RIGA TECHNICAL UNIVERSITY
Faculty of Computer Science and Information Technology
Institute of Applied Computer Systems

Uldis DONIŅŠ
Student of the Doctoral Study Program "Computer Systems"

TOPOLOGICAL UNIFIED MODELING LANGUAGE: DEVELOPMENT AND APPLICATION

Doctoral Thesis

Scientific Supervisor
Dr.habil.sc.ing., Professor
J. OSIS

Riga 2012

ANOTĀCIJA

Neskatoties uz to, ka pastāv kopums ar programmatūras modelēšanas valodām (ieskaitot vienoto modelēšanas valodu UML) un šo valodu atbalstošām programmatūras izstrādes metodēm, programmatūras izstrādes līmenis joprojām ir saglabājies pietiekami zems (izejot no tā, ka: lielu programmatūru izstrādes nereti tiek atceltas; tiek pārtērēts izstrādei atvēlētais budžets un laiks un rezultātā iegūtās programmatūras kvalitāte ir neapmierinoša). Šādam faktam par iemeslu ir problēmvides funkcionēšanas pilnīga vai daļēja ignorēšana programmatūras izstrādes laikā (vairākumā gadījumu tā pastāv un eksistē atdalīti no izveidotā risinājuma), tādejādi veicinot plaisas izveidošanos starp problēmvidi un risinājumu. Šī plaisa galvenokārt veidojas tādēļ, ka netiek veltīta pienācīga uzmanība tam, lai izanalizētu un saprastu problēmvides funkcionēšanu. Atsevišķos gadījumos programmatūra tiek izstrādāta tā, kā to redz un iedomājas tās izstrādātājs, nevis kā to nosaka problēmvides darbības īpatnības. Plaisa starp problēmvidi un piegādāto risinājumu neļauj viennozīmīgi izsekot cēloņseku attiecībām kā problēmvidē, tā piegādātajā risinājumā. Nespējot izsekot šīm cēloņseku attiecībām abās vidēs, programmatūras akceptēšanas process zaudē savu jēgu un nozīmi – programmatūras pasūtītājam nav iespēju pārbaudīt piegādāto risinājumu, to validējot attiecībā pret problēmvidi.

Promocijas darba „*Topoloģiskā vienotā modelēšanas valoda: izstrāde un lietošana*" mērķis ir izpētīt UML valodu un tās lietošanas metodes programmatūras izstrādes procesa nodrošināšanai, rezultātā piedāvājot uzlabotu UML valodas versiju – Topoloģisko vienoto modelēšanas valodu TopUML – un tās izmantošanas metodi formāla programmatūras izstrādes procesa nodrošināšanai un viennozīmīgai cēloņseku atsekojamībai gan problēmvidē, gan programmatūrā.

Promocijas darba ietvaros izstrādātā valoda un tās lietošanas metode ir aprobēta eksperimentālā programmatūras projektēšanas projektā un parauga programmatūras izstrādes projektā. Parauga programmatūra ir izstrādāta organizācijas datu sinhronizēšanai no vairākiem datu avotiem uz vienu datu glabātuvi; pašlaik tā ir uzstādīta un tiek ekspluatēta produkcijas vidē, tādejādi apliecinot TopUML un tās lietošanas metodes izmantošanu programmatūras izstrādes projektos.

Šis darbs sastāv no 224 lappusēm, 71 attēla, 32 tabulām, 14 pielikumiem un 134 informācijas avotu nosaukumiem.

ABSTRACT

Despite that in nowadays exists a bunch of software modeling languages (including the Unified Modeling Language (UML)) and methods that consumes such modeling languages, the way software is built remains surprisingly primitive (by meaning that major software applications are cancelled, overrun their budgets and schedules, and often have hazardously bad quality levels when released). This phenomenon can be explained by the fact that the problem domain exists separately from the solution domain (i.e., by not paying appropriate attention to the analysis of the problem domain functioning). Furthermore, in particular cases the software is built as the developers see the solution and not as the problem domain functions. By reducing or even avoiding proper analysis of the problem domain, the traces between problem and solution domain cannot be established. Without traces between both domains the acceptance process of developed software gets meaningless since the customer cannot fully verify the delivered solution (in terms of relating problem domain to the software).

The main purpose of this doctoral thesis titled "*Topological Unified Modeling Language: Development and Application*" is researching UML and software development methods that support and promote the application of UML. The aim of the research is to create a new version of UML that can be applied in a formal way which allows clearly tracing cause-and-effect relationships between the problem and solution domains.

To achieve the goal of this thesis a new language is developed – Topological Unified Modeling Language (TopUML) – and its supporting software development method – TopUML modeling. The TopUML includes elements that allow formally relating problem and solution domains while the TopUML modeling promotes formal software development process. In short – TopUML modeling ensures that software artifacts are created in a strong conformance with the functioning of the problem domain. The developed language and TopUML modeling method are applied in an experimental project and in a case study project. The experimental project includes software designing for laundry problem domain at the computation and platform independent viewpoints, while during the case study project enterprise data synchronization software is designed and developed which currently operates in production environment.

The doctoral thesis contains 224 pages, 71 figure, 32 tables, and 14 appendices. Bibliography includes 134 information sources.

TABLE OF CONTENTS

Introduction 7
Motivation of the Research 7
Research Area 8
Purpose of the Research 10
Scientific Innovation and Practical Value 11
Approbation of the Work Results 12
Thesis Outline 14
1. Unified Modeling Language – a Standard for Software Design Specification 17
1.1. UML Diagrams 19
1.1.1. Structure Diagrams 20
1.1.2. Behavior Diagrams 22
1.2. Formalism of UML 24
1.2.1. Formalism of UML Version 1.x 24
1.2.2. Formalism of UML Version 2.x 26
1.2.3. The Need of Additional UML Formalization 28
1.2.4. Current UML Formalization Attempts 29
1.3. Benefits of Applying UML 30
1.4. Disadvantages of Applying UML 31
1.5. UML Improvement Options 32
1.5.1. UML Extensibility Mechanisms 33
1.5.2. Improving UML by using Topology 34
1.6. Summary 35
2. Software Designing with UML Modeling Driven Approaches 37
2.1. Current State of the Art 38
2.1.1. Object-Oriented Analysis and Design with Unified Process 39
2.1.2. Business Object-Oriented Modeling 42
2.1.3. Pattern Based Software Design 44
2.1.4. Conceptual Modeling 47
2.1.5. Component Based Development 49
2.1.6. Topological Functioning Modeling for Model Driven Architecture 52
2.1.7. UML Software Design with Microsoft Tools 56

2.2. Benefits and Limitations of UML Modeling Driven Approaches 58
2.3. Summary 61
3. Improving Unified Modeling Language 64
3.1. Profiling UML and Metamodeling 65
3.1.1. Overview of UML Profiles 68
3.2. Developing a Profile for UML 73
3.2.1. Profile Specification Template 75
3.3. Topological Unified Modeling Language – an UML Improvement 76
3.3.1. Topology in UML Diagrams 77
3.3.2. TopUML Diagrams 80
3.3.3. TopUML Profile 81
3.3.4. Metamodels of TopUML Diagrams 86
3.4. Summary 95
4. TopUML Modeling – a Method for Designing Software 97
4.1. Problem Domain Functioning Analysis 101
4.1.1. Topological Space Development 102
4.1.2. Initial Topological Functioning Model Development 103
4.1.3. Refining Topological Functioning Model 109
4.2. Behavior Analysis and Design 112
4.2.1. Use Case Analysis and Design 112
4.2.2. Messages of Objects and their Sequence Analysis and Design 114
4.2.3. Workflow Analysis and Design 116
4.2.4. Workflows and Messaging Design 117
4.3. Structure Analysis and Design 118
4.3.1. Analysis of Objects and their Communication 118
4.3.2. Domain Model Development by Means of Topological Class Diagram 120
4.3.3. Refinement of Topological Class Diagram 122
4.3.4. Modeling System Snapshots 127
4.4. Object State Change and Transition Analysis 128
4.5. Structuring Logical Layout of Software Design 129
4.6. Components and Deployment Design 130
4.7. TopUML Modeling in Comparison with other Modeling Methods 131
4.8. Summary 137
5. Approbation of TopUML Profile and Modeling Method 139

5.1. Business Support Application Development 139
5.1.1. Specification of Laundry System 140
5.1.2. Problem Domain Functioning Analysis 140
5.1.3. Behavior Analysis and Design 147
5.1.4. Structure Analysis and Design 149
5.2. Enterprise Data Synchronization System Development 151
5.2.1. Specification of Enterprise Data Synchronization System 152
5.2.2. Problem Domain Functioning Analysis 153
5.2.3. Behavior Analysis and Design 155
5.2.4. Structure Analysis and Design 158
5.2.5. State Change and Transition Analysis 160
5.3. Empirical Evaluation of TopUML Profile and Modeling Method 162
5.3.1. Goal and Process of Empirical Experiment 162
5.3.2. Participants 164
5.3.3. Results of Experiment 165
5.4. Summary 167
Conclusions 169
Appendices 172
Appendix 1 173
Appendix 2 174
Appendix 3 186
Appendix 4 191
Appendix 5 194
Appendix 6 196
Appendix 7 200
Appendix 8 201
Appendix 9 206
Appendix 10 207
Appendix 11 208
Appendix 12 210
Appendix 13 212
Appendix 14 216
Bibliography 217

INTRODUCTION

The analysis of problem domain and design of desired solution within software development process has a major impact of the achieved result – developed software. While the software developer community uses a set of tools and different techniques to create detailed specification of the solution, the proper analysis of problem domain functioning is ignored or covered insufficiently. One of such techniques is object-oriented software analysis and development which states that there are two fundamental aspects of systems modeling: analysis and design. The analysis defines what the solution needs to do within the problem domain to fit the customer's requirements, and the design states how the solution will be implemented. The design of object-oriented software for the last decade has been leaded by the Unified Modeling Language (UML). UML is an approved industry standard modeling notation for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system [89]. While the UML has elements for designing and specifying artifacts of a software system, it lacks the ability to document the functioning of a problem domain by using computation independent constructs. Since the UML is a notation and not a technique or method, its application within software analysis and design is promoted by a set of different software development methods and approaches.

Motivation of the Research

Despite that in nowadays exists a bunch of software modeling languages (including the UML approved and promoted by Object Management Group (OMG)) and methods that consumes such modeling languages, the way the software is built remains surprisingly primitive (by meaning that major software applications are cancelled, overrun their budgets and schedules, and often have hazardously bad quality levels when released) as outlined by Jones in [55]. This phenomenon can be explained by the fact that the problem domain exists separately from the solution domain (i.e., by not paying appropriate attention to the analysis of the problem domain functioning). Furthermore, in particular cases the software is built as the developers see the solution and not as the problem domain functions. By reducing or even avoiding proper analysis of the problem domain, the traces between artifacts of problem and solution domains cannot be established. Without these traces the acceptance process of developed software gets meaningless since the customer cannot fully verify the delivered solution (in terms of relating functioning of the problem domain to the delivered software).

While the UML has diagrams that describes system from both static and dynamic viewpoint, the language alone is not sufficient for a successful development and delivery of a software system. It should be combined together with appropriate software modeling method or technique. To motivate software developers to pay more attention on the analysis and understanding of problem domain and its functioning, an appropriate models and their application method should be provided. The research results outlined by Jones has proved that the UML and the existing UML modeling driven methods do not provide such appropriate model and modeling guidelines.

Research Area

The research area in the focus of this research is the topological modeling of system functioning (both – the problem and the solution domain). The topological modeling of system functioning was started in the middle of 1960's in Riga Technical University by Janis Osis. The first theoretical foundations of Topological functioning model (TFM) and its application in the topological modeling of system functioning are represented in [99]. The review of topological modeling of system functioning and its evolution in Riga Technical University can be found in [97].

The initial problem domain in which TFM is applied is the diagnostic of mechanic devices (e.g., motor vehicles) based on cybernetics and computer science. Large number of high-quality algorithms and methods related to the TFM application in diagnostic tasks are summarized in [106]. The application of TFM within different problem domains and areas is developed today as well. In fact, the [96] and [99] propose a new foundation of the system theory. The first research of TFM and the category theory is given in [95] while the perspective research direction in the field of software engineering is started in [98] related to the microprogramming optimization.

Topological modeling of system functioning has been successfully applied in the field of medical problems solving and diagnostics by Zigurds Markovics since beginning of 1970's [72]. The work of Markovics is extended with mathematical model composition of elements mini-models [74] and expert knowledge [75]. By continuing to introduce topological modeling of system functioning into medicine the [56] and [73] successfully illustrates its application for therapy selection.

The research work continued by Janis Grundspenkis initially was related to investigations of cycle hierarchies for the purpose of rational diagnostic algorithm

development [45], [50]. Later the topological modeling research direction by Grundspenkis is evolved as structural modeling [49] which is developed to support systematic causal domain model based knowledge acquisition. The knowledge base is divided into two parts: a topological knowledge base and a deep knowledge rule base. The topological knowledge base supports reasoning in logic thus allowing to make decisions about relationships between elements and functioning of a system in normal conditions. The deep knowledge rule base supports diagnostic and predictive reasoning if backward or forward chaining is executed. Structural modeling is applied in two problem domains: early stages of design (to find the structure of the system with physically heterogeneous elements), and technical diagnosis (to acquire model-based conclusions about the functioning of a system under component fault conditions and to make conclusions about failure propagation as well as about causes of parameter value changes) [46]. The essence of structural modeling is the systematic procedure for construction of three structural (i.e., topological) models representing the morphology, functions and behavior of complex technical system [47]. Structural modeling approach has been implemented in automated structural modeling system ASMOS [48]. An agent-based modeling is presented in [131] in order to strengthen dynamic relations between the modeled system's objects.

The topological modeling of system functioning has proved itself as a powerful instrument in the domain of embedded systems where the analysis is done by applying topological function–architecture co-design method [107]. It combines principles of co-design and Model Driven Architecture (MDA) and introduces explicit architectural and computation independent models and formalizes their evolution.

The research direction continued by Osis is related to TFM application in the field of object-oriented analysis and object-oriented software development. Recent research results are published as follows: topological modeling application for business process modeling and simulation [109], TFM application in the software development for mechatronic and embedded systems [93], introducing more formalism in problem domain analysis ([5], [6], [28], and [26]), formal analysis of Computation Independent Model (CIM) within MDA ([94], [100], [104], and [124]), formally specifying Platform Independent Model (PIM) and performing transformation CIM-to-PIM within MDA ([29] and [31]). The theoretical foundations of TFM are summarized and published in monograph [101] where the definitions of TFM are given and the powerfulness of TFM is demonstrated in the context of formal problem domain analysis. This research is the continuation of topological modeling of system functioning evolution within the field of object-oriented analysis and software development.

Purpose of the Research

The goal of the thesis is to supplement UML with theoretical foundations in order to create grounds for converting notation into a formal modeling language and to define modeling method which allows to clearly trace cause-and-effect relationships in both problem and solution domains.

The tasks of thesis in order to achieve the goal are defined as follows:

1. Explore the evolution of UML and its specification in order to outline the positive and negative aspects of the current language's version application within software development thus identifying aspects of UML that should be improved,
2. Identify UML extension mechanisms and options in order to determine the best suitable extension mechanism to implement the extended version of UML,
3. Analyze the main characteristics of a set of UML modeling driven software development approaches and compare their potentialities in formalizing the problem domain and creating solution domain design in accordance with the functioning characteristics of problem domain,
4. Develop template for specifying an UML profile,
5. Develop a new profile – Topological UML (TopUML) – in accordance with the identified aspects of UML that should be improved and according to the developed UML profile specification template,
6. Specify software development method that supports formal application of created TopUML profile that allows to clearly trace cause-and-effect relationships in both problem and solution domains,
7. Approbate the developed profile and its application method in experimental software development project involving into software design process several groups of software development experts, and
8. Approbate the developed profile and its application method in a real software development project providing step-by-step case study exploration.

The research objects are UML and its application methods that support application of UML within software development.

The research subject of the thesis is UML and its application methods, focusing on the formal development of software design models and the establishment of traces between problem domain and solution domain artifacts.

The following **research methods** are used: mathematical model and modeling language to specify problem domain and solution domain, metamodeling method, model transformation, as well as the following parts of mathematic – general topology, combinatory topology, graph theory, and mathematical logic.

Scientific Innovation and Practical Value

The **scientific innovation** of the research is formal modeling of solution domain in strong accordance with the functioning of problem domain by using TopUML and formal software designing and development method which is specially developed to drive the application of TopUML diagrams within software development process. The TopUML together with its modeling method enables less intuitive specification of problem domain and creation of solution domain artifacts. Furthermore, the solution domain artifacts are fully traceable to the problem domain artifacts (and vice versa) thus allowing to evaluate the effect of changes within any of the two domains. Characteristics of the developed TopUML modeling method are compared with the set of currently existing UML modeling driven software development methods and techniques.

The **practical value** of the research is specification of TopUML profile which combines formalism of TFM mathematical topology and specification standard of OMG, and specified software analysis and design method. The developed modeling method enables application of TopUML diagrams in a formal software development process and consists of formally defined designing activities thus enabling solution development in accordance with functioning characteristics of problem domain and clearly tracing cause-and-effect relationships in artifacts of problem and solution domains. Since the TopUML is developed as UML profile, it can be implemented in any existing UML modeling tool that supports definitions of custom profiles.

Specifications, methods and other artifacts created during this research are as follows:

- Specification of TopUML profile in accordance with UML version 2.4.1 specification,
- Definition of TopUML modeling method,
- Monograph that explores step-by-step design development with TopUML modeling for laundry problem domain business support software system, and
- Software designed with TopUML that performs enterprise data synchronization.

Approbation of the Work Results

The main results of the research are presented in the following seven international scientific conferences (two were held in Latvia and five in foreign countries):

1. 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2012), Wroclaw, Poland, June 29-30, 2012
2. 13th International Conference on Enterprise Information Systems (ICEIS 2011), Beijing, China, June 8-11, 2011,
3. 3rd International Workshop on Model-Driven Architecture and Modeling Driven Software Development (MDA & MDSD 2011) in conjunction with 6th International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2011), Beijing, China, June 8-11, 2011,
4. 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development (MDA & MTDD 2010) in conjunction with 5th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2010), Athens, Greece, July 22-24, 2010,
5. 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009), Riga, Latvia, September 7-10, 2009,
6. 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2009), Milano, Italy, May 9-10, 2009, and
7. The 49th Scientific Conference of Riga Technical University, Riga, Latvia, October 13-15, 2008.

The main results of the research are published in the following scientific papers:

1. Donins U. Semantics of Logical Relations in Topological Functioning Model// Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2012) – 2012. (To be published)
2. Donins U., Osis J., Asnina E., Jansone A. Formal Analysis of Objects State Changes and Transitions// Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2012) – 2012. (To be published)
3. Donins U., Osis J. Topological Modeling for Enterprise Data Synchronization System: A Case Study of Topological Model-Driven Software Development// Proceedings of the 13th International Conference on Enterprise Information Systems, Volume 3. - Beijing, China: SciTePress, 2011. - pp. 87-96 [Indexed by Thomson Reuters, Inspec, EI, DBLP, ISTP]

4. Donins U., Osis J., Slihte A., Asnina E., Gulbis B. Towards the Refinement of Topological Class Diagram as a Platform Independent Model// Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 79-88 [Indexed by Thomson Reuters, Inspec, EI, DBLP]
5. Slihte A., Osis J., Donins U., Asnina, E., Gulbis, B. Advancements of the Topological Functioning Model for Model Driven Architecture Approach// Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 91-100 [Indexed by Thomson Reuters, Inspec, EI, DBLP]
6. Asnina E., Gulbis B., Osis J., Alksnis G., Donins U., Slihte A. Backward Requirements Traceability within the Topology-based Model Driven Software Development// Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 36-45 [Indexed by Thomson Reuters, Inspec, EI, DBLP]
7. Slihte A., Osis J., Donins U. Knowledge Integration for Domain Modeling// Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 46-56 [Indexed by Thomson Reuters, Inspec, EI, DBLP]
8. Donins U. Software Development with the Emphasis on Topology// Advances in Databases and Information Systems (Lecture Notes in Computer Science, Vol.5968). - Berlin, Germany: Springer-Verlag, 2010. - pp. 220-228 [Indexed by Springer, SCOPUS, DBLP]
9. Osis J., Donins U. Platform Independent model Development by Means of Topological Class Diagrams// Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development - Portugal: SciTePress, 2010. - pp. 13-22 [Indexed by SCOPUS, Thomson Reuters, Inspec, DBLP]
10. Osis J., Donins U. Formalization of the UML Class Diagrams// Evaluation of Novel Approaches to Software Engineering (Communications in Computer and Information Science (CCIS), Volume 69). - Berlin, Germany: Springer-Verlag, 2010. - pp. 180-192 [Indexed by Springer, SCOPUS]
11. Osis J, Donins U. Modeling Formalization of MDA Software Development at the Very Beginning of Life Cycle// Advances in Databases and Information Systems. 13th East-European Conference, ADBIS 2009: Associated Workshops and Doctoral Consortium,

Local Proceedings. - Riga, Latvia: JUMI Publishing House Ltd., 2009. - pp. 48-61 [ISBN 978-9984-30-163-1]

12. Osis J., Donins U. An Innovative Model Driven Formalization of the Class Diagrams// Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2009). – Portugal: INSTICC Press, 2009. - pp. 134-145 [Indexed by SCOPUS, Thomson Reuters, Inspec, DBLP]
13. Donins U., Osis J. Reconciling Software Requirements and Architectures within MDA// Scientific Proceedings of Riga Technical University, Computer Science (Series 5), Applied Computer Systems (Vol. 38). - Riga, Latvia: RTU Publishing house, 2009. - pp. 84-95 [Indexed by DBLP]

In addition to the scientific papers, a monograph has been published:

1. Doniņš U. Topological Business Systems Modeling and Software Systems Design. - Riga, Latvia: RTU Publishing house, 2011. - 65 p. (in Latvian) [ISBN: 978-9934-10-136-6]

Thesis Outline

The thesis consists of introduction, 5 chapters, conclusions, 14 appendices, and bibliography. The doctoral thesis contains 224 pages, 71 figure and 32 tables. Bibliography includes 134 information sources.

Introduction gives motivation of the thesis, research goal, tasks defined to reach the goal, novelty and practical value of the research together with the approbation and the main results achieved as well.

Chapter 1 represents the research on UML, including the review of its evolution (by paying most attention on the diagrams included in versions 1.x and 2.x as well as on the formalism development used to specify the language). The research on UML shows the benefits and limitations of applying it within software development lifecycle, and identifies UML extension mechanisms and scenarios. As a result UML improvement options are outlined. One of the improvement options is to strengthen UML specification by using topology which is based on formalism of TFM.

Chapter 2 analyzes methods and approaches that support and promote the use of UML within software development process. The analysis of UML modeling driven approaches consists of review of eight mutually unrelated methods. The result of analysis shows the UML diagrams applied, the order in which the diagrams of different types are created, and the activities that are performed to model problem domain in order to develop solution model.

Additional emphasis is paid on the formal relation of problem domain with the solution domain.

The UML extension mechanism – profiles – is covered in *Chapter 3* as well as the development of the TopUML profile. Since the UML is a notation and as such does not include any guidelines of developing and specifying UML profiles, an additional analysis is made of currently existing UML profiles in order to develop the best practice (i.e., template) of profile specification. The application of identified best practice allows better understanding and reading the specification of created profile. TopUML profile is a combination of UML and formalism of TFM and is based on the principles of metamodeling; it extends the UML version 2.4.1 by adding TFM to UML and topological functioning characteristics into UML diagrams.

Since the TopUML specification is developed in accordance with the introduced template of UML profiles and as a profile specifies only the language, *Chapter 4* presents the TopUML modeling – a method intended for systematical application of TopUML within software development analysis and design phase. TopUML modeling is defined as a set of activities. Each activity defines the input (e.g., requirements) and the output artifacts (e.g., TFM) thus allowing the software developers to apply all of the activities as specified by TopUML modeling or by applying subset of the modeling activities. The application of these activities can vary from project to project. Additionally Chapter 4 compares TopUML modeling with the UML modeling driven approaches covered in Chapter 2.

In *Chapter 5* application and approbation of TopUML language and modeling method in the context of experimental software development and case study is explored and described. Experimental software development includes empirical evaluation of TopUML and its modeling. The empirical evaluation is based on practical experiment with two expert groups in which a business support application for the Laundry problem domain is designed. Case study is a step-by-step exploration of TopUML application in real software development project.

The *Conclusions* summarizes the results of this research, gives conclusions and possible future research directions.

Thesis contains fourteen *appendices*:

1. Used Abbreviations,
2. TopUML Specification,
3. Mappings Between TopUML Diagrams,
4. Informal Description of Laundry Functioning,

5. Functional Requirements and System Goals of the Laundry Software System,
6. Functional Features of Laundry Functioning,
7. Closuring of Laundry Functioning Topological Space,
8. Sequence Diagrams Representing Behavior of Laundry,
9. Laundry System Topological Class Diagram,
10. Lattelecom Technology Ltd. Acknowledgement of Software Development by using TopUML Modeling Method,
11. Specification of Enterprise Data Synchronization System,
12. Functional Features of Enterprise Data Synchronization System,
13. Self-Evaluation Questionnaire, and
14. OMG-Certified UML Professional Certificate.

1. UNIFIED MODELING LANGUAGE – A STANDARD FOR SOFTWARE DESIGN SPECIFICATION

Unified Modeling Language (UML) is a graphical language officially defined by OMG for visualizing, specifying, constructing, and documenting the artifacts[1] of a software-intensive system [115]. It offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components [41]. Despite that UML is designed for specifying, visualizing, constructing, and documenting software intensive systems, it is not restricted only for software modeling. UML has been used for modeling hardware, and is commonly used for business process modeling, systems engineering modeling and representing organizational structure, among many other domains [133].

During the two major and a number of revision versions of UML, the definition of UML is evolving. UML version 2.4.1 specification ([88] and [89]) defines the language as follows: "*UML is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms (e.g., J2EE, .NET).*"

The UML is developed in middle of 1990s as a combination of previously competing object-oriented analysis and design approaches: *Booch method* (by Booch; [13]), *Object-Modeling Technique* (OMT; by Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen; [114]), and *Object-Oriented Software Engineering* (OOSE; by Jacobson, Christerson, Jonsson, and Overgaard; [54]), along with other contributions to modeling complex systems (e.g., *statecharts* by Harel [51]). The first version of UML (version 1.1) was approved by OMG in year 1997 [83]; afterwards UML has been revised with several releases (UML 1.3, 1.5, 2.0, 2.1.1, 2.1.2, 2.2, 2.3, and 2.4.1 [91]) by fixing some problems and adding new notational capabilities. The latest standard released by OMG is UML version 2.4.1 (UML version 2.0 is a major rewrite of UML 1.x[2] and was released in 2005).

UML became widely accepted as the standard for object-oriented analysis and design soon after it was first introduced [60] and still remains so today [24][111]. Since the release of

[1] An artifact in software development is an item created or collected during the development process. Example of artifacts includes use cases, requirements, design, code, executable files, etc.
[2] "x" denotes the main version and any subversion of specification.

first UML version a large number of practitioner and research articles and dozens of textbooks have been devoted to articulating various aspects of the UML, including guidelines for using it. Some of the research areas on UML are as follows:

- Formalization of UML semantics (e.g., [34], [52] (both after UML 1.1 was released), and [129] (after UML 2.0 was released)),
- Extending the UML (e.g., [76], [105], and review of a number of UML profiles developed by different researchers and groups [111]),
- Formalizing the way the UML diagrams are developed (e.g., [100] and [102]),
- Ontological analysis of UML modeling constructs (e.g., [133]),
- Empirical assessments (e.g., [24] and [35]),
- Analysis of the UML's complexity (e.g., [33], [120], and [121]),
- Difficulties of learning UML (e.g., [122]) and how to avoid them (e.g., [11]),
- Transformations between UML diagrams (e.g., [67], [63], and [77]),
- Software code generation and related issues with generated code quality (e.g., [65] and [117]), and
- Experiments that evaluate aspects of UML models effectiveness (e.g., [17]).

A list of recent UML research areas can be found also in [10]. The large number of researches regarding UML evolving and strengthening is caused by the basis on which UML was developed. According to Dobing and Parsons [24] the "*UML was not developed based on any theoretical principles regarding the constructs required for an effective and usable modeling language for analysis and design; instead, it arose from (sometimes conflicting) "best practices" (e.g., Booch, OMT, OOSE) in parts of the software engineering community*".

The review of elements that build up UML within this chapter is based on UML version 2.4.1 specification which is divided into two volumes (both volumes cross-reference each other and the specifications are fully integrated):

- *Infrastructure* ([88]) –defines a metalanguage core that can be reused to define a variety of metamodels, including UML, Meta-Object Facility (MOF; [85]), and Common Warehouse Metamodel (CWM; [82]); and the core metamodel on which the Superstructure is based. The Infrastructure of the UML is defined by the *InfrastructureLibrary* package which consists of two subpackages:
    - *Core* – contains core concepts which are used when metamodeling, and
    - *Profiles* – defines the mechanisms that are used to customize metamodels.

- *Superstructure* ([89]) – defines the notation and semantics for diagrams and their elements. The Superstructure metamodel is specified by the UML package, which is divided into a number of subpackages that deal with structural and behavioral modeling.

1.1. UML Diagrams

A system can be specified from different views by drawing different UML diagrams. In context of software development, there are five complementary views that are important in visualizing, specifying, constructing, and documenting software architecture [115]: the use case view, the design view, the interaction view, the implementation view, and the deployment view. Each of these views involves structural modeling (static aspect of a system) as well as behavioral modeling (dynamic aspect of a system). As specified in [15] "*a diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). A diagram is a projection into a system*".

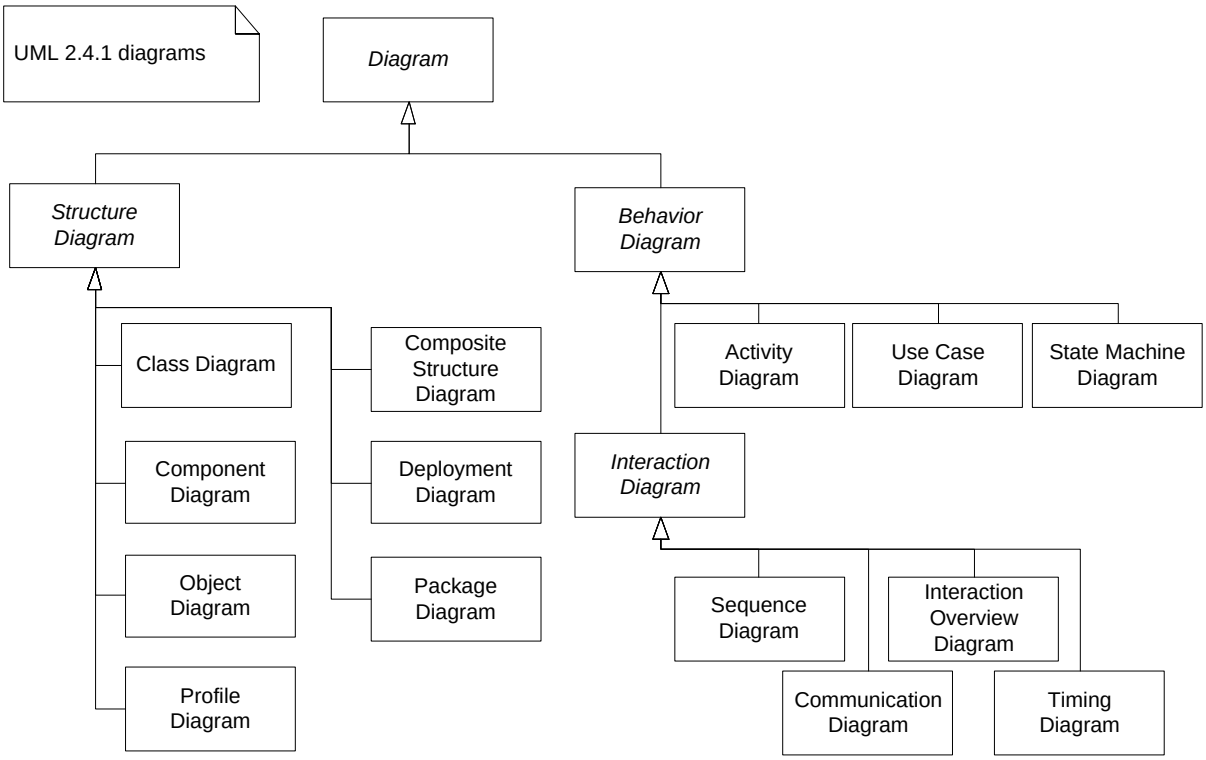


Figure 1.1. Diagram types included in UML version 2.4.1 [89]

The first UML specification (version 1.1) [83] contained nine diagram types: Class diagram, Component diagram, Object diagram, Deployment diagram, Activity diagram, Use Case diagram, Statechart diagram, Sequence diagram, and Collaboration diagram, while the newest UML specification (version 2.4.1) [89] released by OMG in year 2011 contains fourteen diagram types (see Figure 1.1): Class diagram, Component diagram, Object diagram, Composite Structure diagram, Deployment diagram, Package diagram, Profile diagram, Activity diagram, Use Case diagram, State Machine (or Statechart, or State) diagram, Sequence diagram, Communication diagram, Interaction Overview diagram, and Timing diagram.

All of UML diagrams are briefly described in following subsections: Section 1.1.1 describes structure diagrams, Section 1.1.2 – behavior diagrams, and Section 1.1.2.1 – interaction diagrams.

1.1.1. Structure Diagrams

The UML's structural diagrams are aimed to visualize, specify, construct, and document the static aspects of a system [15]. All structure diagrams are listed in Table 1.1 which shows their name, description, main elements, and UML version in which the diagram is included.

Table 1.1

UML structure diagrams

No.	Name	Description	Elements	Version
1.	Class diagram	Class diagram is the most common diagram found in object-oriented systems and it is used to illustrate the static design view of a system. It shows a set of classes, interfaces, and collaborations and their relationships. Class diagram that include active classes is used to address the static process view of a system. Class diagrams are also the foundation for a couple of related diagrams: component and deployment diagrams. [15]	• Class • Interface • Relationship[3] • Enumerator	1.1 – …
2.	Component diagram	A component diagram shows the internal parts, connectors, and ports that implement a component. When the component is instantiated, copies of its internal parts	• Interface • Component • Port	1.1 – …

[3] Denotes all relationships defined in UML specification (e.g., Association, Generalization, Dependency, etc.).

No.	Name	Description	Elements	Version
		are also instantiated. [15] The UML Component Diagram shows how a software system will be composed of a set of deployable components – assembly DLLs, executable files, or web services – that interact through well-defined interfaces and which have their internal details hidden. [69]	• Internal structure • Part • Connector	
3.	Composite structure diagram	A composite structure diagram shows the internal structure of a class or collaboration. The difference between components and composite structure is small. [15]	• Interface • Component • Port • Connector	2.0 – …
4.	Deployment diagram	A deployment diagram shows a set of nodes and their relationships. It is used to illustrate the static deployment view of architecture. Deployment diagram is related to component diagram in that a node typically encloses one or more components. A deployment diagram is a diagram that shows the configuration of runtime processing nodes and the artifacts that live on them. [15]	• Node • Relationship	1.1 – …
5.	Object diagram	An object diagram is a snapshot of the objects in a system at a point in time – it shows one set of objects in relation to one another at one moment in time. It models static design view or static process view of a system from the perspective of real or prototype system. Because it shows instances rather than classes, an object diagram is often called an instance diagram. [41] [15]	• Object • Link • Note • Constraint	1.1 – …
6.	Package diagram	The package is general-purpose mechanism for organizing modeling elements into groups. Packages are used to arrange modeling elements (e.g., classes, interfaces, components, nodes, diagrams, collaborations, use cases, other packages) into larger chunks that it is possible to manipulate them as a group. Packages can also be used to present different views of system's architecture. Well-designed packages group elements that are semantically close and that tend to change together. [15]	• Package • Element • Name • Import relationship • Export	2.0 – …
7.	Profile diagram	The profile diagram contains mechanisms that allow extending and adapting metaclasses from existing metamodels for different purposes; includes the ability to tailor the UML metamodel for different platforms or domains. The profiles mechanism is consistent with the MOF. [89]	• Stereotype • Metaclass • Extension • Profile application • Import	2.2 – …

1.1.2. Behavior Diagrams

UML behavior diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system (modeling dynamic aspects of a system can be considered as representing its changing parts) [41]. All behavior diagrams (except interaction diagrams) are listed in Table 1.2 which shows their name, description, main elements, and UML version, in which the diagram is included. Interaction diagrams are given in next Subsection (1.1.2.1).

Table 1.2

UML behavior diagrams (without interaction diagrams)

No.	Name	Description	Elements	Version
1.	Activity diagram	Activity diagram shows the flow from step to step within a computation. An activity shows set of actions, the sequential or branching flow from action to action, and values that are produced or consumed by actions. Activity diagrams are used to illustrate the dynamic view of a system; they are especially important in modeling the functioning of a system. [15]	• Activity • Action • Edge • Flow • Objects • Constraint	1.1 – …
2.	Use case diagram	Use case diagram shows a set of use cases and actors (a special kind of class) and their relationships; it organizes and models the behaviors of system. [15] Each use case in use case diagram typically is supplemented by a full use case specification – a written statement detailing the preconditions (what must be true before the use case is performed), the sequence of events, and the post-conditions (what must be true after the use case has completed). [69]	• Subject • Use case • Actor • Relationship • Subsystem	1.1 – …
3.	State diagram (or State machine diagram)	State diagram shows a state machine, consisting of states, transitions, events, and activities; it is behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. State diagram is especially important when modeling the behavior of an interface, class, or collaboration; it emphasizes the event-ordered behavior of an object, which is useful when modeling reactive systems. [15]	• Effect • State • Event • Transition • Trigger	1.1 – …

1.1.2.1. Interaction Diagrams

An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagram is the collective name that is given to sequence, communication, interaction overview, and timing diagrams. These diagrams share the same underlying model, although in practice they emphasize different things. By using interaction diagrams, it is possible to reason about flow of control within an operation, a class, a component, a use case, or the system as whole in two ways: 1) focusing on how messages are dispatched across time, and 2) focusing on the structural relationships among the objects in an interaction and then consider how messages are passed within the context of that structure [15]. Interaction diagrams are listed below in Table 1.3 which shows diagram name, description, main elements, and UML version in which the diagram is included.

Table 1.3

UML interaction diagrams

No.	Name	Description	Elements	Version
1.	Sequence diagram	A sequence diagram is an interaction diagram that emphasizes the time ordering of messages; it shows a set of roles and messages sent and received by instances playing these roles. Sequence diagram has two features that distinguish them from communication diagrams: presence of lifetime and focus of control. [15][69]	• Object • Role • Lifeline • Time • Control operator	1.1 – …
2.	Collabo-ration diagram	A collaboration diagram represents a Collaboration (a set of objects related in a particular context), and an Interaction (a set of messages exchanged among the objects within a collaboration to effect a desired operation or result). [83]	• Object • Message • Link	1.1 – 1.5
3.	Communi-cation diagram	Communication diagram accents structural organization of objects that send and receive messages; it shows a set of roles, connectors among roles, and messages sent and received by the instances playing these roles. Communication diagram have two features that distinguish them from sequence diagrams: path and sequence number of messages. Sequence number indicates the time order of message). A communication diagram is a simplified version of the UML version 1.x collaboration diagram. [15] [89]	• Object/Role • Message • Link	2.0 – …

No.	Name	Description	Elements	Version
4.	Interaction overview diagram	Interaction overview diagram define interactions through a variant of activity diagram in a way that promotes overview of the control flow. The lifelines and the messages do not appear at this overview level. Interaction overview diagrams are specialization of activity diagrams that represent interactions. [89]	• Frame • Interaction • Interaction use	2.0 – …
5.	Timing diagram	Timing diagram is used to show interactions when a primary purpose of the diagram is to reason about time; it focuses on conditions changing within and among lifelines along a linear time axis. [89]	• Frame • Message • Timeline • Lifeline	2.0 – …

1.2. Formalism of UML

The UML specification is defined by using a metamodeling approach which adapts formal specification techniques. A metamodel is used to specify the model that comprises UML. In spite of using metamodeling approach, the UML specification method lacks some properties of formal specification methods. The specification of UML cannot be considered as formal specification because of natural language (English) use in it. UML specification [88] underlines that the specification as a metamodel does not eliminate the option of specifying it later by using formal/mathematical language (e.g., Z [126], PVS [110], RAISE [80]). Section 1.2.1 takes closer look at the formalism of UML version 1.x specification and section 1.2.2 takes closer look at the formalism of UML version 2.x (as the UML version 2.x is major revision of UML version 1.x).

1.2.1. Formalism of UML Version 1.x

The specification of UML version 1.5 ([84]) contains language syntax and its static and dynamic semantics. The specification uses a combination of languages - a subset of UML, an object constraint language, and precise natural language to describe the abstract syntax and semantics of the full UML. Thus the UML version 1.x specification uses a formal technique for preciseness improving but at the same time keeping readability of it. Despite that the language structure is described in precise specification that is necessary for tool interaction it is needed to note that the existing description is not a completely formal specification (due to the use of natural language). As stated in [84], a common technique for

specification of languages is to first define the *syntax* of the language and then to describe its *static* and *dynamic semantics*. The *syntax* defines what constructs exist in the language and how the constructs are built up in terms of other constructs. *Static semantics* of a language define how an instance of a construct should be connected to other instances to be meaningful while *dynamic semantics* define the meaning of a well formed construct. These semantics are described using natural language (English).

Summarizing up the UML version 1.x specification, the metamodel of UML is described in a semi-formal way using three views [84]:

- *Abstract syntax* –presented in a form of UML class diagram. The UML metamodel is defined with the set of interrelated packages. Abstract syntax shows the metaclasses defining the constructs and their relationships and also presents some of the well-formedness rules (mainly the multiplicity requirements of the relationships), and whether or not the instances of a particular sub-construct must be ordered. A short informal description in natural language describing each construct is supplied. The first paragraph of each of these descriptions is a general presentation of the construct that sets the context, while the following paragraphs give the informal definition of the metaclass specifying the construct in UML. Each metaclass has its attributes enumerated together with a short explanation. Besides that, the opposite role names of associations connected to the metaclass is also listed in the same way.
- *Well-formedness Rules* – the static semantics of UML metaclasses (except for multiplicity and ordering constraints) are defined as a set of invariants of an instance of the metaclass. These invariants have to be satisfied for the construct to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Each invariant is defined by an expression written using Object Constraint Language (OCL; [132]) together with an informal explanation in English of the expression.
- *Semantics* – defines meanings of the constructs using natural language (English). The constructs are grouped into logical chunks that are defined together. Since only concrete metaclasses have a true meaning in the language, only these are described in this section.

In summary, the UML metamodel is described in a combination of graphic notation, (precise) natural language, and formal language. The use of natural language for specifying language constructs makes its specification semiformal. This semiformal specification of

UML can cause incorrectness and inaccuracy of system models defined with UML (due that statements in natural language can be interpreted with different meanings among different persons (in spite of trying to use it as precise as possible)).

1.2.2. Formalism of UML Version 2.x

At the time of writing thesis the current version of UML is 2.4.1 and the formalism of this version is discussed in this section with regard to infrastructure [88] and superstructure [89] specification. The main goal of major revision of UML within version 2.0 is to increase the precision and correctness of the specification. The set of UML modeling concepts is partitioned into horizontal layers of increasing capability called *compliance levels*. For ease of model interchange, there are only two compliance levels defined for infrastructure specification [88]:

- *Level 0 (L0)* – contains a single language unit that provides capabilities for modeling class-based structures encountered in object-oriented programming languages, and it provides an entry-level modeling capability, and
- *Metamodel Constructs (LM)* – adds an extra language unit for more advanced class-based structures used for building metamodels.

Superstructure specification adds three more compliance levels [89]:

- *Level 1 (L1)* –adds new language units and extends the capabilities provided by Level 0. Specifically, it adds language units for use cases, interactions, structures, actions, and activities.
- *Level 2 (L2)* – extends the language units already provided in Level 1and adds language units for deployment, state machine modeling, and profiles.
- *Level 3 (L3) – represents the complete UML*. It extends the language units provided by Level 2 and adds new language units for modeling information flows, templates, and model packaging.

All compliance levels are defined as extensions to a single core *«UML»* package that defines the common namespace shared by all the compliance levels. Level 0 is defined by the top-level metamodel. The UML version 2.x specification is defined by using a metamodeling approach that adapts formal specification techniques. According to [88], the following are the goals of the specification techniques used to define UML 2.x:

- *Correctness* – improves the correctness of the metamodel by helping to validate it.

- *Precision* – increases the precision of both the syntax and semantics. The precision should be sufficient so that there is nor syntactic nor semantic ambiguity for either implementers or users.
- *Conciseness* – the specification techniques should be parsimonious, so that the precise syntax and semantics are defined without superfluous detail.
- *Consistency* – the metamodeling approach should be complemented by adding essential detail in a consistent way.
- *Understandability* – while increasing the precision and conciseness, the readability of the specification should also be improved. For this reason a less than strict formalism is applied, since a strict formalism would require formal techniques.

The specification technique used in UML version 2.x describes the metamodel in the same way as the version 1.x does, i.e., it uses metamodeling approach and three views (abstract syntax, well-formedness rules, and semantics). Main language constructs are related to metaclasses in the metamodel. Other constructs, i.e., being variants of other ones, are defined as stereotypes of metaclasses in the metamodel. This mechanism allows the semantics of the variant constructs to be significantly different from the base metaclass. Another way of defining variants is the use of metaatributes.

The UML 2.x metamodel contains *infrastructure library* package which defines a reusable metalanguage kernel and a metamodel extension mechanism for UML. The metalanguage kernel can be used to specify a variety of metamodels, including UML itself, MOF, and CWM. In addition, the infrastructure library defines a profile extension mechanism that can be used to customize UML for different platforms and domains without supporting a complete metamodeling capability. The UML profile extension mechanism reduces notation size and efforts for specific task solution and allows creating additional constructs along with the benefit of profile reuse in ordinary UML modeling tools. The architectural alignment among UML, MOF and metadata interchange (XMI) tries to solve the problem of UML model interchange between tools by using the rules of XMI specification.

In spite of trying to use natural language in more precise way, the specification of UML cannot be considered as formal specification because of natural language use – the problem considered in previous subsection still exists. The UML specification still underlines that the specification as a metamodel does not eliminate the option of specifying it later by using formal or mathematical language. However the first steps of formalizing UML

constructs is taken – starting from UML version 2.0 the activity diagram is formally based on Petri nets [89].

1.2.3. The Need of Additional UML Formalization

Since the release of the first UML specification researchers are working and proposing approaches to improve formalization of UML. Researches on UML formalization are performed because the meaning of the language, which is mainly described in English, is too informal and unstructured to provide a foundation for developing formal analysis and development techniques, and because of the scope of the model, which is both complex and large [103]. Despite the fact that the latest UML specification is based on the metamodeling approach, the UML metamodel gives information about abstract syntax of UML but does not deal with semantics in formal way (as discussed previous, the semantics is expressed using natural language). Thus it is hard to determine how a given change in a model influences its meaning and to verify whether a given model transformation preserves the semantics of the model or not. Since UML is method-independent, its specification tends to set a range of potential interpretations rather than providing an exact meaning. [34] [129]

According to [34], the formalization of UML specification has following benefits:

- *Clarity* – the formally stated semantics can act as a point of reference to resolve disagreements over intended interpretation and to clear up confusion over the precise meaning of a construct.
- *Equivalence and Consistency* – a precise semantics provides an unambiguous basis from which to compare and contrast the UML with other techniques and notations, and for ensuring consistency between its different components.
- *Extendibility* – the soundness of extensions to the UML can be verified (as encouraged by the UML authors).
- *Refinement* – the correctness of design steps in the UML can be verified and precisely documented. In particular, a properly developed semantics supports the development of design transformations, in which a more abstract model is diagrammatically transformed into an implementation model.
- *Proof* – justified proofs and rigorous analysis of important properties of a system described in the UML require precise semantics. Proof and rigorous analysis are not currently supported by UML.

- *Tools* – the tools that make use of semantics, for example a code generator or consistency checker, require that semantics to be precise, whether it be expressed as part of the standard or invented in the code by the tool developer.

The current UML semantics are not sufficiently formal to realize all of the above listed benefits. Despite that researches on UML formalization have been made before the release of UML version 2.0, the UML version 2.x specification is not written as a formal specification of language. Therefore there are a number of ongoing UML formalization researches trying to formalize it from different aspects.

1.2.4. Current UML Formalization Attempts

After OMG accepted UML version 1.1 as a standard, a precise UML (pUML) group was found with main goal to bring together international researchers and practitioners who share the aim of developing the UML as a precise modeling language [34]. The aim of pUML group is to work firmly in the context of the existing UML semantics. As a formalization instrument they use several formal notations (e.g., OCL [132] or the formal language Z [126]). The pUML group is an example of researches focusing on formalization of UML semantics. Some of the formalization researches are restricted to the semantics of models, while the others are concerned with the issues of reasoning about models and model transformations. Currently there exist a number of approaches for specifying and formalizing semantics of UML:

- Specifying semantics by formal languages (e.g., using language Z [34] or Object-Z [58]),
- Using category theory – captures relationships between specification objects (e.g., [1] and [20]),
- Using stream theory – as streams is an adequate setting for the formalization of the semantics of concurrent systems (e.g., [16]),
- Using π-calculus or process algebra (e.g., [134]), and
- Using algebraic approaches (e.g., using mathematical notation [129])

As indicated by Evermann in [36], the researches on UML semantics formalization relate to the internal consistency of the UML, not to its relationship to problem domains. To address the relation of UML elements to problem domains, the researches are ongoing on formalizing the way the software is developed by using UML diagrams (e.g., the problem

domain formalization approach [101], or the software development with the emphasis on topology in constructed models [26]) and describing UML constructs by using ontology, thus relating them with problem domains (e.g., [36] and [133]).

By summarizing up the attempts to formalize UML, the following formalization directions emerge:

- Formalizing the semantics of UML,
- Formalizing the way the UML is used, and
- Relating UML constructs to problem domains.

1.3. Benefits of Applying UML

The use of UML for systems' modeling has following benefits ([3], [24], [39], [89], and [92]):

- The *UML is a modeling language and not a method*, methodology, or technique, thus making it independent of particular methods and programming languages. The UML specification defines a number of diagrams and the meaning of those diagrams. A method goes further and describes the steps required to develop the software, which diagrams are developed in what order, and who is responsible for completing certain tasks.
- The *UML is platform independent modeling language* – it can be used to design software for implementation in any programming language.
- It is a modeling language *created from* a set of *widely accepted object-oriented software design methods*, thus ending the endless choose between concurrent notations.
- *UML is a set of standardized object-oriented models*, thus making communication between stakeholders more efficient and meaningful, i.e., if stakeholders are familiar with UML then the created models of system can be more easily communicated between different development teams, customers, and stakeholders.
- Starting with UML version 2.0 it contains *extension mechanisms*. If the set of models provided by UML are not enough for required solution, it is possible to extend UML in a number of allowed ways.
- The UML can be used for both large and complex systems modeling, as well as for small projects.

- As *UML is defined in accordance with XMI*, the models can be transferred between different tools from different tool vendors. Thus making users of UML less dependent on particular modeling tools.

Despite all above mentioned benefits that the application of UML within software development has, it has also a number of disadvantages which are discussed in the next subsection.

1.4. Disadvantages of Applying UML

The specification of UML and the UML itself is not developed basing on any theoretical principles regarding the constructs required for an effective and usable modeling language for analysis and design. UML arose from "best practices" in parts of the software engineering community; in fact these "best practices" at some points are even conflicting [24]. Basically this means that the UML goes without mathematics [93] (except Activity diagrams, which are now (starting from UML version 2.0) based on the formalism and mathematics of Petri nets [89]). UML specification is described using the combination of languages – metamodeling, OCL, and the natural language. This has resulted in a language that contains many modeling constructs, which has thus been criticized on the grounds that it is excessively complex and large. At the same time, the UML has also been criticized for lacking the flexibility to handle certain modeling requirements in specific domains. As a result of this criticism, UML has evolved – starting from UML version 2.0 it allows the development of profiles. [24]

Main disadvantages of UML are as follows ([24], [26], [57] and [105]):

- *Size* – UML is a collection of notations that encompasses a wide range of notations. In addition, the provided extension mechanisms of UML allow modelers to add their own, often ad-hoc, extensions to the language. In short – *UML is large and growing*.
- *Incoherence* – UML has brought together a number of notations from different fields. For example, it is not clear how state diagram relate to class diagram and sequence diagram.
- *Different interpretations* – since the semantics of UML constructs are defined by using natural language, they are interpreted differently by different modelers.

- *Frequent subsetting* – organizations tend to define their own UML subset – guidelines on which parts to use; which not to use; own definitions of semantics where the standard is unclear, inconsistent or untenable for the organization concerned.
- *Lack of causality* – despite the fact that UML contains a set of 14 diagrams, none of the existing diagram allows to clearly trace cause-and-effect relationships between both problem and solution domains. This can be related to the fact, that only Use Case diagram deals with the requirements and computation independent viewpoint modeling.

In regards to the above listed disadvantages of UML, in [123] is presented a list of problems associated with using UML in software development; causes of these problems are various: ambiguous semantics, cognitive misdirection during the development process, inadequate capture of properties of system under consideration, lack of appropriate supporting tools and developer inexperience. By analyzing these problems in detail, part of the researchers claim that some of these problems can be addressed by formalizing UML semantics ([129]), and the most helpful sequencing of modeling techniques ([29]). Others claim that a revision of the UML and its supporting tools is required ([34]). Furthermore, it is assumed that largest part of these problems can be addressed to the ambiguous transition between analysis and design models ([105]).

1.5. UML Improvement Options

According to the UML version 2.4.1 specification ([88] and [89]) and recent researches (e.g., [100], [24], [36], [123], [129], and [133]) in the field of strengthening UML, its use and analysis, the following UML improvement options arise:

- Extending UML by using UML's extensibility mechanisms,
- Formalizing the semantics of UML,
- Formalizing the way the UML is used, and
- Relating UML constructs to problem domains.

The UML can be strengthened by using the mathematical topology. The use of topology reflects extending the UML to support topology in its diagrams, and formalizing the way the UML is applied during software development process. Since this work is dedicated to extend UML by its extensibility mechanisms and formalizing the software development

process, the following two subsections discusses UML extensibility mechanisms and UML improvement by using mathematical topology.

1.5.1. UML Extensibility Mechanisms

The UML version 2.4.1 extensibility mechanisms permit to extend the language in controlled ways; these mechanisms include stereotypes, tagged values, constraints, and profiles. If the enumerated four extension mechanisms does not solve the problem why the language should be extended, then UML metamodel can be extended using MOF. By extending UML using MOF there are no restrictions on what are allowed to do with a UML metamodel. [15] [88]

Stereotypes. A stereotype defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass. In other words - a stereotype extends the vocabulary of the UML, allowing to create new kinds of building blocks that are derived from existing ones but that are specific to problem under consideration.

Stereotype can be considered as a type that defines other types, because each one creates equivalent of a new class in the UML metamodel. When an element is stereotyped (such as node or a class), the UML gets extended by creating a new building block just like the existing one but with its own special modeling properties (each stereotype may provide its own set of tagged values), semantics (each stereotype may provide its own constraints), and notation (each stereotype may provide its own icon). The stereotype *«stereotype»* specifies that the classifier is a stereotype that may be applied to other elements. [15] [88]

Tagged values. A tagged value extends the properties of a UML stereotype, thus allowing creation of new information in element's specification. By using stereotypes it is possible to add new things to the UML; by using tagged values it is possible to add new properties to a stereotype. Tags that apply to individual stereotypes are defined so that everything with that stereotype has tagged value. A tagged value is not the same as class attribute. Rather, a tagged value can be considered as metadata because its value applies to the element specification, not to its instances. [15]

Constraints. A constraint extends the semantics of a UML construct, thus allowing to add new rules or to modify existing ones. Each constraint consists of a textual description in natural language and may be followed by a formal constraint expressed in OCL. If it is not

possible to express the constraint in OCL, then in such case the formal expression can be omitted. [15] [89]

Profiles. The Profile mechanism has been specially defined for providing a lightweight extension mechanism to the UML specification. In UML version 1.1, stereotypes and tagged values were used as string-based extensions that could be attached to UML model elements in a flexible way. In subsequent revisions of UML, the notion of a Profile was defined in order to provide more structure and precision to the definition of stereotypes and tagged values. Since the UML version 2.0 specification this has been carried further, by defining UML extension as a specific meta-modeling technique. Stereotypes are specific metaclasses, tagged values are standard metaattributes, and profiles are specific kinds of packages. A profile defines a specialized version of UML for particular area or solution. Because it is built on standard UML elements, it does not present a new language, and it can be supported by ordinary UML tools. [15] [88] [89]

According to UML version 2.4.1 specification [89], the profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing metamodels). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile.

UML metamodel extension. First-class extensibility is handled by using MOF, where there are no restrictions on what are allowed to do with a UML metamodel (i.e., it is possible to add and remove metaclasses, constraints, and relationships as necessary). [88]

"*There is no simple answer for when you should create a new metamodel and when you instead should create a new profile*" [89].

1.5.2. Improving UML by using Topology

UML improvement by using mathematical topology is based on topology and formalism of Topological Functioning Model (TFM), which is developed at Riga Technical University by Osis [99]. The TFM holistically represents a complete functionality of the system from the computation independent viewpoint. It considers problem domain information separate from the solution domain information. The TFM is an expressive and

powerful instrument for a clear presentation and formal analysis of system functioning and the environment the system works within. [101]

A TFM has topological characteristics: connectedness, closure, neighborhood, and continuous mapping. Despite that any graph is included into combinatorial topology, not every graph is a TFM. A directed graph becomes the TFM only when substantiation of functioning is added to the above mathematical substantiation. The latter is represented by functional characteristics: cause-effect relations, cycle structure, inputs and outputs. [100]

It is acknowledged that every business and technical system is a subsystem of the environment. Besides that a common thing for all system (technical, business, or biological) functioning should be the main feedback, visualization of which is an oriented cycle. Therefore, it is stated that at least one directed closed loop must be present in every topological model of system functioning. It shows the "main" functionality that has a vital importance in the system's life. Usually it is even an expanded hierarchy of cycles. Therefore, a proper cycle analysis is necessary in the TFM construction, because it enables careful analysis of system's operation and communication with the environment. [94]

The UML can be improved by supplementing it with the topological and functioning characteristics of TFM. To allow using topology in UML diagrams, it needs to be extended by using extensibility mechanisms covered in previous subsection. In such case a new kind of UML is created – *Topological Unified Modeling Language* (or *TopUML*). The idea of TopUML is adapted from [93]. The core framework proposal for TopUML profile is presented in [105]. The first research results in [102] shows that *the transfer of topological and functioning characteristics from TFM to UML is sufficient for clearly tracing cause-and-effect relationships in both – problem and solution – domains*.

1.6. Summary

UML is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains and implementation platforms.

The UML version 1.x (the first version – 1.1 – is released in 1997) contains nine diagram types. UML version 2.0 (released in 2005) is a major rewrite of UML version 1.x with the main goal to increase the precision and correctness of the specification. The version 2.0 contains thirteen diagram types, and the version 2.2 adds additional one diagram type –

Profile diagram (now in total UML has fourteen diagram types). At the moment of writing this work the newest version is 2.4.1 and it is released in 2011.

The specification of UML version 2.x is divided into two volumes: Infrastructure (core metamodel); and Superstructure (notation and semantics for diagrams and their model elements). Actually, the Superstructure specification is based on Infrastructure specification. The set of modeling concepts of UML is partitioned into horizontal layers of increasing capability called compliance levels. UML Infrastructure specification defines only two compliance levels (for ease of model interchange): Level 0 (L0), and Metamodel Constructs (LM), while the Superstructure specification adds three more compliance levels: L1, L2, and L3. In fact, the complete UML specification is given in compliance level L3.

While the application of UML within software development has a number of benefits, it also has some disadvantages. The main benefits are: UML is independent of software development methods, techniques and platforms; it has an extension mechanism thus allowing to solve specific modeling tasks; and the models can be transferred between different tools from different tool vendors since UML is defined in accordance with XMI. The main disadvantages of UML application is its size, incoherence, different interpretations, frequent subsetting, and the lack of causality. From these disadvantages rises a set of problems like ambiguous semantics, cognitive misdirection during the development process, inadequate capture of properties of system under consideration, lack of appropriate supporting tools and developer inexperience, and inability to trace cause-and-effect relationships between the existing artifacts in problem domain and created artifacts in solution domain. By taking a closer look at benefits and disadvantages, it is visible that some benefits turn into disadvantages (e.g., independency of software development methods leads to cognitive misdirection during the development process). To address the listed disadvantages, a bunch of researches on UML strengthening and formalization are performed and are still ongoing, e.g., formalizing the semantics of UML, formalizing the way the UML is used, and relating UML constructs to problem domains.

