.1	Goals and Concerns	58
3.7.2	Forces	61
3.7.3	Matching Process	63

3.8	Architectural Solution	66
3.8.1	Software Pattern Definition	68
3.8.2	Instantiation Process	69
3.8.3	Pattern Composition	69
3.8.4	Solution Enhancement	70
3.8.5	Solution Validation	72
3.8.6	Serialization	73
3.9	Code	74
3.10	Summary	75
4	Instantiation of SCARP	77
4.1	uCat Tool	77
4.2	Domain Model Specification	78
4.3	Use Case Specification	80
4.3.1	Entities Extraction	82
4.4	Ontology Creation	83
4.4.1	Types Definition	83
4.4.2	OWL Ontology Generation	85
4.5	Requirement Pattern inference	87
4.5.1	Data Query Mechanism	87
4.5.2	Pattern Specification	87
4.5.3	uQL	88
4.5.4	Pattern Inference	91
4.6	Requirements to Software Patterns	92
4.6.1	Concerns, Goals and Forces definition	92
4.6.2	Matching Process	94
4.7	Software Pattern Instantiation	98
4.7.1	Software Pattern Representation	98
4.7.2	Software Pattern Definition	99
4.7.3	Instantiation Process	100
4.7.4	Composition	103
4.7.5	Enhancement	104
4.7.6	Producing XMI	107
4.8	Generation of Outputs	109
4.8.1	Source Code	109
4.8.2	User Interface Prototypes	110
4.9	Summary	111
5	Validation	113
5.1	Study 1 Description	113
5.1.1	Objectives	115

5.1.2	Study Setup	115
5.1.3	Addressing the Objectives	117
5.1.4	Study Validation	118
5.2	Study 1 Execution	118
5.2.1	Execution of the study	119
5.2.2	Results of the Study	119
5.2.3	Discussion	122
5.2.4	Summary	124
5.3	Study 2 - Pattern Inference Process and Usability	
	Assessment	124
5.3.1	Setup of the Experiment	125
5.3.2	Results of the Experiment	126
5.3.3	Discussion	128
5.4	Threats to Validity	130
5.5	Summary	131
6	Case study	133
6.1	Context	133
6.2	Domain Model	135
6.3	uQL Queries	136
6.4	Software Patterns and Matching Information	138
6.4.1	Concerns	138
6.4.2	Goals	139
6.4.3	Forces	140
6.5	Requirements	143
6.5.1	Register	143
6.5.2	Login	144
6.5.3	Return Home	145
6.5.4	Browse Products	146
6.5.5	View Product	146
6.5.6	Show Highlights	147
6.5.7	View Actions History	147
6.5.8	Search Product	148
6.5.9	Add Product to Cart	148
6.5.10	Checkout	149
6.6	Individuals Classification	149
6.7	Requirement Patterns Inference	150
6.7.1	Inferred Patterns	151
6.8	Produced Solution	152
6.8.1	Parametrization	152
6.8.2	Solution	156

6.9	Discussion	159
6.10	Summary	163
7	Conclusions	165
7.1	Discussion	168
7.1.1	Answer to Research Questions	169
7.1.2	Thesis	170
7.2	Future Work	170
	Appendices	173
A	Inputs and Outputs of SCARP	173
A.1	Domain model	173
A.2	RUST Specification	174
A.3	SPARQL queries	174
A.4	Ontology representing the “ <i>Add product to cart</i> ” use case.	175
A.5	Mapping information	177
A.6	XMI representation of software patterns.	185
B	Inputs and outputs regarding the validation studies	205
B.1	Specifications selected for the first study	205
B.1.1	Use cases	205
B.1.2	Textual descriptions	207
B.2	Scenario descriptions selected for the second study	208
B.3	Requirement patterns catalog	208
B.3.1	Simple Search	208
B.3.2	Catalog	209
B.3.3	Registration	209
B.3.4	List Builder	209
B.4	Software patterns catalog	211
B.4.1	Proxy	211
B.4.2	Command	212
B.4.3	Memento	214
B.4.4	Iterator	215
B.4.5	Composite	217
B.4.6	Flyweight	218
B.4.7	Singleton	220
B.5	Requirement Pattern to Software Pattern Matching Information	221
B.6	Diagrams used in the validation of the produced solution	230

List of Figures

1.1	<i>Waterfall</i> [6] (on the left) and <i>V</i> [36] (on the right) software development methodologies.	2
1.2	<i>Spiral</i> [10] (on the left) and <i>Agile</i> [5] (on the right) software development methodologies.	2
1.3	The SCenario bAsed Rapid software Prototyping (SCARP) approach, as part of a model driven development process.	4
2.1	Overview of the MDA process [70].	13
2.2	Use case diagram (left) and description (right), for the scenario of describing the registration of a product on an eCommerce platform.	15
2.3	LIDA system architecture (adapted from [92]).	19
2.4	Example of Essential Use Case (EUC) to Essential User Interface prototyping (adapted from [65]).	20
2.5	Simplified representation of an ontology.	22
2.6	Representation of a simple ontology (describing the information in Listing 2.4).	24
2.7	Patterns as bridges between problems/contexts and solutions.	27
2.8	Overview of Withall’s pattern catalog (from [120]).	28
2.9	Categorization of software patterns, according to their abstraction level (higher on the bottom, lower on the top) [27].	30
2.10	Design pattern inference approach, adapted from [22].	31
2.11	Pattern composition techniques.	33
2.12	Adaptor and Composite patterns composition by a) Stringing and b) Overlapping.	34
3.1	The SCARP process.	43
3.2	Artifacts in the <i>Application Domain</i>	44
3.3	Setup of SCARP - providing the domain model.	45
3.4	A domain model for an eCommerce context.	46
3.5	SCARP process inputs.	47
3.6	Specification of user requirements step.	48
3.7	Step 1 of SCARP - creating the RUS specification.	48
3.8	Generating the OWL knowledge base.	52

3.9	Step 2 of SCARP - creating the OWL ontology.	52
3.10	Relation between User , System and Actor in the domain model.	53
3.11	Inference of requirement patterns.	54
3.12	Step 3 of SCARP - inferring the requirement patterns.	54
3.13	Representation of a statement in a) Natural language, b) uQL and c) SPARQL Protocol and RDF Query Language (SPARQL).	56
3.14	Transition from requirement patterns to software patterns.	57
3.15	Step 4 of SCARP - Transition from requirement to software patterns.	58
3.16	Attributes of patterns at different levels.	58
3.17	Requirement patterns, concerns and forces for the <i>HasShoppingCart</i> requirement pattern.	63
3.18	Software patterns, goals and forces relationship for the <i>Proxy</i> software pattern.	64
3.19	Software patterns, goals and forces relationship for the <i>Memento</i> software pattern.	64
3.20	Software patterns, goals and forces relationship for the <i>Command</i> software pattern.	64
3.21	Requirement patterns to software patterns process flow.	64
3.22	Summary of the relations between concerns, forces, goals and software patterns, for the <i>HasShoppingCart</i>	66
3.23	Production of the architectural solution.	67
3.24	Step 5 of SCARP - producing an architectural solution from a set of software patterns.	67
3.25	<i>Proxy</i> design pattern structure (adapter from [41]).	68
3.26	<i>Proxy</i> design pattern instantiation.	69
3.27	Composition of <i>Proxy</i> , <i>Memento</i> and <i>Command</i> patterns through overlapping.	70
3.28	Relationship between System and Product extracted from the domain model.	71
3.29	Enhanced version of the solution resulting from the pattern composition process, regarding relationships.	71
3.30	Relationship between User , Age , Name and Address extracted from the domain model.	71
3.31	Enhanced version of the solution resulting from the pattern composition process, regarding relationships and attributes.	72
3.32	Revised solution	73
3.33	Generating source code.	74
3.34	Step 6 of SCARP - producing source code.	74
4.1	uCat architecture.	78
4.2	Domain model plugin user interface.	79
4.3	Indication of an error in the domain model.	79
4.4	Use case diagram specification interface.	81
4.5	Use case scenario specification interface.	81
4.6	Entities visualization plugin.	82
4.7	Types definition plugin, with the automatically extracted information.	84

4.8	Types definition plugin with all types defined.	85
4.9	Ontology generator plugin.	87
4.10	Reasoner plugin depicting a SPARQL query to infer object types, and corresponding result.	88
4.11	uQL requirement pattern specification user interface.	91
4.12	Requirement pattern inference plugin user interface.	92
4.13	Requirement pattern to software pattern matching process.	93
4.14	Interface to create the Requirement Pattern, Software Pattern, Concern, Goal and Forces mapping information.	95
4.15	Requirement pattern to software pattern process flow.	96
4.16	Architectural mapping plugin user interface.	98
4.17	<i>Proxy</i> software pattern structure (adapted from [41]).	99
4.18	<i>Proxy</i> software pattern instantiation.	101
4.19	<i>Memento</i> software pattern instantiation.	101
4.20	<i>Command</i> software pattern instantiation.	102
4.21	Software pattern instantiation user interface.	102
4.22	UML representation of the architecture resulting from the pattern composition process (with merged entities in gray).	105
4.23	Enhancement of the solution via <i>stringing</i> merging operator.	106
4.24	Excerpt of enhanced solution with additional attributes.	106
4.25	Types definition plugin user interface.	107
4.26	Representation in ArgoUML of the XMI produced in SCARP.	109
4.27	Usage of SCARP outputs in order to support the generation of presentation code.	110
4.28	Example of MODUS web page, generated from an output of SCARP.	111
5.1	RUS validation process.	115
5.2	Questionnaire results for numeric questions.	121
5.3	Questionnaire result for Likert scale questions.	121
5.4	Results before and after adjusting the percentages, for the inference of the pattern catalog.	129
5.5	Patterns inference resulting percentages (average values and standard deviation) for <i>Simple Search</i> , <i>Catalog</i> , <i>Registration</i> and <i>List Builder</i> requirement patterns.	129
6.1	Amazon main page.	134
6.2	Amazon shopping cart page.	134
6.3	Domain model for the eCommerce domain, in uCat.	135
6.4	Case study's use case diagram.	144
6.5	" <i>Register</i> " use case description.	144
6.6	" <i>Login</i> " use case description.	145
6.7	" <i>Return home</i> " use case description.	145
6.8	" <i>Browse product</i> " use case description.	146

6.9	“View product” use case description.	146
6.10	“Show highlights” use case description.	147
6.11	“View actions history” use case description.	147
6.12	“Search product” use case description.	148
6.13	“Add product” to cart use case description.	148
6.14	“Checkout” use case description.	149
6.15	Requirement pattern inference result.	151
6.16	Resulting forces matrix.	152
6.17	Resulting inferred software patterns, with best matches highlighted.	152
6.18	Parametrization of the <i>Proxy</i> pattern, for the <i>HasShoppingCart</i> requirement pattern.	152
6.19	Parametrization of all software patterns.	154
6.20	Types for the inferred properties.	155
6.21	Results of the validation study performed in the generated model.	156
6.22	Resulting architecture from the serialization process, with identification of requirement patterns.	158
6.23	Main page of the user interface prototype.	161
6.24	Listing of products in the user interface prototype.	162
B.1	Software patterns template.	211
B.2	<i>Proxy</i> pattern structure.	212
B.3	<i>Proxy</i> pattern instance.	212
B.4	<i>Command</i> pattern structure.	213
B.5	<i>Command</i> pattern instance.	214
B.6	<i>Memento</i> pattern structure.	214
B.7	<i>Memento</i> pattern instance.	215
B.8	<i>Iterator</i> pattern structure.	216
B.9	<i>Iterator</i> pattern instance.	216
B.10	<i>Composite</i> pattern structure.	217
B.11	<i>Composite</i> pattern instance.	218
B.12	<i>Flyweight</i> pattern structure.	219
B.13	<i>Flyweight</i> pattern instance.	220
B.14	<i>Singleton</i> pattern structure.	220
B.15	<i>Singleton</i> pattern instance.	221
B.16	Diagram A.	230
B.17	Diagram B.	230
B.18	Diagram D.	231
B.19	Diagram E.	231
B.20	Diagram F.	232

List of Tables

2.1	Extract of ACE to MLL mapping [73].	16
2.2	Example of conflicts between requirements (adapted from [85]).	35
2.3	Overview of the related work.	39
2.4	Comparison of existing approaches.	40
3.1	RUS formatted “ <i>Add product to cart</i> ” use case.	49
3.2	Result of applying uQL in Listing 3.2 to the knowledge base.	57
3.3	Positive contributions between requirement patterns and software patterns (through forces), as support for the matching process.	65
3.4	Negative contributions between requirement patterns and software patterns (through forces), as support for the matching process.	65
3.5	Matrix describing the relations between forces for the <i>HasShoppingCart</i> requirement pattern.	66
4.1	Example of forces matrix relating goals to concerns.	97
5.1	Questionnaire - part a).	114
5.2	Questionnaire - part b).	114
5.3	Excerpt of a RUS use case specification used in the study.	116
5.4	Study objectives and corresponding tasks.	118
5.5	Excerpt of a collected use case and its RUS version.	120
5.6	RUS evaluation.	121
5.7	Excerpt of use case made by a participant.	121
5.8	Detailed SUS results.	127
6.1	Requirement patterns and corresponding concerns.	140
6.2	Concerns’ forces.	142
6.3	Goals’ forces.	142
6.4	Types specification.	150
B.1	Modify players information.	205
B.2	Cancel a tournament.	206

B.3 Check donative information. 206
B.4 Remove a user. 206
B.5 Mark donative as used. 206
B.6 Change a project. 206
B.7 Allocate items to a certain project phase. 207
B.8 Check a job information. 207

Listings

2.1	Example of a ACE statements [73].	17
2.2	Example of a Gherkin feature describing “ <i>Refund item</i> ” [126].	17
2.3	Example of Boilerplate template [60].	17
2.4	Example of a simple OWL ontology describing the information of a user which clicks in a link.	24
2.5	Example of an SPARQL query to list which objects relate via <i>click</i> object property to any subject.	25
3.1	RUST used for the specification of “ <i>Add product to cart</i> ” use case.	51
3.2	uQL <i>HasShoppingCart</i> requirement pattern.	56
3.3	Proxy software pattern specification in API-like format.	68
4.1	Excerpt of OWL representing the domain model in Figure 4.2.	80
4.2	SPARQL query to identify types for individuals.	83
4.3	SPARQL query to identify the types for User	83
4.4	Excerpt of “ <i>Add product to cart</i> ” use case formalized in OWL	86
4.5	Excerpt of an Use Cases Query Language (uQL) query representing the <i>HasShoppingCart</i> requirement pattern.	89
4.6	Result of the translation of the uQL in Listing 4.5 query to SPARQL.	90
4.7	Proposal of algorithm to automatically generate uQL from a set of specifications.	90
4.8	Excerpt of the ontology supporting the matching process.	94
4.9	Queries to identify positive and negative relations between requirement and software patterns.	96
4.10	Excerpt of the representation of the <i>Proxy</i> software pattern in XMI.	99
4.11	Representation of the <i>Memento</i> software pattern in API-like format.	100
4.12	Representation of the <i>Command</i> software pattern in API-like format.	100
4.13	Excerpt of XMI resulting from composition of <i>Command</i> and <i>Memento</i> software patterns.	103
4.14	SPARQL query to extract individuals of the category <i>TypeOf</i>	104
4.15	SPARQL query to extract individuals of the category <i>PropertyOf</i>	106
4.16	Excerpt of the XMI resulting from the SCARP process.	108
4.17	Excerpt of the Java code generated by ArgoUML, from the produced XMI.	110
5.1	<i>Simple Search</i> requirement pattern, written in uQL.	128

6.1	“ <i>HasAccount</i> ” uQL requirement pattern.	136
6.2	“ <i>HasShoppingCart</i> ” uQL requirement pattern.	136
6.3	“ <i>HasCatalog</i> ” uQL requirement pattern.	137
6.4	“ <i>HasDetails</i> ” uQL requirement pattern.	137
6.5	“ <i>HasSearch</i> ” uQL requirement pattern.	137
6.6	“ <i>HasHighlights</i> ” uQL requirement pattern.	137
6.7	“ <i>HasUpload</i> ” uQL requirement pattern.	138
6.8	“ <i>HasFriendship</i> ” uQL requirement pattern.	138
A.1	Representation of the domain model in OWL.	173
A.2	RUST specification used in this work.	174
A.3	SPARQL query to identify individuals related via <i>TypeOf</i>	174
A.4	SPARQL query to identify individuals related via <i>CompositionOf</i>	174
A.5	SPARQL query to identify individuals related via <i>PropertyOf</i>	175
A.6	Representation in Web Ontology Language (OWL) of the “ <i>Add product to cart</i> ” use case.	175
A.7	Ontology representing the matching information.	177
A.8	XMI representation of the <i>Command</i> software pattern.	185
A.9	XMI representation of the <i>Composite</i> software pattern.	188
A.10	XMI representation of the <i>Flyweight</i> software pattern.	190
A.11	XMI representation of the <i>Iterator</i> software pattern.	194
A.12	XMI representation of the <i>Memento</i> software pattern.	197
A.13	XMI representation of the <i>Proxy</i> software pattern.	200
A.14	XMI representation of the <i>Singleton</i> software pattern.	203
B.1	uQL <i>Simple Search</i> requirement pattern.	208
B.2	uQL <i>Catalog</i> requirement pattern.	209
B.3	uQL <i>Registration</i> requirement pattern.	209
B.4	uQL <i>List Builder</i> requirement pattern.	209
B.5	OWL matching information.	221

Acronyms

ACE	Attempto Controlled English
API	Application Programming Interface
CNL	Controlled Natural Language
CPN	Colored Petri Net
CMS	Content Management Systems
DL	Description Logics
EUC	Essential Use Case
EFW	Entities and Workflow Framework
HSC	Has Shopping Cart
MDA	Model Driven Architecture
Modelery	Models Refinery
MBUID	Model Based User Interface Development
MODUS	MOdel-based Developed User Systems
NFR	Non-Functional Requirements
OMG	Object Management Group
OWL	Web Ontology Language
PIM	Platform Independent Model
PSM	Platform Specific Model
RDF	Resource Description Framework
SWEBOK	Software Engineering Body Of Knowledge
SPARQL	SPARQL Protocol and RDF Query Language
SCARP	SCenario bAsed Rapid software Prototyping
SUS	System Usability Scale
SQL	Structured Query Language
uCat	Use Cases Analysis Tool
UML	Unified Modeling Language
uQL	Use Cases Query Language
URL	Uniform Resource Locator

W3C	World Wide Web Consortium
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Chapter 1

Introduction

1.1 Context

Software engineering is defined by the IEEE as “*The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; (...)*” [99]. The software development process, in particular, corresponds to the set of tasks required to achieve software solutions. As defined in the guide to the Software Engineering Body Of Knowledge (SWEBOK), the process is composed of several phases, including requirements specification, software design, construction, testing and maintenance [11].

Several software development methodologies exist in order to support the software development process. For instance, Pressman [94] categorizes the approaches, or, software development models as *Linear Sequential, Prototyping, RAD, Evolutionary, Component-based, Formal methods* and *Fourth Generation Techniques*. These methodologies can be distinguished from each others by, for instance, how they approach the several phases of the development process, what outputs they create, which kind of models are used, how the development process is managed, and how tests are performed.

Linear methodologies are an example of classical approaches (c.f. Figure 1.1). These methodologies present processes where the development tasks are performed in sequential order. Changes on a previous step imply restarting the processes at that point, since no iteration is envisaged.

A more flexible approach to software development is the one proposed by evolutionary methodologies (c.f. Figure 1.2). In these methodologies, software is produced by a continuous iterative process, which at each cycle adds new features to the final solution.

In the model driven development methodologies (e.g. Model Driven Architecture (MDA)), the software development process is based in models. Models in these methodologies are the main inputs, and their transformation is what produces results. Other approaches resort also to

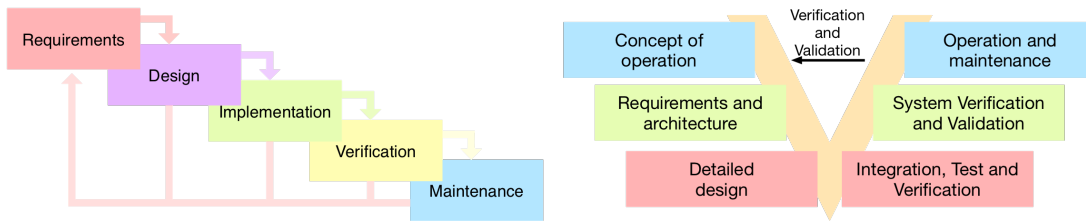


Figure 1.1: *Waterfall* [6] (on the left) and *V* [36] (on the right) software development methodologies.

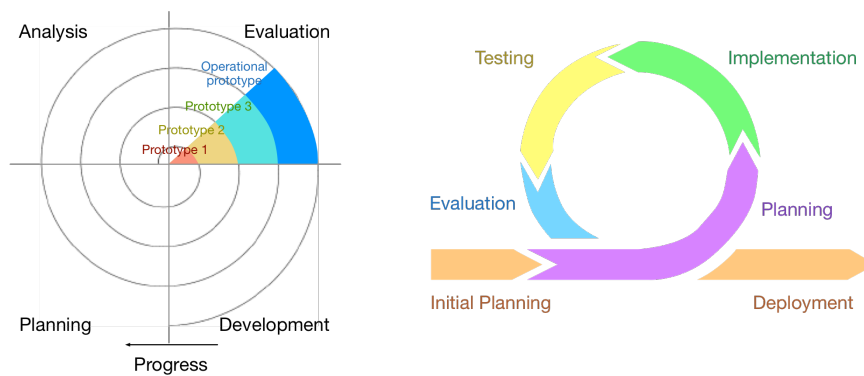


Figure 1.2: *Spiral* [10] (on the left) and *Agile* [5] (on the right) software development methodologies.

models, but not as the primary inputs. In the MDA, different models are defined (e.g. structural and behavioral), and consecutively refined and transformed into other models and into source code, resulting in the production of the software solutions [91]. Using architectural models to create software solutions introduces formalism in the process, making it more systematic and rigorous.

The relevance of model driven development approaches in practice, is supported by its acceptance in industry [106]. Thus, it is worth noting that contributions for the MDA are not only relevant from a theoretical point of view, but are also relevant in practice.

An interesting aspect of MDA, is the possibility of automating the models transformation processes. Indeed, part of its steps and processes have been automated, both regarding the transformation of models, generation of source code, and even reverse processes (as generation of models from source code). Tools and approaches (e.g. [121]) have contributed to the success of the automation, improving the development process itself both in time, costs and quality [70, 106, 117]. More specifically, existing contributions support the transformation of architectural models, into source code. Despite the MDA advances, room for improvement can still be found. Requirements are one of the areas where enhancements can be made.

1.2 Problem

Approaches and tools that support architectural models transformation improve the systematization and quality of the development process, by providing support for rigorous transformations. Requirement models, while relevant for the software development process, have not been addressed with the same emphasis in this MDA vision. Indeed, a gap between requirements models and architectural models can be found, regarding systematic and automatic models transformations between the two levels.

The creation of architectural models, in the context of MDA, is based on the analysis of requirement models. This transition is manually performed, due to the lack of systematic and automated transformation processes. It is acknowledged that manual transition processes are susceptible to errors resulting from interpretation [34]. Although existing approaches have proven to successfully support such a manual processes, the advantages of systematizing the transition should be taken into account.

Software requirements models exist in the same space as architectural models, but the lacking of a systematic transition process can be identified. The transformation between these two kind of models could improve the MDA process, including the requirements in the automatic and systematic transformation chain.

1.3 Proposal

The objective of this work is to tackle the aforementioned issues in the transformation of requirement models into architectural models, by providing the same level of automation existing in the transformation of architectural models in the MDA. The automation supports the inclusion of requirement specifications as part of the MDA process. Such results in an extension of the process, which starts the automated development process in the requirements, rather than in architectural models.

Integrating the requirement models in the MDA provides several advantages. Providing a systematic process to convert requirement models into architectural models supports the mitigation of errors resulting from interpretation, therefore reducing the subjectivity in the process. Automating the process supports a more efficient iterative development process, while reducing errors resulting from manual transformations. When requirements are changed, the effort to specify new models of the proposed solution, or adjusting existing ones is reduced.

The introduction of formalism in the process, as proposed by this work, is acknowledged to improve the quality of the software solutions. At the beginning of the software engineering related tasks, it establishes the basis for a more rigorous development process, hopefully resulting in higher quality solutions. A formalization language with the capability to be understood both by users and developers establishes a common ground for communication, without the cost of classical formal notations. Finally, the formal layer enables also the possibility to apply analysis, validation and verification techniques over the models.

The SCenario bAsed Rapid software Prototyping (SCARP) framework is proposed in order to integrate requirement models as part of the MDA chain (c.f. Figure 1.3). SCARP presents a sequential process composed of several phases, from the requirements specification (c.f. Figure 1.3 *Use cases*), to the architectural solutions production (c.f. Figure 1.3 *Architectural Solution*). This framework is a comprehensive process, which starts with requirement models specification, until the production of architectural artifacts. In this work emphasis is put in the steps concerning the formalization and transformation of the requirement models. For the other steps (e.g. architectural model transformation), it takes advantage of the work already developed by using the MDA (c.f. Figure 1.3 *MDA*), in the context of architectural model specification, transformation and operationalization.



Figure 1.3: The SCenario bAsed Rapid software Prototyping (SCARP) approach, as part of a model driven development process.

In order to support requirement models transformation within the MDA framework, in SCARP, requirements are written in a language with support for automation, while maintaining the readability. The objective of focusing on readability is to support the specification tasks, without the need for complex formats. The automation aspects support the systematic and automatic interpretation of the requirements.

SCARP assumes that the provided requirements have previously been object of study through requirements engineering techniques. They have therefore been analyzed and specified. Similarly, at the end of the SCARP, support for the code generation steps is not addressed, since (for instance) MDA approaches deal with the tasks of operationalizing models. Instead, the proposed approach focus in the generation of architectural models. The resulting architectural models, as architectural prototypes, should then be refined and iterated as part of the software development process.

1.3.1 Thesis

This work proposes to integrate requirement models as part of the MDA process, and as result make requirement models inputs of the process. Currently, the MDA does not provide to requirement models the same emphasis provided to other models, as computable inputs. This integration is possible by providing a framework to automate the transformation of requirement models into architectural models. As result, requirement model are put at the same level of architectural models, in the MDA process. Such is achieved by, on the one hand, providing a language to formalize requirements, and enable their analysis. On the other hand, by providing a systematic process to derive the appropriate architectural models, required for the MDA process.

The mitigation of errors resulting from the manual transformation processes, required to produce architectural models from requirement models, is a relevant motivation in this work. Currently, in practice architectural models are produced by manual processes. As a consequence, they are vulnerable to subjectivity. Furthermore, manual processes result in costs. Automating the transformation process provide two contributions. On the one hand, reduces the interpretation and transformation errors, and on the other hand, reduces the costs of producing architectural models from requirement models. This MDA extension reduces also the costs of changes in requirement models, since the architectural models can be automatically produced. A model driven (compliant) approach, based in the user requirements and in the domain model informations is proposed.

Thesis *“It is possible to create a framework, based on the MDA and support its extension, by including requirement models transformation as part of the process. Functional requirements, specifically use cases, have the required properties in order to be formalized and support automatic transformation techniques, which support their integration in MDA.”*

1.3.2 Research Questions

When defining an approach to prove the aforementioned thesis, a set of research questions emerges. These questions, on the one hand, concern to the viability of the proposed approach and, on the other hand, help shape the course of the work, by defining a set of guiding lines, as a mean to answer those questions. The set of research questions approached in this thesis is the following one.

Question 1 *Is it possible to have a simple, yet expressive language for use case specification, with support for automation?*

It is proposed to explore if use cases can be expressed in a simple language, in order to provide easy automation, while keeping their original information. This question is addressed in Sections 3.4, 3.5, 4.3 and 5.2.

Question 2 *Is it possible to perform software requirement patterns inference over use cases, more specifically in the knowledge base representing them?*

It is proposed to explore if the OWL, and consequently the use cases, contain the required information to support requirement patterns inference, and extraction of meaningful information about the architectures which support them. This question is addressed in Sections 3.6, 4.5 and 5.3.

Question 3 *Can requirement patterns be used to select a set of architectural patterns solving them?*

It is proposed to explore if requirement patterns contain relevant enough information to enable selection of the appropriate software patterns, that support the features they describe. This question is addressed in Section 3.7 and 4.6.

Question 4 *Can a set of architectural patterns be instantiated and combined, within a defined context, in order to achieve prototype architectures?*

It is proposed to explore whether the use cases provide enough information to support a meaningful pattern composition, leading to architectures that are compliant with that use cases. This question is addressed in Section 3.8 and 4.7.

1.3.3 Objectives

In order to answer the proposed research questions, a set of objectives is presented. The objectives go towards supporting each step of the SCARP process. Specifically, regarding the requirements specification, the information inference mechanism, and making the bridge to architectural models. Those objectives go also towards answering the research questions, as follows.

1. Validate the viability of using a simple language to support formal use case specifications, which can be automatically formalized (i.e., transformed into a computable format);
2. Create a software requirement patterns inference mechanism, which enables the possibility to perform pattern inference on a knowledge base;
3. Implement a software requirement patterns to architectural pattern mapping technique, to allow achieving a set of architectural patterns from the knowledge base;
4. Define an architectural pattern instantiation and composition technique, in order to achieve architectural solutions from a set of architectural patterns;
5. Present a systematic and automated process which combines the aforementioned objectives into a single platform.
6. Develop a tool to support the automated process.

1.4 Contributions

The work presented in this thesis addresses several knowledge areas. In each of the addressed areas contributions were made. These contributions are now briefly described. A number of scientific publications resulted from this work and are referenced when relevant, and listed at the end of the section.

Regarding requirements specification, and specifically use cases, a contribution in the form of a language and process was made. Specifically, a Controlled Natural Language (CNL) to support the specification of use case scenarios was developed and validated [24, 26, 28]. The language supports automatic translation into OWL.

There are also contributions on requirements and software patterns. A contribution of this thesis is a process to support the association of requirement patterns to software patterns, in a systematic way. The implementation of a patterns' catalog was started, as a way to support the process. Similarly, an approach to automatically compose those patterns was implemented. Finally, and regarding software patterns, a survey regarding existing types of patterns (and corresponding templates to describe them) was performed. As a result, a hierarchization of existing types of software patterns, and a unified template to specify software patterns were achieved [27].

The Use Cases Analysis Tool (uCat) is another contribution of this work¹. The tool supports the aforementioned SCARP process, in an automated way. The tool serves at the same time as a proof of concept, and as a reference implementation [24, 26].

uCat's development has provided inputs for the development of two other tools. MModel-based Developed User Systems (MODUS), a tool and approach for Model Based User Interface De-

¹Website: <http://myopenx.di.uminho.pt/ucut/>

velopment (MBUID), supports the process of prototyping user interfaces, based on a structural model and domain information. The MODUS approach was influenced by SCARP and can be used to complement it. The outputs from the SCARP can be provided as inputs for MODUS, thus, introducing the generation of user interfaces in the process.

Additionally the uCat tool includes support for integration with the Modelery, a repository for model driven development artifacts, for instance, models and patterns [25, 21]. Storing these artifacts and making them publicly available to other developers fosters both reuse and improvement of knowledge. The integration with the repository enables the possibility to export/import models directly from the online repository.

In total, this work resulted in 7 scientific publications.

- Rui Couto *et al.* *MapIt: A model based pattern recovery tool*. In MOMPES 2012 (see [22]).
- Rui Couto *et al.* *A patterns based reverse engineering approach for java source code*. In SEW 2012 (see [23]).
- Rui Couto *et al.* *Application of ontologies in identifying requirements patterns in use cases*. In FESCA 2014 (see [24]).
- Rui Couto *et al.* *A study on the viability of formalizing use cases*. In QUATIC 2014 (see [26]).
- Rui Couto *et al.* *The modelery: A collaborative web based repository*. In ICCSA 2014 (see [25]).
- Rui Couto *et al.* *The modelery: a model-based software development repository*. In IJWIS 2015 (see [21]).
- Rui Couto *et al.* *Validating an approach to formalize use cases with ontologies*. In FESCA 2016 (see [28]).

Additionally, a technical report was created during this work, and the collaboration on the development of MODUS resulted in a scientific publication.

- Rui Couto *et al.* *A survey on software patterns*. Technical report, HASLab INESC TEC and Universidade do Minho, 2016 (see [27]).
- Marina Machado *et al.* *Modus: uma metodologia de prototipagem de interfaces baseada em modelos*. In INFORUM 2015 (see [84]).

1.5 Document Structure

Besides this introduction, this document is organized in six additional chapters. Chapter 2 presents an overview of related and background work. Work in the areas approached by this thesis is presented, in order to provide an overview about existing approaches, methodologies and tools.

Chapter 3 presents SCARP, the proposed approach to integrate requirements models in MDA. It starts by presenting an overview of the process, and next details each of the steps of the approach.

Chapter 4 presents both how SCARP can be instantiated with concrete input and output formats, as well as the uCat tool. uCat supports the SCARP approach, by providing an interface to support each step of the approach, regarding the specified input and output formats. uCat demonstrates the viability of automating the SCARP approach, and provides detailed guidelines on how to achieve such.

Chapter 5 presents the validation of SCARP and uCat. Two studies regarding the validation of several aspects of both the approach and the tool are presented. The specification of requirements and inference of information from the specifications are also addressed.

Chapter 6 presents a case study for SCARP, and corresponding supporting tool uCat. A set of requirements from the eCommerce domain is presented, to which SCARP is applied. The required inputs and outputs for each step are presented, as well as the resulting artifacts. uCat is used in order to demonstrate the study.

Finally, Chapter 7 concludes this document by providing an overall vision of the performed work, viability of the approach and final remarks. This chapter presents also the answer to the research questions, and leave indications for future work.

Chapter 2

Background

This section presents an overview of background work in several areas relevant to the proposed work. Related works include approaches, tools and methodologies, and their contributions to the proposed objectives are analyzed.

In order to better organize the related work, the following areas are addressed.

1. The MDA framework;
2. Requirement specification languages with automation capabilities;
3. Structured representation of requirements;
4. Knowledge inference on requirements representations;
5. Software patterns;
6. Requirements models based approaches.

2.1 The Model Driven Architecture Framework

The MDA [70] is a framework that allows developers to build software systems focusing on the core business of the applications, instead of platform details such as hardware, programming languages, operating systems or frameworks. Specified by the Object Management Group (OMG), it relies on the Unified Modeling Language (UML) and a number of standards. The MDA has the objective of creating machine readable documents, that have computational capabilities, and therefore, can be used to automated means.

The traditional development processes have some drawbacks that the MDA proposes to solve, namely issues with **productivity**, **portability**, **interoperability** and **maintenance** [70]. Next

is explained how each of those issues affects the traditional development processes, and how the MDA solves them.

Productivity In traditional software development methodologies, models are specified at the beginning of the development process, and meant to be used as guides for the remaining process. However, as the development process goes on (with iterations and improvements), developers tend to forget the models, and focus only on the code. Hence, models are “just papers” that represent the system at the end of the design phase, and of little usage. The MDA focus in the models definitions, and systems are directly generated from the models. Hence, reiterations of the process should occur in the model, not at source level, making the model always synchronized with the real software system.

Portability New technologies emerge at a quick pace. In a short period of time it was possible to see the focus of the applications moving from desktop applications, to web-based ones. The increase in the performance of smartphones turned again the focus from web to mobile. More recently, the lower cost and consumption of processors is moving the focus towards wearable devices. There is a big issue raised by this tendency, as already existing software has only two options to continue to exist. Either the code is ported (or reimplemented as a new version), or left as is, as legacy software, raising then interoperability and security issues. As the MDA focus in the models, only new transformations from models to code must be implemented for each new platform, with the advantage that the transformations can be reused.

Interoperability Monolithic systems are mostly abandoned nowadays. Interoperability is a key feature for software systems. However, the integration of several technologies from several vendors is not always straightforward. The MDA proposes that tools generate models and code from models, but also the bridges between them.

Maintenance and documentation On the one hand, traditional methodologies do not encourage writing documentation. It is usually seen as a tedious and undesired task. On the other hand, models are not used as documentation of the code, as they are usually deprecated as reiterations of the development process occur. Again, as the MDA focus on the models, models are itself the documentation of the process (as they are not abandoned). The transformation tools can support bidirectional transformations, allowing changes in the code to be propagated to the models.

The MDA proposes to solve the presented issues with traditional software development approaches by basing the development process on models. It proposes also two abstraction levels to define the models, which provide different visions of the system to be developed.

Platform Independent Models (PIM) are models which are independent of the technology. They focus in the solution itself, rather than on specific details. The MDA puts the development emphasis in models. Then, tools are used to achieve more specific models.

Platform Specific Models (PSM) are models which are closer to source code, as they include details related with the platforms in which they will be deployed. Context should be provided so the tools can generate Platform Specific Model (PSM) from Platform Independent Model (PIM).

2.1.1 The MDA process

Figure 2.1 provides an overview of the MDA process, as well as a summary of the MDA building blocks. The MDA requires the following inputs:

- High level models, defined in standard languages, and containing all the information of the system being modeled;
- Standard languages, to write those models;
- Transformation definitions, both from PIM to PSM, and from PSM to code;
- Languages to write the transformation definitions, to allow a systematic process;
- Tools that implement the execution of the transformation definitions, to allow the transformations;
- Tools that implement the execution of the transformation of PSM to code, to achieve the final solution.

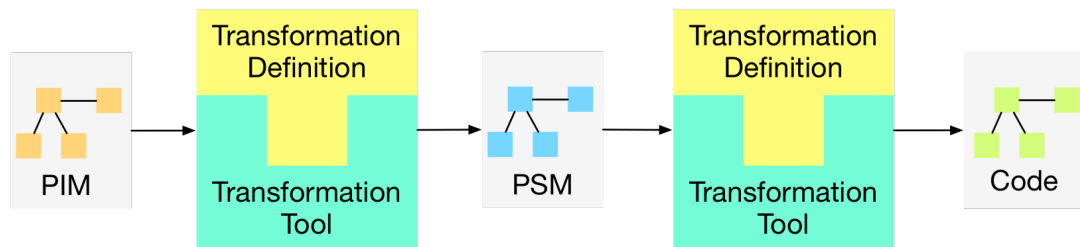


Figure 2.1: Overview of the MDA process [70].

A notable work regarding the MDA is presented by Mellor and Balcer [88]. The authors present an approach, by restricting the specification language, to formalize requirements into verifiable diagrams. From those diagrams is possible to generate code for specific scenarios. Such is possible because the authors present a profile for UML to model structure and behavior of a given system, with the required expressiveness to execute it. The models are as expressive as source code, which enables the automatic generation of code. While suitable for a restricted set of scenarios, this work is a supporting approach for MDA, through industry standards (UML), which shows the viability of MDA.

2.1.2 Discussion

In the MDA, requirement specifications do not have the same emphasis of software models. Indeed, while software models are integrated as part of systematic and automated transformation models, requirements are used as guides to produce software model.

The integration of the proposed approach in the MDA, provides to the proposed approach support not only to use standard technologies to create the models (c.f. UML), but also systematic approaches to refine models until source code. It establishes the basis for an automated process from models until source code.

Several studies have shown the viability of using models to create software solutions (i.e., concretizing the MDA). The approach is more than just a proof of concept, or a good theoretical approach, as it has uses in practice. Those works provide a sound basis for the proposed approach.

2.2 Simplified Languages With Support for Automation

For the proposed approach a simple language with support for automation is required. Representing requirements in simplified formats not always implies losing expressiveness or automation capabilities. Several approaches which propose simplified languages for requirements specification, with support for automation, can be found. This section presents some relevant approaches in the context of this work.

2.2.1 Use Cases

Use cases are a requirements specification methodology based in the description of steps and interactions between different actors and systems. Originally proposed by Jacobson [61], were later adopted and standardized by the OMG. The OMG specified use cases as models for software requirement specifications, through scenarios which describe the interactions between a user and a system (c.f. Figure 2.2), and made them part of the UML.

The OMG does not specify in concrete how use cases should be specified. Such is due to their capability to be used at different stages of requirements analysis and specification, and with different flavors; from more textual description amenable for analysis, to more operational descriptions useful for specification. However, and as use cases describe the system features, several approaches emerged in order to operationalize them.

Tabular representations have proven popular (e.g. as proposed in [38] and [20]). This style of representation is based in an operational actor/system interaction style (as presented in Figure 2.2), where the interactions are described sequentially. Use cases can also contain additional information such as preconditions and postconditions [8].

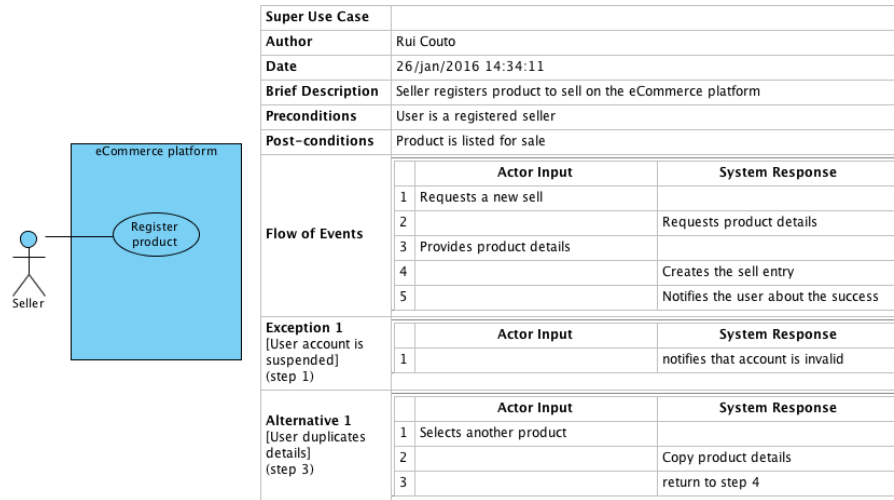


Figure 2.2: Use case diagram (left) and description (right), for the scenario of describing the registration of a product on an eCommerce platform.

Use cases are a powerful approach to describe use case specifications. They provide a simple format for scenarios description. While not directly computable, there have been several proposes in order to improve their specifications, namely the tabular formats. Although, as they are usually written in natural language, they can be combined with approaches which support to formalize and operationalize the requirement specifications, making them amenable to being operationalized and formalized.

2.2.2 Natural Language Processing

One of the most common approaches to support the formalization of software requirements is the application of natural language analysis techniques. Natural language analysis techniques provide the capability to analyze a wide variety of specifications, since they do not rely on any predefined format. These techniques analyze relevant keywords and relations between them, in order to create a meaningful structure. Natural analysis techniques are commonly used to generate architectural models [108, 77], and also to generate formal models as support for applying verification techniques [42, 125]. Some works require some kind of preprocessing on the specifications (e.g. annotation), in order to improve the descriptions, prior to their analysis [101].

The application of natural language processing techniques supports the analysis of existing specifications. However, it is acknowledged the variability of the results when applying these techniques. Providing some structure on these kind of specifications could improve this aspect. The same is true for the complexity of the techniques required to perform both kind of analysis. While analyzing natural language requires specialized tools, analyzing restricted formats can be significantly easier and provide more predictable results.

2.2.3 Computable Formats

Not only textual representations have been used to represent requirements. An alternative is to use some kind of computable format, such as computational models. Representing requirements in this format has the clear advantage of being a direct computable and formalized format, therefore no details are lost in translation or interpretations processes. The major drawbacks are the need for users which clearly understand the modeling language, and a possible loss in readability. It is possible to find authors addressing the modeling of structural and functional aspects, for instance resorting to Colored Petri Net (CPN) [63] or UML extensions [110, 111, 111].

Computable formats reduce the work needed to analyze and formalize a specification. Ultimately reducing the need for translation processes, and loss of information in that processes. However, writing this kind of specifications require all the participant users to understand what is being specified. Consequently, the readability of the specifications is also affected. This kind of specification would benefit from a more flexible format in order to improve the overall readability.

2.2.4 Intermediary Languages

A possible approach to operationalize requirements is to write them in a language which supports their automatic formalization, for being close to a computable format. These kind of formats exist in between the computable formats, which are directly computable, and natural language which requires performing textual analysis. An example of such language is Attempto Controlled English (ACE) [73], a language based in OWL. ACE specifications are close to the same code in OWL, however with improved readability. This approach enables an easy automatic formalization process. ACE is also directly mappable into OWL, while supporting specifications close to natural language (c.f. Table 2.1). The table presents the same expression in ACE and in Manchester-like Language (MLL). MLL is a simplification of the Manchester language [56], one of the languages available to specify OWL ontologies.

Table 2.1: Extract of ACE to MLL mapping [73].

ACE	MLL
$\langle I \rangle$ is a $\langle T \rangle$.	$\langle I \rangle$ HasType $\langle T \rangle$
$\langle I \rangle$ is not a $\langle T \rangle$.	$\langle I \rangle$ HasType not $\langle T \rangle$
$\langle I_1 \rangle \langle R \rangle \langle I_2 \rangle$.	$\langle I_1 \rangle \langle R \rangle \langle I_2 \rangle$
$\langle I_1 \rangle$ does not $\langle R \rangle \langle I_1 \rangle$.	$\langle I_1 \rangle$ not $\langle R \rangle \langle I_1 \rangle$
$\langle I \rangle \langle R \rangle$ a $\langle T \rangle$.	$\langle I \rangle$ HasType $\langle R \rangle \langle T \rangle$

It is easy to understand that this kind of specifications while specifying the format of the supported specifications, restrict the kind of inputs that they support. However, at the same time, they make the process of automatically analyzing the specifications easier.

Hence, a possible approach to achieve a simple language with support for automated analysis is to define a subset of another more complex language (e.g. natural language). Following one such approach, the resulting specifications are more controlled, since they follow a predefined format. The adoption of these languages might limit the complexity of statements. Being the input formats predefined, naturally the specifications cannot have the same complexity of natural languages. Limiting the complexity of statements is not necessarily a negative aspect, and also less complexity is not synonymous of less expressiveness (as demonstrated in [7]).

An approach to define a subset of another language is by reducing the original statements to simpler or more strict formats [20, 87]. For instance, as proposed by Kuhn [73], the ACE is a restricted language, with support for the specification of simple statements (e.g. Listing 2.1). These statements have a high expressiveness, while supporting automation. Another popular approach which resorts also to a predefined input format, with support for automation is Gherkin [126]. In Gherkin, the users input scenarios (c.f. *features*, in Listing 2.2), which have a predefined specification format. The format in which the scenarios are specified supports automatic analysis and generation of Ruby code blocks, in order to perform code verification.

```
Kate sees no officer.  
Kate sees Bill.  
John buys a present.
```

Listing 2.1: Example of a ACE statements [73].

```
Feature: Refund item  
  Scenario: Jeff returns a faulty microwave  
    Given Jeff has bought a microwave for $100  
    And he has a receipt  
    When he returns the microwave  
    Then Jeff should be refunded $100
```

Listing 2.2: Example of a Gherkin feature describing “*Refund item*” [126].

Similarly to Gherkin, boilerplates are an approach to describe functional requirements in a specific format [60]. Specifying a requirement consists in selecting an appropriate template (see Listing 2.3 for an example), and filling the placeholders (e.g. <stakeholder type>). Boilerplate templates are categorized by their objective, and creating a requirement specification consists in selecting the most appropriate template. Although not designed to support use case specifications, boilerplates successfully show how a language based in a simple statements is suitable to create functional requirements as scenario descriptions.

```
The <stakeholder type> shall be able to <capability>  
within <performance> of <event>  
while <operational condition>.
```

Listing 2.3: Example of Boilerplate template [60].

Complex specifications can also be transformed into simpler ones by reducing the set of allowed words in the descriptions [109]. This approach has two major effects. On the one hand, specifications are easier to understand, since the number of synonyms, or even ambiguous words can be removed. On the other hand, it supports to index the specifications to a domain. Naturally, a subset of words is required for creating specifications in each domain. Also, users are more restrict while creating the specifications. Reducing terms in requirement specifications provides the advantage of producing more consistent specifications. Such provides simpler specifications, which are easier to understand by users within the same context.

The usage of CNLs is an interesting approach. On the one hand, it supports specifications in formats which do not impact expressiveness, and are therefore suitable for expressing requirements. On the other hand, specifications written in this format are easier to automate. Naturally, being the formats restricted and predefined, a learning process is required in order for the users to adapt to the language. From the existing approaches and formats, none was designed with the specification of use cases in mind. However, a possibility is to select an existing approach (such as ACE, for instance), in order to support use case scenarios specification.

2.2.5 Annotation

Enhancement of existing specifications in order to perform their formalization is also a recurring approach to support requirements formalization. Annotations help in the analysis process by adding meaning to the terms in specifications. The verification process can then take advantage of annotations in order to check the validity and consistency of specifications. Annotations have several usages, such as defining the meaning of words [72], which later support the generation of corresponding models. Some other works require preprocessing specifications, in order to normalize them according to a certain format [78, 64]. The normalization process results in specifications that are easier to analyze by tools and, therefore, ease the generation of models with support for formalization. It is also possible to find works which improve existing templates (e.g. for use cases [115]), in order to gather more information supporting the formalization process.

Adding meaning to terms in the specifications eases the process of analyzing such requirements. However, manual interaction processes, while providing flexibility, can also be source a of errors. In the case of the annotation process, a full understanding of both the requirements and the context is required. An alternative is to define in beforehand the meaning for terms, in order to support the process. This way, during the specification the input of the users can be reduced.

2.2.6 Operationalization Approaches

The approach proposed in this thesis advocates the usage of use cases as input, in order to produce software architectures. Several approaches propose the usage of use cases as the basis for

the development process (c.f. *use case-based software development*). One of the most influential works regarding use case centered software development was developed by Jacobson [61]. In his work, use cases play the central role to support the development process, from the design phase until testing. Unlike what is the goal here, development progresses through manual steps. Approaches based on use cases have the advantage of producing outputs closer to users' specifications. Furthermore, these approaches support the viability of achieving software outputs in a systematic way, from use case descriptions. However, specifications are prone to errors, and do not contain all the required information to produce adequate solutions. Hence, an adequate design of the methodologies to extract the relevant information, and gather the remaining ones, is required. Most approaches focus on making a direct translation from use cases to architectural outputs. Naturally, performing this step without intermediary representations, hardens the viability to perform any kind of verification prior to generating the outputs.

Several approaches which provide support to automatically generate UML models from specifications can be found in the literature [98, 75, 49, 87, 31]. Overmeyer *et al.* present an interesting approach to generate UML diagrams, the LIDA methodology (c.f. Figure 2.3). The methodology is based in the classification of different words in the specification (e.g. `class`, `attribute`, etc.), and through textual analysis, architectural models are produced [92].

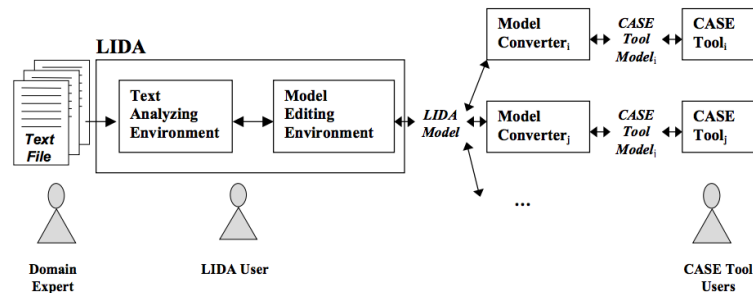


Figure 2.3: LIDA system architecture (adapted from [92]).

Not only the generation of architectural models from requirement specifications has been addressed. It is possible to find approaches which focus on the generation of other kind of outputs. A recurrent kind of output are software prototypes, either executable business logic [79], or the user interface [65]. Kamalrudin and Grundy present an approach to generate user interface prototypes from EUC [65]. Not only the authors demonstrate the viability of producing prototypes from requirements specifications, but also resort to a specification in EUC (e.g. Figure 2.4) to do such. These kind of approaches have well defined objectives, as is the case of verifying the final software, in early stages of development (either by testing features, or interacting with mock user interfaces). The approaches provide useful mechanisms to support verification and mitigate errors in the final solution. However, generated outputs tend to be “throwaway prototypes”, meaning that they do not contribute to the implementation process. The same is true for the used interfaces, since those kind of prototypes do not consider the business layer.

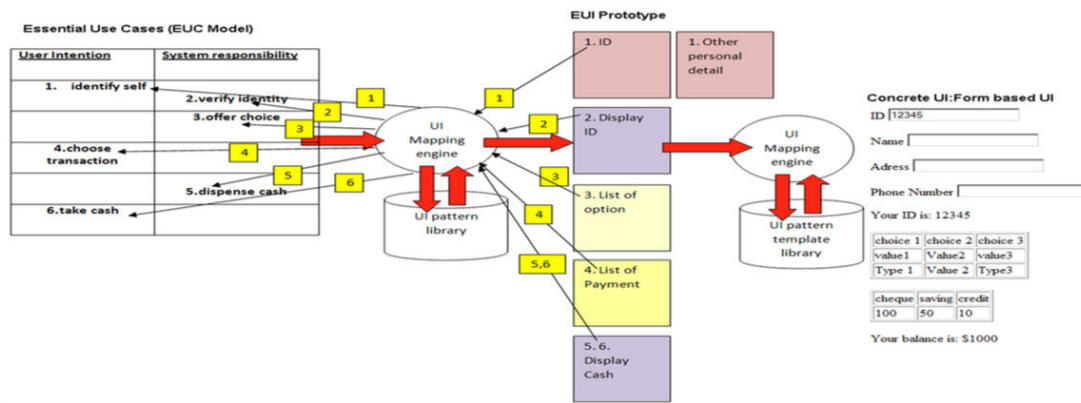


Figure 2.4: Example of EUC to Essential User Interface prototyping (adapted from [65]).

The addressed works provide valuable hints regarding the rationale that requirements are able to convey. They describe processes which successfully support the generation of software architectures directly from requirement specifications. Hence, requirements provide enough information to derive hints regarding the architectures they represent. Furthermore, not only it is possible to generate architectures, but also other kind of useful outputs as user interfaces. Overall, the presented approaches have a well defined objective, of producing a certain kind of output from the requirements. Hence, not a big emphasis is put in their analysis and verification. Furthermore, these works do not focus the integration in the MDA. An alternative approach to automate and improve the analysis of the use cases is to formalize the data they contain in order to apply a verification process, prior to generating these outputs. This way, not only it is possible to produce software outputs, but also to perform a preliminary validation and verification process.

2.2.7 Discussion

Currently existing approaches to support the formalization of requirements vary from textual analysis, to requirements specification in computable formats (e.g. architectural models). The formats in which these requirements are specified vary also, from simple textual descriptions, to more structured formats as use cases.

The decision on the format to adopt in order to specify the requirements must take in account three major factors, depending on the desired objectives. First, there is expressiveness, which is the capability of the language to represent what users intent to convey. Second, the readability of specifications must be considered. On the one hand, specifications can be written in formats, as for instance natural language, which are more flexible regarding the supported terms. These formats tend to be easier to read, but harder to operationalize. On the other hand, more strict formats as CPNs, tend to be harder to read, but easier to operationalize. Finally, the difficulty to automate specifications' processing must be considered.

Considering the three aforementioned aspects, resorting to use cases in a tabular format, as the specification structure in order to fulfill the presented objectives is proposed. As for the specifications itself, the adoption an intermediary format (similar to ACE) to describe the specifications is proposed. The intermediary format presents a compromise between readability, expressiveness and ease of analysis. ACE is not tailored to support use cases, but can be adjusted to do such, since it support the specification of simple statements.

The viability of producing relevant outputs from requirement specifications was addressed by several authors. Several approaches which support the generation of useful models and prototypes can be found. However, the approaches focus directly in the generation of outputs, and do not concern their previous representation in computable formats neither the generation of architectural models. Representing the requirements in such format prior to generating software outputs provides several advantages, such as analysis and validation of the correctness of requirements (therefore, avoiding the propagation of errors to the final solution).

2.3 Representation of Requirements on Knowledge Bases

The usage of knowledge bases to represent requirements information has been proved to be a viable approach. Having a language with support for automation, can provide the required support to automatically generate such knowledge base.

Regarding the proposed approach, three major requirements define the approach to adopt. First, it should allow to formally describe the knowledge in a structured way. Second, it should provide powerful means to query and analyze the knowledge. Finally, the knowledge base must be described in an accepted standard. Such is essential in order to extend and integrate the representation in other works. This section presents approaches which can be adopted in order to fulfill these objectives.

2.3.1 Ontologies

Ontologies are a recurrent format to formalize requirements on a knowledge base, due to their expressiveness. Ontology (a branch of philosophy), is the science of *what is*, of *the kinds* and *structures* of objects, processes, events and relations in every area of reality [104]. In information science, an ontology is a formalization of a specific domain, in terms of objects, properties and relationships. Ontologies provide support for structuring knowledge as concepts and their relationships, and specifying concrete values for those concepts (instances). In Figure 2.5 a simplified representation of an ontology is presented (as an ontology itself). In this case the OWL syntax was used. It is possible to see the concepts as *Class*, *Type* or *Individual*, which define the knowledge representation.

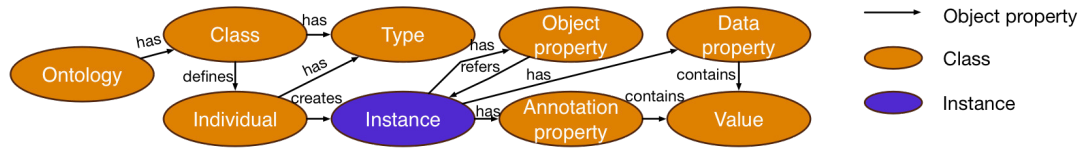


Figure 2.5: Simplified representation of an ontology.

In the context of this work, ontologies are a suitable approach to represent requirements specification. They support the specification of knowledge in a formal and structured way. Specific languages and tools are required to support them.

2.3.2 Ontology Languages

Classical ontology languages are mostly concerned with structuring and representing information (e.g. Ontolingua [47], LOOM [83], OCML [89] and FLogic [67]). Current ontology languages add, on top of classical ones, intercommunication capabilities. This kind of ontologies are becoming more used and popular. They have the same properties as the former ones, but foster interoperability via the Web. Web based ontology languages include RDF Schema, and others based on those such as SHOE [51], XOL [66] and OWL [74]. Perhaps one of the most well known, and a World Wide Web Consortium (W3C) standard, is OWL.

Due to their expressiveness, several authors address the representation of requirements in ontologies. A systematic literature review on methodologies to support the requirements engineering process with ontologies is presented by Dermeval *et al.* [32]. Several benefits on the usage of ontologies as support for requirements engineering are identified, including reducing ambiguity, inconsistency and incompleteness. This work, provides relevant information on the use of ontologies for requirements. First of all, it is a proof that ontologies are powerful means to formalize and analyze requirements. Second, OWL seems to be a good approach for requirements specifications, as it has been adopted by about half of the described approaches. These works do not address, however, the viability of using ontologies to support the software development process itself.

From the analysis of the related work, it is possible to conclude that, from the several available languages, OWL is currently accepted as the *de facto* language to describe ontologies. Hence, it seems to be an appropriate way to support information specification.

2.3.3 Knowledge Base Analysis

The usage of ontologies to represent knowledge not only supports the formalization of information but also supports its analysis. Such is also a recurring approach. Not only ontologies

have been used to represent requirements, but also to reason about them. Several approaches present mechanisms for the formalization of requirements, with emphasis (for instance) on their elicitation [105, 58, 43] and verification [44, 45].

Works performing analysis of requirements resorting to ontologies support the viability of using such kind of representation in order to formalize and analyze knowledge contained in descriptions.

