THESIS ON INFORMATICS AND SYSTEM ENGINEERING C67

Archetypes Based Techniques for Development of Domains, Requirements and Software

Towards LIMS Software Factory

GUNNAR PIHO

TUT PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Informatics

Dissertation was accepted for the defence of the degree of Doctor of Philosophy in Engineering on 14th November 2011.

Supervisor: Professor Jaak Tepandi, Department of Informatics, Tallinn University of Technology

Opponents: Professor Janis Grundspenkis, Faculty of Computer Science and Computer Systems, Riga Technical University

Professor Marlon Dumas, Institute of Computer Science, University of Tartu

Defence of the thesis: 19th December 2011

Declaration:

Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

Signature of candidate:

Date:

Copyright: Gunnar Piho 2011

ISSN 1406-4731
ISBN 978-9949-23-211-6 (publication)
ISBN 978-9949-23-212-3 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C67

Arhetüüpidel tuginevad tehnikad valdkondade, nõuete ja tarkvara arendamiseks

LIMS-i tarkvaravabriku näitel

GUNNAR PIHO

TTÜ
KIRJASTUS

To my family

To my parents

To my teachers

ABSTRACT

Software development is a complex task. Developing dependable software is expensive, takes time and requires knowledge, tools and also techniques. In order to alleviate growing demand for customizable software, a variety of ideas and initiatives (e.g. software product lines, software factories, etc.) for developing software by reusing archetypal components have been proposed.

This work is based on software engineering triptych (from domain model via requirements to software) proposed by Dines Bjørner and on archetypes and archetype pattern base initiative proposed by Arlow and Neustadt. These ideas are used in engineering of domain models for clinical laboratory and in LIMS (Laboratory Information Management System) software development.

Business archetypes and archetype patterns are originally designed and introduced by Jim Arlow and Ila Neustadt. Business archetype patterns (product, party, order, inventory, quantity and rule), composed by business archetypes (person's name, address, phone number, etc.), describe the universe of discourse of businesses as it is, neither referring to the software requirements nor to the software design.

We analysed Arlow and Neustadt's business archetype patterns according to the Zachman Framework and Bjørner's domain analysis methodology. We also compared these archetype patterns with analysis and data model patterns by Hay, Fowler, and Silverston. As a result, the refined and enhanced version of business archetypes and archetype patterns is presented. We propose this refined and enhanced version of archetypes and archetype patterns for engineering of business domains, requirements and software. The clinical laboratory domain model, we designed, is based on this refined and enhanced version of archetypes and archetype patterns. We utilize this clinical laboratory domain model in real life LIMS software development.

The resulted work is archetypes and archetype patterns based techniques for development of domains, requirements and software. In our understanding by using these techniques we can lead software development towards development of software factory. The wider research goal is to develop archetypes and archetype patterns based information systems that software end users, in collaboration with software developers, are able to change safely and easily according to changes in business processes.

KOKKUVÕTE

Tarkvara arendamine on keeruline protsess. Usaldusväärse tarkvara arendamine on kallis, võtab aega ning nõuab teadmisi, töövahendeid ja tehnikaid. Leevendamaks üha kasvavat nõudlust mugandatava tarkvara tootmise järele on viimasel ajal välja arendatud erinevaid arhetüüpsete rakenduste genereerimise ja/või rakenduste komponentidest kokkupanemise ideid ja initsiatiive kas siis tarkvaravabriku või tarkvara tooteliini nime all.

Antud töös on lähtutud Dines Bjørneri tarkvara triptühhoonia (valdkonna mudelist nõuete kaudu korrektse tarkvarani) ning Arlow ja Neustadti arhetüüpide ja arhetüüpmustrite ideedest. Neid ideid on rakendatud laboratooriumi valdkonnamudeli ja sellel valdkonnamudelil põhineva laboratooriumi infosüsteemi arendamisel.

Originaalis Arlow ja Neustadti poolt disainitud äri arhetüüpmustrid (toode, osapool, tellimus, inventar, kvantiteet ja reegel) koosnevad äri arhetüüpidest (inimese nimi, aadress, telefoni number, jne). Sisuliselt on tegemist mudelitega, mis abstraheerivad reaalset ärimaailma nii nagu see on ilma igasuguste viideteta tarkvarale ning tarkvarale esitatavatele nõuetele.

Neid Arlow ja Neustadti poolt pakutud arhetüüpe ja arhetüüpmustreid oleme lähtuvalt Zachmani raamistikust analüüsinud Bjørneri valdkonnaanalüüsi metoodikat kasutades. Ka oleme võrrelnud Arlow ja Neustadti mustreid Hay, Fowleri ja Silverstoni vastavate mustritega. Tulemuseks saime äri arhetüüpide ja arhetüüpmustrite parandatud ja täiendatud versiooni, mida me pakume valdkondade, nõuete ja tarkvara arendamiseks. Neid parandatud arhetüüpe ja arhetüüpmustreid oleme kasutanud meditsiinilaboratooriumi valdkonnamudeli arendamisel. Loodud meditsiinilaboratooriumi valdkonnamudelit aga kasutame reaalse laboratooriumi infosüsteemi arendamisel.

Töö tulemused on esitatud arhetüüpidel põhinevate tehnikatena valdkondade, nõuete ja tarkvara arendamiseks. Me leiame, et arhetüüpidel tuginevate tehnikate abil on võimalik liikuda tarkvaravabrikute arendamise suunas. Kaugemaks eesmärgiks on välja arendada arhetüüpidel ja arhetüüpmustritel tuginevad infosüsteemid, mida lõppkasutajad koos arendajatega on võimelised lihtsalt ja turvaliselt muutma vastavalt muutuvatele ärivajadustele.

ACKNOWLEDGEMENTS

I would like to express my gratitude to many people who supported me while I was working on this dissertation. I thank my supervisor Professor Jaak Tepandi. I thank opponents of current thesis, Professor Janis Grundspenkis from Riga Technical University and Professor Marlon Dumas from University of Tartu. I am grateful to my colleagues from Institute of Informatics (Tallinn University of Technology) and Clinical and Biomedical Proteomic Group (Cancer Research Clinical Centre, Leeds Institute of Molecular Medicine, St James's University Hospital at University of Leeds). Special thanks to Mrs Reet Elling from Tallinn University of Technology for helping me in paperwork during my doctoral studies. I also thank the Tallinn University of Technology, the Estonian Science Foundation, the Estonian Information Technology Foundation, the Estonian Entrepreneurship University of Applied Sciences and University of Leeds for financial support. I thank all of my teachers. I thank my family - my parents Ilmar and Õilme, my wife Sirje and my children Sandra, Laura, Paul and Rasmus.

CONTENTS

ABSTRACT 7
KOKKUVÕTE 8
ACKNOWLEDGEMENTS 9
CONTENTS 10
ABBREVIATIONS 11
1 INTRODUCTION 12
1.1 RESEARCH PROBLEM 12
1.2 RESEARCH APPROACH 14
1.3 CONTRIBUTIONS 15
1.4 HYPOTHESIS 16
1.5 DELIMITATIONS 17
1.6 OUTLINE OF THE THESIS 17
2 ARCHETYPES BASED DEVELOPMENT 18
2.1 FROM DOMAIN VIA REQUIREMENTS TO SOFTWARE 18
2.2 TEST DRIVEN MODELLING 21
2.3 FROM DOMAIN MODEL TO SOFTWARE 29
2.4 VALIDATION AND VERIFICATION 30
2.5 SUMMARY 32
3 MODELS OF ARCHETYPES AND ARCHETYPE PATTERNS 33
3.1 METHODOLOGY 33
3.2 CREATING OF INITIAL MODELS 34
3.3 EVALUATION OF MODELS 41
3.4 FINE TUNING OF MODELS 50
3.5 DEFINITIONS OF MODELS 53
3.6 USING OF MODELS 75
3.7 SUMMARY 82
4 CASE STUDY: CLINICAL LABORATORY SOFTWARE 84
4.1 MOTIVATION FOR LIMS AND LIMS SF DEVELOPMENTS 85
4.2 CLINICAL LABORATORY DOMAIN MODEL 88
4.3 LABORATORY INFORMATION MANAGEMENT SYSTEM (LIMS) 97
4.4 TOWARDS CLINICAL LABORATORY SOFTWARE FACTORY 103
4.5 SUMMARY 117
5 EVALUATION AND ANALYSIS OF ABD 118
5.1 DOMAIN ANALYSIS AND MODELLING 118
5.2 SOFTWARE DEVELOPMENT PROCESSES AND METHODOLOGIES 125
6 CONCLUSION 134
6.1 CONTRIBUTIONS 134
6.2 HYPOTHESIS 135
6.3 FUTURE WORK 138
REFERENCES 140
7 APPENDICES 146
7.1 ORDER LIFECYCLE 146
7.2 USING THE BUSINESS PROCESS ARCHETYPE PATTERN 147
7.3 ELULUGU 163
7.4 CURRICULUM VITAE 164
7.5 LIST OF ARTICLES PUBLISHED BY THE THESIS AUTHOR 165

ABBREVIATIONS

A&AP - Archetypes and Archetype Patterns
ABD - Archetypes Based Development
AP - Archetype Patterns
API - Application Programming Interface
ASTM - American Society for Testing and Materials
CBPG - Clinical and Biomedical Proteomics Group
CGS - The Centimetre-Gram-Second System
CIL - Common Intermediate Language
CIM – Computing Independent Model
CMM – Capability Maturity Model
CMMI – Capability Maturity Model Integration
CRM – Customer Relationship Management
DDD – Domain Driven Design
DM – Domain Model
DSL – Domain Specific Language
DLL – Dynamic Link Library
HL7 – Health Level Seven International
IDE - Integrated Development Environment
IT – Information Technology
LIMS - Laboratory Information Management System
MDA – Model Driven Architecture
MTA - Medical Technical Assistant
OO - Object Oriented
PIM – Platform Independent Model
PSM – Platform Specific Model
RAD - Rapid Application Development
SF – Software Factory
SI – The International System of Units
SPL - Software Product Line
TDD – Test Driven Development
TDM - Test Driven Modelling
QC – Quality Control
UML – Universal Modelling Language
UP – Unified Process
VAT – Value Added Tax
WCF - Windows Communication Foundations
WPF - Windows Presentation Foundations
XP – Extreme Programming

1 INTRODUCTION

We propose archetypes and archetype patterns (A&AP) based techniques for development of domains, requirements and software. We use these techniques in development of a real life laboratory information management system (LIMS) [1] software and LIMS Software Factory.

We have published fourteen conference papers (Appendix 7.5) connected to this thesis. Conference paper [2] summarizing the main points of the current thesis was accepted by 21st European Japanese Conference on Information Modelling and Knowledge Bases (June 6-10, 2011, Tallinn, Estonia). Post conference proceedings of this conference will be published in 2012 by IOS, Amsterdam, in the series "Frontiers in Artificial Intelligence and Applications".

1.1 Research Problem

There are two main challenges in software development: complexity and change. Software engineers have tried to cope with complexity by applying object oriented development techniques [3 pp. 66-86] and formal methodologies [4]. To cope with change, software process methodologies [3 pp. 87-105], including agile software development methodologies [5], have been used. When developing enterprise applications, software engineers have to embrace both complexity as well as change.

Layering is a common technique for complicated software systems [6 p. 17]. Both .Net and Java framework have tools for developing 4-tier software systems [7]. Nowadays 4-tier software architecture is a modification of common 3-tier architecture [6 pp. 19-22]. 3-tier architecture has got a data source layer (accessing data), a domain model layer (defining logic) and a presentation layer (using logic). 4-tier architecture has an additional communication layer (containing and connecting logic).

In our understanding the communication layer and the presentation layer are similar in their nature. The presentation layer gives humans an interface (forms, documents, etc.) to the defined logic (domain model). Similarly, the communication layer gives artificial agents (services, software systems, etc.) an interface (communication protocols, etc.) to the defined logic. This is why in the following we are describing changes only in the presentation (together with the communication layer), the domain and the data source layers. We see following possibilities to change the presentation (and the communication) layer:

- UI.1. Design changes in external shape (form);
- UI.2. Changes in the presentation or in the communication layer without changes in other (domain logic and data source) layers;
- UI.3. Changes in the presentation or in the communication layer which result in need to change the domain logic layer.

Normally, the presentation and the communication layer have no direct access to the data source layer. Therefore, we omit the possibility to change the

presentation (and the communication) layer so that the data source layer has to be changed. Similarly to the changes in the presentation (and the communication) layer, there can be changes in the domain logic and in the data source layers.

The domain logic layer must be designed (good design principle) without any access neither to the presentation nor to the communication layers. It follows that possible types of changes in the domain layer are:

- DM.1. Refactoring [8], which means altering internal structure of the domain logic without changing its nature or external behaviour;
- DM.2. Principal change (changing the nature or the external behaviour) in the domain layer without changes in the data source layer;
- DM.3. Change in the domain layer which also requires the change in the data source layer.

Data source layer has to be designed without any access to other layers. It follows, that changes in the data source layer are:

- DB.1. Refactoring, which means altering internal structure (renaming of tables, renaming of columns, etc.) without any need to transfer data from the old database format to the new one;
- DB.2. Principal change of database layout so that we have to transfer data from the old database format to the new one.

As changes of type UI.1 can be conducted by using tools and technologies like Windows Presentation Foundations (WPF) [9], Windows Communication Foundations (WCF) [10], BizTalk [11] or similar, these types of changes are out of our interest. Refactoring (DM.1, DB.1) is also out of our interest. By refactoring we mean making small changes step by step in order to improve the design of existing code [12 p. 37] or database layout during the development. Refactoring's are not related to the domain nor to software requirements and are supported by different refactoring tools like $ReSharper$ [13]. As compound changes (UI.3 and DM.3) can be reduced to two changes independent from each other (e.g. UI.3=DM.2+UI.2), our main interest in current thesis are independent changes DB.2, DM.2 and UI.2.

We are looking for ways to minimize (better to completely avoid) changes in the domain logic (DM.2) and in the data source (DB.2) layers as these changes are risky and time consuming. We are trying to find possibilities to fulfil user requirements only by making changes in the presentation or in the communication layers (UI.2). It would be nice if these changes can be made by end users even at run-time. Current solutions (e.g. WPF, WCF, BizTalk and similar) are sufficient ($UI1 == UI2$) when the domain logic and the data source layers are designed exactly according to customer's business needs. Unfortunately customer's business needs are constantly changing.

1.2 Research Approach

We use a case-study-based research methodology. The case is Laboratory Information Management System (LIMS) software development in Clinical and Biomedical Proteomics Group (Cancer Research UK Clinical Centre, Leeds Institute of Molecular Medicine, St. James University Hospital at University of Leeds). LIMS represents a class of computer systems designed to manage laboratory information [1].

In research laboratories, like CBPG, business processes are changing constantly and different research groups within the same research laboratory, sometimes even different investigators in one and the same research group, require different business processes and different or differently organized data. While standardized in some ways, such system for scientists has to be flexible and adaptable so, that there are customizable possibilities to describe data, knowledge and also research methods. This is why we decided not to develop only LIMS, but decided to develop a software factory for LIMS.

By Greenfield, *et al.* [3], the software factory is the domain specific RAD (Rapid Application Development) with frameworks, languages, patterns and tools. When general-purpose RAD uses *„logical information about the software captured by general-purpose development artefacts*", then the software factory uses *„conceptual information captured by domain specific models*" [3 p. 564].

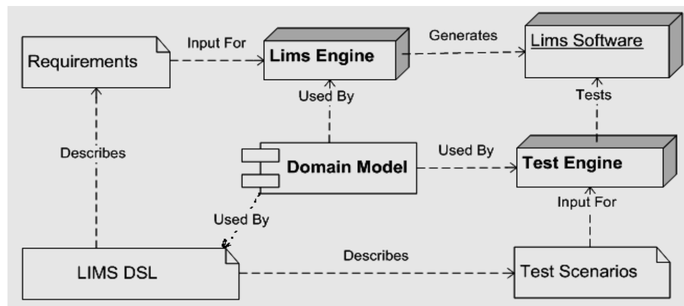

Figure 1-1: The Architecture of the LIMS Software Factory

Figure 1-1 illustrates our research and developments towards LIMS Software Factory (SF). Based on the domain model of laboratory, the LIMS SF architecture consists of the LIMS DSL (domain specific language), the LIMS Engine and the Tests Engine. Requirements for the particular LIMS software will be described with the LIMS DSL. The LIMS Engine has to generate the LIMS software according to these requirements. The Tests Engine has to validate these requirements with respect to the domain model of laboratory and has to verify the generated LIMS software. The Figure 1-1 is based on the software engineering triptych (from domain model via requirements to software). The key point is that all models we are talking about are not only documentation artefacts, but also source artefacts, as common in software factories [3].

In our understanding, to minimize or avoid changes in the domain logic and in the data source layers, as described in Section 1.1, we need a universal domain model that is implemented in the domain logic layer together with supporting database layout and data access layer. This domain model has to be mature, ought not to be changed all the time and it has to be possible to use this domain model to fulfil various user specific requirements for a particular class of software systems. If such domain model (e.g. for clinical laboratory) is available, then in order to make changes in the presentation layer or in the communication layer (UI.2, Section 1.1) we need some tools. By using these tools, the user (preferably end user) has to be able to define and change (preferably at run-time) the formats of user interfaces (web, windows, mobile, etc.) and other electronic documents (printouts, communication protocols, input documents, etc.) according to business requirements. These tools should preferably be supported by DSL that is based on the domain model mentioned above.

In current thesis we concentrate on developments of domain models for laboratory and on possibilities to use these domain models of laboratory in specification and validation of LIMS software requirements and in verification of software. With current thesis, we summarize current status of our research and developments towards LIMS software factory (Figure 1-1) components - LIMS DSL, LIMS Engine and Test Engine. We propose business archetypes and archetype patterns (A&AP) based approach for modelling and development of domain models. A&AP are models of base concepts (e.g. role) from which all concepts of the same kind (e.g. clinician, patient, customer, etc.) are originated. A&AP describe the universe of discourse of businesses as it is, neither referring to the software requirements nor to the software design. Models for business A&AP are originally proposed by Arlow and Neustadt [14]. We have improved these A&AP models and propose archetypes based techniques (ABD) for development of domains, requirements and software. In ABD we utilize these improved A&AP models.

With LIMS software developments in CBPG we are looking for and evaluate possibilities to use proposed A&AP models and ABD techniques in real life software development. Our special interest is to design A&AP and domain models, based on these A&AP, as abstract and universal as possible. We try to find possibilities to specify user requirements (and even domain models) at runtime by using these abstract and universal domain models and A&APs. We are also looking for possibilities to validate so specified requirements and verify software generated according to so specified requirements.

1.3 Contributions

The contributions of current thesis are:

1. *Archetypes Based Development techniques (ABD)* for development of domains, requirements and software;

ABD includes

a. ZF (Zachman Framework) columns based analysis (by asking questions *what*, *how*, *where*, *who*, *when* and *why*) and design (*products*, *processes*, *locations*, *persons*, *events* and *rules*) of domains and requirements by using *archetypes and archetype patterns*.
b. ZF rows based development – from *conceptual* and *semantic* models via *logical*, *physical* and *detailed* models to software *product*.

2. Improved models of *business archetypes and archetype patterns* (A&AP).

   A&AP are models (code artefacts) for independent phenomena (*products*, *processes*, *locations*, *persons*, *events* and *rules*) of ZF.

ABD is presented in Part 2. A&AP models are presented in Part 3. In Part 4 we exemplify the usefulness of ABD and A&AP models in real life software development. In Part 5 we evaluate ABD and A&AP from the perspectives of domain engineering and software development methodologies.

1.4 Hypothesis

We claim that *archetypes based development techniques* (ABD) together with proposed models of *business archetypes and archetype patterns* (A&AP) lead software development towards software factory (SF) development and thence towards possibilities to fulfil user requirements by making changes only in the presentation or in the communication layers as described in Section 1.1.

In our understanding this claim can be summed up in the following conjectural points:

1. Triptych software development (from domain models via requirements to software) is possible and reasonable.
2. We can develop models (frameworks, source artefacts) of A&AP. We can develop domain models by using these A&AP models.
3. We can specify user requirements by using domain and/or A&AP models. We can generate software according to so specified user requirements.
4. We can validate user requirements and verify software by using these models. User requirements can falsify domain as well as A&AP models.
5. We can improve and expand A&AP and domain models. We can reduce risks associated with changes in A&AP and domain models.
6. We can build different tools (generators of UI and other source artefacts, languages for end users to describe requirements, validation and verification tools for requirements and software, etc.) on top of these models. A&AP, domain models and associated tools form software factories. We can develop software factories so, that software end users can change software safety and easily even at runtime by making changes only in the presentation or in the communication layers.

1.5 Delimitations

The following restrictions should be considered:

1. The ABD, A&AP and laboratory domain models are based solely on the author's experiences in development of different software for clinical laboratories.
2. The main focus of thesis is on proof-of-concepts of development directions and strategies for author's current real life LIMS software project.
3. Current real life LIMS software, used in everyday routine of CBPG, should to be taken as prototype software in context of current thesis.
4. The presented A&AP model is designed, but not finally realized.
5. The presented laboratory domain model is designed, but not finally realized.
6. In current version of real life LIMS software only simplified versions of both (A&AP and laboratory domain) models are used.
7. In current version of real life LIMS software only some simple (A&AP based DB layout, generating of UI, some documents based configurations) elements of prospective software factory are used.

1.6 Outline of the Thesis

Archetypes Based Development techniques are described in Part 2. These techniques include *Zachman Framework based analysis* (Section 2.1.1), *triptych software process* (Section 2.1.2) and *test driven modelling* (Section 2.2). We exemplify how requirements can be specified (Section 2.3) and validated (Section 2.4) by using these techniques.

We use these techniques when improving models of *archetypes and archetype patterns*, originally introduced by Arlow and Neustadt (Part 3). We describe methodology (Section 3.1) and create initial models (Section 3.2). In Section 3.3 we evaluate these models by comparing them with models by Fowler [15], Hay [16] and Silverston [17]. We proceed with fine tuning (Section 3.4) and definitions (Section 3.5) of archetypes and archetype patterns and consummate (Section 3.6) with discussions about using these improved models of A&AP in development of domain models.

The usefulness of proposed techniques and models is exemplified in Part 4 where the clinical laboratory domain model (Section 4.2) and real life LIMS development (Section 4.3) is described. In Section 4.4 we propose a theoretical foundation for development of software factories and evolutionary information systems. This theoretical foundation utilizes archetypes and archetype patterns based domain models and P-systems (membrane computing) by G. Paun [18].

Archetypes Based Development techniques (as explained in Part 5) are in agreement with and complement important software development processes and methodologies, such as Bjørner's domain modelling (Section 5.1), Model Driven Architecture (Section 5.2.3), Extreme Programming (Section 5.2.4) and Capability Maturity Model Integration for Development (Section 5.2.5).

2 ARCHETYPES BASED DEVELOPMENT

In the following, we explain the main ideas behind archetypes based techniques for development of domains, requirements and software. We call these techniques ABD (Archetypes Based Development). In explanations we use a simple domain of quantity. The ideas of current part were first published in the paper "The Zachman Framework with Archetypes and Archetype Patterns" [19] presented in the Baltic Database and Information Systems conference, Riga, Latvia, 2010. We presented the ideas of Test Driven Modelling in MIPRO conference, Opatia, Croatia, 2011 [20].

2.1 From Domain via Requirements to Software

According to software engineering triptych, in order to develop software we have to

1) Informally and/or formally describe a domain ($\mathcal{D}$);
2) Derive requirements ($\mathcal{R}$) from these domain descriptions; and
3) Finally from these requirements we have to determine software design specifications and implement the software ($\mathcal{S}$), so that $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ (meaning the software is correct) holds [21].

The term *domain* or *application domain* can be anything to which computing can be applied [22]. In ABD, the archetype patterns, domain models and software requirements are analysed and modelled according to the Zachman Framework (ZF) [23]. ZF for enterprise architecture has been widely accepted as a standard for identifying and organizing descriptive representations that have critical roles in enterprise management and system development. For this reason, the ZF was selected as a reference model for ABD.

ZF is a two dimensional matrix consisting of 6 rows and 6 columns. Each column of ZF describes a *single, independent phenomenon*. These independent phenomena are *things* (what), *processes* (how), *locations* (where), *people* (who), *events* (when) and *strategies* (why). In ABD, these independent phenomena are analysed and developed by using product (what), business process (how), organization structure (where), person (who), order and inventory (when) as well as rule (why) archetype patterns.

2.1.1 Zachman Framework Based Analysis

Table 2-1 illustrates how in ABD we use *product*, *party* and *party relationship*, *order* and *inventory*, *rule*, *quantity* and *money* archetype patterns for modelling of independent phenomena of enterprises described by columns of ZF.

Column 1 (what, things) describes what products (either goods or services) are and how these products are related to each other. Examples of product relations are "produced by using", "produced from", "is a component of", "belongs to", "upgradable to", "substituted by", "complemented by", "compatible with", "incompatible with" and etc. For modelling of products and

product relationships we use the *product* AP (Section 3.5.6). Two additional APs (*quantity,* Section 3.5.1 and *rule*, Section 3.5.3) are needed when modelling products.

Table 2-1: ZF Columns with Archetype Patterns

<table>
<tr><th colspan="7">Business requirements</th></tr>
<tr><th>What</th><th>How</th><th>Where</th><th>Who</th><th colspan="2">When</th><th>Why</th></tr>
<tr><th>Things</th><th>Processes</th><th>Locations</th><th>Persons</th><th colspan="2">Events</th><th>Strategies</th></tr>
<tr><td rowspan="2">Products and services</td><td rowspan="2">Reporting (feedback)</td><td>Organization and organization structure</td><td>Persons</td><td colspan="2" rowspan="2">Business events</td><td rowspan="3">Business rules</td></tr>
<tr><td colspan="2">Party AP</td></tr>
<tr><td>Product AP</td><td colspan="3">Party relationship AP</td><td>Order AP</td><td>Inventory AP</td></tr>
<tr><td colspan="7">Rule AP</td></tr>
<tr><td colspan="7">Quantity and money AP</td></tr>
<tr><td colspan="7">Common infrastructure</td></tr>
</table>

Column 2 (how, processes) describes business processes. Examples of business processes are “buying”, “selling”, “producing”, “planning”, “servicing”, “controlling”, “reporting”, “transporting”, and so on. For modelling of business processes we use the *business process* AP (Section 3.5.9). Business process AP actively manages the progress of business processes by using feedbacks from particular business process managers. Each *process* is a party relationship where a *subordinate* (the role of a person) reports to a *supervisor* (the role of a person). We have designed the *business process* AP as a special case of the *party relationship* AP (Section 3.5.5).

Column 3 (where, location) describes the structure of an organization in terms of organization units and in terms of roles of these organization units. We strongly separated roles from parties (persons, organizations) “playing” these roles. For modelling of locations (organization structure, business environment) we use the *party* (Section 3.5.4) and the *party relationship* (Section 3.5.5) APs.

Column 4 (who, persons) describes persons employed by an organization or parties (persons, organizations) playing some other roles (customers, suppliers, etc.) related to business processes of the organization. For modelling of persons and related parties we use the *party* (Section 3.5.4) and the *party relationship* (Section 3.5.5) APs.

Column 5 (when, events) describes all business events which are somehow related to organization business processes. Examples of these events are “new order from a customer”, “plan is ready”, “some resource has reached the minimal acceptable limit”, “new employee is hired”, and etc. All such kinds of events should be logged and an audit trail should be produced. We model business events by using the *order* (Section 3.5.8) and the *inventory* (Section 3.5.7) APs. With the *order* AP, any request (not only buying and selling) to

change something in the enterprise's *inventory* or in some other list (employees list for instance) can be recorded.

Column 6 (why, strategies) describes strategies in terms of business rules. We use the simple propositional calculus based *rules* archetype pattern (Section 3.5.3) as a base model for strategies.

2.1.2 Zachman Framework Based Implementation

Archetypes Based Development is a software triptych process (from domain model via requirements to software) [22] with business archetypes and business archetype patterns. ABD involves ZF based implementation for A&APs, domain models as well as for requirements (Table 2-2). ZF columns are used for understanding and analysing of A&APs, domains and requirements (Section 2.1.1). ZF rows are used as a methodological guidance for implementing of A&APs, domain models and software. Therefore when ZF columns are indicating "what to implement", then ZF rows are indicating "how to implement".

Table 2-2: ZF Rows and ABD

ZF		**MDA**	**Abstraction**				**Concretization**	
	Model				**A&AP**	**Domain**	**Requirements**	
1	Contextual Scope		domain engineering	domain analysis	Terms	Terms	Terms	triptych software engineering
					Glossary specified as unit tests			
2	Conceptual Business Semantic	CIM			Specs	Specs	Specs	
					A&AP, Domain and Requirements are specified as			
					unit tests	acceptance tests		
3	Logical System	PIM			Design of A&AP	Design in terms of		
						A&AP	DM	
4	Physical Technology	PSM			C#	A&AP based DSL	DM based DSL	
					Source code satisfying specifications for			
					A&AP	D also A&AP	R, D and A&AP	
5	Detailed	Code			Byte-code (CIL) ready to run			
6	Product				DLL used as DSL in concretization		Application	

Row 1 (Contextual model) is a glossary (list of things, objects, assets, etc.) that defines the scope or boundary for A&APs, domains or requirements. For example, the scope for persons in clinical laboratory can include terms like *patient*, *clinician*, *medical technical assistant*, and so on. We specify the scope by unit tests as described in Section 2.2.1.

Row 2 (Semantic model) is a definition of an actual archetype pattern, domain model or user requirements. In ABD, the semantic model of A&APs is specified by unit tests (Sections 2.2.2 and 2.2.3). For the domain models (DM)

and requirements the semantic model is specified by acceptance tests as explained in Section 2.3.

Row 3 (Logical model) is a formal view of A&APs, DMs or requirements in terms of classes, properties, methods and events satisfying the semantic (Row 2) model. For A&APs this is a design of a physical model (code, Row 4) in some general purpose programming language (e.g. C#). For domains, this is a design in an A&APs based language. For software, this is a design in a DM based language. In real life developments (described in Part 4), all our logical models are described in terms of interfaces as explained and illustrated in Section 3.5.

Row 4 (Physical model) is an actual source code in some general purpose programming language for A&APs or embedded (into general purpose programming language) DSL (framework, API) for domains and requirements. A physical model has to satisfy the semantic model (Row 2). *Row 5* (Detailed definition) is a ready to run code (byte code, e.g. CIL in .NET). *Row 6* (Product) is either a DLL or an application. A&APs based DLL is used as DSL for specifying domain models. Domain model (DM) based DLL is used for specifying user requirements.

How we use the A&AP based DLL as embedded DSL for modelling of domains is described in Section 3.6. In ABD, as common for SF (software factories) [3], all models (including contextual, semantic and logical) are source artefacts and not only documentation artefacts.

2.2 Test Driven Modelling

As all models in ABD are course artefacts and not only documentation artefacts, we can utilize Test Driven Development [24] methodology for modelling (analysing and implementing) of domains and for specifying user requirements. In TDM, contextual and semantic models (Table 2-2, Section 2.1.2) are specified by unit tests and logical and physical models (Table 2-2, Section 2.1.2) are developed to satisfy these unit tests.

2.2.1 From Synopsis to Contextual Scope Model

Like Bjørner's domain analysis methodology [25; 26], our TDM methodology starts with synopsis and with sketching (visualizing) of a general picture of a domain. In addition we already start with coding (domain models are DLLs). As soon as the synopsis is ready, we name classes, implement skeletons for each class and sketch the first class diagram. All this is test driven [24].

For example in case of domain model of physical quantity, the synopsis can be as follows: *A physical quantity (for example "10 kilometres") is a numerical value of a measure expressed by a number and a unit. We can: (a) compare two quantities; (b) perform arithmetic operations with quantities; (c) round a quantity; and (d) convert a quantity from one unit to another.*

Based on such synopsis we need at least the following four classes: *quantity*, *unit*, *measure* and *number*. Because the *number* (or similar) is a base class in

any programming language, we omit the implementation of number and implement only *Quantity*, *Unit* and *Measure* classes (Figure 2-1).

Figure 2-1: First Skeleton of the Domain Model of Quantity

Because the procedure is test driven, we formalize the domain scope by unit tests (written in C# like pseudo code) as follows.

$$\begin{aligned} &CanCreate(typeof\ (Quantity)); \\ &CanCreate(typeof\ (Measure)); \\ &CanCreate(typeof\ (Unit)); \end{aligned} \qquad (1)$$

This means, that we have a unit test called $CanCreate$, which takes *type* as a parameter, and by using reflection [27] technology invokes all public constructors with default (null for instance) parameter values.

When we first specify unit tests $CanCreate$, the test environment does not even compile, and the type names *Quantity*, *Unit* and *Measure* are red.

$$\begin{aligned} &CanCreate(typeof\ (Quantity)); \\ &CanCreate(typeof\ (Unit)); \\ &CanCreate(typeof\ (Measure)); \end{aligned}$$

The reason is that there are no such kind of types as *Quantity*, *Unit* and *Measure* in the system. Thus we have to specify these types and sketch the first draft of the domain model of quantity as follows.

```
namespace Archetypes.Quantity {
   public class Quanity {}
   public class Unit {}                    (2)
   public class Measure {}
}
```

As we use Visual Studio 2010 IDE (Integrated Development Environment), the class diagram in Figure 2-1 and the textual representation above are just different views of one and the same code with full reverse engineering features.

After we have specified the quantity domain types *Quantity*, *Unit* and *Measure* (2), the domain requirements, specified by $CanCreate$ unit tests (1), compile and the type names in the listing will change their colour to blue.

Informally this means, that we have a list of domain terms (contextual scope model) under an automated verification. This means, that we can be sure, that at least these types are in system and it is possible to create these types.

In real environment the $CanCreate$ unit test is designed so, that it also tests whether all domain classes are tested. For example, if we specify a new type $DerivedMeasure$ (in $Archetypes.Quantity$ namespace) we also have to

specify the $CanCreate$ test for this type. This means, that the class specification (3) must be explicitly accompanied by the requirement specified by unit test (4).

$$public\ class\ DerivedMeasure\ \{\} \quad (3)$$

$$CanCreate\big(typeof\ (DerivedMeasure)\big) \quad (4)$$

Consequently, as soon as we change (either deliberately or accidentally) the contextual scope of a domain by adding or deleting types (domain terms), the automated verification environment informs us about this inconsistency.

2.2.2 Formalization of Narratives as Unit Tests

Like in Bjørner's domain analysis methodology [22 p. 19], in our TDM methodology we derive narratives from synopsis. By a narrative document Bjørner means a description document which systematically and reasonably explains in natural language the designated universe of discourse [22 p. 19]. For example, we can sketch the following starting narratives describing the quantity domain from the synopsis of physical quantity above (Section 2.2.1).

N.1. There are types Quantity, Measure, Unit and Number;
- N.1.1. With quantity we can associate:
  - N.1.1.1. A unit (e.g. cm) in which a quantity is measured;
  - N.1.1.2. An amount (e.g. 1.86), which is a numerical value of a measurement;
- N.1.2. With unit we can associate:
  - N.1.2.1. A name (e.g. centimetre) as a unique identification for units
  - N.1.2.2. A measure (e.g. distance) which has been measured;
  - N.1.2.3. A factor (e.g. 0.01 if talking about cm and if m is the distance base unit) which shows how many base units a particular unit is equal to;
- N.1.3. With measure we can associate:
  - N.1.3.1. A name (e.g. Distance) as a unique identification for measures
  - N.1.3.2. A formula (e.g. $\frac{Distance}{Time}$) defining the measure.
- N.1.4. We can define following operations with quantity
  - N.1.4.1. Arithmetic
  - N.1.4.2. Comparing
  - N.1.4.3. Rounding
  - N.1.4.4. Converting

Bjørner formalizes narratives by using the RAISE [28; 29] specification language. Differently from Bjørner's domain analysis methodology we do not formalize narratives neither in the RAISE specification language nor in any other specification language like Z [30], B [31], or VDM-SL [32]. We specify our domain narratives as unit tests in ordinal programming language. For example, the quantity narratives above can be specified by unit tests as follows:

```
[TestMethod] public void A07010000_TypesOfQuantityMeasureAndUnit(){}
[TestMethod] public void A07020000_QuantityHasUnitAndAmount() {}
[TestMethod] public void A07030000_UnitHasNameMeasureAndFactor() {}
[TestMethod] public void A07040000_MeasureHasNameAndFormula () {}
[TestMethod] public void A07050100_ArithmeticOperationsWithQuantity() {}
```

```
[TestMethod] public void A07050200_ComparingOperationsWithQuantity() {}
[TestMethod] public void A07050300_RoundingOperationsWithQuantity() {}
[TestMethod] public void A07050400_ConvertingOperationsWithQuantity() {}
```

In summary, narratives describe domains semantically. Narratives, written as unit tests, have the same attributes [22] as narratives written in natural language: they are documents; they describe the domain *systematically*; they can be designed *reasonably comprehensively*; unit test names explain the essence *in natural, yet most likely (application domain-specific) professional language*; and they explain *entities, functions and behaviours (including events)* of a designated universe of discourse. In addition, narratives written as unit tests (source artefacts) are able to test the domain models (implemented as DLLs) automatically.

2.2.3 Specification of Narratives

When we first write unit test based narratives, a body (marked as { } in listing in Section 2.2.2) of each unit test is specified as follows:

```
{  Assert.Inconclusive();  }
```

This means that a narrative is inconclusive - not yet specified. When we run these inconclusive narratives, we get the “yellow” pattern (the uppermost table on Figure 2-2; the circles with question marks are yellow). When a narrative is specified, but the model is not implemented according to these narratives, we will get the “red” pattern (the middle table on Figure 2-2; the circles with crosses are red). In the following some specified narratives are exemplified:

```
[TestMethod] public void A07020000_QuantityHasUnitAndAmount () {
Func < object, Type, bool > isInstanceOfType = (x, y) => x.GetType() == y;
        var q = new Quantity();
        Verify( isInstanceOfType (q.Unit, typeof(Unit) ));
        Verify( isInstanceOfType (q.Amount, typeof(double) ));
}
[TestMethod] public void A07050100_ArithmeticOperationsWithQuantities() {
Func < object, bool > isQuantity = (x) => x.GetType() == typeof(Quantity);
        var q1 = new Quantity();
        var q2 = new Quantity();
        var r = Double.Epsilon;
        Verify( isQuantity (q1 + q2) );
        Verify( isQuantity (q1 − q2) );
        Verify( isQuantity (q1 ∗ q2) );
        Verify( isQuantity (q1 ∗ r) );
        Verify( isQuantity ( r ∗ q1) );
        Verify( isQuantity (q1 / q2) );
        Verify( isQuantity (q1 / r) );
        Verify( isQuantity ( r / q1) );
    }
```

The first test (A0702000) checks that instance of type *Quantity* has properties *Unit* and *Amount* and that the property named *Unit* is a type of *Unit* and that the property named *Amount* is a (primitive) type of *double*. The second exemplified test (A0705010) checks that arithmetic operations are defined and that results of all arithmetic operations are of type *Quantity*.

Result	Test Name
Inconclusive	A07010000_TypesOfQuantityMeasureAndUnit
Inconclusive	A07020000_QuantityHasUnitAndAmount
Inconclusive	A07030000_UnitHasNameMeasureAndFactor
Inconclusive	
Inconclusive	
Inconclusive	
Inconclusive	
Inconclusive	

Result	Test Name
Failed	A07010000_TypesOfQuantityMeasureAndUnit
Failed	A07020000_QuantityHasUnitAndAmount
Failed	A07030000_UnitHasNameMeasureAndFactor
Failed	
Failed	
Failed	
Failed	
Failed	

Result	Test Name
Passed	A07010000_TypesOfQuantityMeasureAndUnit
Passed	A07020000_QuantityHasUnitAndAmount
Passed	A07030000_UnitHasNameMeasureAndFactor
Passed	A07040000_MeasureHasName
Passed	A07050100_ArithmeticOperationsWithQuantities
Passed	A07050200_ComparingOperationsWithQuantities
Passed	A07050300_RoundingOperationsWithQuantities
Passed	A07050400_ConvertingOperationsWithQuantities

Figure 2-2: "Yellow" (question marks), "Red" (crosses) and "Green" (check marks) Patterns of Narratives

In particular, variable r is defined as the smallest positive Double value that is significant in numeric operations or comparisons, so expressions involving r test the boundary conditions.

2.2.4 Structure of Unit Tests (Formally Specified Narratives)

For each narrative, specified as unit test, we keep a simple unified structure:

1. We define the helper function (one or more) for verification of post conditions;
2. We define preconditions;
3. We execute a piece of code;
4. We verify post conditions.

This means, that narratives, specified as unit tests, are in harmony with Hoare triple ($P\{Q\}R$) [33], describing a connection between a precondition (P), a program (piece of code) (Q) and a description of a result (post condition, R) of execution of a program. For example, the narrative A07050100 (Section 2.2.3) defines the helper function named $isQuantity$, which indicates an error for all objects which are not of type *Quantity*. Then we define preconditions *q1*, *q2* (two objects of type *Quantity*), and *r* (smallest possible double value). Finally we verify that results of all defined arithmetic operations are of type Quantity.

After specifying narratives as unit tests we have to implement the quantity model according to specified narratives to get the "green" pattern (the lower table on Figure 2-2; the circles with check marks are green).

In conclusion, after specifying domain narratives as unit tests we have at least some preliminary contextual (ZF Row 1), semantic (ZF Row 2) as well as logical (ZF Row 3) models. Contextual and semantic models are specified as unit tests. A logical model (the skeleton of a physical model) of a domain, satisfying contextual and semantic models, can be presented in the form of class diagram as illustrated in Figure 2-3.

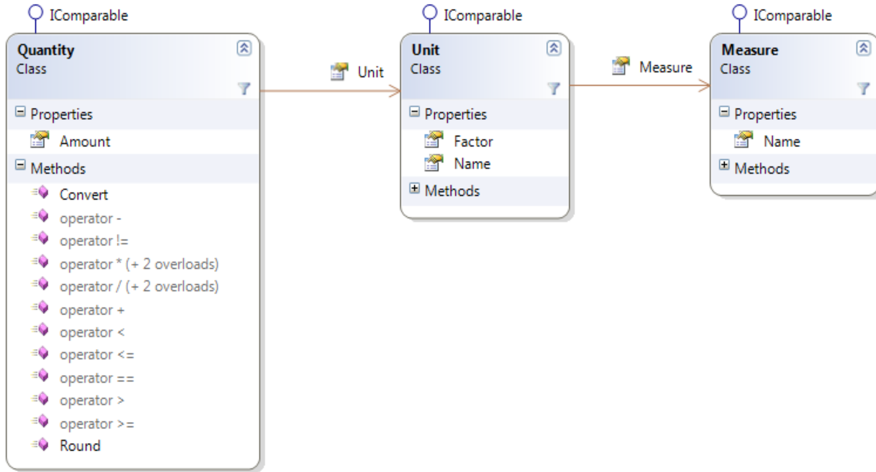

Figure 2-3: Preliminary Version of Quantity Domain Model

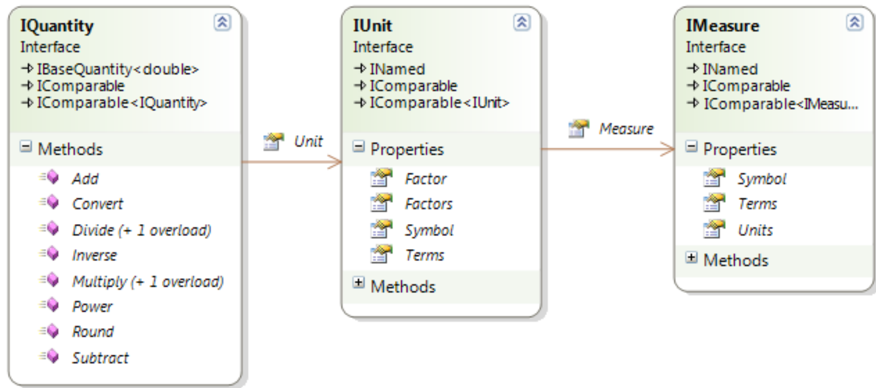

Figure 2-4: Quantity Domain Model in Terms of Interfaces

The other possibility, as shown in Figure 2-4, is to present logical models as interfaces. To keep clear difference between logical and physical models we prefer interfaces as the primary presentation for logical models. Naturally all of these models, contextual, semantic as well as logical, are just preliminary models. It is also clear that all these models are changing and evolving according to further developments. Since narratives (contextual and semantic models) are specified as unit tests, it is relatively safe to change and improve

domain models. That is because unit tests are able to automatically track potential inconsistencies between new developments and work done.

2.2.5 Fine Tuning of Domain Models

Logical model is not a ready to use model of a domain. Logical model is only a blueprint of a domain in terms of classes (or interfaces) and their public members (properties, methods and events) and in principle can be independent from any particular implementation and implementation environment. Think, for example, about possibility to implement the addition ("+") operation of quantity as follows:

```
public static Quantity operator + (Quantity a, Quantity b) {
    return new Quantity();
}
```

This is obviously not correct, but this implementation is just enough to get the "green" pattern according to the narratives we have specified so far. Let us now add a new unit test as follows:

```
[TestMethod]public void 07050101_CanAddQuantitiesOfSameMeasure() {
  Func < Quantity, double, Unit, bool > isAmountAndUnitCorrect =
     (x, y, z) => Verify( (x.Amount == y) && (x.Unit == z));
     var m1 = new Measure { Name = "M1" };
     var m2 = new Measure { Name = "M2" };
     var u1 = new Unit { Name = "U1", Measure = m1, Factor = 1 };
     var u2 = new Unit { Name = "U2", Measure = m1, Factor = 10 };
     var u3 = new Unit { Name = "U3", Measure = m1, Factor = 0.1 };
     var u4 = new Unit { Name = "U4", Measure = m2, Factor = 1 };
     var q1 = new Quantity { Amount = 1, Unit = u1 };
     var q2 = new Quantity { Amount = 10, Unit = u2 };
     var q3 = new Quantity { Amount = 0.1, Unit = u3 };
     var q4 = new Quantity { Amount = 1, Unit = u4 };
     isAmountAndUnitCorrect (q1 + q1, 2, u1) ;
     isAmountAndUnitCorrect (q1 + q2, 2, u1) ;
     isAmountAndUnitCorrect (q2 + q1, 20, u2 );
     isAmountAndUnitCorrect (q1 + q3, 2, u1);
     isAmountAndUnitCorrect (q3 + q1, 0.2, u3 );
     isAmountAndUnitCorrect (q2 + q3, 20, u2 );
     isAmountAndUnitCorrect (q3 + q2, 0.2, u3 );
     Verify( (q1 + q4) == Quantity.UND );
}
```

First we have defined a helper function *isAmountAndUnitCorrect*. This helper function returns *true* if the *Quantity* given by the first parameter has *Amount* and *Unit* equal to values given by the second and third parameters accordingly. Then we define preconditions: two different *Measures*; three different *Units* with different *Factors* for first *Measure*; one unit for second *Measure*; and four different *Quantities* which each have a value measured in different *Unit*.

	Result	Test Name
☐	Passed	A07010000_TypesOfQuantityMeasureAndUnit
☐	Passed	A07020000_QuantityHasUnitAndAmount
☐	Passed	A07030000_UnitHasNameMeasureAndFactor
☐	Passed	A07040000_MeasureHasName
☐	Passed	A07050100_ArithmeticOperationsWithQuantities
☑	Failed	A07050101_CanAddQuantitiesOfSameMeasure
☐	Passed	A07050200_ComparingOperationsWithQuantities
☐	Passed	A07050300_RoundingOperationsWithQuantities
☐	Passed	A07050400_ConvertingOperationsWithQuantities

Figure 2-5: "Red" pattern, when we add new narratives

Finally we verify the correctness of the add operation. According to the specified unit test, we expect the following narratives:

1. We can add quantities with one and the same unit.
2. We can add quantities of one and the same measure. For example we can add meters with centimetres.
3. A unit factor equal to one show that the unit is a base unit. For example, we expect that in SI system the *Meter* is defined with *Factor* equal to one.
4. *Factor* different to one show that the *Quantity* with *Amount* equal to the *Factor* of this *Unit* is equal to the measure's base unit. For example, in SI system, 10 *Decimetres* equals to one *Meter*, because the *Factor* of *Decimetre* in SI system is equal to the 10; the 0.001 *Kilometres* is equal to one *Meter*, because the *Factor* of the *Kilometres* in SI system is equal to 0.001.
5. If the add operation is defined and $u1$ and $u2$ are the units of quantities $q1$ and $q2$ (one and the same measure), then the unit of quantity $q1 + q2$ is $u1$. For example $1\,m\ +\ 2\,dm\ =\ 1.2\,m$.
6. If the add operation is defined and $a1$ and $a2$ are the amounts of quantities $q1$ and $q2$ (one and the same measure) and if $k1$ and $k2$ are the factors of units of quantities $q1$ and $q2$ respectively then the amount of $q1 + q2$ is equal to $a1 + a2 \times \frac{k1}{k2}$. For example $2dm + 1\,m =\ 2\ +\ 1 \times \frac{10}{1}\ dm = 12\ dm$.
7. We expect that $1.2m = 1m + 2dm = 2dm + 1m = 12dm$.
8. If quantities are measured by units of different measures, then the result of add operation is UND (Undefined).