UML can be strengthened by using mathematical topology thus addressing the disadvantage of lacking causality. Next chapter is dedicated to explore currently existing UML modeling driven software development approaches, thus addressing the disadvantages of UML's size, incoherence, different interpretations, and frequent subsetting.

2. SOFTWARE DESIGNING WITH UML MODELING DRIVEN APPROACHES

UML is a notation and as such its specification does not contain any guidelines of software development process (e.g., which diagrams to use in what order). This is pointed out as a benefit of UML application in software development as well as a disadvantage in Section 1.4. Despite that UML is independent of particular methods and approaches, most of the UML modeling driven methods uses use case driven approach [24]. This might be caused by the originators (Booch, Rumbaugh, and Jacobson) of the UML since they recommend a use case driven process in "The Unified Modeling Language User Guide" ([15]). A majority of UML modeling driven approaches since then has endorsed this view, and most contain at least some further prescriptions for applying the UML in modeling (e.g., [64], [112], and [127]).

Since UML modeling driven approaches are elaborated by different authors, their prescriptions sometimes differ. As indicated in [24], "while some accept the original view that only use cases are used to verify requirements with users, others explicitly or implicitly indicate that other UML diagrams can be used for this purpose, e.g., Activity Diagrams can be safely shared with customers, even those unfamiliar with software engineering". There is also difference in the use of use case narratives across various methods due to the lack of guidance on narrative format in the UML specification. The UML specification [89] only states that "use cases are typically specified in various idiosyncratic formats such as natural language, tables, trees, etc. Therefore, it is not easy to capture its structure accurately or generally by a formal model."

A successful software development project can be measured against the deliverables that satisfy and possibly exceed expectations of customer, the delivery schedule that has occurred in a timely and economical fashion, and the created result is resilient to change and adaptation. For software development project to be successful by means of given measurements, it should satisfy the following two characteristics [14]:

- Solution should have a strong architectural vision, and
- A well-managed development lifecycle should be used.

The Architecture description standard [53] defines architecture as "fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution". Besides this definition there exist a large number of architecture definitions. A system that has a good architecture

has also a conceptual integrity. Good software architectures tend to have several attributes in common [14]:

- They are constructed in well-defined layers of abstraction,
- They have a clear separation of concerns between the interface and implementation of each layer, and
- The architecture itself is simple – common behavior is achieved through common abstractions and common mechanisms.

This section discusses the current state of the art of UML based software development approaches by reviewing two aspects of each approach – the process of software development and the artifacts developed. Most attention is paid on the artifacts created by using the UML.

2.1. Current State of the Art

The review of software development methods discusses a number of existing UML modeling driven software development approaches paying the most emphasis and attention on the use and application of UML diagrams (i.e., which diagram types for what purpose are used and in which sequence they should be created). The analysis of UML diagram usage additionally shows if there are included transformation rules or guidelines between different diagram types. Each approach is reviewed by using following structure:

- At first a brief description of the approach is given,
- Overview and analysis is done of development steps involved in the approach, and
- Finally analysis of used UML diagrams is given.

Currently exist dozens of UML modeling driven software development approaches, e.g., software development lifecycles [116], use case driven methods [112], model driven architecture [59], pattern based development [64], component based development [127], and conceptual modeling [81]. Overview of the current state of the art of UML based software development approaches includes approaches that are well known in software development industry [24], formalizes the development process and problem domain [100], and are used in the conjunction of software development tools [69]. The overview of UML modeling driven software development approaches includes thus covering different aspects of software development and its organization:

- Object-oriented analysis and design with Unified Process,
- Business object-oriented modeling,

- Object-oriented analysis and design with patterns,
- Conceptual modeling,
- Component based development,
- Topological Functioning Modeling for Model Driven Architecture, and
- UML software design with Microsoft tools.

Above listed approaches are discussed and reviewed in the further subsections.

2.1.1. Object-Oriented Analysis and Design with Unified Process

Object-oriented software development is a complete conceptual framework that covers the entire software development life cycle and it has the following characterization [112]:

- Affects the way in which the requirements are analyzed and modeled,
- Affects the way the software engineer design the system specification, and
- Affects the way the code itself is structured: the software is implemented by using object-oriented programming languages (e.g., C++ [70], .NET language family (C#, Visual Basic .NET) [79], Java [19]).

One of the well-managed iterative and incremental development lifecycle is Unified software development process (Unified process), where each of the iteration includes its own requirements analysis, design, implementation, and testing activities. Unified process is based on the enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation. The system is developed incrementally over time, iteration by iteration, and thus this approach is also known as iterative and incremental software development. The iterations are spread over four phases: Inception, Elaboration, Construction, and Transition. Each phase consists of one or more iterations. [4] [116]

A set of predefined activities is accomplished within each iteration. The unified process describes work activities as disciplines – a discipline is a set of activities and related artifacts in one subject area (e.g., the activities within requirements analysis). The disciplines described by unified process are as follows [116]:

- *Business modeling* – domain object modeling and dynamic modeling of the business processes,
- *Requirements* – requirements analysis of system under consideration. Includes activities like writing use cases and identifying non-functional requirements,
- *Analysis and design* – covers aspects of design, including the overall architecture,

- *Implementation* – programming and building the system (except the deployment),
- *Test* – involves testing activities such as test planning, development of test scenarios, alpha and beta testing, regression testing, acceptance testing, and
- *Deployment* – the deployment activities of developed system.

The disciplines and phases of unified process are given in Figure 2.1 where it is shown that the relative effort across disciplines changes over time from iteration to iteration, e.g., initial iterations apply greater relative effort on requirements and design.

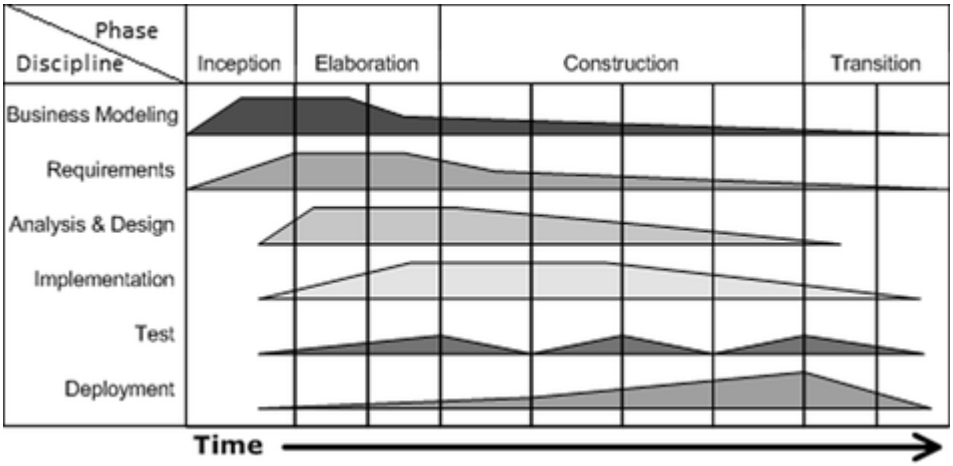


Figure 2.1. Disciplines and phases of unified process [4]

There exist a number of extensions and adaptions of unified process, e.g., Agile Unified Process (AUP) [2] and Rational Unified Process (RUP) [62].

2.1.1.1. Development Process

The development process within unified process is reviewed in the context of its four phases and correspondent iterations. These phases are as follows [116]:

1. *Inception* – establishes a justification or business case for the project, project scope and boundary conditions; outlines the key use cases and requirements, one or more candidate architectures; identifies risks, and prepares a preliminary project schedule and cost estimate.
2. *Elaboration* – primary goals are to identify and mitigate risks and to establish and validate the system architecture. Common processes in this phase include the creation of use case diagrams, domain model (class diagram) and architectural diagrams (package, component, and deployment diagrams). In the elaboration phase a partial implementation of the system is made (it includes architecturally

most significant components). The final elaboration phase artifact is a plan (including cost and schedule estimates) for the next – Construction – phase.

3. *Construction* – the largest phase which involves development of system. The system is built on the foundation created in Elaboration phase. This phase is divided into smaller iterations where each iteration results in an executable release of the software.
4. *Transition* – the final phase in which the system is deployed. Feedback received from an initial release (or initial releases) may result in further refinements to be incorporated over the course of several Transition phase iterations.

2.1.1.2. UML Diagrams Used

Since unified process is a software development lifecycle it uses almost every UML diagram type (see Table 2.1).

Table 2.1

UML diagrams used in Unified Process

No.	Diagram type	Sequence	Information for	Comments
1.	Use Case diagram	1	Sequence diagram, Class diagram, State diagram, Activity diagram	Initial use cases are created during inception phase and later refined in elaboration and construction phases. Use cases help finding conceptual classes using noun phrase identification.
2.	Sequence diagram	2	Communication diagram, Interaction overview diagram	Sequence diagrams are developed during elaboration and construction phases. They should be created for the main success scenario of the use case, and for frequent or complex alternative scenarios. Sequence diagram is generated from inspection of use case.
3.	Class diagram	3	Communication diagram, State diagram, Package diagram, Component diagram	Class diagram is developed during elaboration phase and refined later in construction phase. It is used to represent domain and a system design model. Domain model has conceptual classes with no operations; the key idea behind it is a visual dictionary of abstractions.
4.	State diagram	4	-	State diagrams are developed during elabo-ration and construction phases. The use of state diagrams is emphasized for showing system events in use cases, but they may additionally be applied to any

No.	Diagram type	Sequence	Information for	Comments
				class.
5.	Communicatio n diagram	4	Class diagram	Communication diagrams are developed during elaboration and construction phases. They illustrate object interactions and help to analyze relations between classes.
6.	Activity diagram	4	Interaction overview diagram	Activity diagrams are developed during elaboration and construction phases. Used to visualize business workflows and processes, or use cases.
7.	Interaction overview diagram	5	Interaction overview diagram	Interaction overview diagrams are developed during elaboration and construction phases. Used to visualize business workflows and processes, or use cases.
8.	Package diagram	6	Component diagram, Deployment diagram	Package diagram is developed during elaboration phase and refined later in construction phase. Each package groups a set of cohesive responsibilities.
9.	Component diagram	7	Deployment diagram	Component diagram is developed during elaboration phase and refined later in construction phase. It represents modular, deployable, and replaceable parts of a system.
10.	Deployment diagram	8	-	Deployment diagram is developed during elaboration phase and refined later in construction phase. It shows how instances of components are deployed on instances of processing nodes.

2.1.2. Business Object-Oriented Modeling

Business Object-Oriented Modeling (B.O.O.M.) developed by Podeswa [112] is an UML modeling approach intended to relate business analysis documentation to the object-oriented software development. B.O.O.M. is use case driven analysis approach. A standard UML use case refers to an interaction with any type of system. While analyzing and specifying the system following question arises [112]: "What type of system is being referring to?" Therefore use cases in B.O.O.M. are divided into two logical types: business use cases and system use cases (this distinction of use case types is not a part of the UML but is an UML extension). A business use case is an interaction with a business system while the latter one is an interaction with a software system. A system use case typically involves one active (primary) user and takes place over a single session on the computer. At the end of the system

use case, the user should feel that he or she has achieved a useful goal. The main idea of the B.O.O.M. is to identify and describe business use cases that a planned system will affect, thus analyzing possible changes in business workflows and human roles. Each business use case is analyzed, looking for activities that the application will realize, and this information is specified as system use cases, which further will drive the whole development process. [112]

2.1.2.1. Development Process

The software analysis and design with B.O.O.M. consists of two phases [112]: initiation phase and analysis phase.

Initiation phase involves creation and analysis of business use cases, initial identification of system use cases, and drawing a sketch of business objects involved into system in a form of class diagram. Processes defined by system use cases are identified by going back to the business use case workflow and selecting activities that can benefit from full or partial automation. Activity diagrams are used to model the workflow of each business use case in order to achieve consensus among developers and stakeholders of the business use cases. By the end of initiation phase developers should have overview of the project, initial list of system use cases together with knowledge on users involved within each system use case. System use cases are detailed only at the level to be able to estimate the project (e.g., whether the project development will take days, weeks, or months). The initiation phase includes following steps: modeling business use cases; modeling system use cases; sketching static model (class diagram including key business classes); and setting baseline for analysis.

Analysis phase includes elicitation of detailed requirements from stakeholders, and analysis and documentation of elicited requirements for verification by stakeholders and for use by the developers. To achieve this goal a system use case specifications are created by storyboarding the interaction between users and the proposed system. In parallel with system use case specification, a class diagrams describing key business concepts and business rules that apply to the business objects are developed. For better understanding of business objects life cycle, state diagrams are developed (state diagrams should be developed for at least every key business object). The analysis phase includes following steps: dynamic and static analysis; specifying test plan and implementation plan; and setting baseline for development.

2.1.2.2. UML Diagrams Used

UML diagrams involved into B.O.O.M. application are listed below in Table 2.2. Since the B.O.O.M. covers the requirements and analysis phase of software development

lifecycle, it uses diagrams only needed to analyze the business system and additionally a package diagram to organize developed artifacts.

Table 2.2

UML diagrams applied within B.O.O.M.

No.	Diagram type	Sequence	Information for	Comments
1.	Use case diagram	1	Activity diagram, State diagram, Class diagram, Package diagram	Two types of use cases: business and system use cases. While the first one describes the functionality from a business perspective, the latter one describes functionality by the system perspective. Business use cases are developed before system use cases.
2.	Class diagram	1	Package diagram, Object diagram	Describes key business concepts and rules that apply to business objects. A sketch of class diagram is made during initiation phase and later during analysis phase it is refined to include relationships.
3.	Package diagram	2	-	Package diagram is used as a container to group and organize other diagrams.
4.	Activity diagram	3	-	Activity diagram is used to help developers and stakeholders form a consensus regarding the workflow of each business use case.
5.	State diagram	4	-	The state change of objects in system is specified by using state diagram thus helping to avoid surmising objects behavior over time. State diagram should be created at least for every key business object.
6.	Object diagram	5	Class diagram	Object diagram is used instead of a class diagram in situations that involve more than one object of the same class acting in different roles. It is used while analyzing associations between classes.

2.1.3. Pattern Based Software Design

A general approach in object design is to identify requirements, create domain model and add operations to software classes, and define the messaging between the objects to fulfill the requirements [40]. Deciding what operations belong to which class and how the objects should interact is important and not a trivial task – it takes careful analysis and explanation. According to [64] "this is at the heart of what it means to develop an object-oriented system, not drawing domain model diagrams, package diagrams, and so on." Operations of a class are addressed as responsibilities of this class. Responsibilities answer to two questions: "*What to*

do?" and "*How to do?*"; and they are assigned to classes of objects during the object design. The translation of responsibilities into classes and operations is influenced by the granularity of the responsibility. A responsibility is not the same concept as an operation, but operations are implemented to fulfill assigned responsibilities. Responsibilities are implemented using operations that either act alone or collaborate with other operation and objects. In order to help assign responsibilities, patterns for object-oriented analysis and design has been created by a number of researchers and practitioners (for example, [40], [43], [64], and [118]). In object-oriented analysis and design, a pattern is a named description of a problem and solution pair that is used to design object-oriented systems and can be applied to new contexts. Pattern can be considered as guidelines.

Set of nine GRASP patterns (General Responsibility Assignment Software Pattern) introduced in [64] are designed to address the assignment of responsibilities for objects and analysis of their interaction. The GRASP patterns are as follows:

- *Creator* – responsible for creating an object of a class,
- *Information expert* - leads placing responsibility on the class with the most information required to fulfill it,
- *Controller* – assigns responsibility for dealing with systems events (usually the controller is the first object beyond the user interface layer),
- *Low coupling* – assigns responsibilities in a way that coupling remains low (coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements),
- *High cohesion* – assigns responsibilities in a way that cohesion remains high (cohesion is a measure that shows how appropriate the assigned responsibilities of an element are),
- *Polymorphism* – defines variation of behaviors based on object's type (achieved by using polymorphic operations),
- *Pure fabrication* – creates a special class that has a highly cohesive set of responsibilities and that do not exist in the problem domain (i.e., an artificial class),
- *Indirection* – assign responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled, and

- *Protected variations* – classes with predicted variation or instability are identified and responsibilities to these classes are assigned in a way to create a stable interface around them.

Since patterns can be used in the context of any software development lifecycle and the application of patterns and the order of pattern application are directly influenced by the used lifecycle, the subsection "Development process" is not included for pattern based design.

2.1.3.1. UML Diagrams Used

This section discusses UML diagrams addressed by GRASP design patterns. The UML diagrams used by GRASP design patterns are given in Table 2.3.

Table 2.3

UML diagrams used by GRASP design patterns

No.	Diagram type	Information for	Comments
1.	Use case diagram	Sequence diagram, Communication diagram, Class diagram, State diagram	Use cases are used as a identifications source of controller classes. Controller concept is described by Controller pattern.
2.	State diagram	-	Describes allowed sequence of external system events that are recognized and handled by a system in the context of a use case. Additionally state diagrams can be applied to any class.
3.	Sequence diagram	Class diagram	Sequence diagram is used to show the interaction between objects thus showing also the coupling between them. Sequence diagram is addressed by Creator, Controller, High cohesion, and Indirection patterns.
4.	Communication diagram	Class diagram	Communication diagram is used to show the interaction and relations between objects. Communication diagram is addressed by Creator, Controller, High cohesion, and Indirection patterns.
5.	Activity diagram	Class diagram	Used for visualizing business workflows and processes, or use cases.
6.	Class diagram	Sequence diagram, Communication diagram, State diagram	Since class diagram is the basis for domain model (in the context of Unified process described in section 2.1.1), it is addressed by all nine GRASP design patterns.
7.	Package diagram	-	Typical system is composed of a set of logical packages. Each package groups a set of cohesive responsibilities.

No.	Diagram type	Information for	Comments
			This is the basic practice of modularization to support a separation of concerns.

The GRASP patterns in the context of Unified software development process are focused on Business Modeling, Requirements and Design disciplines, thus the GRASP design patterns uses only part of UML diagram types. This is due the fact that GRASP design patterns are intended for analysis and design of objects. The deployment diagrams are not addressed while they describe the logical structure of objects deployment and not the responsibilities assigned to objects. However these diagrams are addressed by other patterns, for example, Pattern-Oriented Software Architecture or Architectural patterns. Example of Architectural pattern is Layer architecture [18] and Model-View-Controller pattern [40].

2.1.4. Conceptual Modeling

Conceptual modeling can be viewed as an activity related to capturing the knowledge about the desired system functionality. According to [92], "the conceptual schema of an information system is the specification of its functional requirements". In the field of conceptual modeling exists a number of approaches (a set of conceptual modeling approaches are reviewed in [119]). Review of conceptual modeling in this section is based on [92], where the development of conceptual schema is divided into two related parts:

- *Structural schema* – consists of a set of concepts used in a particular domain that constitutes a conceptualization (i.e., ontology) of a domain, and
- *Behavioral schema* – specifies valid changes in the domain state together with the actions that the system can perform (changes in the domain state are domain events and a request to perform an action is an action request event).

The conceptual schema of software system should include the knowledge about the domain and the functions that the system has to perform in order to be able to perform the three main functions of software system [92]:

- *Memory function* – ability to maintain a representation the domain state
- *Informative function* – ability to provide information about the domain state, and
- *Active function* – ability to perform actions that change the domain state.

The state of the domain consists of a set of relevant properties. The meaning of the relevant properties of the domain depends on the purpose for which the system is built. In the

conceptual modeling of information systems it is assumed that a domain consists of a number of objects and the relationships between them, which are classified into concepts. The state of a particular domain consists of a set of objects, a set of relationships, and a set of concepts into which these objects and relationships are classified.

2.1.4.1. Development Process

By looking at the conceptual modeling through the prism of UML and the diagram development sequence an interesting fact comes out – the first model to create is Class diagram (i.e., the structural schema of software system). The software development process by using conceptual modeling according to [92] is shown in Figure 2.2 within the context of two kinds of conceptual schemas that are developed for each software system.

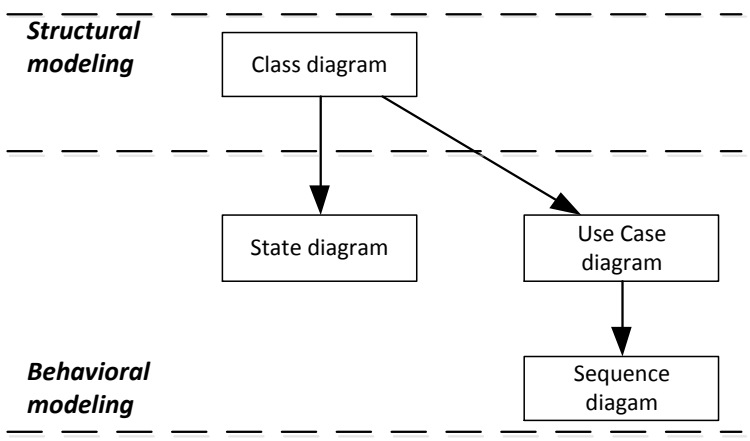


Figure 2.2. Schematic representation of conceptual modeling process

The development of Class diagram is divided into several sub-activities: identification of entities, their relationships (i.e., associations), cardinalities on associations, other relationship types, derivation, taxonomies (i.e., the class hierarchy), and domain events. The domain events within classes are reflected as operations. Each entity identified in structural schema has its own State diagram (or multiple State diagrams) reflecting state changes of it. The set of use cases should be consistent with the set of requests defined in the behavioral schema. This consistency comprises two properties:

- Each request generated by use case should be defined in the behavioral schema.
- Each request defined in the behavioral schema should be generated by one or more use cases.

One way of documenting the mapping of use cases to requests is by including textual references to requests near the places in the use case specification where they are generated. Each use case (the request sequence) can be specified by using Sequence diagram. A Sequence diagram shows, for one particular scenario of a use case, the action requests that the actors generate and their temporal order. The State diagram, Use case diagram, and Sequence diagram together defines the behavioral schema of software system. [92]

2.1.4.2. UML Diagrams Used

Conceptual modeling of software systems [92] uses only five UML diagram types: Class diagram, State diagrams, Use case diagram, Sequence diagram, and Profile diagram (see Table 2.4 below).

Table 2.4

UML diagrams used by conceptual modeling

No.	Diagram type	Sequence	Information for	Comments
1.	Class diagram	1	State diagram, Use case diagram (partly)	Reflects entities and their relationships. Each entity (class) has its own State diagram (or multiple State diagrams) reflecting state changes of it.
2.	State diagram	2	-	Each entity type may be associated with zero, one, or more State diagrams.
3.	Use case diagram	3	Sequence diagram	The set of Use cases should be consistent with the set of requests defined in the behavioral schema.
4.	Sequence diagram	4	-	A Sequence diagram shows, for one particular scenario of a Use case, the action requests that the actors generate and their temporal order.
5.	Profile diagram	5	Class diagram	An entity type defined in the schema of system may also be an entity in the information base of the same system or of another system. A meta entity type is an entity type whose instances are entity types.

2.1.5. Component Based Development

The first promise of developed code reuse is object orientation – classes developed for one project should be usable in the next project. This supposed to deliver high-quality products on time and in budget. Unfortunately as revealed in recent research [55] this is not true – the software projects frequently overrun their budgets and software is developed behind

the planned time. The next step in the direction of reusable software parts is components. According to [127] a component is unit that can be reused or replaced. The ideal way to build a new system in the context of component based development is to take existing components and plug them together. In [32] it is said that for a unit to be reusable it should have following characterization:

- High cohesion,
- Low coupling with the rest of the system,
- A well-defined interface, and
- It should be an abstraction of a well analyzed and understood concept.

Stevens in his book [127] describes component based development in the context of *4+1 architecture view model* [61] which divides software architecture in five *(4+1)* views:

- *Logical view* – shows parts of the system and how they are related together with the functionality that is provided to system users, this view specifies the logical behavior of the system,
- *Process view* – reflects the dynamic aspects of the system, explains the system processes and how they communicate; it addresses several nonfunctional characteristics of system like concurrency, distribution, integrators, performance, and scalability,
- *Development view* – shows system from the perspective of developer and is concerned with software management,
- *Physical (deployment) view* – depicts the system from a system engineer viewpoint; concerned with the layout of software components on the physical layer, as well as the physical connections between these components, and
- *Scenario view (the +1 view)* – description of architecture is illustrated using a small set of scenarios which describe sequences of interactions between objects and processes; scenarios are used to identify architectural elements and to illustrate and validate the architecture design.

2.1.5.1. Development Process

Component based development is oriented on creating reusable software components thus it can be used in the context different software development lifecycles and architectural styles. By applying component based development in the context of 4+1 architectural style as

suggested in [127], the following UML diagrams are developed for each of the architecture view:

- *Scenario view* – Use case diagram,
- *Logical view* – Class diagram, Interaction diagrams, and State diagram,
- *Process view* – Interaction diagrams, State diagram, Activity diagram, and Deployment diagram (used to determine the threads of control of the system),
- *Development view* – Component diagram and Package diagram, and
- *Physical view* – Deployment diagram.

The three case studies provided by Stevens and Pooley in [127] shows a part of a software development project. Within each case study the set of used diagrams differs and the order of diagram development also is different. The three case studies together with developed diagrams are as follows:

- Study process administration – 1) Use case diagram, 2) Class diagram, and 3) Activity diagram;
- Board games – 1) Communication diagram, 2) Class diagram, and 3) State diagram; and
- Discrete event simulation – 1) Class diagram, 2) Use case diagram, 3) State diagram, and 4) Communication diagram.

2.1.5.2. UML Diagrams Used

UML diagrams involved into component based development are listed below in Table 2.5. Since the use of UML diagrams vary from one case study to another (as discussed in previous subsection), the development sequence of UML diagrams cannot be precisely specified.

Table 2.5

UML diagrams used in components based development

No.	Diagram type	Information for	Comments
1.	Use case diagram	?	Specifies a set of scenarios which describe sequences of interactions between objects and processes.
2.	Class diagram	?	Shows objects of the system and how they are related together with the functionality that is provided to system users.
3.	Activity diagram	?	Determines the threads of control in the system.
4.	State diagram	?	
5.	Communication	?	Specifies the logical behavior of the system and determines the

No.	Diagram type	Information for	Comments
	diagram		threads of control in the system.
6.	Sequence diagram	?	
7.	Component diagram	?	Shows the system from the viewpoint of developer.
8.	Deployment diagram	?	Shows the system from the viewpoint of system engineer by showing the topology of components on the physical layer.
9.	Package diagram	?	Shows the system from the viewpoint of developer by grouping together related elements.

2.1.6. Topological Functioning Modeling for Model Driven Architecture

Topological Functioning Modeling for Model Driven Architecture (TFMfMDA) [5] is an approach intended for problem domain analysis and modeling in the context of Model Driven Architecture (MDA, [78]) thus attacking the weakest part of MDA – the Computation independent model (CIM) and its formal transformation to Platform independent model (PIM). In the context of MDA, TFMfMDA uses an extended version of MDA software lifecycle [94] (see Figure 2.3). In the standard MDA lifecycle [59] the feedback from deployment is going back directly to analysis (see "standard feedback" in Figure 2.3) and it is clearly visible that CIM is considered only as a textual requirements without any formal relation to the functionality of the business system and that the requirements and desired behavior of system is not considered at all when changes are needed in the deployed software system.

To avoid ignorance of CIM and analysis of the business system in the context of MDA, TFMfMDA uses capabilities of the universal category logic [9] and is based on the formalism of TFM. The main idea behind TFMfMDA is that the required functionality determines the structure of the planned system [100]. This corresponds to the opinion that there are two stages at the beginning of the problem analysis: the first one is analysis of the problem domain and the second one is analysis of the application domain. Having knowledge about a complex system that operates in the real world, a TFM of this system can be developed. This means that a TFM of the system is tested and can be partially changed and adjusted by functional requirements and vice versa. Usually changes in TFM are initiated if the software system introduces new functionality in the problem domain (e.g., in the context

of library software development project discussed in [100] – sending of SMS notifications is a new functionality introduced to business process through the developed software system).

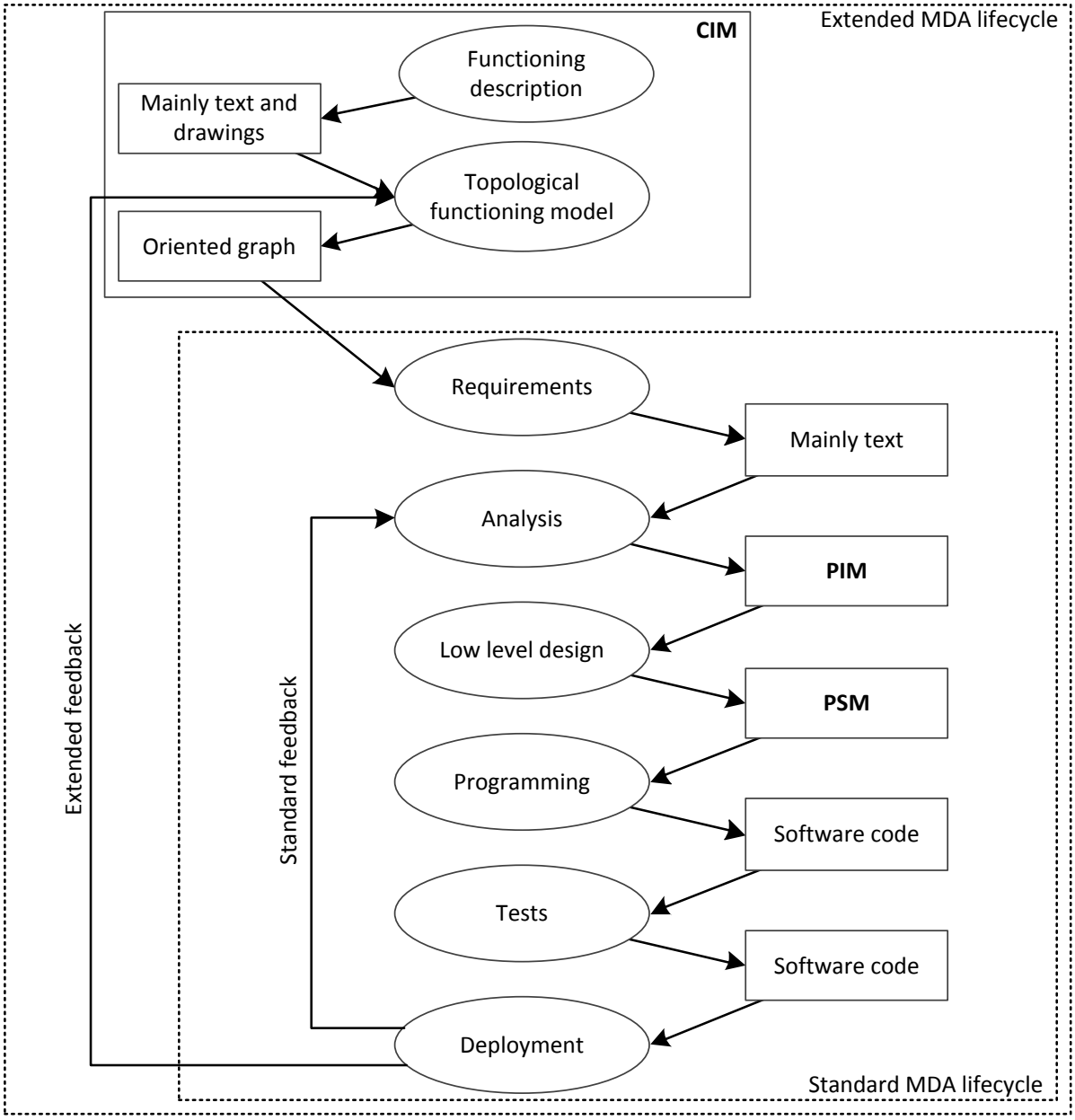


Figure 2.3. Extended and standard MDA software development lifecycle

2.1.6.1. Development Process

The software development process within TFMfMDA approach begins with the analysis and formalization of problem domain as shown in Figure 2.4, where the development is shown in the context of two kinds of information at the beginning of the problem analysis: the problem domain and the application domain. [5]

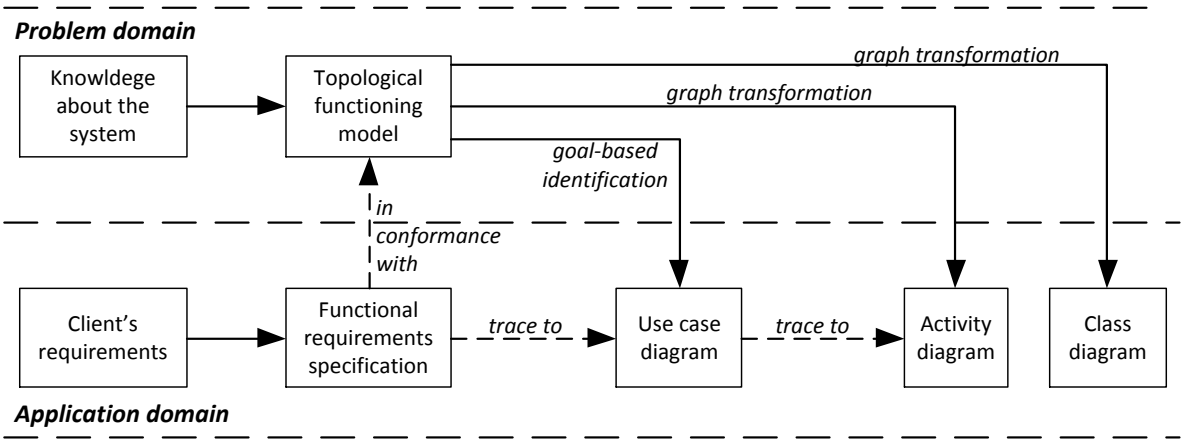


Figure 2.4. Software modeling within TFMfMDA approach

Problem domain analysis and software modeling within the TFMfMDA approach consists following actions:

1. Development of TFM reflecting the problem domain functioning,
2. Functional Requirement Mapping onto TFM,
3. Use case identification from a TFM,
4. Activity diagram development for each identified use case,
5. Conceptual and controller class identification, and
6. Optimal structure identification by using universal categorical logic.

The formal development of TFM within TFMfMDA is given in Figure 2.5.

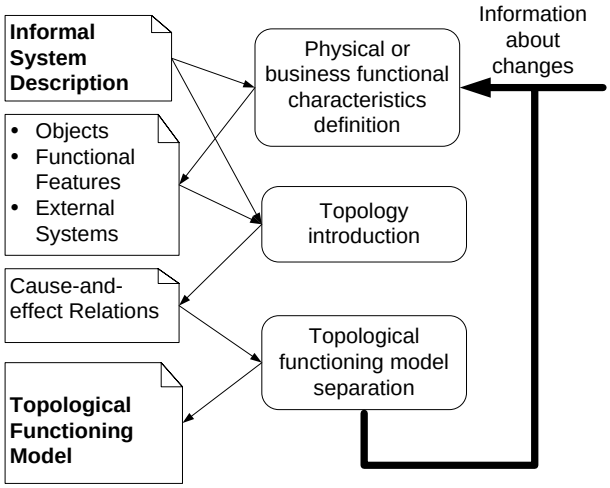


Figure 2.5. TFM development within TFMfMDA approach [100]

After the development of TFM, the functional features are associated with business goals of the system. Associating functional features with business goals provides business use

case and system use case identification according to the problem domain entities. Additionally after those activities functional requirements can be traced back to the system use case diagram. Problem domain concepts are selected and described in an UML Class diagram. The UML Class diagram is developed by performing two transformations: 1) TFM to Problem domain objects graph; and 2) Problem domain objects graph to Class diagram. As a result of this transformation a Class diagram reflecting conceptual classes (i.e., without attributes and operations) and non-directed associations between them is obtained.

2.1.6.2. UML Diagrams Used

TFMfMDA approach uses only three UML diagram types: Use case diagram, Activity diagram, and Class diagram; and additional two diagrams: TFM and Problem domain objects graph (see Table 2.6 below).

Table 2.6

Diagrams used in TFMfMDA approach

No.	Diagram type	Sequence	Information for	Comments
1.	Topological Functioning Model	1	Problem domain objects graph, Use case diagram, and Activity Diagram	TFM is used to formalize problem domain and thus it is the initial diagram developed when using TFMfMDA approach.
2.	Use case diagram	2	-	Since TFMfMDA approach is intended for problem domain analysis using TFM, it does not include guidelines for transformations between standard UML diagrams.
3.	Activity diagram	3	-	
4.	Class diagram	5	-	
5.	Problem domain objects graph	4	Class diagram	TFM is transformed 1:1 into problem domain object graph where each vertex shows only one type of objects.

Despite the fact that TFM can be transformed to Activity diagram, the author of TFMfMDA in [7] states that "it is impossible to create fork and join nodes automatically because the TFM does not hold information of concurrency" (thus TFM can be transformed into "simple" Activity diagram). The transformation from TFM to Class diagram is ambiguous while it is not clear how the control flow showing interaction between objects (i.e., cause-and-effect relationships) in TFM can be transformed into structural relationships (i.e., associations) between classes. The level of ambiguousness is even increased in the initial

stage of TFM to Class diagram transformation – the TFM to Problem domain object graph transformation.

2.1.7. UML Software Design with Microsoft Tools

Since the author of thesis has experience and knowledge background of developing software with Microsoft tools and programming languages and Microsoft takes significant part in the software industry [22] a brief review of UML software design with Microsoft tools is included. The UML software design with Microsoft tools is based on [38], [69], and [113]. With the Microsoft Visual Studio 2010 it is possible to design software with UML in two ways [69]:

- Top-Down Software Design (software code generation from UML models), and
- Down-Top or Reverse Engineering (generate UML models from software code).

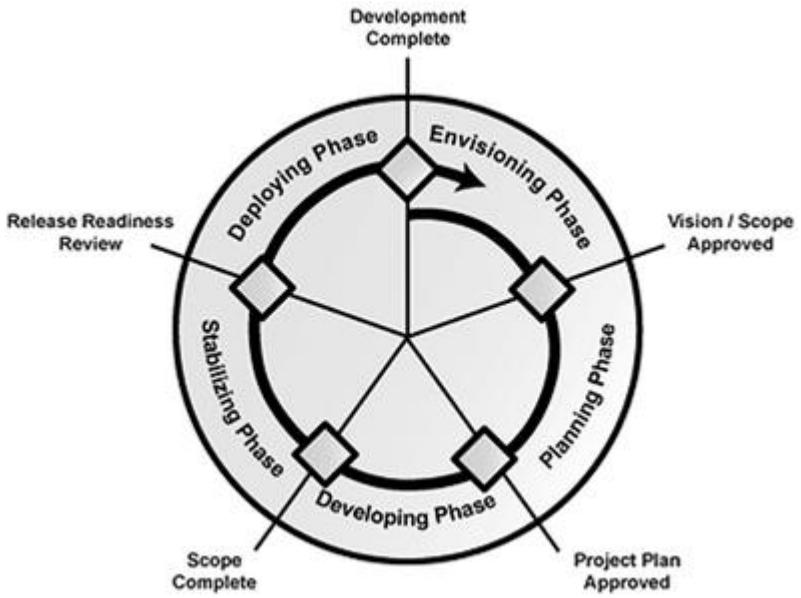


Figure 2.6. Phases of iterations in MSF [130]

Application of UML within Visual Studio 2010 is rather fragmented. In the context of .NET software development, the most logical choice of software development process may be the Microsoft Solutions Framework (MSF) process [130] as enacted by the Team Foundation Server [69]. The MSF is an iterative software development lifecycle where each iteration consists of five phases: Envisioning, Planning, Developing, Stabilizing, and Deploying phase (see Figure 2.6).

.NET software development is the mainstream of Microsoft software development initiatives and tools [113]. By adding Visualization and Modeling Feature Pack to the Microsoft Visual Studio 2010 it is possible to import UML models saved in XMI 2.1 standard. If XMI import file contains diagram types not supported by Microsoft Visual Studio 2010 – they will not be imported. Export to XMI files is not available. [113]

At the moment of writing thesis the Visual Studio 2010 includes only five UML diagram types: Activity diagram, Use case diagram, Sequence diagram, Class diagram, and Component diagram.

2.1.7.1. Development Process

Despite the fact that UML diagrams within Microsoft tools can be created in any order and that there is no need to create all of the available UML diagrams, Loton in his book [69] have described a logical sequence and transformations between available UML diagram types. Logical transformations between different types of UML diagrams within Microsoft Visual Studio 2010 according to [69] are given in Figure 2.7.

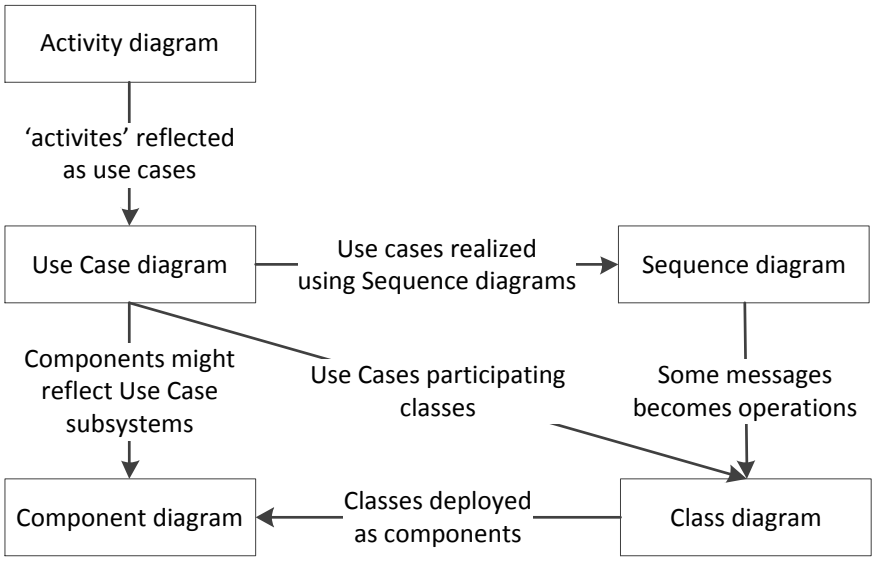


Figure 2.7. Logical transformations between different types of UML diagrams in Microsoft Visual Studio 2010

According to [69] the activities defined in Activity diagram can be reflected as use cases in Use case diagram. Each use case defined in Use case diagram should be realized in the form of a Sequence diagram. If a Sequence diagram is found without corresponding use case (or vice versa) the integrity of the solution can be doubted. For each use case there may

be also a corresponding classes in Class diagram, showing the static relationships between the classes that interact in the use case realizing Sequence diagram. If an inter-object message within the Sequence diagram does not have corresponding operation in the recipient class within Class diagram, the integrity of the solution can be doubted. Finally classes are packaged into components for deployment. In the context of .NET software development the components usually are deployed as executable files and dynamic link libraries.

2.1.7.2. UML Diagrams Used

Microsoft Visual Studio 2010 uses only five UML diagram types: Activity diagram, Use case diagram, Sequence diagram, Class diagram, and Component diagram (see Table 2.7). Additionally to listed five UML diagram types, the Microsoft Visual Studio 2010 includes several Microsoft specific diagrams, like Layer diagram, Directed Graph Document, and domain-specific language (DSL) Class diagram [113].

Table 2.7

UML diagrams used by designing software with Microsoft tools

No.	Diagram type	Sequence	Information for	Comments
1.	Activity diagram	1	Use case diagram	Activities can be reflected as use cases in Use case diagram.
2.	Use case diagram	2	Sequence, Class, and Component diagram	Each use case should be specified with a Sequence diagram while class diagram reflects the participating classes in use case and components may reflect subsystems specified in Use case diagram.
3.	Sequence diagram	3	Class diagram	Each inter-object message should be specified as an operation in class.
4.	Class diagram	4	Component diagram	Shows classes and static relationships between them that interact in the use case realizing Sequence diagram.
5.	Component diagram	5	-	Classes are packaged into components for deployment.

2.2. Benefits and Limitations of UML Modeling Driven Approaches

Since UML itself is a notation and as such it does not contain guidelines on how it can be applied in practice, the largest benefit of presence of UML modeling driven approaches is

that the use of UML is made systematical. If UML is combined together with some approach, it can be used as a powerful tool to analyze and understand both business and software systems and to design planned software system. Despite the fact that UML modeling driven approaches provides a systematical use of UML diagrams, these approaches do cover different parts of a software development lifecycles. Whole software development lifecycle is covered only by the Unified process and MSF, other methods focuses more on analysis (e.g., B.O.O.M., TFMfMDA, and Conceptual modeling) while others – more on design and less on analysis (e.g., Pattern based design, Component based development). This impacts the number of UML diagram types that are used by each of the method. The summary of UML diagrams usage by each method is given in Figure 2.8, where it can be seen that not every UML diagram type is used (UML in total has 14 diagram types). The greatest amount of applied diagram types is for the Unified process.

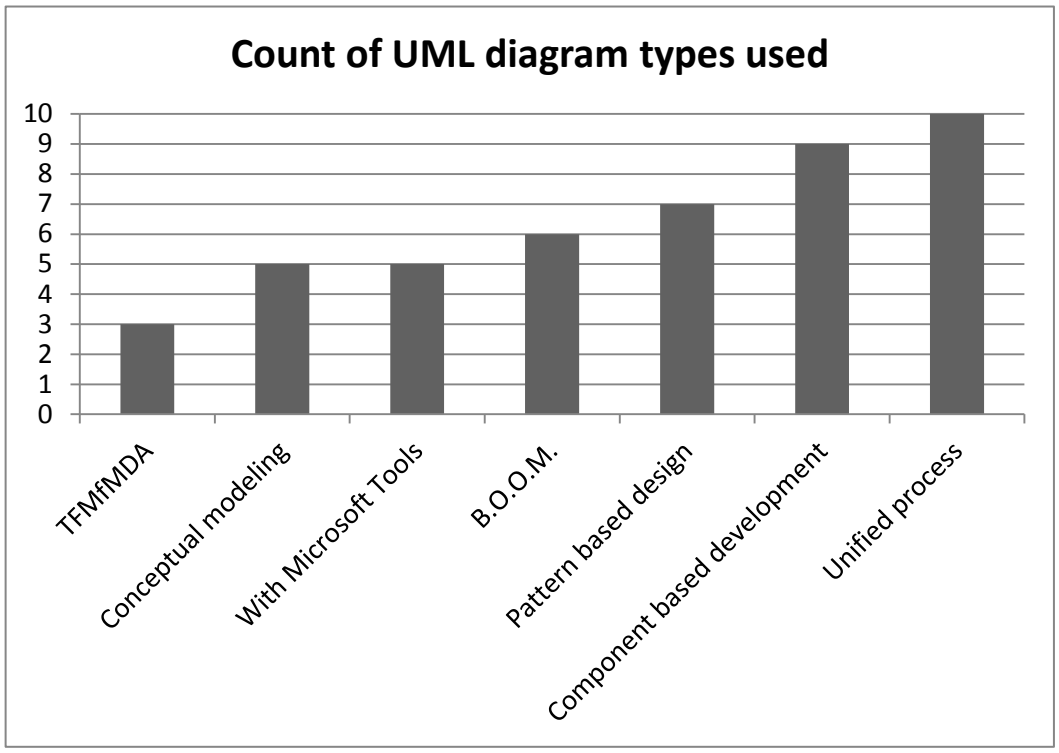


Figure 2.8. Number of UML diagrams used by UML modeling driven methods

While the benefit of applying Unified process is the coverage of whole software development lifecycle, it has some limitations – the Unified process promotes use case driven analysis of business system thus all the process is more or less use case driven (e.g., actors are identified by use cases, system class candidates are extracted from the use case narratives by

using noun analysis). As such the Unified process does not provide a formal way of analyzing and formalizing business system.

The only formal method for problem domain formalization among the reviewed methods is TFMfMDA. It uses TFM as a tool for both problem and solution domain analysis and formalization. When a TFM of system's functioning has been developed, it can be mapped onto functional requirements, goals and use cases. By mapping TFM onto functional requirements, the requirements get validated and missing, overlapping, unrealizable and conflicting requirements are found. If there are requirements that do not map onto developed TFM, then it is a signal that a new functionality is going to be introduced to the functioning of a business system through the new software. TFM can be transformed into Activity diagram and Class diagram (TFMfMDA addresses Class diagram as Conceptual class diagram). The Conceptual class diagram contains conceptual classes (without attributes and operations) and associations between them.

While TFMfMDA has formalized the very beginning of software development lifecycle, its largest limitation is the Conceptual class diagram and its development. TFM describes the functionality of the business and software system (including the responsibilities through the whole system). When TFM is transformed into Conceptual class diagram this important information of responsibilities from TFM is not transferred to Class diagram, thus raising a question: "How the responsibilities carried by classes can be determined?" As underlined by Larman in [64]: "*deciding what operations belong where, and how the objects should interact, is terribly important and anything but trivial. This is a critical step - this is at the heart of what it means to develop an object-oriented system, not drawing domain model diagrams, package diagrams, and so forth.*"

The analysis of UML application in software development industry [24] shows that the five most applied diagram type among UML diagrams are: Class diagram – 84%, Use case diagram – 71%, Sequence diagram – 70%, Activity diagram – 57%, and State diagram – 56% (percentage in braces shows how many of the review's 135 respondents are using corresponding UML diagram type). This fact is tightly related with the UML modeling driven methods – the review of such methods shows that the five most applied UML diagram types within them are: Class diagram – 100%, Use case diagram – 100%, Activity diagram – 86%, Sequence diagram – 71%, and State diagram – 71% (see Figure 2.9).

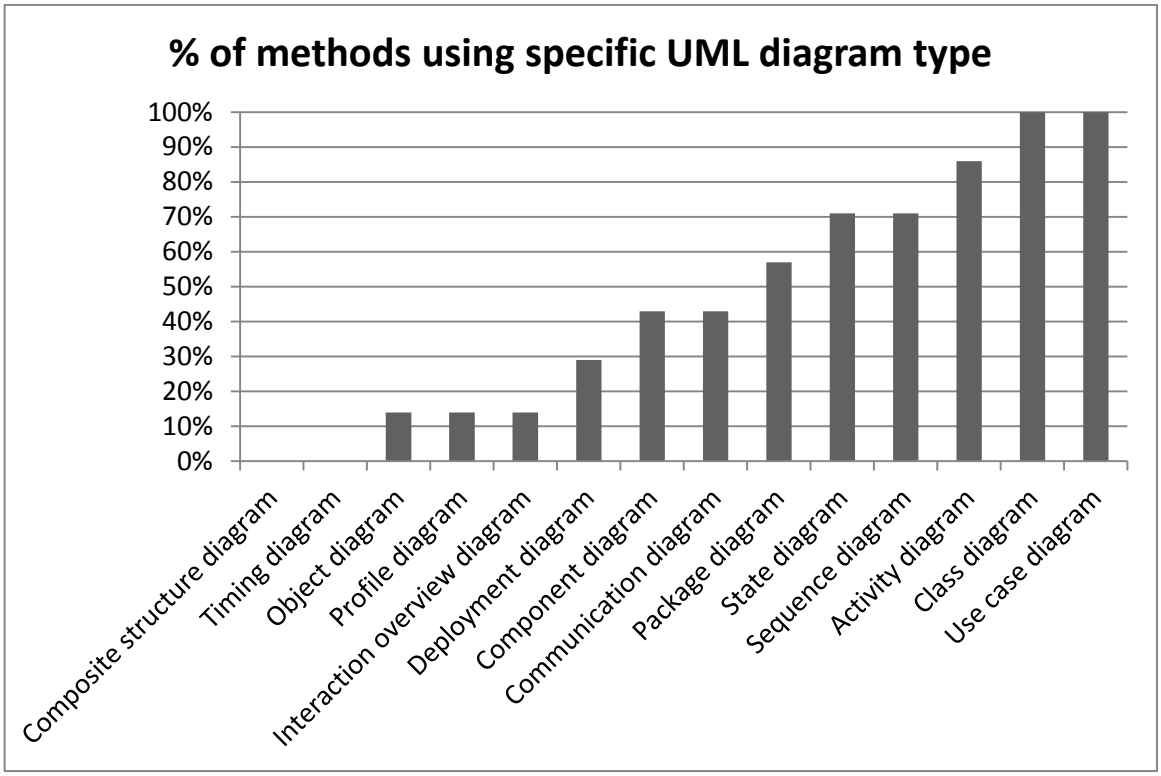


Figure 2.9. Percentage of UML diagram type application by UML modeling driven methods

2.3. Summary

The use of UML modeling driven methods supplements the application of UML in software development. While UML is a notation and as such its specification does not contain any guidelines of its application during software development process, the UML modeling driven methods fulfill this gap. In fact, the application of modeling methods reduces and even solves disadvantages of UML identified in previous chapter:

- *Size* – systematic and consistent software development activities solves issue related with the large amount of UML diagrams and their elements (i.e., UML is applied gradually thus avoiding the need to read whole language specification at once),
- *Incoherence* – through the predefined actions the modeling method tries to develop diagram by diagram thus showing the seams and transitions between diagrams (e.g., developing State diagram for each class in Class diagram),
- *Different interpretations* – UML semantics together with methodical application of UML diagrams creates shared understanding among stakeholders, and

- *Frequent subsetting* – providing UML extension (e.g., profile) and a proper modeling method that uses this extension it is clearly visible how it is related to UML elements and diagrams and how software development can benefit from this extension.

Review of UML modeling methods shows that not every method covers all the software development lifecycle. Among the reviewed methods only Unified process and MSF covers whole software development lifecycle while other methods cover just a specific part of it (e.g., analysis, design) thus impacting the number of UML diagram types that are applied by each of the method. While most of the reviewed UML modeling methods promotes use case driven software development process, the only formal method for business system (or problem domain) formalization among the reviewed methods is TFMfMDA. It uses TFM as a tool for problem domain analysis and formalization. The TFMfMDA covers only TFM, Use case, Activity, and Class diagram development. In fact, Class diagram contains conceptual classes (without attributes and operations) and associations between them, thus the responsibilities of classes are not assigned. Thus the review of UML modeling driven methods leads to the following conclusions:

- None of reviewed methods is sufficient for software development that allows to clearly trace cause-and-effect relationships in both problem and solution domains,
- Modeling methods determine the application of UML diagrams and not the UML itself (review of UML application in industry ([24]) and UML modeling methods review shows that the top five most applied UML diagrams are the same), and
- Due to the partial UML and software development lifecycle coverage and the fragmentary application of UML diagrams the software developers are forced to combine UML with several modeling methods and techniques (instead of taking UML as a notation and one UML modeling driven method) thus the application of UML gets more complicated and incomprehensible.

If the UML disadvantage of lacking causality is solved by supplementing it with mathematical topology and thus creating TopUML profile, then another UML disadvantage emerges – frequent subsetting. To address this issue, a new UML extension needs to be provided together with a proper modeling method – TopUML modeling method. To address issues related with existing UML modeling methods, the TopUML modeling method should include following aspects:

- It should ensure proper analysis of problem and solution domains thus enabling clearly tracing of cause-and-effect relationships in both domains (all software artifacts needs to be an abstraction of a well analyzed and understood problem domain unit),
- It should cover most of the UML diagrams and software development lifecycle to eliminate the need to combine together several modeling methods,
- The developed artifacts should be with high cohesion (achieved through proper analysis of objects and their responsibilities throughout the system), and in addition
- Components of developed system need to have low coupling with the rest of the system and a well-defined interface.

While the next chapter is dedicated to explore UML profiling mechanism and specify TopUML profile, the Chapter 4 defines TopUML modeling method. The TopUML profile and modeling method together solves issues and disadvantages related with UML and its application in software development process.

3. IMPROVING UNIFIED MODELING LANGUAGE

While the UML is intended to be a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a system it has a number of advantages and a number of disadvantages (see Chapter 1). The main disadvantages of UML which emerges the improvement of it are raised from the basis on which UML has been developed. The specification of UML and the UML itself is not developed basing on any theoretical principles regarding the constructs required for an effective and usable modeling language for analysis and design; instead UML arose from (sometimes conflicting) "best practices" in parts of the software engineering community [24]. This means that the UML goes without mathematics. In the field of improving UML and its application in software development the following improvement options are outlined:

- Extending UML by using UML's extensibility mechanisms,
- Formalizing the semantics of UML,
- Formalizing the way the UML is used, and
- Relating UML constructs to the problem domains.

According to the UML specification, the UML extension is divided into two ways – "lightweight" extension and "heavyweight" extension [88]. The lightweight extension is done by using profiles thus defining a new dialect of UML to customize the language for particular platforms, domain and problem solutions. The heavyweight extension (or the first class extension) is done by using metamodeling based on MOF (in this case all the benefits of creating profile are lost and it can be a difficult task to put it into the practice).

If there is need to extend the UML, at first it is needed to draw the scope of UML extension. If the new language will use most of the UML, then profiles are suitable choose for that solution. If the new language uses only small part of UML or there is need to use more complex features of UML such as redefinition, then creating a complete new language by using MOF metamodeling should be considered. The relationship between the UML and the new language under consideration is shown by using Venn diagram in Figure 3.1. The Venn diagram clearly shows that if there is much overlap between the concepts in UML and those within the new language then UML should be extended and if there is little overlap – then MOF based solution should be created. An example of using both approaches (UML profile based extension and MOF based solution) to develop a new language is demonstrated in the UML testing profile [86]. In other words – the UML testing profile specification defines the same language by using two different approaches.

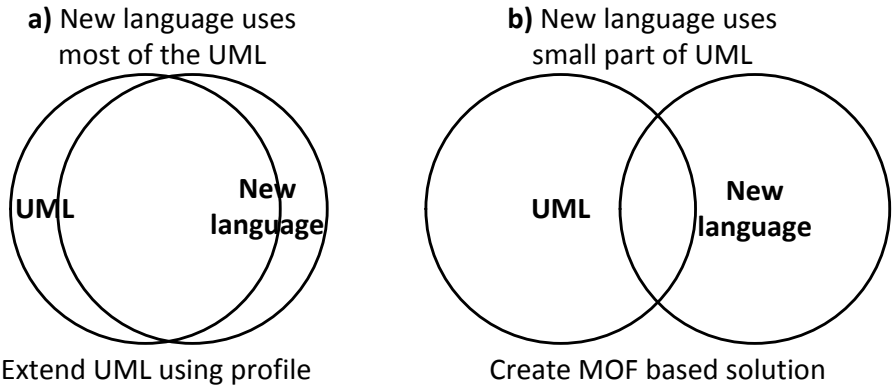


Figure 3.1. Venn diagram showing relationship between UML, the new language under consideration and the suggested language creation solution

The UML specification itself is defined by using metamodeling approach – a metamodel is used to specify the model that builds UML. One of the UML metamodel principles is its extensibility. In fact, the most common and suitable way for improving UML is to use its extensibility mechanisms – the profiles. By improving UML with the profile mechanism, it is possible to adapt and use ordinary UML compliant modeling tools [88]. Thus by creating a profile of UML the costs of adaption in industry for such new language is lowered and it can be adapted faster (in comparison with creating a MOF based solution which forces to implement new modeling tools along with the very new language).

Regardless of which UML extension way is used it is important to add mathematical foundations to the specification, thus making UML and its use more precise and formal. As pointed out in [26] and [103], the UML can be strengthened by using the mathematical topology. The use of topology reflects extending the UML to support topology in its diagrams and formalizing the way the UML is used. Having more precise and formal language makes it less expensive to adapt in software development process and tools [42].

3.1. Profiling UML and Metamodeling

The UML specification is defined using a metamodeling approach that adapts formal specification techniques (it is needed to notice, that this approach lacks some of the rigor of a formal specification method – mainly due to the extensive use of natural language). The purpose of metamodeling is to use metamodel for specifying the model that comprises UML. UML specification is organized in two parts – Infrastructure [88] (represented by *InfrastructureLibrary*) and Superstructure [89] (represented by *SuperstructureLibrary*).

The Infrastructure is represented by two packages (see Figure 3.2): *InfrastructureLibrary* (consists of two subpackages *Core* (contains core concepts used when metamodeling) and *Profiles* (defines the mechanisms that are used to customize metamodels)) and *PrimitiveTypes* (consists of a few predefined primitive types that are commonly used when metamodeling) and it satisfies following design requirements [115]:

- Define a metalanguage core that can be reused to define a variety of metamodels,
- Architecturally align UML, MOF, and XMI so that model interchange is fully supported, and
- Allow customization of UML through profiles and creation of new languages (family of languages) based on the same metalanguage core as UML.

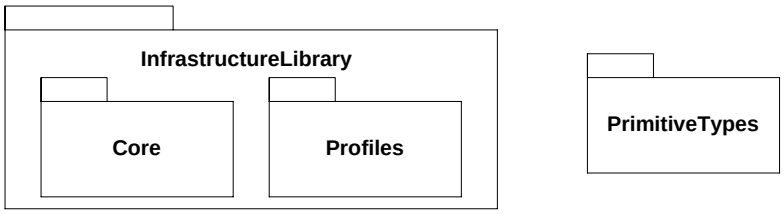


Figure 3.2. Packages of UML's *InfrastructureLibrary* [88]

The *Core* package is a complete metamodel particularly designed for high reusability, where other metamodels at the same metalevel either import or specialize its specified metaclasses (see Figure 3.3 where it is shown how UML, *Profiles*, and MOF each depends on a common core). Common core is defined as a *Core* package, thus enabling to share model elements between UML and MOF and ensuring that UML is defined as a model that is based on MOF as a metamodel.

