Master Thesis
Software Engineering
Thesis no: MSE-2012:96
06 2012

Migration of chosen architectural pattern to Service Oriented Architecture

Piotr Kaliniak

This thesis is presented as part of Degree of
European Master in Software Engineering

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

This thesis is submitted to the School of Computing 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(s):
Piotr Kaliniak
E-mail: piotr.kaliniak@gmail.com

University advisor(s):
Dr. Ludwik Kuźniarz
School of Computing,
Blekinge Institute of Technology, Sweden

External University advisor(s)
Dr. Bogumiła Hnatkowska
Wrocław University of Technology, Poland

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context: Several examples of successful migrations of systems to Service Oriented Architecture (SOA) are presented in the literature. Some of the approaches also try to structure the process of migration to SOA. The reported migration attempts underline the role of architecture of migrated system, but they do not explore the architectural patterns applied in architecture of migrated systems while proper usage of patterns may simplify and improve quality of migration.

Objectives: This work is aimed at elaborating guidelines that support migration from a system that is based on a chosen architectural pattern towards a system based on Service Oriented Architecture.

Methods: Literature review is used as a basic method in the initial steps of the research, that is during investigation of existing techniques of migration to SOA, establishing procedure for selection of the migrated pattern and identifying building blocks of the target architecture. Results of the literature reviews are further analyzed in order to select the migrated architectural pattern and to elaborate the target architecture. The guidelines for migration are the result of the synthesis of the analyzed information.

Results: The migration is realized as a translation between two pattern languages: the first pattern language describes the chosen architectural pattern –Model–View–Controller and the second pattern language describes SOA target architecture, expressed using SOA architectural patterns. The translation is defined by a set of migration guidelines. The approach is also illustrated with migrating an example student project.

Conclusion: The study shows that the usage of an architectural pattern during migration allows to define the migration in a simple, structured and precise way using guidelines that represent a set of subsequent well defined steps that should be applied in order to migrate a specific type of legacy system.

Keywords: Architectural pattern, MVC,migration, SOA, guidelines

ACKNOWLEDGMENT

First, I would like to express my gratitude to my both advisors. Dr. Ludwik Kuźniarz and Dr. Bogumiła Hnatkowska supported my researches and gave me valuable advices and comments. It was a great experience and pleasure to work with you and learn from you.

I am also very grateful to my family and Anna Kiełbaska who were supporting and motivating me from the very begging.

Thank you very much!

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Background

With each passing year, more and more systems become obsolete. New technologies are invented. The old systems become useless not only because of technical challenges, but also maintenance becomes more and more costly and complicated. Those issues may force companies to change their old system. This change can be achieved in two ways. The first way is to replace the existing system with a new one. However, the approach seems to be accurate, there is one very important issue in the way: cost. Replacing the old system is very costly in terms of money and time, which is required to create a new system almost from the scratch [51]. Migration of large projects is a serious and risky task that requires careful analysis of feasibility and required efforts [48]. An example of seriousness of migration is provided by Scott Bolling [17]. He participated in migration of a system. The migration had 25 unsuccessful migration attempts. The second approach is to migrate from old systems to new systems that use the newest software development paradigms and follow the recent technologies.

Available literature related to migration of legacy systems shows that legacy systems migrate not only toward specific applications like Multi-tier Applications [27] or Client Server Applications [63]. The systems migrate also to meet approaches in software development like Object Based Environment [72] [51], Product Line [18] or Service Oriented Architecture [75][48].

Migration to Object Based Environment can be performed with help of wrapping technology. Application of the technology results in creation of wrappers. A wrapper is a new code that encapsulates the legacy code. The wrappers are called "Legacy objects". Those objects provide interfaces that enable usage of wrapped code. Application of this approach supports reimplementation of selected Legacy Objects independently. The legacy objects remain unchanged until a new and equivalent implementation is ready [51]. Wrapping technology distributes migration effort over time.

Migration to Product Line is not as popular as migration toward Object Based Environment. Product Line is meant to increase large-scale software reuse and reduce time to market. Application of Product Line supports well market driven development [18].