These narratives (A07050101), specified as unit tests, give us the "red" pattern as shown in Figure 2-5 (the circle with a cross is red). To get the "green" pattern, we need to supplement the domain model of quantity by implementing the add operation as specified by unit test A07050101.

With such a gradual and step by step upgrading of the model we come closer and closer to the physical domain model of quantity, which has types *Quantity*,

Unit and *Measure* and which is able to perform arithmetic ("+", "−", "×" and "/"), comparing ("==", "!=", "<", "<=", ">" and ">="), rounding and converting operations with given quantities. Process from physical model (code in general purpose programming language) via detailed model (byte code) to product (DLL used as DSL in more concrete implementation) is fully automated thanks to nowadays integrated development environments.

2.3 From Domain Model to Software

According to the software engineering triptych, in order to develop software we first have to informally or formally describe a domain ($\mathcal{D}$); then we somehow have to derive the requirements ($\mathcal{R}$) from these domain descriptions; and finally, from these requirements we have to determine software design specifications and implement the software ($\mathcal{S}$), so that $\mathcal{D}, \mathcal{S} \vDash \mathcal{R}$ holds [21], meaning the software is correct. The key point is that all models we are talking about are not only documentation artefacts, but are source artefacts. Thus the domain model of quantity is ready to run DLL.

As an example, by using the (implemented) domain model ($\mathcal{D}$) of quantity and the (implemented) quantity repository [6 pp. 322-327] (part of quantity domain, registers and holds all measures as well as units, realised as static functions of Quantity archetype), we can specify (write in code) particular requirements ($\mathcal{R}$) for some specific measures and units. For instance, if area, volume and speed are needed as measures, we can specify these requirements as follows:

```
//Measures – units must be defined
Measure Distance = Quantity.RegisterMeasure("Distance","s");
Measure Time = Quantity.RegisterMeasure ("Time","t");
//Derived measures – units must not be defined
Measure Speed =
          Quantity.RegisterMeasure( "Speed","v",Distance,1,Time,−1);
Measure Acceleration =
          Quantity.RegisterMeasure("Acceleration","a",Speed,1,Time,−1);
Measure Area = Quantity.RegisterMeasure("Area","A",Distance,2);
...
// Distance units
Unit Microns = Quantity.RegisterUnit( Distance,"Microns","",1E6);
Unit Millimeters =
          Quantity.RegisterUnit( Distance,"Millimeters","mm",1000);
...
// Time units
Unit MilliSeconds = Quantity.RegisterUnit( Time,"MilliSeconds","ms",1000);
Unit Seconds = Quantity.RgisterUnit( Time,"Seconds","sec",1);
```

First we have defined (registered with methods of quantity repository) two base measures *Distance* and *Time* by specifying their names and symbols. Then we have defined (registered) three derived measures *Speed*, *Acceleration* and *Area*

by specifying their names, symbols and formulas. For example, the formula $Speed = \frac{Distance}{Time}$ has been formalized as $Distance, 1, Time, -1$. In definitions we can use both the base as well as derived measures. For instance, we have used derived measure $Speed$ when defining $Acceleration$. Finally we have defined units for base measures only.

The set of definitions given above is all that is required in order to have software ($\mathcal{S}$), which is able to convert all the area, volume and speed (as well as distance and time) units from one particular unit to another. It also performs all arithmetic, comparison and unit conversion operations with quantities. For instance, the software is able to divide meters with seconds and give an answer in kilometres per hour. The correctness of software ($\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$) according to some particular requirement (for example converting "kilometres per hour" to "meters per second") can now be validated by following acceptance tests [24].

```
var quantity1 = new Quantity("72 km/h");
var quantity2 = new Quantity("10 m/s");
var quantity3 = new Quantity("10 m ");
var quantity4 = new Quantity("2 s");
var quantity5 = quantity3 / quantity 4;
...
Assert.AreSame(Meters, quantity3.Unit);
Assert.AreSame(Distance, quantity3.Unit.Measure);
Assert.AreSame(Speed, quantity2.Unit.Measure);
Assert.AreSame(Speed, quantity5.Unit.Measure);
...
Assert.AreEqual(new Quantity("36 km/h"), quantity2 );
Assert.AreEqual(new Quantity("20 m/s"), quantity1 );
Assert.AreEqual(new Quantity("18 km/h"), quantity5 );
```

In the acceptance tests shown above, we first have defined five quantities: first four by using integrated into domain model parsing and the last one by using divide operation. Next we have verified the units and the measures of quantities. Finally, we have tested some specific relationships. For example, that $36\frac{km}{h} = 10\frac{m}{s}$.

2.4 Validation and Verification

In Part 3, by using methodology we described above, we develop archetypes and archetype patterns (models of independent phenomena describing products, business processes, organization layouts, persons, events and rules) for business domains, requirements and software. In Part 4 we use the same methodology and the developed archetypes and archetype patterns for developing laboratory domain models and laboratory software. The target is to specify user requirements by using DSLs based on domain and A&AP models.

We see possibilities (at least partially) to validate requirements as well as to verify software with domain models developed according to TDM. If with

domain model based DSL (embedded into general purpose language, API) it is possible to prescribe user software requirements, then these requirements are valid (compatible) according to this domain model. If both, domain descriptions specified as unit tests (semantic models of domains, Table 2-2, Section 2.1.2) and software requirements specified as acceptance tests (semantic models of requirements, Table 2-2, Section 2.1.2) are satisfied ("green" pattern in Figure 2-2), then in our understanding the domain model has verified (at least partially) the software which satisfies these requirements.

The question now is: how to validate domain models. Bjørner suggest manual validation, where domain engineers "sit together" with stakeholders and review the model line by line [25 p. 347]. But this is the same way domain models are engineered and developed using TDM: domain specialist together with software engineers write and specify domain narratives by using pair programming for instance. We still have the question: do we get the right model by "sitting together". In our understanding by "sitting together" we cannot validate domain models. By "sitting together" we can just develop domain models. In our understanding domain models cannot be validated. We can only falsify domain models.

Domain models can be falsified by requirements from real life. If a domain model satisfies some of the real life requirements, then we can just say that these requirements have not falsified the domain model. But if with this domain model we cannot satisfy one particular requirement from the real life, then this requirement (in case the requirement is correct) has falsified the domain model. For example, the domain model of quantity, implemented above, satisfies the following requirements:

1. Meter is the base unit of a distance with factor equal to one.
2. Kilometre is the unit of a distance with factor equal to 0.001

But we can falsify our domain model of quantity by using the following real life requirements:

1. Metre is the distance base unit in the SI system with factor equal 1.
2. Centimetre is the distance base unit in the CGS system with factor equal 1.
3. Metre is the distance unit in the CGS system with factor equal to 0.01.
4. Centimetre is the distance unit in the SI system with factor equal to 100.

In our domain model of quantity each unit has got only one factor. In the requirements above, both units have two factors: one for the SI system and the other for the CGS system of units. Because our domain model is implemented according to semantic model specified by unit tests, we can safely upgrade this quantity domain model to satisfy both real life requirements above. For this we have to specify some new narratives (specify semantic models) and upgrade logical and physical models to satisfy these narratives. In the same time, the "old" narratives automatically take care that our upgraded domain model also holds for all the previously specified narratives.

2.5 Summary

Archetypes Based Development (ABD) is a software engineering triptych [22] based software development process with archetypes and archetype patterns [14]. ABD is guided by Zachman Framework [23]. In ABD (Table 2-1), the independent phenomena, described by ZF columns, are analysed and developed by using product (what), business process (how), organization structure (where), person (who), order and inventory (when), as well as rule (why) archetype patterns. Therefore the ZF columns with archetypes describe “what to develop”.

In ABD, we use Test Driven Modelling (TDM). TDM is tied with ZF rows (Table 2-2) and describes “how to develop”. In TDM, we first delimit the scope of phenomena to get a contextual model (according to ZF Row 1). We next specify requirements with unit tests. These unit tests form semantic models (ZF Row 2) of phenomena. By incremental specification and implementation of requirements we get step by step closer to logical (ZF Row 3) and physical (ZF Row 4) models. Logical models are models of phenomena in terms of interfaces (or class designs) and their relationships. Physical models are models of phenomena in some general purpose programming language. Physical models have to satisfy semantic models (ZF, Row 2) specified by unit tests. A detailed model (ZF Row 5) is a model in ready to run byte-code (e.g. CIL in .NET) and a product (ZF Row 6) is a DLL used as embedded DSL for prescribing software requirements or an application used by a customer.

There are (at least) three separated development processes in ABD (Table 2-2):

1. Development of A&APs (archetypes and archetype patterns);
2. Development of domain models; and
3. Development of applications.

TDM is utilized in all of these developments. A general purpose programming language is used in the development of A&APs. The target of the development of A&APs is to get the embedded (into general purpose programming language) DSL for development of domains. In development of domains, the A&APs based embedded DSL is used for specifying domain descriptions (contextual and semantic models of domains). The target of the development of domain models is to get the embedded DSL for developing applications. In application development (customizing and hopefully automated generating in future), the domain model based embedded DSL is used for specifying customer requirements.

Following the software engineering triptych [22], we have a domain ($\mathcal{D}$) model specified by using A&APs based embedded DSL. This domain model is then used for specifying customer requirements ($\mathcal{R}$). According to specified requirements, the software (meaning application) ($\mathcal{S}$) is developed (customized and hopefully can be generate in future). With TDM we ensure the correctness of application ($\mathcal{D}, \mathcal{S} \models \mathcal{R}$) as much as possible.

3 MODELS OF ARCHETYPES AND ARCHETYPE PATTERNS

Patterns in software engineering are widely used for describing general and repeated issues as well as for describing solutions to these issues [34]. Lots of patterns [35; 16; 15; 17; 36; 14] are also developed for modelling of enterprise business logic and data.

One of the problems with these enterprise business logic and data patterns is semantic heterogeneity [37]. Semantic heterogeneity means that models and data schemes describing the same or similar universe of discourse but developed by different independent parties are different. Such semantic heterogeneity is an obstruction when developing interoperable software systems [38]. This is especially critical in distributed healthcare systems, where semantic interoperability is mission critical [39].

Common nowadays solutions for dealing with semantic heterogeneity are data mapping tools similar to Microsoft BizTalk Server [11]. However, it would be useful if different patterns describing one and the same domain (for example the domain of persons and organization) could be combined into one, or at least be subsumed by one, pattern [34]. This clearly requires an assumption that we can analyse, model and unify domains as described by Bjørner [22; 40; 25].

We published the ideas, described in current part of paper, in MIPRO 2007 [41] and in doctoral symposium of Formal Methods conference, Turku, Finland, 2008 [42].

3.1 Methodology

Archetypes and archetype patterns [14] by Arlow and Neustadt are selected as initial models because these patterns have intuitive names and are compatible with ZF [23] (Table 2-1) as well as with triadic model of activity [43].

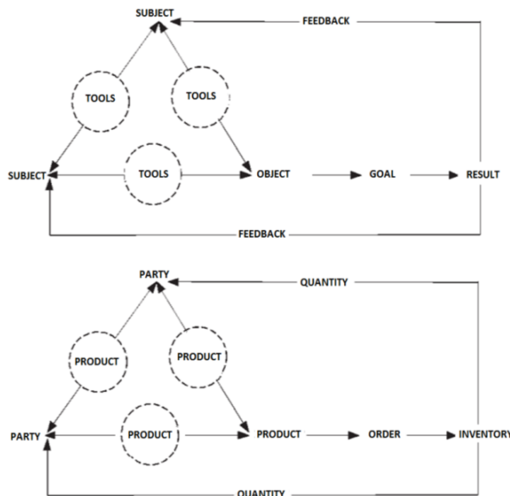

Figure 3-1: Triadic Model and its A&AP Analogue

The triadic model of activity (Figure 3-1) is used as a theoretical base in industrial-organizational psychology to describe human work, mind, culture, and activity. According to the triadic model, all activities are performed either by one or more subjects. Thus the *party archetype pattern* is required. The *product archetype pattern* is also needed. It is because when performing activities, subjects can use tools and the outcome of an activity is some object (product or service). Each activity is triggered by a goal which in a business domain is some kind of order from a client - therefore the *order archetype pattern* is also essential. Each activity has a result, which in businesses will be a record in an inventory list - hence, the *inventory archetype pattern* is required. From an inventory list a subject (manager, etc.) gets feedback about business activities. In businesses, feedback will be measured mainly by money or by some other physical measure - therefore, the *quantity archetype pattern* is required. Finally, arrows in Figure 3-1 are rules describing different conditions which have to be followed. Thus the *rule archetype pattern* is necessary.

We remove from Arlow and Neustadt's archetypes and archetype patterns all operational level attributes so the resulting redesigned archetypes and patterns include only knowledge level attributes as suggested by Fowler [15 p. 26]. Obtained archetypes and patterns are then evaluate by using them for modelling Fowler [15], Hay [35; 16] and Silverston's [17] patterns.

The methodology we use in development of archetypes and archetype patterns is the following:

1. We define initial models;
2. We evaluate these initial models and improve them if necessary;
3. We define the final set of models.

We describe in detail the development of party and party relationship archetype patterns. Thereat we utilize TDM described in Section 2.2. All other archetype patterns are then described only by their logical models. We use these A&APs for developing domain models for a clinical laboratory domain (Section 4.2).

3.2 Creating of Initial Models

We start with a synopsis and a contextual scope model (Section 3.2.1). We then derive narratives (Section 3.2.2), specify these narratives as unit tests (semantic model, Section 3.2.3) and implement the preliminary models (Section 3.2.4) so that these narratives specified as unit tests are satisfied ("green" pattern; the lower table on Figure 2-2; the circles with check marks are green).

3.2.1 Synopsis – Party

We refer to the party and party relationship archetype patterns originally designed by Arlow and Neustadt [14].

The *party archetype pattern* (Figure 3-2) [1] represents a (identifiable, addressable) unit that may have a legal status and has some autonomous control over its actions. *Persons* and *organizations* are types of parties. Party has zero or more *addresses* (phone number, e-mail, web address, postal address) where one and the same address can belong to more than one parties. Party has zero or more *registered identifiers* (passport, VAT number, domain name, stock exchange symbol, etc.). Party *authentication* is the way to confirm that party is who they say they are. Each party can play different *roles* (e.g. one and the same person in a laboratory can be the patient as well as medical technical assistant or clinician). *Preference* stands for a party's (or a role's) choice of or linking for something (like dietary preference) and is typically selected from a set of options. The *capability* is a collection of facts about what a person or organization is capable of doing as well as *body metric* stores information about the human body.

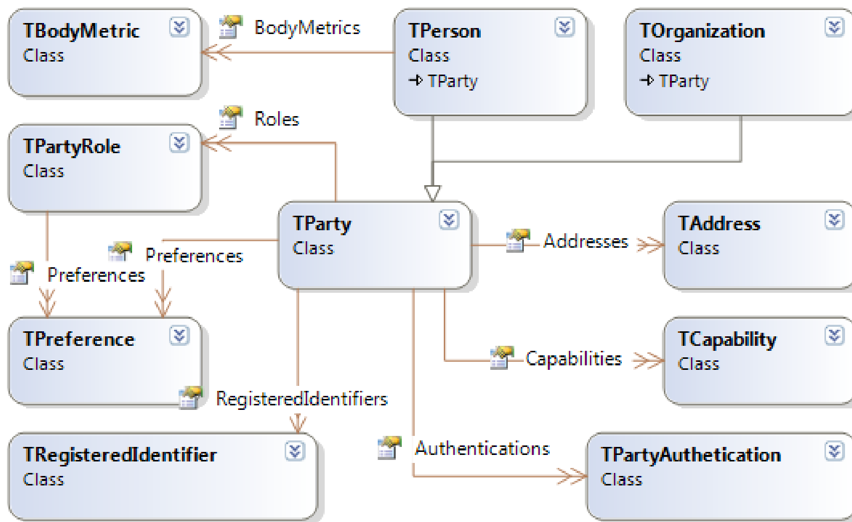

Figure 3-2: The Party Archetype Pattern Abstraction

Figure 3-3 abstracts the *party relationship archetype pattern,* which captures a fact about semantic relationship between two parties in which each party plays a specific role. Binary (more flexible and cleaner than n-ary) relationship is used, which means that one *relationship* binds two related roles called "client" and

[1] Class diagrams are done with integrated into Visual Studio 2010 class diagram tool. Prefix "T" in class names comes from "type" and means "archetype". Inheritance in Visual Studio class diagrams is shown similarly as in UML. A single arrow (meaning is 1 to 0...1 relation) as well as double arrows (meaning is 1 to 0...n relations) are notations for class attributes. In class diagrams only the main relations between the classes are shown, but all the class details (fields, properties, methods and events) are hidden.

"supplier". It has to be clarified that role is always only used to store information that belongs to role itself and not either to a party or to a relationship.

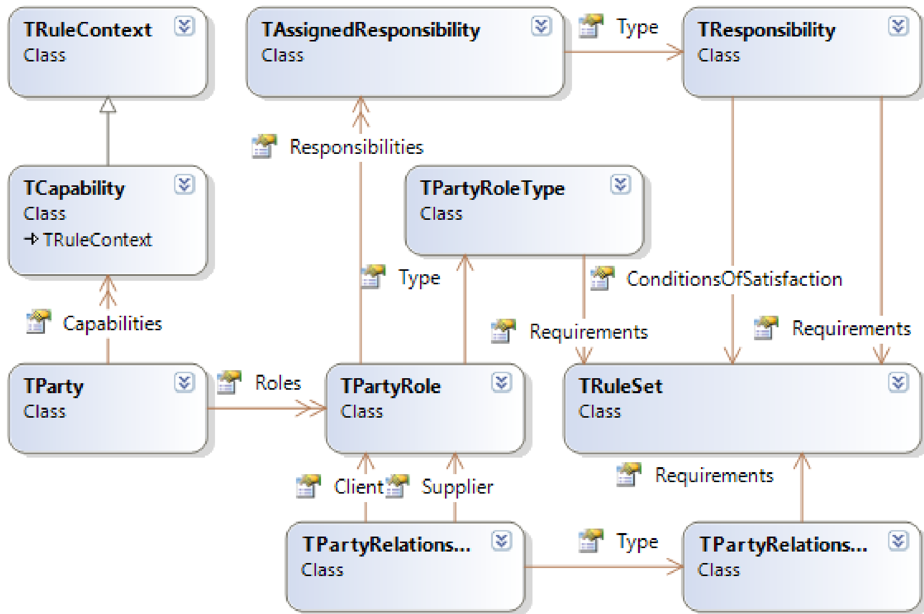

Figure 3-3: The Party Relationship Archetype Pattern Abstraction

Role type is used to store common information for a set of similar role instances as well as *relationship type* is used to store common information for a set of a similar relationship instances. *Responsibility* describes a particular activity that a party, playing a role, may be expected to perform, where the *assigned responsibility* captures the fact that responsibility is assigned to concrete party playing that role. *Condition of satisfaction* as well as the *requirements* for *party role type*, for *party relationship type* and for *responsibility* are *rule sets* (see the rule archetype pattern - Figure 3-14) where the *capability* (rule context) contains information (currently about party) needed for the execution of rules (e.g. can party complete needed responsibilities for role in relationship).

3.2.2 Narratives – Party

First of all we list the party and party relationship related types as follows.

N.3. Party Archetype Pattern
- N.3.1. There are following types :
  - N.3.1.1. *Party* with *Person* and *Organization* subtypes;
  - N.3.1.2. *Address* with *WebPageAddress*, *EmailAddress*, *TelecomAddress* and *GeographicAddress* sybtypes;
  - N.3.1.3. *AddressProperty* and *RegisteredTelecomEquipment* types;
  - N.3.1.4. *Locale* with subtype *IsoCountryCode*;

N.3.1.5. $UniqueIdentifier$ with $PartyIdentifier$ and $PartyRoleIdentifier$ subtypes;
N.3.1.6. $PartySignature$ and $PartyAuthentication$;
N.3.1.7. $RegisteredIdentifier$;
N.3.1.8. *Name* with subtypes $PersonName$ and $OrganizationName$;
N.3.1.9. $Preference$, $PreferenceType$, $PreferenceOption$ and *Attribute*;
N.3.1.10. $Ethnicity$, $BodyMetrics$ and $IsoGender$;
N.3.1.11. $PartyManager$ and $PartySummary$;
N.3.1.12. $PartyRole$, $PartyRoleType$, $PartyRoleConstraint$, $Responsibility$ and $AssignedResponsibility$;
N.3.1.13. $PartyRelationship$, $PartyRelationshipType$, and $PartyRelationshipConstraint$;

Next we associate each type with related attributes. For example narratives, describing the *name* and the *person name* archetype, are written as follows.

N.3.5. With a *name* (either *person* or *organization* name; other possibilities for future study) we can associate;
N.3.5.1. Zero or more $uses$ (legal, trading, artist's, nickname, etc.);
N.3.5.2. A $name$;
N.3.5.3. A $valid\ from$ date;
N.3.5.4. A $valid\ to$ date;
N.3.6. With a *person name* (additionally to a *name*) we can associate;
N.3.6.1. Zero or more *prefixes* (for example Mr, Dr, etc.);
N.3.6.2. Zero or one *given names*;
N.3.6.3. Zero or more *middle names*;
N.3.6.4. A *family name*;
N.3.6.5. Zero or one *preferred* names (for example Bill);
N.3.6.6. Zero or more *suffixes* (for example Jr., PhD, etc.);

That is all for now. This means, that initially narratives list only types and attributes for these types.

3.2.3 Formalization – Party

As described in Section 2.2.2, we specify all narratives as unit tests. We initially have only two types of narratives (enumeration of types and enumeration of type attributes) and therefore only two types of unit tests to start.

In regard to the first type of narratives (enumeration of types), we have to verify that all types are listed in a namespace (e.g. in $Archetypes.PartyPattern$). We also have to verify, that in this namespace there are no types other than specified ones. The following unit test illustrates the foregoing.

```
[TestMethod] public void A03010000_Party_DeclaredClasses() {
  NameSpaceClasses =
    UtilsForReflection.ListNamespaceClasses("Archetypes.PartyPattern");
  IsAbstract(typeof (Party));
  ...
```

```
  CanCreate(typeof(Person));
  CanCreate(typeof(Organization));
  ...
  IsEnum(typeof(IsoGender));
  ...
  IsStatic(typeof(PartyManager));
  ...
  AllTested(NameSpaceClasses);
}
```

First, by using reflection, we get a list of all classes (types in C# are classes) that are declared in the namespace in question. For each class we verify either a class type (abstract, static, enumeration) or an ability to create objects of that class. After verification, a sub-test (*IsAbstract* , *IsStatic* , *IsEnum* and *CanCreate*) deletes a verified class from the list of namespace classes (*NameSpaceClasses*). If a class from the list of namespace classes cannot be found, then a test indicates an error. Finally, when all types have been tested, *NameSpaceClasses* list has to be empty.

Next we verify class attributes for each class. The following code illustrates the verification of narratives for the person name archetype (see narratives from Section 3.2.2).

```
[TestMethod] public void A03060000_PersonName_PropertiesAndMethods() {
  ObjectUnderTest = new PersonName();
  ClassPropertyAndMethodNames =
    GetAllDeclaredPublicPropertiesAndMethods(ObjectUnderTest.GetType());
  HasProperty("Prefixes", typeof(string));
  HasProperty("GivenName", typeof(string));
  HasProperty("MiddleNames", typeof(string));
  HasProperty("FamilyName", typeof(string));
  HasProperty("PreferredName", typeof(string));
  HasProperty("Suffixes", typeof(string));
  AllTested(ClassPropertyAndMethodNames);
  if (!classesToTest.Remove(typeof(PersonName).Name))
      Assert.Fail(msg());
}
```

The general structure of class attribute narratives, specified as unit tests, is mostly the same as the structure for class list narratives (also specified as unit test).

1. We get a list of all class properties and methods by using reflection;
2. We verify a name and a type of each property and method;
3. If a property or method name is not found from the list of class properties and methods, a test indicates an error;
4. When all properties and methods have been tested, the *ClassPropertyAndMethodNames* list has to be empty;

5. We delete the name of the tested class from the list of classes to be tested.

⚠ Test run error Results: 43/44 passed; Item(s) checked: 1

	Result	Test Name	Project
☐	Passed	A01040000_Organization_PropertiesAndMethods	UnitTests
☐	Passed	A01050000_Name_PropertiesAndMethods	UnitTests
☑	Failed	A01060000_PersonName_PropertiesAndMethods	UnitTests
☐	Passed	A01070000_OrganizationName_PropertiesAndMethods	UnitTests
☐	Passed	A01080000_PartyUniqueIdentifier_PropertiesAndMethods	UnitTests

Figure 3-4: One Property is Not Tested

Such method ensures that as soon as we add or delete a class, a property or a method, we will be automatically informed by unit tests that there is something we have not covered by unit tests. This is illustrated in Figure 3-4.

3.2.4 Initial Model - Party

After specification of narratives as unit tests and getting the "green" pattern (see the lower table on Figure 2-2; the circles with check marks are green), we have at least an initial versions of all the models described by ZF rows. We have a contextual model (Row 1, Table 2-2) or a glossary (e.g. narratives N.3.1 in Section 3.2.2) specified as unit tests (e.g. unit test *A03010000_Party_DeclaredClasses* in Section 3.2.3). We also have a semantic model (Row 2, Table 2-2) where narratives are specified as unit tests. For instance, narratives N.3.6 (Section 3.2.2) are specified as unit test *A03060000_PersonName_PropertiesAndMethods*() (Section 3.2.3).

Because models are source artefacts and because we have unit tests, which verify these models, have given us the "green" pattern, we have also at least some preliminary logical (Row 3, Table 2-2), physical (Row 4, Table 2-2) and detailed (Row 5, Table 2-2) models as well as a product (Row 6, Table 2-2). Logical models (for instance as visualized in Figure 3-5) are designs of classes and their relationships realized in code. For example, the logical model (only declaration, not implementation) of the *PartyName* archetype, presented as code as follows:

```
public class PersonName {
  public string Prefixes { get; private set; }
  public string GivenName { get; private set; }
  public string MiddleNames { get; private set; }
  public string FamilyName { get; private set; }
  public string PreferredName { get; private set; }
  public string Suffixes { get; private set; }
}
```

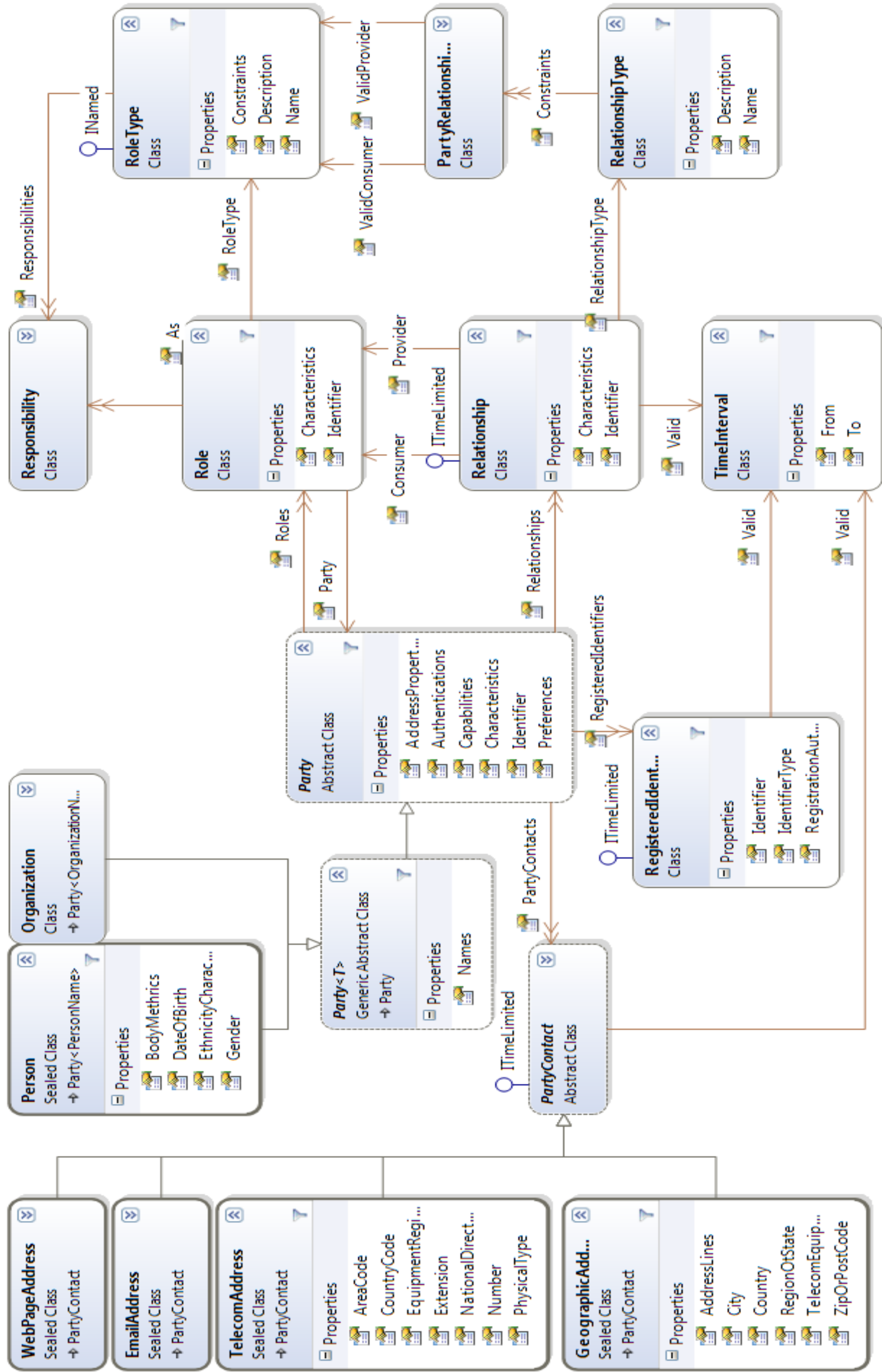

Figure 3-5: Visualization of the Initial Party Archetype Pattern

To keep clear difference between logical and physical models we prefer interfaces (as described in 2.2.4) as primary presentation for logical models as follows:

```
public interface IPersonName {
  string Prefixes { get; }
  string GivenName { get; }
  string MiddleNames { get; }
  string FamilyName { get; }
  string PreferredName { get; }
  string Suffixes { get; }
}
```

3.3 Evaluation of Models

We explain and illustrate the evaluation of *party archetype pattern* by using this pattern for modelling of Fowler's accountability [15] patterns. We then summarize evaluation of other archetype patterns according to patterns by Fowler [15], Hay [16] and Silverston [17].

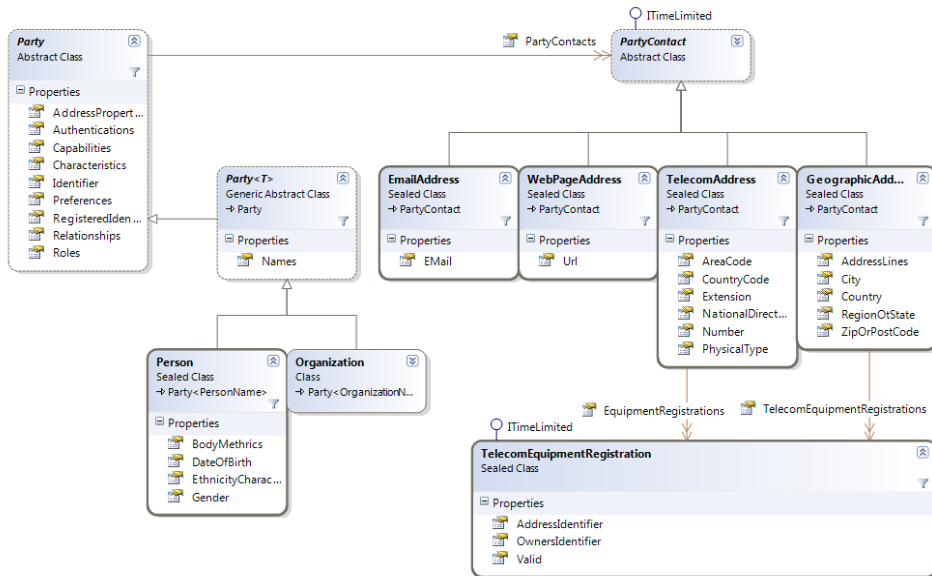

Figure 3-6: Fowler's Party in Terms of the Party AP

Fowler's accountability patterns apply when a person or an organization is responsible to another. Thus the Fowler's accountability is an abstract notion that can represent many specific issues, including organization structures, contracts, and employment. All Fowler's accountability patterns can be modelled by the *party archetype pattern* (Section 3.2.4, Figure 3-5).

3.3.1 Fowler's Party Pattern

The Fowler's *party* [15 pp. 18-19], abstracting *persons* and *organizations*, includes phone numbers, geographical and e-mail addresses. These phone numbers, geographic and e-mail addresses in terms of the *party archetype pattern* are all (*TelecomAddress*, *GeographicAddress, and EmailAddress*) persons' contacts (Figure 3-6).

3.3.2 Fowler's Organization Hierarchies

Fowler has modelled organization hierarchies [15 pp. 19-22] either as a feature of an organization (Fig. 2.4 in [15 p. 20]) or a feature of an organization's structure (Fig.2.6 in [15 p. 22]). The biggest difference between the Fowler's accountability pattern and the party archetype pattern is that in party archetype pattern there is a concept of a *role* (Figure 3-5) while Fowler has not.

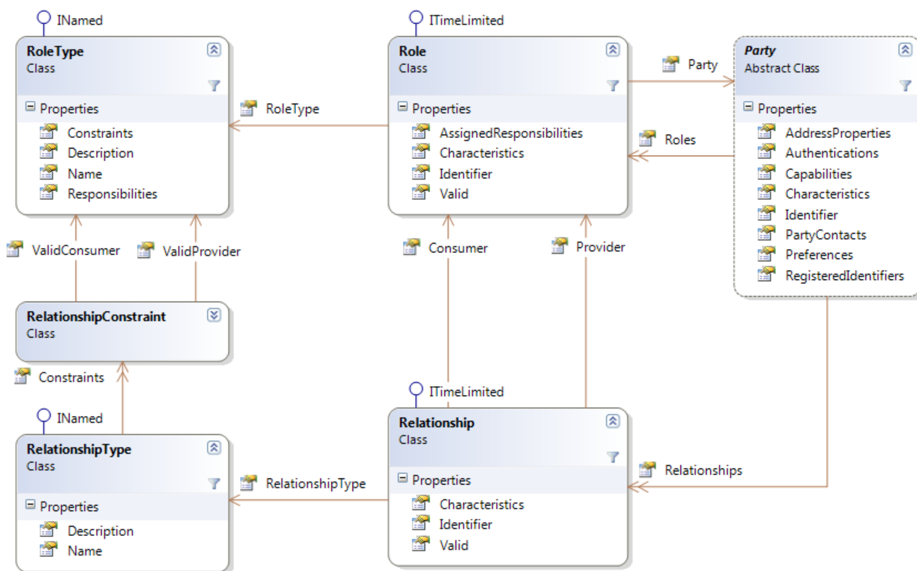

Figure 3-7: Fowler's Organization Structure in Terms of the Party AP

This means that when *operating unit*, *region*, *division* and *sales office* are *organization* subtypes (Fig.2.6 in [15 p. 22]) in Fowler's model, then in the party archetype pattern they are *role types* that organizations can play in relationships with other organizations. In terms of the party archetype pattern, the Fowler's *organization structure* with parent and subsidiary organizations is a party *relationship* with *consumer* and *provider* roles (Figure 3-7). *Operating unit*, *region*, *division* and *sales office* are *role types* in terms of party archetype pattern. A *relationship type* holds rules (constraints) about role types which can form relationships in question.

We follow the Fowler's recommendation to separate knowledge level and operational level [15 p. 26] knowledge. We want domain models to be only

models of knowledge and that requirements (operational level) from customers can be changed at runtime.

It follows that role types (*operating unit*, *region*, *division* and *sales office*) as well as relationship types (*is operating unit*, *is region*, *is division* and *is sales office*) are not "design time" subtypes of *role type* and *relationship type* accordingly, but are "run time" subtypes i.e. singletons [44]. It means that similarly to Kilogram, Meter and Hour (Section 2.3), these terms (*operating unit, region ... is operation unit, is region...*) are not related to domain models (design time), but are related to requirements (run time). The following code illustrates the foregoing.

```
public sealed class SubsidiaryType: RoleType {
  public SubsidiaryType(string name): base(name) {
   Constraints.Add(new PartyRoleConstraint(typeof(Organization).Name));
  }
 }
 public sealed class StructureType: RelationshipType {
    public StructureType(string name,
           SubsidiaryType parent, SubsidiaryType subsidiary ): base(name) {
      Constraints.Add(
            new RelationshipConstraint(parent.Name, subsidiary.Name));
    }
  }
 ...
 private static readonly SubsidiaryType main =
     new SubsidiaryType("Main organization");
 private static readonly SubsidiaryType operatingUnit =
    new SubsidiaryType("Operating Unit");
 private static readonly SubsidiaryType region =
    new SubsidiaryType("Region");
 private static readonly SubsidiaryType division =
    new SubsidiaryType("Division");
 private static readonly SubsidiaryType salesOffice =
    new SubsidiaryType("Sales Office");
 private static readonly StructureType isOperatingUnit =
    new StructureType("Is Operating Unit of Main Organization",
                  main, operatingUnit);
 private static readonly StructureType isRegion =
    new StructureType("Is Region of OperatingUnit", operatingUnit, region);
 private static readonly StructureType isDivision =
    new StructureType("Is Division of Region", region, division);
 private static readonly StructureType isSalesOffice =
     new StructureType("Is Sales Office of Division", division, salesOffice);
```

The *Structure Type* and the *Subsidiary Type* are just simple subtypes (design time, domain model belongings) of *RoleType* and *RelationshipType*. Constraints (restrictions) are declared in these helper types: only an organization

can be in a role of a subsidiary and only two organizations, being in a role of *subsidiary type*, can form a structure (relationship between two roles).

With such models it is possible to realize quite complicated organization structures. For example, we modelled as well as tested the organization structure shown in Figure 3-8.

Organization o1 (see the code above for roles and for relationships types) plays the role of a *main organization* in two relationships: with organization o2 as well as with organization o3. Organization o2 plays the role of an *operating unit* in three relationships: in relationship with o1 (type $isOperatingUnit$), the organization o2 is a provider (Figure 3-8), but in relationships (type $isRegion$) with organizations o4 and o5, the organization o2 is a consumer.

As we can see from Figure 3-8, the organization o5 takes the role of a *region* in two relationships (with organizations o2 and o3) and the role of a *division* in *relationship* with organization o8. This means that in our archetypes based organization structure model, each organization is able to play more than one role.

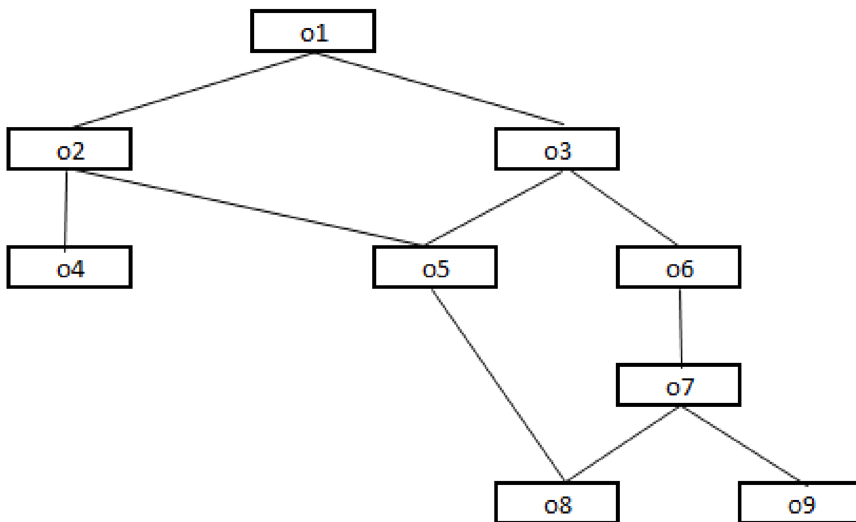

Figure 3-8: Organization Structure as Party Relationships

The condition we have to think about is that an organization, a role and a relationship are all different concepts and should keep the information related either to the organization, the role or the relationship respectively. The following is a snapshot from the code specifying the structure shown in Figure 3-8.

```
Corporation.AddRelationship(o1, o2, isOperatingUnit);
Corporation.AddRelationship(o1, o3, isOperatingUnit);
Corporation.AddRelationship(o2, o4, isRegion);
Corporation.AddRelationship(o3, o5, isRegion);
```

```
    Corporation.AddRelationship(o3, o6, isRegion);
    Corporation.AddRelationship(o2, o5, isRegion);
    Corporation.AddRelationship(o6, o7, isDivision);
    Corporation.AddRelationship(o7, o8, isSalesOffice);
    Corporation.AddRelationship(o7, o9, isSalesOffice);
    Corporation.AddRelationship(o5, o8, isSalesOffice);
```

The code for the *AddRelationship* method is the following

```
public static Relationship AddRelationship (
        Organization parent,  Organization subsidiary,
        StructureType type) {
    PartyManager.RegisterParty(parent);
    PartyManager.RegisterParty(subsidiary);
    var c  =  PartyManager.RegisterRole(
        parent, type.Constraints[0].ValidConsumer as SubsidiaryType);
    var p  = PartyManager.RegisterRole(
        subsidiary, type.Constraints[0].ValidProvider as SubsidiaryType);
    var s  =  PartyManager.RegisterRelationship(c, p, type);
    return s;
  }
```

A party manager (*PartyManager*) is a repository [6 pp. 322-327] for parties, their roles and relationships. The method *RegisterParty* checks, if a party is registered or not. If not, then *RegisterParty* adds the party to the repository. Similarly, two other methods (*RegisterRole* and *RegisterRelationship*) are also implemented. This means that in every period of time, one and the same party is registered only once. The same (registered only once) is true also for one and the same role, role type, relationship as well as for relationship type.

The uniqueness of organizations, roles and relationship types is initially based only on their names. However, more complex, rules-based validation is also possible. The uniqueness of relationships is ensured by allowing each of two organizations to form only one relationship with one and the same relationship type at the same time with each other. Analogously, one and the same organisation can be at one and the same time registered only once in a role with one and the same role type.

For example, the same person (gender has to be female) can exactly once be registered (starting from the birthday of her first child) for the mother's role and can be registered exactly once to be in the role of mother in party relationship (is mother of) with each of his children.

We can test the structure given in Figure 3-8 with acceptance tests as follows.

```
  TestParties( "main", Corporation.Parties(main), o1 );
  TestParties( "operating unit", Corporation.Parties(operatingUnit), o2, o3);
  TestParties( "region", Corporation.Parties(region), o4, o5, o6);
  TestParties( "division", Corporation.Parties(division), o5, o7);
  TestParties( "sales office", Corporation.Parties(salesOffice), o8, o9);
```

$TestParties(\text{"is operating unit"}, Corporation.Parties($
$relationshipType: isOperatingUnit),\ o1, o2, o3);$
$TestParties(\text{"is region"}, Corporation.Parties($
$relationshipType: isRegion), o2, o3, o4, o5, o6);$
$TestParties(\text{"is division"}, Corporation.Parties($
$relationshipType: isDivision), o6, o7);$
$TestParties(\text{"is sales office"}, Corporation.Parties($
$relationshipType: isSalesOffice), o5, o7, o8, o9);$

In acceptance tests above, the *Parties* method ($Corporation.Parties$) gets a list of all registered parties playing either some role type ($main$, $operatingUnit$, $region$ or $division$) or some relationship type ($isOperatingUnit$, $isRegion$, $isDivision$ or $isSalesOffice$) and checks that all given parties (e.g. o1, o2…), and no more, are in this list.

We can model the Fowler's accountability pattern (see Figure 3-10 and explanations from Section 3.3.3 for $Accountability$ and $AccountabilityType$) using the party archetype pattern similarly to Fowler's organization structure pattern (Fig.2.7 in [15 p. 23]). We omit the explanations, as these explanations are exactly the same as shown in the case of organization structure.

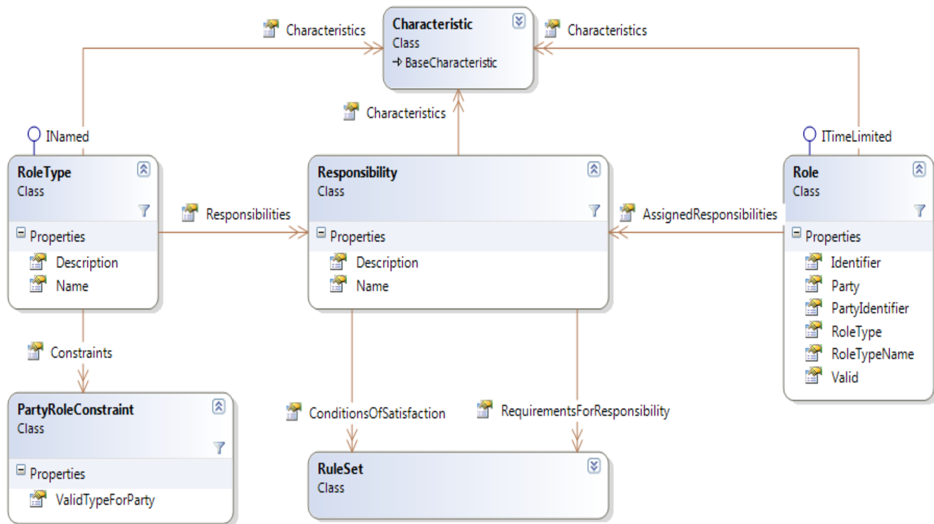

Figure 3-9: Responsibility and Assigned Responsibility

Fowler's discussions about accountability knowledge level [15 pp. 24 - 27] are exactly how we designed the party and the party relationship archetype patterns – separation of knowledge and operational levels. The same is true according to Fowler's discussions about generalization [15 pp. 27 - 28]. A domain model is a generalization (ideally of all possible real world requirements inside one and the same domain) as we have already shown in this section.

Concluding - Fowler's suggested organization structure as well as accountability patterns are special cases and applications of the party and the party relationship archetype patterns.

3.3.3 Fowler's Operating Scope

The Fowler's operating scope [15 pp. 30 - 32] can also be modelled by the party archetype pattern using the *responsibility* and the *assigned responsibility* archetypes (Figure 3-9).

A *role type* holds information (*responsibility*) of an activity that a *party* playing a *role* with specified *role type* may be expected to perform. An *assigned responsibility* of a particular *role* holds the information about responsibilities assigned to the particular *party* playing that *role*.