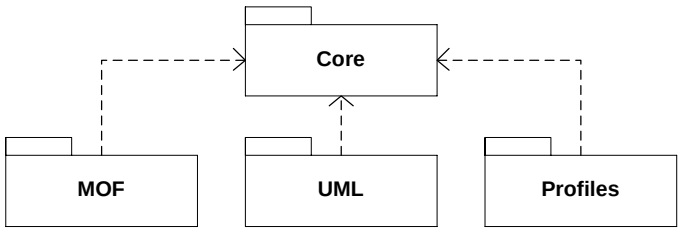


Figure 3.3. Dependencies between Core, MOF, UML and Profiles packages

The mechanisms and tools of improving and extending UML by using profiles are given in the *Profiles* package, which depends on the *Core* package. The *Profiles* package

defines the mechanisms used to tailor existing metamodels towards specific platforms, domains, problem solutions or software process modeling. The primary target for profiles is UML, but it is possible to use profiles together with any metamodel that is based on (i.e., instantiated from) the common core. A profile must be based on a metamodel such as the UML that it extends. A profile of UML is a set of stereotypes – when defining a UML profile the stereotypes are defined to extend classes in the UML metamodel. A stereotype describes how an existing metaclass is extended thus enabling the integration of platform or domain specific terminology or notation in the modeling language.

The UML Superstructure metamodel is specified by the UML package *SuperstructureLibrary*, which is divided into a number of packages that deal with structural and behavioral modeling. One of the primary uses of the UML Infrastructure specification is that it should be reused when creating other metamodels. The *SuperstructureLibrary* reuses the *InfrastructureLibrary* in two different ways: all of the UML metamodel is instantiated from meta-metaclasses that are defined in the *InfrastructureLibrary*, and the UML metamodel imports and specializes all metaclasses in the *InfrastructureLibrary*.

Defining language by using metamodeling includes dealing with three meta-layers that always have to be taken into account:

1. Language specification – the metamodel,
2. User specification – the model, and
3. Objects of the model – the run-time instance of model.

This three-layer structure can be applied recursively thus creating possibly infinite number of meta-layers. The metamodel in one case can be a model in another case (e.g., from the MOF viewpoint the UML is a model, from profile viewpoint the UML is a meta-model). The UML metamodel commonly is viewed as a four layer metamodel hierarchy [115]:

- *M3 meta-metamodeling layer* – defines a language for specifying a metamodel (for example, MOF)
- *M2 metamodeling layer* – defines a language for specifying models (i.e., defines metamodel as an instance of a meta-metamodel, meaning that every element of the metamodel is an instance of an element in the meta-metamodel)(for example, UML, specific profile of UML)
- *M1 modeling layer* – defines languages that describe semantic domains, i.e., to allow users to model a wide variety of different problem domains, such as software, business processes, and requirements (for example, user model which is an instance of UML metamodel).

- *M0 run-time instances layer* – contains the run-time instances of model elements defined in a model.

The application of profiles as the UML extension mechanism has number of positive and negative aspects. When a profiling has been chosen as an extension solution for the development of the new language the following aspects should be kept in mind:

- Positive aspects:
  - Profiles are well described in UML specification,
  - Can add structure, additional constraints, and formal notation,
  - Standard means for icons definition and well defined display options,
  - Application of profiles and how to use them is well defined, and
  - Low development costs – profiles can be used within existing UML tools.
- In some cases it could be needed to remove some existing element or change its existing specification, then the negative aspects of profile application arise:
  - Cannot remove existing constraints,
  - Cannot redefine existing types, and
  - Cannot modify existing structures (i.e., existing relationships between language elements cannot be removed and changed).

3.1.1. Overview of UML Profiles

The first UML profile was presented at UML'99 [42] under the title "Towards a UML Extension for Hypermedia Design" [12]. Since that a number of UML profiles are developed and published [111]. They cover different problem domains, like business process, requirements, ontologies, device capabilities, parallel applications, and hypermedia. Most of the UML profiles have been presented at conferences which are specialized in conceptual modeling and software engineering. As Pardillo has underlined in his research ([111]) the presentation of UML profiles "shows a great disparity regarding both the profile definition process and the quality of the UML-profile presentation", thus leading to difficult comparison, discussion and usage of presented UML profiles.

To better understand the development and structure of UML profile specification this section holds an overview of four UML profiles which are created by different teams. The profiles for review are selected based on following criterions (at least one criterion for each profile is satisfied):

1. Widely accepted and used,
2. Developed at research or scientific institution, or
3. Published by OMG (i.e., publishers of UML specifications).

The review of UML profiles includes analysis of following UML profiles which are selected based on the three criterions listed above:

- *Executable UML* [76] (xUML), see next Subsection (satisfies the first criterion),
- *Topological Functioning Model for Model Driven Architecture* [5] (TFMfMDA, satisfies the second criterion) – created at Riga Technical University by Asnina,
- *Object Modeling Group System Modeling Language* [90] (OMG SysML, satisfies third criterion), and
- *Service Oriented Architecture Modeling Language* [87] (SoaML, satisfies third criterion).

According to [111], the systematic review of UML profiles should include two main aspects of UML profiling practices – profile definition process and profile presentation quality; thus the selected profiles are measured against the following quantitative and qualitative criterions:

- *Qualitative criterions*:
  1. **UML version extended** – shows the version number of UML which is used as a basis for profile development,
  2. **Purpose of the profile** – briefly describes the purpose and goal of the profile,
  3. **Problem domain addressed** – identifies problem domain addressed and the specific task which needs to be solved by the profile,
  4. **Formalization of UML semantics** – identifies if semantics of existing UML elements are formalized,
  5. **Formalization of profile semantics** – identifies if semantics of the new elements (i.e., stereotypes) are expressed in formal statements,
  6. **Extension approach** – evaluates the applied UML extension approach, and
  7. **Specification structure** – evaluates the profile definition structure and whether it follows some specification guidelines or style.
- *Quantitative criterions*:
  1. **Metaclasses extended** – count of metaclasses that are extended,
  2. **Stereotypes defined** – count of stereotypes defined in profile,
  3. **Diagrams extended** – number of UML diagrams that are extended,

4. **Diagrams introduced** – count of diagrams that are new to profile (i.e., defined in profile and does not exist in UML specification), and
5. **Total count of diagrams** – total count of diagrams included into profile.

The result of the evaluation against qualitative criterions is given below in Table 3.1.

Table 3.1

Evaluation of qualitative criterions for selected profiles

No.	Criterion	Profile	Evaluation
1.	UML version extended	xUML	1.4
		TFMfMDA	2.0
		OMG SysML	UML4SysML (based on UML version 2.0)
		SoaML	2.0
2.	Purpose of the profile	xUML	Graphically specifies a system at high level of abstraction, abstracting away both specific programming languages and decisions about the organization of the software. The models are testable, and can be compiled into a less abstract programming language to target a specific implementation. It supports MDA through the specification of PIM, and the compilation of the PIM into PSM. Together with model compiler, xUML models are executable – model compiler turns xUML model into an implementation using a set of decisions about the target hardware and software environment.
		TFMfMDA	An attempt to raise the formalization level of problem domain modeling within MDA at the CIM level and CIM-to-PIM transformation. It includes a new type of diagram – TFM; thus allowing to analyze and model problem and solution domain functioning. According to [100] within software development process this diagram should be created before any other diagram gets constructed. This conforms to the extended MDA life cycle [94].
		OMG SysML	A general-purpose modeling language for systems engineering applications. It supports the specification, analysis, design, verification, and validation of a range of systems (e.g., hardware, software, information, processes, personnel, and facilities) and it is particularly effective in specifying requirements, structure, behavior, allocations, and constraints on system properties to support engineering analysis. The authors of OMG SysML anticipate that it will be customized to model domain-specific applications (for example, such as automotive, aerospace, communication, and information systems).
		SoaML	A language for designing services within a service-oriented architecture (SOA). It supports range of modeling requirements for SOAs, including

No.	Criterion	Profile	Evaluation
			the specification of services systems, individual service interfaces, and service implementations. This is done in a way to support automatic generation of derived artifacts following MDA based approach.
3.	Problem domain addressed	xUML	Software modeling and development within MDA.
		TFMfMDA	Formal problem and solution domain modeling and analysis.
		OMG SysML	Systems engineering.
		SoaML	Modeling of SOA and services.
4.	Formalization of UML semantics	xUML	None.
		TFMfMDA	None.
		OMG SysML	None.
		SoaML	None.
5.	Formalization of profile semantics	xUML	Semantics and constraints expressed in natural language.
		TFMfMDA	Semantics and constraints expressed in natural language and OCL.
		OMG SysML	Semantics and constraints expressed in natural language.
		SoaML	Semantics and constraints expressed in natural language.
6.	Extension approach	xUML	Elements of UML are supplemented with action language and action concepts to make the UML diagrams executable.
		TFMfMDA	UML is extended only where it is necessary to introduce new elements for definition of TFM and elements related to it.
		OMG SysML	Since the OMG SysML profile does not use every modeling element and every diagram which is included in UML version 2.0, initially a narrowed version of UML – UML4SysML – is developed. The OMG SysML specification defines the language architecture in terms of the parts of UML 2 that are reused and the extensions to UML 2.
		SoaML	UML is extended only where it is necessary to accomplish the goals and requirements of SOA and service modeling.
7.	Specification structure	Executable UML	Since this profile is created basing on UML version 1.4, which contains no Profile diagram and the profiling mechanism, the specification style is not convenient with UML specification style. All of the new concepts are defined only using natural language. For the diagrams and elements the authors of xUML try to keep the following specification points: 1. UML diagram (description of which can be supplemented by set of definitions), 2. Diagram element (description of which are supplemented by set of definitions), and 3. How to apply diagram in software development process.
		TFMfMDA	Specification of TFMfMDA includes profile diagram which shows metaclasses and stereotypes extending them. Each stereotype has brief

No.	Criterion	Profile	Evaluation
			description of its semantics in natural language and constraints in OCL. A standalone metamodel is defined for TFM.
		OMG SysML	Specification structure follows the style in which the UML specification itself is written. Specification is divided into parts that contain specific modeling aspects; each part contains definition of elements needed to construct specific diagram. Elements are described in the same way as in UML specification, thus containing determined specification parts.
		SoaML	The same as used for OMG SysML (see above).

The result of the evaluation against quantitative criterions is given below in Table 3.2.

Table 3.2

Evaluation of quantitative criterions for selected profiles

Profile / Criterion	xUML	TFMfMDA	OMG SysML	SoaML
Metaclasses extended	?	6	25	12
Stereotypes defined	?	8	40	20
Diagrams extended	?	0	3	0
Diagrams introduced	0	1	2	0
Total count of diagrams	8	14	9	13

Summary of positive and negative aspects for each evaluated profile is given below in Table 3.3.

Table 3.3

Positive and negative aspects of selected UML profiles

No.	Aspect	Profile	Evaluation
1.	Positive aspects	xUML	Models are testable and executable. Profile contains approach of applying it.
		TFMfMDA	Profile has an approach (called TFMfMDA) for its use in software development.
		OMG SysML	Profile definition structure.
		SoaML	Profile definition structure.

No.	Aspect	Profile	Evaluation
2.	Negative aspects	xUML	Profile definition style – no stereotypes and extended metaclasses modeled.
		TFMfMDA	TFMfMDA approach uses problem domain graph as intermediate model between TFM and class diagram but in the same time this graph is not included in TFMfMDA profile.
		OMG SysML	The introduced requirements diagram is just a repository of textual requirements.
		SoaML	Names of stereotypes are the same as for the metaclasses they extend. It could lead to confusion of used element and misunderstanding of developed diagrams.

The review of profiles shows that there is no unified profile definition template or approach – each author defines profile on its own. After doing systematic review of UML profiles presented at UML and ER conferences from 1999 till 2010, Pardillo in his research [111] has outlined that "*the low presentation quality point out the need of more formal methods and templates to present UML profiles*." Only two of four reviewed profiles – OMG SysML and SoaML – have huge similarities in the profile specification (the specification structure is about 85% the same). The specification of these two profiles follows the overall specification structure of UML; thus if the reader is familiar with the UML specification understanding of these profiles is relieved. Summarizing issues related to UML profile specification techniques and templates, the next subsection provides guidelines for profile development.

3.2. Developing a Profile for UML

Developing a profile for UML should be done in consistent way by using some unified profiling approach or template. Since UML specification contains only the definition of elements that are building up a profile and does not provide guidelines or process on how to apply these elements, before creating a profile for UML it is needed to define guidelines of profile development. While this section summarizes up the contents of profile in the terms of used elements from UML specification, the next subsection gives guidelines of contents for a profile specification.

The UML profile diagram consists of following elements [89]:

- *Metamodel* – a referenced model that is extended through the profile (e.g., UML),

- *Reference* – a dependency relationship with attached stereotype *«reference»* that is directed from profile to referenced metamodel,
- *Profile* – a special package that extends a referenced metamodel by adding stereotypes to it,
- *Metaclass* – a class that is extended by stereotype,
- *Stereotype* – extends existing UML vocabulary by adding a new element to it and it describes how an existing metaclass can be extended enabling the integration of platform or domain specific terminology or notation in the modeling language (a set of stereotypes build up the profile); a stereotype extension is used to indicate that the properties of a metaclass are extended through a stereotype,
- *Extension* – a special binary association, extension end is used to tie an extension to a stereotype when extending a metaclass, and
- *Profile application* – a dependency relationship with attached stereotype *«apply»* between a package and a profile that allows to use the stereotypes from the profile in the model elements of the source package (profile application relationship is directed from package that applies profile to the profile package).

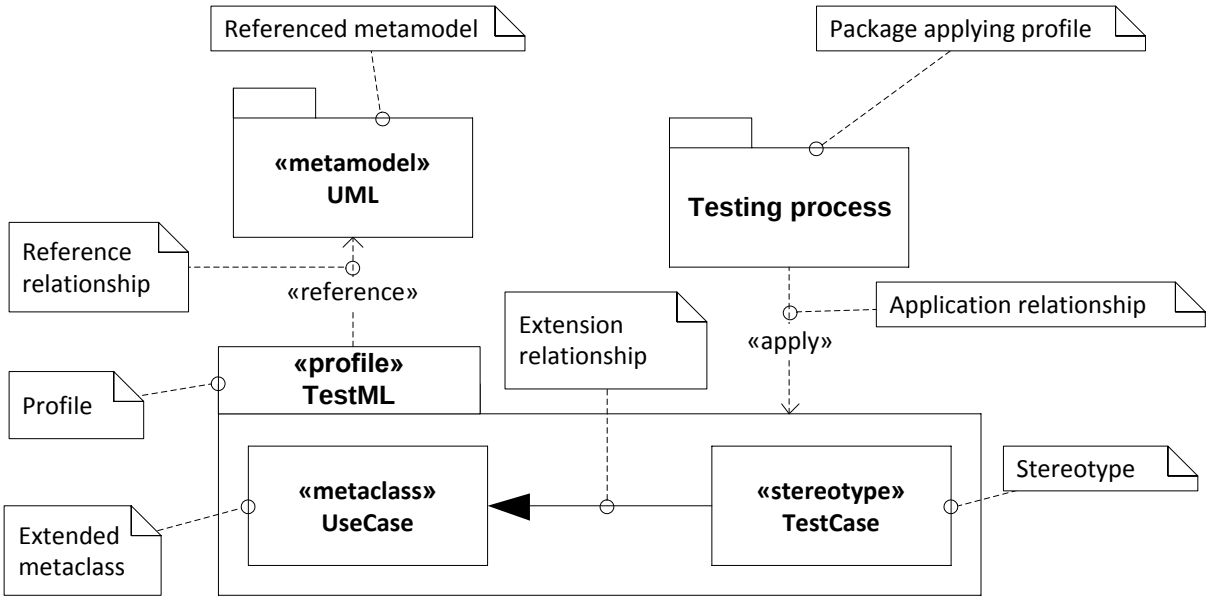


Figure 3.4. Example of profile and profile application

An example showing generic profile with name *"TestML"* is given in Figure 3.4. The example of profile extends UML by adding stereotype *"TestCase"*. The stereotype

"TestCase" extends metaclass *"Class"* by using extension relationship. The profile is applied by the package *"Testing process"*.

3.2.1. Profile Specification Template

As shows the specification of OMG SysML [90] and SoaML [87] the best practice for UML profile specification is to use the same structure as used for UML specification, thus if the reader is familiar with UML specification it is easier to read and understand the specification of specific UML profile. UML specification is created by keeping in mind following aspects [88]: *Correctness*, *Precision*, *Conciseness*, *Consistency*, and *Understandability*.

The profile specification should start with profile diagram showing the referenced metamodel and how the profile extends it. After profile diagram, one or more package diagrams should be provided showing the packages of which the profile consists. UML elements within its metamodel and specification also are grouped into packages. At this point it is advised to reuse the package specification style used in UML specification. Each package and each class in the UML specification (both – infrastructure [88] and superstructure [89] – specifications) has following structure:

- **Package** - this clause provides information for each package and each class in the UML metamodel or profile. Each package specification contains one or more of the following sub clauses (*Classes*, *Diagrams*, and *Instance model*).
    - **Classes** – contains a list of the classes specifying all the constructs defined in package. This sub clause begins with one diagram or several diagrams depicting the abstract syntax of the constructs (i.e., the classes and their relationships) in the package, together with some of the well-formedness requirements (multiplicity and ordering). Then follows a specification of each class in alphabetic order. Each class specification has following sub clauses:
        - **Description** – an informal definition of the metaclass specifying the construct in UML.
        - **Attributes** – list of all attributes for metaclass. Attributes are given together with a short explanation.
        - **Associations** – list of all member ends of associations connected to this class (associations are listed in the same way as attributes).

- **Constraints** – well-formedness rules of the metaclass. These rules specify constraints over attributes and associations defined in the metamodel. Mostly they are defined by using OCL expressions together with an informal explanation of the expression.
- **Additional Operations** (optional) – contains any additional operations on the class which are needed for the OCL expressions.
- **Semantics** - the meaning of a well formed construct is defined using natural language (can include formal definition of construct's semantics).
- **Semantic Variation Points** (optional) – objective of a semantic variation point is to enable specialization of that part of UML for a particular situation or domain.
- **Notation** – presents the notation of the construct (i.e., class).
- **Presentation Options** (optional) - if there are different ways to show the construct, these ways are described in this sub clause.
- **Style Guidelines** (optional) – describes non-normative conventions that are used in representing some part of a model.
- **Examples** (optional) – examples of how the construct is to be depicted.
- **Rationale** (optional) – if there is a reason why a construct is defined like it is, or why its notation is defined as it is, then it is given in this sub clause.

- **Diagrams** – this sub clause is included into specification to describe specific kind of diagram, if this diagram uses the constructs that are defined in this package.
- **Instance model** – shows an example of applying constructs defined in this package.

3.3. Topological Unified Modeling Language – an UML Improvement

Topological Unified Modeling Language (TopUML) is a combination of UML and formalism of Topological Functioning Model (TFM) and is based on the principles of metamodeling and MOF. Idea of Topological UML is adapted from [93] where it is shown that "there is a lack of mathematical formalism by drawing UML diagrams". The main aim of improving UML is by transferring topology and mathematical formalism of TFM to UML. The first research results in [102] shows that it is possible to transform topology from TFM into UML class diagrams thus creating a new diagram called *"Topological class diagram"*

and that this diagram can be refined to have all necessary information for software development [31].

TopUML is developed as a profile of UML and its specification takes advantage of the package merge feature of UML to merge extensions into UML. The created profile provides a UML specific version of the metamodel that can be incorporated into standard UML modeling tools [88]. Since there is no specific profile definition method or approach [111] the TopUML profile definition is done by using results of profile definition analysis as given in Section 3.2. TopUML development is based on following steps:

- Extend UML by using profile mechanism, thus creating TopUML profile, and
- Define guidelines for using TopUML in practice (thus formalizing the way the TopUML is used).

Despite that together with TopUML is defined method on how to apply it in practice, the TopUML language is intended to support multiple approaches and methods (e.g., structured, object-oriented, conceptual). It is assumed that each methodology may impose additional constraints on how a TopUML construct or diagram may be used and applied.

3.3.1. Topology in UML Diagrams

The analysis of UML diagrams included in the version 2.4.1 shows which of the diagrams already have elements that can reflect cause-and-effect relationships and which diagrams should be extended. The extension of a number of UML diagrams is necessary to develop a framework for TopUML and later define TopUML profile diagram for it. The analysis of existing UML diagrams is given in Table 3.4.

Table 3.4

Cause-and-effect relationships in UML version 2.4.1 diagrams

No.	Diagram	Has cause-and-effect relationship	Requires extension	Description
Structure diagrams				
1.	Class diagram	No	Yes	Relationships which are used in Class diagrams do not reflect cause-and-effect relations between classes, e.g., *Dependency* shows that one class uses another without any clues how this class and for what is used, *Association* shows structural relationships between classes (i.e., the structure),

No.	Diagram	Has cause-and-effect relationship	Requires extension	Description
				Generalization shows inheritance between classes. By adding *Topological relationship* to class diagram it is possible to model cause-and-effect relations between classes.
2.	Component diagram	No	No	Component diagram shows relationships between logical components (by using interfaces and ports), i.e., it shows how a software system will be composed of a set of deployable components. Therefore Component diagram do not need to include topological relationship.
3.	Composite structure diagram	No	No	A composite structure diagram shows the internal structure of a class or collaboration. Since the difference between components and composite structure is small, Composite structure diagram do not need to include topological relationship.
4.	Deployment diagram	No	No	A deployment diagram shows a set of nodes and their relationships. Deployment diagrams are used to illustrate the static deployment view of architecture. Therefore Deployment diagram do not need to include topological relationship.
5.	Object diagram	No	No	Object diagram shows instance of classes in class diagram with objects and their corresponding relationships (i.e., a snapshot taken at specific time). Since triggering of topological relationship causes a change in system the snapshot taken is altered thus creating a new snapshot. Therefore Object diagram do not need to include topological relationship.
6.	Package diagram	No	No	Package diagram models contents of packages and relationships between packages, i.e., the structure of packages. Relationships used in package diagrams are *import* and *export*. If an element of one package uses element from another package then there is a relationship between the two packages. Relations between packages are used to enable reuse of package contents. Therefore Package diagram do not need to include topological relationship.
7.	Profile diagram	No	No	Since Profile diagram allows to define a new language belonging to UML language family, it allows to add new language elements or extend existing ones and it does not include information about functioning of systems. The

No.	Diagram	Has cause-and-effect relationship	Requires extension	Description
				Profile diagram is not used to model and design systems. Therefore Profile diagram do not need to include topological relationship.
Behavior diagrams				
8.	Activity diagram	Yes	No	Cause-and-effect relations in Activity diagram is reflected by the control flow from one node to another. Therefore Activity diagram do not need to include additional topological relationship.
9.	Use case diagram	No	Yes	Associations between actors and use cases show that there can be interaction (message sending) between actor and use case. Actor can be a user or another system. Addition of topological relationship to Use case diagram enables to design and show formally defined communication between the use case and actor thus showing who is triggering the communication.
10.	State diagram	Yes	No	Topological relationship within State diagram is reflected by transition relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. Therefore State diagram do not need to include additional topological relationship.
Behavior diagrams: Interaction diagrams				
11.	Sequence diagram	Yes	No	Message sending from one lifeline to another (i.e., a series of messages) establishes a chain of causality in Sequence diagram thus it already has constructs to reflect cause-and-effect relationships. Therefore Sequence diagram do not need to include additional topological relationship.
12.	Communi-cation diagram	Yes	No	Since both Sequence diagram and Communication diagram derive from the same information in UML metamodel message sending from one lifeline to another (i.e., a series of messages) establishes a chain of causality in Communication diagram thus it already has constructs to reflect cause-and-effect relationships. Therefore Communication diagram do not need to include additional topological relationship.
13.	Interaction overview diagram	Yes	No	Interaction overview diagram combines aspects of activity diagrams and sequence diagrams thus it already has constructs to reflect cause-and-effect relationships.

No.	Diagram	Has cause-and-effect relationship	Requires extension	Description
				Therefore Interaction overview diagram do not need to include additional topological relationship.
14.	Timing diagram	Yes	No	Timing diagrams describe behavior of both individual classifiers and interactions of classifiers, focusing attention on time of occurrence of events causing changes in the modeled conditions of the lifelines. Therefore Timing diagram already has constructs to reflect cause-and-effect relationships and do not need to include additional topological relationship.

Analysis of UML diagrams shows that six of fourteen UML diagrams already have constructs that can reflect cause-and-effect relationships within the system. These diagrams are Activity, State (or State machine), Sequence, Communication, Interaction overview, and Timing diagram. The analysis of UML diagrams provides results of diagrams which are missing topological relationships and which one of those should be extended. Extension should be provided for two UML diagrams: Class diagram and Use case diagram.

3.3.2. TopUML Diagrams

According to the TopUML base idea to combine formalism of TFM with UML and to create TopUML in accordance with UML extension mechanisms, the new language includes all diagram types from UML and a new diagram type – Topological functioning model (thus making a family of fifteen diagrams). The analysis of topology in UML diagrams shows that there are two diagrams which should be extended in order to include topological relationship: Class diagram and Use case diagram. Extension of these two diagrams means that it is possible to reflect cause-and-effect relations by using these diagrams. The extended versions of these two diagrams are called *"Topological class diagram"* and *"Topological use Case diagram"*.

The diagrams included into TopUML language specification are as follows: Topological class diagram, Component diagram, Object diagram, Composite structure diagram, Deployment diagram, Package diagram, Profile diagram, Topological functioning model, Activity diagram, Topological use case diagram, State diagram, Sequence diagram,

Communication diagram, Interaction overview diagram, and Timing diagram. The diagram types included in TopUML are shown in Figure 3.5, where with bolder lines are denoted the new diagram – Topological functioning model – and two extended diagrams – Topological class diagram and Topological use case diagram.

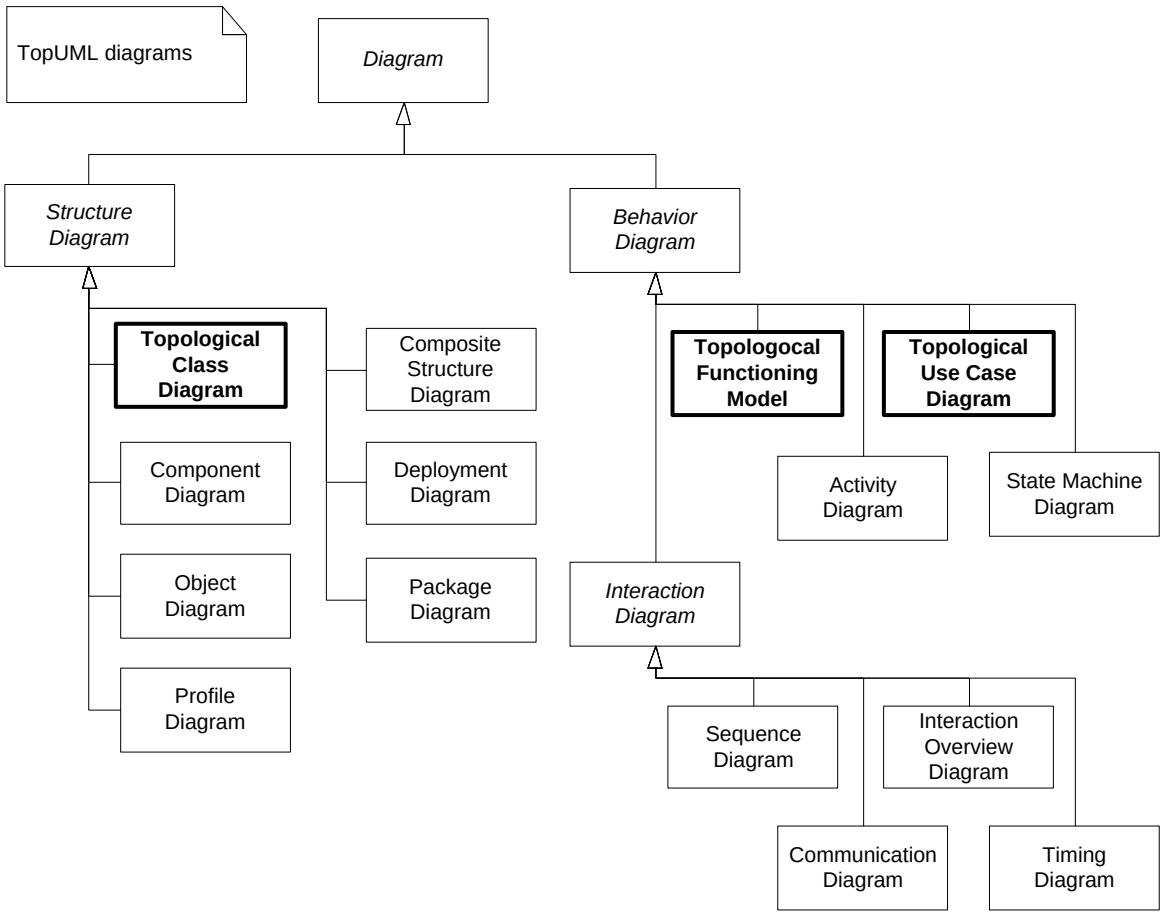


Figure 3.5. Diagram types included in TopUML

3.3.3. TopUML Profile

The profile diagram specifying TopUML language consists of eight stereotypes and two enumerations which are divided into four packages. The top-level profile diagram of TopUML is given in Figure 3.6 which shows the related metamodel and relationships between packages in profile. The packages are used to group together elements basing on their intent and semantics and to ease the evolution of TopUML (i.e., creation of new TopUML versions). The packages that build up TopUML profile are as follows:

- *TopologicalRelationships* – contains constructs related to relationships

- *TopologicalStructure* – contains constructs related to structure representation
- *TopologicalBehavior* – contains constructs related to behavior modeling, and
- *TopologicalModels* – contains diagram types added to UML by TopUML profile.

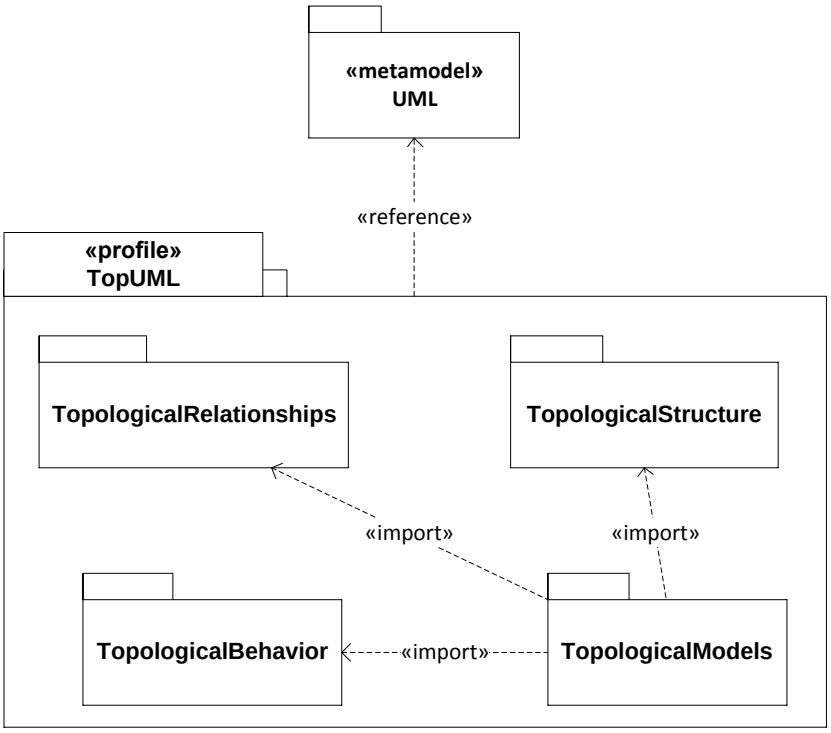


Figure 3.6. TopUML profile top level package

TopUML profile packages are designed to provide the necessary constructs to create Topological functioning model, Topological class diagram, and Topological use case diagram. Stereotypes included into each package are used across multiple diagram types thus making TopUML profile more compact and without needless constructs. Below is given brief description of each package and stereotype included into TopUML profile (full specification of TopUML profile is given in in Appendix 2 on page 174).

The topological relationships package is given in Figure 3.7 as **package *TopologicalRelationships*** and it contains following stereotypes:

- *TopologicalRelationship* – topological relationship is a binary relationship that shows a cause-and-effect relation between two elements – source element and target element. A topological relationship is assertion that indicates that the effect element can be triggered only by the cause element thus showing that effect element is executed only after the cause element executes.

- *LogicalRelationsip* – logical relationship represents logical relation between two or more topological relationships belonging to TFM (an instance of *TopologicalFunctioningModel*). It can show conjunction (*and*), disjunction (*or*), and exclusive disjunction (*xor*).

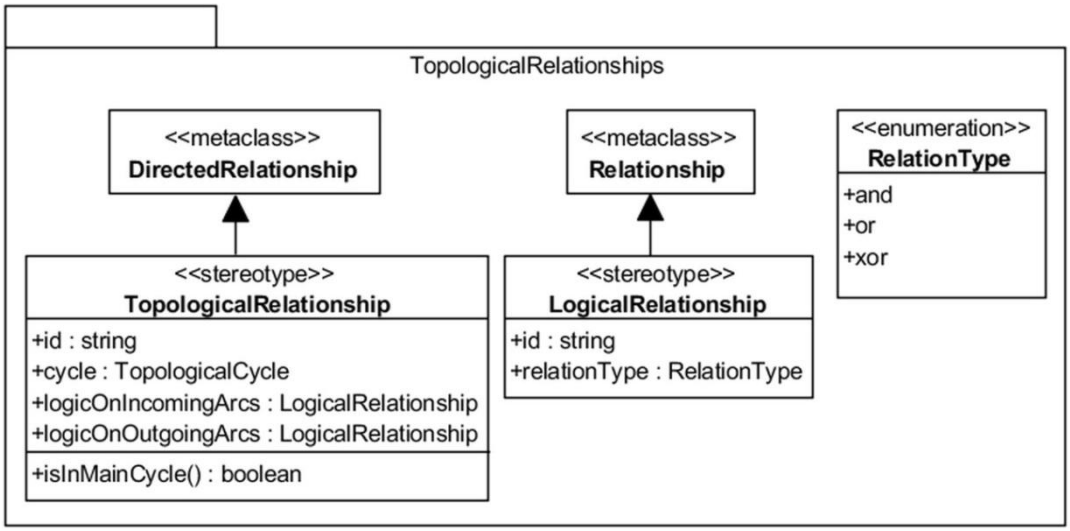


Figure 3.7. *TopologicalRelationships* package

The topological relationships package is given in Figure 3.8 as **package *TopologicalStructure***.

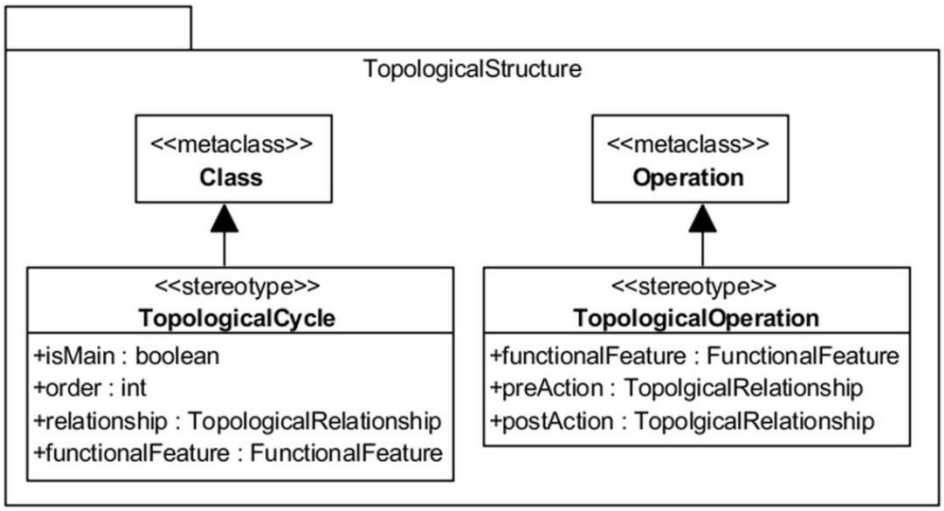


Figure 3.8. *TopologicalStructure* package

TopologicalStructure package contains following stereotypes:

- *TopologicalCycle* – topological cycle represents directed functional cycle of system; it consists of elements and relationships between them. Topological cycle can show the "main" functionality that has a vital importance in the functioning of system, i.e., by interrupting the main cycle the system can no longer function or its functioning is deformed.
- *TopologicalOperation* – topological operation is a behavioral feature of classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior, and related functional features and topological relationships for specifying cause-and-effect relations within system, thus allowing a cause-and-effect relations to be modeled within the system by means of behavioral features (e.g., in topological class diagram).

Topological behaviors package is given in Figure 3.9 as **package *TopologicalBehavior***.

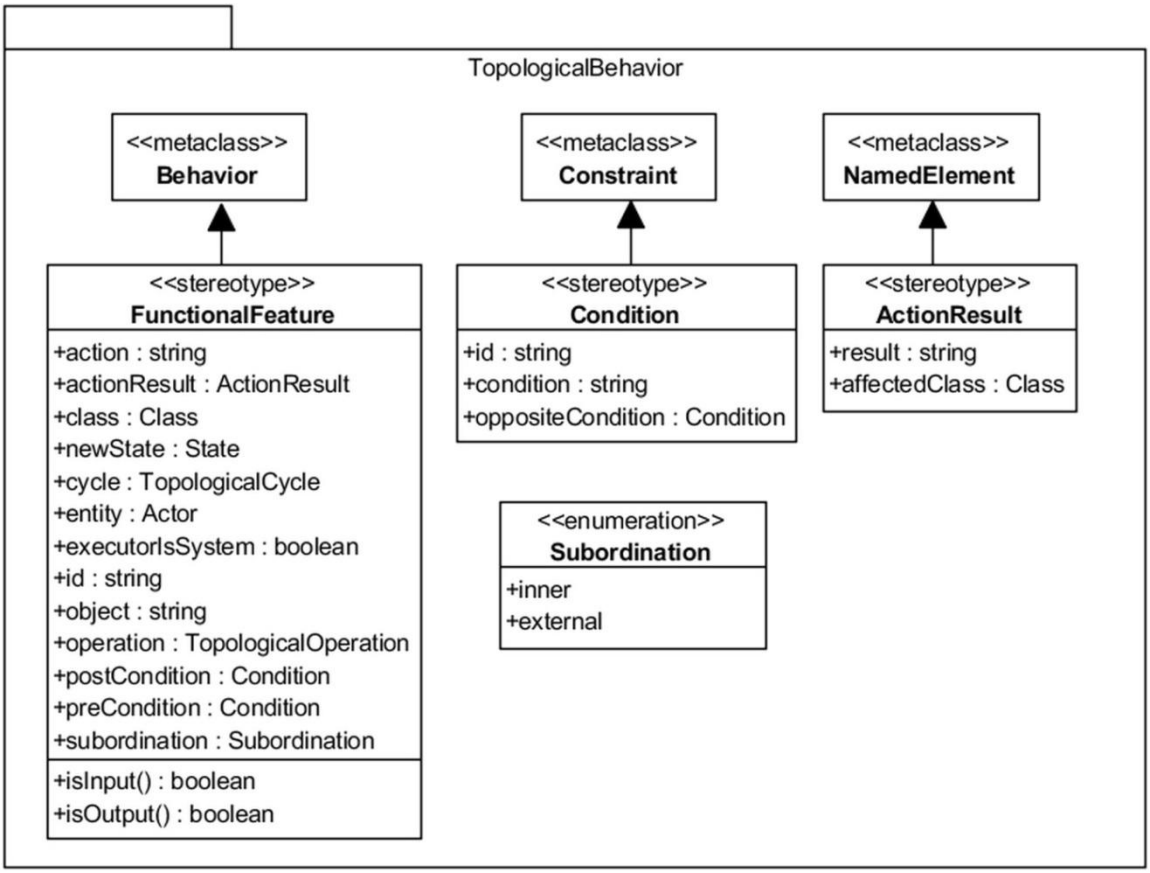


Figure 3.9. *TopologicalBehavior* package

TopologicalBehavior package contains following stereotypes:

- *FunctionalFeature* – functional feature is a description of an atomic business action (i.e., it cannot be separated into a number of other business actions). Each functional feature is a unique tuple (stereotype *FunctionalFeature* is an abstraction of this tuple).
- *Condition* – condition shows pre- and post- conditions within system. To enter the execution of behavior (e.g., functional feature) all preconditions of it should be true and to exit the execution of this behavior all postconditions should be evaluated to true. In the context of business system a condition also can be atomic business rule.
- *ActionResult* – action result specifies a result of object's action together with affected objects. For example, by registering customer in the registration journal a registration entry is created.

The topological models package is given in Figure 3.10 as **package *TopologicalModels***.

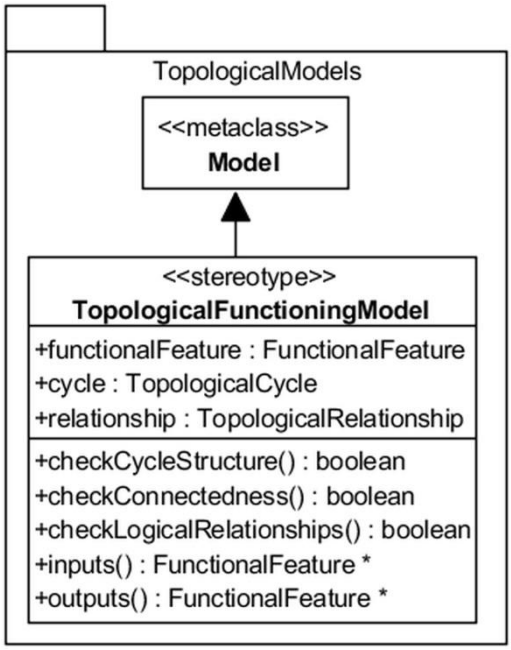


Figure 3.10. *TopologicalModels* package

TopologicalModels package contains one stereotype:

- *TopologicalFunctioningModel* – represents TFM by using UML metamodeling constructs. TFM is a mathematical model that shows functioning of a system in the form of directed graph consisting of functional features and topology between

them. Functional features embed information of systems functioning and its structural description while topology defines cause-and-effect relations between them.

3.3.4. Metamodels of TopUML Diagrams

This section gives metamodels of TopUML diagrams which are new to UML or which have been extended. The following subsections gives metamodels for Topological functioning model, Topological class diagram, and Topological use case diagram. In order to better illustrate meaning of metamodel each subsection briefly describes elements used in metamodel together with instance description of metamodel, and gives an example of metamodel instance. Mappings between elements of TopUML diagrams are given in Appendix 3 on page 186. Mappings are described for diagrams which are new to UML or extended by TopUML profile. All mappings that exist between UML diagrams remain the same. The mappings between standard UML diagrams can be found in various books and researches, like [15], [41], [112], [127], and [128]. Mappings between TopUML diagrams are described in the form of table by giving element of one TopUML diagram type and corresponding element in other kind of TopUML diagram. For better understanding a description of this mapping is given.

3.3.4.1. Metamodel of Topological Functioning Model

To better understand the metamodel of TFM at first is given formal definition of it and its elements. TFM has strong mathematical basis and is represented in a form of a topological space. The TFM has topological characteristics: connectedness, closure, neighbor-hood, and continuous mapping. Despite that any graph is included into combinatorial topology, not every graph is a topological functioning model. A directed graph becomes the TFM only when theoretical substantiation of the systems is added to the above mathematical substantiation. The latter is represented by functional characteristics: cause-effect relations, cycle structure, and inputs and outputs. [94]

TFM enables careful analysis of system's operation and communication with the environment through analysis of functional cycles. Thus it is stated that at least one directed closed loop (i.e., cycle) must be present in every topological model of system functioning. This cycle shows the "main" functionality that has a vital importance in the system's life. Usually it is even an expanded hierarchy of cycles. By interrupting this "main" cycle the

system can no longer function or it functions faulty. To better illustrate “main” cycle in graph representation of TFM, the arcs belonging to this cycle is drew with bolder lines. [100]

Formal Definition of Topological Functioning Model: TFM is represented in a form of a topological space (see equation (1)), where X is a finite set X of functional features X_{id} (see equation (4)) of the system under consideration, and Θ is the topology that satisfies axioms of topological structures and is represented in a form of a directed graph (i.e., Θ is a finite set of topological relationships T_{id} (see equation (3)) between functional features):

$$G = (X, \Theta). \quad (1)[99]$$

Topological space Z represents functioning of the system under consideration and its surrounding environment (i.e., finite set of TFMs exists in topological space where each TFM shows functioning of a specific system). The topological space Z is a system represented by equation (2), where N is a set of internal system functional features and M is a set of functional features of other systems that interact with the system or of the system itself, which affect the external ones:

$$Z = N \cup M. \quad (2)[99]$$

The necessary condition for constructing the topological space Z is a meaningful and exhaustive verbal, graphical, or mathematical system description. The adequacy of a model describing the functioning of a specific system can be achieved by analyzing mathematical and functional properties of such abstract object. [94]

To analyze and show functioning of specific system TFM of this system should be separated from topological space Z. Separation of the TFM from the topological space Z of a problem domain is performed by applying the closure operation over a set of system’s inner functional features (the set N) as it is shown by equation (3), where Xη is an adherence point of set N and capacity of X is the number n of adherence points of N:

$$X = [N] = \bigcup_{\eta=1}^{n} X_{\eta} \quad (3)[99]$$

An adherence point of the set N is a point, whose each neighborhood includes at least one point from the set N. The neighborhood of a vertex Y in a directed graph is the set of all

vertices adjacent to Y and the vertex Y itself. It is assumed here that all vertices adjacent to Y lie at the distance d=1 from Y. An example of TFM is given in Figure 3.11.

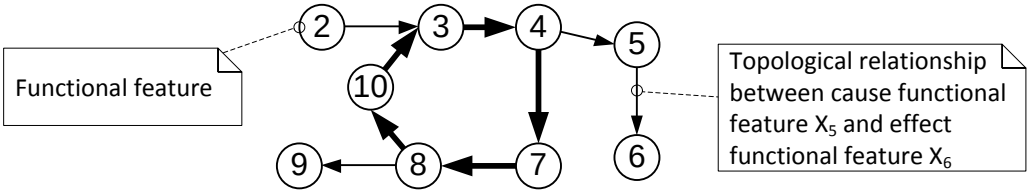


Figure 3.11. Example of Topological functioning model

Formal Definition of Functional Features: Functional feature is a description of atomic business action (i.e., it cannot be separated into a number of other business actions). In [6] it is suggested that each functional feature is a unique tuple; in [103] this tuple is extended to include class and operation reference (elements *Cl* and *Op*). This research redefines tuple to include all necessary elements needed when constructing TFM (the new elements are *Id*, *St*, *Es*, and *S*). Unique tuple definition of functional feature X_{id} is a shown by equation (4):

$$X_{id} = <Id, A, Op, R, O, Cl, St, PrCond, PostCond, E, Es, S>, \text{where} \quad (4)$$

- **Id** – identifier of functional feature,
- **A** – action of object O,
- **Op** – operation which will provide functionality defined by action A (can be acquired when the class diagram is synthesized),
- **R** – result of action A (optional),
- **O** – object that receives the result or that is used in action A (for example, a role, a time period, a catalogue, etc.),
- **Cl** – class which will represent object O in static viewpoint of system (can be acquired when the class diagram is synthesized),
- **St** – new state of object O after performing action A (optional),
- **PrCond** – is a set of preconditions C_{id} (see equation (5)) (optional),
- **PostCond** – is a set of postconditions C_{id} (see equation (5)) (optional),
- **E** – entity responsible for performing action A,
- **Es** – indicates if execution of action A could be automated (i.e., performed without human interaction), and

- **S** – subordination of functional feature (can be internal or external).

Formal Definition of Preconditions and Postconditions: Each precondition or postcondition is a condition C_{id} described by unique tuple given in equation (5). Condition can be considered as an atomic business rule.

$$C_{id} = <Id, Cond, oCond>, \text{ where} \quad (5)$$

- **Id** – identifier of condition,
- **Cond** – condition or an atomic business rule, and
- **oCond** – identifier of opposite condition, i.e., $C_i = \neg C_j$ (optional).

Formal Definition of topological relationships: Cause-and-effect relationship T_{id} is a binary relationship relating two functional features X_{id} and are represented as arcs of a directed graph that are oriented from a cause vertex to an effect vertex. The synonym for cause-and-effect relationship is topological relationship. Each cause-and-effect relationship is a unique tuple represented by equation (6):

$$T_{id} = <Id, X_c, X_e, L_{out}, L_{in}>, \text{ where} \quad (6)$$

- **Id** – unique identifier of topological relationship,
- $\mathbf{X_c}$ – cause functional feature,
- $\mathbf{X_e}$ – effect functional feature,
- $\mathbf{L_{out}}$ – set of logical relationships between topological relationships on outgoing arcs of cause functional feature X_c (optional), and
- $\mathbf{L_{in}}$ – set of logical relationships between topological relationships on incoming arcs of effect functional feature X_e (optional).

Formal Definition of Logical Relations: Logical relation L_{id} shows the logical relationship conjunction (*and*), disjunction (*or*), or exclusive or (*xor*) between two or more topological relationships T_{id}. The type of logical relation denotes system execution behavior (e.g., decision making, parallel actions). Each logical relation is a unique tuple represented by equation (7):

$$L_{id} = <Id, T, Rt>, \text{ where} \quad (7)$$

- **Id** – identifier of logical relationship,

- **T** – set of topological relationships belonging to this logical relationship, and
- **Rt** – logical relationship type (*and*, *or*, or *xor*).

Identification of logical relations L_{id} between cause-and-effect (i.e., topological) relationships T_{id} consists of two activities:

1. Identification of logical relations L_{out} between topological relationships T_{id} that are outgoing from functional feature X_{id}, and
2. Identification of logical relations L_{in} between topological relationships T_{id} that are incoming to functional feature X_{id}.

Example of logical relations between topological relationships is given in Figure 3.12.

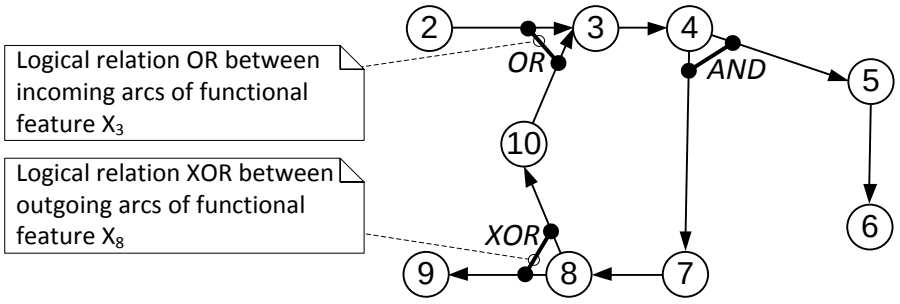


Figure 3.12. Example of logical relations between topological relationships

Definition of TFM metamodel: The metamodel of TFM represents it as an instance of the metaclass *TopologicalFunctioningModel*. In order to define metamodel of TFM the following UML version 2.4.1 metaclasses has been extended: *Model*, *Class*, *NamedElement*, *Behavior*, *Constraint*, *Relationship*, and *DirectedRelationship*.

Metamodel of TFM is given in Figure 3.13 on next page showing all stereotypes, metaclasses and enumerations involved into definition of TFM. Mappings between elements of unique tuple (see equation (4) on page 88) and attributes of stereotype *FunctionalFeature* which is defining functional features is given in Table 3.5.

Table 3.5

Mappings between tuple and stereotype elements representing functional feature X_{id}

	Tuple element	**Class attribute or property with type**	**Multiplicity**
1	Id	id:string	1
2	A	action:string	1
3	Op	operation:Operation	0..1
4	R	actionResult:ActionResult	0..1

	Tuple element	Class attribute or property with type	Multiplicity
5	O	object:string	1
6	Cl	class:Class	0..1
7	St	newState:State	0..1
8	PrCond	preCondition:Condition	0..*
9	PostCond	postCondition:Condition	0..*
10	E	entity:Actor	1
11	Es	executorIsSystem:boolean	1
12	S	subordination:Subordination	1

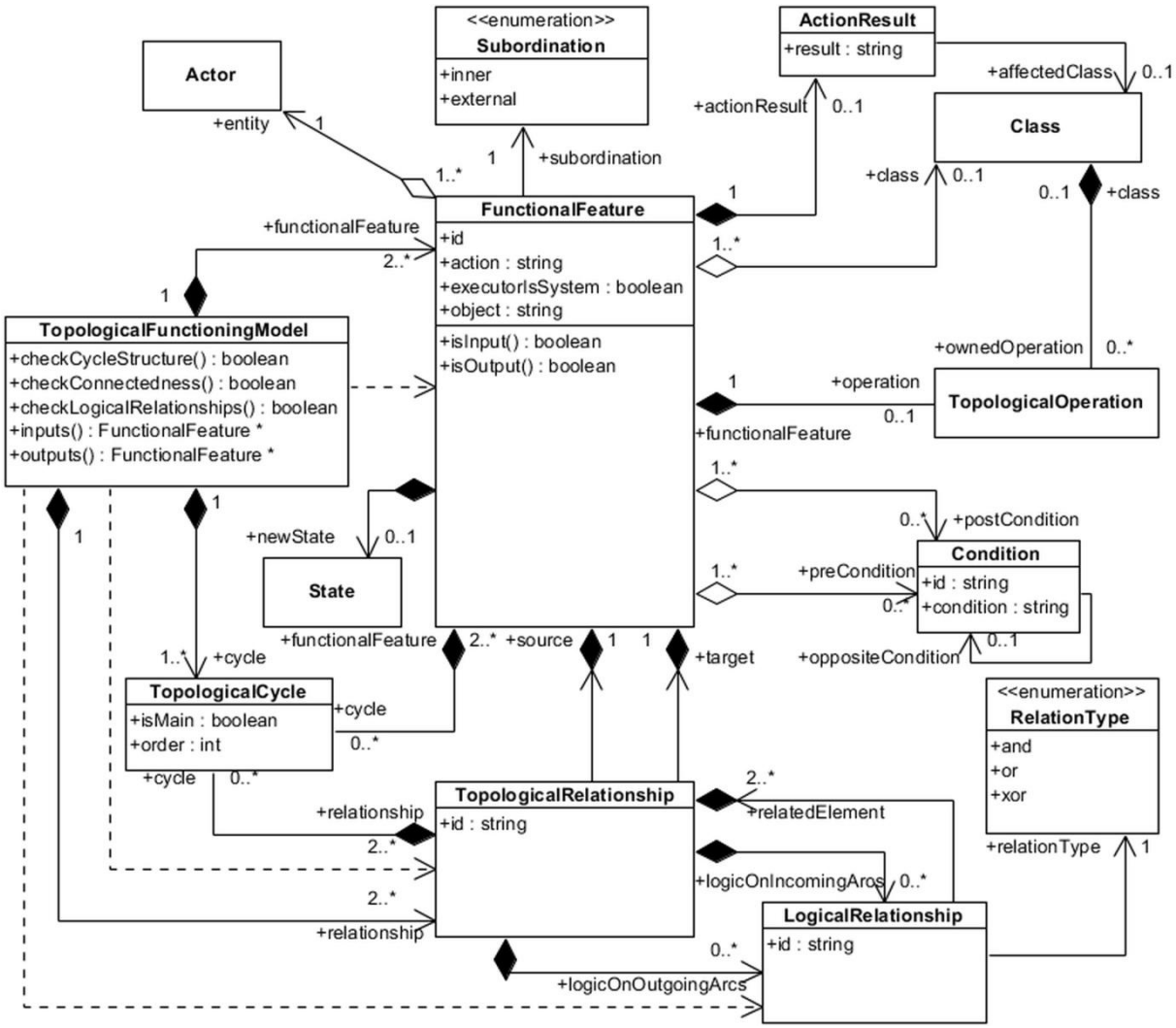


Figure 3.13. Metamodel of Topological functioning model

Mappings between elements of unique tuple (see equation (5) on page 89) and attributes or properties of stereotype *Condition* which is defining pre- and post- conditions of functional feature is given in Table 3.6.

Table 3.6

Mappings between tuple and stereotype elements of pre- and post- conditions C_{id}

	Tuple element	**Class attribute or property with type**	**Multiplicity**
1	Id	id:string	1
2	Cond	condition:string	1
3	oCond	oppositeCondition:Condition	0..1

Mappings between elements of unique tuple (see equation (6) on page 89) and attributes or properties of stereotype *TopologicalRelationship* which is defining topological relationships between functional features is given in Table 3.7.

Table 3.7

Mappings between tuple and stereotype elements of topological relationships T_{id}

	Tuple element	**Class attribute or property with type**	**Multiplicity**
1	Id	id:string	1
2	Xc	source:FunctionalFeature	1
3	Xe	target:FuntionalFeature	1
4	Lout	logicOnOutgoingArcs:LogicalRelationship	0..*
5	Lin	logicOnIncomingArcs:LogicalRelationship	0..*

Mappings between elements of unique tuple (see equation (7) on page 89) and attributes or properties of stereotype *LogicalRelationship* which is defining logical relations between topological relationships is given in Table 3.8.

Table 3.8

Mappings between tuple and stereotype elements of logical relationships L_{id}

	Tuple element	**Class attribute or property with type**	**Multiplicity**
1	Id	id:string	1
2	T	relatedElement:TopologicalRelationship	2..*
3	Rt	relationType:RelationType	1

According to metamodel of TFM in order to create TFM an instance of class *TopologicalFunctioningModel* should be instantiated. Each instance consists from at least two

functional features (instances of class *FunctionalFeature*), two cause-and-effect relationships (instances of class *TopologicalRelationship*) and at least one functioning cycle (instance of class *TopologicalCycle*).

3.3.4.2. Metamodel of Topological Class Diagram

Metamodel of Topological class diagram is given in Figure 3.14, where classes with bolder lines show elements that are added to the metamodel of regular UML class diagram.

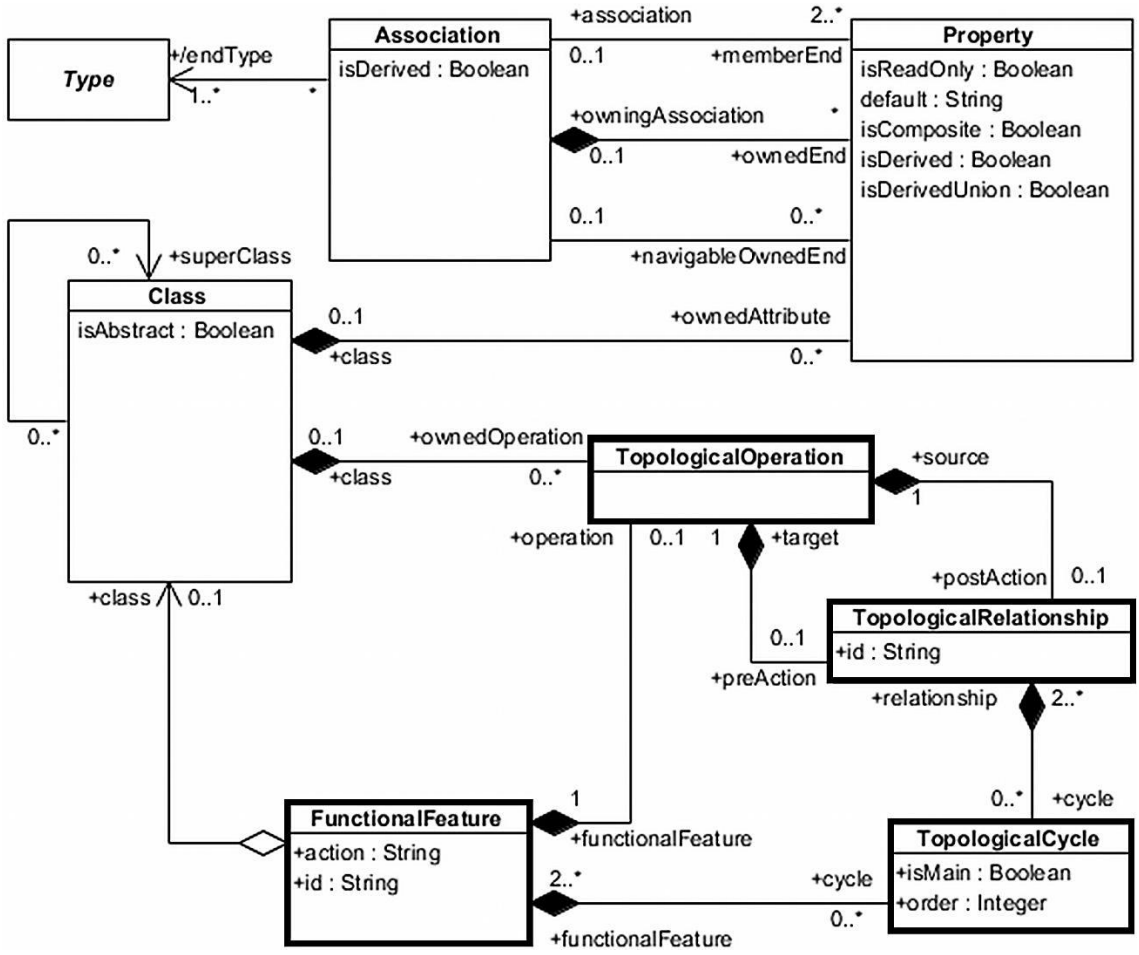


Figure 3.14. Metamodel of Topological class diagram

Topological class diagram is based on ordinary UML class diagram – it is extended to include topological relationships thus allowing to trace causality relations between problem and solution domains. Additionally Topological class diagram is able to show functioning cycles of a system (including the main functional cycle), input and output classes (e.g., classes which are communicating with the external environment). The presence of functional cycles

allows classifying classes thus the classes participating in functional cycles can be marked and highlighted in the solution.

