Master Thesis
Software Engineering
Thesis no: MSE-2005:14
June 2005

Transformation of Rational Unified Process analysis model to design model according to architectural patterns

Andrzej Bednarz

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Andrzej Bednarz
Address: ul. Kr. Marysieńki 5/2,
51-200 Wrocław, Poland
E-mail: andbed@tlen.pl

External advisor:
Dr. Bogumiła Hnatkowka
Institute of Applied Informatics
Wroclaw University of Technology

University advisor:
Dr. Ludwik Kuźniarz
School of Engineering

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Applying Rational Unified Process (RUP) in a project means to develop a set of models before the system could be implemented. The models depict the essentials of the system from requirements to detailed design. They facilitate getting a system that has appropriate and rich documentation (therefore highly maintainable) and addresses user needs. However, creation of the models may cause overheads since a lot of work has to be put to elaborate the artefacts.

In this paper a method that makes RUP more efficient is proposed. The method makes use of the fact that every subsequent model is developed basing on the previous model. In other words, models are successively transformed from requirements up to executable code. In particular, design model bases on an analysis model. The proposed method applies automatic model transformation from an analysis model to a design model. Firstly, an approach for performing automatic transformation is chosen. Secondly, a tool applying this approach is implemented. Finally, the transformation tool is tested and evaluated in an empirical study.

The results show that automation of model transformation may be beneficial, and therefore can help in getting better systems in shorten time.

Keywords: transformation of UML models, RUP, software architecture, patterns

CONTENTS

ABSTRACT ..... **I**

CONTENTS ..... **II**

1 INTRODUCTION **1**

1.1 BACKGROUND AND MOTIVATION 1
1.2 RELATED WORKS 2
1.3 OUTLINE OF THE THESIS 3
1.4 READING GUIDELINES 3
1.5 THE RESEARCH 3
1.5.1 Goals and objectives *3*
1.5.2 Research questions *4*
1.5.3 Research methodology *4*
1.5.4 Expected outcomes *4*

2 DESCRIPTION OF RUP METHODOLOGY **5**

2.1 INTRODUCTION 5
2.2 OVERVIEW OF THE ANALYSIS MODEL 6
2.3 OVERVIEW OF THE DESIGN MODEL 9
2.4 TRANSFORMATION FROM AN ANALYSIS TO A DESIGN MODEL IN THE THESIS CONTEXT 11

3 OVERVIEW OF THE ARCHITECTURAL PATTERNS **15**

3.1 INTRODUCTION 15
3.2 FEATURES OF LAYERED ARCHITECTURE 16
3.3 EXAMPLES OF LAYERED ARCHITECTURAL PATTERNS 18
3.4 LAYERED ARCHITECTURE IN DESIGN MODEL 25
3.5 SUMMARY OF THE LAYERED ARCHITECTURAL PATTERNS 27

4 OVERVIEW OF TECHNIQUES FOR REALIZATION OF UML MODEL TRANSFORMATION **29**

4.1 INTRODUCTION 29
4.2 FEATURES OF THE DESIRED TRANSFORMATION TECHNIQUE 30
4.3 OVERVIEW OF THE TRANSFORMATION APPROACHES 30
4.4 SUMMARY AND CHOICE OF THE TRANSFORMATION APPROACH 34

5 REALIZATION OF THE TRANSFORMATION **36**

5.1 CHARACTERISTICS OF THE IMPLEMENTATION ENVIRONMENT 36
5.2 DESIGN OF THE TRANSFORMATION TOOL 38
5.2.1 Functional requirements *38*
5.2.2 Quality requirements *38*
5.2.3 Analysis *38*
5.2.4 Design *39*
5.2.5 Implementation *41*
5.2.6 Verification and Validation *41*
5.3 TRANSFORMATION LANGUAGE 41
5.3.1 Expressing Architecture Schema *42*
5.3.2 Expressing Transformation Rules *43*
5.3.3 Sharing Architectural Patterns *49*
5.4 CONCLUSIONS FROM DEVELOPING 50

6 EXPERIMENT **52**

6.1 INTRODUCTION 52
6.2 PROBLEM STATEMENT 52
6.3 EXPERIMENT PLANNING 53
6.3.1 Context *53*
6.3.2 Hypotheses *54*

6.3.3 Experiment variables *54*
6.3.4 Experiment design *54*
6.3.5 Instrumentation *57*
6.3.6 Validity discussion *57*
6.4 EXPERIMENT OPERATION 59
6.5 DATA ANALYSIS 59
6.5.1 Time spent on performing the transformations *59*
6.5.2 Number of faults in the transformations *62*
6.5.3 Hypothesis testing *63*
6.6 INTERPRETATION OF RESULTS 63
6.6.1 Pilot study results *63*
6.6.2 Experiment results *64*
6.7 DISCUSSION AND CONCLUSIONS 65

7 SUMMARY AND SUGGESTIONS FOR FURTHER RESEARCH 67

REFERENCES 69

APPENDIX A 72

APPENDIX B 80

APPENDIX C 81

1 INTRODUCTION

1.1 Background and motivation

The software engineering discipline grew up from the crisis of software development. The software systems were developed with time and cost overrun, and applications failed to provide expected features. As we can see in the report of The Standish Group (Standish, 1994), the situation was very bad and even in 1993 only 16,2% projects were completed with full success. The reasons why the crisis emerged are very complex but some crucial are lack of proper methods, tools, and procedures with dealing with more and more complex systems.

As an answer for the crisis, software engineering discipline occurred. It was clear that without good software development strategies, methods, tools, and best practices projects would fall repeatedly. The history of software development methodologies is long; however, nowadays two approaches are leading: heavy and lightweight (called agile). At the beginning, we can notice strong development of heavy methodologies but they were neither sufficient nor proper for every type of the projects. Therefore, agile approach emerged. Agile proponents found many weaknesses of heavy approach and proposed other solutions. According to Cutter Consortium report (Charette 2001), both approaches are broadly used nowadays; both have some significant advantages and disadvantages.

One of the reasons why heavy methodologies are criticized is the fact that a lot of time has to be spent on activities which do not provide even single line of code. For instance, when heavy approach is applied, large amount of time and effort have to be spent on drawing carefully several UML models. These models are used for business analysis, system analysis, and design, and do not lead (directly) to executable code (only detailed design can lead to generation of some source code). Although they require additional time, they facilitate designing highly maintainable system (with good architecture) that addresses real needs of the user. Moreover, the system is created in a carefully planned way. These are some of the reasons why large companies often use heavy methodologies (Charette 2001).

Other approaches to software development try to resolve the issue of time consuming modelling in different ways. Agile methods, even if they use UML models, do not care much about them. The models can be just in a form of sketch on the blackboard or on the sheet of paper that is thrown away when is no longer needed. In this way, agile methods do not squander too much time on creating UML models (Beck 1999). Opposite approach came from companies and institutions associate in Object Management Group (OMG). They have proposed a standard called Model Driven Architecture (MDA) (OMG 2005). In foundations of this approach lies the fact that in the process of system modelling only the first model (for instance business model) is built from a scratch. Every subsequent model is a result of certain transformation of the previous one, e.g. Business Model is transformed to Platform Independent Models, they to Platform Specific Models and so forth. Transformations can be performed up to executable code. Although transformations of UML models are the essential part of MDA, it is still not clear how to perform them. There are many investigations at this moment to find the most suitable manner to perform such transformations (Sendall and Kozaczynski 2003).

In this research, we propose another approach to address issues concerned with development process and time-consuming modelling. If agile approach demonstrated that heavy approach wastes too much time on modelling and MDA approach shows that models could be transformed one to another, why not consider enhancement of

modelling in heavy methods – which are already widely used on the market – to make them more efficient by automatic transformations. These days, there are some model transformation techniques developed mainly to use with MDA. We believe that they can be applied also in heavy methodology. Therefore, the idea of this research is to attempt to make heavy methodologies more efficient by computer aided UML model transformation. To achieve this we have to take one of the heavy methodologies, specify one of the UML model transformations within it, and choose one UML model transformation approach. Afterwards, apply the chosen approach to perform specified transformation and evaluate the results.

First of all, we have to choose one of the heavy methodologies. The most suitable is Rational Unified Process (RUP) because it is the most significant representative of heavy methodologies, as Cutter Consortium shows in their report (Charette 2001).

A transformation worth consideration within RUP is the transformation from analysis model to design model according to architectural patterns because of two main reasons. Firstly, it is relatively easy to perform such a transformation in an automatic way. Secondly, this transformation is excruciating for system architects – they have to draw large diagrams with many elements. Success in this field would make the work of system architect more efficient and would show that other similar improvements are possible.

1.2 Related works

Transformation of UML models has been in the past few years very intensely investigated, mainly in the context of MDA, e.g. (OPSLA 2003). Essential comments on contemporary UML model transformation approaches are enclosed in the (Sendall and Kozaczynski 2003) article. The authors classify UML model transformation approaches into three groups: direct model manipulation, intermediate representation, and transformation language support. They describe the ideas of the approaches, evaluate theirs implementation, and because none of theses approaches is sufficient, they describe features of the most desirable approach.

In the next paper Sendall (2003) put forward an idea of model transformation language called Genermorphous, which is a sort of combination of two significant approaches: generative and graph transformations. This approach, according do authors, allows for the most powerful and easy-to-use transformation technique.

The other classification and evaluation of UML model transformation approaches could be found in an article by Czarnecki and Helsen (2003). They distinguish several model transformation approaches in a slightly different manner than Sendall and Kozaczynski (2003). They also cannot find the approach that would be unambiguously the most appropriate one. They point out that more experiments and practical experience is required to find the best approach.

All these resources deliver many useful considerations, but they are focused on faintly different assumptions comparing to this paper – only from MDA perspective. In this research, the evaluation of model transformation approaches has different criteria. It is desirable to apply model transformation in the particular case. Considered transformation is the transition from RUP analysis model to RUP design model. We have not found any works concerning enhancement (in similar sense) of RUP.

Beside the approach to perform considered transformation, there is also a need to reflect on target architecture in RUP design model. Therefore some architectural patterns will be considered. Many resources about software architecture patterns can be easily found, e.g. in (Bosch 2000) or in (Fowler 2002). We have to emphasize the fact that what we call architectural pattern here, Bosch (2000) calls architectural style. This paper contains discussion that is not considered in (Bosch 2000) or (Fowler 2002)

– an attempt to define an architectural pattern in a way that can be applied in an automatic transformation and can be shared between different projects.

1.3 Outline of the thesis

The paper is organized as follows. First part of the thesis – first four chapters introduce the reader to the main concepts of considered transformation. Second part of the thesis (chapters from 5 to 7) provides description of the investigations and their results.

Chapter 2 contains description of RUP methodology. RUP analysis model and design model are presented in details since they are the source and the outcome of the considered transformation. After that, in Chapter 3, an overview of the architectural patterns is presented. Architectural pattern concerns defining transformation rules. In Chapter 4, several approaches of performing UML model transformation are discussed and the most appropriate approach is chosen. The following Chapter 5 describes implementation of the transformation. The subsequent Chapter 6, contains description of an experiment with human subject which was carried out in order to evaluate the developed transformation tool. Finally, summary, conclusions, and suggestions for further research are presented.

1.4 Reading guidelines

The thesis is intended for software engineers. Readers should be familiar with Object-Oriented terminology. In particular, the knowledge of UML is indispensable. Basic terms, like *a class*, *an use-case*, *an association*, etc. are not explained. Additionally, the reader should have basic knowledge of the empirical research methods since there are not explained terms like *null hypothesis, treatment*, *descriptive statistics*. They are discussed in separate works on research methods, e.g. in (Wohlin et al. 2000).

If readers have already good knowledge of RUP methodology, they can skip a part of Chapter 2 (2.1, 2.2, and 2.3) which contains description of RUP analysis model and design model. Accordingly, parts of the chapters describing architectural patterns and UML model transformation approaches can be skipped for advance readers in these issues.

1.5 The Research

1.5.1 Goals and objectives

The main goal of the thesis is to verify if applying an automatic UML models transformation approach can improve the work of the software architect in Rational Unified Process.

The following objectives need to be addressed.

- To evaluate current approaches to automatic UML model transformations and choose the most appropriate one
- To define a way of describing target architecture in RUP design model
- To elaborate and to evaluate prototype of the tool implementing transformation form RUP analysis model to design model

1.5.2 Research questions

Several research questions will be addressed in the study. The main is:

- How the work of the architect in Rational Unified Process can be improved by introducing tool support for UML model transformation?

The other questions that will be addressed are:

- Which UML model transformation approach is the most suitable to perform transformation from RUP analysis model to design model
  - What are the criteria of choosing the most appropriate technique for analysis to design model transformation?
  - How the process of UML model transformations should look like?
  - Which technique is the most appropriate and why?
- How to define target architecture in RUP design model that is undemanding for automate processing in a way that will be convenient to an architect

1.5.3 Research methodology

A literature study will be performed to investigate what are the current approaches to UML model transformations. Then evaluation of these approaches in context of applying them in RUP analysis to design model transformation will be performed to choose the most appropriate approach.

After that, an experiment will be performed to examine usefulness of chosen approach. To perform the experiment a prototype of a tool that applies chosen approach will be developed. Finally, two groups of architects will perform transformation from RUP analysis model to design model; the first one using developed tool and the second one without it.

1.5.4 Expected outcomes

Based on the research questions, the following outcomes are foreseen:

- Results of the experiment confirming or denying the idea of improving work of an architect in RUP by automate model transformation
- Evaluation of current approaches to model transformation and choice of the most appropriate one for transformation of RUP analysis model to design model according to architectural patterns
- A way of defining software architecture for automate transformation
- A prototype of the tool (or plug-in) performing automate model transformation

2 DESCRIPTION OF RUP METHODOLOGY

2.1 Introduction

Rational Unified Process (RUP) is a specialization of a generative software development process – Unified Software Development Process (USDP), shortly called Unified Process (UP) (Jacobson et al. 1999). RUP is based on UP, thus all of the facts about the development process presented below for UP are valid for RUP as well. However, some additions and refinements are introduced into RUP (e.g. about usage of the tools, documents schemas); it is mentioned where appropriate. The following considerations are based on (Jacobson et al. 1999) and on the RUP documentation (RUP 2003).

The fundamental fact about UP is that UP is use case driven, architecture centric, iterative and incremental. These three essential features of UP can be summarised as follows.

Process is use-case driven because it proceeds through a series of activities that derive from use-cases. Firstly, use-cases are identified, described, and analysed. Secondly, they are designed and implemented. Finally, they are the source for testers to construct test cases.

UP is architecture centric what means that the role of software architecture is very important in the process. Good architecture ensures that quality requirements are fulfilled and a system is able to encompass all functional requirements. Neither use-cases nor system architecture is developed in isolation. They are both developed in tandem to make a proper system that addresses users' needs and fulfils quality requirements.

The process is iterative and incremental in a way that it consists of some phases and workflows (in RUP called disciplines) which define software development process in details. On Figure 1, below, the core workflows, phases, and some exemplary iterations are presented.

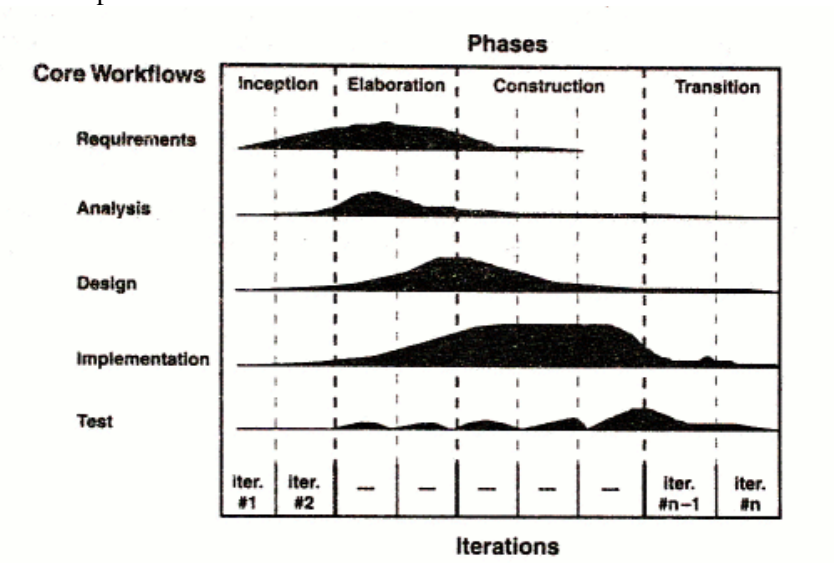

Figure 1. Core workflows, phases and exemplary iterations in UP, source (Jacobson et al. 1999: 104)

Briefly, after requirements elicitation, analysis model is created. Analysis model addresses functional requirements in terms independent of implementation platform.

Afterwards, design model is developed. It takes analysis model and all quality requirements (non-functional requirements) as an input and provides architectural design and detailed design which defines how a system works. After developing design model, it comes to implementation, testing, and deployment. All these activities are performed in an iterative and incremental manner within successive phases (Figure 1). The point is that every workflow is present in every iteration. However, certain workflow in successive iterations takes larger or lower attention. For instance, within several beginning iteration more emphasized are requirements and analysis whereas in latter iterations it is put more effort to implementation and testing.

Very important characteristic of UP is that it leads to development of a working system as well as a set of artefacts. According to Jacobson et al. (1999), "artefact is a general term for any kind of information created, produced, changed, or used by workers in developing the system".

This thesis concerns transformation of two artefacts: analysis model and design model. Model, according to OMG (2004), is "an abstraction of the physical system, with a certain purpose". This certain purpose influences contents of the model, determines what is allowed and what is irrelevant. In other words, a model is an abstraction of the modelled system from a certain point of view at the appropriate level of detail (level of abstraction). A point of view may be for instance design view or test view. As results of activities in UP, a set of models is created. The primary model set of the UP consists of: use case model, analysis model, design model, deployment model, implementation model, and test model. For the sake of conciseness, in the remainder of this chapter, only two models: analysis and design, which are the objects of the considered transformation, are presented in details.

2.2 Overview of the analysis model

A source model of the transformation is the analysis model in Rational Unified Process, thus this chapter contains detailed elaboration of it. Analysis model, which is basically a conceptual object model for analysis workflow, is created between use-case model (requirements) and design model, as it is presented on Figure 2. It provides for them an output and an input, respectively. The analysis model facilitates refinement of the requirements and allows to outline internals of the system.

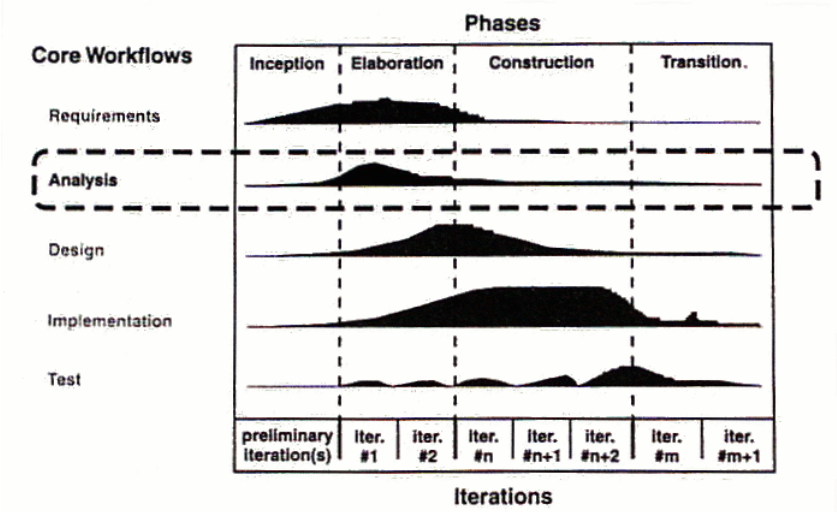

Figure 2. The focus of analysis, source (Jacobson et al. 1999: 179)

Analysis model is considered before design and implementation because it deals with some other concerns. In analysis, non-functional (quality) requirements are not taken into consideration whereas they are essential during design and implementation. In analysis the focus is set on refining and structuring the functional requirements. Design and implementation require consideration of functional requirements according to software architecture, foreseen components, constraints put on the system, performance and maintainability requirements, etc. (The following section contains more detailed description of design). Overall, the reason to distinguish analysis from design is following the separation of concerns principle. Analysis is considered mainly in the initial iterations (see Figure 2).

Analysis model consists of several artefacts elaborated by different workers, in a way that is presented on Figure 3.

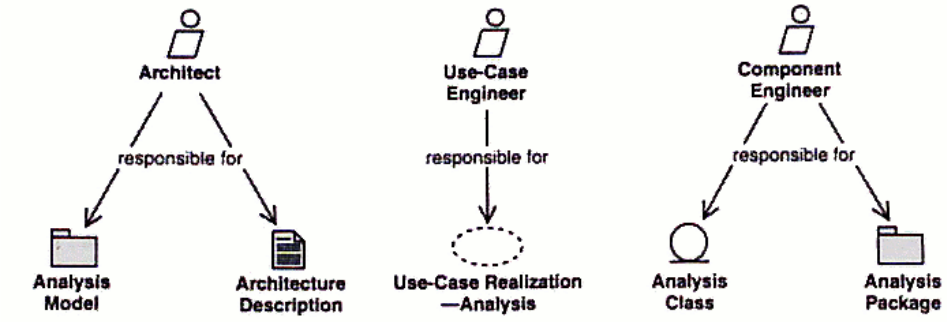

Figure 3. The workers and artefacts involved in analysis, source (Jacobson et al. 1999: 175)

Architect is responsible for the correctness, consistency, and readability of analysis model. Analysis model is correct when it refines and realizes all the functionality expressed by use-cases but nothing more. Additionally, architect is responsible for architecture description – definition and description of architecturally significant elements of the analysis.

Use-case engineer's main responsibility is to ensure integrity of one or several use-case realizations, which should fulfil the requirements. This means that all textual descriptions and diagrams of the use-case realizations are appropriate, comprehensive and address their aim.

Component engineer is responsible for defining and maintaining one or several analysis classes. This involves specification of responsibilities, attributes, relationships, and special requirements of the classes. Each analysis class should fulfil the requirements that come from appropriate use-case realization. Component engineer also has to ensure integrity and correctness of one or more analysis packages. Usually, component engineer is responsible for an analysis package as well as for analysis classes that it contains.

The flow of activities of the analysis workflow is as follows. Architect initiates the analysis model creation by identifying key analysis packages, evident analysis classes, relations between them, and some general requirements (about, for instance, performance, robustness). After that, use-case engineer realizes use cases with analysis classes. The classes get some requirements (expressed by responsibilities, attributes, relations) which are then specified by component engineer. All the time during analysis, architect finds new classes, packages, relations, requirements and the analysis model evolves. Accordingly, component engineer refines and put more details into analysis packages as development of the model proceeds.

There are several artefacts produced in analysis model. The main are: analysis classes, analysis packages, use-case realizations, and architecture description (view of the analysis model).

According to Jacobson et al. 1999, “an analysis class represents an abstraction of one or several classes and/or subsystems in the system’s design”. The characteristics worth mentioning about analysis classes are the following. An analysis class disregards quality requirements; the focus is set on handling functional requirements. The purpose of this feature is to keep analysis class more evident and clear, with direct reference to domain context. The analysis class is usually of larger granularity than its matching part from design and implementation. Consequently, the class can contain operations, however usually better would be some higher-level responsibilities. Similarly, relations, attributes, and types of attributes are also on higher level, straightforwardly associated with the problem domain, with no direct mapping to programming language.

For the sake of simplicity, analysis class fits one of three stereotypes: boundary, control, or entity. Each of these stereotypes introduces specific semantics in a clear and intuitive way. These three stereotypes go along with the separation of concerns principle. To model interactions between actors and the system the boundary classes are used. They represent abstraction of graphical user interface (GUI) components, communication interfaces, and other interfaces to external systems (e.g. printer interface). Entity classes are used to model persistent data. They represent some persistent concepts like a real-life object, or a real-life event. Usually, they are mapped directly from business entities. The main difference between them is the fact that an analysis class represents an object handled by the system which is developed, whereas a business class represents an object in the problem domain. The last type of analysis class is control class. Control class is used to model control behaviour referred to one or more use cases. It represents the dynamics of the system, managing control flows and the main tasks. It does not have to handle every kind of behaviour, but it has to coordinate activities of all objects used to implement a use case.