The Fowler's operating scope pattern (Fig.2.14 in [15 p. 31]) in terms of the party archetype pattern is visualized in Figure 3-10 and is specified as follows.

```
public sealed class Post: Role {
  public Post(Party o, RoleType t) : base("", o.Identifier.Value, t.Name) {}
 }
public sealed class Accountability: Relationship {
  public Accountability(Post p, Post s, AccountabilityType t) :
      base ("", t.Name, p.Identifier.Value, s.Identifier.Value ) {}
}
public sealed class PostType: RoleType {
   public PostType(string name): base(name) {
     Constraints.Add(new PartyRoleConstraint(typeof (Person).Name));  }
}
public sealed class AccountabilityType: RelationshipType {
  public AccountabilityType (string name, RoleType parent,
      RoleType subsidiary):  base(name) {
          Constraints.Add(
               new RelationshipConstraint(parent.Name, subsidiary.Name)); }
}
public class OperatingScope: Responsibility {
   public object Location { get { return Attributes[0].Value; }}
}
public sealed class ClinicalCareScope: OperatingScope {
   public object ObservationConcept {
        get { return Attributes[1].Value; }}
}
public sealed class ProtocolScope: OperatingScope {
   public int Amount { get { return (int)Attributes[1].Value; } }
   public object Protocol { get { return Attributes[2].Value; } }
}
public sealed class ResourceProvision: OperatingScope {
   public Quantity Quantity {
         get { return Attributes[0].Value as Quantity; } }
   public ProductType ResourceType {
```

```
        get { return Attributes[0].Value as ProductType; } }
}
public sealed class SalesTerritory: OperatingScope {
  public ProductType ProductType {
        get { return Attributes[0].Value as ProductType; } }
}
```

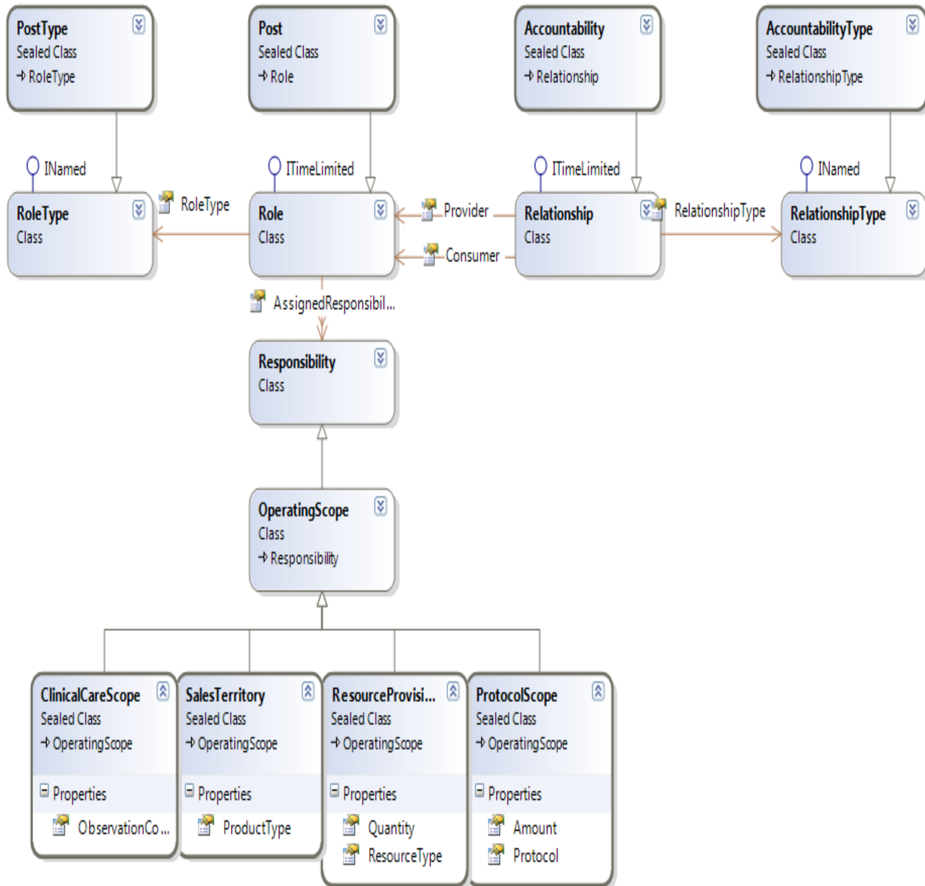

Figure 3-10: Operating Scope and the Party Archetype Pattern

As shown in this example, in the party archetype pattern a *post* is not a *party* as in Fowler's patterns [15 pp. 32 - 33]. In terms of the party archetype pattern a *post* is a *role* only a *person* can play (see constraint in specifications for particular *role type*) in relationships with organizations. Operating scope and their subtypes in terms of A&APs are "*assigned responsibilities*" of a role. Accountability is party relationship and accountability type is party relationship type (Figure 3-10).

3.3.4 Conclusions

We exemplified, that accountability patterns described by Fowler [15 pp. 17 - 33] are special cases of the party and the party relationship archetype patterns. The biggest difference in comparison to the party and the party relationship archetype pattern is that Fowler's patterns of accountability have no independent concept for party *roles* and party *role types*. The lack of the *party role* leads to the fact, that in addition to a *person* and an *organization* (two subtypes of party) Fowler has a third subtype of a party - a *post*. In our opinion this is contradicting the reality. For example, in our opinion there are no persons who essentially are doctors, patients, students, etc. However, there are persons who for some period of time carry out some of these roles (including *posts*).

Table 3-1: Fowler's Analysis Patterns and Archetype Patterns

Fowler's Analysis Pattern	***Archetype Pattern***
Accountability	Party and Party Relationship
Quantity, Measurement and Observations	Quantity
Observations for corporate finance	Party, Product, Inventory, Order, Quantity and Money
Referring to objects	Unique identifier, registered identifier, name
Inventory and accounting	Inventory
Planning	Party relationship
Contract and portfolio	Order together with Inventory
Derivative financial trades	Product (services)

The lack of a party role concept forces Fowler to admit that if Dr. Edwards is both a GP (general practitioner) and a paediatrician, we can record that only by creating a special GP/paediatrician party type, with both GP and paediatrician as super types [15 p. 28]. Such a double inheritance is unnecessary, if we have a concept of a role.

Table 3-2: Hay's Data Models and the Archetype Patterns

Hay's Data Pattern	***Archetype Pattern***
Enterprise and Its World	Party and Party Relationship
Things of the Enterprise	Product
Procedures and Activities	Party and Party Relationship
Contracts	Order together with Inventory
Accounting	Inventory
The Laboratory	Lifting of different archetype patterns
Material Requirements Planning	Party Relationship, Process
Process Manufacturing	Party Relationship, Process
Documents	Not covered

Table 3-1 summarizes Fowler's analysis patterns [15] from the perspective of archetype patterns. Table 3-2 summarizes Hay's data patterns [16] from the perspective of archetype patterns. Table 3-3 summarizes Silverstone's universal

data patterns [17] from the perspective of archetype patterns. All these patterns describe similar phenomena of universe of discourse but are modelled differently. Models, where one and the same thing is modelled differently, are *semantically heterogeneous* [37].

Despite similarities between these patterns we found ideas from Fowler, Hay and Silverstone's patterns that we use in improved models of archetypes and archetype patterns.

From Fowler's quantity we took a concept of measure and an idea for modelling formulas. The concept of "attributes" we use in different archetypes is based on Fowler's observation and measurement. From Fowler's planning pattern and from Silverston's work effort data model pattern we took ideas for our proposed business process archetype pattern.

Table 3-3: Silverston's Data Patterns and the Archetype Patterns

Silverston's Data Pattern	***Archetype Pattern***
People and Organizations	Party and Party Relationship
Products	Product
Ordering Products	Order
Shipments	Party Relationship, Process
Work Effort	Inventory and Order, Process
Invoicing	Order, Process
Accounts and Budgeting	Inventory and Order
Human Resources	Party and Party Relationship
Enterprise Data Model	Inventory and all other archetypes
Sales Analysis	Inventory and all other archetypes

From Hay we got the idea to expand "discrete" models to "continuous" ones which we use in our business process archetype pattern. We use some Hay's ideas and patterns like the *laboratory* pattern and the *material requirements planning* pattern in the domain model of laboratory (Section 4.2) we developed.

When I first read about archetypes and archetype patterns by Arlow and Neustadt, I had a feeling that archetype pattern for a *document* is missing. This lacuna is filled by Hay when describing documents. We plan to use Hay's document model together with Bjørner's document [26] model and Lindqvist and Christensen's electronic document [45] model as a base model for the development of the domain model for documents. In Section 4.4 we describe an idea where archetype patterns based business domain models and document domain models are cornerstones for development of software factories.

3.4 Fine Tuning of Models

After evaluation, the initial model should be finalized. Current unit tests (contextual and semantic models) are controlling that:

1. Needed types (and no more) exist and it is possible to create (use) these types;
2. These types have needed (and no more) properties and methods.

In fine tuning stage we will supplement a semantic model to ensure not only the existence of necessary properties and methods, but also their correct runtime behaviour as we have shown in Section 2.2.5. Take a look at the following test (semantic model, narrative specified as unit test).

```
[TestMethod] public void A03160200_WebPageAddress_CorrectValues(){
 Func < WebPageAddress, string, DateTime, DateTime, bool > isCorrect =
   (v, u, x, y) = (v.Url == u) && (v.Valid.From == x) && (v.Valid.To == y);
   string a = "";
   DateTime f = DateTime.MinValue;
   DateTime t = DateTime.MaxValue;
   var w = new WebPageAddress();
   Assert.IsTrue( isCorrect(w, a, f, t));
   a = "www.www.ee";
   f = DateTime.Now.AddYears(−5);
   t = DateTime.Now.AddYears(5);
   w = new WebPageAddress(a, new TimeInterval(f, t));
   Assert.IsTrue( isCorrect(w, a, f, t) );
}
```

With this unit test we verify that the class *WebPageAddress* can be created without parameters as well as with parameters and that after creation the created *WebPageAddress* object holds correct property values.

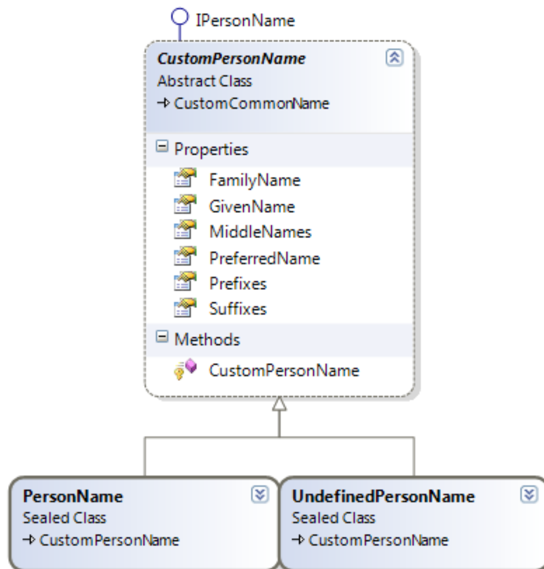

Figure 3-11: Custom, Class and Undefined Pattern of Physical Model

The pattern of this unit test is as described in Section 2.2.4.

1. We define a helper function for verification of post conditions (e.g. *isCorrect*);

2. We define preconditions (e.g. a, f, t, and w);
3. We execute a piece of code (e.g. $new\ WebPageAddress()$);
4. We verify post conditions (e.g.$Assert.IsTrue(isCorrect(w, a, f, t));$).

The logical model of the resultant party archetype pattern is shown in Figure 3-15 and the party relationship archetype pattern is shown in Figure 3-16. As mentioned previously (Section 2.2.4 and Section 3.2.4), for logical models we use the "language" of interfaces. The physical model for the person name ($PartyName$) archetype (illustrated in Figure 3-11) is following.

```
public abstract class CustomPersonName:
   CustomCommonName, IPersonName {
 protected CustomPersonName(
     string prefixes = "", string givenName = "", string middleName = "",
     string familyName = "", string preferredName = "", string suffixes = "",
     string uses = "", bool isDefault = false, IFromToTime valid = null)
       : base(uses, isDefault, valid) {
        Prefixes = prefixes?? string.Empty;
        GivenName = givenName?? string.Empty;
         MiddleNames = middleName?? string.Empty;
        FamilyName = familyName?? string.Empty;
        PreferredName = preferredName?? string.Empty;
        Suffixes = suffixes?? string.Empty;
  }
  public string Prefixes { get; private set; }
  public string GivenName { get; private set; }
  public string MiddleNames { get; private set; }
  public string FamilyName { get; private set; }
  public string PreferredName { get; private set; }
  public string Suffixes { get; private set; }
}
public sealed class PersonName : CustomPersonName {
      public PersonName(
        string prefixes =, string givenName =, string middleName =,
        string familyName =, string preferredName =, string suffixes =,
        string uses = "", bool isDefault = false, IFromToTime valid = null)
         : base(prefixes, givenName, middleName, familyName,
                      preferredName, suffixes, uses, isDefault, valid) { }
}
public sealed class UndefinedPersonName : CustomPersonName {
     private static readonly IPersonName instance =
          new UndefinedPersonName();
     private UndefinedPersonName() { }
     public static IPersonName Instance { get { return instance; }
     }
  }
```

We define logical models (Row 3 of ZF, Table 2-2), according to semantic (Row 2 of ZF, Table 2-2) models specified as unit tests (described in Section 2.2), by using interfaces. We then use open for extensions common archetypes (abstract classes) where we implement all needed functionality. Finally, we have closed for modification archetypes (sealed classes). We also use closed for modifications special cases ($UndefinedPersonName$ for instance) as suggested by Fowler [6].

3.5 Definitions of Models

We briefly describe the resultant archetype patterns we propose for development of business domains.

3.5.1 Quantity Archetype Pattern

A quantity (Figure 3-12) is an amount of something measured according to some standard of measurements.

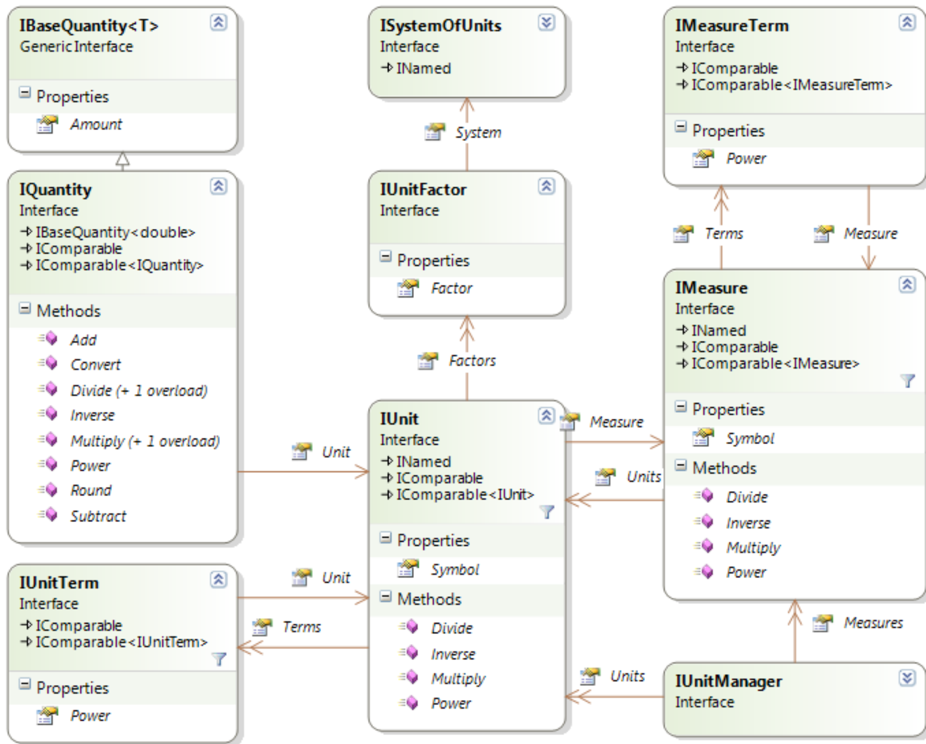

Figure 3-12: Quantity Archetype Pattern

Differently to the *quantity archetype* designed by Arlow and Neustadt [14], we have the concept of measure (inspired by Fowler's quantity pattern [15]) and a unit factors based conversion mechanism (modified idea inspired by Borland Delphi IDE). The *unit* archetype ($IUnit$) represents a standard of measurement.

Unit attributes are *name, description* (both derived from $IName$), *symbol, measure* (Table 3-4), *terms* and *factors*.

Unit *terms* are used to define composed units precisely for automatic unit conversion. For instance, the terms for unit *litre* ($dm * dm * dm = dm^3$) and *newton* ($kg * \frac{m}{s^2}$) are illustrated in Table 3-5. *Unit factor* is an amount of base units equal to a unit in question. For one and the same unit more than one unit factors are possible. For example, the unit kilogram has the factor equal to 1 in SI (International System of Units) system but the factor equal to 1000 in CGS (Centimetre Gram Second System of Units) system. Similarly to the *unit* and the *unit term* archetypes there are the *measure* and the *measure term* archetypes, exemplified in Table 3-6. Additionally to properties, some common methods (Table 3-7) are defined for *quantity*, *unit* and *measure* archetypes

Table 3-4: Simple unit attributes

Unit	**Semantics**	***Name***	***Symbol***	***Measure***	***Description***
Metre	Unit of distance	“metre”	“m”	Distance	“The metre is….”
Kilogram	Unit of mass	“kilogram”	“kg”	Mass	“The kilogram is …”
Second	Unit of time	“second”	“s”	Time	“The second is ….”

Table 3-5: Attributes of the Unit and Unit Terms Archetype

Unit	***Name***	***Symbol***	***Measure***	***Unit*** $\boldsymbol{Term(s)}$
Litre	“litre”	“l”	Area	dm^3
Newton	“newton”	“N”	Force	kg^1, m^1, s^-2

Table 3-6: Attributes of the Measure and the Measure Term Archetypes

Measure	***Name***	***Symbol***	$\boldsymbol{MeasureTerm(s)}$
Time	“time”	“T”	
Distance	“distance”	“L”	
Mass	“mass”	“M”	
Area	“area”	“A”	L^3
Force	“force”	“F”	M^1, L^1, T^3

Table 3-7: Common Methods of the Quantity Pattern

Method	***Quantity***	***Unit***	***Measure***
Add	$1m + 2dm = 1.2m$	Not defined	Not defined
$ConvertToUnit$	$1m = 10dm$	Not defined	Not defined
Divide	$\frac{10m}{5} = 2m;\ \ \frac{10m}{2s} = 5\frac{m}{s}$	$m : s\ = \frac{m}{s}$	$V : L = A$
Inverse	$(10s)^{-1} = 0.1Hz$	$s^{-1} = Hz$	$T^{-1} = f$
Multiply	$5\frac{m}{s} * 2s = 10m,$ $10m * 2 = 20m$	$\frac{m}{s} * s = m$	$L * L = A$
Power	$(10dm)^3 = 1000l$	$dm^3 = l$	$L^3 = V$
Round	$1.378m \approx 1.4m$	Not defined	Not defined
Subtract	$1m - 2dm = 0.8m$	Not defined	Not defined

3.5.2 Money Archetype Pattern

The *money archetype pattern* (Figure 3-13) is a special case of the quantity archetype pattern.

Money archetype represents an amount of a specific currency. A currency is accepted in one or more location. *Money* has properties *amount* and *currency*. A currency (*ICurrency*) has properties *name* (e.g. "Euro"), *description* (e.g. "Currency of the European Community") (derived from *INamed*), *introduction date* and *expiration date* (derived from *ITimeLimited*), *alphabetic code* (e.g. "EUR"), *numeric code* (e.g. 978 according to ISO 4217), *major unit symbol* (e.g. "€"), *minor unit symbol* (e.g. "c") and *ratio of minor unit to major unit* (e.g. 100). A currency can be an ISO currency (defined by ISO 4217) and a non ISO currency (e.g. customer loyalty credits).

Figure 3-13: Money Archetype Pattern

The *payment* archetype ($IPayment$) represents money paid by one party to another, in return for goods or services. The *payment method* ($IPaymentMethod$) archetype represents a medium by which a payment can be made. *Cash* ($ICash$), *check* ($ICheck$) and *payment card* ($IPaymentCard$) are payment methods.

Similarly to unit conversion (Section 3.5.1) each currency has one or more exchange rates. An exchange rate (time limited) is a conversion factor to a "base" currency. Normally the "base" currency is a domestic currency and the exchange rate of a domestic currency is equal to 1.

For example in some EU countries the domestic currency is EUR and the exchange rate for EUR in those countries is therefore 1. However, the currency conversion mechanism allows using multiple "base" currencies. For this reason each exchange rate has a type. In addition to the use of multiple "base" currencies, this mechanism provides an opportunity to define different rates for sales, purchases, major clients and etc. Which exchange rate must be used in concrete currency conversion is determined by exchange rate type ($IExchangeRateType$) applicability rules.

3.5.3 Rule Archetype Pattern

We use semantically the same rule pattern (Figure 3-14) as designed by Arlow and Neustadt [14].

The rule archetype ($IRule$) represents a business constraint and is defined by a sequence of rule elements ($IRuleElement$). The rule context archetype ($IRuleContext$) contains an informational context for the evaluation of a rule. Also in a rule context, information is represented by a sequence of rule element ($IRuleElement$) archetypes. Rule elements are either operators ($IOperator$) or variables ($IVariable$). Rules can be grouped into rule sets ($IRuleSet$).

An active rule ($IActiveRule$) represents a type of rule that automatically executes an activity after the evaluation of a rule. It is possible to override a rule by a given special value ($IRuleOverride$). *Why*, *when* and *who* established an override are the properties of the *rule override* archetype.

Although Arlow and Neustadt have a third type of rule element, namely proposition ($IProposition$), this difference is technical rather than substantive. In our model a proposition is just a Boolean type variable ($IVariable$). Such a technical realization allows us to easily determine what type of rule elements may be elements of the rule ($IRule$) archetype and what can be elements of the rule context ($IRuleContext$) archetype. If rule elements of $IRule$ archetype can be all rule element types ($IOperator$ and $IVariable$ including $IProposition$), then rule elements of $IRuleContext$ archetype can only be variables ($IVariable$), i.e. including propositions ($IProposition$).

Similarly to Arlow and Neustadt's rule archetype pattern, operators can be logical operators ($and,\ or,\ xor$ and not) and comparison operators ($equal,\ not\, equal,\ greater,\ not\, greater,\ less,\ not\, less$). For every generic variable ($IVariable < T >$) operations (*equal*, *not equal*, *greater*, *not greater*,

less, *not less*) have to be defined according to a variable type T. Additionally to propositions (T is Boolean), logical operations ($and,\ or,\ xor$ and not) have to be defined. In principle, currently used operators with defined operations can be complemented with arithmetic operators and arithmetic operations related to these operators.

Figure 3-14: Rule Archetype Pattern

3.5.4 Party Archetype Pattern

In real world a party represents an identifiable, addressable unit that may have a legal status and that normally has an autonomous control over its actions [14 p. 122]. In information systems, the party archetype pattern (Figure 3-15) is a representation of information about persons and organizations needed for information systems in order to achieve their specific business goals.

A party identifier ($IParty.Identifier$) uniquely identifies a *party*. A *registered identifier* ($IRegisteredIdentifier$) has been assigned (e.g. VAT registration number, national insurance number, passport number, credit card number, domain names, etc.) to a *party* by a recognized statutory body and is

valid for some period of time. A party authentication ($IParty.Authentication$) is an agreed and trusted way to confirm that *parties* are who they say they are.

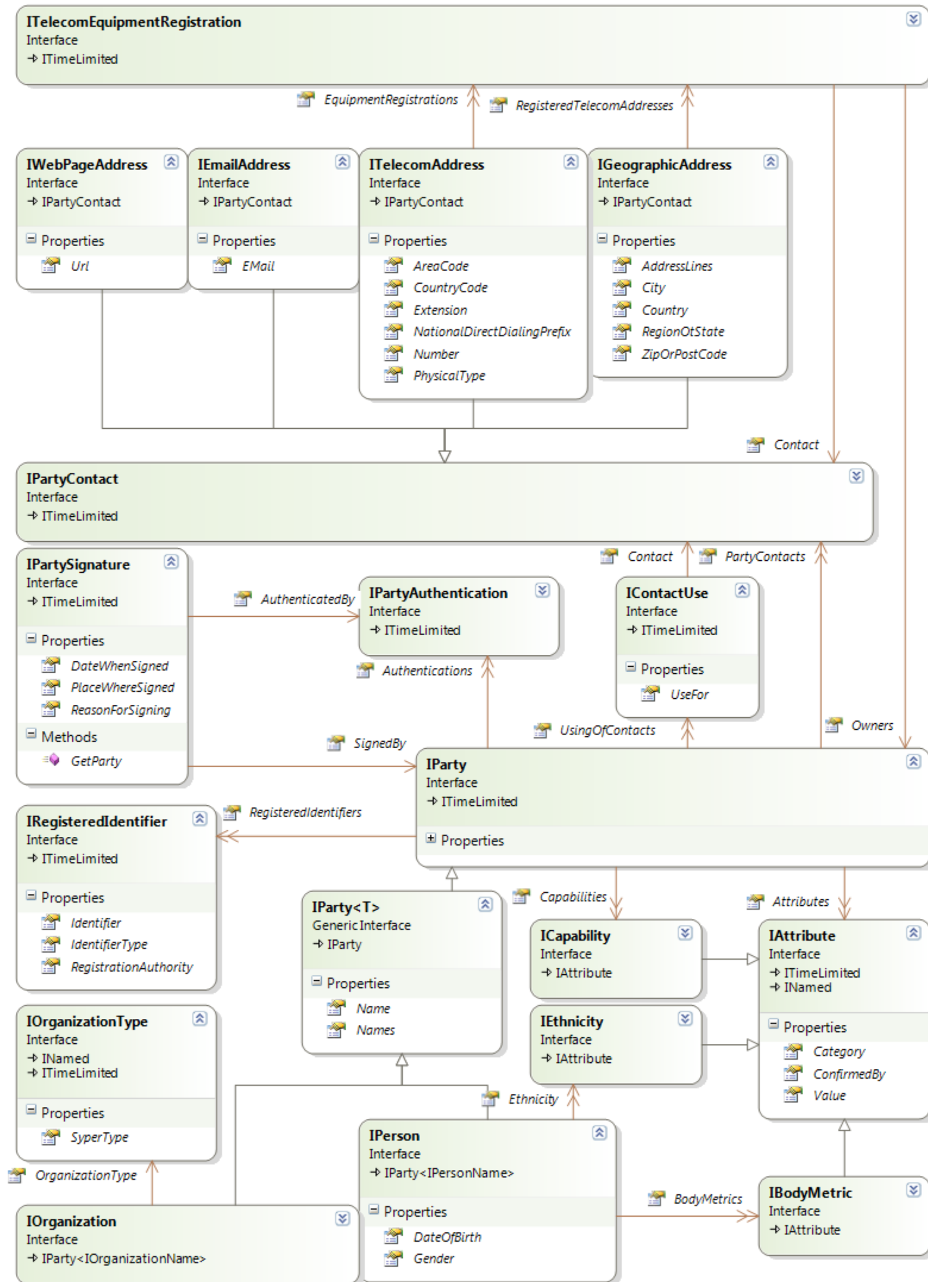

Figure 3-15: Party Archetype Pattern

A party contact ($IWebPageAddress$, $IEMailAddress$, $ITelecomAddress$ and $IGeographicAddress$) represents information that is used to contact a party. Differently to Arlow and Neustadt's contacts, we added the *telecom equipment registration* ($ITelecomEquipmentRegistration$) and the *using of contact* ($IContactUse$) archetypes. Such a solution allows to register more than one telecom addresses (old equipment such as phones and faxes) in any particular geographic address and to register any telecom address to more than one party.

For generalization (e.g. $ICapability$, $IEthnicit$ and $IBodyMetric$ are all *attributes*) and flexibility (to add new attributes even at runtime) reasons we introduced the *attribute* ($IAttribute$) archetype into the party archetype pattern. Attribute has *valid from*, *valid to*, *name*, *category*, *value* and *conformed by* properties. For example the notation "person is 176 cm tall, measured by Dr Smith at 3rd of May 2000" is an attribute with *category* denoting "body metrics", *name* denoting "is tall", *value* denoting "176 cm", *confirmed by* denoting "Dr Smith", and *valid from* denoting "3rd of May 2000".

P*erson* and *organization* are types of parties. Additionally to common party ($ICommonParty$) properties, the *person* archetype has some specific properties. An *ethnicity* ($IEthnicity$) is an attribute which is used to classify people according to their racial, national, religious, linguistic, cultural origin or by other background. A *body metric* ($IBodyMetric$) is an attribute which is used to store information about a human body such as size, weight, hair colour, eye colour, clinical laboratory measurements, diagnosis, and etc. The *organization* archetype inherits most of its attributes from the common party archetype.

Differently from Arlow and Neustadt, there is the *organization type* ($IOrganizationType$) property and no organization subtypes (*organization unit* and *company*). This gives us possibilities to flexibly adjust types of organizations, even during runtime and gives organizations possibilities to flexibly redesign organization structures as exemplified in Section 3.3.2 and illustrated in Figure 3-8. For example, an organization can reorganize marketing team via marketing department to marketing division.

3.5.5 Party Relationships

A *party relationship* (Figure 3-16) captures the fact that there is a semantic relationship between two parties in which each party plays a specific role. There are only binary relationships (flexible, conceptually cleaner and easier to understand) between exactly two parties. This means that every n-ary relationship is reduced to two or more binary relationships.

The *party role* ($IRole$) archetype captures semantic of a role (e.g. mother, father, customer, patient, student, etc.) played by a *party* in a particular *party relationship* ($IRelationship$). Not separating parties from roles they are involved with is a quite common design mistake in information systems.

I have seen a hospital information system, which refused to recognize physician of this hospital as a patient. As persons we are in roles of students, clients, patients or physicians in relationship with other parties for some limited period of time. A *role* is related with a party (e.g. Gunnar is a software developer in Clinical and Biomedical Proteomics Group) in some period of time with some concrete requirements for responsibility and conditions of satisfaction.

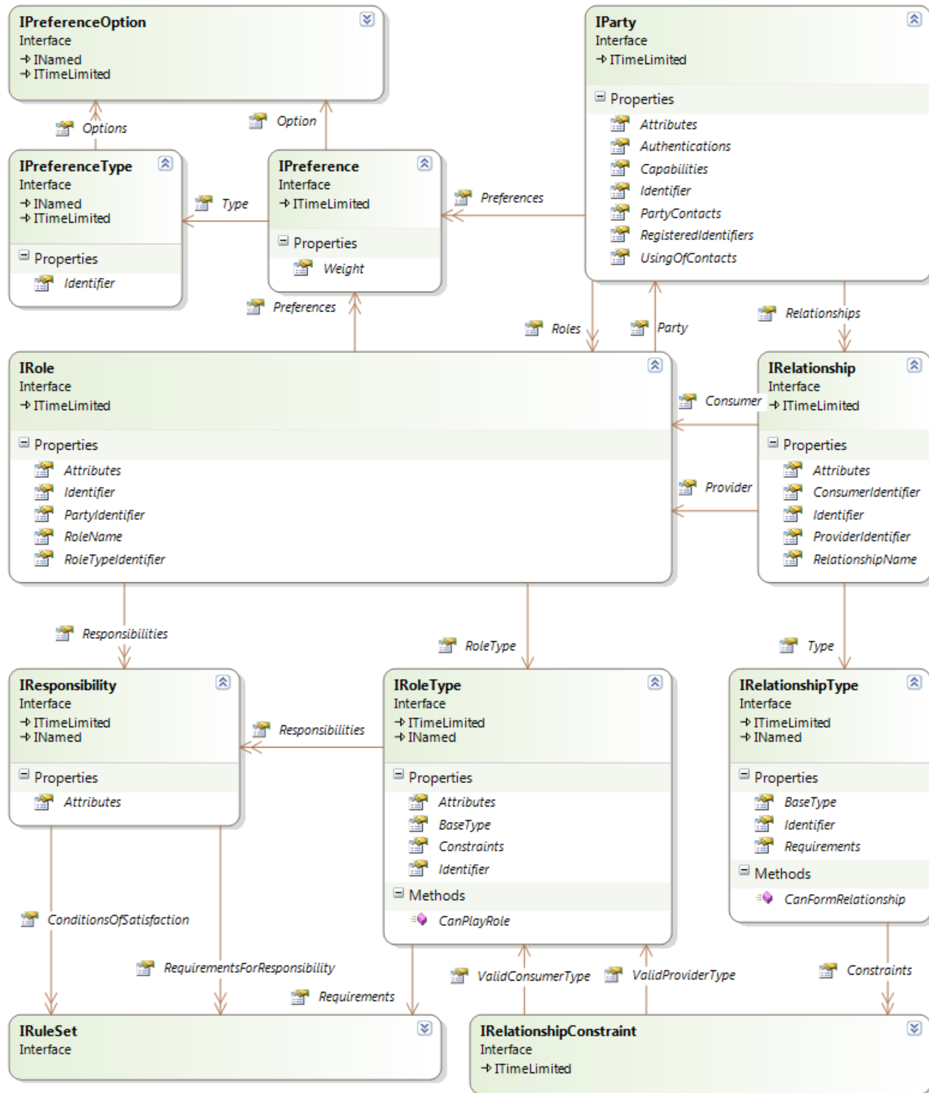

Figure 3-16: Party Relationship Archetype Pattern

A *role type* (*IRoleType*) (e.g. software developer in Clinical and Biomedical Proteomics Group) provides a way to store all the common information for a set of party role instances. Using *subtype* property (collection) of a role type

archetype it is possible to build complicated role type hierarchies. The *party role constraint* ($IRuleSet$, Section 3.5.3) specifies the type of a party and other conditions that are needed for the party to play a specific role.

A *preference* ($IPreference$) represents a party's or role's expressed choice of (or linking for) something. It is often a set of possible or offered options [14 p. 150]. General preferences (e.g. dietary preferences) held by a party ($IParty$); specific preferences (e.g. working time preferences) held by a party playing a particular party role ($IRole$). *Preference type* ($IPreferenceType$) is specified by a *name*, a *description* and a range of possible *options* ($IPreferenceOption$) for a preference. Each *preference* specifies exactly one option from a range of options listed in its *preference type*. Preference type may be related to a specific *product* or a *service* (Section 3.5.6).

3.5.6 Product Archetype Pattern

The *product archetype pattern* (Figure 3-17) represents a generalized model for products (goods or services) parties (persons or organizations) produce, use, sell or buy. Products can be unique things (e.g. *Mona Lisa* by Leonardo da Vinci), identical things (e.g. loaves of bread), identifiable things (e.g. cars with unique serial numbers), measured things (e.g. flour measured in tons or kilograms) or services.

The *product type* ($IProductType$) archetype describes common properties of a set of goods or services [14 p. 208]. The *product instance* ($IProductInstance$) archetype represents a specific instance of a product type [14 p. 208]. Each product type has a unique *identifier*. Each product instance can be uniquely identified by a *serial number*.

The product feature type ($IProductFeatureType$) archetype and the *product feature instance* ($IProductFeatureInstance$) archetype are used for product specifications. The product *feature type* archetype represents a type of a product feature ($e.g. colour$) and its possible values (e.g. *{blue, green, yellow, red}*). The product *feature instance* archetype represents a specific feature (such as *colour*) and its value (e.g. *blue*). Each *product type* has a set of *possible features* ($IProductFeatures$). Possible features can be *mandatory* or *optional*. Differently from Arlow and Neustadt [14], where a possible feature value is just a collection of objects, in our model a possible value is a collection of attributes ($IAttribut$).

The batch archetype ($IBatch$) describes a set of product instances of a specific product type that are to be tracked together for example for quality control purposes [14 p. 215]. Differently from Arlow and Neustadt, we have defined batch as a special product instance and therefore derived from $IProductInstance$ archetype. We also have added a batch type ($IBatchType$) as a special product type inherited from $IProductType$. The reason for this is that, for instance, batches of samples in clinical laboratories can sometimes be just normal products we have to manage and audit trail. This means, that like

any product instance, each batch has a serial number and is composed according to specifications defined by a batch type.

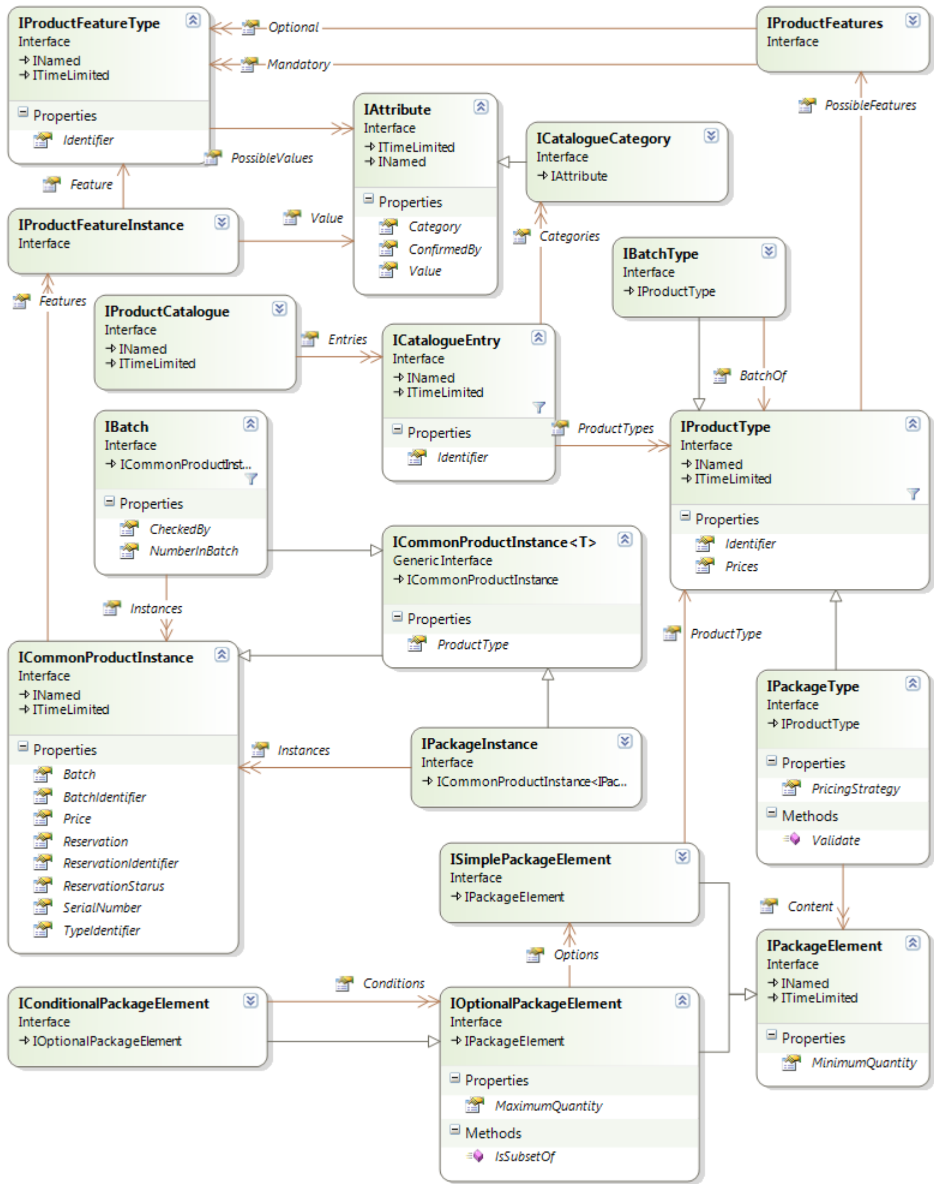

Figure 3-17: Product Archetype Pattern

Each batch contains only one type of *product instances* identified by *BatchOf* property. Product instances in batches can be identified by serial numbers. Batches may be optionally validated by one or more parties. Validations are indicated by party signatures (*IPartySignature*). In our batch archetype,

differently from Arlow and Neustadt's, dates (sell by, best before, etc.) and also allowed maximum and minimum product instances are "product features" and therefore are described by product *feature type* and product *feature instance*. In comparison to the batch model we introduce, the original model, designed by Arlow and Neustadt, is a special case.

The product catalogue ($IProductCatalogue$) archetype represents a persistent store of product information [14 p. 221]. The catalogue entry ($ICatalogueEntry$) archetype represents information about a specific type of product held in a product catalogue [14 p. 224]. In our product catalogue model, differently to Arlow and Neustadt's product catalogue model [14 pp. 221-225], each catalogue entry is time limited and a category is an attribute ($IAttribute$) and not only a string.

A selection of different products grouped together as a unit is often called a package. The package is also a product and therefore we have a *package type* ($IPackageType$) and a *package instance* ($IPackageInstance$) archetypes inherited accordingly from the *product type* and the *product instance* archetypes. Although our package model is based on rule-driven package specification explained and illustrated by Arlow and Neustadt [14 pp. 230-242], we have introduced some significant changes.

The *package type* is a *product type* that specifies a content of a package. The content of a package is a collection of *package elements* ($IPackageElement$). A package element can be either simple ($ISimplePackageElement$), optional (*IOptionalPackageElement*), or conditional (*IConditionalPackageElement*).

Packages with fixed content can be modelled by using simple package elements. An example of such package is a meat package containing for example a pound of minced meat, one chicken, four pork chops and so one. The simple package element archetype has product type (minced meat, chicken, etc.) and amount ($IQuantity$, see Section 3.5.1) properties. With *optional package element* it is possible to model packages similar to meal sets in restaurants where a customer, for example, can pick one starter, one main course and one dessert from a fixed selection of starters, main courses and desserts. With *conditional package elements* it is possible to add conditions to optional packages. For example, if a customer orders "*Calamars marinés aux feuilles de citron*" for the starter, he/she cannot order "*Assortemente do chariot de desserts*" for the dessert. Concluding, packages in our product model, instead of *minimum* and *maximum* integer values, have *minimum* and *maximum* quantities and all package types (simple, optional and conditional) are modelled similarly by using general and unified concept of a package element.

Our product relationship (Figure 3-18) pattern, differently from Arlow and Neustadt's product relationship, is similar to the party relationship pattern (Section 3.5.5). This means, that in addition to the *product relationship* ($IProductRelationship$) archetype, we have the *product relationship type* ($IProductRelationshipType$) archetype. The product relationship archetype

stores all common information for a set of product relationships and describes constraints for valid "provider" and "consumer" product types in product relationship.

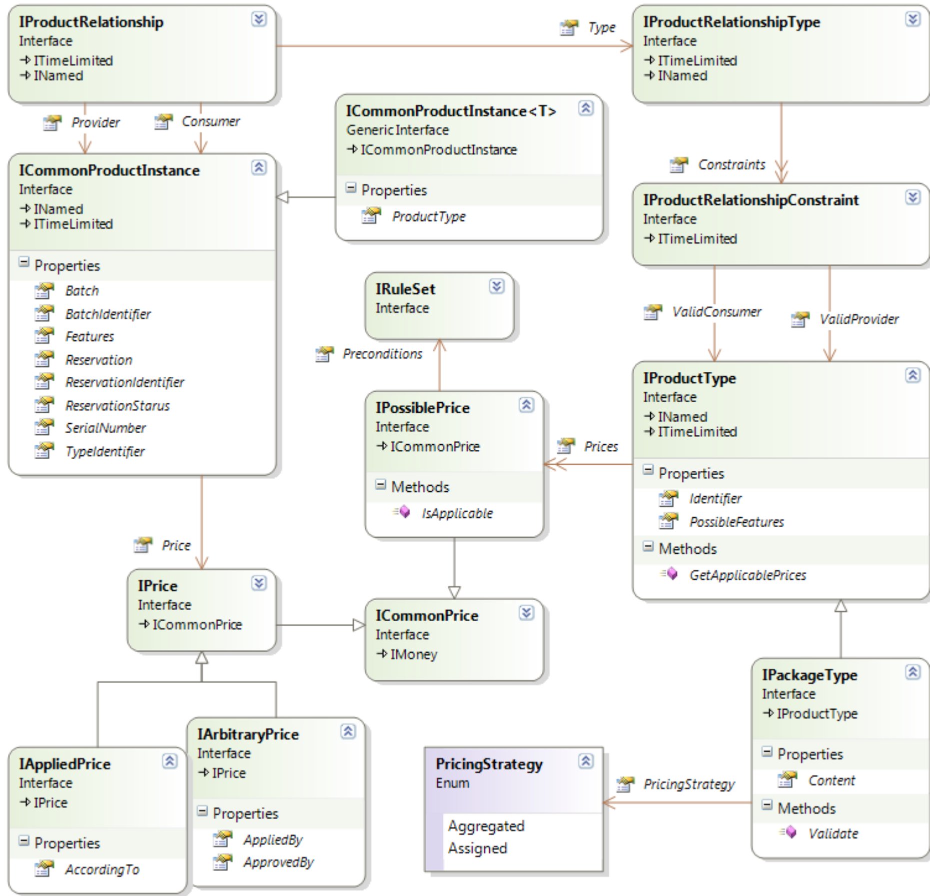

Figure 3-18: Product Relationship and Pricing

We have a different model (comparing to Arlow and Neustadt) also for pricing (Figure 3-18). The *product type* archetype stores possible prices (*IPossiblePrice*). *Possible prices* have a set of preconditions (rule set, Section 3.5.3) in order to apply. The *product instance* holds a price, which can be either an applied (*IAppliedPrice*) or an arbitrary (*IArbitraryPrice*) price. Applied prices are prices from a collection of possible prices of a *product type* and are applied only when preconditions are fulfilled. Arbitrary prices can be applied by some party and must be signed by an applier. In some cases an arbitrary price must be approved by other authorized persons. The pricing of packages can be either assigned (a package has a set price) or aggregated (the price of a package depends on prices of package components).

There are some special products (Figure 3-19) in the product archetype pattern. These special products are measured product, service, unique product and identical product.

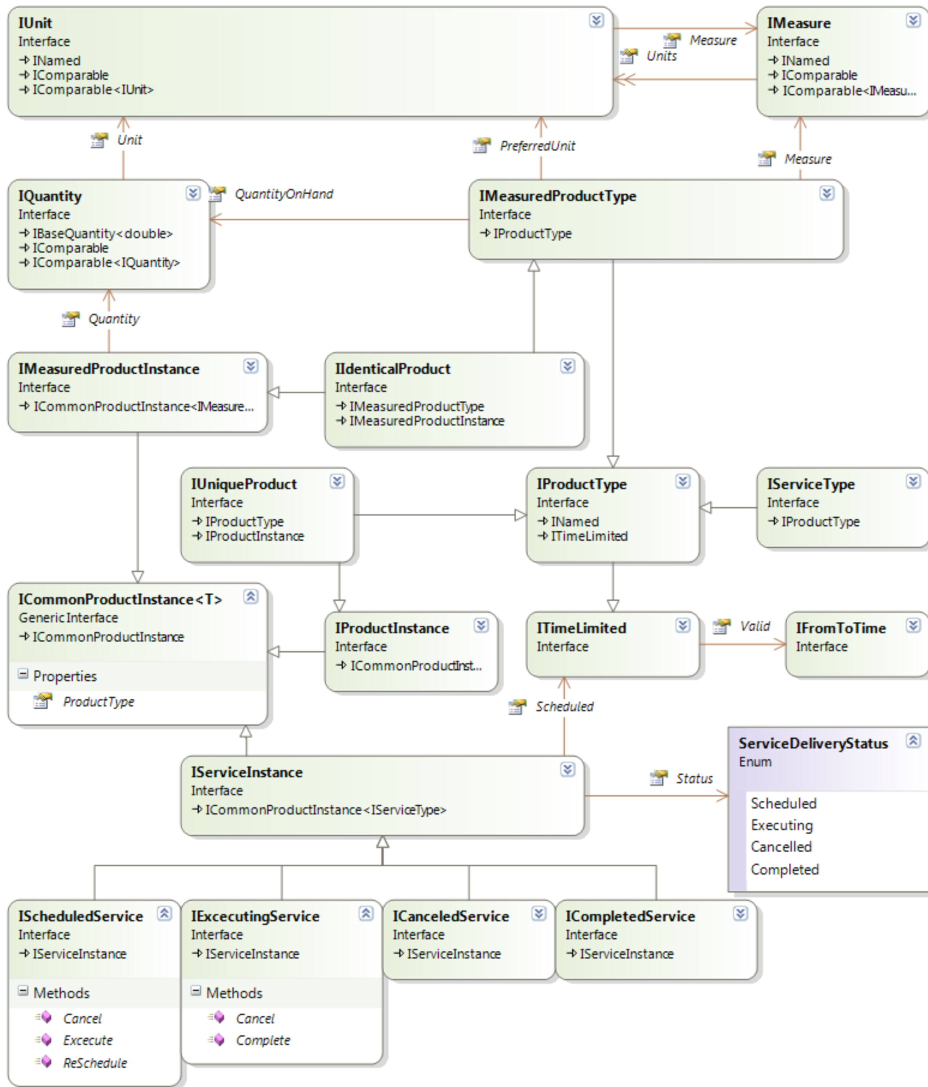

Figure 3-19: Special Product Types

The measured product type (*IMeasuredProductType*) and the measured product instance (*IMeasuredProductInstance*) archetypes are used for products where it is important to account the quantity of a product. The measured product type has measure (e.g. height, weight) and preferred unit (e.g. metre, kilogram) properties. The measured product type also records quantity on hand. The measured product instance has a quantity property.