Migration trend aims also at migration toward Service Oriented Architecture (SOA). SOA is an approach to software development[13] that integrated business processes and IT infrastructure. Architecture of a system designed in SOA-way is a package of loosely coupled interoperable services that exchange data and expose their functionalities via interfaces. A service is *"an asset that corresponds to real-world business activities or recognizable business functions"*[59].The concept of SOA changed during the years and evolved from concepts like distributed computing and modular programming [68]. Successful migrations toward SOA have been achieved in many domains including banking, electronic payment applications and development tools [48]. In spite of all those successful migrations, a person that decided to migrate toward SOA has to be aware that migration toward SOA is not an ultimate solution to all problems with old systems. Migration to SOA needs an additional time that is needed to understand this concept. Misunderstanding raises misconceptions [47]that may be costly. Choice of SOA relates to both technical and design challenges, because not every system can be presented as a set of reusable services or the cost of such adaptation disqualifies SOA as a solution[47].

Literature presents several examples of techniques of migration toward SOA, but three of them are especially important. They present different types of migration to SOA and they are well described. The first is Taxonomy Analysis that analyzes the existing code [84].The next technique suggests to treat legacy system as a "black box" and to wrap its code into services [77]. The last approach is rather a family of five approaches. It is called Service Migration and Reuse Technique (SMART) [10].

Taxonomy Analysis is a method that uses an existing code and documentation as an input. Analysis of the documentation set boundaries of the system and identifies services [84]. The services are described on business level. Analysis of the available code provides a dendogram [84]. This diagram in form of a tree presents the most often-used terms and relations between them. This approach is semi automatic because a decision person (like an architect) has to set cutting points that divide the tree into subtrees. Those subtrees are further implemented as services.

Wrapping as a technique of migration to SOA is similar to migration toward Object Based Environment. The migrated system is wrapped into a set of services that cover all the use cases covered by the actual system. The wrapping

technique conducts a three steps migration [23] The first step is meant to identify use cases and candidate services that cover those use cases. The description of this step does not provide own way of identification of services, instead it refers to works of Sneed [76] or SMART [49][10].The identified services are wrapped into services during the second step. The last step is deployment and validation. This step establishes infrastructure and deploys already created services. The services are further tested in order to assure that the migrated system fulfills requirements.

SMART is a family of five approaches of migration toward SOA [10]. The basic SMART approach is SMART-Migration Pilot (SMART-MP). SMART-MP identifies services and their components. This technique estimates potential risks and tries to provide a migration pilot with strategies for migration of the whole systems. Four remaining approaches are tailored version of the basic case. SMART Service Migration Feasibility (SMART-SMF) focuses mainly on feasibility of migration and its risks. SMART Enterprise Service Portfolio (SMART-ESP) dedicated for companies that decided to migrate their system but they did not identify all the services. SMART Environment (SMART-ENV) is meant for companies that did not select the target platform for migrated system. This approach aims at selection of this platform with analysis of its implications like risks and cost. The last family member is SMART System. This technique supports migration from initial estimations and analysis, through implementation and selection of environment till the end of migration. The SMART family provides guidelines for migration, but the guidelines are not complete [3]. They neglect impact of architecture of migrated systems on the process of migration and the target architecture.

Descriptions of those three approaches allows identifying following advantages and drawbacks:

Taxonomy analysis
Advantages

1. Identifies relations between services
2. The technique is systematic
3. Execution of the technique can be performed semi automatic
4. Provides a lot of information about the migrated system

Drawbacks

1. The technique does not consider architectural patterns that are applied in architecture of migrated systems

2. Requires documentation - this causes problems because documentation of an old system may be missing or not maintained

Wrapping
Advantages

1. The technique is systematic
2. Execution of the technique is semi automatic

Drawbacks

1. The technique does not consider architectural patterns that are applied in architecture of migrated systems
2. A full list of use cases is needed. The use cases are described in documentation that may be missing or not maintained
3. The technique bases on inputs and outputs of the system. It is hard to define all possible combinations of input-output pairs.

SMART
Advantages

1. The technique is systematic
2. The technique produces many artifacts that help in understanding the migrated system
3. SMART has a few variants that are tailored to different needs

Drawbacks

1. The technique does not consider architectural patterns that are applied in architecture of migrated systems
2. Application of the technique requires a lot of time

1.2 Problem Statement

Identified techniques of migration to SOA do not consider architectural patterns applied in the migrated system (see first drawback of each technique).