The subsequent analysis artefact is a use-case realization. It is used to describe how a particular use case is realized, how analysis classes interact to perform the use case. The use-case realization can contain several other artefacts. They are: a textual flow-of-events description, class diagram with analysis classes that participate in realization of the use case, and interaction diagrams (collaboration or sequence) that present certain scenarios of the use case.

The next artefact, analysis package provides a mean to partition aforementioned analysis artefacts. An analysis package can contain use-case realizations, analysis classes and analysis subpackages. Analysis packages should be characterized by high cohesion (i.e. the level of uniformity and consistency of the internals of the package) and low coupling (i.e. the level of interdependency between packages).

The last significant analysis artefact is architecture description. This artefact provides architectural view of the analysis model. It contains architectural significant elements of the analysis model. They are: top-level analysis packages and their relations, key analysis classes, important or exceptional use-case realizations.

More details and guidelines about analysis model can be found in (Jacobson et al.1999), and (RUP 2003).

The intention of analysis is to deal with functional requirements in a way that make them well structured, comprehensive, more maintainable, expressed in the developers’ language. However, development of the analysis model is a time-consuming activity. It helps to understand requirements but the system itself is not being developed. Thus, sometimes analysis model is skipped in order to lower costs of the project. If analysis model, or at least a part of it, can be directly moved to (used by) design, the benefits will be worth consideration; because in such case we could obtain high-quality requirements specification with low cost.

In (RUP 2003) analysis is not considered as a separate discipline; instead there is one discipline *Analysis & Design*. It contains all the activities and artefacts aforementioned, however it is emphasized that analysis model is optional. The roles and responsibilities are faintly different what is not very important from our point of

view. Activities and tasks of use-case engineer and component engineer are performed by one role – a designer. Nevertheless, Rational tools, and documentation repositories fully support creation and maintenance of analysis model.

2.3 Overview of the design model

Design comes after analysis; however it is developed in the iterative and incremental manner (as mentioned above). Figure 4 illustrates this. Design model is created basing on analysis model but the crucial consideration is to address quality requirements and to provide enough information for implementation workflow.

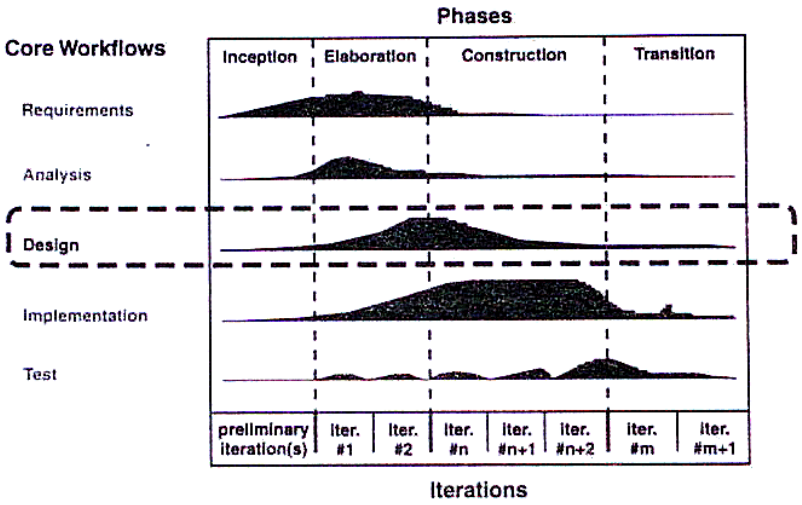

Figure 4. The focus of design, source (Jacobson et al. 1999: 217)

Design is a workflow when a system gets its shape and form according to chosen implementation platform – its abilities and constraints. Moreover, during design the following issues have to be considered: choice of operating system, concurrency technologies, database systems, programming language abilities and constraints, reusability and interoperability of the components, transaction management, interfaces between subsystems, and so forth. However, the most important is to impose a high-level structure of the system. The workers involved in design and the artefacts, which they produce, are presented on Figure 5.

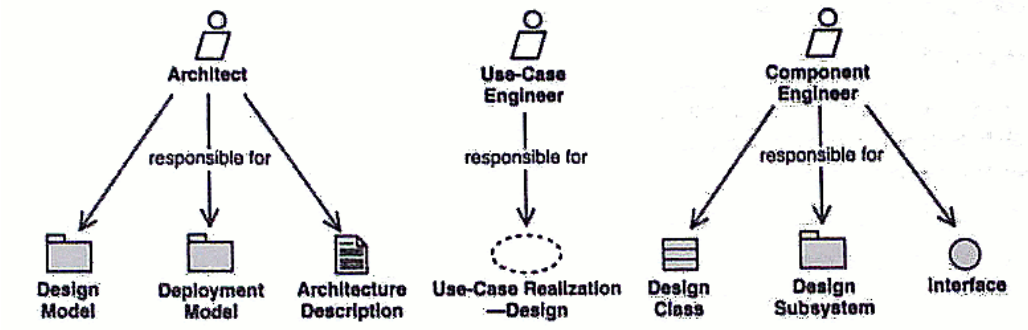

Figure 5. The workers and artifacts involved in design, source (Jacobson et al. 1999: 215)

The architect is responsible for the overall design model; its consistency, correctness, readability, and accordance to analysis model (and later to implementation model). Additionally, architect is responsible for deployment model that contains

nodes of the system. Architecture description elaborated by architect contains architecturally significant parts of the abovementioned models.

Use-case engineer takes care of one or more use-case realizations in design. This includes, similarly to analysis workflow, ensuring integrity of the use-case realizations, which should fulfil the requirements. This includes guaranteeing that all textual descriptions and diagrams of the use-case realization are appropriate, comprehensive and address their aim.

Component engineer is responsible for the details in design model. Basing on requirements provided by use-case realization, component engineer describes operations, attributes, relations, and other features of the design classes. The classes have to fulfil all the requirements which are made on it in use-case realization and architecture description. Component engineer takes care also of one or more subsystems, maintain their integrity, correctness, and completeness. Moreover, subsystems usually should have defined interfaces, which are correct and minimize.

The workflow of the design is as follows. The architect initializes creation of design model by outlining major subsystems and their interfaces, main design mechanisms, and essential design classes. Additionally, the architect points main nodes in deployment model and creates other facets of design part of architecture description. After that, use-case engineer creates use-case realizations outlining design classes, subsystems, and interfaces. Finally, component engineer specifies how the requirements of the classes are fulfilled in terms of operations, attributes, relationships, interfaces.

A more detailed description of design artefacts is needed to have holistic view of the design model. There are several artefacts created in design workflow, the essential are: design class, use-case realization in design, design subsystem, interface, and architectural view of the design model.

Design classes are abstractions of classes (or similar constructs) in an implementation of the system. They are specified in the implementation language. In implementation workflow operations are mapped to methods. Attributes are mapped directly to implementation attributes (regarding type of visibility, i.e. *public*, *protected*, or *private*). Stereotypes are mapped to appropriate constructs (e.g. a class with stereotype *EJB* could be mapped to a set of classes with appropriate interfaces and method stubs).

Design use-case realization presents how design objects collaborate to fulfil requirements. It is traced straightforwardly from analysis use-case realization. Design use-case realization contains a set of class diagrams, interaction diagrams, and textual description. They are all means to describe some flows of events or scenarios of the use case in terms of participating design objects.

Design subsystems are just means of partitioning the artefacts of design model (may be recursively) in easier to manage parts. Moreover, subsystem may provide an interface that can be used from external. As with analysis packages, design subsystems should be highly cohesive and loosely coupled. It is likely that analysis package is mapped to design subsystem.

Interfaces can by provided by classes or subsystems. They specify some operations. They should be realized by proper methods. Interfaces are means of separating the specification of functionality (operations) from their implementation (methods). If a client uses an interface, it is independent from its implementation, what is an advantage in some cases (e.g., when it comes to change implementation, client does not have to be changed).

Architectural view of the design contains architecturally significant elements of design model. These elements are: decomposition of the design into packages and subsystems, essential design classes, design use-case realizations which have impact on the whole architecture, and the organization of these classes, packages, and subsystems into layers. RUP (2003) defines in more details how a document with architecture description should look like at the stage of design. It includes several

views, according to separation of concerns principle: the logical view, process view (concurrency issues), and deployment view.

More details and guidelines about design model can be found in (Jacobson et al.1999), and in (RUP 2003).

Summarizing, after analysis, we know a conceptual way that system realizes functional requirements. However, this knowledge is insufficient. There is a need to define design model that addresses quality requirements and defines how the system works in terms of interfaces, design classes, and subsystems. These design elements are expressed according to terms of the implementation platform. Design model is an input for implementation workflow providing all necessary specifications, documents, and models.

2.4 Transformation from an analysis to a design model in the thesis context

As aforementioned, the architect is a person (a role) with main responsibility for analysis model and design model. Architect is responsible for main concepts both in analysis and in design. In analysis workflow essential analysis classes, subsystems, use-case realizations are outlined. In design workflow, the architect has already defined initial set of classes but he has to consider software architecture. Additionally, the architect identifies subsystems, interfaces, and most obvious design classes depended on the implementation platform. The most obvious are important for the architect because they have impact on the architecture, e.g. active classes in terms of threads or processes. Architect defines also generic design mechanisms: architectural styles and patterns, key mechanisms, modelling conventions which are needed to fulfil special requirements, e.g. performance, persistency, distribution. Briefly, the main task is to define architecture for the system. During design, the architect usually base on experience gained from developing comparable software systems or systems in similar problem domains.

The design process takes as input artefacts from earlier workflows (use-case model, supplementary requirements, analysis model, architecture description from analysis). The result of the design provides several artefacts: outline of layers, subsystems, design classes, interfaces, deployment model, and design architecture description.

The idea of the thesis is to improve the work of the architect, to make this transformation more efficient, to automate a part of this transformation. There are some attempts to make whole transformation straightforward, to perform everything with a tool with minimal contribution from software architect. This approach comes from OMG and is called MDA. However, this holistic approach is very hard to implement and brings many difficulties. The largest software companies try to make MDA working. The approach presented in this thesis goes in different way. It suggests introducing some improvements in small steps – improving already existing development methodologies in an evolutionary way. Therefore, we were seeking for a transformation that brings quantifiable benefits (e.g. makes transformation faster), and it is possible to implement it in considerable amount of time.

The chosen transformation regards transformation from analysis model to design model, however with some constraints. It takes analysis model as input, but only static part of it (analysis classes, packages, and their relations to each other), and returns initial design model in terms of design classes, layers, subsystems, and their relations. Such transformation does not regard only analysis model and design model. It requires also a way of defining an architectural pattern. Therefore, architectural patterns are further discussed in Chapter 3. Here, more detailed view of the source and the result of

the transformation is provided. Additionally, an example illustrating this issue is presented.

The source of the transformation is an analysis model. As aforementioned, it consists of use-case realizations, analysis classes, and packages. We take into consideration analysis classes, their grouping into packages, and their relations (created in use-case realizations). Below, on Figure 6, an example of some analysis classes and analysis packages from an analysis model is presented. Relationships between classes result from use-case realizations. The domain of the system is a students' organization that arrange some events for students (e.g. parties, meetings). It has some money from university for its operation. The money have to be carefully accounted. The model is significantly simplified for the sake of conciseness. There are two use cases. First one concerns browsing the finances of the organization and second one generating reports from operation of the organization. According to guidelines in (Jacobson et al. 1999) and (RUP 2003), and description presented in Section 2.2, each use-case realization consist of boundary, control, and entity classes. Usually analysis model contains some collaboration diagrams, or sequence diagrams that presents behaviour of the classes; however they are not respected in the considered transformation, and in consequence will not be presented here.

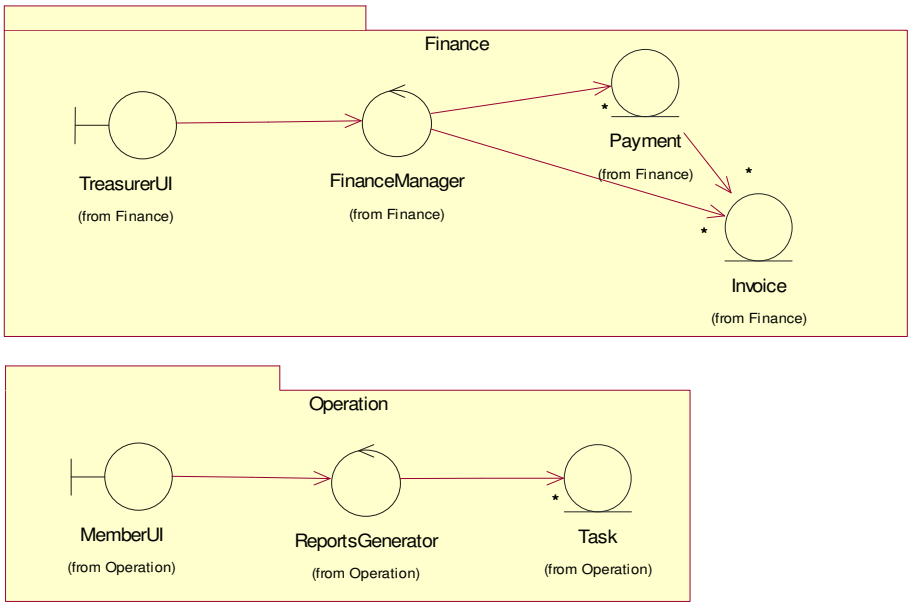

Figure 6. Example of analysis classes and packages in analysis model.

The result of the transformation contains design classes, packages, relations derived from analysis classes, and also design classes, relations, layers derived from architectural pattern and implementation platform. All of these elements must be set in implementation environment.

On the Figure 7, an exemplary result of the transformation is provided. There is introduced a structure according to *Model-View-Controller* (MVC) pattern. MVC pattern is further discussed in Section 3.3. Briefly, there are three layers: model, view, controller which contain packages, classes, and relations derived from the analysis model. The only new class derived from the architectural pattern is the *Gateway* class. Relations to this new class are also derived from the pattern. The intention of this class is to encapsulate database operations. All the entity classes use *Gateway* class to keep persistence. On the Figure 7 analysis packages that became subsystems are not explicitly drawn for the sake of clarity. However they exist, what can be easily seen

since each class has a subtitle with the name of the parent package. Classes get first letter from the name of the layer to which they belong.

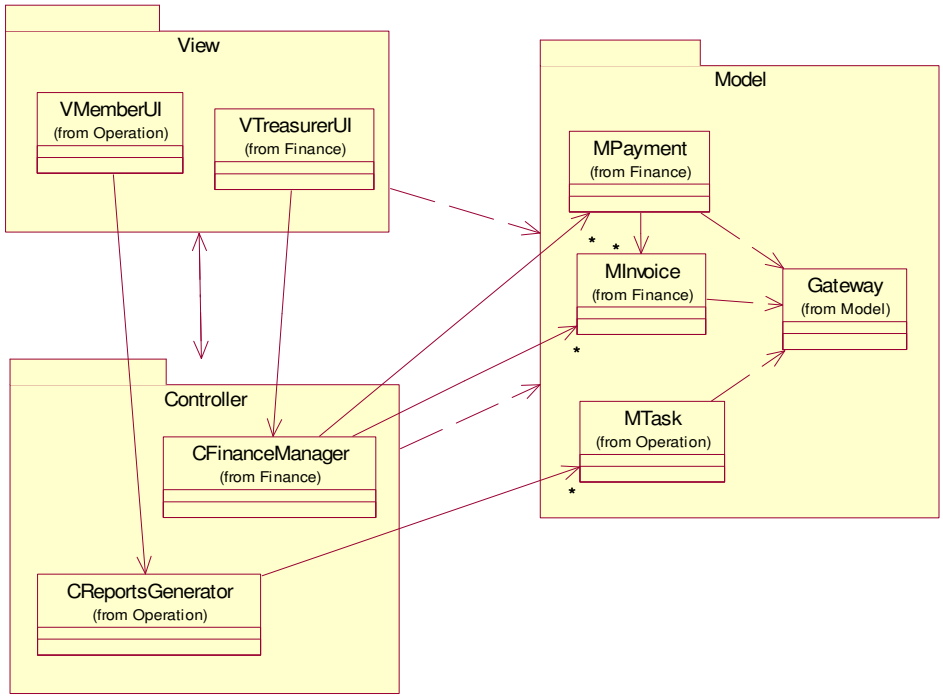

Figure 7. Example of initial design model

There are several variations of MVC pattern; here is presented only one simplified example. Good transformation tool should facilitate making changes in the definition of a pattern. More detailed view of the functional requirement for such transformation tool is presented in Section 5.2.1. The most important foreseen features are: possibility to define overall architecture in a flexible and maintainable way, possibility to adjust transformation details (which elements from analysis are taken into account), possibility to share architectural patterns definitions between projects.

Tool-supported transformation of the analysis model to design model can help architect in several ways. It can increase efficiency of work, facilities the transformation, facilitate exchanging of experience and good practises (by sharing architectural patters).

Two essential advantages of this transformation are worth emphasizing. It allows to reuse common part of many systems, i.e. architectural patterns. Pattern could be defined only one time and reused in any subsequently developed system with the same (or similar) architecture. Moreover, this approach makes analysis model even more valuable. Sometimes developers omit development of analysis model in order to avoid additional costs. When they do not have analysis model, understanding of requirements and their specification might be much worse. By using a tool that creates design model from analysis model directly, additional costs are not spoiled because every analysis class is automatically transformed to design class (classes). In other words, creating analysis model means creating simultaneously design model. Developers, after all, they have to create design classes anyway.

To sum up, the transformation chosen to implement is a transformation of RUP analysis model to design model according to architectural patterns. It does not take whole analysis model and does not give as a result complete design model. Instead, it transforms the static part of the analysis model (analysis classes, packages, and their

relationships) to initial design model (design classes derived from analysis classes associated to packages and layers and their relations). However, before it is possible to perform this transformation, there is a need to define architectural patterns. Architectural patterns are discussed in the subsequent chapter.

3 OVERVIEW OF THE ARCHITECTURAL PATTERNS

3.1 Introduction

As aforementioned, we consider a transformation from an analysis model to a design model according to chosen architectural pattern. The architectural pattern is like a schema or framework which defines the overall system architecture. Due to some misunderstandings in vocabulary, at the beginning of this chapter some definitions are provided.

The notion of a *pattern* comes from Christopher Alexander. He wrote several books in the domain of civil and architectural design suggesting some well-known solutions for common problems (Alexander et al. 1977). Alexander introduced the first definition of a pattern. "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Gang of Four introduced patterns into computer science in 1994. Theirs milestone work (Gamma et al. 1994) describes and discuses a number of useful design patterns.

Before the term *architectural pattern* is discussed, good understanding of *architecture* is needed. There are a lot of slightly different definitions of the term. However, it is clear that architecture relates to the top-level structure of the system. It is about the major parts of the system and their relations to each other. We will use the definition provided by Bass et al. (1998). “The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them”.

There can be also a misunderstanding about a term *architectural patterns.* What we call *architectural pattern* here, Bosch (2000) in turn names *architectural style.* What is more confounding, Bosch (2000) uses also the term *architectural pattern* but as “a rule on the architecture that specifies how the system will deal with one aspect of its functionality, e.g. concurrency or persistence” (in RUP this concept is called generic mechanism). We take the term *architectural pattern* as a rule that deals with system architecture. An architectural pattern specifies how the top-level elements are organized, how they are related to each other, what are their main responsibilities. To impose an architectural pattern means to reorganize the architecture of the system completely. Design pattern, on the other hand, concerns internals of the subsystems or components, or the relationships between them (Gamma et al. 1994). However, the distinction between architectural and design patterns is sometimes blurred and it is hard to say whether a certain pattern is a design or rather architectural pattern.

The main concerns when imposing an architectural pattern is the matter of fulfilling quality requirements made on the system. Architectural pattern improves one or several quality attributes of the system and is neutral or negative for other quality attributes. For instance, blackboard pattern (Bosch 2000) is characterized by high maintainability (it is easy to add and change components) but low reliability (it is difficult to debug; the execution is not deterministic). Therefore, the choice of a proper architectural pattern depends mainly on the quality requirements of the system.

Architectural patterns can be merged to some extent but usually one pattern is predominant. Moreover, such merging can decrease conceptual integrity and increase complexity. However, we can figure out an architecture of the system that is layered but some of its components share the same resources according to blackboard pattern. In practise, a combination of architectural patterns is sometimes indispensable.

Architectural patterns are considered in (Buschmann et al. 1996: 12) where the most appropriate, from our point of view, definition is provided. "An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them." This interpretation is also used in (RUP 2003).

Only layered architectural patterns are considered in this thesis. It does not mean that layered architecture is the only choice when it comes to architectural design. There are other patterns as well, e.g. pipe and filters, microkernel, or blackboard (Bosch 2000). However, in this study we consider enterprise systems only, therefore we decided to deal with the most popular and most commonly used patterns among enterprise systems – layered patterns. Fowler (2002) considers architectures of enterprise applications and concludes that layered architecture of some form is used by most non-trivial applications. The other types of systems, e.g. real-time systems, can have other requirements and other architectural patterns may be suitable for them.

More details about all kind of patterns can be found in (Bosch 2000), (Fowler 2002), (Gamma et al. 1994), (Buschmann et al. 1996), (RUP 2003), and (Eeles 2002). The following discussion of layered architectural patterns bases on these works.

3.2 Features of layered architecture

Layers are the means of decomposing system into a set of smaller parts. Layered decomposition concerns logical partition of the system and is different from functional decomposition into components (however in some cases both decompositions may lead to the same results). Layered decomposition is horizontal; each layer provides additional level of abstraction. Every layer rests upon a subordinate layer that provides some services (interfaces).

Buschmann et al. (1996: 31) provide a definition we will use: "The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at particular level of abstraction".

A well-known example of layered architecture is the OSI model for communication protocols (Figure 8) defined by the International Organization for Standardization (ISO). It consists of seven layers and each of them uses various services of the lower layer to operate (Eeles 2002). Thanks to this system, it is easy to build network applications on the top of the OSI model. These applications can use highly abstract interfaces and does not have to implement low-level communication protocols.

There are several variations of the layered architecture. The strict variant allows for communication only between the neighbouring layers and only in one direction. This allows to decrease coupling (i.e. the level of interdependency) and decreasing of it is usually very desirable. An example for that kind of layering can be aforementioned OSI model. The top-level layer, application layer, can only use services provided by the first lower layer, presentation layer. Application layer cannot use any services provided by other layers, e.g. session layer or physical layer. Moreover, presentation layer can use only services provided by session layer but cannot use, although they are neighbouring, application layer.

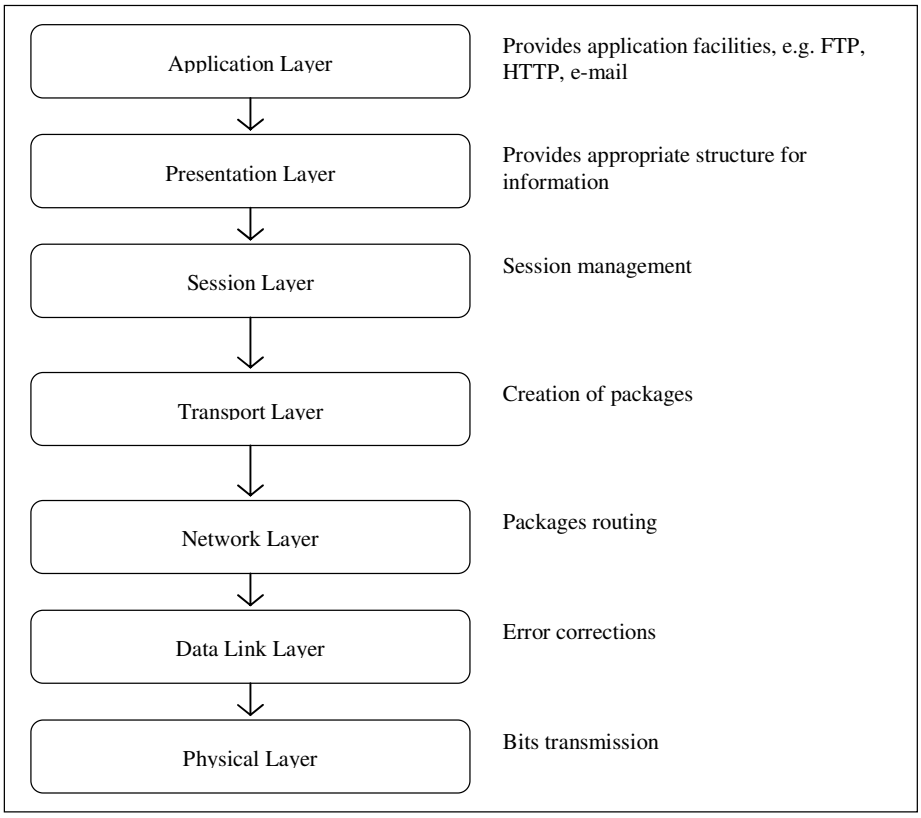

Figure 8. An example of strict layered architecture – OSI model for communication protocols.