A service is a process or an activity that is offered for sale [14 p. 254]. The service type (*IServiceType*) is a product type (*IProductType*). A service is available for a period of time. The service instance (*IServiceInstance*) is a product instance. The service instance has *scheduled* and *actual* time periods. Additionally to the product instance, the service instance has a *service delivery status* (*ServiceDeliveryStatus*), which reflects the lifecycle of service execution.

We have introduced the *state pattern* [12 pp. 57-60] to the service archetype. According to the state pattern, different service states (*IScheduledService*, *IExecutingService*, *ICancelledServic* and *ICompletedService*) are represented by individual classes. The effect is moving behaviour methods (*Cancel*, *Executing*, *Complete* and *ReScheduling*) to where they belong. Additionally, such swapping of a single field into a bunch of separate classes is in agreement with the Single Responsibility Principle [12 p. 60].

Most products in modern times are mass produced. However, some products are unique or so called “one-off” products. The *unique product* (*IUniqueProduct*) in our model, differently from Arlow and Neustadt’s, inherits both the *product type* and the *product instance* archetypes. This means, that depending on the context a *unique product* acts either as a normal *product type* or a normal *product instance*. It also means that batches of unique products (e.g. archaeological findings or museum specimens) are possible.

The *identical product* archetype is applied, when a product instance (mass production) is an identical copy of a product type. Also the identical product (*IIdenticalProduct*) archetype in our model has double inheritance. The *identical product* archetype is inherited from the *measured product type* and from the *measured product instance*. This means, that the *identical product* archetype, depending on the context, acts either as a normal *product type* or a normal *product instance*.

3.5.7 Inventory Archetype Pattern

The *inventory archetype pattern* (Figure 3-20) represents a model for managing a stock (or store) of products (goods or services). The *inventory* (*IInventory*) archetype represents a collection of *inventory entries* (*IInventoryEntry*) held in a stock by a business. An *inventory entry* records a *product type* (*IProductType*) and a collection of *available instances* of that product type (*ICommonProductInstance*).

In comparison to the *inventory archetype pattern* by Arlow and Neustadt [14 pp. 267-301], the inventory in our model belongs explicitly to some party (*IInventory.BelongsTo*). The *inventory archetype pattern* is designed as a repository [6 p. 322] and because of performance reasons is designed to support the lazy load pattern [6 p. 200]. According to the *lazy load* pattern, an object does not contain all data it needs, but knows how to get it when necessary. This means that *inventory* does not contain all *inventory entries*, but only knows how to get them.

The same is true for the *inventory entry* archetype. The inventory entry archetype, by knowing an identifier of a *product type* (*IProductTypeIdentifier*) and a *collection of product serial numbers* (*IProductSerialNumber*) is able to get information about a *product type* (*IProductType*) and *product instances* (*IProductInstance*) when needed.

There are two *inventory entry types:* inventory entry for *products* and inventory entry for *services*. A *product inventory entry* (*IProductInventoryEntry*) is an inventory entry that holds a set of available *product instances* of the same *product type*. Each *product inventory entry* may have a *restock policy* (*IRestockPolicy*) which, by a set of rules (see Sections 3.5.3), determines when inventory items need to be reordered. Outstanding purchase orders (*IPurchaseOrder*, see Section 3.5.8) are used to calculate a *quantity of an ordered items* for a particular period.

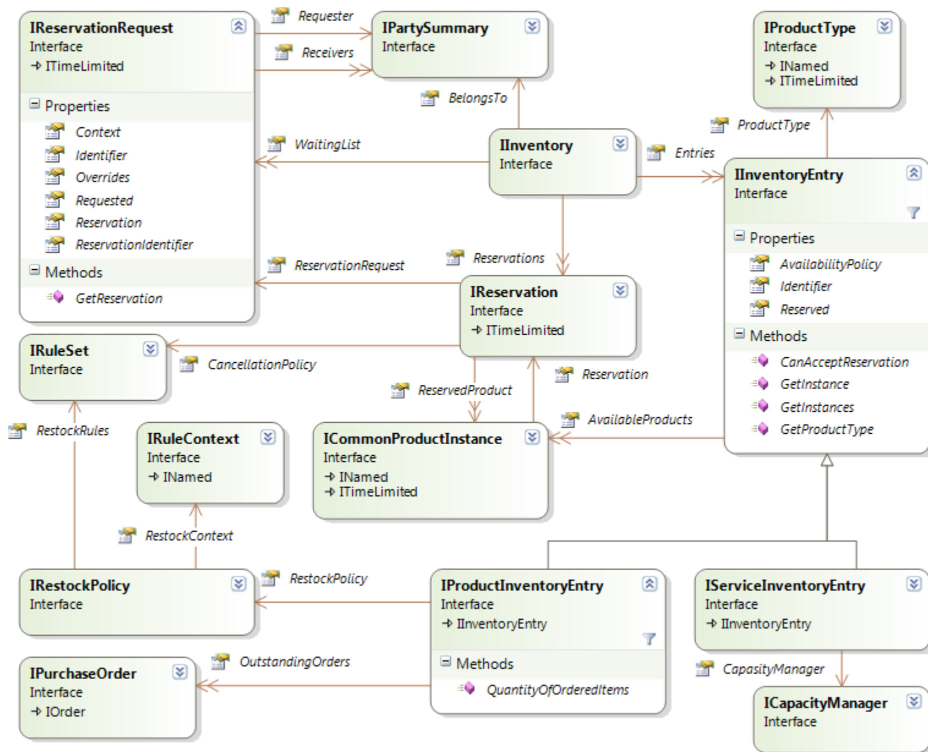

Figure 3-20: Inventory Archetype Pattern

A service inventory entry (*IServiceInventoryEntry*) is an inventory entry that holds a set of service instances of the same service type. A capacity manager (*ICapasityManager*) manages the utilization of a capacity by releasing service instances.

One of the key functions of an *inventory* is to decide whether *product* or *service instances* are available for sale. The *reservation archetype*

($IReservation$) represents an assignment of one or more *product instances* to one or more *receivers*. The *reservation request* ($IReservationRequest$) archetype represents a request from a *requester* ($IPartySummary$) for a *reservation* to be made. Each reservation has a unique identifier ($IReservationIdentifier$) which uniquely identifies a reservation. This *reservation identifier* is used for product reservation. For example if $reservation$ property of $ICommonProductInstance$ is not $IUndefinedReservation$ then this product instance is reserved.

The availability policy ($IInventoryEntry.AvailabilityPolicy$) archetype and cancellation policy ($IReservation.CancellationPolicy$) archetype are rule sets (Section 3.5.3) describing either availability or cancellation rules.

3.5.8 Order Archetype Pattern

The main archetypes of the *order archetype pattern* (Figure 3-21) are the *order manager*, the *order* and the *order line*.

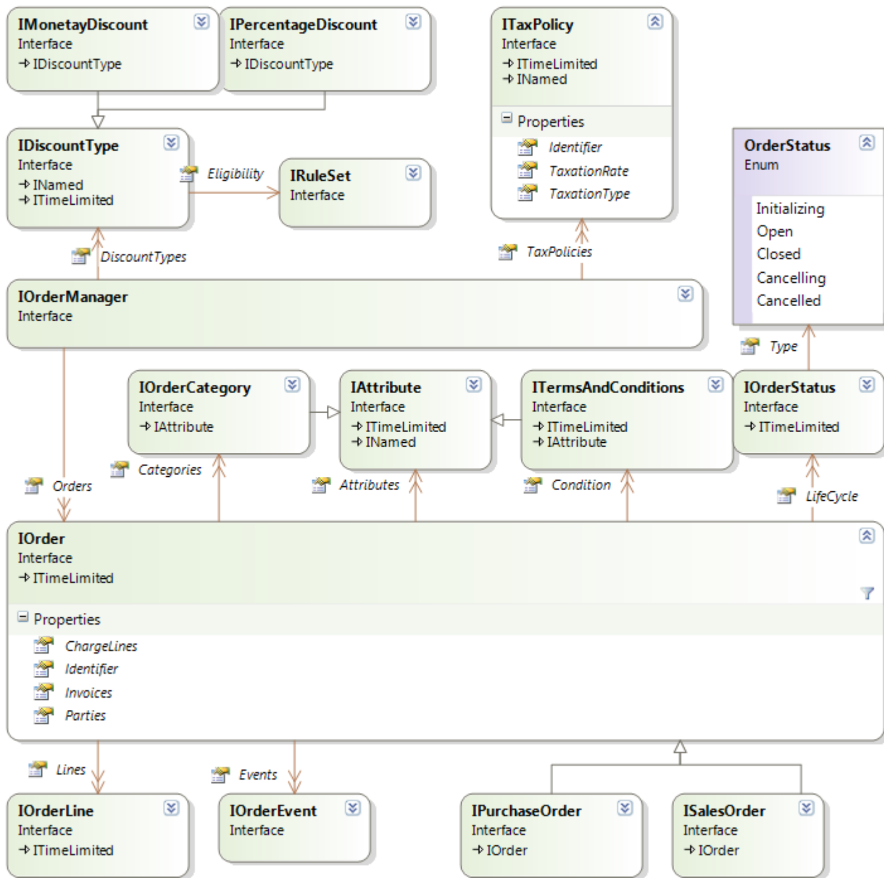

Figure 3-21: Order Archetype Pattern

In comparison to the *order archetype pattern* by Arlow and Neustadt [14 pp. 303-389], our model has a different *order status* management system and is designed as a static document which maintains information about company events (e.g. *order is created*, *payment is made*, *payment is accepted*, *delivery is sent*, and etc.). The dynamic part of Arlow and Neustadt's order archetype pattern (designed by activity diagrams) is removed from the order archetype pattern and is designed (Appendix 7.2) by using the *business process archetype pattern* (Section 3.5.9).

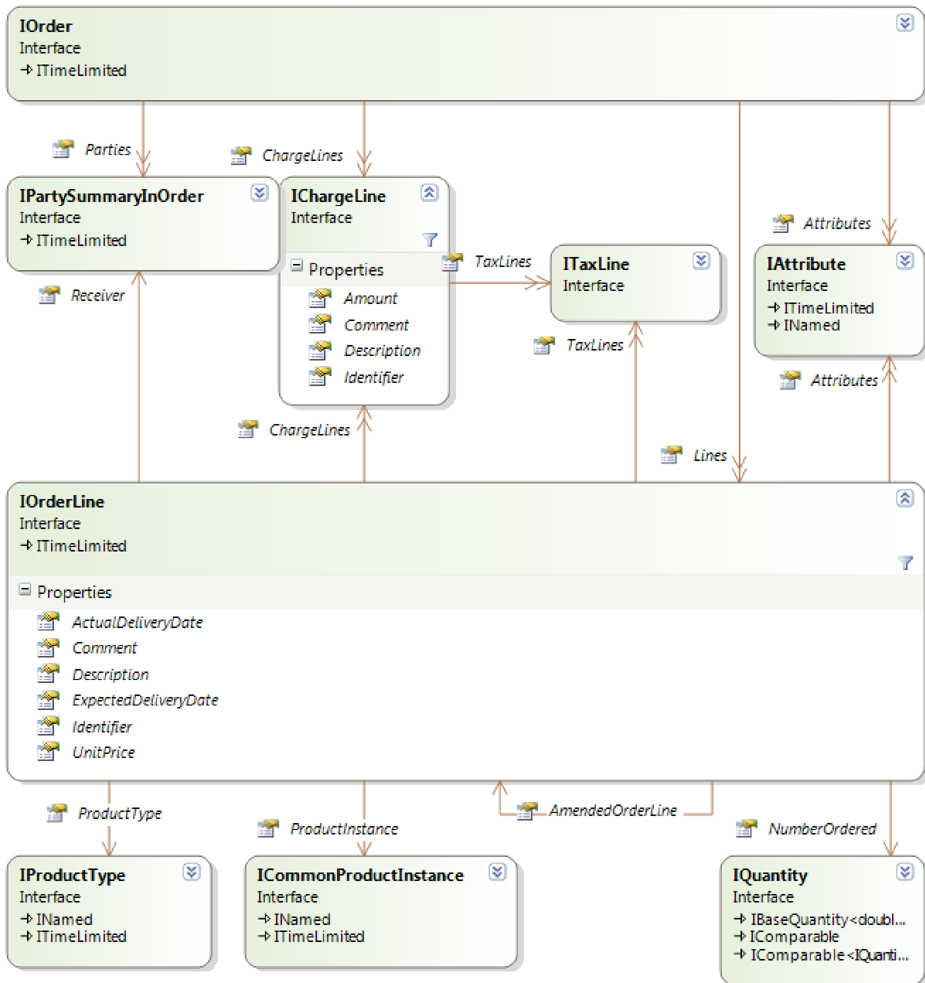

Figure 3-22: Order Line Archetype

The order archetype, according to Arlow and Neustadt [14 p. 304], represents a record of a request from a buyer to a seller to supply some goods or services. There are two order types: a *purchase order* (*IPurchaseOrder*) and a *sales order* (*ISalesOrder*).

Orders can be categorized ($IOrderCategory$) and in addition to common properties, *date created* (*order* is inherited from $ITimeLimited$) and *terms and conditions* ($IOrderTermsAndConditions$), the *order* archetype has *attributes* ($IAttribute$) property. With *attributes* property different customer requirements (or even domain) specific features, like sales channel (shop, Internet, etc.) or discount context (e.g. $IRuleContect$ describing information in order to use discounts), can be modelled.

The *order line* ($IOrderLine$) archetype (Figure 3-22) represents a part of an order that is a summary of particular goods or services ordered by a buyer [14 p. 310]. The *order line* archetype has *product type*, *product instance*, *amount* of ordered items, unit *price* of an item, *expected* and *actual delivery* dates, and other properties. By using an order line *receiver* property, products in each order line can be delivered separately.

Additionally, an *order line* has responsibility to manage *charges* and *taxes* related to the particular order line. A *charge line* represents an additional charge (packaging, transporting, etc.) for an *order line* [14 p. 319]. The *charge line* is described by *amount* of money, *comment* and *description* attributes. The description indicates what the additional charge is for (e.g. packaging, handling, shipping, and etc.) and the comment is just for recording additional information.

The *tax on line* ($ITaxOnLine$) archetype represents a tax charged on an *order line* ($IOrderLine$) or on a *charge line* ($IChargeLine$) [14 p. 320]. The *tax on line* ($ITaxOnLine$) archetype, by pointing to the *tax policy* ($ITaxPolicyIdentifier$) archetype, records what *taxation type* has been applied and what *taxation rate* is used. The *order manager* ($IOrderManager$) archetype is responsible for managing prevailing *tax policies*.

The *order manager archetype* manages a collection of orders ($IOrder$), tax policies ($ITaxPolicy$) and discount types ($IDiscountType$). The order manager $(IOrderManager)$ is designed as repository [6 pp. 322-327]. This means, that the order manager archetype appears as an in-memory collection of domain objects ($IOrder, ITaxPolicy\ and\ IDiscountType$), although objects can be physically stored in a database or in some other storage. The order manager manages all amendments in these collections. With CRUD (create, read, update and delete) operations it is possible to add new, to search, to change and to delete existing entities. As all of our archetypes are designed as read only software artefact then changes can only be made by sending a clear and explicit request to repository for repository to perform these needed changes. All CRUD requests, with explicit requester's identifier, must be sent to authorized repository which records request, checks requesters' privileges, and only if requester is allowed to make such a request, it completes the request. With such logic we can audit trail all requests and changes, and if needed, we are also able to restore the previous situation with built in undo and redo features of repository.

A party can be in different roles (*vendor*, *sales agent*, *payment receiver*, *order initiator*, and etc.) related to ordering of goods and services. In our

understanding these party roles, related to orders, are more requirements than domain related features. Therefore we propose collection based solution instead of modelling these roles explicitly (as done by Arlow and Neustadt, Figure 9.5 in [14 p. 316]). Such a solution allows us to meet different requirements from customers even at runtime.

Order *events* ($IOrderEvent$) and order *status* ($IOrderStatus$) (Figure 3-23) are used for an order lifecycle management. The order *lifecycle* is driven by certain notable occurrences or *order events*. An order event can be *authorized* by one or more parties through party signatures. The *date authorized* property records a date and a time when all required authorizations are obtained and the *date processed* property records a date and a time when this event is fully processed.

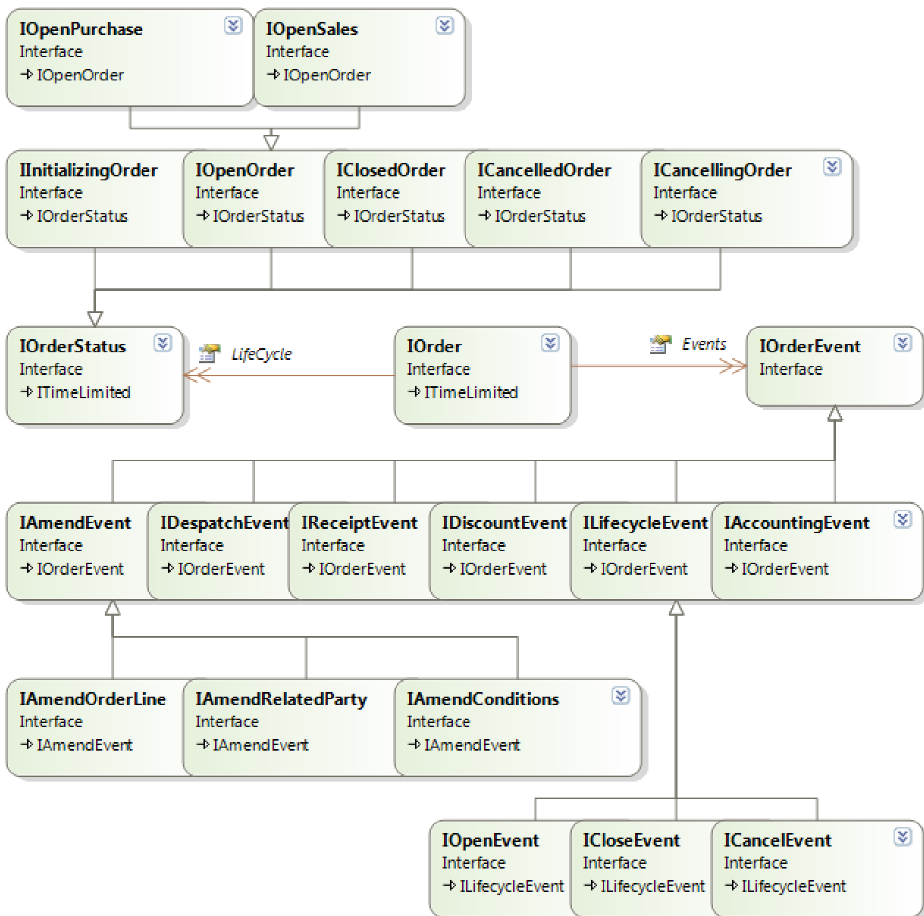

Figure 3-23: Order Status and Events

The *order status* ($IOrderStatus$) archetype represents a particular order state and contains possible activities that can be performed with an order in current

state [14 p. 326]. Differently from Arlow and Neustadt, where the order status is just an attribute holding enumerations (*initializing*, *open*, *closed* and *cancelled*), we have modelled order status by using *state pattern* from Nilsson [12 pp. 57-60].

With the state pattern, we encapsulate different states as individual concrete classes ($IInitializingOrder$, $IOpenOrder$, $IClosedOrder$, $ICancelledOrder$ and $ICancellingOrder$) inherited from an abstract base class ($IOrderStatus$). Such swapping of a single field into a bunch of separate classes results in moving behaviour methods to where they belong and satisfies the Single Responsibility Principle [12 p. 60].

More precise description of order status and order events is given in Appendix 7.1. In the order archetype pattern, only results of sales and purchases processes will be recorder. This means, that the order archetype pattern acts as documentation for sales and order processes. Payment (Appendix 7.2.3), purchases (Appendix 7.2.4) and sales (Appendix 7.2.5) processes are business processes described in the next section.

3.5.9 Process Archetype Pattern

Business archetype patterns (Party, Party Relationship, Product, Inventory, Order, Rule), designed by Arlow and Neustadt [14], are in good harmony with Zachman Framework (Section 2.1.1) and can be used for describing independent business phenomena (ZF columns). Arlow and Neustadt have archetypes for recording things (product archetype pattern), locations (party relationship archetype pattern), persons (party archetype pattern), events (inventory and order archetype patterns) and strategies (rule archetype pattern). Arlow and Neustadt have no archetype pattern for recording processes. This is the reason why we have designed the process archetype pattern (Figure 3-25).

It is important to note, that we are talking only about business processes. This means, that there is (Figure 3-25) at least some *outcome* ($IOutcome$) somehow reflected in company's accounts (information recorded either by the order or the inventory archetype patterns). This also means, that there is a subordinate party role ($IRelationship.Provider$) responsible to reporting to some supervisor party role ($IRelationship.Consumer$).

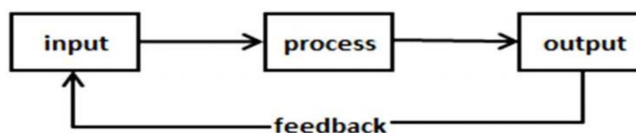

Figure 3-24: Process and Feedback

Thus, the metaphor of our business process model is a subordinate's report (or feedback, Figure 3-24) to a supervisor. Therefore processes in our model are described by communications between two parties (persons, organizations, or even artificial agents) "playing" some roles. In our understanding this metaphor

is quite powerful for modelling of different kinds of business processes and even business plans.

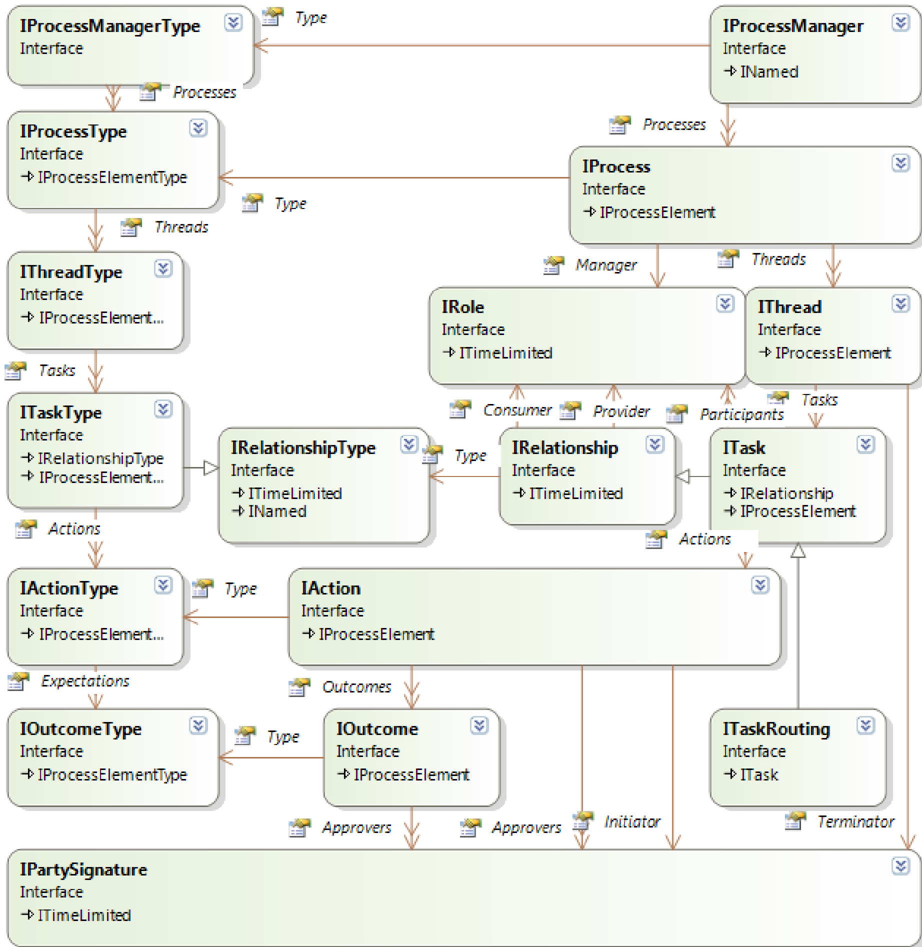

Figure 3-25: Business Process Archetype Pattern

Similarly to movies, that emulate dynamic reality by sequences of static pictures, the business process archetype pattern emulates dynamics of business processes by sequences of reports (feedbacks). More reports from trusted and different parties means better and more implicit picture about the whole process as a dynamic phenomenon. The business process archetype pattern can also be used for planning of business processes. When an actual report is a feedback about what has already happened, then a plan is a "feedback" about business processes we hope will occur in future. By comparing plans, modelled as expected future reports, and actual reports we can monitor the compliance of plans and reality and correct the plans if needed.

Therefore we have modelled business processes as communications (reports, feedbacks). Arlow and Neustadt have designed the CRM (*Customer Relationship Management*) archetype pattern [14 pp. 187-201] on top of the *party relationship archetype pattern*.

We have designed the business process archetype pattern similarly - on top of the party relationship archetype pattern. This means, that each task ($ITask$, part of business process, Figure 3-25) is a party relationship ($IRelationship$, see Section 3.5.5) which binds together consumer and provider roles.

Other *process archetype pattern* (Figure 3-25) archetypes are the following. The *process manager* archetype ($IProcessManager$) records all possible *processes* ($IProcess$) of allowed *process types* ($IProcessType$) described by the *process manager type* ($IProcessManagerType$). An example of a process manager type is *sales manager*. Examples of *process types* are different sales types (debited sales, invoiced sales, credited sales and prepaid sales).

Each business process consists of one or more *business threads* ($IThread$). A business thread is described by a *thread type* ($IThreadType$). Allowed thread types are listed in a *business process type*. Examples of *thread types* in a *sales business process* are sales initialization, receiving of payments, despatch of deliveries, change of sales conditions and cancellation of sales. Each business process (Figure 3-25) has a manager ($IProcess.Manager$), which is a party role ($IRole$).

One and the same thread (e.g. receiving of payments) can include more than one task (e.g. receiving a payment). This is why we need threads (e.g. all payments) and tasks (e.g. particular payment) in our model. Therefore each business thread (Figure 3-25) consists of one or more *business tasks* ($ITask$). A business task has *task type* ($ITaskType$) property. Allowed thread task types are listed in thread types. A task has a task manager ($(ITask\ as\ IRelationship).Provider$) and participants ($ITask.Participants$) properties. A *task manager* is responsible for reporting to a senior manager ($(ITask\ as\ IRelationship).Consumer$). Similarly to threads that consist of one or more tasks, tasks consist of one or more actions ($IAction$). Each *action* has an action type ($IActionType$) and one or more outcome ($IOutcome$) properties. Each outcome has an outcome type ($IOutcomeType$).

Business processes often require some kind of approvals. Process threads can be terminated ($IThread.Terminator$), actions can be initiated ($IAction.Initiator$) and approved ($IAction.Approvers$), as well as outcomes can be approved ($IOutcome.Approvers$) by authorized persons. The party signature archetype ($IPartySignature$) is used for such approvals.

As business processes vary and can be changed often, we have designed the business process archetype pattern to be managed by rules (Figure 3-26). By using the rule archetype pattern's (Section 3.5.3) rule set ($IRuleSet$) and rule context ($IRuleContect$) archetypes, we can formally describe and validate wide variety of business requirements used by business processes. This means, that

each process element type ($IProcessType$, $IThreadType$, $ITaskType$, $IActionType$ and $IOutcomeType$ - inherited from $IProcessElementType$) has $RuleSet$ property, and each process element ($IProcess$, $IThread$, $ITask$, $IAction$ and $IOutcome$ - inherited from $IProcessElement$) has $RuleContext$ property.

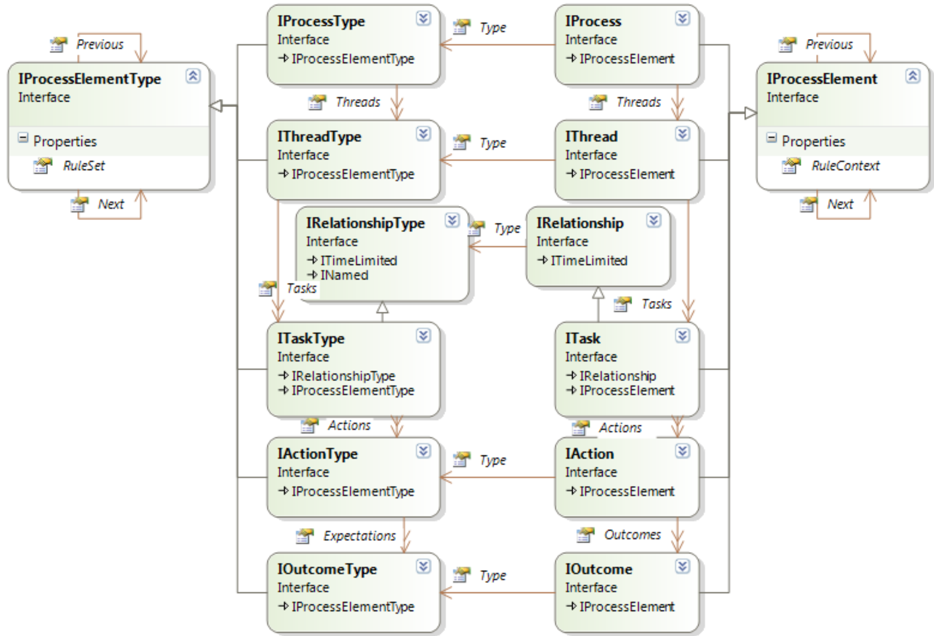

Figure 3-26: Rule Based Process Management

For example, to establish that the buyer has rights to withdraw within 14 days from purchases transaction, the *receive purchase cancellation task type* must have the following simple rule

$$R = \{CancellationDate, InitDate, DIFF_IN_DAYS, AllowedDays, LESS\}$$

Depending on *cancellation* and *initiation* dates, a seller either accepts or declines the sales cancellation. For example, when a seller receives the cancellation request which gives context $C = \{28.12.2010, 20.12.2010, 14\}$, the buyer has rights to withdraw.

Some concrete common business processes (communication, reporting, payment, purchase and sales) are described in Appendix 7.2.

3.6 Using of Models

A domain stakeholder [25] is a person or an organization united somehow in interest or dependency on the domain. Each stakeholder has some roles, rights, duties as well as specifically identified perspective or view on a domain.

By analysing stakeholders' views, rights and duties, we get knowledge about the domain.

Lindqvist and Christensen [45] have generalized stakeholders to a global administrator, a local administrator, a person and a third party. We use the same classification along with the assumption, that a stakeholder is a role that persons and/or organizations are playing.

Figure 3-27 illustrates parties' roles in clinical laboratory. Stakeholders' relationships in clinical laboratory are illustrated in Figure 3-28.

A *person* is an individual capable of sample analysis in a laboratory. Persons are laboratory employees. A role that these persons, employed by a laboratory, are playing, in the laboratory, is the role of a MTA (Medical Technical Assistant). Some MTA's can also be in the role of a local administrator.

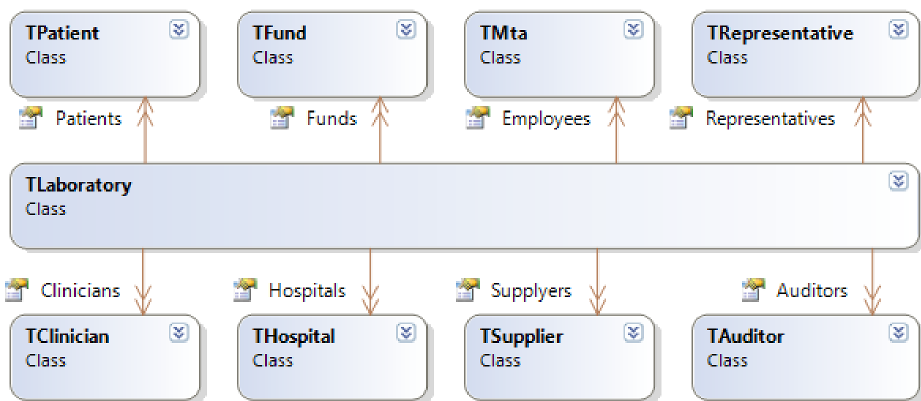

Figure 3-27: Abstraction of Clinical Laboratory Related Party Roles

A *local administrator* is the administrator of a particular laboratory. A *local administrator* takes care of maintaining infrastructure in a laboratory.

A *global administrator* is an administrator who tracks and uniquely identifies organizations and individuals. Examples of people's identifiers are passport numbers, social security numbers and identity card numbers. Examples of companies' identifiers are domain names, stock exchange symbols, registered names and office addresses.

A *third party*, in a laboratory, is a person or an organization, which causes MTA to analyse samples or is somehow affected by sample analysis process in the laboratory. A third party does not interact directly with the sample analysis process.

Third parties in a laboratory are:

Laboratory – a sample analysing company. A MTA works for a laboratory.
Patient – a person whose sample (blade, serum, urine, etc.) is being analysed.
Clinician – a person who treats patients. A clinician requests (initiates) a sample analysis in a clinical laboratory and clinicians receive analysis reports from a laboratory MTA.

Hospital – a company where clinicians are working and patients can be hospitalized. A laboratory can be a department of a hospital.
Auditor – a company (institution) which for example audits QC in a clinical laboratory. A laboratory has to send periodical reports about laboratory QC to auditors.
Fund – a company or a fund offering health insurance for patients.
Supplier – a company which maintains laboratory equipment and sells reagents and spare parts to laboratories.
Representative – a person who represents a company (hospital, auditor, fund or supplier) or a person (patient, clinician).

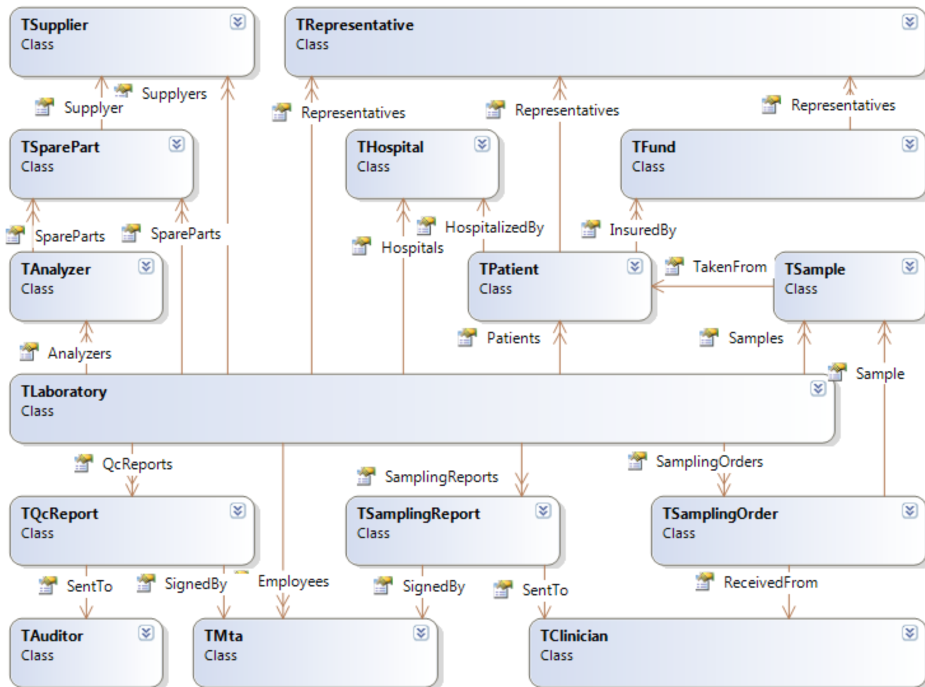

Figure 3-28: Laboratory Stakeholders Relationships Abstraction

The following narratives (L.3) illustrate clinical laboratory stakeholders (roles related to a laboratory) and their relationships.

L.3. Organizations and persons are parties in a clinical laboratory
- L.3.1. All parties are uniquely identified (they have some registered identifiers like passport number or VAT registration number) by a global administrator (some government or other legal organization).
- L.3.2. Each party can play one or more roles in a laboratory.
  - L.3.2.1. Roles that persons can play in a laboratory are: MTA, patient, physician, and representative of an organization or other person.
  - L.3.2.2. Roles that organizations can play are: laboratory, hospital, auditor, supplier, and fund.

L.3.2.3. Each company's role (laboratory, hospital, auditor, supplier, or found) has zero or more representatives (persons representing a company).

L.3.3. Party relationship is a relationship between two parties, where each party plays a specific role.

L.3.3.1. Laboratory employs one or more MTAs.

L.3.3.1.1. Responsibilities are assigned to MTAs. Meaning that persons who play this role have to have some capabilities (education, experience, certificates, etc.).

L.3.3.1.2. Some MTA's are local laboratory managers.

L.3.3.2. A clinician can treat one or more patients.

L.3.3.2.1. More than one clinician can treat one and the same patient at one and the same time.

L.3.3.2.2. A person who, is a clinician, has capabilities (education, experience, certificates, etc.). Clinician's role has requirements, which can give a person (playing this role) some specific rights (responsibilities) (e.g. to treat children, to treat some special disease, etc.).

L.3.3.2.3. A clinician can order sample analysis only for patients he/she treats.

L.3.3.3. A laboratory tests (determines) samples.

L.3.3.3.1. It is possible that a laboratory has to collect samples.

L.3.3.3.2. Sample determination (can include validation, decision making, etc.) can require some specific information (e.g. dietary, age, gender, active medicaments, putative/actual diagnosis, etc.) about patients.

L.3.3.4. Patients can be either hospitalized or not.

L.3.3.4.1. Only one hospital can hospitalize one and the same patient at the same time.

L.3.3.4.2. There are a fixed number of beds in a hospital.

L.3.3.4.3. A hospitalized patient is in one bed or moves from one bed to another.

L.3.3.5. A clinician is either a sole proprietor (for example some GP's - general practitioners), or has to be employed by at least one health care company.

L.3.3.5.1. When ordering sample analysis, a clinician has to assign only one health care company to the sampling request.

L.3.3.6. Patient's health can be insured by one or more health insurance funds.

L.3.3.7. A laboratory delivers sampling invoices (depending on rules or agreements) either to hospitals, funds, clinicians or patients.

L.3.3.7.1. If an invoice is sent, then only one payer has to be marked.

L.3.3.8. One or more auditors can audit different activities in a laboratory.

L.3.3.8.1. Most common audit in laboratories is QC audit.

L.3.3.8.1.1. QC reports have to be sent periodically to auditors.

L.3.3.8.1.2. QC report has to be signed by one or more MTAs

L.3.3.9. One or more suppliers can be maintaining laboratory equipment and/or supplying spare parts and other supplies needed for sample analysis.

According to laboratory stakeholders' skeleton (Figure 3-28) and the party role and the party relationship archetypes (Figure 3-29), we need:

1. Specific role types (*patient, MTA, clinician, and representative*) that persons can play;
2. Specific role types (*laboratory, hospital, auditor, supplier and fund*) that organizations can play;

3. Specific relationship types (*laboratory MTA, laboratory manager, hospital clinician, hospital patient, laboratory patient, clinician patient, etc.*).

For complete explanations of the party and the party relationship archetype patterns please see Sections 3.5.4 and 3.5.5. The concise logical model of party relationships is illustrated in Figure 3-29.

There are two possibilities to model laboratory related role and relationship types (domain models in general). We call these "runtime" and "design time" techniques. The "runtime" technique is described in Section 2.3.

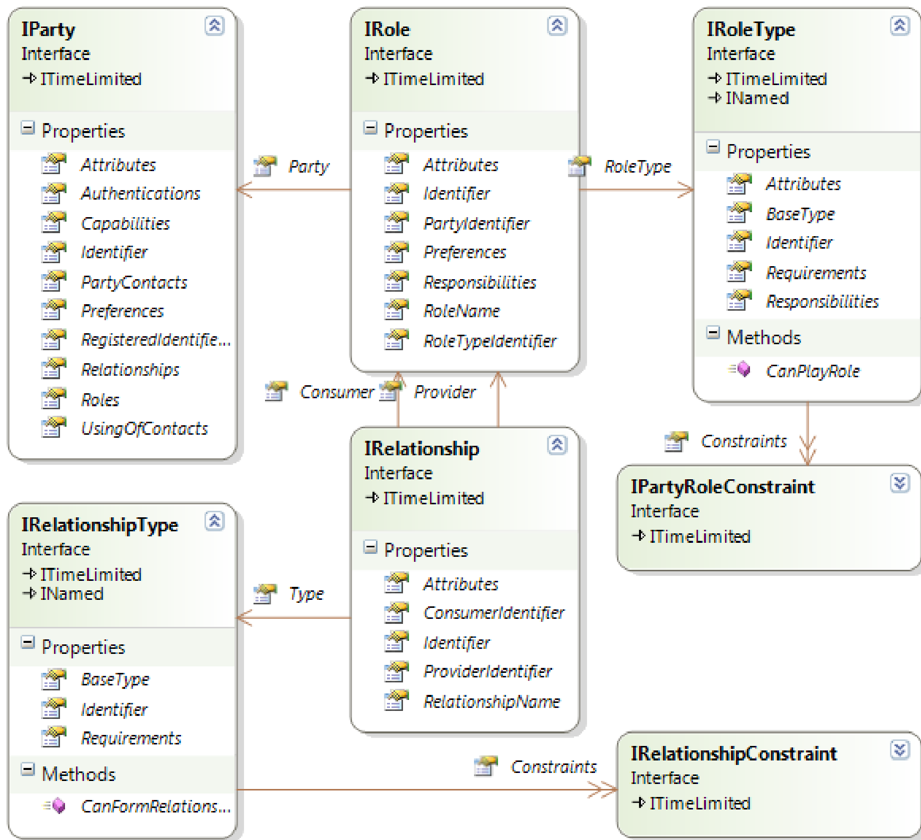

Figure 3-29: Concise Logical model of Role and Relationship

We believe, based on our current knowledge, that this "runtime" technique allows us to change domain models even at runtime. This means, that laboratory related (or more generally domain related) role and relationship types are not special classes or interfaces, but are singleton unique values in a domain. This possible "runtime" model of laboratory related role and relationship types is illustrated in the following pseudo code.

```
// Role types the person can play
IRoleType Mta = RoleType.Register ("MTA", typeof(IPerson), ...);
IRoleType Clinician = RoleType.Register ("Clinician", typeof(IPerson), ...);
IRoleType Patient = RoleType.Register ("Patient", typeof(IPerson), ...);
IRoleType Representative =
     RoleType.Register ("Representative", typeof(IPerson), ...);

// Role types the organizations can play
IRoleType Laboratory =
     RoleType.Register ("Laboratory", typeof(IOrganization), ...);
IRoleType Hospital =
     RoleType.Register ("Hospital", typeof(IOrganization), ...);
IRoleType Fund = RoleType.Register ("Fund", typeof(IOrganization), ...);
IRoleType Auditor
      = RoleType.Register ("Auditor", typeof(IOrganization), ...);
IRoleType Supplyer
      = RoleType.Register ("Supplyer", typeof(IOrganization), ...);

// Relationship types
IRelationshipType LaboratoryMta
      = RelationshipType.Register (... Laboratory, Mta, ...);
IRelationshipType LaboratoryManager
      = RelationshipType.Register (... Laboratory, Mta, ... );
IRelationshipType OrderingClinician
      = RelationshipType.Register (... Laboratory, Clinician, ...);
IRelationshipType HospitalClinician
      = RelationshipType.Register (... Hospital, Clinician, ...);
...
```

First, in the code above, role types that only persons (*typeof*(*IPerson*)) can play are specified. Next, role types only organizations (*typeof*(*IOrganization*)) can play are specified. Finally relationship types, which can be formed between persons playing roles of particular types, are specified.

The technique, shown in the listing above, is in essence similar to normal OO (object oriented) modelling technique ("design time" modelling of classes and interfaces) shown in Figure 3-30.

The main difference between these techniques is not what (laboratory domain in both cases) we model, but how we model – either by using "design time" class/interface technique or "runtime" singleton instances technique. Even the *inheritance* is supported by the "runtime" model to some extent using *BaseType* attributes (Figure 3-29).

However, in domain analysis, we prefer (at least currently) the normal OO ("design time") technique and we normally use the "runtime" modelling technique for specification of customer requirements. The "design time" technique allows us to specify domain terms (clinician, patient, etc.) that we can use as DSL when specifying customer requirements.

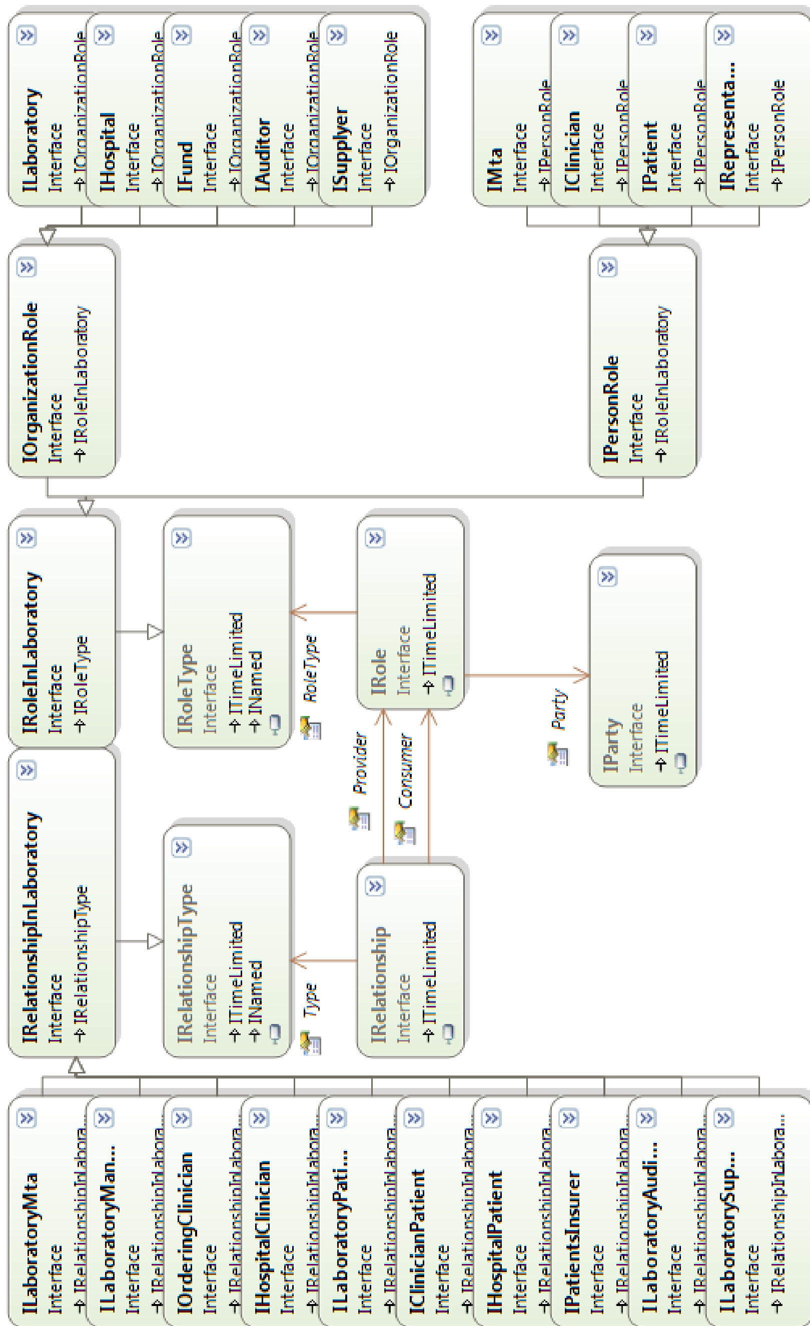

Figure 3-30: Laboratory Role and Relationship Types

3.7 Summary

According to Arlow and Neustadt [14], a *business archetype* is a primordial thing that occurs consistently and universally in business domains and in business software systems. A *business archetype pattern* is a collaboration of business archetypes. We modified and complemented these archetypes and archetype patterns originally designed by Arlow and Neustadt. As common for software factories [3], all our models are source artefacts. However, we distinguish the idea of archetype and archetype patterns (A&AP) from the implementation (A&API). When we designed the A&AP and implemented the A&API, we used the following principles.

Separation of knowledge and operational levels is one of our A&APs design principles. For instance, there are no concrete units (kilogram, metre, second, etc.) in our quantity archetype pattern. This is because we strongly separate knowledge (abstraction - unit as general concept) and operational (concept concretisations – kilogram, metre, second, etc.) levels from the archetype patterns as suggested by Fowler [15 pp. 8, 24-27]. Such concretizations (kilogram, metre, second, etc.) are neither issues of archetypes and archetype patterns nor domains but are issues of business requirements for some particular software. Such a separation technique gives us flexibility to change and modify software according to requirements. The goal is to provide modifications of requirements even at runtime in order to add evolutionary properties to software systems.