Usage of architectural patterns during migration is important because they express some common structure of the system. A migration technique that bases on an architectural pattern could provide a structured and systematic way of migration for systems characterised by this particular pattern. Additionally, it

was noticed that identification of design pattern helps in identification of services [7].

Aim

This thesis is meant to elaborate guidelines that support migration from a system that is based on a chosen architectural pattern toward a system based on Service Oriented Architecture.

Objectives
The objectives of the thesis are as follows:

1. To investigate existing migration toward SOA approaches.
2. To investigate existing architectural patterns in order to select the pattern for migration.
3. To investigate SOA in order to provide understanding of SOA in the context of this work.
4. To elaborate the target architecture.
5. To elaborate translation between selected architectural pattern and the target architecture
6. Illustrate an application of the translation.

Research question:
To fulfill the aim and the objectives, the thesis addresses and answers following research questions:

- RQ.1 What are the existing techniques of migration toward SOA?
- RQ.2 What are drawbacks and advantages of identified techniques of migration toward SOA?
- RQ.3 What process employ in order to select the pattern for migration?
- RQ.4 Which pattern should be chosen for migration?
- RQ.5 What elements should be the building blocks of the target architecture?
- RQ.6 How the target architecture should look like?
- RQ.7 How to translate the selected architectural pattern into the target architecture?

- RQ.8 What are the drawback and advantages of the new technique ?

Expected outcomes: The following outcomes are expected to be elaborate by the thesis.

- EQ.1 List of techniques of migration toward SOA with descriptions
- EQ.2 List of drawback and advantages of found techniques of migration toward SOA
- EQ.3 Process for selection of the migrated architectural pattern, motivation behind the activities
- EQ.4 Architectural pattern for the migration
- EQ.5 List of elements building the target architecture
- EQ.6 Target architecture
- EQ.7 Guidelines for migration
- EQ.8 Example illustration of implementation of the migration according to the guidelines
- EQ.9 Advantages and drawback of the guidelines for migration to SOA

1.3 Research Methodology

This study is conducted in two domains: Architectural Patterns and Service Oriented Architecture. Both of the domains are covered by a number of publications.

Entry points for the research are books containing body of knowledge. The books were identified during elaborating the topic for this thesis. For Architectural Patterns, the series of books by Frank Buschmann [22][71][45][20][21] is considered as a body of knowledge. In turn, body of knowledge for Service Oriented Architecture or more precisely Architectural Patterns in SOA is described in *"SOA Design Patterns"*[32] by T.Erl. In case of migration to SOA, the entry point are techniques for migration to SOA described in introduction (see section 1.1). Those approaches and other further identified are deeper investigated in order to get better understanding what is done and how it is done.

The basic research methods applied in this thesis are Literature Review, Analysis and Synthesis. The core of the literature review applied in this thesis are books containing bodies of knowledge and publications describing identified techniques for migration. Those sources are investigated in the first order. If they

do not provide satisfying information then other books, conference and journal papers are studied. Web pages are referenced mainly when some knowledge about technology is needed. Literature review is conducted until a satisfying information is not found. The review takes into account year of publication of found sources. The newest publications are revised in the first order. The main sources of journals and conference papers are: IEEE, Inspec, Springer and ACM digital library. Literature review in domain of Architectural Patterns and SOA is conducted separately. Information provided by literature review is further analysed and synthesised

Literature Review in the domain of Architectural Patterns is applied in order to establish procedure for selection of architectural pattern(answer for RQ.3).The procedure is further applied and results of each activity is analysed. The pattern for migration is chosen based on those results (answer for RQ.4).

In case of SOA, literature review is conducted twice. The purpose of the first review is to investigate the existing methods of migration to SOA (answer for RQ.1). The second literature review is meant to find elements that will build the target architecture (answer for RQ.5). The identified methods of migration are further analysed in order to identify their advantages and drawbacks (answer for RQ.2). Design of the target architecture (answer for RQ.6) is result of analysis outcome of literature review providing building blocks of the target architecture.