There are some more relaxed types of layered patterns which allow for interconnection to all lower-level layers (not only to immediate below). Furthermore, there are some patterns in which a layer can invoke any other layer (even higher-level layers). This obviously increases the level of dependency between layers and makes maintainability lower. However, some systems require such kind of architecture.

As Maciaszek (2004: 190) notices, the strict layered architecture is not quite realistic. Although it would be ideal to have only top-down dependency structure, in reality the bottom-up dependencies exist. However, they "can be made relatively harmless by skilful design and programming". Several techniques are used to achieve this, more details could be found in (Maciaszek 2004). We will not consider them in the remainder of this chapter since we deal with conceptual modelling rather than detailed design.

The benefits of layers usage are hard to overestimate. They allow to develop easier to understand systems since responsibilities are divided into several parts. They allow to increase reuse of the code because some responsibilities are common to applications, for example persistency. They let to introduce parallel development since every layer can be developed separately, only interfaces are the matter of the developers of the two neighbouring layers. Additionally, layers facilitate increasing security of the system in an easy way – just additional security layer could be introduced. Furthermore, the most essential advantage is that layers allow to apply the separation of concerns principle and therefore to increase to large extent maintainability and flexibility. They are usually essential quality requirements in enterprise systems.

Layers also bring in some shortcomings. Additional levels of abstraction make performance lower since direct invocations are always faster. Crossing the layer border requires additional computation for, for example, switching between data

representation. Additionally, entities have to be represented on every level of abstraction what may lead to overheads and to maintainability decrease due to cascading changes, when, for example, an additional field of the entity has to be introduced in each layer. However generally, advantages are overwhelming shortcomings in enterprise applications and usage of layered patterns is greatly beneficial.

There is also important distinction between a *tier* and a *layer*. Although some authors treat them as synonyms, e.g. (Eeles 2002), we believe these terms should not be mixed, according to Fowler (2002). We treat a tier as implying a physical separation. For instance, a client-server application is a two-tier application when part of it runs on the client machine and second part runs on the server machine. Layers decomposition not necessarily concerns physical separation. Many layers can also run on a single machine but it still will be a multi-layered architecture. However, layering often goes along with physical separation in order to gain high performance.

In this study we deal with architectural patterns at the level independent from programming language, development tools, and implementation platform. Thanks to that, our solution can be applicable regardless type of implementation platform. Of course, the next step in design would be introducing implementation specific elements. Moreover, for a particular implementation environment some patterns are better (easier) to deal with. So, later or sooner we have to consider implementation specific issues. We will stay at the level independent from a platform though. The next step – into implementation specific solutions – could be a field for further studies and is out of scope of this work.

According to Eeles (2002), layering can be based on many different characteristics. The author lists some of these characteristics: responsibility, reuse and describe them in details. Eeles (2002) put forward also other possible bases for layering strategies: security, ownership, and skill set. Moreover, they discuss multi-dimensional layering which base on a combination of the layering strategies, e.g. a combination of reuse-based layers with responsibility-based layers. We take into account, however, only a responsibility-base layering strategy that is regarded by Eeles (2002) as “probably the most commonly used”. Moreover, Fowler (2002) does not even consider other strategies describing architectural patterns for enterprise applications; therefore we may be sure that we deal with the crucial one. The aspects, which are neglected in this layering strategy, are recalled at the end of the following section.

3.3 Examples of layered architectural patterns

According to Fowler (2002), typical layered architecture for enterprise applications consists of three principal layers: presentation, domain, and data source. The highest layer is intended to provide presentation and to handle interaction with a user. The middle layer provides business logic, computations, and contains domain-specific solutions. The lowest layer concerns all data operations (persistency, transactions, messaging). However, in practise three layers are sometimes insufficient and it is advantageous to introduce additional layers.

Layers are represented in UML as standard packages since they are means of grouping several elements together. A layer package can be stereotyped as a *layer*.

The first discussed here layering pattern is a client-server pattern. One layer, the client layer, usually works on the client machine. It is responsible for user-interface. The application code can work on both client layer or server layer or could be divided into these two layers. The first type of client-server architecture is called with *rich client*, the second *thin client* and the third is a mixed architecture. Server layer could be just a database system. The *rich client* is considered firstly.

This pattern is commonly used in applications developed by tools like Visual Studio, Powerbuilder and Delphi. These development environments facilitates to large extent building data-intensive applications. They allow to use SQL-aware GUI components. Therefore, building a screen of such an application means to drag and drop appropriate controls on the form and connect them to the database (by adjusting property sheets). Fowler (2002) claims that this approach works very well but only with relatively simple application. If domain logic get more complex, these applications become difficult to work with. Usually handling domain logic is encompassed in GUI components what causes problems with making the changes, extending the functionality and it is likely to provide duplicated code. However, this kind of applications remains the fastest-developed desktop applications and they are used in some cases. Figure 9 illustrates considered architecture. Client layer contains presentation and domain logic and uses services provided by lower layer – the server layer. Server layer contains data source and is unaware of higher layer.

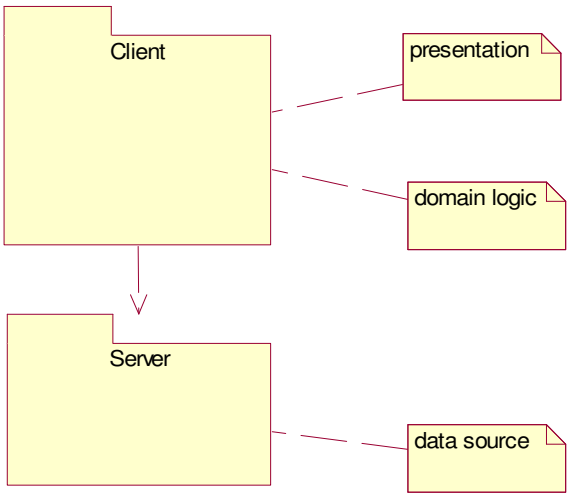

Figure 9. A client-server pattern with rich client

A client-server architecture with *thin client* deals with domain logic in different way. Domain logic is included into server layer, the rest facts about client-server architecture is similar. This pattern could be implemented in several different ways, for example, when domain logic is expressed in stored procedures in the database. The other systems built according to thin client paradigm are sometimes implemented in web scripting languages (PHP or ASP). This variant of client-server architecture is presented on the Figure 10.

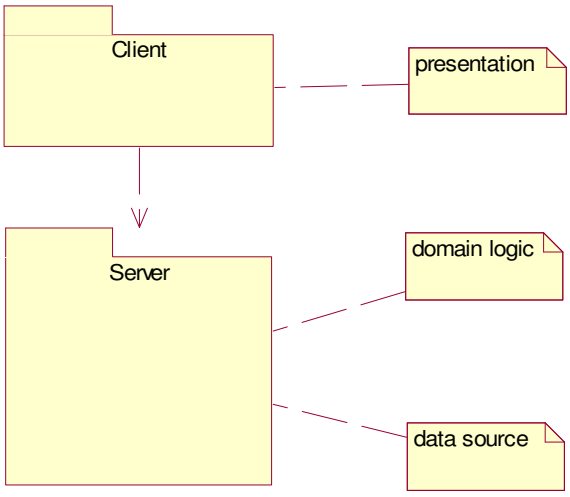

Figure 10. A client-server pattern with thin client

When object-oriented systems became more popular, more sophisticated architectural patterns emerged. At the beginning, domain logic was moved to the third layer. The three-layer architecture follows the separation of concerns principle. Web applications caused that the three and more layered architectural patterns became more popular and nowadays they are the base for most enterprise applications architectures.

The following patterns is the three-layer architectural pattern. It consists of three layers: presentation, domain, and data source (names according to (Fowler 2002)), see Figure 11. Presentation layer is responsible for user interface, for interaction between the system and the user (displaying data, interpreting commands and forwarding them to lower layer). It can be implemented in several ways, from simple command line to HTML interface displayed in a browser, or rich windows graphical user interface (GUI). It is relatively easy to change presentation layer without affecting the others. Presentation is not only involved in human-computer interaction, it is intended to handle requests of others systems too. For example, presentation could provide web services that are used by other external systems.

Domain layer is responsible for domain logic (business logic). It concerns validation of data that is forwarded from presentation or taken from data source, performing computations, and controlling the execution flow.

Data source layer is responsible for operations on data. It is responsible also for communication with external systems which provide data operation services (transaction monitors, messaging systems, etc.). This layer could be also seen as a mean of accessing other external systems. It can communicate with other systems by, for example, web services to provide needed resources.

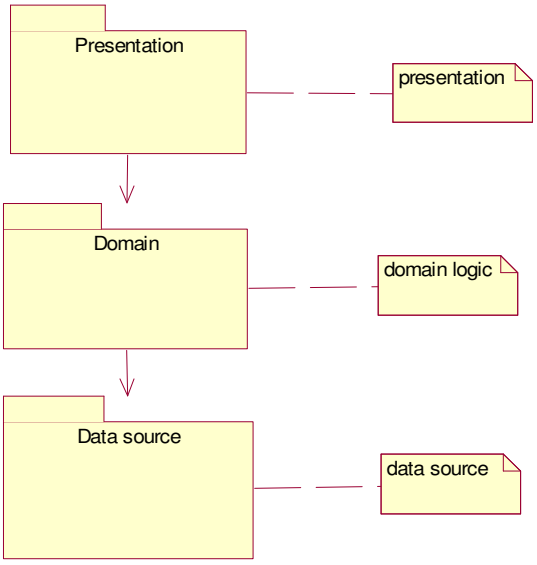

Figure 11. Three-layer architecture

There are a lot of variations of three-layer architecture. Another example is a Model-View-Controller pattern that is presented below on Figure 12. We use the definition of this pattern provided by Buschmann et al. (1996: 125). Presentation is separated into two layers: *View* (display information) and *Controller* (handle user input and events). The core functionality (from domain) and data are introduced in *Model* layer.

Unfortunately, there are some misunderstandings and confusion with responsibilities of layers in MVC pattern. The concept of MVC is quite old and was used in various systems in different contexts. Trygve Reenskaug introduced MVC for the first time for the Smalltalk platform in the late 70's as a fundamental part of Smalltalk-80 (Buschmann et al. 1996: 126). Analysis classes in Unified Process (Section 2.2) base on the principles of MVC architecture. Many patterns, for example PCMEF – a pattern discussed below, also uses MVC as a base. The problem is that some authors, for example Jacobson et al. (1999: 185) consider *controller* layer as involved is business logic processing also (not only events handling, interpreting of mouse clicks, keyboard typing). Some authors, for example Maciaszek (2004: 202), suggest putting bulk of the program's logic (so probably also domain logic) into *control* layer. However, we believe that MVC layer's responsibilities clearly stated by Buschmann et al. (1996: 126) or Fowler (2002) define the strict MVC variant.

This pattern facilitates changing the view in case of need (for example: textual, a table, a box plot), and supports several user interface paradigms (two examples: usage of keyboard, clicking on GUI). MVC pattern requires introduction of several design patterns, for example, observer, strategy, and composite (Gamma et al. 1994).

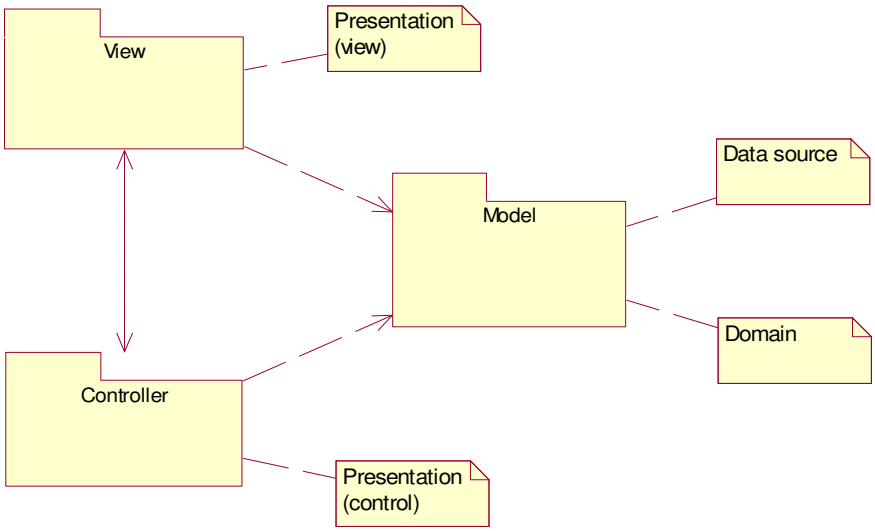

Figure 12. Model-View-Controller (MVC) pattern

Every layer from the set of three principal layers could be further divided. Fowler (2002) mentions patterns with more than three layers, although he believes that the three principal layers (presentation, domain, data source) are always present in system architecture and they represent the simplest layering schema for enterprise systems. The layers could be cut differently (like in MVC pattern) and still be useful, however there is always a reference to the three principal layers. The examples of patterns with more than three layers are presented below.

On the Figure 13 a five-layer architecture is presented. It contains two more layers which are a kind of mediating layers between the basic three layers. The *controller/mediator* layer is a mean of mediating between presentation and domain layer. It is responsible for handling flow of an application; for example, screen navigation sequence. The second newly introduced layer, *data mapping* layer, is a mean of mediating between the domain and data source layers. It is responsible for transferring data from domain objects to a data representation in data source (database).

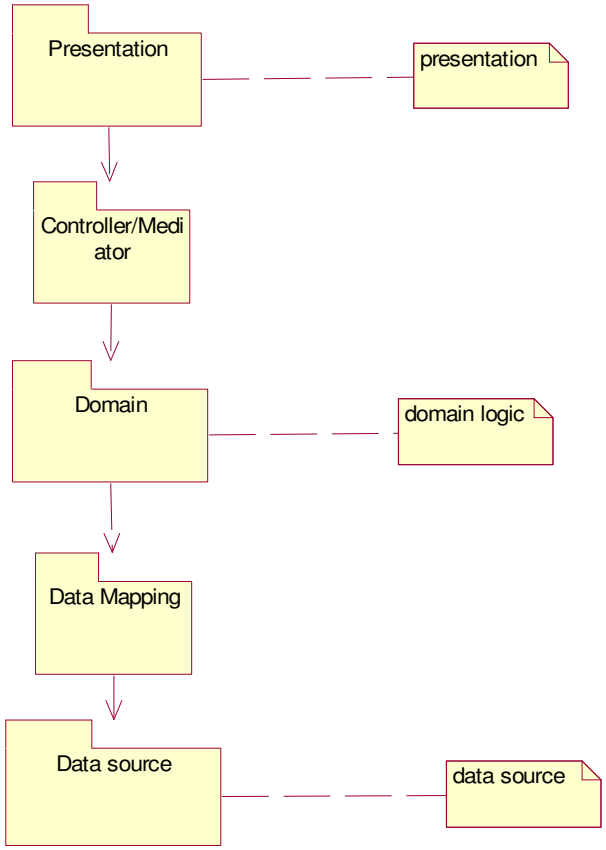

Figure 13. Five-tier architecture, source (Fowler 2002)

On Figure 14 a PCMEF+ architectural pattern is presented. It is an interesting variation of layered architecture. It is described by Maciaszek (2004) as a result of studies performed with many groups of students. The pattern consists of four principal layers: *presentation*, *control*, *domain*, and *foundation*. The *domain* layer is initially partitioned into *entity* and *mediator*. According to Maciaszek (2004), when relating PCMEF+ to MVC: *presentation* is *view*, *control* is *controller*, *entity* is *model*. However, after analysing the source code for *control* layer, it occurred that it contains domain logic (unlike *controller* layer from MVC). *Foundation* is a newly introduced layer to handle communication with databases, web services and other sources of the data. *Mediator* is a layer that mediates between *entity* and *foundation* providing that all needed objects are loaded through *foundation* and put into *entity* layer.

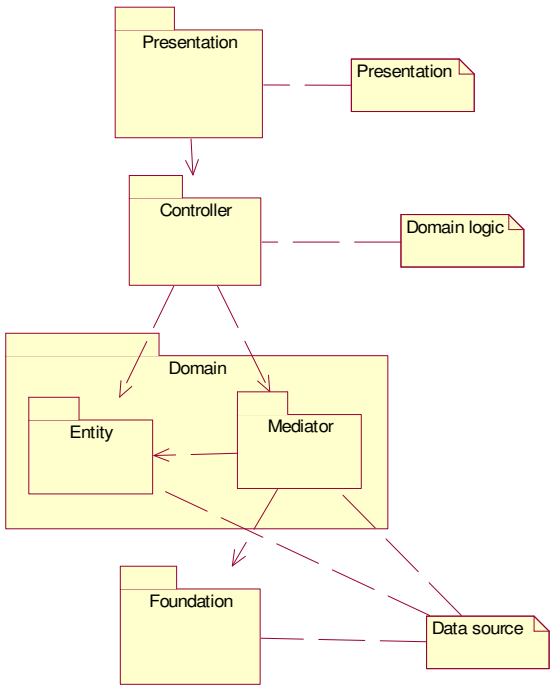

Figure 14. PCMEF architectural pattern, source (Maciaszek 2004)

As Eeles (2002) claims other ways of layering are also possible. All the already defined patterns consider just one layering strategy – responsibility based. As mentioned, there exist also other strategies; the most noteworthy is the reuse-based strategy. This strategy can result in a system that is presented on the Figure 15. Lower layers are more reusable than higher since they are less application specific. On the Figure 15 *Personal organizer* uses more reusable components *Address book* and *Calculator* which in turn use base components (so the most reusable) *Filestore* and *Memory* management, and *Math*. Reusability is one of the most important qualities in software development, therefore this kind of layering facilitates consideration of reusability. If reusability is the main concern of a system, this could be the leading strategy. However, in most systems a responsibility-based layering has to be considered primarily in order to build a proper system. Components have an internal structure and in enterprise applications they are usually partitioned into three aforementioned principal layers. Moreover, responsibility-based layering do not exclude reusability considerations. Sometimes, a sort of mixed layering (multi-dimensional layering) is applied.

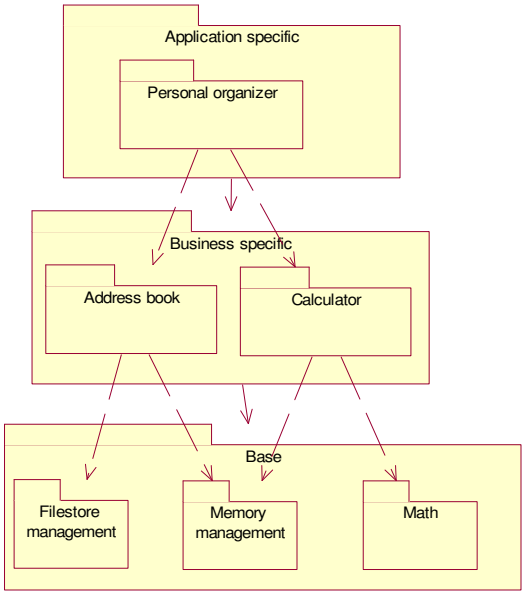

Figure 15. Reuse-based layering strategy, source Eeles (2002)

The architectural patterns can be enriched by some additional constructions. For example, MVC pattern could include observer pattern or data gateway. However, we deal with static of the models only, so additional constructions would mean to introduce some design classes, subsystems, and their relationships. The way of defining architectural pattern should enable to express such constructions.

In the remainder of the thesis we deal with responsibility-based layered architectures only. This is due to the fact that this kind of architecture is probably the most commonly used, as abovementioned, and this kind of architecture can be in explicit relation to analysis model. Therefore, a transformation could be applied. The relation could be expressed since analysis classes are considered in terms of their responsibilities (*boundary*, *entity*, *control*) and they could be mapped to design constructs which also concern responsibilities (*presentation*, *domain*, *data source*).

3.4 Layered architecture in design model

Presented layered patterns have influence on design model, implementation model. Moreover, the models in accordance with one particular pattern can be structured in several different ways. The following example presents typical approaches to structure the design model when applying three-layered architecture.

We can think of an exemplary system from students' organizations domain (introduced in Section 2.4) that consists of two parts: finances management, and operation management. In parallel, the system (and each subsystem) can be divided into three layers: presentation, domain, and data source. Layered breakdown is seen as a horizontal decomposition whereas subsystem breakdown is seen as a vertical decomposition. There are several ways to reflect these decompositions in the design model (and therefore in the code).

Below, examples of incorporating layers into design model are presented by the screenshots of the tree-views from Rational Rose.

First approach takes layered decomposition as the predominant. Each layer contains classes from every component (Figure 16). The classes are not further

partitioned into packages (subsystems). This approach could be applied in simple applications.

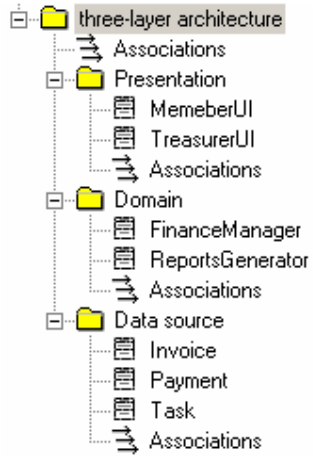

Figure 16. Simple classes partitioning into layers

Another approach introduces further partitioning (Figure 17). Each layer is partitioned according to subsystems belonging to. This approach may be useful in more complex models when a system consists of many subsystems with many classes.

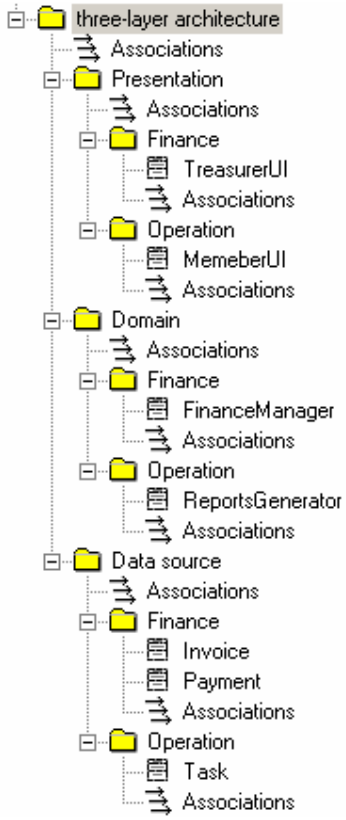

Figure 17. Partitioning into layers and subsystems

The order of partitioning could be also different, as presented on the Figure 18. The other possibility is to make subsystems partitioning the top-level divide. This could be appropriate when building an application from separate components is the

main concern, when domain-related concepts have to be supported explicitly. This is further discussed in Eeles (2002).