Bjørner's real world modelling principle is the next design principle we use. This is an opposite of the „*stop trying to model the real world*“ [46] principle. According to Bjørner's real world principle, we have to analyse *„what already exists"* and describe *„the world as it is"* [22 p. 18]. This is why there is the *measure* archetype in the quantity archetype pattern we designed, and why instead of Arlow and Neustadt's [14] *derived units* (e.g. $\frac{km}{h}$) we have the *derived measure* (e.g. $Speed = \frac{Distance}{Time}$) archetype.

Using of constructions similar to RDF (Resource Description Framework) *triplets* is the next principle we use. This means, that most of our archetypes (see Section 3.5) have *attributes* property holding a collection of {category, predicate, object, authorized by, authorized when} records which are used similarly to RDF {subject, predicate, object} triplets.

Good object-oriented design principles. As our archetypes and archetype patterns are not just documentation artefacts, but are source artefacts, we use principles of good object-oriented design [47]. For example, we use the *Single Responsibility Principle (SRP)* by designing archetypes and archetype patterns so that they are responsible only for domain knowledge but are not responsible for infrastructure, presentation and data access. This means, that A&APs are designed and A&API is implemented to be infrastructure ignorant (similarly to persistence ignorance [12 pp. 183-184]).

We also use the *Open-Closed Principle* (closed for modifications, open for extensions) [47] by using interfaces and common classes as shown in the following example of the *PartyName* archetype.

```
public interface IPersonName: ICommonName {
    string Prefixes{ get; }
    string GivenName{ get; }
    string MiddleNames{ get; }
    string FamilyName{ get; }
    string PreferredName{ get; }
    string Suffixes{ get; }
}
public abstract class CustomPersonName:
   CustomCommonName, IPersonName {...}
public sealed class PersonName: CustomPersonName {...}
public sealed class UndefinedPersonName: CustomPersonName {...}
```

We define logical models (Row 3 of ZF, Table 2-2), according to semantic (Row 2 of ZF, Table 2-2) models specified as unit tests (described in Section 2.2), by using interfaces. We then use open for extensions common archetypes (abstract classes) where we implement all needed functionality. Finally, we have closed for modification archetypes (sealed classes). We also use closed for modifications special cases (*UndefinedPersonName* for instance) as suggested by Fowler [6].

Liskov Substitution Principle (LSP) is also used when designing archetypes and archetype patterns. For example, we never inherit *student* (as described in [12]) or *patient* from *person*, because no such kinds of persons as patients or students exist in real world. In designing of archetypes and archetype patterns we strongly separate *parties* (John Smith for instance) from *roles* (being a student or a patient) parties are involved with.

We use *the principle of comprehensive test* when implementing A&API. This means, that all our models are implemented according to the test driven modelling techniques (Section 3.2) where we utilize the test driven development [24] techniques for domain analysis and modelling.

4 CASE STUDY: CLINICAL LABORATORY SOFTWARE

We use ABD (Part 2) in development of real life LIMS software and LIMS SF (Software Factory) in CBPG (Clinical and Biomedical Proteomics Group, Cancer Research UK Clinical Centre, Leeds Institute of Molecular Medicine, St. James University Hospital) at University of Leeds under the project called MyLIS.

LIMS represents a class of computer systems designed to manage laboratory information [1]. MyLIS is sample management software designed for clinical research laboratories and intended to satisfy such important criteria of modern information systems as interoperability [38] and dependability [48]. A wider research goal is to develop LIMS that evolves in an evolutionary way together with business processes.

Figure 1-1 illustrates our research and developments towards Software Factory for LIMS. Based on business archetypes and archetype patterns based domain model of laboratory, the LIMS Software Factory architecture consists of LIMS Domain Specific Language (DSL), LIMS Engine and Tests Engine.

The same methodology we described in Part 2 for development of archetypes and archetype patterns is also used for development of specific (e.g. laboratory) domains (Table 2-2).

1. A glossary and semantic models are both specified as unit tests;
2. Logical design of domain models is specified in terms of A&APs and the implementation of domain models is realized so that all unit tests (semantic models, both for domains and A&APs) hold;
3. We use archetypes and archetype patterns based models (Section 3.5), implemented as DLL (as embedded DSL), to get a specific domain model (e.g. laboratory);
4. Specific domain models (e.g. laboratory, implemented as DLL) are used as embedded DSL when specifying requirements.

We started with LIMS developments, under the code name MyLIS, in September 2008. The prototypic MyLIS has been used in CBPG laboratory from the end of 2009 and is currently in its third version, used by three different CBPG research groups with different requirements. Although in CBPG MyLIS is used in everyday laboratory routine, it acts also as a test polygon where we test, evaluate and verify A&APs based techniques and LIMS SF ideas.

MyLIS development is agile in a sense that we stared with a very simple laboratory domain model based on a very simple A&AP model. The version of A&AP model, we explained in Part 3, is more mature than the model we use in the current working version of MyLIS. Models, used in the working MyLIS version, are similar to the initial party model described in Section 3.2.4 and visualized in Figure 3-5. This means, there are no interfaces based logical models (Section 3.5), no influences from Fowler [15], Hay [16] and Silverston's [17] models and no custom-class-undefined patterns illustrated in Figure 3-11.

The foregoing is also true for the laboratory domain model used in the current MyLIS version. Therefore, laboratory domain models, described in Section 4.2, are currently under development.

In Section 4.1, we describe motivation and strategies used in MyLIS developments. In Section 3.6, based on laboratory related party roles, we described how archetypes and archetype patterns are used in domain development. In Section 4.2, we describe domain models of laboratory. In Section 4.3, we are talking about MyLIS developments where ABD techniques and laboratory domain models are utilized. In Section 4.4, we describe our research and developments towards LIMS SF and evolutionary information systems.

We have published three conference papers [42; 49; 50] related to MyLIS research and developments.

4.1 Motivation for LIMS and LIMS SF developments

LIMS [1; 51] is a complicated software system. Medical laboratories differ from other laboratories in the sense that medical laboratory data are classified as sensitive patient data and therefore these are subject to data protection laws. However, research laboratories differ from other laboratories in the sense that business processes used by research laboratories are constantly changing and different research groups within the same research laboratory, sometimes even different investigators in one and the same research group, require different and customizable business processes. At the same time, research laboratories and researchers require exchange of information and interoperability of software systems in a global manner.

The 2020 Science Group (Venice, July 2005) [52], a group of internationally distinguished scientists, considering the future of science and the role, also impact, of computing and computer science on sciences, including revolutionizing medicine and healthcare, highlighted that *"...end-to-end scientific data management, from data acquisition and data integration, to data treatment, provenance and persistence*" is one of the immediate and important science challenges for the year 2020. They also indicated that "*a first step in that direction is peer-to-peer and service-oriented architectures*" and that "*the development of an infrastructure for scientific data management is therefore essential*".

It would be good if, for example, cancer researchers were able to share and distribute their data and knowledge with other cancer researchers around the world automatically and conveniently. It would be also good if cancer researchers could use in their research clinical data about patients around the world.

Such a world-wide, peer-to-peer and service oriented infrastructure for scientific data management requires that data and knowledge are semantically understandable for machines. It is also important that patients' clinical data and patients' sensitive personal data are strongly separated.

While standardized in some ways, such system for scientists has to be flexible and adaptable so, that there are customizable possibilities to describe data, knowledge and research methods. This system has to be tied with mathematical methods and have flexible data processing features. It also needs advanced authorizing and security features.

We see ABD for development of domains, requirements and software as a promising idea to build such an infrastructure for medical research (including cancer research) scientist. The kernel of techniques in question is a universal, well-designed, semantic model (archetypes and archetype patterns describing objects, subjects, processes, locations, events and rules) used for describing domain models and requirements.

Table 4-1: Meta-modelling with archetypes and archetype patterns

Layer	Content	Changes	Tool
Meta-meta-model (M3)	Objects, properties, ... programming language	Rarely, design time	e.g. C# language
Meta-model (M2)	Archetypes and archetype patterns (e.g. business archetypes and archetype patterns)	Rarely, design time	Code
Model (M1)	Domain model (e.g. clinical laboratory domain model)	Sometimes, run-time and/or design time	Data (or code)
Reality (M0)	Requirements	Often, run-time	Data

We see possibilities to implement database layouts, communication protocols and graphical user interfaces on top of archetypes and archetype patterns based models (Section 3.5) and neither on top of laboratory domain models (Section 4.2) nor on top of particular requirements from a laboratory. We see possibilities to modify data descriptions and research methods even at runtime and distribute as well as compare them with others.

In our understanding this can be archived by meta-models (M2, Table 4-1), containing archetypes and archetype patterns, and by domain models (M1), designed on top of these meta-models. Meta-models as well as domain models should reflect the universe of discourse, contain only knowledge level and not operational level information and be designed infrastructure ignorant (Section 3.7).

This is why (only knowledge level information), for example, in the quantity archetype pattern, there are no concrete measures, units and unit converting factors. Therefore, there are no such types (classes in C#, design time artefacts) as Kilometre, Hour and Kilogram or even such types (classes in C#) as Distance, Time and Mass. These concrete measures, units and unit converting factors are coming from domains or from particular software requirements and they are data (objects in C#, run-time artefacts). This is why (infrastructure ignorance) we have included repositories into archetypes and archetype patterns

(quantities, inventory, party manager, orders manager) and designed UI by using reflection technology.

Some of these data (describing either domain or requirement knowledge) are special data - they are "singletons". Not singletons [44], but "singletons". A singleton ensures a type (class) which only has one instance and provides a global point of access to it. The "singleton" Kilogram, for example, ensures globally one and the same semantic meaning. We realized "singletons" similarly to the *singleton registry* pattern by Fowler [6 p. 483].

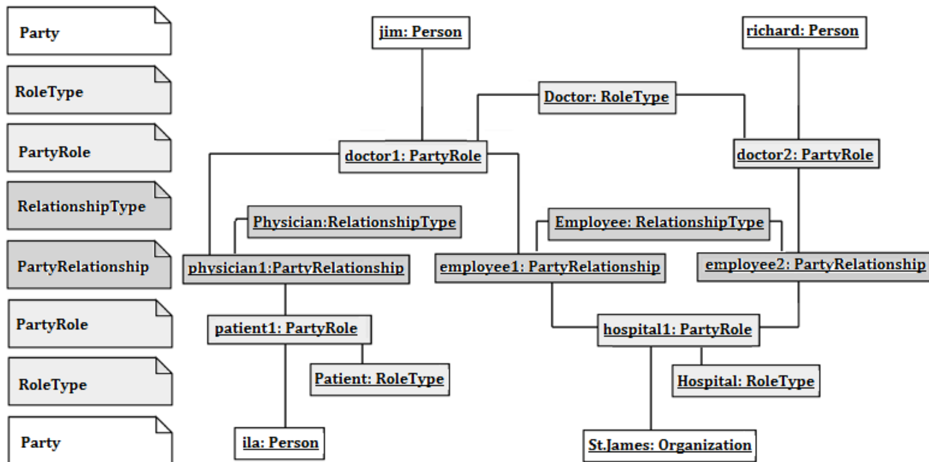

Figure 4-1: Example of Doctors and Patients in Hospital

All other archetype patterns are designed similarly to quantity archetype pattern. For example, the party archetype pattern (Figure 3-29) has types *RoleType* and *RelationType*. The laboratory domain model (Figure 3-30) has role types *Fund*, *Patient*, *Hospital*, *Supplier*, *Laboratory*, *Physician*, *Auditor* also *MTA* (Medical Technical Assistant) and relationship types "patient is insured by a fund", "MTA works for a laboratory", "physician works for a hospital", "the patient has an attending physician", and etc. All this can be modelled and realized in software systems so that run-time changes to domain models and requirements are possible.

This, for example, means (Figure 4-1), that we have a meta-model (M2) with *Party*, *RoleType*, *PartyRole*, *RelationshipType*, and *PartyRelationship* types according to the party archetype pattern (Figure 3-29). These types (classes in C#) are realized in code. Next we have the laboratory domain model with terms like *Doctor*, *Patient* and *Hospital*. *Doctor*, *Patient* and *Hospital* are not subclasses of *RoleType* class, but are instances ("singletons", objects) of *RoleType* class. Similarly, *Physician* and *Employee* are "singleton" instances of the *RelationshipType* class. In Figure 4-1, two persons, Ila and Jim, are related so that in *PartyRelationship* indicated as *physician1* with *RelationshipType* indicated as *Physician*, Jim plays a *PartyRole*, indicated as

doctor1, with *RoleType*, indicated as *Doctor* and Ila plays a *PartyRole* indicated as *patient1* with *RoleType* indicated as *Patient*.

4.2 Clinical Laboratory Domain Model

In the following we develop the domain model for clinical laboratories. This laboratory DM is based on the ASTM standard laboratory guidelines [1; 51]. We use the A&AP model (Section 3.5, realized as DLL) as a DSL similarly to laboratory stakeholders' case explained in Section 3.6.

In laboratory domain analysis and laboratory DM design we follow independent phenomena described by the columns of ZF (Table 2-1). This means that by asking common questions *what* (products), *how* (processes), *where* (locations), *who* (persons), *when* (events) and *why* (motivations), we analyse and model different clinical laboratory domain facets.

We found, that there are no laboratory specific aspects for events (*when*), modelled using the inventory (Section 3.5.7) and the order (Section 3.5.8) archetype patterns. The same is also true regard to motivations (*why*), modelled using the rule (Section 3.5.3) archetype pattern. In Section 3.6 we designed laboratory stakeholders we used for modelling locations (*where*, the structure of laboratory) and persons (*who*, generally parties). Therefore, in current section we have to model only laboratory products (Section 4.2.1) and processes (Section 4.2.2).

4.2.1 Products and Services in Laboratory

Products and services are phenomena and concepts of domain, which are fundamental to all other domain facet (or domain phenomena). The clinical laboratory domain is a domain of sample management. Main products in clinical laboratory are *analyser*, *sample*, *tube*, and *rack*. Main service in a clinical laboratory is sample *determination* (testing).

A *sample* is a small part of a material or a product intended to be a representative of the whole [51]. Each sample must be uniquely identified and the location of a sample in a laboratory, a sample login, distribution and final sample elimination and utilization has to be carefully tracked. Samples are normally kept in sample tubes. Figure 4-2 abstracts the life-cycle of samples in a laboratory. The first positive identification leads samples to registered state. Normally, each sample in a laboratory has to be accompanied by a sample order which determines tests ordered by ordering clinicians. When a sample and the accompanying order are both in a laboratory (in principle they can arrive at different times), the sample is in open state. This means, that the sample is ready for determinations. Open samples are normally located in distribution workstations. In distribution workstations samples will be distributed to analysis or storage workstations. In analysis workstations samples are analysed - results of some sample attributes are determined. In storage workstations samples are held for possible late determination (e.g. for possible redetermination or possible subsidence purposes before determination), stationary storage (e.g.

long-term storage in -80C refrigerators) or utilization. Before utilization all samples should be marked as eliminated from laboratory.

A *tube* or a sample tube is a cover of samples. It is possible to get different tube types with different sizes produced by different vendors. All such information (capacity, vendor, etc.) can be important in sample determination.

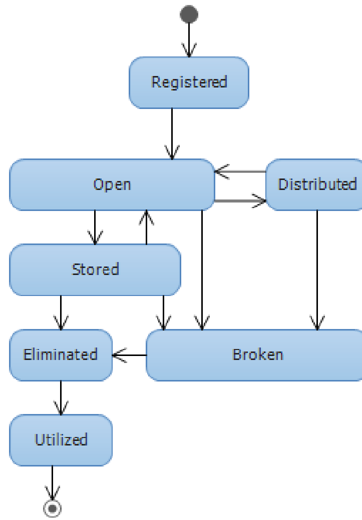

Figure 4-2: Life-cycle of Samples

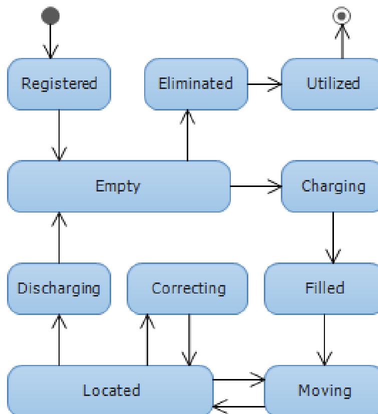

Figure 4-3: Life-cycle of Racks

A *rack* is a container of samples in tubes. Each rack is uniquely identified and the location of a rack as well as the content of a rack must be carefully tracked. Figure 4-3 abstracts the life-cycle of racks in a laboratory. Registered (uniquely identified and inventoried) rack is normally in empty state. Racks are charged with samples in distribution workstations. After that, filled racks are normally moved to some analysis workstations, where racks are located until all rack

samples are determined. Racks can also be moved to some storage workstations for stationary storage or to distribution workstations for correction or discharge. Racks can also be discharged in some workstations where samples are eliminated from a laboratory before utilization of samples. Obsolete or broken racks have to be marked as “eliminated” in the inventory list before utilization.

A *determination* is a single result of sample analysis [51]. Determination has properties: type (*test*) indicating what was determined, object (sample) determined, value of the determination, date when the sample was determined, by which analyser the sample was determined and one or more MTA signatures. A *test* (e.g. cholesterol, sugar, protein) is a type of determination. Tests may be grouped into *profiles* (e.g. complete blood picture or small blood picture).

An *analyser* is a laboratory tool which is used for sample determination. Analysers are located in laboratory workstations. A *workstation* is a place in a laboratory (the main organization unit in laboratory) where samples are distributed, analysed (determined) or stored for later use. Samples are located in workstations or move from one workstation to another. Products and their relationships in clinical laboratory are illustrated in Figure 4-4.

Figure 4-4: Abstraction of Laboratory Products

The logical model of laboratory domain products and product relationships is shown in Figure 4-5. The product archetype pattern is described in Section 3.5.6. There are *analyser* (*IAnalyserType*) and *sample* (*ISampleType*) product types and corresponding product instances (*IAnalyser* and *ISample*) in the laboratory domain model.

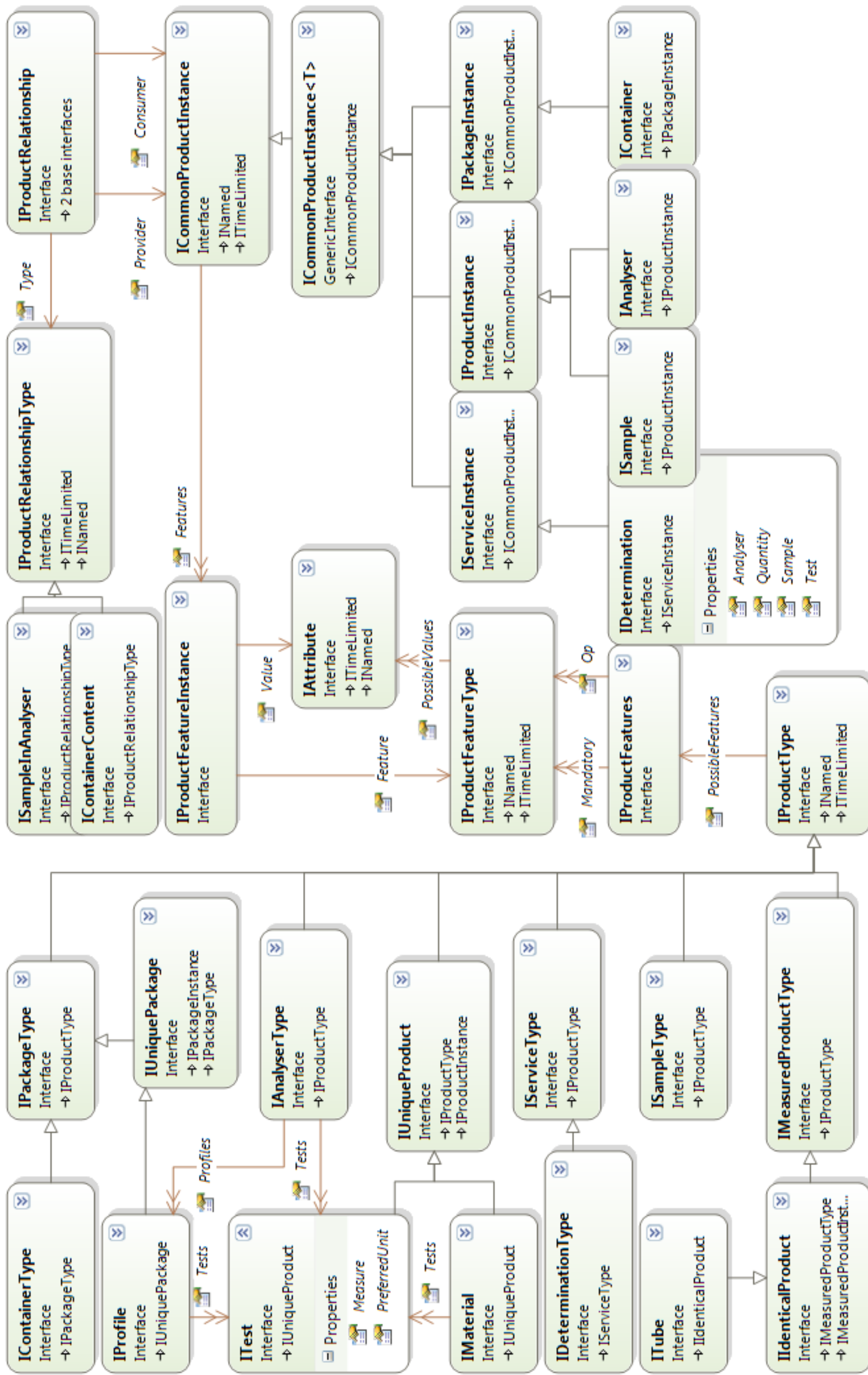

Figure 4-5: Laboratory Products and Product Relationships

There are package type ($IContainerType$) and corresponding package instance ($IContainer$) for modelling *containers* in the domain model of laboratory. There are service type ($IDeterminationType$) and corresponding service instance ($IDetermination$) for modelling *determinations* in the laboratory domain model. There are unique products for modelling *tests* ($ITest$) and *materials* ($IMaterial$), identical products for modelling *tubes* ($ITube$) and unique packages for modelling *profiles* ($IProfile$) in the clinical laboratory domain model.

In principle it is possible to describe all characteristics of products using *product feature* ($IProductFeatureInstance$) and *product feature type* ($IProductFeatureType$) archetypes. Shortcuts have been designed for these product features and feature types, as shown in Figure 4-5, to facilitate understanding. For instance, the *determination* ($IDedermination$) has “shortcuts” to *Analyser*, *Quantity*, *Sample* and *Test* features.

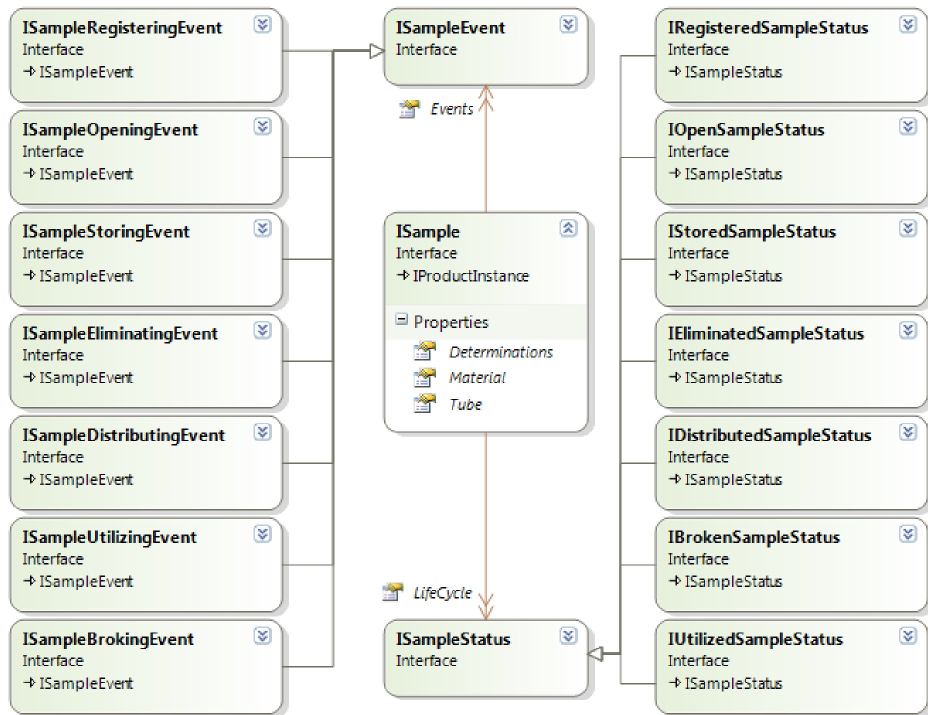

Figure 4-6: Sample Status and Events

Defining such shortcuts is basically the only reason why we do not have implicit classes for party roles (Section 3.6) and why we have implicit classes for product instances ($IDetermination$, $ISample$, $IAnalyzer$ and $IContainer$) in the laboratory domain model.

Implicitly defined relationships are relationships between containers and their elements ($IContainerContent$) and relationships between samples and analysers ($ISampleInAnalyser$).

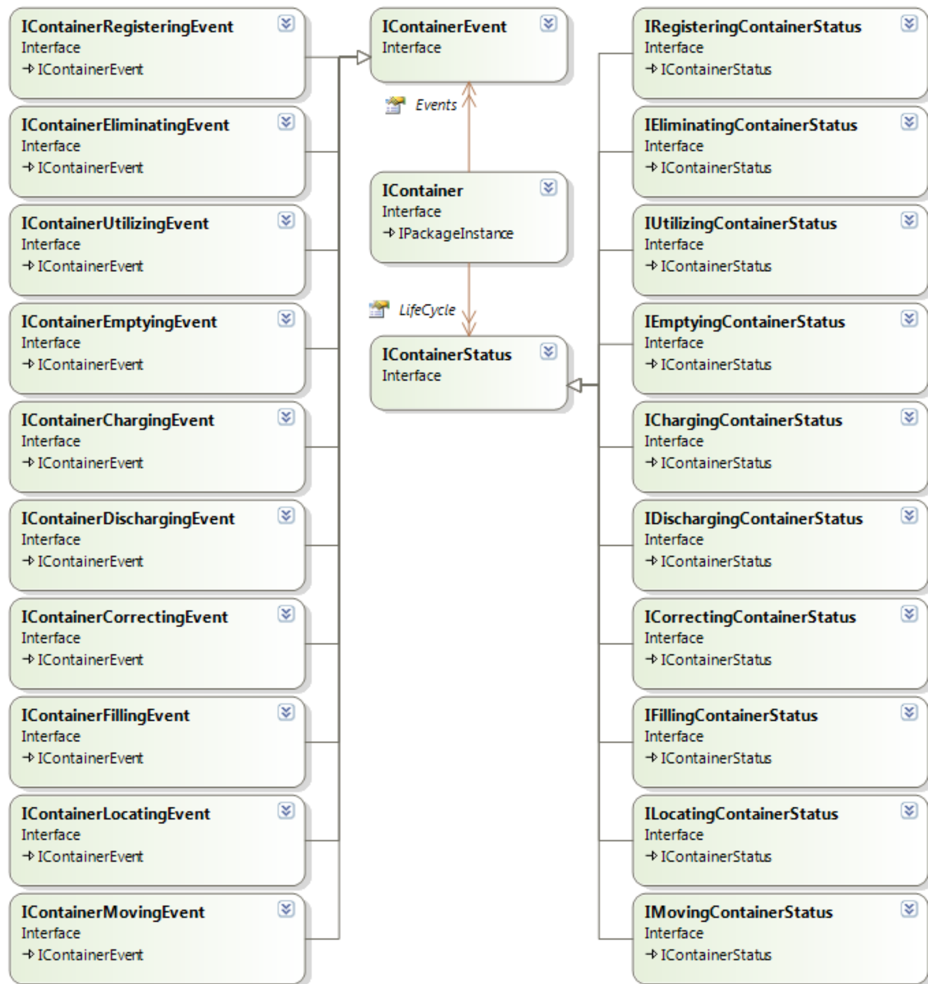

Figure 4-7: Container Status and Events

The question is, why only these and why there are no implicitly designed relationships between *samples* and *determinations* (or between *determinations* and *analysers*; or between *determinations* and *tests*). The answer is simple: if a *determination* is made, then it is made by a particular *analyser*, for a particular *sample* and what was determined was a particular *test*. These properties of a *determination* will never change or if they will, then this is just a correction of recording mistakes. At the same time, relationships *container-content* and *sample-analyser* are in continuous change and all these changes should be audit tracked.

Similarly to *order status* and *order events* (Figure 3-23), *container* and *sample* have *status* and *event* properties for managing their lifecycles as shown in Figure 4-6 and Figure 4-7.

4.2.2 Laboratory Business Processes

4.2.2.1 Sample Determination

A laboratory is an organization managing samples in order to analyse these samples (determination of sample properties). It has to be mentioned that more than one laboratory is possible in the same physical location and these laboratories can share laboratory equipment as well as employees.

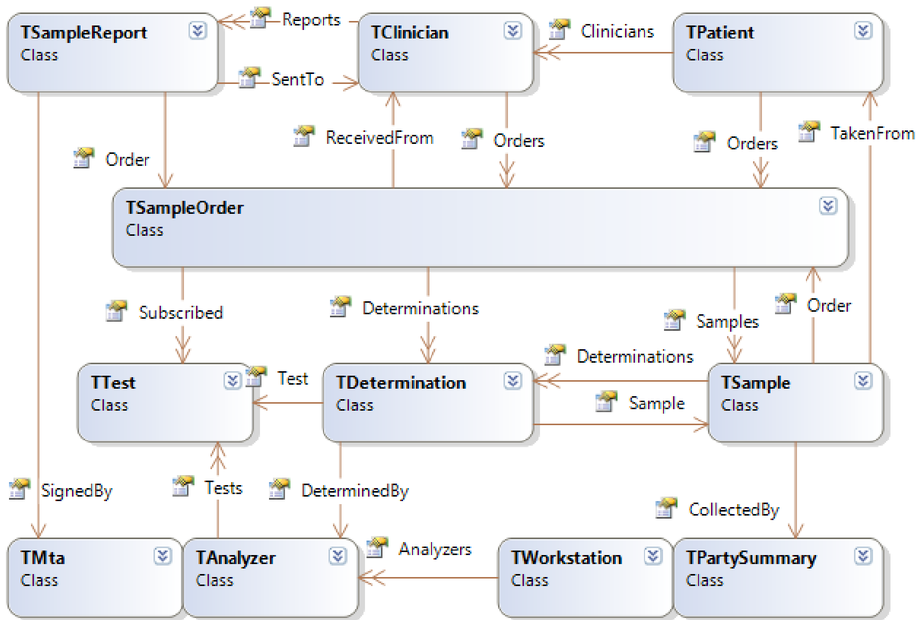

Figure 4-8: Sample Determination in Laboratory

Sample determination process in laboratory is described by ASTM Standard Guide for LIMS [51]. The initiation of a request for testing/sampling starts the sampling process in a laboratory. Manual, phone, process-driven, time or calendar-based, etc. orders for sampling are possible. Laboratory obtains different kind of information (client, biography, requested test(s), safety…) needed for sampling from a sample order. *Sample collection* can precede or follow *sample order*. Unique labels for samples (barcodes) and some documents (collection lists) can be generated during collection and/or login process.

Schedule work process includes adjusting sample priorities and reassigning laboratory work as required. Control samples, and QC samples can also be added to scheduled workflow if needed.

Analysis process contains sample measurement (determination) and data capturing. After the analysis process, the results are reviewed by a qualified person (*verification and correction process*). Once determination results are verified, results can be *reported* to a customer. Some laboratories are able to make *interpretation* and support decision making. *Re-tests* (the same sample will be re-tested) and *re-samplings* (a new sample will be first collected from a patient and then this new sample will be tested) can be initiated at multiple points in laboratory workflow. Figure 4-8 abstracts sample determination in laboratories.

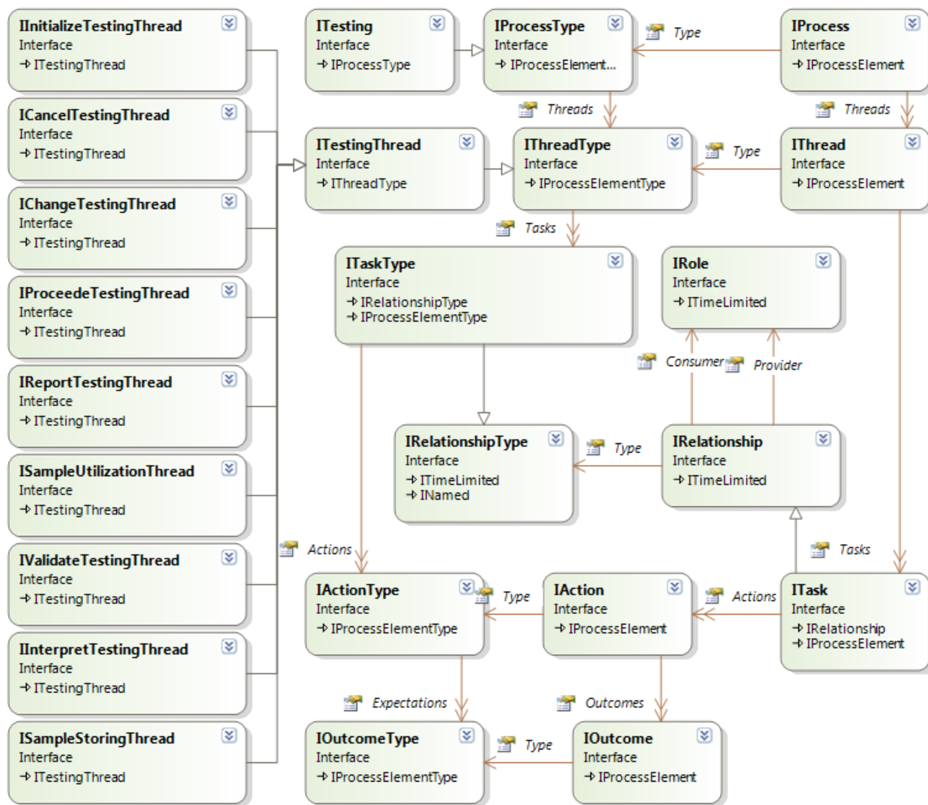

Figure 4-9: Logical Model of Sample Determination

As sample determination is similar to selling of services, we have modelled sample determination (Figure 4-9) similarly to selling (Figure 7-14) process using the business process archetype pattern (Figure 3-25). This means, we have a process which process type is testing (*ITesting*). Each testing process will be initialized by a sampling order and includes testing threads (*ITestingThread*) described by laboratory standard workflow [51]. Each of the testing threads consists of testing tasks (party relationship between two party roles). Each task has activities and each activity has outcomes related to a laboratory inventory

list. Threads in laboratory testing process are initialization, cancellation, changing, preceding, reporting, sample utilization, validation, interpretation and sample storing. As all these sample determination threads are designed similarly to sales threads, described in Appendix 7.2.5, we omit here detailed explanations of these sample determination threads.

4.2.2.2 Quality Control Process

Quality control process (Figure 4-10) in a laboratory is similar to sample determination process in a laboratory (Figure 4-8). In the following, we only describe some important differences rather than the whole QC process.

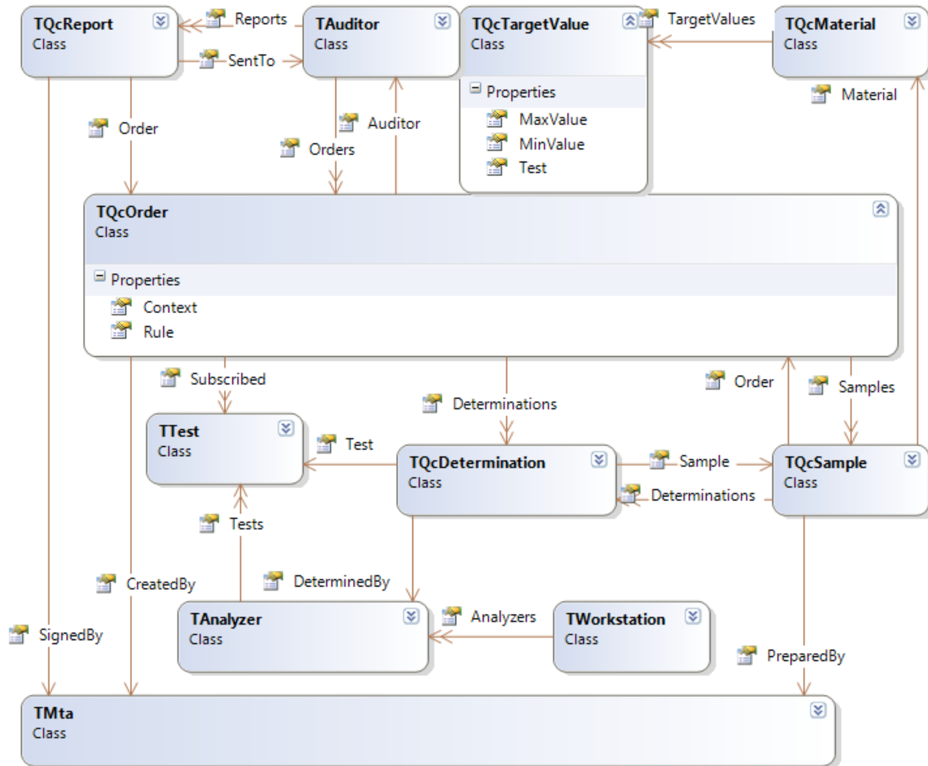

Figure 4-10: Quality Control in Laboratory

In general, QC process is a sample determination process. The difference is that QC samples are made of QC materials and QC materials have targeted values for each test. Therefore the aim of QC determination is to check a determination quality of laboratory analysers. Normally QC process in a laboratory is prescribed by rules and regulations from QC auditors. Examples of QC rules and regulations in laboratories are RiliBÄK [18] and Westgard QC [22]. These rules prescribe when and how the parameters of QC samples should be determined; which are acceptable tolerances from targeted values; how to deal

with non-acceptable differences between targeted values and values obtained by determination; and how to report about QC. Based on these rules, MTAs either create QC orders manually or the generation of QC orders can be automated by rules (Section 3.5.3). Simple example of such a rule is to perform QC determination every morning before laboratory routine. Example of more complicated rules is to perform QC determinations after every thousand normal determinations or after every working hour.

As business process archetype (Figure 3-26) is designed to be managed by rules, it is possible to add different QC rules to needed process archetype pattern archetypes.

4.2.2.3 Planning and Monitoring of Material Requirements

There is a set of different accessories used in sample determination processes in laboratories. Spare parts, reagents and QC materials are examples of these accessories. In terms of A&APs, all these accessories are products and can be modelled using the *measured product* (Figure 3-19) archetype. Therefore by using the inventory archetype pattern (Section 3.5.7) it is possible to monitor the quantity of each of these accessories and by using rules (3.5.3) it is possible to automatically generate purchase orders (7.2.4) and send these orders to supplies.

4.2.2.4 Laboratory Automation

Laboratory automation is how online analysers receive information about what determinations have to be applied to some specific samples and how clinicians and GPs (general practices), by using HIS (Hospital Information Systems), can order tests and receive determination reports automatically. The key to laboratory automation are *communication protocols* ([53; 54; 55]). Communication protocols describe rules and formats of messages sent and received between laboratory instruments and LIMS system or between LIMS and HIS systems. Informally communication between LIMS system and analysers as well as HIS systems is similar to communication between two parties described in Appendix 7.2.1. This informal similarity and our experience in developing OCS (Online Control Server, mentioned in Section 4.4.3) for laboratory automation is why we are working towards a *laboratory automation archetype pattern*. This laboratory automation archetype pattern is based on OSI (Open Systems Interconnection) model, ASTM [54] and HL7 [55] standards and utilizes the business process archetype pattern (Section 3.5.9). This *laboratory automation archetype pattern* is for future study.

4.3 Laboratory Information Management System (LIMS)

The architecture of MyLIS software is illustrated in Figure 4-11. This architecture is derived from the architecture proposed by Helander [12 pp. 467-477]. The data access layer implements object-relational mapping of persistent data. The domain model layer has three sub-layers. Each sub-layer is realized as

DLL (framework, API) and acts as DSL embedded into programming language (C# in case) for the sub-layer above it. Business archetype patterns sub-layer is the DSL for the Laboratory Domain Model sub-layer, which itself is the DSL for the Clinical Laboratory Domain Model sub-layer[2]. On top of the domain model layer, we have a relatively thin service layer [6 pp. 30-32], where requirements for particular laboratory are specified. The Clinical Laboratory Domain Model acts as DSL for these concrete, specified in the service layer, user requirements.

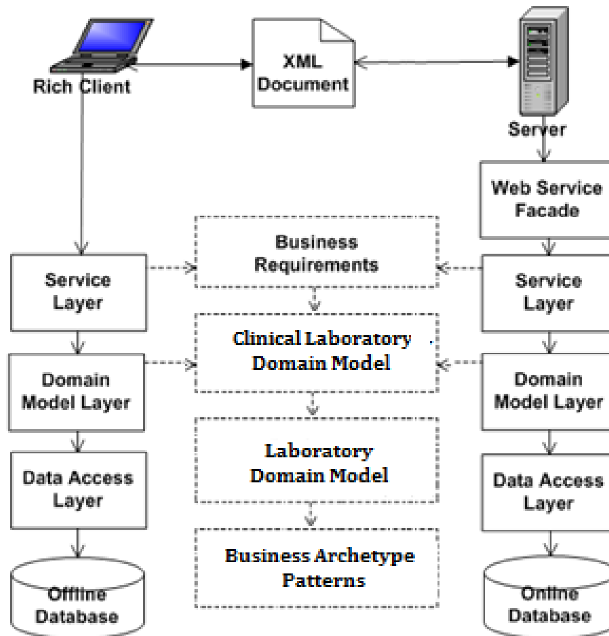

Figure 4-11: The Architecture of MyLIS Software

As the target is to change requirements and domain models even at runtime, the presentation (rich client, web client) as well as communication (XML document based communication interface between server and client) layer artefacts are implemented using reflection [27] (.NET reflection for example).

For this the presentation layer uses access modifiers (private, protected, internal and public) for selecting properties of objects to show in user interface.

[2] In current version we have not separated domain models for clinical laboratory and laboratory domains. We have just one clinical laboratory domain model on top of archetypes and archetype patterns. Refactoring of clinical laboratory models to two separated models is a task for future study.

Figure 4-12: Screenshot of MyLIS user interface

For example, browsing forms show public read-only and read-write properties but editing forms show only public read-write properties. MyLIS user interface, shown in Figure 4-12, is developed so that UI generator analyses the structure (type and properties) of an object and generates UI according to this information at runtime. We also have a small "language" for this; we use this language to describe which properties and in which order to show in UI. For example, the UI, illustrated in Figure 4-12, is generated according to the following scripts.

```
static string[] GridRange = new[] { "Type", "Name", "Symbol", "Code", "Valid" };
static string[] PropertyRange = new[] {"Type", "Name", "Symbol", "Code", "Valid",
        "Description", "Barcode", "Components" };
static string[] EditRange = new[] {"Type", "Name", "Symbol", "Code", "Valid",
        "Description", "Components" };
```

First, the *GridRange* lists the properties, by their names, the master grid shows. Next, the *PropertyRange* lists the properties the detail panel (left side panel of main form) shows. Finally, the *EditRange* lists the properties the edit dialog shows.

In context of document format based changes of information systems, we propose in Section 4.4, this means that we have document formats (*GridRange*, *PropertyRange*, *EditRange*) which describe documents (user interfaces). When we change document formats, the user interfaces, and therefore the information system, will change. As document formats are properties, it is possible to change the values of these properties at runtime using, for example, reflection technology.

We already use such "*document formats*" technology in number of places in current version of MyLIS. For example, the following script (content of file) first describes the automatically generated dialog, shown in Figure 4-13, and then prints the barcode (Figure 4-14) according to entered, using this dialog, values.

```
;Parafin Block Storage Rack Drawer Label
;{a} PB
;{b} I RackFrom = 990
;{c} I RackTo = 990
;{d} C Rack = Foreach(b,c)
;{e} I DrawerFrom = 90
;{f} I DrawerTo = 90
;{g} C Drawer = Foreach(e,f)
;{0} C Barcode = a:2|d:3|g:2
;{1} I Amount = 90
;
m m
J
H 100
S l1; 0,0,9,13,37
O R
B 4,2,0, CODE128, 4, 0.2; {0}
```

$T\ 8{,}8{,}0{,}5, pt6;\ \{0\}$
$B\ 29.8{,}2.2{,}0, DATAMATRIX, 0.4; \{0\}$
$A\ \{1\}$

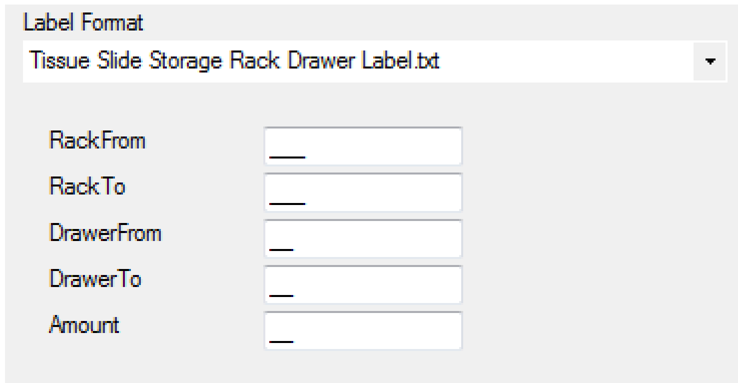

Figure 4-13: Fragment of Generated Barcode Printing Dialog

Figure 4-14: Example of Generated Barcode

MyLIS database layout (under the development, current working version uses object images serialized into flat files) is designed according to A&APs. For example, the medical laboratory has terms patient, physician and MTA (medical technical assistant). However, in the database layout we do not see tables for patients, physicians and MTAs. Database layout (for patients, physicians and MTSs) is designed using only archetypal concepts of the party archetype pattern.

We use the *single table inheritance* pattern proposed by Fowler [6]. In this pattern, the inheritance hierarchy of classes is represented as a single table which has fields for all properties of various classes. As domain model classes, as well as classes designed according to user requirements (if any), are designed so that they are only concretizations of A&AP classes (for example, using attributes technology described in Section 3.5.4), we do not have to know all the derived classes before we design database layouts. Because of the same reason (all classes are concretizations of A&APs) there will be no "empty" fields in database layouts. Such "empty" fields are inevitable side effects of *single table inheritance* pattern as also pointed out by Fowler.

For example, ethnicity, body metrics, date of birth and gender (Figure 3-15) are properties of the person ($IPerson$) archetype. Person as well as organization ($IOrganization$) are both concretizations of the general party ($IParty$) archetype. If mentioned specific person properties (ethnicity, body metrics, date of birth and gender) are all attributes (with category values "ethnicity", "body

metrics", "date of births" and "gender"), then, using *single table inheritance* pattern, the database table for both party as well as for organization classes can be designed as shown in Figure 4-15.

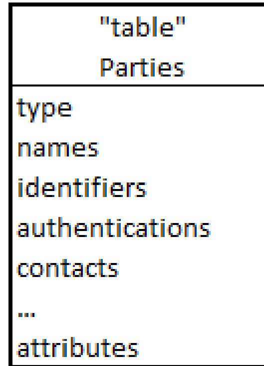

Figure 4-15: Example of Design of Database Table

	A	B	C	D	E	F	G	
1	ID	Type	Name	Symbol	Code	Descriptic	Valid	Barcode
2	STU3225035602	LeedsRtb	Acute Kidney Injury	AKI	30		Always	RegisteredBa
3	STU3733834480	NotLeedsRtb	ARTB	ARTB	34		Always	RegisteredBa
4	STU2848697922	LeedsRtb	Benign Urological	BENIGN	09		Always	RegisteredBa
5	STU1911128967	NotLeedsRtb	Bile	BILE	33		Always	RegisteredBa
6	STU3856181510	NotKnown	Bladder cancer	BLADDER	81		Always	RegisteredBa
7	STU1738294942	NotLeedsRtb	Bladder tissue	BTISSUE	40		Always	RegisteredBa
8	STU1702323915	Completed	Bradford Ov	BRADFORD	51		Always	RegisteredBa
9	STU1101839550	Completed	Breast	BREAST	52		Always	RegisteredBa
10	STU643427237	Completed	Castlemans	CEA	54		Always	RegisteredBa
11	STU2046679989	Completed	Castlemans	CASTLEMANS	53		Always	RegisteredBa
12	STU638379272	Completed	Colo Rectal	COLORECTAL	55		Always	RegisteredBa
13	STU2101544680	Completed	Completed Study	COMPLETED	98		Always	RegisteredBa
14	STU4138547360	LeedsRtb	Controls cataract clini	CC	06		Always	RegisteredBa

Figure 4-16: Example of Generated Excel Reports

Such A&APs based database design should theoretically allow different commercial databases (e.g. Oracle, MySQL, MS SQL etc.) to work with MyLIS software. This is because we use databases only to store data (tables and views) and we do not use database engines for storing logic (stored procedures, triggers, etc.). It should also give a possibility to upgrade user and even domain requirements either without or with minor changes in the database layout and therefore without needs to map data from one DB layout to other.