2.3.4 Web Ontology Language

OWL is a declarative ontology language, based in the W3C standards XML, RDF, and RDF Schema (RDF-S), which fosters interoperability. RDF is a standard model for data interchange (over the Web). It combines the identification of data through URIs as well as the relationships between them, is what is usually referred to as a “triple” (Resource Description Framework (RDF) triples). The data results in a labeled graph for data representation. RDF-S provides the vocabulary to model information in RDF, while eXtensible Markup Language (XML) corresponds to the format in which the information is described.

OWL has three different levels of expressiveness: OWL Full, OWL Description Logics (DL), and, OWL Lite. While OWL Full supports mixing OWL statements with RDF, creating more expressive documents, at the cost of no reasoners support for such specifications, OWL DL provides maximum expressiveness, while providing inference capabilities. It enforces restrictions on the allowed statements, in order to enable reasoning capabilities. Finally, OWL Lite is a simplified version of OWL DL, with less expressiveness.

OWL supports the definition of several constructs to build ontologies¹. Some of the most relevant constructs available to specify ontologies include the following.

Class similarly to Object Oriented Languages, represents a concept in the domain (e.g. *Actor*).

The classes can have instances, or *Individuals*. Classes define the structure of the ontology.

Individual represent an element which belongs to a *Class* (e.g. *user*), an *instance*. The individuals (and their relations) are the data of the knowledge base. Individuals relate between themselves via *Object Properties*.

ObjectProperty represent the relation between individuals (e.g. *clicks*). An individual related to another individual via an object property forms an RDF triple (e.g. *user clicks link*).

DataProperty represent data in the individuals (e.g. *id* of type integer). They make it possible to add information to individuals beyond their names.

AnnotationProperty can also be associated with individuals (e.g. *transient*), denoting an annotation.

Data Type represent the types supported by OWL (e.g. integer, decimal, string, etc.).

¹Full list of elements in <http://www.w3.org/TR/owl2-manchester-syntax/>, last accessed in 2016-05-26

OWL ontologies can be written in several interchangeable languages. Listing 2.4 presents an example on an ontology, written in the Manchester OWL Syntax [56], which provides a more easy to read format than other representations. The ontology contains information about a user clicking a link. `User` is an individual of class `Actor`. `Clicks` is the object property (a *fact*) that relates it to the `Link`. Figure 2.6 presents an overview of the ontology. In this work, ontologies will be described in the Manchester OWL Syntax.

```

Ontology: <http://www.url.com>

ObjectProperty: <http://www.url.com#clicks>

Class: <http://www.url.com#Actor>

Individual: <http://www.url.com#link>

Individual: <http://www.url.com#user>
  Types:
    <http://www.url.com#Actor>
  Facts:
    <http://www.url.com#clicks> <http://www.url.com#link>

```

Listing 2.4: Example of a simple OWL ontology describing the information of a user which clicks in a link.



Figure 2.6: Representation of a simple ontology (describing the information in Listing 2.4).

Some languages have the objective of easing the specification of OWL ontologies, as is the case of Manchester Syntax [56] and Rabbit [50], or even higher level formats as the aforementioned ACE [73]. ACE further improves readability of the statements, when compared with Manchester or Rabbit syntax, by providing a simple format for interactions description. Hence, ACE can be adjusted in order to support the specification of (use case) requirements, even if it was not designed specifically to support that kind of specification. Adopting one such approach, not only supports formalizing requirements, but also reasoning about them. On the one hand, being ACE closer to natural language, makes it easier to write, and consequently formalize, the requirements. On the other hand, the language formalizes the requirements, which supports the reasoning process.

2.3.5 Discussion

Knowledge bases are a possible approach to formalize requirements information. Several approaches exist to support such specifications, but ontologies stand out for their capabilities to structure knowledge, while providing inference capabilities. Several authors successfully addressed the usage of ontologies in order to represent requirements and reason about them. The

presented works do not address, however, representation of use cases in ontologies. Nevertheless, these works, can be used as a basis for a more specific application of ontologies to represent use cases.

From the several languages available to create ontologies, OWL seems to be the most appropriate as it is widely adopted and supported by several tools. OWL has the expressiveness to define requirements ontologies as well as their instances. Adoption of the OWL DL subset supports also performing the query process on that knowledge.

2.4 Knowledge Inference Mechanism

Knowledge inference mechanisms support the analysis of the formalized information in knowledge bases, through well defined processes and languages. From several usages for the inference techniques, one is to analyze the relations between requirements. Such enables the possibility to apply pattern inference mechanisms. Implementing a pattern inference mechanism is required in order to recognize the high level requirements expressed in specifications, as part of the proposed approach. Next follows some approaches which support information and pattern inference processes.

2.4.1 Inference in OWL

One of the main reasons to adopt OWL as the language to define requirements is the available query engines. The inference mechanisms support knowledge extraction in order to analyze and interpret the data specified in the knowledge bases.

Several languages exist to query OWL knowledge bases, as is the case of *DL query* which resorts to the Manchester Syntax [57], *Terp* [102], *SWRL* [90] and *SQWRL*. However, the *de facto* language for data query is *SPARQL* [2], as proposed by the W3C. *SPARQL* is a structured query language (much like *Structured Query Language (SQL)*). *SPARQL* queries can either result in data sets or *RDF* graphs. Hence, this powerful and standard query language was selected.

```
PREFIX : <http://www.url.com#>
SELECT ?object
WHERE {
    ?s :clicks ?object
}
```

Listing 2.5: Example of an *SPARQL* query to list which objects relate via *click* object property to any subject.

Listing 2.5 shows an example of a *SPARQL* query, to identify individuals which are the target of the *clicks* object property. The prefix identifies the ontology to be queried, and the words

preceded by “?” denote variables (e.g. `?object`). The `Where` block, describes the query itself, in a triple format.

Considering the proposed objectives, exploring the SPARQL seems to be appropriate, since it is the most adopted (and supported) language.

2.4.2 Available OWL Tools

Several tools support the development and use of OWL (see W3C’s website² for a list). The Protégé IDE is perhaps the most well known [71]. There are also several implementations of reasoners supporting OWL (e.g., Chainsaw, FaCT++, JFact, HermiT, Pellet, RacerPro). Two reasoners stand apart from the others, namely HermiT (used in Protégé), and Apache Jena. Apache Jena is an open-source framework for Java which supports creating, manipulating and querying RDF (and OWL) ontologies.

Another alternative to interaction with ontologies is the *OWL API* [55]. It is another open source Java framework with the same objectives of Apache Jena. Both frameworks would suit the approach needs. The selected tool to support the proposed work is Apache Jena. First, it supports the creation of ontologies, as required by the proposed approach. Second, being an Apache framework, it will continue to have support and updates, as the Apache projects tend to be reliable.

Since is part of the objectives to have a tool which supports the inference process, Apache Jena seems to be an adequate way to integrate OWL specification and query mechanisms into Java applications.

2.4.3 Discussion

In order to represent requirements information, knowledge bases are a recurrent approach. It is possible to find several works addressing such representations resorting to ontologies. Ontologies are expressive enough to handle such specifications, while providing query mechanisms.

Regarding the proposed approach, these works provide valuable insights on how to perform the formalization of the information, and which kind of verification techniques it is possible to apply. They further provide hints on which specific technologies are most common, as is the case of OWL and SPARQL. Despite none of the works addressing specifically the representation of use cases, they provide valuable hints on how to achieve such, and regarding the viability of such approach. Both exploring the capability of OWL to represent requirements’ information, and SPARQL’s capability to query them is proposed.

²W3C list of relevant OWL tools, <http://www.w3.org/2001/sw/wiki/OWL>, last visited February 2016

2.5 Software Patterns

This section presents an overview on software patterns' related works. Approaches concerning their definition, representation, instantiation, usage and inference are presented, in order to discuss the viability of formalize and operationalize them. Not an extensive amount of works can be traced in all areas, but some insights can be obtained from the existing ones.

2.5.1 Patterns

The term *Pattern* can be traced back to Christopher Alexander's work [1]. A pattern is defined as a well known solution, for a given kind of problem, that arises in a specific context. Given a problem, patterns enable solutions to be reused without the need to recreate them all over again. Patterns exist within a certain context. Hence, given a problem/context pair, patterns provide the corresponding solution (c.f. Figure 2.7). Such an idea of a pattern was later adopted to software engineering by Gamma *et al.* [41]. Since then, there have been a large amount of works related with patterns [96].



Figure 2.7: Patterns as bridges between problems/contexts and solutions.

In software engineering patterns appear in several flavors, depending on their objectives. Two kind of patterns are specially relevant in the context of this work. First, are the requirements patterns. Specifically, for a given set of stakeholders needs, the matching requirement patterns present a good requirements specification. The second relevant kind consists of design and architectural patterns. Design patterns provide solutions for common issues which occur at the design level (i.e. organizations of classes). Architectural patterns produce solutions for issues which occur at a higher architectural level (i.e. organizations of classes groups).

2.5.2 Requirement Patterns

Requirement patterns are both applicable on software engineering and requirements engineering, with works addressing their specification, management and application. While not as popular as other kinds of patterns (e.g. design and architectural patterns), several works regarding requirement patterns can be found. Requirement patterns are referred to in literature also as software requirement patterns. In this work, in order to differentiate them from patterns which describe solutions (e.g. design and architectural patterns), they are simply referred to as requirement patterns.

A systematic literature review on requirement patterns was performed by Silva and Benitti [29]. Three interesting outputs resulted from the literature review.

- As expected, the usage of templates in order to specify requirement patterns is common practice;
- While some authors focus on the specification of non functional requirements with requirement patterns, there is evidence that they can also be used to specify functional requirements;
- Requirement patterns should support composition processes.

Regarding the proposed objective, it is relevant the fact that there is evidence of their usage, specifically to specify functional requirements.

Several authors address the definition and specification of requirement patterns [119, 40]. A relevant contribution is provided by Withal [120], which presents a catalog of requirement patterns, addressing aspects such as their usage, classification and identification. Another relevant work is presented by Yan, which catalogs and classifies 72 eCommerce patterns [123].

The organization of requirement patterns in catalogs has been addressed by several works in order to support different scenarios. Apart from the aforementioned work by Withal [120] (one of the most complete, with 41 patterns, distributed by 8 categories as depicted in Figure 2.8), it is possible to find catalogs which document domain specific patterns, as Content Management Systems (CMS) [93], or address specific aspects of the systems, as usability [16], trust [54], sustainability [97] or legal aspects [53].

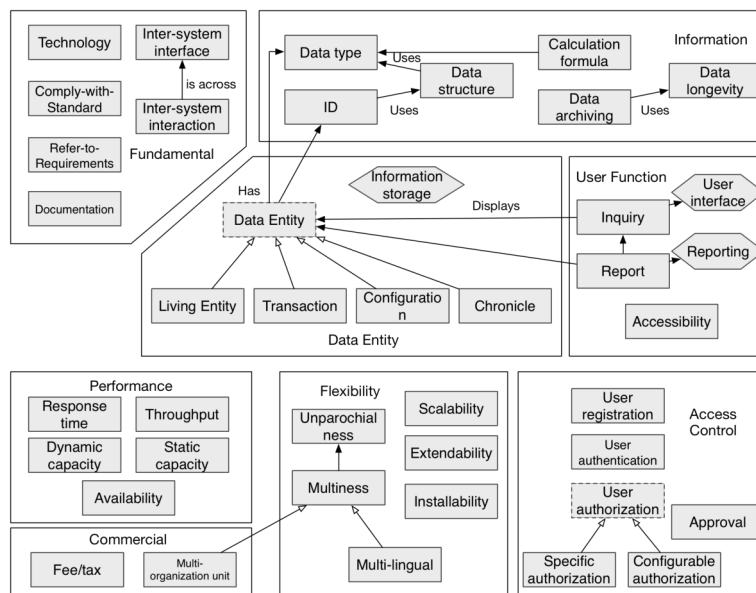


Figure 2.8: Overview of Withal's pattern catalog (from [120]).

Analysis on the adoption of requirement patterns shows that, despite the difficulties, they are being adopted as recurrent practice, for instance in the industry [39, 30]. Works which address the usage of requirement pattern in systematic ways contribute to their adoption. Authors present approaches to formalize requirements, and to reason about them [39, 30]. Existing approaches tend, however, to focus in the requirements engineering aspects. Tools and approaches focus in handling, managing and analyzing requirements and their relations (e.g. support for reuse), as means to support the software development process. In the context of this work, the inference of requirement patterns from requirement specifications is proposed, in order to automate the analysis of requirements.

Analysis of existing works on requirement patterns shows that authors are concerned mostly with the analysis and verification of specifications, in order to improve the specifications themselves. It was not possible to find, however, works providing a systematic and automated oriented transition process from requirement patterns to software specifications. Since requirement patterns represent aspects of the final solution, implementing such a transition provides support to generate solutions closer to the intended specifications.

2.5.3 Patterns Categorization

The pattern instantiation process consists in producing specific outputs (e.g. models, or code), from a pattern specification. In the context of this work, the instantiation of software patterns, in order to produce architectural models, is proposed.

Opposing to requirement patterns, the quantity of works concerning software patterns is extensive, therefore, it is not trivial to find a clear categorization of existing works. A literature review was performed in the context of this work. From this review two relevant outputs were produced. First, a template was specified with the intent of supporting all kind of patterns. Second, two levels of patterns were defined, namely *Patterns* and *Patterns Sets*, in which existing approaches can be categorized. Figure 2.9 presents the result of the categorization process.

The *Pattern sets* are subdivided in three categories. *Pattern Catalogs* consist in a set of software patterns described and organized according to their nature, in order to support their usage. *Pattern Languages* consist in subset of patterns and their relations. *Pattern compounds* are small sets of patterns which are usually found together, or are known for producing good solutions when combined.

The patterns themselves can represent information at different levels of abstraction. *Requirement patterns*, as aforementioned, represent common requirements found in different specifications. *Analysis patterns* represent knowledge at analysis level, without concerning source code or implementation. *Architectural Patterns*, as presented for instance by Buschmann *et al.* [15], present architectural solutions for specific situations, with corresponding forces and drawbacks. *Design Patterns*, as proposed by Gamma *et al.*, define design-level solutions for well known problems for Object Oriented projects. Finally, *Idioms* define platform specific solutions.

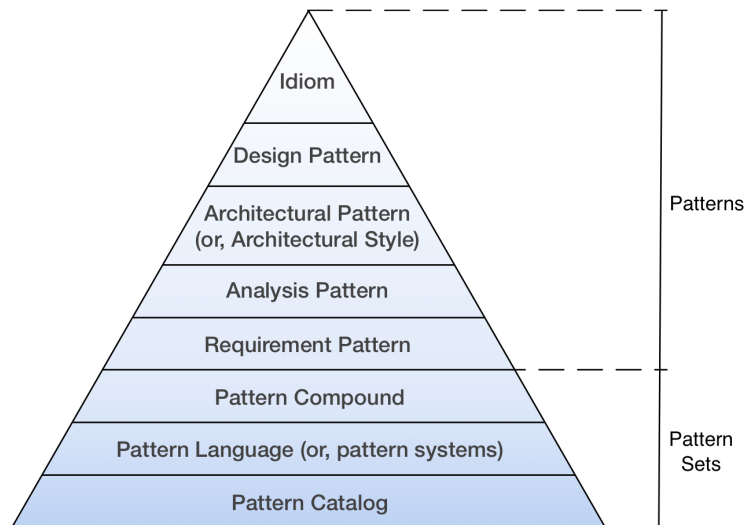


Figure 2.9: Categorization of software patterns, according to their abstraction level (higher on the bottom, lower on the top) [27].

Regarding the production of models as input for the MDA process, they are achieved by composing software patterns. Thus, apart from the requirement patterns, two other patterns are of special interest. On the one hand, architectural patterns which specify how to organize groups of classes, at a high level of abstraction. On the other hand, design patterns specify how the individual classes should be organized. More information can be found in the technical report resulting from the literature review [27].

2.5.4 Pattern Inference

The inference of patterns from knowledge bases has already been the subject of study. It is possible to find works from the late 90's regarding pattern inference on several formats [113]. For the proposed approach, the objective consists in the inference of requirement patterns, which in their genesis are similar to software patterns. As result, presented work focuses in software patterns.

One of the most influential works regarding software patterns, specifically design patterns, was presented by Gamma *et al.* [41]. Such work, in a form of a pattern catalog, defined the basis for many following works. There are other influential works describing software patterns at other levels of abstraction, as is the case of architectural patterns [37, 15] and requirement patterns [119]. The definition of patterns is essential in order to perform their inference.

When patterns started to be defined, several works emerged regarding their inference (see, for instance, [96] for a survey). Software pattern inference was addressed in several aspects. Some

authors propose their inference by analyzing source code in which they are contained [113, 100]. These approaches are mostly useful for documentation and reengineering tasks, since they provide a better understanding of the source code organization. Directly analyzing the source code avoids the need to pre-process the code and apply transformation techniques. The analysis is made directly in the artifacts themselves.

An alternative to analyzing only the source code, is to generate an intermediary representation. An example of such is to generate a UML model from the respective source code, and map it into a ontology, written for instance in Prolog [68]. Creating an intermediary representation, as a knowledge base, is more expressive, but there is the need to apply an analysis process to the source code, prior to generating the knowledge base. This translation process represents an additional cost, and might lead to loss of information. In the case of analyzing only source code, there are also tools supporting software pattern inference. An example is the Ptidej tool suite, with the objective of evaluating and enhancing software quality through patterns [48].

Previous works by the authors provided relevant background to support the objectives proposed in this thesis [22, 23]. The authors developed an approach to analyze Java source code, generate high level models (class diagrams), and in those models infer the existence of design patterns, resorting to a knowledge base and an inference engine, as depicted in Figure 2.10. These works provided knowledge in order to extend such approach, and support the proposed objectives.

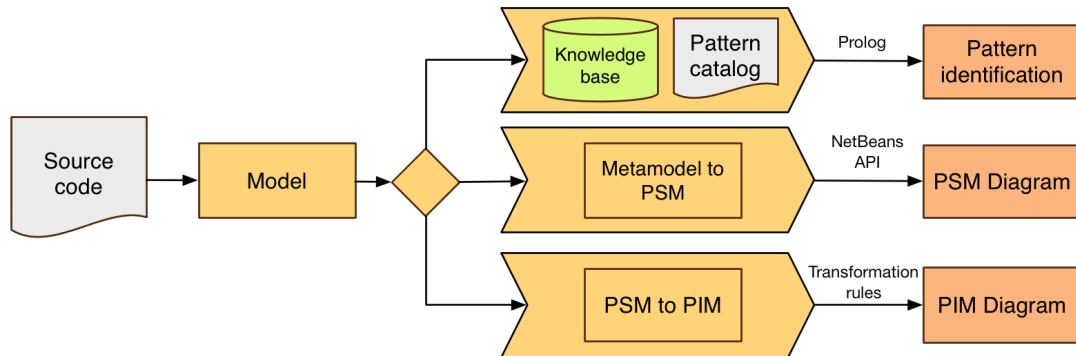


Figure 2.10: Design pattern inference approach, adapted from [22].

The specific inference of software patterns from OWL knowledge bases has also been addressed [69, 59]. The approaches analyze source code in order to generate intermediary representations, which are next formalized in OWL. Resorting to OWL's inference engine, the authors are able to perform software pattern inference. These works are of special interest, since the proposed approach requires support for inferring requirement patterns from a knowledge base.

As part of the objectives proposed in this work, several approaches were presented which can be adjusted in order to support the formalization of requirements in OWL. The aforementioned approaches all deal with software patterns. For requirement patterns, specially considering their

inference, there are not a large amount of works available. However, since requirement patterns are similar to software patterns, it is possible to develop a new inference approach based on the existing ones.

2.5.5 Software Pattern Instantiation

Software patterns were not designed with the objective of being automatically instantiated. Instead, it is the responsibility of developers to specify concrete instances of the patterns. As part of the objectives for the proposed approach, this instantiation process is required to be semi-automated. By supporting automatic instantiation, patterns can be used in a composition process to achieve a unified solution.

One of the fields specified in software patterns catalogs (e.g. [41, 15]) is the sample implementation (e.g. *Structure* and *Implementation*), which guides the application of that patterns. In order to support a semi-automatic process, it is possible to resort to the provided sample implementation to document patterns in an appropriate format. Being the sample implementation defined as a structure, similar to a template, creating the concrete instances consists in concretizing such templates.

The instantiation process required for the proposed approach concerns the *design* and *architectural patterns* level. Instances of both kind of models can be materialized as architectural models, for instance as UML class diagrams, in order to represent the structural aspects.

By analyzing existing pattern catalogs and the pattern structures, it is proposed to define a template format with support for instantiation. This template format generates then instances of the proposed implementation, with interaction from the users.

2.5.6 Discussion

The amount and scope of works regarding patterns has established a relatively large body of knowledge in that area. Nevertheless, regarding the automation of the instantiation process, there are currently no suitable approaches which satisfy the proposed objective. A process is required to create concrete software patterns, in such a way that, while being semi-automatically performed, the process generates outputs according to the correspondent pattern.

In order to implement an approach to systematically support the instantiation process (with automation in mind), currently documented patterns were analyzed. Existing patterns tend to be specified alongside with their structure, or sample implementation. Abstracting such definition provides support to reuse them, by supporting the definition of instances.

2.6 Software Pattern Composition Process

Software patterns can be composed resorting to different operators, in order to be incorporated as part of the same solution. The process' automation is required in order to support the automatic generation of architectural models. This section describes the analysis of existing approaches concerning the composition, model generation and code production processes.

2.6.1 Composition Techniques

Three major categories of pattern composition techniques can be found in the literature, namely *Connection*, *Combination* and *Inclusion* (c.f. Figure 2.11) [114, 112]. Connection of patterns consists in connecting elements from one pattern to another pattern. Such results in the two patterns existing separately, but still associated. Combination of patterns is based on unifying compatible classes existing in different patterns, in a single class. Finally Inclusion of patterns, consists in a pattern being included as part of another one.

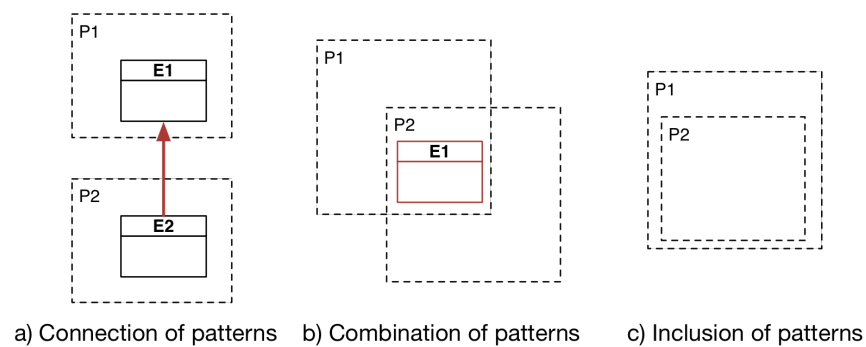


Figure 2.11: Pattern composition techniques.

Two major merging operators support the three pattern composition techniques, specifically *Stringing* and *Overlapping* (c.f. Figure 2.12) [121, 114]. Stringing consists in connecting elements of the two patterns (e.g. through an association), in order to make a connection between those two elements. Overlapping is a merging operator which combines the information of two entities, existing in different patterns, merging then the patterns.

2.6.2 Discussion

Applying the pattern composition techniques is straightforward to perform. The process consists in selecting the appropriate components, and applying the aforementioned operators. Existing works successfully describe how the process should be performed. There are however issues which result from the composition process. Two are worth mentioning, namely the traceability of the

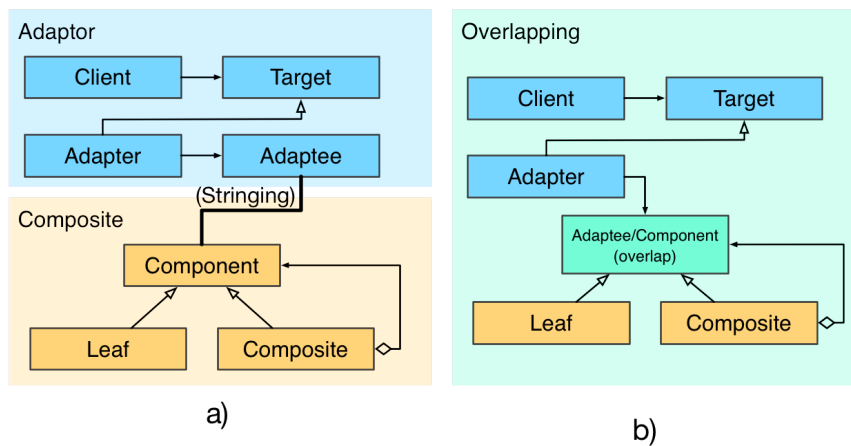


Figure 2.12: Adaptor and Composite patterns composition by a) Stringing and b) Overlapping.

original patterns, and the selection of the appropriate components to compose. Approaches that address the traceability aspects, propose improvements on the merging operators and processes, in order to keep some information which supports traceability [112]. Remains also to implement an automated process, according to the selected pattern representation format.

2.7 From Requirement Patterns to Software Patterns

The association of requirement patterns to software patterns enables the possibility to make the transition from the requirements level, to the architectural one. This mapping process consists in selecting the most appropriate software pattern, to support a certain requirement pattern. This selection process is performed by analyzing how the software patterns support the requirement patterns.

2.7.1 Selecting Patterns for Requirements

The selection of the appropriate software patterns to support an architecture can be performed by analyzing the requirements information. This is only natural, since requirements describe the desired features in the final solutions, and patterns contain a set of contributions to the solutions in which they are integrated. Usually, the selection of software patterns, based in requirements specification, is performed through manual processes [46, 18].

Of special interest is the systematic approach presented by Bass *et al.* to select patterns, in order to support usability [4]. High level requirement specifications are decomposed into smaller components (scenarios), which can be analyzed at a lower level. Patterns are characterized by

their benefits to usability, and the composition process consists in selecting the most appropriate benefits to support a certain scenario. Bass *et al.* approach, despite being designed to support usability (patterns), can be adapted to support software and requirement patterns. Decomposing the patterns to lower level components, enables the possibility to compare their concerns and contributions, thus, supporting a matching process.

In order to have an automated process, it remains to provide a systematic approach to analyze the relations between requirement patterns concerns and software patterns contributions.

2.7.2 Relating Requirements

In the Non-Functional Requirements (NFR) area it is possible to find several works addressing the interactions among several requirements (see [86] for a survey). It is known that requirements interact between them, not only affecting the final solution, but also other existing requirements [17, 118]. A special kind of interaction is the conflict. Conflicts denote contradictory concerns among different requirements, meaning that when a requirement is set, it conflicts with the objective of other requirements defined in beforehand [33, 9].

The identification of conflicts between requirements is based in the objectives of each requirement, and how each contributes to the solution. Mairiza *et al.* present a systematic approach to analyze conflicts between requirements [85]. In that approach, requirements are described by their types, and their interactions are analyzed. Such results in a table of conflicts, as presented in Table 2.2. In the table, the symbol X means an absolute conflict, * a relative conflict and O no conflict. The conflicts for the remaining relationships (empty cells) are unknown.

Table 2.2: Example of conflicts between requirements (adapted from [85]).

NFRs	Functionality	Interoperability	Maintainability	Performance	Portability
Functionality	O		*	*	
Interoperability				X	
Maintainability	*		O	X	
Performance	*	X	X	*	X
Portability				X	

The methodologies to analyze conflicts on non-functional requirements can be adapted to support the analysis of conflicts in patterns. If patterns are described at the same level of abstraction (corresponding to the NFRs types), it is then possible to generate a similar approach to clearly define the interactions between patterns, achieving then a systematic approach. It remains then to formally specify, at the same level, characteristics of both kinds of patterns (i.e. requirement and software), in order to support one such approach.

2.7.3 Forces

Software patterns can be described by their low level impact in the solutions. In the patterns community, there is a term to describe those low level characteristics, the *forces*. The term force is used to denote (c.f. Buschmann [15]): “*any aspect of the problem that should be considered when solving it, such as Requirements (what the solution must fulfill), Constraints (things to consider), and Desirable properties (that the solution should have)*”.

In general, forces represent the pros and cons of adopting a specific pattern. They may complement or contradict each other. For instance the extensibility of a system might contradict minimization of its source code, since a generic solution tends to be more complex in terms of code. In Buschmann’s words [15], “*forces are the key to solving the problem. The better they are balanced, the better the solution to the problem*”. Software pattern catalogs tend to describe the forces (e.g. intent) for each cataloged pattern [15, 41]. The application of different patterns, and the effects of the forces on the resulting application, are usually presented in a tabular format, where for each force is shown the impact on other forces. This format is usually called the forces matrix.

Requirements patterns, can also be described by the forces that compose them. Specifically, it is possible to analyze their goals, which can be characterized by their forces, similarly to non functional requirements [46].

2.7.4 Discussion

It is possible to find several approaches to map requirement patterns to software patterns. Existing works focus in the analysis of low level aspects of both requirements and patterns, in order to perform their matching. However, approaches tend to not formalize the information, therefore presenting hard to automate approaches. In the non-functional requirements area, approaches have been proposed that relate requirements by their contribution (either positive or negative to other requirements) in order to analyze their conflicts, and produce a conflicts table. Describing requirements at the same abstraction level is also proposed to enable their comparison.

Based in the presented approaches, it is proposed to analyze the viability of specifying requirement and software at the same level of abstraction. Of special interest are the forces, which can be used in order to analyze conflicts, and support an automated process. The objective to have a mapping process from requirements to software patterns consists then in selecting the most appropriated software pattern, based in its contribution for the requirement pattern.

2.8 Similar Approaches and Supporting Tools

It is possible to find approaches which have objectives close to the ones proposed in this work. These approaches were analyzed in order to understand their contributions, and later be used to implement the proposed approach. Furthermore, it is relevant to understand to which extent existing works cover the proposed objectives.

2.8.1 Requirements Based Approaches

Similar approaches can, overall, be classified in two major groups. First, there are works which resort to the analysis of textual specifications, and identify relevant terms which are directly translated into entities. Relations between those terms are then translated into relations between the entities [3, 92]. One approach worth mentioning is presented by Liu, where use cases are formalized and described according to a vocabulary, with predefined meaning [81]. In the second category of works, are the approaches that take the specifications on a predefined format, which can afterwards be converted into source code [103]. For instance, Bulajic *et al.* present an approach which generates source code, but focus in the validation of business requirements, during the requirement negotiation process [14].

Presented approaches successfully address the generation of software outputs (e.g. models and source code), based on requirement specifications. The processes are not, however, focused in contributing to the MDA, by extending the process. Instead, they implement a different vision, where models are not required to support the different steps of the processes.

2.8.2 Supporting Tools

The tools which can help to achieve the proposed objectives include the Apache Jena framework to support the representation and query of OWL ontologies, in Java [62]. The Protégé tool provides several facilities in order to develop and test the required ontologies [71].

As a way to support the storage, sharing and spreading of the inputs and outputs of the proposed approach, it was developed the Modelery [21, 25]. Modelery consists in an online models repository, which provides an API, in order to enable the integration of the models' upload and download features in modeling tools.

2.8.3 Discussion

The existence of approaches with similar objectives to the proposed ones is acknowledged. While these approaches support the generation of different kinds of artifacts, they differ from the proposed in this work. Indeed, none of the approaches concerns the extension of the MDA process, by formalizing requirements and automating the architectural model generation process.

Instead, the approaches focus in the generation of specific outputs instead of models. Thus, to the best of the author's knowledge, no work capable of covering the proposed objectives exists.

Supporting tools provide a valuable help in order to achieve the proposed objectives. They contribute for several steps of the proposed process, namely in the representation and analysis of requirements in an intermediary format. Both the analysis and interchange of ontologies is also supported by existing tools.

2.9 Summary

This chapter provides an overview of the works related with the objectives for the proposed approach. Several areas were addressed, as well as their current contributions and issues to tackle, in order to accomplish the proposed objectives.

There are several existing approaches in order to specify requirements in a simple format. From natural languages, to more formal formats, different authors present different visions. A possible approach is to find a middle term, with a controlled language, which can easily be formalized.

Structuring requirements information can be achieved in several ways. From the analyzed ones, OWL seems to be a suitable approach for its capabilities to specify and organize knowledge. Indeed, if a proper specification language is selected, the requirements formalization into OWL can be achieved without comprehensive textual analysis techniques.

Some existing knowledge bases representations support the information extraction process, by providing query languages. Hence, formalizing the requirements in an appropriate knowledge base, provides support for such query languages, and consequently for information inference. This information inference mechanism tackles the objective of performing pattern inference in the knowledge base.

Patterns in software engineering are well established nowadays, from their specification and composition to their inference. As expected, the presented approaches are generic enough in order to be adjusted to the individual needs. Hence, based on existing works, it is possible to develop a specific approach. Since patterns are well documented and several authors have analyzed them at several levels of abstraction, their mapping into other kinds of patterns is possible by analyzing their lower level characteristics.

Patterns are also currently well documented, with characteristics such as sample implementation and examples of usage. Hence, creating concrete instances is possible by defining generic formats for their specification. Their composition has also been object of study, with two merging operators (stringing and overlapping), supporting the composition of architectures from a set of pattern instances. The MDA provides the framework to take these resulting architectures, and produce computable outputs as result.

Table 2.3: Overview of the related work.

Process	Implemented
Requirements operationalization	✓ e.g. [92, 65]
Requirements to intermediary representation	✓ e.g. [63, 73]
Use case representation	✓ e.g. [61, 38, 20]
Use case formalization	×
Use cases to intermediary representation	×
Use cases to ontology	×
Architectural to Architectural Models	✓ e.g. [70]
Architectural to Software Models	✓ e.g. [70]
Software pattern inference	✓ e.g. [22, 23, 96]
Software pattern composition	✓ e.g. [114, 112]
Requirement pattern inference	×
Requirement pattern to software pattern	×
Requirement Model to Software model	×

Finally, approaches with similar objectives to the presented ones do not tackle the same objective of producing architectural models in order to take advantage from the MDA. Existing tools and libraries can, however, support the implementation of a new tool supporting the proposed approach.

In conclusion, it is possible to find a large amount of works in the literature regarding the several aspects of the proposed approach. Indeed, these works provide valuable contributions to specify a new approach. In order to take advantage of existing works, there are adjustments which need to be made, since they were designed with different objectives. Table 2.3 summarizes the current state of the several addressed areas. It is possible to see that use cases formalization, transformation and requirement pattern inference are the areas with less focus.

Although many areas are covered by existing approaches, they are not combined into a single solution. As an example, use case specification languages with support to automatic transformation have not been integrated with requirement patterns. An overview of approaches with similar objectives to the proposed ones is presented in Table 2.4. In the table is shown that, from existing approaches, none is able to fully automate the requirement models to software models process, through a systematic pattern process. While the objectives of SCARP go towards a semi-automatic process, it is not proposed to produce software solutions as complete as other approaches. For SCARP is, however, proposed to deal with patterns in order to improve the transformation process.

Table 2.4: Comparison of existing approaches.

	Jacobson [61]	Overmyer [92] <i>et al.</i>	Kamalrudin and Grundy [65]	SCARP goals
Available	✓	×	✓	–
Automatic	×	✓	✓	–
Model Based	✓	–	✓	✓
Requirements Models	×	×	×	✓
Intermediary representation	×	×	×	✓
Self contained Steps	✓	×	✓	✓
Use cases	✓	×	✓	✓
Evolutionary prototypes	✓	✓	×	✓
Executable outputs	✓	✓	✓	×

Chapter 3

The SCARP Approach

This chapter presents the SCenario bAsed Rapid software Prototyping (SCARP) approach. SCARP has the objective of providing a systematic process to produce software artifacts (e.g. architectural information and prototypes), based on the textual specification of requirements.

The approach is a comprehensive process, composed of several steps addressing the different areas involved in going from requirements specification to architectural artifacts production. Dealing with different areas results in a multi step process. SCARP was designed to support the automation of some of these steps, in order to provide a systematic process to produce software artifacts from requirements specifications. Requirements elicitation and refinement is not addressed in SCARP. Instead, it is considered to be a previous step, part of Requirements Engineering processes.

The diversity of steps results from the need to address the formal representation of requirements, extraction of information, and consequent transformations until source code. Thus, the process starts with requirements formalization resorting to a CNL. The formalization is then transformed into an intermediary format with support for querying. Such enables the systematic and automated analysis of the requirements information. The automated analysis process supports, for instance, the extraction of requirement patterns, which are mapped into software patterns. Performing composition of those patterns results in architectural outputs, supporting the specified requirements. SCARP has an iterative nature, supported by systematic and automated processes. Hence, if the resulting outputs are not as expected, it is possible to repeat the process, adjusting the inputs.

Not the same emphasis was equally given to all phases, since for some of them good solutions already exist (c.f. Chapter 2). Some of the phases are described in more detail, as for instance the use case formalization processes, while to other phases, due to having known solutions to support them (as the code generation from XML Metadata Interchange (XMI)), less emphasis is given. This chapter describes, for each step of SCARP, which are the tasks and issues to tackle.

3.1 Automation of a Model based Process

Automation in model based software development methodologies supports, among others, its application in a systematic and consistent manner, ensuring repeatability. Systematization provides also the viability to develop a tool supporting the process.

Regarding the previously defined objectives (see Section 1.3.3), works addressed so far tend to support both systematization and automation of specific aspects of the problem. Hence, part of the systematization consists in taking advantage from these aforementioned works.

3.1.1 MDA and the Proposed Approach

The proposed approach relies in models in order to support several steps. Furthermore, models at two levels are required: those in which patterns are applied, and, those containing the information required to produce software prototypes. Hence, the MDA suits the proposed approach. It is possible to map the outputs from the approach to corresponding MDA artifacts (c.f. Figure 2.1), as follows.

High level models (PIM) are the models generated from the combination of architectural patterns.

Standard Languages can be industry standards (as proposed by Mellor and Balcer, presented next [88]), therefore UML.

Transformation definitions need some information about the context (see Entities and Workflow Framework (EWF) in the next section).

Languages to write the transformation are part of the informations related with the context.

Tools are artifacts needed in order to automate the transformation process (and part of the proposed objectives).

By using the MDA it is possible to achieve automatic generation of code, by providing the required artifacts. Hence, there is the need to implement them, in order to complete the process.

3.1.2 The Entities and Workflows Framework

The MDA transformation processes (both from PIM to PSM, and from PSM to code), require inputs related with the specific platform in which these models are being deployed. This information supports the models refinement process, and both automated or manual transformation processes. If provided in an appropriate format, this information can support the automated transformation process.

In order to provide support for that information, the EWF is proposed. The purpose of such framework is to support the automated transformation process, by providing context to the entities in the models. Such allows not only to add meaning and structure to the entities, but also behavior. Hence, the EWF supports the transformation of the models, by specializing the entities in the models, leading then to source code meaningful for the EWF context.

In practice, the EWF specifies the application domain, while providing some more information. On the one hand, it provides the domain information relevant for the application being developed. Such includes the existing entities, and how they relate. On the other hand, the EWF provides inputs specific for the domain. Such includes query and mapping information. Briefly, the EWF will provide the transformation definition required by the MDA in order to perform the models' transformation.

3.2 SCARP Overview

The SCARP process is composed of two groups of tasks. First, there is the parametrization tasks, which include the specification of the domain model and transformation information, as well as the specification of the pattern catalogs (c.f. Figure 3.1 **Application domain** and **Process Inputs**). The second group of SCARP tasks starts with the formalization of use cases scenarios by the user (Figure 3.1 **I**), resorting to a CNL to describe the scenarios. The use cases are then converted into an intermediary knowledge base, specifically in OWL (Figure 3.1 **II**). OWL enables the possibility to query the formalized use cases information, in order to extract requirement patterns (Figure 3.1 **III**). The requirement patterns can be associated with software patterns (Figure 3.1 **IV**). Composing software patterns through appropriate techniques leads to architectural solutions (Figure 3.1 **V**). The architectural solutions correspond to UML models, that can be used to produce, for instance, source code (Figure 3.1 **VI**).

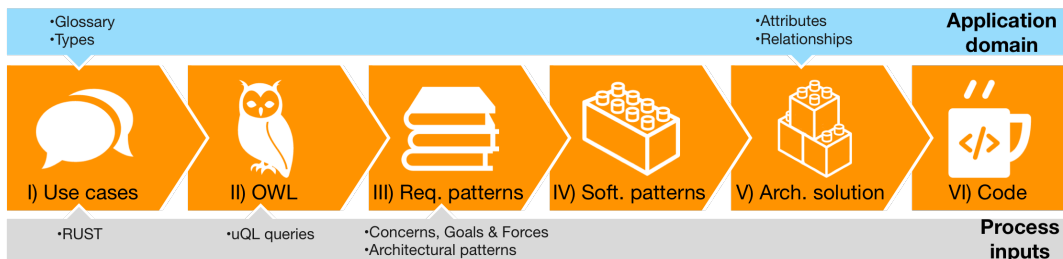


Figure 3.1: The SCARP process.

In order to support SCARP three types of reusable inputs are required. First, the domain model is required. In a domain model the existing entities and corresponding relationships (for a specific domain) are described. Second, the process inputs are required. They parametrize how the process is performed. Queries and transformation rules are examples of process inputs.

When changing the application domain, the domain inputs can be reused, however the domain model should be adjusted accordingly. These two inputs correspond to the EWF. Finally, the use cases must be provided. They correspond to the starting point of the SCARP process.

Two actors are expected to interact with SCARP. On the one hand, there are the actors which perform the approach setup. Such includes the specification of reusable inputs, as the domain model and queries which is the specification of the EWF. On the other hand are the actors which use the approach to generate the architectural outputs. The latter ones provide, for instance, the use case specifications and types information.

The next section details each step of SCARP. For each step the required processes are presented, as well as the inputs supporting the processes.

3.3 SCARP Parametrization

The existence of previous knowledge is assumed in order to support the extraction of domain relevant information in SCARP. This step of parametrizing the application context is required, prior to the execution of the process itself, and corresponds to the first group of tasks to apply SCARP. Such is performed by providing the **Application Domain**, as well as the patterns and matching information process inputs (c.f. Figure 3.2), corresponding to the EWF.

3.3.1 Domain Model Specification

The **Application Domain** is a reusable artifact used in SCARP, and represented by a *Domain Model*, which defines the kind of application that the process will generate. The *Domain Model* supports also the extraction of a *Glossary* of terms existing in the domain, the *Relations* between the entities, as well as the *Types* and *Attributes* of the entities.

The level of detail of the domain model will affect the process itself, since the process relies in this model.

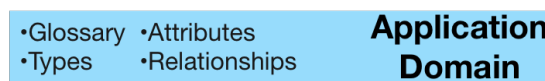


Figure 3.2: Artifacts in the *Application Domain*.

Figure 3.3 details (resorting to a UML activity diagram) the information flow for the *Application Domain* specification process. The user has the responsibility to specify such *Application Domain*. The user starts by **Creating the Domain Model**, which results in a specific **Domain Model instance** (for the application context). In SCARP the domain model instance is

used to **Extract information** related with that domain. Such process consists in producing a knowledge base containing the **Glossary** of terms related with the domain (e.g. **Actor**, **User**, **Email**), the **Types** information (e.g. **Actor**, **Person**, **Field**) and **Attributes** for those terms (e.g. **Email**, **Age**, **Name**). All those outputs are useful for the following steps.

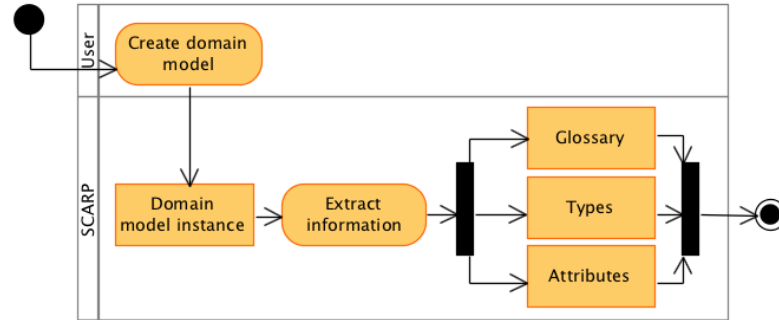


Figure 3.3: Setup of SCARP - providing the domain model.

The *Glossary* corresponds to the set of existing entities in the *Domain Model*. The entities represent all the known terms to the process in an open-world assumption, meaning that for instance, new terms can be added later. The glossary information is extracted by iterating all the domain model elements, and extracting their names and relationships.

The *Types* and *Attributes* information corresponds to a subset of the *Glossary* of terms. By analyzing how the entities in the domain model relate to each other (i.e. the names of the relationships between two entities), each entity is defined as having a type. In order to define this information, three categories of relationships were specified, namely *CompositionOf*, *PropertyOf* and *TypeOf*. The role of the categories is as follows.

CompositionOf classifies the relationships which state that an entity *contains* another entity.

PropertyOf classifies the relationships stating that an entity is a *property* of another entity.

TypeOf classifies the relationships which state that an entity *has the type* of another entity.

Figure 3.4 presents an example of a domain model, which can be provided to SCARP. Example of entities contained in the model are **User**, **Actor**, **Cart** and **Order**. Regarding their relationships, it is possible to identify **contains**, **has** and **is**. If the **contains** relationships is classified as the type *CompositionOf*, then the **User** contains **Cart**. Similarly, if **is** is classified as *TypeOf*, therefore a **User** is an **Actor**. Finally, considering that **has** belongs to *PropertyOf*, a **Cart** contains a **Order**.

The domain model specification has the objective of being lightweight, without restricting the names of entities and relationships. By categorizing the different kinds of existing relationships in the domain model, it is possible to assign meaning to the domain model elements.

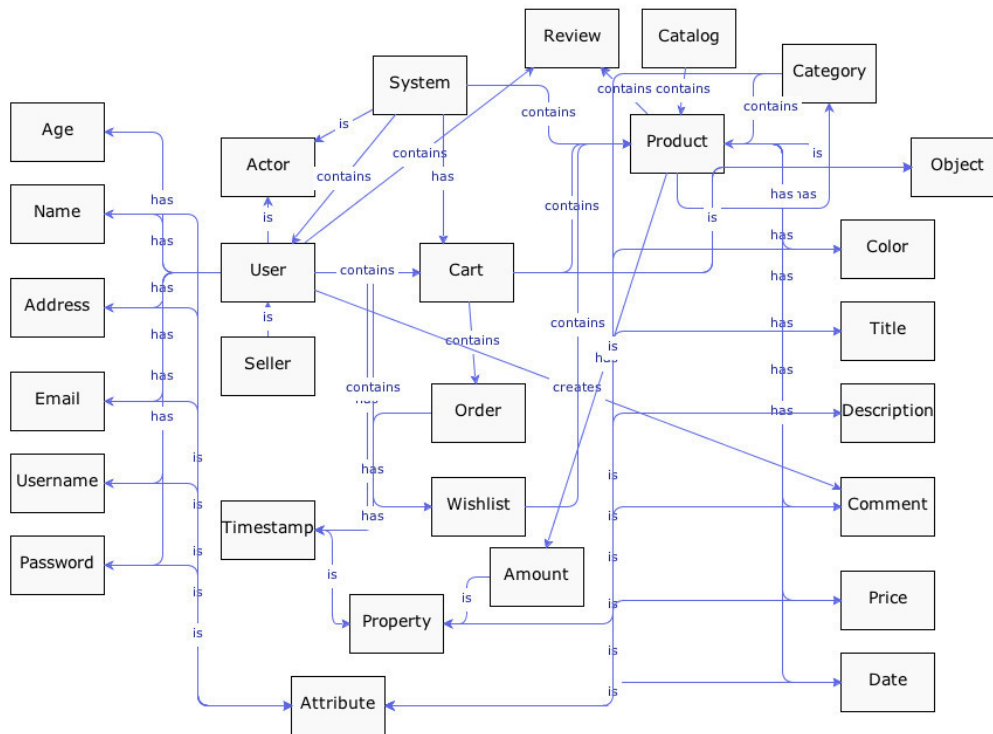


Figure 3.4: A domain model for an eCommerce context.

In order to support SCARP, the *Domain Model* is formalized into a knowledge base. Such is required in order to enable the possibility to query the information.

3.3.2 Patterns catalogs

SCARP requires also the existence of two pattern catalogs in order to perform the process. On the one hand, a requirement patterns catalog is required. This catalog must describe the requirement patterns to be inferred in the specifications. The catalog must be specified in the form of queries (see Appendix B.3 for an example of a requirement patterns catalog). On the other hand, a software pattern catalog is required. The catalog can contain, for instance, the specification of design and architectural patterns, to be used in order to achieve architectural models. These patterns should be described in a format that abstracts the user from the implementation details, and at the same time in a format which eases their operationalization (see Appendix B.4 for an example of a software patterns catalog).

The requirement to software pattern matching process is based on the analysis of the relations between these two types of patterns. Thus, the process requires the characterization of both kinds of patterns, namely regarding the concerns of requirement patterns, and goals of software patterns. This mapping information should be described in a format with support for automatic analysis, namely in an OWL ontology (see Appendix B.5 for an example of matching information). This information corresponds to the Process Inputs, depicted in Figure 3.5.

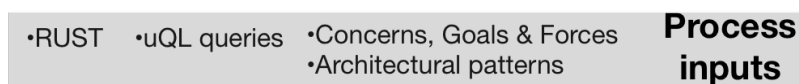


Figure 3.5: SCARP process inputs.

3.4 Use Cases Specification

The second group of tasks starts with the use case specification process. This corresponds also to the first step of SCARP itself (c.f Figure 3.6 I)). The specification process consists in the formalization of use cases. The output of this step is provided as input to the next step, in order to create a knowledge base representing the information contained in the use cases.

Hence, the first step of SCARP starts with the specification of textual requirements, as presented in Figure 3.7. Use cases represent the requirements to be analyzed in order to guide the elaboration of software artifacts. They contain the information required to extract hints about the system they describe.

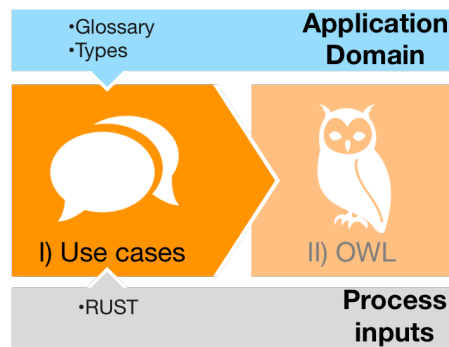


Figure 3.6: Specification of user requirements step.

Use cases, if written in natural language, are hard to analyze and operationalize. To overcome this difficulty, it was decided to use a CNL to support formalizing the specifications. Restricted Use Case Statement (RUS) is the CNL which supports the requirement specification process. The RUS language provides the core features to support the descriptions.

RUS relies on a template language, Restricted Use Case Statement Template (RUST), in order to be described. The objective of the template language is twofold. On the one hand, it supports the validation of RUS statements, regarding their syntactical validity. On the other hand, it specifies how the information in RUS should be interpreted, and mapped into a knowledge base.

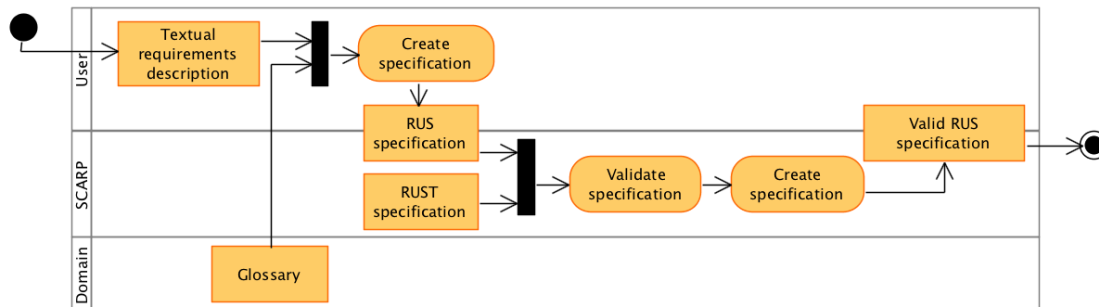


Figure 3.7: Step 1 of SCARP - creating the RUS specification.

The process of creating the specification consists in describing a use case in a tabular format, using valid RUS statements. Knowing that the language used to create the specification is RUS, the users start the process by taking previously defined **Textual requirements descriptions**, and proceeds to **Create a specification** from them. An example of textual requirement description is as follows.

There are several steps to follow in order to add a product to a shopping cart. When a user selects a product, the system will show it. Next, the user has the opportunity to select the amount, color and size to buy, and click in the “add” button. The system will process the request, by reading the amount, color and size, and adding the product to the shopping cart. The system will also update the cart total, and show a success or error message, depending if the process succeeds.

While creating the use cases, the SCARP users have the **Glossary** (extracted from the domain) available in order to create specifications consistent with the domain. The **RUST specifications**, are used in order to **Validate the (RUS) specifications** (i.e. check if the RUS statements comply with RUST). As output, a **Valid RUS specifications** is produced. The corresponding valid RUS specification for the previously presented textual description is as presented in Table 3.1. It represents the information of the presented textual description, expressed as valid RUS statements. In Figure 3.7 it is possible to see that **Valid RUS specifications** are represented between User and SCARP swimlanes. That representation means that the specifications are available both to the user, to see and manipulate, and to SCARP, in order to process them.

Table 3.1: RUS formatted “Add product to cart” use case.

Step	User input	System response
1	user selects a product	
2		system shows the product
3	user selects the amount, color and size	
4	user selects add	
5		system reads the amount, color and size
6		system adds the product
7		system updates the cart
8		system shows success

3.4.1 RUS

RUS was not developed in order to express highly complex specifications, but focus on the specification of sequential scenarios. Inspired in ACE [73], RUS was designed to handle simple statements, based on the concept of triples $\langle S, P, O \rangle$. The triple elements correspond, respectively to the *Subject*, *Predicate* and *Object* of the sentence. This format provides support to write statements such as **user selects product**, where **user** is the subject, **selects** is the predicate and **product** is the object. In order to improve the expressiveness of RUS statements, it is also possible to add keywords to this triple format. The keywords correspond to words that can decorate the triples, as for instance $\langle S, k1, P, k2, O \rangle$. In practice, the keywords correspond to elements such as prepositions (e.g. *in*, *on*) or determiners (e.g. *the*, *a*). Adding these keywords enables the possibility to create more expressive statements, such as **user selects a product**.

RUS defines two requirements for the specifications. First, use cases are specified with small and simple statements, which comply to the triples format. Second, use cases specifications follow

a tabular format (c.f. [38]). The main scenario should be written as the success case, while possible variations should be written as exceptions and alternatives to the main scenario.

One of the objectives of RUS is to allow statements to keep their readability, despite being converted from natural language to a CNL. Further discussing regarding this objective can be found in two studies, that validate that RUS is expressive enough to handle several kinds of use case scenarios specifications [26, 28] (c.f. Section 5.2).

As stated above, RUS relies on a template language, RUST, in order to perform the process of validation and extraction of information. The use case in Table 3.1 is a valid RUS specification, since all the statements are RUST compliant. Being the specification RUST complied, it is then considered a valid RUS specification.

3.4.2 RUST

In order to define which triples are valid, the RUST template language was created. The templates define how the RUS statements should be written, which both guides the user on how to write them, and, enables the possibility to validate them in runtime. RUST defines also how to the statements should be processed, in order to map them into OWL.

As with RUS, the base of RUST is the concept of triple, for example the RUST statement $\langle S \rangle \langle P \rangle \langle O \rangle$, defines a statement with respectively a *subject*, *predicate* and *object* (c.f. RDF triple). An example of a statement complying with this format is, for instance:

```
system shows product
```

In this case, **system** is the subject, **shows** the predicate and **product** is the object. RUST supports also the specification of statements with a variable number of *objects*. A RUST statement which supports such has the last element of the triples defined as $\langle O \rangle^+$, namely $\langle S \rangle \langle P \rangle \langle O \rangle^+$. An example of a RUS statement supported by this format is:

```
user selects the amount, color and size
```

In this case, the statement is decomposed in the equivalent triple ones, namely:

```
user selects amount
user selects color
user selects size
```

The definition of the valid decorator keywords is also performed in RUST. When creating a RUST, any keyword existing between the triples is then considered a decorator keyword, as for instance $\langle S \rangle k1 \langle P \rangle k2 \langle O \rangle$. In this case, **k1** and **k2** are decorator keywords. As an example, it is possible to define the RUST expression $\langle S \rangle \langle P \rangle \text{the} \langle O \rangle$, which supports the specification of the RUS statement `system updates the cart`.

Considering the specification in Table 3.1 it is possible to see the existence of several kind of statements, such as `user selects add`, `user selects a product`, `system shows the product` or `user selects the amount, color and size`. With the triple format in mind, it is easy to identify the corresponding RUST, which supports those statements. The first statement is the most basic one, simply with three components, therefore the associate RUST is `<S> <P> <O>`. The second one, adds an *a*, between the two last components, therefore `<S> <P> a <O>`. Similarly, for the third one, `<S> <P> the <O>`. The last one, predicts a variable number of arguments, therefore `<S> <P> the <O>+`.

RUST defines also how the extracted triples are mapped into OWL. This process is defined in a simple format to specify how the triples information should be interpreted. Nevertheless, the format to define requirements information is a tuple, which specifies what is an individual, and what are its properties. Namely, by specifying `<Individual: i, property: p>`, where *i* denotes the individual, and *p* denotes the associated property. An example is the definition of facts, in the form `<Individual: i, Facts, f>`. A concrete instance of this example is `Individual:,<S>,Facts:,<P> <O>`. This RUST states that, for a given triple, the subject will be the individual, and the predicate and object will be the associated facts. Other kinds of existing properties are the definition of equivalence (for instance between individuals, and between classes), and annotations (available for all elements). This format was designed with future improvement in mind, and is compatible with the specification of other kind of OWL operators.

Studies performed in the context of this work suggest that a small set of RUST statements is able to support a large number of specifications, since all the statements tend to be very simple [26, 28]. Indeed, only a subset of the RUST presented in Listing A.2 is needed to cover the statements in Table 3.1.

```

<S> <P> <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> a <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> in <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> in a <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> in the <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> the <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> the <O>+ -> Individual: ,<S>,Facts: ,<P> <O>+

```

Listing 3.1: RUST used for the specification of “*Add product to cart*” use case.

3.5 OWL Generation

The second step of SCARP consists in the translation of the use case specification previously written in RUS, into an OWL ontology. The ontology represents an intermediary representation of the use cases information. The information is used as basis for the requirement pattern inference process (Figure 3.8 III).



Figure 3.8: Generating the OWL knowledge base.

Figure 3.9 details the process which supports generating the knowledge base from the provided RUS specification. Apart from the use cases specification, from the previous step, this step requires also the glossary and types information from the domain model, in order to create the corresponding ontology.

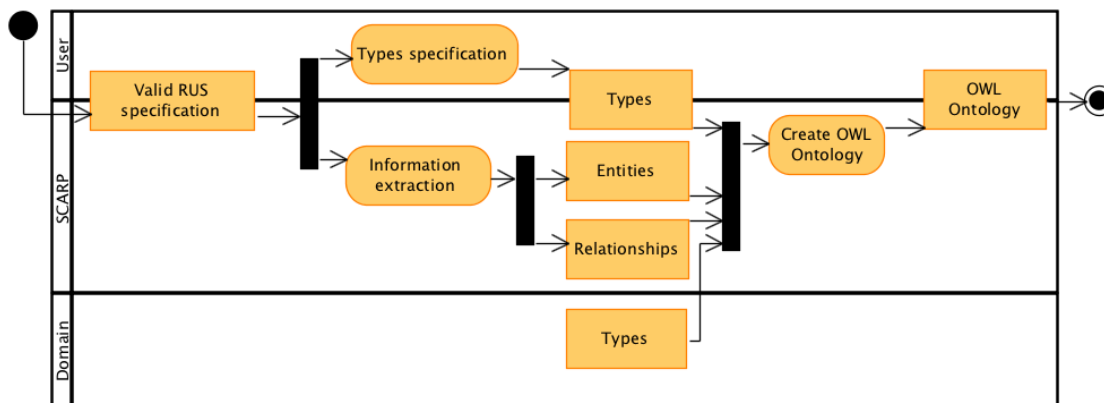


Figure 3.9: Step 2 of SCARP - creating the OWL ontology.