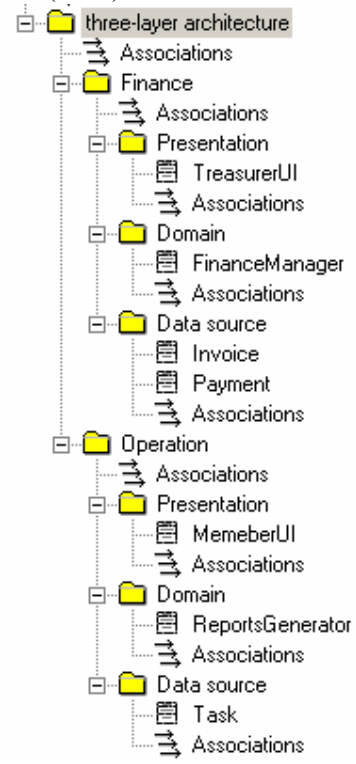

Figure 18. Partitioning into subsystems and layers

Other ways of partitioning are also possible. For the sake of conciseness, however, we will not consider any other because we mentioned the essential and commonly used.

The problem here is the fact that design architecture has to be expressed in some way that could be used in automatic transformation. Defining target architecture is strongly depended on transformation approach and is further discussed in the next Chapter 4. The way of defining target architecture – if it is easy and comprehensible for the architect – is also the key feature that decides about efficiency in performing considered transformation.

3.5 Summary of the layered architectural patterns

This chapter contains an overview of the layered architectural patterns. The main characteristics are given, some examples are included, and discussion of the ways of applying the patterns in design model is carried out.

The patterns are presented because they are indispensable part of the considered transformation. The analysis model is transformed according to an architectural pattern. The analysis model together with the chosen architectural pattern provide an input for the transformation. The analysis model is the source of analysis classes, their relationships, and analysis packages. An architectural pattern is the source of ideas how design model is organized. It defines the main parts of the design model (layers), their responsibilities and relations to each other.

The transformation maps analysis elements into design constructions. Analysis classes are usually mapped according to their stereotype. As aforementioned, the analysis classes can have one of the three stereotypes: *boundary*, *control, or entity*. The classes therefore, could be mapped to presentation, domain, and data source

layers, respectively. The exact way of mapping depends on the architect. Analysis packages are usually transformed to design subsystems. Analysis relationships are mapped straightforwardly to relationships between appropriate elements in the design model but may be discarded.

There is a need, therefore, for a way of defining the architectural pattern that would be easy to use by the user, easy to process be a transformation tool and whose functionality would allow to express needed patterns and transformation rules. The approach of defining architectural patterns is strongly depended on transformation approach, so it is further considered in the following chapter.

4 OVERVIEW OF TECHNIQUES FOR REALIZATION OF UML MODEL TRANSFORMATION

4.1 Introduction

Sendall and Kozaczynski (2003) claim that many activities within UML model transformation could be performed as automated processes. The idea of the thesis is to apply one of the approaches to transform analysis model to design model and verify if this automation is beneficial. Sendall and Kozaczynski (2003) define model transformation as "taking one or more source models as input and producing one or more target models as output, following a set of transformation rules". Before the transformation could be implemented, we have to analyse existing approaches and choose the most appropriate one.

There are known many UML model transformation approaches. They were developed in the context of MDA transformations mainly. One of the taxonomies of the UML model transformation approaches put forward Sendall and Kozaczynski (2003). The authors divide transformation approaches into three main groups:

1. Direct Model Manipulation
2. Intermediate Representation
3. Transformation Language Support

The first approach gives a user opportunity to manipulate a model using a set of procedural APIs. This approach is introduced in many modelling tools, for instance, Rational Rose, Rational XDE, Eclipse. APIs provide access to the internal model representation and allow to manipulate it with some popular programming languages. Thanks to that, developers are usually familiar with these languages and can immediately begin to develop transformation without extra training. However, the major shortcoming of this approach is the fact that the possibilities of implementing transformations are constrained by the set of provided APIs. Moreover, since popular programming languages are general-purpose rather than intended to express transformations, they lack special construction that facilitates transformations. As a result, implementing transformations can be difficult or even impossible, time-consuming and maintainability of the transformation code could be low.

The second approach concerns transformation of XMI (XML Metadata Interchange; XML – Extensible Markup Language). XMI is a standardized way of representing UML models. Some modelling tools can export and import a model in this form. The transformation relates to manipulating the XML document since XMI is a strict XML document. Every kind of XML manipulation is facilitated by alarge amount of tools. For instance, for transformation of XML document to another XML document the Extensible Stylesheet Language Transformation (XSLT) could be used. Advantages and disadvantages of this approach are further discussed in the following section.

The third approach provides a language for specifying model transformations. The language is intended to have only one destination – model transformation and offers special means (constructions, methods, mechanisms) for performing it. This is foreseen as the most promising approach.

All these approaches have advantages as well as drawbacks described briefly by Sendall and Kozaczynski (2003). Therefore, it is not trivial to decide which approach is the most suitable one in our case. There is an obligation to consider in details the

needs and the context of the transformation. The more detailed view on the available techniques is presented in the subsequent section.

4.2 Features of the desired transformation technique

Before the transformation approaches are described, the considerations on the features of desired approach are presented. The selected approach will be applied to transformation of an RUP analysis model to design model according to a given architectural pattern.

First of all, as the transformation description indicates, the solution should be easy to introduce in companies applying RUP methodology and using Rational tools. Therefore, the solution should require as little additional tools and trainings as possible. The most suitable solution would be the one that no new tools are needed to buy and that provides good integration with existing processes and tools.

Secondly, applying chosen solution should be efficient. Once implemented, transformation should be easy-to-use (interactive mode), easy to maintain, and, above all, reusable.

Finally, the solution should also allow to implement certain range of transformations. We take into account transformations according to certain architectural patterns, as described in Chapter 3, responsibility-based layered patterns.

Summarizing, the main concerns are the ability to implement the transformation according to certain architectural patterns, usability, flexibility, and maintainability. Additionally, the amounts of time needed to implement and to make changes are also important.

4.3 Overview of the transformation approaches

There are several techniques and tools worth considering throughout seeking the most suitable approach. The essential techniques, in our opinion, are described briefly below, and summarised with comparison in the following Table 1. It is worth to emphasize, however, that we are unable to analyse and verify all of the tools and approaches that have emerged on the market because new tools are coming almost every day. Moreover, some tools are unavailable due to their cost and inability to get trial versions. The analysis is performed basing on our experiences, several peer-reviewed articles about transformations, and seeking through Internet. Probably, there exist or soon will exist better solutions than it is here considered. Following the specified criteria, it would be easier to evaluate them and to find suitable approach.

Rational Rose and XDE

The first group of UML model transformations is direct model manipulation. The approach is applied in many modelling tools. One of them is Rational Rose. It is crucial and worth consideration tool because it is commonly used. Companies applying RUP as well as universities and various software engineering schools teaching RUP often use Rational Rose as the main modelling tool.

Rose provides a set of procedural APIs. It allows to perform some basic operations on the model. It allows to query the model about existence and properties of model elements, delete elements, and add or change model elements.

Full specification of API is accessible in Rose *Help* menu. It is possible to perform model operations in several ways. Firstly, Rose provides developing environment that allows to write scripts in a language that is similar to MS Visual Basic. Secondly, any

dll compliant environment (for example, Visual C++, C#) can access Rose functionality by a COM client. Finally, some other languages are supported through mappings to COM aware structures. For example, usage of Java is enabled by providing special Java library (*jar* file) which can communicate with special *dll* what makes COM operations possible.

The essential advantage of using Rose is the fact that Rose is already used in many companies applying RUP. Therefore, cost of introducing transformation with this tool would be relatively little. Moreover, according to Sendall and Kozaczynski (2003), developers are familiar with procedural languages and it is easier for them to encode transformations in this way.

There are also several disadvantages of using Rose as a mean of transformation. APIs provided by the tool constrain number of possible operations on the model and therefore limit the number and types of transformations. Usage of procedural language may be inconvenient solution to specify transformations. Procedural languages are general-purpose and do not have any special constructions (mechanisms) to facilitate transformation. Therefore, it may require a lot of time and effort to implement transformation. Furthermore, a transformation once encoded may be hard to maintain and extend (Sendall and Kozaczynski 2003).

Some of the problems are addressed in another tool by IBM Rational – XDE Modelling Tool. It brings some significant improvements – provides special transformation language based on patterns. This language allows to specify transformation on the high level of abstraction, so the main drawback is reduced. However, the number of types of transformations is still limited. Moreover, the language was not intended to perform model-to-model transformation but rather model-to-code. Additionally, the problem with XDE is that only a part of the specification of the pattern language is published. The published part concerns reading and querying the model to facilitate code generation. The unpublished part of the documentation concerns updating a model. The part describing updates was not published because the API related to making changes in the model is not stable yet and may be changed in the future versions of XDE. Furthermore, XDE is distributed with two developing environments only: Visual Studio .Net and Eclipse/WebSphere. Thus, usage of XDE is additionally constrained.

XMI

XMI is tightly coupled with other OMG standards, therefore a short introduction on OMG work (four layer meta-modeling framework) is provided at the beginning of this section according to (OMG 2002a).

The family of OMG standards for modelling software systems and architectures contains complementary specifications: The Unified Modelling Language (UML), The Meta-Object Facility (MOF), and The XML Metadata Interchange (XMI). UML specification includes a formal definition of UML metamodel, a graphic notation for expressing UML models, and an XMI format for UML model interchange. MOF is defined in the following way: “The Meta-Object Facility (MOF) Specification defines an abstract language and a framework for specifying, constructing, and managing technology neutral metamodels. A metamodel is an effect an abstract language for some kind of metadata”.

The relations between MOF, UML, and a computer system are as follows, according to (OMG 2002a: xii). Aspects of a computer system can be defined by UML model. UML model can be defined by UML metamodel. UML metamodel in turn can be defined by the MOF meta-metamodel. In other words, one can say that UML metamodel is an “instance-of” the MOF meta-metamodel. UML model is an “instance-of” the UML metamodel. And computer system is an “instance-of” the UML model.

The XMI specification defines technology mappings from MOF metamodels to XML (XML DTDs and XML Schemas). Therefore, any MOF compliant model (metamodel) can be expressed by XMI. The XMI specification defines physical representation of these meta-models and models. This means that UML models can be interchanged, exported to and imported from XMI format according to XMI specification.

As aforementioned, very promising fact about XMI is that it is a XML document. A transformation tool that manipulates UML models represented in the XMI format would be very useful because it could be applied in every case when UML modelling is used, regardless operating system or vendor's modelling tool. Benefits coming from using XMI are hard to overestimate. Usage of XMI can lead to universal, independent from tools and platforms solutions that can be broadly used. Furthermore, there are already developed several tools and techniques helpful in manipulating of XML documents, including XSLT (for transformation tasks) and XML Path Language (XPath – for querying the XML document).

However, there are many drawbacks of using XMI. XMI is not fully supported nowadays by the vendors of the UML modelling tools. Moreover, XMI standard is changing quite frequently. This causes that tools are compliant with incompatible versions of XMI. Current version of XMI is 2.0 (01.06.2005) but many modelling tools are compatible with versions 1.1 or 1.2 (this is rapidly changing though).

Moreover, unfortunately, XMI did not lead to holistic model interchange. XMI do not cover the graphical aspect of model interchange (OMG 2003b: 1). Albeit XMI covers all aspects of UML models, the problem is that the UML metamodel does not define a standard manner of expressing diagram definitions. OMG has noticed this shortcoming and proposed a specification called "Diagram Interchange" for new version of UML (2.0) in (OMG 2003b) to "enable smooth and seamless exchange of documents compliant to the UML standard between different software tools". This specification is nearly finished because it has a status of "final adopted". However, at the present time, in practise, exporting a model from one tool and importing to another (through XMI files) usually results in loosing some or even all diagram information. Thanks to OMG standardization work, this is likely to change in the future.

Additionally, modelling tools have subtle differences in implemented UML metamodels (Walkowiak 2004). These differences may cause interchanging of UML models impossible or difficult to large extent.

Furthermore, Peltier at al. (2001) recall their experience with XMI transformation tasks and conclude that writing transformation program by XMI may be long and painful. The authors used XLST to transform one model into another. The program was about 7000 lines and, in their opinion, was unreadable and very hard to maintain. Writing transformation of XMI in XSLT requires a lot of skills, basically being expert in the XSLT, MOF, and XMI specification. Another disadvantage is the fact that executing XSLT program is performed in batch mode. A transformation is not, therefore, user-friendly. It is hard to implement user interaction and it is hard to notify about errors.

MTrans – XMI extension

Due to the large number of advantages of using XMI, the idea of creating a transformation tool which is based on XMI is investigated in some research centres, for example (Wagner 2001), and (Peltier et al. 2001). An impressive solution called *MTrans* is inquired at France Telecom R&D (Peltier et al. 2001).

The authors described a language that is placed on the top of XSLT, i.e. XSLT is generated from the new language. The proposed solution facilitates defining transformation providing special constructions. As Peltier et al. (2001) show,

expressing even complex transformation is relatively easy (functional form), especially when comparing to XSLT transformations.

However, the main disadvantage of this approach, as well as many other similar approaches, is the fact that they are immature. There are no tools supporting it, specification is not complete and further investigations are still needed. The other, less important drawbacks, are that the transformation is still performed in the batch mode (like XMI transformation using XSLT) and specifying transformation must be held in a strict functional form which in some cases might be inconvenient (Sendall and Kozaczynski 2003).

Domain language

Deursen et al. (1998) put forward an idea of domain language, "A Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain". This kind of languages also arises within the domain of UML model transformation.

One of these languages uses a technique which treat UML models as graphs. This approach was suggested by Agrawal et al. (2003). The authors present a specification that is called *The Graph Rewriting and Transformation Language* (*GReAT* for short). Although the language has large enough expressive power, the usage and usefulness of this approach is not very high due to difficulties in defining transformations. The source model, destination model, and a set of explicit transformation actions are encompassed in one graph only. This makes comprehension of this method difficult and causes problems with expressing transformations (Sendall and Kozaczynski 2003).

Another approach worth mentioning is the transformation framework based on a logic programming language described by Whittle (2002). Unfortunately, again, defining transformation by writing rules in a logic language (named Maude) is not simple. Developers would have to spend considerable amount of time encoding the transformations. They would be hard to understand and maintain (Kozaczynski and Sendall 2003).

Although Sendall and Kozaczynski (2003) present several other techniques and tools implementing domain language approach, they cannot find any good enough approach to model transformation. Thus, they propose a new language. Firstly, they specify desired features, e.g. it should be executable, fully expressive, unambiguous, offer graphical constructs, declarative. In their later work they specify in more details their language (Sendall 2003). However, they did not develop implementation of this language, it is still in the initial phase.

A domain language seems to be the most suitable approach to model transformation. However, the usage of domain languages is limited to the experimental area as long as the specification is immature, there is not any leading, broadly accepted approach and there is not developed reference implementation (a tool).

Query/View/Transformation

Object Management Group noticed the need for efficient UML model transformation approach since model-to-model transformation is a key concept for the MDA. Transformation is discussed in broader document about model manipulation – Query/View/Transformation (QVT) Request For Proposal (OMG 2002b). The concept of QVT is further discussed and reviewed in Gardner et al. (2003). However, the exact date of finishing the QVT specification is unknown. Nevertheless, QVT is very promising approach considering most of the needed operations on the UML model.

Query part describes the way of selecting/filtering part of models. *View* specifies how to get a view (a part of the model) from source model. *Transformation* concerns a set of rules defining the transformation process. The *transformation* concept includes simple transformations (single elements in the source model to single elements in the target model) but also expressions, naming, complex transformations, resilience to errors and many other issues (Gardner et al. 2003). This approach is the most holistic and seems to present the most appropriate approach for model transformation. However, the specification is not ready yet. After finishing the specification, a considerable amount of time is needed to develop tools applying this approach. Some attempts have already begun emerging, for instance (QVTPartners 2004). However, they are immature and provide tiny functionality and, obviously, cannot be considered as a serious solution for transformation issue yet. Nonetheless, it is anticipated as the most promising approach that will have large impact on the MDA industry (and therefore transformation issues).

4.4 Summary and choice of the transformation approach

The following Table 1 presents the main points of the aforementioned issues and facilitates comparing different model transformation approaches. Rows represent transformation approaches (tools) and columns represent features of the transformation which are worth consideration from the thesis point of view. The features base on the characteristic of the desired transformation approach presented in Section 4.2.1. We take an uncomplicated scale of evaluating. Three values are possible *0*, *1*, and *2*. They could be interpreted in the following way. *0* indicates that a feature is not supported, *1* that a feature is supported partially and *2* that it is fully supported. The first feature, which we take into account, is fitness into RUP, i.e. if it is probable that the tool is already used in companies applying RUP. This is important because we consider a transformation within RUP. Secondly, effortless in introducing is considered, i.e. if it is easy to encode the transformation by developers (who are familiar to popular procedural languages). Thirdly, we take into account if the transformation implemented according to an approach may be easy-to-use and user-friendly for the end user. For instance, interactive mode is seen as better solution than *batch* mode. The following feature concerns architecture definition. Architectural pattern is a part of transformation rules. It should be easy to define and maintain. The next feature is expressiveness. If it is possible to express more transformations, the approach is better. Finally, tool and documentation support is considered. Better solutions have support provided by their vendors and have comprehensive and accessible documentation.

Table 1. Summary of UML model transformation techniques

	Fits into RUP	Effortless in introducing	Usability to end-user	Architecture definition support	Expressiveness	Tool & documentation support	Sum
Rose	2	1	2	2	1	2	10
XDE	2	1	2	2	1	1	9
XMI	0	0	1	1	2	2	6
MTrans	0	2	1	1	2	0	6
Domain language (*GReAT*)	0	2	2	1	2	0	7
QVT	0	2	2	2	2	0	8

Rose gets almost maximum number of points. The only significant weaknesses are the following. Provided API restricts number of operations on the model and that the only possibility is to use procedural languages which are general-purpose rather than intended to implement transformations. Architectural patterns could be expressed in UML using the tool; so all aspects of the transformation can be included in Rose.

XDE is the newer product of IBM Rational and its features are evaluated in a similar way. The essential difference from Rose is that part of the XDE documentation (about updating the model) is not published (because API is not stable yet).

Intermediate representation in the XMI form has many shortcomings. Although it is possible to express every kind of transformation and there are a lot of tools and their documentations, direct XMI manipulation is difficult to adapt with RUP (exports and imports are needed), complicated to use by developers and not user-friendly (*batch* mode, weak errors notification). Architecture should be expressed separately from transformation, e.g. in additional XML file.

MTrans is similar to XMI, however it is more friendly for developers (it contains special constructions for expressing the transformation). On the other hand, there is no tool applying MTrans comprehensively and documentation is not available.

Domain languages, GReAT in particular, have features similar to MTrans. The only difference is that they are more user-friendly – can act in interacting mode and be more flexible. The architectural pattern may be expressed in an easy and effective way because domain languages provide special constructions (e.g. graph constructions) for transformation rules.

QVT seems to be the most promising approach, it may address all of our needs. However, it is in the planning phase yet. Tool support and documentation are not provided and the support for RUP is weak (only as importing and exporting models in the future).

This evaluation provides general overview of the existing tools and approaches. The approaches (tools) are described in the previous section where the main advantages and drawbacks are listed. There are also references to more detailed descriptions in external papers. We have a view on the transformation approaches at the sufficient level of details to decide which approach fulfils our requirements. The presented Table 1 is not considered as the only criteria in decision process, however, it is used to justify our choice stronger. The additional and the most important criteria is whether it is possible to implement transformation nowadays since we have to perform the transformation. Only several tools fulfil this requirement (Rational Rose, XDE, and XMI).

As it is showed in the Table 1, the most suitable solution to implement considered UML model transformation is direct model manipulation approach implemented in Rational Rose. We believe this solution can be applied with success to implement transformation of RUP analysis model to a design model according to architectural patterns.

5 REALIZATION OF THE TRANSFORMATION

5.1 Characteristics of the implementation environment

Direct model manipulation and Rational Rose is chosen as the most suitable mean of performing the transformation. The implementation of the transformation base on the APIs provided by Rose (Rational 2001).

Rational Rose is designed to enable extending its capabilities. This can be achieved in several ways. The essential are: menu customization, writing Rose Scripts, executing Rose functions from within another application through Rose Automation object, access to Rose elements (i.e. classes, properties, methods) by including Rose Extensibility Library in any COM compliant environment, and writing extensions which are called *add-ins* (the same as *plug-ins* in some other environments).

Rose is itself component based and is defined in the Rose Extensibility Interface (REI) Model. "The REI Model is essentially a metamodel of a Rose model, exposing the packages, classes, properties, and methods that define and control the Rose application and all of its functions" (Rational 2001). Therefore access to REI gives powerful mean to perform model transformations. Using the REI, all Rose elements can be manipulated. REI could be accessed in two ways, as presented on the Figure 19

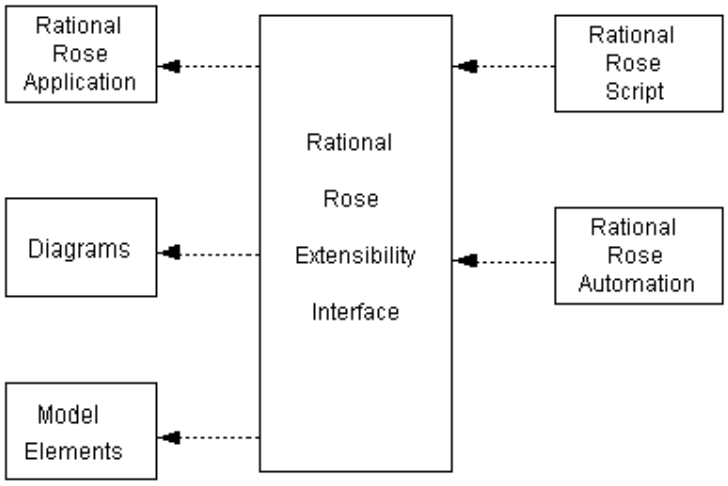

Figure 19. Context of the Rational Rose Extensibility Interface, source (Rational 2001)

As aforementioned, one can communicate with REI through Rose Scripts or Rose Automation. Both ways lead to usage of the same API. REI API is described in the online Help provided together with Rational Rose. REI is used to manipulate the components which are on the left side of the Figure 19. They are: Rose application, diagrams, and model elements. These three components offer access to almost whole Rose functionality.

Rose provides an environment to write, test, debug, and compile scripts. The Rose Scripting language base on the *Summit BasicScript* language (therefore it is very similar to *Microsoft Visual Basic*) but some extensions are introduced. The additions to the Rose Script include special classes and functions to manipulate the model (e.g. class *Class*, class *Category*) and Rose itself (e.g. class *RoseApp*).

Rose Automation is intended to enable interoperability between Rose and other applications in Windows environment. According to (Rational 2001), Rose can act as

an automation controller, “you can call an OLE automation object from within Rose script”, e.g. execute functions in Word or Excel. Additionally, Rose can act as an automation server, “you can call its OLE automation object from within other OLE-compliant applications”. It is possible to use Rose through automation controller implemented in Visual Basic, C++, and other COM compliant environments.

The last concept worth mentioning is the add-in architecture. The add-ins concept is similar to plug-ins in other environments (e.g. plug-ins in the Eclipse platform). It is possible to implement some functionality using Rose Script or Rose Automation, combine it with menu customization, properties enhancement, reactions for events, and other optional elements and provide a single installable package. Add-ins could be seen as additional, independent components which can extent Rose functionality. The number of ways of implementing add-ins and their potential functionality is so large that is out of scope of this paper to describe them. The following Figure 20 outlines the architecture of add-ins.