To achieve above described goal a class *TopologicalOperation* is used instead of class *Operation* thus allowing to relate two operations with topological relationship (similarly as Association relationship relates at least two properties). Each operation specifies a cause and an effect of it (the cause is specified by using attribute *preAction* and the effect is specified by using attribute *postAction*). Attribute *preAction* leads to related topological relationship and the attribute target of this relationship leads to pre-operation (the cause), while attribute *postAction* leads to related topological relationship and the attribute source of this relationship leads to post-operation (the effect). The topological relationship defines the causality within Topological class diagram while association defines the structure of objects (more precisely their specification – classes).

Each topological operation is related with functional feature that specifies it. This relation allows establishing direct traceability between Topological functioning model and Topological class diagram. Besides that relations in Topological class diagram are perceived formally from problem domain.

3.3.4.3. Metamodel of Topological Use Case Diagram

Topological use case diagram extends ordinary UML Use case diagram by adding topological relationship between actor and use case (i.e., between classes *Actor* and *UseCase*) thus allowing to define actor for use case diagram directly from TFM by automatically relating functional features with corresponding use case. Since both classes *Actor* and *UseCase* are generalizations of class *BehavioredClassifier*, the source and target of topological relationships (i.e., both ends of a binary relationship) is connected to the *BehavioredClassifier*.

The metamodel of the Topological use case diagram is represented in Figure 3.15, where classes with bolder lines show elements that are added to the metamodel of regular UML Use case diagram.

Topological use case diagram allows development of use cases based on the functional features and functionality they describe – functional features are mapped onto use cases. The input and output functional features define communication with the external environment. When functional features have been mapped onto use cases, the identification of actors is performed. The actors in Topological use case diagram are entities from functional features.

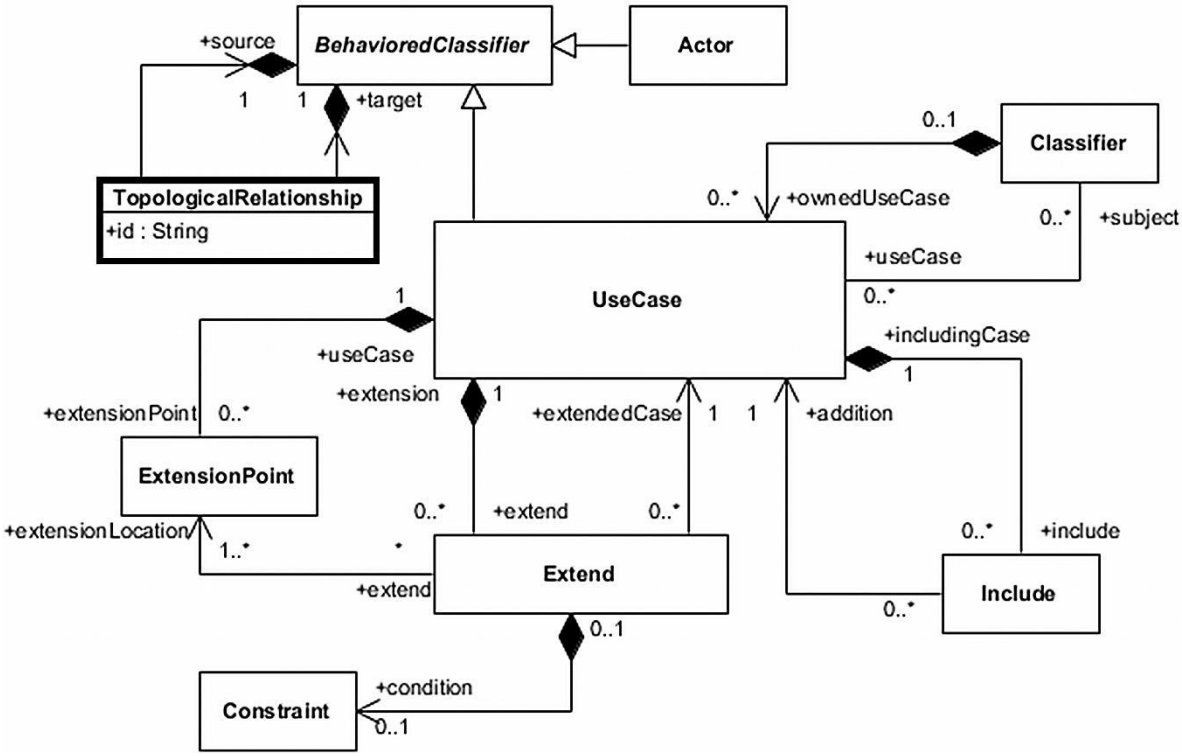


Figure 3.15. Metamodel of Topological use case diagram

Additionally *«include»* and *«extend»* relationships between use cases can be added automatically by analyzing topological and logical relationships between the mapped functional features. The scope of subsystems is defined formally by analyzing the functional cycles within TFM. If more than one functioning cycle is found – subsystems are designed in a way that each subsystem includes functionality of one functional cycle. Thus the Topological use case diagram is developed in strong accordance with the functioning of the problem and solution domains.

3.4. Summary

When extending UML it is needed to draw the scope of required extension: if the new language will use most of the UML, then profiles are more suitable; and if the new language uses only small part of UML or there is need to use more complex features of UML such as redefinition, then creating a complete new language by using MOF metamodeling should be considered. In fact, UML specification itself is defined using metamodeling approach (a metamodel is used to specify the model that builds UML). One of the UML metamodel principles is its extensibility. The most common and suitable way for improving UML is to

use its extensibility mechanisms – the profiles (in this scenario it is possible to adapt and use ordinary UML compliant modeling tools). Thus, by creating a profile of UML, the costs of adapting it in practice is lowered and it can be adapted faster (in comparison with creating a new modeling language which leads to implementing new modeling tools).

Since the UML specification is a specification of a notation, it does not include any guidelines for profile definition and specification, thus leading to current situation when UML profiles are developed in inconsistent ways. This makes it hard to read and understand profiles proposed and created by different authors. To overcome this issue a set of UML profiles are analyzed resulting in profile specification template – the best practice for UML profile specification is to use the same structure as used for UML specification. Thus, if the reader is familiar with UML specification it is easier to read and understand the specification of specific UML profile.

Development of UML profile provides a way to solve one of its disadvantages – lacking causality. Addition of mathematical topology to the UML through TopUML profile ensures that such profile has elements which are allowing to clearly trace causal relations in both – problem and solution – domains. TopUML is a combination of UML and formalism of TFM. The main aim of improving UML is by transferring topology and mathematical formalism of TFM to UML. At first an analysis of the existing UML diagrams is performed, which shows that extension should be provided for two UML diagrams: Class diagram and Use case diagram. The definition of TopUML profile follows the identified specification template. In addition TopUML profile is supplemented with mappings between its diagrams and diagram elements thus showing transformation patterns between different diagram types.

While this chapter specifies TopUML profile thus giving tools to clearly trace cause-and-effect relations in problem and solution domains, the next chapter defines a modeling method for applying TopUML profile in practice. Together the TopUML profile and modeling method solves issues and disadvantages related with UML and its application in software development process.

4. TOPUML MODELING – A METHOD FOR DESIGNING SOFTWARE

TopUML modeling for problem domain modeling and software systems designing is a model-driven modeling method. It combines TFM and its formalism with elements and diagrams of TopUML. The TFM considers problem domain information separate from the solution domain information and holistically represents a complete functionality of the system from the computation independent viewpoint while TopUML has elements for representing system design at the platform independent viewpoint and platform specific viewpoint (in the context of MDA). The TopUML modeling method covers modeling and specification of systems in computation independent and platform independent viewpoints.

The application of TopUML modeling ensures proper analysis of system functioning by identifying and analyzing functioning cycles. The functioning cycle is a common thing of all system (technical, business, or biological) functioning. Therefore, it is stated that at least one directed closed loop must be present in every TFM of system functioning [99]. It shows the main functionality that has a vital importance in the system's life (i.e., by destroying the main functioning cycle the system can no longer function or it is seriously malfunctioning). Usually the main functionality is even an expanded hierarchy of cycles [45]. Therefore, a proper cycle analysis is necessary in the very beginning of software development lifecycle, because it enables careful analysis of system's operation and communication with the environment [101]. By using TopUML the information of system functioning from TFM is transferred to design models and diagrams thus allowing marking and evaluating the most important objects and components within system and to assign proper responsibilities[4] to the objects in a formal way.

Suggested transitions between TopUML diagrams are given in Figure 4.1, where the diagrams are shown as object nodes and the edges between them as object flow within Activity diagram. TopUML modeling does not include development of Profile, Timing and Composite structure diagrams as the TFM shows timing within functioning of a system, Component diagram specifies structure of components. Profile diagram is not addresses while it is intended to specify a new profile of UML (not a software design). It is possible to automate transitions between TopUML diagrams while the validation of the acquired diagrams is needed by the domain experts. The development of TFM can be partly automated as shown in [108] where the business use cases are transformed into TFM.

[4] Operations of a class are addressed as responsibilities of this class. Responsibilities answer to two questions: "*What to do?*" and "*How to do?*"; and they are assigned to classes of objects during the object design. [64]

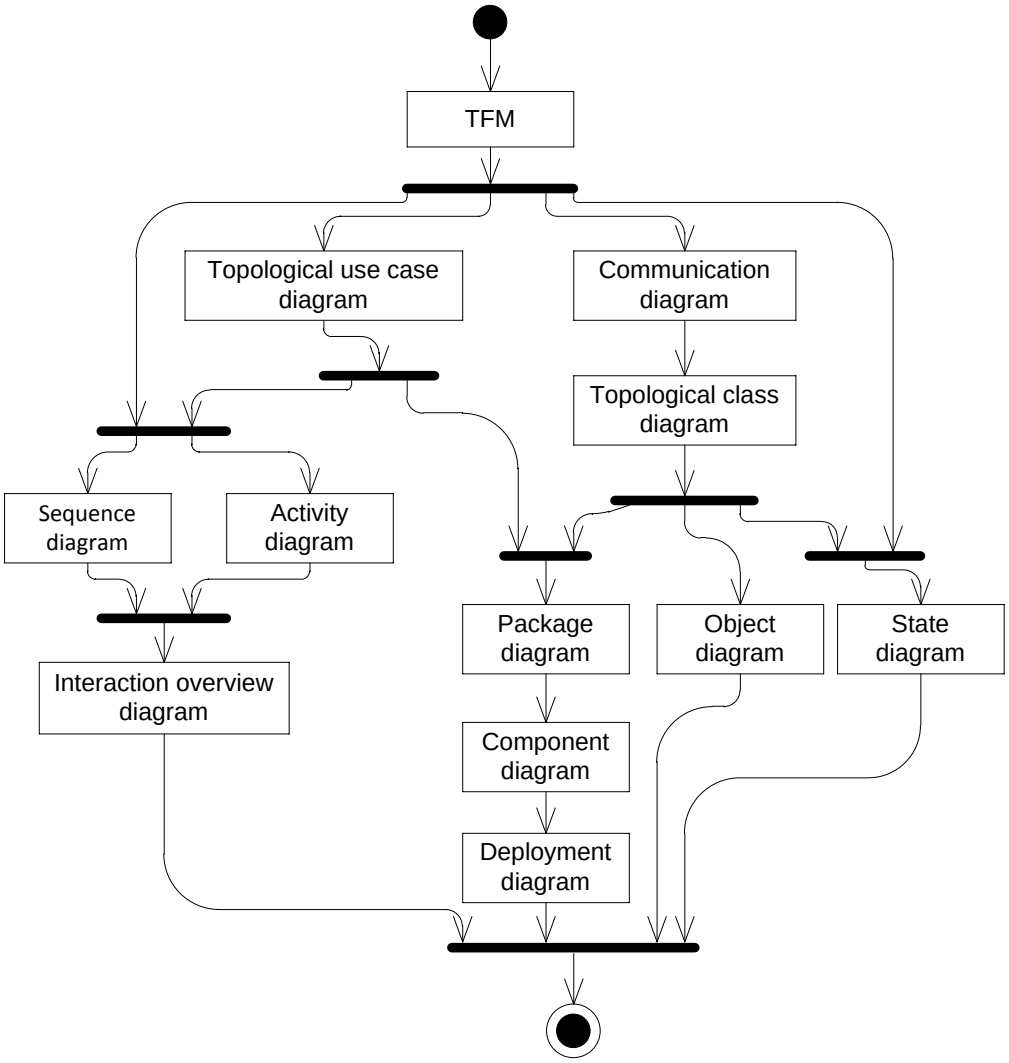


Figure 4.1. Transitions between TopUML diagrams

The TopUML diagrams that are used within TopUML modeling are listed in Table 4.1, where a development order of the diagram is given as well as the diagrams to which it can be transformed or has information for development.

Table 4.1

Diagrams used within TopUML modeling method

No.	Diagram type	Order	Information for	Notes
1.	Topological Functioning Model	1	Topological use case diagram, Sequence diagram, Activity diagram,	Initial TFM is developed by analyzing functional characteristics of the problem domain. The refinement of TFM includes adjusting TFM to the functional requirements of the desired software system since the

No.	Diagram type	Order	Information for	Notes
			Communication diagram, State diagram	requirements can introduce new functionality to the problem domain. By refining TFM the functional requirements are validated, i.e., the TFM shows missing, overlapping, conflicting, and incorrect requirements [5].
2.	Topological use case diagram	2	Sequence diagram, Activity diagram, Package diagram	The scope of use cases is set either by functional requirements or by system goals. The functionality represented by each use case is obtained from the TFM according to the mappings between functional features and functional requirements.
3.	Communica-tion diagram	2	Topological class diagram	Communication diagram is used as an intermediate model between TFM and Topological class diagram. It is developed by transforming TFM – the functional features representing the same object type are merged and the cause-and-effect relations become links between lifelines.
4.	Sequence diagram	3	Interaction overview diagram	Sequence diagram shows the messaging between actors and objects. Usually a set of Sequence diagrams is created – one for each use case. Use case is used to set the scope of Sequence diagram while TFM is used to set the messages and their order.
5.	Activity diagram	3	Interaction overview diagram	Activity diagram shows the workflow of a use case. Usually a set of Activity diagrams is created – one for each use case. Use case is used to set the scope of Activity diagram while TFM is used to set the action nodes and edges.
6.	Topological class diagram	3	Package diagram, State diagram, Object diagram	Topological class diagram is used to represent a domain model and a system design model. The key idea behind domain model is a visual dictionary of abstractions. The topological relations between classes show the causal relations between entities in the problem and solution domains.
7.	Interaction overview diagram	4	-	Defines interactions through a variant of Activity diagram in a way that promotes overview of the control flow. Interaction overview diagram focus on the overview of the flow of control.
8.	Object diagram	4	-	Object diagram can be developed during the refinement process of Topological class diagram when the associations are analyzed. It is useful in situation when object of one type plays more than one role at a time.

No.	Diagram type	Order	Information for	Notes
				Object diagram also can be used to provide examples of system at a specific time
9.	State diagram	5	-	State diagrams are used to show the state transitions of objects; one diagram is created for each object type.
10.	Package diagram	6	Component diagram	Package diagram is used to organize and group classes into logical structure – packages. Each package represents a subsystem and groups a set of cohesive responsibilities of classes.
11.	Component diagram	7	Deployment diagram	Component diagram represents modular, deployable, and replaceable parts of a system; one component is created for each package.
12.	Deployment diagram	8	-	Component diagram shows how instances of components are deployed on instances of nodes. The content of Deployment diagram is denoted by components and nonfunctional requirements.

Problem domain analysis and software system design with TopUML modeling method consists of following six activities as given below in Activity diagram in Figure 4.2:

1. Problem domain functioning analysis,
2. Behavior analysis and design,
3. Structure analysis and design,
4. State change and transition analysis,
5. Structuring logical layout of design, and
6. Components and deployment design.

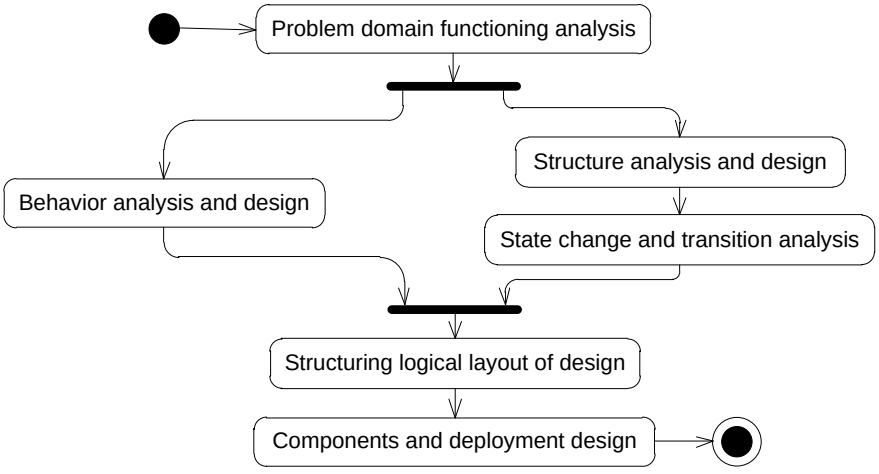


Figure 4.2. TopUML modeling activities

The activities of TopUML modeling within the software development project can be applied in any order and only part of the activities can be used. There is one restriction – inputs of each activity should be provided. TopUML modeling method is guidelines of TopUML profile application in software development; it does not restrict the use and application of TopUML diagrams. Each modeling activity in detail is described in the following subsections, where the inputs, outputs and included subactivities and actions of the main activity are given. TopUML modeling process in detail is described with a set of activity diagrams followed by description in a natural language.

4.1. Problem Domain Functioning Analysis

TFM development consists of three activities (see Figure 4.3, where "*Mappings**" denotes the mappings between functional features and functional requirements):

1. Topological space development,
2. Initial TFM development, and
3. TFM refinement.

The inputs of problem domain functioning analysis activity are functioning description and functional requirements. Functioning description can be in any form; it needs to cover full description of problem domain functioning.

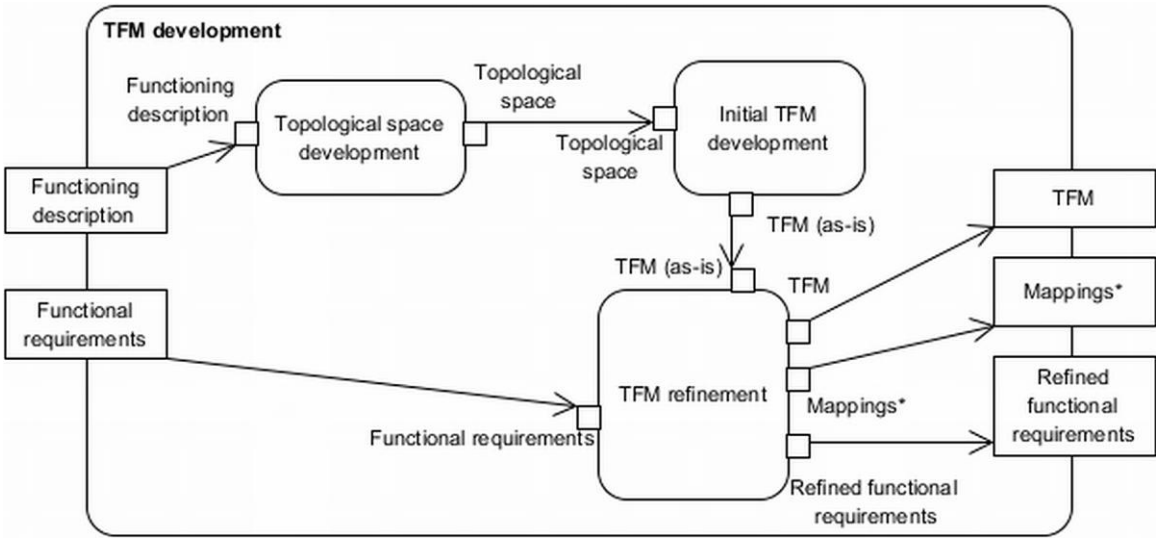


Figure 4.3. Topological Functioning Model development activities

Output of this activity is TFM (both the one representing functioning of problem domain and the one representing functionality of desired software system), mappings between functional features and functional requirements, and refined functional features.

The TFM development activities in detail are described in the subsequent three subsections (each subsection describes one activity).

4.1.1. Topological Space Development

Topological space development consists of two actions (see Figure 4.4):

1. Definition of physical or business functional characteristics, and
2. Introduction of topology Θ.

The **input** of this activity is functioning description of the system. Functioning description can be in different formats, e.g., informal structured textual description [99], business use cases (discussed in [112]).

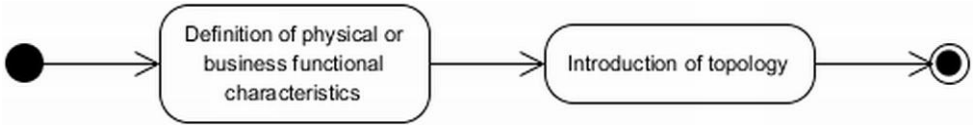


Figure 4.4. Development of topological space

Definition of physical or business functional characteristics consists of the following activities [5]:

1. Definition of objects and their properties from the problem domain description;
2. Identification of external systems and partially-dependent systems; and
3. Definition of functional features using verb analysis in the problem domain description, i.e., by finding meaningful verbs.

As a result a set of functional features X_{id} are defined. At the lowest abstraction level one functional feature should describe only one atomic business action (it cannot be further divided into a set of business actions) [27]. By using the topological characteristic *continuous mapping* of TFM, the abstraction level of functional features can be changed at any time when needed (e.g., the initial TFM can be detailed out by lowering the abstraction level of functional features thus showing more or less detailed description of problem domain functioning). Each functional feature is a unique tuple as shown by equation (4) on page 88.

Introduction of topology $\mathcal{O}$ (in other words – creation of topological space) means establishing cause-and-effect (topological) relations T_{id} between identified functional features. Topological relations are represented as arcs of a directed graph that are oriented from a cause vertex to an effect vertex, where each vertex is a functional feature. Topological space Z is a system represented by equation (2) on page 87, where N is a set of inner system functional features and M is a set of external functional features (i.e., the external functional features show links and communication with the external environment). Topological space represents the system under consideration together with the environment in which this system exists.

Output of this activity is a set of functional features and a set of topological (cause-and-effect) relationships between them. Together they make topological space of the system under consideration.

4.1.2. Initial Topological Functioning Model Development

Initial TFM development involves completion of two actions (see Figure 4.5).

Figure 4.5. Development of initial TFM

The **input** of this activity is topological space.

Separation of TFM from topological space is done by applying the closure operation as shown by equation (3) on page 87 over a set of system's inner functional features (the set N), where Xη is an adherence point of the set N and capacity of X is the number n of adherence points of N. Initial TFM can be called "*TFM as-is*" where "*as-is*" means that the TFM represents the functioning of the problem domain without the impact of planned software system. Construction of initial TFM can be iterative. Iterations are needed if the information collected for TFM development is incomplete or inconsistent or there have been introduced changes in system functioning or in software requirements.

Identification of logical relations between topological relationships consists of two steps since there are two kinds of logical relationships L_{id} – one kind is between arcs that are outgoing from functional features X_{id} and the other kind is between arcs that are incoming to functional features X_{id}. The logical relationships between outgoing arcs are denoted with L_{out}

and the logical relationships between incoming arcs – L_{in}, thus the identification of logical relations between cause-and-effect (i.e., topological) relationships consists of two subactions:

1. Identification of logical relations L_{out} between topological relationships T_{id} that are outgoing from functional feature X_{id} (see section 4.1.2.1), and
2. Identification of logical relations L_{in} between topological relationships T_{id} that are incoming to functional feature X_{id} (see section 4.1.2.2).

Within TFM can be defined three types of logical relations L_{id}: conjunction (*and*), disjunction (*or*), and exclusive disjunction (*xor*). Within each logical relation L_{id} participate two or more topological relationships T_{id}. The following two subsections cover the identification of logical relations L_{out} and L_{in}.

Output of this activity is initial TFM representing the functioning of the business system.

4.1.2.1. Identification of Logical Relations between Outgoing Topological Relationships

Logical relations L_{out} between topological relationships that are outgoing from functional feature indicate necessity of decision making or branching. In the case of making decision only part of effect functional features is executed, but in the case of branching all of the effect functional features are executed (i.e., system performs parallel processing). [25]

Depending on the relationship type *Rt* of logical relation L_{out}, system execution behavior is defined as follows:

- *Conjunction* (*AND*) – system is running in parallel by executing all effect functional features of topological relationships participating in this logical relation,
- *Disjunction* (*OR*) – system can be running in parallel by executing one, part of or all effect functional features of topological relationships participating in this logical relation, and
- *Exclusive disjunction* (*XOR*) – only one effect functional feature of topological relationships participating in this logical relation is executed.

The algorithm of identifying logical relations L_{out} between outgoing arcs of functional features is given in Figure 4.6 while the rules for identification of logical relation L_{out} type and patterns for transforming TFM into Activity diagram are given in Table 4.2, where Rt denotes relation type, X_e – effect functional features, and C_{id} – preconditions of X_e.

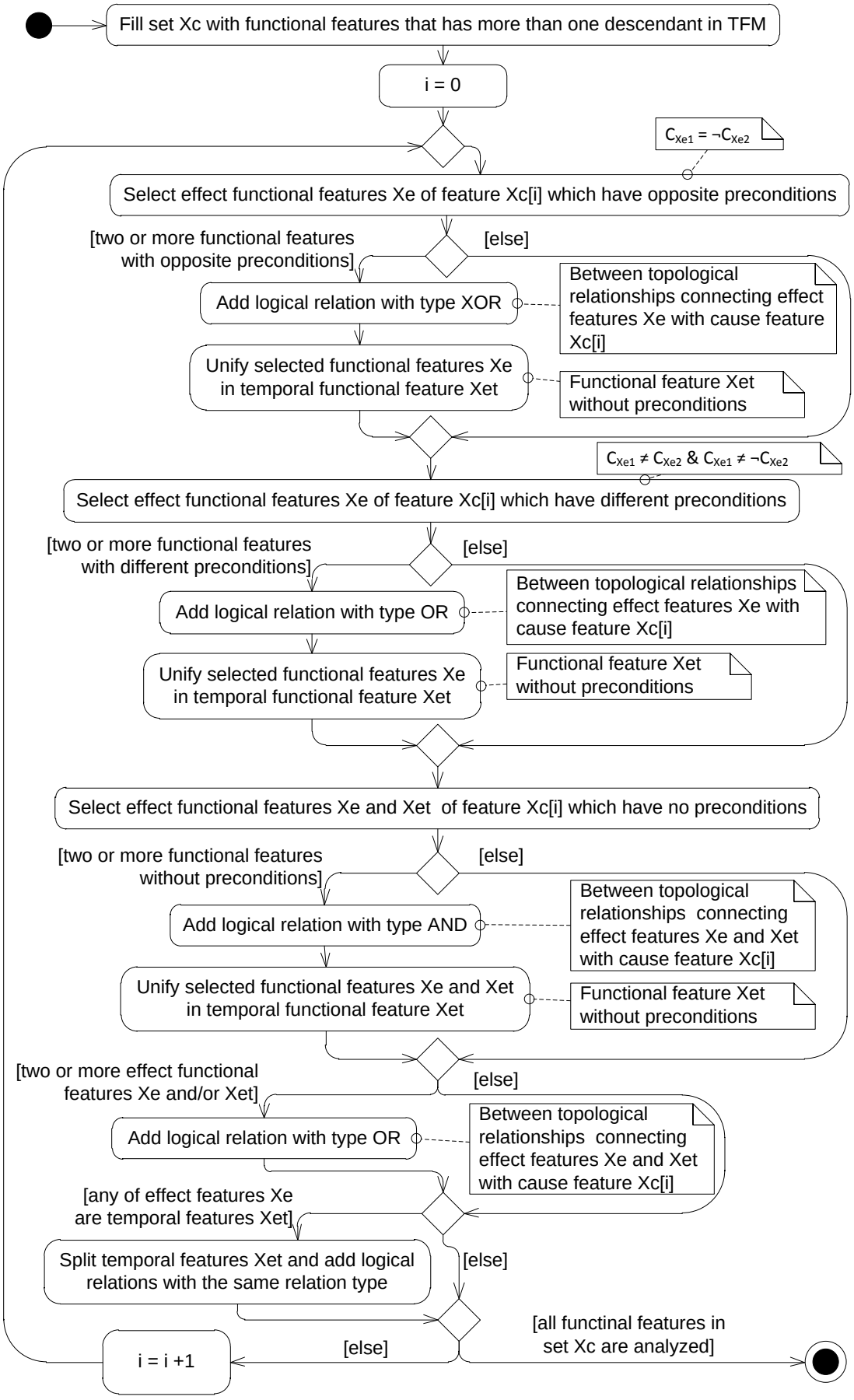
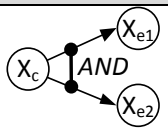
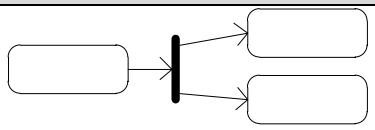
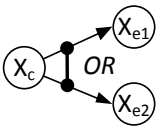
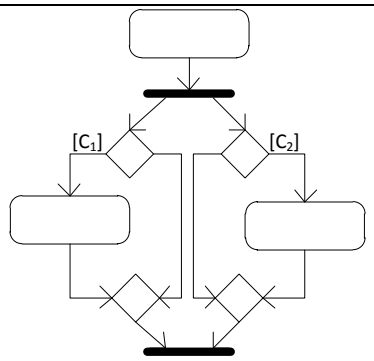
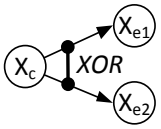
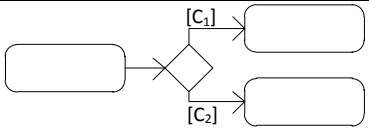


Figure 4.6. Identification of logical relations L_{out} between outgoing topological relationships

Table 4.2

Rules for identification of logical relations L_{out} and patterns for transforming TFM into Activity diagram

Rt	X_e	C_{id}		Example of L_{out}	Pattern of Activity diagram
AND	X_{e1}	Ø			
	X_{e2}	Ø			
OR	X_{e1}	C_1	$C_1 \neq C_2$ & $C_1 \neq \neg C_2$		
	X_{e2}	C_2			
XOR	X_{e1}	C_1	$C_2 = \neg C_1$		
	X_{e2}	C_2			

The analysis of logical relations L_{out} is critical when transforming TFM into other diagram types, while these relations contain information about decision making, parallel execution, and branching. Thus, by using TFM and logical relations L_{id} it is possible to build advanced diagrams of other type (e.g., Activity and Topological use case diagrams). This is in opposition to the opinion in [7] that TFM contains information sufficient to create only basic Activity diagrams (i.e., without forking, joining, and decisions).

4.1.2.2. Identification of Logical Relations between Incoming Topological Relationships

Logical relations L_{in} between topological relationships that are incoming to functional feature indicate that there is decision or branching made before the effect functional feature. If there was branching before the effect functional feature, then before executing this functional feature there should be joining and system can continue its execution only after all arcs are joined [25]. This reflects the mathematical foundations of Petri nets [21]. Depending on the relation type of logical relation L_{in}, system execution behavior is defined as follows:

- *Conjunction* (*AND*) – system is executing in parallel thus effect functional feature X_e can be executed only when all direct predecessor functional features (i.e., all

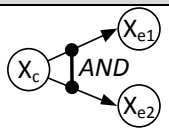
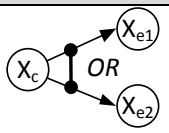
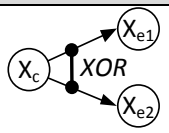
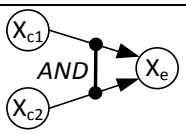
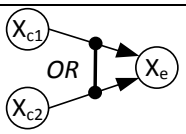
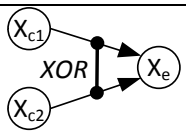
cause functional features X_c in the distance d=1) of topological relationships T_i participating in logical relation L_{id} are executed,

- *Disjunction* (*OR*) – system can be executing in parallel by executing one, part of or all cause functional features X_c of effect functional feature X_e at the distance d=1 of topological relationships T_i participating in this logical relation, and
- *Exclusive disjunction* (*XOR*) – only one cause functional feature X_c of effect functional feature X_e at the distance d=1 of topological relationships T_{id} participating in this logical relation L_{id} is executed.

Relation type of logical relations L_{in} is denoted by corresponding logical relation L_{out} (for relationships which are branched within TFM) and by the pre- and post-conditions (for the relationships that come from the inputs of TFM). The rules for identification of logical relations L_{in} between incoming arcs of functional features are given in Table 4.3, where Rt denotes relation type, L_{out} – logical relations between outgoing arcs, and L_{in} – logical relations between incoming arcs.

Table 4.3

Rules for identification of logical relations L_{in} between outgoing arcs

Rt	**AND**	**OR**	**XOR**
Source **L_{out}**			
Corresponding **L_{in}**			

The algorithm for identification of logical relations L_{in} between incoming arcs of functional features is given in Figure 4.7. To find required functional features within TFM, graph traversing algorithms are used (e.g., backtracking algorithm [71]). The identification process of logical relations L_{in} is different from the identification process of logical relations L_{out}. The main difference is in the fact that logical relations L_{in} are added according to the existing logical relations L_{out} and pre- and post- conditions of functional features, while logical relations L_{out} are added by taking into account only pre- conditions of functional features.

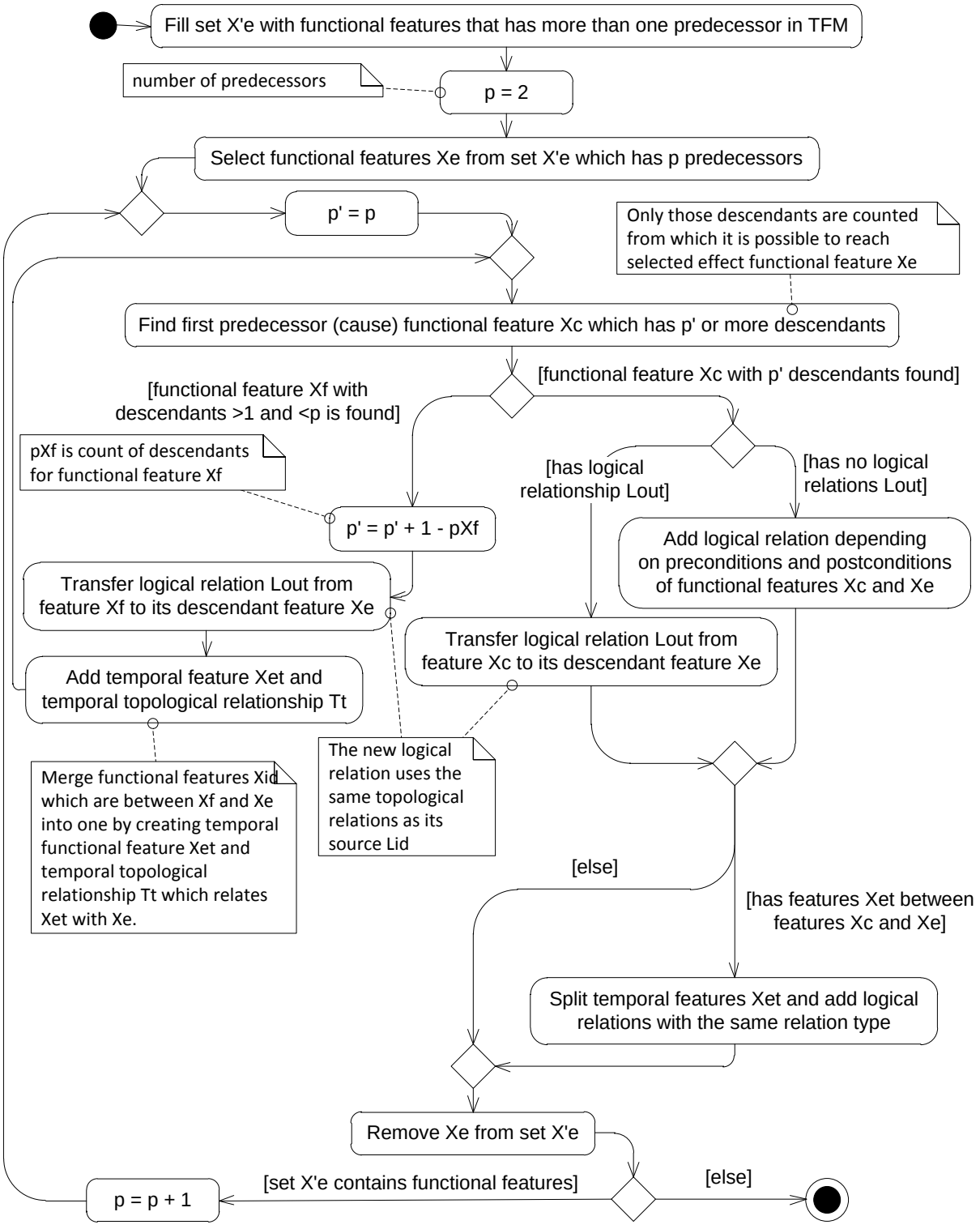


Figure 4.7. Identification of logical relations between incoming topological relationships

Logical relations L_{in} contain important information when transforming TFM into other diagram types, e.g., Activity diagram. Depending on the type of logical relation L_{in} in the Activity diagram join (*Rt = AND*) and decision making should be created (*Rt = OR* or *XOR*).

4.1.3. Refining Topological Functioning Model

TFM refinement process consists of following steps (see Figure 4.8):

1. Mapping functional features on functional requirements,
2. Checking for missing and incomplete functional requirements,
3. Checking for missing functional features, and
4. Searching for overlapping functional requirements.

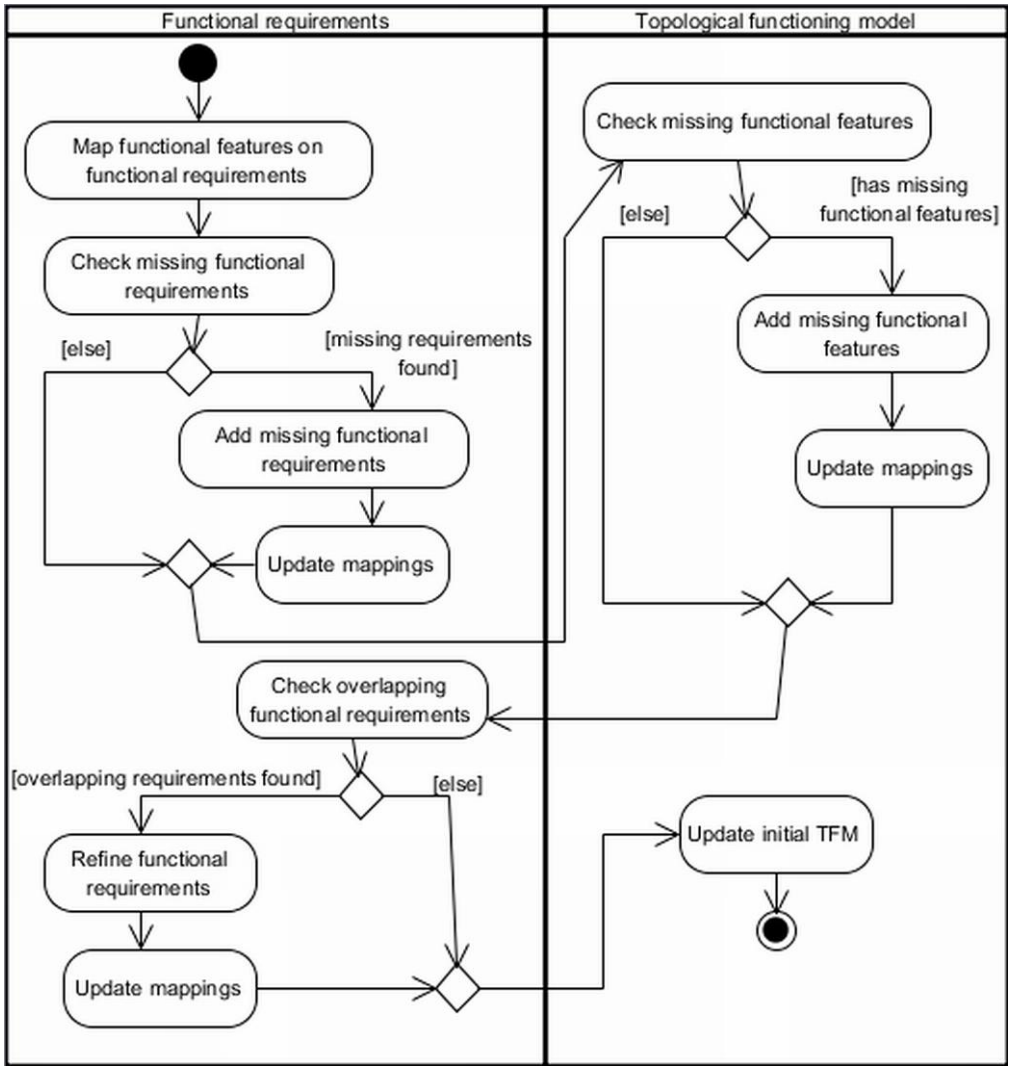


Figure 4.8. TFM refinement process

The **input** of this activity is initial TFM and functional requirements.

By **mapping functional features on functional requirements** the latter are validated in conformance with the constructed TFM. Functional features specify functionality that exists in the problem domain, but requirements specify functionality that should exist in the

solution domain. Therefore it is possible to make mappings between requirements and functional features of the TFM. As a result of requirements validation, both TFM and requirements are checked. In [100] it is suggested to represent requirement mappings between functional requirements and functional features by using arrow predicates. An arrow predicate is a construct in universal categorical logic. Universal categorical (arrow diagram) logic for computer science is explored in detail in [23].

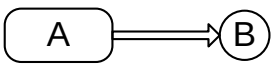
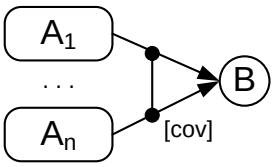
There are five types of mappings and corresponding arrow predicates defined for mapping requirements onto TFM [5]:

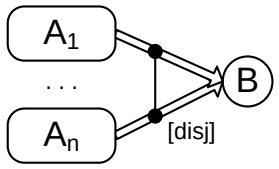
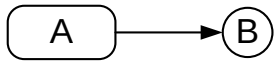
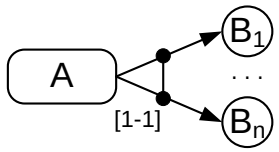
1. *One to One*: one requirement completely specifies what will be implemented in accordance with one functional feature,
2. *Many to One*: set of requirements overlap the specification of what will be implemented in accordance with one functional feature,
3. *One to Many*: one requirement incompletely specifies some functional feature, or one requirement completely specifies several functional features. It can be so because of one of the following reasons:
    a. the requirement joins several requirements and can be split up, or
    b. functional features are more detailed than the functional requirements,
4. *One to Zero*: One requirement specifies some new or undefined functionality. In this particular case it is necessary to define possible changes in the problem domain functioning, and
5. *Zero to One*: Specification does not contain any requirement corresponding to the defined functional feature. This means that it could be a missing requirement.

Graphical representations of arrow predicates are given below in Table 4.4.

Table 4.4

Arrow predicates used to map functional features with functional requirements [100]

No.	Type	Description	Graphical representation
1.	One to One	Inclusion predicate is used if the functional requirement A completely specifies what will be implemented in accordance with the functional feature B	
2.	Many to One	Covering predicate is used if functional requirements A_1, A_2, …, A_n overlap the specification of what will be implemented in accordance with the functional feature B	

No.	Type	Description	Graphical representation
		Disjoint (component) predicate is used if functional requirements A_1, A_2, …, A_n together completely specify the functional feature B and do not overlap each other	
3.	One to Many	Projection is used if some part of the functional requirement A incompletely specifies some functional feature B	
		Separating family of functions is used if one functional requirement A completely specifies several functional features B_1,…,B_n	

The mappings between functional features and functional requirements allow to:

- **Check for missing requirements** – presence of *One to many* (with *Projection* predicate) or *Zero to one* mapping type indicates that requirements do not cover the full functionality of the problem domain. Missing functional requirements should be added or existing ones extended and mappings should be updated in order to cover main functional cycle of the problem domain.
- **Check for missing functional features** – if at least one mapping with type *One to zero* exists, it indicates that the functional requirements introduce new functionality to the problem domain. Missing functional features and cause-and-effect relationships should be defined and mappings should be updated.
- **Identify overlapping functional requirements** – presence of *Many to one* (with *Covering* predicate) mapping type or if there existence of more than one functional requirement with more than one mapping type associated with some functional feature indicates that a number of functional requirements define functionality that will be implemented by the same functional feature.

Output of this activity is refined TFM representing the functionality of the desired software system (refined TFM can be addressed as "*TFM to-be*"), mappings between functional features and functional requirements, and refined functional requirements.

4.2. Behavior Analysis and Design

System behavior analysis and design consists of following four activities (see Figure 4.9, where "*Mappings**" denotes the mappings between functional features and functional requirements):

1. Use case analysis,
2. Messages and their sequence analysis,
3. Workflows analysis, and
4. Workflows and messaging analysis.

The input of this activity is refined functional requirements, refined TFM and mappings between functional features and functional requirements. Output of this activity is subsystems, use cases and actors (Topological use case diagram), messages and their sequence (Sequence diagram), and workflows (Activity and Interaction overview diagram). The system behavior analysis and design activities in detail are described in the subsequent four subsections (each subsection describes one activity).

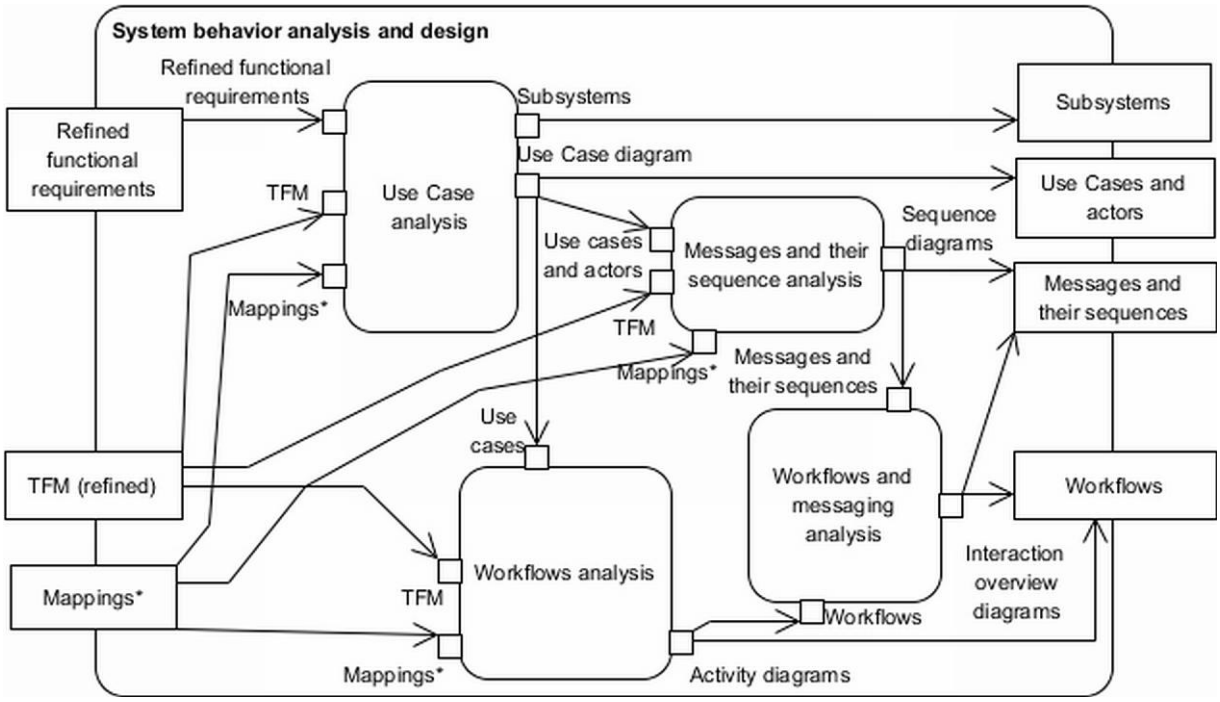


Figure 4.9. Process of system behavior analysis and design

4.2.1. Use Case Analysis and Design

Use case analysis and design consists of following four actions (see Figure 4.10):

1. Identification of use cases and actors,
2. Mapping functional features onto use cases,
3. Identification of use cases and actors, and
4. Establishing relationships between use cases.

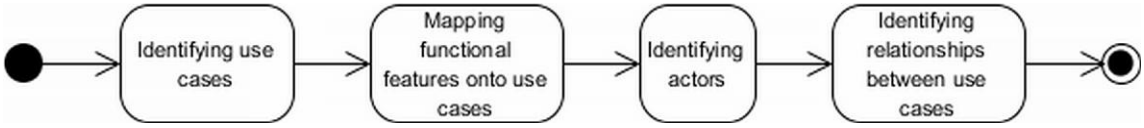


Figure 4.10. Use case development process

The **input** of this activity is refined functional requirements, TFM (refined) and mappings between functional features and functional requirements.

Identification of use cases is performed by creating one use case for each requirement. The alternative is to use system goals which are identified by the problem domain experts. If system goals need to be identified during problem domain analysis a TFMfMDA approach can be used [100]. TFMfMDA uses goals [66] in order to identify use cases and concepts from the description of the system (in the form of informal description, expert interviewing, etc.). The next step is to **map functional features onto use cases** according to mappings between functional features and functional requirements by expert. The expert finds an input and an output functional feature for each use case – all functional features that correspond to particular use case should be in one chain. When functional features have been mapped onto use cases, the **identification of actors** is performed. Since actors in Use case diagram show interaction between system and external systems or entities [89], they are obtained from topological space. The actors are entities from functional features and the set of actors are identified by Equation (8), where E is a set of functional features defining external entities, X is a set of functional features belonging to TFM, and M is a set of functional features of other systems. [29]

$$E = X \setminus M \tag{8}$$

The cause and effect relation between one functional feature belonging to set E and the other to set X defines topological relationship between use case and actor, since all use cases are mapped to functional features.

The final action is to **establish relationships between use cases** according to mappings between functional features and them. According to these mappings *«include»* and *«extend»* relationships can be automatically established between use cases by analyzing logical relations L_{out} between topological relationships. Logical relations should be analyzed for the first predecessor functional feature (which has two or more descendants, but within the scope of predecessor use case) of the use case's input functional features:

- The *«include»* relationship is added in two cases:
    1. No branching for the predecessor functional features, and
    2. Logical relation L_{out} with type AND.
- The *«extend»* relationship is denoted by presence of logical relation L_{out} with type OR and XOR on the topological relationships outgoing from functional feature.

The example of adding relationships between actors and use cases and between use cases themselves is illustrated in section 5.2 on page 151 which shows the TopUML approbation in the context of enterprise data synchronization system development.

The designed Use case diagram is supplemented with the information of subsystems. The scope of subsystems is determined by analyzing functional cycles within TFM – subsystems are extracted from TFM by applying closuring operation on the set of functional features belonging to a particular functioning cycle.

Output of this activity is use case diagram showing use cases, actors, relationships between them, and subsystems. In the context of B.O.O.M. [112] this Use case diagram is addressed as system use cases.

4.2.2. Messages of Objects and their Sequence Analysis and Design

Analysis and design of messaging between objects consists of three actions (see Figure 4.11) in which one Sequence diagram is developed for each use case:

1. Setting scope of sequence diagram,
2. Adding actors, and
3. Establishing messages between objects and their sequence.

Figure 4.11. Acquisition of Sequence diagram

The **input** of this activity is use cases, TFM, and mappings between functional features and functional requirements.

Scope of each Sequence diagram is set by the scope of corresponding use case (i.e., the Sequence diagram contains the description of the same functionality that is included into corresponding use case). Actors are added to Sequence diagrams directly from the corresponding use case. The TFM and mappings allows establishing objects with lifelines (merged functional features), messages they send each other (cause-and-effect relationships). If one use case has included another use case (e.g., A includes B), then the sequence diagram for use case A should include interaction use (*ref*) for the sequence diagram of use case B.

Messages between objects and their sequence are established by transforming part of TFM according to the scope of the corresponding use case. During TFM transformation all vertices with the same type of objects should be merged. While merging these vertices all topological relations between them are kept. The cause-and-effect relations from TFM serve as message sending between objects. The interaction operators are added by taking into consideration logical relations L_{out}:

- ***alt*** – added for logical relations L_{out} with type OR and XOR,
- ***opt*** – added for logical relations L_{out} with type OR, and
- ***par*** – added for logical relations L_{out} with type AND.

In the case of adding interaction operators *alt* and *opt* their guards are set as the preconditions of the corresponding effect functional feature.

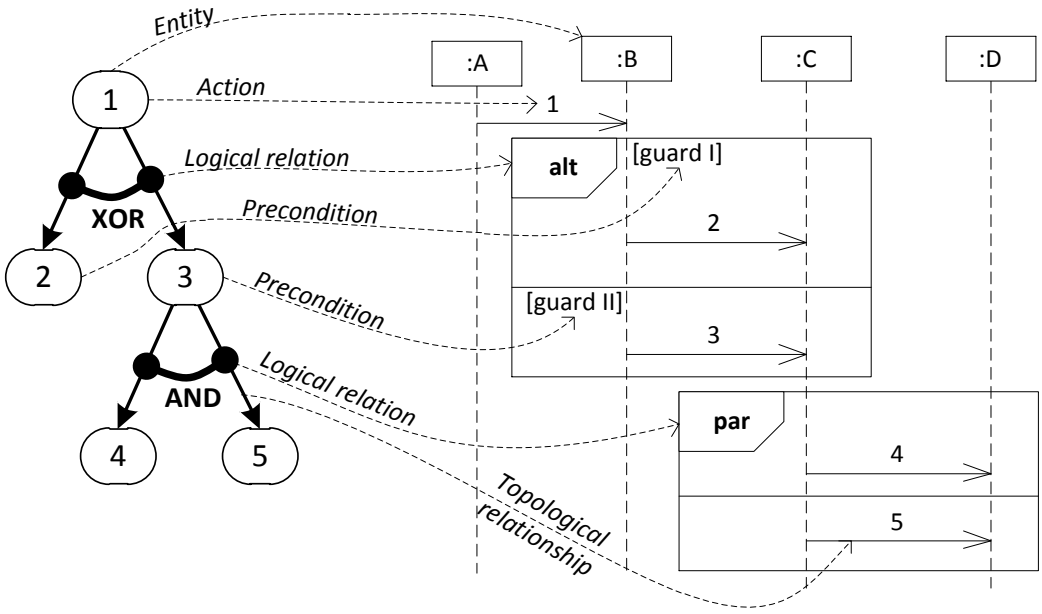


Figure 4.12. Mappings between elements of TFM and Sequence diagram

An example showing TFM to Sequence diagram transformation is given in Figure 4.12, where on the left side is given fragment of TFM and on the right side – fragment of Sequence diagram (the dashed arrows from TFM to Sequence diagram shows corresponding elements of TFM in the Sequence diagram). The **output** of messages of objects and their sequence analysis and design activity is Sequence diagram for each use case showing the objects and message sending between them.

4.2.3. Workflow Analysis and Design

Analysis and design of workflows consists of three actions (see Figure 4.13) in which one Activity diagram is developed for each Use case:

1. Setting scope of activity diagram,
2. Adding actions, and
3. Establishing control flow and logic.

Figure 4.13. Acquisition of Activity diagram

The **input** of this activity is use cases, TFM, and mappings between functional features and functional requirements.

Scope of each Activity diagram is set by the scope of corresponding Use case (i.e., the Activity diagram contains the description of the same functionality that is included into corresponding use case). The TFM and mappings allows **establishing actions and the control flow** between actions – functional features are transformed into action nodes and topological relationships into activity edges. **The logic of control flow** (i.e., decision, merge, fork, and join) is defined in accordance with the logical relations Lout within TFM as shown in the Activity diagram pattern in Table 4.2 on page 106. An example showing TFM to Activity diagram transformation is given in Figure 4.15, where on the upper side is given fragment of TFM and on the lower side – fragment of Activity diagram (the dashed arrows from TFM to Activity diagram shows corresponding elements of TFM in the Activity diagram).

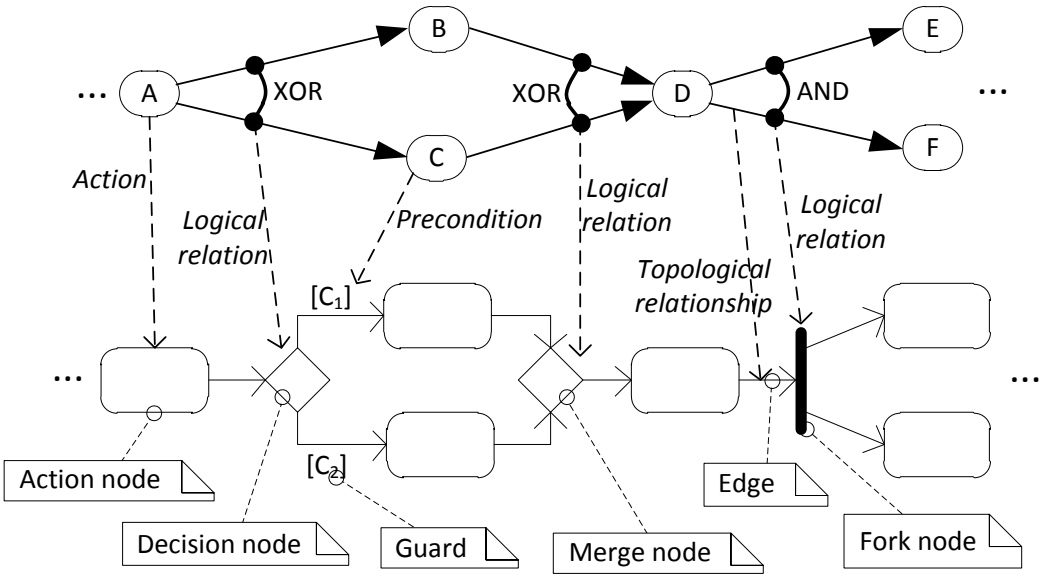


Figure 4.14. Example of TFM to Activity diagram transformation

The **output** of this activity is Activity diagram for each use case representing its workflow.

4.2.4. Workflows and Messaging Design

The **input** of this activity is Activity diagrams (representing the workflow of use case) and Sequence diagrams (showing the objects and messages they send each other).

The workflow and messaging design together is represented with interaction overview diagram which define interactions through a variant of Activity diagram in a way that promotes overview of the control flow. Interaction overview diagram focus on the overview of the flow of control where the nodes are of type *Interaction* or *InteractionUse*. The lifelines and the messages do not appear at this overview level. [89]

The Interaction overview diagrams are developed by merging created Activity diagrams and Sequence diagrams. While the first one gives the information about control flow, the latter shows objects and messaging between them. The obtained diagram can be helpful in order to better understand the overall process of system and the control flow relations between Sequence diagrams.

As **output** a set of Interaction overview diagrams are created.

4.3. Structure Analysis and Design

Domain model analysis and design is based on the Topological class diagram and consists of the following activities (see Figure 4.15):

1. Analysis of objects structure and communication,
2. Domain model development, and additionally
3. Modeling snapshots of the system.

The input of this activity is refined TFM and the output of this activity is the domain model in the form of Topological class diagram, Communication diagram, and Object diagram. The Object diagram is developed as an additional artifact when analyzing relationships between classes; it is advised to draw object diagram when one type of objects plays more than one role in the system [112].