Having the **RUS specification** (resulting from the last step), the process starts with the **Information extraction** process. The process extracts from the specification **Entities** (c.f. **subject predicate** and **object**), as well as **Relationships** between the entities.

Taking, for instance, the first two statements of the use case description in Table 3.1, we have **user selects a product** and **system shows the product**. From these statements it is possible to extract:

- Individuals: **user**, **product** and **system**;
- Predicates: **selects** and **shows**;
- Facts: (**user**) **selects** **product** and (**system**) **shows** **product**.

When mapping this information to OWL, individuals and facts have direct representations. The predicates are represented as *object properties*. The object properties are numbered, according to the order in which they appear in the specification.

The previously presented ontology contains only the entities' information. As aforementioned, the ontology requires also the **types** information. Types in SCARP are available to the user, which can specify and modify them. Hence, Figure 3.9 presents them as part of both the **User** and **SCARP** swimlanes. Figure 3.10 presents an excerpt of the domain model. In order to extract the types information, the terms in the *TypeOf* category in the domain model are analyzed. The figure depicts the relationship between the entities **User**, **System** and **Actor**. It is possible to see both **User** and **System** are connected through a connection named **is**, to the **Actor**. Because **is** is considered a relation of the group *TypeOf*, it is possible to infer that **User** and **System** have the type *Actor*.

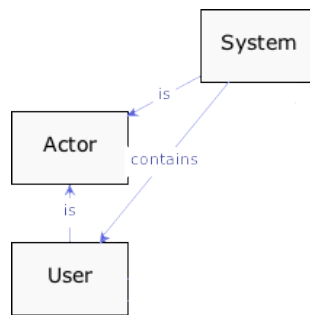


Figure 3.10: Relation between **User**, **System** and **Actor** in the domain model.

In order to complete the ontology is necessary to define additional types, and associate them with the corresponding entities. The types information is both extracted from the relations *TypeOf*, and manually defined by the users.

Providing the types information results in a more complete and accurate ontology. For entities with unspecified types, the generic type *Object* is assumed. At the end of this process, all the information required to produce the ontology is gathered.

3.6 Requirements Patterns

The third step of SCARP (c.f. Figure 3.11), consists in the analysis of the knowledge base containing the use cases information, in order to identify requirement patterns. In SCARP, requirement patterns are inferred in order to obtain hints regarding the desired features in the final solution. The identified features convey a set of properties to address, which in turn can provide hints concerning the architectural solutions which support such properties.

Inferring **Requirement Patterns** consists in applying requirement pattern queries (uQL/SPARQL) to **OWL** (c.f. Figure 3.12).

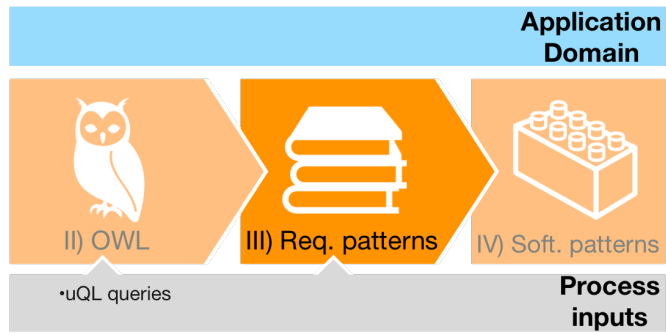


Figure 3.11: Inference of requirement patterns.

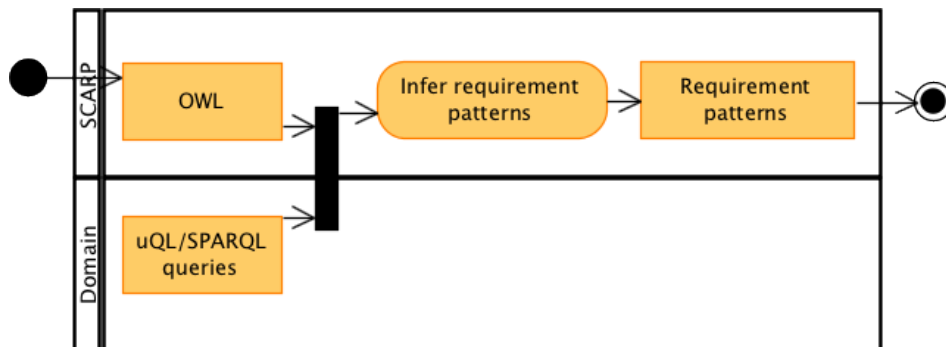


Figure 3.12: Step 3 of SCARP - inferring the requirement patterns.

In order for the requirement pattern inference process to be performed, a set of patterns is required. Requirement patterns are specified as SPARQL queries to be applied to the ontology. The results of applying a SPARQL query provide information regarding which conditions of that query were met, and corresponding context. Thus, describing a requirement pattern as a SPARQL query, provides the possibility to understand if a requirement pattern exists in an ontology, and in which context. In this case, the relevant context information corresponds to the quantity of conditions that exist in the ontology. Having the information regarding the quantity of met conditions, it is possible to understand to which extent the pattern does really exist in the ontology.

Consider for instance, the triples $t1 = \text{“user selects product”}$, $t2 = \text{“system shows product”}$, $t3 = \text{“system lists products”}$ and $t4 = \text{“system creates log”}$. Consider also that the *HasShoppingCart* (*HSC*) requirement pattern contains the triples $t1$ and $t2$, $HSC = \{t1, t2\}$, and a pattern *HasCategories* (*HC*), contains the triples $t2$ and $t4$, $HC = \{t2, t4\}$. If the ontology *Requirements*, contains $t1, t2, t3$, defined as $Requirements = \{t1, t2, t3\}$, then the inference process consists in checking if the ontology contains the triples defined in the patterns. For *HasShoppingCart* and *HasCategories*, the results are $HSC \subset Requirements$ and $HC \not\subset Requirements$, meaning that *HasShoppingCart* exists in the ontology, and *HasCategories* does not exist.

Rather than knowing only if a pattern exists or not in the knowledge base, it is more interesting to know how likely is the pattern to exist. In practice, partial matching consists in identifying the intersection of the pattern triples with the ontology ones, rather than knowing if the pattern is a subset of the ontology. For patterns $HSC = \{t1, t2\}$ and $HC = \{t2, t4\}$, the intersection with the ontology $Requirements = \{t1, t2, t3\}$ can be performed. The results are $HSC \cap Requirements = \{t1, t2\}$, and since the result is equal to the set *HasShoppingCart*, this corresponds to a complete match of *HasShoppingCart* in the ontology, and $HC \cap Requirements = \{t2\}$. In this case, the pattern matches only partially (specifically, in half of the triples). Hence, *HasCategories* exists in the specification, but only partially.

When defining the patterns, the relevance of each triple to that pattern it is also defined. Regarding *HasCategories*, if $t2$ is 3 times more relevant than $t4$, that means the pattern has a matching percentage of 75%.

In practice, the specification of requirement patterns queries can be performed by analyzing descriptions of known patterns, and writing them as queries. An example of the description of a recurring feature is as follows.

The user selects (or opens) a product, and then the system presents (shows, or displays) such product. Next, the user selects the properties for that product, and adds it to the shopping cart. The system reads those properties, and updates the internal data. Next, the system adds the product to the cart, and updates its representation.

This feature describes the process of adding a product to a shopping cart (in an eCommerce domain). The next step consists in creating the corresponding query, by analyzing each statement of the description. SPARQL queries to define requirement pattern can become, however, very verbose. Hence, and since knowledge is mainly described resorting to triples, a new and simpler language to describe requirement patterns is proposed, the uQL. Listing 3.2 presents the corresponding uQL query, for the aforementioned feature.

```
(user) (selects|opens) (?product) 10
(system) (shows|presents|displays) (?product) 20
(user) (selects) (?property) 10
(user) (?) (add) 10
(system) (reads|fetches) (?property) 10
(system) (updates) (data) 20
(system) (adds) (?product) 10
(system) (updates) (cart) 10
```

Listing 3.2: uQL *HasShoppingCart* requirement pattern.

In uQL, each statement corresponds to a condition to be met, that is, to a triple which should exist in the specification. The statements support also the *or* logical operator (c.f. |), named (c.f. ?product) and anonymous (c.f. ?) variables. Finally, each condition has an associated percentage, denoting the weight of that statement in the specification, represented by the number in front of each statement.

In order to provide an overview over the complexity of the three languages, Figure 3.13 presents the three different representations for the first statement of the *HasShoppingCart* requirement pattern. First, it is shown the statement in natural language (a)), and the corresponding uQL (b)). It is also possible to see the SPARQL (c)) representation.

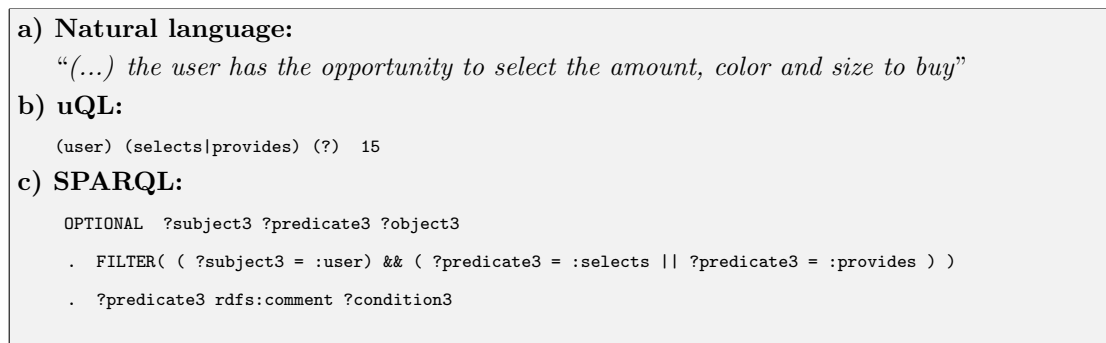


Figure 3.13: Representation of a statement in a) Natural language, b) uQL and c) SPARQL.

3.6.1 Pattern Inference

When a uQL query is applied to the knowledge base, it indicates which statements match. Since statements have an associated weight, the sum of weights of matching statements indicate the matching percentage of the pattern, for the given knowledge base.

Table 3.2: Result of applying uQL in Listing 3.2 to the knowledge base.

uQL condition number	1	2	3	4	5	6	7	8
Match result	true	true	true	true	true	false	true	true
Match percentage	10%	20%	10%	10%	10%	20%	10%	10%
Match result (sum of percentages)	80%							

For the given pattern in Listing 3.2, the result of the matching process is presented in Table 3.2. The table presents, for each condition of the query, the corresponding percentage, and if that condition matched in the knowledge base. For instance, condition number 1 (c.f. `(user) (selects|opens) (?product) 10`) has been found in the knowledge base, and has a weight of 10%. Adding the matching percentage of the conditions, it is possible to state that in this case the pattern would have a matching percentage of 80%. It remains for the user to define a reference percentage value, in order to filter the significative patterns.

3.7 Software Patterns

The fourth step of SCARP (see Figure 3.14) consists in the transition from requirement patterns into software patterns. Requirement patterns describe features, which can be associated with *concerns*. Software patterns on their turn provide functionalities, with the intent to support requirements' concerns. Hence a methodology to establish the bridge between requirements and patterns is proposed.

The inputs for the instantiation process, are requirement patterns, software patterns and the mapping information. As result a set of software patterns is produced, which support the identified requirements. Since software patterns can be seen as building blocks of software, their identification contributes to support producing architectural solutions.

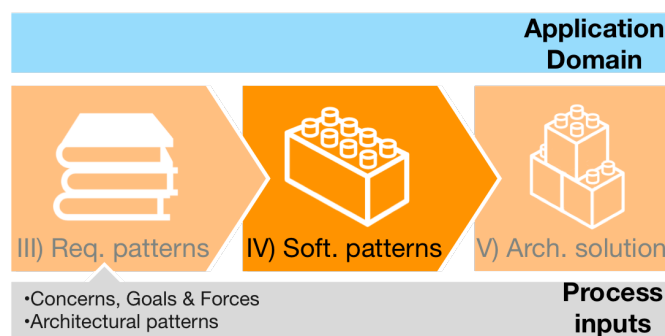


Figure 3.14: Transition from requirement patterns to software patterns.

The process of achieving software patterns from requirement patterns is depicted in Figure 3.15.

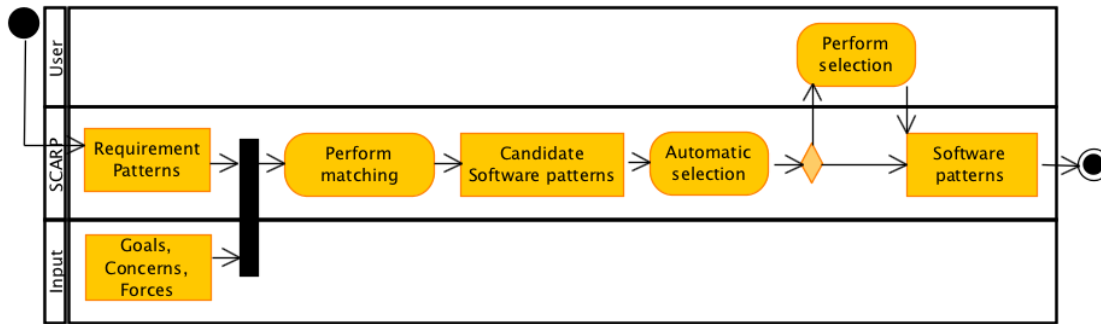


Figure 3.15: Step 4 of SCARP - Transition from requirement to software patterns.

The first step of the process is the input of the **Requirement Patterns**. These patterns result from the inference process in the previous step. The **Goals**, **Concerns** and **Forces**, part of the process inputs, provide the required information in order to match the requirement patterns to software patterns (i.e. **Perform matching**). This matching process results in a set of **Candidate software patterns**. It is required to either manually **Perform selection** of adequate patterns, or perform an **Automatic match** of patterns. The process ends with a set of **Software patterns**, representing the solutions achieved to better handle the provided requirement patterns.

3.7.1 Goals and Concerns

Patterns can be characterized by different aspects, representing different levels of abstraction. In SCARP three levels of abstraction are considered, namely *Concerns*, *Goals* and *Forces*, as depicted in Figure 3.16.

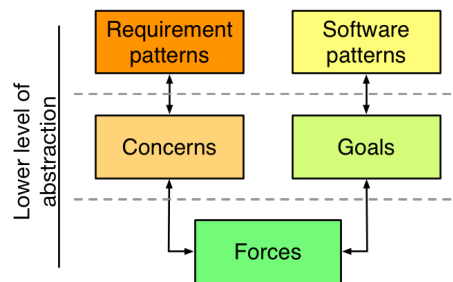


Figure 3.16: Attributes of patterns at different levels.

Requirement patterns' *Concerns* provide information about properties to be fulfilled, in the system to be implemented. *Goals* on their turn, provide information regarding the benefits that an architectural pattern provides when applied. In the one hand, the users' needs (c.f. concerns)

can be identified. On the other hand, the advantages provided by the usage of a certain pattern (c.f. goals) can be identified. These two information groups can be seen as complementary. The process of matching concerns to goals consists in identifying which goals are the most suitable to support a set of concerns. However, there is still a gap between those two artifacts. They exist at the same abstraction levels, but represent different aspects. The specification of the concerns to goals mapping information is required in order to automate the transformation process. The process requires the identification of both concerns and goals, and consequent decomposition into forces.

Identifying concerns in requirement patterns can be achieved by reading and understanding the meaning of each pattern being addressed. Patterns are by nature well defined and described. That makes it possible to extract the concerns they convey. Following an approach similar to Bass *et al.* [4], by reading the description of the patterns it is possible to extract their concerns.

For the process supporting a shopping cart in an eCommerce platform (c.f. *HasShoppingCart*), for instance the Amazon website ¹, has the following description:

If you want to order an item, click the Add to Basket button on the item's product detail page. Once you've added an item to your Basket, you can keep searching or browsing until your Basket contains all the items you want to order. You can access the contents of your Basket at any time by clicking the Basket button on any page.

Tip: If your Shopping Basket is empty or items are missing from it, it's likely that you're not logged into your account. Placing an available item in your Shopping Basket doesn't reserve that item. Available stock is only assigned to your order after you click Place your order and receive an e-mail confirmation that we've received your order.

You can modify an item in your Shopping Basket: To change the quantity, enter a number in the Quantity box and click Update.

Note: We strive to provide customers with great prices, and sometimes that means we limit quantity to ensure that the majority of customers have an opportunity to order products that have very low prices or a limited supply. We may also adjust your quantity in checkout to reflect your recent purchases of a quantity-limited item. To remove an item from your Shopping Basket, click Delete. To wait until another day to buy some of the items in your Shopping Basket, click Save for later. This will move the item to your Saved for Later list located below the Shopping Basket. Click Move to Basket next to an item when you're ready to purchase it.

Note: Items in your Shopping Basket will always reflect the most recent price displayed on the item's product detail page. This price may differ from the price the item when you first placed it in your Basket.

¹<http://www.amazon.co.uk/gp/help/customer/display.html/?nodeId=502528&qid=1448537481>, last visited on 2015-11-26

By reading the description it is possible to extract several relevant functional features. In the presented sentences it is possible to identify concerns:

- “If you want to order an item, click the Add to Basket button on the **item’s product detail page**.” - states that a product contains details. Hence, the *detailable* concern;
- “You can **access the contents of your Basket** at any time by clicking the Basket button on any page” - states that both the cart contains all the items, and belongs to a user. Hence, respectively the *listable* and *manageable* concern;
- “Placing an available item in your Shopping Basket doesn’t reserve that item. Available stock is only assigned to your order after **you click Place your order** and receive an e-mail confirmation that we’ve received your order.” - states that there is an explicit process to process the order. Hence, the *processable* concern;
- “You can **modify** an item in your Shopping Basket: To change the quantity, enter a number in the Quantity box and click Update.” - clearly states that the user has the ability to modify the cart content. Hence, the *editable* concern.

The concerns identification process depends on the interpretation of the descriptions, making the process vulnerable to subjectivity. As result, some concerns of the description might be missing in the extracted ones. Identified concerns support the architectural artifact identification process. Concerns missing from the identification process can have as consequence missing architectural artifacts. If such problem occurs, the concerns information should be refined, and the matching process repeated. The refinement of the patterns is a continuous process. In this example it was possible to identify, for the *HasShoppingCart* pattern, the associated *detailable*, *listable*, *manageable*, *processable* and *editable* concerns.

Identifying goals on software patterns can be done by analyzing pattern catalogs. Pattern catalogs (c.f. [41]), contain several fields to describe a pattern. One of those fields is the *intent*, which details what benefits the patterns provide when used. Usually, the intent field contains enough information to derive several goals. Another field that is useful to identify goals is the *description*.

Considering, for instance, the *Proxy* pattern [41], the following intent can be found: “Provide a surrogate or placeholder for another object to control access to it.” The pattern intent is very clear, but it is possible to further read in the pattern’s applicability: “A protection proxy controls access to the original object”. Hence, this pattern has the goal:

- *Delegate* - act as a bridge, or, surrogate to handle the client requests.

The *Memento* pattern has the intent: “Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.” From the pattern intent, it is possible to highlight the capability this pattern provides to “capture and externalize (...) state”. Furthermore, it allows to later “restore” the state. In the pattern applicability is is

possible to read: “a snapshot of (some portion of) an object’s state must be saved so that it can be restored to that state later”. From the description, it is then possible to extract the goals:

- *State* - the possibility to keep the state for a set of objects whose state should be maintained;
- *Edit* - the possibility to edit the state, with the intent (for instance) to restore object states.

Another example is the *Command* pattern, with the intent: “Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.” In its intent, it is possible to highlight the “parameterize clients with different requests”. From the pattern’s applicability, we can read “parameterize objects by an action to perform”. Hence, the goal:

- *Process* - represents the capability to perform a parametrized processing operation on objects.

3.7.2 Forces

In the patterns community the term *Force* (originally borrowed from architecture and Christopher Alexander’s work [1]) is used to denote (c.f. Buschmann): “any aspect of the problem that should be considered when solving it, such as Requirements (what the solution must fulfill), Constraints (things to consider), and Desirable properties (that the solution should have)”. The identification of a pattern’s forces can be done by analyzing the catalog in which the pattern is described.

In the SCARP’s context, *Forces* identify the specific impact of a *Concern* or *Goal* in the final solution. Pattern’s forces enable the possibility to match concerns to goals, as forces are used to describe both requirement and software pattern at the same level.

Forces on Concerns

By analyzing existing concerns, a set of forces can be defined (as proposed by other authors [85]). Several concerns, including for instance **Manageable**, **Processable**, **Editable** were analyzed and the following set of forces extracted.

- Abstraction - provide a layer to abstract the content;
- Separation - create a level of indirection to mediate access to content;
- Decoupling - provide means to loosely couple two elements;
- Coupling - enforce the relation between two elements;
- Direction - remove intermediary elements in the access of two elements;
- Indirection - support delegating a request;

- Computability - the capability for an element to be processed;
- Versioning - the capability to keep one or more states of an object;
- Flexibility - the capability to adjust in runtime;
- Constraint - removes flexibility to ensure consistency.

Considering the impact of forces on concerns, the following associations can be made. In the association, the symbol + or - denotes if that concern is positively or negatively affected by that force.

- Manageable:
 - Abstraction (+) - When an object is managed by another one benefits from abstracting what is being owned;
 - Separation (+) - An indirection in the reference for the owned object is beneficial as it keeps the owned data transparent to the user;
 - Decouple (+) - Decoupling the managed object allows it to evolve separately;
 - Couple (-) - Coupling the object to the owner, negatively affects the owner when the object changes;
 - Direct (-) - Direct linking the owner to the object can provide too much control over it.
- Processable
 - Computability (+) - A processable object, benefits from the possibility to be processed;
 - Indirection (+) - Delegating a task to another entity introduces flexibility in the process;
 - Separation (-) - An indirection level on processable introduces an undesired intermediate step of complexity.
- Editable
 - Versioning (+) - Being able to keep one or several versions benefits the fact of an object being editable;
 - Flexibility (+) - If we have the concern of editing, then having the capability to adjust it in runtime is a positive effect;
 - Constraint (-) - Removing flexibility in the solution is undesired for the editable concern.

The information regarding the requirement patterns, concerns, and forces is summarized in Figure 3.17.

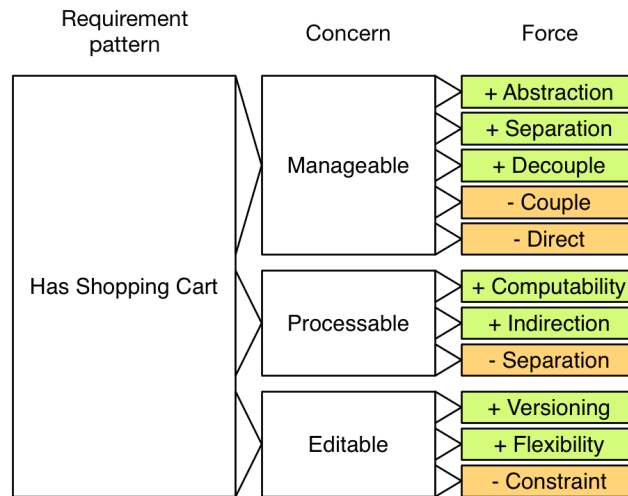


Figure 3.17: Requirement patterns, concerns and forces for the *HasShoppingCart* requirement pattern.

After having defined the forces for the concerns, it is now required to identify the forces on the goals. Such is essential to have both information described at the same level.

Forces on Goals

Software patterns are usually documented and described with their associated forces. Pattern catalogs (e.g. [15]) contain for each pattern the corresponding force. As an example, and according to Buschmann, the proxy pattern contains the forces *Efficiency*, *Decoupling*, *Separation*, and on the negative side *Efficiency* and *Overkill* [15]. It is also possible to say that this pattern supports abstraction, hence the *Abstraction* goal. The proxy pattern, contains only the *Delegate* goal, therefore all forces belong to that goal. The relation regarding the software patterns, goals and forces is summarized in Figure 3.18. Figures 3.19 and 3.20 present the same information, for the *Memento* and *Command* patterns, respectively.

In the context of SCARP, concerns, goals and forces are represented as an ontology. Such approach enables the possibility to automatically analyze the relations between those elements.

3.7.3 Matching Process

The automatic matching process, as represented in Figure 3.21, is composed of two parts. Part **A)** consists in the extraction of concerns from requirement patterns, and respective forces. In **B)**, concern forces are matched against goals forces, in order to identify the corresponding goals. From the goals it is possible to identify the software patterns.

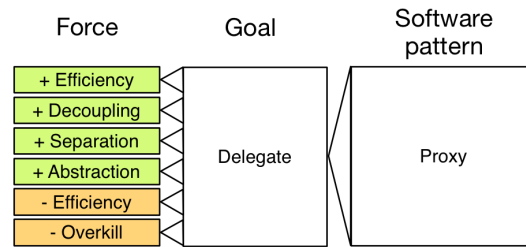


Figure 3.18: Software patterns, goals and forces relationship for the *Proxy* software pattern.

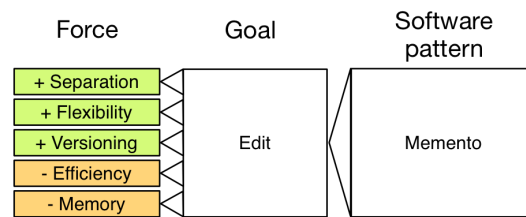


Figure 3.19: Software patterns, goals and forces relationship for the *Memento* software pattern.

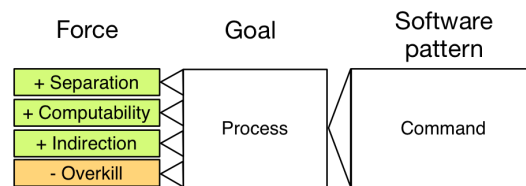


Figure 3.20: Software patterns, goals and forces relationship for the *Command* software pattern.

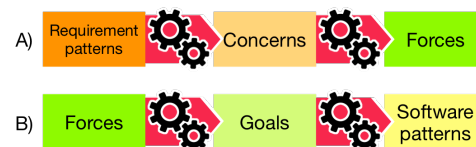


Figure 3.21: Requirement patterns to software patterns process flow.

Performing the matching process consists in the application of the queries presented in Listing 4.9 (see page 96), to the ontology representing the requirements information. The result of the matching process in the Has Shopping Cart (HSC) pattern is presented in Table 3.3 and Table 3.4. From the table it is then possible to extract, for instance, that the *Flexibility* force has a positive impact both on the *Editable* concern and on the *Edit* goal, therefore relates positively the Memento and *HasShoppingCart* patterns.

Table 3.3: Positive contributions between requirement patterns and software patterns (through forces), as support for the matching process.

Requirement pattern	Concern	Force	Goal	Software Pattern
HasShoppingCart	Manageable	Separation	Edit	Memento
HasShoppingCart	Manageable	Separation	Process	Command
HasShoppingCart	Manageable	Separation	Delegate	Proxy
HasShoppingCart	Manageable	Abstraction	Delegate	Proxy
HasShoppingCart	Processable	Computability	Process	Command
HasShoppingCart	Processable	Indirection	Process	Command
HasShoppingCart	Editable	Flexibility	Edit	Memento
HasShoppingCart	Editable	Versioning	Edit	Memento

Table 3.4: Negative contributions between requirement patterns and software patterns (through forces), as support for the matching process.

Requirement pattern	Concern	Force	Goal	Software Pattern
HasShoppingCart	Processable	Separation	Edit	Memento
HasShoppingCart	Processable	Separation	Process	Command
HasShoppingCart	Processable	Separation	Delegate	Proxy

Figure 3.22 presents the combination of the aforementioned tables. On the left (**Concerns**) the existing concerns for the *HasShoppingCart* requirement pattern are described. For each *Concern*, the corresponding *Goals*, which are related via the same *Forces* are presented. On the right (*Software Patterns*) the software patterns for the identified *Goals* are presented.

Following the pattern selection algorithm, the mapping for the *Manageable* pattern (c.f. Figure 3.22) is as follows.

- Manageable → Delegate: 2 positive forces: *Abstraction*, *Separation*, and 0 negative forces.
- Manageable → Process: 1 positive force: *Separation*, and 0 negative forces.
- Manageable → Edit: 1 positive force: *Separation*, and 0 negative forces.

Based in this information, it is possible to state that the most suitable goal to respond to the *Manageable* concern, is the *Delegate*. By applying this process iteratively to each concern and goal, it is possible achieve the information of how each concern matches a goal. The resulting output is the forces matrix (c.f. Table 3.5).

The forces matrix, as presented on Table 3.5, shows how each pattern influences the other ones. On the left, the table shows the identified concerns. On the top, the *Goals* related with the *Concerns* are shown. Each cell of the table presents the positive (+) and negative (-) amount of forces which relate each *Concern/Goal* pair. For instance the **Processable/Process** pair, is related via two positive forces (+2) and one negative force (-1).

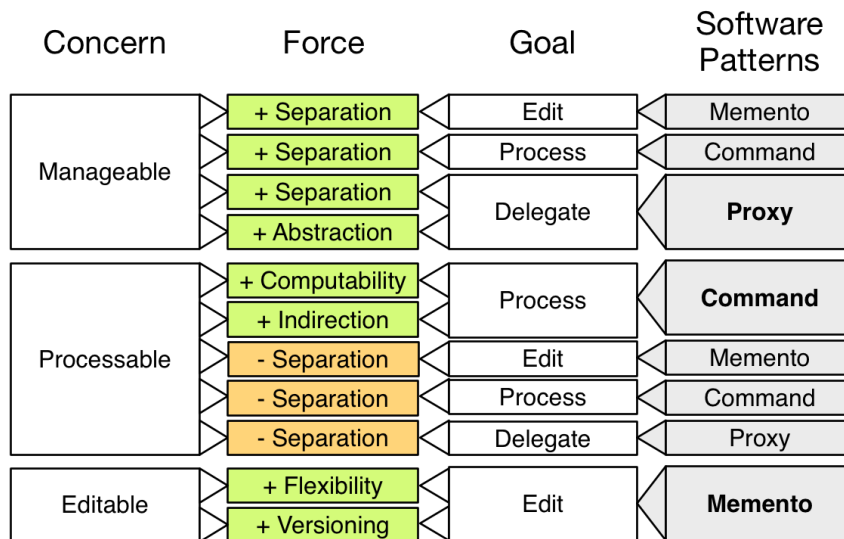


Figure 3.22: Summary of the relations between concerns, forces, goals and software patterns, for the *HasShoppingCart*.

Table 3.5: Matrix describing the relations between forces for the *HasShoppingCart* requirement pattern.

		Goals		
		Delegate	Edit	Process
Concerns	Processable	(+0, -1)	(+0, -1)	(+2, -1)
	Editable	(+0, -0)	(+2, -0)	(+0, -0)
	Manageable	(+2, -0)	(+1, -0)	(+1, -0)

Based on the forces matrix, it is possible to conclude that the best match for *Manageable* concern is the *Delegate* goal, and *Delegate* is supported by the *Proxy* software pattern. Hence, *Proxy* is a software pattern suitable for *HasShoppingCart* (c.f. Figure 3.22). Similarly, the *Editable* and *Processable* concerns have the best match with *Edit* and *Process* (respectively). Thus, *Command* and *Memento* are also patterns required to support the *HasShoppingCart* requirement pattern. This resulting matching information can be used to either automatically select the software pattern (as presented), or guide users to manually perform the pattern selection process.

3.8 Architectural Solution

The fifth step of SCARP (c.f. Figure 3.23) consists in the transition from software patterns, into a corresponding architecture. The process relies in the input of the software patterns from the

previous step, and information regarding the attributes and relationships of the entities, from the domain model. The produced architectural model represents a solution which supports the specified use cases. There are several software pattern catalogs available to support this step of SCARP. Gamma *et al.*'s design patterns catalog [41] was selected in order to illustrate SCARP, since it is one of the most popular patterns catalog available. The process to achieve an

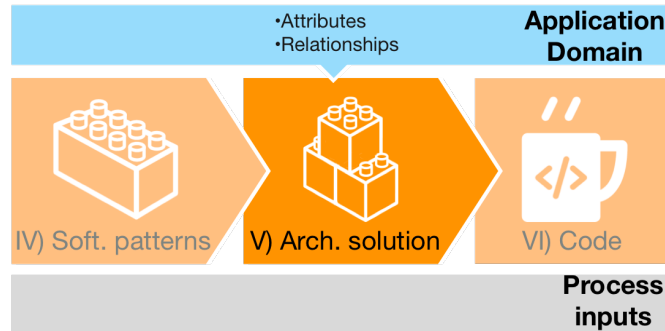


Figure 3.23: Production of the architectural solution.

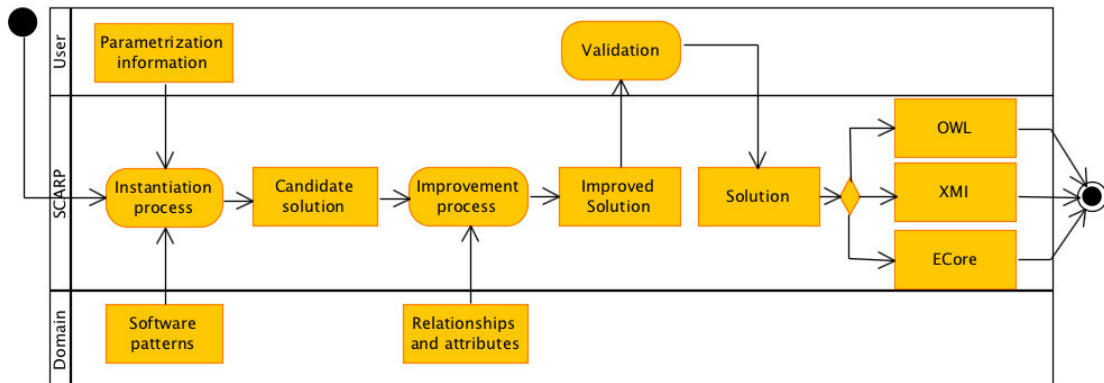


Figure 3.24: Step 5 of SCARP - producing an architectural solution from a set of software patterns.

architectural solution from a set of software patterns (c.f. Figure 3.24) starts with the **instantiation process**, of **Software patterns**. This process requires the input of those **software patterns** and, with **parametrization information** provided by the user, SCARP is able to perform the **instantiation process**. Such process produces a candidate solution, by merging the software patterns (resorting to *stringing* and *overlapping* operators). The **improvement process** enhances the achieved solution, and requires information regarding **relationships and attributes** from the domain, and combines it in the candidate solution. This process results in an **improved solution**, which after a **validation** from the user, results in the final **solution**. The achieved solution can be serialized into **XMI**, or other computable formats.

The iteration of early architectural models has been successfully adopted in practice, for instance, through evolutionary prototyping [80, 52]. Evolutionary prototyping is the process of creating an early prototype, which is iteratively refined until becoming the final solution. In SCARP, the produced models can be used as basis for such an approach. The refinement of software models can occur at the requirements level, while produced models (prototypes) can be iterated at the MDA level.

3.8.1 Software Pattern Definition

The chosen approach to support the automated processing of patterns, is to represent them as an Application Programming Interface (API)-like format, composed of a **name** and a set of **parameters** (e.g. `name(parameter1, parameter2, ...)`). In this case, the **name** of the pattern corresponds to the name of the method, while the **parameters** correspond to the name of the pattern constituents. The *Proxy* (c.f. Figure 3.25) pattern contains the elements *client*, *Subject*, *RealSubject* and *Proxy*. The representation of this pattern is then `proxy(client, subject, realSubject, proxy)`. Alongside with the definition of the pattern it is necessary to provide information regarding the role of each constituent, or participant (c.f. [41]). Hence, and according to the proxy pattern definition, the `client` triggers the request, the `proxy` “maintains a reference that lets the proxy access the real subject” or the interface that handles the request, the `subject` “defines the common interface for *RealSubject* and *Proxy*”, and *RealSubject* “defines the real object that the proxy represents”. Following an approach similar to Java and Javadoc, a pattern signature is defined as presented in Listing 3.3, which includes also the pattern intent to better document the pattern.

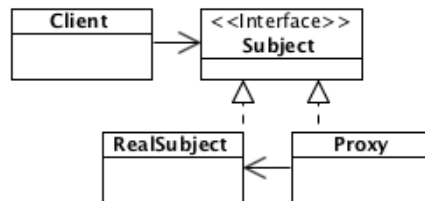


Figure 3.25: *Proxy* design pattern structure (adapter from [41]).

```

/**
 * @intent Provide a surrogate or placeholder for another object to control access to it.
 * @param client      Triggers the request.
 * @param subject     Defines the common interface for RealSubject and Proxy.
 * @param realSubject Defines the real object that the proxy represents.
 * @param proxy       Entry point to handle the request.
 */
proxy(client, subject, realSubject, proxy);

```

Listing 3.3: Proxy software pattern specification in API-like format.

3.8.2 Instantiation Process

Having the pattern specified in the API-like format, the instantiation process corresponds then to a *invocation* of that API. Such supports the creation of concrete software pattern instances, as support for the composition process. Hence, it is necessary to define the concrete names for the parameters, according to the provided documentation. With this approach it is possible to automate the pattern instantiation process, while abstracting the user from the pattern's details.

In the inference process the **Proxy** pattern was identified as a software pattern to be integrated in the *HasShoppingCart*, in support for the *Manageable* concern. The parametrization information consists in the definition of the parameters to be provided to the instantiation process. Hence, the **client** owning all the instances is the **System** itself. The **realSubject** being abstracted corresponds to the shopping **Cart**, which is the entity being held and accessed. Thus, the **subject**, being an interface, is the **ICart**. Finally, the object to intermediate the access to the cart (i.e. **Proxy**) can be the **User**, which can handle the interactions with the cart. According to this definition, the specification of the **Proxy** pattern is `proxy(System, ICart, Cart, User);`. The parametrization information is summarized next. The result of the *Proxy* pattern instantiation is shown in Figure 3.26.

- **client: System** - The system triggers changes in the cart;
- **subject: ICart** - The interface which defines the cart actions;
- **realSubject: Cart** - The implementation of the cart;
- **proxy: User** - The user handles the instance of the cart.

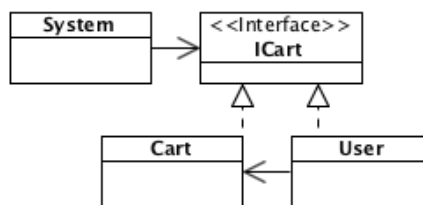


Figure 3.26: Proxy design pattern instantiation.

Recurrently applying this process to all inferred software patterns, results in a set of pattern instances which are representative of the solution to be implemented. These instances contain the classes to exist in the solution.

3.8.3 Pattern Composition

After the instantiation process, follows the unification of the several instances into a single solution. Although there is no standard approach, several authors propose different alternatives to perform the composition process. In this work the usage of *stringing* and *overlapping* [122]

composition approaches is explored. Despite the large amount of available work regarding pattern composition, *stringing* and *overlapping* are two of the most well known and accepted approaches.

The instances composition process is based in the overlapping technique. Overlapping supports performing a unification operation over the classes. The unification process is performed by considering classes with the same name to represent the same entity, and merging those classes into a single one.

To apply the composition technique to *Proxy*, *Memento* and *Command* patterns instances, the identification of common classes is necessary. For instance, the shopping **Cart** is a class common to all three instances. Hence, and according to the overlapping technique, **Cart** class represents the same entity in all the instances. By applying the same process to the remaining classes, the architectural solution presented in Figure 3.27 is achieved. In the figure, the *Proxy* pattern is represented in grey, the *Memento* pattern in blue and the *Command* in yellow.

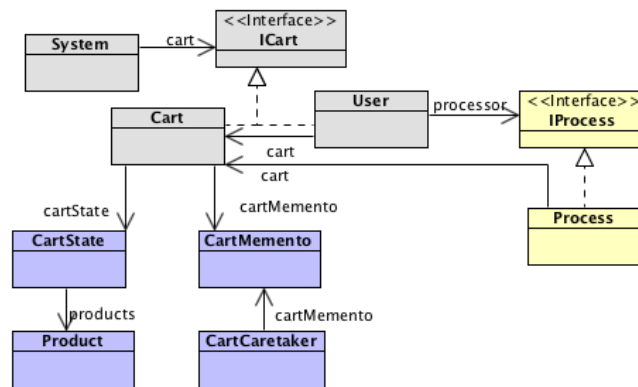


Figure 3.27: Composition of *Proxy*, *Memento* and *Command* patterns through overlapping.

The solution presented in Figure 3.27 was produced from the instantiation of the *Proxy*, *Commando* and *Memento* software patterns. After the instantiation, a pattern composition process followed, resorting to the overlapping operator. **User** and **Cart** classes were identified as common entities in all pattern instances, and merged, resulting in the presented architecture.

3.8.4 Solution Enhancement

The solution enhancement step, performed over the unified solution, resorts to the stringing composition technique. In the domain model it is possible to see, for instance, that **System** is related with **Product** via an association of the category *CompositionOf*, in this case **contains** (c.f. Figure 3.28). Therefore, a new association needs to be added to the solution.

By analyzing the domain model relations of the type *CompositionOf* for the remaining classes in the solution, the improvement of the solution is possible, as depicted in Figure 3.29.

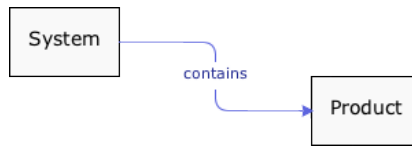


Figure 3.28: Relationship between **System** and **Product** extracted from the domain model.

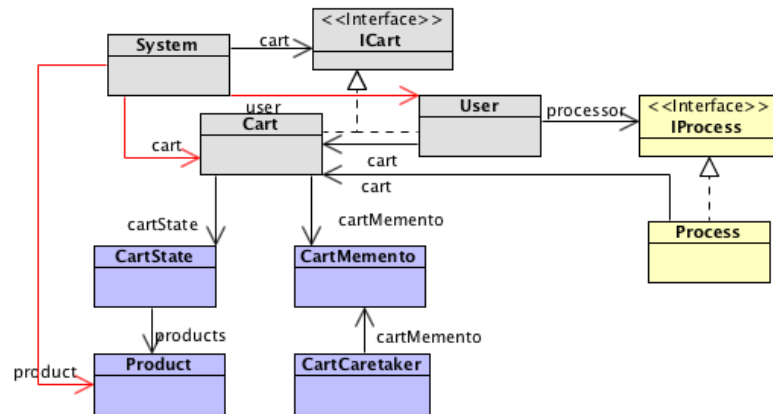


Figure 3.29: Enhanced version of the solution resulting from the pattern composition process, regarding relationships.

The final step of the composition process is the enhancement of the classes, by providing their attributes. The entities related via associations of the category *PropertyOf* in the domain model, as is the example of **has** relationship are analyzed. It is possible to see, for instance, that, the entity **User** is related (among others) with **Age**, **Name** and **Address** (c.f. Figure 3.30). Hence, the two entities represent attributes of the **User** class.

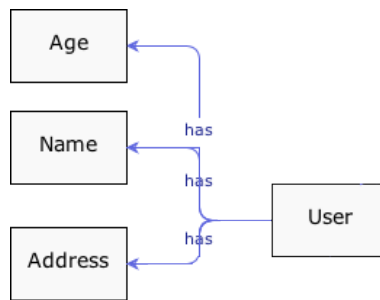


Figure 3.30: Relationship between **User**, **Age**, **Name** and **Address** extracted from the domain model.

Having identified the attributes, it remains to specify their types. This step requires input from the SCARP users. An example of types definition, for the three described attributes is as follows.

- Name is a String;
- Age is an Integer;
- Address is a String.

After defining the types, the enhanced solution can be generated. The result is presented in Figure 3.31.

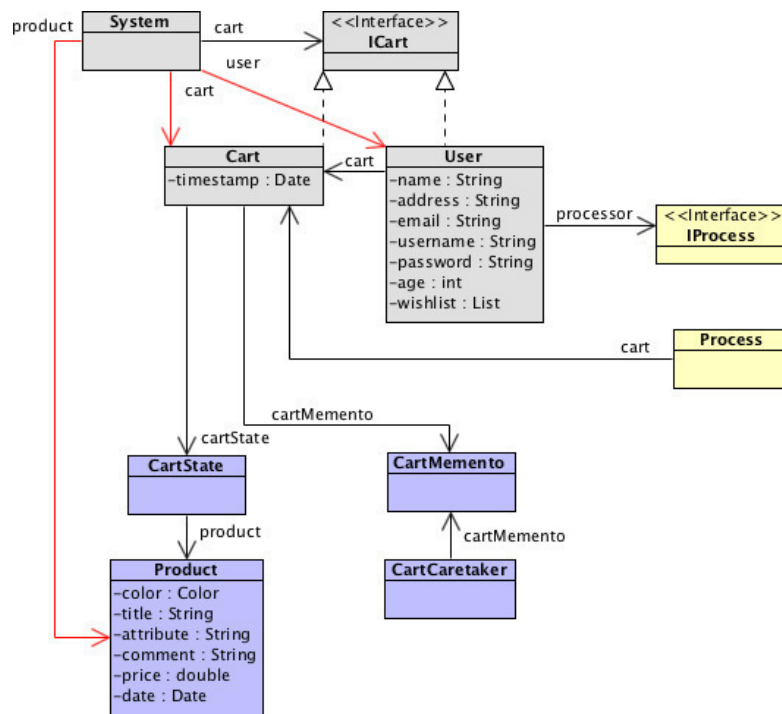


Figure 3.31: Enhanced version of the solution resulting from the pattern composition process, regarding relationships and attributes.

3.8.5 Solution Validation

After the creation of the unified and improved solution in SCARP, it remains to validate the produced solution. Since both composition and enhancement are automated processes, there is a possibility for incorrect information being added to the solution. Two main scenarios can occur regarding incorrect and incomplete information in the final solution. The first scenario is the addition of unnecessary (e.g. duplicate) relationships between entities, and the second one is the addition of inappropriate (e.g. erroneous) attributes.

The last step of the composition process consists in manually verify the resulting model. An example of an error resulting from the enhancement process, is the association between **System** and **Cart**. Indeed, **System** should not be associated with **Cart**, since an association to **ICart**

already exists. Thus, this association should be removed. The user has also the capability to finish the model itself, adding the remaining attributes. An example of the revised solution is the one presented in Figure 3.32.

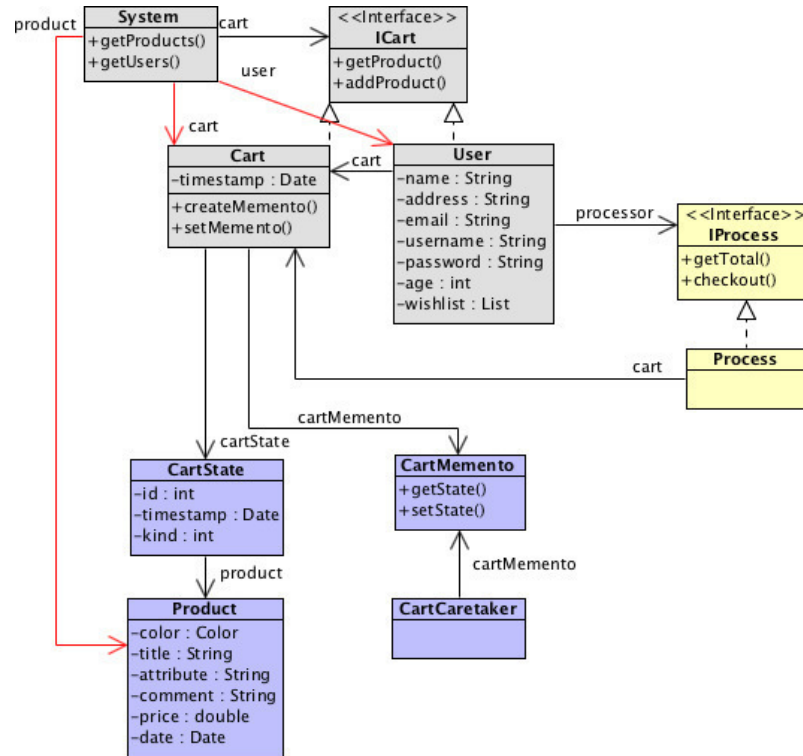


Figure 3.32: Revised solution

It is expected that the users perform this verification process in order to validate the automatically generated solution. After this validation the final solution is then produced. This final solution corresponds to the architecture which supports the requirements specified at the beginning of the process, considering the provided domain model. The following steps consist in transforming the solution into an interchangeable format.

3.8.6 Serialization

After performing the solution validation, it is required to select a format to serialize the result. The serialization process consists in translating the previous information into a computable format (e.g. XMI).

The objective of the serialization is twofold. On the one hand, it is intended to support the next step of SCARP, by producing source code. On the other hand, it is designed to support interoperability with other processes and tools, by producing formats which are known to be

widely adopted. In SCARP, a substantial emphasis was put in the adoption of standard formats. The adoption of standards, demonstrates the open nature of SCARP, regarding the operability with other approaches and tools.

3.9 Code

The final step of the SCARP approach (Figure 3.33, VI) consists in generating code, from the architectural model provided as output of the previous step. The produced code corresponds to the implementation of the architecture obtained so far from the SCARP process. The code corresponds also to the final output of SCARP. The process starts with the input of the

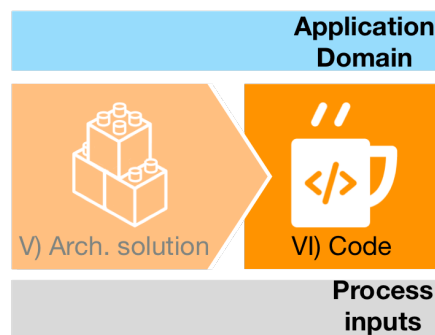


Figure 3.33: Generating source code.

solution achieved in the previous step. The user performs the **tool selection** process, in order to choose a suitable MDA tool for the desired outputs. The tool proceeds then to the **code generation** process, producing the **source code** which represents the solution.

Achieving source code from UML diagrams (as represented in XMI, for instance), is a well known process, addressed by the MDA. The only relevant question to address in this step is the selection of the tool which supports the code generation process.

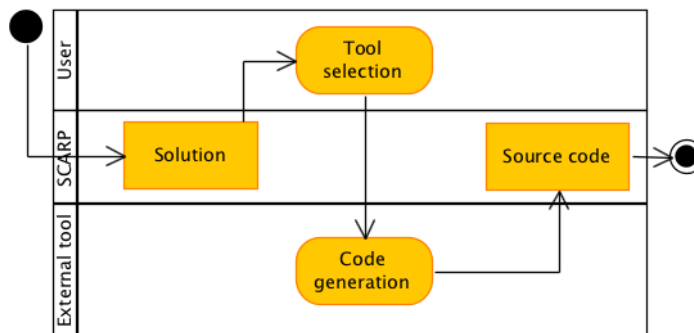


Figure 3.34: Step 6 of SCARP - producing source code.

3.10 Summary

This chapter presented the SCARP approach, as a process which supports the generation of architectural artifacts, from a set of requirement specifications. In order to do such, requirements representation techniques, pattern inference and matching processes, pattern instantiation and composition approaches were described. Each step of the approach was described, explaining the corresponding required input, processes and outputs. An overview of the process was also provided, by presenting how the steps link to each other (via their inputs and output), and how the process integrates in the MDA.

Chapter 4

Instantiation of SCARP

This chapter describes uCat, a tool implemented to support the SCARP process. For each step of SCARP, uCat provides several features which support the process. Alongside the description of the decisions taken to provide a concrete implementation for SCARP steps, the chapter presents how uCat supports each step.

4.1 uCat Tool

uCat was developed in Java in order to provide native multi-platform support. In order to make it portable, there are also no external dependencies (e.g. databases or reasoners). All the required features were embedded in the application. Providing a platform independent and dependency free application was useful to support the validation studies, and will help foster the adoption of the tool in the future. The use of standard technologies and languages (e.g. OWL, XML, SPARQL), is relevant to both foster interoperability and enable the possibility to use external tools to help in the process.

Figure 4.1 depicts uCat's architecture. A modular format (built upon a plugin architecture) was selected in order to implement the solution. Such approach improves the application quality, allowing each plugin to exist independently. Also, it was possible to perform an incremental development, by implementing the plugins as required. Another objective of the plugin architecture is to provide a simple way to extend uCat. On the one hand, adding new features can be done through the implementation of new plugins. These plugins can communicate with existing ones via an internal messaging system, without the need to modify the latter. On the other hand, existing plugins can be individually improved. Finally, the plugin architecture supports the adjustment of uCat according to the user needs. Considering, for instance, there is only the need to create specifications, plugins concerning inference (e.g. pattern inference) can be removed. Indeed, for the studies presented in Chapter 5, several plugins were removed from uCat in order to allow users to focus in the features being evaluated.

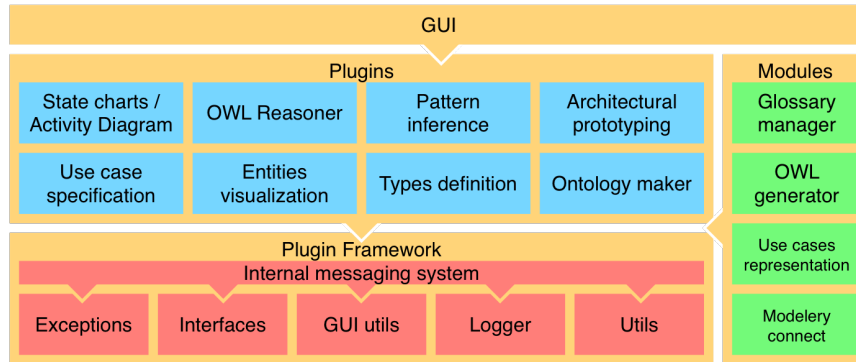


Figure 4.1: uCat architecture.

At the base of the architecture is a plugin framework, which supports the modular development process. The plugin framework is parametrizable through a manifest file, which must contain the name of the application, version and included plugins. The framework provides also support for the Java interfaces that defines a plugin's behavior. The GUI utils component provides common GUI elements, such as alert windows or input/output dialogs. Also part of the framework is the Logger, which supports logging the user activities. Finally, the plugin framework provides some utils in support for file read/write and control of the main window. The internal messaging system allows the plugins to communicate with each others and with the framework itself. The contribution of each plugin is described in the following sections.

4.2 Domain Model Specification

In uCat, the *domain model specification* plugin provides a visual editor, where it is possible to create or load the domain model for the application being generated. The domain model is created resorting to a subset of the language available to create UML class diagrams. The plugin supports the creation of entities and their relationships in order to create the domain model. An example of a domain model created in this plugin is presented in Figure 4.2. On the left are shown the elements that can be added to the model (**entities** and **relationships**), and on the right the model itself. The plugin supports also exporting the data from the domain model to an OWL file, saving the diagram as an image, and some customization options (as colors and fonts).

In order to support automatic information extraction, the domain model representation in OWL is required. The process to create an ontology containing the domain model information is composed of two steps. First, each entity of the domain model is converted into an OWL instance. Each relationship is converted into an OWL object property, which associates two individuals.

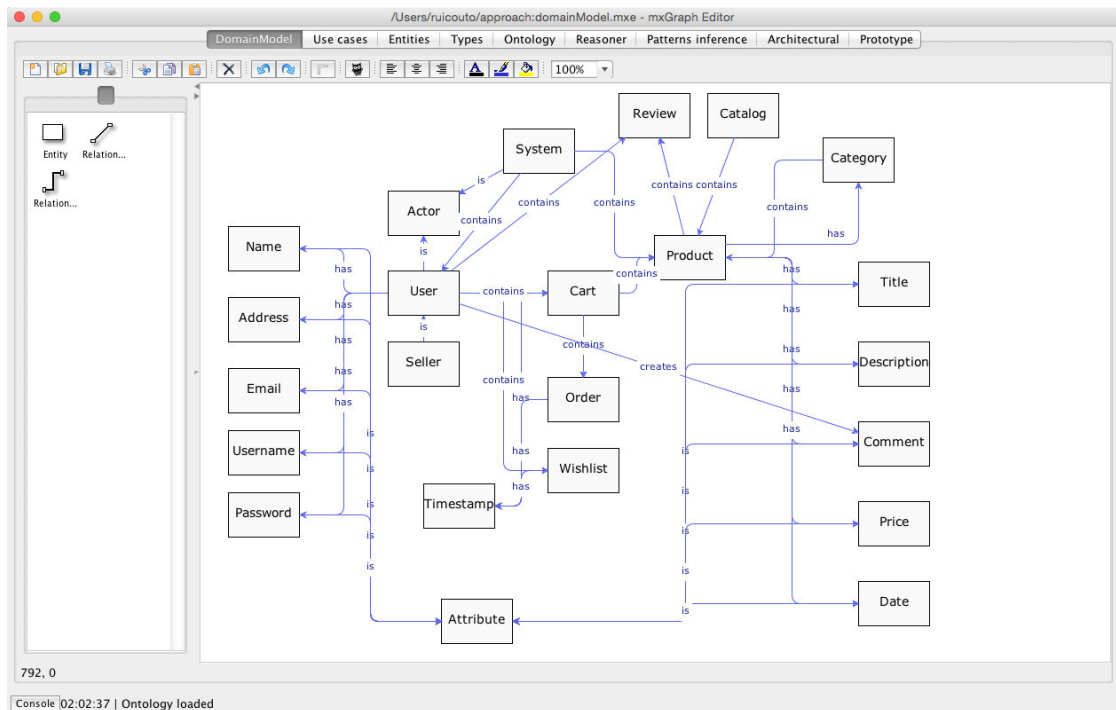


Figure 4.2: Domain model plugin user interface.

uCat verifies the consistency of the model prior to the generation of the knowledge base. Specifically, missing names for entities and relationships are checked. Also, warnings are shown for entities names not belonging to any category of the defined types. An example of the error for a missing name in a relationship is shown in Figure 4.3.

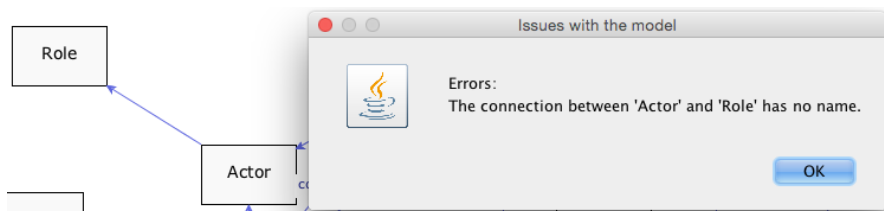


Figure 4.3: Indication of an error in the domain model.