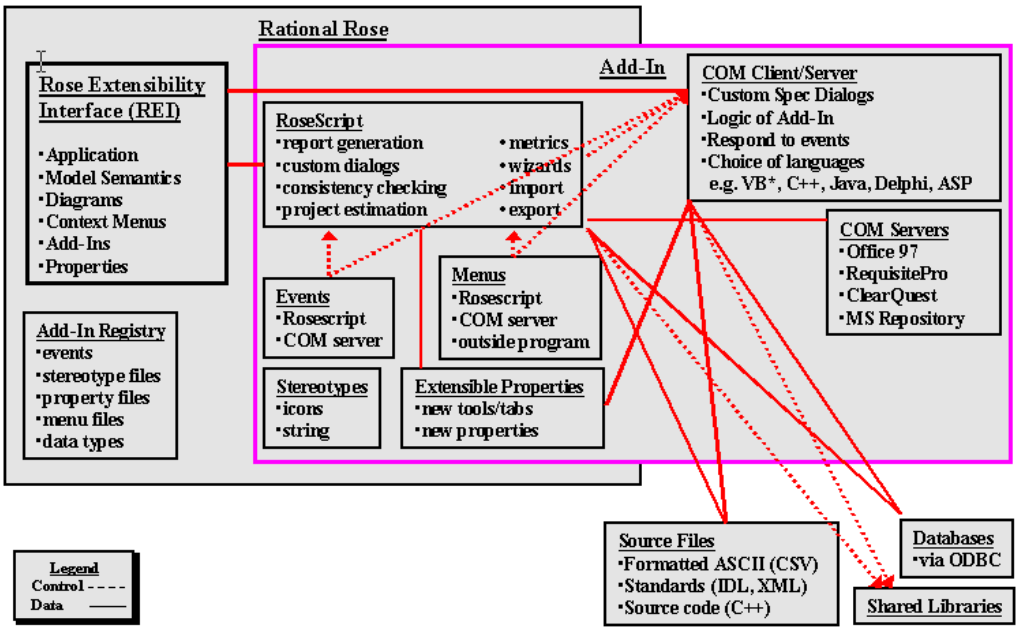

Figure 20. The Rational Rose Add-in architecture, source (Rational 2001)

As presented on the Figure 20, add-ins use REI in order to perform some operations. Add-ins could be implemented in Rose Script language or as COM Clients/Servers. They can define its own responds for the events that Rose generates, customize menus, define new stereotypes, extend the properties of the Rose elements (e.g. add new properties to classes in the model), and add an own help system.

The more detailed specification of REI, add-ins, and Rose Script, as well as some exemplary code is provided in Rose online Help and in (Rational 2001).

Developed transformation tool makes use of the add-in architecture. Details are presented in the remainder of this chapter.

5.2 Design of the transformation tool

5.2.1 Functional requirements

Functional requirements of the transformation tool base on the need of performing considered transformation. When we have analysis model developed, we want to transform it into design model according to an architectural pattern. Therefore, before the transformation can take place, architectural pattern should be defined. It should be defined according to depicted transformation language. It is also important to gain benefits of reusability – once developed solutions can be shared be many development projects. It is desired to have ability to export and import already defined architectural patterns.

The developed transformation tool would be a prototype rather than complete transformation system. It is possible to implement only the basic features in the considerable amount of time. However, the implemented set of features must enable to perform the experiment (Chapter 6) verifying usefulness of the tool.

The initial name of the transformation tool is *ArchiTransAB*.

5.2.2 Quality requirements

There are several essential quality (non-functional) requirements which were taken into account during design of the transformation tool. The quality requirements are discussed in order of importance.

Usability is the key quality attribute of the transformation tool. It concerns such issues like response times, GUI design, easiness of learning and general comfort of use. This is crucial feature since usability has large impact on the way that users deal with the tool. The easiness of use is the key to success in the evaluation of the tool (Chapter 6).

Maintainability concerns the ability of the system to introduce changes and corrections. It requires that system is easy to understand, easy of testing, easy of changing. The importance of this feature comes from the fact that changes in any software are indispensable, especially in prototypes and initial versions of the software. If it was hard to bring in any changes, the system probably would not be maintained and developed in the future.

Another important quality requirement is high flexibility. Flexibility is about easiness of extending the functionality, augmenting the scope of the tool. It is essential because the prototype is intended to be extended in the future.

Performance concerns the general efficiency with which the system works. It is certainly not the key quality attribute of the system, more important are the aforementioned. The only requirement regarding performance, is to keep it at reasonable level. The transformation cannot last longer than 10-20 seconds (depending on the size of the transformed models).

Portability is about the ability of the system to migrate to new operating environments. Since Rose is available on different platforms (Windows, UNIX), it would be desirable to have possibility to run the transformation tool on all of these platforms as well.

5.2.3 Analysis

Analysis is simplified to large extent due to small size of the transformation tool. The UP analysis workflow described in Chapter 2 is considered as too complex to be

applied here. The usage scenarios (similar to use-cases) seem to be appropriate technique for such simplified projects.

Performing transformation

1. User starts the transformation using Rose menu items.
2. System displays welcome screen and ask the user to indicate in a model the following folders (packages):
   - input for the transformation (analysis model)
   - transformation rules (architectural patterns)
   - where to put outcome (design model).
3. If there are more than one architectural pattern the user is asked to choose one architectural pattern
4. The transformation is performed

Defining a Pattern

Defining a pattern allows a user to define an architectural pattern, according to defined architectural language (see Section 5.3). The pattern will be processed during transformation from an analysis model to a design model.

Sharing Patterns

Sharing patterns consist of two separate parts: importing and exporting.

1. User chooses one or more architectural patterns to export and saves the patterns definitions on the hard disk.
2. User chooses a file from hard disk and import one or more architectural patterns definitions into the model.

Obviously, every scenario has a flow when error handling is considered.

5.2.4 Design

Implementation Approach

The transformation tool can be implemented in several ways. The implementation approach makes large impact on the design since some environments are object oriented, some are structural, some are mixed. There are two main approaches (see Figure 19), Rose Script and Rose Automation. There are also a few ways of implementing the Automation – several programming environments could be used. Due to author's abilities, only Java and Visual C++ were taken into account. It is possible to use Rose Automation through Java because IBM Rational Company provides special libraries (*java2rei.jar* and *java2rei.dll*) that make Java able to employ COM object. However, such compliance has some shortcomings. The errors notification is weak; Java program does not get information about the type of the error. Additionally, some functions do not work well; for example, there are problems with closing the Rose application after executing Java program. Furthermore, GUI has to be implemented in Java, which is different from Rose GUI. Moreover, any changes in the program require additional tools and knowledge from developers. Programs are also not portable, they can run only on Windows platform.

Microsoft (MS) Visual C++ has also some disadvantages. Similarly to Java, transformation tool has to be implemented in external environment what decrease maintainability; GUI is not the same as in Rose and programs are not portable.

Additionally, code in Visual C++ is considerably longer because it requires more implementation details compared to other methods (for, for example, putting into work Automation controller). MS Visual C++ has considerable advantage – high performance. However, performance is not the key quality requirement (see previous section).

Finally, after considerations and series of tests, Rose Script has been chosen as the most appropriate approach. This approach could fulfil quality requirements in the most suitable way. Firstly, it is characterized by better usability. Users of the transformation tool already have been using Rose; therefore they are familiar with Rose GUI and menu systems. Rose Script allows to define the same GUI elements as Rose uses. Secondly, maintainability is higher since no additional environments are needed to make changes the code. Performance was tested by execution several scripts which confirmed that Rose Script fulfil performance requirement. Finally, portability is higher since Rose Script runs on the same platforms as Rose does.

Core elements

The implementation environment influences design of the transformation tool. Since Rose Script has been chosen as the most suitable implementation approach, the program is constructed in structural form. Rose Script, as a scripting language, does not support object oriented techniques.

The developed program consists of a set of procedures and functions. The main procedure (*Sub Main*) gets an object *RoseApp* that is used to manipulate Rose application and perform operations on the currently opened model. The essential operations are obtaining collections of classes or packages (packages are named Categories in REI), finding proper elements, performing operations on these elements (changing properties, deleting from model, adding new elements, etc.).

The outline of the program performing the considered transformation is presented on the Figure 21. The figure contains procedures hierarchical diagram.

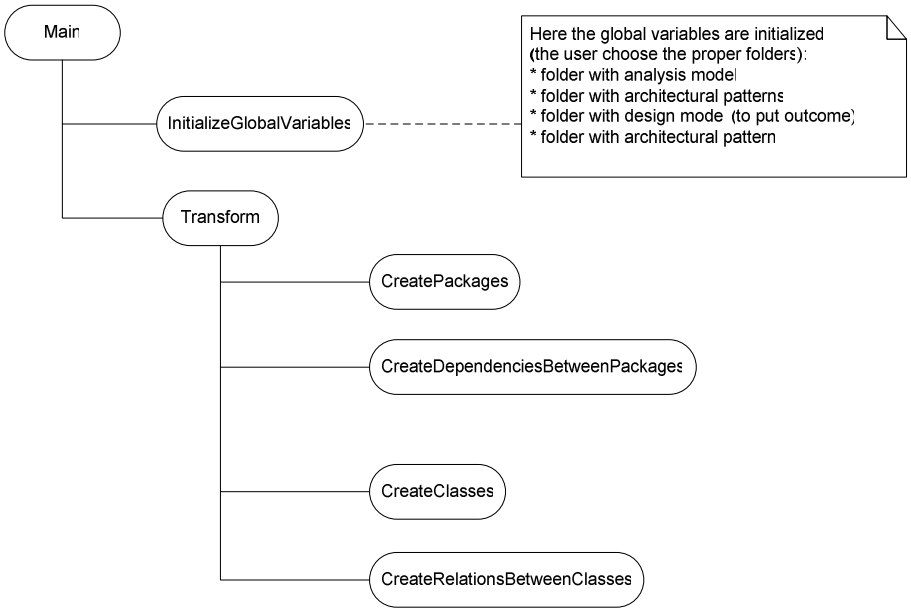

Figure 21. Outline of the structure of the program performing considered transformation

Figure 21 presents the diagram containing structure of the transformation program. Since the program is implemented in scripting language, there are not any forks. The flow is straightforward; the procedures are called one after another. *Main* procedure calls two subprocedures: *InitializeGlobalVariables* and *Transform*. *Transform* in turn deals separately with packages and classes. Initially it creates packages, afterwards dependencies between them. Accordingly, *Transform* deals with classes.

Creation of relationships between classes is the most complex procedure. It requires taking into account relationships from analysis model and from architectural pattern definition and creating a kind of combination of them. It is presented in the depiction of transformation language below (Section 5.3).

5.2.5 Implementation

The transformation tool was implemented in Rose Script using the internal Rose script editor. GUI dialogs were also developed using built-in Rose component (*Dialog Editor*). The program was implemented according to abovementioned design. We managed to satisfy the key quality requirements. This was achieved not only by choice of the implementation environment (Rose Script), but mainly by careful design and implementation. The usability is augmented by the GUI design which is in accordance with Rose GUI. Maintainability is augmented by readable names of procedures and variables, top-down structure of procedures, comments in the code, and several special code constructions. The same features increase flexibility. Portability is accomplished by usage of Rose Script; program could be run on every platform on which Rose is implemented.

The following Section 5.3 contains details of the implemented transformation language.

Program may be delivered in a form of one file – *setup.exe* which installs all needed elements (adds to menu an item executing program, updates registry, extend properties of packages, classes, relationships).

5.2.6 Verification and Validation

According to Sommerville (2000), the verification issues concern the fact whether the software conforms to its specification. It was investigated by several tests. Program was executed with test data (several analysis models, and several architectural patterns).

The validation process answers for the question if the software does what the user really needs. Validation was hard to perform since the developed system was not a typical bespoke system. However, the requirements and the matter of fulfilling them by developed tool were consulted with experienced UML modelling teachers from two universities: Bogumiła Hnatkowska (Wrocław University of Technology) and Ludwik Kuźniarz (Blekinge Institute of Technology).

Verification and validation were also the key issue in the experiment with human subject, where several participants tested the system, see Chapter 6.

5.3 Transformation language

In order to perform transformation two aspects have to be considered. Firstly, how to define architecture schema, and secondly, how to specify transformation rules. In other words, a transformation language has to be defined.

In the following sections, the transformation language is described in details.

5.3.1 Expressing Architecture Schema

An UML model was found as the simplest way of defining architecture schema. Only some UML elements are valid on the schema: packages, classes and relationships between them.

Packages

Standard UML packages are means of grouping and partitioning. They can be nested to as many levels as the tool (i.e. Rose) allows. The only allowed relationship between packages is dependency. Packages contain classes. Top-level packages represent layers – only different kinds of layered architecture are considered as valid architecture schemas (Chapter 3). On the following Figure 22 an example of two-layered architectural pattern is presented. Packages represent *Client* layer and *Server* layer. *Client* layer is dependent on *Server* layer (i.e. it uses functionality provided by Server).

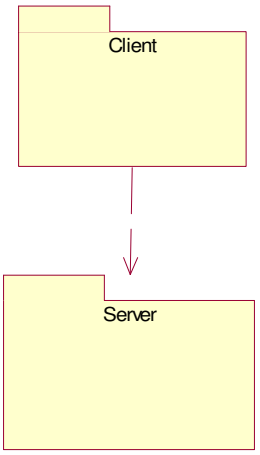

Figure 22. Example of the two-layered pattern

Classes

There can be used standard UML classes. Classes play the main role in defining transformation rules. Two main groups of classes can be distinguished. Classes from the first group are these that will be directly mapped to the design model. Classes from the second group are auxiliary classes. They are used to express transformation rules and are not mapped directly to the design model.

Relationships

The following relationships can be used in the definition of architecture pattern. They are processed by transformation plug-in. The semantics of the relations is in accordance with UML specification. Example of the allowed relationships is presented on the Figure 23. They are:

- Associations

- Dependency (dependency relation with stereotype *trace* is allowed only to express transformation rules, details in the following section)
- Generalization

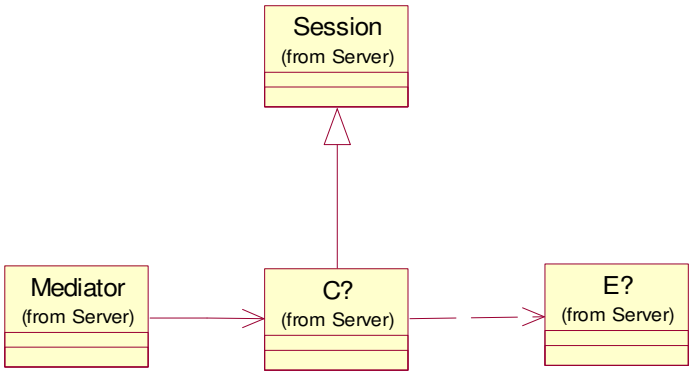

Figure 23. Examples of allowed relations between classes: association, generalization, and dependency

The syntax of the transformation language can be expressed in a metamodel. It is presented in Appendix C. The transformation language metamodel is essentially a subset of Rose metamodel. Therefore, a metamodel of the transformation language was developed basing on the Rose metamodel presented in the Rose online Help.

5.3.2 Expressing Transformation Rules

Packages

Every package from a definition of architectural pattern is created in the output folder. The only relation between packages which is processed by the tool is dependency, regardless its stereotype, name and documentation. The additional packages can come from analysis model if analysis classes were grouped into analysis packages. Analysis packages can be maintained or discarded; a user decides about that by adjusting special properties, default is to preserve analysis packages. This is further discussed in the Section *Special properties* below.

At the beginning of the transformation, an initial design model (layers and packages) is created basing on the packages from architectural pattern.

Classes

As aforementioned, from the point of view of the transformation, two types of classes are distinguished. The first group – classes which are directly mapped to the design model – we will call as classes of type A.

The second group – classes that are means for expressing transformation rules (auxiliary classes) – could be easily found since they are related to each other by dependency relationship with stereotype *trace*. Both the client and the supplier of every dependency relationship with stereotype *trace* are auxiliary classes.

Because client and supplier of the dependency relation *trace* play different roles in the transformation process, we further divide auxiliary classes into two groups. Clients are named as classes of type B and suppliers are named as classes of type C.

Summarizing, we can define three groups of classes (each group is named with a subsequent letter from alphabet):

- **A** – without any dependency relation with stereotype *trace*
- **B** – as a client for dependency relation with stereotype *trace* (in other words, the class is *traced* from another class)
- **C** – as a supplier for dependency relation with stereotype *trace*

On the Figure 24 below classes of these three types are showed. Class *Mediator* is an example of class type A, class *Facade* of class type B and *NewClass5* of type C. Class *Facade* and *NewClass* are the auxiliary classes because they are related to each other by the dependency with stereotype *trace*.

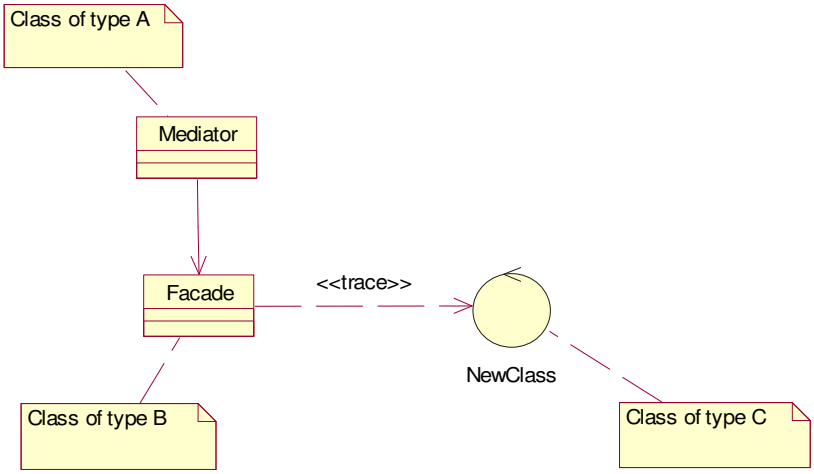

Figure 24. Example of different types of classes

Before it comes to classes' transformation, there already exists initial design model with layers and packages (as described above).

First type of classes – A (without any dependency relation with stereotype *trace*) – is transformed to the design model without any changes. Appropriate design classes are placed in the design model in the same way that classes of type A are placed in the architectural pattern definition. Every property of a class of this type is placed in the new class in design model as well.

The classes from the analysis model to the design are transformed according to the following rules.

We need to define two things before transformation could be performed. Firstly, which classes we take from the analysis model. Secondly, where we want to put them in the design model. Classes of type C are used to indicate which classes from analysis are taken and classes of type B are used to indicate where they are put in the design model. This is presented on the Figure 25.

Since we distinguish only three types of classes in the analysis model (Section 2.2) – *boundary*, *entity*, *control*, there are only three types of classes of type C in the pattern definition. The only feature of the classes of type C taken into account during transformation is their stereotype. Classes of type C can have, therefore, only *boundary*, *entity*, or *control* stereotype.

Worth to emphasize is the fact that the properties of the new classes in design model are a combination of properties from two classes on which it is based: an analysis class and a patterns class of type B. However, the transformation of properties can be adjusted to some extent by *Special properties*, this is described in details below.

New class gets a name based on the pattern class of type B in the following way. Instead of character '?' the name of appropriate analysis class is put. The following

three figures present example of transformation rule (Figure 25) applied to the analysis model (Figure 26) and the result of the transformation (Figure 27).

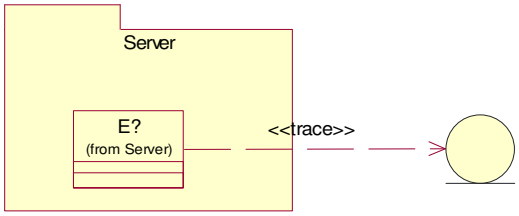

Figure 25. Example of the transformation rule

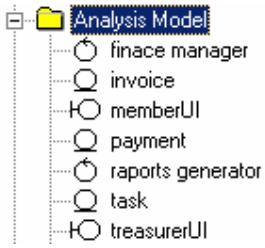

Figure 26. Example of the analysis model (source for transformation)

Figure 27. The rule from Figure 25 applied to above analysis model resulted in this initial design model

There are some constraints when it comes to put the classes into the definition of an architectural pattern. The constraint bases on the type of the class. The classes from group C have to be placed in the root folder of the architectural pattern (i.e. out of any package which represents a layer). The rest of the classes (from groups A and B) have to be put inside some packages that represent layers in architectural pattern.

Figure 28 presents a package *Client-Server* which contains a definition of an architectural pattern. Classes are distributed in the following way. Classes of type C (*NewClass1*, *NewClass2*, *NewClass3*) are out of any package but classes of type B (*Service1*, *Service2*, *Invoker*) and of type A (*Mediator*) are inside the packages.

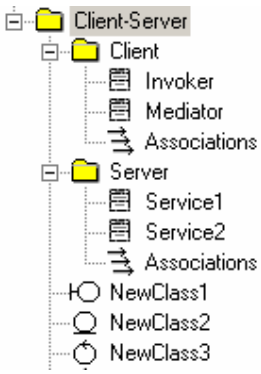

Figure 28. Example of proper distribution of classes into the packages in an architectural pattern

Relations

Three abovementioned types of relationships are taken into account during transformation process. When transforming associations several different cases can be distinguished.

Associations between classes of type A are created in output model in the same way as they exist in the definition of an architectural pattern. Associations between a class of type A and a class of type B is transformed in the following way. In the output folder the class of type A will be associated with all classes which are created basing on class of type B. The following figures illustrate this.

On the Figure 29 a class of type A (*Mediator*) is associated with a class of type B (*E?*). Because the class of type B (*E?*) is traced from a class with stereotype *entity*, it is replaced in the output model by three classes (since analysis model contains three *entity* classes, Figure 26). Therefore, after the transformation class *Mediator* (type A) is associated with three new design classes (*Einvoice, Epayment, Etask*) which base on the class *E?* (type B) (Figure 30).

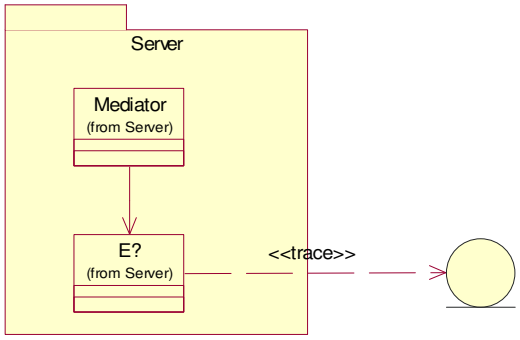

Figure 29. Example of transformation rule with associations between class type A (Mediator) and class type B (E?)

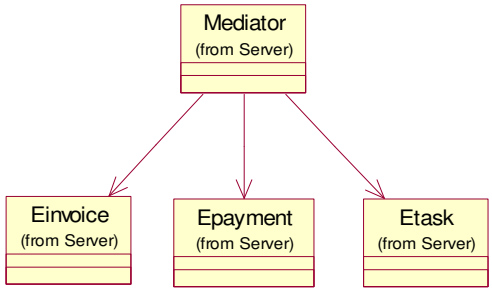

Figure 30. Result of the transformation, source association is multiplied

The third kind of transformation of associations concerns associations between two classes of type B. This is a special case because for every class of type B many classes are created in design model. Therefore, an association between two classes of type B means that every design class created basing on the first class is associated with every design class created basing on the second class of type B. Figure 31 illustrates an architectural pattern with two classes of type B associated to each other.

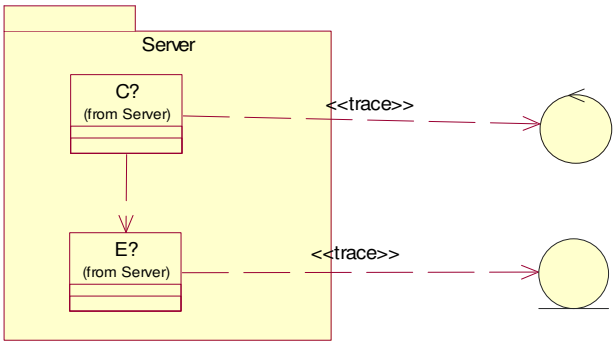

Figure 31. Example of transformation rule with associations between two classes of type B

Figure 32 presents result of the transformation. Every class which is created basing on the class *C?* is associated with every class which is created basing on the class *E?*.