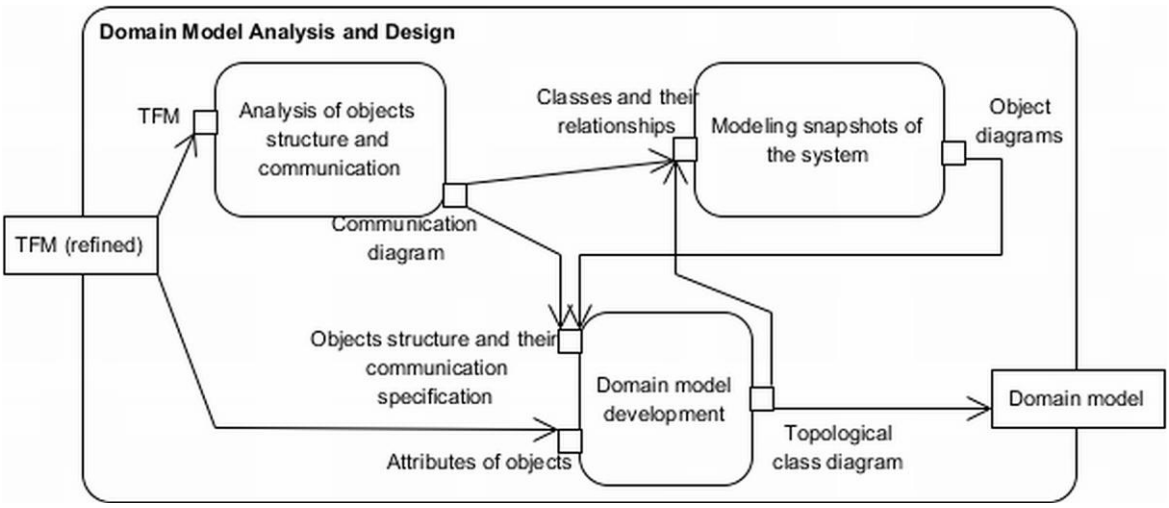


Figure 4.15. Development of domain model

The domain model analysis and design activities in detail are described in the subsequent subsections (each subsection describes one activity). Additional subsection is designated for initial Topological class diagram refinement.

4.3.1. Analysis of Objects and their Communication

Analysis of objects and their communication is based on the TFM transformation into Communication diagram. When transforming TFM into Communication diagram the following TFM elements are used:

- *Functional features* – source for lifeline identification and message sending from object to object,
- *Topological relationships* – from which lifeline to which lifeline the message is sent and the message sending sequence, and
- *Logical relations* – message sending concurrency.

The development process of Communication diagram is given in Figure 4.16).

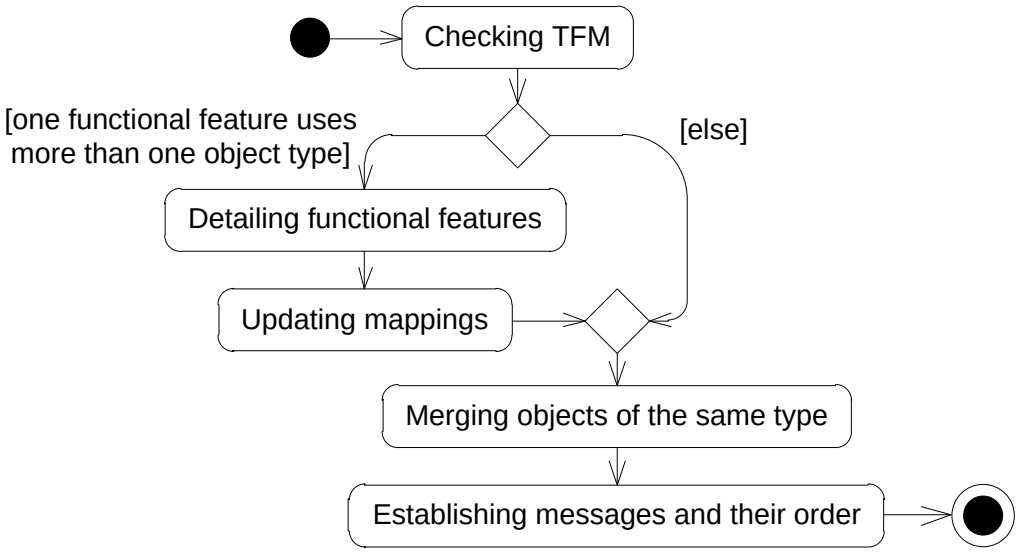


Figure 4.16. Development of Communication diagram

The **input** of this activity is refined TFM.

In order to obtain a Communication diagram, it is necessary to check if each functional feature of the TFM reflects only one type of object. If some of functional feature reflects more than one type of object then it is needed to decompose it to the level where one functional feature uses only one type of objects. If TFM has been successfully checked it can be transformed into Communication diagram. The first step in transformation is to merge functional features with objects of the same type in one lifeline (the lifeline represents the *class* attribute of the functional feature). While merging functional features into lifelines the relationships with other lifelines should be retained (if there is more than one topological relationship then only one link is added between lifelines). The count of topological relationships between merged functional features denotes the count of messages sent between lifelines represented by those functional features. Messages can be obtained from functional features because one functional feature represents one atomic business action. The message that is sent to a lifeline is an operation attribute of the functional feature (e.g., if functional

feature B is descendant of functional feature A, then in Communication diagram the lifeline representing A sends a message to lifeline representing B and this message is the value of *operation* attribute of B). Actors to Communication diagram are added from the input functional features – value of the *entity* attribute is used.

For a better understanding of TFM to Communication diagram transformation, a small fragment of TFM consisting of two functional features A and B is used (see Figure 4.17), where A is an input functional feature of TFM.

An example showing TFM to Communication diagram transformation is given in Figure 4.17, where on the upper side is given fragment of TFM and on the lower side – fragment of Communication diagram (the dashed arrows from TFM to Communication diagram shows corresponding elements of TFM in the Communication diagram). Fragment of TFM consists of two functional features A and B, where A is an input functional feature.

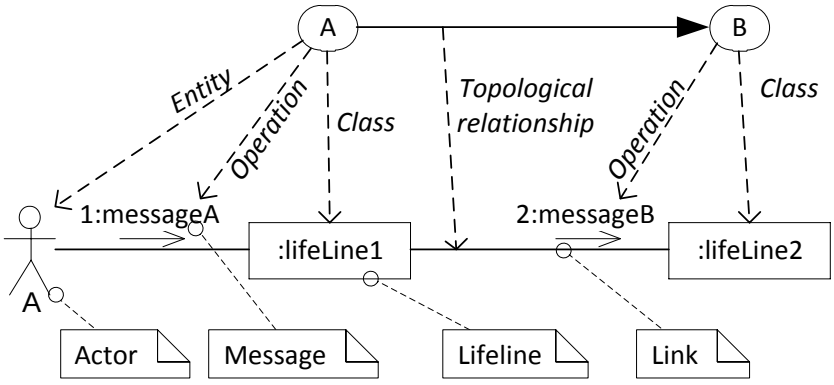


Figure 4.17. Example of TFM to Communication diagram transformation

As **output** of this activity the partial domain model in the form of Communication diagram is created.

4.3.2. Domain Model Development by Means of Topological Class Diagram

Domain model development by means of Topological class diagram consists of four activities (see Figure 4.18):

1. Adding classes and operations,
2. Adding topological relationships between classes,
3. Identifying attributes, and

4. Refining initial Topological class diagram (this activity is covered in next subsection – 4.3.3 on page 122).

The **input** of this activity is refined TFM and Communication diagram, while the **output** is a domain model in the form of Topological class diagram.

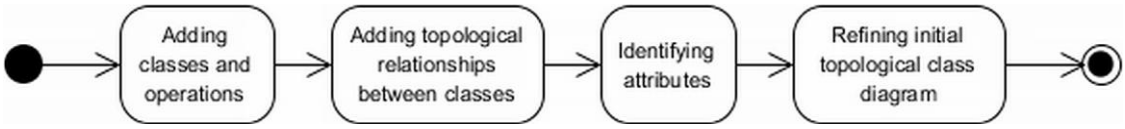


Figure 4.18. Development of topological class diagram

At first the Communication diagram is used for **adding classes and operations** to the Topological class diagram – lifelines are transformed into classes and messages into operations. The next step is **adding topological relationships** between classes. Since the notation of Topological class diagram allows variations of topological relationship graphical representation, it is advised to draw only one directed arrow in the same direction between classes (the arrow will show the cause and the effect operations). The example of Communication to Topological class diagram transformation is given below in Figure 4.19, where on the upper side is given fragment of Communication diagram and on the lower side – fragment of Topological class diagram.

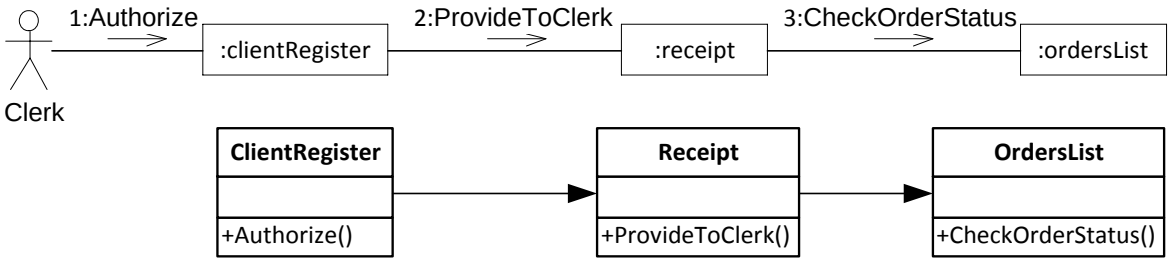


Figure 4.19. Example of Communication to Topological class diagram transformation

After the classes and topological relationships between them have been established the next step is **identification of attributes**. This can be achieved by taking into consideration attributes of the object represented by functional feature. If the functional feature is well specified the class attribute of it is determined. If the class attribute is not determined, it can be specified in several ways (e.g., by analyzing functioning description of the system and

searching nouns that represents attributes of the object [29], performing expert interviews [112], or by using ontology [133]).

By transforming Communication diagram an initial Topological class diagram is obtained (with attributes, operations, and topological relations between classes). A topological relation shows the control flow within the system. If static relations should be included (such as associations, generalization, etc.) then initial topological class diagram should be refined. The refinement process of initial Topological class diagram is described in the next subsection.

4.3.3. Refinement of Topological Class Diagram

The refinement of Topological class diagrams is aimed to lower abstraction level of it. By lowering abstraction level the diagram gets additional information which is needed during the software development and later also during its maintenance. The refinement process consists of six actions [31] (see Figure 4.20):

1. Identify generalizations (basing on topological relationships, attributes, operations, and responsibilities),
2. Define interfaces (both provided and required),
3. Identify structural relationships between classes (aggregations, compositions, and associations),
4. Identify enumerations,
5. Check for additional relationships (such as dependencies and realizations), and
6. Revise topological class structure.

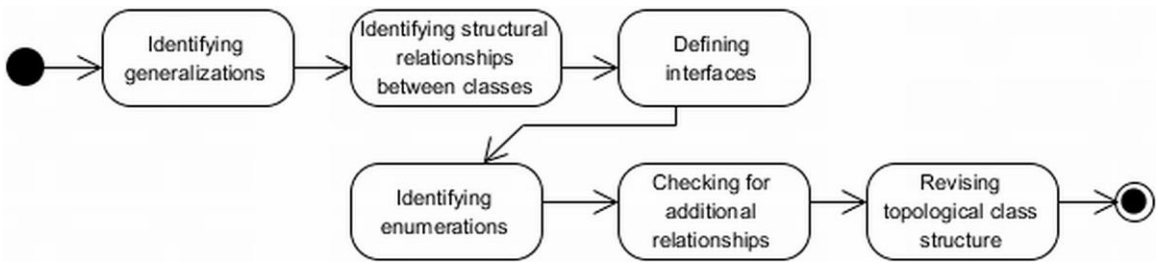


Figure 4.20. Refinement process of topological class diagram

The actions of refinement process are described in detail in the subsequent subsections. As a result of applying refinement process, a rich Topological class diagram with lower abstraction level is obtained.

4.3.3.1. Identifying Generalizations

A generalization is a relationship between a general kind of thing (called the superclass or parent) and a more specific kind of thing (called the subclass or child). Generalization sometimes is called an "*is-a-kind-of*" relationship. If subclass has one superclass then it is single inheritance. If subclass has two or more superclasses then it is multiple inheritance. [15]

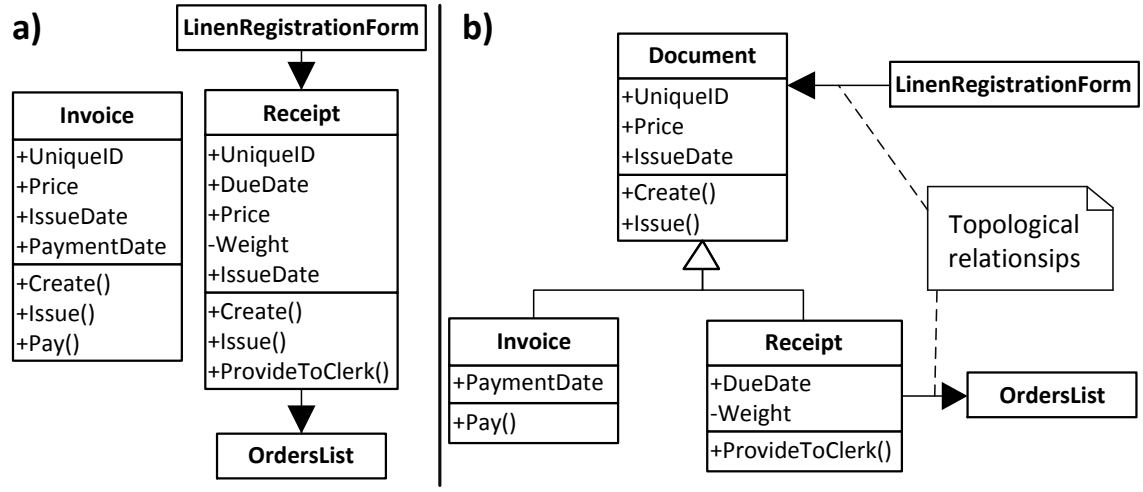


Figure 4.21. Initial topological classes (a) and generalized topological classes (b) showing topological and generalization relationships between them

The generalizations can be identified in two ways. The first way is to review initial topological classes which are obtained from the TFM. To find a generalization you need to look for the same responsibilities, topological relationships, attributes, and operations that are common to two or more classes. The set of common responsibilities, topological relationships, attributes, and operations can be elevated to a more general class. If this general class does not exist it can be created. Since topological relationships define control flow within system, by introducing general classes and generalization relationships it is possible that the more general class is placed at the end of topological relationship and the more specific class is placed at the beginning of topological relationship (see Figure 4.21). In order to help identifying generalizations, during the review process of initial topological classes, an

additional attention can be paid on anywhere where the initial topological classes indicates that there is more than one "*kind of*" thing (for example, two kinds of documents (see Figure 4.21b). This indicates a possible generalization.

The second way is by doing additional interviews with stakeholders. During the interviews the interviewees are asked if any of the classes are variations on others [112]. By applying both ways in generalization identification a more formal (by reviewing initial topological classes) and less formal (by making interviews) approaches are used. The reviewing process is more formal because it is based on sets of already existing information. Reviewing and introduction of generalization relationships (together with superclasses) can be automated. By using together reviewing and interviewing an additional model checking gets performed.

4.3.3.2. Defining Interfaces for Collaboration with Environment

An interface is a collection of operations that are used to specify a service of a class or a component. Graphically, an interface may be rendered as a stereotyped class in order to expose its operations and other properties. Interfaces may also be used to specify a contract for a use case or subsystem. [88]

A line around the topological class diagram which is obtained by applying transformations on the TFM can be drawn, thus showing the boundary of the system under consideration. The next step is to identify the operations and the signals that cross this boundary. These operations and signals can be found by analyzing both the TFM and the topological space of the system (the TFM shows the functioning of the system, but topological space shows the system within the *(surrounding)* environment). This analysis shows the inputs and outputs of the system. The input functional features within TFM indicate the **provided interfaces**, but the output functional features indicate the **required interfaces**. Required (imported) interfaces are modeled by using dependency relationships, and provided (exported) interfaces are modeled by using realization relationships. An example of showing analysis of TFM and topological space and the resulting interfaces is given in Figure 4.22, where in the middle is given fragment of TFM and on the sides – fragment of Topological class diagram (the dashed arrows from TFM to Topological class diagram shows corresponding elements of TFM in the Topological class diagram).

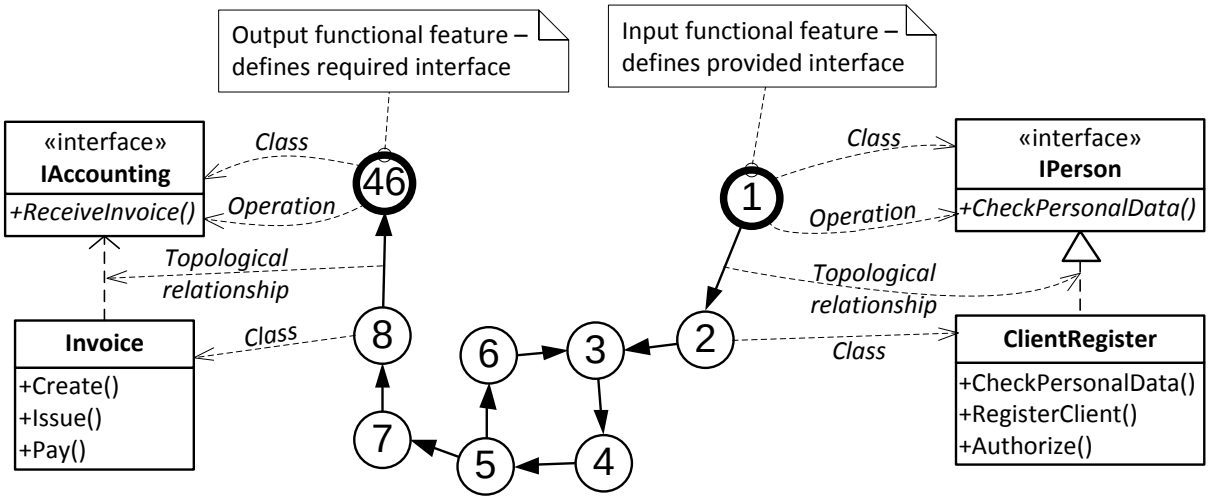


Figure 4.22. Fragment of topological space and examples of provided and required interfaces

By using the guidelines given in [15] it is possible to model interfaces within the system as a seams between different parts of the system.

4.3.3.3. Identifying Structural Relationships

The identification of physical relationships between entities involved in the system consists of three steps. At first it is needed to check and find the whole and part relationships – aggregations and compositions.

Aggregation is a "*has a*" relationship meaning that an object of the whole has objects of the part. [41] If objects are related with an aggregation then by destroying the object of the whole, the objects of the part is not destroyed. Aggregation is a special kind of association. According to guidelines given in [112], aggregation can be placed between objects if a part object can belong to more than one whole object and the part continues to exist when the whole is destroyed. Words that suggest aggregation include "*collection*", "*list*", and "*group*".

Composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it. This means that, in a composite aggregation, an object may be a part of only one composite at a time and by destroying whole, the parts are destroyed with it [41]. According to guidelines given in [112], composition can be placed between objects if a part is totally "owned" by the whole and the part ceases to exist when the whole is destroyed. Words that suggest composition include "*composed of*" and "*component*".

After identification of aggregations and compositions, the next step is identification of **associations** between classes. An association is a structural relationship that specifies that

objects of one thing are connected to objects of another. Given an association connecting two classes, it is possible to relate objects of one class to objects of the other class [88]. According to guidelines given in [15], associations can be placed between objects if it is needed to navigate from objects of one type to objects of another. This is a data-driven view of associations.

4.3.3.4. Identifying Enumerations

An enumeration is a data type whose values are enumerated in the model as enumeration literals [88]. Enumeration is a kind of data type, whose instances may be any of a number of user-defined enumeration literals. An enumeration may be shown using the classifier notation (a rectangle) with the keyword *«enumeration»*.

The enumeration within a system can be found in two ways. The first way is to review initial topological classes which are obtained from the TFM of the system under consideration. To find enumerations at first you need to look for attributes which can contain only a restricted set of values. In the context of the laundry system, an example of the restricted set of values is the requested washing type. The second thing is to search for objects which can change its state value during its lifetime. In the context of the laundry system, an example of such object is washing request. The washing request can have different states, for example, *new*, *registered*, *in washing*, *completed*, *paid*. The second way is by doing additional interviews with stakeholders. During the interviews the interviewees are asked if any of the attributes has only limited list of allowed values or if there exist a states of things involved into system. If such lists of values or states exist, then enumerations should be defined for each such list. An example of identified enumerations is given in Figure 4.23.

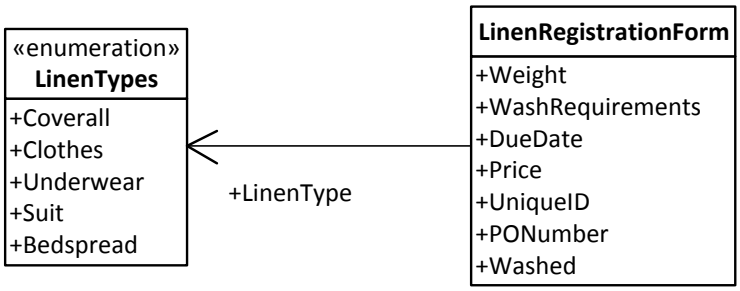


Figure 4.23. Example of identified enumeration in linen registration form

4.3.3.5. Revising Topological Class Structure

The final step in topological class diagram refinement is the revising of topological class structure. The revising of topological class structure should be done using fol-lowing guidelines (revising guidelines for generalizations are based on guidelines given in [112]):

1. Any classes that have the same topological relationships or associations to other classes should be identified. If such classes exist, a decision of adding additional generalized class should be made. If generalized class is added, then common topological relationships and associations should be moved to it.
2. Any classes that have the same attributes or operations as other classes should be identified. If such classes exist, a decision of adding additional generalized class that will contain common attributes and operations should be made.
3. Every generalized class in the topological class diagram should be justified. The point of introducing a generalized class is to provide a convenient, single place to put rules that affect a number of specialized classes. There should be at least one attribute, operation, or relationship that can be ascribed to the generalized class.
4. As a final revising step of generalized classes is that each generalized class should have at least two specializations, with two exceptions:
    a. The generalized class is concrete, and
    b. It is anticipated that more specializations will be added in the future.
5. Since the system is connected with the environment (through inputs and out-puts), at least one provided and one required interface should be identified. Revising of interfaces should follow these rules:
    a. The count of operations defined within provided interfaces should be the same as count of input functional features within TFM.
    b. The count of operations defined within required interfaces should be the same as count of output functional features within TFM.

After the revising process has been finished, the initial topological class diagram is refined and the abstraction level of it has been lowered. Mainly the abstraction level should be lowered in order to introduce generalized classes, structural relationships, and interfaces.

4.3.4. Modeling System Snapshots

The **input** of this activity is Topological class diagram or Communication diagram.

Since object diagram shows a complete or partial view of the structure of a modeled system at a specific time moment [89] it can be is used instead of a Topological class diagram in situations that involve more than one object of the same class acting in different roles [112] or to provide examples of a system at a specific time. An Object diagram focuses on particular set of object instances and attributes, and the links between the instances. A set of object diagrams provides insight into how a view of system is expected to evolve over time. Only those aspects of a model that are of current interest need be shown on an Object diagram. When Topological class diagram is transformed into a set of Object diagrams, the classes become instance specifications, and associations – links.

As the **output** of this activity a set of object diagrams is created describing the objects details of the domain model.

4.4. Object State Change and Transition Analysis

Object state change and transition analysis is based on the State diagram and consists of one activity (see Figure 4.24). The **input** of this activity is refined TFM and classes (either from Topological class diagram or lifelines from Communication diagram) and the **output** of this activity is one State diagram for each class. It is advised to analyze state changes of complex or most important objects in the system. The most important objects are those that are participating in the main functioning cycle of the system.

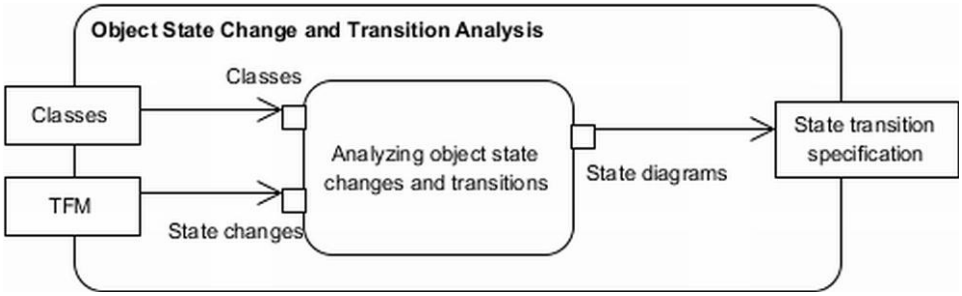


Figure 4.24. Analyzing object state changes and transitions

The first action is to **scale down TFM** which is performed by removing functional features which does not represent the object under consideration but in the same time retaining cause-and-effect relations. For example, assume that TFM consists of three functional features A, B, and C and are in the following causal chain: A-B-C. The A and C

represent the same object while B represents another object. The resulting (scaled down) TFM is as follows: A-C.

States for each class **are obtained** from the functional features of refined TFM (functional feature has an attribute named *newState* as shown in Figure 3.9 on page 84). If the execution of functional feature involves the change of the corresponding object's state, then the attribute *newState* has value, otherwise the value is not set. State transitions are obtained by transforming cause-and-effect relationship between functional features. [30]

The special states (initial state and final state) are added to the obtained State diagram as follows:

- The **initial state** is added before the states that are obtained from the functional features which are the inputs of the downscaled TFM (the ones which has no predecessors), and
- The **final state** is added after the states that are obtained from the functional features which are the outputs of the downscaled TFM (the ones which has no descendants).

The example of transforming generic example of TFM into state diagram is given in Figure 4.25, where the upper part shows fragment of TFM and lower part – fragment of State diagram.

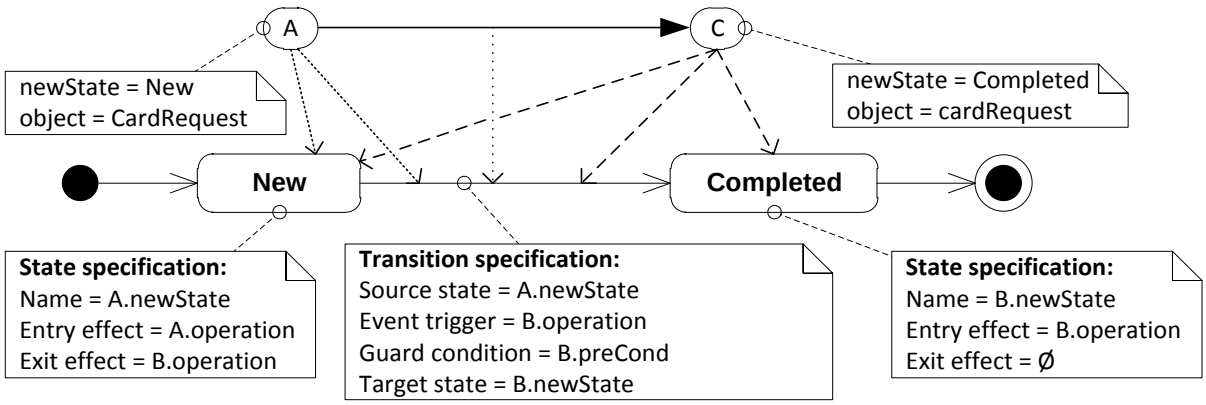


Figure 4.25. Example of TFM to State diagram transformation

4.5. Structuring Logical Layout of Software Design

Logical layout of software design is structured in accordance with the defined subsystems in the system behavior analysis and design activity and the **input** of this activity is

subsystems (Use case diagram) and classes with their relationships (Topological class diagram) as given in Figure 4.26.

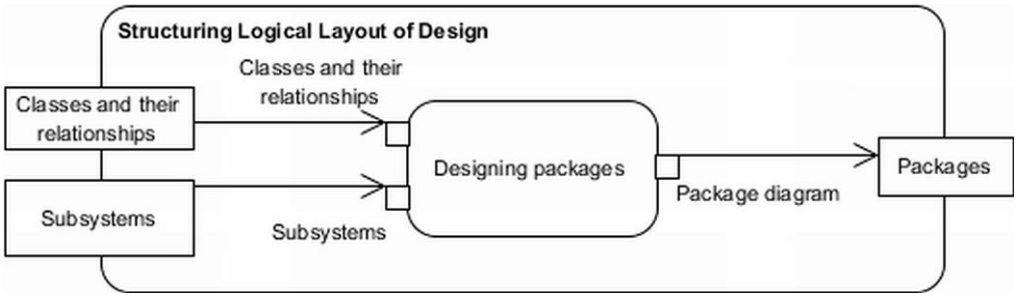


Figure 4.26. Structuring the logical layout of software design

The logical layout is depicted by using Package diagram where each package initially represents one subsystem. The contents of packages are added from the Topological class diagram accordingly to the use cases in each system and the mappings between functional features and Use cases. Thus each package gets a set of classes that are responsible for particular subsystem. If needed the initial packages can be split up by grouping classes by their responsibilities. The **output** of this activity is Package diagram structured according to subsystems and responsibilities of classes.

4.6. Components and Deployment Design

The components and deployment design consists of two consequent activities (see Figure 4.27): designing components, and planning deployment.

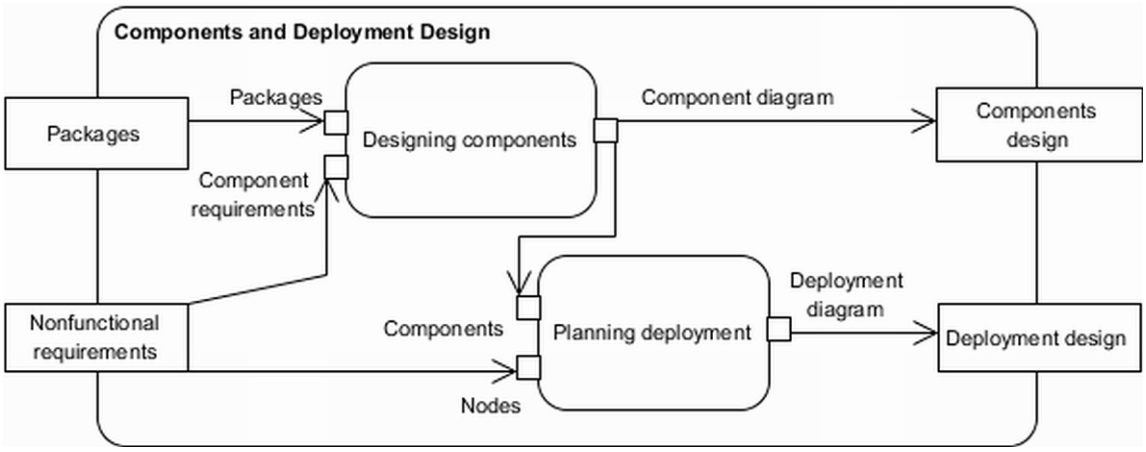


Figure 4.27. Overview of designing components and planning deployment

The **input** of this activity is packages from Package diagram and nonfunctional requirements, and as the **output** a component design (Component diagram) and deployment design (Deployment diagram) is created. The components and deployment design activities in detail are described in the subsequent subsections (each subsection describes one activity).

Components designing within TopUML modeling are performed according to the packages and nonfunctional requirements. The **input** of this activity is Package diagram and nonfunctional requirements. The components designing process consists of two subsequent actions (see Figure 4.28):

- **Defining components** – the initial components are designed, one component for each package, and
- **Refining components** – initial components are refined according to nonfunctional requirements.

For example, the nonfunctional requirements may include security requirements by stating that executable files responsible for logging into system should be deployed separately of other components.

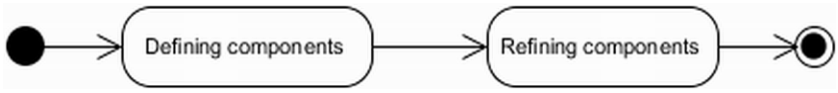


Figure 4.28. Components designing process

As the **output** a Component diagram is developed.

Deployment planning within TopUML modeling is made according to the components and nonfunctional requirements. During the deployment planning the components are assigned to the nodes as specified by nonfunctional requirements. According to the UML specification [89], "a node is computational resource upon which artifacts may be deployed for execution". As the output a Deployment diagram is created which represent the assignment of software artifacts to nodes.

4.7. TopUML Modeling in Comparison with other Modeling Methods

This section compares TopUML modeling with UML modeling driven approaches discussed in Section 2. Each of the method is evaluated against a set of criterions divided into four groups:

- **Analysis and design models** – shows the summary of diagram types used and the transformation guidelines provided,
- **Problem domain analysis and design** – sets the emphasis on the evaluation of the existing and desired domain model designing and the identification of the boundaries of them,
- **Requirements management** – describes the requirements management aspect of the compared methods (like requirements traceability), and
- **Usage** – evaluates the practical application of each method in the software development.

The criterions in each group are selected from a set of modeling methods and techniques reviews and guidelines ([5], [37], [68], and [111]) basing on the purpose of TopUML modeling.

The first evaluated group of criterions is analysis and design models and it consists of following evaluation criterions:

1. **Count of diagrams used** – shows the number of diagrams used by each method.
2. **The first diagram created** – indicates the first diagram that is created by particular method (this criterion indicates the diagram that is driving the further software development process, e.g., some methods are called use case driven).
3. **Transformations provided** – if method includes transformation guidelines for all diagrams addresses, then the value is *all*; if part of the diagrams are covered – *partial*; and if no guidelines for transformations are given – *none*.
4. **Transformation automation level** – if the transformation process can be done automatically (i.e., without presence of human) then the value is *automatic*; if some interaction is required by the human – *semi-automatic*; or *manual* (no guidelines and mappings between diagrams and diagram elements are given).

The result of the evaluation against analysis and design models criterions is given below in Table 4.5.

Table 4.5

Evaluation of analysis and design models criterions

No.	Criterion	Method	Evaluation
1.	Count of diagrams used	TopUML modeling	12 – all except Profile diagram, Timing diagram, and Composite structure diagram
		OOAD with Unified Process	10 – all except Composite structure diagram, Object

No.	Criterion	Method	Evaluation
			diagram, Profile diagram, and Timing diagram
		B.O.O.M.	6 – Use case diagram, Activity diagram, State diagram, Class diagram, Package diagram, Object diagram
		Conceptual modeling	5 – Class diagram, State diagram, Use case diagram, Sequence diagram, and Profile diagram
		Component based development	9 – Composite structure diagram, Object diagram, Profile diagram, Interaction overview diagram, and Timing diagram
		TFMfMDA	5 – three UML diagrams: Use case diagram, Activity diagram, and Class diagram; one diagram from TFMfMDA profile: TFM; and unspecified diagram type – Problem domain objects graph
2.	The first diagram created	TopUML modeling	TFM
		OOAD with Unified Process	Use case diagram
		B.O.O.M.	Use case diagram and Class diagram (in parallel)
		Conceptual modeling	Class diagram
		Component based development	Not specified, varies from case study to case study
		TFMfMDA	TFM
3.	Transfor-mations provided	TopUML modeling	All
		OOAD with Unified Process	Partial
		B.O.O.M.	Partial
		Conceptual modeling	Partial
		Component based development	None
		TFMfMDA	All
4.	Transfor-mation auto-mation level	TopUML modeling	Semi-automatic – some diagram types requires additional actions from expert or designer
		OOAD with Unified Process	Semi-automatic – depending on the design patterns used
		B.O.O.M.	Manual
		Conceptual modeling	Semi-automatic
		Component based development	Manual
		TFMfMDA	Semi-automatic

The second evaluated group of criterions is problem domain analysis and design and it consists of following evaluation criterions:

1. **Representation of "as-is" domain model** – diagram types used to specify and analyze the existing functioning of the problem domain. This is an important criterion since it is needed to understand the functioning of the existing problem

domain (e.g., business system) and only then introduce the new concepts and functions into it.

2. **Representation of "to-be" domain model** – diagram types used to specify and analyze the desired functioning of the solution.
3. **"As-is" boundary identification** – approach used for the existing ("as-is") domain boundary identification (i.e., before the new software system is introduced to the problem domain).
4. **"To-be" boundary identification** – approach used for the desired ("to-be") domain boundary identification (i.e., after the new software system is introduced to the problem domain) since the new software system can introduce a new functionality in the problem domain.

The result of the evaluation against problem domain analysis and design criterions is given below in Table 4.6.

Table 4.6

Evaluation of problem domain analysis and design criterions

No.	Criterion	Method	Evaluation
1.	Representation of "as-is" domain model	TopUML modeling	TFM
		OOAD with Unified Process	Use case diagram
		B.O.O.M.	Business use cases and Activity diagram
		Conceptual modeling	Partly by Class diagram
		Component based development	None
		TFMfMDA	TFM
2.	Representation of "to-be" domain model	TopUML modeling	TFM, Communication diagram, Topological class diagram, and Object diagram
		OOAD with Unified Process	Use case diagram, Activity diagram, and Class diagram
		B.O.O.M.	Use case diagram and Class diagram
		Conceptual modeling	Class diagram
		Component based development	Scenario and Logical view of 4+1 architectural style
		TFMfMDA	TFM, Class diagram with conceptual classes
3.	"As-is" boundary identification	TopUML modeling	Initial TFM – the result of applying topological space closure operation
		OOAD with Unified Process	Intuitive – based on initial estimation of Use cases
		B.O.O.M.	Business use cases – an initial estimation based on interviews
		Conceptual modeling	Intuitive
		Component based development	None

No.	Criterion	Method	Evaluation
		TFMfMDA	TFM as the result of topological space closure operation
4.	"To-be" boundary identifi-cation	TopUML modeling	Refined TFM – the result of mapping TFM on functional requirements or goals
		OOAD with Unified Process	Intuitive – based on analysis of Use cases
		B.O.O.M.	System use cases – identified and elaborated basing on the business use cases
		Conceptual modeling	Intuitive
		Component based development	Intuitive
		TFMfMDA	User goals finding in accordance with the TFM

The next evaluated group of criterions is requirements management and it consists of following evaluation criterions:

1. **Functional requirements** – The way of the functional requirement specification
2. **Nonfunctional requirements** – The way of the nonfunctional requirement specification
3. **Requirements conformance** – evaluates the conformance of functional requirements to the existing domain functioning.
4. **Functional requirements traceability** – support of the functional requirements traceability between developed artifacts.
5. **Nonfunctional requirements traceability** – support of the nonfunctional requirements traceability between developed artifacts.

The result of the evaluation against requirements management criterions is given below in Table 4.7.

Table 4.7

Evaluation of requirements management criterions

No.	Criterion	Method	Evaluation
1.	Functional require-ments	TopUML modeling	Textual description, business use cases, etc.
		OOAD with Unified Process	Use case diagram
		B.O.O.M.	System use case diagram
		Conceptual modeling	Conceptual schema (all UML diagrams applied by this method)
		Component based development	Use case diagram (the Scenario view of 4+1 architecture style)
		TFMfMDA	Textual description, Use case diagram
2.	Non-	TopUML modeling	Textual description

No.	Criterion	Method	Evaluation
	functional require-ments	OOAD with Unified Process	Textual specification within use cases
		B.O.O.M.	Textual specification within system use cases
		Conceptual modeling	None
		Component based development	Components should conform to a characterization of a unit to be reusable
		TFMfMDA	Partially by textual specification within use cases
3.	Require-ments confor-mance	TopUML modeling	Mappings between requirements and the initial and refined TFM
		OOAD with Unified Process	Intuitive or based on knowledge of expert
		B.O.O.M.	Based on business use cases
		Conceptual modeling	Intuitive or based on knowledge of expert
		Component based development	Intuitive or based on knowledge of expert
		TFMfMDA	Mappings between requirements and the TFM "as-is"
4.	Functional require-ments traceability	TopUML modeling	Trace links from requirements to functional features of TFM, Use cases, Topological class diagram and other developed artifacts (TopUML modeling enables traceability from the very beginning of software development lifecycle to the code)
		OOAD with Unified Process	Requirements are traced to Use cases and to other design and implementation artifacts.
		B.O.O.M.	Trace links from business use cases to system use cases and other developed artifacts
		Conceptual modeling	Intuitive; trace links between diagrams created in the conceptual schema
		Component based development	Intuitive; trace links from Scenario view (Use cases) to other views in the context of 4+1 architecture style
		TFMfMDA	Trace links between requirements, TFM, Use cases and other diagrams used
5.	Non-functional require-ments traceability	TopUML modeling	Trace links from nonfunctional requirements to Component diagram and Deployment diagram
		OOAD with Unified Process	Traced together with use cases
		B.O.O.M.	Traced together with system use cases
		Conceptual modeling	Not supported
		Component based development	Intuitive; based on the experience of designer
		TFMfMDA	Not supported

The last evaluated group of criterions is Usage and it consists of following evaluation criterions:

1. **Type of validation** – evaluates how the method is validated. There can be three different ways on validating a method depending on the purpose of the validation and the conditions for empirical investigation [37]: *survey*, *case study*, and *experiment*. A survey is an investigation performed in retrospect when the method has been used for a certain period of time. A case study is an observational study in which data is collected for a specific purpose throughout the study. An experiment is a formal and controlled investigation (it includes also the theoretical examples).

The result of the evaluation against usage criterions is given below in Table 4.8.

Table 4.8

Evaluation of usage criterions

No.	Criterion	Method	Evaluation
1.	Type of validation	TopUML modeling	Case study (as given in section 5.2 and [29])
		OOAD with Unified Process	Survey
		B.O.O.M.	Survey
		Conceptual modeling	Experiment (a conceptual modeling "case study" in [92] is shown in the context of already existing system so it cannot be considered as a real case study)
		Component based development	Case study
		TFMfMDA	Experiment

The conducted evaluation of UML modeling driven methods shows the TopUML modeling on the background of other methods. Since TopUML modeling and TFMfMDA both are based on TFM, several characteristics of them are equal or similar.

4.8. Summary

TopUML modeling is based on the formalism of TFM and the TopUML profile. While TFM provides formal abstraction and understandability of the problem domain, other TopUML diagrams allows to model system from different viewpoints. TFM is used in two forms – "*as-is*" and "*to-be*", where the first one describes the existing functioning of the problem domain while the latter one – the required functioning of the solution (i.e., software system). Within TopUML modeling TFM serves as the root model for behavior and structure specification and as a tool for validation of the functional requirements as well as of the

software system. The problem domain analysis and software design within TopUML modeling consists of six activities, covering behavior, structure, layout, and deployment analysis and design. *By following the TopUML modeling activities one by one, the system gets designed in top-down way and the developed design artifacts are in strong accordance with the functioning of problem domain, thus the TopUML modeling ensures that causal trace links exist between artifacts of both problem and solution domains.* By following the TopUML modeling activities, designed software artifacts have following characteristics:

- High cohesion – application of TFM for system functioning analysis and formal transformation of TFM to other diagram types ensures appropriate assignment of responsibilities to objects and classes (this actually is leading to the next statement),
- Every design artifact is an abstraction of a well analyzed and understood problem domain artifact,
- Low coupling with the rest of the system – relations between elements initially are identified within TFM and defined as topological relationships between functional features describing functioning of a problem and solution domains, and
- Well-defined interface – the result of performing closuring operation of topological space is TFM reflecting functioning of system under consideration; the TFM obtained from topological space shows inputs and outputs (one of the functional characteristic of TFM). TopUML modeling uses the inputs and outputs of TFM do define required and provided interfaces of system.

The comparison of TopUML modeling and other UML modeling driven methods shows evaluation of a set of criterions. Since TopUML modeling and TFMfMDA both are based on TFM, several characteristics of them are equal or similar. The TopUML modeling solves one of the weakest points of the TFMfMDA approach – assignment of responsibilities to appropriate classes. TFMfMDA uses a powerful tool to analyze functioning of the problem domain – the TFM, but the TFMfMDA lacks the ability to transfer responsibilities of objects from TFM to the classes. TopUML modeling solves this issue by transforming the TFM into Topological class diagrams. Proposed method is able to identify and reflect functioning cycles of a system (including the main functional cycle) within developed diagrams. The presence of functional cycles allows classifying classes thus the classes participating in functional cycles can be marked and highlighted in the solution.

5. APPROBATION OF TOPUML PROFILE AND MODELING METHOD

The implementation and approbation of TopUML profile and modeling method is shown and discussed in the context of two software designing projects:

1. *Business support application development* – shows a practical experiment in which the software is designed for a laundry problem domain, and
2. *Enterprise data synchronization system development* – covers a case study of software development project in which software is developed to perform enterprise data synchronization by taking data from multiple data sources and placing into centralized data storage.

A case study is considered as an observational study in which data is collected for a specific purpose throughout the study, and an experiment is a formal and controlled investigation [37]. Each of the discussed projects consumes slightly different parts of TopUML modeling method, e.g., the experiment uses system goals to set scopes of Sequence diagrams while the case study uses use cases.

Additionally this chapter discusses empirical evaluation of TopUML modeling provided by two expert groups participating in the experiment of business support application development. The modeling and development knowledge of experts before the experiment is determined by using self-evaluation questionnaire thus showing preliminary knowledge of each participant.

5.1. Business Support Application Development

Business support application development is demonstrated on the basis of practical experiment in which a software design for laundry functioning at the platform independent viewpoint is developed. The laundry business system is an experimental system created to demonstrate the capabilities of TopUML profile and modeling method. Experimental software designing includes creation of artifacts according to the TopUML modeling method. These artifacts are as follows and they are given in the subsections of this section:

- *Initial and refined TFM* in accordance with informal system description and functional requirements,
- *Sequence diagrams* and *Interaction overview diagram* in accordance with TFM and system goals,

- *Communication diagram* in accordance with TFM, and
- *Topological class diagram* in accordance with Communication diagram and TFM.

The used theory within experiment and obtained results are published in [105] (TFM transformation into Topological class diagram), [31] (the Topological class diagram refinement), and [27]. The latter reference is a guidance manual which is made in accordance with the knowledge gained by elaborating laundry experiment with two expert groups.

5.1.1. Specification of Laundry System

Specification of laundry business system includes following artifacts:

1. Informal description of laundry functioning (Appendix 4 on page 191), where nouns are denoted by *italic*, verbs are denoted by **bold**, and action pre- and post-conditions are underlined,
2. Functional requirements defined for laundry software system (Appendix 5 on page 194), and
3. Eight system goals of laundry business and software systems by the problem domain expert (Appendix 5 on page 194).

5.1.2. Problem Domain Functioning Analysis

Problem domain functioning analysis is based on the development of TFM representing functioning of laundry which involves the following activities:

1. *Topological space development* – functional features are identified during the analysis of system functioning informal description and establishment of cause-and-effect relationships between them thus creating a topological space of the laundry functioning,
2. *Initial TFM development* – TFM is obtained by performing closure operation [99] of the inner functional features thus taking out the functioning description of the system from the topological space, and
3. *TFM refinement* – functional features are mapped onto functional requirements thus showing missing functional features (i.e., the functionality that do not exist in the problem domain currently) and missing functional requirements.

All the activities in detail are shown in the subsequent subsections.

5.1.2.1. Topological Space Development

As the result of definition of physical or business functional characteristics for the laundry software system development project has been defined a set of 85 functional features (see Table 5.1 where a set of 5 functional features are given; full list of defined functional features is given in Appendix 6 on page 196). These functional features were identified during the analysis of laundry business system – the informal description of it.

Table 5.1

Functional features of laundry functioning

ID	Object action (A)	Precondition (PrCond)	Entity (E)	(S)
6.	Preparing client card	If the person does not have the client card yet	Clerk	Inner
7.	Client card issue to the client		Clerk	Inner
8.	Authorizing client status	If the person is registered (and) if the client has the client card	Clerk	Inner
9.	Requesting linen registration form	If client has client card	Client	Inner
10.	Creating linen registration form		Clerk	Inner

After definition of functional features a topology Θ is introduces between them (i.e., establishing cause-and-effect relations between functional features). Cause-and-effect relations are represented as arcs of a directed graph that are oriented from a cause vertex to an effect vertex. The identified cause-and-effect relations between the defined functional features are illustrated by the means of the topological space in Figure 5.1 where it is clearly visible that cause-and-effect relations form functioning cycles.

All cycles and sub-cycles should be carefully analyzed in order to completely identify existing functionality of the system. The main cycle of system functioning (i.e., functionality that is vital for the system's life) should be found and analyzed before starting further analysis. In laundry business case study the main functioning cycle of laundry functioning represents registering client's linen, washing it and returning to client. Functional features belonging to the main functioning cycle are as follows: "*8-9-10-11-12-13-14-15-16-17-18-19-20-22-23-27-28-29-32-33-34-42-40-41-44-45-48-51-8*" (denoted with the boldest arrows in Figure 5.1). In the case of studying a complex system, a TFM can be divided into a series of sub-systems according to the identified cycles.

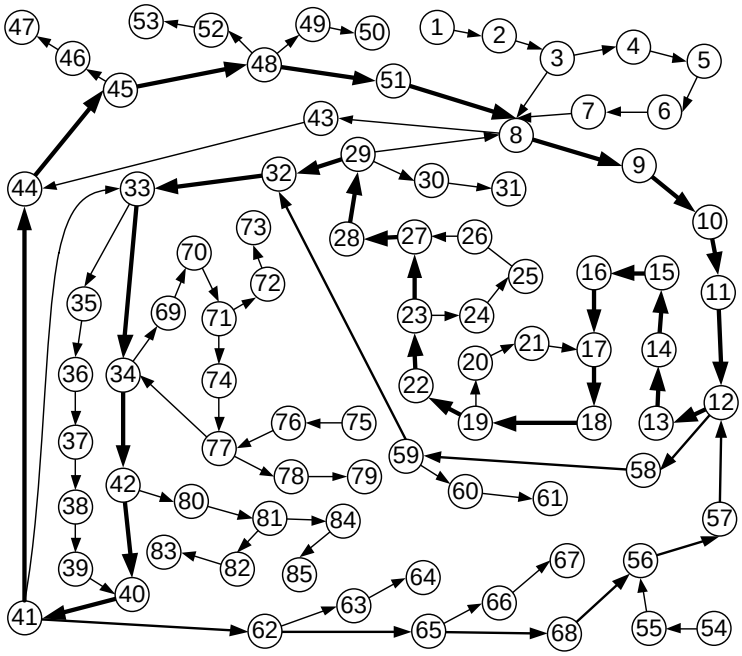


Figure 5.1. Topological space of laundry functioning

5.1.2.2. Initial Topological Functioning Model

According to the equation (2) on page 87 all the identified functional features given in Table 5.1 are split in two sets– the set N (inner functional features) the set M (external functional features and system functional features that affect the external environment):

1. N = { 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 48, 51, 56, 57, 58, 59, 62, 65, 68, 69, 70, 71, 74, 77, 80, 81 };
2. M = { 1, 2, 31, 46, 47, 49, 50, 52, 53, 54, 55, 60, 61, 63, 64, 66, 67, 72, 73, 75, 76, 78, 79, 82, 83, 84, 85 }.

The TFM is obtained by performing closuring operation (see equation (3) on page 87) over the set of system inner functional features (the set N). The example illustrating how the closuring operation is applied over the set N is given in Appendix 7 on page 200. The set X (*the TFM*) of laundry functioning is as follows: X={2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44}. The resulting graph showing TFM of laundry problem domain functioning is given in Figure 5.2.

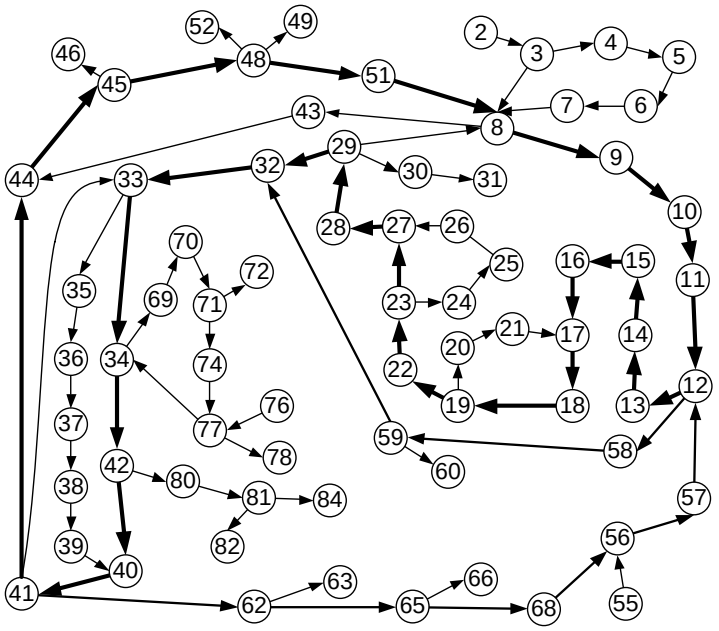


Figure 5.2. Topological functioning model of laundry functioning

5.1.2.3. Refining Topological Functioning Model

By mapping functional features onto functional requirements it is visible which functional requirements define new functionality (a functionality that currently does not exists in the problem domain) and which functional requirements are missing (a functionality that currently exist in problem domain but is not covered by functional requirements). The mappings between functional features and functional requirements are as follows (the symbol Ø is used if a functional requirement describes functionality which is not present in the problem domain):

FR1 = { 2, 3 };

FR2 = { 4, 5, 6, 7 };

FR3 = { Ø };

FR4 = { 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31};

FR5 = { 23, 24, 25, 26 };

FR6 = { 30, 31 };

FR7 = { 34, 42 };

FR8 = { 32, 33, 41 };

FR9 = { Ø };

FR10 = { 43, Ø };

FR11 = { 44 };

FR12 = { 45, 46, 48 };

FR13 = { 49, 51 };

FR14 = { 55, 56, 57, 58, 59, 60, 62 };

FR15 = { 60 };

FR16 = { 62, 65, 66 };

FR17 = { 63, 68 };

FR18 = { 69, 70, 71, 72, 74, 76, 77, 81, 82 }; and

FR19 = { 80, 82 }.

The mappings between functional features and functional requirements using arrow predicates is given in Figure 5.3 where it can be seen that the most used relation type is One-to-Many. One-to-Many shows that one functional requirement completely specifies functionality defined by several functional features.

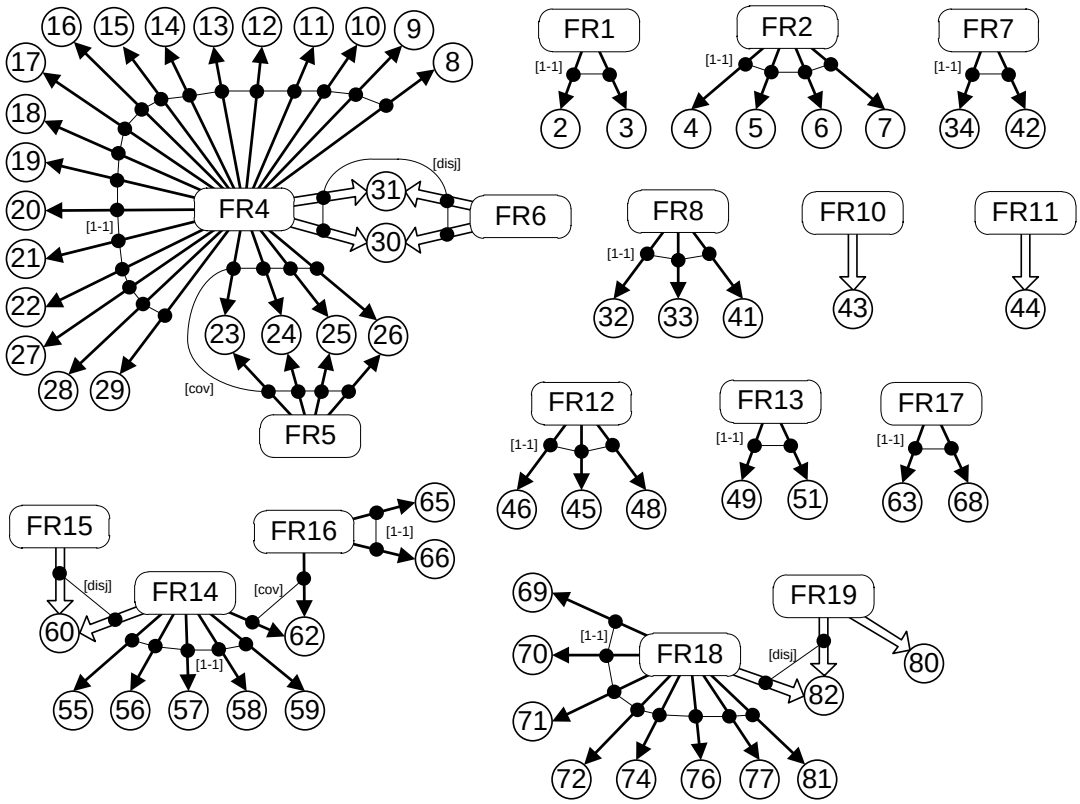


Figure 5.3. Mappings between functional features and functional requirements illustrated by arrow predicates

The mapping between the functional requirements FR18 and FR 19 and the functional feature 82 is of type Many-to-One and the disjoint (component) predicate is used (the same mapping type is used also between other functional requirements and features). Disjoint predicate means that the related requirements together completely specify the functional feature and do not overlap each other.

The Many-to-One mapping type can be reflected also with the covering predicate which is used if a set of functional requirements overlap the specification of what will be implemented in accordance with one functional feature. Example of covering predicate is mapping between functional requirements FR14 and FR16 and functional feature 62.

The functional requirements that define new functionality are FR3, FR9 and FR10. In order to implement this new functionality it is needed to refine initial TFM by adding missing functional features to it. The new functional features are given below in Table 5.2. Functional requirements map onto the new functional features as follows: **FR3** = {86}; **FR9** = {87, 88}; and **FR10** = { 43, 89 }.

Table 5.2

Additional functional features of laundry business system

ID	Object action (A)	Precondition (PrCond)	Entity (E)	
86.	Registering client e-mail address	If client has e-mail address	Clerk	Inner
87.	Preparing e-mail that linen has been washed	If e-mail address of client has been registered	Clerk	Inner
88.	Sending e-mail that linen has been washed		Clerk	Inner
89.	Submission of received e-mail letter	If client wants to take back washed linen basing on e-mail letter	Client	Inner

The second acquirement of functional requirements validation is the ability to find missing requirements. From the mappings it is clearly visible that functional features 35, 36, 37, 38, 39, 40, 52, 78, and 84 has no corresponding functional requirements. Since the functional features 35, 36, 37, 38, and 39 are part of second order sub-cycle (the sub-cycle in which a dry-cleaning is ordered from collaboration partner) and functional feature 40 is part of the main functioning cycle the missing requirements are vital for the system functioning (missing functional requirements should be defined or existing ones should be extended). If these functional features were not a part of cycles that are important to system's functioning, then they can be removed from the TFM of laundry system and thus not implemented in software system. The new functional requirements are given below in Table 5.3.

Table 5.3

Missing functional requirements of the laundry software system

ID	Requirement
FR20	System should provide functionality for placing dry-cleaning orders at collaboration partners, including linen registration form updating.
FR21	After linen has been washed it should be possible to update linen registration form with mark that linen is washed and can be returned to the client.
FR22	When a payment of provided washing services is received from client, the system should ensure

ID	Requirement
	functionality that supports payment to the collaboration partner for the washing services used.
FR23	The system should ensure bank transfer preparation of the bill for determents order.
FR24	The system should collect monthly costs for the servicing of washing machines.

The new functional requirements map onto functional features as follows: **FR20** = {35, 36, 37, 38, and 39}, **FR21** = {40}, **FR22** = {52}, **FR23** = {78}, and **FR24** = {84}.

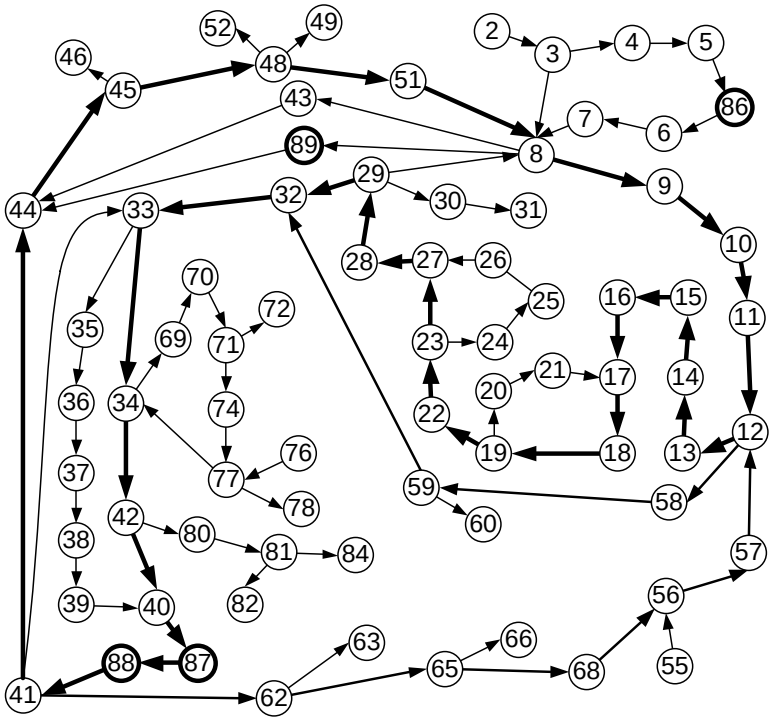


Figure 5.4. Refined Topological functioning model of laundry business system after functional requirements validation

After identification of missing functional features it is needed to update informal description of system functioning and to update the set of inner functional features N and set of external functional features M. After updating sets N and M it is needed to renew neighborhood of each element of the set N. Following is the update of neighborhood of each element of the set N which is affected by the new functional features. Refined TFM of laundry functioning is given in Figure 5.4 (the new functional features are highlighted with boldest vertices).