Information gathered during selection of the architectural pattern for migration and elaboration of the target architecture is further synthesised in order to elaborate translation between the chosen architectural pattern and the target SOA (answer for RQ.7). Finally the information gather during analysis of identified migration to SOA techniques is synthesised with the elaborated guidelines (answer for RQ.8).

Overview of the described process is presented in figure 1.1. The EO.8 is not present in the figure because it is not a direct outcome of research questions.

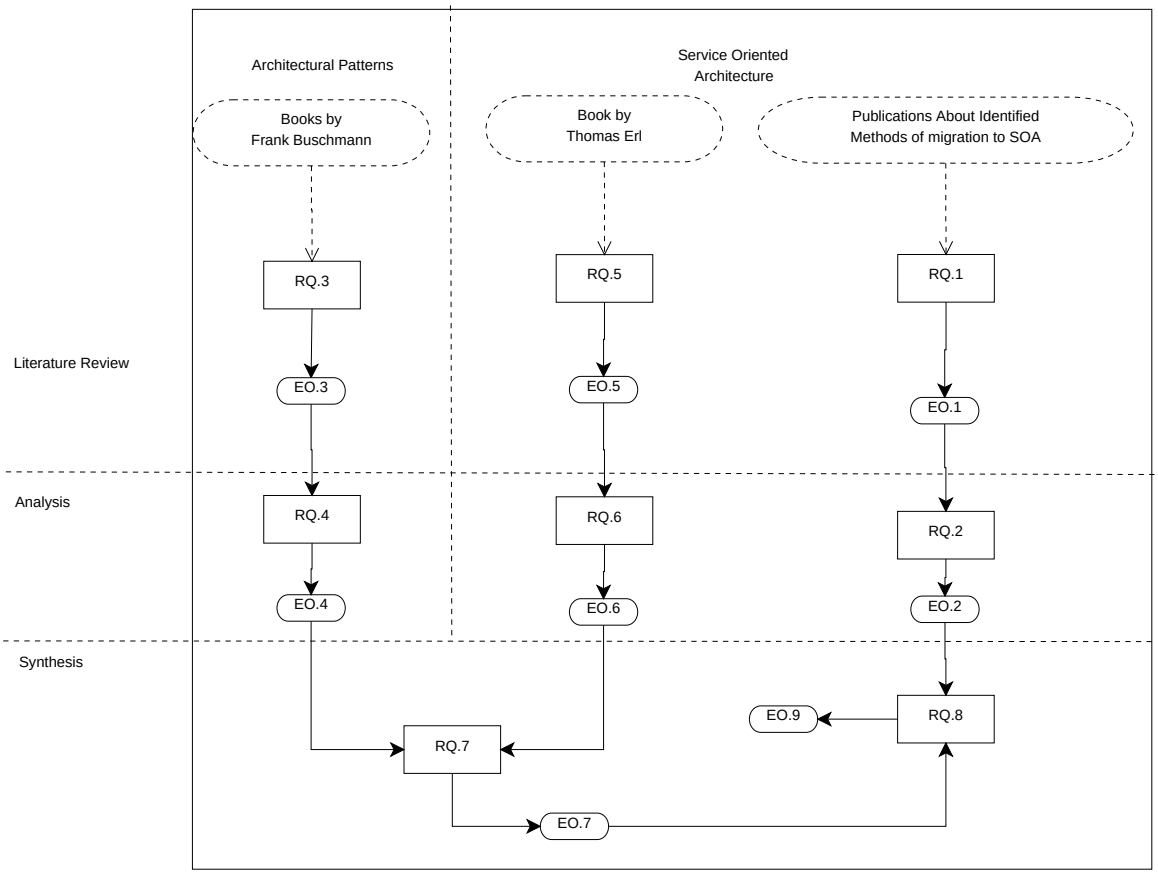

Figure 1.1: Relation between research questions, outcomes, methodologies and domains of interest

1.4 Outline of the thesis

The thesis is organised as follows:

Chapter 1 presents background information about the problem of migration of a legacy system along with researches, objectives to achieve and problem statement.

Chapter 2 presents related works.