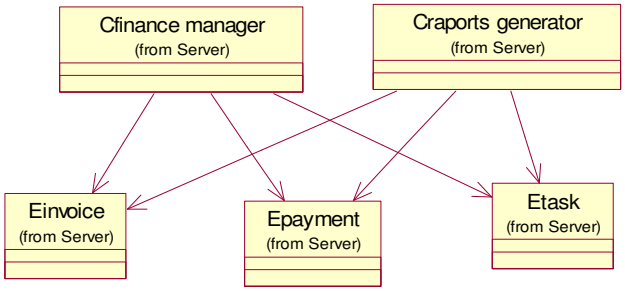

Figure 32. Result of the transformation according to above transformation rule

What can be easily seen, this result of the transformation is usually unintended. Therefore, such transformation rule is ignored as default. However, this setting can be changed by adjusting special properties of the association (details in the section *Special properties* below). Moreover, there is also one specific case in transformations of this kind when association exists between two classes of type B which are traced from the class of the same stereotype. In this case, desired design model could be created in a way that two output classes, which are based on the same analysis class, are associated to each other. This is default setting but it may be easily changed by adjusting special properties of the association (details in section *Special properties* below). The Figure 33 and Figure 34 illustrate this case. Classes *E?* and *Helper-?* are traced from a class with stereotype *entity*. The transformation leads to design model with classes associated in pairs, according to their origin. For example, classes *Einvoice* and *Helper-invoice* are associated because they are transformed from *invoice* analysis class.

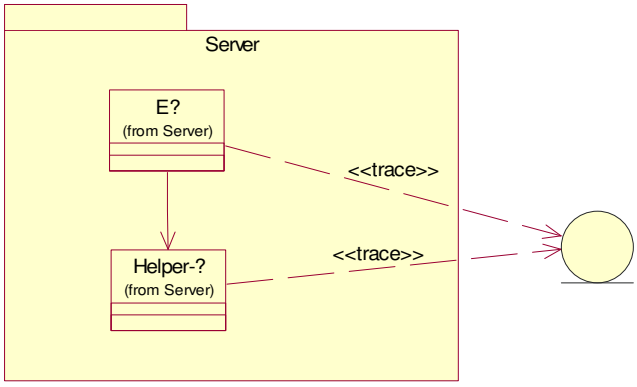

Figure 33. Example of transformation rule with two classes of type B traced from the same class

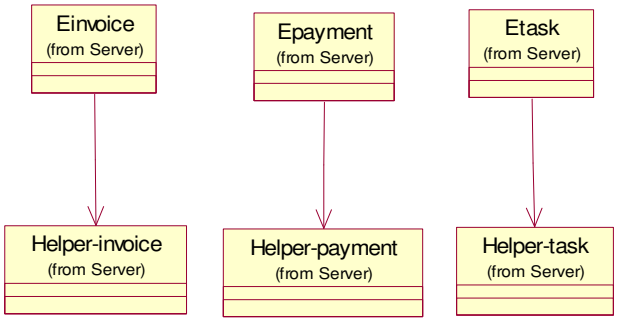

Figure 34. Result of default transformation

Associations, which exist in analysis model, are put into the output model (between the classes that are created basing on appropriate analysis classes), unless it is deliberately changed in special properties of the class.

Dependency and generalization relations are transformed according to already described rules for associations. The only exception is the fact that dependency relations from analysis model are not taken into account.

Special properties

Rational Rose gives possibility to set some meta-properties to classes, relations, and packages. They can be viewed and changed by highlighting an element and choosing from context menu (right key of the mouse) *Open specification* item. A dialog appears where several tabs can be found. One of them is *ArchiTransAB* tab (after the plug-in is installed) which contains properties used by transformation plug-in. The properties in the tab *ArchiTransAB* are discussed according to type of the element.

Classes contain three properties and each of them can have one of two or three values (Table 2). When a class has set the property *Analysis Relations* to default (*Keep all*), all of its analysis relations are transformed to design model. This is sometimes unintended, for example when we want to put a mediating layer between domain and data source. The relationships from analysis model should be in this case discarded. It is achieved be setting property *Analysis Relations* to *Discard all relations*. This property can have one more value – *Keep only inside a package*. This means that only inside a package analysis relations are held. This allows for, for example, keeping

relations between entity based classes (they can be put into one package *entity*) but discard relations outside the package (to control based classes).

Property *Keep Analysis package* allows to adjust whether we want to maintain analysis packages. Default is *true* since analysis packages are often mapped to subsystems (so packages in design model) (Jacobson et al. 1999).

The last class property – *Keep Analysis properties* – allows to decide if we want to inherit properties (attributes, operations, etc.) from analysis class or not.

Table 2. Properties and values of the *class* element

Property	Value
Analysis Relations	Keep all relations (default) Keep only inside a package Discard all relations
Keep Analysis package	True (default) False
Keep Analysis properties	True (default) False

These properties are useful only for classes of type B because they are based on analysis classes. All the properties and relations which come from analysis model can be adjusted only by the pattern class specification.

Relationships in a pattern definition have only one property with three possible values (

Table 3). Default is *Smart create* because this is the probably most commonly used for relationships between classes of type B (this is described above, Figure 34 illustrates this). The *Create all* value means to create all potential relationships what is often unintended (described above, Figure 32 illustrate this). There is also the possibility to skip the relationship and not include it into design model (*Skip it* value).

Table 3. Properties and values of the *relationships* elements

Property	Value
Create Relations in output	Create all Smart create (default) Skip it

5.3.3 Sharing Architectural Patterns

When some Architectural Patterns are defined in a model, there is a need to share them with other models to gain the benefits of reuse. This need is just the problem of importing and exporting part of the model. There are several ways of exporting and importing models and parts of the models in Rational Rose. The recommended way is by using control units. This mode is described below in details.

Exporting

- To export a pattern (patterns), a folder with architectural patterns should exist (usually *Architectural Patterns* in *Logical View*)
- In the tree view of a model in Rational Rose highlight a folder with architectural patterns (usually *Architectural Patterns*) or a folder with one of the architectural patterns, e.g. *Client-Server*

- From context menu (right key of the mouse) or from menu *File* choose option *Units* / *Control*
- Enter the name of a file which will store a set of patterns or one of the patterns and click *Save*

Importing

- In the tree view of a model in Rational Rose create a folder *Architectural Patterns* or create appropriate folder for one pattern (e.g. *Client-Server*)
- Highlight this folder, then from context menu (right key of the mouse) or from menu *File* choose option *Units* / *Control*
- Enter the name *Temp* and press *Save*. This file is almost empty because it stores empty folder, it will not be needed any more
- Highlight this folder, then from context menu (right key of the mouse) or from menu *File* choose option *Units* / *Reload*
- In dialog box select a file that should be imported
- After that a dialog box appears to save a folder before reloading, *No* should be chosen

5.4 Conclusions from developing

Developed tool fulfils its requirements. However, there are foreseen several improvements which could be included in future versions. The most obvious are:

- Automatic creation of traceability diagrams. These diagrams show which design elements are traced from analysis elements. Traceability diagrams are required is some type of projects.
- Support for iterative and incremental development method. This includes possibility to update architectural pattern, analysis model, or design model after they are used in transformation. When design model there is already developed, desirable would be to perform next transformation (with updated analysis model, design model, or architectural pattern) that seamlessly integrates with previous transformation. For instance, in subsequent iteration an analysis package is added. The package should be transformed into subsystem according to architectural pattern and added into existing design model.
- Support for more sophisticated architectural patterns. Although it is possible to express the basic architectural patterns, it is foreseen that other patterns would be required and more flexible transformation rules.

Although the goal was achieved, developed tool – *ArchiTransAB* (the add-in for Rational Rose) allows to perform the considered transformation from RUP analysis model to design model according to architectural patterns, there were elicited some shortcomings of chosen approach. API provided by the modelling tool constraints the number and types of operations. Some operations are not allowed. There are also some inconsistencies, e.g. relationship *dependency* is sometimes called *uses* relationship (during defining additional properties).

The chosen implementation technique – Rose Script has also some shortcomings. It does not support Object-Orientation and lacks some useful constructions, e.g. it does not contain collections with built-in sorting (like *TreeSet* in Java). Nevertheless, the considered transformation was developed according to its specification.

However, design and implementation of the tool is insufficient to address the main aim of the thesis (verifying if applying automatic UML models transformation

approach can improve the work of an architect in Rational Unified Process). Therefore, the developed tool is further investigated in an empirical study. In the following chapter, the experiment with human subject is depicted. It validates the transformation tool and shows that it can be used in an experimental environment with certain results.

6 EXPERIMENT

6.1 Introduction

In the previous part of the thesis, the problem of model transformation is described and discussed. The most appropriate approach to perform defined model transformation is chosen (direct model manipulation with Rational Rose) and a tool that applies the approach is developed (ArchiTransAB). According to Dawson (2000: 16), regardless of kind of development project is tackled, a researcher is expected to include critical evaluation of the work. This chapter contains an experiment with human subject which is the evaluative part of the thesis.

The field of the investigation is a usage of the developed tool. Directly, only the developed tool is evaluated; however, some conclusions on approaches to UML model transformations are stated as well. The intention of the experiment is not to verify technical abilities of the tool but to investigate if changing model transformation technique from manual to automatic (in this specific implementation of the specific approach) improves the efficiency of work of a software architect.

The experiment has been chosen as the most appropriate research method because of two main reasons. Firstly, it is an empirical method. Empirical methods, according to Wohlin et al (2000), are proper to evaluate new methods, techniques, tools, languages, and parts of the process, and therefore excellently fit into software engineering. Furthermore, empirical studies are crucial in the evaluation of human-based activities, and when there is a need to evaluate the use of software products. Wohlin et al. (2000: 3) claim, “Experimentation provides a systematic, disciplined, quantifiable and controlled way of evaluating human-based activities”. We need to evaluate new tool but when it is used by humans. The human perspective is so important in software engineering because even if a product is technically perfect, but nobody uses it, it is not very valuable. Secondly, experiment is relatively easy to perform. It requires laboratory environment only and experiment provides high level of control. Other research methods could be also appropriate. It would be desirable, for instance, to carry out a case study performed on an on-going development project. However, due to time and budget constraints, experiment study has been chosen as the most appropriate approach. Experimentation in software engineering is further discussed in (Wohlin et al. 2000).

6.2 Problem statement

The study is based on the initial hypothesis that it is better to use automatic UML model transformation approach rather than manual. However, as aforementioned, there are several approaches to automatic UML model transformation. Additionally, every approach can be implemented in several ways. In the previous chapters it is described how one approach is chosen and how it is implemented. The experiment concerns only this particular implementation. Nevertheless, we can provide evidence for the fact that applying particular approach may lead to increase of efficiency in model transformation task. The selected approach gives some opportunities and puts some constraints when it comes to its implementation. Every developed tool that applies an approach uses some of its opportunities and respects all of its constraints. If our tool is successful, other, developed in a similar way, can be successful as well.

It has not been investigated whether introducing automated model transformations indeed decreases development time and thus costs. Although it can automate some

tasks – therefore make it faster, it requires additional work from software architect (express transformation rules, learn to use new techniques and tools). We have to consider if advantages are overwhelming shortcomings. The main shortcomings are: the need to learn more (how to use new tool) and the need to spent considerable amount of time with no direct results, but just developing additional artefacts. It is required to arrange some transformation schemas and rules before it comes to perform the transformation.

The experiment is performed in an academic environment since this is an initial study. The disadvantages of using students can be balanced by the low cost involved in the experiment. Investigations in real companies require more time and money and can be performed in further studies. However, students are performing tasks of a software architect. Thus, experiment is intended to take the software architect point of view. We have six subjects in the study. They are a group of volunteers from the population of software engineering students at Wrocław University of Technology. Due to the lack of randomization of students, the study cannot be judged as being controlled experiment but quasi-experiment (Wohlin et al. 2000: 43).

The main effect investigated in the experiment is the efficiency of using developed tool facilitating UML models transformation. The investigated aspects are total time needed to perform the transformation and total number of faults (quality).

Details of the considered transformation, the needed source and allowed destination models are described in the previous chapters.

Concisely, the definition of the experiment can be formulated in the following way. Analyze model transformation tool for the purpose of evaluation with respect to their effectiveness from the point of view of the software architect in the context of M.Sc. students performing UML model transformations.

At the end of the experiment, subjects are asked to fill in a questionnaire. The questionnaire is intended to elicit information about the level of knowledge in using RUP. Additionally, the participants are asked some questions to get qualitative data concerning their personal reflection on using developed tool. The answers are not part of the experiment or any formal investigation, however, they contain useful remarks which complement the conclusions.

6.3 Experiment planning

6.3.1 Context

The experiment is set in the context of several M.Sc. students at Wrocław Univerity of Technology. The students are in their forth and fifth year at the university (graduate). All of them are on master course in Software Engineering (SE). They have required skills in UML modelling because they have finished course *Software Systems Development* (theory and practice). Therefore, they should be familiar with Rational tools and with foundation of software architect's tasks within RUP (this knowledge is needed to finish the mentioned course). Before agreeing to take part in the experiment, they have confirmed that they had required knowledge. Moreover, all of the participants fill out a questionnaire which verifies their competence in RUP.

The subjects are chosen by convenience since they are volunteers. Students of the fourth and fifth year were given an announcement which encouraged them to take part in the experiment. Although they are a sample from all Software Engineering (SE) students, they are not a random sample. They are probably more familiar with UML and software design than the average of SE students. They have confirmed that by agreeing to participate in the experiment, because prerequisite was good knowledge of RUP and Rational tools. In the experiment, students are playing the role of software

architect in RUP during system design. They perform transformation of analysis model to design model according to architectural patterns.

The experiment concerns specific problem of performing UML model transformations. It addresses a real problem, i.e. usefulness of the developed tool.

Objects of the study are UML analysis model and definitions of target architectures. Any analysis model can be transformed to design model according to any architecture pattern (see Chapter 2 for details). The analysis model is taken from real students' project. It concerns a domain of operation and finances of a students' organization. The model is simplified to large extent due to time constraints. In other case, a transformation could last too long. The model consists of seven analysis classes and their relations. We consider one analysis model but two architecture patterns. One relatively simple pattern is Model-View-Controller (MVC); it is discussed in Chapter 3 and in (Fowler 2002). The second, more complex pattern is PCMEF; it is discussed in Chapter 3 and in (Maciaszek and Liong 2004). All the supplied materials to the participants are included in Appendix A.

6.3.2 Hypotheses

The experiment tests a null hypothesis. There is defined also alternative hypothesis. They are formulated in the following way:

- Null hypothesis (H_0): there is no difference in effectiveness of transformation of analysis model to design model between manual and tool supported approach
- Alternative hypothesis (H_1): introduction of tool support in considered transformation of analysis model to design model improves effectiveness

What is meant by *effectiveness* is presented in the following section.

6.3.3 Experiment variables

There is one independent variable which is manipulated – UML model transformation technique. The treatments are manual and automatic (tool supported) approach.

Two depended variables are the measure of effectiveness of the used technique in model transformation:

- Total time (TT) – the time in minutes which was required to perform the transformation
- Total number of faults (TF) – number of faults in target design model after transformation

A transformation is performed in more effective way when it is performed faster and the number of faults is not bigger. Detailed description of depended variables and the way of measurement is discussed at the end of subsequent section.

Type of the architectural pattern could be considered as the next independent variable because two different patterns are tackled in the experiment. However, it was introduced to reduce threats to conclusion validity. It is not investigated what is the relation between these two patterns and how this relation influences depended variables. This could be a field for investigations for another study. Only some general conclusions are stated. If the results are valid for two patterns, it is more likely that they are valid regardless type of the pattern. More details about threats to validity are presented in Section 6.3.6

6.3.4 Experiment design

The experiment consists of a set of trials that are combination of treatment, subjects and objects (Wohlin et al. 2000: 34). The treatments are two model transformation approaches: manual and tool supported.

The type of the experiment design is paired comparison design (Wohlin et al. 2000: 55). It means that each subject uses both treatments on the same object. The motivation for that kind of experiment was to find differences in performing the transformation between manual and tool-supported way for each person separately. Thanks to this, we do not have to consider individual proficiency in using Rational Rose, UML, or RUP. If a person performs transformation longer due to unfamiliarity with Rose, he or she will perform the transformation longer in both cases (automatic and manual). However, the order may be significant for the results of the experiment. It may take more time to complete assignment with the first treatment than to complete it with the second because of the learning effect (they get to know the architecture and domain model). To reduce this effect the order is assigned randomly to each subject (when they enter the experimental room one of the two forms is handed randomly). Moreover, we ensure reduction of this effect by providing detailed experiment guideline with description of the desired architecture and oral explanations.

The experiment consists of two series of tests. First set is conducted with a simple architectural pattern (MVC) and the second with a complex pattern (PCMEF+). With both of them, every participant performs the considered transformation from analysis to design using both treatments: manual and tool supported. Participants perform this transformation in random order, some of them firstly manual, some of them firstly tool supported, so there are two different groups. Moreover, every participant has a test when first treatment is manual and second is tool supported. Below, Table 4 and Table 5 present assigning the treatment to subjects (numbers represent order, grey colour represents group 1).

Table 4. Assignment of the treatment to subjects in tests using MVC pattern

Subjects	**Treatment 1 (manual)**	**Treatment 2 (tool supported)**
A	1	2
B	2	1
C	1	2
D	2	1
E	1	2
F	2	1

Table 5. Assignment of the treatment to subjects in tests using PCMEF pattern

Subjects	**Treatment 1 (manual)**	**Treatment 2 (tool supported)**
A	2	1
B	1	2
C	2	1
D	1	2
E	2	1
F	1	2

The tool, as described in Chapter 5, can transform analysis model to design model according to architectural patterns. However, the tool can be used in several different ways, therefore the results could be diverse. The problem is that a pattern, which we would like to use, is already expressed in the pattern language or is not yet expressed. In the first case, to transform means to press barely few buttons in the tool. In the second case, to transform means to spent initially considerable amount of time on defining and expressing the pattern. An in-between case is when we have to modify a

pattern already expressed in pattern language to get desired solution. Due to the fact that in the optimistic case (when we have already expressed pattern) transformation lasts very shortly – few seconds to import the pattern and execute transformation – it is not investigated in the experiment. However, this situation is meaningful for the conclusions and is considered in the summary.

The two depended variables which are measured in the experiment are mentioned in the previous section. Total time (TT) is measured from the beginning of the transformation task – the subjects write down in the form current time. After finishing the transformation task, current time is also written down. The final time minus start time gives total time (TT). Experiment supervisor who checks developed models assesses the second variable, total number of faults (TF). The correct result of the transformation – design model could be easily found since analysis model and architectural pattern determine unambiguously design model. Every fault – every difference to the correct model (e.g. lack of a class, operation, association) is counted as one. Some of the faults are more significant than others, e.g. missing class means more than missing attribute. However, omitted class is counted as one fault. The reason for that is the fact that if there is no class, there is not any of its attributes, operations, relations to other classes, and therefore number of faults is greater. The other faults are counted accordingly, e.g. missing package is a serious fault and all of its classes has inappropriate parent package – number of faults is large.

The number of faults is not considered as the key factor. This assumption is related the specificity of the transformation task. The intention of introducing faults measurement is to ensure that all elements of the transformation are considered. Otherwise, some participants could neglect some elements and thanks to that get better times of the transformations. The key measure is the time of the transformation (TT). Number of faults is measured only to verify if participants transformed the models accordingly and completely. Since the result design model is explicitly presented in the supplied materials (Appendix A) and the models are relatively simple (only 7 classes) it should be uncommon to make faults. As defined in Section 6.3.3, the transformation is performed in more effective way when it is performed faster and the number of faults is not bigger. If the number of faults is very little and meaningless, it will be neglected (only total time will be taken into account). In the other case, relative measure will be considered (total time divided by number of faults).

The essential assumption is also the fact that software architect knows the desired software architecture. Otherwise, the time of performing the transformation could depend on individual experience and knowledge of the architectural pattern. To make sure that the experiment is valid, we ensure the sufficient level of knowledge by providing comprehensive experiment guideline. It contains all the needed information about architecture. The desired architecture populated by classes is explicitly drawn in the guideline. Furthermore, the architectural pattern is explained orally and individually in case of need. The individual explanations are in fact influence of the supervisor on the experiment. This influence may be hard to define and therefore may make the experiment replication more difficult. It may be seen as a threat to validity of the study. However, all additional explanations are carried out according to the guideline which is handed to the participants (Appendix A). The explanations merely clarify guideline instruction and are required in some cases, e.g. when some instructions are not comprehensible enough or simply when participants want to make sure they understood the instructions.

More detailed discussion about threats to validity of the study is presented in Section 6.3.6.

6.3.5 Instrumentation

Measurement instruments are:

- Forms with identifier of a subject, and a table with three columns: identifier of the test, start time, end time of a test (which is filled by the participants itself), and number of faults (which is filled by experiment supervisor afterwards)
- A digital clock which participants can easily see to determine the current time

All the supplied materials to the subjects are included in Appendix A. The next instrument is an experiment guideline. It explains how to perform tool supported transformation. It contains description of the pattern language and some examples. Additionally, experiment process is described in details to make sure everything is clear and everybody knows what to do. Additionally, it allows to decrease number of potential questions. The description of the experiment process contains a list of actions that form the experiment, short reminder of the key Rational Rose features, and definition of the architectural patterns that are used in experimental transformations.

In addition to the guideline, subjects gain short training (demonstration) in the usage of transformation tool. The time spent on getting familiar with the guideline and on training is not being count to the total time of performing transformations. However, this time can be easily found since every participant writes down the time when they start performing the transformation (so the time they finish reading guideline).

Prepared objects of the study are:

- Analysis model. A model prepared in Rational Rose is taken from one of the students assignments. This assignment was done several years ago, so students could not see it before. The domain is activity of one of the student organizations – its operation and finances. The model is simplified be removing some of the analysis classes to make the test shorter. The simplified model with seven analysis classes is considered as being enough complex to get meaningful results. Model is provided in Rational Rose model file (mdl).
- Architecture patterns: MVC and PCMEF in the form of textual description with diagrams on the sheet of paper

Before the real experiment, a pilot study with two participants is conducted. Pilot study should be helpful in determining confounding factors.

6.3.6 Validity discussion

According to Wohlin et al. (2000), experimental study has threats to its validity which can be divided into four groups.

Conclusion validity concerns ability to draw correct conclusions on relationship between the treatment and the outcome. The main threat here is the low sample size and thus not large statistical power. This is due to the little number of volunteers that took part in the experiment. Further studies require larger sample size. Nevertheless, the number of subjects is sufficient to draw meaningful conclusions.

Internal validity ensures that the treatment really causes the outcome. Most of the internal validity threats are addressed through experiment design. Lack of random sampling might be seen as confounding factor, however, sampling by convenience was done in order to find appropriate participants. Only students with good RUP and Rational tools knowledge are considered as suitable for the experiment since the tool is aimed to software architects (not whole population of software developers). It was not possible to make random sampling on the students (we could not make the

participation in the experiment obligatory); moreover it was not easy to find students with good RUP knowledge. Lack of random sampling, therefore, is seen as a threat and may lead to worse external validity but it is unavoidable and do not make the results invalid.

A serious threat to validity of the study is also the fact that every second transformation of the same model with the same architectural pattern may be faster. In other words, when automatic approach is tested on the second place, it may take shorter time because a subject already knew how to perform the transformation and not because tool-supported is better. Therefore, blocking is introduced – one group firstly performs the transformation manually, and the second group automatically. If the time reduction is only the matter of learning effect (or this effect is dominating), it will be easily noted. The threat that in one group there are more UML fluent subjects that in the other is considered as being negligible since randomization is carried out and, what is more important, relative times for every participant are taken into account. The essential are differences between times of manual and automatic transformation for each participant separately.

The influence of supervisor is not seen as a significant threat. Supervisor's role was reduced to clarifying the guideline (Appendix A) and experiment instructions. Negligible amount of additional information were given to the participants.

Construct validity concerns relationship between observation and theory. The main threat here is reliability of treatment implementation. This threat concerns application of a treatment to the subject. The risk is that the implementation in not similar between different occasions, i.e. automation of the model transformation can be various. However, in this study, the conclusions are drawn only about the considered implementation. The conclusions can only state that this particular implementation may lead to improvements and not that any implementation leads to similar results. Moreover, because even single solution (i.e. developed tool) can be used in the experiment in many different ways. We will try to use it in the most standard way. Experiment is run according to the guideline which can be found in Appendix A, therefore potential replication of the experiment is facilitated.