The summary of TFM refinement is as follows: four new functional features were added basing on the functional requirements, and five missing functional requirements were identified.

5.1.3. Behavior Analysis and Design

System behavior of laundry functioning is analyzed and represented by using Sequence diagrams and Interaction overview diagram. According to the TopUML modeling, functional features should be grouped into sets in order to set the scope for each sequence diagram. The primary source for scope identification in this case is system goals. The mappings between system goals and functional features are given in Table 5.4 together with the names of underlying Sequence diagrams.

Table 5.4

Mappings between system goals, Sequence diagrams, and functional features of laundry functioning

Goal	Sequence diagram name	Corresponding functional features
SG1	Checking and registering clients	2, 3, 4, 5, 6, 7, 8, and 86
SG2	Registering client linen for washing	9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, and 32
SG3	Registering and receiving linen from collaboration partners	12, 32, 55, 56, 57, 58, 59, and 60
SG4	Fulfilling registered linen washing orders	33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 62, 87, 88
SG5	Returning linen to client and receiving payment	43, 44, 45, 46, 48, 49, 51, 52, and 89
SG6	Returning linen to partner and receiving payment	63, 65, 66, and 68
SG7	Monitoring and ordering supplies of determents	69, 70, 71, 72, 74, 76, 77, and 78
SG8	Monitoring the usage of washing machines and ordering servicing for them	80, 81, 82, and 84

In accordance with the system goals a set of eight Sequence diagrams has been developed to represent the behavior of laundry functioning. All sequence diagrams are given in Appendix 8 on page 201 while the relations between sequence diagrams are reflected with Interaction overview diagram as given in Figure 5.5. Interaction overview diagram shows the workflow and messaging design together in one diagram (in fact, it is a combination of Activity and Sequence diagram elements); it is developed by merging created Activity diagrams and Sequence diagrams. While the first one gives the information about control flow, the latter shows objects and messaging between them.

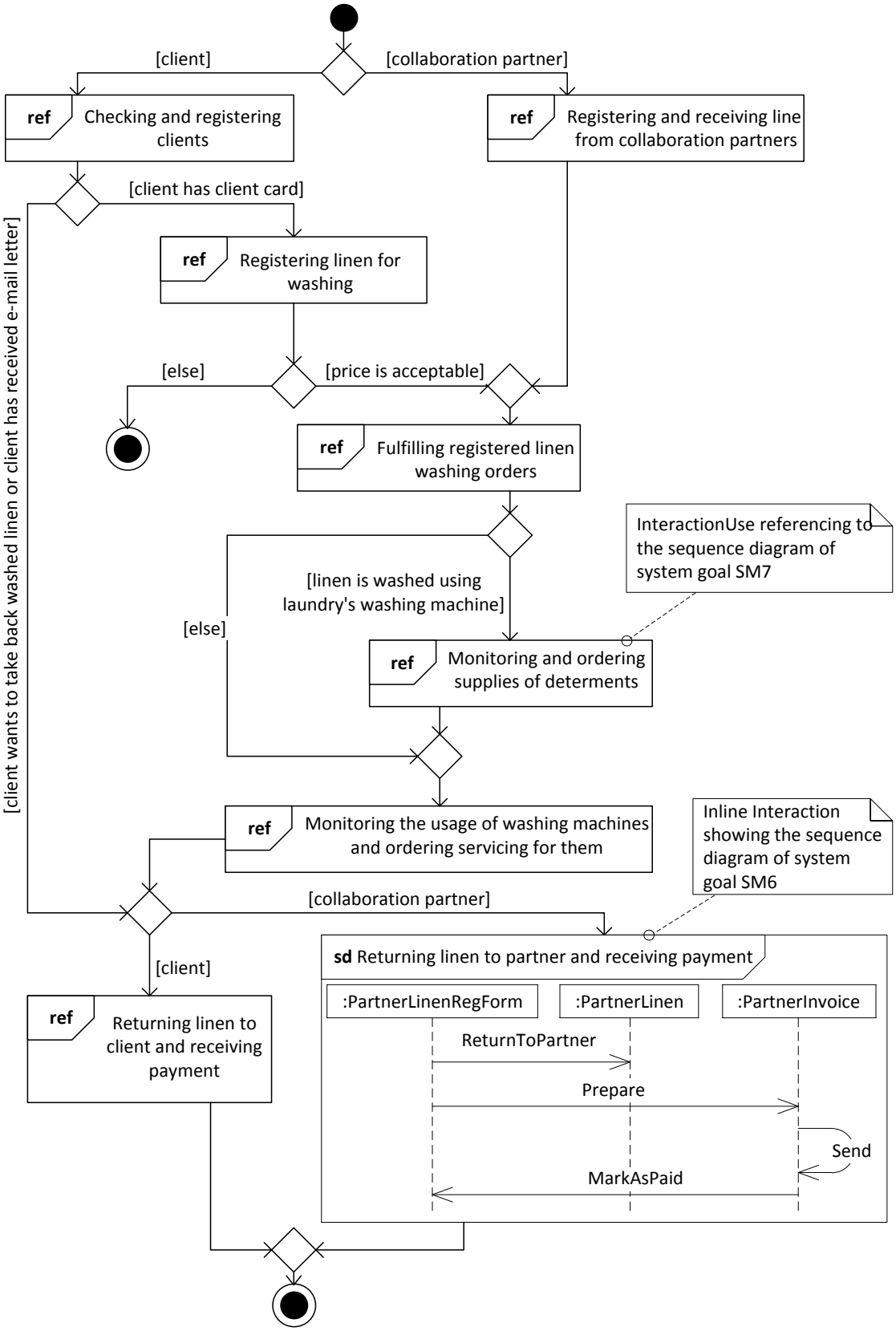


Figure 5.5. Interaction overview diagram created in accordance with system goals and developed TFM of laundry functioning

Analysis of Interaction overview diagram allows identifying and understanding relationships between various Sequence diagrams. Interaction diagram given in Figure 5.5 shows that there is a decision making in the context of laundry functioning on which Sequence diagram will be executed at which conditions. If there are only Sequence diagrams without an Interaction diagram, then the relationships between the first ones are underestimated.

5.1.4. Structure Analysis and Design

The main goal of domain model analysis and design is to develop a Topological class diagram which contains classes together with their attributes, operations, and topological relationships between them. To identify classes and assign the right responsibilities a TFM is used. Thus, the creation of domain model is made more formal in comparison to other UML modeling driven approaches (except the TFMfMDA approach which involves identification of conceptual classes, i.e., classes with no responsibilities). This section shows the praxis of developing such structure model – at first transforming TFM into Communication diagram and then into Topological class diagram.

5.1.4.1. Analysis of Objects and their Communication

Analysis of objects and their communication within TopUML modeling consists of TFM transformation into Communication diagram. To obtain a Communication diagram of laundry functioning, it is necessary to detail each functional feature of the TFM to a level where it uses only one type of objects as described in section 4.3.1 on page 118. If some of functional feature contains more than one type of object then it is needed to refine it to the level where one functional feature uses only one type of objects. The first step in transformation is to merge functional features with objects of the same type in one lifeline. While merging functional features into lifelines the relationships with other lifelines should be retained (if there is more than one topological relationship then only one link is added between lifelines). The message that is sent to a lifeline is an operation attribute of the functional feature. Communication diagram showing arrival of person and submission of linen for washing is given in Figure 5.6 on next page.

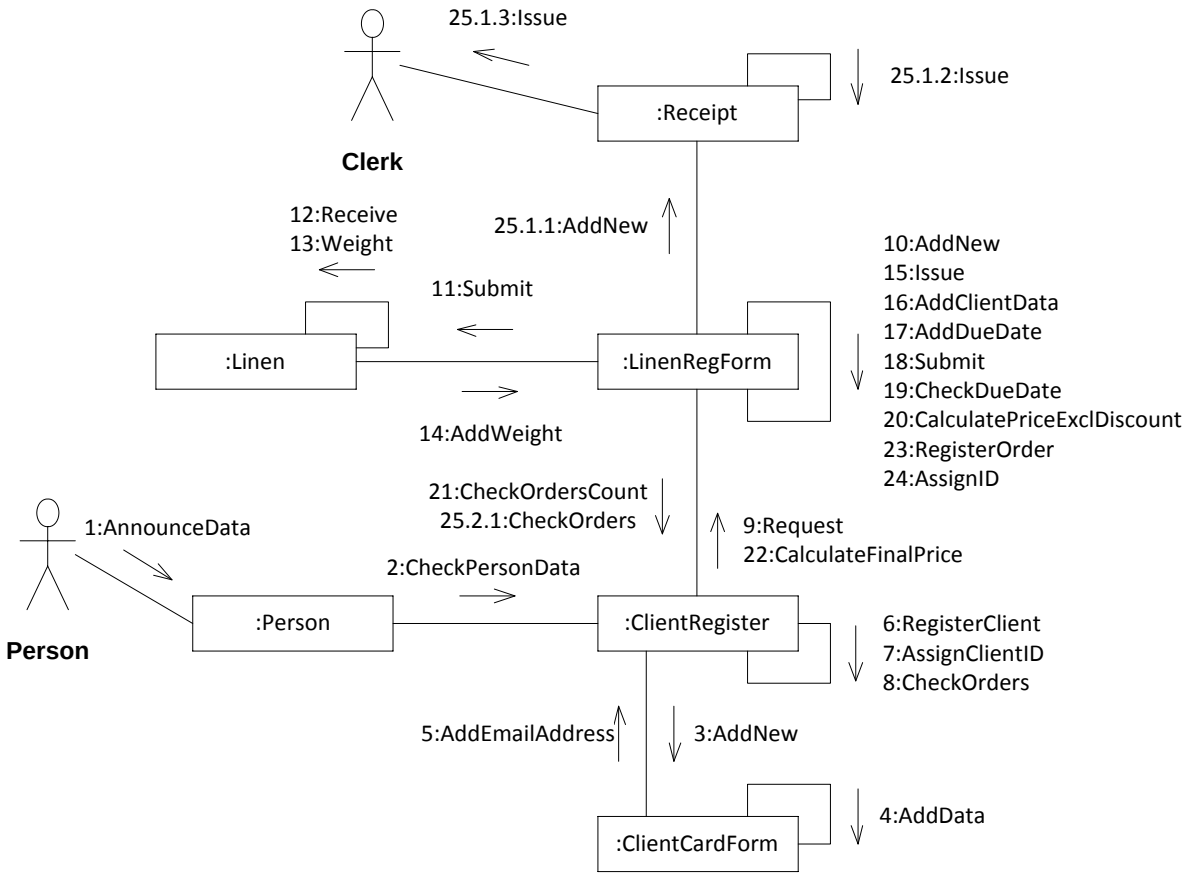


Figure 5.6. Communication diagram: arrival of person and submission of linen for washing

5.1.4.2. Domain Model Development by Means of Topological Class Diagram

Topological class diagram is constructed after creation of Communication diagram by applying transformations on it. The developed topological class diagram of laundry system can be seen in Figure 5.7 where the classes are shown with hidden attributes and operations. With the boldest lines in developed topological class diagram is maintained the main functional cycle which is defined by the expert within the constructed TFM [93]. This reflects the idea proposed in [94] and [100] that the holistic domain representation by the means of the TFM enables identification of all necessary domain concepts and, even, enables to define their necessity for a successful implementation of the system.

The Topological class diagram of laundry system showing attributes and operations is given in Appendix 9 on page 206. The operations are obtained during TFM transformation to Communication diagram and the attributes are added from TFM while transforming Communication diagram into Topological class diagram. The responsibilities of classes are assigned as operations. Thus by using TFM in software development the classes are identified and responsibilities are assigned directly from the problem domain.

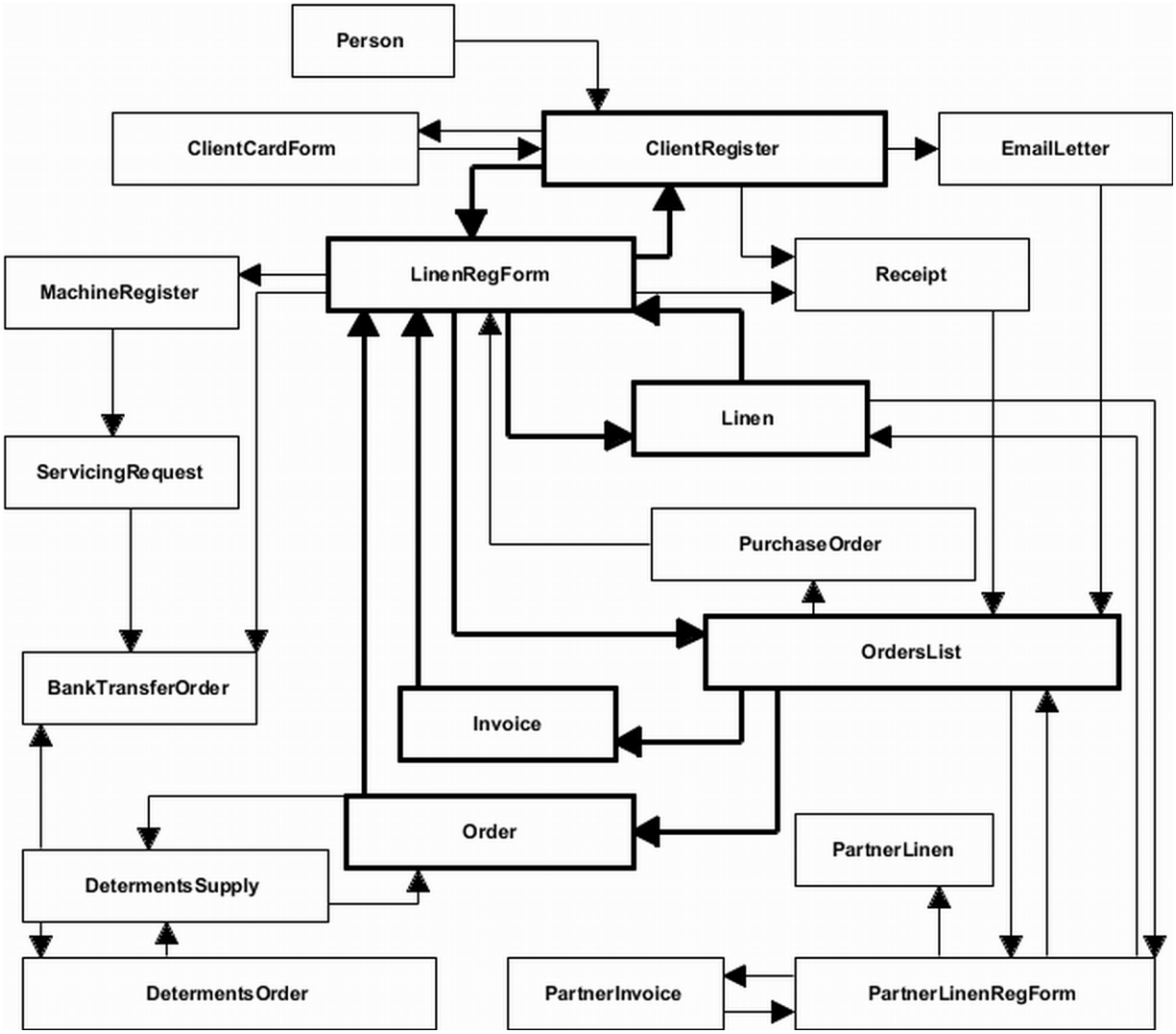


Figure 5.7. Topological class diagram representing laundry system

Above given Topological class diagram can be considered as initial because it contains classes and topological relations between them and it is at high abstraction level. By reviewing and refining initial class diagram associations, generalizations, dependencies, and other relationships defined in UML are added. In this way the abstraction level of diagram is lowered and structural relations between classes are added. The refinement of initial Topological class diagram is covered within the case study explored in next subsection.

5.2. Enterprise Data Synchronization System Development

The main idea of this section is to explore a case study of applying TopUML modeling in real software project. In this software project case study a service application is

developed which application is aimed to synchronize enterprise employee data. Synchronization is done by taking data from multiple data sources and placing in one central data storage. The case study covers full software development life cycle (the software now is at maintenance phase, the case study covers implementation phase). The software is developed in Lattelecom Technology Ltd Software Development Department (see acknowledgement in Appendix 10 on page 207). TopUML modeling case study includes development of TopUML diagrams. These diagrams are as follows and they are given in the subsections of this section:

- *Initial and refined TFM* in accordance with informal system description and functional requirements,
- *Topological use case diagram* defined in accordance with developed TFM and mappings between functional features of TFM and determined functional requirements,
- *Sequence diagrams* and *Activity diagram* in accordance with TFM and Use cases,
- *Communication diagram* obtained by performing transformations on TFM,
- *Initial and refined Topological class diagram* in accordance with Communication diagram and TFM, and
- *State diagram* within the case study is prepared for the main object of data synchronization system by applying transformations on TFM. The main object is determined by its membership to the main functioning cycle.

The exploration of enterprise data synchronization system development case study by using TopUML modeling is published in [29].

5.2.1. Specification of Enterprise Data Synchronization System

Specification of data synchronization system includes following artifacts:

1. Informal description of data synchronization functioning, where nouns are denoted by *italic*, verbs are denoted by **bold**, and action pre- and post- conditions are underlined, and
2. Functional and nonfunctional requirements defined for data synchronization software system.

The specification artifacts are available in Appendix 11 on page 208.

5.2.2. Problem Domain Functioning Analysis

The development of TFM representing enterprise data synchronization system functioning involves the following activities:

1. Topological space development,
2. Initial TFM development, and
3. TFM refinement (including the identification of logical relations).

All the activities in detail are shown in the subsequent subsections.

5.2.2.1. Topological Space Development

Within enterprise data synchronization software system development project has been defined 30 functional features (see Appendix 12 on page 210). These functional features are identified during the analysis of enterprise data synchronization system – the informal description of it. After definition of functional features the topology Θ (cause-and-effect relationships) are identified between those functional features. The identified cause-and-effect relations between the defined functional features are illustrated by the means of the topological space (see Figure 5.8).

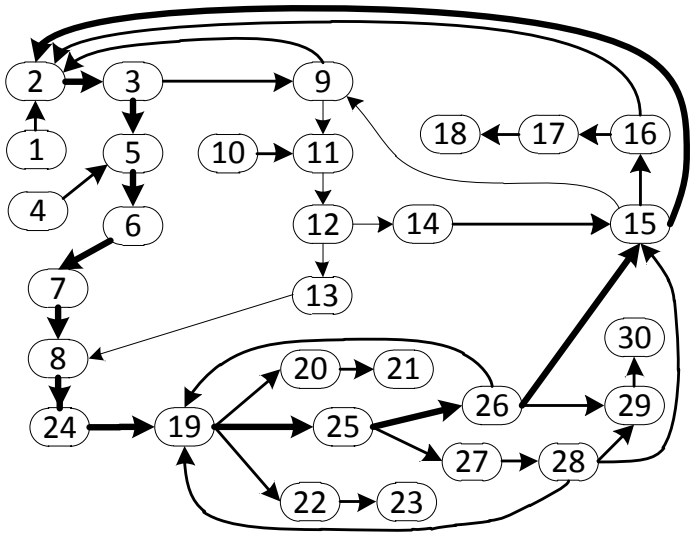


Figure 5.8. Topological space of enterprise data synchronization system

In the Figure 5.8 is clearly visible that cause-and-effect relations form functioning cycles. In enterprise data synchronization system case study the main functioning cycle

represents getting data from source data base and import files and editing those data in target data base, and is as follows: 2-3-5-6-7-8-24-19-25-26-15-2.

5.2.2.2. Initial Topological Functioning Model

According to the equation (2) on page 87 all the identified functional features given in Table 5.1 are split in two sets:

- N = {2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 19, 24, 25, 26, 27, 28}; and
- M = {1, 4, 10, 17, 18, 20, 21, 22, 23, 29, 30}.

In order to get all of the system's functionality – the set X – the closuring operation (see equation (3) on page 87) is applied over the set N (detailed example of applying closuring operation is given in section 5.1.2.2 on page 142). The obtained set X (*the TFM*) after applying closuring operation is as follows: X={2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 19, 20, 22, 24, 25, 26, 27, 28, 29}. The resulting graph is given in Figure 5.9.

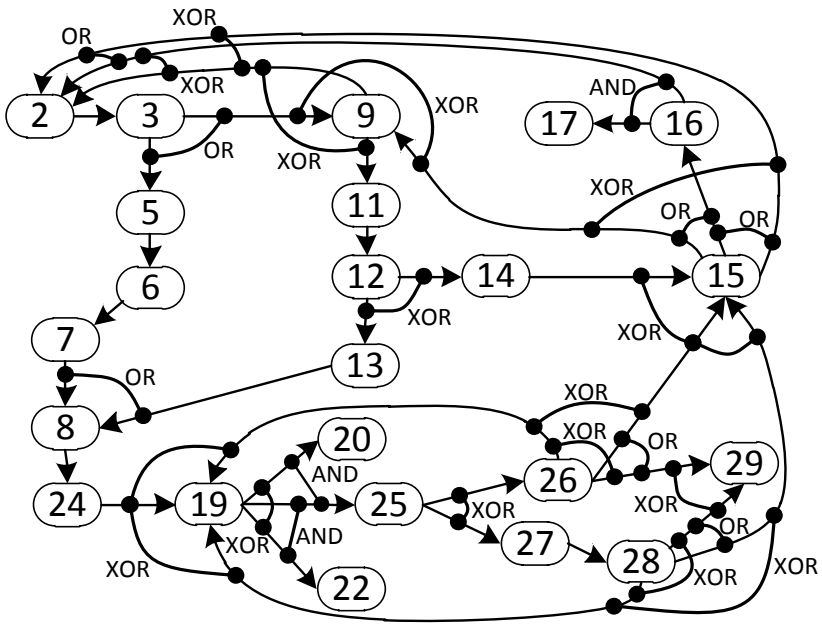


Figure 5.9. TFM representing enterprise data synchronization system functioning and logical relations between cause-and-effect relationships

The TFM as given in Figure 5.9 shows logical relations between cause-and-effect relationships. The logical relations are divided into two sets: L_{out} (logical relations between topological relationships which are outgoing from functional feature) and L_{in} (logical relations between topological relationships which are incoming into functional feature); they are identified by using algorithm described in section 4.1.2 on page 103.

To better understand identification of logical relations a small fragment of TFM given in Figure 5.9 is used consisting of four functional features: 19, 20, 22, and 25. The functional feature 20 has a precondition C_1 ("*If data from the particular row exists*") and functional feature 22 has a precondition C_2 ("*If data from the particular row does not exist*") while functional feature 25 has no preconditions as given in Appendix 12 on page 210. The relation between preconditions C_1 and C_2 is as follows: $C_1 = \neg C_2$; thus indicating that between the arcs that are outgoing from functional feature 19 to functional features 20 and 22 (19→20 and 19→22) the logical relation with type exclusive disjunction (XOR) exist. Since functional feature 25 has no preconditions a logical relations with type conjunction (AND) is added between topological relationship 19→20 and 19→25, and 19→22 and 19→25.

5.2.2.3. Refining Topological Functioning Model

The result of requirements validation is that both TFM and functional requirements are checked. In order to validate functional requirements and the constructed TFM, mappings between functional requirements and functional features should be established. Since in this case study Use cases are used to model requirements, the set of mappings of functional requirements include both functional features and functional requirements. The established mappings are as follows:

FR1 = {FR1/[1-6]};
FR1/1 = {2, 3};
FR1/2 = {5, 6, 7, FR1/5};
FR1/3={9, 11, 12, 13, 14, FR1/[4-5]};
FR1/4 = {15, 16, 17};
FR1/5 = {8, 24, 19, 20, 22, FR1/6}; and
FR1/6 = {25, 26, 27, 28, 29}.

The identified mappings shows that there are no missing requirements and no missing functional features thus the refined TFM is equal to the initial TFM.

5.2.3. Behavior Analysis and Design

Every requirement is modeled with use case:

- **FR1** = "Employee data synchronization",
- **FR1/1** = "Obtaining configuration information",
- **FR1/2** = "Obtaining data from source data base",
- **FR1/3** = "Obtaining data from import files",
- **FR1/4** = "Logging faulty import file",

- **FR1/5** = “Importing data in target data base”, and
- **FR1/6** = “Logging import status”.

According to the mappings between functional features and requirements and logical relations in TFM the *«include»* and *«extend»* relationships are automatically established between Use cases. Since actors in Use case diagram show interaction between system and external systems or entities, they are obtained from topological space – actors are *entities (E)* from functional features and the set of actors are identified by equation (8) given on page 113. Topological relation between one functional feature belonging to set E and the other to set X defines relation between use case and actor (since all use cases are mapped to functional features). The developed Topological use case diagram is given in Figure 5.10.

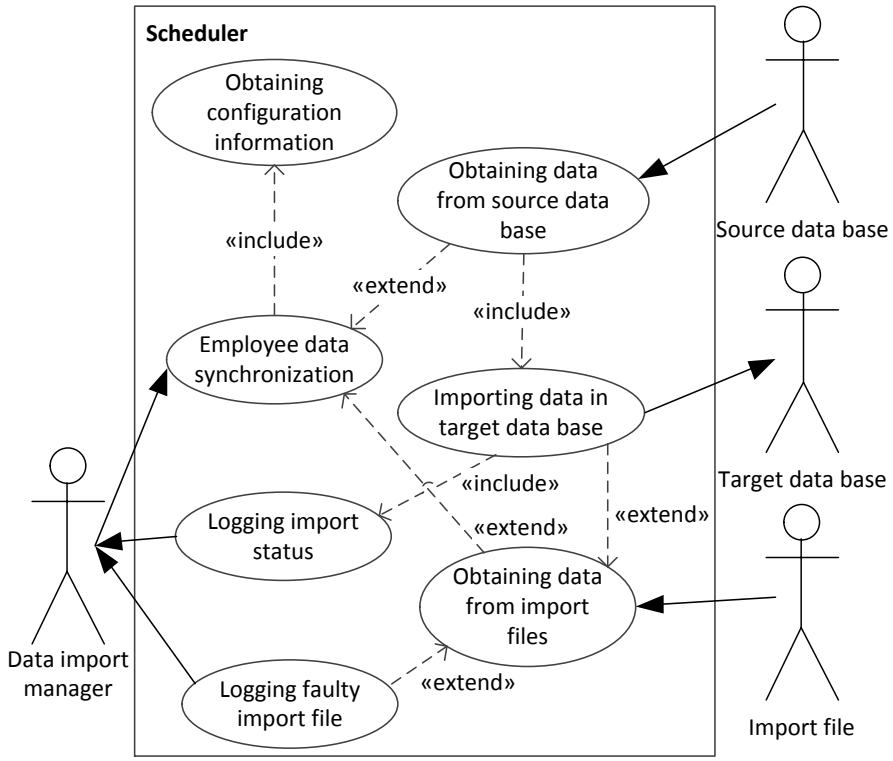


Figure 5.10. Use case diagram of enterprise data synchronization system

5.2.3.1. Messages of Objects and their Sequence Analysis and Design

Since in this case study use cases are used to model requirements, the use cases define the number and the scope of sequence diagrams. The scope of sequence diagrams defines a set of functional features which are included in each sequence diagram. A total set of seven sequence diagrams is created. Sequence diagram for use case “Importing data in target data

base" (which reflects functional requirement FR1/5) is given in Figure 5.11. As FR1/5 mappings includes also functional requirement FR1/6, the corresponding Sequence diagram contains *ref* interaction use to Sequence diagram "Logging import status".

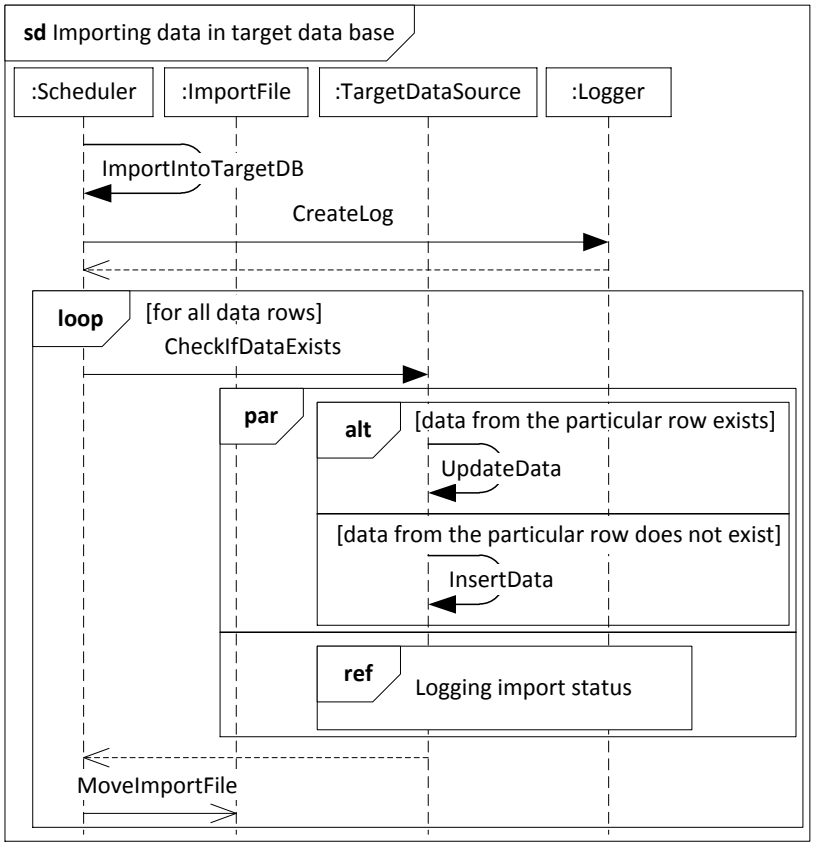


Figure 5.11. Sequence diagram "Importing data in target data base"

5.2.3.2. Workflow Analysis and Design

Workflows within system are analyzed and designed by using the Activity diagram. The scope and count of activity diagrams are denoted by the Use cases and their mappings with functional features (i.e., equal to Sequence diagrams described in previous subsection). According to the defined Use cases and established mapping, a total set of seven Activity diagrams is created. Activity diagram representing the use case "Importing data in target data base" is given in Figure 5.12. As FR1/5 mappings includes also functional requirement FR1/6, the corresponding sequence diagram contains *ref* interaction use to Activity diagram "Logging import status".

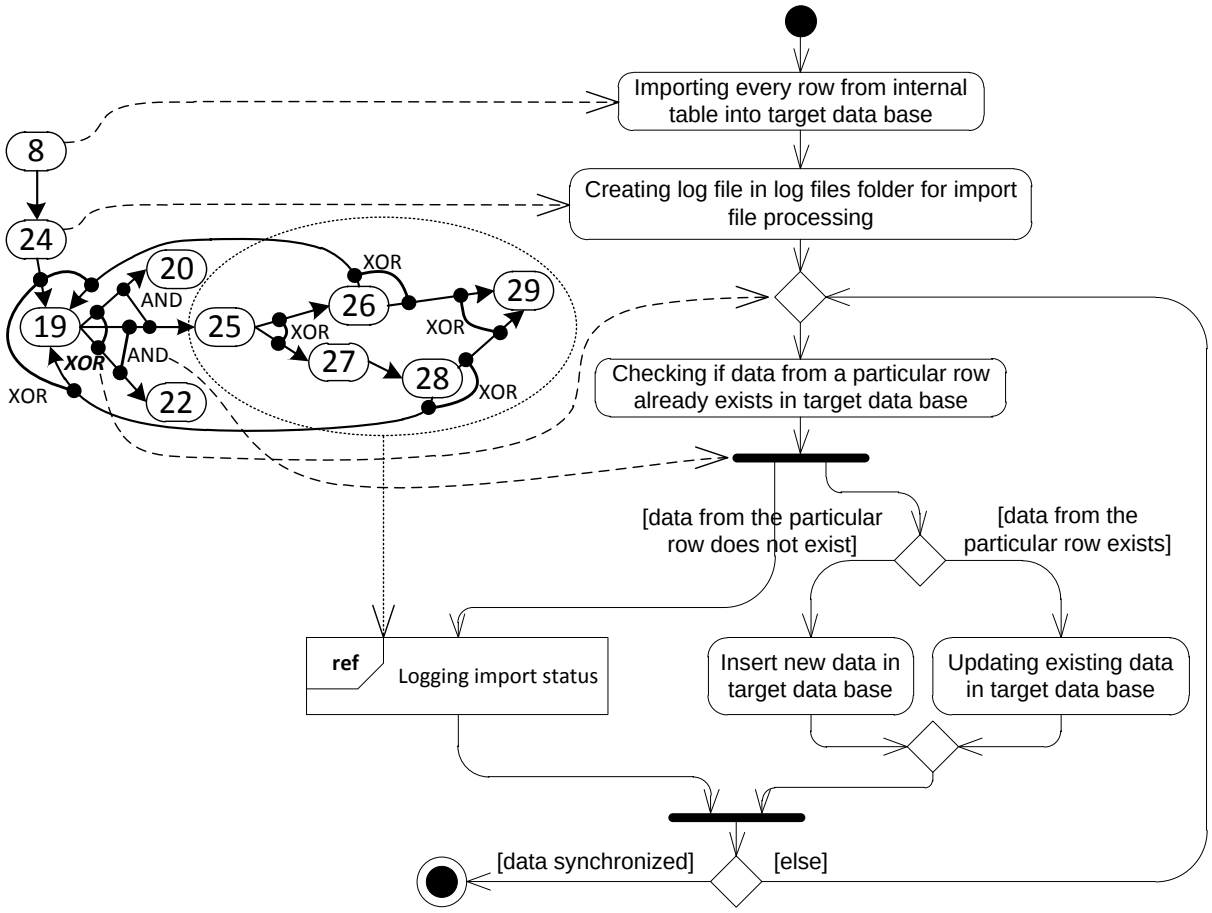


Figure 5.12. Part of TFM representing functioning of enterprise data synchronization system and activity diagram representing workflow of Use case "Importing data in target data base"

5.2.4. Structure Analysis and Design

The main goal of domain model analysis and design is to develop a Topological class diagram which contains classes together with their attributes and responsibilities. To identify classes and assign the right responsibility to each one of them a TFM is used. This section shows the transformation of TFM into Communication diagram and Communication diagram into Topological class diagram (together with refinement of it).

5.2.4.1. Analysis of Objects and their Communication

The next step within TopUML modeling approach is development of Communication diagram by transforming TFM of enterprise data synchronization system functioning. To obtain a Communication diagram, it is necessary to detail each functional feature of the TFM

to a level where it uses only one type of objects. Developed Communication diagram representing data synchronization with source data base is given in Figure 5.13.

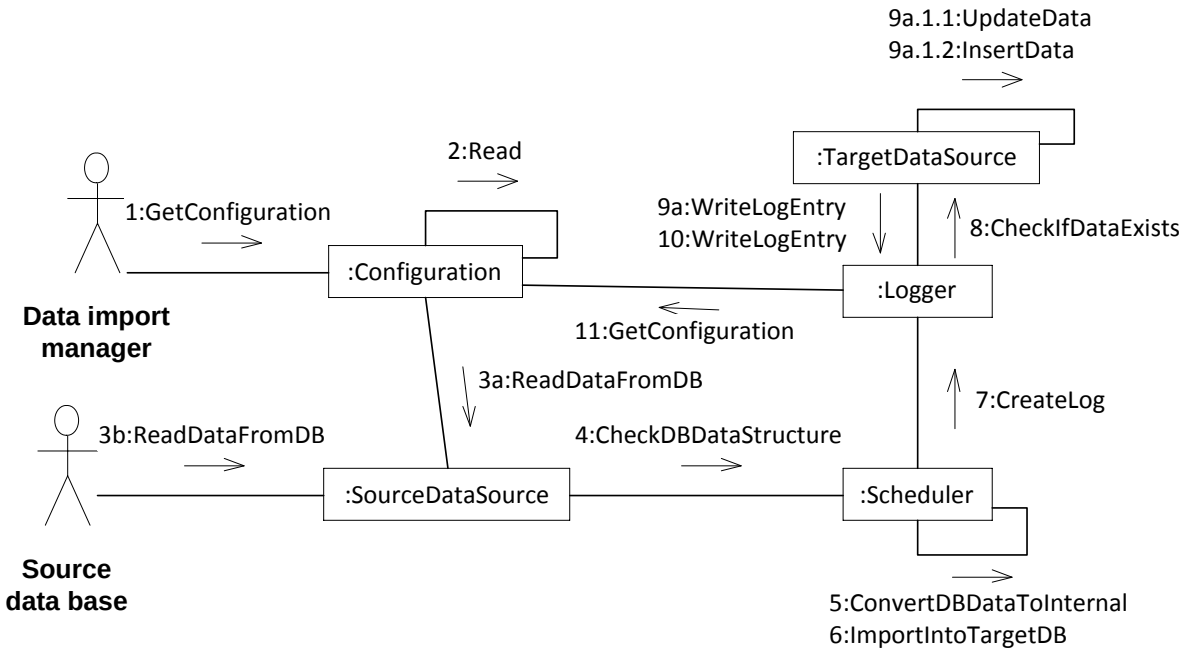


Figure 5.13. Communication diagram representing data synchronization with source data base

5.2.4.2. Domain Model Development by Means of Topological Class Diagram

Topological class diagram is constructed after creation of Communication diagram by applying transformation on it and TFM. The resulting Topological class diagram after transforming Communication diagram and TFM is considered as initial since it contains classes (with attributes and responsibilities) and topological relations between them. The operations are obtained during TFM transformation to Communication diagram and the attributes are added from TFM while transforming Communication diagram into Topological class diagram. Responsibilities of classes are assigned as operations.

To add other relationship types (e.g., associations, generalizations, and dependencies), required and provided interfaces in accordance to inputs and outputs of TFM to the initial Topological class diagram, the initial Topological class diagram should be refined in accordance to the steps given in section 4.3.3 on page 122.

The refined topological class diagram of enterprise data synchronization system is given in Figure 5.14, where with bolder topological relationships is denoted main functioning cycle.

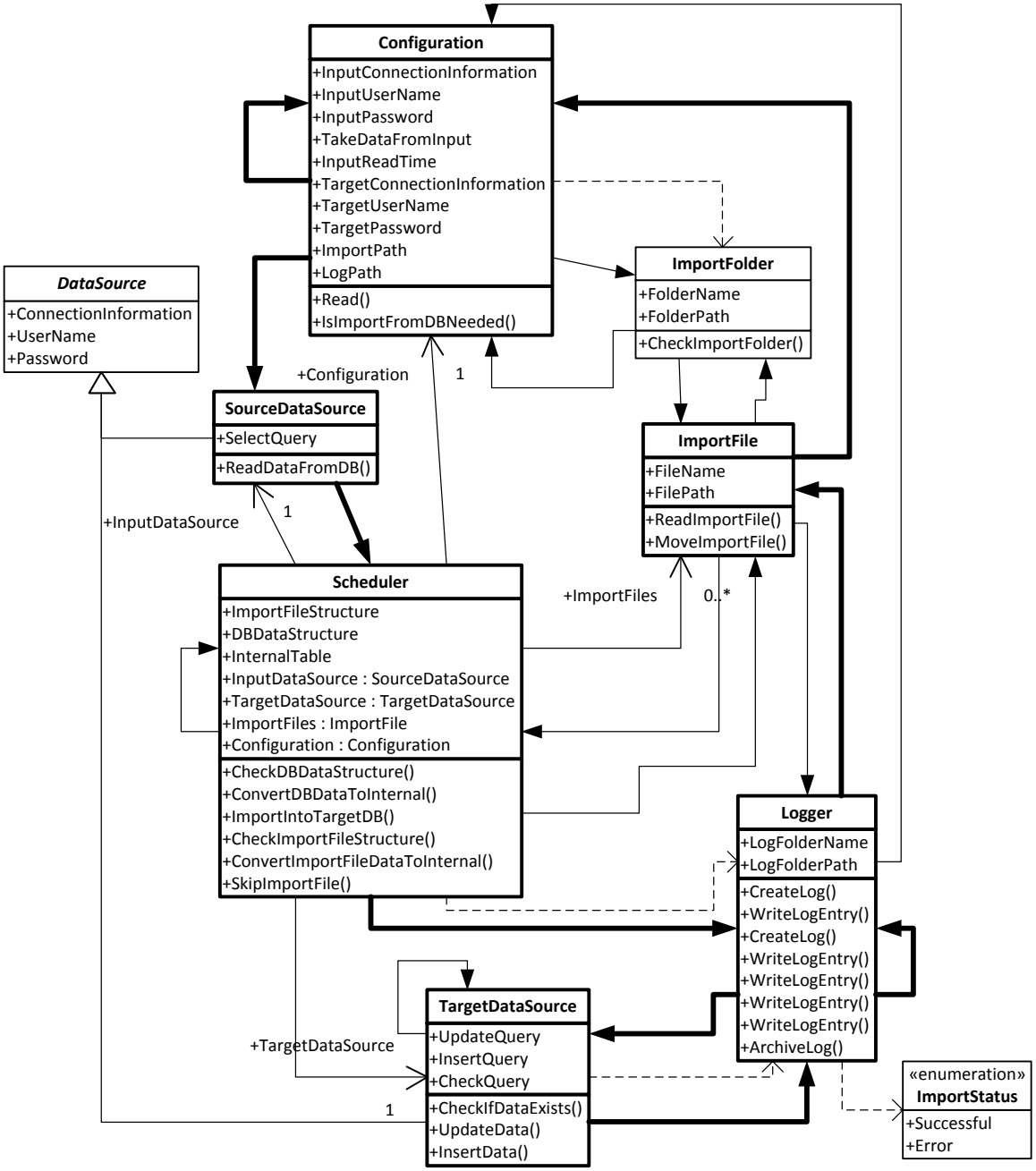


Figure 5.14. Topological class diagram of enterprise data synchronization system

5.2.5. State Change and Transition Analysis

State changes and transitions within system are analyzed and designed by using the State diagram. The scope and count of activity diagrams is denoted by the main classes in planned system (i.e., classes belonging to the main functioning cycle of Topological class diagram). This section discusses state diagram created for one of the main object – "*Scheduler*". Table 5.5 on next page contains specification of functional features that are used to define state diagram for object "*Scheduler*".

Table 5.5

Functional features specifying functioning of object "*Scheduler*"

ID	Object action	Precondition	New state
5	Reading all data from source data base	If import should be performed from source data base	Reading data
6	Checking if read data structure is according to specification		Checking data
7	Putting the read data into temporal internal table	If data structure is according to specification	Importing
9	Checking import folder		Reading data
12	Checking if import file data structure is according to specification		Checking data
13	Converting the read data from import file into temporal internal table	If import file structure is according to specification	Importing
15	Moving import file to processed files folder		Completing import
19	Checking if data from a particular row already exists in target data base		Importing
25	Logging data row from temporal internal table		Logging status
29	Archiving log file	If data import is completed	Completing import

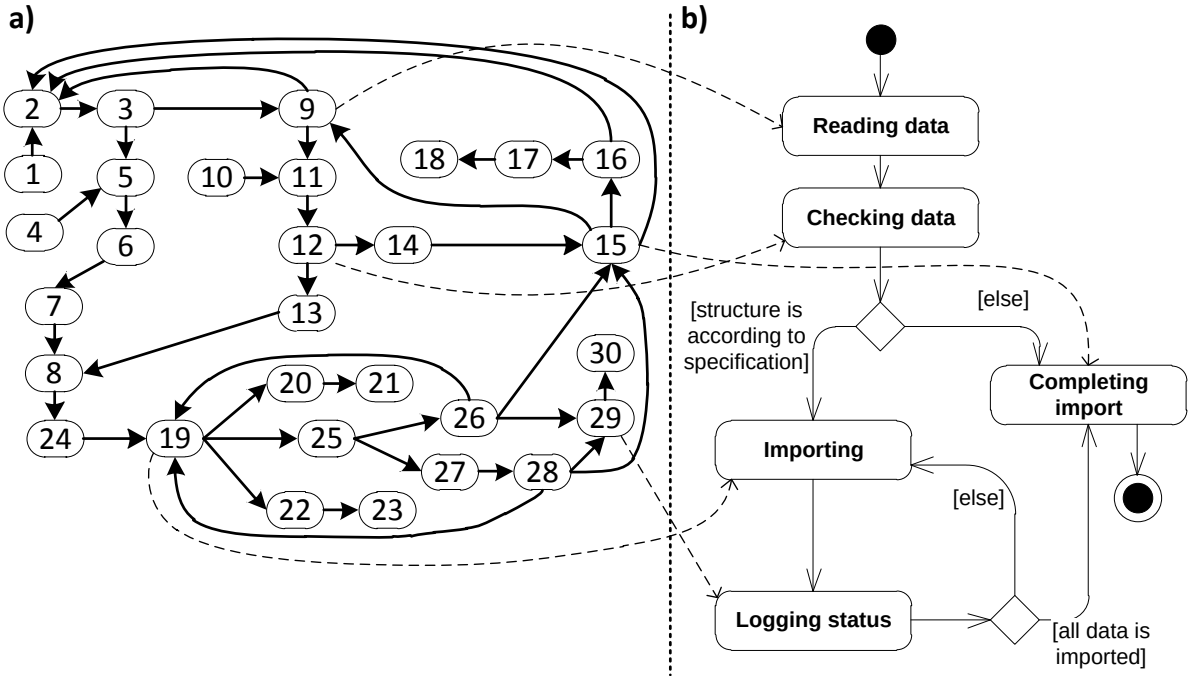


Figure 5.15. TFM of enterprise data synchronization system functioning (a) and State diagram for object "*Scheduler*" (b)

The specification of functional features shows that this object in total has five different states: 1) "*Reading data*", 2) "*Checking data*", 3) "*Importing*", 4) "*Logging status*", and 5) "*Completing import*". The resulting State diagram is given in Figure 5.15.

5.3. Empirical Evaluation of TopUML Profile and Modeling Method

The empirical evaluation of TopUML profile and modeling method is based on practical experiment with two expert groups in which a business support application for the Laundry problem domain is designed. Description of experimental software designing with experts is organized in following subsections:

1. Goal and process of empirical experiment – sets the goal of practical experiment and describes the experiment process showing the produced artifacts at each TopUML modeling step,
2. Participants – gives the descriptive information of experiment participants: education degree, self-evaluation of software modeling, designing and development, and experience in software development,
3. Results of experiment – summarizes the results and competency gained during the TopUML modeling experiment, including positive and negative aspects of modeling method and recommendations of participants.

The main result of practical experiments is publication of guidance manual "*Topological business systems modeling and software systems design*" [27], which includes both – the TopUML modeling theory and a practical example of applying it within software designing.

5.3.1. Goal and Process of Empirical Experiment

The main goal of the experiment is to design business support software by applying TopUML profile and modeling method with the experts of software development, to review the TopUML modeling usage in practice, and to collect feedback from the experiment participants. At the beginning of the experiment the participants are introduced with the problem domain and the specified software requirements. Laundry business system is selected as problem domain for both practical experiments. The informal description of problem domain is given in Appendix 4 on page 191 and the set of specified functional requirements is

given in Appendix 5 on page 194. The process of empirical experiment is divided into eight practical sessions as given in Table 5.6, where goal, tasks, and results (including developed artifacts) are described for each session.

Table 5.6

Practical sessions of TopUML modeling experiment

No.	Goal	Tasks	Results
1.	Evaluate participants knowledge, introduce TopUML modeling and problem domain	1. Fill out self-evaluation questionnaire (Appendix 13 on page 212) 2. Briefly describe TopUML modeling 3. Hand-out informal description of Laundry functioning 4. Define functional features	1. Evaluation of participants 2. Set of functional features
2.	Develop topological space of Laundry functioning	1. Define topological space 2. Identify cause-and-effect (topological) relationships between functional features	Topological space of Laundry functioning
3.	Develop TFM	1. Define TFM 2. Perform closure operation of previously developed topological space	TFM of Laundry functioning
4.	Validate functional requirements and developed TFM	1. Hand-out specification of functional requirements and system goals 2. Define mappings between functional features and functional requirements 3. Refine TFM and functional requirements	1. Mappings between functional features and functional requirements 2. Refined TFM 3. Refined functional requirements
5.	Transform refined TFM into problem domain objects graph	1. Define problem domain objects graph 2. Transform TFM into problem domain objects graph	Problem domain objects graph
6.	Develop Sequence diagrams	1. Define transformations between TFM and Sequence diagram 2. Transform TFM into a set of Sequence diagrams	Set of Sequence diagrams (one for each system goal)
7.	Develop Topological class diagram	1. Define Topological class diagram 2. Transform problem domain objects graph into Topological class diagram	Topological class diagram
8.	Review the TopUML modeling and experiment	1. Review experiment process 2. Review TopUML profile and modeling method 3. Summarize reviews in report	Report of experiment and TopUML modeling method

The experiments do not cover full TopUML modeling method because of the time limitation and it is partly outside the TopUML modeling while the experiment uses Problem domain object graph instead of Communication diagram. The application of Problem domain objects graph is shown in [103].

5.3.2. Participants

Total number of participants in empirical experiment is 32 (the participants are split in two separate groups). The first group is made with 17 participants holding master degree in computer science while the second group is made with 15 participants holding bachelor degree in computer science. Average age of participants is 26 years and average work experience in software development is 3.7 years.

At the beginning of each practical experiment the participants are asked to fill out self-evaluation questionnaire with questions grouped into following sections (questionnaire form is given in Appendix 13 on page 212): Modeling languages and notations, UML diagrams (8 most used diagrams according to [24]), Software development lifecycles, Programming languages, and Modeling tools.

Self-evaluation results are given in Figure 5.16 and Figure 5.17, where the first one shows knowledge evaluation of modeling languages and notations and the latter one – UML diagrams.

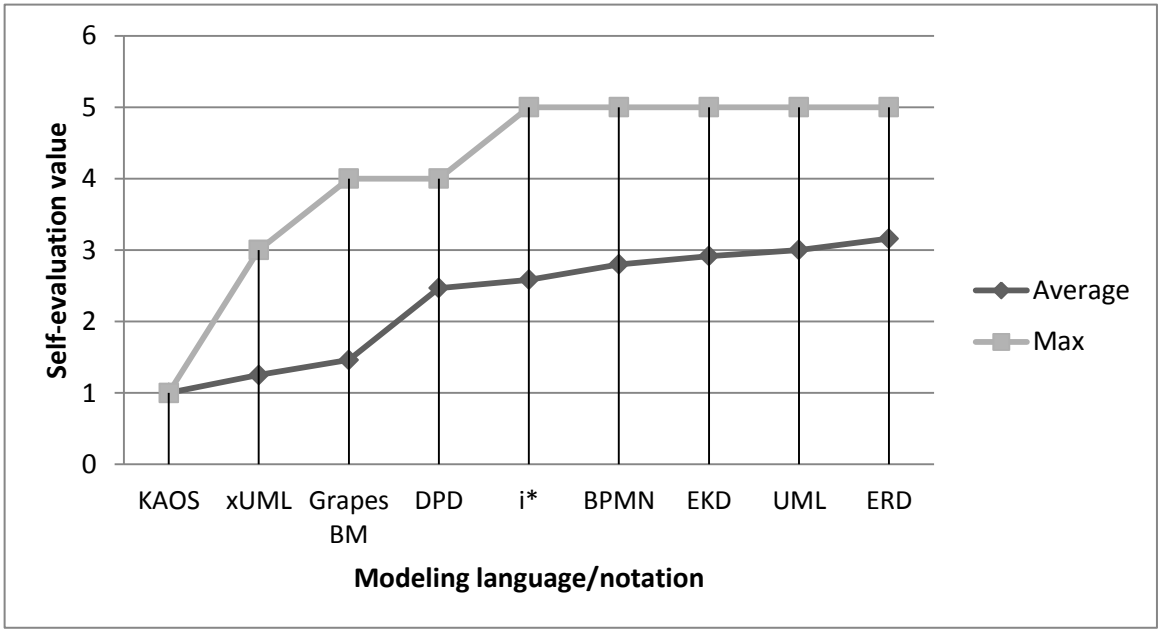


Figure 5.16. Participants' self-evaluation results of modeling language/notation knowledge

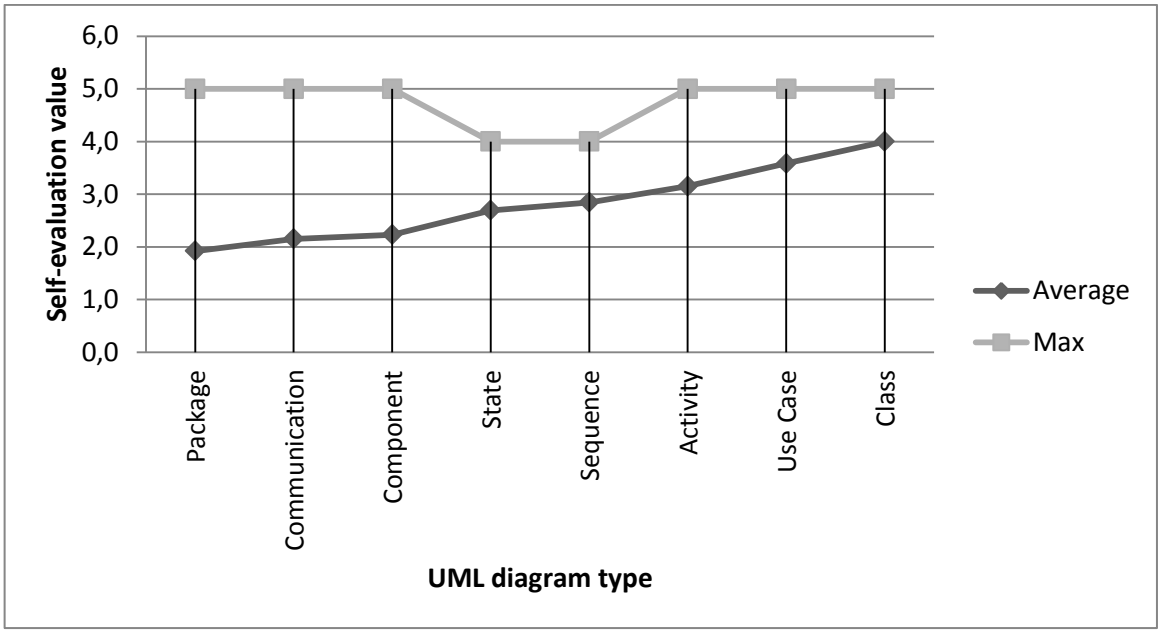


Figure 5.17. Participants' self-evaluation results of UML diagram type knowledge

5.3.3. Results of Experiment

The results of practical experiments allow empirically evaluate application of TopUML profile and modeling method within software designing. Participants and their results during development process are evaluated by using quantitative and qualitative measurements. The quantitative measurement involves evaluation of participant results by evaluating all participants with the best results obtained in each practical session. For example, the participant defining the largest set of functional features gets the largest amount of points – 10 (in the scale 0 to 10 where 0 is the smallest and 10 is the largest value). The qualitative measurement is the subjective evaluation of participant results set by the head of experiment. For example, the quality of developed functional features varies from participant from participant. While the largest part of participants had no difficulties defining functional features, there are some participants defining functional features containing more than one atomic business action. The average value of participants' evaluation in both experiments is 8 (5 being the smallest value and 10 the largest value). Distribution of evaluation values in the scale 5 to 10 is given in Figure 5.18. The TopUML modeling application results gained by participants holding bachelor degree and participants holding master degree are almost the same thus showing that it can be applied by modelers having different amount of preliminary knowledge.

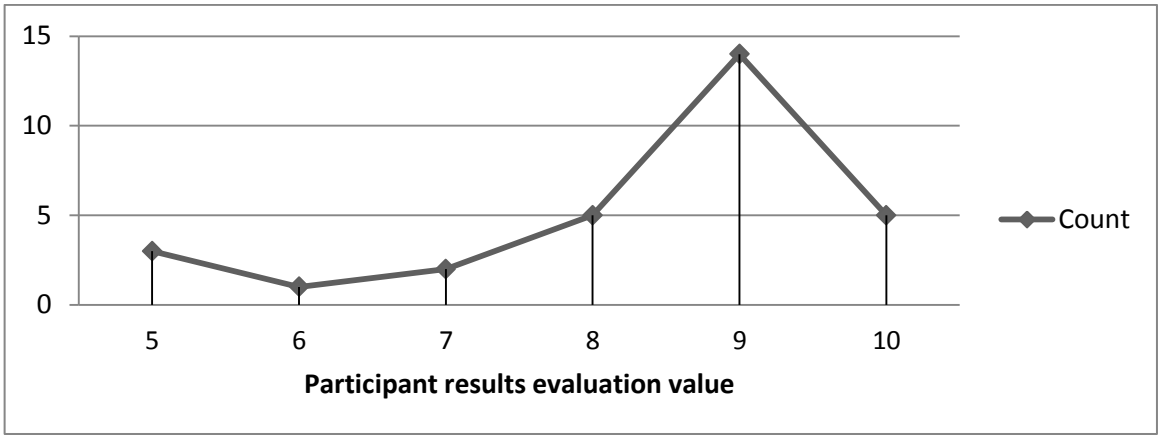


Figure 5.18. Evaluation of results produced by experiment participants

By summarizing the feedback from participants of both experiments the following benefits of using TopUML modeling can be outlined:

- Theoretical and scientific basis of the method,
- Formal nature of TopUML modeling (in contrast to the methods based on Use case modeling),
- Transformation from computation independent viewpoint to the platform independent viewpoint,
- Possibility to automate transformations between developed diagrams, and
- Reduced possibility of rewriting software code (due to the improper analysis of problem domain).

The reviews of performed experiments include following recommendations and limitations marked by participants:

- *Checking correctness of developed models* – the method should include mechanism of automatically checking the correctness and quality of the developed models. In fact, the first checking of developed TFM is the identification of logical relationships, mappings establishment between functional features and functional requirements, and identification of functional cycles, thus ensuring that TFM is constructed properly. The TFM can be in addition validated against problem domain ontology as suggested in [125]. TFM validation against errors in it ensures that the other models developed using TopUML modeling are in accordance with problem domain functioning.
- *Tool support* – since the experiments are performed without any TopUML tool support, participants outline that a tool supporting TopUML would improve the

usability of TopUML modeling in practice. Since the TopUML modeling describes formal transformations between different diagram types, the tool would improve also the transition process between diagrams. The transitions between diagrams could be made semi-automatically.

- *Advancements of TopUML modeling* – modeling method should allow developing models in different scenarios and in different order. This suggestion could be related to the development scenario presented in practical sessions since the TopUML modeling is a set of activities. Each activity can be performed in any order; the only constraint is the input parameters of each TopUML modeling activity.

5.4. Summary

The application and approbation of TopUML profile and modeling shows that the main goal of implementing new modeling language in the form of UML profile and its usage method has been reached – the artifacts produced during the design phase (e.g., classes, responsibilities of classes, use cases, and so on) are identified and defined in a strong accordance with the problem domain. Every artifact that is created can be traced from the problem domain. In fact the traceability works in both directions – from and to problem domain [8].

The first discussed software development project shows that the TopUML modeling can be applied in iterations where the first iteration draws a concrete border of the desired system while the sub-iterations are needed to lower the abstraction level of the developed models. The detailing of developed models should be performed by detailing functional features and the TFM of the problem domain functioning thus keeping the strong relevance of produced artifacts with the problem domain. The TFM is cogent model for detailing the specification of functioning because one of its topological characteristics is continuous mapping – more details can be added by not losing the existing specification. To ensure that the created models in software development project are detailed out enough for software implementation, a decision of experts can be used. The consensus of the created solution can be obtained by using voting mechanism as suggested in [44], where intermediate model is used to seam together requirements and the architecture of software.

The second discussed software development project (case study) shows that it can be enough to make only one iteration within TopUML modeling – developed models are detailed

enough for developers to be able implementing software. The enterprise data synchronization software project is chosen as case study for approbation of TopUML profile and modeling method since its scope, work estimation and delivery plan fits in the time frame of this thesis.

Summary of both the experiment and the case shows that TopUML modeling deals with the Computation independent viewpoint and the Platform independent viewpoint within MDA; the Platform specific viewpoint is not addressed. The TopUML modeling is concerned with the "*what*" and not "*how*". The "*what*" means what is what and what should do what (i.e., identification of classes and their responsibilities) while the "*how*" means how the responsibility will be implemented in a specific platform or technology (e.g., how the data will be saved in data base). The "how" part is dependent of the technology used to implement the software and even within one technology there can be a bunch of implementation options. For example, if the .NET framework is chosen to implement software, the communication with data base can be implemented in several ways [79].

The empirical evaluation of TopUML modeling is based on the results gained by two independent expert groups in practical experiment in which software is designed for the laundry problem domain. Within the experiment the first iteration is made by drawing the borders and scope of desired software system. Each participant before the experiment is asked to fill out self-evaluation questionnaire regarding the knowledge of software modeling. During the experiment, participants outlined several disadvantages (like lack of supporting tool) and several advantages (like formal problem domain analysis, formal relation of problem domain to developed models, and improved traceability options) of the proposed method. By summarizing up the experience and knowledge obtained during the experiment with experts the guidance manual [27] of TopUML modeling is prepared and published.