Chapter 3 presents selection of a pattern for migration with procedure and results

Chapter 4 describes background information about SOA and presents the target architecture with its justification.

Chapter 5 presents guidelines for migration with their motivation. Additionally an example application of the guidelines is presented.

Chapter 6 summarised the work, provides future works and answers research questions.

Chapter 2

Related Work

Motivation behind migration is a result of two factors. The first factor is the fact that old but very often mission critical systems cannot be developed or maintained anymore because it becomes too expensive. This factor is called [74] *"Legacy System dilema"*. The second factor describes why a company does not want to develop a new application from the scratch. This may include for instance lack of documentation of the system. Additionally, the system can be a mature application that is a result of many years of development and maintenance. When a company decides to migrate an application, it always has to face two very general problems [74]. The first problem is caused by own, old application which was developed for years and may be lacking in documentation. The second problem is a target system, namely the architecture and migration decision that are associated with selection of the target system.

One of possible migration directions is migration toward Service Oriented Architecture which brings some very promising benefits like code reuse and easy further integration (see 4.7).But even if the choice is made, still an open question remains: *How to conduct the migration?* According to Z.Zhang and H.Yang there are three types of migration toward SOA [84].The first approach called White–Box requires high understanding of the system and allows to make significant changes in code if there is such need. The second approach is focused rather on input and output of the system and is called Black–Box. The third approach lies between white and Black–Box techniques and it is called Grey–Box technique. The "Gray–Box" technique includes code analysis, modification of the code and analysis of behaviour of the system as a whole entity.

How does migration is important is also evidenced by the fact that it became a subject of standardisation. The standard, named Architectural–Driven Modernisation (ADM) is meant to facilitate migration efforts[58]. This standard considers system on three levels of abstraction. The standard investigates also role of patterns in the system. Migration towards Service Oriented Architecture became also a subject of interest of architects dealing with non–SOA architectural patterns like patterns identified in the second chapter.

This chapter is meant to investigate techniques of migration toward SOA in order to set context of the work and identify advantage and drawback of the existing methods. The study aims at mature and well described migration techniques but techniques that base on architectural patterns are also briefly presented. The chapter is organised as follows:

1. Migration toward SOA – "colour" techniques
   This section presents three example techniques that correspond to White–Box, Black–Box and Grey–Box approaches. Descriptions provided by available literature give an overview of studies conducted in the field of migrations toward SOA. Additonally, the section presented advantages and drawback of the presented techniques.

2. Other approaches
   The section presents three other approaches. The first approach is not dedicated for migration toward SOA, but the steps taken during migration reflect steps taken during creation of SOA. The remaining two techniques present different approaches of migration of an architectural pattern to SOA.

3. Summary
   The summary presents result of investigation in the field of migration toward SOA.

2.1 Migration toward SOA – "colour techniques"

This section presents briefly examples of migration techniques described byZ.Zhang and H.Yang[84].

2.1.1 White–box: Service–Oriented Migration and Reuse Technique

Service–Oriented Migration and Reuse Technique (SMART) is rather a family of processes then a single process. It is meant to support organisations in making initial decision about feasibility of migration system towards SOA [10]. SMART evolved from the Options Analysis for Reengineering (OAR) method developed at the SEI. [49][10]

Family members

However SMART family consists of five members, the members are just tailored variants of the first technique–SMART–Migration Pilot (SMART–MP), therefore this "root" approach in opposite to remaining family members is described more detailed.

SMART–MP – SMART Migration Pilot focuses mainly on early project analysis. The analysis determinates potential challenges, trade–offs, risks and cost by identifying potential services and their components [10]. Migration Planning tries also to recognise pilot project and appropriate migration strategies [10]. According to SMART–MP description[10], the technique consists of four following elements :

1. Element 1 –SMART–MP Process
   The first element is a process that is meant to analyze the gap between both the actual and a target system by gathering information about legacy components and potential services build on them. Gap analysis leads to development of an initial migration strategy.

2. Element 2 –Service Migration Interview Guide (SMIG)
   Interview Guide aims at identification of potential risk and costs associated with the migration process. The identification is based on answering questions, which are categorized in more than 60 categories [10].