The other threat is the fact that complexity of real-world analysis and design models is not the same as used in the experiment. To minimise this threat, a model from real project was taken. However, due to experiment constraints (time, resources) the model was simplified to large extent. Simplification of analysis model is not seen as a threat. Contrary, automation – in our case – shows better results with large models (analysis model is transformed in the same way regardless of the number of analysis classes and analysis packages). The other issue is considered with design model. It can be impossible to express more complex design models in the proposed transformation language. Although it is possible to express all layered architectures described in Chapter 3 (responsibility-based) and even two of them are used in the experiment, this threat is impossible to eliminate. It requires further investigations on software architectures used in the industry.

External validity is about the issue how general are the findings, if they are valid outside the context. If prerequisites of the experiment are fulfilled (RUP is applied, analysis model is developed, layered architecture is used) and the context is similar (subjects have similar experience to students' experience, models are simplified) the results should be valid. However, to generalize the results to wider population, further investigations are needed. Hypothesis should be tested with less homogenous subjects with professional software architects and in a real industry environment.

6.4 Experiment operation

Two rounds combine the experiment. First one concerns transformation to MVC architecture and the second one concerns transformation to PCMEF architecture. Subjects are divided into two groups. In the round number one, first group start with manual transformation, second starts with tool supported transformation. In the next round the order is inverted.

After preparation of materials, guidelines, and forms for data collection, the experiment has been performed. It was conducted in approximately two hours on a sample of six students. Firstly, students were given the experiment guideline. Then, short introduction to Rational Rose, transformation tool, and modelling issues was carried out. After that, participants performed a simple supervised transformation to train usage of the tool. The next task was to perform the transformation in two rounds by themselves. At the end, the forms filled by participants and the *mdl* files with resulted models were collected. Finally, supervisor checked the models and wrote down in the forms the number of faults found in models.

The initial data validation showed that participants understood forms and filled them out correctly. However, one *mdl* file with a model was corrupted. It did not contain all needed design models. There was not present one result from the second round – the result of manual transformation to PCMEF architecture. Due to inability to count number of faults in the lacking model, the results of this round for this participant were dismissed.

6.5 Data analysis

6.5.1 Time spent on performing the transformations

We use descriptive statistics to visualize the data gathered as a first step in data analysis. Figure 35 contains a bar plot that shows the amounts of time that participants needed to complete transformation task in the first round – transformation to the architecture based on MVC pattern. The advantage of applying tool supported technique can be easily seen. In every pair of results, time needed to perform transformation manually was longer. The mean for manual is 12.3 minutes, whereas mean for automatic is only 7.8. The standard deviation is 4.0 and 3.0 respectively. On average, participants performed the transformation using the developed tool 4.5 minutes faster.

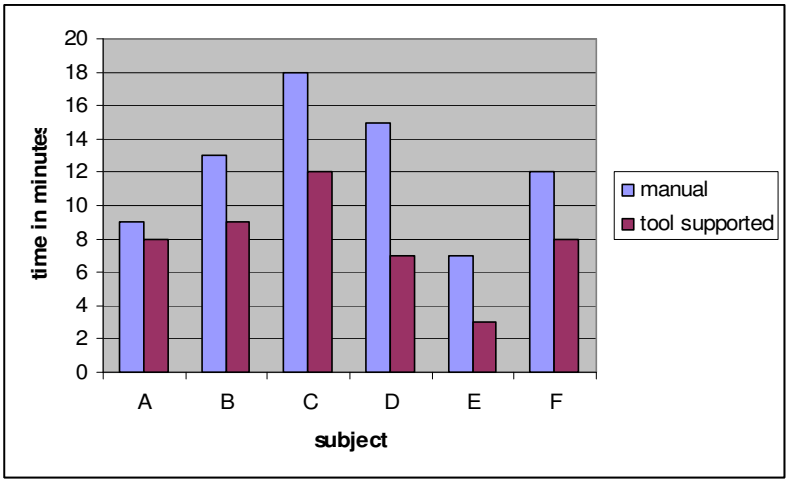

Figure 35. The amount of time needed to perform transformation; MVC architecture

Figure 36 shows analogous results for second round, for transformation to PCMEF architectural pattern. Due to initial data validation, results of one participant are omitted. The mean for manual transformation is 17.6 min. with standard deviation of 4.6. The mean for tool supported transformation is 16.6 with a standard deviation of 7. Because of much larger than others standard deviation, some outliers are foreseen. The outlier could be participant *B* whose result is much different from the others. To gain better understanding of the collected data and to find potential outliers, a box plot is done.

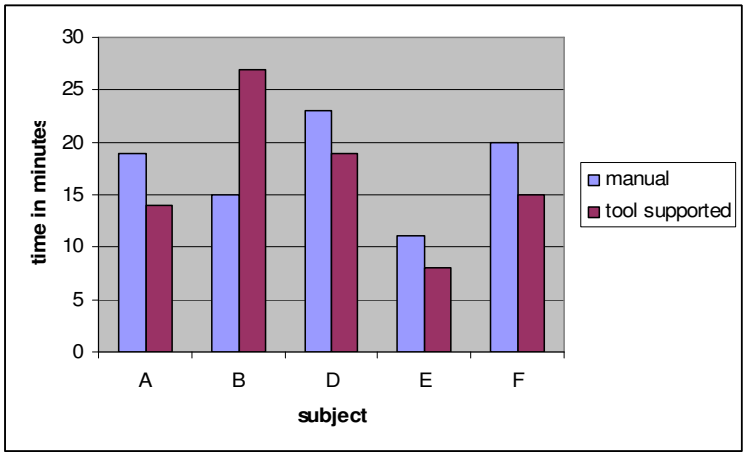

Figure 36. The amount of time needed to perform transformation; PCMEF architecture

The box plot concerns intervals between amounts of time needed to perform the transformation manually and using the tool (in ascending order they are: -12, 1, 3, 4, 5, 5). The box plot, showed on the Figure 37, is constructed according to Wohlin et al. (2000: 88). The needed values are median, upper and lower quartiles, upper and lower tails. The median = 3.5, lower quartile (lq) = 1, upper quartile (uq) = 5. To indicate upper tail of the box we need to add to the upper quartile the length of the box (uq – lq) multiplied with 1.5. Accordingly, to indicate lower tail we need to subtract from the lower quartile the length of the box multiplied with 1.5. A tail cannot be higher than maximum value and lower than minimum value, therefore lower tail = -5 and upper tail is equal to upper quartile = 5.

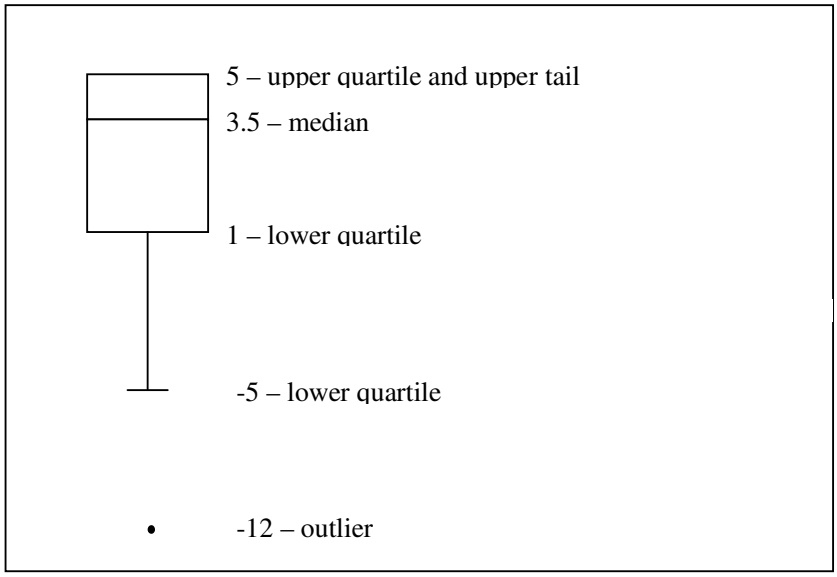

Figure 37. A box plot for time intervals between manual and tool supported transformation; PCMEF architecture

As it is showed on Figure 37, one outlier is indicated. It is an outlier from a statistical perspective. As foreseen, this is the result of participant *B* (see Figure 36). We decided to perform data reduction in this special case and to remove results of participant *B* from the set of valid results of second round of transformations (with PCMEF architecture). This result shows that participant *B* performed manual transformation much faster than tool-supported. We have to notice two facts. Firstly, it was the second round of transformations, so most probably participant already knew how to use transformation tool (they had already performed several transformations). Secondly, participant *B* was in the group that carried out tool supported transformation as the first one. This could mean that participant *B* misunderstood architectural pattern PCMEF, and needed a lot of time to get familiar with it. When he finally understood it, he performed the transformation immediately. This is also confirmed by the fact that experiment supervisor noted that participant B needed additional explanations before finishing the first PCMEF transformation. Due to the fact that we assume good knowledge of desired architectural pattern, such situation is unwanted and probably will not repeat in an experiment replication. Taking all these things into consideration, we decided to dismiss it.

After data reduction, the values of second round have changed, see Figure 38. The mean for manual transformation is 18.25 min. with standard deviation of 5.1. The mean for tool-supported transformation is 14 with a standard deviation of 4.5. Data is more regular and on average participants needed 4.5 minutes less to perform the transformation using the developed tool.

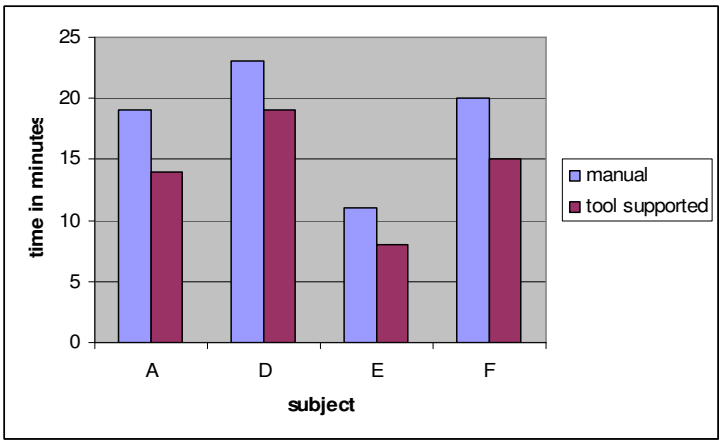

Figure 38. The amount of time needed to perform transformation; PCMEF architecture; after data reduction

6.5.2 Number of faults in the transformations

The second depended variable which was measured concerns quality of the outcome of the transformation. The quality was measured as the number of faults in design model, lower number of faults – higher quality. It is the narrowest definition of quality which is in fact very broad and ambiguous concept (Kan 2002). The architectural patterns and analysis models were relatively simple and detailed guidelines were handed to the participants. All these aspects supported small number of faults that could occur in the outcome of the transformation. These are the reasons why the numbers of faults in assessed models are very low. Moreover, there are no serious faults like missing classes, wrong packaging of classes or misplaced relations. Most of them are just forgotten attribute, not setting the class to be an abstract class, forgotten dependency between packages (layers). These faults are not architecturally important. There are slightly more faults in models from manual transformations. There is even no need to put any bar plots here because they are uninteresting and meaningless (most bars would be 1 or 0 points high). Although the details of number of faults (separately for each participant) are not presented here, all experiment data are attached to the thesis in Appendix B. The mean of number of faults for tool-supported transformation in both rounds is around 0.5. Just half of the participants made no more than one fault (standard deviation approximately 0.5). The mean for manual transformation is in both rounds around 1 (standard deviation approximately 0,8). Only one participant made more faults than 2 in one of the transformations. In the first round this participant made 5 faults but only of one type. Associations between classes were not put into design model. Probably, this is just a matter of forgetting. In each case the number of faults in tool-supported transformation was lower or the same as in the manual transformation.

The following Figure 39 presents total number of faults when performing transformation in both rounds. It is used to illustrate the difference between two approaches. Tool-supported transformation led to less faults (4) and therefore provided higher-quality models.

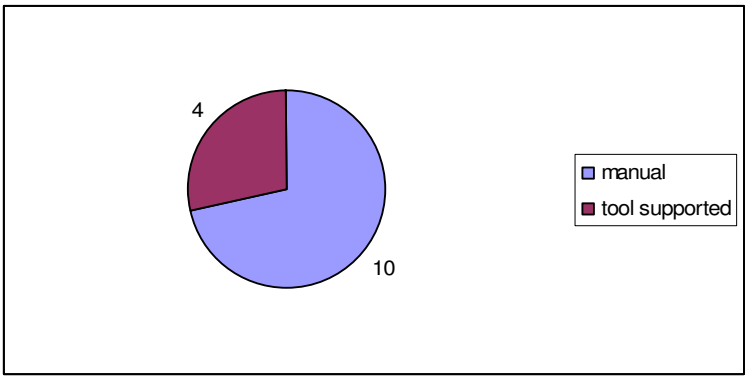

Figure 39. Comparison of total number of faults in manual and tool-supported transformations

For the hypothesis testing, we take only time as the effectiveness. The reasons for that are discussed in this section above and in the design of the experiment (Section 6.3.4).

6.5.3 Hypothesis testing

As aforementioned, the type of the experiment is paired comparison. It is characterized by one factor (transformation technique) and two treatments (manual and tool-supported). For this kind of test Wohlin et al. (2000: 92) advise to use paired t-test, Wilcoxon test or sign test. The first test belongs to a set of parametric tests whereas the latter two belong to non-parametric family. Additionally, Wohlin et al. (2000) suggest using non-parametric tests since non-parametric tests are more general and should be used when it is possible. Therefore we decided to use Wilcoxon test. There are some requirements that have to be fulfilled before Wilcoxon test can be applied. The pair of values should be comparable (which value is greater) and there should be possibility to rank the differences. Both requirements are fulfilled in our hypothesis testing.

Details of the Wilcoxon test can be found in (Wohlin et al. 2000: 103) which are based in turn on (Siegel and Castellan 1988). In general, differences between pairs are compared ($d_i = x_i - y_i$), then put into ascending order and ranked. After that, minimum from sum of the ranks of positive d_i and sum of the ranks of positive d_i is taken. Finally, the result is compared to statistical tables. In this study, for both rounds of experiment (counted separately or together), the null hypothesis can be rejected and the alternative hypothesis can by supported with the significance level (p) less than 0.05.

6.6 Interpretation of results

6.6.1 Pilot study results

Before the real experiment, a pilot study with two participants has been performed. It allowed determining and eliminating the essential shortcomings of the anticipated experiment. Every participant played a role of one group. Lessons learned are the following:

- Too much time needed on experiment causes subjects impatience. The large amount of time is caused by complicated architectures and complex analysis model. They should be simplified in the experiment.

- Introduction to Rose is indispensable. Although all of the participants are familiar with Rose, a short remainder of the key (from transformational point of view) features is needed.
- Good guidelines are indispensable to decrease the amount of questions and to ensure that everything is clear.

Results (amounts of time and number of faults) of the pilot study are similar to the results from experiment and will not be considered here.

6.6.2 Experiment results

In the experiment 6 subjects participated performing together 12 pairs of transformation. Two results were discarded due to data validation and data reduction. Experiment has proved the alternative hypothesis – that introduction of tool support in transformation of an analysis model to a design model improves effectiveness (in terms of the times since faults were insignificant). The results can be summarized in the following Table 6. It is easy to see the benefits of applying tool-supported approach in the table. On average, tool-supported transformation took 4,7 minutes less and results were more consistent (standard deviation was lower). The average increase of effectiveness is about 30%.

Table 6. Listing of general experiment results

	Manual		Tool-supported		Average difference	Average increase of effectiveness
	Mean	Standard deviation	Mean	Standard deviation		
1st round (MVC)	12.3 min	4.0	7.8 min	3.0	4.5 min	37%
2nd round (PCMEF)	18.3 min	5.1	14.0 min	4.5	4.3 min	23%
Together	**14,7**	**5,2**	**10,3**	**4,7**	**4,4 min**	**29,9%**

The interesting fact is the difference between the first round one the second round. Relatively simple architectural pattern was used in the first round (MVC). It took less time to perform transformation and the benefits of using tool-supported approach were greater (average increase of efficiency much higher). Although investigating the reasons for difference between transformation times (with different patterns) are not the field of this study, some general intuitive explanation can be given. PCMEF pattern is more complex and requires much more work than MVC. The additional work needed for creation of the pattern definition hardly paid off. This could mean that complex patterns are not worth tool-supported approach. However, we must remember that only 7 analysis classes were transformed. Moreover, architectural patterns have to be defined only once. They can be shared between development projects. If we had already defined PCMEF pattern, the transformation would be much faster (only few seconds for importing the pattern and pressing the button to transform). This issue is further discussed in conclusion section.

We also measured the start time of performing the first transformation. It indicates how much time a participant needed to get familiar with experiment guideline and how quickly they understood the instructions. The amount of time for every subject was approximately the same. Differences between participants were about 2 minutes maximum. This is not foreseen as a factor influencing the results, therefore will not be discussed further.

Participants filled a questionnaire at the end of the experiment. The questionnaire was intended to investigate their personal experience. All of them had finished *Software Systems Development* course at university and had similar level of knowledge, as foreseen in experiment design. Therefore, blocking according to their experience was unnecessary (Wohlin et al. 2000).

Additionally, the questionnaire gave possibility to the participants to assess the developed tool with their individual opinions. Generally, participants had positive feelings about the tool (usability and functionality) and suggested some small improvements. Some interesting ideas about extending the functionality (e.g. adding diagrams with *trace* dependency) were put forward. These suggestions should be considered in the future versions of the tool (this is discussed in Section 5.4 where possible extensions of the tool are considered). The opinions emphasized also the advantage of implementing the transformation tool in the well known environment – Rational Rose. Time needed for training and getting familiar with the tool were, in their opinion, considerably shorter.

6.7 Discussion and conclusions

We have shown by the experiment that tool-supported approach is more effective. On average, it took 30% less time to perform a transformation from analysis model to design model according to architectural pattern. This is a serious improvement which should be remarkable for software architects and project managers.

However, one more question could be posed. Is tool-supported approach worth introducing when the time it saves is only about 4 minutes? Time needed for training was significantly longer (introduction before experiment took about 30 minutes). The cost of introducing was not investigated (development of the tool took about one month). We will try to answer this question in the following paragraph.

The crucial fact is that in the experiment only the worst case was investigated – when usage of the tool-supported approach is least beneficial. There were used several means for providing the worst case. Firstly, analysis model was extraordinary simplified. It consisted of 7 classes only. The number of classes does not influence the speed of tool-supported transformation – rules are expressed for all of the analysis classes at once. On the other hand, speed of manual transformation depends directly on the number of analysis classes. More analysis classes – more time is needed to create design model. The real systems developed in industry can consist of hundreds of classes and therefore the time of the manual transformation should be considerably longer. Secondly, we had to express architectural pattern before the tool-supported transformation was carried out. When we have the pattern already expressed, transformation lasts very shortly – only few seconds to import the pattern and execute transformation. This is the aim of the possibility of importing and exporting definitions of patterns – to share them between many projects and to gain benefits of reuse. Again, it was not investigated in the experiment. In industry, there are several patterns most commonly used, therefore the need for defining the new ones from scratch is rare (Fowler 2002). Usually, just simple variations of well known patterns are applied. We have shown that introduction of tool-supported transformation brought benefits even in the worst case, so it is anticipated that the increase of effectiveness would be much greater in projects which do not need defining patterns from scratch.

Additionally, the results show that other similar improvements are possible, not only in the context of Rational Unified Process. As long as UML models are used, some conclusions can be applied in any process that uses UML models, even in agile methodologies.

Nevertheless, we are aware of constraints of the experimental research method. The remarkable reflection about experiment is that it will never provide the final answer for questions (Wohlin et al. 2000: 39). Further investigations, especially in industry, in less homogenous environment are needed to confirm our hypothesis. However, we believe that our experiment provides remarkable arguments for using automation in transformation of UML models. We believe that the experiment can be easily replicated and therefore the hypotheses strengthened. To facilitate a replication

of the experiment, complete experimental process is described in details. Additionally, all guidelines, models, descriptions of used patterns, and full results are included in the appendices.

7 SUMMARY AND SUGGESTIONS FOR FURTHER RESEARCH

The main goal of the thesis – to verify if applying automatic UML models transformation approach can improve transformation task – was accomplished. We showed in the empirical study that applying one of the transformation approaches can improve the work of the architect in RUP. Before the experiment was performed, the approaches to UML model transformation had been described and evaluated, and a way of expressing architectural patterns had been elaborated.

The features of the transformation approach worth emphasizing are: easiness of use by the developers (programming language of high level of abstraction with good API definition), high usability for the users (known interface, interactive mode, consistent modelling environment – without switching from one tool to another, possibility to interchange patterns). These were the essential issues that influenced the choice of the transformation approach. The chosen approach – direct model manipulation introduced in Rational Rose – allowed to implement the considered transformation with success. Empirical study showed that usage of this tool is beneficial and therefore that applying tool-supported transformation may lead to improvements.

The other achievement of the thesis is also an idea on making analysis model more valuable. Developers sometimes skip analysis model due to work overhead. However, analysis brings better understanding of the system requirements which is always good (the system should fulfil requirements). There are a lot of discussions about usefulness of the analysis model, e.g. (Comp.object 2000). The thesis can be an argument *for* using the analysis model. In the method presented in the thesis, the analysis model is not only a mean for better understanding requirements, but also it is used to obtain initial version of design model in automatic way. Therefore, cost of introducing design model (thanks to analysis model) could be considerably reduced.

One of the thesis achievements is also a method for defining software architecture that is easy to use for an architect and allows to perform some computer-aided operations that deal with software architecture. The method is general enough to define many different software architecture patterns and simple enough to define these patterns by an architect in an efficient way. The method can be applied in many cases, everywhere where layered architecture is considered.

Nevertheless, the most important achievement of the thesis is, in author's opinion, experimental presentation that improvements of RUP are feasible and do not require much additional work neither for the developers nor for the users.

The main shortcomings of the study is the fact that the method was not evaluated in the industrial environment. This shortcoming was caused by the time and resources constraints. Additionally, information introduced in the analysis model could be used in larger extent (only static part of it is used). Usage of behavioural aspects could be considered in the further studies. Moreover, architectural patterns could be enriched with more sophisticated design elements. Finally, design model could be more advance in the sense that some stereotypes to classes could be assigned. These stereotypes should be in accordance with implementation environment and should allow for, at least partial, code generation. Potential extensions of the transformation tool are very broad and may be implemented in the future.

Potential further researches include investigating the tool in an industry environment, for instance by pursuing a case study. Additionally, more sophisticated architectural patterns taken from real projects in industry could be considered. It is not certain that proposed transformation language will be suitable for every kind of architectural patterns.

The results of the study can be usable in every development methodology when UML modelling is used, even in agile methods. Partially, they show that MDA approach is very promising and may be the answer for seeking better software development methodology. However, MDA is a holistic approach and requires revolutionary changes in companies developing systems for years according to other methodologies. Although the presented method uses the same means of improvements – transformation of UML models, it is much different from MDA. It proposes evolutionary changes in already used and accepted heavy software development methodology.

REFERENCES

Agrawal A., Karsai G., and Shi F. (2003) *A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations*, International Journal on Software and Systems Modeling, (Submitted), 2003

Alexander C., Ishikawa S., Silverstein M., et al. (1977) *A pattern language*, Oxford University Press, New York

Baas L., Clements P., Kazman R. (1998) *Software Architecture In Practise*, Addison-Wesley

Beck K. (1999) *Extreme programming explained: embrace change*, Addison-Wesley Longman Publishing Co., Inc.

Bosch J. (2000) *Design & use of software architectures*, ACM Press Books, Addison-Wesley

Buschmann F, Meunier R., Rohnert H., Sommerlad P., and Stahl M. (1996). *Pattern-Oriented Software Architecture—A System of Patterns*, New York, NY: John Wiley and Sons

Charette R. (2001) *The Decision Is In: Agile Versus Heavy Methodologies*, Cutter Consortium

Comp.object (2000) *Do you keep your Analysis model up to date*, comp.object mailing list discussion <http://groups.google.com/groups?hl=pl&lr=&group=comp.object>

Creswell J. W. (2003) *Research Design – Qualitative, Quantitative and Mixed Methods Approaches*, Sage Publications

Czarnecki K. and Helsen S. (2003) *Classification of Model Transformation Approaches,* proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA. Anaheim, (October 2003)

Deursen A. van, and Klint P. (1998) *Little languages: Little maintenance*, Journal of Software Maintenance, 1998, pp. 75-93.