CONCLUSIONS

The goal of the doctoral thesis was to supplement UML with theoretical foundations in order to create grounds for converting notation into a formal modeling language and to define modeling method which allows to clearly trace cause-and-effect relationships in both problem and solution domains. The main result of the work is TFM supplementation with logical relations, specification of TopUML profile and definition of modeling method which is used to put into practice the created profile. All of the specified tasks for achieving the goal of thesis are completed and the following results and conclusions are obtained:

1. Results of analyzing UML, its specification and application in software development are as follows:
    a. Despite of the benefits gained by using UML within software development process, analysis of its specification and application shows a number of disadvantages and limitations,
    b. Although that language extension mechanisms are provided starting with the UML version 2.0, the development of profiles is a difficult task while the UML specification is a specification of a notation and thus it defines only elements of language, their notation and semantics,
    c. By analyzing a number of existing UML profiles an opinion is established that the most suitable way for specifying an UML profile is by using the same specification structure as used in UML specification,
    d. Modeling methods determine the application of UML within software development process and not the UML itself (review of UML application in industry and UML modeling methods review shows that the top five most applied UML diagrams are the same), thus a part of UML disadvantages and limitation can be solved by using an appropriate modeling method, and
    e. The fragmental application of UML diagrams and partial software development lifecycle coverage within analyzed methods do not eliminate disadvantages of UML at a sufficient level.
2. Above mentioned results of performed analysis lead to the conclusions that the following should be completed to achieve the goal of thesis:
    a. Supplement UML with the theoretical foundation, thus developing specification of TopUML profile, and

b. Define effective and usable modeling method for TopUML profile application within software development.

3. TFM supplementation with logical relations gives adequate base for transformation of TFM into other diagrams with complex structure.
4. By adding formalism of TFM mathematical topology to the UML, a modeling language specification is obtained which creates grounds for converting notation into a formal modeling language and which contains sufficient language elements to clearly identify, specify and trace cause-and-effect relationships.
5. The proposed modeling method includes the following aspects: proper analysis of problem and solution domains, application of most of the TopUML diagrams, it covers most of software development lifecycle; thus the identified UML disadvantages (size, complexity, incoherence and different interpretations) are reduced and even eliminated.
6. Application of TFM as a root model to synthesize other diagrams ensures that in a formal way are achieved following results:
    a. All artifacts created for solution domain are in conformance with the functioning characteristics of problem domain,
    b. It is possible to clearly trace cause-and-effect relationships within and between developed artifacts at the same and different abstraction levels,
    c. The developed artifacts are with high cohesion, and
    d. Components of developed system have low coupling with the rest of the system and a well-defined interface.
7. Developed TopUML profile and its modeling method have been successfully applied in an experimental software analysis and design project, as well as in a real software development project.
8. Approbation of proposed profile and modeling method shows that TopUML modeling deals with the Computation independent viewpoint and the Platform independent viewpoint within MDA. The TopUML modeling is concerned with the "*what*" and not "*how*":
    a. The "*what*" means what is what and what should do what (i.e., identification of classes and their responsibilities), and
    b. The "*how*" means how the responsibility will be implemented in a specific platform or technology.
9. Performed approbation of TopUML profile and modeling method together with two independent expert groups in a practical software design experiment gives an empirical

evaluation for the proposed modeling language and method. The following advantaged are outlined: theoretical foundations, formal modeling activities, transformations between models. Previously mentioned advantages together reduce the risk of rewriting software code.

The future research directions are as follows:

- Development of a tool supporting the TopUML profile. Since the TopUML is developed as a profile of UML version 2.4.1, it can be introduced to any tool that supports its extensions with UML profiles thus eliminating the need of a completely new tool creation.
- Research on the application of TopUML for platform independent viewpoint transformation into platform specific viewpoint and generating software code.

APPENDICES

Appendix 1

USED ABBREVIATIONS

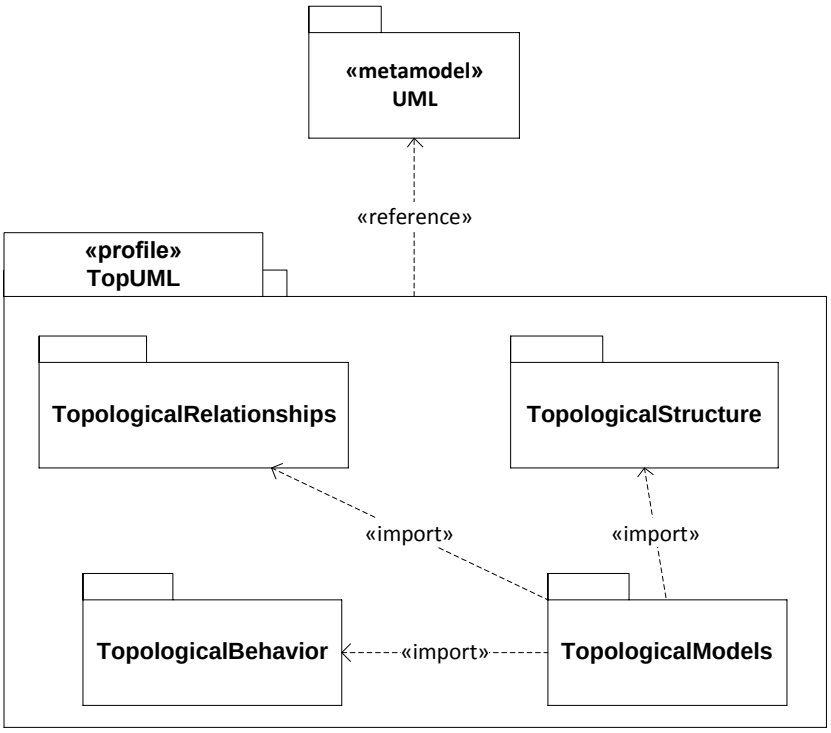
No.	Abbreviation	Description
1.	ASMOS	Automated structural modeling system
2.	AUP	Agile Unified Process
3.	B.O.O.M.	Business Object-Oriented Modeling
4.	CIM	Computation Independent Model
5.	CWM	Common Warehouse Metamodel
6.	DSL	domain-specific language
7.	GRASP	General Responsibility Assignment Software Pattern
8.	MDA	Model Driven Architecture
9.	MOF	Meta-Object Facility
10.	MSF	Microsoft Solutions Framework
11.	OCL	Object Constraint Language
12.	OMG	Object Management Group
13.	OMG SysML	Object Modeling Group System Modeling Language
14.	OMT	Object-Modeling Technique
15.	OOSE	Object-Oriented Software Engineering
16.	PIM	Platform Independent Model
17.	pUML	Precise UML
18.	PVS	Prototype Verification System
19.	RAISE	Rigorous Approach to Industrial Software Engineering
20.	RUP	Rational Unified Process
21.	SOA	Service-oriented architecture
22.	SoaML	Service Oriented Architecture Modeling Language
23.	TFM	Topological functioning model
24.	TFMfMDA	Topological Functioning Modeling for Model Driven Architecture
25.	TopUML	Topological Unified Modeling Language, Topological UML
26.	UML	Unified Modeling Language
27.	UP	Unified software development process, Unified process
28.	XMI	XML Metadata Interchange
29.	XML	Extensible markup language
30.	xUML	Executable UML, profile of UML for execution
31.	Z	Z notation

Appendix 2

TOPUML SPECIFICATION

The profile diagram specifying TopUML language consists of eight stereotypes and two enumerations which are divided into four packages. The top-level profile diagram of TopUML is given in Figure "TopUML profile top level package" below which shows the related metamodel and relationships between packages in profile. The packages are used to group together elements basing on their intent and semantics and to ease the evolution of TopUML (i.e., creation of new TopUML versions). The packages that build up TopUML profile are as follows:

- *TopologicalRelationships* – contains constructs related to relationships
- *TopologicalStructure* – contains constructs related to structure representation
- *TopologicalBehavior* – contains constructs related to behavior modeling, and
- *TopologicalModels* – contains diagram types added to UML by TopUML profile.

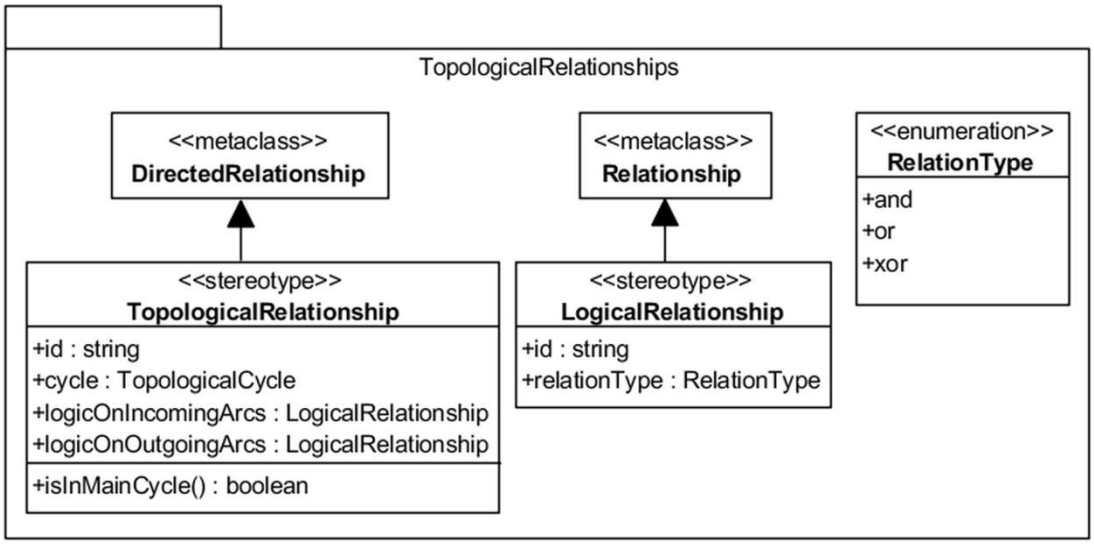


TopUML profile top level package

Stereotypes and enumerations included in packages are specified in subsequent four subsections. Stereotypes are related to corresponding metaclasses by extension relationship.

Topological Relationships Package

The topological relationships package contains all additional relationships that are created by TopUML profile. These relationships provide the necessary constructs to create Topological functioning model, Topological class diagram, and Topological use case diagram. The relations introduced are used across multiple diagram types thus making TopUML profile more compact and without needless constructs. The topological relationships package is given in Figure "*TopologicalRelationships* package" as package *TopologicalRelationships*.



TopologicalRelationships package

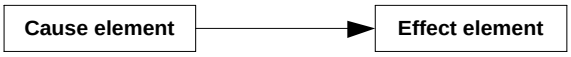
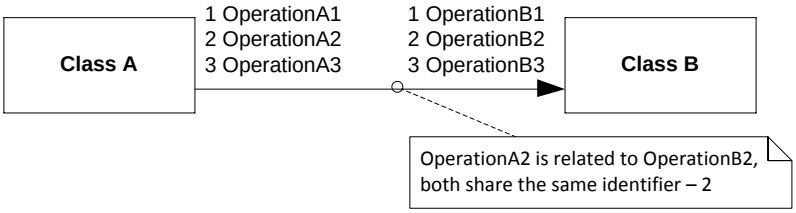
Topological relationship in TopUML is reflected by stereotype *TopologicalRelationship*. Specification of *TopologicalRelationship* is given below in Table "Specification of stereotype *TopologicalRelationship*".

Specification of stereotype *TopologicalRelationship*

Clause	Specification
Description	Topological relationship represents a cause-and-effect relation between two elements – source and target element. Extends «metaclass» DirectedRelationship.
Attributes	• + id : string [1] o Specifies the identifier of topological relationship.
Associations	• + cycle : TopologicalCycle [0..*] o Specifies the cycle to which this topological relationship belongs. Topological relationship can belong to many cycles at a time.

Clause	Specification

	• + logicOnIncomingArcs : LogicalRelationship [0..*]
	  ○ Specifies logical relationship to which this logical relationship belongs. Topological relationship can belong to many logical relationships at a time.
	• + logicOnOutgoingArcs : LogicalRelationship [0..*]
	  ○ Specifies logical relationship to which this logical relationship belongs. Topological relationship can belong to many logical relationships at a time.

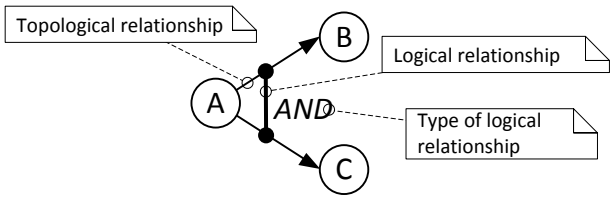
Constraints	[1] Logical relationship logicOnIncomingArcs should belong to TFM (an instance of TopologicalFunctioningModel). [2] Logical relationship logicOnOutgoingArcs should belong to TFM (an instance of TopologicalFunctioningModel). [3] If TopologicalRelation relates instances of UseCase and Actor, then source cannot be the same type and/or instance as target. [4] If TopologicalRelation relates instances of UseCase and Actor, then target cannot be the same type and/or instance as source.
Additional Operations	• + isInMainCycle() : boolean ○ Check that topological relationship belongs to main functioning cycle.
Semantics	Topological relationship is a binary relationship that shows a cause-and-effect relation between two elements – source element and target element. A topological relationship is assertion that indicates that the effect element can be triggered only by the cause element thus showing that effect element is executed only after the cause element executes.
Notation	Topological relationship is shown as arrow with filled arrowhead pointing from cause element to effect element (the arrowhead is placed at the effect side); it is directed only in one way – from cause to effect. 
Presentation Options	In the case of topological class diagrams where behavioral features (operations) are related with topological relationship, multiple topological relationships can be merged by listing cause and effect behavioral features together with their identifiers on the ends of relationship (identifier for related behavior features on both ends of relationship is the same). Identifiers are local to one topological relationship; it can be any user-specific symbol. 

Logical relationship in TopUML is reflected by stereotype *LogicalRelationsip*. Specification of *LogicalRelationsip* is given below in Table "Specification of stereotype *LogicalRelationsip*".

Specification of stereotype *LogicalRelationsip*

Clause	Specification
Description	Logical relationship represents a logical relation between two or more topological relationships. It extends «metaclass» Relationship.

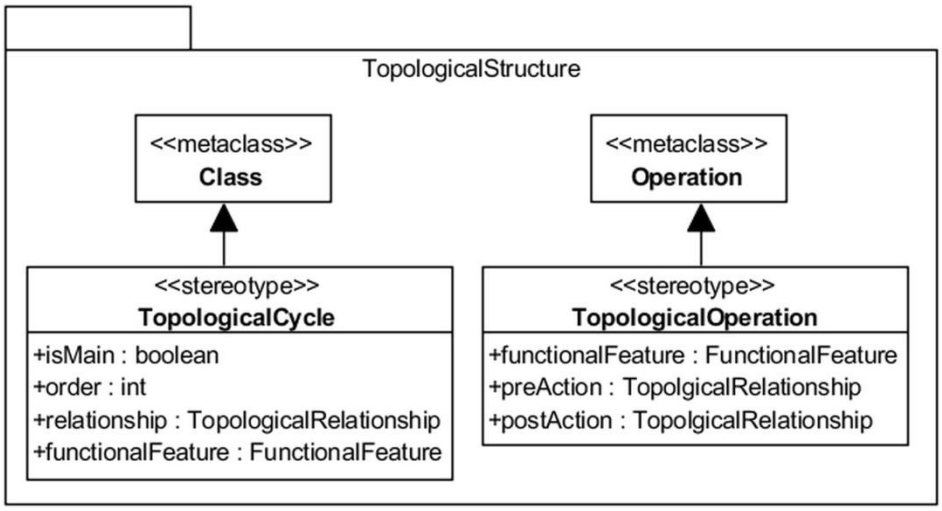
Attributes	• + id : string [1]
	  ○ Specifies the identifier of logical relationship.
	• + relationType : RelationType [1]
	  ○ Specifies the type of logical relation; defined values: *and*, *or*, and *xor*.

Associations	No additional associations.
Constraints	[1] Minimal count of related elements is two (i.e., relatedElement : Element [2..*]). [2] relatedElement can be only of type TopologicalRelationship. [3] Related topological relationships should belong to TFM (an instance of TopologicalFunctioningModel). [4] Either target element or source element of related topological relationships should be the same.
Semantics	Logical relationship represents logical relation between two or more topological relationship belonging to TFM (an instance of TopologicalFunctioningModel). It can show conjunction (and), disjunction (or), and exclusive or (xor).
Notation	Logical relationship is drawn as solid line connecting related topological relationships. If topological relationship is included in logical relationship, a point is set on the place where line representing logical relationship crosses arrow representing topological relationship. Next to the line of logical relationship is added label denoting the type of it. 

Topological Structure Package

The topological structure package contains all additional structure elements that are created by TopUML profile. These elements provide the necessary constructs to create Topological functioning model, Topological class diagram, and Topological use case diagram. The elements introduced are used across multiple diagram types thus making TopUML profile more compact and without needless constructs. The topological structure

package is given below in Figure "*TopologicalStructure* package" as package *TopologicalStructure*.

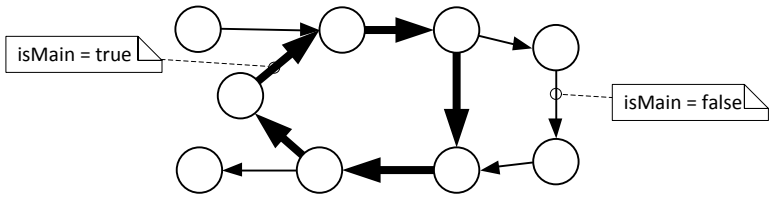


TopologicalStructure package

Topological cycle in TopUML is reflected by stereotype *TopologicalCycle*. Specification of *TopologicalCycle* is given below in Table "Specification of stereotype *TopologicalCycle*".

Specification of stereotype *TopologicalCycle*

Clause	Specification
Description	Topological cycle represents directed functional cycle of system; it consists of elements and relationships between them. Extends «metaclass» Class.
Attributes	• + isMain : Boolean [1] o Indicates if cycle is main functioning cycle of system. • + order : Integer [1] o Shows the order of functioning cycle.
Associations	• + relationship : TopologicalRelationship [2..*] o Contains all topological relationships belonging to this functioning cycle. • + functionalFeature : FunctionalFeature [2..*] o Contains all functional feature belonging to this functioning cycle.
Constraints	[1] Only one main functional cycle is allowed in system. [2] Can contain elements and relations between them which form the oriented cycle.
Semantics	It is acknowledged that every business and technical system is a subsystem of the environment. Besides that a common thing for all system functioning should be the main feedback, visualization of which is an oriented cycle. Therefore, it is stated that at least one directed

Clause	Specification
	closed loop (main functioning cycle) must be present in every topological model of system functioning. It shows the "main" functionality that has a vital importance in the functioning of system, i.e., by interrupting the main cycle the system can no longer function or its functioning is deformed.
Notation	There is no general notation for a TopologicalCycle. It is formed by elements and relationships between them.
Presentation Options	The main topological cycle (isMain = true) can be highlighted by drawing bolder lines of relationships belonging to it.
Examples	Example shows diagram with two cycles – one main cycle and one ordinary cycle. 

Topological operation in TopUML is reflected by stereotype *TopologicalOperation*. Specification of *TopologicalOperation* is given below in Table "Specification of stereotype *TopologicalOperation*".

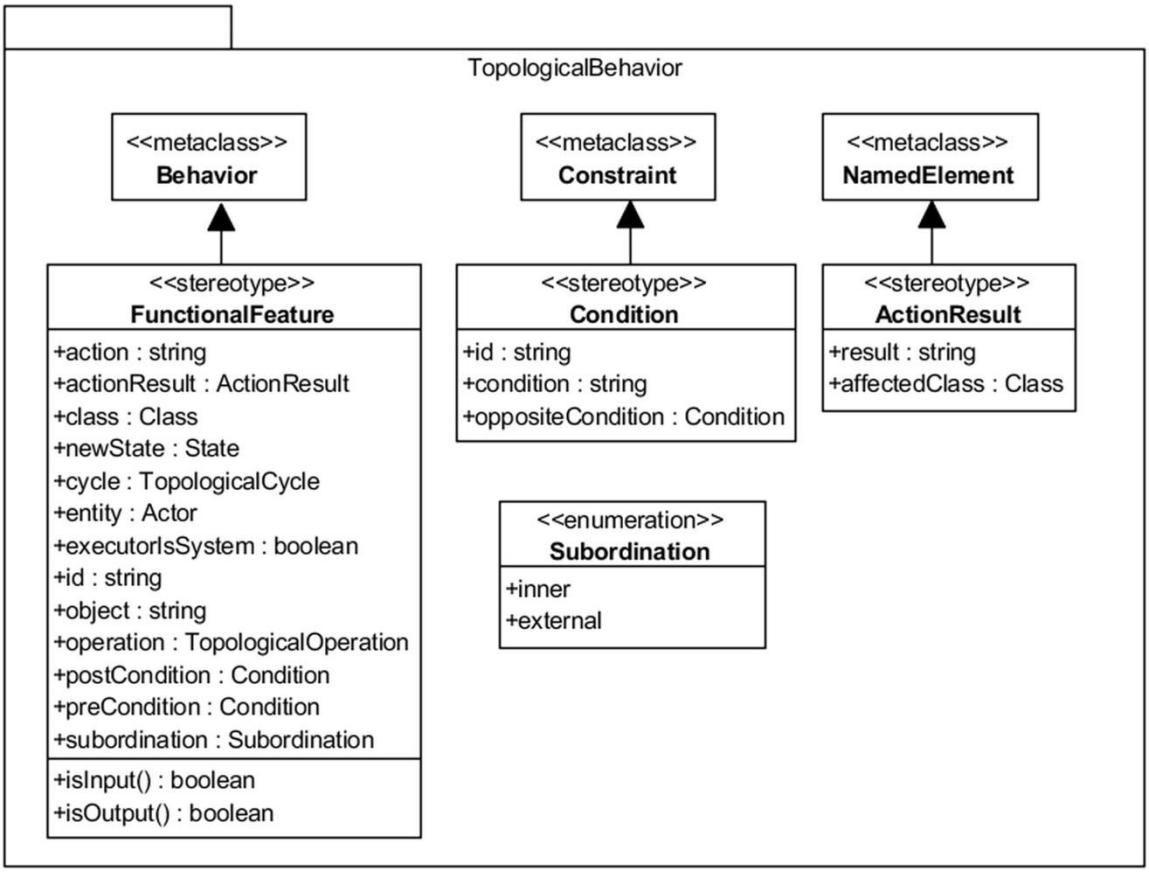
Specification of stereotype *TopologicalOperation*

Clause	Specification
Description	Topological operation is extension of «metaclass» Operation which is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior, and related functional features and topological relationships for specifying cause-and-effect relations within system.
Attributes	No additional attributes.
Associations	• + functionalFeature : FunctionalFeature [1] o Reference to functional feature which specifies this operation. • + preAction : TopologicalRelationship [0..1] o Specifies relationship to cause action for this action. • + postAction : TopologicalRelationship [0..1] o Specifies relationship to effect action of this action.
Constraints	[1] If topological relationship specifying cause action belongs to main functioning cycle (preAction.isMain -> true), then this operation cannot be suppressed. [2] If topological relationship specifying effect action belongs to main functioning cycle (postAction.isMain -> true), then this operation cannot be suppressed.
Semantics	Topological operation is a behavioral feature of classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior, and related functional features

Clause	Specification
	and topological relationships for specifying cause-and-effect relations within system, thus allowing a cause-and-effect relations to be modeled within the system by means of behavioral features (e.g., in topological class diagram).
Notation	No additional notation – uses the same as extended «metaclass» Operation.

Topological Behavior Package

The topological behaviors package contains all additional behavioral constructs that are created by TopUML profile that are necessary to create Topological Functioning model, Topological Class diagram, and Topological use case diagram. The relations introduced are used across multiple diagram types thus making TopUML profile more compact and without needless constructs. The topological behaviors package is given below in Figure "*TopologicalBehavior* package" as package *TopologicalBehavior*.

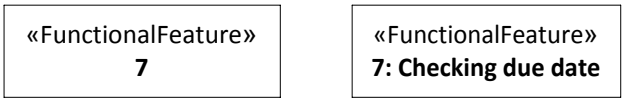


TopologicalBehavior package

Functional feature in TopUML is reflected by stereotype *FunctionalFeature*. Specification of *FunctionalFeature* is given below in Table "Specification of stereotype *FunctionalFeature*".

Specification of stereotype *FunctionalFeature*

Clause	Specification
Description	Stereotype FunctionalFeature extends «metaclass» Behavior and is an abstraction of functional feature.
Attributes	• + action : String [1] o Action performed by object (see attribute this.object). • + executorIsSystem : Boolean [1] o Indicates if execution of action A could be automated (i.e., performed without human interaction). • + id : String [1] o Identifier of functional feature. • + object : String [1] o Object that receives the result or that is used in action A (for example, a role, a time period, a catalogue, etc.). • + subordination : Subordination [1] o Specifies subordination of functional feature, can be *inner* or *external*.
Associations	• + actionResult : ActionResult [0..1] o Result of action (this.action) performed by object (this.object). • + class : Class [0..1] o Represents object (this.object) in static viewpoint of system (can be specified when the class diagram is synthesized). • + newState : State [0..1] o Represents the new state of object (this.object) after performing action (this.action). • + cycle : TopologicalCycle [0..*] o Shows the functioning cycles to which this functional feature belongs. • + entity : Actor [1] o Entity responsible for performing action specified by this functional feature. • + operation : TopologicalOperation [0..1] o Specifies functionality defined by action (this.action). Operation can be specified when the class diagram is synthesized. • + preCondition: Condition [0..*] o Set of preconditions, where precondition can be an atomic business rule. • + postCondition : Condition [0..*] o Set of postconditions, where precondition can be an atomic business rule.
Constraints	[1] Each functional feature is participating in at least one topological relationship either as a

Clause	Specification
	target or as a source. [2] In order for control flow to enter into functional feature, if any precondition is present it should be evaluated to true. [3] In order for control flow to leave functional feature, if any postcondition is present it should be evaluated to true.
Additional Operations	• + isInput() : boolean o Check if functional feature is input functional feature. • + isOutput() : boolean o Check if functional feature is output functional feature.
Semantics	Functional feature is a description of an atomic business action (i.e., it cannot be separated into a number of other business actions). Each functional feature is a unique tuple (stereotype FunctionalFeature is an abstraction of this tuple).
Notation	Functional feature is represented in a form of circle with label inside showing identifier of it. 
Presentation Options	Functional feature can be represented as a class showing its stereotype name «FunctionalFeature» above the identifier or identifier with action of functional feature. The format of displaying functional feature name is as follows: <id>[': '<action>] 

Precondition and postcondition in TopUML are reflected by stereotype *Condition.* Specification of *Condition* is given below in Table "Specification of stereotype *Condition*".

Specification of stereotype *Condition*

Clause	Specification
Description	Condition is an abstraction of pre- and post- conditions in system. It extends «metaclass» Constraint.
Attributes	• + id : String [1] o Identifier of condition. • + condition : String [1] o Boolean expression written in natural language or machine readable language.
Associations	• + oppositeCondition : Condition [0..1] o Relation to opposite condition of this condition, i.e. this.condition = ¬this.oppositeCondition.condition
Constraints	No additional constraints.
Semantics	Condition shows preconditions and postconditions within system. To enter the execution of behavior (e.g., functional feature) all preconditions of it should be true and to exit the execution

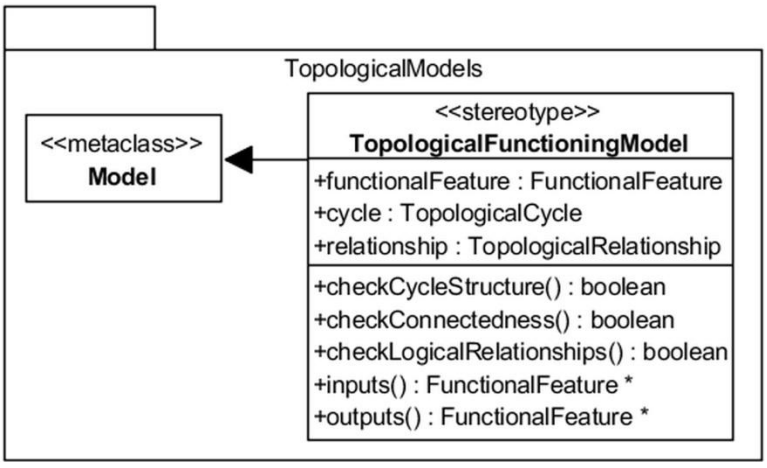
Clause	Specification
	of this behavior all postconditions should be evaluated to true. In the context of business system a condition also can be atomic business rule.
Notation	There is no notation for condition. It is shown only as attribute of functional feature.

Result of action in TopUML is reflected by stereotype *ActionResult*. Specification of *ActionResult* is given below in Table "Specification of stereotype *ActionResult*".

Specification of stereotype *ActionResult*

Clause	Specification
Description	Action result is an abstraction of result which is achieved by object performing action as specified by functional feature. Extends «metaclass» NamedElement.
Attributes	• + result : String [1] o Textual description of result of action performed by object specified in functional feature.
Associations	• + affectedClass : Class [0..1] o Related class which is affected by the result of action performed by object specified in functional feature.
Constraints	[1] Instance of ActionResult should belong to instance of FunctionalFeature.
Semantics	Action result specifies a result of object's action in functional feature specification. The action result shows also affected objects during its execution. For example, by registering customer in the registration journal a registration entry is created.
Notation	There is no notation for condition. It is shown only as attribute of functional feature.

Topological Models Package

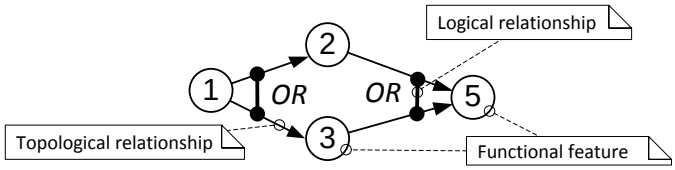
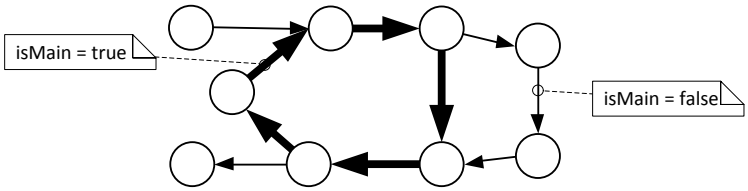
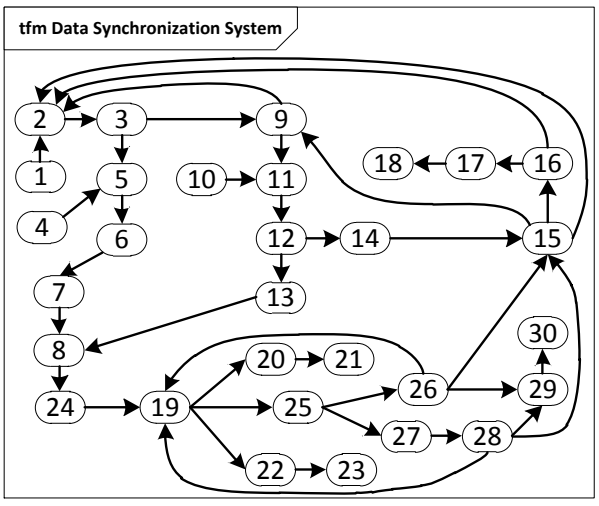


TopologicalModels package

The topological models package contains additional model that are introduced to UML by TopUML profile – Topological functioning model. The topological models package is given in Figure "*TopologicalModels* package" on previous page. Topological functioning model in TopUML is reflected by stereotype *TopologicalFunctioningModel*, its specification is given below in Table "Specification of stereotype *TopologicalFunctioningModel*".

Specification of stereotype *TopologicalFunctioningModel*

Clause	Specification
Description	TopologicalFunctioningModel is an abstraction of Topological functioning model in the metamodeling terms; it extends «metaclass» Model.
Attributes	No additional attributes.
Associations	• + functionalFeature : functionalFeature [2..*] o Functional feature is an atomic business action (i.e., it cannot be separated into a number of other business actions). • + cycle : TopologicalCycle [1..*] o Topological cycle represents directed functional cycle of system; it consists of functional features and relationships between them. • + topologicalRelationship : TopologicalRelationship [2..*] o Topological relationships relating functional features.
Constraints	[1] Instances of TopologicalRelationship can relate only instances of FunctionalFeature (i.e., source and target of TopologicalRelationship can by only of type FunctionalFeature). [2] TopologicalFunctioningModel should contain at least two elements of type FunctionalFeature. [3] TopologicalFunctioningModel should contain at least two elements of type TopologicalRelationship. [4] TopologicalFunctioningModel should contain at least one instance of type TopologicalCycle. [5] TopologicalFunctioningModel should contain one instance of type TopologicalCycle with its attribute isMain set to value true (isMain = true).
Additional Operations	• + checkCycleStructure() : boolean o Check that created model contains functioning cycles. • + checkConnectedness() : boolean o Check that all functional features are connected with cause-and-effect relationships. • + checkLogicalRelationships() : boolean o Identify and set logical relations within model. In addition identification of logical relations does an additional model checking by verifying correctness and allowance of specified preconditions and postconditions. • + inputs() : FunctionalFeature * o Array of functional features that are input functional features

Clause	Specification
	(this.functionalFeature[].isInput). • + outputs() : FunctionalFeature * ○ Array of functional features that are output functional features (this.functionalFeature[].isOutput).
Semantics	TopologicalFunctioningModel represents Topological functioning model (TFM) by using UML metamodeling constructs. TFM is a mathematical model that shows functioning of a system in the form of directed graph consisting of functional features and topology between them. Functional features embed information of systems functioning and its structural description while topology defines cause-and-effect relations between them.
Notation	Topological functioning model is displayed as a directed graph showing functional features as circles and topological relationships (i.e., cause-and-effect relations) between them. Topological relationship is denoted with solid line and filled arrowhead to the target (i.e., effect) functional feature. Additionally logical relations can be shown between topological relationships. Logical relationship is drawn as solid line connecting related topological relationships (connection with topological relationship is denoted by bold point). Topological functioning model in diagram header is denoted with keyword tfm. 
Style Guidelines	The main topological cycle (isMain = true) can be highlighted by drawing bolder lines of relationships belonging to it. Example shows diagram with two cycles – one main cycle and one ordinary cycle. 
Examples	

Appendix 3

MAPPINGS BETWEEN TOPUML DIAGRAMS

Mappings between elements of TFM and elements of Communication and Sequence diagrams are given below in Table "Mappings between elements of TFM and elements of Communication and Sequence diagrams".

Mappings between elements of TFM and elements of Communication and Sequence diagrams

	TFM element	Communication and Sequence diagram element	Description
1.	Class specified by functional feature	Lifeline	Each functional feature specifies object which is performing action. During analysis of system object is specified by class.
2.	Operation specified by functional feature	Message	Each functional feature specifies an atomic business action which later is specified by topological operation.
3.	Sequence of functional features	Message sequence number[1] Message order[2]	Message sequence number is denoted by the sequence number of functional feature. The sequence of functional features is defined by problem domain expert.
4.	Logical relations	Message sending concurrency	Logical relations in TFM give additional information about execution concurrency of functional features, thus allowing to define concurrency within Communication diagram.

[1] Communication diagram

[2] Sequence diagram

Mappings between elements of TFM and elements of Activity diagram are given below in Table "Mappings between elements of TFM and elements of Activity diagram".

Mappings between elements of TFM and elements of Activity diagram

	TFM element	Activity diagram element	Description
1.	Action of object specified by functional feature	Action	Each functional feature specifies an atomic business action which is represented by action of object and later is specified by topological operation. In activity diagram one action represents one functional feature from TFM.
2.	Cause-and-effect (i.e., topological) relationship	Edge	Functional features are connected by topological relationship which is represented by straight line with arrowhead at effect side (i.e., it points from cause to effect). In activity diagram one edge

	TFM element	Activity diagram element	Description
			represents one topological relationship from TFM.
3.	Logical relationship with type *xor* (and partially *or*)	Decision and merge node	Logical relations in TFM give additional information about execution concurrency of functional features and decision making within system. Thus exclusive or (*xor*) within Activity diagram is represented with decision node and corresponding merge node. Disjunction (*or*) is represented in a mixture of decision and fork nodes.
4.	Preconditions of functional feature	Guards on edges outgoing from decision node	Preconditions of functional feature in TFM are represented as guards on edges incoming to corresponding action in Activity diagram.
5.	Logical relationship with type *and* (and partially *or*)	Fork and join node	Logical relations in TFM give additional information about execution concurrency and decision making within system. Thus conjunction (*and*) within Activity diagram is represented with fork node and corresponding join node. Disjunction (*or*) is represented in a mixture of decision and fork nodes.
6.	Functional feature	Initial node	In basic scenario when input functional feature is transformed into an action, an initial node is added before this action. In more advanced scenario TFM can be split up in several parts and each part represented by its own Activity diagram. In such case initial node is added before action which is obtained from the input functional feature of that TFM part.
7.	Functional feature	Final node	In basic scenario when output functional feature is transformed into an action, a final node is added after this action. In more advanced scenario TFM can be split up in several parts and each part represented by its own Activity diagram. In such case final node is added after action which is obtained from the output functional feature of that TFM part.

Mappings between elements of TFM and elements of Topological use case diagram are given below in Table “Mappings between elements of TFM and elements of Topological use case diagram”.

Mappings between elements of TFM and elements of Topological use case diagram

	TFM element	Topological use case diagram element	Description
1.	TFM or part of	Subject	TFM itself defines subject of Topological use case diagram. If

	TFM element	Topological use case diagram element	Description
	TFM		TFM is divided into parts according to subsystems then each part of TFM defines the subject.
2.	Entity of functional feature	Actor	Actor is an entity of input and output functional features.
3.	Functional features	Use case	Use case is defined by a set of functional features. All functional features within one set should be connected – there should be no separated functional features. The set of functional features included in one use case (i.e., the scope of use case) is denoted by expert, by functional requirement, or by goal.
4.	Topological relationship	Topological relationship	The topological relationship from input functional feature to the descendant functional feature denotes topological relationship pointing from actor to use case. The topological relationship from predecessor of output functional feature to the output functional feature denotes topological relationship pointing from use case to actor.
5.	Cause-and-effect (i.e., topological) relationship	Relationship between Use cases	Relationship between Use cases are denoted by the existence of topological relationship between functional features belonging to use cases.
6.	Logical relationship	Extend relationship	The type of relationship between use cases is denoted by the type of logical relationship in TFM. The disjunction (*or*) and exclusive or (*xor*) denote extend relationship between use cases.
7.	Cause-and-effect (i.e., topological) and logical relationship	Include relationship	The type of relationship between use cases is denoted by the type of logical relationship in TFM. The conjunction (*and*) denote include relationship between use cases. If there is no logical relationship between topological relationships in TFM, then it indicates that there exist include relationship between use cases.

Mappings between elements of TFM and elements of Topological class diagram are given below in Table “Mappings between elements of TFM and elements of Topological class diagram”.

Mappings between elements of TFM and elements of Topological class diagram

	TFM element	Topological class diagram element	Description
1.	Class specified by functional feature	Class	Each functional feature specifies object which is performing action thus during analysis of system the object is specified by

	TFM element	Topological class diagram element	Description
			class. A class in Topological class diagram represents one class which is obtained by merging all functional features specifying the same class.
2.	Attributes of class specified by functional feature	Attribute of class	Each functional feature specifies an atomic business action which involves specification of affected data and data fields. Later this information can be specified as attributes of corresponding class.
3.	Operation specified by functional feature	Operation of class	Each functional feature specifies an atomic business action which later is specified by topological operation in TFM.
4.	Topological relationship	Topological relationship	Topological relationship within TFM is drawn between two functional features while in Topological class diagram it is drawn between two topological operations. In fact each topological operation is defined by one functional feature, so the topological relationship is transferred 1:1 from TFM into Topological class diagram.
5.	Result of action and class specified by functional feature	Association	An association within Topological class diagram can be added between class specified by functional feature and class specified by result of action of the same functional feature. By further analysis of the action context an aggregation or composition can be set of this association.

Mappings between elements of TFM and elements of State diagram are given below in Table "Mappings between elements of TFM and elements of State diagram". Each functional feature specifies an object performing certain action. By transforming TFM into State diagrams a set of State diagrams is obtained. The count of obtained State diagrams is denoted by count of distinct objects specified by functional features.

Mappings between elements of TFM and elements of State diagram

	TFM element	State diagram element	Description
1.	Object state specified by functional feature	State	Each functional feature specifies an object performing certain operation. If during execution of this action changes the state of object performing this action, functional feature specifies the new state of the object. Object state from functional feature is transformed into state in State diagram.
2.	Object state specified by	Initial state	When information from input functional feature is transformed into a state, an initial state is added before this state.

	TFM element	State diagram element	Description
	functional feature		
3.	Object state specified by functional feature	Final state	When information from output functional feature is transformed into a state, a final state is added after this state.
4.	Topological relationship	Transition	If during execution of action specified by functional feature is changed the state of object performing this action then incoming topological relationship defines transition from previous state to the new state.
5.	Operation specified by functional feature	Event	Each functional feature specifies an atomic business action which later is specified by topological operation in TFM. If functional feature specifies the new state of object, the operation is transformed into the event triggering transition from one state to another.
6.	Operation specified by functional feature	Entry effect	If current functional feature specifies the new state of object, the operation is transformed into the entry effect of this new state.
7.	Operation specified by functional feature	Exit effect	If descendant functional feature specifies the new state of object, the operation of this descendant functional feature is transformed into the exit effect of current state.
8.	Preconditions of functional features	Guard condition	If current functional feature specifies the new state of object, the preconditions of this functional feature are transformed into the guard conditions.
9.	Logical relationship with type *and* (and partially *or*)	Fork and Join	A logical relation in TFM give additional information about execution concurrency of functional features, thus conjunction (and) within State diagram is represented with fork and corresponding join. Disjunction (*or*) indicates of possible fork and join.

INFORMAL DESCRIPTION OF LAUNDRY FUNCTIONING

When a *person* **arrives** at laundry he **announces** his *identification data* to *clerk* and **gets checked** by *clerk*. All laundry *clients* **are registered**. *Registration* **is done** by the *clerk*. To become a registered *client* a *person* **should fill out** and **submit** *client card form*. Any *person*, who **is registered** in the *laundry client register* and who **has filled out** *client card* is considered as a *client*. If the person is not a registered client yet, the *clerk* **performs** the client registration. If the client does not have the *client card* yet, the *clerk* **makes** it anew. The *client card* **bears** the information about the client (*name*, *surname* and *address*) and every client card has its own *unique identification number*. Registered *clients* with *client card* have the **right to use** the services provided by laundry.

If a *client* wants to give *linen* for washing at laundry, he **requests** the *linen registration form* from *clerk*. After **creation of** *linen registration form*, *client* **gives** *linen* to *clerk*. *Clerk* then **weights** received *linen* and **registers** *linen weight* into *linen registration form*. When *weight of linen* **is registered** *clerk* **gives out** *linen registration form* to the *client*. *Client* then **fills out** all necessary *data* (*linen type*, *requirements for washing*, *due date*). After *client* **has filled out** *linen registration form*, he **gives** it back to *clerk*. *Clerk* **checks** if it is possible to **wash** *linen* till given *due date*. If it is not possible to fulfill washing request till requested *due date*, *clerk* **checks** for nearest possible *due date*, **announce** it to *client* and **gives** back *linen registration form* to *client* for **filling out** new *due date*. *Client* **writes** down new *due date* and **returns** *linen registration form* back to *clerk*. After every 20 *washing orders*, the *client* **gets** a *discount* for the 21st order. The *discount amount* **is** equal to 25% of the *average price* of the previous 20 *washing orders*. If the discount is greater than the price for 21st order, the remainder of the *discount* **is not transferred** to the next *washing order*. If it is possible to fulfill washing request till requested *due date*, *clerk* **calculates** *price* of the new *order* and **notifies** the *client*. If the *price* **is acceptable** for *client*, *clerk* **registers** *linen registration form* and **assigns** *unique identification number* to the *order*. *Clerk* **prepares** *receipt* of the received *order* by filling out *order unique identification number*, *due date*, *price* and *weight* of the *linen* and **gives** it out to *client*.

Received laundry washing *orders* **are registered** in the *orders list*. After **registering** new *order* into *orders list* it **is sorted** to **determine** whether the *order* **can be fulfilled** by using laundry's *washing machines*. The *order* **is passed** for *collaboration partner* if *dry-cleaning* **is needed**. **To pass** the *order* to *collaboration partner* a *purchase order* **is created**

by *clerk*. *Clerk* **assigns** *a number* to *purchase order* and **fills out** *purchase order* with necessary *data* (*linen type* and *weight*, *requirements for washing*, *due date*, *order unique identification number*). *Purchase order number* **is registered** in *linen registration form* by *clerk*. After *purchase order* **is prepared**, the *linen* **gets transferred** to *collaboration partner* for washing. When *collaboration partner* **returns** washed *linen*, *clerk* **makes** *a mark* in *linen registration form* that *linen* **is received** from *collaboration partner* and **is** ready for giving back to *client*. If *linen* **is washed** by using one of laundry's *washing machines*, *clerk* **makes** *a mark* in *linen registration form* in which *washing machine linen* was washed and that *linen* **is** ready for giving back to *client*. After completing an *order* it **is took out** from the *orders list*.

If *client* **wants to take back** washed *linen*, he **provides** *receipt* to *clerk*. After receiving *receipt clerk* **checks** the *order list* whether or not the *order* has been fulfilled and whether the *client* can get back *linen*. If the *order* is fulfilled *clerk* **prepares** *invoice* of *provided service* and **issues** it to the *client*. After *client* **has paid** the *invoice*, *clerk* **gives** the *linen* to *client* and **makes** *a mark* in *linen registration form* that *linen* has been given back to *client*. If *linen* **is returned** to the *client*, *clerk* **marks** the fact of returning in the *linen registration form*. If the *linen* **was washed** by the *collaboration partner* then after receiving a *payment* from the *client* a *bank transfer order* **is created** and the *amount* as stated in *purchase order* **is paid** to the *collaboration partner*.

Additionally the *laundry* **can accept** *washing requests* from *collaboration partners*. The **process** of registering the *washing request* from *collaboration partner* **is** the same as for *clients* (i.e., a *linen registration form* **is created** and **filled out** for each *washing request*). The *linen registration form* **is created** and **filled out** when the *collaboration partner* **submits** *washing request*. Since this *washing request* **contains** the *linen type* and *weight* the *linen* **is not weighted** again. When the *linen* **is received** from *collaboration partner* the *due date* and *price* of fulfilling the *washing request* **gets wrote** in the *linen registration form*. When the *linen registration form* **is completed** a *notice* to the *collaboration partner* **is sent**. After the *linen* **has been washed** the *clerk* **makes** a *mark* in the *linen registration form* that *linen* **is ready** for returning and **prepares** an *invoice*. When the *invoice* **is completed** it **is sent** to the *collaboration partner*. The washed *linen* **is returned** to the *collaboration partner* before the *invoice* is paid. The *invoice* **is paid** after the *client* of *collaboration partner* **has paid** for the washing to the *collaboration partner*. When the *invoice* **is paid** by the *collaboration partner*, the *clerk* **makes** a *mark* in *linen registration form* that the *invoice* **is paid**.

The *supplies* of *determents* **are monitored to ensure** that *linen* washing within the laundry's *washing machines* **is not interrupted**. If the *supplies* **are less** than required for

fulfilling *washing requests* for a month the *determents* **are bought** in the *amount* **required** to fulfill *washing requests* for a *quarter*. Before the *determents* **are ordered** a l*ist of required items* **is composed**. The *determents* **are bought** from the *determents trader* which is a collaboration partner of laundry. The *determents trader* **accepts** also *electronic orders* which should be in the form of *structured text file*. *Determents trader* **confirms** that he has received the *order*. Together with ordered *items* a *bill* **is received**. The *bill* **should be paid** by *accountant* in the *term* given on it.

A *usage counter* **is assigned** to each *washing machine*. The *counter* **is incremented** by 1 for every time the *washing machine* **is used** for washing the *linen*. After the *counter* of particular *washing machine* **has exceeded** 250 the *washing machine* **gets serviced** thus ensuring continuity of the *laundry* operation and high quality of *linen* washing. The *washing machines* **are serviced** by a *service center* (an *agreement* **is signed** between *laundry* and *service center*). The *agreement* **states** that the *provided services* **should be paid** once in month – at the 10^{th} day of the next month.

Appendix 5

FUNCTIONAL REQUIREMENTS AND SYSTEM GOALS OF THE LAUNDRY SOFTWARE SYSTEM

Functional requirements of the laundry software system

ID	Requirement
FR1	The system should ensure that the linen in laundry gets placed for washing and issued after washing only by registered clients.
FR2	If a client is not registered in the laundry, the system should provision new client registration by setting up a registration form for each new client.
FR3	In client registration form it should be possible to register the client's e-mail address.
FR4	System should implement placement of new orders according to the process defined by the laundry (including the preparation of registration form for order and unique identification number assignment).
FR5	System should calculate the amount of discount for the client's 21st washing request; the discount should be applied to it.
FR6	After client has placed new order for washing, the system should prepare and issue a receipt. Basing on this receipt client will be able to get back its linen after washing.
FR7	System should contain functionality which provides possibility to record in linen registration form if linen is washed by laundry's washing machines and if so then also which washing machine was used.
FR8	System should contain list of orders of linen washing. The list should implement following functionality: insertion of new orders, sorting of registered (inserted) orders, and removal of fulfilled orders.
FR9	After the fulfilled order has been removed from order's list system should send e-mail notification to client's e-mail address (if the e-mail address is registered in client's registration form).
FR10	In order to receive washed laundry client must present the receipt of this order or present the received e-mail from the laundry system.
FR11	System should implement checking of order's list to ensure if the given order has been fulfilled.
FR12	The system should be able to prepare an invoice to client for used laundry services and it should be able to mark in linen registration form that client has paid the issued invoice.
FR13	System should enable to make a mark in linen registration form that linen is issued to the client.
FR14	Washing requests should be accepted from collaboration partners according to the process defined for clients, including the preparation of linen registration form.
FR15	A notice should be prepared and sent to collaboration partner when the linen registration form is prepared.
FR16	After the linen has been washed a mark that linen is ready for returning in the linen registration form should be made, and an invoice should be prepared and sent to the collaboration partner.
FR17	The system should allow returning the washed linen to the collaboration partner before the invoice is paid. When the invoice is paid by the collaboration partner, a mark in linen registration form should be

ID	Requirement
	made stating that the invoice is paid.
FR18	A monitoring and supplementing of the reserves of determents should be established. The system should monitor the usage counters of washing machines and order a servicing of the particular washing machine if the counter has exceeded 250.
FR19	To ensure that the washing machines are accordingly serviced, the system should contain a register of laundry's washing machines. The register should contain following functionality: • The increment of washing machine usage counter by 1 every time it is used to wash linen, and • When the washing machine is serviced, its usage counter should be set to 0.

System goals of the laundry software system

ID	System goals
SG1	Checking and registering clients
SG2	Registering linen for washing
SG3	Registering and receiving linen from collaboration partners
SG4	Fulfilling registered linen washing orders
SG5	Withdrawal of washed linen and charging client for laundry services after the completion of order
SG6	Withdrawal of linen to the collaboration partner after fulfillment of order and receiving payment for the washing services provided
SG7	Ensure the continuity of the laundry operation by monitoring and ordering supplies of determents
SG8	Ensure the continuity of the laundry operation by monitoring the usage of washing machines and ordering servicing for them

Appendix 6

FUNCTIONAL FEATURES OF LAUNDRY FUNCTIONING

ID	Object action (A)	Precondition (PrCond)	Entity (E)	
1.	A person arriving at laundry		Person	External
2.	Announcing personal identification data		Person	External
3.	Checking of personal data with the laundry's client register		Clerk	Inner
4.	Handing out client card form	If the person is not registered yet	Clerk	Inner
5.	Fulfilling client card form		Person	Inner
6.	Preparing client card	If the person does not have the client card yet	Clerk	Inner
7.	Client card issue to the client		Clerk	Inner
8.	Authorizing client status	If the person is registered (and) if the client has the client card	Clerk	Inner
9.	Requesting linen registration form	If client has client card	Client	Inner
10.	Creating linen registration form		Clerk	Inner
11.	Giving linen to clerk		Client	Inner
12.	Taking linen from client		Clerk	Inner
13.	Weighting received linen		Clerk	Inner
14.	Registering linen weight in linen registration form		Clerk	Inner
15.	Giving out linen registration form to client		Clerk	Inner
16.	Filling out linen registration form (linen type, requirements for washing)		Client	Inner
17.	Filling out order due date in linen registration form		Client	Inner
18.	Giving back linen registration form to clerk	If linen registration form is filled out	Client	Inner
19.	Checking due date		Clerk	Inner
20.	Giving back linen registration form to client	If it is not possible to fulfill washing till requested due date	Clerk	Inner
21.	Announcing nearest possible due date to client		Clerk	Inner
22.	Calculating price without discount of the new order		Clerk	Inner
23.	Checking the count of previous orders		Clerk	Inner
24.	Calculating average price of the previous 20 washing orders	If it is 21st order of the client or if 20 orders are made after previous	Clerk	Inner

ID	Object action (A)	Precondition (PrCond)	Entity (E)	
		discount		
25.	Calculating 25% of the average price of the previous 20 washing orders		Clerk	Inner
26.	Calculating price with discount of the new order		Clerk	Inner
27.	Announcing the final price of the new washing order	If the due date and linen weight is fixed	Clerk	Inner
28.	Registering linen registration form	If price is acceptable for client	Clerk	Inner
29.	Assigning unique identification number to the order		Clerk	Inner
30.	Preparing receipt of the received order by filling out order unique identification number, due date, price and weight of the linen		Clerk	Inner
31.	Giving out receipt to client		Clerk	External
32.	Registering order in laundry orders' list		Clerk	Inner
33.	Sorting order list		Clerk	Inner
34.	Fulfilling order by using laundry's washing machines	If order can be fulfilled by using laundry's washing machines	Clerk	Inner
35.	Creating purchase order by filling out purchase order (linen type and weight, requirements for washing, due date, order unique identification number)	If dry-cleaning is needed	Clerk	Inner
36.	Assigning purchase order number to purchase order		Clerk	Inner
37.	Registering purchase order number in linen registration form		Clerk	Inner
38.	Transferring linen to collaboration partner		Clerk	Inner
39.	Receiving linen from collaboration partner		Clerk	Inner
40.	Marking that linen has been washed		Clerk	Inner
41.	Taking out completed order from the orders list		Clerk	Inner
42.	Marking in linen registration form in which machine linen was washed		Clerk	Inner
43.	Providing receipt to clerk	If client wants to take back washed linen basing on receipt	Clerk	Inner
44.	Checking the order list whether or not the order has been fulfilled		Clerk	Inner
45.	Preparing invoice of provided service	If the order is fulfilled	Clerk	Inner
46.	Issuing invoice to client		Clerk	External
47.	Paying for received service according to invoice		Client	External

ID	Object action (A)	Precondition (PrCond)	Entity (E)	
48.	Noticing the payment of washing services in linen registration form	If the invoice is paid	Clerk	Inner
49.	Giving out linen to client	If the linen has been washed	Clerk	External
50.	Receiving washed linen		Client	External
51.	Marking in linen registration form that linen has been returned to client	If the linen has been returned	Clerk	Inner
52.	Creating bank transfer order	If the linen was washed by collaboration partner	Clerk	External
53.	Paying to collaboration partner		Accountant	External
54.	Receiving linen for washing from collaboration partner		Collaboration partner	External
55.	Submitting linen washing request		Collaboration partner	External
56.	Checking washing requests submitted by collaboration partner		Clerk	Inner
57.	Filling out linen registration form for collaboration partner washing request		Clerk	Inner
58.	Calculating price for collaboration partner washing request	If washing request was made by collaboration partner	Clerk	Inner
59.	Determining due date for collaboration partner washing request		Clerk	Inner
60.	Preparing notice of registered linen washing request		Clerk	External
61.	Sending notice to collaboration partner		Clerk	External
62.	Marking in linen registration form that linen has been washed	If the linen has been washed	Clerk	Inner
63.	Returning linen to collaboration partner	If a washing request submitted by collaboration partner has been fulfilled	Clerk	External
64.	Receiving the washed linen		Collaboration partner	External
65.	Preparing invoice for washing services provided to collaboration partner		Clerk	Inner
66.	Sending invoice to collaboration partner		Clerk	External
67.	Paying the invoice		Collaboration partner	External
68.	Making a mark in linen registration form that the	If the collaboration partner	Clerk	Inner

ID	Object action (A)	Precondition (PrCond)	Entity (E)	
	invoice is paid	has paid the invoice		
69.	Reviewing the supplies of determents	If linen is washed by using laundry's washing machine	Clerk	Inner
70.	Composing a list of required determents	If the supplies are less than required for fulfilling washing requests for a month	Clerk	Inner
71.	Ordering required determents		Accountant	Inner
72.	Preparing the order of required determents		Clerk	External
73.	Submitting determents order to determents trader		Clerk	External
74.	Receiving confirmation of determents order submission		Clerk	Inner
75.	Delivering determents order		Determents trader	External
76.	Receiving ordered determents		Clerk	External
77.	Stocking up supplies of determents		Clerk	Inner
78.	Preparing bank transfer to pay bill for determents order		Clerk	External
79.	Paying the determents order bill		Accountant	External
80.	Increasing the usage counter of particular washing machine		Clerk	Inner
81.	Preparing order for servicing particular washing machine	If the usage counter of particular washing machine has exceeded 250	Clerk	Inner
82.	Ordering servicing of washing machine		Clerk	External
83.	Servicing washing machine		Service center	External
84.	Summarizing the costs of servicing washing machines for a month		Clerk	External
85.	Paying for servicing of washing machines		Accountant	External

Appendix 7

CLOSURING OF LAUNDRY FUNCTIONING TOPOLOGICAL SPACE

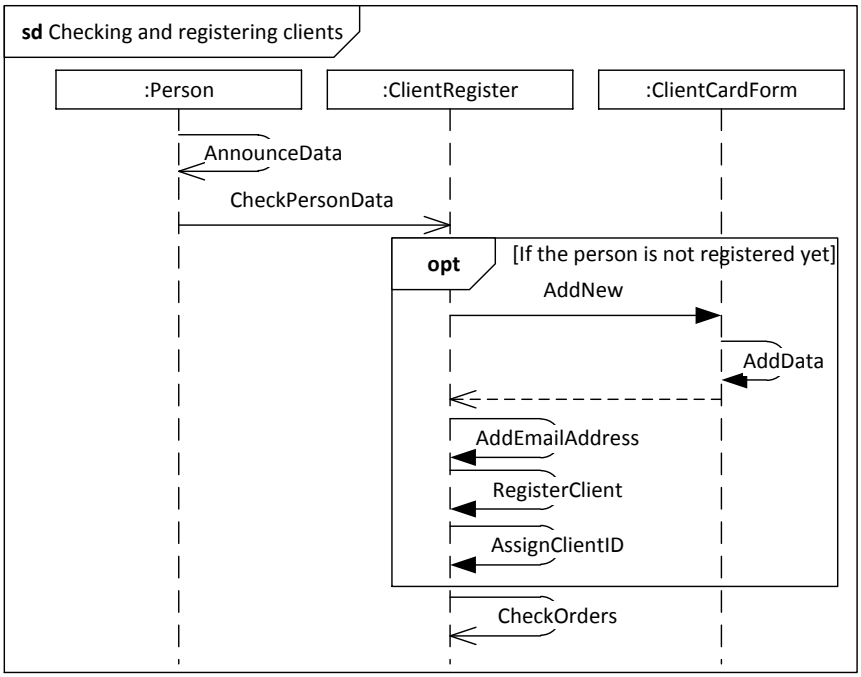
The example below illustrates how the closuring operation is applied over the set N in order to get TFM of laundry functioning:

$\mathbf{X_3}$ = {2, 3, 4, 8}
$\mathbf{X_4}$ = {3, 4, 5}
$\mathbf{X_5}$ = {4, 5, 6}
$\mathbf{X_6}$ = {5, 6, 7}
$\mathbf{X_7}$ = {6, 7, 8}
$\mathbf{X_8}$ = {3, 7, 9, 29, 43, 51}
$\mathbf{X_9}$ = {8, 9, 10}
$\mathbf{X_{10}}$ = {9, 10, 11}
$\mathbf{X_{11}}$ = {10, 11, 12}
$\mathbf{X_{12}}$ = {11, 12, 13, 57, 58}
$\mathbf{X_{13}}$ = {12, 13, 14}
$\mathbf{X_{14}}$ = {13, 14, 15}
$\mathbf{X_{15}}$ = {14, 15, 16}
$\mathbf{X_{16}}$ = {15, 16, 17}
$\mathbf{X_{17}}$ = {16, 17, 18, 21}
$\mathbf{X_{18}}$ = {17, 18, 19}
$\mathbf{X_{19}}$ = {18, 19, 20, 22}
$\mathbf{X_{20}}$ = {19, 20, 21}
$\mathbf{X_{21}}$ = {17, 20, 21}
$\mathbf{X_{22}}$ = {19, 22, 23}
$\mathbf{X_{23}}$ = {22, 23, 24, 27}
$\mathbf{X_{24}}$ = {23, 24, 25}
$\mathbf{X_{25}}$ = {24, 25, 26}
$\mathbf{X_{26}}$ = {25, 26, 27}
$\mathbf{X_{27}}$ = {23, 26, 27, 28}
$\mathbf{X_{28}}$ = {27, 28, 29}
$\mathbf{X_{29}}$ = {8, 28, 29, 32}
$\mathbf{X_{30}}$ = {29, 30, 31}
$\mathbf{X_{32}}$ = {29, 32, 33, 59}
$\mathbf{X_{33}}$ = {32, 33, 34, 35}
$\mathbf{X_{34}}$ = {33, 34, 42, 69, 77}
$\mathbf{X_{35}}$ = {33, 35, 36}
$\mathbf{X_{36}}$ = {35, 36, 37}
$\mathbf{X_{37}}$ = {36, 37, 38}
$\mathbf{X_{38}}$ = {37, 38, 39}
$\mathbf{X_{39}}$ = {38, 39, 40}
$\mathbf{X_{40}}$ = {39, 40, 41, 42}
$\mathbf{X_{41}}$ = {33, 40, 41, 44, 62}
$\mathbf{X_{42}}$ = {34, 40, 42, 80}
$\mathbf{X_{43}}$ = {8, 43, 44}
$\mathbf{X_{44}}$ = {41, 43, 44, 45}
$\mathbf{X_{45}}$ = {44, 45, 46, 48}
$\mathbf{X_{48}}$ = {45, 48, 49, 51, 52}
$\mathbf{X_{51}}$ = {8, 48, 51}
$\mathbf{X_{56}}$ = {55, 56, 57, 68}
$\mathbf{X_{57}}$ = {12, 56, 57}
$\mathbf{X_{58}}$ = {12, 58, 59}
$\mathbf{X_{59}}$ = {32, 58, 59, 60}
$\mathbf{X_{62}}$ = {41, 62, 63, 65}
$\mathbf{X_{65}}$ = {62, 65, 66, 68}
$\mathbf{X_{68}}$ = {65, 68, 56}
$\mathbf{X_{69}}$ = {34, 69, 70}
$\mathbf{X_{70}}$ = {69, 70, 71}
$\mathbf{X_{71}}$ = {70, 71, 72}
$\mathbf{X_{74}}$ = {71, 74, 77}
$\mathbf{X_{77}}$ = {74, 76, 77, 78, 34}
$\mathbf{X_{80}}$ = {42, 80, 81}
$\mathbf{X_{81}}$ = {80, 81, 82, 84}

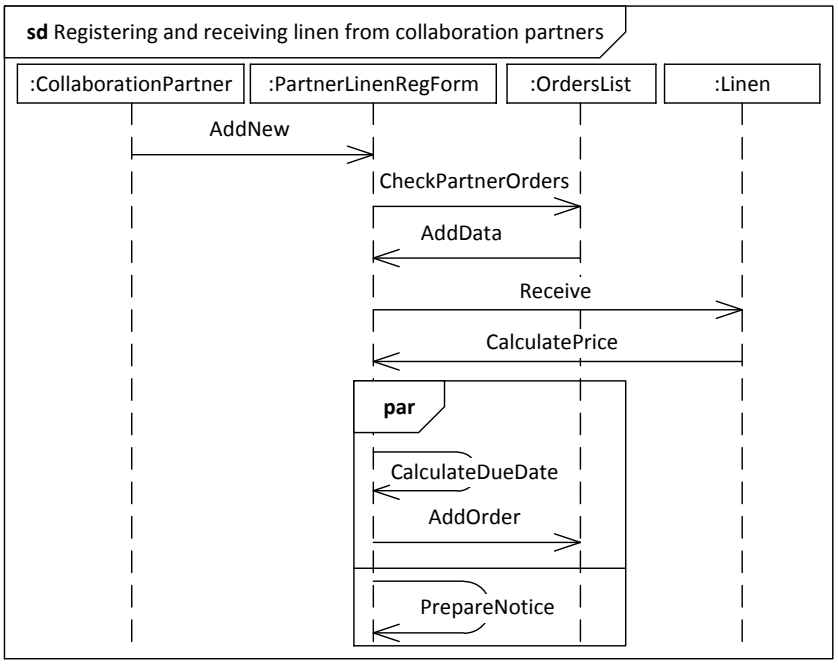
Appendix 8

SEQUENCE DIAGRAMS REPRESENTING BEHAVIOR OF LAUNDRY

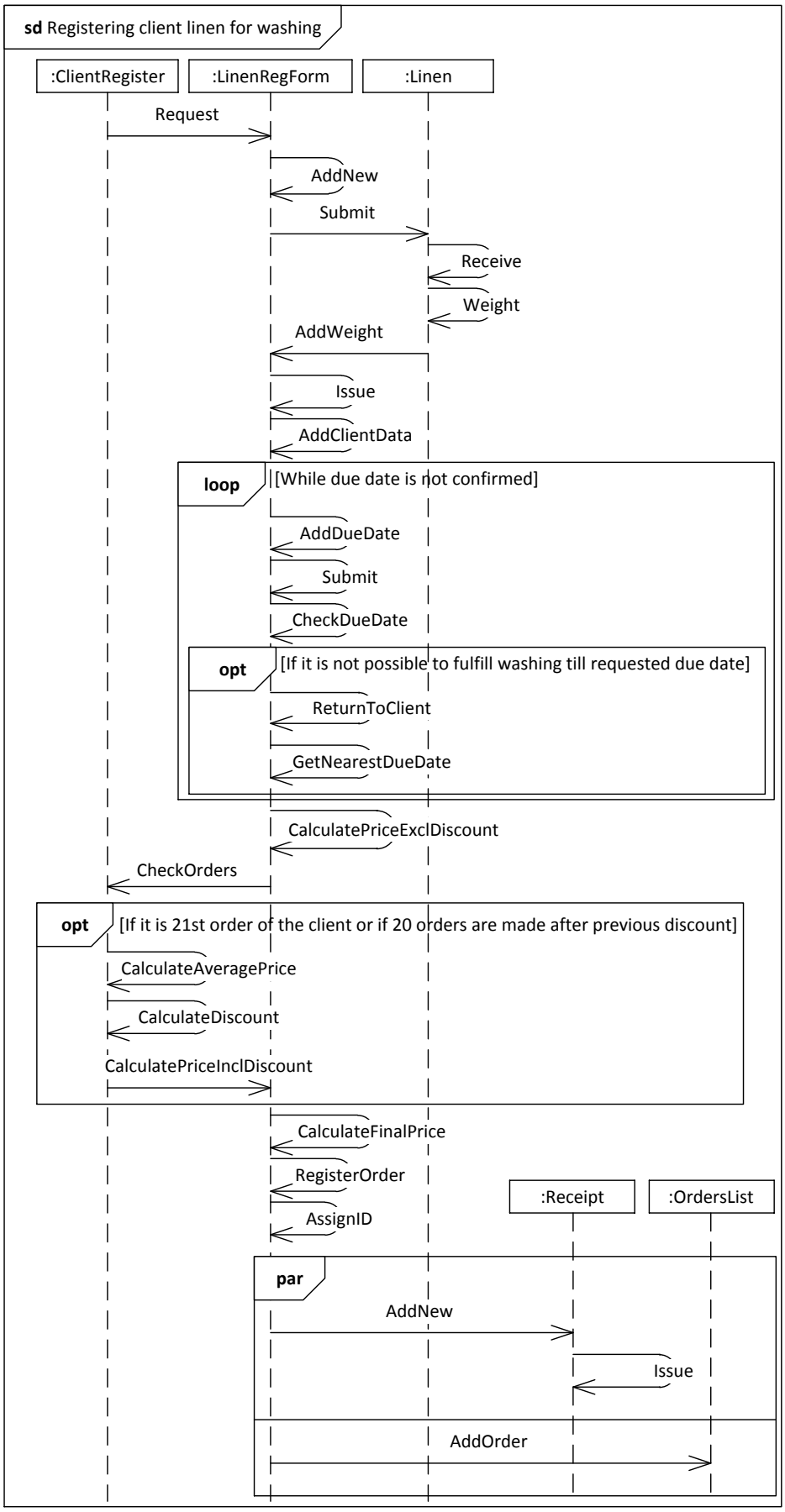
Sequence diagram "Checking and registering clients"



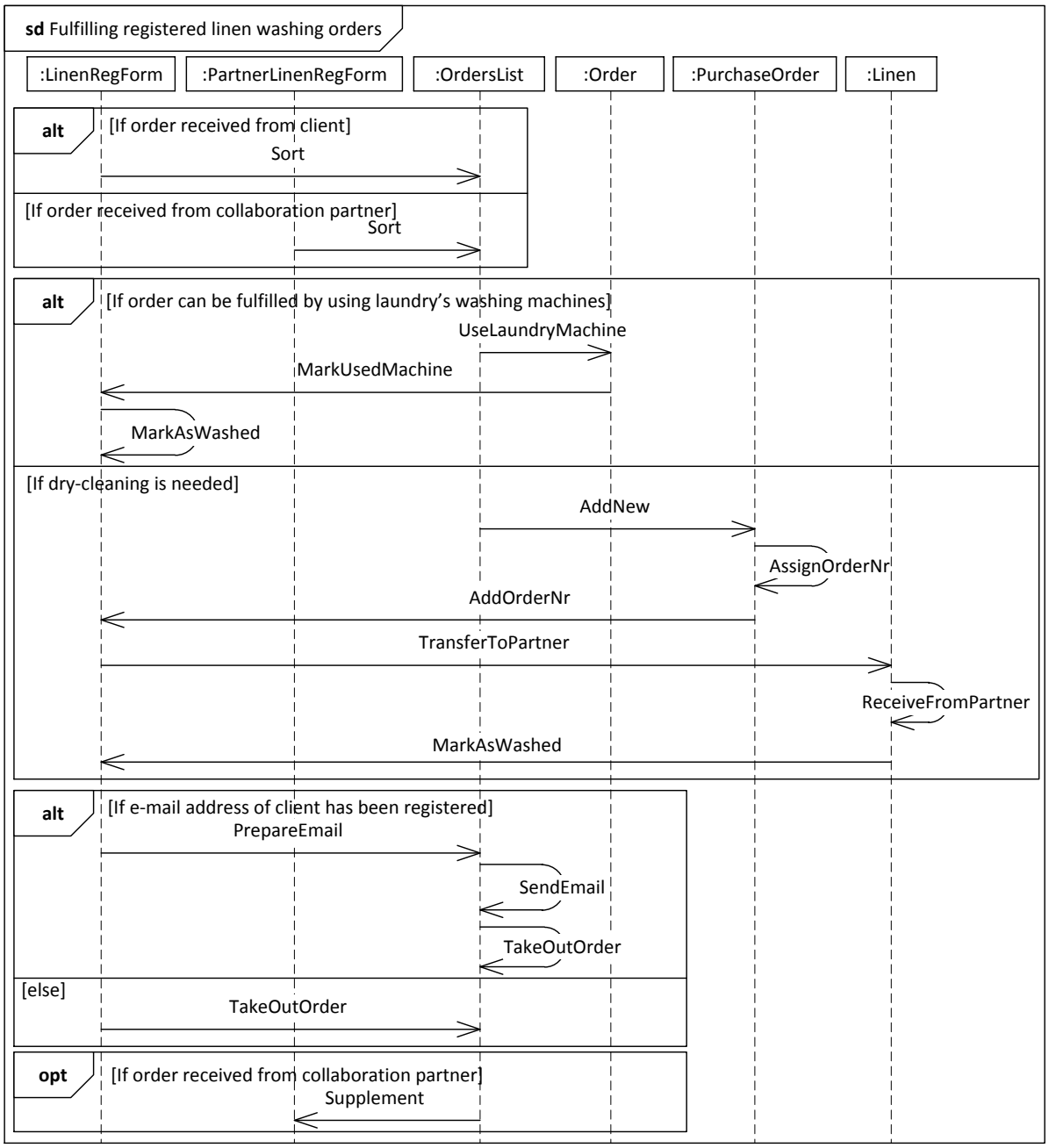
Sequence diagram "Registering and receiving linen from collaboration partners"



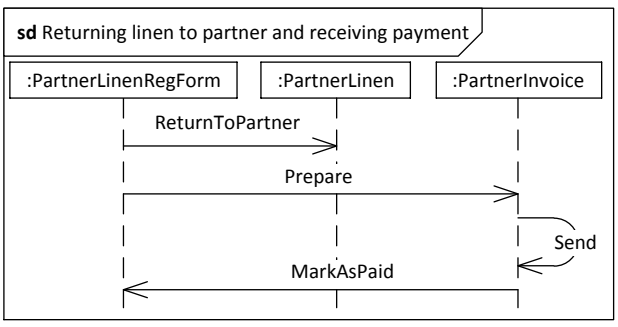
Sequence diagram “Registering client linen for washing”



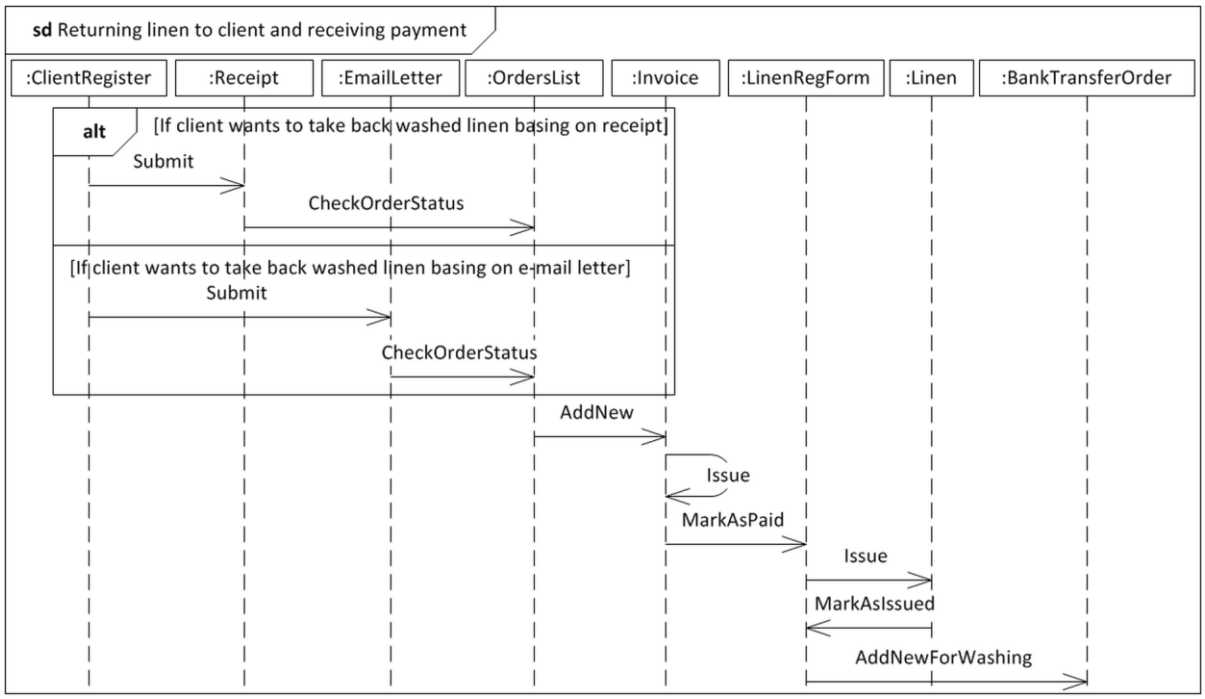
Sequence diagram “Fulfilling registered linen washing orders”



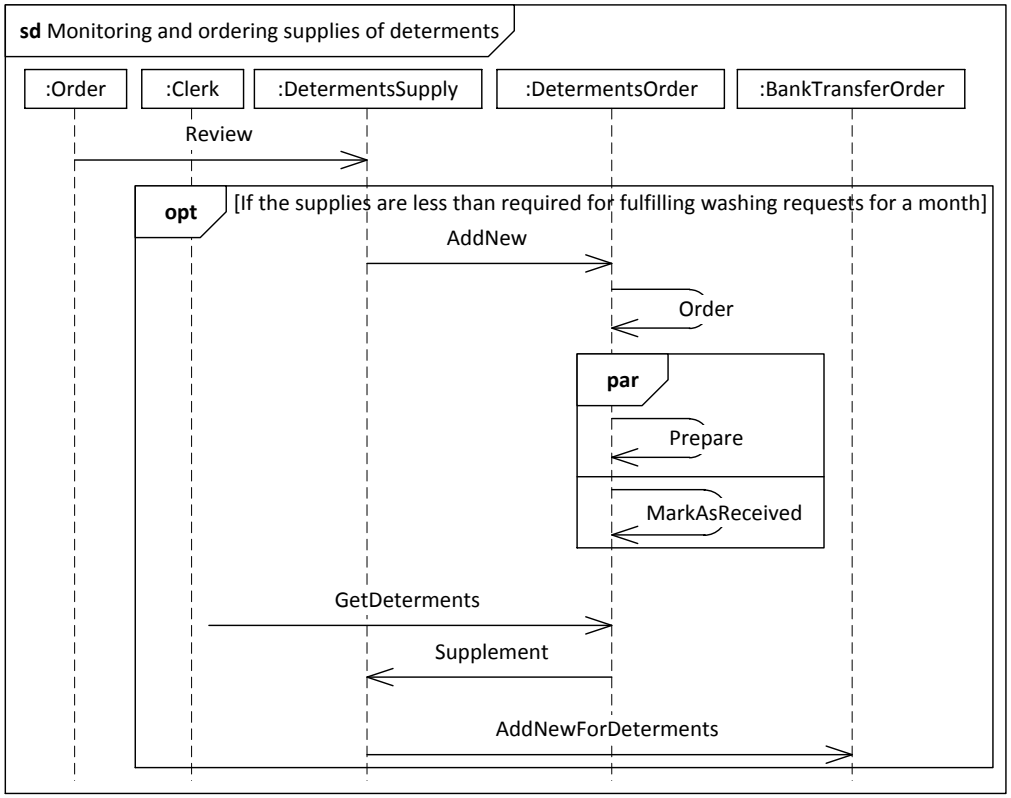
Sequence diagram “Returning linen to partner and receiving payment”



Sequence diagram “Returning linen to client and receiving payment”

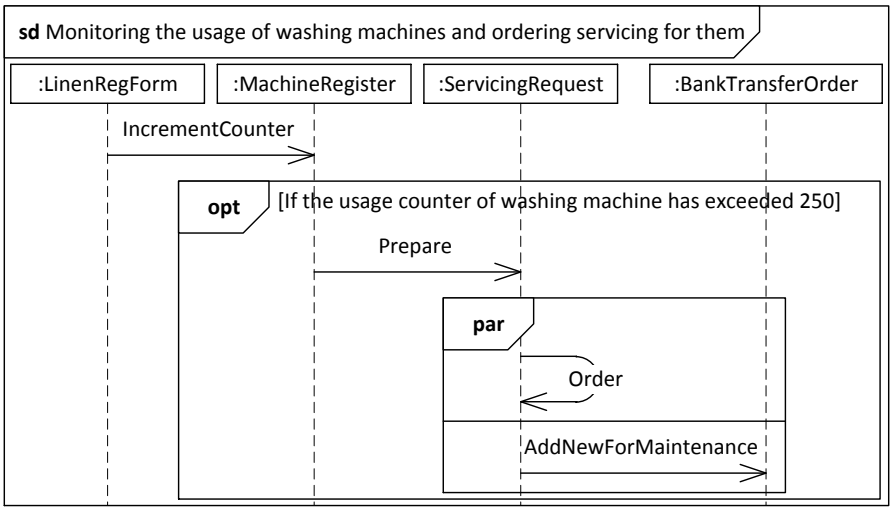


Sequence diagram “Monitoring and ordering supplies of determents”

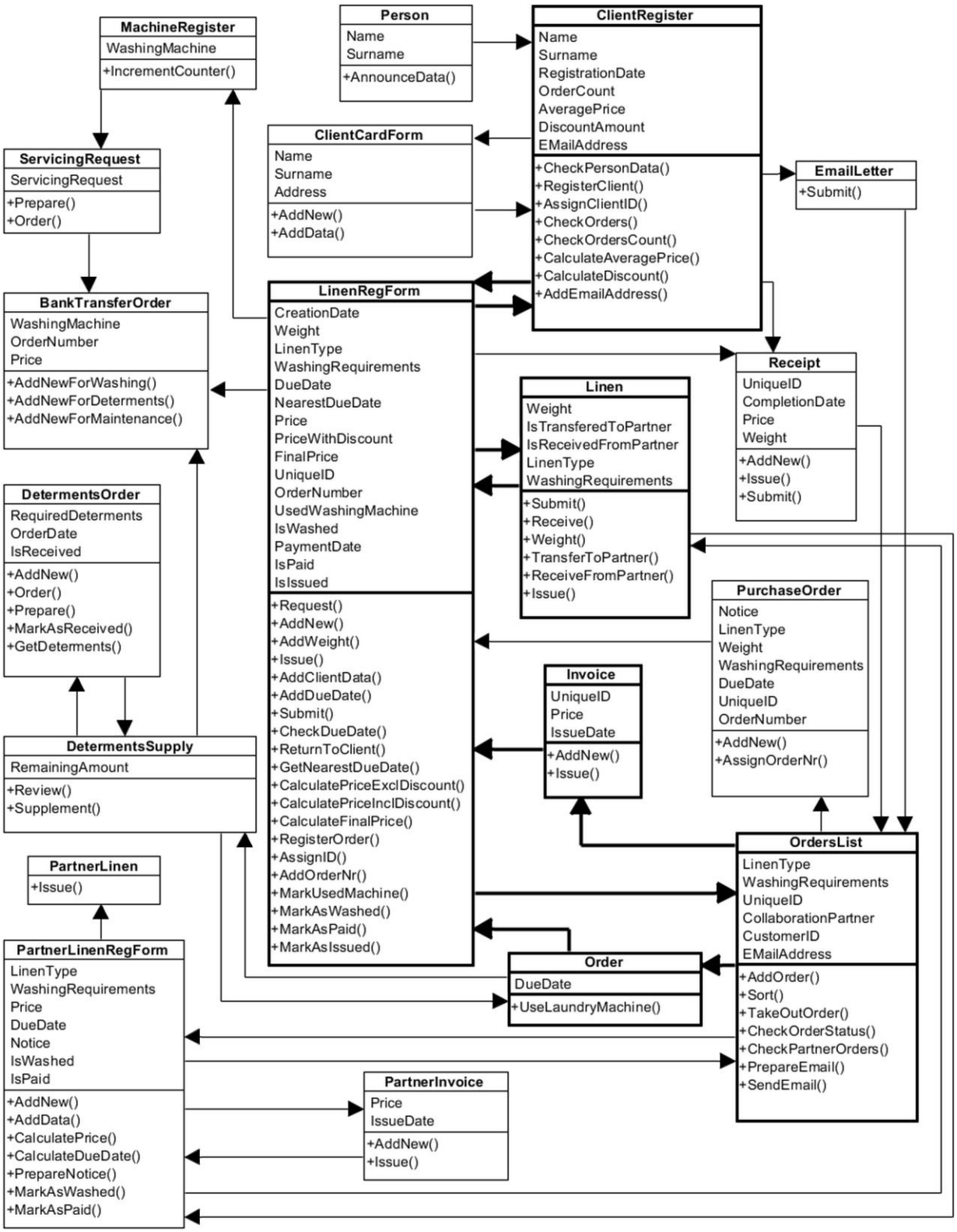


Sequence diagram

“Monitoring the usage of washing machines and ordering servicing for them”



LAUNDRY SYSTEM TOPOLOGICAL CLASS DIAGRAM



SIA Lattelecom Technology
Vienotais reģ. nr. 40003016728
PVN reģ. nr. LV40003016728

Dzirnavu iela 16, Rīga LV-1010
Tālr. +371 67020000
Fakss +371 67020009

E-pasts: technology@lattelecom.lv
www.lattelecom.technology.lv

Appendix 10

April 16, 2012

ACKNOWLEDGEMENT

This is to acknowledge that software intended to collect data from multiple data sources and to store in one unified data storage was designed by using topological functioning modeling approach as recommended by Uldis Donins. The software was developed in the end of year 2010 and the first half of year 2011 in Software Development Department by a development team consisting of four persons: project manager, systems analyst, software architect and developer. At the moment of writing this acknowledgement, the developed software operates in customer's production environment. Software itself is developed by using Microsoft .NET Framework 4.0 and C# programming language.

During the project's elaboration phase Uldis Donins successfully demonstrated the abilities and potentialities of applying Topological functioning model to describe and model functionality of the desired software system. As this was the first pilot project of applying Topological functioning model within Lattelecom Technology, all models were drawn by using Microsoft Visio and later manually translated into a software code. The application of topological functioning modeling allows saving development efforts and time as the solution structure is obtained from analysis models, better evaluating of changes requested by customer, tracing code back to requirements, more easily shifting the maintenance works to other developer team.

The main results of project are presented in the following research paper:

- Donins U., Osis J. Topological Modeling for Enterprise Data Synchronization System: A Case Study of Topological Model-Driven Software Development // Proceedings of the 13th International Conference on Enterprise Information Systems, Volume 3, China, Beijing, June 8-11, 2011. – pp. 87.-96.

_____ / Rinalds Sprogis /
Executive Director, CEO

LATTELECOM TECHNOLOGY
SABIEDRĪBA AR IEROBEŽOTU ATBILDĪBU

_____ / Aivis Students /
Head of Software Development Department

Appendix 11

SPECIFICATION OF ENTERPRISE DATA SYNCHRONIZATION SYSTEM

Informal System Description

Scheduler every five minutes **reads** *configuration data* from *configuration file*. *Configuration data* **includes** following *parameters*: connection information of input data source, username and password for reading input data, flag to indicate if data should be taken from input data source, time at which to make import from input data source, connection information of target data source, username and password for editing data in target data source, path to import files folder, path to log folder.

After *configuration data* **is read**, *scheduler* **checks** if import from *source data base* **should be performed**. *Import* from *source data base* **is performed** at specified time which is given in *configuration data* as *parameter*. If import **should be performed** from *source data base*, then *scheduler* **reads** all data from *source data base* by using *query statement* given in *configuration file*. After all *data* **is read**, *scheduler* **checks** if read data structure is according to specification. Data from source data bases has following structure: surname, forename, job title, address, e-mail address, telephone number, gender, start date, expiry date, department, and company code. If data structure is according to specification, then *scheduler* **puts** the read data into temporal *internal table*. After converting read data to temporal *internal table* every *row* from this table **is imported** into *target data base*.

After *configuration data* is read and import from *source data bases* **is performed** (if needed), *scheduler* **checks** *import folder*. If *CSV file* (the *import file*) **is found** in that folder, *scheduler* **reads** the *import file*. *Import file* has following structure: surname, forename, job title, address, e-mail address, telephone number, gender, start date, expiry date, department, and company code. *Scheduler* then **checks** that read *import file* **corresponds** to predefined *import file* structure. If *import file* structure is according to specification, then *scheduler* **converts** the read data into temporal *internal table*. After converting read data into temporal *internal table* every *row* from this table **is imported** into *target data base*. If *import file* structure is not prepared according to specification, the *import file* **is skipped**, **moved** to *processed files folder* and a *log file* **is created** in *log files folder* stating that particular *import file* was not imported into *target data base*.

For every *row scheduler* **checks** if *data* from a particular *row* already **exists** in *target data base*. If *data* from the particular *row* **exists** then **update** of *existing data* **is performed** in *target data base*. If *data* from the particular *row* **does not exist** then **insert** of *new data* **is performed** in *target data base*. By updating or inserting data in *target data base scheduler* **prepares** *log file* in *log files folder* for every import file and for every time data is imported from *source data base*. In *log file* **is logged** every data *row* from *temporal internal table* in order to unify *log files* from different data sources. For every *row* from *source data* an *import status* **is logged**. There are two *import statuses*: *successful* and *error*. *Successful* status **is logged** when *import* **is successful** for particular row. *Error* status **is logged** when *import* **is not successful** for particular row. If *error* **is logged** then *error description* also **is logged** in order to allow *data import manager* to watch for *un-imported data*. After data import is completed the *log file* **is archived**. After importing data from *import file*, the *import file* **is moved** to *processed files folder*.

Functional Requirements

ID	Requirement
FR1	Employee data synchronization should be done between input data sources and target data source. This requirement includes requirements FR1/[1-6]
FR1/1	By starting synchronization process a configuration information should be taken from configuration file
FR1/2	If needed, data from source data base should be taken
FR1/3	Data should be taken from import files in CSV format
FR1/4	If import CSV file is with wrong data structure, the processing of particular file should be skipped and faulty import file should be logged
FR1/5	All data obtained from either source data base or import files should be placed in target data base
FR1/6	When importing data in target data base all rows from source data should be logged together with import status for each particular data row.

Nonfunctional requirements

ID	Requirement
NFR1	Employee data synchronization mechanism should be implemented in a way that it runs every 5 minutes after previous data synchronization has been completed
NFR2	Synchronization mechanism should run using Microsoft .NET Framework 4.0 [79]

Appendix 12

FUNCTIONAL FEATURES OF ENTERPRISE DATA SYNCHRONIZATION SYSTEM

ID	Object action	Precondition	Entity	Object	Inner or External
1	Creating data synchronization parameters		Data import manager		External
2	Acquiring synchronization parameters		Configuration	Configuration	Inner
3	Checking if import from source data base should be performed		Configuration	Configuration	Inner
4	Creating data in source data base		Source data base		External
5	Reading all data from source data base	If import should be performed from source data base	Scheduler	SourceDataSource	Inner
6	Checking if read data structure is according to specification		Scheduler	Scheduler	Inner
7	Putting the read data into temporal internal table	If data structure is according to specification	Scheduler	Scheduler	Inner
8	Importing every row from internal table into target data base		Scheduler	Scheduler	Inner
9	Checking import folder		Scheduler	ImportFolder	Inner
10	Creating CSV import file		Import file		External
11	Reading the import file	If CSV file (the import file) is found in import folder	Scheduler	ImportFile	Inner
12	Checking if import file data structure is according to specification		Scheduler	Scheduler	Inner
13	Converting the read data from import file into temporal internal table	If import file structure is according to specification	Scheduler	Scheduler	Inner
14	Skipping importing of import file	If import file structure is not prepared	Scheduler	Scheduler	Inner

ID	Object action	Precondition	Entity	Object	Inner or External
		according to specification			
15	Moving import file to processed files folder		Scheduler	ImportFile	Inner
16	Creating log file in log files folder		Scheduler	Logger	Inner
17	Writing into log file that particular import file was not imported into target data base		Scheduler	Logger	External
18	Receiving log file for unimported CSV file		Data import manager		External
19	Checking if data from a particular row already exists in target data base		Scheduler	TargetDataSource	Inner
20	Updating existing data in target data base	If data from the particular row exists	Scheduler	TargetDataSource	External
21	Receiving updated information		Target data base		External
22	Insert new data in target data base	If data from the particular row does not exist	Scheduler	TargetDataSource	External
23	Receiving new information		Target data base		External
24	Creating log file in log files folder for import file processing	If data is read from import file	Scheduler	Logger	Inner
25	Logging data row from temporal internal table		Scheduler	Logger	Inner
26	Logging Successful status	If import is successful for particular row	Scheduler	Logger	Inner
27	Logging Error status	If import is not successful for particular row	Scheduler	Logger	Inner
28	Logging error description	If error is logged	Scheduler	Logger	Inner
29	Archiving log file	If data import is completed	Scheduler	Logger	External
30	Receiving archived import log file		Data import manager		External

Appendix 13

SELF-EVALUATION QUESTIONNAIRE

Name, Surname: _____

Age: _____

Education:

	University	**Obtained degree**
1.		
2.		

Additional education (e.g., professional courses, certifications, etc):

1.	
2.	
3.	
4.	

Modeling languages and notations:
Evaluate knowledge level in scale from 1 to 5, where: 1 – only theoretical knowledge or little experience, 5 – great knowledge.

	Modeling language/ notation	**Knowledge level**
1.	Grapes BM (GRADE)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
2.	Business Process Modeling Notation (BPMN)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
3.	Data Flow Diagram (DFD)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
4.	Entity-Relationship Diagram (ER or ERD)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
5.	Unified Modeling Language (UML)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
6.	Executable UML (xUML)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
7.	KAOS	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
8.	i*	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
9.	Enterprise Knowledge Development (EKD)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
10.		☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5

UML diagrams:
Evaluate knowledge level in scale from 1 to 5, where: 1 – only theoretical knowledge or little experience, 5 – great knowledge.

	UML diagram type	**Knowledge level**
1.	Class diagram	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
2.	Component diagram	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
3.	Package diagram	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
4.	Activity diagram	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
5.	Use case diagram	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
6.	State diagram	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
7.	Sequence diagram	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
8.	Communication diagram	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
9.		☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
10.		☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5

Software development lifecycles:
Evaluate knowledge level in scale from 1 to 5, where: 1 – only theoretical knowledge or little experience, 5 – great knowledge.

	Software development lifecycle	**Knowledge level**
1.	Waterfall (classical)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
2.	Iterative waterfall	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
3.	V-type model	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
4.	Prototyping	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
5.	Rational Unified Process	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
6.	Model Driven Architecture	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
7.	Agile lifecycles (except Extreme programing)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
8.	Extreme programming (XP)	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
9.		☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
10.		☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5

Programming languages:
Evaluate knowledge level in scale from 1 to 5, where: 1 – only theoretical knowledge or little experience, 5 – great knowledge.

	Programming language	**Knowledge level**
1.	C	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
2.	C++	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
3.	Visual Basic	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
4.	Visual Basic .NET	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
5.	C#	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
6.	Java	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
7.	Python	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
8.	PHP	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
9.	Delphi	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
10.		☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5

Modeling tools:
Evaluate knowledge level in scale from 1 to 5, where: 1 – only theoretical knowledge or little experience, 5 – great knowledge.

	Modeling tool	**Knowledge level**
1.	GRADE	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
2.	IBM Rational Rose	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
3.	Eclipse	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
4.	Microsoft Visio	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
5.	Sparx Enterprise Architect	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
6.	PowerDesigner	☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
7.		☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5

Employer: _____

Position: _____

Number of employees: _____

Largest IT project (as participant): number of members _____, months _____

Work experience in software development (years): _____

Main responsibilities at employer regarding software development tasks
(strike out unnecessary)**:**

	Responsibility	**Technology/Approach/Language applied**
1.	Software programming	
2.	Designing of software systems (including modeling)	
3.	Data base programming	
4.	Designing of data bases	
5.	Data mining from data bases	
6.	System analysis	
7.	Software testing	
8.		
9.		
10.		

Technology/Approach/Language applied in practice:

	Technology/Approach/Language
1.	
2.	
3.	
4.	
5.	

Appendix 14

OMG-CERTIFIED UML PROFESSIONAL CERTIFICATE

BIBLIOGRAPHY

[1] Alksnis G. Formal Specification Languages and Category Theory Within the Framework of MDA// Computer Science, Applied Computer Systems, Vol.26, Nr.5, Scientific Proceedings of Riga Technical University, Riga, Latvia: RTU Publishing, 2006. - pp. 33-41

[2] Ambler S. Agile Modeling: Effective Practices for EXtreme Programming and the Unified Process. - New York, USA: John Wiley & Sons, 2002. - 400 p.

[3] Ambler S. Elements of UML 2.0 Style. - New York, USA: Cambridge University Press, 2005. - 200 p.

[4] Arlow J., Neustadt I. UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design. - Upper Saddle River, NJ, USA: Addison-Wesley, 2nd ed., 2005. - 624 p.

[5] Asnina E. Formalization of Problem Domain Modeling within Model Driven Architecture. Doctoral thesis. - Riga, Latvia: RTU Publishing house, 2006. - 195 p.

[6] Asnina E. The Formal Approach to Problem Domain Modelling Within Model Driven Architecture// Proceedings of the 9th International Conference "Information Systems Implementation and Modelling" (ISIM'06), Přerov, Czech Republic: Jan Štefan MARQ, 2006. - pp. 97-104

[7] Asnina E. A Formal Holistic Outline for Domain Modeling// Advances in Databases and Information Systems. 13th East-European Conference, ADBIS 2009: Associated Workshops and Doctoral Consortium, Local Proceedings. - Riga, Latvia: JUMI Publishing House Ltd., 2009. - pp. 400-407

[8] Asnina E., Gulbis B., Osis J., Alksnis G., Donins U., Slihte A. Backward Requirements Traceability within the Topology-based Model Driven Software Development// Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 36-45

[9] Barr M., Wells C. Category theory for computer science. – Hertfordshire, United Kingdom: Prentice Hall International, 2nd ed., 1995. - 344 p.

[10] Batra D. Unified Modeling Language (UML) Topics: The Past, the Problems, and the Prospects// Journal of Database Management. - 2008. - 19(1) - pp. 1-7

[11] Batra D., Satzinger J. Contemporary Approaches and Techniques for the Systems Analyst// Journal of Information Systems Education. - 2006. - 17(3) - pp. 257–265

[12] Baumeister H., Koch N., Mandel L. Towards a UML Extension for Hypermedia Design// «UML»'99: The Unified Modeling Language. Beyond the Standard (Lecture Notes in Computer Science, Vol. 1723). – Berlin, Germany: Springer, 1999. - pp. 614-629

[13] Booch G. Object Oriented Analysis and Design with Applications. - Upper Saddle River, NJ, USA: Addison-Wesley, 2nd ed., 1993. - 608 p.

[14] Booch G., Maksimchuk R., Engel M., Young B., Conallen J., Houston K. Object-oriented analysis and design with applications. - Upper Saddle River, NJ, USA: Addison-Wesley, 3rd ed., 2007. - 720 p.

[15] Booch G., Rumbaugh J., Jacobson I. The Unified Modeling Language User Guide. - Upper Saddle River, NJ, USA: Addison-Wesley, 2nd ed., 2005. - 475 p.

[16] Breu R., Hinkel U., Hofmann C., Klein C., Paech B., Rumpe B., Thurner V. Towards a Formalization of the Unified Modeling Language // ECOOP'97 – Object-Oriented Programming, 11th European

Conference (Lecture Notes in Computer Science, Vol. 1241). – Berlin, Germany: Springer, 1997. - pp. 344-366

[17] Burton-Jones A., Meso P. Conceptualizing Systems for Understanding: An Empirical Test of Decomposition Principles in Object-Oriented Analysis// Information Systems Research. - 2006. - 17(1) - pp. 38-60

[18] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. Pattern-Oriented Software Architecture: A System of Patterns. - West Sussex, England: John Wiley & Sons Ltd., 1996. - 476 p.

[19] Darwin I. Java Cookbook. - Sebastopol, USA: O'Reilly, 2nd ed., 2004. - 864 p.

[20] DeLoach S., Hartrum T. A Theory-Based Representation for Object-Oriented Domain Models// IEEE Transactions on Software Engineering. - 2000. - Volume 26, Issue 6. - pp. 500-517

[21] Desel J., Juhás G. "What is a Petri Net?" Informal Answers for the Informed Readers// Unifying Petri Nets, Advances in Petri Nets (Lecture Notes in Computer Science, Vol. 2128). - Berlin, Germany: Springer, 2001. - pp. 1-25

[22] Digital Software Magazine: The Software Decision Journal / Internet. - http://www.softwaremag.com/

[23] Diskin Z., Kadish B., Piessens F., Johnson M., Universal Arrow Foundations for Visual Modeling// Theory and Application of Diagrams: First International Conference, Diagrams 2000 (Lecture Notes in Artificial Intelligence, Vol. 1889). - Berlin, Germany: Springer, 2001. - pp. 345-360

[24] Dobing B., Parsons J. Dimensions of UML Diagram Use: Practitioner Survey and Research Agenda// Principle Advancements in Database Management Technologies: New Applications and Frameworks. – Hershey, New York, USA: Information Science Reference, 2010. - pp. 271-290

[25] Donins U. Semantics of Logical Relations in Topological Functioning Model// Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2012) – 2012. (To be published)

[26] Donins U. Software Development with the Emphasis on Topology// Advances in Databases and Information Systems (Lecture Notes in Computer Science, Vol.5968). - Berlin, Germany: Springer-Verlag, 2010. - pp. 220-228

[27] Doniņš U. Topological business systems modeling and software systems design. - Riga, Latvia: RTU Publishing house, 2011. - 65 p. (in Latvian)

[28] Donins U., Osis J. Reconciling Software Requirements and Architectures within MDA// Scientific Proceedings of Riga Technical University, Computer Science (Series 5), Applied Computer Systems (Vol. 38). - Riga, Latvia: RTU Publishing house, 2009. - pp. 84-95

[29] Donins U., Osis J. Topological Modeling for Enterprise Data Synchronization System: A Case Study of Topological Model-Driven Software Development// Proceedings of the 13th International Conference on Enterprise Information Systems, Volume 3. - Beijing, China: SciTePress, 2011. - pp. 87-96

[30] Donins U., Osis J., Asnina E., Jansone A. Formal Analysis of Objects State Changes and Transitions// Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2012) – 2012. (To be published)

[31] Donins U., Osis J., Slihte A., Asnina E., Gulbis B. Towards the Refinement of Topological Class Diagram as a Platform Independent Model// Proceedings of the 3rd International Workshop on Model-

Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 79-88

[32] D'Souza D., Wills A. Objects, Components, and Frameworks with UML: The Catalysis Approach. - Upper Saddle River, NJ, USA: Addison-Wesley, 1998. - 816 p.

[33] Erickson J., Siau K. Theoretical and Practical Complexity of Modeling Methods// Communications of the ACM. - 2007. - 50(8) - pp. 46-51

[34] Evans A., Kent S. Core Meta-Modelling Semantics of UML: The pUML Approach// «UML»'99: The Unified Modeling Language. Beyond the Standard (Lecture Notes in Computer Science, Vol. 1723). - Berlin, Germany: Springer, 1999. - pp. 140-155

[35] Evermann J., Wand Y. Ontological Modeling Rules For UML: An Empirical Assessment// Journal of Computer Information Systems. - 2006. - 46(5) - pp. 14-29

[36] Evermann J., Wand Y. Towards Ontologically-Based Semantics for UML Constructs// Conceptual Modeling – ER 2001, 20th International Conference on Conceptual Modeling (Lecture Notes in Computer Science, Vol. 2224). - Berlin, Germany: Springer, 2001. - pp. 341-354

[37] Fenton N., Pfleeger S. Software Metrics: A Rigorous and Practical Approach. - Scottsdale, Arizona, USA: Coriolis Group, 2nd ed., 1996. – 649 p.

[38] Filev A., Loton T., McNeish K., Schoellmann B., Slater J., Wu C. Professional UML Using Visual Studio.Net: Unmasking Visio for Enterprise Architects. - Indianapolis, IN, USA: Wiley Publishing Inc., 2002. - 368 p.

[39] Fowler M. Why use the UML?// Software Development. - 1998. - Volume 6, Issue 3

[40] Fowler M. Patterns of Enterprise Application Architecture. - Upper Saddle River, NJ, USA: Addison-Wesley, 2002. - 560 p.

[41] Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. - Upper Saddle River, NJ, USA: Addison-Wesley, 3rd ed., 2003. - 208 p.

[42] France R., Rumpe B. «UML»'99: The Unified Modeling Language. Beyond the Standard (Lecture Notes in Computer Science, Vol. 1723). - Berlin, Germany: Springer, 1999. - 724 p.

[43] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. - Upper Saddle River, NJ, USA: Addison-Wesley, 1994. - 416p.

[44] Grünbacher P., Egyed A., Medvidovic N. Reconciling software requirements and architectures with intermediate models // Software and Systems Modeling. - 2004. - Volume 3, No 3. - pp. 235-253

[45] Grundspenkis J. Fault Localisation Based on Topological Feature Analysis of Complex System Model// Diagnostics and Identification. - Riga: Zinatne, 1974. - pp. 38-48 (in Russian)

[46] Grundspenkis J. Reasoning Supported by Structural Modelling// Intelligent Design, Intelligent Manufacturing and Intelligent Management. - Kaunas, Lithuania: Technologija, 1999. - pp. 57-100

[47] Grundspenkis J. Structural Modelling of Complex Technical Systems in Conditions of Incomplete Information: A Review// Modern Aspects of Management Science. - Riga, Latvia: RTU Publishing house, 1997. - pp. 111-136

[48] Grundspenkis J. Structural Modelling with ASMOS in the Early Stages of Design// Software for Manufacturing. – Amsterdam, Holland: North-Holland Publishing Company, 1989. - pp. 229-239

[49] Grundspenkis J. The Synthesis and Analysis of Structure in Computer Aided Design// Computer Applications in Production and Engineering: Proceedings of the First International Conference. – Amsterdam, Holland: North-Holland Publishing Company, 1983. - pp. 301-316

[50] Grundspenkis J., Blumbergs A. Investigation of Complex System Topological Model Structure for Analysis of Failures// Issues of Technical Diagnosis. - Rostov-on-Don, Russia: Rostov Institute of Building Engineering, 1981. - pp. 41-48 (in Russian)

[51] Harel D. Statecharts: A visual formalism for complex systems// Science of Computer Programming. - 1987. - Volume 8, Issue 3. - pp. 231-247

[52] He X. Formalizing UML Semantics// 25th Annual International Computer Software and Applications Conference (COMPSAC'01). - Chicago, Illinois, USA: IEEE Computer Society, 2001. - pp. 277

[53] International Organization for Standardization (ISO): ISO/IEC/IEEE 42010:2011 "Systems and software engineering -- Architecture description", 2011. - 37 p.

[54] Jacobson I., Christerson M., Jonsson P., Overgaard G. Object-Oriented Software Engineering: A Use Case Driven Approach. - Upper Saddle River, NJ, USA: Addison-Wesley, 1992. - 552 p.

[55] Jones C. Positive and Negative Innovations in Software Engineering// International Journal of Software Science and Computational Intelligence. - 2009. - Volume 1, Issue 2. - pp. 20-30

[56] Karpics I., Markovics Z. Development and Evaluation of Normal Performance Recovery Method of a Functional System // Proceedings of 9th IEEE International Symposium on Applied Machine Intelligence and Informatics (SAMI), 2011, Slovakia, Smolenice, 2011. - pp. 171-175

[57] Kent S. The Unified Modeling Language// Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches. - Cambridge, England: Cambridge University Press, 2001. - pp. 126-151

[58] Kim S., Carrington D. Formalizing the UML Class Diagram Using Object-Z// «UML»'99: The Unified Modeling Language. Beyond the Standard (Lecture Notes in Computer Science, Vol. 1723). - Berlin, Germany: Springer, 1999. - pp. 83-98

[59] Kleppe A., Warmer J., Bust W. MDA Explained. The Model Driven Architecture: Practice and Promise. - Upper Saddle River, NJ, USA: Addison-Wesley, 2003. - 192 p.

[60] Kobryn C. UML 2001: A Standardization Odyssey// Communications of the ACM. - 1999. - Volume 42, Issue 10. - pp. 29-37

[61] Kruchten P. The 4+1 view model of architecture// IEEE Software. - 1995. - Volume 12, Issue 6. - pp. 42-50

[62] Kruchten P. The Rational Unified Process: An Introduction. - Upper Saddle River, NJ, USA: Addison-Wesley, 2003. - 336 p.

[63] Lano K., Kolahdouz-Rahimi S. Model-Driven Development of Model Transformations// Theory and Practice of Model Transformations (Lecture Notes in Computer Science, Volume 6707). - Berlin, Germany: Springer-Verlag, 2011. - pp. 47-61

[64] Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. - Upper Saddle River, NJ, USA: Prentice Hall, 3rd ed., 2005. - 736 p.

[65] Lazar I., Motogna S., Parv B., Lazar C. Realizing Use Cases for Full Code Generation in the Context of fUML// Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development - Portugal: SciTePress, 2010. - pp. 80-89

[66] Leffingwell D., Widrig D. Managing Software Requirements: a Use Case Approach. Upper Saddle River, NJ, USA: Addison-Wesley, 2nd ed., 2003. - 544 p.

[67] Li D., Li X., Stolz V. QVT-Based Model Transformation using XSLT// ACM SIGSOFT Software Engineering Notes. - 2011. - Volume 36, Issue 1. - pp. 1-8

[68] Loniewski G., Insfran E., Abrahao S. A systematic Review of the Use of Requirements Engineering Techniques in Model-Driven Development// Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science, Volume 6395). - Berlin, Germany: Springer-Verlag, 2010. - pp. 213-227

[69] Loton T. UML Software Design with Visual Studio 2010. – Breinigsville, PA, USA: LOTONtech Limited, 2010. - 136 p.

[70] Loudon, K. C++ Pocket Reference. - Sebastopol, USA: O'Reilly, 2003. - 138 p.

[71] Luger, G.: Artificial Intelligence: Structures and Strategies for Complex Problem Solving. - Upper Saddle River, NJ, USA: Addison-Wesley, 5th ed., 2005. - 928 p.

[72] Markovica I., Markovics Z. Mathematical Model of Pathogenesis of Hard Differentiable Diseases// Cybernetics and Diagnostics, Volume 4. - Riga: Zinatne, 1970. - pp. 21-28 (in Russian)

[73] Markovitch Z., Markovitcha I. Modelling as a Tool for Therapy Selection// Proceedings of the 14th European Simulation Multiconference "Simulation and Modelling". - 2000. - pp. 621-623

[74] Markovitch Z., Rekners Ya. Synthesis of Systems Model on Basis of Topological Minimodels// Automatic Control and Computer Sciences. - 1998. - Volume 32, Issue 3. - pp. 59-66

[75] Markovitch Z., Stalidzans E. Expert Based Model Building Using Incidence Matrix and Topological Models// Proceedings of the 12th European Simulation Symposium "Simulation in Industry 2000". - 2000. - pp. 328-332

[76] Mellor S., Balcer M. Executable UML: A Foundation for Model-Driven Architecture. - Upper Saddle River, NJ, USA: Addison-Wesley, 2002. - 416 p.

[77] Mens T., Van Gorp P. A Taxonomy of Model Transformation// Electronic Notes in Theoretical Computer Science. - 2006. - Volume 152. - pp. 125-142

[78] Miller J., Mukerji J. (editors): MDA Guide Version 1.0.1 / Internet. - http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf

[79] Nagel C., Evjen B., Glynn J., Watson K., Skinner M. Professional C# 4.0 and .NET 4. - New York, USA: John Wiley & Sons, 2010. - 1536 p.

[80] Nielsen M., Havelund K., Wagner K., George C. The RAISE Language, Method and Tools// Formal Aspects of Computing. - 1989. - Volume 1 Issue 1. - pp 85-114

[81] Ņikiforova O. System Modeling in UML with Two-Hemisphere Model Driven Approach// Scientific Proceedings of Riga Technical University, Computer Science (Series 5), Applied Computer Systems (Volume 43). - Riga, Latvia: RTU Publishing house, 2010. - pp. 37-44

[82] OMG: Common Warehouse Metamodel Specification Version 1.1 / Internet. - http://www.omg.org/spec/CWM/1.1/PDF/

[83] OMG: Unified Modeling Language Specification Version 1.1 / Internet. - http://www.omg.org/cgi-bin/doc?ad/97-08-11

[84] OMG: Unified Modeling Language Specification Version 1.3 / Internet. - http://www.omg.org/spec/UML/1.5/PDF/

[85] OMG: Meta Object Facility (MOF) Core Specification Version 2.0 / Internet. - http://www.omg.org/spec/MOF/2.0/PDF/

[86] OMG: UML Testing Profile Version 1.0 / Internet. - http://www.omg.org/spec/UTP/1.0/PDF/

[87] OMG: Service Oriented Architecture Modeling Language (SoaML) / Internet. - http://www.omg.org/spec/SoaML/1.0/Beta2/PDF

[88] OMG: Unified Modeling Language Infrastructure Version 2.4.1 / Internet. - http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/

[89] OMG: Unified Modeling Language Superstructure Version 2.4.1 / Internet. - http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

[90] OMG: OMG Systems Modeling Language (OMG SysML) / Internet. - http://www.omg.org/spec/SysML/1.2/PDF

[91] OMG: OMG Formally Released Versions of UML / Internet. - http://www.omg.org/spec/UML/

[92] Olive A. Conceptual Modeling of Information Systems. - Berlin, Germany: Springer, 2007. - 455 p.

[93] Osis J. Extension of Software Development Process for Mechatronic and Embedded Systems// Proceeding of the 32nd International Conference on Computer and Industrial Engineering. - Limerick, Ireland: University of Limerick, 2003. - pp. 305-310

[94] Osis J. Formal Computation Independent Model within the MDA Life Cycle// International Transactions on Systems Science and Applications. - 2006. - Volume 1, Number 2. - pp. 159-166

[95] Osis J. Investigating Troubles of Complex System Functioning and Category Theory // Cybernetic and Diagnosis, Volume. 4. - Riga: Zinatne, 1970. - pp. 15-20 (in Russian)

[96] Osis J. Mathematical Description of Complex System Functioning// Cybernetic and Diagnosis, Volume 4. - Riga: Zinatne, 1970. - pp. 7-14 (in Russian)

[97] Osis Ya. Scientific Research on System Modelling at RTU// Scientific Proceedings of Riga Technical University, Computer Science (Series 5), Applied Computer Systems (Volume 8). - Riga, Latvia: RTU Publishing house, 2001. - pp. 6-17 (in Latvian)

[98] Osis J. Some Questions of Microprogramming Optimization using Topological Model Properties // Cybernetic Methods in the Diagnosis - Riga: Zinatne, 1973. - pp. 30-34 (in Russian)

[99] Osis J. The Topological Model of System Functioning// Automatics and Computer Science, Volume 6. - Riga, Latvia, 1969. - pp. 44-50 (in Russian)

[100] Osis J., Asnina E. Enterprise Modeling for Information System Development within MDA// Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008). - Chicago, Illinois, USA: IEEE Computer Society, 2008. - pp. 491

[101] Osis J., Asnina E. Model-Driven Domain Analysis and Software Development: Architectures and Functions. – Hershey, New York, USA: IGI Global, 2011. - 487 p.

[102] Osis J., Donins U. An Innovative Model Driven Formalization of the Class Diagrams// Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2009). – Portugal: INSTICC Press, 2009. - pp. 134-145

[103] Osis J., Donins U. Formalization of the UML Class Diagrams// Evaluation of Novel Approaches to Software Engineering (Communications in Computer and Information Science (CCIS), Volume 69). - Berlin, Germany: Springer-Verlag, 2010. - pp. 180-192

[104] Osis J, Donins U. Modeling Formalization of MDA Software Development at the Very Beginning of Life Cycle// Advances in Databases and Information Systems. 13th East-European Conference, ADBIS 2009: Associated Workshops and Doctoral Consortium, Local Proceedings. - Riga, Latvia: JUMI Publishing House Ltd., 2009. - pp. 48-61

[105] Osis J., Donins U. Platform Independent model Development by Means of Topological Class Diagrams// Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development - Portugal: SciTePress, 2010. - pp. 13-22

[106] Osis J., Gefandbein J., Markovitch Z., Novozhilova N. Diagnosis based on graph models. (By the Examples of Aircraft and Automobile Mechanisms). - Moscow, Russia: Transport, 1991. - 244 p. (in Russian)

[107] Osis J., Silins J. Topological Function-Architecture Co-Design of Embedded Systems// Advances in Databases and Information Systems. 13th East-European Conference, ADBIS 2009: Associated Workshops and Doctoral Consortium, Local Proceedings. - Riga, Latvia: JUMI Publishing House Ltd., 2009. - pp. 424-431

[108] Osis J., Šlihte A. Transforming Textual Use Cases to a Computation Independent Model// Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development - Portugal: SciTePress, 2010. - pp. 33-42

[109] Osis J., Sukovskis U., Teilans A. Business Process Modeling and Simulation Based on Topological Approach// Proceedings of the 9th European Simulation Symposium and Exhibition. - Passau, Germany, 1997. - pp. 496-501

[110] Owre S., Rushby J., Shankar N. PVS: A Prototype Verification System// 11th International Conference on Automated Deduction (Lecture Notes in Artificial Intelligence, Volume 607). -Berlin, Germany: Springer, 1992. - pp. 748-752

[111] Pardillo J. A Systematic Review on the Definition of UML Profiles// Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science, Volume 6394). - Berlin, Germany: Springer-Verlag, 2010, - pp. 407-422

[112] Podeswa H. UML for the IT Business Analyst. - Boston, MA, USA: Course Technology PTR, 2nd ed., 2009. - 372 p.

[113] Randolph N., Gardner D., Anderson C., Minutillo M. Professional Visual Studio 2010. - New York, USA: John Wiley & Sons, 2010. - 1224 p.

[114] Rumbaugh J., Blaha M., Premerlani W., Eddy F. Lorensen W. Object-Oriented Modeling and Design. - Englewood Cliffs, NJ: Prentice Hall, 1991. - 528 p.

[115] Rumbaugh J., Jacobson I., Booch G. The Unified Modeling Language Reference Manual. - Upper Saddle River, NJ, USA: Addison-Wesley, 2nd ed., 2004. - 721 p.

[116] Scott K. The Unified Process Explained. - Upper Saddle River, NJ, USA: Addison-Wesley, 2001. - 208 p.

[117] Sejans, J., Nikiforova, N. Practical Experiments with Code Generation from the UML Class Diagram// Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 57-67

[118] Shalloway A., Trott J. Design Patterns Explained: A New Perspective on Object-Oriented Design. - Upper Saddle River, NJ, USA: Addison-Wesley, 2nd ed., 2004. – 480 p.

[119] Shekhovstov V. On the Evolution of Quality Conceptualization Techniques// The Evolution of Conceptual Modeling (Lecture Notes in Computer Science, Volume 6520). – Berlin, Germany: Springer-Verlag, 2011. - pp. 117-136

[120] Siau K., Cao Q. Unified Modeling Language (UML) - a Complexity Analysis// Journal of Database Management. - 2001. - Volume 12, Issue 1. - pp. 26-34

[121] Siau K., Cao, Q. How Complex Is the Unified Modeling Language?// Advanced Topics in Database Research. - 2002. - Volume 1. - pp. 294-306

[122] Siau K., Loo P. Identifying Difficulties in Learning UML// Information Systems Management. - 2006. - Volume 23, Issue 3. - pp. 43-51

[123] Simons A., Graham I. 37 Things that Don't Work in Object-Oriented Modeling with UML// Proceedings of ECOOP 98 Workshop on Precise Behavioral Semantics. - Universitat Muchen, 1998. - pp. 209-232

[124] Slihte A., Osis J., Donins U. Knowledge Integration for Domain Modeling// Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 46-56

[125] Slihte A., Osis J., Donins U., Asnina, E., Gulbis, B. Advancements of the Topological Functioning Model for Model Driven Architecture Approach// Proceedings of the 3rd International Workshop on Model-Driven Architecture and Modeling-Driven Software Development. - Beijing, China: SciTePress, 2011. - pp. 91-100

[126] Spivey J. The Z Notation: A Reference Manual. - Prentice Hall, 2nd ed., 1992. - 150 p.

[127] Stevens P., Pooley R. Using UML: Software Engineering with Objects and Components. - Harlow, England: Addison-Wesley, 2nd ed., 2005. - 250 p.

[128] Van Der Straeten R., Mens T., Simmonds J., Jonckers V. Using Description Logic to Maintain Consistency between UML Models// «UML» 2003 – The Unified Modeling Language. Modeling Languages and Applications (Lecture Notes in Computer Science, Volume 2863). – Berlin, Germany: Springer-Verlag, 2003. - pp. 326-340

[129] Szlenk M. UML Static Models in Formal Approach// Balancing Agility and Formalism in Software Engineering (Lecture Notes in Computer Science, Volume 5082). - Berlin, Germany: Springer-Verlag, 2008. - pp. 129-142

[130] Turner M. Microsoft Solutions Framework Essentials: Building Successful Technology Solutions. - Redmond, Washington, USA: Microsoft Press, 2006. - 300 p.

[131] Valkovska I., Grundspenkis J. Representation of Complex Agents by Frames for Simulation of Internal Relationships in Structural Modelling// Proceedings of the 19th European Conference on Modelling and Simulation (ECMS 2005). - Riga, Latvia: RTU Publishing house, 2005. - pp. 151-156

[132] Warmer J., Kleppe A. The Object Constraint Language: Getting Your Models Ready for MDA. - Upper Saddle River, NJ, USA: Addison-Wesley, 2nd ed., 2003. - 240 p.

[133] Xueming L., Parsons J. Ontological Semantics for the Use of UML in Conceptual Modeling// ER (Tutorials, Posters, Panels & Industrial Contributions). - 2007. - pp. 179-184

[134] Zhao Y., Zong-Yuan Y., Xie J. Pi-Calculus Based Assembly Mechanism of UML State Diagram and Validation of Model Refinement// Proceedings of International Conference on Electronic Computer Technology 2009 (ICECT 2009). - Macau, China, 2009. - pp 604-609