For independence and performance reasons each client has also an offline (local) database. This is because the job should get done even with no connection to application and/or database servers. Naturally, this needs some built in synchronization mechanisms for data stored in databases.

We also have a customizable MS Excel import/export feature that allows data import from (and export to) MS Excel tables using A&APs based converting. This is implemented similarly to UI generator described above. For example, the following script generates Excel file illustrated in Figure 4-16.

```
static string[] ImportRange = new[] {"ID", "Type", "Name", "Symbol", "Code",
        "Description", "Valid", "Barcode", "Components"};
```

We use such customizable Excel file techniques also in generating customizable user reports.

Similarly to the A&APs based Excel interface, we are also working towards A&APs based XML interface that allows exchange of data and therefore ensures interoperability with other software systems.

4.4 Towards Clinical Laboratory Software Factory

Figure 1-1 illustrates our research and developments towards Software Factory [3] for Laboratory Information Management Systems (LIMS) [1]. Figure 1-1 is based on the software engineering triptych [22]. According to the software engineering triptych, in order to develop software we must first informally or formally describe a domain ($\mathcal{D}$); then we somehow have to derive requirements ($\mathcal{R}$) from these domain descriptions; and finally from these requirements we have to determine software design specifications and implement the software ($\mathcal{S}$), so that $\mathcal{D}, \mathcal{S} \models \mathcal{R}$ (meaning that the software is correct) holds [21].

All models we are talking about are not only documentation artefacts but are source artefacts as common for software factories [3]. It means that the (laboratory) domain model in Figure 1-1 is implemented as DLL.

Let us consider *syntax*, *semantics* and *pragmatics*. *Pragmatics* could be a necessity. Need to cure and be cured. Need to teach and be taught. Need to produce and sell products. Those needs are explicit, bound with particular people and organizations. Perhaps pragmatics are requirements that every particular person or organization have for a system. Maybe this necessities/requirements/pragmatics can be explained to others. How accurately, however, is a different question (perhaps you know the story about the swing – the swings, that the child wanted, that the father understood he wanted and finally the craftsman built, were all different). Perhaps it could be that pragmatics is $\mathcal{R}$ in the Bjørner's equation.

Semantics could be a domain - collection of concepts and relationships between those concepts. Perhaps semantics is a rationalized and generalized abstract language satisfying needs of pragmatics. Semantics could be $\mathcal{D}$ in the Bjørner's equation.

Syntax should then be $\mathcal{S}$ in the Bjørner's equation or some part of $\mathcal{S}$. For instance, when we speak then in addition to speech (syntax), an appropriate infrastructure is also needed (mouth, ears, knowledge for presenting semantics according to specific syntax and knowledge for understanding presented semantics).

Based on what was said previously, it seems that the Bjørner's equation $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ is relatively general equation. Statement „$software\ is\ correct$" could also be more generally comprehensible. For us to be able to explain our necessities R so that those necessities get satisfied, we need a corresponding infrastructure S (syntax - language, alphabet, image, sound, taste, smell, etc. – as a part of infrastructure) and a system of concepts D that is independent of that infrastructure. Information system Π would then be described as $\Pi = (\mathcal{R}, \mathcal{D}, \mathcal{S})$ and this information system is correct if $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ applies.

4.4.1 Paun's P-systems

Let us consider the same thing (Information system Π) in P-system's (Membrane Computing) notation [18].

$$\Pi = (O, C, \mu, \omega_1, \omega_2, \ldots, \omega_m, R_1, R_2, \ldots, R_m, i_0)$$

- O *is a finite and non-empty alphabet of objects* [18]. At the moment we do not specify if they are domain or infrastructure objects.
- $C \subset O$ *is a set of catalysts* [18]. For example, if $ddf \rightarrow faa$, where $O = \{a, d, f\}$, then $C = \{f\}$. This means that f (e.g. the format of a document) is not important by itself. It is only needed to be able to extract a (e.g. domain archetype) from d (e.g. document). More generally: f is infrastructure (eyes, ears, knowledge) that is needed to extract a (semantics) from d (syntax). Or the other way round – to put semantics (a) into the form of a specific syntax (d).
- μ *is a membrane structure, consisting of* m *membranes* [18].
  Notice similarities between the membrane structure (Figure 4-17) and the structure of a company. Companies are structured into divisions; divisions into departments; departments into working groups; and working groups into posts (not workers). That can be given formally
  $$[_1\ [_2\ ]_2\ [_3\ ]_3\ [_4\ [_5\ ]_5\ [_6\ [_8\ ]_8\ [_9\ ]_9\ ]_6\ [_7\ ]_7\ ]_4\ ]_1.$$
  It seems to me that the concept of party role and relationship (Section 3.5.5) is an important concept for specifying organization structures. Let us consider an organizational structure of a laboratory for example. Laboratory is not a party but a role that is performed by a particular juristic person, e.g. Lab Ltd, for a certain period of time. Note that the change of juristic person does not change anything in the laboratory (the role). The same applies for posts. There are posts (also party roles) in the structure of company that a certain party (person) is hired to carry out for a period of time.

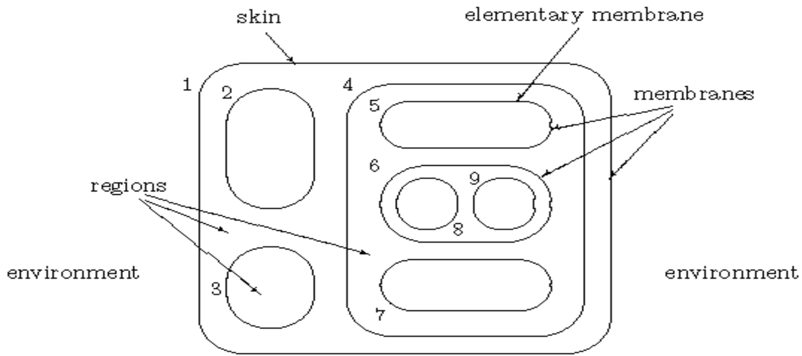

Figure 4-17: A Membrane as an Abstraction of a Company (picture from [18]).

- $\omega_1, \omega_2, \ldots, \omega_m$ *are strings over O representing multisets of objects present in regions* $1,2,..m$ *of a membrane structure* [18]. Every structural unit in a company receives documents (d) (information received in speech or mimics is also equivalent to a document). In every structural unit there are also descriptions of these documents i.e. formats (f) of these documents. For instance, a staff department knows exactly what should be the format (f_i) of a document (d_i) to compose an employment contract document (d_j) according to contract document format (f_j). On the other hand, the staff department is not able to deal with those documents about which they have no corresponding formats. For example, the staff department of a company has not got any idea, what to do with a confirmation document about acquisition of a new car for the company. Best thing to do with those documents is to forward them to the transportation or accounting department. In conclusion: ω_i is a set of documents (d_i) (infrastructure objects), formats of these documents (f_i) (catalysts, infrastructure objects) and corresponding archetypal knowledge (a_i) (archetypal domain objects) that are in a structural unit i (organization or organization unit) at any given time.
- $R_1, R_2, \ldots, R_m$ *is a finite set of evolution rules associated with regions* $1,2..m$ *of a membrane structure* [18]. Situation described in previous clause would be described by following simple rules.

$$d_i f_i f_j \rightarrow d_j f_i f_j a_i$$
$$a_i f_k \rightarrow d_k f_k$$

It means that according to a document d_i (e.g. order), the recruitment of a worker is started. As a result the employment contract (d_j) is made with the worker and archetypal knowledge (a_i) about the worker is created. That archetypal knowledge is then written into a specific document d_k (e.g. record in a list of workers) of the company. It is necessary to have

descriptions of all documents (f_i, f_j, f_k) as they are company-specific. Archetypal knowledge a_i is derived from a domain model.

- i_0 *is either one of the membrane labels* $1,2,..m$, *or* 0, *representing results of computation* [18]. In our example i_0 would indicate the company's ledger, where the "budget" of the system (company) is continuously "recorded" in real time.

In conclusion, information systems have dictionaries, catalysts, structure, data and rules ($\Pi = (O, C, \mu, \omega_1, \omega_2, \dots, \omega_m, R_1, R_2, \dots, R_m, i_0)$) as follows:

- Dictionary ($O = (O_A, O_B, O_C, O_D)$) containing concepts of the following domains:
  - O_A – Agent (for example party Lab Ltd) archetypes and archetype patterns;
  - O_B – Domain (business where agent operates, for example clinical laboratory) archetypes and archetype patterns;
  - O_C – Rule (calculus that agent uses, for example business rules in laboratory) archetypes and archetype patterns;
  - O_D – Document (language that agent is able to use, for example input, output and internal documents in laboratory) archetypes and archetype patterns;
- Catalysts ($C \subset \mathcal{S}; c_i \in C; o_b^j \in O_B;\ o_d^k \in O_D; C = \{c_i\};\ c_i = \{(o_b^j, o_d^k)\}$):
  - Parts of an infrastructure ($C \subset \mathcal{S}$);
  - Describe how language should be understood, i.e. how external language is translated to internal concepts;
  - Are document formats;
  - Describe ($c_i = \{(o_b^j, o_d^k)\}$) the information communicated by a document by using document and domain archetypes. $\{(o_b^j, o_d^k)\}$ is a set of relationships between archetype elements. (o_b^j, o_d^k) is a relationship between two elements and should be read as: the element k of the document archetype d contains the value j of the domain archetype b;
  - Catalysts can be described as requirements. $\forall c_i \in C\ \exists r_\alpha \in \mathcal{R}$ so that $[c_i] = r_\alpha$ (realizes and satisfies requirement) where $c_i == c_j\ \wedge [c_i] = r_\alpha\ \wedge [c_j] = r_\beta => r_\alpha == r_\beta$ (= means "by definition"; == notes identity).
- Structure ($\mu \subset \mathcal{S}; \mu_i \in \mu; o_a^j \in O_A;\ o_b^k \in O_B; \mu = \{\mu_i\};\ \mu_i = \{(o_a^j, o_b^k)\}$):
  - A part of an infrastructure ($\mu \subset \mathcal{S}$);
  - Emulates the real world. For instance, in company's information systems the structure of a company is emulated. That is because in the real world every specific activity is performed in a specific company's structural unit by a person, working on a certain post;
  - Describes an agent ($\mu_i = \{(o_b^j, o_a^k)\}$) who is „educated in a specific domain". $\{(o_b^j, o_a^k)\}$ is a set of relationships. (o_b^j, o_a^k) is a relationship,

that should be read as: agent a is a domain object (archetype) b and the agent's property j (e.g. result of conducted calculations) is the property k (e.g. inventory book) of the archetype b;
  - Structures can be described as requirements. $\forall \mu_i \in \mu \; \exists r_\alpha \in \mathcal{R}$ so that $[\mu_i] = r_\alpha$ with $\mu_i == \mu_j \; \wedge [\mu_i] = r_\alpha \; \wedge [\mu_j] = r_\beta => r_\alpha == r_\beta$;
  - One structural element ($i_0 = \mu_i$) can contain the result ($result \; =$) of the calculation.
- Data ($\omega_1, \omega_2, \dots, \omega_m$):
  - They are documents and catalysts;
  - Catalysts are document formats that convert internal domain language into language (i.e. document) understood by an external environment;
  - Documents are parts of an infrastructure ($\mathcal{S}$). They are syntax – inputs and outputs.
  - Every agent understands documents it sends and receives. Therefore it can execute such operations as $f = getFormat(d_i)$ and $decode(d_i, f)$
- Rules of calculation ($R_1, R_2, \dots, R_m$):
  - Are parts of a infrastructure ($\mathcal{S}$);
  - Essentially algorithms;
  - Can be changed using requirements ($\mathcal{R}$);
  - In a specific structure it describes what has to be done with a document when it arrives.

4.4.2 Relationship between Equations of Paun and Bjørner

Based on discussions we gave above, it seems that there is a relationship between equations $\Pi = (\mathcal{R}, \mathcal{D}, \mathcal{S})$ and $\Pi = (O, C, \mu, \omega_1, \omega_2, \dots, \omega_m, R_1, R_2, \dots, R_m, i_0)$ as follows:

1. Domain, $\equiv (O)$. It seems that four abstract (archetypal) models (agent, domain, calculus and language – or other names) are needed;
2. Software, $\mathcal{S} \equiv (C, \mu, \omega_1, \omega_2, \dots, \omega_m, R_1, R_2, \dots, R_m, i_0)$, containing catalysts, structure, resources for every structural element, algorithms for every structural element, the result;
3. Requirements, $\mathcal{R} \equiv ([C], [\mu], [\omega_1, \omega_2, \dots, \omega_m], [R_1, R_2, \dots, R_m], [i_0])$, are infrastructure descriptions in the language of domain objects. $[x]$ has to be read as a description of x.

4.4.3 World of information systems and agents – informal explanation

An agent is the one that performs „calculus". For that it needs resources (catalysts and rules). It is independent and autonomous, reacts to events (arrival of documents) and is able to learn. This means, that we can change (teach) languages (catalyst), derivation rules (rules), knowledge (domain) and develop the agent (make the agents structure more complicated and more perfect). Input

and output documents are events for agents. Documents are processed according to document formats and according to given rules (resources).

There is a man (man is an agent). The man learnt to play the lute (acquired a domain). The man heard birds singing (language) and studied (derivation rules) to mimic birds singing. The man leant musical notations (new language) and can play the lute (new derivation rule) according to these musical notations. We are dealing with an agent who understands the "calculus" of sounds and can convert (calculate) an input, given in musical notations (document), into an output document presented in sound.

There is a scientist (scientist is an agent). The scientist started to observe the movement of moon and other astronomical objects (acquired a domain). He knows numbers and can write them (language, document). He started to take notes of the movement of moon and other astronomical objects using numbers (conversion rules between numbers and movement of astronomic objects). By analysing these notes, the scientist found relationships and constructed formulae (new language) describing movements of astronomical objects. Using those formulae the scientist leant to predict movements of astronomical objects and solar (and lunar) eclipses (new derivation rules). We are dealing with an agent who understands the calculus of astronomical objects and is able to calculate positions of these astronomical objects.

There is a businessman (businessmen is an agent). The businessman leant to buy fancy spices from India, deliver them to Europe and sell them there (domain). He learnt to write down his deliveries and sales into his notebook (language). He leant to predict his profit (derivation rules). We are dealing with an agent who knows the calculus of business and can calculate the balance of his business.

A software developer developed a software (software is an agent) which is able to communicate with laboratory equipment. This software (Online Control Server) was based on OSI (Open Systems Interconnection) model and ASTM standard protocol E1394-97 which together form a domain of communication protocols for communicating with laboratory analysers. This software was able to communicate with different laboratory devices using different protocols (languages) by converting those native laboratory protocols into E1394-97 standard protocol and vice versa. It was possible to add new protocols (languages) at run-time. This software is an agent which is able to convert (and understand) different communication protocols into E1394-97 protocol and vice versa.

A software developer is developing LIMS. This LIMS knows (learns to know) the domain of a laboratory according to ASTM LIMS standard guide E1578-06. This domain is based on business archetypes. That LIMS knows languages of Excel, XML and SQL. This LIMS can convert information from Excel, XML and SQL documents into laboratory domain concepts and vice versa (calculus). It should be possible to add other derivation rules ("calculus" methods) like statistical analysis methods, data mining methods and etc. into

this LIMS. We are dealing with an agent who aids people working in laboratories to process laboratory data.

4.4.4 *Example as an Informal Proof of Concept*

An agent (staff department for instance) performs calculations (keeps records of employees). We have objects $O = (O_A, O_B, O_C, O_D)$. We have an archetypal agent O_A. According to P-systems, this agent has knowledge (O_b, O_c, O_d), language (C), structure (μ), events (arriving documents ϖ), rules (R) and the result i_0 of agents work (calculation).

$$\boldsymbol{O_a} = \{IAgent = \{O_b, O_c, O_d, C, \mu, \varpi, R, i_0\}\}$$

We have, for example, archetypal domain knowledge about employees. Let an employee be represented just by a simple record (object) containing the name (assume the name to be unique), the start date of the employment contract and the end date of the contract. We have also a staff department with lists of document formats, documents and employees records.

$$
\begin{aligned}
&\boldsymbol{O_b} = \{ \\
&\quad IStaffDepartment = \{IList < IEmployee > Emps, IList < IDoc > Docs, \\
&\qquad IList < IDocF > Formats\}, \\
&\quad IEmployee = \{IName\ Name,\ IEmployed\ Employed\}, \\
&\quad IEmployed = \{IDate\ From, IDate\ To\}, \\
&\quad IName, \\
&\quad IDate \\
&\}
\end{aligned}
$$

We also have an archetypal calculus. These are operations that agent knows.

$$
\begin{aligned}
&\boldsymbol{O_c} = \{\ IList < T > = \{ \\
&\qquad void\ Add(T); \{T\}Fnd(IPredicate); void\ Del(T); \\
&\qquad void\ Upd(T, T); int\ Cnt(IPredicate)\ \} \\
&\}
\end{aligned}
$$

This means, that with every list (T means either $IEmployee, IDoc$ or $IDocF$) an agent is able to perform Add, Find, Delete, Update and Count operations.

We have an archetypal document and an archetypal document format.

$$
\begin{aligned}
&\boldsymbol{O_d} = \{ \\
&\quad IDoc\{IDocF\ Format, IContent\ Content; IDate\ SignedWhen; IName\ SignedBy\}; \\
&\quad IDocF = IList < ICol >; \\
&\quad IContent = IList < IRow >; \\
&\quad IRow; \\
&\quad ICol \\
&\}
\end{aligned}
$$

We have catalysts. Catalysts are parts of infrastructure that can be defined by requirements i.e. descriptions of real documents corresponding to real situations. Currently, for simplicity reasons, formats only describe how many

rows and columns a document consists of and tie every column with a specific domain concept or include predicates (document has to be composed according to these predicates) like formats of requests $\boldsymbol{IRequestEmployeesListDocF}$ and $\boldsymbol{IInfoRequestDocF}$ shown in pseudo code below. In this pseudo code, there is the format of recruitment documents ($\boldsymbol{IRecruitmentDocF}$), the format of dismissal documents ($\boldsymbol{IDismissalDocF}$), the format of error correction documents ($\boldsymbol{ICorrectionDocF}$), the format of documents for requesting lists of employees ($\boldsymbol{IRequestEmployeesListDocF}$), the format of employees list documents ($\boldsymbol{IEmployeesListDocF}$), the format of documents for requesting information ($\boldsymbol{IInfoRequestDocF}$) and the format of information documents ($\boldsymbol{IInfoDocF}$).

$\boldsymbol{C} = \{$

$\boldsymbol{IRecruitmentDocF} = IDocF(\,Content.Rows.Count == 1$
$\wedge Content.Colums.Count == 2$
$\wedge Content.Column[0] == IEmployee.Name$
$\wedge Content.Column[1] == IEmployee.Employed.From\,);$

$\boldsymbol{IDismissalDocF} = IDocF(Content.Rows.Count == 1$
$\wedge Content.Colums.Count == 2$
$\wedge Content.Column[0] == IEmployee.Name$
$\wedge Content.Column[1] == IEmployee.Employed.To\,);$

$\boldsymbol{ICorrectionDocF} = IDocF(Content.Rows.Count == 2$
$\wedge Content.Colums.Count == 3$
$\wedge Content.Column[0] == IEmployee.Name$
$\wedge Content.Column[1] == IEmployee.Employed.From$
$\wedge Content.Column[2] == IEmployee.Employed.To);$

$\boldsymbol{IRequestEmployeesListDocF} = IDocF($
$Content.Description == IPredicate);$

$\boldsymbol{IEmployeesListDocF} = IDocF($
$Content.Description == IDocF(IRequestEmployeesListDocF)$
$\wedge Content.Rows.Count == n \wedge Content.Colums.Count == 3$
$\wedge Content.Column[0] == IEmployee.Name$
$\wedge Content.Column[1] == IEmployee.Employed.From$
$\wedge Content.Column[2] == IEmployee.Employed.To\,);$

$\boldsymbol{IInfoRequestDocF} = IDocF($
$Content.Column[0] == IStaffDepartment.Count(IPredicate)$
...
$\wedge Content.Column[n] == IStaffDepartment.Count(IPredicate)\,);$

$\boldsymbol{IInfoDocF} = IDocF(\,Content.Description == IDocF(IInfoRequestDocF)$
$\wedge Content.Rows.Count == n$
$\wedge Content.Colums.Count == IInfoRequestDocF.Content.Colums.Count$
$\wedge Content.Column[0] == IInforRequestDocF.Content.Column[0]$
...
$\wedge Content.Column[n] == IInfoRequestDocF.Content.Column[n]\,);$

}

We have a structure of an agent. This structure corresponds to a real situation and can be described using requirements.

$\boldsymbol{\mu} = \{IStaffDepartment: StaffDepartment\}$

We have input and output data (documents) according to document formats (catalysts).

$\boldsymbol{\varpi} = \{$
$\quad \boldsymbol{IRecruitmentDoc} = IDoc(Format == IRecruitmentDocF);$
$\quad \boldsymbol{IDismissalDoc} = IDoc(Format == IDismissalDocF);$
$\quad \boldsymbol{ICorrectionDoc} = IDoc(\ Format == ICorrectionDocF);$
$\quad \boldsymbol{IRequestEmployeesListDoc} = IDoc(Format ==$
$\qquad IRequestEmployeesListDocF);$
$\quad \boldsymbol{IEmployeesListDoc} = IDoc(Format == IEmployeesListDocF);$
$\quad \boldsymbol{IInforRequestDoc} = IDoc(Format == IInfoRequestDocF);$
$\quad \boldsymbol{IInforDoc} = IDoc(Format == IInfoDocF);$
$\}$

We have rules of calculations. These rules can be presented as Hoare triplets [33] ($\{Pre - Conditions\}Rules\{Post - Conditions\}$) and describe what to do when a document arrives to a staff department.

$\boldsymbol{R} = \{$
$\quad \boldsymbol{IRecruitmentDoc}: \boldsymbol{d} \rightarrow ($
$\qquad \{ \quad e = new\ Empoloyee\ (\ Name: d.Val[\text{Name}],$
$\qquad\qquad Employed: new\ Employe(From: Name: d.Val[\text{From}]\ ));$
$\qquad\quad StaffDepartment.Fnd(o => o.Name == e.Name) == null)$
$\qquad \};$
$\qquad StaffDepartment.Add(e, d);$
$\qquad \{StaffDepartment.Fnd(o => o == e)! = null)\}$
$\quad)$

$\quad \boldsymbol{IDismissalDoc}: \boldsymbol{d} \rightarrow ($
$\qquad \{ \quad e = new\ Empoloyee(\ Name: d.Val[\text{Name}],$
$\qquad\qquad Employed: new\ Employe(To: Name: d.Val[\text{To}]\ ));$
$\qquad\quad StaffDepartment.Fnd(o => o.Name == e.Name)! = null)$
$\qquad \};$
$\qquad StaffDepartment.Del(e, d);$
$\qquad \{StaffDepartment.Fnd(o => o.Name == e.Name)! = null)\}$
$\quad)$

$\quad \boldsymbol{ICorrectionDoc}: \boldsymbol{d} \rightarrow ($
$\qquad \{ \quad e1 = new\ Empoloyee(\ Name: d.Val[\text{Name,0}],$
$\qquad\qquad Employed: new\ Employe(From: Name: d.Val[\text{From,0}]\ ));$
$\qquad\quad e2 = new\ Empoloyee(\ Name: d.Val[\text{Name,1}],$
$\qquad\qquad Employed: new\ Employe(From: Name: d.Val[\text{From,1}]\ )\ );$
$\qquad\quad StaffDepartment.Fnd(o => o == e1)! =$
$\qquad\qquad null) \land StaffDepartment.Find(o => o == e2) == null)$
$\qquad \};$
$\qquad StaffDepartment.Upd(e1, e2, d);$
$\qquad \{StaffDepartment.Fnd(o => o == e1)! = null)$
$\qquad\quad \land StaffDepartment.Find(o => o == e2) == null)$

```
    }
  )
  IRequestEmployeesListDoc: d → (
    {StaffDepartment.Fnd(True).Length > 0};
    l = StaffDepartment.Fnd(d.Content as IPredicate);
    new IDoc(IEmployeesListDoc, l);
    {l.Length ≤ StaffDepartment.Fnd(True).Length}
  )
  IInfoRequestDoc: d → (
    { StaffDepartment.Fnd(True).Length > 0};
    l = StaffDepartment.Fnd(d.Content as IPredicate);
    new IDoc(IInfoDoc, l);
    {StaffDepartment.Fnd(True).Length > 0}
  )
}
```

We have a result of calculations

$$\boldsymbol{i_0} \rightarrow \boldsymbol{IRequestEmployeesListDoc}(TDate\ d{:}IEmployee.Employed.From \leq d \wedge IEmployee.Employed.To \geq d)$$

4.4.5 There is an Agent

It appears that an information system is an agent that performs calculations and can be described as $\Pi = (O, C, \mu, \omega_1, \omega_2, \dots, \omega_m, R_1, R_2, \dots, R_m, i_0)$.

1. It is collaborative. It receives messages $(\omega_1, \omega_2, \dots, \omega_m)$ from a surrounding environment (messages can be documents) and sends messages to the surrounding environment.
2. It is autonomous. It communicates only by sending and receiving documents.
   a. It decides, according to catalyst (C), if a document is correct (in compliance with the document format),
   b. It decides, according to rules ($R_1, R_2, \dots, R_m$), how to process the document
3. It has an ability to learn. In my opinion this can be realised as follows:
   a. We send messages which include new knowledge (O);
   b. We send messages which include new languages (C)
   c. We send messages which include new agents' structure (μ)
   d. We send messages which include new agents' liabilities $R_1, R_2, \dots, R_m$
   e. Note: I agree that in the beginning this is "learning" and not learning. This means, that we just change the data the system (agent) uses for calculations.

4.4.6 Immutable objects

States of all objects in a system can only be changed with a constructor. This means, that all class properties are read-only.

```
public class ClassName {
    public Cosnstructor (t1 x1 = null, t2 x2 = null, .... tn xn = null )
    public t1 X1 {get; private set;}
    public t2 X2 {get; private set;}
    ...
    public tn Xn {get; private set;}
    ...methods
}
```

This makes it possible to log all changes in a system and avoids changes that are made by accident. Changes are dealt only by a repository (we exemplified the techniques we use in Section 3.5.8). For every particular object there is only one repository in the system. Every repository is responsible for a set (all employees for example) or for a subset (e.g. employees of particular department) of objects. Messages (language), a repository accepts, are the following:

$\boldsymbol{Repository} =$
$\{\{void\ Add(T); \{T\}Fnd(IPredicate); void\ Del(T); void\ Upd(T,T); \ldots\}\}$

Authentication (who is the sender of a message) parameters can be added. A repository must be implemented so that it logs every change and there should always be a possibility to roll back changes.

4.4.7 Dependability – informative meaning

Even if we can describe information systems according to relationships $\Pi = (\mathcal{R}, \mathcal{D}, \mathcal{S})$ and $\Pi = (O, C, \mu, \omega_1, \omega_2, \ldots, \omega_m, R_1, R_2, \ldots, R_m, i_0)$, the question, what does it mean that $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ holds, still exists. Let we have an information system

$$\Pi = (O, C, \mu, \omega_1, \omega_2, \ldots, \omega_m, R_1, R_2, \ldots, R_m, i_0)$$

In the following, we propose some ideas for dependability criteria of information systems.

Let Ω be an amount of all possible (whatever it may mean) messages (documents) and $\Omega_\Pi \subset \Omega$ the amount of those messages that the given system knows.

1. Availability
   a. If we (authorization is not important) do not send any messages to the system, the system will never go down.
   b. If we (authorization is not important) at an arbitrary moment of time send the system an arbitrary message $q_n \in \Omega$, the system will not go down;
   c. If we (authorization is not important) at an arbitrary moment of time send the system a lot of arbitrary messages $q_i \ldots q_j \in \Omega$, the system will not go down;

   d. There is a “death message”, that can be sent by an authorized “death coachman”, which takes the system down.
2. Reliability
   a. If we (some authorized agent) at an arbitrary moment of time send the system an arbitrary message $q_n \in \Omega_n$ then the system sends us an ACK (Acknowledgement) or a NAK (Negative Acknowledgement) message in an agreed time period T.
   b. If we (some authorized agent) at an arbitrary moment of time send the system an arbitrary message $q_n \in \Omega \setminus \Omega_n$ (not recognized by system), then the system sends us a NAK message in an agreed time period T (perhaps necessary, although this is stressing for the system).
   c. If during a time period T we do not receive ACK nor NAK messages then following possibilities exist:
      i. For example, a DoS (denial-of-service) attack or a DDoS (distributed DoS) attack is undergoing and therefore information moves slowly;
      ii. The system (e.g. during DoS or DDoS attack) deals with preserving itself and ignores all messages or some messages selectively;
      iii. The system is down (is killed by “death message” from “death coachman”).
3. Safety
   a. A system does only what is described by requirements given by catalyst C and rules $R_1, R_2, \dots, R_m$
4. Integrity
   a. Processing of a document is an „atomic“ activity. It either occurs or does not. There are no intermediary possibilities.
5. Maintainability
   a. „Health checks“ of a system can take place every day or by other given rules. For example, the whole set of tests (i.e. unit and acceptance tests) can be started whenever it is needed.
   b. „Health monitoring“ of a system can take place all the time. “Health monitoring” means logging of all system events (what documents were received, what changes they made and what documents were sent out). This feature was implemented into the Online Control Server (OCS, mentioned in Section 4.4.3). With OCS logs it possible to track errors. It was also possible to emulate whole previous laboratory days according to log files after any changes made in the system. In addition, it was also possible to do the stress tests (e.g. 1 minute of real time is 1, 0.1 or 0.01 seconds when played back) for the system using real laboratory data.
   c. Evolutionary criteria (change) should also be followed. The condition is that whenever the system is changed, dependability criteria have to be preserved.
6. Confidentiality.
   a. All documents (input and output messages) have to be signed;

b. System reacts (and sends ACK and NAK messages) only to correctly signed documents.
c. All documents can be encrypted.
d. All documents that are sent are also signed by the system (agent)

4.4.8 Evolutionary criteria

It is possible to explain informatively the meaning of $\Pi = (\mathcal{R}, \mathcal{D}, \mathcal{S})$ and $\Pi = (O, C, \mu, \omega_1, \omega_2, \dots, \omega_m, R_1, R_2, \dots, R_m, i_0)$. But what can we do with these explanations? Is it possible, for instance, to derive evolutionary criteria from these equalities?

Table 4-2 Evolutionary versus non-evolutionary information systems

	Non-evolutionary (one-off software developed exactly according to requirements from clients)	Software factory (one-off software **generated** for a client)	**Evolutionary** (Software factory is a part of a system used by a client)
O	Static, code (**classes**, e.g. Unit)	Changes are made by describing new requirements in a software factory.	Changes are made by sending appropriate messages to systems
C	Static, code or permanent data in a database (client based static objects possessing relatively constant values that do not change during the lifetime of a system. E.g. kilogram)	Changes are made by describing new requirements in a software factory.	Changes are made by sending appropriate messages to systems
μ	Static, **architecture** of a system	Changes are made by describing new requirements in a software factory.	Changes are made by sending appropriate messages to systems
ω_i .	Dynamic, all **data**	Dynamic, all **data**	Dynamic, all **data**
R_i ..	Static, **algorithms** in use	Changes are made by describing new requirements in a software factory.	Changes are made by sending appropriate messages to systems
i_0	Static, locations where **results of calculation** are written	Changes are made by describing new requirements in a software factory.	Changes are made by sending appropriate messages to systems

Perhaps being able to change (add and delete as objects are immutable) the following is enough for the system to be evolutionary.

1. Being able to change dictionaries (O, implemented as source artefacts) of agents, domains, calculus and documents. For example, defining of new objects from existing objects. E.g., defining a new domain concept

patient so, that *patient* is a *role* that can only be played by a *party* who is a *person*.

2. Being able to change document formats (catalysts,C) that the system uses for communications.
3. Being able to change the structure (μ) of a system (agent).
4. Data ($\omega_1, \omega_2, \dots, \omega_m$) can be changed anyway. We can add as many lines as needed to documents or send as many documents as needed.
5. Being able to change calculation rules ($R_1, R_2, \dots, R_m$).
6. Being able to change the location (i_0) where calculation results are held.

Table 4-2 describes how changes are implemented in three differently developed information systems.

Changes are made by describing new requirements in a software factory means that there are ready to use pieces (DLLs) that can be put together and generated according to requirements from clients. Information about where and what is running is kept in software factories. As there are several clients with different requirements and configurations, a software factory must be instrumented with a quite complicated version control system which includes a data mapping system from an old system to a new one and an automated testing system. In our understanding, it is possible to avoid data mappings when database layouts are based on A&AP (as described in Section 4.3) and neither on domain models nor on requirements.

An algorithm for "*can be changed by sending appropriate messages to system*" can be something as follows:

1. A document about changes is received (like every document it has to correspond to the form of a document):

   $\boldsymbol{EvolDoc}(Description, Content, Tests, SignedWhen, SignedBy)$

2. A signature of the document is checked and if the signature does not meet authorization rules, the message will be ignored.
3. If the document format is wrong, a NAK (Negative Acknowledgement) message will be sent to the requester and the process of changing of system is cancelled.
4. Which part or subpart of the system (O, C, μ, R or i_0) is about to be changed is determined from the document *description*.
5. A copy is made of the corresponding part ($Content_{old}$) and of tests testing this part ($Tests_{old}$).
6. Changes are made to the corresponding part ($Content$) and to tests testing this part ($Tests$).
7. Tests are started (Including all tests from the previous set of tests except these tests that are amended).
8. If an error occurs while testing, the initial state is restored and a NAK message with an error message indicating a test error is sent to the requester.
9. The log file from a previous day is taken and emulated.

10. If an error occurs in the emulation, then the initial state is restored and a NAK message is sent.
11. The system is running a new version.
12. Possibly there have to be an Undo and a Redo features.
13. Possible recovery of the initial version, in case of errors, can follow tests as well as emulations of previous days.

4.5 Summary

Implementation and testing of the LIMS Software and LIMS Software Factory elements proves feasibility of A&AP models (Section 3.5), laboratory domain models (Section 4.2) and archetypes based techniques (Part 2) in real life systems. Prototypic MyLIS has been used in CBPG (Clinical and Biomedical Proteomics Group) from the end of 2009. It is currently in its third version and is presently used by three different CBPG research groups with different requirements. When in CBPG this software is used in everyday laboratory routine, then for us this LIMS is a test polygon where we evaluate and verify our LIMS Software Factory ideas.

We see the P-systems ($\Pi = (O, C, \mu, \omega_1, \omega_2, \dots, \omega_m, R_1, R_2, \dots, R_m, i_0)$), described by dictionaries (O), catalysts (C), structure (μ), data ($\omega_1, \omega_2, \dots, \omega_m$) and rules ($R_1, R_2, \dots, R_m$), as a roadmap towards evolutionary information systems (Section 4.4). Proposed interpretation of P-systems is based on agent's metaphor. An autonomous agent (e.g. enterprise) is active in a domain (e.g. laboratory) and by communicating (e.g. sending and receiving documents) with an external environment processes (i.e. calculates) information (e.g. tests laboratory samples).

In this interpretation, dictionaries (O) are archetypal concepts used for modelling agents, domains, documents and calculations. Catalysts (C) can be interpreted as document formats describing how external languages (documents) is translated into internal domain knowledge. A structure (μ) is a description (e.g. organization structure) of an agent. An agent is able to communicate with an environment by receiving and transmitting data ($\omega_1, \omega_2, \dots, \omega_m$) and according to algorithms, described by rules ($R_1, R_2, \dots, R_m$), is able to process calculations. An agent keeps results of calculations in some of its structural units (i_0), e.g. in the general ledger of an accounting department.

We described and explained this P-system based approach and derived dependability as well as evolutionary criteria for information systems using this approach. We see this P-system based approach as roadmap to develop information systems that software end users are able to change according to changes in business processes.

5 EVALUATION AND ANALYSIS OF ABD

In the current part, we briefly evaluate and analyse ABD (Archetypes Based Development). We consider *domain analysis and modelling* and *software development processes and methodologies* topics.

We have published [56] the ideas described in Section 5.2 in post conference proceedings of Baltic DB&IS 2010, published in 2011 by IOS, Amsterdam, in the series "Frontiers in Artificial Intelligence and Applications".

5.1 Domain Analysis and Modelling

We analyse ABD, described in Section 2, by comparing it with Dines Bjørner's software triptych principle [22] and with Bjørner's domain analysis methodology [25]. Bjørner's domain analysis methodology is based on domain stakeholders as well as on pragmatically chosen domain facets. Domain facets, according to Bjørner, are: (1) *intrinsic*; (2) *business processes*; (3) *supporting technologies*; (4) *management and organization*; (5) *rules, regulations and scripts*; and (6) *human behaviour*. In the following analysis we follow the domain engineering research topics proposed by Bjørner in 2007 [21]. We start with *domain analysis* research topics (R5...R13) (Bjørner has numbered the research topics as R1 to R17), continue with *infrastructure* (R3, R4) research topics, and proceed with *lifting and projecting* (R2) research topic. We follow with *requirements* (R15, R16), *domain models* (R17), *domain theories* (R14) and we consummate with the $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ *relation* (R1) research topics.

5.1.1 Research Topic R5 – Intrinsic

Bjørner uses *domain intrinsic* [25 p. 264] for these phenomena and concepts of a domain which are fundamental to any of the other (business processes, supporting technologies, management and organization, rules and regulations and human behaviour) domain facets. For instance, in clinical laboratory the intrinsic (Section 4.2.1) can be a *sample*, an *analyser*, a *rack* and a *determination*. These are all products (goods or services) businesses use or make or which are somehow related to business processes and can be abstracted by product archetype pattern (Section 3.5.6). In ABD, instead of the term *intrinsic,* we use the term *product* for all of these concepts of domain, which are things and of which we can ask a question "what".

5.1.2 Research Topics R6 and R7 - Support Technologies

Support technology is a domain facet carrying out business processes [25]. For example, there is support technology radar, which "observes" flight traffic [21]. The radar technology is not perfect. Its positioning of flights follows some probabilistic or statistical pattern [21]. In ABD, business processes and support technologies (also process) are modelled by using business process archetype pattern (Section 3.5.9) which metaphor is report or feedback.

By using feedback, it is possible to collect information about processes. Similar to movies that emulate dynamics of reality by showing sequences of static pictures, the feedback (progress report) emulates business process dynamics by static "pictures" concerning the business process. Based on *party relationship archetype pattern,* each *progress report* (possibly *from* address *to* address) is a party relationship where a *subordinate* reports to a *supervisor*. With this approach, we have a set of reports which is not perfect, but with some probabilistic pattern describes the whole business process.

The quality control (QC) procedure in a laboratory is a support technology. Quality control is similar to the laboratory's main business process. When in the laboratory the main business process is sample testing (some sample parameters are measured) in order to get some information about samples (e.g. reports from analyser to work-area manager), then in laboratory QC procedure QC samples (artificial samples with known parameters) are tested in order to get information about the testing procedure (reports from analyser to QC manager). The QC technology is not perfect, but with some probabilistic or statistical pattern we still can say something about the quality of measurement in a laboratory.

5.1.3 Research Topics R8 and R9 - Management and Organization

In ABD, we strongly separate parties (persons, organizations, artificial agents) from roles (patient, physician, hospital, etc.) these parties are involved with within business domains. We use the *party relationship archetype pattern* (Section 3.5.5) for modelling of *management and organization* (Section 3.3.2).

We only use binary relationships, which mean that one *relationship* binds exactly two roles called "consumer" and "provider". It has to be clarified that the role is always only used to store information that belongs to the role itself and not to a party or to a relationship. *Role type* is used to store common information for a set of similar role instances and *relationship type* is used to store common information for a set of a similar relationship instances. With such party relationship archetype pattern we are able to model quite complicated organization structures as for instance is exemplified in Figure 3-8.

5.1.4 Research Topic R10 - Rules and Regulations

Rules and regulations are prescriptions that have to be followed in order to do business. Rules and regulations are either followed or not. In this sense we can look at rules and regulations as logical statements. This is why, in ABD, all *rules and regulations* are modelled by using the *rule archetype pattern* (Section 3.5.3). A *rule* in the rule archetype pattern is a constraint on the operation which semantic is defined by sequence of *rule elements*. Rule elements can be *operators*, *propositions* and *variables*. Operator is either a Boolean operator (e.g. and, or, xor, not) or a quantifier operator ($=, != , <, >, <=, >=$). When a *rule* represents some kind of a mask or a pattern, then a *rule context* contains an informational context for the evaluation of the rule. When evaluating a rule, we will get either the Boolean value true or false.

5.1.5 Research Topics R11 and R12 - Human Behaviour

Human behaviour is a quality spectrum (careful, diligent, accurate, sloppy, delinquent, criminal, etc.) of carrying out assigned responsibilities. There are the following properties for modelling of responsibilities, capabilities and conditions of satisfactions in party and party relationship archetype patterns (3.5.4).

- Capability (*party*, e.g. “Java programming skill at level 8 out of 10”)
- Responsibility (*role type*, e.g. “motivating the team”)
- Condition of satisfaction (rule set, *responsibility*, e.g. “average score for staff motivation >= 7 out of 10 on staff feedback”)
- Assigned responsibility (*role*, responsibility assigned to the specific party role in specific relationship)

In ABD, we use party and feedback (Section 3.5.9, concretization of party relationship) archetype patterns for modelling of human behaviour. This means, that the monitoring and control of human behaviour is a management process where supervisor gives periodical feedback to subordinates. This feedback is based on stated capabilities, responsibilities and on conditions of satisfaction.

5.1.6 Research Topic R13 - Sufficiency of Domain Facets

Instead of domain facets based methodology, in ABD we use ZF (Zachman Framework) with archetypes and archetype patterns based methodology. We found that the Bjørner’s domain facets based domain analysis method is a special case of the domain analysis methodology based on ZF with archetype patterns (Table 5-1).

Table 5-1: ABD and Bjørner’s Domain Analysis Methodology

	Column 1	**Column 2**	**Column 3**	**Column 4**	**Column 5**	**Column 6**
ZF	What (Things)	How (Processes)	Where (Location)	Who (Persons)	When (Events)	Why (Strategies)
ABD	Product AP	Progress Report AP	Party and Party Relationship AP		Order and Inventory AP	Rule AP
Bjørner	Intrinsic	Main Business Process; Related Processes	Management and Organization	Stakeholders; Human Behaviour		Rules, Regulations and Scripts

For example, the set of intrinsic concepts (basic concepts to any other domain facet) is the subset of all products and services businesses use, buy or sell and can be analysed and modelled by using product archetype pattern (column 1). The main business process together with related processes, management and human behaviour can be analysed and modelled by the business process (reporting or feedback) archetype pattern (column 2). The organization structure can be analysed and modelled using the party relationship archetype pattern (column 3). Stakeholders can be modelled by the party archetype pattern (column 4). Rules, regulations and scripts can be analysed and modelled according to the rule archetype pattern (column 6).

In addition to the Bjørner's facets, the ZF based approach has order and inventory archetype patterns for analysing and modelling of business events (column 5). Such an orders based modelling of events is also used by the REA system [57]. Behind such modelling is the fact that generally all events in businesses are triggered by some kind of orders being either written or verbal.

5.1.7 Research Topics R3 and R4 - Infrastructure Components

We do not use the term infrastructure as it is defined in the World Bank report in 1994 [58] or as used by Bjørner in [21] when posing research topics. The main target of ABD is to generate tailored software automatically according to requirements and domain models. Thus, by the infrastructure we mean the following [12 p. 279].

- Authorization and authentication
- Integration (service requests and responses)
- Data management and access (persistence)
- Presentation
- Logging

By definition, the domain descriptions describe the universe of discourse as it is, without any references neither to software requirements nor to the software design [25 pp. 7-9]. This is why in ABD, domain models are developed as POCO (Plain Old CRL Object; coming from POJO - Plain Old Java Object) objects and are free from any infrastructure-related distractions.

This infrastructure ignorant (similar to persistence ignorance [12 p. 183]) approach, that we use in the engineering of domain models, is in harmony with SRP (single responsibility principle) [47] which states, that every object should have only a single responsibility. In ABD, the domain model is responsible only for acquiring domain knowledge and neither for infrastructure nor for requirements.

If for instance, we are talking about the clinical laboratory, then the clinical laboratory domain describes products, business processes, organization structure, persons, events and business rules used in a laboratory. These descriptions are then used by LIMS [51; 1] software. If for example, the generic LIMS workflow [51] includes features to support laboratory processes (generate

sample request, sample collection, sample distribution, etc.), then the domain model of laboratory describes these processes.

There is a difference between laboratory domain processes and corresponding LIMS processes. When, for example, the sample collection domain process manages samples, then the corresponding LIMS processes manage information (records) about these samples. Additionally, the LIMS processes should deal with information technology related infrastructure (authorization, integration of service requests and responses, data management and access, presentation and logging).

In our understanding these infrastructure components (authorization, integration...) are also domains (infrastructure domains) and can be analysed and modelled similarly to business domains. These infrastructure domain models are playing key roles in proposed foundations (Section 4.4) for developing software factories and evolutionary information systems.

5.1.8 Research Topic R2 - Lifted Domains and their Projections

The *transportation domain* is an abstraction of the more concrete *road*, *rail*, *sea* and *air* transportation domains [21]. For Bjørner such abstracted domains are "lifted" from more concrete domains and concrete domains are "projections" of abstracted domains.

If, for instance, a lifted (abstracted) domain (lets name this domain as party relationship) has types of '*party*', '*party role'*, '*party role type'*, '*party relationship*' and '*party relationship type*' (specified as $IParty$, $IPartyRole$, $IPartyRoleType$, $IPartyRelationship$ and $IPartyRelationshipType$, Figure 3-29), then for example in projected concretizations (for example in domain "party relationships in clinical laboratory") we would probably have to concretize only types of $IPartyRoleType$ and $IPartyRelationshipType$ as shown in Figure 3-30 and as described in Section 3.6.

One possibility for concretisation can be realized by using inheritance as shown in Figure 3-30. Role types in the clinical laboratory (*Medical Technical Assistant*, *Patient*, *Physician*, *Hospital*, *Laboratory*, *Workarea*, etc.) are all general '*party role types*' ($IPartyRoleType$) and party relationship types in the clinical laboratory (*Patient is Hospitalized*, *Manager in Laboratory*, *Medical Technical Assistant in Laboratory*, etc.) are all general '*party relationship types*' ($IPartyRelationshipType$).

The other possibility is to use instantiation as for quantity requirements is shown in Section 2.3. In this case, '*role types*' and '*party relationship types*' in clinical laboratory can be instantiated as "singleton" as shown in Section 3.6.

We use both techniques in our clinical LIMS software factory developments (Part 4). Although, based on our current experiences, it seems to us that from the point of evolutionary information systems (Section 4.4), the instantiation of "singletons" will probably be a better and more flexible solution. At the same time the normal OO inheritance gives clear and simple domain terminology.

5.1.9 Research Topic R15 and R16 – Requirements

In contrast to the procedure based (how to do) software development methods and models we agree, that "*to develop and research a number of requirements-specific domain (software) development models is a grand challenge*" [21]. We see the archetypes and archetype patterns based development (Section 2) as possible requirements-specific development method which combines both "what to do" and "how to do" elements. In our understanding, guided with ZF columns, the ABD includes "what to do" elements and guided with ZF rows the ABD includes "how to do" elements. Still, future developments, research and evaluations are needed.