An example of the resulting ontology is presented in Listing 4.1. In the ontology, the existence of several `ObjectProperties` (e.g. `is`, `has`) is specified, as well as their categories via OWL classes (c.f. `TypeOf`, `PropertyOf`, `CompositionOf`). The definition of a connection type is done via the `Domain` property in the ontology. It is also specified (for instance) that `Product` is part of `Cart`, since the entities are related via an association of the type `CompositionOf`. The representation of the domain model entities is done via OWL `Individual` (e.g. `Cart`, `User`). See Appendix A.1 for the full specification.

```

Ontology: <dm>

Class: owl:Thing
Class: <dm#TypeOf>
Class: <dm#PropertyOf>
Class: <dm#CompositionOf>

ObjectProperty: <dm#contains>
  Domain:
    <dm#CompositionOf>

ObjectProperty: <dm#is>
  Domain:
    <dm#TypeOf>

ObjectProperty: <dm#has>
  Domain:
    <dm#PropertyOf>

Individual: <dm#User>
  Types:
    owl:Thing
  Facts:
    <dm#is> <dm#Actor>,
    <dm#has> <dm#Username>

Individual: <dm#Cart>
  Types:
    owl:Thing
  Facts:
    <dm#contains> <dm#Product>

```

Listing 4.1: Excerpt of OWL representing the domain model in Figure 4.2.

4.3 Use Case Specification

uCat provides a plugin (the *use cases specification* plugin) which supports the input of both use case diagrams and corresponding scenarios. For creating the diagram, the plugin supports the specification of actors, use cases and relationships. It is possible to add several actors, and for each actor several use cases. Figure 4.4 on the left shows the available elements to create the diagrams, specifically **Actors**, **Use cases**, **Systems** and **Associations**. On the right is presented an example of an use case diagram.

The specification of the use case scenario for each use case, is done in a user input/system response fashion (c.f. Fowler’s approach [38]), in a tabular format. In the scenario the user inputs and the system responses are specified, as shown in Figure 4.5. The figure depicts the main (or success) scenario for that use case. The plugin supports also the specification of alternative and exception

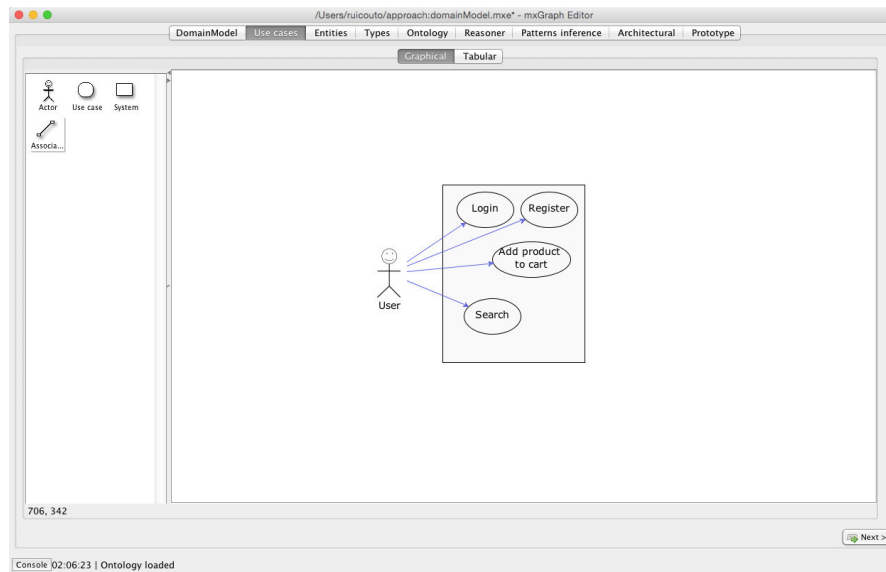


Figure 4.4: Use case diagram specification interface.

scenarios. Figure 4.5 presents an example of an alternative in step 3, and an exception in step 6. Lines where alternatives occur have a blue background, while lines where exceptions occur have a red background. The plugin performs runtime validation of the statements, against RUST. In Figure 4.5 it is possible to see an invalid statement on line 9, which is shown in red font.

N	User input	System response
1	user selects a product	
2		system shows the product
3	user selects the amount, color and size	
4	user selects add	
5		system reads the amount, color and size
6		system adds product
7		system updates the cart
8		system shows success
9	user ends the process itself	

Figure 4.5: Use case scenario specification interface.

4.3.1 Entities Extraction

After specifying the use cases, the subject, predicate and objects present in that specifications can be automatically extracted. That information corresponds to three sets of information, which are useful for users to validate the use cases. uCat contains the *entities visualization* plugin, which provides the possibility to visualize the information extracted from the domain model and the use case specifications.

The purpose of this plugin is twofold. On the one hand, it provides a preliminary verification step. The users have the possibility to see which information was extracted, and if there is (for instance) missing information. On the other hand, the plugin supports the verification of the consistency of the extracted information, against the domain model. In the plugin user interface the subjects, predicates and objects extracted from the specifications are shown, and terms that are not part of the domain model highlight in red. An highlighted word will be an hint for a missing term on the domain model or a misspelled word.

Figure 4.6 shows the information extracted from the use cases (specified in Figure 4.4). It is possible to see, for instance, that the words **session**, **amount**, **login** and **add** are highlighted. The first two words, **session** and **amount**, correspond to terms missing in the domain model, so the user should update it accordingly. Users should analyze the extracted information (i.e. these, and remaining words) in order to update the domain model or use case specifications as required.

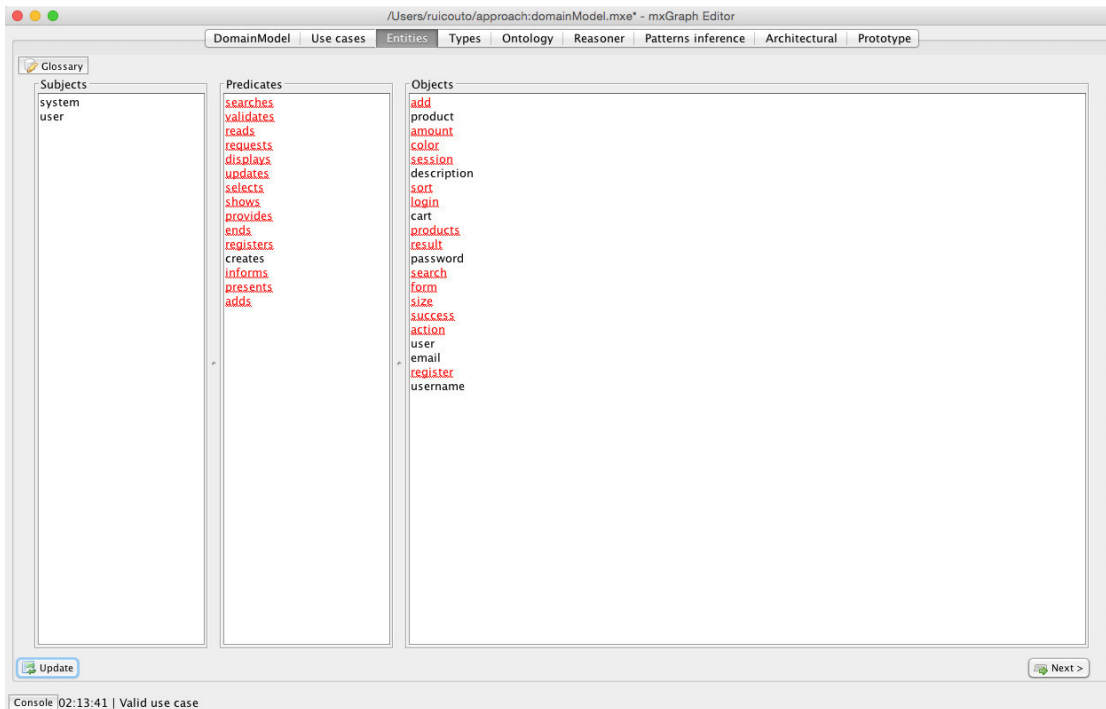


Figure 4.6: Entities visualization plugin.

The *entities visualization* plugin neither generates nor manipulates the internal data. Its sole purpose is to provide a simple overview of the extracted data to support a preliminary validation step. Indeed, no input is required by the user in this step to proceed with the process.

4.4 Ontology Creation

Creating the ontology that represents the requirements information is possible by automatically analyzing the RUS specifications. It is, however, necessary to have the specification of the types for the extracted entities. Two sources provide the types information, namely the domain model, and manual user input.

4.4.1 Types Definition

The automatic types information extraction is supported by the query presented in Listing 4.2. The query extracts all the types for all the entities in the domain model. Specifically, all the subject and objects, related via an association of the *TypeOf* category are requested. This results in the list of all instances and associated known types (see Appendix A.3 for the remaining queries).

```
PREFIX : <http://www.rmsc.com#>
SELECT ?subject ?type
WHERE { ?subject ?predicate ?type .
        FILTER(?predicate rdf:type :TypeOf)
}
```

Listing 4.2: SPARQL query to identify types for individuals.

The previously presented query can be further specialized in order to return only the types for a specific entity. Thus, in Listing 4.3 the query has been tailored by changing the `?subject` variable to `User`. The result of the query will provide the known types for `User`. The presented query does not address transitivity of properties. Consequently, if an entity `User` is known to have the type of another entity `Actor`, and `Actor` is known to have another type, e.g. `Object`, it is not assumed that `User` has the type `Object`. However, it is worth noting that SPARQL has enough expressiveness to handle such an extension to support transitivity analysis.

```
PREFIX : <http://www.rmsc.com#>
SELECT ?type
WHERE { :User ?predicate ?type .
        FILTER(?predicate rdf:type :TypeOf)
}
```

Listing 4.3: SPARQL query to identify the types for `User`.

Depending on the level of detail of the domain model, it might not be possible to infer all the types information. In that case, users will have to manually specify the missing types. uCat contains

a plugin which simultaneously supports the analysis of the extracted types, and specification of new ones. The *types definition* plugin is depicted in Figure 4.7. On the left the individuals, and corresponding inferred types are show. On the right, the types used to specify those individuals are presented.

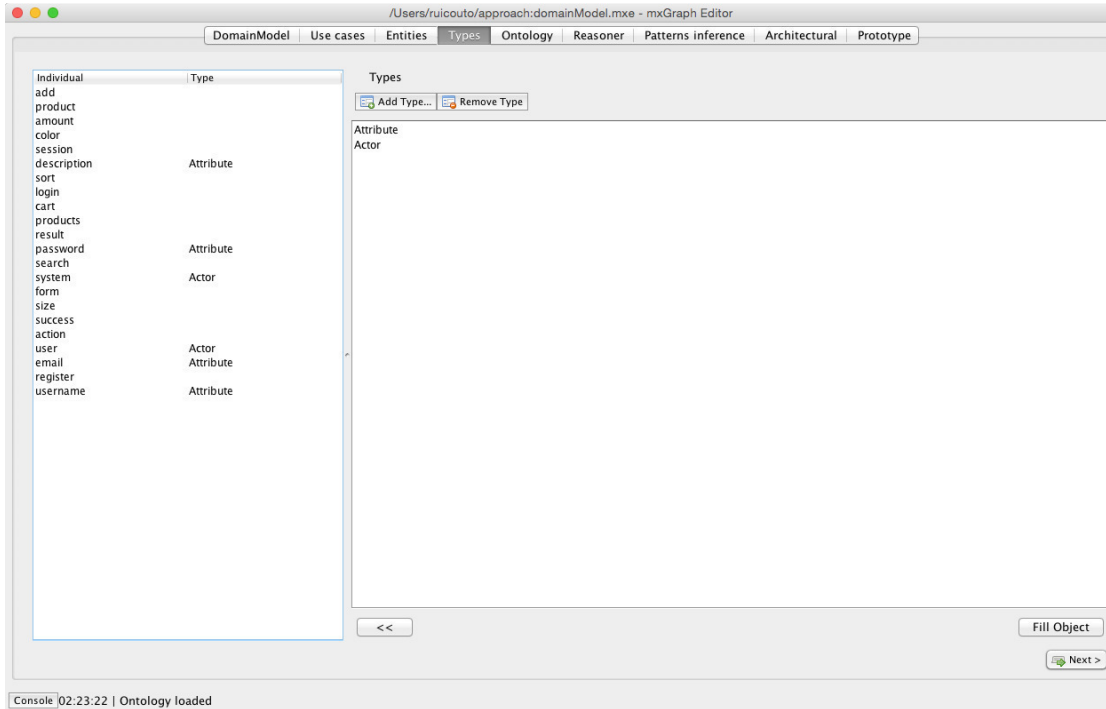


Figure 4.7: Types definition plugin, with the automatically extracted information.

As it is possible to see in Figure 4.7, there are some missing types. Manually specifying the types corresponds to the second step of the types definition process. In this step the user has also the possibility to check if the types' association is correct, and adjust it if necessary. Here, there are two options available. The first one requires the user to manually specify each missing type. The second one disregards the individuals' types, considering that all have a generic type. SCARP supports both kinds of information for the process of generating architectural prototypes. By following the manual types definition approach, the types **Action**, **Collection**, **Object**, **Data** and **Property** were defined and associated with the respective individuals. Figure 4.8 presents the complete types table.

After the definition and attribution of the types to the individuals, the plugin updates the internal representation of the use cases data. This adds more information, that is essential in order to proceed with the SCARP process.

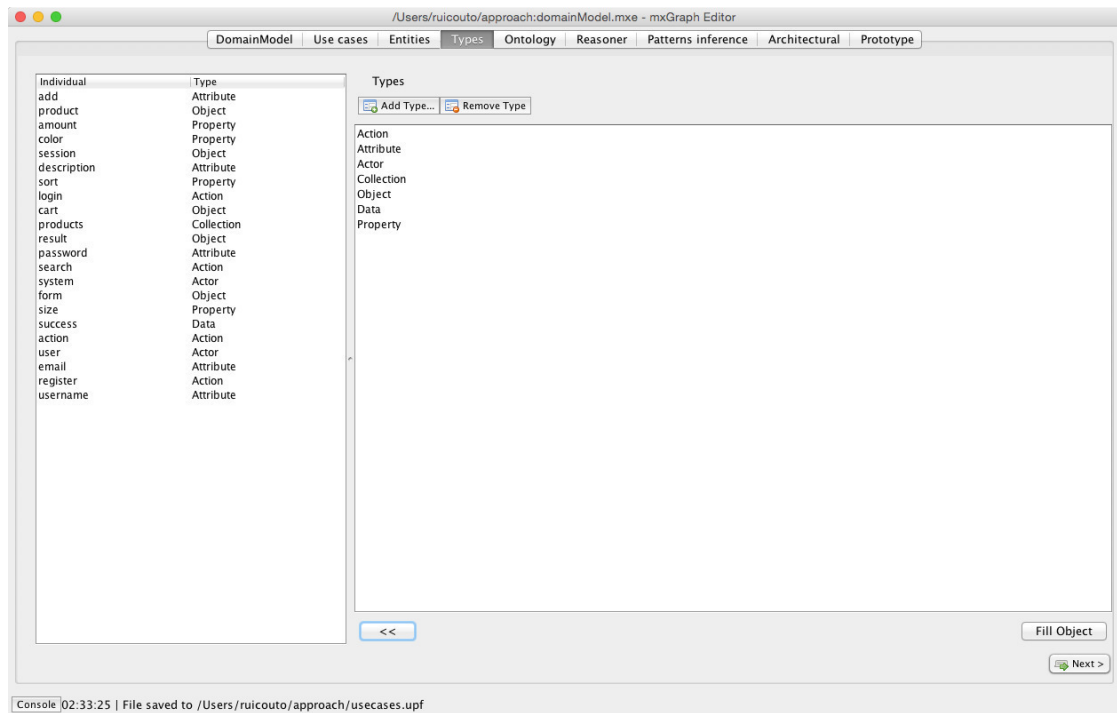


Figure 4.8: Types definition plugin with all types defined.

4.4.2 OWL Ontology Generation

Combining the declared instances, object properties and types information supports the creation of ontologies, as is the case with the one presented in Listing 4.4 (see Appendix A.4 for the full specification). At this point the ontology is ready to be used, in the case of SCARP, ready to be queried. The ontology can also be exported to be used in other tools (e.g. Protégé).

The achieved ontology contains the RUS representation of the *Add product to cart* use case (as presented in Figure 4.5), enhanced with the types information. Combining this information with OWL's query capabilities establishes the basis to apply inference techniques.

The *ontology generation* uCat plugin was implemented, to support parametrization and generation of OWL ontologies, by combining the previously extracted information. The users have the possibility to define both the name and the Uniform Resource Locator (URL) to identify the elements in the ontology. Figure 4.9 shows the user interface for this plugin.

After selecting the option to create the ontology, the tool automatically creates and shows a graphical representation of the ontology. However, the representation is only useful when considering small use case examples. When considering several use cases, the diagrams tend to become too complex. In any case, the main objective is to process it automatically. The tool provides also the possibility to see the source code of the generated ontology, in RDF/XML. Generating

the ontology is required in order to proceed with SCARP.

```
Prefix: j.0: <http://www.url.com/Requirements/>
Ontology: <http://www.url.com/Requirements>
AnnotationProperty: rdfs:comment
Datatype: rdf:PlainLiteral
ObjectProperty: j.0:selects
  Annotations:
    rdfs:comment "1"
ObjectProperty: j.0:shows
  Annotations:
    rdfs:comment "2"
Class: j.0:Object
Class: j.0:Actor
Individual: j.0:product
  Types:
    j.0:Object
Individual: j.0:system
  Types:
    j.0:Actor
Facts:
  j.0:shows j.0:product
Individual: j.0:user
  Types:
    j.0:Actor
Facts:
  j.0:selects j.0:product
(...)
```

Listing 4.4: Excerpt of “*Add product to cart*” use case formalized in OWL

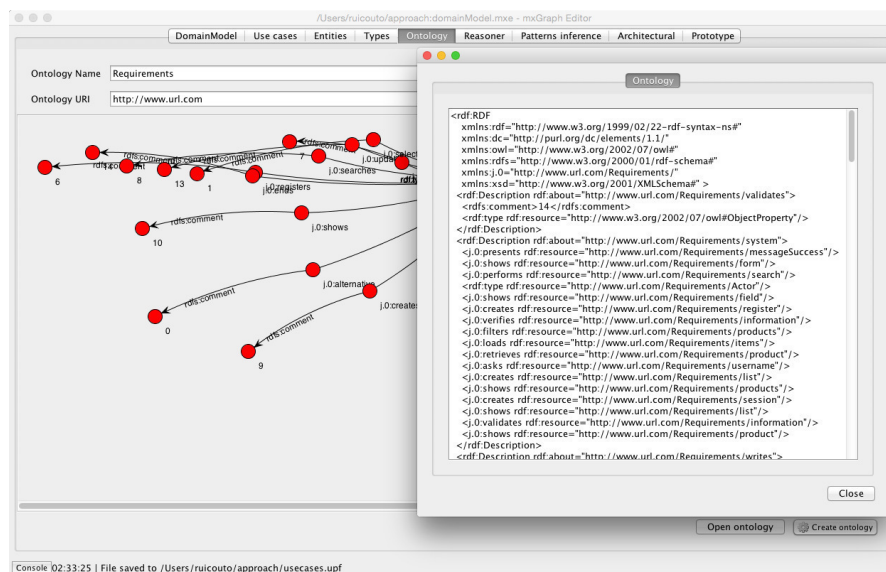


Figure 4.9: Ontology generator plugin.

4.5 Requirement Pattern inference

Two requirements exist in order to perform the pattern inference process. First, the existence of a query mechanism which supports the analysis of the ontology. Second, a set of queries, representing the requirement patterns to be inferred.

4.5.1 Data Query Mechanism

The first requirement for pattern inference is supported by SPARQL and its corresponding query mechanisms. In uCat, the *reasoner* plugin provides a functionality for users to query the generated OWL ontology. In this plugin, the user has the possibility to write, apply and see the result of SPARQL queries. Figure 4.10 presents the plugin. On the upper part it is possible to write the SPARQL queries in textual format. On the bottom part the query result is presented, both in textual and tabular format.

This plugin is intended to enable users to analyze the ontology generated from the specified requirements, through a query language. Similarly to the entities visualization plugin, this plugin neither changes nor adds information to the internal representation.

4.5.2 Pattern Specification

The second requirement to support the pattern inference process is the specification of requirement patterns. Defining the patterns is achieved by analyzing software specifications, similarly

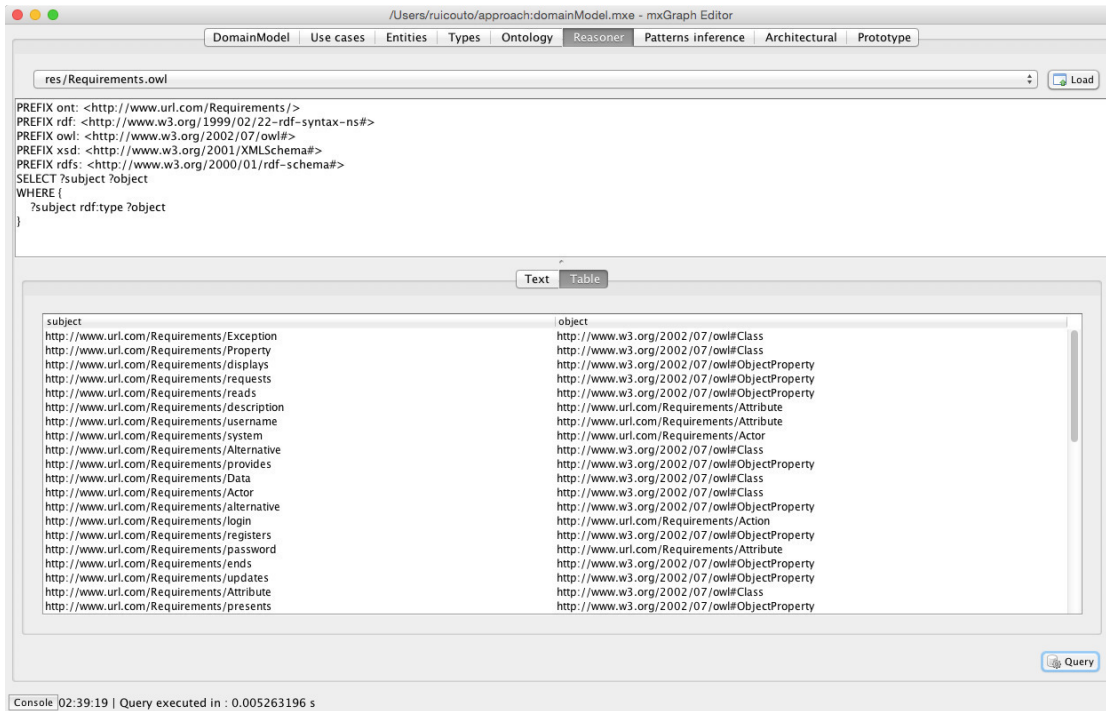


Figure 4.10: Reasoner plugin depicting a SPARQL query to infer `object` types, and corresponding result.

to the definition of other kind of patterns. By analyzing requirement specifications describing the same requirement, it is possible to identify their similarities, achieving then a requirement pattern [120]. Textual requirements defined by simple and clear statements are the best candidates for creating the corresponding requirement patterns. In Section 4.3 a description of a requirement which describes a feature of support a shopping cart, was presented. A requirement pattern, namely the *HasShoppingCart*, can be extracted from the description.

When such a kind of specification (for a given feature) is found to be recurrent it is then possible to consider it as a description of a requirement pattern. In this case it is possible to extract that the pattern *HasShoppingCart* (c.f. *Feature*) is composed of several steps. One of the steps, is for instance a *user* which *selects* a *product*. Another step is performed by the *system* which reads the *amount*, *color* and *size*, previously defined by the user.

4.5.3 uQL

The automatic and systematic transformation of uQL into SPARQL is possible since SPARQL also follows a triple format, similar to RDF. The process consists in creating a SPARQL query for each uQL pattern. Each uQL statement corresponds to a SPARQL condition. A set of rules is used in order to perform the transformation, as follows.

- Each statement results in an optional SPARQL triple which corresponds to the condition to exist, i.e.
`OPTIONAL { ?subject ?predicate ?object }`
- Each *or* (c.f. “|”) condition is converted into a set of SPARQL *or* (c.f. “||”), allowing a variable to have several values, i.e.
`?var = :v1 || var = :v2 ...`
- Each condition group results in a `FILTER`, which demands the condition to have those properties, i.e.
`FILTER (?subj = :s1 && ?pred = :p1 || ?pred = :p2 && ?obj = :o1 ...)`
- From each condition is extracted the comment of the predicate, which supports understanding if the condition is met in the knowledge base, i.e.
`?predicate rdfs:comment ?condition1`

An example of an uQL query is presented in Listing 4.5. Each element of a uQL triple represented by “()” corresponds to, respectively, predicate, subject and object. As described in Section 3.6, the keywords define the name of the predicates, subjects and objects (e.g. `user` is a subject). The `|` denotes the logical *or*, denoting that it can be one of the alternatives (e.g. `provides|inserts` means either `provide` or `inserts` will match). The number in front of the statement (c.f. 10) denotes the weight, in percentage of that statement for the requirement pattern. Finally, adding variables to the queries is also possible, by preceding a keyword with a `?` before a name (e.g. `?user`), or anonymous variables (c.f. `(?)`). A variable represents the same entity across several conditions.

```
(user) (selects) (?product) 10
(system) (shows) (?product) 10
(user) (provides|inserts) (?) 15
...
```

Listing 4.5: Excerpt of an uQL query representing the *HasShoppingCart* requirement pattern.

Following the aforementioned process for the uQL query presented in Listing 4.5 results in the SPARQL query presented in Listing 4.6.

```

PREFIX : <http://www.url.com/Requirements/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?condition1 ?condition2 ?condition3
WHERE {
  OPTIONAL { ?subject ?predicate ?object
    . FILTER ( ( ?subject = :user )
      && ( ?predicate = :selects ) )
    . ?predicate rdfs:comment ?condition1 } .

  OPTIONAL { ?subject2 ?predicate2 ?object2
    . FILTER ( ( ?subject2 = :system )
      && ( ?predicate2 = ont:shows ) )
    . ?predicate2 rdfs:comment ?condition2 } .

  OPTIONAL { ?subject3 ?predicate3 ?object3
    . FILTER ( ( ?subject3 = :user )
      && ( ?predicate3 = :provides ||
        ?predicate3 = :inserts ) )
    . ?predicate3 rdfs:comment ?condition3 }
}

```

Listing 4.6: Result of the translation of the uQL in Listing 4.5 query to SPARQL.

In this example, the presented requirement pattern was derived from its description. However, and according to the pattern's philosophy, they are typically extracted from existing documents. uQL patterns can be automatically extracted from requirement descriptions. If a large number of descriptions known to contain a given pattern are available, it is possible to derive the pattern from them. An example of the algorithm to do that is presented in pseudo-code, in Listing 4.7.

```

//uniform statements
1. for all specifications
  1.1. for all statements
    1.1.1 reduce to triple format (c.f. <S,P,O>)
//create the uQL statement
2. Group statements representing the same step
3. for each predicate on the same group
  3.1. ns = number of synonyms found in specifications
  3.2. ns2 = number of statements in the next step which refer to this predicate
  3.3. if ns > N or ns2 > V
    3.3.1. then use the subjects as alternatives (i.e. a|b|...|n)
    3.3.2. else use a variable to represent the subjects (e.g. ?v)
  3.4. repeat for object
//set weights
4. for all specifications
  4.1. for all statements
    4.1.1. ps = percentage of specifications that contain this statement
    4.1.2. statement.weight = ps

```

Listing 4.7: Proposal of algorithm to automatically generate uQL from a set of specifications.

There are several parameters that can be tuned, namely N or V (c.f. Listing 4.7, line 3.3.). N corresponds to a number of different terms found to describe the same concept. Hence, N corresponds to a low number of alternatives (e.g. 4). V corresponds to the number of statements that in the step $n + 1$ refer to the terms in step n . Hence, V should correspond to a large percentage of statements (e.g. 60%).

Defining the weight of the statements is a challenge itself, since finding a meaningful value is mandatory in order to both achieve a sound approach, and infer meaningful patterns (i.e. reduce the number of false positives). The process of generating automatic uQL queries was not further explored, and is left for future work.

4.5.4 Pattern Inference

The pattern inference process is supported by the *pattern inference* plugin. In this plugin it is possible to specify (or load) uQL queries, which will compose the pattern catalog. The pattern catalog can then be applied to the knowledge base, in order to infer the patterns therein. Figure 4.11 shows the plugin interface to specify the patterns. In the left part is presented the uQL specification interface, where the user can write the uQL queries, and also see the generated SPARQL. On the right is presented the list of patterns created so far, and also their corresponding SPARQL.

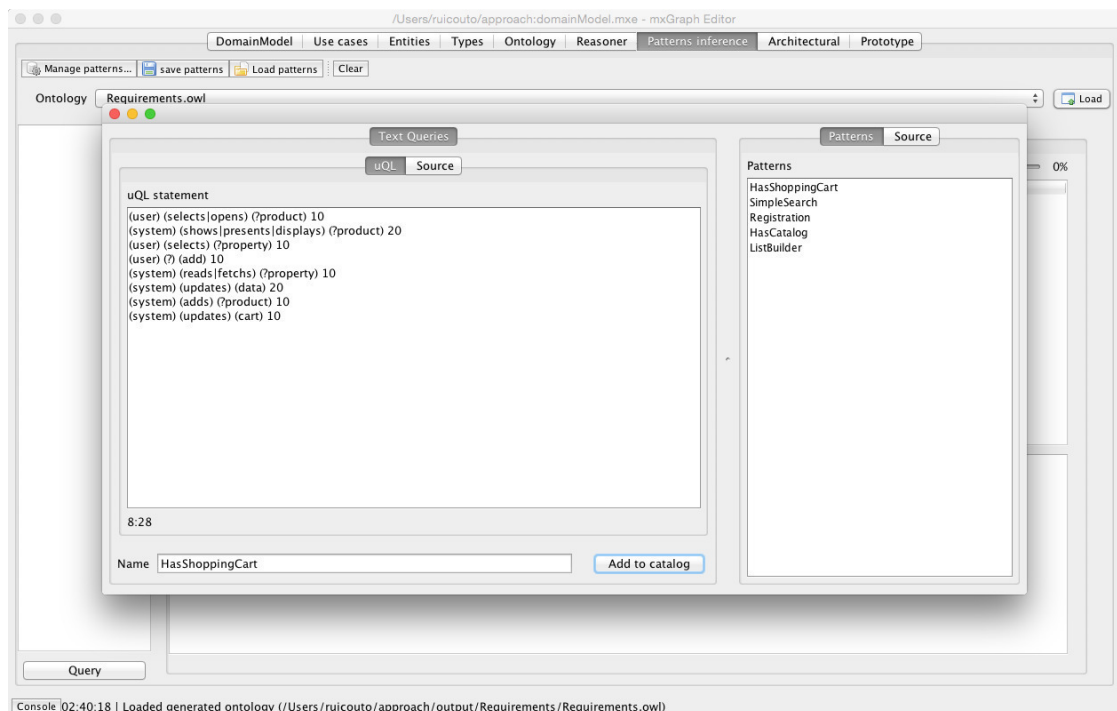


Figure 4.11: uQL requirement pattern specification user interface.

After specifying the queries, the plugin supports the process of automatically applying them to the knowledge base. As result, are shown not only the inferred patterns, but also their matching percentage. The plugin interface is shown in Figure 4.12. This interface presents on the left the list of available patterns to query. On the right it presents the result of the inference process, and for each pattern the matching percentage.

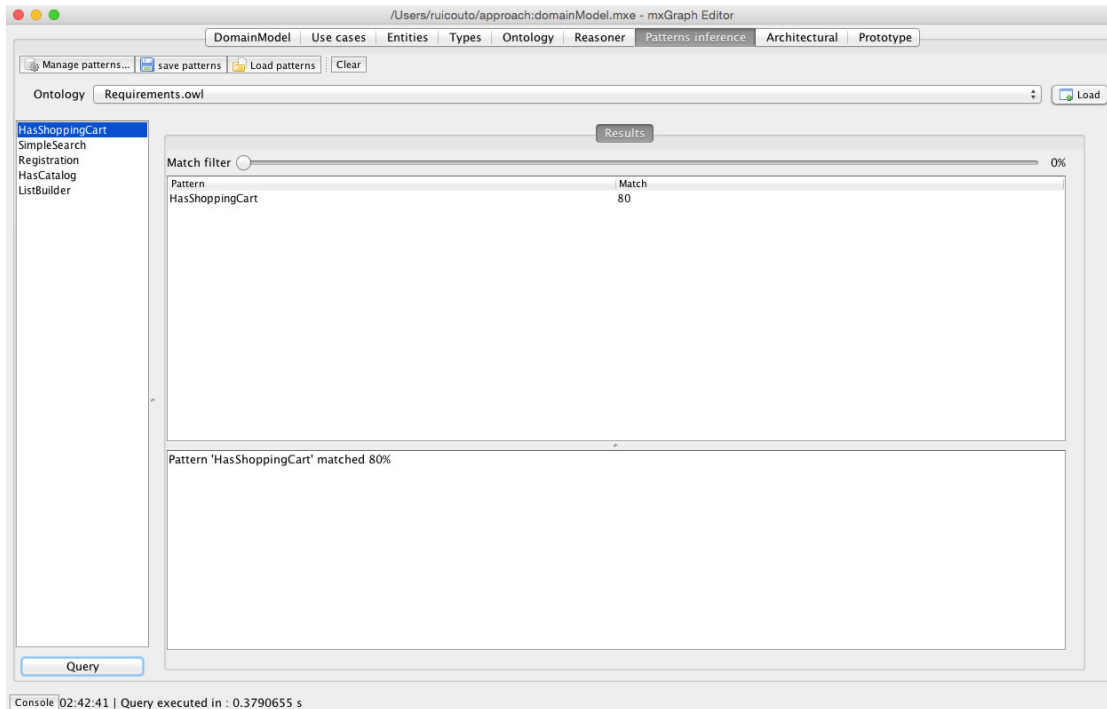


Figure 4.12: Requirement pattern inference plugin user interface.

4.6 Requirements to Software Patterns

The requirement to software pattern matching process is composed of two main parts. First, there is the identification of the requirement and software patterns properties (c.f. Figure 4.13 1) and 2)). Second, there is the analysis process (c.f. Figure 4.13 c)). The analysis process results in a set of candidate software patterns, selected based on the specified forces.

4.6.1 Concerns, Goals and Forces definition

The requirements to software patterns matching process assumes the definition of the requirement pattern concerns, software pattern goals and corresponding forces in beforehand (c.f. Figure 4.13 a) and d)).

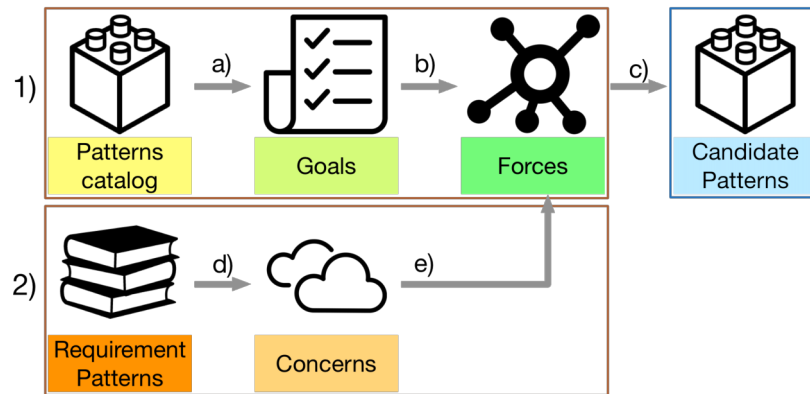


Figure 4.13: Requirement pattern to software pattern matching process.

An example of Requirement Pattern to Concern association is the requirement pattern *HasShoppingCart*, containing the *Manageable* concern, c.f. `HasShoppingCart => { Manageable }`. An example of Software Pattern to Goal, is the *Proxy* software pattern which contains the *Delegate* goal, c.f. `Proxy => { Delegate }`. Finally, an example of forces association, the *Manageable* concern has the *Separation* goal, c.f. `Manageable => { Separation }`. Similarly, the *Delegate* goal has also the *Separation* concern, c.f. `Delegate => { Separation }`. The forces association has been defined c.f. `HasShoppingCart => { Manageable => { Separation } }` and `Proxy => { Delegate => { Separation } }`. By having both patterns described at the forces level, the tool is able to perform the matching process, which will identify for a given requirement pattern and respective concern, which is the corresponding goal. Having identified the goal, the tool proposes the candidate software patterns, which satisfy the provided requirement patterns.

The tool uses the the matching information in order to perform the process. An ontology was defined which represents the *Requirement patterns*, with their associated *Concerns* and corresponding *Forces*. The ontology contains also the description of the *Software patterns*, with their corresponding *Goals* and *Forces*. An excerpt of the defined ontology is presented in Listing 4.8. For each concept there is a corresponding class (e.g. `RequirementPattern`, `Concern`). The relation between the different concepts is done via object properties (e.g. `hasGoal`, `hasConcern`). The forces can be defined as a positive or negative contribution (with `hasForceP` and `hasForceN`, respectively). In Listing 4.8 it is possible to see a simplified version of an excerpt of the ontology which supports the formalization of the information. In the listing it is possible to see the classes representing for instance `Goal`, `Concern` and `Force` (see Appending A.5 for the full specification). It is also possible to see the object properties which identify the relationship between those elements, as for instance `hasForceP` (for a force with a positive impact), `hasForceN` (for a force with negative impact) or `hasGoal`.

```

Ontology: <http://www.url.com/mapping>
AnnotationProperty: rdfs:comment

Datatype: xsd:string

ObjectProperty: :hasForceP
ObjectProperty: :hasForceN
ObjectProperty: :hasGoal
ObjectProperty: :hasConcern

Class: :RequirementPattern
Class: :Concern
Class: :Force
Class: :Goal
Class: :SoftwarePattern

```

Listing 4.8: Excerpt of the ontology supporting the matching process.

4.6.2 Matching Process

The plugin provides a graphical interface to help building the ontology containing the mapping information, as part of the application setup. Figure 4.14 presents the interface provided. It is possible to see on the left the input fields to specify the requirement patterns, and for each requirement pattern the set of concerns. It is also possible to specify for each concern the corresponding forces and their impact on the concern (i.e. positive or negative). Similarly, on the right are presented the input fields to specify the software patterns, and for each software pattern its set of goals. Finally, for each goal the set of forces and their nature. After specifying this information, in the plugin is generated an ontology (as the one presented in Listing 4.8), which formalizes the specified information.

In Figure 4.14 it is possible to see that the *HasShoppingCart* requirement pattern contains the *Editable*, *Manageable* and *Processable* concerns. For instance the *Editable* concern, contains the *Versioning* and *Flexibility* positive forces, and *Constraint* negative force. Regarding the software patterns, we have the *Command*, *Flyweight*, *Memento* and *Proxy* software patterns. For instance the *Proxy* software pattern has the *Delegate* concern. The *Delegate* concern contains the *Decoupling*, *Separation* and *Abstraction* positive forces, and, *Overkill* and *Efficiency* negative forces. This information is then formalized into the corresponding ontology, and ready to support the mapping process.

The software pattern inference process, for a given set of requirement patterns (c.f. **Perform matching**, in Figure 3.15) is a process composed of two main parts. The process results in the candidate software patterns, as presented on Figure 4.15.

In Figure 4.15 Part 1, starts by querying the knowledge base in order to identify the **concerns** for a given **requirement pattern**. Having the concerns, it is possible to extract the associated **Forces**. Considering the *HasShoppingCart* pattern, the format of the expected result is as follows.

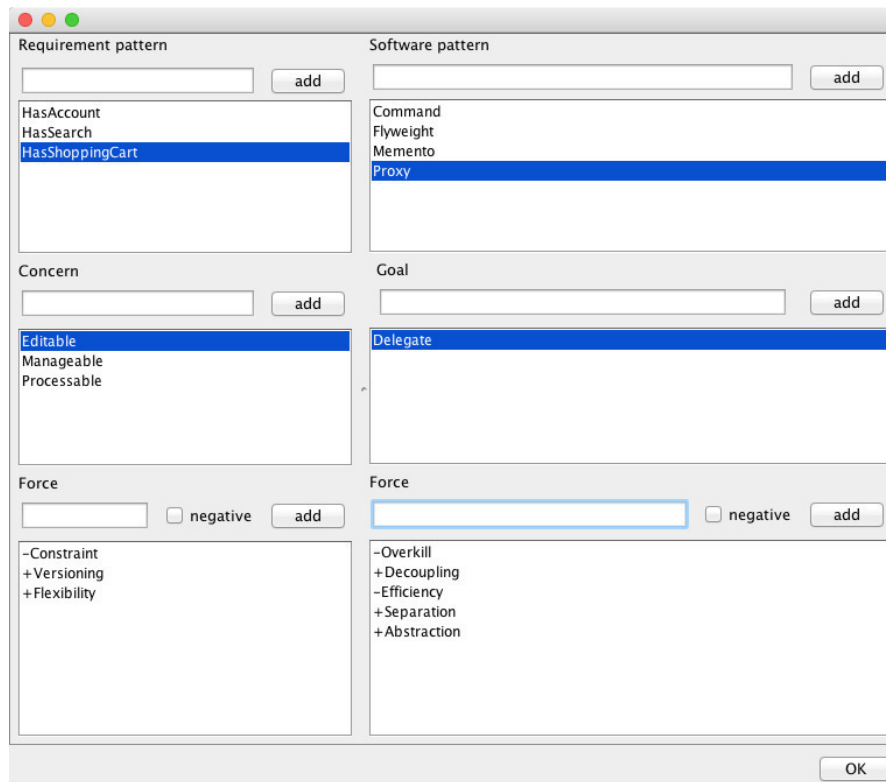


Figure 4.14: Interface to create the Requirement Pattern, Software Pattern, Concern, Goal and Forces mapping information.

```
{ HasShoppingCart => { Manageable => { +Separation, ... }, ... }, ... }
```

Part 2 of the matching process, consists in reaching the suitable software patterns, given a set of forces. Hence, for the given forces, the suitable goals are inferred, which have matching forces. Finally, from the goals is possible to achieve the software patterns. The format of the expected result would be as follows.

```
{ +Separation => { Delegate => { Proxy, ... },
  -Separation => { Abstraction => { Bridge, ... }, ... }
```

Part 2 of the matching process results in several goals that match the inferred forces. Also, a goal can be related with several software patterns. An example would be as follows.

```
Separation => { Delegate, Abstraction }
and
Delegate => { Proxy }, Abstraction => { Bridge }
```

Hence, a methodology is needed to select the most appropriate pattern to satisfy a given goal. A matrix was used in order to identify the best match for a given set of patterns (c.f. [4]). The

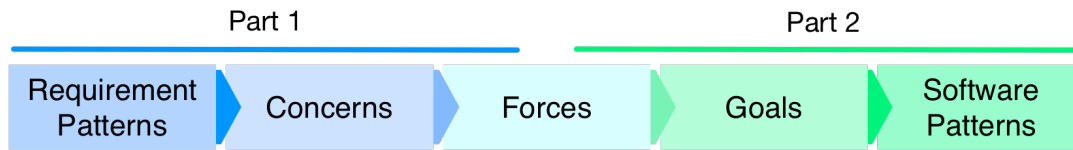


Figure 4.15: Requirement pattern to software pattern process flow.

same approach was used for matching requirement and software patterns. The forces matrix for the presented example is as depicted in Table 4.1. In the matrix, the positive and negative relations between forces are represented, respectively, by positive (+P) and negative (-N) values.

```

PREFIX : <http://www.url.com/mapping#>
SELECT ?rp ?c ?f ?g ?sp
WHERE {
  ?rp :hasConcern ?c .
  ?c :hasForceP ?f .
  ?g :hasForceP ?f .
  ?sp :hasGoal ?g
}

PREFIX : <http://www.url.com/mapping#>
SELECT ?rp ?c ?f ?g ?sp
WHERE {
  ?rp :hasConcern ?c .
  ?c :hasForceN ?f .
  ?g :hasForceP ?f .
  ?sp :hasGoal ?g
}

```

Listing 4.9: Queries to identify positive and negative relations between requirement and software patterns.

The queries in Listing 4.9 support the extraction of the relation between requirement patterns (?rp), concerns (?c), forces (?f), goals (?g) and software patterns (?sp). This is the information used to create the forces matrix. Specifically, the first query supports extracting the positive contributions between goals and concerns, and the second query supports extracting negative contributions.

The Forces Matrix summarizes the impact of adopting requirement patterns, in order to address a software pattern (and vice-versa). The matrix is automatically extracted from the formalized information, due to OWL's reasoners, by applying the aforementioned queries in Listing 4.9.

uCat's *Architectural Matching* plugin, as presented on Figure 4.16, was implemented to perform the pattern matching process. The interface handles the requests to perform the matching process, and to display its resulting information. In order to request the execution of the matching process, the user should select the *Match* option (depicted on Figure 4.16, on the top). Once

Table 4.1: Example of forces matrix relating goals to concerns.

		Goals		
		Delegate	Abstraction	Goal n
Concerns	Manageable	(+2, -0)	(+0, -1)	...
	Editable	(+2, -1)	(+1, -0)	...
	Concern n	(+P, -N)	(+P, -N)	...

requested, the plugin resorts to the ontology containing the mapping information, and performs the matching process. From the matching process results the forces matrix, which summarizes the information regarding the Concerns/Goals contributions.

Figure 4.16 depicts the results of the matching process, for the *HasShoppingCart*, *HasAccount* and *HasSearch* requirement patterns. The plugin's user interface is divided in four major vertical sections. In the first section (from left to right), are presented both the requirement patterns to handle (inferred in the previous plugin), as well as the mapping information provided so far. The second section (*Forces matrix*), presents the resulting forces matrix, which shows for each concern, the impact on each goal. The third section (*Matching result*), presents the automatic inference result, as the most appropriate goal to answer each concern, and the corresponding software pattern. Finally, the fourth section (*Software Patterns*) lists the software patterns to be instantiated, required to handle the inferred requirement patterns.

From the provided mapping information (c.f. Figure 4.14), the plugin was able to generate the forces matrix. In the matrix, it is possible to see the concerns *Editable*, *Processable*, *Manageable* and *Manageable*, as well as their relations with the goals *Delegate*, *Edit*, *Share* and *Process*. The mapping information supports also the inference of the values of the impact of each concern on goals. For instance, the concern *Manageable* has a positive impact of 2 on the *Delegate* goal. It is also possible to see that +2 is the positive contribution with the higher value for this goal, therefore that goal is the most appropriate one. Similarly, for the *Editable* concern, the *Edit* goal was inferred to be the most appropriate one, with a value of 2, for *Processable* the *Process* goal was inferred with a value of 2, and for the *Manageable*, *Delegate* was inferred with a value of 3. Hence, and in respect to the forces matrix, the most appropriate software pattern to handle the *Manageable* concern is the *Proxy*, for the *Editable* concern is the *Memento*, for the *Processable* concern is the *Command*, and finally for the *Manageable* concern is the *Proxy*.

This resulting list of patterns, plus the association to the requirement patterns they handle, are the outputs produced by this plugin. At this point, all the information required in order to start the pattern instantiation process is gathered. That step is supported by the *Prototype* plugin, described in Section 4.7. The user has then the possibility to adjust the suggested solution, and proceed to the architectural step.

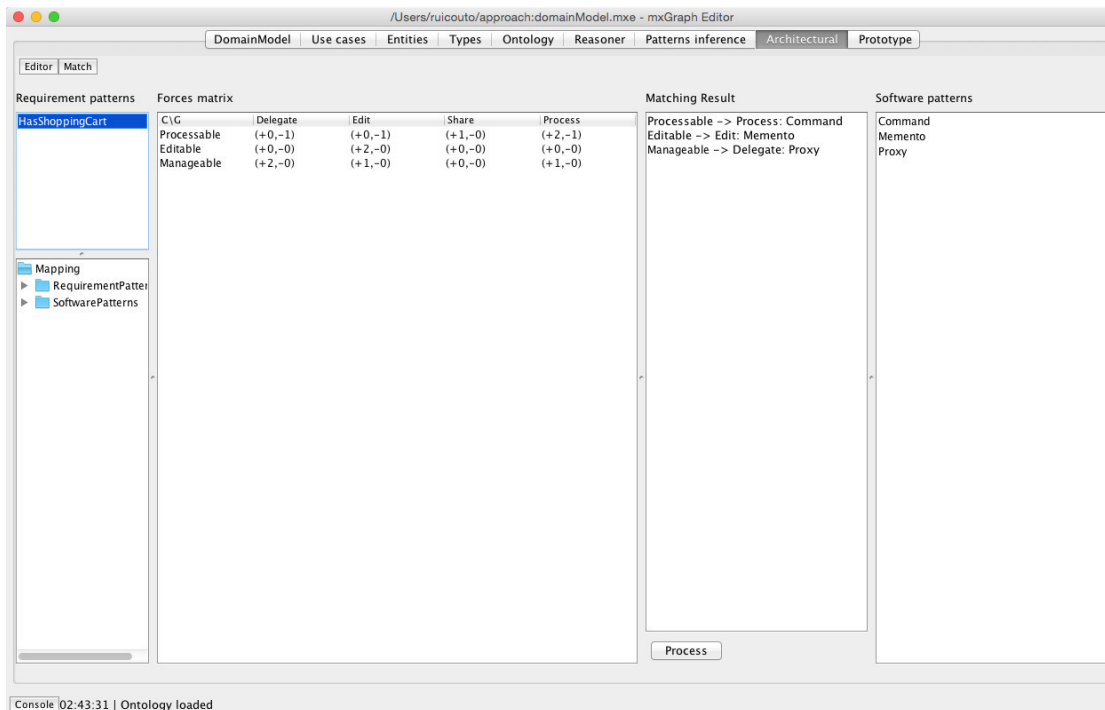


Figure 4.16: Architectural mapping plugin user interface.

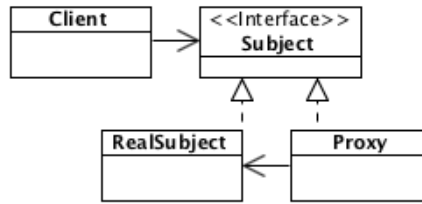
4.7 Software Pattern Instantiation

The software pattern instantiation process consists in creating and composing concrete software pattern instances. Taking the inferred patterns from the previous step, candidate solutions are created.

As described in Section 3.8, in order to support this instantiation process, four major requirements must be fulfilled. First, it is required to represent software patterns in an appropriate way to support their instantiation. Second, there is the need to describe requirement patterns in a format that can be automatically managed. Third, combining the two previous requirements, it is needed a process to create concrete pattern instances. Finally, the fourth requirement is to provide a composition technique able to compose the pattern instances.

4.7.1 Software Pattern Representation

Software patterns can be represented in XMI. Such eases their automated manipulation and instantiation, according to the first requirement. Since software patterns are typically represented as UML class diagrams, XMI is the logical selection. As an example, the Proxy pattern depicted in Figure 4.17, is represented in XMI in Listing 4.10 (see Appendix A.6 for the full list of used patterns).

Figure 4.17: *Proxy* software pattern structure (adapted from [41]).

In the presented listing it is possible to see the representation of the structure elements, such as the class `Proxy`, or the interface `Subject`, as well as the relation (**Abstraction**) between them. Having this representation, it is now possible to automate the instantiation process, by replacing the structure elements with concrete values.

```

<UML:Class xmi.id = 'ID1' name = 'Proxy' visibility = 'public'
  isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
  isAbstract = 'false' isActive = 'false'>
  <UML:ModelElement.clientDependency>
    <UML:Abstraction xmi.idref = 'ID2' />
  </UML:ModelElement.clientDependency>
</UML:Class>

<UML:Interface xmi.id = 'ID3' name = 'Subject' visibility = 'public'
  isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
  isAbstract = 'false' />

<UML:Abstraction xmi.id = 'ID2'
  isSpecification = 'false'>
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = 'ID4' />
  </UML:ModelElement.stereotype>
  <UML:Dependency.client>
    <UML:Class xmi.idref = 'ID1' />
  </UML:Dependency.client>
  <UML:Dependency.supplier>
    <UML:Interface xmi.idref = 'ID3' />
  </UML:Dependency.supplier>
</UML:Abstraction>
  
```

Listing 4.10: Excerpt of the representation of the *Proxy* software pattern in XML.

4.7.2 Software Pattern Definition

The second requirement to support the instantiation process is the definition of software patterns. In this specific context, specifying the software patterns consists in abstracting their representation into a simplified and computable format, with the objective of easing their usage.

The selected approach was to represent software patterns as an API-like format (see Section 3.8.1). In Listing 3.3, Listing 4.11 and Listing 4.12 the descriptions, respectively, of the *Proxy Memento* and *Command* software patterns are presented. Both XMI and the API-like format are required, since the former supports the manipulation of the patterns, and the latter their specification.

```
/**
 * @intent Without violating encapsulation, capture and externalize an object's
 *         internal state so that the object can be restored to this state later.
 * @param originator Who triggers the request for creating states.
 * @param memento    Stores internal state of the Originator object.
 * @param caretaker  The responsible for the memento's safekeeping.
 * @param state      The representation of the state being keep.
 * @param item       The items belonging to the state.
 */
memento(originator, memento, caretaker, state, item);
```

Listing 4.11: Representation of the *Memento* software pattern in API-like format.

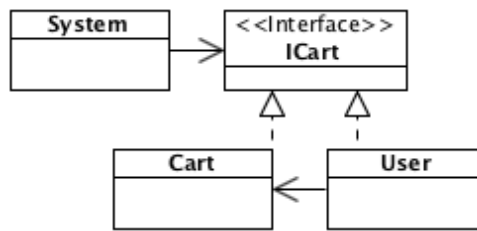
```
/**
 * @intent Encapsulate a request as an object, thereby letting you parameterize
 *         clients with different requests, queue or log requests, and support
 *         undoable operations.
 * @param client Who triggers the operation.
 * @param receiver Knows how to perform the operations associated with carrying out
 * @param invoker Asks the command to carry out the request.
 * @param command Declares an interface for executing an operation.
 * @param concreteCommand Implements Execute by invoking the corresponding operation
 *         on Receiver.
 */
command(client, receiver, invoker, command, concreteCommand);
```

Listing 4.12: Representation of the *Command* software pattern in API-like format.

4.7.3 Instantiation Process

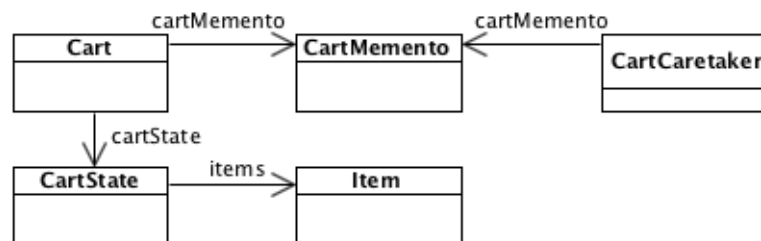
In the previous step the pattern definition process was presented. The result of the inference process listed the *Proxy*, *Memento* and *Command* software patterns in order to support the *Has-ShoppingCart* requirement pattern. Hence, considering the eCommerce domain, and resorting to the API-like pattern definition, the parameters for the *Proxy* pattern are as follows, resulting in the architecture depicted in Figure 4.18, which supports the third requirement.

- **client:** *System* - The system triggers changes in the cart;
- **subject:** *ICart* - The interface which defines the cart actions;
- **realSubject:** *Cart* - The implementation of the cart;
- **proxy:** *User* - The user handles the instance of the cart.

Figure 4.18: *Proxy* software pattern instantiation.

For the *Memento* proxy, the parameters are as described next, and the instance depicted in Figure 4.19.

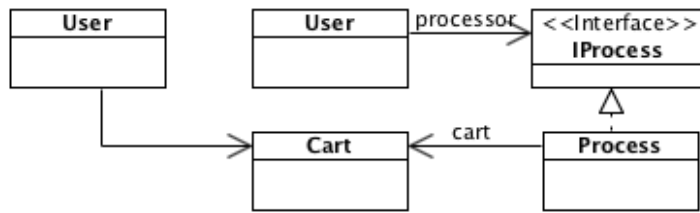
- originator: `Cart` - The cart is who needs to keep the states;
- memento: `CartMemento` - The memento for the cart;
- caretaker: `CartCaretaker` - The caretaker for the cart state;
- state: `CartState` - The state for the cart;
- item: `Product` - The products of the application.

Figure 4.19: *Memento* software pattern instantiation.

The definition of the parameters for the *Command* pattern is described next, and the instance depicted in Figure 4.20.

- client: `User` - The user will trigger and requests the actions (e.g. checkout);
- receiver: `Cart` - The cart is who knows what action to perform;
- invoker: `User` - The user is who asks the process to be performed;
- command: `IProcess` - The interface to define the class to process;
- concreteCommand: `Process` - The class which implements the process code.

uCat is able to interpret the aforementioned software pattern representations and definitions in order to support SCARP. The instantiation process consists in requesting for each pattern component, a specific name. The plugin provides then the interfaces required to perform such,

Figure 4.20: *Command* software pattern instantiation.

by showing for each required software pattern a) its context (i.e., the requirement pattern in which it will be integrated), and its content, b) the list of its constituents, i.e., names of the classes, and corresponding description, and finally c) the user interface input fields to specify the names of the constituents. The interface to support this process is shown in Figure 4.21.

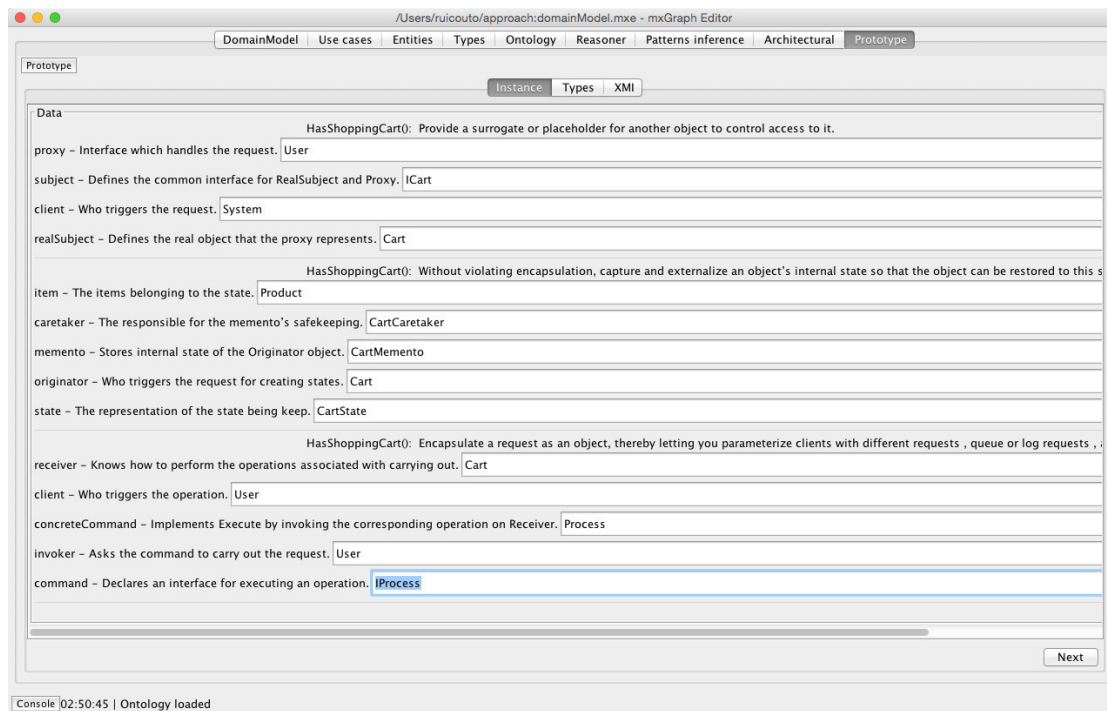


Figure 4.21: Software pattern instantiation user interface.

In Figure 4.21 it is possible to see that for the first software pattern to support the *HasShoppingCart* requirement pattern (i.e. *Proxy*), the names for the *Proxy*, *Subject*, *Client* and *RealSubject* constituents are requested. The corresponding names provided for those constituents are *User*, *ICart*, *System* and *Cart*. Similarly, for the remaining fields were given meaningful names in the eCommerce domain.