Dawson C.W. (2000) *The essence of computing projects: a student's guide*, Prentice Hall, Harlow UK

Eeles P. (2002) *Layering Strategies*, Rational Software White Paper

Fowler M. (2002) *Patterns of Enterprise Application Architecture*, Addison-Wesley

Gamma E., Helm R., Johnson R., and Vlissides J. (1994) *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison Wesley Longman

Gardner T., Griffin C., Koehler J., Hauser R. (2003) *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*, OMG < http://www.omg.org/docs/ad/03-08-02.pdf> (30.06.2004)

Jacobson I., Booch G., Rumbaugh J. (1999) *The Unified Software Development Process*, Addison-Wesley

Kan S.H. (2002) *Metrics and Models in Software Quality Engineering, Second Edition*, Addison Wesley

Maciaszek L., Liong B. L. (2004) *Practical Software Engineering. A Case Study Approach,* Addison-Wesley

OMG (2002a) *Meta-Object Facility (MOF™)*, version 1.4, OMG <www.omg.org/mof> (01.01.2005)

OMG (2002b) *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, OMG < http://www.omg.org/public_schedule> (30.06.2004)

OMG (2003a) Gardner T., Griffin C., Koehler J., Hauser R., *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*, OMG, < http://www.omg.org/docs/ad/03-08-02.pdf > (21.08.2003)

OMG (2003b) *UML 2.0 Diagram Interchange Specification*, final adopted specification, OMG, <http://www.omg.org/technology/documents/modeling_spec_catalog.htm> (01.06.2005)

OMG (2004) *UML 2.0 Superstructure Specification*, ver.04-10-02, <www.omg.org/uml> (15.05.2005)

OMG (2005) *OMG Model Driven Architecture* – Home Page <http://www.omg.org/mda/> (01.06.2005)

OPSLA (2003) *Generative Techniques in the context of Model Driven Architecture*, proceedings of the 2nd OOPSLA'03 Workshop, October 2003, <http://www.softmetaware.com/oopsla2003/mda-workshop.html> (10.10.2004)

Peltier M. (2002) *MTrans, a DSL for model transformation*, Proceedings of the Sixth International Enterprise Distributed Object Computing Conference

Peltier M., Bezivin J., Guillaume G. (2001) *MTRANS: a general framework, based on XSLT, for model transformations*, Workshop on Transformations in UML, Italy 2001

QVTPartners (2004) <http://www.qvtp.org/> last updated 04.02.2004 (last viewed 01.01.2005)

Rational (2001) *Using the Rose Extensibility Interface*, Rational Software Corporation, <ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/rose.html> (01.06.2005)

RUP (2003) Documentation of Rational Unified Process provided with Rational tools distribution, Rational Software Corporation (provided with Rational Suite)

Sendall S. (2003) *Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance?*, OOPSLA '03 Workshop "Generative techniques in the context of MDA" (Oktober 2003)

Sendall S. and Kozaczynski W. (2003) *Model Transformation: The Heart and Soul of Model-Driven Software Development*, IEEE Software 20, No. 5, 42–45 (September/October 2003)

Siegel S. and Castellan N. J. (1988) *Nonparametric Statistics for the Behavioral Sciences*, 2nd ed. New York, McGraw-Hill

Sommerville I. (2000) *Software Engineering (6th Edition)*, Addison Wesley

Standish (1994) *The CHAOS Report*, Standish Group, <http://www.standishgroup.com/sample_research/chaos_1994_1.php> (10.10.2004)

Wagner A. (2001) *A Pragmatic Approach to Rule-Based Transformations within UML using XMI.difference*, WITUML: Workshop on Integration and Transformation of UML models (held at ETAPS 2001)

Walkowiak A. (2004*) Badanie spójności modeli UML* (*Investigation of UML model consistency*), master thesis, Wrocław University of Technology

Whittle J. (2002) *Transformations and Software Modeling Language: Automating Transformations in UML*, Jézéquel, Hußmann & Cook (Eds.): Proceedings of UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany. Lecture Notes in Computer Science no. 2460, pp. 227-242, Springer, 2002.

Wohlin C., Runeson P., Höst M., Ohlsson M. C., Regnell B., Wesslén A. (2000) *Experimentation in software engineering: an introduction*, Kluwer Academic Publishers

APPENDIX A

The appendix contains all the materials used in the experiment.
1 – experiment guideline
2 – experiment forms
3 – analysis model used in the experiment

All the materials are presented in a form that was handed to the participants. However, participants were listening to the supervisor introduction and comments on the materials. Therefore, they had considerable larger amount of information.

1 – Experiment guideline

Introduction into experimentation in Software Engineering

- What is research
- What is experiment
- This experiment:
  - The intention is not to verify technical abilities of the tool but to investigate if changing model transformation technique from manual to automatic (in this specific implementation of the specific approach) improves the efficiency of work of a software architect

Summary of Rational Rose features

- Diagram is only a view of a model
- Creating and deleting relations (delete, ctrl+d)
- Creating and putting classes into packages (in tree view)
- Setting properties, stereotypes, names of the class (*open specification*)
- Inserting attributes and operations (*ins*)

Introduction into usage of plug-in ArchiTransAB

- What does it do
- Supported features: basic and advance
- Try using: menu *Tools* / *ArchiTransAB* / *Perform transformation*
- Additional info in document *How to use ArchiTransAB plug-in*

Training in using ArchiTransAB

1. Create package (folder) *Client-Server-my*
2. Create subpackage *Client*
3. Create subpackage *Server*
4. In the *Client* package create class *B?*
5. In the *Server* package create class *C?* and *E?*
6. In the root of *Client-Server* package create classes with default names with stereotypes *boundary*, *control*, *entity*
7. Create a class diagram and put all the classes onto it
8. Create dependencies with stereotype *trace* from *B?* to *boundary*, from *C?* to *control*, from *E?* to *entity*
9. Perform transformation *Tools* / *ArchiTransAB* / *Perform transformation*
10. Check results by drawing a diagram

Conducting experiment

Every participant of the experiment will be given one form. The form indicates the order of transformation and provides space for writing down start and end time of every task.

Rename output folders by adding ending *man* for manual transformation, and *tool* for tool-supported.

First part – MVC

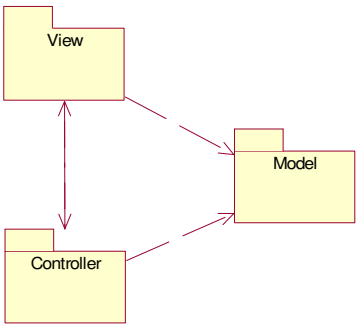

Figure 40. Overview of the MVC pattern

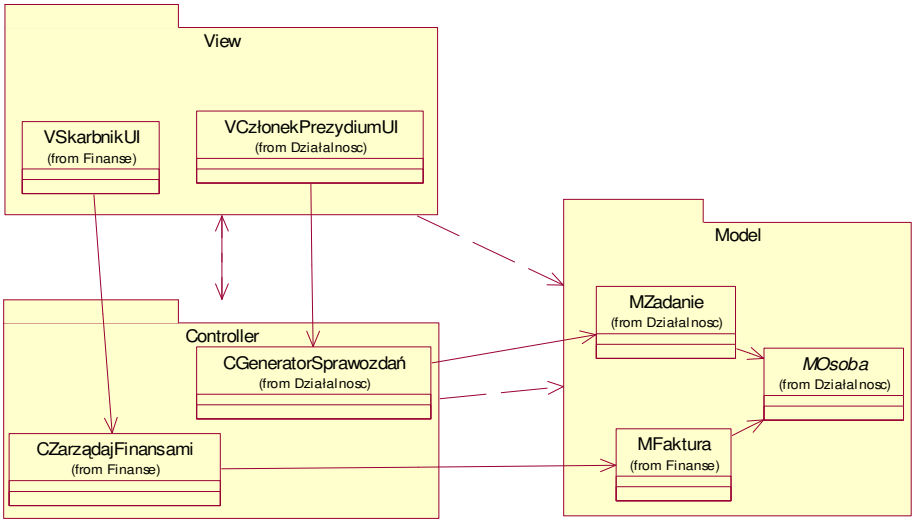

Figure 41. Result of the transformation

Do not forget!

- Double dependency: from View to Controller and from Controller to View
- Create new names (before name from analysis, a letter is added: V, C, or M)
- MOsoba is an abstract class
- MFaktura has one attribute, value: Currency = 0

Second part – PCMEF+

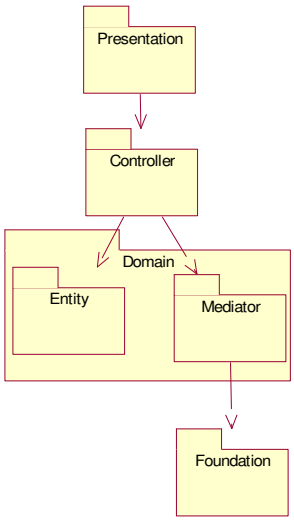

Figure 42. Overview of the PCMEF pattern

Do not forget!

- Create new names (before name from analysis, a letter is added)
- EOsoba and FOsoba is an abstract class
- EFaktura and FFaktura has one attribute, value: Currency = 0

When creating a pattern remember!

- In Entity package (class E?), set property *Analysis Relations* to *Keep only inside a package*
- In Foundation package (class F?) set property *Analysis Relations* to *Discard all relations*
- Edit properties of association from Controller (C?) to MFacade and from MFacade to Entity (E?) and dependency from MFacade to Foundation (F?) and set *Create Relations in output* to *Create All*

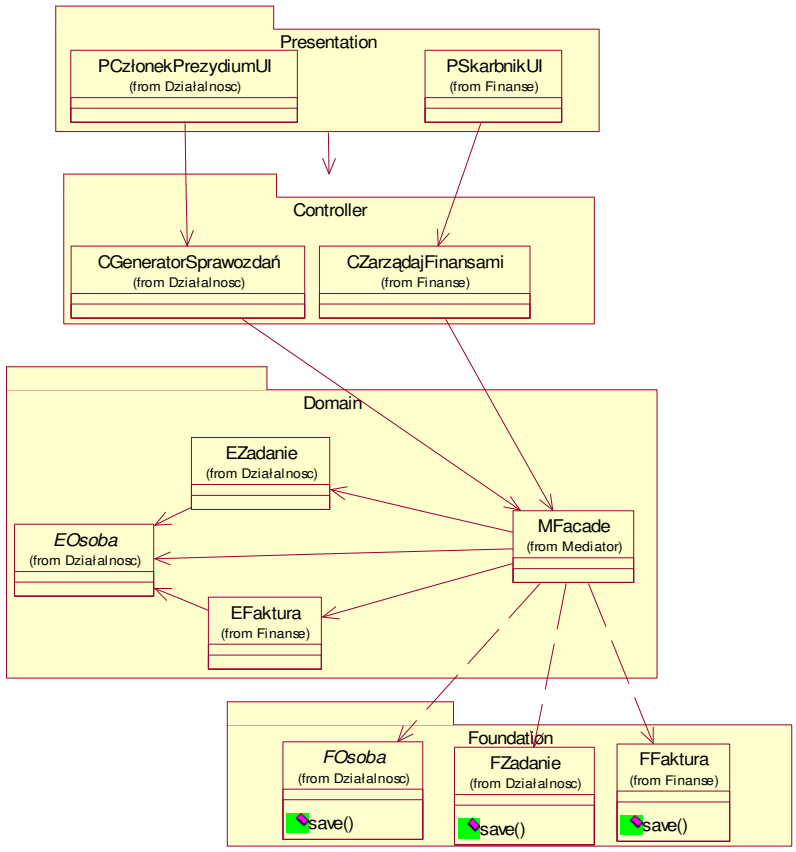

Figure 43. Result of transformation to PCMEF

How to arrange transformation

Prerequisite for performing transformation is to have following items in a model in Rational Rose:

- Analysis model with analysis classes (recommended folder to keep this model is *Logical View \ Analysis Model*) – this is an input for transformation
- Destination folder (recommended is *Logical View \ Design Model*) – this is an output for the transformation
- A folder with one or more architectural patterns. A folder with architectural patterns should be created by a user if it does not already exists (recommended location is *Logical View \ Architectural Patterns*). Patterns can be created in one of the two ways. A pattern can be imported (see section *Sharing Architectural Patterns* in *Transformation Language* paper) or can be created from scratch (see *Transformation Language* paper for details).

How to perform transformation

Performing transformation consist of few steps. Firstly, choose menu *Tools/ArchiTransAB/Perform Transformation*. After that, a dialog appears in which three folders have to be specified:

- Architectural Patterns folder – a folder that includes one or more patterns, default is *Logical View \ Architectural Patterns*
- Analysis Model folder – a folder that include analysis classes (analysis classes can be nested in subpackages), default is *Logical View \ Analysis Model*
- Destination folder – a folder in which initial design model after transformation will be placed, default is *Logical View \ Design Model*

In the next step, if it is defined in Architectural Patterns folder more than one pattern, a dialog appears which allows to choose one pattern. After that, a transformation is performed and initial design model is created in the Destination folder. If any faults occur, information dialog shows warnings.

How to verify results

As a result, a new folder with the name of the pattern occurs in the Destination folder. In this new folder, initial design model is created. It can be browsed with a tree view in Rational Rose (left panel). Relationships of the new classes could be seen in three ways:

- In the tree view (only associations)
- In specification dialog of the class (tab *Relations*)
- After creation of new class diagram and placed classes on it (graphical view)

Other properties of a class (e.g. attributes, operation, documentation) can be seen in specification dialog in appropriate tabs.

2 – forms

Group 1, Name: ……..

First test - MVC

Manual transformation	
Start time:	
End time:	
Number of faults:	
Tool supported transformation	
Start time:	
End time:	
Number of faults:	

Second test – PCMEF

Tool supported transformation	
Start time:	
End time:	
Number of faults:	
Manual transformation	
Start time:	
End time:	
Number of faults:	

Group 2, Name: ……..

First test - MVC

Tool supported transformation	
Start time:	
End time:	

Number of faults:	
Manual transformation	
Start time:	
End time:	
Number of faults:	

Second test – PCMEF+

Manual transformation	
Start time:	
End time:	
Number of faults:	
Tool supported transformation	
Start time:	
End time:	
Number of faults:	

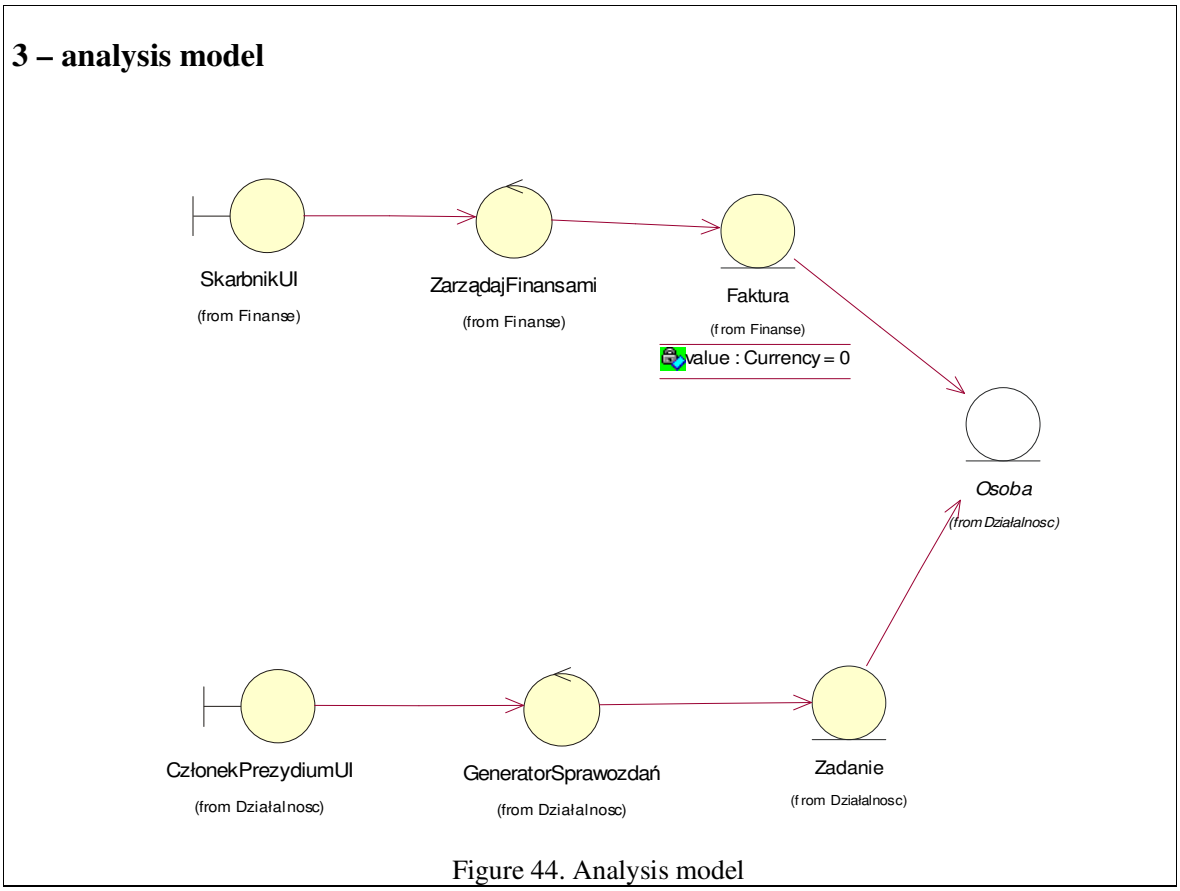

Figure 44. Analysis model

APPENDIX B

Appendix B contains all data collected during experiment.

Table 1. Times MVC

	manual	tool supported
A	9	8
B	13	9
C	18	12
D	15	7
E	7	3
F	12	8
mean	12,333333	7,833333
standard deviation	3,9832985	2,926887

Table 2. Faults MVC

	manual	tool supported
A	2	0
B	1	1
C	1	1
D	0	0
E	1	0
F	1	1
Mean	1	0,5
standard deviation	0,632456	0,547723

Table 3. Times PCMEF

	manual	tool supported
A	19	14
D	23	19
E	11	8
F	20	15
mean	18,25	14
standard deviation	5,1234754	4,546061

Table 4. Times PCMEF after data reduction

	manual	tool supported
A	19	14
D	23	19
E	11	8
F	20	15
mean	18,25	14
standard deviation	5,1234754	4,546061

Table 5. Faults PCMEF

	manual	tool supported
A	0	0
D	2	1
E	1	0
F	1	0
mean	1	0,25
standard deviation	0,816497	0,5

APPENDIX C

Appendix C contains UML class diagrams presenting metamodel of the transformation language.

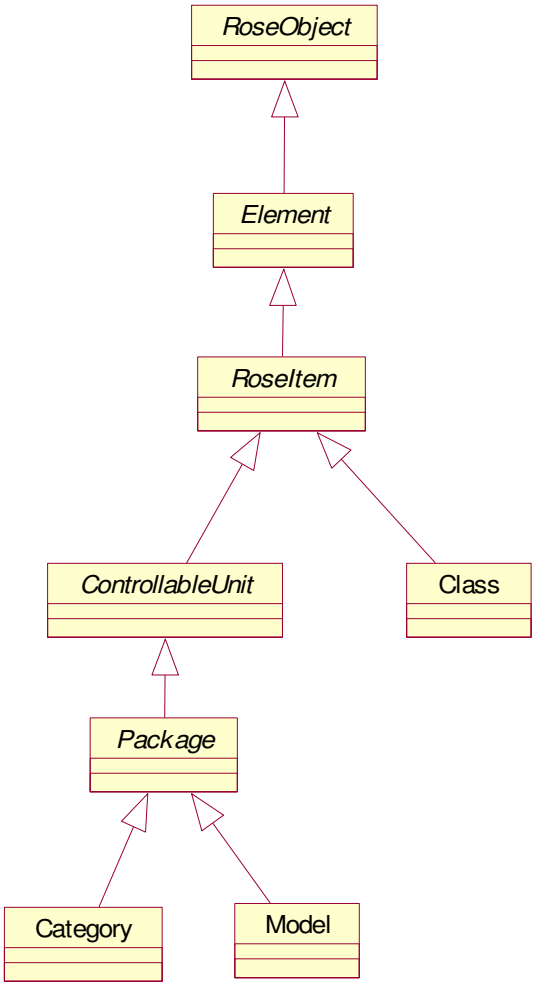

Figure 45. A part of the transformation language metamodel with class and package

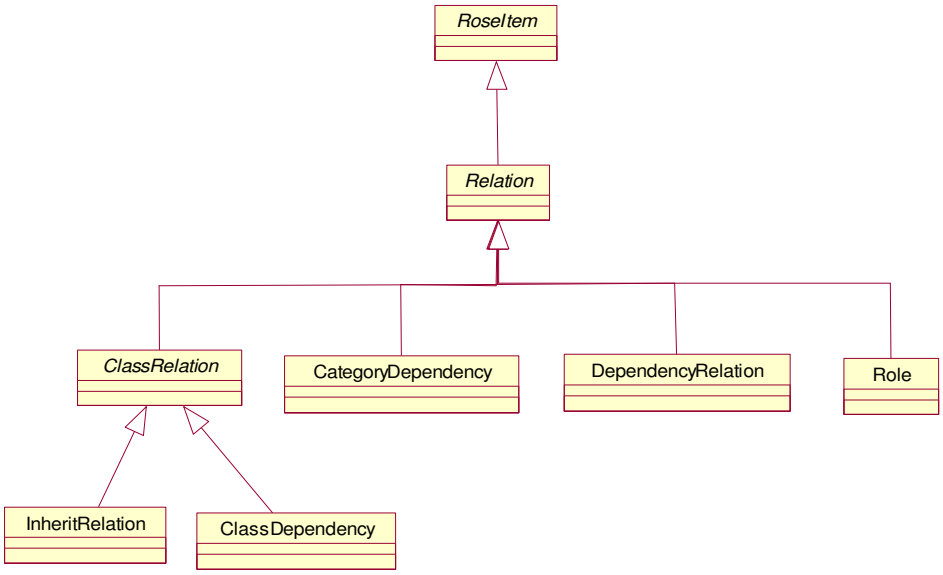

Figure 46. A part of the transformation language metamodel with class and package with relations

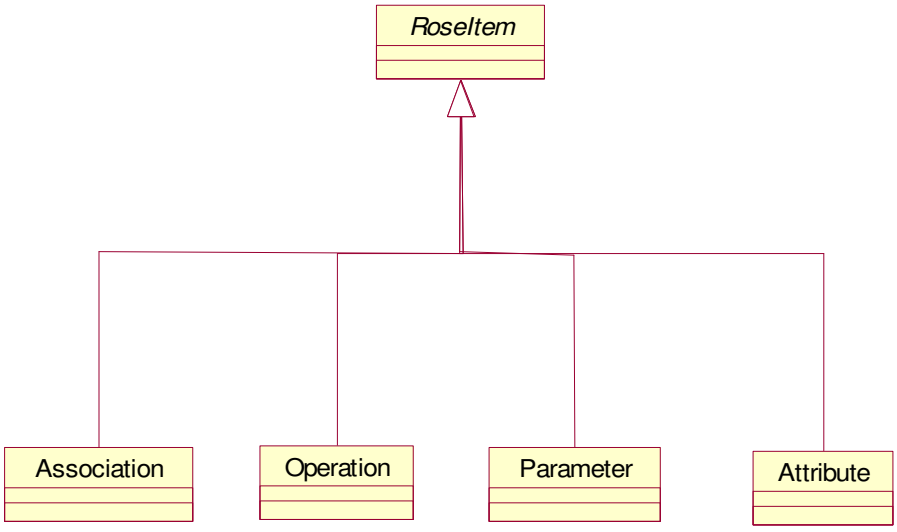

Figure 47. A part of the transformation language metamodel with other elements