5.1.10 Research Topics R14 and R17 - Domain Models and Theories

As pointed out by Bjørner, it is a grand challenge to develop and research families of domain models [21]. Despite some progress, to use the archetypes and archetype patterns based methods for development and validation of clinical laboratory domain models and information systems, plenty of research effort is still needed. We expect that the domain models will enable more efficient development, deployment, and support of self-development evolutionary information systems as explained in Section 4.4.

5.1.11 Research Topic R1 - The $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ Relation

In Section 2.3, the $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ relation (from domain model via requirement to software) is exemplified by using a simple domain of quantity. As the domain model of quantity ($\mathcal{D}$) (Figure 3-12) is realized in code as DLL, we can say, that there is a formal (machine readable) description of the quantity domain similar to the following simplified version

```
namespace Archetypes {
  public class TQuanity {   }
  public class TUnit {
     ...
     public string Name { get; private set; }
     public string Symbol { get; private set; }
     public string Description { get; private set; }
     public double Factor { get; private set; }
     ...
  }
  public class TMeasure {   }
}
```

We also have the formal prescription of requirements ($\mathcal{R}$) as exemplified in Section 2.3. The question now is: do we have and if we have, then what is in this quantity example the formal specification of the software design ($\mathcal{S}$) of the software which is able to convert quantity from one particular unit to another

and perform arithmetic and rounding operations with quantities (for instance, divides meters with seconds and gives the answer in kilometres per hour).

If the domain description ($\mathcal{D}$) is a model of an application domain (quantity currently) in some language and if the requirements prescription ($\mathcal{R}$) prescribes in some language what the software is expected to do and if the software design ($\mathcal{R}$) specifies in some executable programming language, how execution may proceed, then why cannot the software design be a domain model realized in some programming language.

It seems now, that the domain model description ($\mathcal{D}$) and the software design ($\mathcal{S}$) are one and the same. However, they do not match exactly. First, the software design is not only a domain description in programming language, but also a tool for prescribing requirements. Secondly, we can use technology of interfaces to fully separate semantics of the domain model from its realization. In the following example code, the description of term "unit" (term from quantity domain) is specified as interfaces by using programming language C#.

```
public interface IUnit : INamed, IComparable, IComparable<IUnit> {
        IMeasure Measure { get; }
        Double Factor { get; }
        string Symbol { get; }
        ReadOnlyCollection<IUnitTerm> UnitTerms { get; }
        ISystemOfUnits SystemOfUnits { get; }
        double AmountToBaseUnitAmount(double amount);
        double AmountFromBaseUnitAmount(double amount);
        IUnit Power(int power);
        IUnit Inverse();
        IUnit Multiply(IUnit unit);
        IUnit Divide(IUnit unit);
    }
```

Now in quantity domain, we have the description of the domain ($\mathcal{D}$) (as interfaces in programming language C#), the design of the software ($\mathcal{S}$) (implements the domain descriptions as DLL in programming language C#) and the prescription of requirements ($\mathcal{R}$).

So designed software acts as domain specific language (DSL) embedded into general purpose programming language C#. So prescribed requirements prescribe in provided DSL what the software is expected to do and the C# compiler generates software according to $\mathcal{D}, \mathcal{S}$ and $\mathcal{R}$ so that $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ (means that the software is correct) holds.

But how we can be sure, that the $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ holds and what does it mean that the software is correct? Without loss of generality, this assertion can be in some form of a pre/post condition of $\mathcal{S}$ [21]. Now, if $\mathcal{P}$ indicates pre conditions and $\mathcal{Q}$ indicates post conditions, then according to Hoare triple [33] we can write $\mathcal{P}\{\mathcal{S}\}\mathcal{Q}$ and interpret it as follows - "If the assertion $\mathcal{P}$ is true before execution of a software with a software design $\mathcal{S}$, then the assertion $\mathcal{Q}$ will be true after execution of the software".

We can interpret the software design $\mathcal{S}$ (in object oriented world) as

$$\left\{\mathcal{S}_1^1, \mathcal{S}_1^2, \dots \mathcal{S}_1^\alpha, \mathcal{S}_2^1, \mathcal{S}_2^2, \dots \mathcal{S}_2^\beta, \dots \mathcal{S}_n^1, \mathcal{S}_n^2, \dots \mathcal{S}_n^\omega\right\}$$

In this interpretation, the element $\mathcal{S}_i^\xi$ (the lower index indicates a class and the upper index indicates an element of the class) is some class element (method, property, field or event). Let us assume (good software design, no copy and paste programming techniques), that there are no duplications in software design.

Formally this means, that

$$\mathcal{S}_i^\xi = \mathcal{S}_j^\zeta => i = j \wedge \xi = \zeta$$

Let us also assume, that every single part $\mathcal{S}_i^\xi$ of software design is unit tested [24]. This means, that for every i and for every ξ there is a unit test with pre and post conditions ($\mathcal{P}_i^\xi$, $\mathcal{Q}_i^\xi$) so that $\mathcal{P}_i^\xi\{\mathcal{S}_i^\xi\}\,\mathcal{Q}_i^\xi$. We can read this as follows: "Based on our best current knowledge this small amount of software is correct because according to the unit test the assertion $\mathcal{P}_i^\xi$ is true before execution of this amount of software designed as $\mathcal{S}_i^\xi$ and the assertion $\mathcal{Q}_i^\xi$ is true after execution of this amount of software".

Let us assume now, that a part of software is designed to satisfy domain descriptions $\left\{\mathcal{S}_1^1, \mathcal{S}_1^2, \dots \mathcal{S}_1^\alpha, \ \dots \mathcal{S}_k^1, \mathcal{S}_k^2, \dots \mathcal{S}_k^\xi\right\}$ and a part of this software $\left\{\mathcal{S}_{k+1}^1, \mathcal{S}_{k+1}^2, \dots \mathcal{S}_{k+1}^{\xi+1}, \ \dots \mathcal{S}_n^1, \mathcal{S}_n^2, \dots \mathcal{S}_n^\omega\right\}$ is designed to satisfy requirement prescriptions. Assuming, that all software design is covered by unit tests, it may be correct to say that the prerequisite, the $\mathcal{D}, \mathcal{S} \Vdash \mathcal{R}$ holds, is that all unit tests ($\mathcal{P}_i^\xi\{\mathcal{S}_i^\xi\}\,\mathcal{Q}_i^\xi$) have to pass.

If so, then it could be wise to describe domains as well as prescribe requirements in terms of unit tests (contextual and semantic models) as we explained in Section 2.2. Based on these ideas we see possibilities to expand and elaborate the archetype based domain analysis and modelling methodology, integrate it with information systems self-development approach, and work out techniques for integrating domain models with software factories (Section 4.4).

5.2 Software Development Processes and Methodologies

Sometimes it seems to me, that the main issue in software development is whether to do extreme programming or not to do extreme programming. Not that I find KISS (*Keep It Simple and Stupid*), YAGNI (*You Aren't Going to Need It*), DSTCPW (*Do the Simplest Thing that Could Possibly Work*) and other *extreme programming (XP)* [5] and agile software development [59] truths and practices useless. I just do not believe that they are absolute and universal. Based on my experience, these agile practices can exist in harmony even with

statements such as „...*formal techniques apply in all phases, stages and steps of software engineering, and in the development of all kinds of software* ...“ [22 p. 9]. I also agree, that to not use formal techniques „... *would be tantamount to cheating the customer — also known as criminal neglect* ...“ [22 p. 31].

5.2.1 "What to do" versus "How to do"

Based on my experience as a software developer and on some software development process knowledge [60], it seems to me that the majority of software development methodologies try to reduce the software development risks by making things right.

However, studies show [61] that risks in software development are related not only to the software development process ("how to do"), but are also related to *unrealistic and unarticulated project goals* ([61], the main reason of failure), with *badly defined system requirements* ([61], the third reason of failure from top) and with *inability to handle the project's complexity* ([61], the eighth reason of failure from top), which I think are more "what to do" (domain) problems rather than "how to do" (process) problems.

Table 5-2: Effectiveness and Efficiency in Software Development

Effectiveness – Level of achieving the objectives (High ↑ / Low ↓)	Bad	Good
High	**Right thing is done wrongly.** Software developers achieve their objectives though they use software development process which is not correct and mature. Waste of resources. *Improving of software development process can make here things better.*	**Right thing is done correctly.** Software developers achieve their objectives by using software development process which is correct and mature. They use resources rationally.
Low	**Wrong thing is done wrongly.** Software developers do not achieve their objectives.	**Wrong thing is done correctly.** Software developers do not achieve their objectives though they use a software development process which is correct and mature.

Bad ← Efficiency – Using of resources → Good

I do not know exactly the background of Charette's work [61], but if „... *IT projects rarely fail for just one or two reason* ...“ [61] and „... *failures, in fact, can be treated to combination* ... “ [61], and if each such combination includes one of the mentioned reasons (unrealistic goals, badly defined requirements or

complexity), then I will be brave to speculate that, if we do not know exactly what to do, then it absolutely does not matter which development procedure we use (how to do) in software development – we do not get useful results anyway.

There are many such examples in the world (even the story of LINUX operating system) where the top-level software is developed by using the lowest maturity level (Initial) [62] software development process. According to a study, which was done some ten years ago [63], 70% of the software companies have worked on the CMM first level. Not all of these projects have gone wrong after all. However, based on the Standish Group research, Greenfield etc. [3] argue that only 16% of all software projects are successful, 51% will require considerably more time and money than originally planned and 31% of the software projects are terminated primarily because of their poor quality.

In my understanding, if we do not know exactly what is needed (business or domain requirements of software), it is absolutely irrelevant which of the software development processes we will use – this software will never be ready or even if it will be ready, then it will not be usable. If, however, we know exactly what to do (the business requirements of software) then this software will be ready at some point and will be usable. But by using mature software development process, we can develop this software more efficiently and economically - that is to say more profitably. Table 5-2 summarizes my understanding about software development processes.

5.2.2 DDD and TDD from Software Triptych Perspective

Dines Bjørner [22] has formulated the relationship (probably derived [3] from "satisfaction of the requirement" relationship by Jackson and Zave [64]) between the software development process and software requirements as the *software engineering triptych*, which consists of following stages.

1. From domain analysis [26] to the formal domain model.
2. From the formal domain model via specifying and proper selection of domain features to software requirements.
3. From software requirements (for example by using test driven development [24]) to the dependable [48] and correct software.

According to the software engineering triptych and on condition that someone (for example, a user incorporated into the software development team – one of the main Extreme Programming [5] practises) knows exactly what should be requirements for the software under the development and provided that those requirements are not just too inconsistent, the Test Driven Development (TDD) [24] and Extreme Programming [5] as well as the entire agile software development should be adequate to produce high-quality and dependable software.

[3] Thanks to Daniel M. Berry for this comment.

Such customer-requirements-specific one-off software development with "*stop trying to model the real world*" [46] strategy should ensure that the third stage of the software engineering triptych - from the software requirements to the dependable software – is of sufficient quality. As reported by Paulk [65], the extreme programming development is at relatively high level from the perspective of CMM.

However, software requirements from the customer can sometimes be very controversial. In addition, the client may simply forget to explain something or forget to talk about some of exceptional cases, which can transform the entire big picture that developers have got so far. A tool against such conflicting claim should be Domain Driven Design (DDD) [66] introduced by Evans[4]. Software development with DDD in combination with TDD should provide a much better result than software development without TDD and DDD. It seems to me, that the software development by using DDD and TDD is like the application of the second stage of the software triptych in reverse – from specific customer requirements to the domain model.

Software developed using DDD, should be more dependable than one developed without DDD. Unfortunately DDD supports mostly one-off software developments (developed domain model is based on concrete requirements) and the developed software can be used only by those companies whose business process is compatible with the business process realised in the software or by those companies who are willing to adapt their business processes according to the software.

The author of thesis faced such problem in 1999-2005 when developing the Multilab™ LIMS software [67; 68] for small and medium sized clinical laboratories. All the laboratories (approximately 60 laboratories in Germany) that implemented the Multilab™ software were to change their business process in a greater or lesser extent.

However, what can be done with these companies who for some reason do not want or cannot change their business processes[5]? Company's unique business process can be the most valuable strategy to make profitable business.

In current thesis, the proposed archetypes and archetype patterns based development techniques for developing domains, requirements and software is designed according to the software triptych. We see this proposed archetypes and archetype patterns based development as one of those requirements-specific

---

[4] A very useful DDD book [12] is written by Nilsson.

[5] Thanks to the Clinical and Biomedical Proteomics Group (Cancer Research UK Clinical Centre, Leeds Institute of Molecular Medicine, St James's University Hospital at University of Leeds), who was not willing to change their business process, it was possible to fund this project and to complete the thesis.

(what to do) development methods [21] rather that process-specific (how to do) development methods. With ABD we see possibilities to lead software development towards software factory and thence towards possibilities for end users to evolve software systems in evolutionary way together with business processes.

5.2.3 ABD and MDA

Table 5-3 summarizes how, in our opinion, the software triptych, ZF and Model Driven Architecture (MDA) [69] activities are related.

Upper (first and second) rows of ZF correspond to requirements from some concrete enterprise in the context of the software triptych and to the Computing Independent Model (CIM) in the context of MDA. We interpret CIM as a conceptual and business level model that is a product of the enterprise requirements analysis process. Middle (third and fourth) rows of ZF correspond to the domain part in the context of the software triptych and to the Platform Independent Model (PIM) in the context of MDA. The PIM is interpreted as a logical design model. Lower (fifth and sixth) rows of ZF correspond to software part in the context of the software triptych and to the Platform Specific Model (PSM) in the context of MDA.

Table 5-3: The Rows of ZF in the Context of Software Triptych and MDA

Triptych	ZF Rows	MDA
Requirements	1,2	CIM (conceptual, business, analysis)
Domain	3,4	PIM (logical design)
Software	5,6	PSM (physical implementation)

5.2.4 ABD and XP

Extreme Programming (XP) [5] is an agile software development methodology with basic practices like test driven development, pair programming, planning game, continuous integration, small releases, metaphor, simple design, refactoring, collective ownership, 40-hour week, coding standards and so on. Table 5-4 (made by using the similar table from [65]) summarizes ABD and XP activities.

While XP is for development of tailored one-off software for customer and is based on customer requirements, the ABD is for development of software factories (SF) so that tailored one-off software for specific customer requirements can be generated automatically (at least partially) by using SF tools and other artefacts.

Table 5-4 has two columns for ABD. The "ABD for SF" column summarizes how to use XP activities when developing software factory artefacts. The "ABD for Software" column summarizes the activities needed for generating software from SF according to customer needs.

Table 5-4: Comparing XP and ABD

Common-sense	XP extreme	XP practice	ABD for SF	ABD for Software
Manage requirements	Review requirements all the time	On site customer	On site domain specialist	Requirements are coded in DSL so that customer can validate them
Code reviews	Review code all the time	Pair programming	Pair programming	Code is generated (largely) automatically
Testing	Test all the time	Unit testing, functional testing	Unit tests based domain modelling	Domain model validates and verifies requirements
Design	Design is everybody's daily business	Refactoring	Refactoring towards archetype patterns	Archetypes and archetype patterns based predefined by SF design
Simplicity	Simplest design that supports the system's current functionality	The simplest thing that could possibly work	The simplest abstraction that could possibly work	
Architecture	Everybody works to refine the architecture	Metaphor	Based on ZF with archetype patterns	Archetypes and archetype patterns based predefined by SF architecture
Integration testing	Integrate and test several times a day	Continuous integration	Continuous integration	
Short iterations	Short (sec, min, hours) iterations	Planning game	Archetypes based planning (game)	
Manage versions	Plan and release frequently small units of business functionality	Frequent small releases	Archetype patterns based releases	Requirements based step by step releases with possibilities to undo and redo.

We mostly use XP practices in combination with domain analysis and domain modelling activities when developing A&AP based software factory artefacts.

Still, instead of an “on site customer” we need an “on site” domain specialist. Instead of XP practices, where everyone can change design (refactoring) as well as refine the architecture (metaphor) towards the simplest thing and design that can possibly works, in ABD, when developing SF artefacts, we have relatively fixed ZF with archetype patterns based architecture. Therefore the refactoring is mostly towards efficient and universal use of archetype patterns.

ABD uses the XP unit testing practice in domain analysis and modelling (Section 2.2). This means, that all domain narratives are specified (contextual and semantic models) as unit tests [24]. We call this approach Test Driven Modelling.

When the SF is ready, we can hopefully generate one-off software automatically (Figure 1-1). This means that by using a domain model based DSL (domain specific language) we “code” customer requirements. The DSL has to be designed so that a domain specialist is able to understand this DSL and is able to validate correctness of so specified requirements.

The software generated will be based on these requirements. Requirements will be first validated according to the domain model and the generated software will be verified according to requirements as well as according to domain model. As the final validation and verification can be conducted only when the software is deployed into the real environment and used by the customer in real everyday business, it is wise to implement and deploy requirements step-by-step. For these purposes the undo and redo mechanisms for requirements as well as for data have to be implemented in SF artefacts.

ABD complements XP by focussing on understanding of the domain (what to do) and on modelling domains formally, on the decision analysis and resolution (by selecting a solution that meets multiple demands of relevant stakeholders), on requirements development (by describing customer requirements in terms of domain) and on validation and verification by validating requirements against domain models and verification of software according to specified requirements.

5.2.5 ABD and CMMI for Development

CMMI (Capability Maturity Model Integration) for Development *“is a process improvement maturity model for the development of products and services”* [62]. It describes best practises for improving maturity of software development. In the following, we refer to CMMI for Development as to "CMMI".

ABD addresses (Table 5-5) many of the CMMI Level 2 *requirements management* process area (*PA*) specific practices through its use of the domain model, synopsis (similar to stories in XP) and narratives (similar to XP tasks). When XP integrates feedback on customer expectations and needs by

emphasizing short release cycles and continual customer involvement, the ABD maintains "common understanding" through the ZF with archetypes and archetype patterns by asking questions what, how, where, who, when and why. Although requirements from customers might evolve dramatically over time, in our understanding, a properly abstracted and formalized domain model simplifies the introduction of changes to specifications, as requirements are in terms of domain model. In addition, it reduces the risks involved with introducing these changes, as Test Driven Modelling enables us (at least partially) to validate user requirements according to domain models and to verify software according to so specified user requirements.

Table 5-5: ABD Satisfaction of CMMI Process Areas

Level	**Key process areas**	**Satisfaction**
2: Managed	Requirements Management	++
	Project Planning	+
	Project Monitoring and Control	+
	Measurement and Analysis	+
	Process and Product Quality Assurance	+
	Configuration Management	-
	Supplier Agreement Management	-
3: Defined	Organizational Process Focus	-
	Organizational Process Definition	-
	Organizational Training	-
	Integrated Project Management	-
	Risk Management	+
	Decision Analysis and Resolution	-
	Requirements Development	++
	Product Integration	++
	Technical Solution	++
	Validation	++
	Verification	++
4: Quantitatively Managed	Organizational Process Performance	-
	Quantitative Project Management	-
5: Optimized	Organizational Innovation and Deployment	-
	Causal Analysis and Resolution	-

++ largely, + partly and - not addressed in ABD

Although the archetype patterns based system architecture establishes the project's main direction, in ABD the project plan (*project planning* PA) is not detailed for the project whole life cycle. Still, by analysing and designing requirements in terms of archetype patterns based domain models together with

XP practices like short iterations (1-3 weeks) and small releases (2-6 months) enables developers to identify and manage their plans.

ABD addresses *project monitoring* similarly to XP by using “big visual chart”, project velocity, and commitments for small releases. The “big visual chart” in XP means an open workspace together with white board based information reflecting the projects progress and close communication between project members and an onsite customer. An overall schedule and budget in XP are calculated by figuring the estimated time for the work factored with the project velocity (40-hours weeks, implemented tasks per developer per week and etc.). By using small releases, the feedbacks for commitments from real users from the real environment provide reassurance and the opportunity to intervene fast. All these activities are also ABD activities. Differently from XP, where the development team is a lot like an explorer with a compass, the ABD team is also equipped with a decent map – the ZF with archetypes and archetype patterns gives additional possibilities (where we are, how much is to go) for measurement and analysis of both work products as well as development processes.

ABD addresses Level 3 *risk management PA* (manage risks with continuing and forward-looking activities that include identification of risk parameters) partly through activities described already in project monitoring and control PA. Additionally some preventative activities like customer readable simple synopsis and narratives, archetypes and archetype patterns based design, refactoring, coding standards, unit testing and especially unit testing based modelling are all elements of risk management.

Requirements development PA (identifies customer needs and translates these needs into high-level conceptual solutions) is addressed in ABD through describing customer requirements in terms of the archetypes and archetype patterns based domain models. Translation of customer requirements into domain model (or A&AP) terms is one of the key features of ABD. The archetypes based domain model is also the key feature that addresses Level 3 *product integration* (generate the best possible integration sequence by integrating product components) and the *technical solution* (develops technical data packages for product components) PA’s. The same is also true for *validation* (incrementally validate products against customer’s needs) and *verification* (ensure that selected work products meet specified requirements) which are both natural components of ABD (Section 2.2). By formal analysis of requirements through using archetypes and archetype patterns based domain analysis and modelling techniques, the subjective nature of requirements, design and architecture decisions will be reduced in order to select solutions that meet multiple demands of relevant stakeholders.

In conclusion we can say that by using ABD it is possible to cover some institutional practices that the CMMI for Development identifies as key elements for good engineering and management.

6 CONCLUSION

6.1 Contributions

This work is based on software engineering triptych (from domain models via requirements to software) proposed by Dines Bjørner and on archetypes and archetype pattern base initiative proposed by Arlow and Neustadt. These ideas are used in engineering of domain models for clinical laboratory and in LIMS (Laboratory Information Management System) software development.

The resulted work is archetypes and archetype patterns based techniques for engineering of domains, requirements and software. In our understanding by using these techniques we can lead software developments towards software factory developments. The wider research goal is to develop archetypes and archetype patterns based information systems that software end users, in collaboration with software developers, are able to change safely and easily according to changes in business processes.

The contributions of thesis are *Archetypes Based Development (ABD)* techniques for development of domains, requirements and software (Part 2) and improved models of *Business Archetypes and Archetype Patterns* (A&AP) (Part 3). The ABD includes (Section 2.1.1) ZF (Zachman Framework) columns based analysis (by asking questions *what*, *how*, *where*, *who*, *when* and *why*) and design (*products*, *processes*, *locations*, *persons*, *events* and *rules*) of domains and requirements by using *archetypes and archetype patterns*. The ABD also includes ZF rows based development (Section 2.1.2) – from *conceptual* and *semantic* models via *logical*, *physical* and *detailed* models to software *product*. In ABD the validation and verification (Section 2.4) of requirements and software is based on the Test Driven Modelling (Section 2.2) techniques. Business A&APs (Part 3), used in ABD, are models (code artefacts) used for modelling independent phenomena (*products*, *processes*, *locations*, *persons*, *events* and *rules*) of ZF.

In Part 4 we exemplified the usefulness of ABD and A&AP models in real life software developments. We presented the domain model of laboratory (Section 4.2), where the ABD and A&AP models were utilized. We also described LIMS software (Section 4.3) developed for and already used in everyday laboratory routine by Clinical and Biomedical Proteomic Group (Cancer Research Clinical Centre, Leeds Institute of Molecular Medicine, St James's University Hospital at University of Leeds). We also presented possibilities to use domain models as objects in P-systems (4.4). In our understanding this P-systems based approach leads us towards information systems that software end users, in collaboration with software developers, are able to evolve in an evolutionary way according to changes in business processes.

While implementation and testing of the LIMS Software proves feasibility of archetypes based techniques in real life systems, these A&APs based techniques

are also in agreement with and complement important software development processes and methodologies, such as Bjørner's domain modelling, MDA (Model Driven Architecture), XP (Extreme Programming) and CMMI (Capability Maturity Model Integration) for Development as explained in Part 5.

6.2 Hypothesis

We claimed (Section 1.4) that *archetypes based development techniques* (ABD) together with proposed models of *archetypes and archetype patterns* (A&AP) lead software development towards software factory (SF) development and thence towards possibilities to fulfil user requirements by making changes only in the presentation or in the communication layers.

This claim we summed up into 6 conjectural points (Section 1.4) about which, based on our work done, we can say the following.

1. Triptych software development (from domain models via requirements to software) is possible and reasonable.

In Part 2 we explained and exemplified *archetypes based development* (ABD) techniques by using simple domain model of quantity. ABD includes *Zachman Framework based analysis* (Section 2.1.1), *triptych software process* (Section 2.1.2) and *test driven modelling* (Section 2.2). Archetypes and archetype patterns (A&AP) (Part 3) are an integrated part of the ABD. We use A&APs as DSL (Domain Specific Language) for developing domain models (Sections 3.6 and 4.2). So developed domain models we use as DSLs for specification (Section 2.3) and for verification (Section 2.4) of requirements. We use ABD techniques in development of real life software. ABD is also in agreement with and complement important software development processes and methodologies such as Bjørner's domain modelling, Model Driven Architecture, Extreme Programming and Capability Maturity Model Integration for Development as shown in Part 5.

2. We can develop models (frameworks, source artefacts) of A&AP. We can develop domain models by using these A&AP models.

We presented the A&AP model in Section 3.5. This model is an improved version of A&APs originally proposed by Arlow and Neustadt [14]. We separated from these Arlow and Neustadt models the knowledge and operational levels as suggested by Fowler [15] and added the archetype pattern for business processes. We evaluated (Section 3.3) A&AP models by comparing them with models by Fowler [15], Hay [16] and Silverston [17] and found that all these patterns describe similar phenomena of businesses but are modelled differently i.e. these models are *semantically heterogeneous* [37]. Our presented A&AP models are in harmony with *Zachman Framework* (Table 2-1) and Triadic Model of Activity (Figure 3-1) and is designed to abstract the universe of discourse of businesses as it is, neither referring to the software requirements

nor to the software design. Presented A&AP models are a framework, realized in .NET with C# by using ABD, described in Part 2.

We exemplified the development of domain models according to ABD techniques (Section 3.6) and presented the design of the domain model of laboratory (Section 4.2). We evaluated the development process of domain models, we suggest and use, by comparing it (Section 5.1) to domain analysis and development methodology by Bjørner. We found that our proposed ZF based methodology complements domain facets methodology proposed by Bjørner (Table 5-1). Proposed laboratory domain model (Section 4.2) is designed according to ASTM LIMS Standard Guide [1; 51]. The verification of compliance with other important laboratory and health care standards like Health Level Seven [55], openEHR [70] and communication protocols between laboratory instruments and laboratory software [53; 54] is for future study.

3. We can specify user requirements by domain and/or A&AP models. We can generate software according to so specified user requirements.

We presented our ideas how user requirements can be specified by domain models in Section 2.3. In real life LIMS software development, for specification of user requirements, we use laboratory domain model based DSL. This DSL is realized as embedded into general purpose programming language (C#) API (framework). We use this laboratory domain model based DSL for specifying user requirements similarly as we used A&AP based DSL for specifying domain models (as exemplified in Section 3.6).

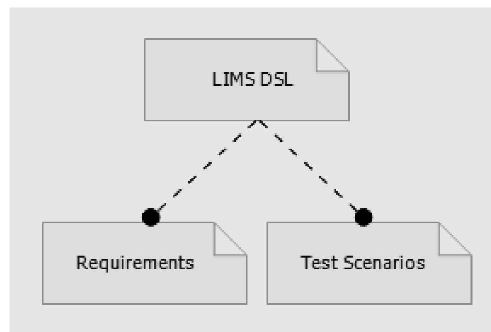

Figure 6-1: Joint Specification of Requirements and Test Scenarios.

In design-time, using DSLs embedded into general purpose languages is good enough and in our understanding this technique is suitable for analysing of domains and for development of domain models. Unfortunately this technique does not work at run-time. Thus we need some languages and tools to describe user requirements and test scenarios as illustrated in Figure 1-1. It would be most beneficial, however, to specify requirements and test scenarios jointly as illustrated in Figure 6-1. This task will be for future study.

Figure 1-1 illustrates our developments towards software factory. Currently we do not generate software automatically and therefore this will be the main

task for future studies and developments. Still we already generate user interfaces as we briefly described in Section 4.3 and we have some simple techniques to work with documents and document formats as we proposed in Section 4.4 and already use for some simple cases as described in Section 4.3

4. We can validate user requirements and verify software by using these models. User requirements can falsify domain as well as A&AP models.

We discussed and exemplified this in Section 2.4. We see possibilities (at least partially) to validate requirements as well as to verify software with domain models developed according to TDM (Test Driven Modelling, Section 2.2). If with domain model based DSL (embedded into general purpose language, API) it is possible to prescribe user software requirements, then these requirements are valid (compatible) according to this domain model. If both, domain descriptions specified as unit tests and software requirements specified as acceptance tests, are satisfied ("green" pattern in Figure 2-2), then in our understanding the domain model has verified (at least partially) these requirements. If a domain model satisfies some of the real life requirements, then we can just say that these requirements have not falsified the domain model. But if with this domain model we cannot satisfy one particular requirement from the real life, then this requirement (in case the requirement is correct) has falsified the domain model.

5. We can improve and expand A&AP and domain models. We can reduce risks associated with changes in A&AP and domain models.

In ABD (Part 2) we use Test Driven Modelling (TDM) (Section 2.2). TDM utilizes Test Driven Development [24] methodology for modelling (analysing and implementing) of domains and for specifying user requirements. In TDM we first delimit the scope of phenomena to get a contextual model (according to ZF Row 1). We next specify requirements with unit tests. These unit tests form semantic models (ZF Row 2) of phenomena. By incremental specification and implementation of requirements we get step by step closer to logical (ZF Row 3) and physical (ZF Row 4) models. Logical models are models of phenomena in terms of interfaces (or class designs) and their relationships. Physical models are models of phenomena in some general purpose programming language. Physical models have to satisfy semantics (ZF, Row 2) specified by unit tests.

6. We can build different tools (generators of UI and other source artefacts, languages for end users to describe requirements, validation and verification tools for requirements and software, etc.) on top of these models. A&AP, domain models and associated tools form software factories. We can develop software factories so, that software end users can evolve software in an evolutionary way even at runtime by making changes only in the presentation or in the communication layers.

We see the P-systems ($\Pi = (O, C, \mu, \omega_1, \omega_2, \dots, \omega_m, R_1, R_2, \dots, R_m, i_0)$), described by dictionaries (O), catalysts (C), structure (μ), data ($\omega_1, \omega_2, \dots, \omega_m$) and rules ($R_1, R_2, \dots, R_m$), as a roadmap towards software factories and evolutionary information systems (Section 4.4). Proposed interpretation of P-systems is based on agent's metaphor. An autonomous agent (e.g. enterprise) is active in a domain (e.g. laboratory) and by communicating (e.g. sending and receiving documents) with an external environment processes (i.e. calculates) information (e.g. tests laboratory samples). This task will be for future study.

6.3 Future Work

Besides future improvements, developments and evaluations of ABD, A&APs, domain models (e.g. implementing laboratory automation patterns as described in Section 4.2.2.4) and developments of LIMS software towards LIMS software factory (as proposed in Section 4.4), one of the possible future tasks is to analyse and improve the degree of formality of the Test Driven Modelling (TDM) (Section 2.2) features and possibilities.

For example, according to Bjørner's domain analysis [26], the first narratives of quantity domain (Section 2.2.2) in RAISE specification language could be the following:

$\boldsymbol{type}$
$\quad Quantity, Measure, Unit, Real, String$
$\boldsymbol{value}$
$\quad Observe_Unit: Quantity -> \ Unit$
$\quad Observe_Amount: Quantity -> \ Real$
$\quad Observe_Measure: Unit -> \ Measure$
$\quad Observe_UnitName: Unit -> String$
$\quad Observe_UnitFactor: Unit -> Real$
$\quad Observe_MeasureName: Measure -> \ String$
$\boldsymbol{type}$
$\quad OpKind \ == \ arithmetic|comparing|rounding|converting$

In TDM, the same is specified by using unit tests as shown and explained in Section 2.2. However, is there a significant difference in the degree of formality between these two specifications? If there is, then which of them is more formal and if the TDM is less formal, then how can the formality of TDM be improved. In our understanding the real value of any formal method is validation of requirements and verification of software to increase the software dependability. We discussed this issue in Sections 2.4 and 5.1.11, but more mature and accurate research is needed.

The other future research goal can be developing domain and requirements specification languages with integrated TDM features on top of archetypes and archetype patterns (A&AP). Currently we use A&APs based DSLs, realized as APIs (or framework) and embedded into general purpose language (C#), for specifying domain models (clinical laboratory for example), as described in Section 2.3.

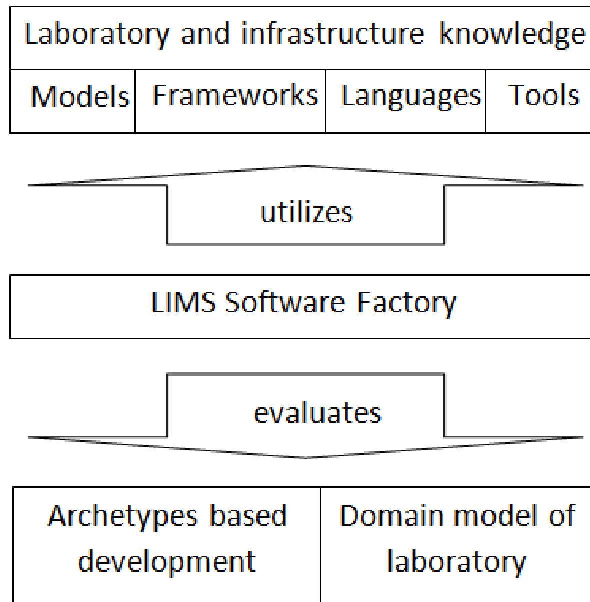

Figure 6-2: Towards LIMS Software Factory.

In design-time, using DSLs embedded into general purpose languages is good enough and in our understanding this technique is suitable for analysing domains. Unfortunately this technique does not work at run-time. Thus we need some languages and tools to describe user requirements and test scenarios at runtime as illustrated in Figure 1-1. It would be most beneficial, however, to specify requirements and test scenarios jointly as illustrated in Figure 6-1.This means that one specification results in two outputs: requirements for generating software and test scenarios for verifying these requirements and for validating generated software.

Domains and requirements specification languages research topic is closely related to the evolutionary self-development research topic we explained in Section 4.4. In this section we discussed possibilities to build evolutionary self-development information systems as P-systems [18] where domain model (clinical laboratory for example) concepts together with document domain concepts, agent domain concepts and calculus domain concepts are used as alphabet (language) elements of P-systems. We see this evolutionary self-development research topic as fundamental towards LIMS Software factory (Figure 6-2) and evolutionary information systems where end users, in collaboration with software developers, are able to evolve software in an evolutionary way according to changes in business processes, by making changes only in the presentation or in the communication layers.

REFERENCES

[1] ASTM., *E1578-06 Standard Guide for Laboratory Information Management Systems (LIMS).* s.l. : ASTM International, 2006.

[2] Piho, G., Tepandi, J. and Roost, M., "Archetypes Based Techniques for Modelling of Business Domains, Requirements and Software." Tallinn : s.n., 23-27 May 2011. 21st European Japanese Conference on Information Modelling and Knowledge Bases.

[3] Greenfield, J., et al., *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* s.l. : Wiley, 2004.

[4] Heitmeyer, C., "Managing Complexity in Software Development with Formally Based Tools." *Electronic Notes in Theoretical Computer Science (ENTCS).* December 2004, Vol. 108, pp. 11-19 . [doi>10.1016/j.entcs.2004.11.004] .

[5] Beck, K., *Extreme Programming Explained: Embrace Change.* s.l. : Addison-Wesley, 2000.

[6] Fowler, M., *Patterns of Enterprise Application Architecture.* Boston, MA : Addison-Wesley, 2003.

[7] Chappell, D., "Comparing .NET and Java: The View from 2006." Barcelona : Microsoft TechEd Developers, 2006.

[8] Fowler, Martin., "Refactoring." [Online] 22 08 2011. http://www.refactoring.com/.

[9] Microsoft., "Microsoft Presentation Foundation." [Online] [Cited: 22 08 2011.] http://windowsclient.net/.

[10] —. "Windows Communication Foundation." [Online] [Cited: 22 08 2011.] http://msdn.microsoft.com/en-us/library/ms731082.aspx.

[11] —. Microsoft BizTalk Server. [Online] http://www.microsoft.com/biztalk/en/us/default.aspx.

[12] Nilsson, J., *Applying Domain-Driven Design and Patterns.* Boston, MA : Addison-Wesley, 2006.

[13] JetBrains., "ReSharper 6." [Online] [Cited: 22 08 2011.] http://www.jetbrains.com/resharper/.

[14] Arlow, J. and Neustadt, I., *Enterprise Patterns and MDA: Building Better Software With Archetype Patterns and UML.* s.l. : Addisson-Wesly, 2003.

[15] Fowler, M., *Analysis Patterns: Reusable Object Models.* s.l. : Addison-Wesley, 2005.

[16] Hay, D. C., *Data Model Patterns, First Edition : A Metadata Map (The Morgan Kaufmann Series in Data Management Systems).* s.l. : Morgan Kaufmann, 2006.

[17] Silverston, L., *The Data Model Resource Book 1. A Library of Universal Data Models for All Enterprises.* s.l. : Wiley, 2001. Vol. 1.

[18] Paun, G., "Introduction to Membrane Computing." [Online] 2004. [Cited: 30 08 2011.] http://psystems.disco.unimib.it/download/MembIntro2004.pdf.

[19] Piho, G., Tepandi, J. and Roost, M., "The Zachman Framework with Archetypes and Archetype Patterns." [ed.] J. Barzdins and M. Kirikova. Riga, Latvia, Baltic DB&IS, July 5-7, 2010 : University of Latvia Press, 2010. Databases and Information Systems: Proceedings of the Ninth International Baltic Conference. pp. 455-570.

[20] Piho, G., et al., "Test Driven Domain Modelling." Opatia, Horvatia : s.n., 23-27 May 2011. 34th International Convention on Information and Communication Technology, Electronics and Microelectronics. accepted by MIPRO 2011.

[21] Bjørner, D., "Domain Theory: Practice and Theories (A Discussion of Possible Research Topics)." Macau SAR, China : The 4thInternational Colloquium on Theoretical Aspects of Computing - ICTAC, 2007.

[22] Bjørner, D., *Software Engineering, Vol. 1: Abstraction and Modelling.* Texts in Theoretical Computer Science, the EATCS Series. : Springer, 2006.

[23] Zachman, J. A., "A Framework for Information Systems Architecture." *IBM Systems Journal.* 1987, Vol. 26, 3.

[24] Beck, K., *Test-Driven Development: By Example.* Boston, MA : Addison-Wesley, 2003.

[25] Bjørner, D., *Software Engineering, Vol. 3: Domains, Requirements, and Software Design.* Texts in Theoretical Computer Science, the EATCS Series. : Springer, 2006.

[26] —. *Documents: A Domain Analysis.* [Online] http://www2.imm.dtu.dk/~db/5lectures/document.pdf.

[27] Wikipedia., Reflection (computer programming). [Online] http://en.wikipedia.org/wiki/Reflection_(computer_programming)#cite_note-0.

[28] The RAISE Language Group., *THE RAISE Specification Language: The BCS Practioner Series.* s.l. : Prentice Hall, 1992. ISBN 0-13-752833-7.

[29] The RAISE Method Group., *The RAISE Development Method: BCS Practitioner Series.* s.l. : Prentice Hall, 1995. ISBN 0-13-752700-4.

[30] Woodcock, J.C.P. and Davies, J., *Using Z: Specification, Proof and Refinement.* Prentice Hall : s.n., 1996.

[31] Abrial, J.-R., *The B Book: Assigning Programs to Meanings.* UK : Cambridge University Press, 1996.

[32] Bjørner, D. and Jones, C., [ed.], "The Vienna Development Method: The Meta-Language." LNCS, s.l. : Springer, 1978, Vol. 61.

[33] Hoare, C. A. R., "An axiomatic basis for computer programming." *Communications of the ACM.* October 1969, Vol. 12, 10, pp. 576–580,583. http://sunnyday.mit.edu/16.355/Hoare-CACM-69.pdf.

[34] Beydoun, G., et al., "FAML: A Generic Metamodel for MAS Development." *IEEE Transactions on Software Engineering.* November/December 2009, Vol. 35, 6, pp. 841-863.

[35] Hay, D. C., *Data Model Patterns: Conventions of Thought.* s.l. : Dorset House Publishing, 1996.

[36] Silverston, L., *The Data Model Resource Book 2. A Library of Data Models for Specific Industries.* s.l. : Wiley, 2001.

[37] Halevy, A.Y., "Why Your Data Won't Mix: Semantic Heterogeneity." *Queue.* 2005, Vol. 3, 8, pp. 50-58.

[38] Wache, H., "Solving Semantic Interoperability Conflicts." Brussel : s.n., 02 February 2009. Methodology workshop: Modelling eGovernment entities Methodologies and Experiences under review. http://www.semic.eu/semic/view/documents/presentations/SEMIC-EU-Methodology-Wache-Solving-Conflicts.pdf;jsessionid=A065C6F205788A2F1E4E3366E9F24D10.

[39] Heard, S., et al., "Templates and Archetypes: how do we know what we are talking about?" [Online] 2003. [Cited: 21 06 2010.] http://www.openehr.org/publications/archetypes/templates_and_archetypes_heard_et_al.pdf.

[40] Bjørner, D., *Software Engineering, Vol. 2: Specifications of Systems and Languages.* Texts in Theoretical Computer Science, the EATCS Series. : Springer, 2006.

[41] Piho, G., "Archetype patterns based method of prescribing enterprise software requirements." [toim.] D. Čišic, et al. Rijeka (Croatia) : Studio Hofbauer, 2007. MIPRO 2007 proceedings: MIPRO 2007, Opatia (Croatia), May 21-25. Kd. Business Intelligence Systems, lk 236 - 241.

[42] Piho, G., "Towards archetypes based domain model of clinical laboratory." [ed.] E. Troubitsyna. Turku, Finland : TUCS (Turku Centre for Computer Science) General Publications, 2008. Proceedings of Doctoral Symposium held in conjunction with Formal Methods 2008. Vol. 48, pp. 33 - 42.

[43] Bendy, G.Z. and Harris, S.R., "The Systematic-Structural Theory of Activity: Applications to the Study of Human Work." *Mind, Culture, and Activity.* 2005, Vol. 12, 2, pp. 128-147.

[44] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA : Addison-Wesley, 1995.

[45] Lindqvist, A. and Christensen, B., *Abstract Document System - instantiated for patient medical records.* The Department of Computer Science and Engineering, Institute of Informatics and Mathematical Modelling. Copenhagen : Technical University of Denmark, 2004. Master thesis. IMM-THESIS-2004-23.

[46] Wirfs-Brock, R. J., "Looking for Powerful Abstractions." *IEEE Software.* Jan/Feb 2006, Vol. 23, 1, pp. 13-15.

[47] Martin, R. C., *Agile Software Development: Principles, Patterns, and Practices.* New Yersey : Prentice Hall, 2002.

[48] Avižienis, A., Laprie, J.-C. and Randell, B., *Fundamental Concepts of Dependability. Research Report N01145.* s.l. : LAAS-CNRS, April 2001.

[49] Piho, G., et al., "From archetypes-based domain model of clinical laboratory to LIMS software." Opatia, Croatoa, 24-28 May 2010 : IEEE, 2010. MIPRO, 2010 Proceedings of the 33rd International Convention. Vol. Digital Economy, pp. 1179-1184. ISBN: 978-1-4244-7763-0.

[50] Piho, G., et al., "From Archetypes Based Domain Model via Requirements to Software: Exemplified by LIMS Software Factory." Opatia, Horvatia : s.n., 2011. 34th International Convention on Information and Communication Technolology Technology, Electronics and Microelectronic (MIPRO, May 23-27, 2011).

[51] ASTM., *E1578-93 (Reapproved 1999) Standard Guide for Laboratory Information Management Systems (LIMS).* s.l. : ASTM International, 1999.

[52] Science 2020., "Towards 2020 Science." [Online] 2005. http://research.microsoft.com/towards2020science/background_overview.htm.

[53] ASTM., *E1381-02 Standard Specification for Low-Level Protocol to Transfer Messages Between Clinical Laboratory Instruments and Computer Systems (Withdrawn).* s.l. : ASTM International, 2002.

[54] —. *E1394-97 Standard Specification for Transferring Information Between Clinical Instruments and Computer Systems (Withdrawn).* s.l. : ASTM International, 2002.

[55] HL7., HL7 Standards. [Online] http://www.hl7.org/.

[56] Piho, G., Tepandi, J. and Roost, M., "Evaluation of Archetypes Based Development." [ed.] J. Barzdins and M. Kirikova. *Frontiers in Artificial*

Intelligence and Applications. Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, 2011, Vol. 224, pp. 283 - 295.

[57] Hruby, P., *Model-Driven Design Using Business Patterns.* Berlin Heidelberg : Springer-Verlag, 2006.

[58] The World Bank., *World Development report 1994: Infrastructure for Development.* s.l. : Oxford Universaity Press, 1994. http://www-wds.worldbank.org/external/default/WDSContentServer/WDSP/IB/1994/06/01/000009265_3970716142907/Rendered/PDF/multi0page.pdf.

[59] Cocburn, A., *Agile Software Development.* Boston, MA : Addisson-Wesley, 2002.

[60] Piho, G., *Introducing XP-methodology in a Small Estonian Software Company. Master thesis (in Estonian).* Tallinn : Tallinn University (former Tallinn Pedagogical Univerity), 2003. http://www.cs.tlu.ee/instituut/opilaste_tood/magistri_tood/2003_sugis/Gunnar_Piho/Gunnar_Piho_Mag_Too.pdf.

[61] Charette, R.N., "Why Software Fails." *IEEE Spectrum.* Sept. 2005. http://www.spectrum.ieee.org/sep05/1685.

[62] CMMI product team., *CMMI for Development, Version 1.2, CMU/SEI-2006-TR-008.* s.l. : Software Engineering Institute, 2007. http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tr008.pdf.

[63] Douglas, S., et al., *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects.* s.l. : Wiley, 2000. Vol. 2.

[64] Jackson, M. and Zave, P., "Deriving specifications from requirements: an example." New York : ACM, 1995. Proceedings of the 17th international conference on Software engineering (ICSE '95). pp. 15-24.

[65] Paulk, M. C., "Extreme Programming from a CMM Perspective." Raleigh, NC 23-25 July : Paper for XP Universe, 2001. http://www.sei.cmu.edu/cmm/papers/xp-cmm-paper.pdf.

[66] Evans, E., *Domain-Driven Design: Talking Complexity in the Heart of Software.* Boston, MA : Addison-Wesley, 2004.

[67] Sysmex., Multilab - Laboratory Information Management System. *www.sysmex-europe.com.* [Online] http://www.sysmex-europe.com/Products/Clinical%20Information%20Solutions/LIS/multilab/.

[68] MesHR., Informatička i upravljačka rješenja u laboratoriju. *MulitLab®.* [Online] http://www.meshr.hr/proizvodi/info_rjesenja/#2.

[69] Alhir, S. S., "Understanding the Model Driven Architecture (MDA)." [Online] 2003. [Cited: 29 08 2011.] http://www.methodsandtools.com/archive/archive.php?id=5.

[70] openEHR., "openEHR Archetypes." [Online] 2007. [Cited: 28 05 2009.] http://www.openehr.org/svn/knowledge/archetypes/dev/index.html. Release 1.0.1.

7 APPENDICES

7.1 Order Lifecycle

There are a number of different events (Figure 3-23) which we have to audit trail during the order lifecycle. These events depend on the order status. One class of events are lifecycle events ($ILifecycleEvent$) which change the status of the order. An order that has no lifecycle events is in the initializing state ($IInitializingOrder$). The *open event* ($IOpenEvent$) can occur only if the order is in the *initializing state*. After the *open event*, the order will be either in $IOpenPurchaseOrder$ or in $IOpenSalesOrder$ which are both sub-states of the *open state* ($IOpenOrder$). We separated abstract $IOpenOrder$ into two concrete sub-types $IOpenPurchaseOrder$ and $IOpenSalesOrde$, because in these sub-types different *payment*, *despatch* and *receipt* events can occur.

The order in open state ($IOpenSatus$) can be closed (sale is completed) or cancelled. If all the sales transactions (payments, delivery) have been completed, the order will be transformed into the closed state ($IClosedOrder$) by the close event ($ICloseEvent$). In some situations (depending on terms and conditions) the sale can be cancelled. If this is so, by using cancel event ($ICancelEvent$) the order will be transformed into the *cancelling* station ($ICancellingStatus$). After all the loose ends (return of items and refunds) are completed, the order will be transferred from the cancelling station by *close event* ($ICloseEvent$) into the cancelled state ($ICancelledStatus$).