After defining the names, when the user selects the *Next* option (Figure 4.21 on top), the plugin creates the internal representation, and extract the fields contained in each constituent.

4.7.4 Composition

The fourth and final requirement of the instantiation process is the composition of the achieved instances. uCat resorts to *overlapping* to combine entities represented by the same name. In the *Proxy*, *Memento* and *Command* patterns, the *User* class represents the same concept in all patterns, as the user of the eCommerce application. The same is true for the *Cart* class. Thus, classes with the same name can be merged into a single one, composing both patterns into a single architecture. The overlapping technique occurs at the XMI level, by merging the information of both XMI files into a single one. As result, is produced the representation of *User* represented in Listing 4.13. In the XMI is possible to see that it was added a connection from *User* to *Cart*, and that *Cart* has two connections, one to *CartState*, other to *CartMemento*. The *Cart* class represents also the merged information.

```

<!-- Class User -->
<UML:Class xmi.id = 'id-user' name = 'User' ... >
  <UML:ModelElement.clientDependency>
    <UML:Abstraction xmi.idref = 'id-abstraction-user' />
  </UML:ModelElement.clientDependency>
</UML:Class>
<!-- Class Cart -->
<UML:Class xmi.id = 'id-cart' ...>
  <UML:ModelElement.clientDependency>
    <UML:Abstraction xmi.idref = 'id-abstraction-cart' />
  </UML:ModelElement.clientDependency>
</UML:Class>
<!-- User to Cart association -->
<UML:Association xmi.id = 'id-1' ...>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = 'id-2' ...>
      <UML:AssociationEnd.participant>
        <UML:Class xmi.idref = 'id-user' />
      </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
    <UML:AssociationEnd xmi.id = 'id-cart' ...>
      <UML:AssociationEnd.participant>
        <UML:Class xmi.idref = 'id-cart' />
      </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>
<!-- Interface ICart -->
<UML:Interface xmi.id = 'id-icart' ... />
<!-- User to ICart association -->
<UML:Abstraction xmi.id = 'id-abstraction-user' ...>
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = 'id-abstraction-stereotype' />

```

```

</UML:ModelElement.stereotype>
<UML:Dependency.client>
  <UML:Class xmi.idref = 'id-user' />
</UML:Dependency.client>
<UML:Dependency.supplier>
  <UML:Interface xmi.idref = 'id-icart' />
</UML:Dependency.supplier>
</UML:Abstraction>
<!-- Cart to ICart association -->
<UML:Abstraction xmi.id = 'id-abstraction-cart' ...
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = 'id-abstraction-stereotype' />
  </UML:ModelElement.stereotype>
  <UML:Dependency.client>
    <UML:Class xmi.idref = 'id-cart' />
  </UML:Dependency.client>
  <UML:Dependency.supplier>
    <UML:Interface xmi.idref = 'id-icart' />
  </UML:Dependency.supplier>
</UML:Abstraction>

```

Listing 4.13: Excerpt of XMI resulting from composition of *Command* and *Memento* software patterns.

Figure 4.22 depicts the graphical representation of the aforementioned XMI. In the figure it is possible to see that both pattern instance were merged into a single solution, where the class **User** is the connection of both patterns. It is possible to see in gray the entities which were merged, into the ones highlighted in bold (c.f. **User** and **Cart**).

4.7.5 Enhancement

The enhancement process is composed of two phases. In the first phase, the tool adds extra associations between pattern elements through the *stringing* operator. In the second phase, attributes are added to the classes.

The first phase relies in the domain model, in order to extract the possible additional relations between the classes. The relationship associations are extracted with the query presented in Listing 4.14. This query extracts the domain model connections of the type *CompositionOf*, which give indications for classes which should be connected via an association.

```

PREFIX : <http://www.rmsc.com#>
SELECT ?subject ?Object
WHERE {
  ?subject ?predicate ?object .
  FILTER(?predicate rdf:type :TypeOf)
}

```

Listing 4.14: SPARQL query to extract individuals of the category *TypeOf*.

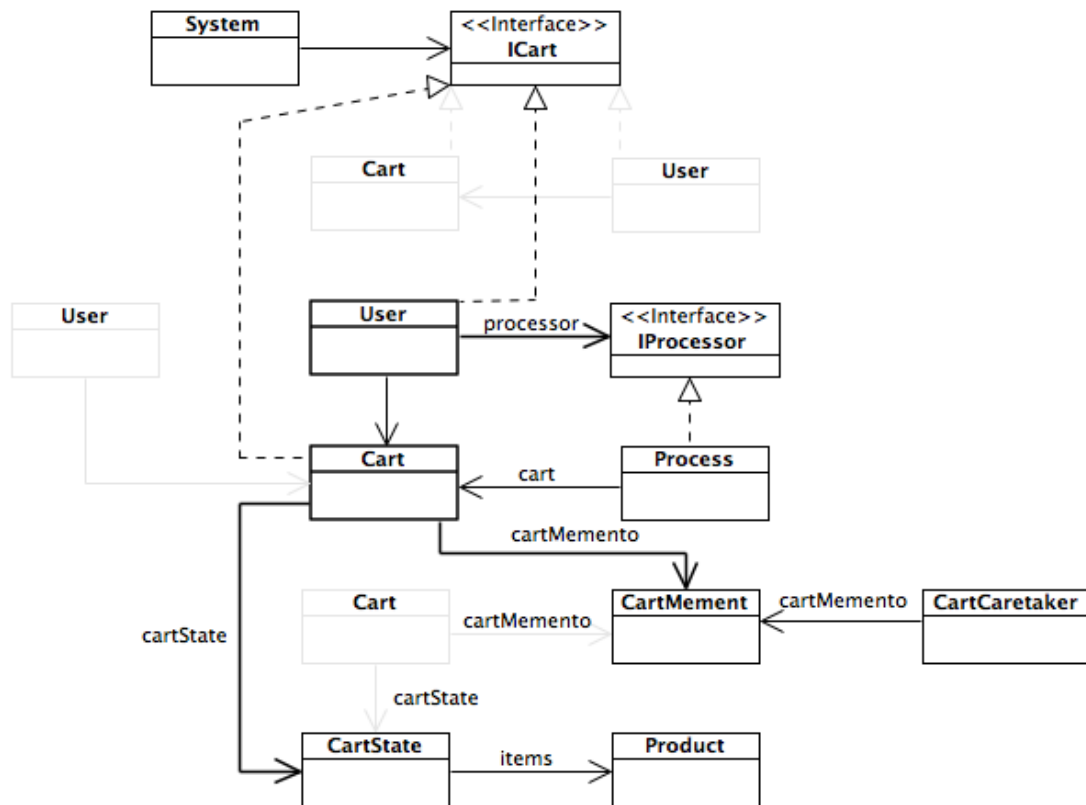


Figure 4.22: UML representation of the architecture resulting from the pattern composition process (with merged entities in gray).

From the domain model, uCat extracts that the entities **System** and **Product** are related via the *CompositionOf* property. Hence, a connection should be added between those entities. The same is true for **Cart** and **Product**. The resulting architecture of the first step of the improvement process is presented in Figure 4.23.

The second part of the enhancement process consists in adding attributes to the classes of the solution. Once again, the domain model is essential in order to extract that information. To identify which attributes belong to each class, a query to analyze the *PropertyOf* properties is used, as presented in Listing 4.15.

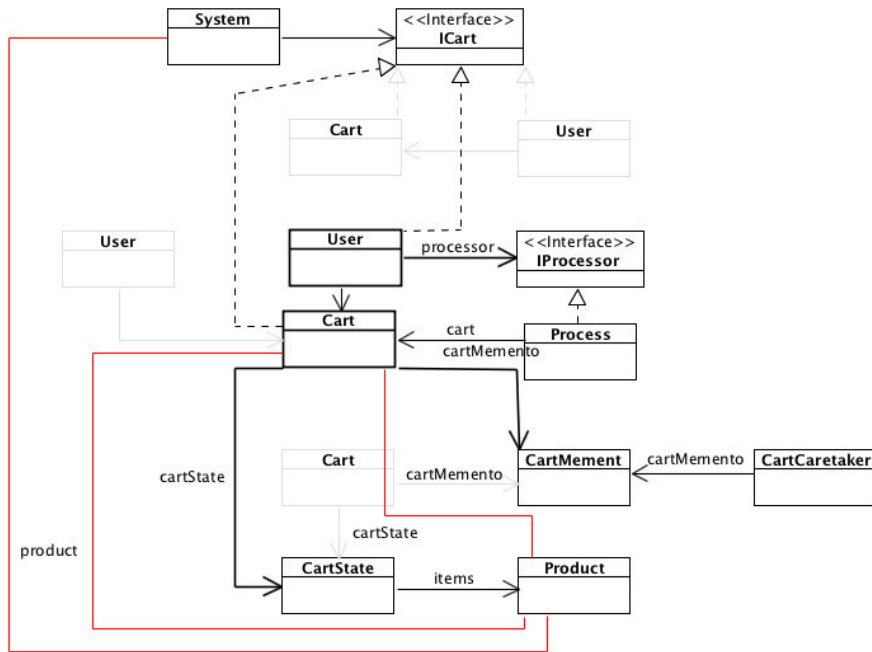


Figure 4.23: Enhancement of the solution via *stringing* merging operator.

```

PREFIX : <http://www.rmssc.com#>
SELECT ?subject ?object
WHERE {
  ?subject ?predicate ?object .
  FILTER(?predicate rdf:type :PropertyOf)
}

```

Listing 4.15: SPARQL query to extract individuals of the category *PropertyOf*.

From the domain model, is possible to extract, for instance, that entities *Name*, *Address*, *Email*, *Username* and *Password* are related with the entity *User* via *PropertyOf*, hence they are its attributes. As result, they are added to the corresponding entity, as presented in Figure 4.24.

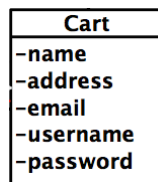


Figure 4.24: Excerpt of enhanced solution with additional attributes.

4.7.6 Producing XMI

In order to generate the XMI, a final step of concrete types specification, for the classes attributes, is required. While the enhancement process has the capability to add attributes to classes from the information in the domain model, their types must be manually specified. Hence, it remains for the user to specify the types, in order to generate the final XMI. XMI makes it possible to link the results of the SCARP process to other MDA tools.

uCat provides a plugin in order to support the types definition. As depicted on Figure 4.25, for each constituent of each pattern a list of fields is presented, to specify the corresponding types. For instance, the class `User` contains the `Password` field, for which the `String` type was defined. Similarly, for the remaining fields the types `String` for the `Username`, `String` for the `Address`, `Integer` for the `Age`, `String` for the `Name` and `String` for the `Email` were defined.

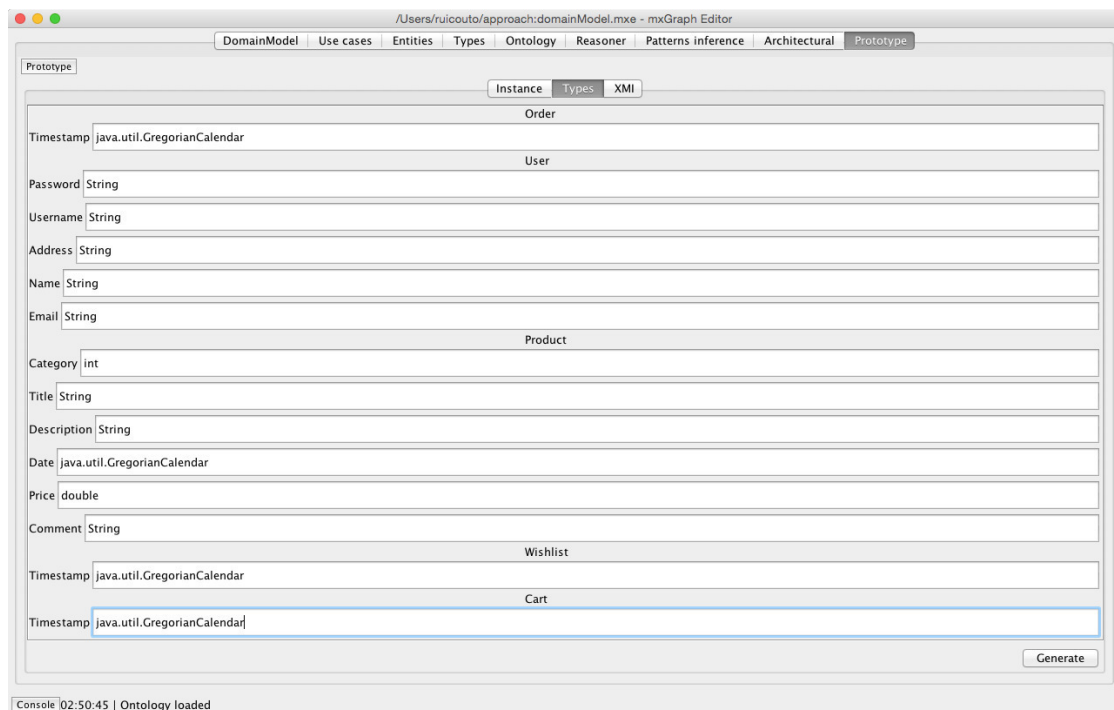


Figure 4.25: Types definition plugin user interface.

uCat supports the aforementioned processes, of software patterns interpretation and manipulation, instantiation, composition and enhancement, as well as the types definition. At the end of the process, the tool is also able to generate the corresponding XMI code, as presented in Listing 4.16. In the Listing it is possible to see for instance the classes `User` and `Cart`, as well as their relationships. It is also possible to see the relation between `System` and `Product`, which resulted from the enhancement process. The plugin supports also exporting the architecture to other formats, as for instance Ecore, since uCat provides an internal representation for the architectures. This way, the plugin can easily be extended to generate other standard formats.

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp =
'Mon Mar 14 14:55:29 WET 2016'>
  <XMI.content>
    <UML:Model xmi.id = 'id' name = 'model' isSpecification = 'false' isRoot
      = 'false' isLeaf = 'false' isAbstract = 'false'>
      <UML:Namespace.ownedElement>

        <UML:Class xmi.id = 'id1' name = 'Cart' visibility = 'public'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
          isAbstract = 'false' isActive = 'false'>
          ...
        </UML:Class>

        <UML:Class xmi.id = 'id2' name = 'System' visibility = 'public'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
          isAbstract = 'false' isActive = 'false'>
          ...
        </UML:Class>

        <UML:Class xmi.id = 'id3' name = 'User' visibility = 'public'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
          isAbstract = 'false' isActive = 'false'>
          ...
        </UML:Class>

        <UML:Association xmi.id = '...' ...>
          <UML:Association.connection>
            <UML:AssociationEnd xmi.id = '...' ...>
              <UML:AssociationEnd.participant>
                <UML:Class xmi.idref = 'id1' />
              </UML:AssociationEnd.participant>
            </UML:AssociationEnd>
            <UML:AssociationEnd xmi.id = '...' ...>
              <UML:AssociationEnd.participant>
                <UML:Class xmi.idref = 'id3' />
              </UML:AssociationEnd.participant>
            </UML:AssociationEnd>
          </UML:Association.connection>
        </UML:Association>

      </UML:Namespace.ownedElement>

      ...

    </UML:Model>
  </XMI.content>
</XMI>

```

Listing 4.16: Excerpt of the XMI resulting from the SCARP process.

4.8 Generation of Outputs

From the produced XMI it is now possible to generate concrete software outputs. As an example, two possibilities are described in this section. The first one corresponds to source code, and the second one to a user interface prototype. These outputs make it possible to perform a preliminary validation of the solution to be implemented.

4.8.1 Source Code

Generating source code from XMI is one of the possible approaches to operationalize the produced output. Several tools and approaches, notably the OMG MDA, already propose a solutions to this problem. For demonstration purposes, it is possible to take the models resulting from the SCARP, and use a tool to generate source code. Examples of tools able to do that are ArgoUML, EclipseEMF, Visual Paradigm, among many others.

The resulting XMI was imported into ArgoUML (c.f. Figure 4.26), and the corresponding Java source code automatically generated. An excerpt of the resulting code is shown in Listing 4.17. It is possible to see, for instance, in the class `User`, the attributes `Password` and `Username`, as well as the association with `Cart`. The objective of SCARP is not to produce final solutions. Instead, the focus is in generating architectural solutions which are iterated and improved as part of the software development process, as proposed by the MDA.

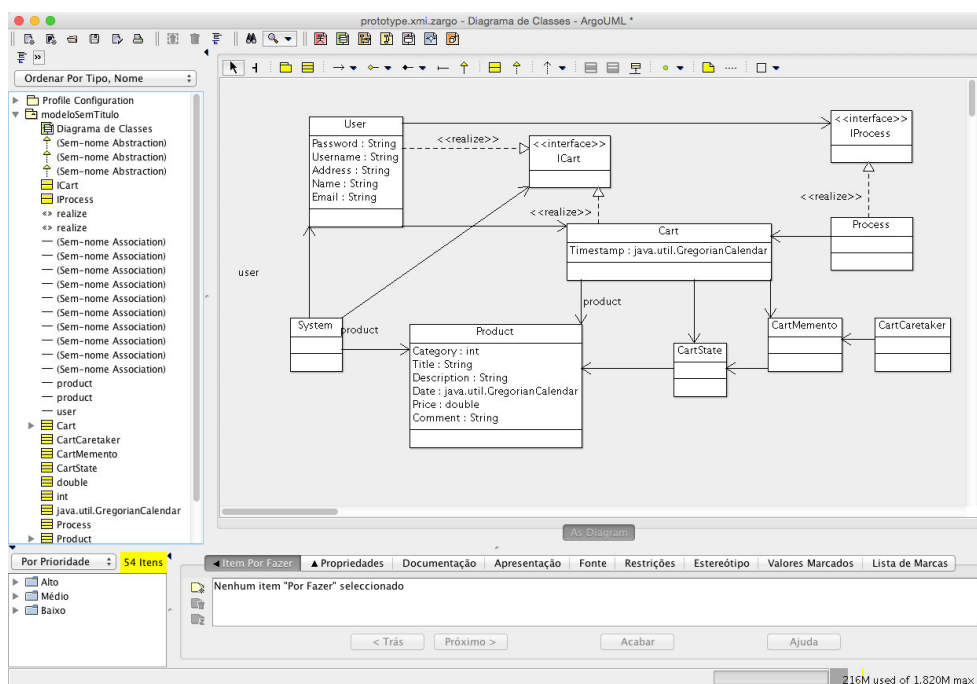


Figure 4.26: Representation in ArgoUML of the XMI produced in SCARP.

```

public class User implements ICart {
    private String Password;
    private String Username;
    private String Address;
    private String Name;
    private String Email;
    public Vector myCart;
    public Vector myIPProcess;
}

public class Cart implements ICart {
    private java.util.GregorianCalendar Timestamp;
    public Vector myCartMemento;
    public Vector myCartState;
    public Vector product;
}

public interface ICart {
}

```

Listing 4.17: Excerpt of the Java code generated by ArgoUML, from the produced XMI.

4.8.2 User Interface Prototypes

The integration of the SCARP process with other model based tools was explored. Specifically, with the MODUS approach and tool [84]. The MODUS approach, developed during the elaboration of this work, supports the automatic generation of UI prototypes from UML structural models (c.f. class diagrams). Similarly to SCARP, MODUS relies on the domain information, in order to make several assumptions, and automatically generate meaningful solutions. In practice, the models resulting from the Architectural Solution step (Figure 3.1, **IV**), were used as input for MODUS. As a result, MODUS was able to automatically generate an interface, as depicted in Figure 4.27.

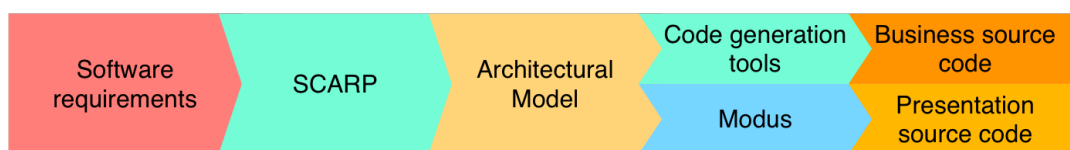


Figure 4.27: Usage of SCARP outputs in order to support the generation of presentation code. For this example, an Ecore representation (instead of XMI) was generated, and used as input for MODUS. MODUS supports the automatic generation of user interface for an eCommerce application. It is possible to explicitly see in Figure 4.28 the elements contained in the diagram shown on Figure 4.26. For instance, it is possible to see that pages for user (*profile*) and shopping cart (*My cart*) exist. Furthermore, the user is represented by its *username*, *name*, *email* and *password*.

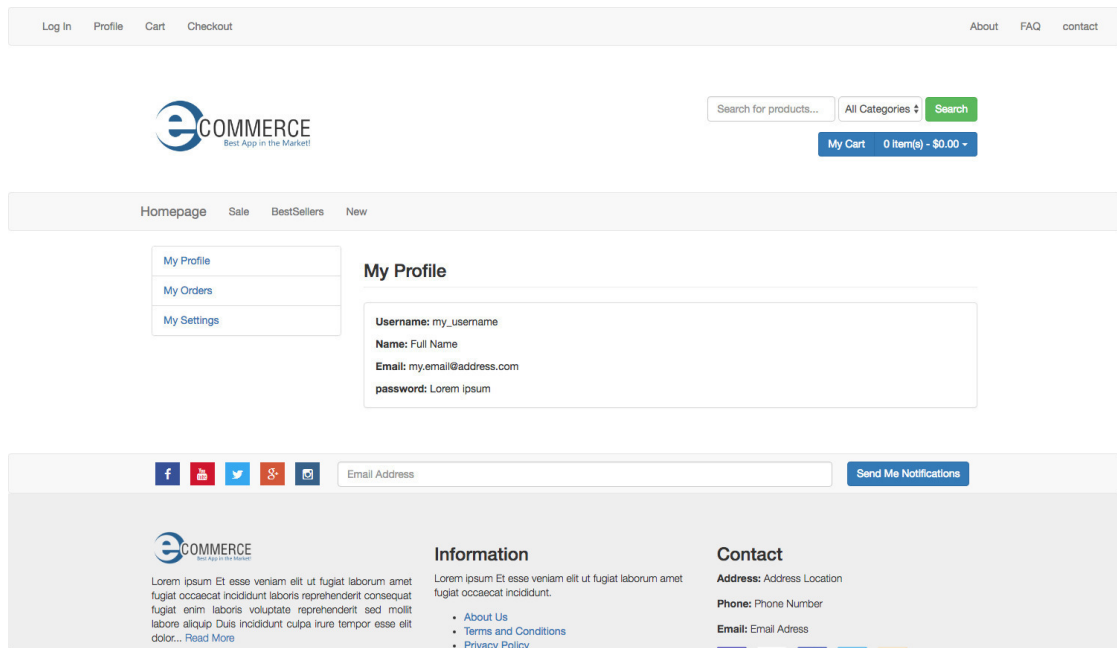


Figure 4.28: Example of MODUS web page, generated from an output of SCARP.

4.9 Summary

This chapter presented the decisions made in order to support the several steps of SCARP. For each step, a concrete implementation in terms of technologies, inputs and outputs was presented. Also, examples of inputs, which lead to the production of a software architecture were presented. Such architecture can be serialized into several formats, as is the case of XMI. The decisions were required to support the implementation of uCat, the tool that implements the processes described in each step of SCARP. Furthermore, uCat has the objective of operationalizing and automating some of the steps, in order to generate the architectural outputs (c.f. XMI and Ecore).

The suitability of SCARP to support Model Based User Interface Development is illustrated by the UI prototypes produced by MODUS. The interoperability and integration of SCARP with different tools and approaches (c.f. Visual Paradigm, ArgoUML [19], MODUS, etc.), is only possible resorting to standards. Indeed, SCARP supports exporting the generated data to XMI, the OMG standard for model interchange, it also supports representing the produced models in ECORE, which are useful to support a large set of model based tools (c.f. Eclipse EMF [107]), and which is the input format for the MODUS approach.

Chapter 5

Validation

This chapter is presents the validation of SCARP regarding the three following dimensions.

- Viability of the specification process, through assessment of both RUS and RUST;
- Acceptance of both the approach and tool by the users;
- Viability of the pattern inference process.

Two studies were carried out in order to perform the validation. The first study assessed both the first and second topics. The users were requested to create and interpret RUS specifications, while using the uCat tool. Their feedback regarding the experience with the tool/language was collected. The second study addressed the last topic, by requesting different users to write the same specifications in RUS, and perform pattern inference on those specifications.

The performed studies resulted in publications regarding the achieved results [26, 24, 28].

5.1 Study 1 Description

In the first study, which addressed the expressiveness and overall acceptance of the language, a set of questions about RUS (which provide also feedback for RUST) and uCat was defined (c.f. Table 5.1 and 5.2). From the questions a set of tasks for users to perform was extracted. A task consists in an exercise the user should perform (e.g. create or interpret a specification). By analyzing the result of the tasks, it was possible to answer the proposed questions. Data was collected during and at the end of the modeling tasks.

At the end, a questionnaire composed of two parts, and described in Table 5.1 and Table 5.2, was applied. Note that questions 1 to 3 and question 16 are open (although a numerical answer was expected), while the remaining questions, up to number 24, were answered in a 7-point Likert scale (0 meaning low and 7 meaning high). Questions 25 to 30 were answered with text.

Table 5.1: Questionnaire - part a).

	Question
1	Number of statements which required major changes in order to be mapped into RUS
2	Number of statements which lost their meaning
3	Number of unsupported statements
16	Minutes spent in adjustments (to match RUS)

Table 5.2: Questionnaire - part b).

	Question (low to high)
4	How much sense does the user interface makes
5	How familiar was the terminology
6	How much the tool helps in the specification
7	How easy to use is the RUS
8	How easy to understand is the RUS
9	How much easier is RUS to understand than NL
10	How easy is it to manipulate RUST
11	Is NL easier to use than RUS
12	Is NL easier to understand than RUS
13	Likelihood to adopt RUS
14	How easy is it to understand RUST
15	How clear is the language
17	How close to NL is RUS
18	How easy was it to understand the tool
19	How this tool is preferred over VP
20	How useful and adequate was the output
21	How acceptable are the tools' limitations
22	How easy to use is the tool
23	How much the user liked the language
24	How much the user liked the tool

The tool evaluation concerns usability aspects. Quesenbery [95] addresses five usability dimensions, as follows.

1. *Effective* corresponds to how complete and accurately the work is completed;
2. *Efficient* Corresponds to how quickly the work can be performed;
3. *Engaging* Corresponds to how well the interface captures the user attention, and how satisfying is it to use;
4. *Error Tolerant* Corresponds to how well the application prevents and helps the user recover from errors;
5. *Easy to Learn* to how well the application supports the learning process through usage.

These dimensions are addressed in this study, in the following manner: **1.** is addressed by questions 1, 19 and 21; **2.** by question 17; **3.** by questions 4, 13, 23 and 24; **4.** by questions 2, 3 and 20; **5.** questions 5-12, 14-18 and 22.

5.1.1 Objectives

The objective of the study was to analyze how RUS performs with real users, and, how uCat performs at supporting the language. The objectives for RUS, according to Section 3.4.1, were as follows.

- 1.1. Provide formalism to use cases with minimal extra costs for the user - such includes for instance a seamless transition from natural language to RUS, without losing the meaning or expressiveness of the original statements; furthermore, it is not intended for users to have a background in formal methods.
- 1.2. Support the users' specification needs - RUS should support the users' use case specification needs, in order to make SCARP viable; at the same time, improving the specification process is intended, by encouraging the use of templates (or patterns), by demanding users to follow the RUS.
- 1.3. Be easy enough to understand and manipulate according to the users' needs - if so, the users will be more likely to adopt it.

As uCat supports RUS, tool specific objectives are somehow related, and were elicited as follows.

- 2.1. Be easy to learn - this objective corresponds to a tool that is easy to use, and does not require an extensive adaptation period; it is also evaluated how suitable the interface is regarding the use of familiar and self explanatory terminology.
- 2.2. Have a good acceptance - it is evaluated how likely it is that uCat will be able to complement or even substitute other UML supporting tools, regarding the formalism it provides.
- 2.3. Provide a good support for the language - the tool must be able to both support and improve the usage of RUS.

5.1.2 Study Setup

In order to evaluate the expressiveness of the language, a number of tasks was defined (see Figure 5.1) as described next.

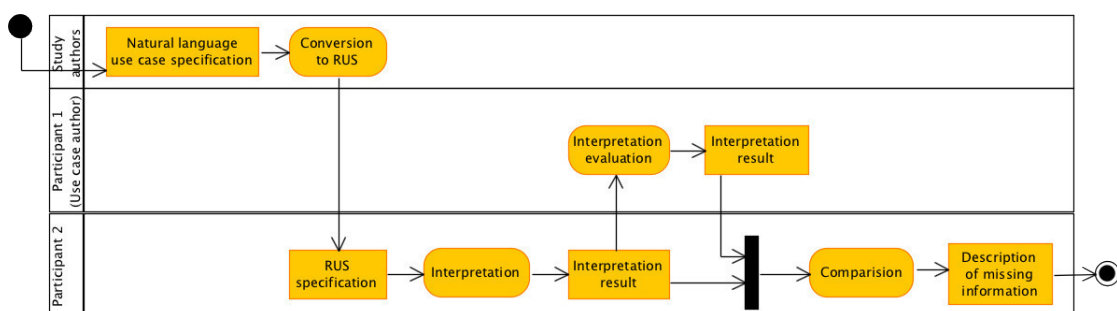


Figure 5.1: RUS validation process.

1. A number of use cases were converted into RUS beforehand, by the authors of the study;
2. The participants were asked to interpret and textually describe the RUS use cases;
3. The textual descriptions were handed to the original use case authors, which evaluated them;
4. The participants were presented with the original use cases, and asked to point out any missing information from the RUS version.

The use cases used resulted from a previous unrelated work performed by the participants, in the context of a software engineering masters, where several functional requirements were described. The requirements, are described in the form of use cases, related to either social projects' or a football tournaments' management system. From all the specifications, the works with higher grades were selected. From those works, the use cases with the higher number of steps were chosen (see Appendix B.1 for the set of selected use cases).

Table 5.3: Excerpt of a RUS use case specification used in the study.

	User Input	System Response
1	user inserts name	
2		system searches tournament
3		system shows tournaments
4	user selects tournament	
5		system shows tournament
6	user selects remove	
7		system requests confirmation
8	user provides confirmation	
9		system removes tournament
10		system informs success

The use case presented in Table 5.3 is one of the use cases used in the study, presented here for illustration purposes. It describes the process of removing a tournament from a football team management system. The process consists in a user providing the name of the tournament to be removed, which the system searches for. The system presents the existing tournaments, from which the user selects the one to remove, confirms the action, and the system proceeds to remove it.

In total, 8 distinct use case scenarios were selected (c.f. Appendix B.2), and formalized in RUS (see Table 5.5 for an excerpt of such descriptions). The process resulted in a set of RUS use cases for a domain known to the test subjects, while avoiding their contact with the language. Next, scripts for the participants were created, containing the tasks for each stage and instructions about how to perform them. Figure 5.2 and 5.3 present the results of the questionnaire.

In order to evaluate the expressiveness of the language and the acceptance of the tool, the same users were asked to perform a set of additional steps.

5. A new scenario describing textually a system's functionality was provided to the participants, which were asked to write the corresponding use case, using the tabular representation and natural language;

6. At this point, both the language and the tool were introduced. The users were asked to convert the use case into RUS, with the tool;
7. A new scenario was presented, and the users asked to write it in RUS, on the tool;
8. The use cases were handed to other users, which interpreted them; each author evaluated the descriptions' correctness;
9. A RUS entry was presented, and the users were asked to write the corresponding RUST.

In relevant tasks the time required to perform them was measured. At the end, a questionnaire was applied.

The scenarios for tasks 5 and 7 concerned a web application context. Three different scenarios were used: "Upload a model to a repository", "Download a model from a webpage" and "Register on a group on a web application". The scenarios were written in Portuguese as that was the participants' native language. The scenarios were translated into english for this work. As an example the latter scenario was as follows (see Appendix B.1 for all scenarios descriptions):

A user clicks in "groups" link. The system shows the available groups. Next, the user views the list, and selects one to register to. The user selects "register", the system registers the user in the group and shows a success message. If the group is private, after selecting "register" the system sends a message to the group author (with an admission request) and shows an information message, instead of a success message.

As it will be described in the next sections, several groups of test subjects were involved in the study. In all cases the participants were students from an Informatics Engineering course, at the University of Minho. They had obtained final grades on the 80th percentile in a previous software engineering course. They all had previous contact with the use cases *user input/system response* style, and tabular representation, as proposed by Fowler. The participants performed the study in an isolated environment and without interaction with each other. None of them had previous contact either with RUS, the tool or SCARP. To perform the study, each participant had a computer and a printed script of the study, and individually performed each task. Every task was previously explained before its execution. The steps were carried out sequentially, and all the users performed them at the same time.

5.1.3 Addressing the Objectives

Objective 1.1 is addressed by measuring the overall time required by the users in order to adapt to RUS, and by how correct their specifications are. In tasks 2, 3 and 4, the language acceptance is measured, and in 6, 7 and 8 the time required by the users. Objective 1.2 is mainly evaluated by the reports about how the users understood predefined statements. Task 3 measures how able is RUS to support the use case specifications. Objective 1.3 is evaluated with two approaches. First by measuring how valid the users' inputs are, regarding a provided set of RUST entries. Second by analyzing how the users were able to manipulate RUST and the corresponding required time.

In order to evaluate objective 2.1, the time spent using the tool is measured, as well as the number of tries required in order to successfully create a use case specification. Evaluation of objective 2.2 relies on the users' feedback, about the likelihood of adopting the language. Objective 2.3 is measured by the capability of the users to write the use case specification (overall number of supported statements) and take advantage of the language's capability (for instance, alternatives and exceptions). Table 5.4 relates each objective to the set of corresponding tasks. The outputs produced by tasks 1 and 5 do not contribute directly to the objectives, but are required for the remaining tasks.

Table 5.4: Study objectives and corresponding tasks.

Objective	Task 2	Task 3	Task 4	Task 6	Task 7	Task 8	Task 9	Feedback
1.1	×	×	×	×	×	×		
1.2		×						
1.3				×	×	×	×	
2.1				×	×			
2.2								×
2.3				×	×	×		

5.1.4 Study Validation

A preliminary session was carried out with five participants, to validate the study's design. The study itself was performed with the authors of the use case descriptions. All users were able to perform the study and the data collected was useful for analysis. No major issues were found, affecting the users' performance. No questions were also raised regarding the questionnaire, neither complaints regarding the duration of the study. Nevertheless, a number of adjustments was identified, which are described in the next section.

5.2 Study 1 Execution

After the validation, the study was applied to a larger set of participants [28]. A more clear organization of the study (i.e. the stages and objectives) reflects adjustments resulting from the validation study (c.f. the previous section). Specifically, it is worth noting the separation of the study in two stages.

STAGE 1 aimed to validate the expressiveness of the language. It addressed three of the main objectives defined for the approach:

Objective 1 Provide formalism with a minimal effort for the users;

Objective 2 Provide a language which is expressive enough to support use case specifications;

Objective 3 Provide a language which is easy to use.

STAGE 2 addresses the tool's support for RUS. It addressed three more objectives of the approach (in this case, related specifically to uCat):

Objective 4 Be easy to learn how to use;

Objective 5 Be acceptable as a complement/substitution for other tools;

Objective 6 Provide a good support for the RUS language.

5.2.1 Execution of the study

A group of 18 participants (16 male, 2 female), with ages between 20 and 25 years with a mean of 21, was selected to perform the evaluation stages. Participants were distributed by four sessions. Each session took about 130 minutes: 30 minutes for presenting the tool and the language, 90 minutes for the study, 10 minutes for the questionnaire. In each session, the participants were gathered in a room, and asked to perform the scripts individually. No time limits were imposed. In each session the participants performed both the first and the second stages of the study in sequence.

5.2.2 Results of the Study

In both stages the participants demonstrated autonomy while performing the tasks. As presented next, in some steps some questions arose, but all regarding minor issues.

Table 5.6 summarizes the results from the first stage of the study regarding a) whether the use case descriptions produced in Task 1 were correct (Task 2); and b) whether there were differences between the Natural Language and the RUS use case descriptions (Task 3). In the second task, only 15 of the 18 participants answered the questions.

In the second stage of the study, while writing the specifications in natural language was straightforward, writing them in the tool generated some questions. Most common questions regarded input mismatches (for instance, a trailing space in a statement, or how to write a multiple word entity such as “work plan”). All the questions were easily answered, not affecting the study in a negative way. An example of a specification, both in natural language and in RUS is presented in Table 5.5.

An excerpt of a RUS use case produced by a participant is shown in Table 5.7, starting from a Natural Language use case. It corresponds to the success case for the “*Register on a group on a web application*” scenario from Section 5.2.1, and it is a case where the author of the Natural Language use case considered the RUS version a correct version of the original NL use case.

Figures 5.2 and 5.3 present the questionnaires results up to question 24 (mean value for numeric answers, and mode for Likert-scale answers, respectively). These results will be discussed next.

Table 5.5: Excerpt of a collected use case and its RUS version.

Original version			RUS version		
	User input	System response		User input	System response
1	inserts project identifier		1	user inserts project_id	
2		confirms project existence	2		system confirms project
3	inserts work plan		3	user inserts work.plan	
4		confirms insertion of work plan	4		system confirms insertion

The open questions in the questionnaire enabled participants to express their experience. The first question was: “*What became harder by using RUS (over NL)?*”. From the participants which decided to answer the question, three answered that nothing became harder, while eight referred the need to learn and adjust to the RUS syntax. Six participants answered that it was to map more complex statements into RUS. On the contrary, when asked “*What became easier by using RUS (over NL)?*”, seven participants answered that it was the interpretation, as the descriptions became simpler. Four participants referred the standardization of the specification, while three referred that it became easier to create specifications. Three participants stated that it became easier to create specification (vs Natural Language), and one participant answered that it was easier to create correct specifications.

Another question asked was: “*What did you like in the language?*”. Thirteen participants answered that it was its simplicity, two participants referred the standardization of the specifications, one the interpretation of use cases produced by other authors, another that it speeds up the specification process, and one mentioned nothing. To the question: “*What did you dislike in the language?*”, nine participants answered that there was nothing that they disliked, three participants reported the need to adapt Natural Language, other three the limited set of keywords to support statements, a clear indicator that they did not understand RUST. Other two participants reported the required learning time, and one participant how alternative scenarios could be specified (although that is an issue related more with the tool).

It was also asked if the users preferred the RUS format (supported by uCat), or another free text format input tool they knew. Ten participants stated that they preferred RUS because of the standardization provided by the format, six participants mentioned the lightweight interface, and two the easier way to specify alternatives. One user stated that it becomes closer to a programming process. When asked the question: “*What did you like in the tool?*” most participants (seven), mentioned its simplicity, four the capability to validate the use cases while specifying them, three the formatted input, and two the representation of the information. One participant referred the familiar interface, and another the possibility for the tool to be a viable replacement for other tools. On the contrary, when asked: “*What did you dislike in the tool?*” ten participants pointed nothing, three mentioned the restrictions on the specifications format, and three proposed improvements in the alternative scenarios specification. One referred minor bugs, and another stated that specifications became harder to read.

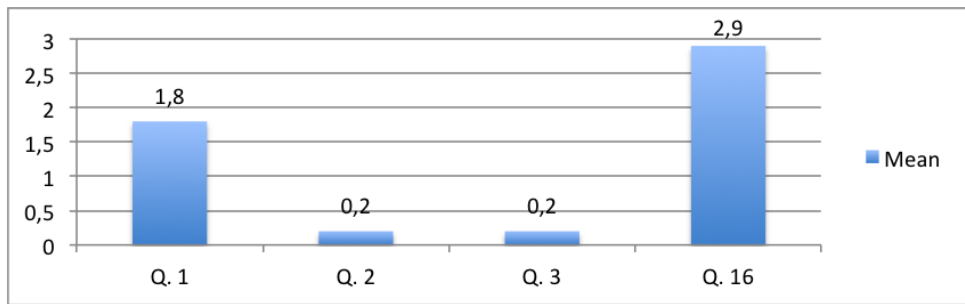


Figure 5.2: Questionnaire results for numeric questions.

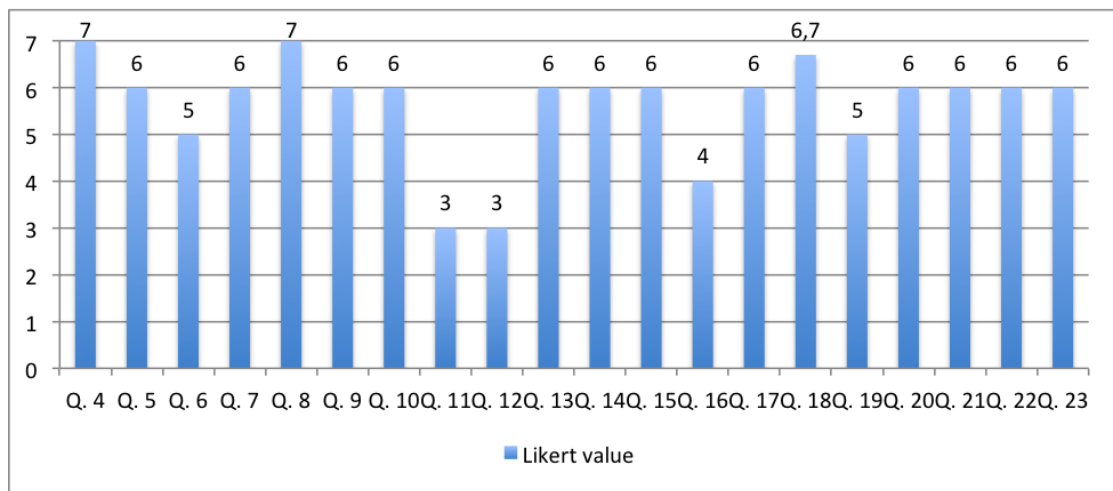


Figure 5.3: Questionnaire result for Likert scale questions.

Table 5.6: RUS evaluation.

Question	✓	~	×
a) Use case description is correct	12	2	1
b) Versions are identical	14	4	0

(✓ – correct; ~ – had issues; × – incorrect)

Table 5.7: Excerpt of use case made by a participant.

	User Input	System Response
1	user clicks groups	
2		system provides the groups
3	user selects group	
4	user clicks register	
5		system register user
6		system provides success_message

5.2.3 Discussion

Regarding the performed tasks, both the participant's results and observed behaviors provide relevant insights. **Tasks 1 and 2** provide hints about how the participants were able to correctly understand the presented RUS. Only two of the descriptions produced by the participants were considered to have minor issues. One resulted from an interpretation error, where a use case element (**user**) was used instead of another (**player**). The other consisted on a poor description of the scenario. Participants described the steps of the specification, instead of interpreting the scenarios. On a positive note, none of the interpretation errors was due to either the language or the translation process.

The results from **Task 3** show that the participants were able to both understand and express the use cases in RUS without issues, and that the language had enough expressiveness. None of the users reported missing information, which would lead to changes in the RUS use case descriptions. In fact, only five participants referred some missing detail; one stated that the RUS version was more compact, and the remaining stated that both descriptions had the same information.

From **Tasks 4, 5 and 6** it is possible to draw several conclusions. The participants required an average of 27 seconds per statement (s/s) when creating the tabular Natural Language use case descriptions. While writing the same description in RUS, the users required an average of 70s/s. That is somehow understandable considering that the users had no previous training, and therefore needed an adaptation period. However, when writing a description for the second time, the participants required an average of 52s/s (18s/s less), corresponding to an improvement of about 25%. A two-sample T-test (for iteration 1 and 2) was applied, for the null hypothesis that there is no difference between the two populations means. The result, for a confidence level of 95%, states that the mean value of iteration 1 is greater. This clearly indicates that through practice, the users were able to reduce the time required to write RUS statements.

In **Task 7** participants were able to correctly describe the use cases written in RUS. From the 18 descriptions, just in two were missing details pointed out. These were not related with the language description, rather with brevity of the users descriptions. This is another positive indication about the expressiveness of RUS.

The presented results, plus the participants' feedback, clearly indicate that they were receptive to SCARP. Participants were able to perform the tasks independently. The formalization of the use cases was possible with minor overhead, and there was no loss of information in the process. Only one of the specifications produced in the second stage contained issues. In that case, the participant clearly did not understand the purpose of the language, as the remaining specifications were correct. Analysis of the specifications from the third stage shows that, overall, participants made a correct usage of RUS syntax. All the specifications were valid, and correctly described the corresponding scenarios. The lack of issues and the participants' autonomy are good indicators regarding both the tool and language.

The specifications varied in the used verbs, the number of statements, and in the names given to entities (e.g. the terms *search_link* and *searchLink* were used by different participants to refer to a “search link”). For instance, to describe the “Download a model from a webpage” scenario, (from stage 2), 7 participants used 7 or 8 statements, while the other 3 used 6, 12 and 13. Each scenario presented in **Task 8**, varied in the number of statements required by different participants to describe them in RUS. Namely, scenario **1)** and **2)** varied from 5 to 9 statements, scenario **3)** from 7 to 12 statements and scenario **3)** from 3 to 9 statements. These results demonstrate the existence of variability in the descriptions.

Next it is described in more detail how each objective from Section 5.2 relates to the results of the questionnaire from stage 2.

Objective 1 is related to questions 9, 11, 12 and 16. The answers show that participants preferred the RUS approach over Natural Language. Several factors contributed to this result (as presented next), but overall the participants liked the standardization provided by the language.

Objective 2 is related to questions 1, 2, 3 and 22. On questions 1, 2 and 3, on average 1.8 statements (for an average of 14.2 statements) required some kind of adjustment when mapping into RUS. This result is a good indicator that the language has a good expressiveness. From question 22, it is possible to conclude that the participants expressed empathy with the language. The participants considered also that the RUS language is easy to learn and RUST is easy to understand and manipulate (mode 6, on questions 8 and 14 and 10 respectively), which contributes to achieve this objective. These answers are in line with the results from the aforementioned tasks.

Objective 3 is related to questions 7 to 15. From these questions, it is possible to concluded that the participants considered the tool to be easy to understand and use, even when compared with Natural Language.

Objective 4 is related to questions 5, 17, 19, 21 and 23. With a mode of 6 for these questions, the tool had a good acceptance by the participants (being easy to use and learn). Question 23 provides also feedback for the tool, and follows the trend of the other questions.

Objective 5 is related to questions 4, 18 and 20. These questions have modes of 6 and 7. The results show that participants are highly receptive of using uCat as replacement/complement to other tools.

Objective 6 is related to questions 6, 19 and 21, which have modes of 5 and 6. This result shows that the participants consider the tool able to support the language.

Beyond the questionnaire, direct observation during the study supported concluding that the tool played a relevant role in the specification process. First, by ensuring the correctness of the specifications, as the tool forces the participants to input valid RUS statements, since only valid specifications are accepted. Second, the tool provided runtime feedback regarding the statements: once the user finished writing a statement, it was immediately verified and highlight if incorrect.

Thus, the test subjects not only knew if the specification was valid, but what statements were invalid.

5.2.4 Summary

Regarding the language, the participants successfully both interpreted it and created new specifications, even without previous training. Not only were they able to produce specifications in an acceptable time interval, but practice further reduced the time required to write them. The tool's performance results are also positive. It performed well and generated positive feedback, successfully supporting all the tasks in the study. As a whole, results indicate that the proposed use cases formalization approach is feasible, and that the tool provides good support for the approach. It is possible to write use cases, without a major effort and without losing expressiveness, such that they can be automatically formalized.

Performing the validation study was not only relevant to validate the formalization approach, but also the supporting tool. It is not only relevant to have tools which support formalization mechanisms, but it is also important for them to have a good acceptance by the final users. By successfully validating uCat, it was possible to gather users' feedback, improve the tool, and ultimately foster the adaptation of formalization techniques. Results indicate uCat is an adequate tool to provide support for the specification process required to support the approach. Study 2 further explores this aspect.

5.3 Study 2 - Pattern Inference Process and Usability Assessment

The second study was developed in order to access the following aspects:

- OWL's capability to formalize use cases;
- OWL's expressiveness to support the rationale required by the patterns inference process;
- SPARQL potential to infer requirement patterns from such knowledge;
- The capability of uQL to specify requirement patterns;
- The usability of the tool.

Furthermore, from that study, a set of requirement patterns, with adjusted percentages, was extracted.

5.3.1 Setup of the Experiment

In order to perform the study a group of 21 participants (18 male, 3 female) with an average age of 24 years, and approximately 2,5 years of experience in specifying use cases was selected. All were students from a master course. The study was performed as part of a class assignment during approximately 1,5 hours.

The study started by presenting both the RUS language and the uCat tool (during approximately 20 and 10 minutes, respectively). It was demonstrated how the tool supports the language, by specifying a use case in the tool (adding a product to the shopping cart). Participants had the opportunity to interact with uCat and to create themselves a specification, in order to get used to the tool. This took about 10 minutes. Note that, given their background, all participants were well versed in using software modeling tools. For this stage participants had to carry out a single task. They were asked to specify a set of five textual descriptions of usage scenarios for an eCommerce website in RUS. Each scenario was designed to capture an eCommerce pattern. Next follows the list of patterns used to create the scenarios (see Appendix B.2 for the scenarios' specification).

Simple search "Provide users with powerful, yet simple, search mechanisms." [82];

Catalog pattern "The catalog pattern organizes information about the products sold in a web site." [35];

Session "Many objects need access to shared values, but the values are not unique throughout the system." [124];

List builder "The users need to build up and manage a list of items" [116].

Shopping basket "A mechanism that keeps track of items selected by the user." [116].

As an example, the first scenario is as follows:

A user clicks in the "search" link in the website. Then, the system shows a field where the user should insert the keyword to search, as well as the search criteria (price, date, etc.). When the user clicks "ok", the system performs a search (based in the given criteria), creates a result list and shows such list to the user. Finally, the user checks the resulting list.

This task took about 50 minutes to complete. Again no time restrictions were imposed. After completing the specification, participants answered the System Usability Scale (SUS) questionnaire [12]. The specifications were kept to validate the requirement patterns. To perform the study, all the participants were gathered in the same room. Each participant had a laptop and was asked to install uCat. A printed page containing all the usage scenarios and another containing the SUS questionnaire were printed and handed.

The identification of the requirement patterns occurred in four steps. First, syntactical errors were removed from the specifications, as addressing them is not relevant for the valida-

tion process. Second, each use case specification was analyzed, to identify recurring properties across the several specifications. For instance, for all users was observed that the first statement of the use case search scenario could be generalized as follows: “(the) user clicks or asks for the search, searchLink or search.link”. Third, a set of generic enough statements was wrote with the objective to support the specifications. Such lead, for instance, to the statement (system) (shows|asks|displays) (?). The set of statements constitutes a requirement pattern. Fourth, the patterns were tested in order to check if they correctly identified the statements they where specified to identify.

The evaluation process occurred in two steps. First, each pattern was applied to the knowledge base in order to identify both the success ratio and the false positives. This is useful to understand if a pattern is identifying the correct knowledge. Second, each pattern was iteratively analyzed to identify the most relevant statements. The percentages for those statements were adjusted, and applied again. The results were compared with the previous percentages. The percentage adjustment process supports tuning the patterns’ percentages, in order to improve the correct identification ratio, and lower the false positives percentage.

5.3.2 Results of the Experiment

The participants were autonomous while performing the study. They were able to use the tool and create the specifications without external intervention regarding the specification process. Example of minor questions that arose during the study are as follows: “should I use a verb here?”, or, “should I split this specification in two statements?”, or how to perform some action in the tool: “how do I save the specification?”, or even minor questions related with the scenarios “should I specify the action of viewing an item?”. All these minor questions were solved, and the study proceed without issues.

The overall feedback regarding the tool was positive. The participants had previous contact with modeling tools, and stated that uCat was lighter, since it was faster to open and register user actions, having a small size and an interface that responded quickly. Users expressed preference for uCat, over previous used tools. Some participants noted that they would rather use this tool in the classes as it leads to more standardized specifications.

Regarding the “*Perform search in the site*” scenario, the specifications were as expected. The users correctly resorted to the provided syntax and created meaningful specifications. A RUS specification provided by a participant is as follows:

	User Input	System Response
1	user clicks in search	
2		system shows the search_field
3	user inserts the keyword, search_criteria	
4		system performs search
5		system creates list
6		system shows list
7	user checks list	

It is possible to see that the scenario correctly describes the textual description, and is a correct RUS specification. All the specifications followed the same formats with similar descriptions. There were slight differences regarding the names of entities and in the choice of steps, which was expected and useful in order to evaluate the pattern inference process.

Two issues arose regarding the specifications. First, one test subject was not able to use the syntax correctly. The test subject did not write the specifications as expected. Second, some test subjects (as English is not their native language), performed some syntactical errors. Despite the errors not affecting the process itself, they can affect the inference process.

From all participants, 14 were able to correctly describe the use cases. From the 14 participants, 4 pointed out some issue with the description (not deterrent for the study itself). Regarding the 4 that had issues, two participants pointed out missing descriptions of the alternative scenarios (despite their being in the specification), one pointed out a missing bit of context information (it was not stated that the group could be private, according to the aforementioned scenario), and another misunderstood “information” for “success message”. Hence, the participants overall produced correct use case specifications for the given scenario.

A SUS questionnaire was applied. It consists of ten questions about a system which are answered using a 5 point Likert scale (from 1 – Strongly disagree – to 5 – Strongly agree). From the answers provided, a score is calculated which has been shown to have a strong correlation with the perceived usability of the user interface being analyzed. uCat scored a value of 74, meaning that it has higher perceived usability than (approximately) 72% of all products tested [13]. The corresponding grade is **B**. Although the individual results of SUS (c.f. Table 5.8) are not as relevant as the global score value, they provide further details regarding the users’ feedback.

Table 5.8: Detailed SUS results.

Question	Mode
I think that I would like to use this system frequently.	3
I found the system unnecessarily complex.	2
I thought the system was easy to use.	4
I think that I would need the support of a technical person to be able to use this system.	2
I found the various functions in this system were well integrated.	3
I thought there was too much inconsistency in this system.	2
I would imagine that most people would learn to use this system very quickly.	4
I found the system very cumbersome to use.	1
I felt very confident using the system.	4
I needed to learn a lot of things before I could get going with this system.	2

5.3.3 Discussion

The lack of issues regarding the use of both the language and the tool is a positive outcome. In the previous study both the language and the tool were successfully validated (c.f. Section 5.2). The results from this study are another positive hint regarding the language expressiveness and tool support.

This study had a shorter training phase than the previous one, but still the participants were able to correct specifications. This shows that the users were able to quickly learn and make a correct use of the language.

Regarding the invalid specification produced by one of the participants. That case is taken as an exception, since it was an isolated case, most likely from not paying attention while the process was being explained. The user provided specifications as “`system creates result list and shows`”. As explained at the beginning of the study, such is not supported in the RUS. Instead, it was expected a two statement specification, for instance: “`system creates result_list`” and “`system shows result_list`”. As for the syntactical errors, they can be mitigated. In order to achieve this a glossary was developed and integrated in the tool. On the one hand it provides syntactical verification, on the other hand helps to limit the used terms regarding the specification context.

A set of four requirement patterns was achieved, by analyzing and extracting the similarities from the use case specifications.

Listing 5.1 presents the *simple search* pattern (written in uQL) resulting from the described process (see Appendix B.3 for the applied pattern catalog).