There are three sub-types ($IAmendOrderLine$, $IAmendRelatedParty$ and $IAmendCondtions$) of the abstract *amend event* ($IAmendEvent$). All of these sub-types can only occur, when the order is in the open state ($IOpenOrder$). With the *amend order line* all the changes in order lines can be audit trailed. As mentioned earlier, all our archetypes are read only, and therefore it is impossible to change any part of any archetype without a clear request. Therefore, instead of changing some properties in the order line, the old order line is marked as cancelled and a new order line is created. The *amend order line event* points to both, newly created as well as cancelled, order lines. The *amend order line event* ($IAmendOrderLine$) also points to the returned items, if such items exist. The *amended order line identifier* ($AmendedOrderLineIdentifier$) is used to interconnect amended order lines. Similarly to the *amend order line event*, with *amend related party event* ($IAmendRelatedParty$), all the changes in *related parties* are audit trailed. Like *amend order line event* pointing to the cancelled and newly created order lines, so *amend related party event* points to the cancelled and newly created *parties*. The third amend event ($IAmendConditions$) tracks changes in sale conditions. The logic behind $IAmendConditions$ is exactly the same as the logic behind *amend order line* and *amend related party* events.

The *discount event* ($IDiscountEvent$) can occur only when the order is in the open state ($IOpenOrder$). There can be different discounting reasons.

Discounts on orders may be granted for high-value customers, for customers who raised the order by the internet, for customers in specific geographic regions and etc. There are two different discount types ($IPercentageDiscount$and $IMonetaryDiscount$), managed by order manager ($IOrderManager$). The discount type contains a set of rules (see Section 3.5.3) that describe the conditions ($Eligibility$) under which a particular discount may be applied. The discount event ($IDiscountEvent$) points to the discount which calculates the discounted price for the order.

The despatch event archetype ($IDespatchEvent$) is an event that can be applied only to a sales order when the sales order is in the open state ($IOpenSalesOrder$). The despatch event records goods or services sent to the delivery receiver. Despatch event records properties such as the date ($Date$) on which the despatch was made, the unique despatch identifier ($Despatch$) and shipment instructions ($Instructions$). It also points to the despatch line ($IDespatchLine$) which records the amount of despatched items ($IDespatchedItem.Amount$) for the particular order line ($IOrderLine$).

The delivery receiver may reject some of the despatched items. Rejected items will be recorded by the rejected items ($IRejectedItem$) archetype. Items received by delivery are recorded by the receipt event ($IReceiptEvent$) archetype. The receipt event can be applied to a purchase order only when the purchase order is in open state ($IOpenPurchaseOrder$). Receipt event contains the *delivery identifier* that links to a specific delivery of goods or services and the *delivery date* on which the delivery was received. Similarly to the delivery event the receipt event points both to the receipt line ($IReceiptLine$) which records the amount of received items ($IReceiptItem.Amount$) of the particular order line ($IOrderLine$) and to the rejected items ($IRejectedItem$), in case some items are rejected.

The order payment ($IOrderPayment$) archetype represents a *payment* ($IPayment$) made or accepted [14 p. 343]. Order payment has attributes *from account* and *to account* and is used for recording respective bank accounts. The *payment* ($IPayment$) archetype is described in the money archetype pattern (Section 3.5.2). Six accounting events ($IAccountingEvent$) can be applied to an open order. Three of these events - send invoice ($ISendInvoice$), accept payment ($IAcceptPayment$) and make refund ($IMakeRefund$) - can be applied to a sales order in the open state ($IOpenSales$). The other three of these events - accept invoice ($IAcceptInvoice$), make payment ($IMakePayment$) and accept refund ($IAcceptRefund$) - can be applied to a purchase order in the open state ($IOpenPurchase$).

7.2 Using the Business Process Archetype Pattern

7.2.1 Communication and CRM

When we have a common process archetype pattern (Figure 3-25), the communication and the CRM archetype pattern from Arlow and Neustadt [14

pp. 187-201] can be modelled as a special case of process archetype pattern as shown in Figure 7-1.

A communication process manager ($IProcessManager$) with a communication manager ($ICommunicationManager$) type ($IProcessManagerType$) manages all customer communications as a set of processes ($IProcess$) with $ICommunicationCase$ process type ($IProcessType$).

Figure 7-1: Logical Model of Communication

Communication ($ICommunication$) is a task type ($ITaskType$). A task with a task type $ICommunication$ captures details of communications between two parties. For simplicity, communication always originates from one provider role (e.g. customer service representative) and is received by one consumer role (e.g. either customer or customer agent – someone who represents a customer) [14 p. 198]. However, many other party roles may be participants in the communication in question.

Each such communication (task, party relationship) can have attributes such as *date sent* (business has initiated the communication), *date received* (customer has initiated the communication), *content* (summary of conversation), *from address* (address where the communication originated), *to address* (address where the communication was received), and etc. See Section 3.5.4 for the custom attributes system we use in our archetypes.

Communication routing ($ICommunicationRouting$) is a special case of task routing that represents a handover between customer service representatives. A customer service representative is a party role played by someone who acts on behalf of, and with the authorization and authority of the customer service department [14 p. 197].

The communication case ($ICommunicationCase$) is a type of process, which holds a collection of all communications (task with type of $ICommunication$) about a specific topic related to a specific customer (the party role type). The communication process can have the following attributes: *title* (summarizes the nature of the communication case), *description* (short description of the communication case), *raised by* (pointer to the party role that raised the case), *start date*, *end date*, *priority* and so on. The communication thread ($ICommunicationThread$) represents a sequence of communications about a particular topic.

For each task (Figure 3-26), the communication may also be a source of zero to many actions and any action may have zero or more outcomes.

7.2.2 Reporting

Similarly to communication and CRM (Appendix 7.2) we can use the process archetype pattern (Figure 3-25) to model reporting's (Figure 7-2).

The reporting process manager ($IProcessManager$) with a process manager type ($IReportManager$) manages all reports as a set of processes ($IProcess$) with a $IReportCase$ process type ($IProcessType$).

Report ($IReport$) is a task type ($ITaskType$) which captures details of reports between two parties. For simplicity, a report always originates from one report provider role (e.g. subordinate) and is received by one consumer role (e.g. manager). However, many other party roles may also be participants.

Each such report (task, party relationship) can have attributes like *date sent* (the date and time when the report was initiated), *date received* (the date and time when the report was received), *content* (summary of the report), *from address* (address where the report was originated), *to address* (address where the report was received), and etc.

Report routing ($IReportRouting$) is a special case of task routing that represents a handover either between subordinates or between managers.

The report case ($IReportCase$) is a type of process, which holds a collection of all reports (task with type of $IReport$) about a specific topic related to a specific subordinate (the party role type). The routing process can have the following attributes: *title* (summarizes the nature of the reporting case), *description* (short description of the reporting case), *raised by* (pointer to the party role that raised the case), *start date*, *end date*, *priority* and so on.

The reporting thread ($IReportingThread$) represents a sequence of reports about a particular topic. As common for each task (Figure 3-26), the report may also be a source of zero to many actions and any action may have zero or more outcomes.

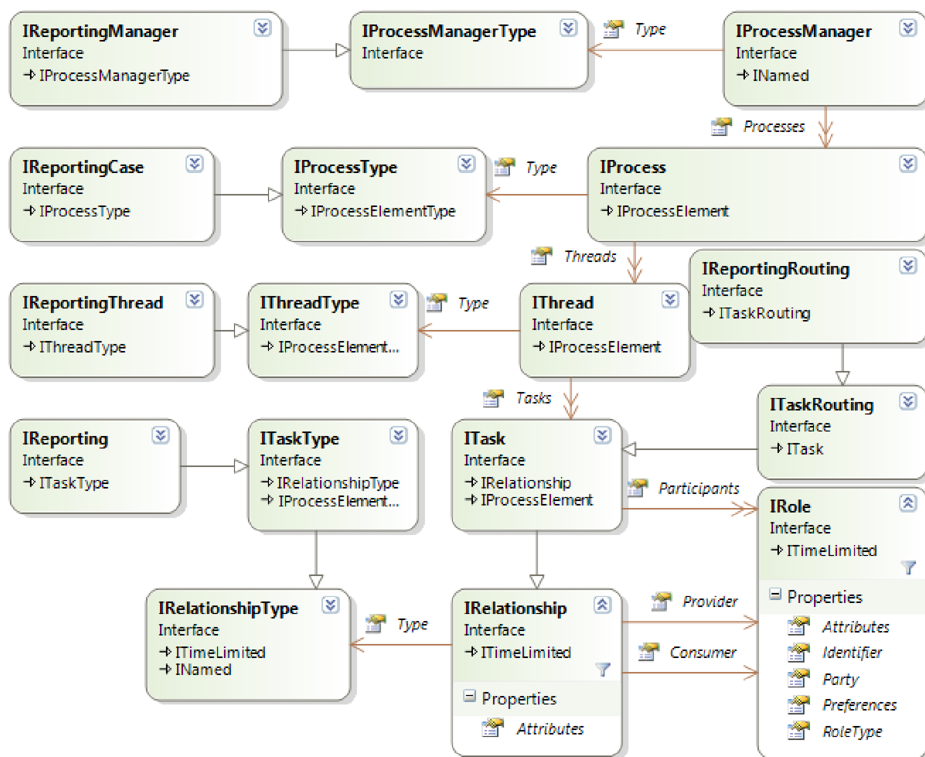

Figure 7-2: Logical Model of Reporting

7.2.3 Payments

In this section we are preparing for sales (Appendix 7.2.5) and purchase (Appendix 7.2.4) processes by modelling different payment strategies, that have been described and modelled by using activity diagrams by Arlow and Neustadt [14 pp. 346-348]. When Arlow and Neustadt models are documentation artefacts, then our models are source artefacts (as normal for software factories) and are all concretizations of the process archetype pattern (Figure 3-25) described in Section 3.5.9.

In Figure 7-3 the major action types (*IActionType*) and major outcome types (*IOutcomeType*) of payment process is illustrated. The "major" means that an addition to accepting activities and outcomes (e.g.*IAcceptPayment*, *IPaymentIsAccepted*), there can be also declining activities (like *IDeclinePayment*, *IPaymentIsDeclined*).

Each outcome from sales (Appendix 7.2.5) and purchase (Appendix 7.2.4) processes is related through an order event (*IOrderEvent*) with order (*IOrder*) or order line (*IOrderLine*) archetypes. This is illustrated for example in Figure 7-20. As each order is related through an order line and an inventory entry also with inventory, we can say (at least in buying and selling context), that order

and inventory archetype patterns can be used for recording (logging) of different business events as we already stated in Section 2.1.1 when we discussed how Zachman Framework columns and archetype patterns are related.

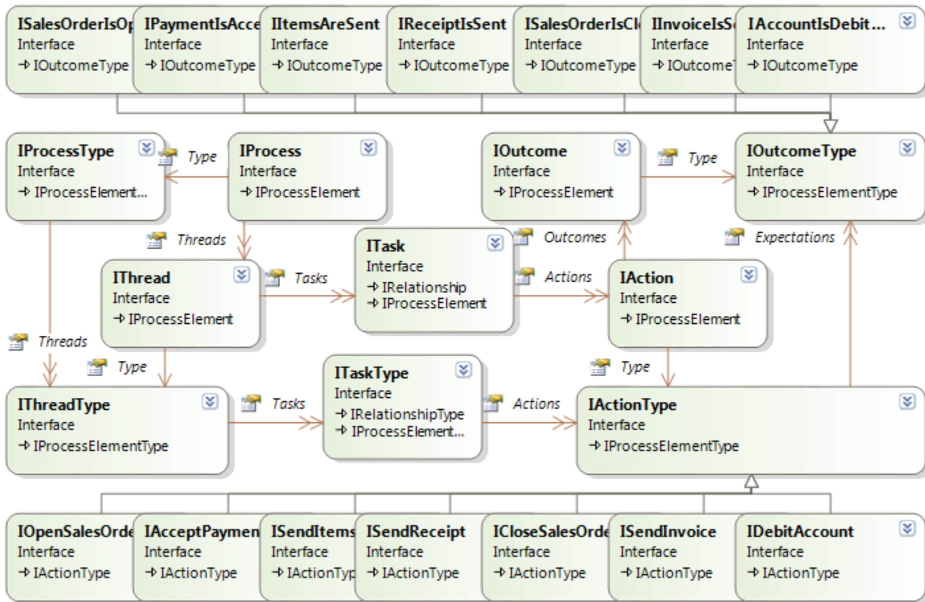

Figure 7-3: Logical Model of Payment Actions and Outcomes

The major activities of payment processes are *open sales order* (*IOpenSalesOrder*), *accept payment* (*IAcceptPayment*), *send items* (*ISendItems*), *send receipt* (*ISendReceipt*), *close sales order* (*ICloseSalesOrder*), *send invoice* (*ISendInvoice*) and *debit account* (*IDebitAccount*). It follows then that the major outcomes of payment processes are *sales order is open* (*ISalesOrderIsOpen*), *payment is accepted* (*IPaymentIsAccepted*), *items are sent* (*IItemsAreSent*), *receipt is sent* (*IReceiptIsSent*), *sales order is closed* (*ISalesOrderIsClosed*), *invoice is sent* (*IInvoiceIsSent*) and *account is debited* (*IAccountIsDebited*).

The order of these activities and rules (each process element and process element type can be ordered as well as attributed by rules, Figure 3-26) for executing these activities depend on the sales type. Depending on companies selling strategies, the type of sale can be prepaid (*IPrepaidSale*, Figure 7-4), credited (*ICreditedSale*, Figure 7-5), invoiced (*IInvoicedSale*, Figure 7-6) and debited (*IDebitedSale*, Figure 7-7).

In a prepaid sale (Figure 7-4), no deliveries will be despatched (*IDespatchDelivery*) before the full payment. Prepaid sale is common case between individuals and a business or when the customer is unknown and therefore has no relationship of trust with the business [14 p. 346]. Prepaid sale is initialised by a buyer by sending a purchase order along with full payment.

Receipt of a purchase order ($IReceivePurchaseOrder$) activates a task with $IOpenSalesOrder$ action, then receiving a payment ($IReceivePayment$) the vendor normally activates a task with accept payment ($IAcceptPayment$) action.

The task despatch delivery ($IDespatchDelivery$), with the sequenced activities send items ($ISendItems$), send receipt ($ISendReceipt$) and close sales order ($ICloseSalesOrder$) starts (rule based) when sales order is open ($ISalesOrderIsOpen$) and payment is accepted ($IPaymentIsAccepted$) outcomes have been achieved.

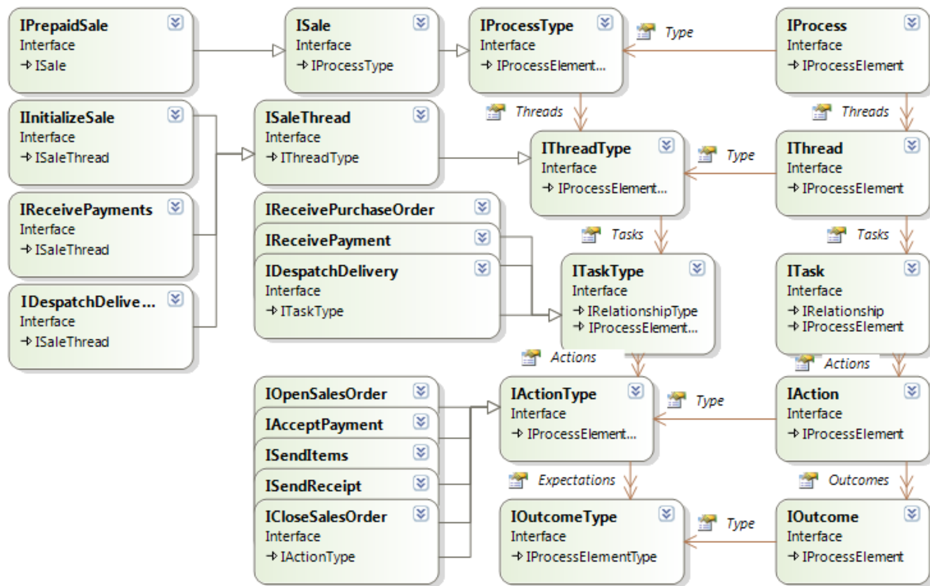

Figure 7-4: Logical Model of Prepayment

Although the simplest prepaid sale processes can be modelled without sale threads ($IInitializeSale$, $IReceivePeyments$ and $IDespatchDeliveries$), we retain them for reasons of universality (buyer can make more than one payments and items can be delivered to different receivers) and for compatibility (with other sale types).

The credited sale ($ICreditedSale$, Figure 7-5) differs from prepaid sale ($IPrepaidSale$, Figure 7-4) in that deliveries will be despatched directly after receiving a purchase order from buyer. Essentially the task despatch delivery ($IDespatchDelivery$), with sequenced activities send items ($ISendItems$) and send invoice ($ISendInvoice$) commences (rule based) when a sales order is open ($ISalesOrderIsOpen$). Again, the deliveries can be despatched to many different receivers in different deliveries. The receive payments ($IReceivePayments$) thread with possible accept payment ($IAcceptPayment$), send receipt ($ISendReceipt$) and close sales order ($ICloseSalesOrder$) actions commences when the first payment is received

($IReceivePayment$ task) by vendor and ends when the full payment according to purchased items is done. Many different payments are possible and naturally the actions of send receipt ($ISendReceipt$) and close sales order ($ICloseSalesOrder$) can be activated (rule based) when all the required payments (according to invoice) are accepted.

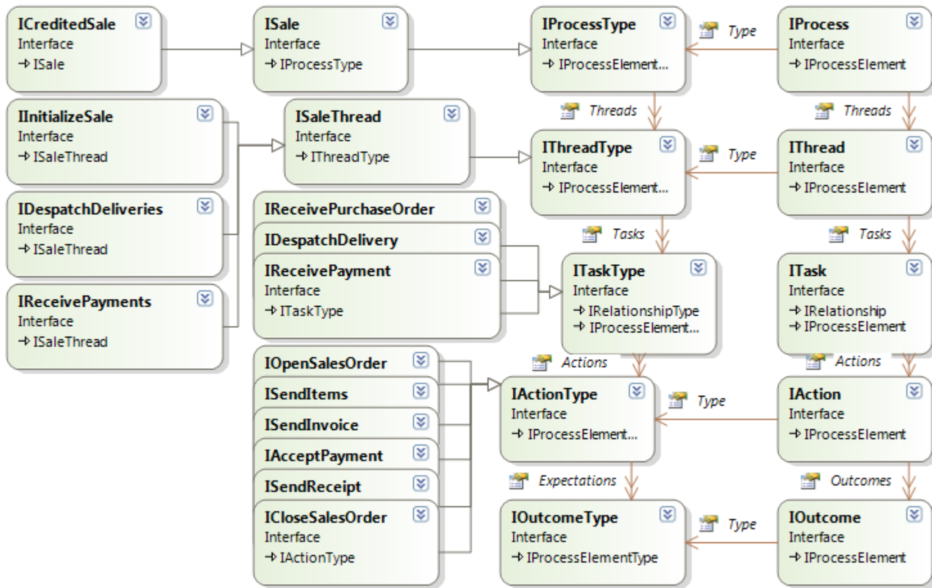

Figure 7-5: Logical Model of Credited Payment

In an invoiced sale ($IInvoicedSale$, Figure 7-6), the buyer pays after the purchase order has been received ($IReceivePurchaseOrder$) by the vendor and in advance of receipt of the goods [14 p. 347]. The $IReceivePurchaseOrder$ task is the only task in the sale initialise ($IInitializeThread$) thread but includes two actions ($IOpenSalesOrder, ISendInvoice$).

After the full payment from buyer is received ($IReceivePayments$ thread with one or more $IReceivePayment$ tasks with $IAcceptPayment$ activity), the vendor delivers the products (goods or services) to the *delivery receivers* within an agreed time period. The despatch deliveries ($IDespatchDeliveries$) thread consists one or more despatch delivery ($IDespatchDelivery$) tasks with at least one $ISendItems$ activity. The last (or the only one) $IDespatchDelivery$ tasks includes two further activities ($ISendReceipt$ and $ICloseSalesOrder$).

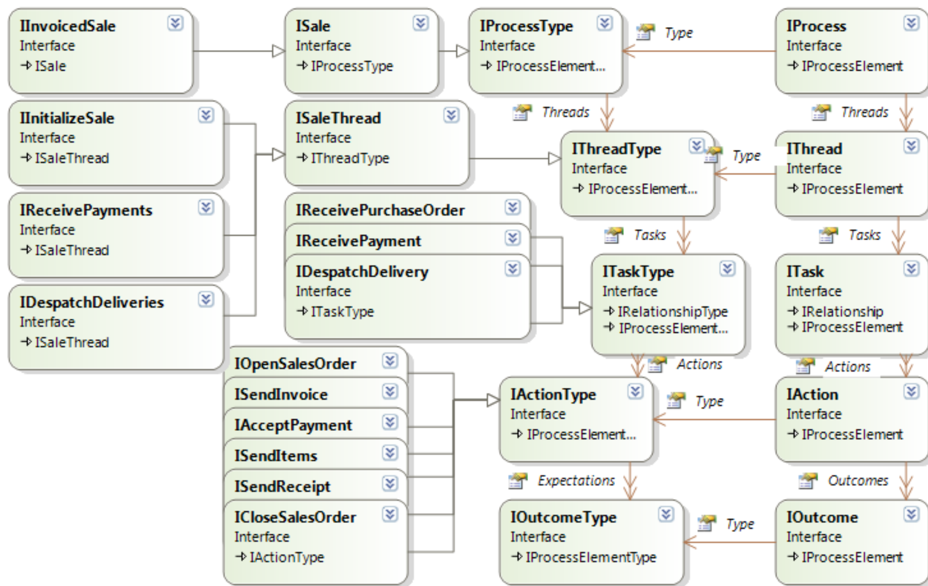

Figure 7-6: Logical Model of Invoiced Payments

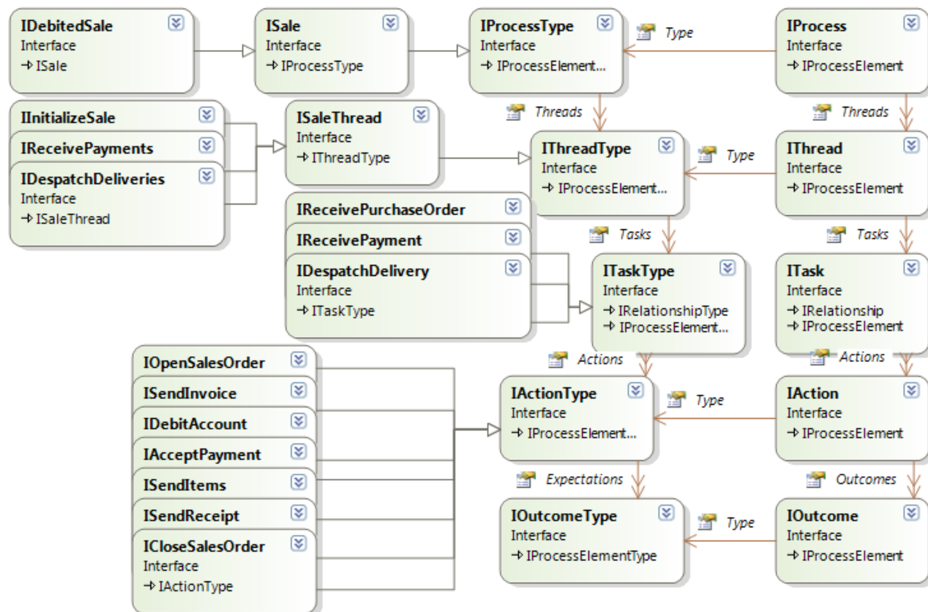

Figure 7-7: Logical Model of Debited Payments

A debited sale (*IDebitedSale*, Figure 7-7) occurs in both B2B (business – to - business) transactions and in individual-to-business transactions, when the individual has an account with the business [14 p. 348].

Receiving a purchase order (*IReceivePurchaseOrder*) activates a task with *IOpenSalesOrder* action followed by send invoice (*ISendInvoice*) and debit account (*IDebitAccount*) activities. The receiving of a payment (*IReceivePayment*) is still the task of separate *IReceivePayments* thread and normally with accept payment (*IAcceptPayment*) action.

The despatch delivery (*IDespatchDelivery*) task, with sequenced activities send items (*ISendItems*), send receipt (*ISendReceipt*) and close sales order (*ICloseSalesOrder*) commences (rule based) when sales order is open (*ISalesOrderIsOpen*) and payment is accepted (*IPaymentIsAccepted*) outcomes have been achieved.

7.2.4 Purchases

As there are four possible payment methods, we also have four possible purchases. These are *ICreditedPurchase*, *IDebetdPurchase*, *IInvoicesPurchase* and *IPrepayedPurchase* (Figure 7-8).

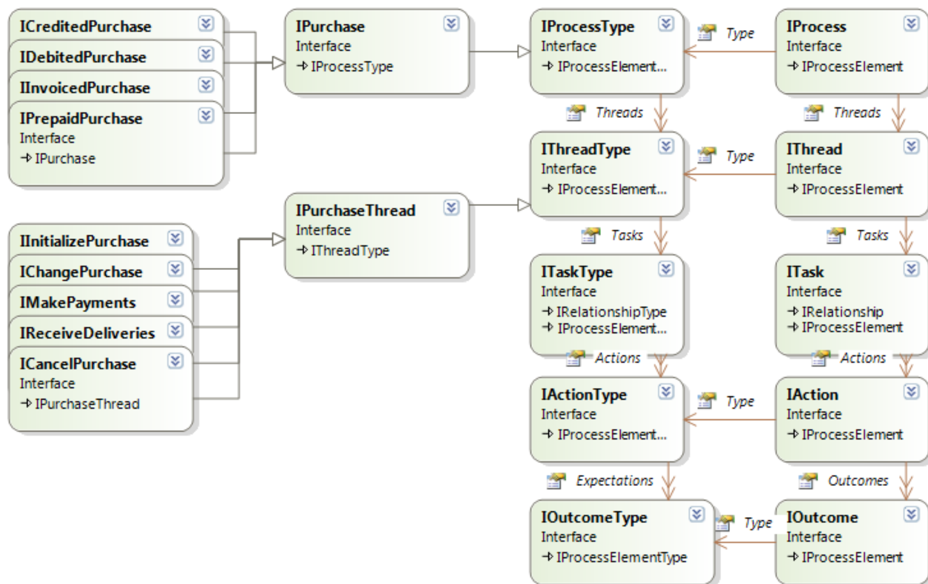

Figure 7-8: Logical Model of Purchases

All of these purchases include *initialize* (*IInitializePurchase*, Figure 7-9), *change* (*IChangePurchase*, Figure 7-11), *make payments* (*IMakePayments*, Figure 7-12), *receive deliveries* (*IReceiveDeliveries*, Figure 7-13) and *cancel* (*ICancelPurchase*, Figure 7-10) threads. Although for some purchase types, these threads can be firmly related to each other and can begin immediately after one has finished (for example in case of *IPreparedPurchase*, where the

make payments starts immediately after the purchase order is sent), we still keep these threads separately.

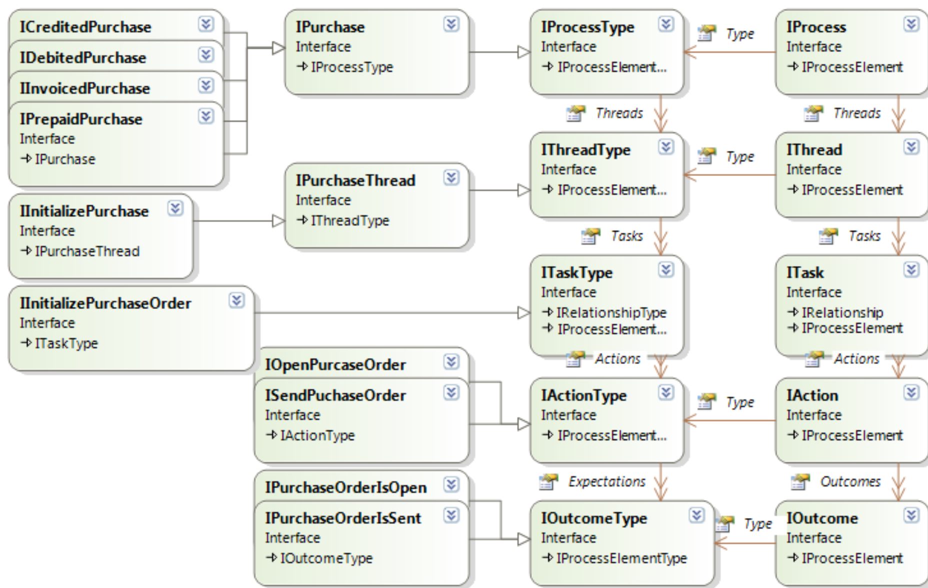

Figure 7-9: Logical Model of Purchase Initialization

Purchase initialization thread has one task with *IOpenPurchaseOrder* and *ISendPurchaseOrder* activities. The *cancel* purchase thread (*ICancelPurchase*, Figure 7-10) can include four tasks – initialize purchase decline, receive sales decline, return purchases, and receive refund. Both vendor and buyer have rights to initialize cancellation. Buyer commences the cancellation (*ICancelPurchase*) by sending a decline (*ISendDecline* action of *IInitializePurchaseDecline* task) to the vendor. When the vendor accepts decline (see sales cancellation Figure 7-16), then delivered purchases (if any) should be returned (*IReturnPurchases*) and any payments received should be refunded (*IReceiveRefund*). Different rules and rights can and should be followed by both sides, when cancelling purchases.

Similarly both parties can also initialize purchase amendments (*IChangePurcahse*, Figure 7-11). The purchase change thread has the same (or similar) tasks and activities as the cancel purchase thread has. When both parties have accepted amendments (see also sales change, Figure 7-17), then any delivered purchases should be returned (*IReturnPurchases*) and any payments received should be refunded (*IReceiveRefund*).

Figure 7-10: Logical Model of Purchase Cancellation

Figure 7-11: Logical Model of Purchase Change

Figure 7-12: Logical Model of Make Payments

Figure 7-13: Logical Model of Receive Deliveries

We already have discussed payments in Section 7.2.3. The make payments thread with receive invoice and initialize payment tasks is illustrated in Figure 7-12. Figure 7-13 illustrates the receive deliveries thread. This thread consists of one or more tasks $IReceiveDelivery$. The whole delivery can be accepted or declined and each separate item within the delivery can be accepted or rejected.

7.2.5 Sales

As purchases and sales are interrelated we also have four possible sales – $ICreditedSale$, $IDebetedSale$, $IInvoicedSale$ and $IPrepaidSale$ (Figure

7-14). All of these sales include initialize sale, change sale, receive payments, despatch deliveries and cancel sale threads. The sale initialization thread ($IInitializeSale$, Figure 7-15) has only one task ($IReceivePurchaseOrder$) with two activities $IOpenSalesOrder$ and $IDeclinePurchase$.

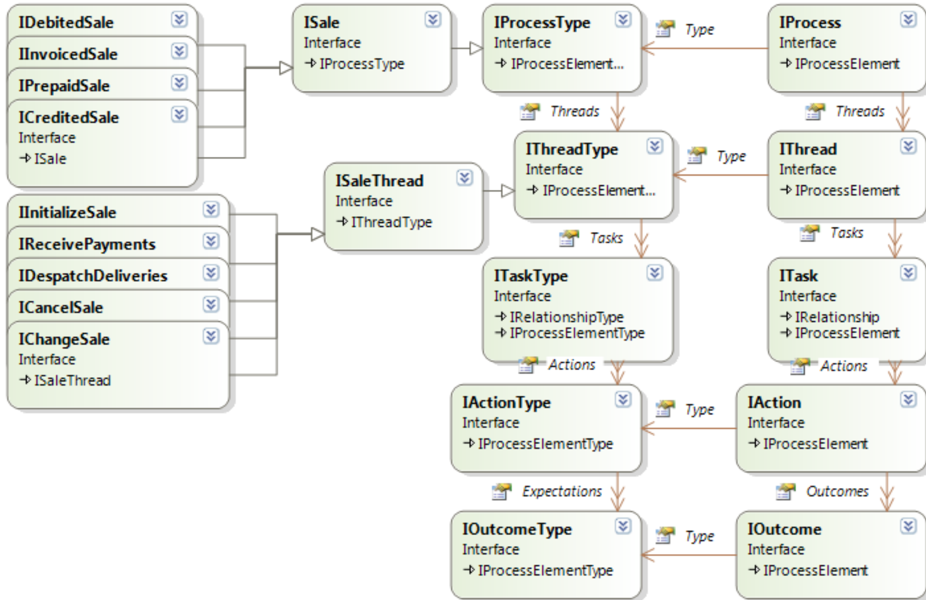

Figure 7-14: Logical Model of Sales

The cancel sale thread ($ICancelSale$, Figure 7-16) can include four tasks – receive purchase decline, initialize sales decline, receive purchased items, and refund. As was the case for cancellation, both parties can also initialize sale amendments ($IChangeSale$, Figure 7-17). The sale change thread has same (or similar) tasks and activities as the cancel sale thread. When both parties have accepted the amendments (see also purchase change, Figure 7-11), then delivered purchases (if any) should be returned ($IReceivePurchases$) and any payments received should be refunded ($IRefund$). The receive payments thread with one or more receive payment task and accept or decline payment activities is illustrated in Figure 7-18. Figure 7-19 illustrates the despatch deliveries ($IDespatchDeliveries$) thread. This thread consists of one or more tasks $IDespatchDelivery$. One delivery tasks includes three action types ($IPrepareItem$, $ICreateShipment$, $IDespatchShipment$) with their corresponding outcomes. The $IPrepareItem$ action can (but must not) be generated for every item in delivery.

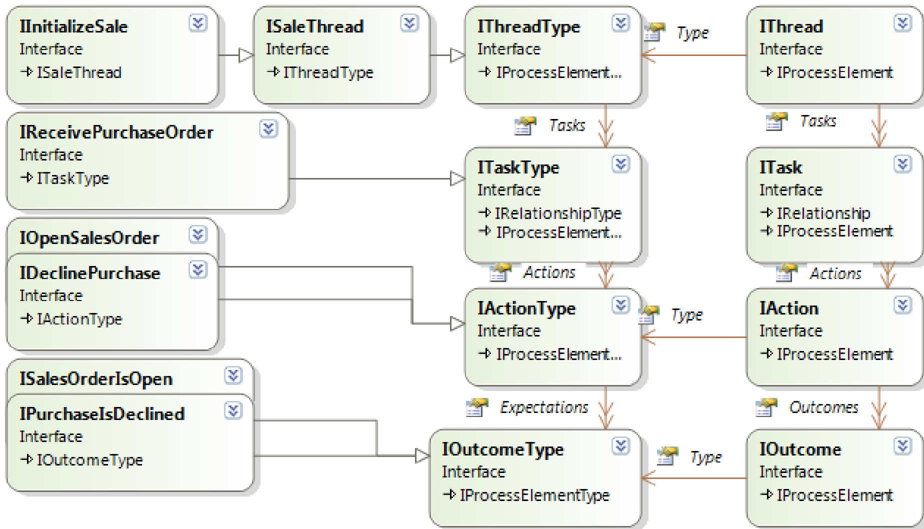

Figure 7-15: Logical Model of Sales Initialization

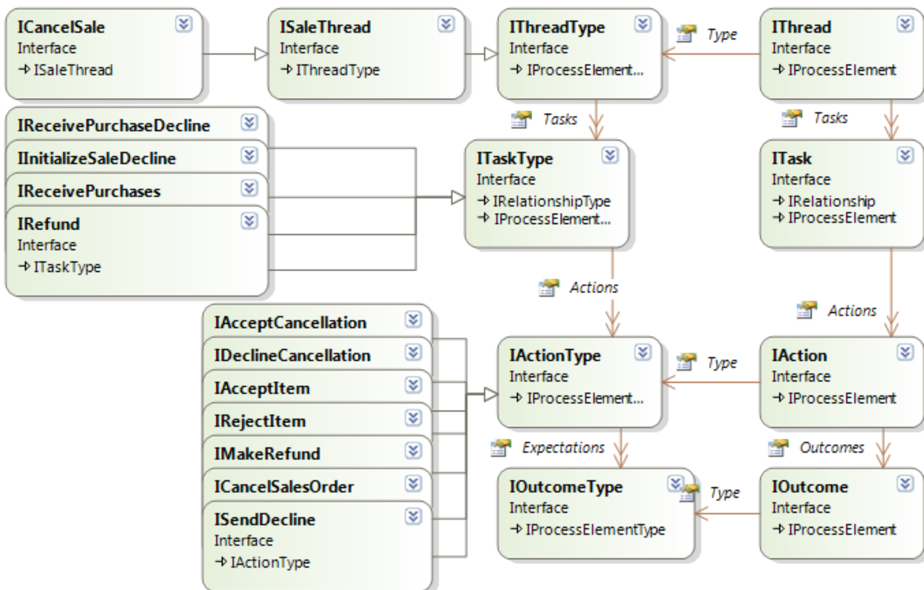

Figure 7-16: Logical Model of Sales Cancellation

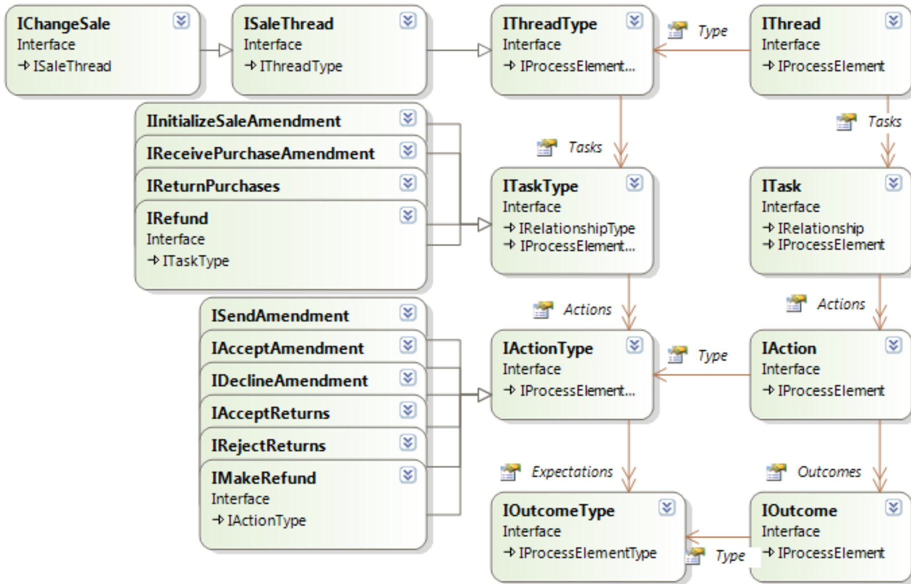

Figure 7-17: Logical Model of Sales Change

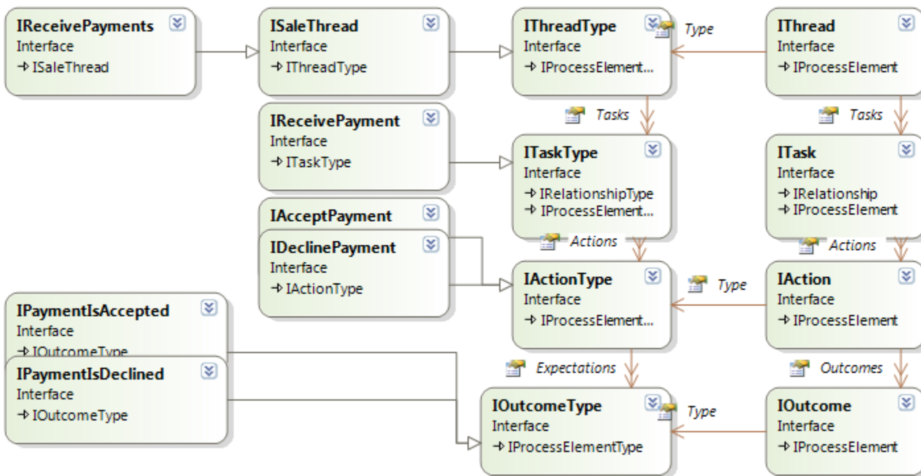

Figure 7-18: Logical model of Receive Payments

Figure 7-19: Logical Model of Despatch Items

Figure 7-20: Logical Model of Sales, Purchases and Order Events

7.3 Elulugu

Nimi Gunnar Piho

Telefon +3725111236

E-kiri gunnar.piho@computer.org

Haridustee

2005-2011 **Tallinna Tehnikaülikool**, Informaatikainstituut, **doktorant**.

2001–2003 **Tallinna Pedagoogikaülikool**, Infotehnoloogia juhtimine, **magistratuur**

1974–1979 **Tallinna Pedagoogiline Instituut**, matemaatika ja füüsika õpetaja.

1971–1974 **Valga 1. Keskkool**

Töökogemus

09. 2008 - **Bioinformaatik** - Clinical and Biomedical Proteomics Group, Cancer Research UK Clinical Centre, Leeds Institute of Molecular Medicine,St James's University Hospital (Beckett Street, Leeds LS9 7TF, UK), **University of Leeds** – laboratooriumi infosüsteemi arendamine.

2005-2008 **Mainori Kõrgkool,** lektor/IT Instituudi direktor.

1999–2005 **Systek Informationsystems GmbH** (Germany), tarkvara arendaja/arendusmeeskonna juht

1998–1999 **Medisoft AS**, tarkvara arendaja

1998–1998 **Sofimation OY** (Finland), tarkvara arendaja

1990–1997 Erinevad valitavad ametikohad (Kolila vallavanem; Talupidajate Keskliidu juhatuse liige; Eesti Kongressi liige; Maapanga juhatuse liige)

1987–1990 **Tervishoiuministeeriumi arvutuskeskus**, tarkvara arendaja.

1980–1986 **Sideministeeriumi arvutuskeskus**, tarkvara arendaja

7.4 Curriculum Vitae

Name	Gunnar Piho
Phone	+372 51 11236
E-mail	gunnar.piho@computer.org

Education

Sept 2005 -	**Tallinn University of Technology**, Department of Informatics, **PhD studies**.
2001 - 2003	**Tallinn Pedagogical University**, Management of Information Technology, **master studies**.
1974 - 1979	**Tallinn Pedagogical Institute**, Mathematics.
1971 - 1974	**Valga Secondary School No 1**

Work experience

2008 -	**Bioinformatician** - Clinical and Biomedical Proteomics Group, Cancer Research UK Clinical Centre, Leeds Institute of Molecular Medicine, St James's University Hospital (Beckett Street, Leeds LS9 7TF, UK), **University of Leeds** – developing of LIMS (Laboratory Information Management Systems) for particular laboratory.
2005-2008	**Mainor Business School,** Lecturer / Director of IT Institute.
1999–2005	**Systek Informationsystems GmbH** (Germany), software developer/team leader in Estonian office.
1998–1999	**Medisoft AS**, software developer.
1998–1998	**Sofimation OY** (Finland), software developer
1990–1997	Active in politics (Chairman of Kohila district; member of the board of Estonian Farmers Union; member of the Estonian Congress, member of the board of Estonian Land Bank)
1987 – 1990	**Computing Centre at Ministry of Healthcare**, software developer.
1980 – 1986	**Computing Centre at Ministry of Communication**, software developer.

7.5 List of Articles Published by the Thesis Author

1. Piho, G.; Tepandi, J.; Roost, M.; Parman, M.; Puusep, V. (2011). **From Archetypes Based Domain Model via Requirements to Software: Exemplified by LIMS Software Factory**. MIPRO 2011 - 34th International Convention on Information and Communication Technology, Electronics and Microelectronics: Telecommunications and Information, Opatia, Horvatia, May 23-27, 2011. (3.2)
2. Piho, G.; Tepandi, J.; Parman, M.; Puusep, V.; Roost, M. (2011). **Test Driven Domain Modelling**. MIPRO 2011 - 34th International Convention on Information and Communication Technology, Electronics and Microelectronics: Telecommunications and Information, Opatia, Horvatia, May 23-27 2011. (3.2)
3. Piho, G.; Tepandi, J.; Roost, Mart (2011). **Archetypes Based Techniques for Modelling of Business Domains, Requirements and Software**. 21st European Japanese Conference on Information Modelling and Knowledge Bases, June 6-10, 2011, Tallinn, Estonia. (3.2)
4. Piho, G.; Tepandi, J.; Roost, M. (2011). **Evaluation of the Archetypes Based Development**. Barzdins, J.; Kirikova, M.; (Ed.). Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010 (283 - 295).IOS Press, Frontiers in Artificial Intelligence and Applications, Volume 224 (3.1)
5. Piho, G.; Tepandi, J.; Parman, M.; Perkins, D. (2010). "**From archetypes-based domain model of clinical laboratory to LIMS software**." Opatia, Croatoa, 24-28 May 2010 : IEEE, 2010. MIPRO, 2010 Proceedings of the 33rd International Convention, Volume: Digital Economy, pages 1179-1184, ISBN: 978-1-4244-7763-0. (3.2)
6. Piho, G.; Roost, M.; Perkins, D.; Tepandi, J.; (2010). "**Towards archetypes-based software development**." [ed.] T. Sobh and K. Elleithy, Springer, 2010. Innovations in Computing Sciences and Software Engineering: Proceedings of the CISSE 2009, pages 561-566, DOI: 10.1007/978-90-481-9112-3_97, ISBN: 978-90-481-9111-6. (3.1)
7. Piho, G.; Tepandi, J.; Roost, M. (2010). "**Domain analysis with archetype pattern based Zachman framework for enterprise architecture**." [ed.] A K Mahmood, et al. Kuala Lumpur, Malaisia, 15th - 17th June 2010 : IEEE, 2010. Proceedings The 4th International Symposium on Information Technology 2010, Vol 3 - Knowledge Society and System Development and Application, pages 1351-1356, ISBN 978-1-4244-6716-7. (3.1)
8. Piho, G., Tepandi, J. ja Roost, M., "**The Zachman framework with archetypes and archetype patterns.**" [ed.] J. Barzdins ja M. Kirikova. Riga, Latvia, Baltic DB&IS, July 5-7, 2010 : University of Latvia Press,

2010. Databases and Information Systems: Proceedings of the Ninth International Baltic Conference, pages 455-570. (3.2)

9. Tepandi, J.; Piho, G.; Liiv, I. (2010). "**Domain engineering for cyber defence visual analytics: a case study and implications.**" Tallinn, Estonia : CCD COE Publications, 2010. CCDCOE Conference on Cyber Conflict, pages 59-77. (3.2)
10. Piho, G (2008). “**Towards archetypes based domain model of clinical laboratory**”. In: Proceedings of Doctoral Symposium held in conjunction with Formal Methods 2008: Doctoral Symposium of 15th International Symposium on Formal Methods, May 26-30 2008, Turku, Finland. Troubitsyna, E. (Ed.) Turku, Finland: Turku Centre for Computer Science, 2008, (TUCS General Publications; 48), pages 33 - 42.
11. Piho, G. (2008). “**Towards archetypes and archetype patterns based software engineering techniques of domains, requirements and software**”. Nordic workshop and doctoral symposium on dependability and security (NODES 08) August 29, 2008, Marguse, Estonia. , 2008, 31 - 36.
12. Piho, G. (2008). “**A Quantity: A simple example of software development with domain analysis**”. In: Collection of articles from second annual conference of Doctoral School of Information and Communication Technology (IKTDK): 25.-26. April 2008, Voore (Estonia): Tallinn: Tallinn Technical University Press, 2008, pages 73 - 76.
13. Piho, G. (2007). “**Archetype patterns based method of prescribing enterprise software requirements**”. In: MIPRO 2007 proceedings: MIPRO 2007, Opatia (Croatia), May 21-25, 2007. (Ed.) Čišic, D.; Hutinski, Ž.; Baranovic, M.; Sandri, R.; Rijeka (Croatia): Studio Hofbauer, 2007, (Business Intelligence Systems), pages 236 - 241. (3.2)
14. Piho, G. (2007). “**Archetypes and archetype patterns based engineering techniques of domains, requirements and software: the clinical LIMS software factory**”. In: Collection of articles from second annual conference of Doctoral School of Information and Communication Technology (IKTDK). Viinistu (Estonia), 11.-12. Mai 2007. Tallinn: Tallinn Technical University Press, 2007, pages 65 - 68.

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo.** Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components.

48. **Vineeth Govind**. DfT-Based External test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pav**el **Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.