```
(user) (clicks|asks) (search|search_link|searchLink) 13
(system) (shows|asks|displays) (?) 13
(user) (inserts|enters) (keyword|date|price|criteria) 13
(user) (clicks) (ok) 13
(system) (performs) (search) 12
(system) (creates) (result_list|list|resultlist|result|results) 12
(system) (shows|displays|display|presents) (list|result_list|results) 12
(user) (checks) (list|resulting_list|info|result|results) 12
```

Listing 5.1: *Simple Search* requirement pattern, written in uQL.

These patterns went through a fine tuning process. Figure 5.4 depicts the improvement in the identification ratio of the *Catalog* pattern, before and after adjusting the percentages. The mean value of the identification of the *Catalog* pattern improved from 77% to 80%, while the false positives decreased approximately 1.5%. This is an indication that queries can be tuned to better identify the patterns (and reduce false positives). The systematic process of analyzing the provided specifications, and generating the requirement patterns could be, relatively easily, automated. Such would allow to automatically deduce a uQL query from a set of specifications. Furthermore, the larger the number of specifications, the more accurate the queries would be.

5.3. STUDY 2 - PATTERN INFERENCE PROCESS AND USABILITY ASSESSMENT 129



Figure 5.4: Results before and after adjusting the percentages, for the inference of the pattern catalog.

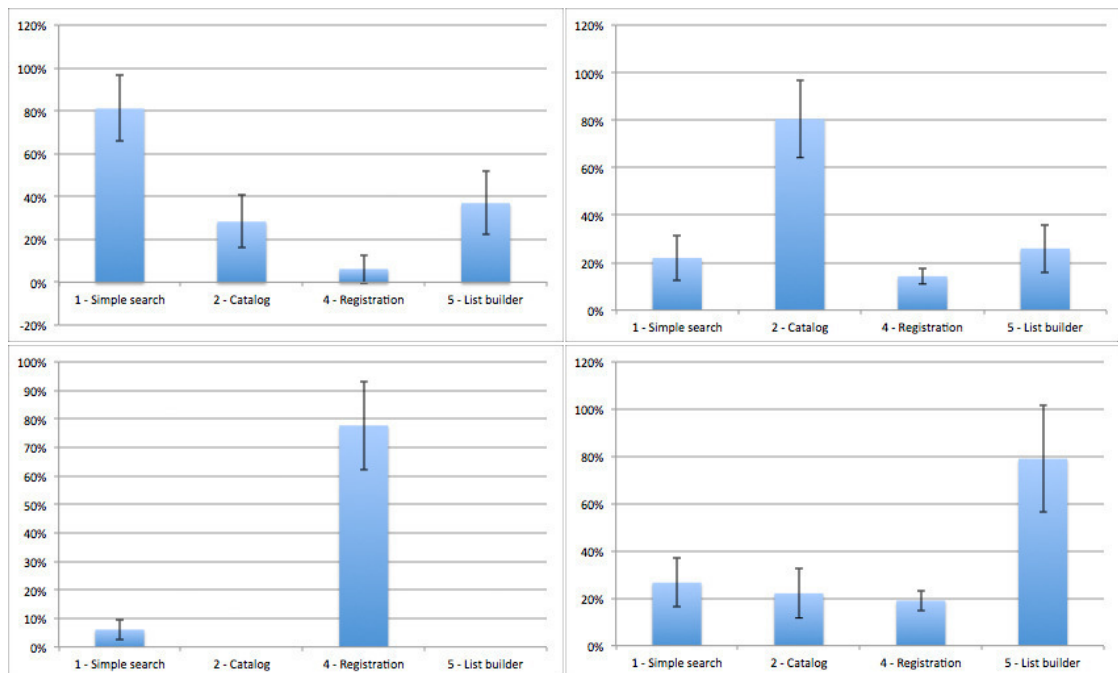


Figure 5.5: Patterns inference resulting percentages (average values and standard deviation) for *Simple Search*, *Catalog*, *Registration* and *List Builder* requirement patterns.

Figure 5.5 presents an overview of the result of applying the final versions of the four patterns to the use case specifications. In each chart in the figure is represented the average value of the match percentage for the corresponding pattern, across all the users. Each chart corresponds to a pattern, respectively “Simple search”, “Catalog”, “Session” and “List Builder”.

An overall analysis of the chart indicates that all patterns have been successfully inferred, with a match percentage of 80%. The standard deviation, on the upper bound was about 100% match, and on the lower bound above 50% match. The false positives had an average value under 30%, standard deviation under 20% for the lower bound and under 40% for the upper bound.

The first chart represents the inference of the “Simple search”. It is possible to see that the first bar (corresponding to the pattern) clearly indicates the presence of the pattern. The false positives have relatively low values, being “List builder” the higher. Such is however somehow expected. The list builder pattern has an overlapping with the “Simple search”, as both patterns share some steps regarding the production of a list of results. The relatively high value of the “List Builder” pattern is an indication that the “Simple search” contains part of the “List Builder”. In the second and fourth charts, for the “Catalog” and “List builder” patterns respectively, the inference is clear. The “Registration” pattern had a relatively more complex specification than the other catalogs. As presented, such resulted in a more clear identification of the patterns, and drastically reduced the false positives.

Despite the variation on the use case descriptions (both in terminology and number of statements), the achieved queries allows to correctly identify the requirement patterns. Indeed, the requirement patterns were identified with a confidence of (approximately) 80%. That means that, for the given specifications, in each description that contained a pattern, that pattern was identified with an 80% or more precision. Regarding false positives, they all had a value under 30%.

5.4 Threats to Validity

While the results provide positive feedback, a number of aspects must be taken into consideration. With 18 and 21 test subjects in each group a reasonable level of confidence in the results is attained. Naturally, increasing the sample size would result in more reliable results. It is however worth noting that SUS in particular is known to provide good results for small sample sizes (5 test subjects is usually considered an acceptable number for early stage evaluations, according to [76]).

The participants’ background is a difficult issue to address. Ideally, the study should have a more diverse collection of test subjects. However, that was not easy to achieve. The fact that a considerable number of participants did not have professional experience, might have affected, for example, their willingness to accept new tools as the time invested in the tools they currently use and the cost of adopting new tools is not overly large. On the contrary, the diversity of computers

used by the participants (e.g. operating systems) might have affected the time measurements performed on some tasks. Ideally all the participant should have had a similar setup, however that was beyond control.

Still on the topic of the variability of conditions, and despite the efforts to avoid it, different sessions were performed at different hours. While some groups performed the tasks in the morning, other performed them the end of the afternoon. The effects on the study are arguable, but since some tasks required focus, the fatigue of the participants could have affected them.

Finally, it can be argued that the provided scenario descriptions were too detailed, and close to the use cases language, making it easy to perform the translation from natural language to use cases. However, the focus at this stage was on use case specification, not requirements analysis, and in order for the study to have a viable size, it was decided to write simple and small statements that the all users could translate in reasonable time. Subjective sentences were introduced however, as much as possible, to give some room for variations in the specifications. Indeed, for the same specification, different participants presented specifications with a different number of lines.

5.5 Summary

This chapter presented the results of two validation studies, performed on the SCARP approach. Overall, the results of the studies answered a set of questions, namely:

- Is RUS expressive enough to handle requirement specifications?
The study results have provided a good feedback regarding the expressiveness of the language.
- Is the proposed knowledge base representation and inference mechanism suitable to support requirement pattern inference?
It was possible to apply the implemented pattern inference mechanism to the provided specifications, and successfully infer requirement patterns.
- Is uCat able to support the specification process?
Study results show that uCat performed well in supporting the specifications, with the tool being used in the studies themselves.
- How is uCat's usability?
The applied SUS questionnaire had a result value of 74, corresponding to a grade B, which is an indicator of good usability.

Chapter 6

Case study

This chapter presents the application of SCARP to an example in eCommerce domain. For each step of the approach, the specific inputs and outputs are presented. SCARP requires two kind of inputs. On the one hand, there are the reusable inputs, part of the process setup. These inputs are composed by the domain model, uQL requirement queries catalog, software patterns catalog, and requirement patterns to software patterns matching information (Sections 6.2 - 6.4). On the other hand, there are the specific inputs for the solution being developed, composed by the use case specifications and types definition (Sections 6.5 - 6.6). This demonstration of SCARP, supported by uCat, has also the purpose of documenting the approach.

6.1 Context

The case study to illustrate the application of SCARP resorts to the eCommerce domain. In order to assess the viability of SCARP, three major groups of features will be addressed. First, the features related with products for sale. Second, the users which interact with the system. Finally, the orders, which represent the users' intentions to buy a product. An eCommerce website (Amazon¹) was used in order to contextualise the case study, and to identify the main entities.

Products One of the core features of eCommerce platforms is presenting and selling products.

Hence, the concept of *product* is required. Figure 6.1 presents the main page of the Amazon website, presenting several products (c.f. **C**). It is possible to see that products can be cataloged by several factors (c.f. **A**), or highlighted (c.f. **B**). The process of placing an *order* is also related with the products.

Accounts There are features related with users, in order to support the shopping process itself.

¹<http://www.amazon.co.uk/>, last visited on 2016-06-23

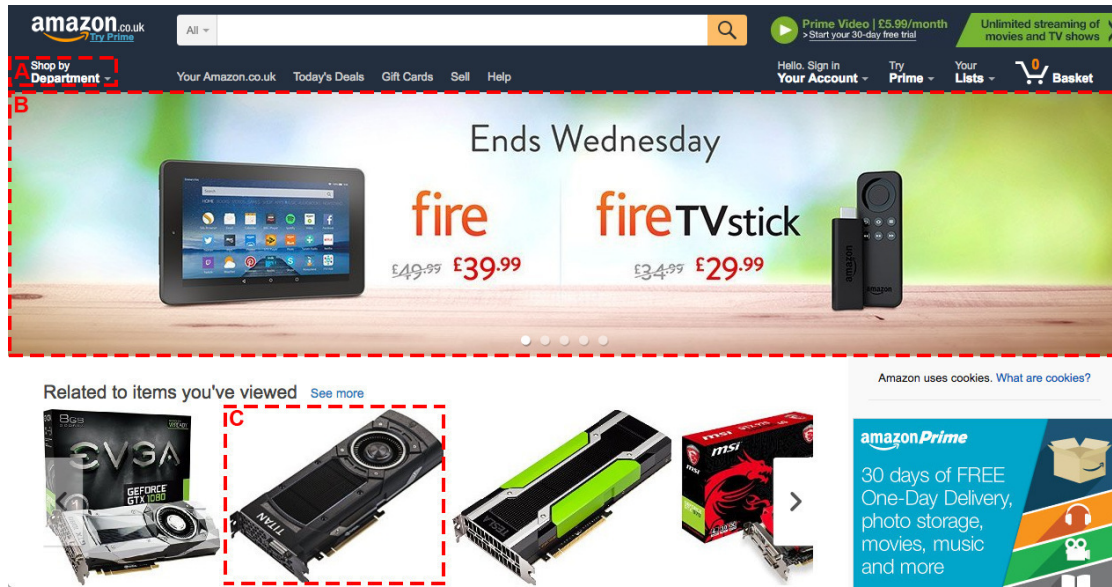


Figure 6.1: Amazon main page.

Figure 6.2 presents the shopping cart page of the Amazon website, where it is possible to see the account related features. Such features include for instance the concepts of user account (c.f. A) and shopping cart (c.f. B).

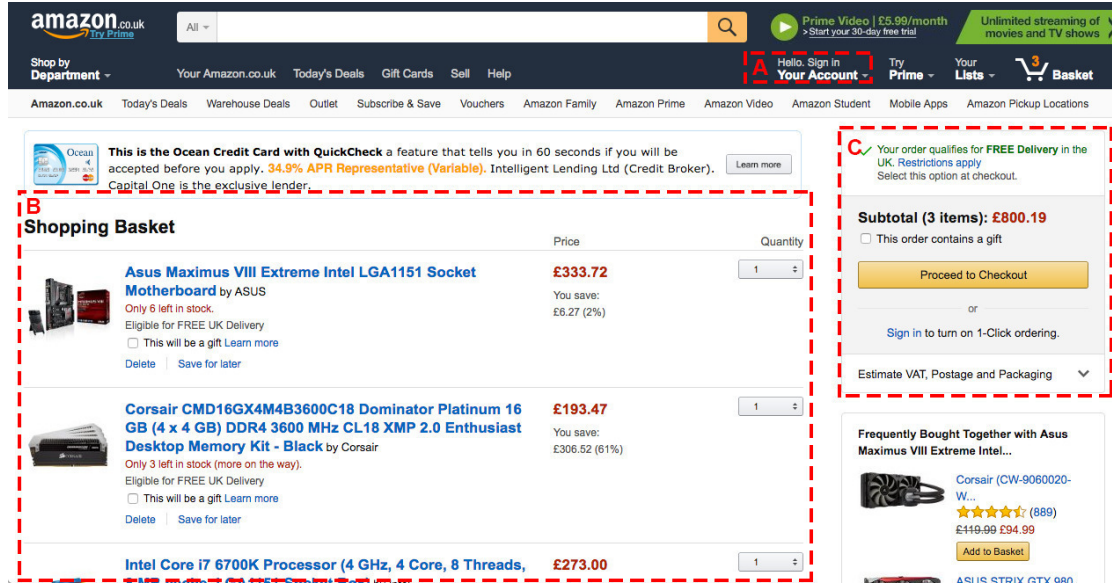


Figure 6.2: Amazon shopping cart page.

Order The process of buying products results in the production of an order. The order cor-

responds to the user request to buy products, which includes the payment and shipping processes. In Figure 6.2 the action to request an order can be seen (c.f. C), “*Proceed to Checkout*”).

6.2 Domain Model

The domain model that was provided to SCARP is specified in Figure 6.4. The diagram shows several entities related with the user, c.f. **User** and **Seller**. Related with the products, the entities **Cart**, **Product** and **Order**, for instance, are defined. There are also entity specific elements, such as the existence of attributes as **Name**, **Address** and **Email**.

Regarding the relations between domain entities, the types for each relation category were specified as follows.

is is contained in *TypeOf*.

contains is contained in *CompositionOf*.

has is contained in *PropertyOf*.

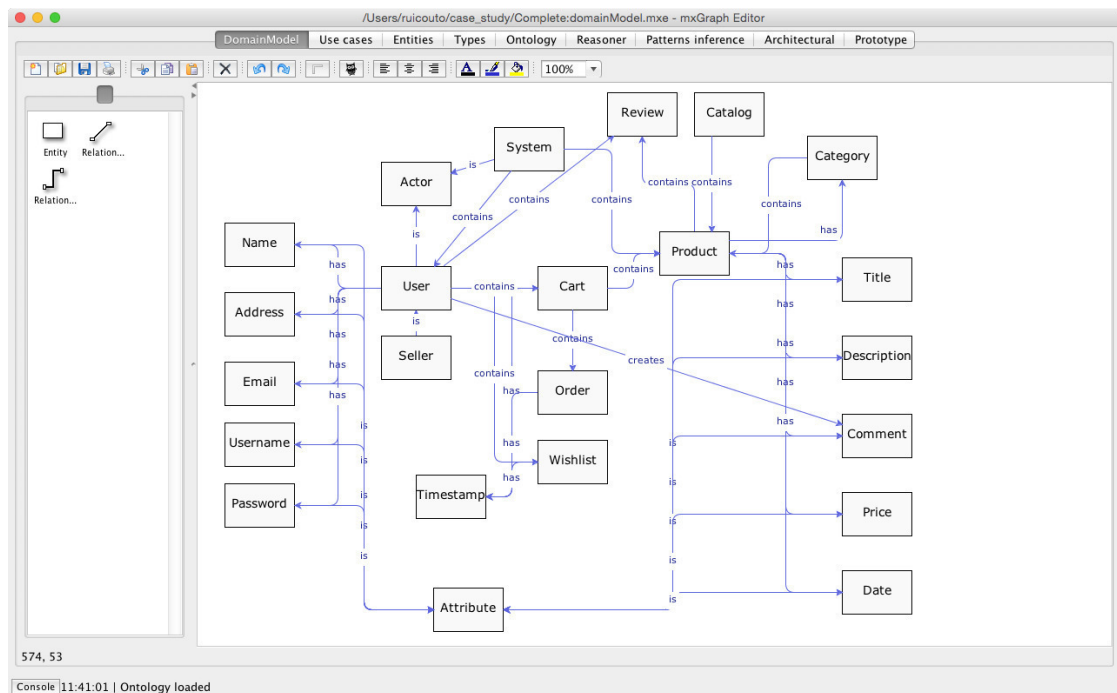


Figure 6.3: Domain model for the eCommerce domain, in uCat.

6.3 uQL Queries

An example of a pattern catalog (as uQL queries) which can be used to support the pattern inference process is described next. The catalog is composed of six patterns, which address different features to be found in the final solution. Presented patterns were extracted from an existing pattern catalog [123], and specified in uQL in order to be supported by SCARP (c.f. Chapter 3). The patterns percentages have been manually adjusted, by putting more emphasis on statements more relevant to the patterns itself.

The first pattern corresponds to support for an account feature, therefore the *HasAccount* requirement pattern. This pattern represents the feature of supporting the representation of the user information, which can be later associated with the user credentials. Its description is presented in Listing 6.1.

```
(user) (requests) (registration|register) 5
(system) (validates) (user) 10
(system) (creates) (?account) 20
(system) (activates) (?account) 20
(user) (requests) (login) 5
(system) (requests) (?credentials) 10
(user) (provides) (?credentials) 10
(system) (verifies|validates) (?credentials) 10
(system) (creates) (session|user_session) 10
```

Listing 6.1: “*HasAccount*” uQL requirement pattern.

The shopping cart feature corresponds to a virtual basket, where the user collects the products to buy. The basket is associated with the corresponding user, and later used to perform the purchase process. The corresponding uQL for the *HasShoppingCart* requirement is presented in Listing 6.2.

```
(user) (selects) (?product) 8
(user) (selects) (?details) 8
(system) (reads|fetches) (?details) 8
(system) (validates) (?details) 8
(system) (adds) (?product) 11
(system) (refreshes) (cart|shopping_cart|basket) 8
(system) (calculates) (?amount) 8
(system) (shows) (?amount) 8
(system) (requests) (?value) 8
(user) (provides) (?value) 8
(system) (validates) (?payment) 8
(system) (posts|request) (shipping) 9
```

Listing 6.2: “*HasShoppingCart*” uQL requirement pattern.

A catalog of products represents a categorization of the products in a system. Hence, products can be represented in different aspects, in order to ease the products’ browsing process. This requirement is represented by the *HasCatalog* pattern, described in Listing 6.3

```
(user) (selects) (?) 20
(system) (loads) (?) 20
(system) (creates) (?result) 25
(system) (shows|presents) (?result) 25
(user) (checks|sees) (?result) 10
```

Listing 6.3: “*HasCatalog*” uQL requirement pattern.

Details of objects correspond the information associated with the corresponding objects. This pattern describes the need for a feature which represents object which need to have some kind of associated information, represented as the *HasDetails* requirement pattern. The pattern is described in Listing 6.4.

```
(user) (selects) (?item) 16
(system) (loads) (?item) 10
(system) (creates) (?result) 18
(system) (processes) (?result) 20
(system) (presents|shows|displays) (?result) 20
(user) (checks|consults|sees) (?item) 16
```

Listing 6.4: “*HasDetails*” uQL requirement pattern.

Search features represent the capability to perform a selection of objects existing on a system, by providing some kind of parameter. This requirement is described as the *HasSearch* requirement pattern, presented in Listing 6.5.

```
(system) (shows) (?) 10
(user) (inputs) (?term) 12
(system) (receives|processes) (?term) 15
(system) (performs) (search|query) 16
(system) (creates|processes) (?result) 16
(system) (presents|shows) (?result) 16
(user) (views|receives) (?result) 15
```

Listing 6.5: “*HasSearch*” uQL requirement pattern.

The requirement to highlight objects in a system by giving them some emphasis is represented as the *HasHighlights* requirement pattern. Listing 6.6 presents the pattern representation.

```
(user) (arrives) (?) 18
(system) (reads) (historic) 21
(system) (loads) (?) 21
(system) (creates) (?highlights) 22
(system) (shows) (?highlights) 18
```

Listing 6.6: “*HasHighlights*” uQL requirement pattern.

Uploading a file to a system is represented by the *HasUpload* requirement pattern. The pattern is presented in Listing 6.7.

```
(user) (select) (?item) 20
(system) (retrieves|fetches) (?item) 20
(system) (?) (permissions) 20
(system) (processes) (?item) 20
(user) (downloads) (?item) 20
```

Listing 6.7: “*HasUpload*” uQL requirement pattern.

Some platforms support the concept of friendship, in order to associate two users. The pattern representing the friendship feature, *HasFriendship*, is presented in Listing 6.8.

```
(user) (selects) (?profile) 18
(system) (loads|opens) (?profile) 16
(system) (presents|displays|shows) (?profile) 16
(user) (requests) (friendship|connection) 18
(system) (creates|sends) (request) 16
(system) (?notifies) (success) 16
```

Listing 6.8: “*HasFriendship*” uQL requirement pattern.

6.4 Software Patterns and Matching Information

Software patterns are inferred from the requirement pattern. This section presents examples of inputs which support the identification of software patterns. The matching process needs matching information, which is also presented in this section. The information presented in this section has been extracted accordingly with the processes described in Chapter 3.

6.4.1 Concerns

A set of concerns must be associated with requirement patterns. In this context, the following ones were defined.

- **Browsable** - represents support for browsing a collection of objects;
- **Editable** - represents support for editing and managing the state of an object (c.f. Section 3.7.1);
- **Manageable** - represents the support to manage access to an object (c.f. Section 3.7.1);
- **Processable** - represents the support to process an object (c.f. Section 3.7.1);
- **Recursive** - represents the capability to aggregate information in a recursive way (c.f. Section 3.7.1);
- **Shareable** - represents the capability to share objects;
- **Viewable** - represents the capability to display the information of an object.

These concerns represent attributes extracted from the requirement patterns interpretation. As consequence, other kinds of concerns could be specified. In order to understand how each of the requirement patterns is related with the extracted concerns, the description of the requirements pattern catalog is presented.

The *HasAccount* pattern corresponds to a user account. In a system, it corresponds to the representation of the client in the system side. As result, a single instance provides a possible approach for its representation. The need for a representation of a single information which is shared through the system justifies the *Shareable* concern.

The *HasShoppingCart* requirement pattern contains three concerns. First, and since a shopping cart represents the items of some user, it must be owned by other class, therefore managed by it, or *Manageable*. Second, the shopping cart aggregates the information of objects with the ultimate goal of producing an order. As consequence, its content should be *Processable*. Finally, objects within the shopping cart must be managed by the cart itself, therefore the *Editable* concern.

The *HasCatalog* pattern corresponds to a catalog, which consists in a set of items needs to be managed by another entity. Such results in the *Manageable* concern. The main objective of the catalog is to provide users means to browse through the items it contains, hence the *Browsable* concern.

The *HasDetails* pattern corresponds to item details, which can be more or less specific. It is usual to find comments or reviews as part of items descriptions. Such information is commonly presented in recursive structures, where it is possible to respond to comments. The *Recursive* concern was found as result.

The *HasSearch* pattern corresponds to the support for objects in a system, and has two consequences. On the one hand, there is the need to generate some output from the search process, hence, the *Manageable* concern. On the other hand, the result must support inspection to create a representation, thus the *Viewable* concern.

The *HasHighlights* pattern is similar to a user account, as the highlights need only a single representation shared through the system, hence, *Shareable*. As is the case with a search feature, the highlights need to be processed in order to create a representation, hence *Viewable*.

Table 6.1 summarizes the presented information about requirement patterns, and corresponding concerns.

6.4.2 Goals

The identification of goals on software patterns is done by analyzing their descriptions. A possible identification of goals on patterns (according to the patterns' descriptions) is as follows.

Proxy pattern is associated with **Delegate goal**. The *Proxy* pattern delegates the control of an object to another object.

Table 6.1: Requirement patterns and corresponding concerns.

Requirement pattern	Concern
HasAccount	Shareable
HasShoppingCart	Manageable
	Processable
	Editable
HasCatalog	Manageable
	Browsable
HasDetails	Recursive
HasSearch	Viewable
	Manageable
HasHighlights	Shareable
	Viewable

Command pattern is associated with **Process goal**. The *Command* pattern supports the parametrization of requests as objects.

Memento pattern is associated with **Edit goal**. The *Memento* pattern keeps the representation of several states, which can be restored while editing an object.

Iterator pattern is associated with **Explore goal**. The *Iterator* pattern provides support to explore efficiently a collection of objects.

Composite pattern is associated with **Compose goal**. The *Composite* pattern supports the hierarchical composition of objects.

Flyweight pattern is associated with **Handle goal**. The *Flyweight* pattern provides a structure to handle a large set of objects.

Singleton pattern is associated with **Unified goal**. The *Singleton* pattern provides support to define a single (hence, unified) instance of a class.

6.4.3 Forces

The final required input to perform the matching process is the definition of forces (see Section 3.7.2). An initial set of forces, inspired in the work by Mairiza [85], was defined as follows.

Abstraction corresponds to the level of abstraction of the structure by providing a simpler interface.

Aggregation indicates the impact on the support of data structures aggregation.

Computability corresponds to providing support for structures which support computable operations.

Constraint indicates the impact of reduction of flexibility in the solution.

Coupling indicates the impact on the association between two elements.

Decoupling separates two structures (e.g. definition from implementation).

Direction corresponds to the integration of two structures by removing intermediary elements.

Efficiency corresponds to a solution which effectively solves the problem in a meaningful way.

Flexibility indicates the existence of a solution which is flexible in terms, for instance, of implementation, and evolution.

Feeding indicates the capability to provide a continuous feed of data.

Filtering corresponds to the capability to support filtering data.

Indirection introduces an indirection via an intermediary element.

Memory corresponds to the impact in the memory of the solution.

Nesting indicates how well the solution supports nested structures.

Overkill corresponds to the application of a complex solution from which only a subset of features is used.

Performance corresponds to a solution which performs a good performance when applied.

Simplicity indicates how simple a solution is in terms of structure.

Versioning corresponds to support of different object versions (or states).

The association of the forces with the corresponding concerns and goals is a manual process. A possible association is as presented in Table 6.2 for concerns, and Table 6.3 for goals. Each force contains a positive (c.f. +) or negative (c.f. -) nature, regarding the concern or goal it belongs to. As an example, *Abstraction* is something required to support the *Manageable* concern, while *Coupling* something to avoid. The nature of the forces in the goals represents how that force contributes to the goal it belongs to. As an example, while the *Explore* goal introduces *Indirection*, the *Unified* goal has the contrary effect.

Having presented the reusable inputs for the approach setup, the following sections present the SCARP process itself.

Table 6.2: Concerns' forces.

Concern	Force
Manageable	+Abstraction
	+Decoupling
	+Indirection
	-Coupling
	-Direction
Processable	+Computability
	+Indirection
	-Coupling
Editable	+Versioning
	+Flexibility
	-Constraint
Browsable	+Feeding
	+Filtering
	+Efficiency
	-Constraint
Recursive	+Indirection
	+Nesting
Viewable	+Simplicity
	+Efficiency
	+Performance
	-Overkill
Shareable	+Memory
	+Efficiency
	+Direction
	-Overkill

Table 6.3: Goals' forces.

Goal	Force
Delegate	+Efficiency
	+Decoupling
	+Abstraction
	-Performance
	-Overkill
Process	+Computability
	+Indirection
	+Coupling
	-Overkill
Edit	+Versioning
	+Flexibility
	-Efficiency
	-Memory
Explore	+Feeding
	+Aggregation
	+Indirection
	+Filtering
	+Performance
Compose	-Direction
	+Indirection
	+Nesting
	+Aggregation
	-Efficiency
	-Simplicity
Handle	-Memory
	+Performance
	+Indirection
	+Efficiency
	-Coupling
Unified	-Overkill
	+Memory
	+Efficiency
	+Direction
	-Indirection

6.5 Requirements

Based in the two described features (or requirements) categories, a set of requirements was defined. In order to comply with SCARP, the requirements were described as use cases, resorting to RUS. The following set of use cases was specified. It is worth mentioning that only requirements for the front-end of the eCommerce application are being considered.

Register Create a user account, by providing the details to identify a user.

Login The capability to perform login into a user account.

Return home At any moment, the user should be able to see the homepage.

Browse products The capability to browse existing products in the platform.

View product A product contains several details, which the user should be able to view.

Show highlights Highlights represent products that for some reason have a bigger emphasis.

View actions history While browsing through products, the user actions should be logged.

Search product Searching products should be possible (e.g. by their name).

Add product to cart With the objective of purchasing a product, the user should have a virtual shopping cart.

Checkout The final step of purchasing a product consists in providing the shipping and payment details, in order to checkout the request.

Figure 6.4 presents the use case diagram containing the described requirements.

6.5.1 Register

The register feature consists in creating a user account. Figure 6.5 presents the use case description in the tabular format. The use case consist in providing a set of details, which are validated and registered by the system. Such results in the creation of the user account.

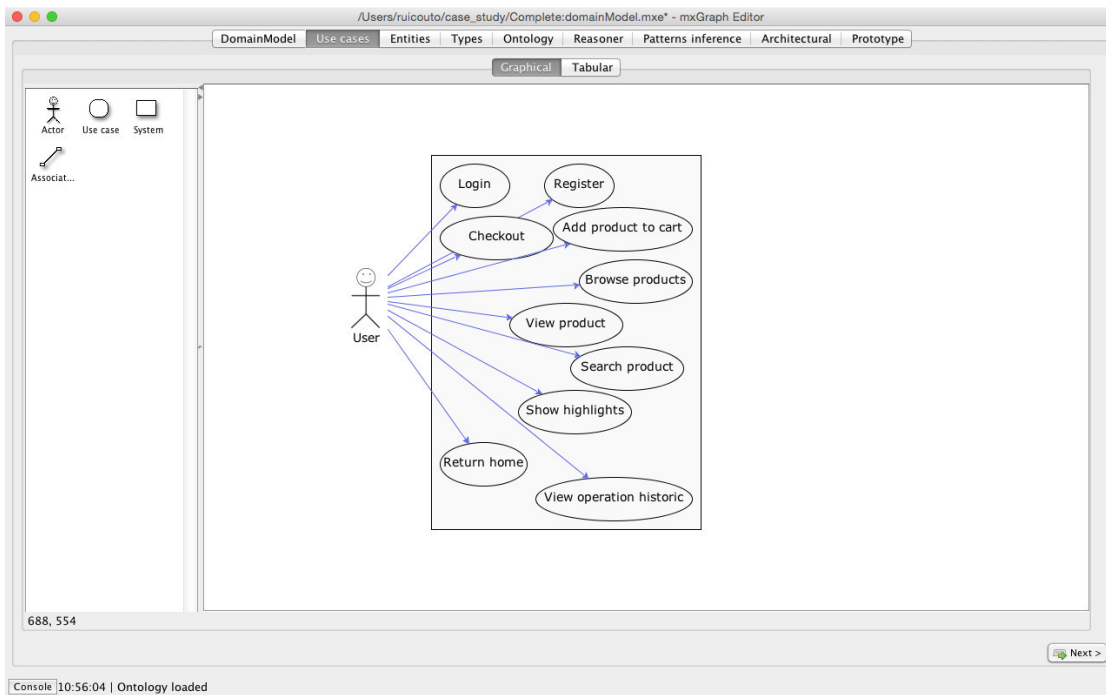


Figure 6.4: Case study's use case diagram.

N	User input	System response
1	user requests registration	
2		system requests username, password, email, name and address
3	user provides username, password, email, name and address	
4		system validates username, password, email, name and address
5		system creates the account
6		system notifies the user
7		system requests email_validation
8	user provides email_validation	
9		system activates account

Figure 6.5: “Register” use case description.

6.5.2 Login

The login feature describes the process of supporting the user authentication. The actions required to perform the login process are presented in Figure 6.6. The process to perform the login consists in providing the login credentials, which are verified against the system information.

Use cases

Name

N	User input	System response
1	user requests login	system requests username and password
2		system requests username and password
3	user provides username and password	system validates username and password
4		system creates a session
5		system creates a session
6		system informs of success

Figure 6.6: “Login” use case description.

6.5.3 Return Home

Since the home page is the starting point for the interactions with the application, the feature to return to the home page is required. At any moment, the user should be able to easily return to the main page, as described in Figure 6.7. Returning to the home page consist just in requesting it. This use case can be represented as an *extend* of other use cases, but the RUS language does not support such specification yet.

Use cases

Name

N	User input	System response
1	user requests homepage	system redirects to homepage
2		system redirects to homepage

Figure 6.7: “Return home” use case description.

6.5.4 Browse Products

Browsing products is the most essential feature in the platform. Figure 6.8 presents the process required to do such. This use case description presents the products organized by categories.

N	User input	System response
1	user selects category	system loads products
2		system creates representation
3		system shows representation
4		
5	user sees products	

Figure 6.8: “Browse product” use case description.

6.5.5 View Product

Being a product characterized by its details (e.g. name, price), it is essential for the users to have the capability to view them. The process of viewing details is as described in Figure 6.9. The steps consist in selecting a product, which the system presents to the user.

N	User input	System response
1	user selects a product	system loads details
2		system loads comments
3		system creates page
4		system shows page
5		
6	user sees product	

Figure 6.9: “View product” use case description.

6.5.6 Show Highlights

Products can be advertised, by giving them a special emphasis. In practice, the products are displayed with a bigger emphasis in some page, when the user enters the platform. The required steps are presented in Figure 6.10. The process to view the highlighted products starts when the user arrives the main page, and the system generates the appropriate information.

The screenshot shows a web application window titled 'Use cases'. At the top, there is a 'Name' field containing 'Show highlights'. Below this is a table with three columns: 'N', 'User input', and 'System response'. The table contains four rows of data. Below the table, there are five buttons: 'Add row', 'Validate', 'Add exception', 'Add alternative', and 'Create'.

N	User input	System response
1	user arrives main_page	
2		system loads highlights
3		system creates banner
4		system shows banner

Figure 6.10: “*Show highlights*” use case description.

6.5.7 View Actions History

The user should have the possibility to view the actions history, as for instance the viewed products during the session. The steps required to view the history are described in Figure 6.11. The process consists in requesting the actions history, which the system will process and shows the user.

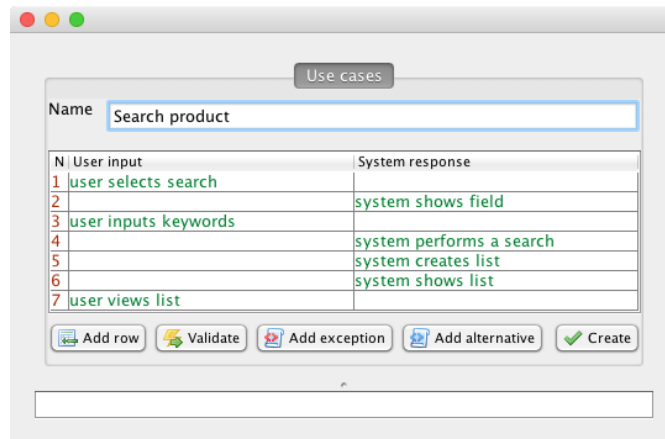
The screenshot shows a web application window titled 'Use cases'. At the top, there is a 'Name' field containing 'View actions history'. Below this is a table with three columns: 'N', 'User input', and 'System response'. The table contains four rows of data. Below the table, there are five buttons: 'Add row', 'Validate', 'Add exception', 'Add alternative', and 'Create'.

N	User input	System response
1	user clicks history	
2		system loads actions
3		system creates list
4		system shows list

Figure 6.11: “*View actions history*” use case description.

6.5.8 Search Product

Searching products is one of the most common features while exploring products in an eCommerce platform. The steps, as described in Figure 6.12, consist in selecting the appropriate feature, and providing the keywords that the system will search.

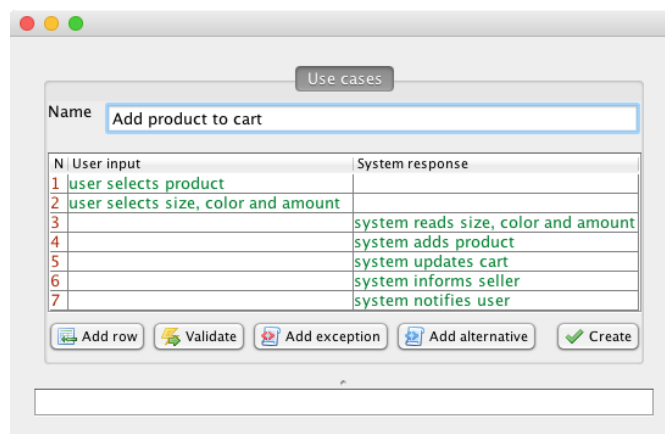


N	User input	System response
1	user selects search	
2		system shows field
3	user inputs keywords	
4		system performs a search
5		system creates list
6		system shows list
7	user views list	

Figure 6.12: “Search product” use case description.

6.5.9 Add Product to Cart

Adding a product to a shopping cart is essential for the buying process. The steps for this process, as described in Figure 6.13, consists in selecting a product and its corresponding properties. The system proceeds to update the user information accordingly.



N	User input	System response
1	user selects product	
2	user selects size, color and amount	
3		system reads size, color and amount
4		system adds product
5		system updates cart
6		system informs seller
7		system notifies user

Figure 6.13: “Add product” to cart use case description.

6.5.10 Checkout

The final step of the buying process consists in the checkout. This process, as described in Figure 6.14, consists in requesting the checkout operation (from the shopping cart), and the providing payment information. The system deals with the payment and shipment request processes.

N	User input	System response
1	user opens the shopping_cart	
2	user selects the checkout	
3		
4		system processes the items
5		system calculates total
6		system shows the total
7		system requests payment
8	user provides the payment	
9		system validates payment
10		system requests shipping system informs user

Figure 6.14: “Checkout” use case description.

6.6 Individuals Classification

The specification of the types for each individual is required to generate the ontology. By analyzing each one of them it is possible to define a set of types, which are able to aggregate the entities. The process of defining new types depends on the user’s interpretation, however types can later be adjusted. The initial set of types defined for the entities is as follows.

Input Describes any entity that supports interaction with the user (e.g. text field, input value).

Action Describes an entity that represents an action to be performed in the system.

Object Describes a generic object.

Process Describes an entity representing, or related to, a process.

Result Describes an entity that represent a resulting output from a process.

Table 6.4: Types specification.

Individual	Type	Individual	Type
seller	<i>User</i>	amount	Input, Attribute
color	Attribute	comments	<i>Attribute</i>
keywords	Input	address	<i>Attribute</i>
session	Object	banner	Object
email_validation	Process	history	Object
login	Process	list	Object
representation	Object	system	Actor
cart	Object	highlights	Object
products	Object	size	Attribute
total	Result	field	Input
password	<i>Attribute</i>	success	Result
search	Action	name	<i>Attribute</i>
main_page	Object	registration	Process
shipping	Process	page	Object
payment	Input	category	Attribute
details	Input	user	<i>Actor</i>
checkout	Process	items	Object
email	<i>Attribute</i>	actions	Action
shopping_cart	Object	account	Object
product	Object	username	<i>Attribute</i>
homepage	Object		

Table 6.4 presents the association of entities to types. The types for `seller`, `password`, `email`, `comments`, `address`, `system`, `name` and `username` entities were automatically extracted from the domain model, and are presented in italic font.

6.7 Requirement Patterns Inference

The requirement patterns catalog was applied to the ontology representing the requirements, in order to infer existing patterns. As a result, a matching percentage was obtained for each pattern.

The matching results is as presented in Figure 6.15. The figure presents for each requirement pattern, how likely is it to exist in the requirements specification.

In order to filter which patterns really exist in the specification, a true positives match percentage threshold is required. Empirical results (c.f. Chapter 5) have shown that matching percentages above 50% might be a good indication of true positives. In this case, from all patterns, only

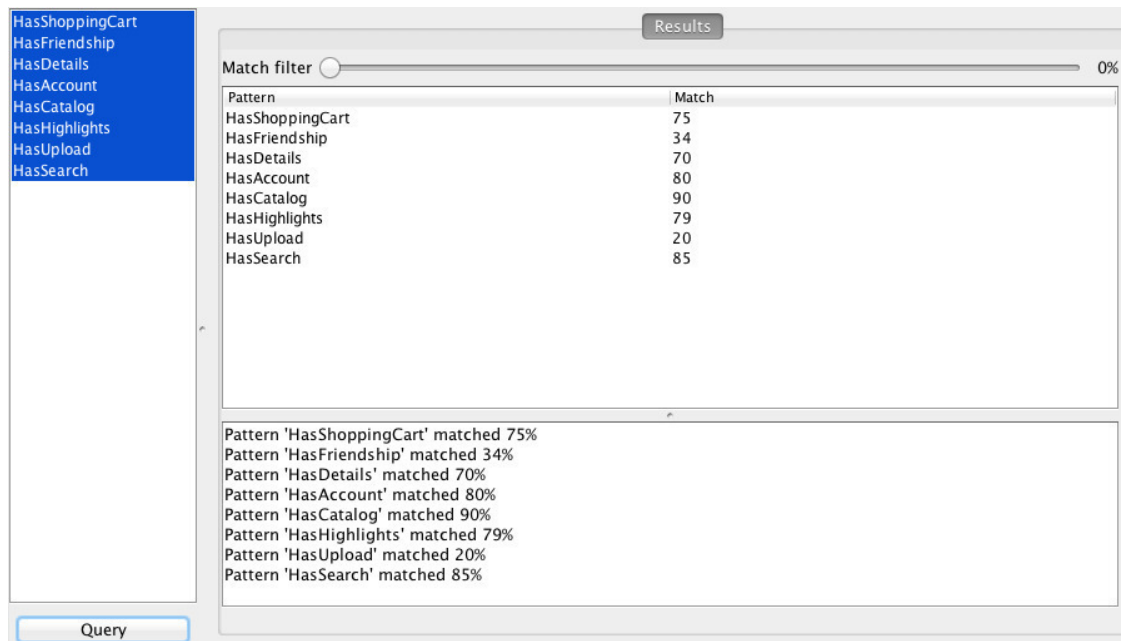


Figure 6.15: Requirement pattern inference result.

HasUpload and HasFriendship have a matching value under 50% (20% and 34%, respectively). The remaining patterns have a matching value over 50%, therefore these six patterns exist in the specification. Indeed, for the accepted patterns, the minimum match value corresponds to 70% for the *HasDetails* requirement pattern.

6.7.1 Inferred Patterns

Based in the provided specifications of concerns, goals and forces, producing the forces matrix is possible, which depicts the relation of these elements. Figure 6.16 shows the resulting forces matrix. In the matrix is highlighted, for each concern, the most relevant contribution, which defines the most appropriate goal.

In order to create the forces matrix, each concern is compared against each goal, regarding their forces. For instance, the result of matching the *Processable* concern against the *Process* goal has two positive and one negative matches (c.f. (2,-1)). The positive matches correspond to *Computability* and *Indirection*, which have a positive nature in both the concern and the goal. The negative match corresponds to the *Coupling* force, which while unwanted by *Processable*, is provided by *Process*.

Figure 6.17 summarizes the matching result information, associating a software pattern to each requirement pattern. The resulting set correspond to the software patterns required to instantiate.

Forces matrix							
C \ G	Unified	Delegate	Edit	Explore	Handle	Compose	Process
Processable	(+0,-0)	(+0,-0)	(+0,-0)	(+1,-0)	(+1,-0)	(+1,-0)	(+2,-1)
Editable	(+0,-0)	(+0,-0)	(+2,-0)	(+0,-0)	(+0,-0)	(+0,-0)	(+0,-0)
Manageable	(+0,-0)	(+2,-0)	(+0,-0)	(+1,-1)	(+1,-1)	(+1,-1)	(+1,-2)
Shareable	(+3,-0)	(+1,-0)	(+0,-0)	(+0,-0)	(+1,-0)	(+0,-0)	(+0,-0)
Browseable	(+1,-0)	(+1,-0)	(+0,-0)	(+2,-0)	(+1,-0)	(+0,-0)	(+0,-0)
Viewable	(+1,-0)	(+1,-0)	(+0,-0)	(+1,-0)	(+2,-0)	(+0,-0)	(+0,-0)
Recursive	(+0,-0)	(+0,-0)	(+0,-0)	(+1,-0)	(+1,-0)	(+2,-0)	(+1,-0)

Figure 6.16: Resulting forces matrix.

Matching Result	
Manageable	-> Delegate: Proxy
Processable	-> Process: Command
Editable	-> Edit: Memento
Recursive	-> Compose: Composite
Shareable	-> Unified: Singleton
Browseable	-> Explore: Iterator
Viewable	-> Handle: Flyweight

Figure 6.17: Resulting inferred software patterns, with best matches highlighted.

6.8 Produced Solution

Generating the final solution in SCARP requires the prior definition of the parameters for classes. After the parametrization, a XMI solution is produced.

6.8.1 Parametrization

In order to generate the final solution, the specific names for the software patterns' elements need to be provided. For instance, the *Proxy* software pattern is composed by the `proxy`, `subject`, `client` and `realSubject`, as presented in Figure 6.18. uCat provides alongside with the context in which the pattern is being applied (i.e., the *HasShoppingCart* requirement pattern), the objective of the software pattern. Each parameter is also described regarding its role in the solution.

HasShoppingCart(): Provide a surrogate or placeholder for another object to control access to it.	
proxy - Interface which handles the request.	User
subject - Defines the common interface for RealSubject and Proxy.	ICart
client - Who triggers the request.	System
realSubject - Defines the real object that the proxy represents.	Cart

Figure 6.18: Parametrization of the *Proxy* pattern, for the *HasShoppingCart* requirement pattern.

The parametrization consists in setting specific names for each field. In this case, the **proxy** element is the **User**, since the user will handle the shopping cart, therefore the associated requests. The **subject**, being the interface for the cart, is the **ICart**. Who triggers requests in the shopping cart (e.g. add and remove products) is the **system** itself. Finally, the **realSubject**, which represents the proxy is the shopping **Cart**. This process was applied to all the parameters of the inferred software patterns, as shown in Figure 6.19².

The properties inferred from the domain model require their types definition. Hence, a possible parametrization is as shown in Figure 6.20. In the figure is possible to see, for each class and each attribute, an input field to specify the types. For the **Product** class, the type **String** was defined for the **Category**, **Title**, **Description** and **Comment**, since all these properties can be described textually. For the **Date**, a specific Java type was selected, the **LocalData**. Finally, the **Price**, being a numerical value, is represented by a **double**.

²In order to avoid reducing the font size to an unreadable size, some pattern descriptions are cropped on the right.

<p>HasShoppingCart(): Provide a surrogate or placeholder for another object to control access to it.</p> <p>proxy - Interface which handles the request. <code>User</code></p> <p>subject - Defines the common interface for RealSubject and Proxy. <code>ICart</code></p> <p>client - Who triggers the request. <code>System</code></p> <p>realSubject - Defines the real object that the proxy represents. <code>Cart</code></p>
<p>HasShoppingCart(): Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored</p> <p>item - The items belonging to the state. <code>Product</code></p> <p>caretaker - The responsible for the memento's safekeeping. <code>CartCaretaker</code></p> <p>memento - Stores internal state of the Originator object. <code>CartMemento</code></p> <p>originator - Who triggers the request for creating states. <code>Cart</code></p> <p>state - The representation of the state being keep. <code>CartState</code></p>
<p>HasShoppingCart(): Encapsulate a request as an object, thereby letting you parameterize clients with different requests , queue or log re</p> <p>receiver - Knows how to perform the operations associated with carrying out. <code>Cart</code></p> <p>client - Who triggers the operation. <code>User</code></p> <p>concreteCommand - Implements Execute by invoking the corresponding operation on Receiver. <code>Process</code></p> <p>invoker - Asks the command to carry out the request. <code>User</code></p> <p>command - Declares an interface for executing an operation. <code>IProcess</code></p>
<p>HasDetails(): Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects a</p> <p>component - defines the interface for the composed objects. <code>IDetails</code></p> <p>composite - represents an element with children of the same type. <code>Comment</code></p> <p>client - manipulates objects in the composition through the Component interface. <code>Product</code></p> <p>leaf - represents a leaf in the composition, without children. <code>FinalComment</code></p>
<p>HasAccount(): Ensure a class only has one instance, and provide a global point of access to it.</p> <p>singleton - creates and encapsulate the single object instance. <code>User</code></p>
<p>HasCatalog(): Provide a surrogate or placeholder for another object to control access to it.</p> <p>proxy - Interface which handles the request. <code>CatalogProxy</code></p> <p>subject - Defines the common interface for RealSubject and Proxy. <code>ICatalog</code></p> <p>client - Who triggers the request. <code>System</code></p> <p>realSubject - Defines the real object that the proxy represents. <code>Catalog</code></p>
<p>HasCatalog(): Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.</p> <p>iterator - defines the interface to access the and traverse elements. <code>ProductIterator</code></p> <p>concreteAggregate - implements the aggregate interface. <code>ProductAggregate</code></p> <p>concreteIterator - implements the iterator. <code>ConcreteProductIterator</code></p> <p>client - requests the iterator. <code>Catalog</code></p> <p>aggregate - defines the interface for creating iterator objects. <code>IProductAggregate</code></p>
<p>HasHighlights(): Ensure a class only has one instance, and provide a global point of access to it.</p> <p>singleton - creates and encapsulate the single object instance. <code>Highlights</code></p>
<p>HasHighlights(): Use sharing to support large numbers of fine-grained objects efficiently.</p> <p>concreteFlyweight - Extension of the concrete instances. <code>Product</code></p> <p>client - Who triggers the request. <code>Highlights</code></p> <p>flyweightFactory - <code>ProductFlyweightFactory</code></p> <p>flyweight - The class of the instances to be managed. <code>IProduct</code></p>
<p>HasSearch(): Provide a surrogate or placeholder for another object to control access to it.</p> <p>proxy - Interface which handles the request. <code>ProductProxy</code></p> <p>subject - Defines the common interface for RealSubject and Proxy. <code>ProductSearch</code></p> <p>client - Who triggers the request. <code>System</code></p> <p>realSubject - Defines the real object that the proxy represents. <code>ProductFlyweightFactory</code></p>
<p>HasSearch(): Use sharing to support large numbers of fine-grained objects efficiently.</p> <p>concreteFlyweight - Extension of the concrete instances. <code>Product</code></p> <p>client - Who triggers the request. <code>SearchProxy</code></p> <p>flyweightFactory - <code>ProductFlyweightFactory</code></p> <p>flyweight - The class of the instances to be managed. <code>IProduct</code></p>

Figure 6.19: Parametrization of all software patterns.

Order	
Timestamp	java.time.LocalDate
User	
Password	String
Username	String
Address	String
Name	String
Email	String
Product	
Category	String
Title	String
Description	String
Date	java.time.LocalDate
Price	double
Comment	String
Wishlist	
Timestamp	java.time.LocalDate
Cart	
Timestamp	java.time.LocalDate

Figure 6.20: Types for the inferred properties.

6.8.2 Solution

The XMI representation resulting from the serialization process was imported into the ArgoUML tool. After minor adjustments, the resulting diagram is as presented in Figure 6.22. The figure corresponds to a UML class diagram, resulting from the software pattern instantiation and composition process. The figure depicts the resulting classes and interfaces (e.g. `User`, `Cart`, `Product`), as well as the respective relationships. The figure highlights also the set of classes which support each requirement pattern. For instance, the *HasDetails* requirement pattern is supported by the `IDetails` interface, `Comment` and `FinalComment` classes. The figure successfully shows the existence of architectural structures to support each requirement pattern, in the form of a software pattern. The final solution is composed of the inferred requirement patterns and the supporting software patterns, parametrized according to the aforementioned values.

A preliminary study was applied to the resulting architecture. Five participants, all Software Engineers with a master's degree, were asked to compare the resulting architecture (c.f. Figure 6.22) against other five architectural models, manually specified by other participants as part of another study (see Appendix B.6 for the remaining diagrams). The participants answered a questionnaire, in a Likert scale, to perform the evaluation of each model, as presented.

1. The model is incomplete.
2. I would add classes to the model.
3. I would add relations to the model.
4. I would use this model as a starting point for the development of the eCommerce business layer.
5. The model is too complicated.

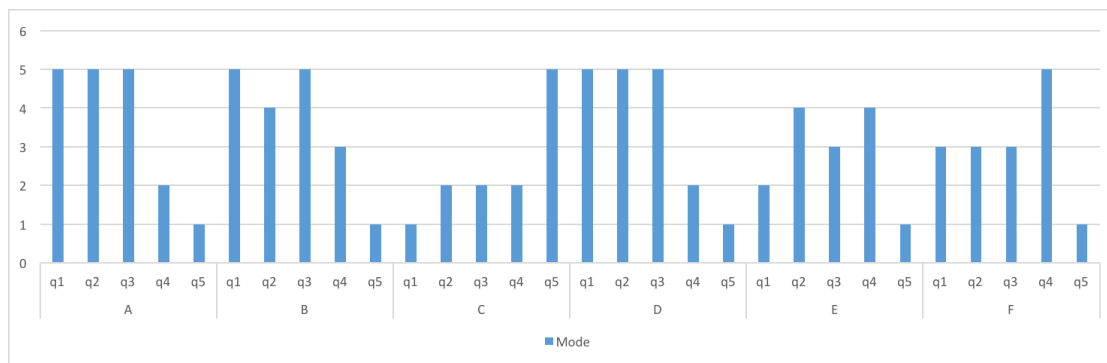


Figure 6.21: Results of the validation study performed in the generated model.

The results are presented in Figure 6.21, where **C** corresponds to the model generated by SCARP. In questions **1** and **2**, the model **C** scored the lowest values. Such means that all other diagrams need more classes and relationships than **C**, therefore, it is the most complete. In question **3**,

the participants have shown preference for other models, as is the case of A, B or D. Such is related with the following question, **5**, in which the participants consider the diagram C the most complex diagram. When asked why they prefer other models, the participants stated that a longer analysis process would be needed to understand the model, while acknowledging that the diagram is the most complete. However, they state also that with the other models they have more flexibility, since the models are simpler. It is worth noting that the participants did not take part of the SCARP parametrization process.

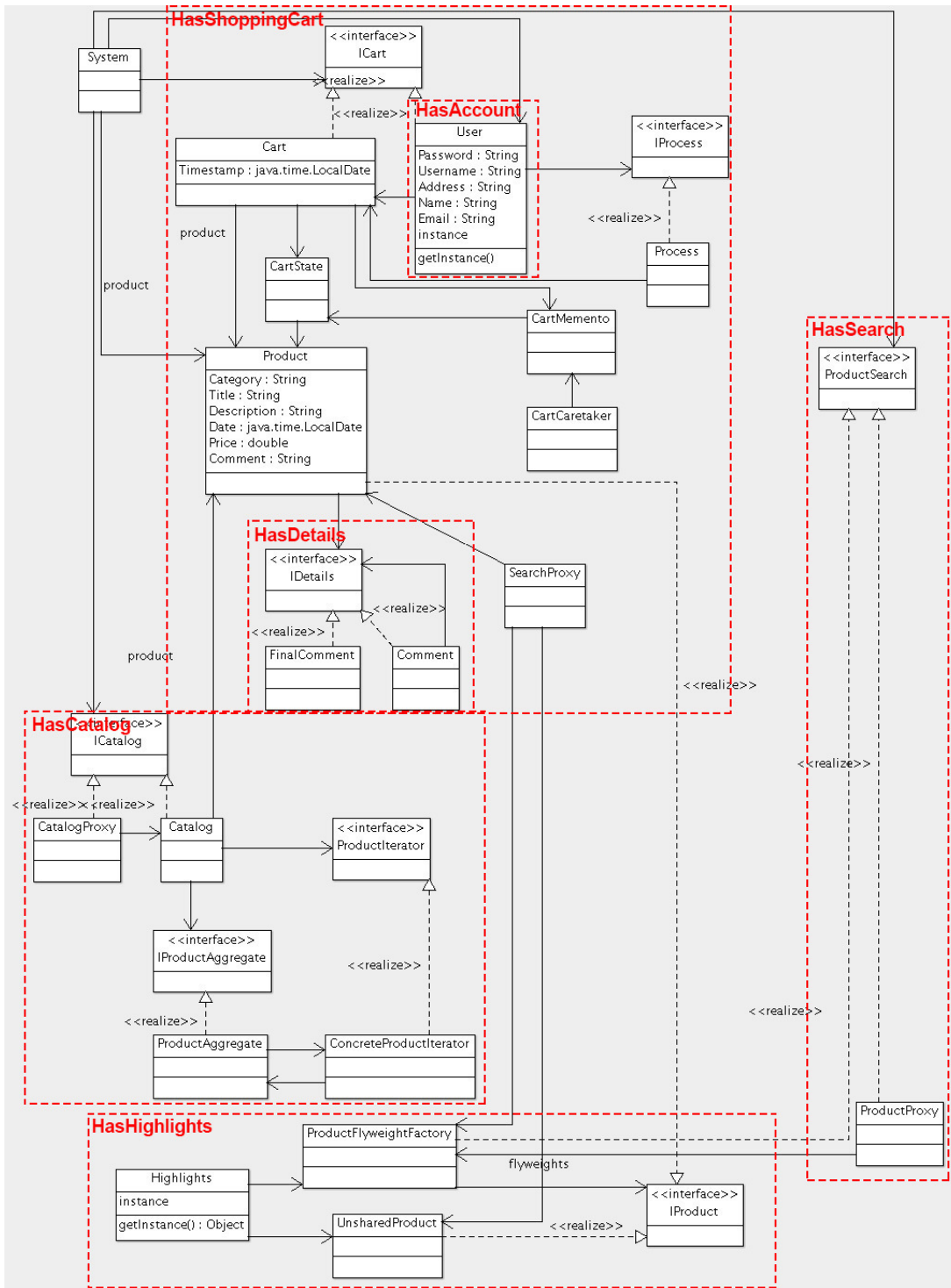


Figure 6.22: Resulting architecture from the serialization process, with identification of requirement patterns.

6.9 Discussion

After the application of this eCommerce case study, there are four considerations to be made. First, the domain model is an artifact that should be detailed and refined. On the one hand, the domain model has impact on the final solution, regarding automatic types inference, properties and attributes information extraction. Hence, it ultimately affects the produced solution. On the other hand, the domain model can be reused for other projects in SCARP. In consequence, all the effort put in refining it is reused for other projects.

Second, there is a certain degree of subjectivity in the identification of the concerns and associated forces. Since this step relies on the interpretation of pattern descriptions, different users will have different interpretations. As a consequence, different requirement and software patterns might be identified as part of the final solution. The intent of SCARP is not the generation of final architectures, but instead presenting a first version to be evolved. Resulting solutions will be influenced by the decisions made during the process itself, which enables the possibility to parametrize and adjust the solution according to the users' needs. The possibility to parametrize SCARP introduces flexibility in the approach.

Third, the results of the requirement pattern to software pattern matching process depends also on users' interpretation. Defining a matching percentage to consider that the pattern really exists in the specification (i.e., is not a false positive) will affect the resulting patterns. On the one hand, reducing the percentage, leads to a larger set of patterns, but increases also uncertainty. On the other hand, increasing the value, while raising certainty, reduces the number of patterns. Defining the adequate percentage level is a process which might require some tuning before proceeding with the process, despite the empirical results indications (c.f. Section 5). This parametrization enables the possibility to explore different architectures, in order to iterate the produces solutions.

Fourth, the instantiation process is also dependent on users' interpretations, and directly affects the resulting architecture, but expertise and reuse could improve the process.

Regarding these considerations, three conclusions can be made. First, there is some degree of subjectivity involved in the process to take in account, regarding the inputs parametrization. While introducing variability in the solutions, the parametrization enables flexibility in and customization of the process. Second, it is worth mentioning that for the presented case study, the requested features are present in the final solution in the form of software patterns. The transition from the requirement specification to architectural artifacts is one of SCARP objectives, and it was demonstrated in this case study. Third, SCARP is an iterative process. When a generated architecture contains errors, the used inputs should be refined and adjusted in order to solve these errors, and a new architecture should then be produced. This process results in architectures which became more refined in each step, this iterative model derives in one final architecture.

Regarding this case study, a set of configuration inputs was specified. It was possible to successfully define a set of use cases, in which several requirement patterns were identified, with

different matching levels. Such illustrates the viability of producing an ontology to represent requirements, which can also be queried to analyze the formalized information. A set of parameters to support the matching process (c.f. concerns, goals and forces) was defined. The inputs provided successfully supported the transition from requirement to software patterns. The parameters defined to support the software pattern instantiation process successfully lead to an initial architecture. It was also possible to achieve a set of architectural artifacts, corresponding to the initial set of requirements. The artifacts are represented as a unified solution, in a class diagram.

SCARP supports the generation of software models, from requirements' specifications. The produced models are initial solutions which should be iterated. From this step, a MDA process can proceed, in order to generate several kind of outputs. Specifically, the production of source code regarding the structural aspects is a viable approach. The behavioral aspects have not been addressed as part of this work, but the possibility to generate such outputs from the use cases can be explored. However, the generation of prototype user interfaces was explored, with MODUS.

MODUS defines a process to generate user interface prototypes from UML class diagrams. MODUS takes advantage from the domain informations, and by analyzing the entities and relationships, is able to generate webpages relevant for the specified models. MODUS uses as input a class diagram (as produced by SCARP), and was also applied to the eCommerce domain. Hence, in order to generate the user interface prototype, the output of SCARP was provided as input for MODUS. Figure 6.23 presents the resulting interface, in the form of a web page. In the figure is possible to see, among others, the created pages for *Shopping cart*, *User account*, *Products* and *Categories*.

Figure 6.24 presents a listing of products, where it is possible to see the product names and prices. These attributes in particular were extracted from the domain model (in Figure 6.4), and associated with the corresponding `Product` entity, which resulted in a class representing such information.

As a final remark, it is worth mentioning that the relevance and impact of the refinement process, and, impact of the subjectivity (regarding some of the inputs) need to be properly evaluated. Despite the success of the preliminary study on the quality of the solution, a more extensive study on the produced solution should be performed. This validation process is out of this work's scope, therefore left for future work.

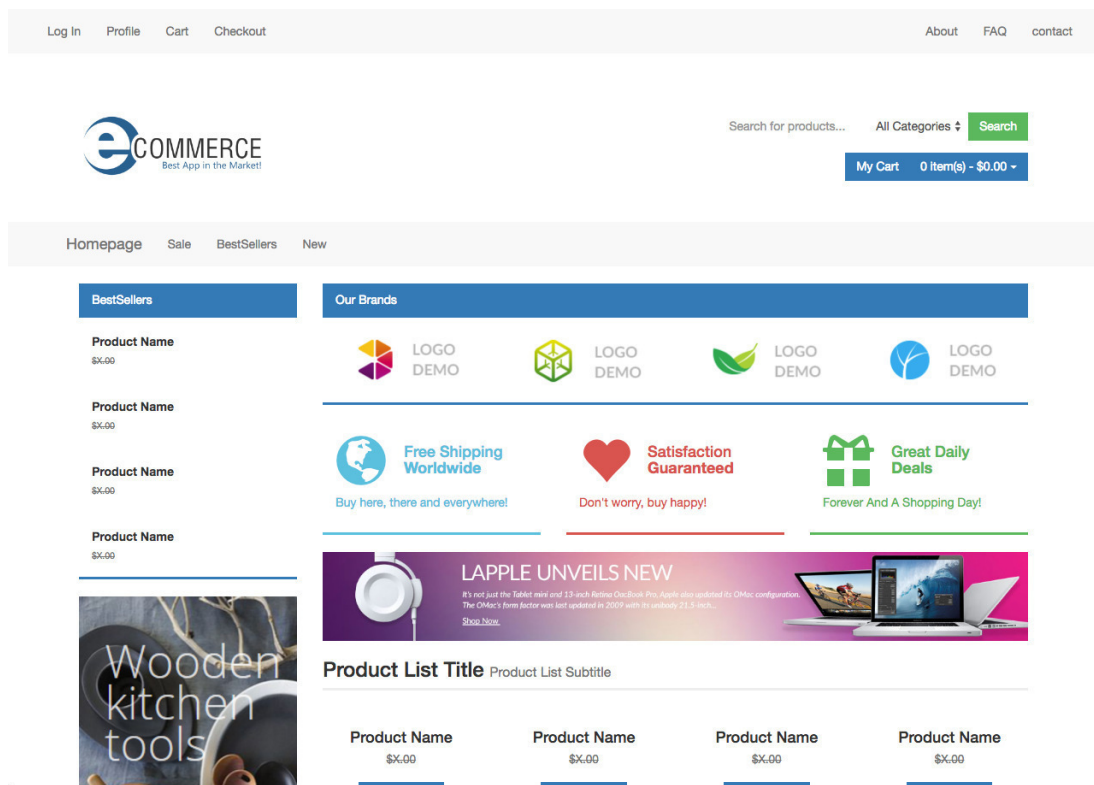


Figure 6.23: Main page of the user interface prototype.

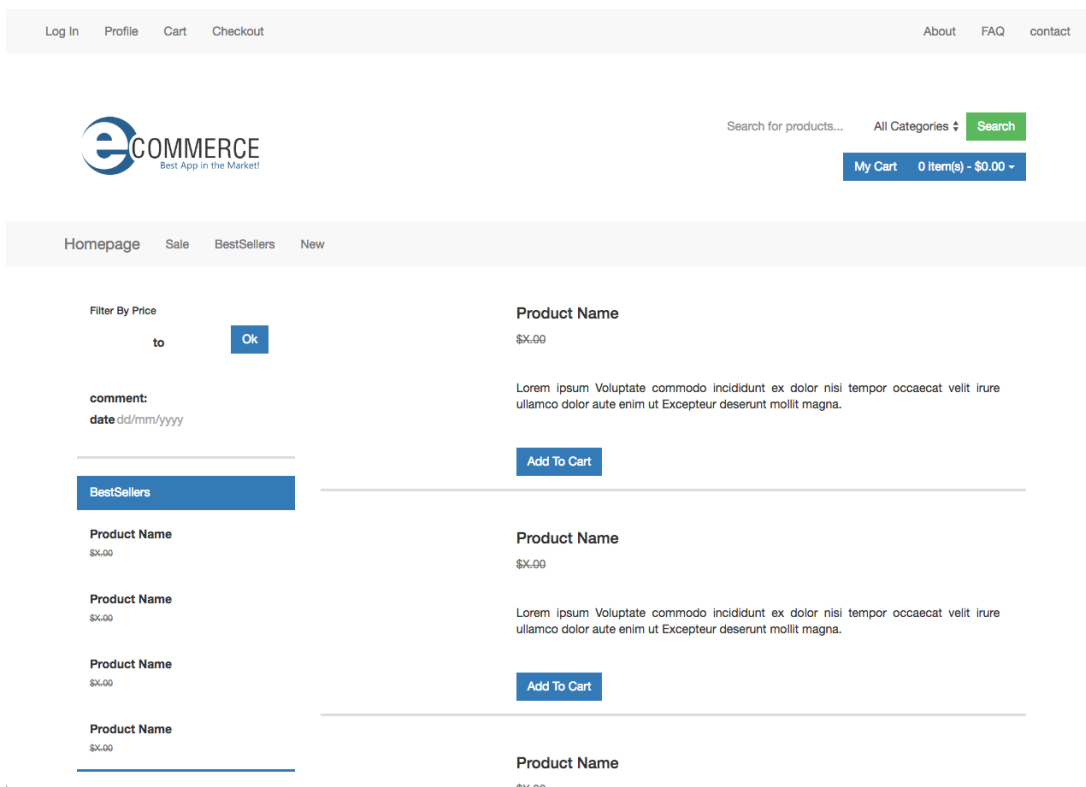


Figure 6.24: Listing of products in the user interface prototype.

6.10 Summary

This chapter provides support for arguing about the applicability of the SCARP approach with a specific example. Indeed, resorting to the approach, it was possible to achieve an architectural prototype, starting from a set of use cases. The inputs provided to support the approach were also presented (c.f. Figure 3.1). The final result was a prototype architecture, build by composing software patterns, in order to have a better solution. The generated solution is a model, and it can be used as input for other model based approaches. As an example it was supplied as input to the MODUS approach. This resulted in a user interface that supports the produced architectural model.

Chapter 7

Conclusions

Different approaches support the software development process. As an example, linear approaches support development via a sequential process, and iterative approaches present incremental processes. The Model Driven Architecture (MDA) is a software development process based on models. In the MDA, models are defined and transformed into other models, and ultimately into source code.

The usage of models makes the MDA an interesting approach. On the one hand, models introduce formalism in the software development process. Defining software as models instead of informal formats, make the process more rigorous, and enables the possibility to apply automated analysis techniques. On the other hand, tools developed around the MDA support the (semi) automatic transformation of these models, improving the software development process.

Architectural models are successfully used as support for the software development process in the context of the MDA. Different tools and techniques support their analysis and transformation. Requirement models, however, do not usually have such kind of support for automation. As result, requirement models are not integrated as part of the MDA in the same way as software models.

The transformation of requirement models into architectural models depends on manual processes, in order to support the integration in the MDA. Non automated processes are known to introduce subjectivity in the translation process, and open margin for interpretation errors. Automation of the transformation of requirements to architectural model is a plausible approach to mitigate such errors.

This work presented the SCenario bAsed Rapid software Prototyping (SCARP) approach, to support and improve the integration of requirement models in the MDA process. SCARP defines a process composed of several steps, in order to automate those transformations. The process starts with the formalization of requirements (as models), in a language emphasizing both automation and readability. The emphasis on readability supports the definition of a language

both understood by actors (which specify the requirements), and developers (which interpret the specifications). The emphasis in systematization supports the semi-automatic transformation of requirement into requirement models, with support for information analysis. The analysis capabilities support information extraction, such as requirement patterns. These patterns are used to extract architectural hints, specifically, their association with software patterns. Software patterns are instantiated and composed in order to create an architectural solution. To support automate this process a tool, Use Cases Analysis Tool (uCat), was developed. The tool provides features and interfaces to support all the SCARP steps, namely the requirements specification, requirement pattern inference and architectural pattern matching, as well as the parametrization and production of architectural solutions.

With SCARP it is possible to automate the integration of requirement models into the MDA process. Requirement models are automatically analyzed and formalized, which reduces the impact of interpretation errors. Furthermore, the formalization language provides a unified format to specify requirements. Simplifying the resulting specifications, also make them easier to understand. The transformation from requirements specifications, to requirement patterns, and consequently to software pattern is also automated in SCARP. The models resulting from this process are represented in standard formats, fostering the interoperability of SCARP with other tools. It is worth noting that the presented approach was defined as a comprehensive process, composed of several steps. In order to perform some of the steps, existing MDA tools and techniques were adopted. uCat was developed to fill in the gaps.

Besides developing SCARP and uCat, two studies were performed in the context of this work, in order to validate the acceptance of applying SCARP (and uCat) to the software development process. The first study focused in the expressiveness of the specification language, and in uCat's support of the specification process. The results of the study indicate that the language is suitable and has a good acceptance by the users. uCat provided also an appropriate support for the process. In the second study the viability of the requirement pattern inference process was validated. The results indicate that it is possible to perform pattern inference in requirement specifications. Furthermore, the patterns are extracted with an associated matching ratio, and such helps developers to select which patterns to apply. Finally, the usability of the tool was also addressed using the System Usability Scale (SUS), scoring a value of 74 which corresponds to a B grade.

This work details also how to put SCARP in practice, by describing the required inputs for each step of the approach. The outputs resulting from each step and corresponding contribution to the process were also presented. A specific instantiation of SCARP is presented, though a case study, in the eCommerce domain. A requirements specification model was created, from which a set of requirement patterns were extracted. Software patterns were inferred from the requirement patterns, instantiated and composed, leading to an architectural model. The case study supports both the viability of the approach and documents how SCARP can be applied.

Several contributions result from this work. In specific, the SCARP approach and uCat tool were implemented, as well as a set of scientific publications. The list of contributions is as follows.

SCARP is the envisaged approach, consisting in a process which integrates requirement models into the MDA process, by automating the transition of requirement models to architectural models. The definition of SCARP is the major contribution of this work.

uCat is the tool supporting SCARP. The tool provides support for the requirements specification, and the following steps in order to produce architectural models.

“A study on the viability of formalizing use cases.” corresponds to a scientific publication, which presents the study performed to validate SCARP, as well as a set of preliminary results. This publication was relevant to support the work in Chapter 3.4.

“Validating an approach to formalize use cases with ontologies.” corresponds to a publication, which presents the results of the SCARP validation study, performed with uCat. This publication was relevant for the work presented in Chapter 3.4 and 5.

“Application of ontologies in identifying requirements patterns in use cases.” corresponds to a publication, which describes an early vision of SCARP, namely regarding the requirements specification, and proposal for pattern inference. This publication (as the previous one) further supports the work in Chapter 3.6.

“A survey on software patterns.” corresponds to a survey, which categorizes software patterns at different levels of abstraction. This categorization was useful to support the work described in Chapters 3.7 and 4.

The following publications concern indirect support of the presented work, and were produced during the project.

“A patterns based reverse engineering approach for Java source code.” corresponds to a publication, which presents an approach to perform software pattern inference from Java source code. This work was relevant in order to support the requirement pattern inference process, performed in SCARP, as described in Chapter 3.6.

“MapIt: A model based pattern recovery tool.” corresponds to a publication, which describes the process of software pattern inference from source code (as the previous one). This publication further supports the work described in Chapter 3.6.

“Modus: uma metodologia de prototipagem de interfaces baseada em modelos.” corresponds to a publication, which described Model-based Developed User Systems (MODUS), an approach to produce user interface prototypes from a architectural model (namely, class diagrams). This work was useful in order to test the expressiveness of the architectural models produced in SCARP, in support for the work described in Chapters 6 and 4.

“*The modelery: a model-based software development repository*” corresponds to a publication, which describes Models Refinery (Modelery), an online repository of software artifacts (as architectural models). uCat supports the interaction with Modelery in order to store and retrieve models from the repository. This publication was relevant for the work described in Chapter 6.

“*The modelery: A collaborative web based repository.*” corresponds to a publication, which further described Modelery. This publication was relevant to support the work described in Chapter 6.

7.1 Discussion

This work has presented SCARP, an approach to support the integration of requirement models in the MDA process. SCARP resorts to the requirements formalization in order to produce requirement models, and support their operationalization. uCat, the supporting tool, was also presented. The viability of the approach was illustrated by a case study, and supported by the performed studies.

The implementation of SCARP was made possible by the integration of different approaches and technologies. The formalization of the requirements was possible resorting to a Controlled Natural Language (CNL), which is itself a computable format. The computable format which supports the requirements representation is a knowledge base (specifically, an ontology), which provides the capability to structure and query knowledge. Finally, both requirement and software patterns play an important role in the presented approach. Both their identification and instantiation is essential to achieve software solutions.

SCARP succeeds in presenting an approach to formalize requirements and supports their automatic transformation into architectural models. Furthermore, the approach consists in a parametrized process, which provides the capability to customize the process, in order to adjust the produced solutions.

The existence of limitations in SCARP is acknowledged, and some of them worth mentioning. Knowledge in use cases specification is assumed for both developers and authors of the requirement models. This knowledge is essential in order to produce usable specifications, and interpret them. The same is true for requirement and software patterns. Due to the role of software patterns in the process, users need to understand its role. Knowledge in software patterns is also required for the pattern selection process, and helps in the instantiation process. A setup phase is required, prior to the application of SCARP. More specifically, the domain model, requirement patterns, software patterns and matching information must be defined in beforehand. SCARP was designed to support model driven development methodologies. As result, the suitability for other kind of methodologies (e.g. Agile) was not addressed. Finally, it is acknowledged that SCARP is a comprehensive process, and not all phases were addressed with the same detail level.

7.1.1 Answer to Research Questions

At the beginning of this work, a set of questions were defined in order to guide the research work. These questions were addressed across this document, and respective answers can be summarized as follows.

Question 1 *Is it possible to have a simple, yet expressive language for use case specification, with support for automation?*

Resorting to the Restricted Use Case Statement (RUS) Controlled Natural Language (CNL), defining an expressive language was possible. Being RUS a CNL, the language enables the possibility to perform automatic transformations of the specifications. Despite the simplicity of RUS specifications, they have shown to be capable of handling several kinds of specifications. Both existing specifications and textual scenario descriptions were translated into RUS, without losing information. The two performed studies provide evidence of the viability of RUS.

Question 2 *Is it possible to perform software requirement patterns inference over use cases, more specifically in the knowledge base representing them?*

Being proven the viability of formalizing requirement specifications as well as representing them in a knowledge base, it was possible to develop a requirement pattern inference mechanism. The performed study shows that requirement patterns were successfully found in the specifications.

Question 3 *Can software requirement patterns be used to select a set of architectural patterns solving them?*

A matching process was implemented, in order to select a set of architectural patterns from requirement patterns. For each requirement pattern, one or more software patterns were identified and an association was defined. The performed case study further supports this question, by demonstrating how the inferred software patterns support the generation of an architectural model.

Question 4 *Can a set of architectural patterns be instantiated and combined, within a defined context, in order to achieve prototype architectures?*

An (API-like) format and a process were defined in order to, respectively, represent and instantiate software patterns. Merging pattern instances was possible resorting to well known pattern merging operators (namely, *stringing* and *overlapping*). This result is supported by the case study, in which a set of software patterns is successfully instantiated and composed into an architectural model.

In conclusion, it is possible to say that the research questions were successfully addressed during this work. Evidences, such as studies, case studies and outputs, were given in order to answer the research questions.

7.1.2 Thesis

The thesis presented in this work is that it is possible to extend the MDA process, including the requirement models as part of the process. Furthermore, a process to support such, and automate the process can be achieved.

The presented work successfully supports this thesis. In SCARP, requirements are formalized as models, and through a systematic process, produce an architectural solution. Furthermore, the systematization of the process results in its capability to automate the process steps, and (semi) automatically produce architectural solutions. The implementation of uCat further supports the thesis. uCat is able to successfully support the automation of SCARP steps. Both the implementation of the approach and the tool (demonstrated through a case study), and corresponding validation, were useful to support the presented thesis.

7.2 Future Work

This work was developed within a specific context, with emphasis in the operationalization of requirement models. Being the MDA a complex and comprehensive process, several improvements can be made. These improvements were left for future work since they were not part of the scope of the project.

At the moment, the domain model provides information for some steps of the process. The entities, and three categories of relationships are being taken in to account. It is possible to define further categories of relationships, and add meaning to certain entities. The domain models used in the approach, also do not specify the relationships multiplicity. Further exploring the information in the domain model is proposed as future work.

While the RUS language performed well in the performed validation studies, OWL is a language capable of handling more complex statements. The extension of RUS in order to support the specification of more complex statements, for instance supporting causality (e.g. user adds product to cart) is proposed. As future work is also proposed the improvement of the use cases syntax, namely support for *includes* and *extends*, and further exploration of the already supported alternative and exception scenarios.

In this work, SCARP was mainly applied to an eCommerce domain (although other domains are addressed in the validation studies). It remains to apply the presented approach to other domains, in order to further prove the viability of the approach against other domains.

The focus of SCARP is the production of architectural solutions. The behavioral aspects are not in the scope of this work, despite their relevance for the final solutions. Hence, the extraction of behavioral aspects from the use case specifications, and consequent transformation in architectural models is proposed, in order to complement the outputs currently produced. MODUS

already goes somehow in that direction, but assumes a pre-defined set of domain dependent behaviors.

The formalization of requirements, as a architectural model, opens several possibilities. One possible approach is the application of model checking techniques in order to check the consistency of the requirement specifications. Such supports the analysis of requirements, prior to the generation of the architectural models, in order to increase the quality of the resulting solutions. The analysis of the formalized requirements is proposed as future work.

Another aspect worth exploring is traceability of requirements and patterns in the produced models. While this work focused in the direct process of generating architectural models, the reverse process of extracting requirements from the models is equally relevant. On the one hand, supports the propagation of changes in the models into the specifications. On the other hand, supports the analysis of existing solutions, in order to analyze them at a higher level of abstraction.

In order to further validate SCARP, other aspects should be validated. Since the performed studies do not cover all the aspects of the approach, performing additional studies would help to justify the viability of SCARP.

In Chapter 4 was presented a proposal of an algorithm to automatically generate uQL from a set of specifications. The algorithm can be further analyzed and refined, in order to be concretized. Such is proposed by applying the algorithm to several publications, and compare the resulting patterns against both the original specifications, and manually defined patterns.

Finally, the produced architectures details information could be improved. Namely, regarding the multiplicities, name of relationships and attributes, and separation in packages. On the one hand, the domain model could be further explored to extract such informations. On the other hand, the flexibility to store and reuse knowledge generated in SCARP could also be of great assistance. Additional formats could be specified for the output format of the produced solutions, in order to support the integration of the outputs with other works.

Appendix A

Inputs and Outputs of SCARP

A.1 Domain model

```
Prefix: dc: <http://purl.org/dc/elements/1.1/>
Prefix: owl: <http://www.w3.org/2002/07/owl#>
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: xml: <http://www.w3.org/XML/1998/namespace>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>

Ontology: <http://www.dm.com>

ObjectProperty: <http://www.dm.com#contains>
  Domain:
    <http://www.dm.com#CompositionOf>

ObjectProperty: <http://www.dm.com#is>
  Domain:
    <http://www.dm.com#TypeOf>

ObjectProperty: <http://www.dm.com#has>
  Domain:
    <http://www.dm.com#PropertyOf>

Class: <http://www.dm.com#TypeOf>

Class: owl:Thing

Class: <http://www.dm.com#PropertyOf>

Class: <http://www.dm.com#CompositionOf>

Individual: <http://www.dm.com#Product>
  Types:
```

```

    owl:Thing

Individual: <http://www.dm.com#Username>
Types:
    owl:Thing

Individual: <http://www.dm.com#Actor>
Types:
    owl:Thing

Individual: <http://www.dm.com#User>
Types:
    owl:Thing
Facts:
    <http://www.dm.com#is> <http://www.dm.com#Actor>,
    <http://www.dm.com#has> <http://www.dm.com#Username>

Individual: <http://www.dm.com#Cart>
Types:
    owl:Thing
Facts:
    <http://www.dm.com#contains> <http://www.dm.com#Product>

```

Listing A.1: Representation of the domain model in OWL.

A.2 RUST Specification

```

<S> <P> <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> a <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> in <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> in a <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> in the <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> the <O> -> Individual: ,<S>,Facts: ,<P> <O>
<S> <P> the <O>+ -> Individual: ,<S>,Facts: ,<P> <O>+

```

Listing A.2: RUST specification used in this work.

A.3 SPARQL queries

```

PREFIX : <http://www.rmsc.com#>
SELECT ?subject ?type
WHERE { ?subject ?predicate ?type .
        FILTER(?predicate rdf:type :TypeOf)
}

```

Listing A.3: SPARQL query to identify individuals related via *TypeOf*.

```

PREFIX : <http://www.rmsc.com#>
SELECT ?subject ?type
WHERE { ?subject ?predicate ?type .

```

```

    FILTER(?predicate rdf:type :CompositionOf)
}

```

Listing A.4: SPARQL query to identify individuals related via *CompositionOf*.

```

PREFIX : <http://www.rmsc.com#>
SELECT ?subject ?type
WHERE { ?subject ?predicate ?type .
    FILTER(?predicate rdf:type :PropertyOf)
}

```

Listing A.5: SPARQL query to identify individuals related via *PropertyOf*.

A.4 Ontology representing the “Add product to cart” use case.

```

Prefix: dc: <http://purl.org/dc/elements/1.1/>
Prefix: j.0: <http://www.url.com/Requirements/>
Prefix: owl: <http://www.w3.org/2002/07/owl#>
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: xml: <http://www.w3.org/XML/1998/namespace>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>

Ontology: <http://www.url.com/Requirements>

AnnotationProperty: rdfs:comment

Datatype: rdf:PlainLiteral

ObjectProperty: j.0:updates
  Annotations:
    rdfs:comment "6"

ObjectProperty: j.0:shows
  Annotations:
    rdfs:comment "2"
  SubPropertyOf:
    j.0:exception

ObjectProperty: j.0:loads
  Annotations:
    rdfs:comment "1"
  SubPropertyOf:
    j.0:alternative

ObjectProperty: j.0:reads
  Annotations:
    rdfs:comment "4"

ObjectProperty: j.0:exception

```

Annotations:
rdfs:comment "-1"
ObjectProperty: j.0:alternative
Annotations:
rdfs:comment "0"
ObjectProperty: j.0:selects
Annotations:
rdfs:comment "3"
ObjectProperty: j.0:adds
Annotations:
rdfs:comment "5"
ObjectProperty: j.0:ends
Annotations:
rdfs:comment "7"
Class: j.0:Data
Class: j.0:Property
Class: j.0:Alternative
Class: j.0:Object
Class: j.0:Actor
Class: j.0:null
Class: j.0:Action
Class: j.0:Attribute
Class: j.0:Exception
Individual: j.0:color
Types:
j.0:Property
Individual: j.0:success
Types:
j.0:Data
Individual: j.0:size
Types:
j.0:Property
Individual: j.0:insuccess
Types:
j.0:null

```

Individual: j.0:action
  Types:
    j.0:Action

Individual: j.0:product
  Types:
    j.0:Attribute

Individual: j.0:system
  Types:
    j.0:Alternative,
    j.0:Exception,
    j.0:Actor
  Facts:
    j.0:loads j.0:allProducts,
    j.0:reads j.0:color,
    j.0:shows j.0:insuccess,
    j.0:reads j.0:size,
    j.0:shows j.0:success,
    j.0:updates j.0:cart,
    j.0:adds j.0:product,
    j.0:reads j.0:amount,
    j.0:shows j.0:product

Individual: j.0:amount
  Types:
    j.0:Attribute

Individual: j.0:user
  Types:
    j.0:Actor
  Facts:
    j.0:selects j.0:color,
    j.0:selects j.0:amount,
    j.0:selects j.0:size,
    j.0:selects j.0:product,
    j.0:ends j.0:action

Individual: j.0:allProducts
  Types:
    j.0:null

Individual: j.0:cart
  Types:
    j.0:Object

```

Listing A.6: Representation in OWL of the “Add product to cart” use case.

A.5 Mapping information

```

Prefix: : <http://www.url.com/mapping#>
Prefix: owl: <http://www.w3.org/2002/07/owl#>

```

```

Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: xml: <http://www.w3.org/XML/1998/namespace>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Prefix: untitled-ontology-62: <http://www.url.com/mapping#>

Ontology: <http://www.url.com/mapping#>

AnnotationProperty: rdfs:comment

Datatype: xsd:string

ObjectProperty: untitled-ontology-62:hasForce

ObjectProperty: untitled-ontology-62:hasForceP

ObjectProperty: untitled-ontology-62:hasForceN

ObjectProperty: untitled-ontology-62:hasGoal

ObjectProperty: untitled-ontology-62:hasConcern

Class: untitled-ontology-62:RequirementPattern

Class: untitled-ontology-62:Concern

Class: untitled-ontology-62:Force

Class: untitled-ontology-62:Goal

Class: untitled-ontology-62:SoftwarePattern

Individual: untitled-ontology-62:Composite
  Annotations:
    rdfs:comment "/**
* @intent Compose objects into tree structures to represent part-whole hierarchies. Composite
  lets clients treat individual objects and compositions of objects uniformly.
* @param client manipulates objects in the composition through the Component interface.
* @param component defines the interface for the composed objects.
* @param leaf represents a leaf in the composition, without children.
* @param composite represents an element with children of the same type.
*/"^^xsd:string
  Types:
    untitled-ontology-62:SoftwarePattern
  Facts:
    untitled-ontology-62:hasGoal untitled-ontology-62:Compose

Individual: untitled-ontology-62:Overkill
  Types:
    untitled-ontology-62:Force

Individual: untitled-ontology-62:Performance
  Types:

```

untitled-ontology-62:Force
Individual: untitled-ontology-62:Recursive
Types:
untitled-ontology-62:Concern
Facts:
untitled-ontology-62:hasForceP untitled-ontology-62:Indirection ,
untitled-ontology-62:hasForceP untitled-ontology-62:Nesting
Individual: untitled-ontology-62:Filtering
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:Nesting
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:Processable
Types:
untitled-ontology-62:Concern
Facts:
untitled-ontology-62:hasForceN untitled-ontology-62:Coupling ,
untitled-ontology-62:hasForceP untitled-ontology-62:Computability ,
untitled-ontology-62:hasForceP untitled-ontology-62:Indirection
Individual: untitled-ontology-62:Abstraction
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:HasCatalog
Types:
untitled-ontology-62:RequirementPattern
Facts:
untitled-ontology-62:hasConcern untitled-ontology-62:Manageable ,
untitled-ontology-62:hasConcern untitled-ontology-62:Browseable
Individual: untitled-ontology-62:Decoupling
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:Feeding
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:Handle
Types:
untitled-ontology-62:Goal
Facts:
untitled-ontology-62:hasForceP untitled-ontology-62:Indirection ,
untitled-ontology-62:hasForceN untitled-ontology-62:Overkill ,
untitled-ontology-62:hasForceP untitled-ontology-62:Efficiency ,
untitled-ontology-62:hasForceP untitled-ontology-62:Performance ,
untitled-ontology-62:hasForceN untitled-ontology-62:Coupling

Individual: untitled-ontology-62:Viewable

Types:

untitled-ontology-62:Concern

Facts:

untitled-ontology-62:hasForceP untitled-ontology-62:Overkill,
 untitled-ontology-62:hasForceP untitled-ontology-62:Simplicity,
 untitled-ontology-62:hasForceP untitled-ontology-62:Efficiency,
 untitled-ontology-62:hasForceP untitled-ontology-62:Performance

Individual: untitled-ontology-62:Versioning

Types:

untitled-ontology-62:Force

Individual: untitled-ontology-62:HasDetails

Types:

untitled-ontology-62:RequirementPattern

Facts:

untitled-ontology-62:hasConcern untitled-ontology-62:Recursive

Individual: untitled-ontology-62:Memory

Types:

untitled-ontology-62:Force

Individual: untitled-ontology-62:Singleton

Annotations:

rdfs:comment "/*

* @intent Ensure a class only has one instance, and provide a global point of access to it.

* @param singleton creates and encapsulate the single object instance.

*/"^^xsd:string

Types:

untitled-ontology-62:SoftwarePattern

Facts:

untitled-ontology-62:hasGoal untitled-ontology-62:Unified

Individual: untitled-ontology-62:Shareable

Types:

untitled-ontology-62:Concern

Facts:

untitled-ontology-62:hasForceN untitled-ontology-62:Overkill,
 untitled-ontology-62:hasForceP untitled-ontology-62:Efficiency,
 untitled-ontology-62:hasForceP untitled-ontology-62:Direction,
 untitled-ontology-62:hasForceP untitled-ontology-62:Memory

Individual: untitled-ontology-62:Indirection

Types:

untitled-ontology-62:Force

Individual: untitled-ontology-62:Constraint

Types:

untitled-ontology-62:Force

Individual: untitled-ontology-62:Flexibility
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:Simplicity
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:Explore
Types:
untitled-ontology-62:Goal
Facts:
untitled-ontology-62:hasForceP untitled-ontology-62:Aggregation,
untitled-ontology-62:hasForceP untitled-ontology-62:Performance,
untitled-ontology-62:hasForceP untitled-ontology-62:Filtering,
untitled-ontology-62:hasForceP untitled-ontology-62:Feeding,
untitled-ontology-62:hasForceN untitled-ontology-62:Indirection,
untitled-ontology-62:hasForceP untitled-ontology-62:Indirection
Individual: untitled-ontology-62:Separation
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:Command
Annotations:
rdfs:comment "/*
* @intent Encapsulate a request as an object, thereby letting you parameterize clients with
different requests , queue or log requests , and support undoable operations.
* @param client Who triggers the operation.
* @param receiver Knows how to perform the operations associated with carrying out.
* @param invoker Asks the command to carry out the request.
* @param command Declares an interface for executing an operation.
* @param concreteCommand Implements Execute by invoking the corresponding operation on
Receiver.
*/"^^xsd:string
Types:
untitled-ontology-62:SoftwarePattern
Facts:
untitled-ontology-62:hasGoal untitled-ontology-62:Process
Individual: untitled-ontology-62:Direction
Types:
untitled-ontology-62:Force
Individual: untitled-ontology-62:Process
Types:
untitled-ontology-62:Goal
Facts:
untitled-ontology-62:hasForceP untitled-ontology-62:Computability,
untitled-ontology-62:hasForceP untitled-ontology-62:Indirection,
untitled-ontology-62:hasForceN untitled-ontology-62:Overkill,
untitled-ontology-62:hasForceP untitled-ontology-62:Coupling

Individual: untitled-ontology-62:Edit

Types:

untitled-ontology-62:Goal

Facts:

untitled-ontology-62:hasForceN untitled-ontology-62:Memory ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Flexibility ,
 untitled-ontology-62:hasForceN untitled-ontology-62:Efficiency ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Versioning

Individual: untitled-ontology-62:Browseable

Types:

untitled-ontology-62:Concern

Facts:

untitled-ontology-62:hasForceN untitled-ontology-62:Constraint ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Feeding ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Filtering ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Efficiency

Individual: untitled-ontology-62:Computability

Types:

untitled-ontology-62:Force

Individual: untitled-ontology-62:Unified

Types:

untitled-ontology-62:Goal

Facts:

untitled-ontology-62:hasForceP untitled-ontology-62:Memory ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Direction ,
 untitled-ontology-62:hasForceN untitled-ontology-62:Indirection ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Efficiency

Individual: untitled-ontology-62:Delegate

Types:

untitled-ontology-62:Goal

Facts:

untitled-ontology-62:hasForceN untitled-ontology-62:Overkill ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Decoupling ,
 untitled-ontology-62:hasForceN untitled-ontology-62:Efficiency ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Abstraction ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Efficiency

Individual: untitled-ontology-62:Sharing

Types:

untitled-ontology-62:Force

Individual: untitled-ontology-62:Compose

Types:

untitled-ontology-62:Goal

Facts:

untitled-ontology-62:hasForceN untitled-ontology-62:Efficiency ,
 untitled-ontology-62:hasForceP untitled-ontology-62:Aggregation ,
 untitled-ontology-62:hasForceN untitled-ontology-62:Memory ,
 untitled-ontology-62:hasForceN untitled-ontology-62:Simplicity ,

<pre> untitled-ontology-62:hasForceP untitled-ontology-62:Nesting, untitled-ontology-62:hasForceP untitled-ontology-62:Indirection </pre>
<pre> Individual: untitled-ontology-62:HasHighlights Types: untitled-ontology-62:RequirementPattern Facts: untitled-ontology-62:hasConcern untitled-ontology-62:Viewable, untitled-ontology-62:hasConcern untitled-ontology-62:Shareable </pre>
<pre> Individual: untitled-ontology-62:Proxy Annotations: rdfs:comment "/* * @intent Provide a surrogate or placeholder for another object to control access to it. * @param client Who triggers the request. * @param subject Defines the common interface for RealSubject and Proxy. * @param realSubject Defines the real object that the proxy represents. * @param proxy Interface which handles the request. */"^^xsd:string Types: untitled-ontology-62:SoftwarePattern Facts: untitled-ontology-62:hasGoal untitled-ontology-62:Delegate </pre>
<pre> Individual: untitled-ontology-62:Coupling Types: untitled-ontology-62:Force </pre>
<pre> Individual: untitled-ontology-62:Flyweight Annotations: rdfs:comment "/* * @intent Use sharing to support large numbers of fine-grained objects efficiently. * @param client Who triggers the request. * @param flyweightFactory Creates and manages flyweight objects. * @param flyweight The class of the instances to be managed. * @param concreteFlyweight Extension of the concrete instances. * @param unsharedConcreteFlyweight A flyweight not shared. */"^^xsd:string Types: untitled-ontology-62:SoftwarePattern Facts: untitled-ontology-62:hasGoal untitled-ontology-62:Handle </pre>
<pre> Individual: untitled-ontology-62:Aggregation Types: untitled-ontology-62:Force </pre>
<pre> Individual: untitled-ontology-62:Iterator Annotations: rdfs:comment "/* * @intent Provide a way to access the elements of an aggregate objectsequentially without exposing its underlying representation. * @param client requests the iterator. </pre>

```

* @param iterator defines the interface to access the and traverse elements.
* @param concreteIterator implements the iterator.
* @param aggregate defines the interface for creating iterator objects.
* @param concreteAggregate implements the aggregate interface.
*/^^xsd:string
  Types:
    untitled-ontology-62:SoftwarePattern
  Facts:
    untitled-ontology-62:hasGoal  untitled-ontology-62:Explore

Individual: untitled-ontology-62:Efficiency
  Types:
    untitled-ontology-62:Force

Individual: untitled-ontology-62:Concurrency
  Types:
    untitled-ontology-62:Force

Individual: untitled-ontology-62:HasShoppingCart
  Types:
    untitled-ontology-62:RequirementPattern
  Facts:
    untitled-ontology-62:hasConcern  untitled-ontology-62:Editable ,
    untitled-ontology-62:hasConcern  untitled-ontology-62:Manageable ,
    untitled-ontology-62:hasConcern  untitled-ontology-62:Processable

Individual: untitled-ontology-62:Manageable
  Types:
    untitled-ontology-62:Concern
  Facts:
    untitled-ontology-62:hasForceN  untitled-ontology-62:Coupling ,
    untitled-ontology-62:hasForceP  untitled-ontology-62:Indirection ,
    untitled-ontology-62:hasForceN  untitled-ontology-62:Indirection ,
    untitled-ontology-62:hasForceP  untitled-ontology-62:Abstraction ,
    untitled-ontology-62:hasForceP  untitled-ontology-62:Decoupling

Individual: untitled-ontology-62:HasAccount
  Types:
    untitled-ontology-62:RequirementPattern
  Facts:
    untitled-ontology-62:hasConcern  untitled-ontology-62:Shareable

Individual: untitled-ontology-62:Memento
  Annotations:
    rdfs:comment "/*
* @intent Without violating encapsulation, capture and externalize an object?s internal state
  so that the object can be restored to this state later.
* @param originator Who triggers the request for creating states.
* @param memento Stores internal state of the Originator object.
* @param caretaker The responsible for the memento?s safekeeping.
* @param state The representation of the state being keep.
* @param item The items belonging to the state.
*/^^xsd:string

```

```

Types:
  untitled-ontology-62:SoftwarePattern
Facts:
  untitled-ontology-62:hasGoal  untitled-ontology-62:Edit

Individual: untitled-ontology-62:Editable
Types:
  untitled-ontology-62:Concern
Facts:
  untitled-ontology-62:hasForceN  untitled-ontology-62:Constraint,
  untitled-ontology-62:hasForceP  untitled-ontology-62:Versioning,
  untitled-ontology-62:hasForceP  untitled-ontology-62:Flexibility

Individual: untitled-ontology-62:HasSearch
Types:
  untitled-ontology-62:RequirementPattern
Facts:
  untitled-ontology-62:hasConcern  untitled-ontology-62:Viewable,
  untitled-ontology-62:hasConcern  untitled-ontology-62:Manageable

```

Listing A.7: Ontology representing the matching information.

A.6 XMI representation of software patterns.

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Mon Dec 14
  22:48:26 WET 2015'>
  <XMI.header>    <XMI.documentation>
    <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
    <XMI.exporterVersion>0.34(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan
      2010) $ </XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000B72'
      name = 'modeloSemTitulo' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000B73'
          name = 'Client' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' isActive = 'false' />
        <UML:Class xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000B74'
          name = 'Receiver' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' isActive = 'false' />
        <UML:Association xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000B75'
          name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
            'false'>
          <UML:Association.connection>
            <UML:AssociationEnd xmi.id = '-64--88-1-65--c06f953:151a294185c
              :-8000:000000000000B76'
              visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
                = 'unordered'

```

```

    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-64--88-1-65--c06f953:151a294185c:-8000:00000000000000
        B73' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-64--88-1-65--c06f953:151a294185c
    :-8000:00000000000000B77'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-64--88-1-65--c06f953:151a294185c:-8000:00000000000000
        B74' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>
<UML:Class xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:00000000000000B78'
  name = 'Invoker' visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' isActive = 'false' />
<UML:Interface xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:00000000000000B79'
  name = 'Command' visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' />
<UML:Class xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:00000000000000B7A'
  name = 'ConcreteCommand' visibility = 'public' isSpecification = 'false'
  isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
  <UML:ModelElement.clientDependency>
    <UML:Abstraction xmi.idref = '-64--88-1-65--c06f953:151a294185c
      :-8000:00000000000000B81' />
  </UML:ModelElement.clientDependency>
</UML:Class>
<UML:Association xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:00000000000000B7B'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
  'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '-64--88-1-65--c06f953:151a294185c
      :-8000:00000000000000B7C'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
        = 'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-64--88-1-65--c06f953:151a294185c:-8000:00000000000000
        B7A' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-64--88-1-65--c06f953:151a294185c
    :-8000:00000000000000B7D'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-64--88-1-65--c06f953:151a294185c:-8000:00000000000000

```

```

        B74' />
      </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000B7E'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
  'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '-64--88-1-65--c06f953:151a294185c
      :-8000:000000000000B7F'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
      = 'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000
        B78' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-64--88-1-65--c06f953:151a294185c
    :-8000:000000000000B80'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
    'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Interface xmi.idref = '-64--88-1-65--c06f953:151a294185c
      :-8000:000000000000B79' />
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Abstraction xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000B81'
  isSpecification = 'false'>
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = '-64--88-1-65--c06f953:151a294185c
      :-8000:000000000000B82' />
  </UML:ModelElement.stereotype>
  <UML:Dependency.client>
    <UML:Class xmi.idref = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000B7A
    ' />
  </UML:Dependency.client>
  <UML:Dependency.supplier>
    <UML:Interface xmi.idref = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000
    B79' />
  </UML:Dependency.supplier>
</UML:Abstraction>
<UML:Stereotype xmi.id = '-64--88-1-65--c06f953:151a294185c:-8000:000000000000B82'
  name = 'realize' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
  isAbstract = 'false'>
  <UML:Stereotype.baseClass>Abstraction</UML:Stereotype.baseClass>
</UML:Stereotype>
</UML:Namespace.ownedElement>
</UML:Model>

```

```
</XMI.content>
</XMI>
```

Listing A.8: XMI representation of the *Command* software pattern.

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Wed Jun 01
14:24:03 WEST 2016'>
  <XMI.header>    <XMI.documentation>
    <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
    <XMI.exporterVersion>0.34(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan
2010) $ </XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E77'
      name = 'modeloSemTitulo' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E78'
          name = 'Client' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
        <UML:Interface xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E79'
          name = 'Component' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false'>
        <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E7A'
          name = 'Composite' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
        <UML:ModelElement.clientDependency>
          <UML:Abstraction xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
            :-8000:000000000000E7B'>
        </UML:ModelElement.clientDependency>
      </UML:Class>
      <UML:Abstraction xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E7B'
        isSpecification = 'false'>
      <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
          :-8000:000000000000E7C'>
      </UML:ModelElement.stereotype>
      <UML:Dependency.client>
        <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E7A'
          >
      </UML:Dependency.client>
      <UML:Dependency.supplier>
        <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000
          E79'>
      </UML:Dependency.supplier>
    </UML:Abstraction>
    <UML:Stereotype xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E7C'
      name = 'realize' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
    <UML:Stereotype.baseClass>Abstraction</UML:Stereotype.baseClass>
  </UML:Stereotype>
```



```

<UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E7D'
  name = 'Leaf' visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:ModelElement.clientDependency>
  <UML:Abstraction xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:000000000000E81' />
</UML:ModelElement.clientDependency>
</UML:Class>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E7E'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
  'false'>
<UML:Association.connection>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:000000000000E7F'
  visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
  = 'unordered'
  aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.participant>
  <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000
    E7A' />
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
  :-8000:000000000000E80'
  visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
  'unordered'
  aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.participant>
  <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:000000000000E79' />
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Abstraction xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E81'
  isSpecification = 'false'>
<UML:ModelElement.stereotype>
  <UML:Stereotype xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:000000000000E7C' />
</UML:ModelElement.stereotype>
<UML:Dependency.client>
  <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E7D
  ' />
</UML:Dependency.client>
<UML:Dependency.supplier>
  <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000
    E79' />
</UML:Dependency.supplier>
</UML:Abstraction>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000E83'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
  'false'>
<UML:Association.connection>

```

```

    <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:00000000000000E84'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
        = 'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
        E78' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:00000000000000E85'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:00000000000000E79' />
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
</UML:Namespace.ownedElement>
</UML:Model>
</XMI.content>
</XMI>

```

Listing A.9: XMI representation of the *Composite* software pattern.

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Wed Jun 01
  14:56:06 WEST 2016'>
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
      <XMI.exporterVersion>0.34(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan
        2010) $ </XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000128D'
      name = 'modeloSemTitulo' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
    <UML:Namespace.ownedElement>
      <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000128E'
        name = 'Client' visibility = 'public' isSpecification = 'false' isRoot = 'false'
        isLeaf = 'false' isAbstract = 'false' isActive = 'false' />
      <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000128F'
        name = 'FlyweightFactory' visibility = 'public' isSpecification = 'false'
        isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false' />
      <UML:Interface xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:0000000000001290'
        name = 'Flyweight' visibility = 'public' isSpecification = 'false' isRoot = 'false'
        isLeaf = 'false' isAbstract = 'false' />
      <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:0000000000001291'

```

```

name = 'ConcreteFlyweight' visibility = 'public' isSpecification = 'false'
isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:ModelElement.clientDependency>
  <UML:Abstraction xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:000000000000129C'/>
</UML:ModelElement.clientDependency>
</UML:Class>
<UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:0000000000001292'
  name = 'UnsharedConcreteFlyweight' visibility = 'public' isSpecification = 'false'
  isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:ModelElement.clientDependency>
  <UML:Abstraction xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:000000000000129E'/>
</UML:ModelElement.clientDependency>
</UML:Class>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:0000000000001293'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
    'false'>
<UML:Association.connection>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:0000000000001294'
    visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
      = 'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:000000000000128E'/>
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
  :-8000:0000000000001295'
  visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
    'unordered'
  aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:000000000000128F'/>
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:0000000000001296'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
    'false'>
<UML:Association.connection>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:0000000000001297'
    visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
      = 'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:000000000000128E'/>

```

```

    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:0000000000001298'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:0000000000001291'>/>
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:0000000000001299'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
    'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:000000000000129A'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
        = 'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
        :-8000:000000000000128E'>/>
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:000000000000129B'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:0000000000001292'>/>
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Abstraction xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000129C'
  isSpecification = 'false'>
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:000000000000129D'>/>
  </UML:ModelElement.stereotype>
  <UML:Dependency.client>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:0000000000001291'>/>
  </UML:Dependency.client>
  <UML:Dependency.supplier>
    <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:0000000000001290'>/>

```

```

</UML:Dependency.supplier>
</UML:Abstraction>
<UML:Stereotype xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000129D'
  name = 'realize' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
  isAbstract = 'false'>
  <UML:Stereotype.baseClass>Abstraction</UML:Stereotype.baseClass>
</UML:Stereotype>
<UML:Abstraction xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000129E'
  isSpecification = 'false'>
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:00000000000129D' />
  </UML:ModelElement.stereotype>
  <UML:Dependency.client>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:000000000001292' />
  </UML:Dependency.client>
  <UML:Dependency.supplier>
    <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:000000000001290' />
  </UML:Dependency.supplier>
</UML:Abstraction>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:0000000000012A0'
  name = 'flyweights' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
  isAbstract = 'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:0000000000012A1'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering =
        'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
        :-8000:00000000000128F' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:0000000000012A2'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:000000000001290' />
  </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
</UML:Namespace.ownedElement>
</UML:Model>
</XMI.content>

```

</XMI>

Listing A.10: XMI representation of the *Flyweight* software pattern.

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Wed Jun 01
14:22:51 WEST 2016'>
  <XMI.header>    <XMI.documentation>
    <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
    <XMI.exporterVersion>0.34(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan
2010) $ </XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000D66'
      name = 'modeloSemTitulo' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000D67'
          name = 'ConcreteAggregate' visibility = 'public' isSpecification = 'false'
          isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
          <UML:ModelElement.clientDependency>
            <UML:Abstraction xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
              :-8000:000000000000D72' />
          </UML:ModelElement.clientDependency>
        </UML:Class>
        <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000D68'
          name = 'ConcreteIterator' visibility = 'public' isSpecification = 'false'
          isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
          <UML:ModelElement.clientDependency>
            <UML:Abstraction xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
              :-8000:000000000000D74' />
          </UML:ModelElement.clientDependency>
        </UML:Class>
        <UML:Interface xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000D69'
          name = 'Aggregate' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' />
        <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000D6A'
          name = 'Client' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' isActive = 'false' />
        <UML:Interface xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000D6B'
          name = 'Iterator' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' />
        <UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000D6C'
          name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
          'false'>
          <UML:Association.connection>
            <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
              :-8000:000000000000D6D'
              visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
              = 'unordered'
              aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
            <UML:AssociationEnd.participant>

```

```

        <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
            D6A' />
    </UML:AssociationEnd.participant>
</UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:00000000000000D6E'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
        'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
        <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
            :-8000:00000000000000D69' />
    </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000D6F'
    name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
        'false'>
    <UML:Association.connection>
        <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
            :-8000:00000000000000D70'
            visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
                = 'unordered'
            aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
        <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
            D6A' />
    </UML:AssociationEnd.participant>
</UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:00000000000000D71'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
        'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
        <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
            :-8000:00000000000000D6B' />
    </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Abstraction xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000D72'
    isSpecification = 'false'>
    <UML:ModelElement.stereotype>
        <UML:Stereotype xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
            :-8000:00000000000000D73' />
    </UML:ModelElement.stereotype>
    <UML:Dependency.client>
        <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000D67
            ' />
    </UML:Dependency.client>
</UML:Dependency.supplier>

```

```

    <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
      D69' />
  </UML:Dependency.supplier>
</UML:Abstraction>
<UML:Stereotype xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000D73'
  name = 'realize' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
  isAbstract = 'false'>
  <UML:Stereotype.baseClass>Abstraction</UML:Stereotype.baseClass>
</UML:Stereotype>
<UML:Abstraction xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000D74'
  isSpecification = 'false'>
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:00000000000000D73' />
  </UML:ModelElement.stereotype>
<UML:Dependency.client>
  <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000D68
    ' />
</UML:Dependency.client>
<UML:Dependency.supplier>
  <UML:Interface xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
    D6B' />
</UML:Dependency.supplier>
</UML:Abstraction>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000D76'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
  'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
      :-8000:00000000000000D77'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
        = 'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
        D68' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:00000000000000D78'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
      D67' />
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000D79'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
  'false'>

```



```

<UML:Association.connection>
  <UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
    :-8000:00000000000000D7A'
    visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
      = 'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
      D67'>/>
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
  :-8000:00000000000000D7B'
  visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
    'unordered'
  aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.participant>
  <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:00000000000000
    D68'>/>
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
</UML:Namespace.ownedElement>
</UML:Model>
</XMI.content>
</XMI>

```

Listing A.11: XMI representation of the *Iterator* software pattern.

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Mon Dec 14
  22:10:00 WET 2015'>
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
      <XMI.exporterVersion>0.34(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan
        2010) $ </XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:00000000000010FC'
      name = 'modeloSemTitulo' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:00000000000010FD'
          name = 'Originator' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          ,
          isLeaf = 'false' isAbstract = 'false' isActive = 'false'>/>
        <UML:Class xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:00000000000010FE'
          name = 'Memento' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' isActive = 'false'>/>
        <UML:Class xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:00000000000010FF'
          name = 'Caretaker' visibility = 'public' isSpecification = 'false' isRoot = 'false'

```

```

isLeaf = 'false' isAbstract = 'false' isActive = 'false'>/>
<UML:Class xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:0000000000001100'
name = 'State' visibility = 'public' isSpecification = 'false' isRoot = 'false'
isLeaf = 'false' isAbstract = 'false' isActive = 'false'>/>
<UML:Class xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:0000000000001101'
name = 'Item' visibility = 'public' isSpecification = 'false' isRoot = 'false'
isLeaf = 'false' isAbstract = 'false' isActive = 'false'>/>
<UML:Association xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:0000000000001102'
name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
'false'>
<UML:Association.connection>
<UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
:-8000:0000000000001103'
visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
= 'unordered'
aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.participant>
<UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
:-8000:00000000000010FD'>/>
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
:-8000:0000000000001104'
visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
'unordered'
aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.participant>
<UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
:-8000:00000000000010FE'>/>
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:0000000000001105'
name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
'false'>
<UML:Association.connection>
<UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
:-8000:0000000000001106'
visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
= 'unordered'
aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.participant>
<UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
:-8000:00000000000010FF'>/>
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
:-8000:0000000000001107'
visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
'unordered'
aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.participant>

```

```

        <UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
            :-8000:00000000000010FE'>
    </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:0000000000001108'
    name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
        'false'>
    <UML:Association.connection>
        <UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
            :-8000:0000000000001109'
            visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
                = 'unordered'
            aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
        <UML:AssociationEnd.participant>
            <UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
                :-8000:00000000000010FD'>
            </UML:AssociationEnd.participant>
        </UML:AssociationEnd>
    <UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
        :-8000:000000000000110A'
        visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
            'unordered'
        aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
        <UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
            :-8000:0000000000001100'>
        </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:000000000000110B'
    name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
        'false'>
    <UML:Association.connection>
        <UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
            :-8000:000000000000110C'
            visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
                = 'unordered'
            aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
        <UML:AssociationEnd.participant>
            <UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
                :-8000:0000000000001100'>
            </UML:AssociationEnd.participant>
        </UML:AssociationEnd>
    <UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
        :-8000:000000000000110D'
        visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
            'unordered'
        aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
        <UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec

```

```

        :-8000:0000000000001101'/'>
      </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-64--88-1-65--67b626f9:151a280f3ec:-8000:000000000000110E'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
    'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
      :-8000:000000000000110F'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
        = 'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
        :-8000:00000000000010FE'/'>
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '-64--88-1-65--67b626f9:151a280f3ec
  :-8000:0000000000001110'
  visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
    'unordered'
  aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.participant>
  <UML:Class xmi.idref = '-64--88-1-65--67b626f9:151a280f3ec
    :-8000:0000000000001100'/'>
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
</UML:Namespace.ownedElement>
</UML:Model>
</XMI.content>
</XMI>

```

Listing A.12: XMI representation of the *Memento* software pattern.

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Mon Dec 14
  22:52:40 WET 2015'>
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
      <XMI.exporterVersion>0.34(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan
        2010) $ </XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:0000000000000865'
      name = 'modeloSemTitulo' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
    <UML:Namespace.ownedElement>
      <UML:Class xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:0000000000000866'

```

```

    name = 'Client' visibility = 'public' isSpecification = 'false' isRoot = 'false'
    isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:Interface xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:0000000000000867'
    name = 'Subject' visibility = 'public' isSpecification = 'false' isRoot = 'false'
    isLeaf = 'false' isAbstract = 'false'>
<UML:Class xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:0000000000000868'
    name = 'RealSubject' visibility = 'public' isSpecification = 'false' isRoot = '
        false'
    isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:ModelElement.clientDependency>
    <UML:Abstraction xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:000000000000086A'>
    </UML:ModelElement.clientDependency>
</UML:Class>
<UML:Class xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:0000000000000869'
    name = 'Proxy' visibility = 'public' isSpecification = 'false' isRoot = 'false'
    isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:ModelElement.clientDependency>
    <UML:Abstraction xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:000000000000086C'>
    </UML:ModelElement.clientDependency>
</UML:Class>
<UML:Abstraction xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:000000000000086A'
    isSpecification = 'false'>
<UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:000000000000086B'>
    </UML:ModelElement.stereotype>
<UML:Dependency.client>
    <UML:Class xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:0000000000000868'>
    </UML:Dependency.client>
<UML:Dependency.supplier>
    <UML:Interface xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:0000000000000867'>
    </UML:Dependency.supplier>
</UML:Abstraction>
<UML:Stereotype xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:000000000000086B'
    name = 'realize' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
    isAbstract = 'false'>
    <UML:Stereotype.baseClass>Abstraction</UML:Stereotype.baseClass>
</UML:Stereotype>
<UML:Abstraction xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:000000000000086C'
    isSpecification = 'false'>
<UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:000000000000086B'>
    </UML:ModelElement.stereotype>
<UML:Dependency.client>
    <UML:Class xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:0000000000000869'>
    </UML:Dependency.client>
<UML:Dependency.supplier>

```

```

    <UML:Interface xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
      :-8000:0000000000000867' />
  </UML:Dependency.supplier>
</UML:Abstraction>
<UML:Association xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:000000000000086E'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
    'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4
      :-8000:000000000000086F'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
        = 'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:0000000000000866' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4
    :-8000:0000000000000870'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Interface xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
      :-8000:0000000000000867' />
  </UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
<UML:Association xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4:-8000:0000000000000871'
  name = '' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract =
    'false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4
      :-8000:0000000000000872'
      visibility = 'public' isSpecification = 'false' isNavigable = 'false' ordering
        = 'unordered'
      aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
    <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
        :-8000:0000000000000869' />
    </UML:AssociationEnd.participant>
  </UML:AssociationEnd>
  <UML:AssociationEnd xmi.id = '-64--88-1-65--1265bbf9:151a2b0c5f4
    :-8000:0000000000000873'
    visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering =
      'unordered'
    aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.participant>
    <UML:Class xmi.idref = '-64--88-1-65--1265bbf9:151a2b0c5f4
      :-8000:0000000000000868' />
  </UML:AssociationEnd.participant>

```

```

    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>
</UML:Namespace.ownedElement>
</UML:Model>
</XMI.content>
</XMI>

```

Listing A.13: XMI representation of the *Proxy* software pattern.

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Wed Jun 01
14:09:56 WEST 2016'>
  <XMI.header>    <XMI.documentation>
    <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
    <XMI.exporterVersion>0.34(6) revised on $Date: 2010-01-11 22:20:14 +0100 (Mon, 11 Jan
2010) $ </XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Model xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000C63'
      name = 'modeloSemTitulo' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
      isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000C64'
          name = 'Singleton' visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
          <UML:Classifier.feature>
            <UML:Attribute xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000C65'
              ,
              name = 'instance' visibility = 'public' isSpecification = 'false' ownerScope =
              'instance'
              changeability = 'changeable' targetScope = 'instance'>
            <UML:StructuralFeature.multiplicity>
              <UML:Multiplicity xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
                :-8000:000000000000C66'>
                <UML:Multiplicity.range>
                  <UML:MultiplicityRange xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
                    :-8000:000000000000C67'
                    lower = '1' upper = '1'/>
                </UML:Multiplicity.range>
              </UML:Multiplicity>
            </UML:StructuralFeature.multiplicity>
            <UML:StructuralFeature.type>
              <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000
                C68'>
              </UML:StructuralFeature.type>
            </UML:Attribute>
            <UML:Operation xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000C69'
              ,
              name = 'getInstance' visibility = 'public' isSpecification = 'false' ownerScope
              = 'instance'
              isQuery = 'false' concurrency = 'sequential' isRoot = 'false' isLeaf = 'false'

```

```

    isAbstract = 'false'>
    <UML:BehavioralFeature.parameter>
      <UML:Parameter xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1
        :-8000:00000000000000C6A'
        name = 'return' isSpecification = 'false' kind = 'return'>
      <UML:Parameter.type>
        <UML:Class xmi.idref = '-84-26-39-97-10a9aaf0:1550c12bec1
          :-8000:00000000000000C68'>/>
      </UML:Parameter.type>
    </UML:Parameter>
  </UML:BehavioralFeature.parameter>
</UML:Operation>
</UML:Classifier.feature>
</UML:Class>
<UML:Class xmi.id = '-84-26-39-97-10a9aaf0:1550c12bec1:-8000:000000000000C68'
  name = 'Object' visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' isActive = 'false'>/>
</UML:Namespace.ownedElement>
</UML:Model>
</XMI.content>
</XMI>

```

Listing A.14: XMI representation of the *Singleton* software pattern.

Appendix B

Inputs and outputs regarding the validation studies

B.1 Specifications selected for the first study

This appendix presents the specifications used for the first study. It is worth mentioning that the original specifications were described in Portuguese, the native language of the participants.

B.1.1 Use cases

Table B.1: Modify players information.

	User	System
1		Lists players
2	Selects player	
3		Presents player information
4	Performs changes	
5	Submits changes	
6		Validates changes
7		Informs success

Table B.2: Cancel a tournament.

	User	System
1	Inserts name of the tournament to cancel	
2		Searches tournament
3		Presents list of tournaments with matching names
4	Selects desired tournament	
5		Presents tournament information
6	Selects remove option	
7		Asks removal confirmation
8	Confirms tournament removal	
9		Removes tournament
10		Informs success of removal
	Exception: No results (Step 2)	
a 1		Informs that no matching tournaments were found
	Exception: No confirmation (Step 6)	
b 1		Informs of removal cancel

Table B.3: Check donative information.

	User	System
1	Informs donator information	
2		Verified donator
3		Asks donator information
4	Informs donative to donative to check	
5		Verifies donative
6		Provides donative information

Table B.4: Remove a user.

	User	System
1	inserts username	
2		verifies username
3	inserts password	
4		verifies password
5		user is removed

Table B.5: Mark donative as used.

	User	System
1	selects donative to mark as used	
2		verifies if donative exists
3	asks for selected donative to be marked as used	
4		marks donative as used
5		informs that donative was marked as used

Table B.6: Change a project.

	User	System
1		presets list of projects
2	selects project	
3	provides information to change	
4		validates new information
5		register changes in the system

Table B.7: Allocate items to a certain project phase.

	User	System
1		presents list of all project phases
2		presents list of items available
3	selects the phase to which add items	
4	selects which item to allocate	
5	selects the quantity to allocate	
6		allocates the quantity of items to the project phase

Table B.8: Check a job information.

	User	System
1	asks for job information	
2		verifies if job has started
3		presents job information

B.1.2 Textual descriptions

“Search a product” scenario description

A user clicks in the “search” link in the website. Then, the system shows a field where the user should insert the keyword to search, as well as the search criteria (price, date, etc.). When the user clicks “ok”, the system performs a search (based in the given criteria), creates a result list and shows such list to the user. Finally, the user checks the resulting list.

“Upload a model to a repository” scenario description

A user clicks in “new model” link. Then, the website shows a page with several fields to fill, specifically name, description, image and file. The user fills those fields, and clicks in “submit”. The system registers the fields, uploads the model, and shows a success message. If any of the fields is missing, an error message is shown, and the process finishes.

“Download a model from a webpage” scenario description

A user clicks in “models” link. The system shows a list of modes. Then, the user browses the list, and selects the desired model. The user selects “download”, the system processes the requests, and starts the download. If the model is private, after selecting “open”, the system sends a message to the model’s author (with a request to view the model), and shows an informative message to the user, instead of a success message.

B.2 Scenario descriptions selected for the second study

“List products from category and open details” scenario description

A user clicks in a category. The system filters the products by the category, and creates a new list which will then present to the user. The use might then check the result, from which will select one. The system will then retrieve the product details, and present them to the user. Finally, the user checks the product.

“Registration and sign in” scenario description

The user clicks in *register*. The system will then present a form where it requests the username, the password and the email. The user will then provide such informations, that the system will verify. If the informations are valid, the system will also create the register. The user will next click in *login*, and the system will request both the username and password. After validating those informations, the system creates a new session, and presents a success message.

“Check historic” scenario description

The user starts by clicking in *historic*. The system will the load the items that the user have viewed, and creates a list with such informations. Next, the system shows the list to the user, which can check it.

B.3 Requirement patterns catalog

Based in the patterns described in [123], a requirements pattern catalog is presented. The intent is presented alongside each pattern.

B.3.1 Simple Search

The *Simple Search* requirement pattern is described as: “*Offer a search functionality consisting of a search label, a keyword and a filter*”.

```
(user) (clicks|asks) (search|search_link|searchLink) 13
(system) (shows|asks|displays) (?) 13
(user) (inserts|enters) (keyword|date|price|criteria) 13
(user) (clicks) (ok) 13
(system) (performs) (search) 12
(system) (creates) (result_list|list|resultlist|result|results) 12
(system) (shows|displays|display|presents) (list|result_list|results) 12
(user) (checks) (list|resulting_list|info|result|results) 12
```

Listing B.1: uQL *Simple Search* requirement pattern.

B.3.2 Catalog

The *Catalog* requirement pattern is described as: “*A catalog class to be a product collection*”.

```
(user) (clicks) (category) 22
(system) (filters|performs) (products) 7
(system) (creates) (list|new_list) 10
(system) (shows|presents|displays) (list|product_list|results) 5
(user) (checks) (list|result|results) 5
(user) (selects|clicks) (product|result) 5
(system) (retrieves) (?info) 15
(system) (presents|shows) (?info) 15
(user) (checks) (product) 16
```

Listing B.2: uQL *Catalog* requirement pattern.

B.3.3 Registration

The *Registration* requirement pattern is described as: “*Offers users a registration after first use of the data to avoid reenter the same information*”.

```
(user) (clicks) (register|register_link|registerLink) 12
(system) (presents|displays|shows) (form) 8
(system) (requests|asks|presents) (username|password|email) 8
(user) (provides|enters|inserts) (?data) 8
(system) (verifies|validates) (?data) 8
(system) (creates) (register|user) 8
(user) (clicks) (login|loginLink|login_link) 8
(system) (requests|asks) (username|password) 8
(user) (inserts|provides|enters) (username|password) 8
(system) (verifies|validates) (information|informations|info) 8
(system) (creates) (session|new_session) 8
(system) (presents|displays|shows) (success_message|successmessage|successMessage|message) 8
```

Listing B.3: uQL *Registration* requirement pattern.

B.3.4 List Builder

The *List Builder* requirement pattern is described as: “*Present the total list and provide editing functionality on selected items*”.

```
(user) (clicks|asks) (search|search_link|searchLink) 20
(system) (shows|asks|displays) (?) 10
(user) (inserts|enters) (keyword|date|price|criteria) 20
(user) (clicks) (ok) 10
(system) (performs) (search) 10
(system) (creates) (result_list|list|resultlist|result|results) 10
(system) (shows|displays|display|presents) (list|result_list|results) 15
```

```
(user) (checks) (list|resulting_list|info|result|results) 15
```

Listing B.4: uQL *List Builder* requirement pattern.

B.4 Software patterns catalog

The presented catalog is based in [41], and described according to the unified template produced as part of the survey regarding software patterns [27]. The template is composed of the fields presented in Figure B.1.

<p>Pattern name and classification The pattern name and category;</p> <p>Intent The problem it tries to solve;</p> <p>Also known as Alternative names;</p> <p>Motivation The reason for this pattern to exist;</p> <p>Applicability When is this pattern applicable;</p> <p>Structure A generic implementation of the pattern;</p> <p>Participants A list of constituents of the pattern;</p> <p>Collaborations How the pattern interacts with other components;</p> <p>Consequences Consequences of using this pattern;</p> <p>Implementations Guidelines for implementing the pattern;</p> <p>Sample code Pseudocode to illustrate the pattern (not applicable in this context);</p> <p>Known uses Examples where the pattern is known to be used;</p> <p>Related patterns Patterns related with this one;</p> <p>Forces Side effects of applying this pattern. Differs from consequences by its specificity;</p> <p>Example resolved An example implementation.</p>

Figure B.1: Software patterns template.

B.4.1 Proxy

Pattern name and classification *Proxy*, structural design pattern.

Intent Provide a surrogate or placeholder for another object to control access to it.

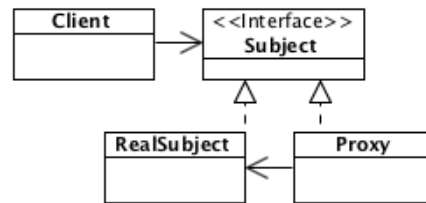
Also known as Surrogate.

Motivation One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.

Applicability Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.

Structure (Figure B.2).

Participants Proxy - maintains a reference that lets the proxy access the real subject; Subject - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected; RealSubject - defines the real object that the proxy represents.

Figure B.2: *Proxy* pattern structure.

Collaborations Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Consequences The Proxy pattern introduces a level of indirection when accessing an object.

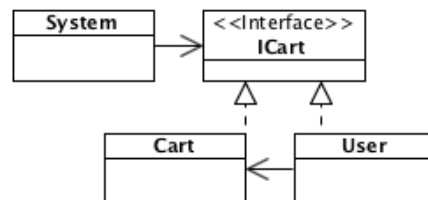
Implementations Overloading the member access operator in C++; using `doesNotUnderstand` in Smalltalk; Proxy doesn't always have to know the type of real subject.

Known uses NEXTSTEP uses proxies as local representatives for objects that may be distributed. The usage of proxies in Smalltalk to access remote objects is known. Provide side-effects on method calls and access control with "Encapsulators" is possible resorting to the proxy pattern.

Related patterns Adapter, Decorator.

Forces Efficiency, Decoupling, Separation, Abstraction, Efficiency, Overkill.

Example resolved (Figure B.3)

Figure B.3: *Proxy* pattern instance.

B.4.2 Command

Pattern name and classification *Command*, behavioral design pattern.

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Also known as Action, Transaction.

Motivation Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Applicability The Command pattern can be used when the parameterization of objects by an action to perform is required; when specifying, queueing, and executing requests at different times; supporting undo; support logging changes so that they can be reapplied in case of a system crash; structuring a system around high-level operations built on primitives operations.

Structure (Figure B.4)

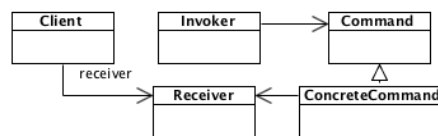


Figure B.4: *Command* pattern structure.

Participants Command - declares an interface for executing an operation; ConcreteCommand - defines a binding between a Receiver object and an action and implements Execute by invoking the corresponding operation(s) on Receiver; Client - creates a ConcreteCommand object and sets its receiver; Invoker - asks the command to carry out the request; Receiver - knows how to perform the operations associated with carrying out a request.

Collaborations The client creates a ConcreteCommand object and specifies its receiver. An Invoker object stores the ConcreteCommand object. The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute. The ConcreteCommand object invokes operations on its receiver to carryout the request.

Consequences Command decouples the object that invokes the operation from the one that knows how to perform it. Commands are first-class objects, they can be manipulated and extended like any other object. Commands can be assembled into a composite command. In general, composite commands are an instance of the Composite pattern. It's easy to add new Commands, because existing classes don't have to be changed.

Implementations: A command can have a wide range of abilities, as supporting undo and redo, avoiding error accumulation in the undo process. A possible implementation is by using C++ templates.

Known uses Command pattern appears in the THINK class library. Unidraw's command objects are unique in that they can behave like messages. The implementation of functor objects that are functions, in C++ is possible.

Related patterns Composite, Memento, Prototype.

Forces Separation, Computability, Indirection.

Example resolved (Figure B.5)

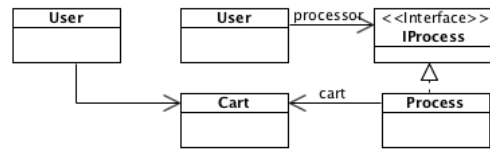


Figure B.5: *Command* pattern instance.

B.4.3 Memento

Pattern name and classification *Memento*, behavioral design pattern.

Intent Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Also known as Token.

Motivation Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.

Applicability Use the Memento pattern when a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later. A direct interface to obtaining the state would expose implementation. Details and break the object's encapsulation.

Structure (Figure B.6)

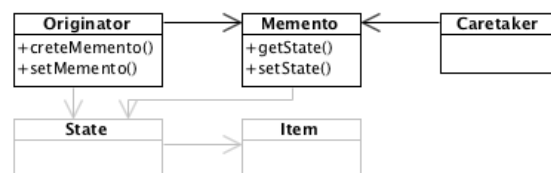


Figure B.6: *Memento* pattern structure.

Participants Memento stores internal state of the originator object, protects against access by objects other than the originator. Originator creates a memento containing a snapshot of its current internal state. Caretaker is responsible for the memento's safekeeping. Never operates on or examines the contents of a memento.

Collaborations A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator. Mementos are passive.

Consequences Preserving encapsulation boundaries. It simplifies the Originator class. Using mementos might be expensive. Supports defining narrow and wide interfaces. There are hidden costs in caring for mementos.

Implementations When implementing mementos two considerations must be taken in account, namely language support, and storing incremental changes.

Known uses Unidraw's support for connectivity through its CSolver class. The QOCA constraint-solving toolkit stores incremental information in mementos.

Related patterns Command, Iterator,

Forces Separation, Flexibility, Versioning.

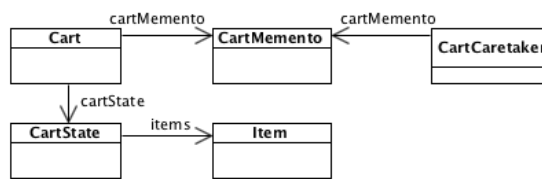


Figure B.7: *Memento* pattern instance.

Example resolved

B.4.4 Iterator

Pattern name and classification: *Iterator*, behavioral design pattern.

Intent Provide a way to access the elements of an aggregate objects sequentially without exposing its underlying representation.

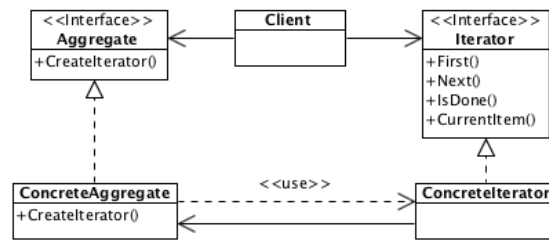
Also known as Cursor.

Motivation An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.

Applicability The Iterator pattern can be used to access an aggregate object's contents without exposing its internal representation, to support multiple traversals of aggregate objects, to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Structure (Figure B.8)

Participants Iterator - defines an interface for accessing and traversing elements. ConcreteIterator - implements the Iterator interface. Keeps track of the current position in the traversal of the aggregate. Aggregate - defines an interface for creating an Iterator object. ConcreteAggregate - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Figure B.8: *Iterator* pattern structure.

Collaborations A `ConcreteIterator` keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences The *Iterator* pattern supports variations in the traversal of an aggregate. Iterators simplify the `Aggregate` interface. More than one traversal can be pending on an aggregate.

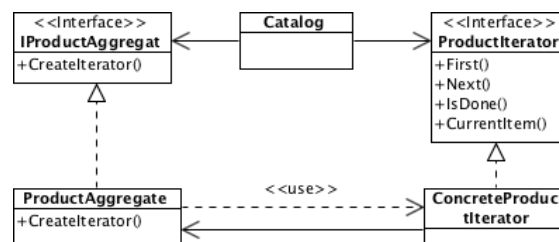
Implementations A fundamental issue is deciding which party controls the iteration, the iterator or the client that uses the iterator. The iterator is not the only place where the traversal algorithm can be defined. It can be dangerous to modify an aggregate while you're traversing it. The minimal interface to `Iterator` consists of the operations `First`, `Next`, `IsDone`, and `CurrentItem`. Iterators may have privileged access.

Known uses Smalltalk uses iterators implicitly. Polymorphic iterators are provided by the ET++ container classes. ObjectWindows 2.0 provides a class hierarchy of iterators for containers.

Related patterns Composite, Factory Method, Memento.

Forces Feeding, Aggregation, Indirection, Filtering, Performance, Direction.

Example resolved (Figure B.9).

Figure B.9: *Iterator* pattern instance.

B.4.5 Composite

Pattern name and classification *Composite*, structural design pattern.

Intent Compose objects into tree structures to represent part-whole hierarchies.

Also known as N/A

Motivation The key to the Composite pattern is an abstract class that represents both primitives and their containers.

Applicability Represent part-whole hierarchies of objects. Ignore the difference between compositions of objects and individual objects. Treat all objects in the composite structure uniformly.

Structure (Figure B.10)

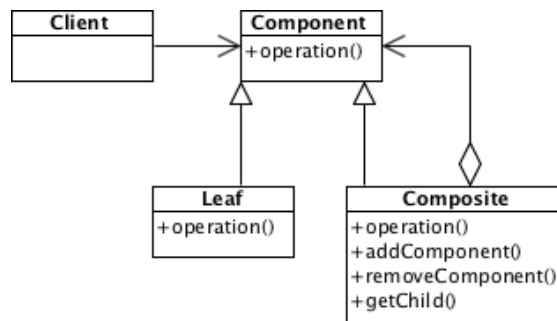


Figure B.10: *Composite* pattern structure.

Participants Component - declares the interface for objects in the composition, implements default behavior for the interface common to all classes, declares an interface for accessing and managing its child components, and, optionally defines an interface for accessing a component's parent in the recursive structure. Leaf - represents leaf objects in the composition, has no children, and defines behavior for primitive objects in the composition. Composite - defines behavior for components having children, stores child components, and implements child-related operations in the Component interface. Client - manipulates objects in the composition through the Component interface.

Collaborations Clients use the Component class interface to interact with objects in the composite structure.

Consequences Defines class hierarchies consisting of primitive objects and composite objects. Makes the client simple. Makes it easier to add new kinds of components. Can make the design overly general.

Implementations Explicit parent references. Sharing components. Maximizing the Component interface. Declaring the child management operations. Declaring the child manage-

ment operations. Child ordering. Caching to improve performance.

Known uses Examples of the Composite pattern can be found in almost all object-oriented systems. The original View class of Smalltalk Model/View/Controller was a Composite, and nearly every user interface toolkit or framework has followed in its steps. The RTL Smalltalk compiler framework uses the Composite pattern extensively. Another example of this pattern occurs in the financial domain, where a portfolio aggregates individual assets.

Related patterns Command, Chain of Responsibility, Decorator, Flyweight, Iterator, Visitor.

Forces Indirection, Nesting, Aggregation, Efficiency, Simplicity, Memory.

Example resolved (Figure B.11)

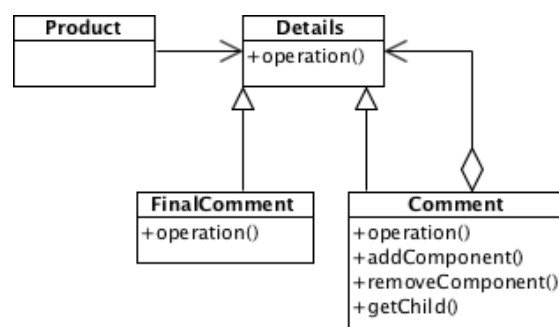


Figure B.11: *Composite* pattern instance.

B.4.6 Flyweight

Pattern name and classification *Proxy*, structural design pattern.

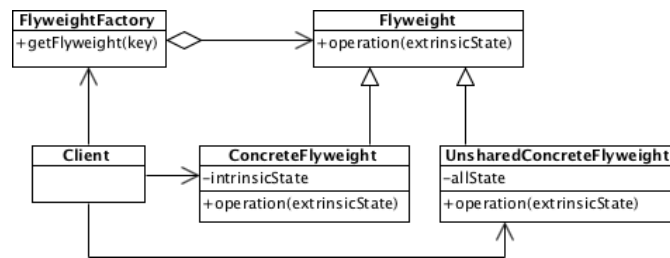
Intent Use sharing to support large numbers of fine-grained objects efficiently.

Also known as N/A

Motivation Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.

Applicability The Flyweight pattern's effectiveness depends heavily on how and where it's used. Should be applied when all of the following are true: an application uses a large number of objects, storage costs are high because of the sheer quantity of objects, most object state can be made extrinsic, many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed, the application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Structure (Figure B.12)

Figure B.12: *Flyweight* pattern structure.

Participants **Flyweight** - declares an interface through which flyweights can receive and act on extrinsic state. **ConcreteFlyweight** - implements the **Flyweight** interface and adds storage for intrinsic state, if any. A **ConcreteFlyweight** object must be sharable. Any state it stores must be intrinsic, i.e., it must be independent of the **ConcreteFlyweight** object's context. **UnsharedConcreteFlyweight** - not all **Flyweight** subclasses need to be shared. The **Flyweight** interface enables sharing but doesn't enforce it. It's common for **UnsharedConcreteFlyweight** objects to have **ConcreteFlyweight** objects as children at some level in the flyweight object structure. **FlyweightFactory** - creates and manages flyweight objects, ensures that flyweights are shared properly. When a client requests a flyweight, the **FlyweightFactory** object supplies an existing instance or creates one, if none exists. **Client** - maintains a reference to flyweight(s), and computes or stores the extrinsic state of flyweight(s).

Collaborations State that a flyweight needs to function must be characterized as either intrinsic or extrinsic. Clients should not instantiate **ConcreteFlyweights** directly.

Consequences Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. The more flyweights are shared, the greater the storage savings. The **Flyweight** pattern is often combined with the **Composite** pattern to represent a hierarchical structure as a graph with shared leaf nodes.

Implementations Removing extrinsic state. Managing shared objects.

Known uses The concept of flyweight objects was first described and explored as a design technique in *InterViews 3.0*. *ET++* uses flyweights to support look-and-feel independence. For each widget class there is a corresponding **Layout** class (e.g., **ScrollbarLayout**, **MenuBarLayout**, etc.). The **Layout** objects are created and managed by **Look** objects.,

Related patterns **Composite**, **State**, **Strategy**.

Forces Efficiency, Decoupling, Abstraction, Performance, Overkill

Example resolved (Figure B.13)

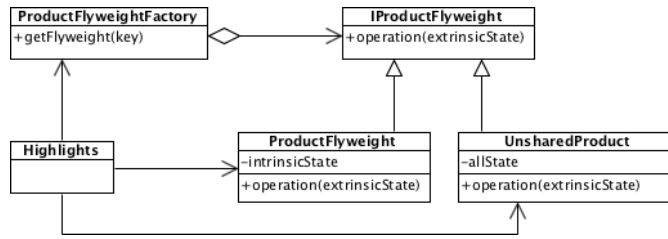


Figure B.13: *Flyweight* pattern instance.

B.4.7 Singleton

Pattern name and classification *Singleton*, creational design pattern.

Intent Ensure a class only has one instance, and provide a global point of access to it.

Also known as N/A

Motivation It's important for some classes to have exactly one instance. For instance, although there can be many printers in a system, there should be only one printer spooler.

Applicability There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point. When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure (Figure B.14)

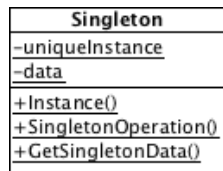


Figure B.14: *Singleton* pattern structure.

Participants Singleton - defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++). May be responsible for creating its own unique instance.

Collaborations Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences Controlled access to sole instance. Reduced name space. Permits refinement of operations and representation. Permits a variable number of instances. More flexible than class operations.

Implementations Ensuring a unique instance. Subclassing the Singleton class.

Known uses An example of the Singleton pattern in Smalltalk-80 is the set of changes to the code, which is ChangeSet current. The InterViews user interface toolkit uses the Singleton pattern to access the unique instance of its Session and WidgetKit classes, among others.

Related patterns Abstract Factory, Builder, Prototype.

Forces Performance, Indirection, Efficiency, Coupling, Overkill.

Example resolved (Figure B.15)

User
<u>-uniqueInstance</u>
<u>-data</u>
+Instance()
+SingletonOperation()
+GetSingletonData()

Figure B.15: Singleton pattern instance.

B.5 Requirement Pattern to Software Pattern Matching Information

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY untitled-ontology-62 "http://www.url.com/mapping#" >
  <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="http://www.url.com/mapping#"
  xml:base="http://www.url.com/mapping"
  xmlns:untitled-ontology-62="http://www.url.com/mapping#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <owl:Ontology rdf:about="http://www.url.com/mapping"/>

  <!--
  //////////////////////////////////////
  //
  // Object Properties
  //
  //////////////////////////////////////
  -->
  <!-- http://www.url.com/mapping#hasConcern -->
  <owl:ObjectProperty rdf:about="&untitled-ontology-62;hasConcern"/>
```

```

<!-- http://www.url.com/mapping#hasForce -->
<owl:ObjectProperty rdf:about="&untitled-ontology-62;hasForce"/>

<!-- http://www.url.com/mapping#hasForceN -->
<owl:ObjectProperty rdf:about="&untitled-ontology-62;hasForceN"/>

<!-- http://www.url.com/mapping#hasForceP -->
<owl:ObjectProperty rdf:about="&untitled-ontology-62;hasForceP"/>

<!-- http://www.url.com/mapping#hasGoal -->
<owl:ObjectProperty rdf:about="&untitled-ontology-62;hasGoal"/>

<!--
////////////////////////////////////
//
// Classes
//
////////////////////////////////////
-->
<!-- http://www.url.com/mapping#Concern -->
<owl:Class rdf:about="&untitled-ontology-62;Concern"/>

<!-- http://www.url.com/mapping#Force -->
<owl:Class rdf:about="&untitled-ontology-62;Force"/>

<!-- http://www.url.com/mapping#Goal -->
<owl:Class rdf:about="&untitled-ontology-62;Goal"/>

<!-- http://www.url.com/mapping#RequirementPattern -->
<owl:Class rdf:about="&untitled-ontology-62;RequirementPattern"/>

<!-- http://www.url.com/mapping#SoftwarePattern -->
<owl:Class rdf:about="&untitled-ontology-62;SoftwarePattern"/>

<!--
////////////////////////////////////
//
// Individuals
//
////////////////////////////////////
-->
<!-- http://www.url.com/mapping#Abstraction -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Abstraction">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Aggregation -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Aggregation">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Browseable -->

```

B.5. REQUIREMENT PATTERN TO SOFTWARE PATTERN MATCHING INFORMATION 223

```
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Browseable">
  <rdf:type rdf:resource="&untitled-ontology-62;Concern"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Constraint"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Feeding"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Filtering"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Command -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Command">
  <rdf:type rdf:resource="&untitled-ontology-62;SoftwarePattern"/>
  <rdfs:comment rdf:datatype="&xsd:string"/>/**
* @intent Encapsulate a request as an object, thereby letting you parameterize clients
  with different requests, queue or log requests, and support undoable operations.
* @param client Who triggers the operation.
* @param receiver Knows how to perform the operations associated with carrying out.
* @param invoker Asks the command to carry out the request.
* @param command Declares an interface for executing an operation.
* @param concreteCommand Implements Execute by invoking the corresponding operation on
  Receiver.
*/</rdfs:comment>
  <hasGoal rdf:resource="&untitled-ontology-62;Process"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Compose -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Compose">
  <rdf:type rdf:resource="&untitled-ontology-62;Goal"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Aggregation"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Indirection"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Memory"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Nesting"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Simplicity"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Composite -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Composite">
  <rdf:type rdf:resource="&untitled-ontology-62;SoftwarePattern"/>
  <rdfs:comment rdf:datatype="&xsd:string"/>/**
* @intent Compose objects into tree structures to represent part-whole hierarchies. Composite
  lets clients treat individual objects and compositions of objects uniformly.
* @param client manipulates objects in the composition through the Component interface.
* @param component defines the interface for the composed objects.
* @param leaf represents a leaf in the composition, without children.
* @param composite represents an element with children of the same type.
*/</rdfs:comment>
  <hasGoal rdf:resource="&untitled-ontology-62;Compose"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Computability -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Computability">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>
```

```

<!-- http://www.url.com/mapping#Concurrency -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Concurrency">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Constraint -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Constraint">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Coupling -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Coupling">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Decoupling -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Decoupling">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Delegate -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Delegate">
  <rdf:type rdf:resource="&untitled-ontology-62;Goal"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Abstraction"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Decoupling"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Overkill"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Direction -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Direction">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Edit -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Edit">
  <rdf:type rdf:resource="&untitled-ontology-62;Goal"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Flexibility"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Memory"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Versioning"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Editable -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Editable">
  <rdf:type rdf:resource="&untitled-ontology-62;Concern"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Constraint"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Flexibility"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Versioning"/>
</owl:NamedIndividual>

```

B.5. REQUIREMENT PATTERN TO SOFTWARE PATTERN MATCHING INFORMATION 225

```

<!-- http://www.url.com/mapping#Efficiency -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Efficiency">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Explore -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Explore">
  <rdf:type rdf:resource="&untitled-ontology-62;Goal"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Aggregation"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Feeding"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Filtering"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Indirection"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Indirection"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Performance"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Feeding -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Feeding">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Filtering -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Filtering">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Flexibility -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Flexibility">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Flyweight -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Flyweight">
  <rdf:type rdf:resource="&untitled-ontology-62;SoftwarePattern"/>
  <rdfs:comment rdf:datatype="&xsd:string"/>/**
* @intent Use sharing to support large numbers of fine-grained objects efficiently.
* @param client Who triggers the request.
* @param flyweightFactory Creates and manages flyweight objects.
* @param flyweight The class of the instances to be managed.
* @param concreteFlyweight Extension of the concrete instances.
* @param unsharedConcreteFlyweight A flyweight not shared.
*/</rdfs:comment>
  <hasGoal rdf:resource="&untitled-ontology-62;Handle"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Handle -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Handle">
  <rdf:type rdf:resource="&untitled-ontology-62;Goal"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Coupling"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Indirection"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Overkill"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Performance"/>

```

```

</owl:NamedIndividual>

<!-- http://www.url.com/mapping#HasAccount -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;HasAccount">
  <rdf:type rdf:resource="&untitled-ontology-62;RequirementPattern"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Shareable"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#HasCatalog -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;HasCatalog">
  <rdf:type rdf:resource="&untitled-ontology-62;RequirementPattern"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Browseable"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Manageable"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#HasDetails -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;HasDetails">
  <rdf:type rdf:resource="&untitled-ontology-62;RequirementPattern"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Recursive"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#HasHighlights -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;HasHighlights">
  <rdf:type rdf:resource="&untitled-ontology-62;RequirementPattern"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Shareable"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Viewable"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#HasSearch -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;HasSearch">
  <rdf:type rdf:resource="&untitled-ontology-62;RequirementPattern"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Manageable"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Viewable"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#HasShoppingCart -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;HasShoppingCart">
  <rdf:type rdf:resource="&untitled-ontology-62;RequirementPattern"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Editable"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Manageable"/>
  <hasConcern rdf:resource="&untitled-ontology-62;Processable"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Indirection -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Indirection">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Iterator -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Iterator">
  <rdf:type rdf:resource="&untitled-ontology-62;SoftwarePattern"/>
  <rdfs:comment rdf:datatype="&xsd:string">/**
* @intent Provide a way to access the elements of an aggregate objects sequentially without

```

B.5. REQUIREMENT PATTERN TO SOFTWARE PATTERN MATCHING INFORMATION 227

```
        exposing its underlying representation.
* @param client requests the iterator.
* @param iterator defines the interface to access the and traverse elements.
* @param concreteIterator implements the iterator.
* @param aggregate defines the interface for creating iterator objects.
* @param concreteAggregate implements the aggregate interface.
*/</rdfs:comment>
    <hasGoal rdf:resource="&untitled-ontology-62;Explore"/>
  </owl:NamedIndividual>

  <!-- http://www.url.com/mapping#Manageable -->
  <owl:NamedIndividual rdf:about="&untitled-ontology-62;Manageable">
    <rdf:type rdf:resource="&untitled-ontology-62;Concern"/>
    <hasForceP rdf:resource="&untitled-ontology-62;Abstraction"/>
    <hasForceN rdf:resource="&untitled-ontology-62;Coupling"/>
    <hasForceP rdf:resource="&untitled-ontology-62;Decoupling"/>
    <hasForceP rdf:resource="&untitled-ontology-62;Indirection"/>
    <hasForceN rdf:resource="&untitled-ontology-62;Indirection"/>
  </owl:NamedIndividual>

  <!-- http://www.url.com/mapping#Memento -->
  <owl:NamedIndividual rdf:about="&untitled-ontology-62;Memento">
    <rdf:type rdf:resource="&untitled-ontology-62;SoftwarePattern"/>
    <rdfs:comment rdf:datatype="&xsd:string"/>/**
* @intent Without violating encapsulation, capture and externalize an object?s internal state
    so that the object can be restored to this state later.
* @param originator Who triggers the request for creating states.
* @param memento Stores internal state of the Originator object.
* @param caretaker The responsible for the memento?s safekeeping.
* @param state The representation of the state being keep.
* @param item The items belonging to the state.
*/</rdfs:comment>
    <hasGoal rdf:resource="&untitled-ontology-62;Edit"/>
  </owl:NamedIndividual>

  <!-- http://www.url.com/mapping#Memory -->
  <owl:NamedIndividual rdf:about="&untitled-ontology-62;Memory">
    <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
  </owl:NamedIndividual>

  <!-- http://www.url.com/mapping#Nesting -->
  <owl:NamedIndividual rdf:about="&untitled-ontology-62;Nesting">
    <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
  </owl:NamedIndividual>

  <!-- http://www.url.com/mapping#Overkill -->
  <owl:NamedIndividual rdf:about="&untitled-ontology-62;Overkill">
    <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
  </owl:NamedIndividual>

  <!-- http://www.url.com/mapping#Performance -->
  <owl:NamedIndividual rdf:about="&untitled-ontology-62;Performance">
    <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
  </owl:NamedIndividual>
```

```

</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Process -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Process">
  <rdf:type rdf:resource="&untitled-ontology-62;Goal"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Computability"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Coupling"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Indirection"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Overkill"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Processable -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Processable">
  <rdf:type rdf:resource="&untitled-ontology-62;Concern"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Computability"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Coupling"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Indirection"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Proxy -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Proxy">
  <rdf:type rdf:resource="&untitled-ontology-62;SoftwarePattern"/>
  <rdfs:comment rdf:datatype="&xsd:string">/**
* @intent Provide a surrogate or placeholder for another object to control access to it.
* @param client Who triggers the request.
* @param subject Defines the common interface for RealSubject and Proxy.
* @param realSubject Defines the real object that the proxy represents.
* @param proxy Interface which handles the request.
*/</rdfs:comment>
  <hasGoal rdf:resource="&untitled-ontology-62;Delegate"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Recursive -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Recursive">
  <rdf:type rdf:resource="&untitled-ontology-62;Concern"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Indirection"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Nesting"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Separation -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Separation">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Shareable -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Shareable">
  <rdf:type rdf:resource="&untitled-ontology-62;Concern"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Direction"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Memory"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Overkill"/>
</owl:NamedIndividual>

```


B.5. REQUIREMENT PATTERN TO SOFTWARE PATTERN MATCHING INFORMATION 229

```

<!-- http://www.url.com/mapping#Sharing -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Sharing">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Simplicity -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Simplicity">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Singleton -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Singleton">
  <rdf:type rdf:resource="&untitled-ontology-62;SoftwarePattern"/>
  <rdfs:comment rdf:datatype="&xsd:string">/**
* @intent Ensure a class only has one instance, and provide a global point of access to it.
* @param singleton creates and encapsulate the single object instance.
*/</rdfs:comment>
  <hasGoal rdf:resource="&untitled-ontology-62;Unified"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Unified -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Unified">
  <rdf:type rdf:resource="&untitled-ontology-62;Goal"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Direction"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceN rdf:resource="&untitled-ontology-62;Indirection"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Memory"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Versioning -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Versioning">
  <rdf:type rdf:resource="&untitled-ontology-62;Force"/>
</owl:NamedIndividual>

<!-- http://www.url.com/mapping#Viewable -->
<owl:NamedIndividual rdf:about="&untitled-ontology-62;Viewable">
  <rdf:type rdf:resource="&untitled-ontology-62;Concern"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Efficiency"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Overkill"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Performance"/>
  <hasForceP rdf:resource="&untitled-ontology-62;Simplicity"/>
</owl:NamedIndividual>
</rdf:RDF>

```

Listing B.5: OWL matching information.

B.6 Diagrams used in the validation of the produced solution

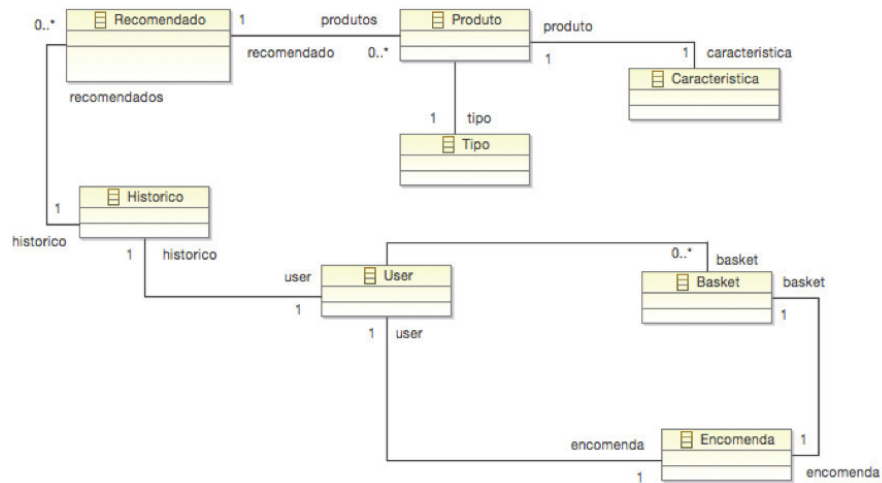


Figure B.16: Diagram A.

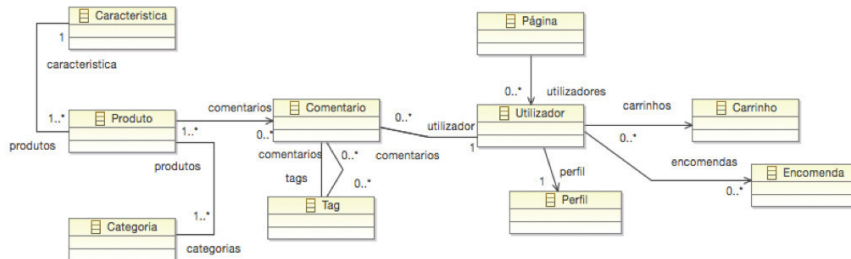


Figure B.17: Diagram B.

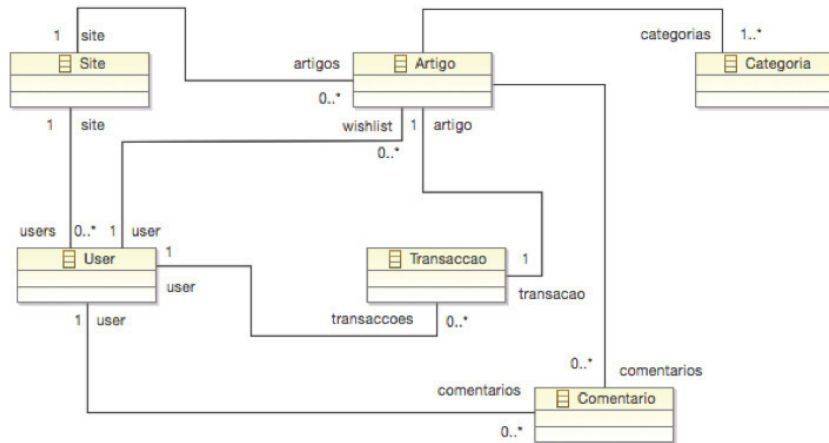


Figure B.18: Diagram D.

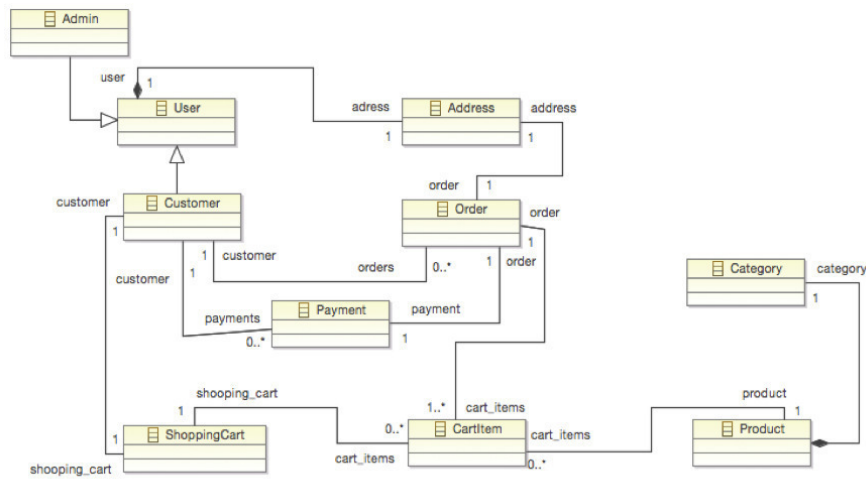


Figure B.19: Diagram E.

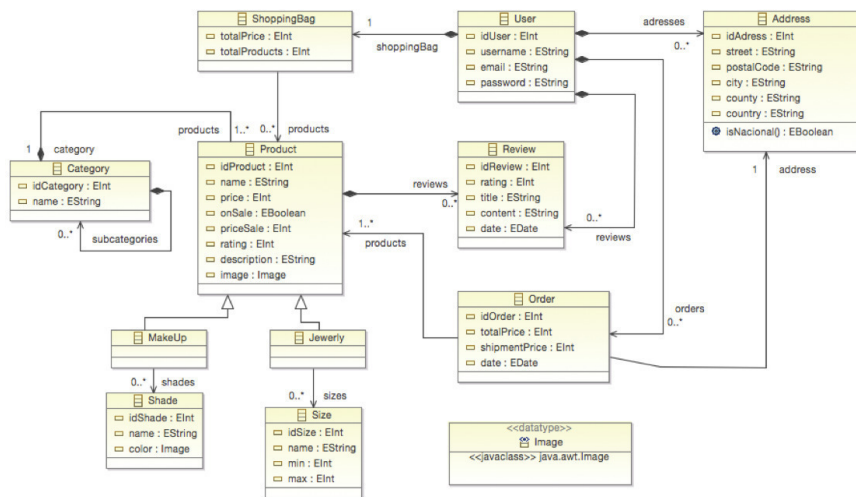


Figure B.20: Diagram F.

Bibliography

- [1] Christopher Alexander. *The timeless way of building*, volume 1. Oxford University Press, 1979.
- [2] Marcelo Arenas and Jorge Pérez. Querying semantic web data with sparql. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 305–316, New York, NY, USA, 2011. ACM.
- [3] C. Arora, M. Sabetzadeh, L. Briand, F. Zimmer, and R. Gnaga. Automatic checking of conformance to requirement boilerplates via text chunking: An industrial case study. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 35–44, Oct 2013.
- [4] Len Bass, Bonnie John, and Jesse Kates. Achieving usability through software architecture. Technical Report CMU/SEI-2001-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2001.
- [5] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.
- [6] Herbert D Benington. Production of large computer programs. In *ICSE*, volume 87, pages 299–310, 1987.
- [7] Robert Biddle, James Noble, and Ewan D. Tempero. Essential use cases and responsibility in object-oriented development. In *Computer Science 2002, Twenty-Fifth Australasian Computer Science Conference (ACSC2002), Monash University, Melbourne, Victoria, January/February 2002*, pages 7–16, 2002.
- [8] Kurt Bittner. *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [9] B. Boehm and H. In. Identifying quality-requirement conflicts. *IEEE Software*, 13(2):25–35, Mar 1996.
- [10] B.W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

- [11] Pierre Bourque and Richard E. Fairley, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOOK v3.0*. IEEE CS, 2014 version edition, 2014.
- [12] John Brooke. SUS-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.
- [13] John Brooke. SUS: a retrospective. *Journal of Usability Studies*, 8(2):29–40, 2013.
- [14] Aleksandar Bulajic, Radoslav Stojic, and Samuel Sambasivam. The generalized requirement approach for requirement validation with automatically generated program code. *Interdisciplinary Journal of Information, Knowledge, and Management*, 9, 2014.
- [15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [16] J. C. Campos and M. D. Harrison. Systematic analysis of control panel interfaces using formal tools. In *XVth International Workshop on the Design, Verification and Specification of Interactive Systems (DSV-IS 2008)*, number 5136 in Lecture Notes in Computer Science, pages 72–85. Springer-Verlag, July 2008.
- [17] Lawrence Chung, Julio Cesar, and Sampaio Prado Leite. Non-functional requirements in software engineering, 1999.
- [18] Lawrence Chung and Brian A. Nixon. Dealing with non-functional requirements: Three experimental studies of a process-oriented approach. In *Proceedings of the 17th International Conference on Software Engineering, ICSE '95*, pages 25–37, New York, NY, USA, 1995. ACM.
- [19] Software Freedom Conservancy. ArgoUML website, <http://argouml.tigris.org/>, July 2011.
- [20] Larry L Constantine and Lucy AD Lockwood. *Software for use: a practical guide to the models and methods of usage-centered design*. Pearson Education, 1999.
- [21] Rui Couto, António Manuel Nestor Ribeiro, and José Francisco Creissac Freitas de Campos. The modelery: a model-based software development repository. *IJWIS*, 11(2):205–225, 2015.
- [22] Rui Couto, Antonio Nestor Ribeiro, and José Creissac Campos. Mapit: A model based pattern recovery tool. In *Model-Based Methodologies for Pervasive and Embedded Software, 8th International Workshop, MOMPES 2012, Essen, Germany, September 4, 2012. Revised Papers*, pages 19–37, 2012.
- [23] Rui Couto, Antonio Nestor Ribeiro, and José Creissac Campos. A patterns based reverse engineering approach for java source code. In *35th Annual IEEE Software Engineering Workshop, SEW 2012, Heraclion, Crete, Greece, October 12-13, 2012*, pages 140–147, 2012.

- [24] Rui Couto, Antonio Nestor Ribeiro, and José Creissac Campos. Application of ontologies in identifying requirements patterns in use cases. In *Proceedings 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2014, Grenoble, France, 12th April 2014.*, pages 62–76, 2014.
- [25] Rui Couto, Antonio Nestor Ribeiro, and José Creissac Campos. The modelery: A collaborative web based repository. In *Computational Science and Its Applications - ICCSA 2014 - 14th International Conference, Guimarães, Portugal, June 30 - July 3, 2014, Proceedings, Part VI*, pages 1–16, 2014.
- [26] Rui Couto, Antonio Nestor Ribeiro, and José Creissac Campos. A study on the viability of formalizing use cases. In *9th International Conference on the Quality of Information and Communications Technology, QUATIC 2014, Guimaraes, Portugal, September 23-26, 2014*, pages 130–133, 2014.
- [27] Rui Couto, António Nestor Ribeiro, and José Creissac Campos. A survey on software patterns. Technical report, HASLab INESC TEC and Universidade do Minho, 2016.
- [28] Rui Couto, Antonio Nestor Ribeiro, and José Creissac Campos. Validating an approach to formalize use cases with ontologies. In *Proceedings of the 13th International Workshop on Formal Engineering Approaches to Software Components and Architectures, FESCA@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016.*, pages 1–15, 2016.
- [29] Rodrigo Cezario da Silva and Fabiane Barreto Vavassori Benitti. Padrões de escrita de requisitos: um mapeamento sistemático da literatura. In *Anais do WER11 - Workshop em Engenharia de Requisitos, Rio de Janeiro-RJ, Brasil, Abril 28-29, 2011*, 2011.
- [30] Rodrigo Cezario da Silva and Fabiane Barreto Vavassori Benitti. Sers: Uma ferramenta de apoio ao reuso de requisitos. Technical report, Universidade do Vale do Itajaí, 2011.
- [31] Deva Kumar Deeptimahanti and Ratna Sanyal. Semi-automatic generation of UML models from natural language requirements. In *Proceeding of the 4th Annual India Software Engineering Conference, ISEC 2011, Thiruvananthapuram, Kerala, India, February 24-27, 2011*, pages 165–174, 2011.
- [32] Diego Dermeval, Jssyka Vilela, IgIbert Bittencourt, Jaelson Castro, Seiji Isotani, Patrick Brito, and Alan Silva. Applications of ontologies in requirements engineering: a systematic review of the literature. *Requirements Engineering*, pages 1–33, 2015.
- [33] Christof Ebert. Putting requirement management into praxis: dealing with nonfunctional requirements. *Information and Software Technology*, 40(3):175 – 185, 1998.
- [34] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool. In *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard*, pages 97–105, 2001.

- [35] Eduardo B Fernandez, Yi Liu, and Rouyi Pan. Patterns for internet shops. *Procs. of PLoP*, 2001.
- [36] Kevin Forsberg and Harold Mooz. The relationship of system engineering to the project cycle. In *INCOSE International Symposium*, volume 1, pages 57–65. Wiley Online Library, 1991.
- [37] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
- [38] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.
- [39] X. Franch, C. Quer, S. Renault, C. Guerlain, and C. Palomares. *Managing Requirements Knowledge*, chapter Constructing and Using Software Requirement Patterns, pages 95–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [40] Xavier Franch. Software requirement patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1499–1501. IEEE Press, 2013.
- [41] E. Gamma. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2004.
- [42] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. Automatically extracting requirements specifications from natural language. *arXiv preprint arXiv:1403.3142*, 2014.
- [43] Gabriella Gigante, Francesco Gargiulo, and Massimo Ficco. A semantic driven approach for requirements verification. In David Camacho, Lars Braubach, Salvatore Ventocinque, and Costin Badica, editors, *Intelligent Distributed Computing VIII*, volume 570 of *Studies in Computational Intelligence*, pages 427–436. Springer International Publishing, 2015.
- [44] Arda Goknil, Ivan Kurtev, and Klaas Berg. A metamodeling approach for reasoning about requirements. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 310–325, Berlin, Heidelberg, 2008. Springer-Verlag.
- [45] Arda Goknil, Ivan Kurtev, Klaas van den Berg, and Jan-Willem Veldhuis. Semantics of trace relations in requirements models for consistency checking and inferencing. *Software and Systems Modeling*, 10:31–54, 2011. 10.1007/s10270-009-0142-3.
- [46] Daniel Gross and Eric Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6:18–36, 2000.
- [47] Thomas R. Gruber. Ontolingua: A mechanism to support portable ontologies. Technical report, Stanford University, 1992.
- [48] Y. G. Guéhéneuc. Ptidej: Promoting Patterns with Patterns. In *proceedings of the 1st ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.

- [49] David Harel and Rami Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. *Software and System Modeling*, 2:2003, 2002.
- [50] Glen Hart, Martina Johnson, and Catherine Dolbear. Rabbit: developing a control natural language for authoring ontologies. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 348–360, Berlin, Heidelberg, 2008. Springer-Verlag.
- [51] Jeff Heflin, James Hendler, and Sean Luke. Shoe: A knowledge representation language for internet applications. Technical report, University of Maryland, 1999.
- [52] S Hekmatpour. Experience with evolutionary prototyping in a large software project. *SIGSOFT Softw. Eng. Notes*, 12(1):38–41, January 1987.
- [53] Axel Hoffmann, Thomas Schulz, Holger Hoffmann, Silke Jandt, Alexander Roßnagel, and Jan Marco Leimeister. Towards the use of software requirement patterns for legal requirements. In *2nd International Requirements Engineering Efficiency Workshop (REEW) 2012*, 2012.
- [54] Axel Hoffmann, Matthias Söllner, and Holger Hoffmann. Twenty software requirement patterns to specify recommender systems that users will trust. In *20th European Conference on Information Systems (ECIS)*, 2012.
- [55] Matthew Horridge and Sean Bechhofer. The owl api: A java api for owl ontologies. *Semantic Web*, 2(1):11–21, 2011.
- [56] Matthew Horridge, Nick Drummond, John Goodwin, Alan L. Rector, Robert Stevens, and Hai Wang. The manchester OWL syntax. In *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA, November 10-11, 2006*, 2006.
- [57] Matthew Horridge and Peter F. Patel-schneider. P.f.: Manchester syntax for owl 1.1. In *In: OWLED 2008, 4th international workshop OWL: Experiences and Directions (2008) Live Extraction 1223*, 2008.
- [58] Haibo Hu, Y Dan, Y Chunxiao, F Chunlei, and L Ren. Detecting interactions between behavioral requirements with owl and swrl. *World Academy of Science, Engineering and Technology*, 72:330–336, 2011.
- [59] Heyuan Huang, Shensheng Zhang, Jian Cao, and Yonghong Duan. A practical pattern recovery approach based on both structural and behavioral analysis. *Journal of Systems and Software*, 75(2):69 – 87, 2005. Software Engineering Education and Training.
- [60] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements engineering*. Springer Science & Business Media, 2010.
- [61] Ivar Jacobson, M Christerson, P Jonsson, and G Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1992.

- [62] Apache Jena. semantic web framework for java, 2007.
- [63] Jens B. Jørgensen, Simon Tjell, and João M. Fernandes. Formal requirements modelling with executable use cases and coloured petri nets. *Innovations in Systems and Software Engineering*, 5(1):13–25, 2009.
- [64] J. Jurkiewicz and J. Nawrocki. Automated events identification in use cases. *Information and Software Technology*, 58(0):110 – 122, 2015.
- [65] M. Kamalrudin and J. Grundy. Generating essential user interface prototypes to validate requirements. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 564 –567, nov. 2011.
- [66] Peter D Karp, Vinay K Chaudhri, and Jerome Thomere. Xol: An xml-based ontology exchange language, 1999.
- [67] Michael Kifer and Georg Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 134–146, New York, NY, USA, 1989. ACM.
- [68] Dae-Kyoo Kim and Lunjin Lu. Inference of design pattern instances in uml models via logic programming. In *Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on*, pages 10 pp.–, 2006.
- [69] Damir Kirasić and Danko Basch. Ontology-based design pattern recognition. In *Proceedings of the 12th international conference on Knowledge-Based Intelligent Information and Engineering Systems, Part I*, KES '08, pages 384–393, Berlin, Heidelberg, 2008. Springer-Verlag.
- [70] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [71] Holger Knublauch, Matthew Horridge, Mark A Musen, Alan L Rector, Robert Stevens, Nick Drummond, Phillip W Lord, Natalya Fridman Noy, Julian Seidenberg, and Hai Wang. The protege owl experience. In *OWLED*, 2005.
- [72] Petr Kroha. Preprocessing of requirements specification. In *Database and Expert Systems Applications, 11th International Conference, DEXA 2000, London, UK, September 4-8, 2000, Proceedings*, pages 675–684, 2000.
- [73] Tobias Kuhn. The understandability of owl statements in controlled english. *Semantic Web*, 4(1), 3 2013.
- [74] Lee W Lacy. *OWL: representing information using the Web Ontology Language*. Trafford Publishing, 2005.

- [75] Mathias Landhäuser, Sven J. Körner, and Walter F. Tichy. From requirements to UML models and back: how automatic processing of text can support requirements engineering. *Software Quality Journal*, 22(1):121–149, 2014.
- [76] James R. Lewis. Sample sizes for usability studies: Additional considerations. Tech. Report 54.711, IBM, 1992, October 1992.
- [77] Ke Li. Towards semi-automation in requirements elicitation: mapping natural language and object-oriented concepts. In *RE05*, pages 5–7, 2005.
- [78] Liwu Li. A semi-automatic approach to translating use cases to sequence diagrams. In *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of*, pages 184–193, Jul 1999.
- [79] Xiaoshan Li, Zhiming Liu, Jifeng He, and Quan Long. Generating a prototype from a uml model of system requirements. In R.K. Ghosh and Hrushiksha Mohanty, editors, *Distributed Computing and Internet Technology*, volume 3347 of *Lecture Notes in Computer Science*, pages 255–265. Springer Berlin Heidelberg, 2005.
- [80] Horst Lichter, Matthias Schneider-Hufschmidt, and Heinz Züllighoven. Prototyping in industrial software projects—bridging the gap between theory and practice. In *Proceedings of the 15th International Conference on Software Engineering, ICSE '93*, pages 221–229, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [81] Dong Liu. *Automating Transition from Use Cases to Class Model*. PhD thesis, University of Calgary, 2003.
- [82] Fernando Lyardet, Gustavo Rossi, and Daniel Schwabe. Patterns for adding search capabilities to web information systems. In *EuroPLOP*, pages 189–202, 1999.
- [83] Robert MacGregor and Raymond Bates. The loom knowledge representation language. Technical report, DTIC Document, 1987.
- [84] Marina Machado, José Campos, and Rui Couto. Modus: uma metodologia de prototipagem de interfaces baseada em modelos. In *INFORUM 2015*, number 1 in 7, pages 17–32, 2015.
- [85] Dewi Mairiza, Didar Zowghi, and Nur Nurmuliani. Towards a catalogue of conflicts among non-functional requirements. In *ENASE 2010 - Proceedings of the Fifth International Conference on Evaluation of Novel Approaches to Software Engineering, Athens, Greece, July 22-24, 2010*, pages 20–29, 2010.
- [86] Dewi Mairiza, Didar Zowghi, and Nurie Nurmuliani. Managing conflicts among non-functional requirements. In *Australian Workshop on Requirements Engineering*. University of Technology, Sydney, 2009.
- [87] G. S. Anandha Mala and G. V. Uma. Automatic construction of object oriented design models [UML diagrams] from natural language requirements specification. In *PRICAI*

- 2006: *Trends in Artificial Intelligence, 9th Pacific Rim International Conference on Artificial Intelligence, Guilin, China, August 7-11, 2006, Proceedings*, pages 1155–1159, 2006.
- [88] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [89] Enrico Motta. An overview of the ocml modelling language. In *the 8th Workshop on Methods and Languages*. Citeseer, 1998.
- [90] Martin O’Connor, Samson Tu, Csongor Nyulas, Amar Das, and Mark Musen. Querying the semantic web with swrl. In *Proceedings of the 2007 international conference on Advances in rule interchange and applications*, RuleML’07, pages 155–159, Berlin, Heidelberg, 2007. Springer-Verlag.
- [91] OMG. MDA Guide Version 1.0.1. <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>, June 2003.
- [92] S.P. Overmyer, L. Benoit, and R. Owen. Conceptual modeling through linguistic analysis using lida. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 401–410, May 2001.
- [93] C. Palomares, C. Quer, X. Franch, S. Renault, and C. Guerlain. A catalogue of functional software requirement patterns for the domain of content management systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC ’13*, pages 1260–1265, New York, NY, USA, 2013. ACM.
- [94] Roger S Pressman. *Software engineering: a practitioner’s approach*. Palgrave Macmillan, 2005.
- [95] Whitney Quesenbery. Dimensions of usability: Defining the conversation, driving the process. In *UPA 2003 Conference*, 2003.
- [96] Ghulam Rasool, Patrick Maeder, and Ilka Philippow. Evaluation of design pattern recovery tools. *Procedia Computer Science*, 3(0):813 – 819, 2011. World Conference on Information Technology.
- [97] K. Roher and D. Richardson. Sustainability requirement patterns. In *Requirements Patterns (RePa), 2013 IEEE Third International Workshop on*, pages 8–11, July 2013.
- [98] Doug Rosenberg and Matt Stephens. *Use Case Driven Object Modeling with UML Theory and Practice*. Apress, Berkely, CA, USA, 2nd edition, 2013.
- [99] Approved September. Ieee standard glossary of software engineering terminology. *Office*, 121990(1):1, 1990.
- [100] Nija Shi and R.A. Olsson. Reverse engineering of design patterns from java source code. In *Automated Software Engineering, 2006. ASE ’06. 21st IEEE/ACM International Conference on*, pages 123–134, Sept 2006.

- [101] V. Simko, D. Hauzar, P. Hnetyuka, T. Bures, and F. Plasil. Formal verification of annotated textual use-cases. *The Computer Journal*, sep 2014.
- [102] Evren Sirin, Blazej Bulka, and Michael Smith. Terp: Syntax for owl-friendly sparql queries. In *OWLED*, 2010.
- [103] Michal Smialek and Wiktor Nowakowski. *From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice*. Springer, 2015.
- [104] Barry Smith. Ontology. In Luciano Floridi, editor, *Blackwell Guide to the Philosophy of Computing and Information*, pages 155–166. Blackwell, 2003.
- [105] Amina Souag, Camille Salinesi, Ral Mazo, and Isabelle Comyn-Wattiau. A security ontology for security requirements elicitation. In Frank Piessens, Juan Caballero, and Nataliia Bielova, editors, *Engineering Secure Software and Systems*, volume 8978 of *Lecture Notes in Computer Science*, pages 157–177. Springer International Publishing, 2015.
- [106] Miroslaw Staron. *Adopting Model Driven Software Development in Industry – A Case Study at Two Companies*, pages 57–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [107] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
- [108] Kalaivani Subramaniam, Dong Liu, Behrouz Homayoun Far, and Armin Eberlein. UCDA: use case driven development assistant tool for class model generation. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering, Banff*, pages 324–329, 2004.
- [109] Sara Tena, David Díez, Paloma Díaz, and Ignacio Aedo. Standardizing the narrative of use cases: A controlled vocabulary of web user tasks. *Information & Software Technology*, 55(9):1580–1589, 2013.
- [110] S. Tiwari, S.S. Rathore, S. Gupta, V. Gogate, and A. Gupta. Analysis of use case requirements using sfta and sfinet techniques. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, pages 29–38, July 2012.
- [111] Saurabh Tiwari and Atul Gupta. Does increasing formalism in the use case template help? In *Proceedings of the 7th India Software Engineering Conference, ISEC '14*, pages 6:1–6:10, New York, NY, USA, 2014. ACM.
- [112] MinhTu Ton That, Salah Sadou, Flavio Oquendo, and Isabelle Borne. Composition-centered architectural pattern description language. In Khalil Drira, editor, *Software Architecture*, volume 7957 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2013.

- [113] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 230–238, 1999.
- [114] I. Tounsi, M. H. Kacem, A. H. Kacem, and K. Drira. An approach for soa design patterns composition. In *2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 219–226, Oct 2015.
- [115] L. Valente and B. Feijo. Extending use cases to support activity design in pervasive mobile games. In *Computer Games and Digital Entertainment (SBGAMES), 2014 Brazilian Symposium on*, pages 193–201, Nov 2014.
- [116] Martijn van Welie. welie.com, April 2015.
- [117] Pablo Martin Vera. Component based model driven development: An approach for creating mobile web applications from design models. *International Journal of Information Technologies and Systems Approach (IJITSA)*, 8(2):80–100, 2015.
- [118] K. Wiegers and J. Beatty. *Software Requirements*. Developer Best Practices. Pearson Education, 2013.
- [119] Karl Wiegers and Joy Beatty. *Software Requirements*. Microsoft Press, third edition, 2013.
- [120] Stephen Withall. *Software requirement patterns*. Pearson Education, 2007.
- [121] SherifM. Yacoub and HanyH. Ammar. Uml support for designing software systems as a composition of design patterns. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 149–165. Springer Berlin Heidelberg, 2001.
- [122] S.M. Yacoub and H.H. Ammar. *Pattern-oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley, 2004.
- [123] Li Yan. *E-Commerce Patterns*. PhD thesis, Carleton University, 2004.
- [124] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. *Urbana*, 51:61801, 1998.
- [125] Tao Yue, Lionel C. Briand, and Yvan Labiche. atoucan: An automated framework to derive UML analysis models from use case models. *ACM Trans. Softw. Eng. Methodol.*, 24(3):13, 2015.
- [126] Roberto Zen. *Gherkin and Cucumber*. PhD thesis, University of Trento, 2013.