3. Element 3 –SMART Tool
   The tool supports data collection and organization. The support of assignation of answers to the question to related risks [10].The tool uses SMIG as a framework to create a draft migration strategy.

4. Element 4 –Artifact Templates
   The element consists of templates for following artifacts [10]: Migration Issues List, Business Process–Service Mapping, Service Table, Component Table, Service – Component Alternatives Table and Migration Strategy.

Elements presented above are further used during migration process by SMART activities (see figure 2.1). SMART–MP technique distinguishes six iterative processes:

1. Process 1 –Establish Context
   Establishing of the context is the first SMART–MP process. The process is meant to gather basic information about the system by describing its business and technical context. Business context includes identification of stakeholders and identification of their goals. The technical context includes identification of candidate services and analysis of legacy system. Services identified during this step are generic because generalisation helps to determinate project feasibility and provides required experience if the decision about migration is made [10]. However, this is the first SMART process, it generates three out of eight documents [10].

   (a) Stakeholder List –contains list of stakeholders with contact data and information to elicit from each of them.

   (b) Characteristic List –contains a set of predefined characteristics meant to support decision about migration feasibility. The information refers to legacy system's components.

   (c) Migration Issues List –the list contains information about potential migration issues of both the system and single components

2. Process 2 –Define Candidate Services
   Service Candidate Definition is restricted to a small number (typically 3–4) of well defined in the previous step services. A good candidate keeps high cohesion, high reuse (see 4.3.1) and clearly defined input and output. Finally, selected services are specified more precisely in terms of Quality of Service (QoS) and definition of input–output.

3. Process 3 –Describe Existing Capability
   Capability description is meant to localize and gather information about components of legacy system that contain functionality required by previously selected services [10]. The description of component includes both architectural view like design paradigms, system quality and technical view like implementation platform, language, code documentation, code size [10], age of the components, code complexity and interfaces for users and systems [49].Asides from gathering technical and business description, components should be described from quality point of view, by taking into consideration their change history, outstanding problems, likehood of meeting long term needs and historical cost of development and maintenance [49]. All this information is presented in two formal artifacts:

   (a) Component Table – contains only components meant for migration to services. The table captures also characteristics described in characteristic list.

   (b) Service Table –contains a list of possible services derived from component table with characteristics of those services.

4. Process 4 – describes Target SOA Environment
   Description of a target environment includes major components of target environment with influencing technologies and standards, guidelines for service implementation, their interaction with environment and quality of services (QoS) [10].

5. Process 5 –Analyse the Gap
   The gap analysis is meant to provide preliminary approximation of costs and risks of particular services allocated from the legacy system. The approximation is based on necessary changes of each required component. In turn, the allocation is based on characteristic of SOA environment and requirements coming from previously identified services. Gap analysis has

two main goals. The first is to identify gaps in knowledge about the legacy system. The second is to create Component Option Table which is used as an input to the final process.

(a) Component Option Table –maps possibilities between Service Table and Component Table. The table contains also mapping used in system Commercial Off The Shelf (COTS) components. There are two possible ways in which the process can be conducted [49]. The first is *"manual"* allocation of components prepared by SMART team.The second approach is enriched by supporting tools. The supporting tools are used mainly for analysis of the code that does not have complete documentation or quality of the code is not determined.

6. Process 6 –Develop Strategy
Strategy development is the last and the final SMART–MP process. The process makes use of all the previously gathered information including migration issues and information related to cost and risk estimations. Both cost and risk estimations have to be determined precisely, because they set additional constrains for the project. They also decide about feasibility of the project. In turn, migration issues include [10]identification of a pilot project by identification of initial set of services and the order of their implementation. The issues address guidelines created during describing of the target environment and specify possible migration paths like wrapping or re–writing [49]. In result, two artifacts are created:

(a) Migration alternatives table – alternative table takes as an input Components Service Options Table and produces feasible migration strategies. The strategies differ mainly in a list of components selected for migration, order of processes and code modification.

(b) Service Migration Strategy – migration strategy is a final document that is a result of all the processes and artifacts. The strategy describes services that can be implemented and what components would create them. The document contains also description of the processes required to accomplish the migration and full cost and risk analysis.

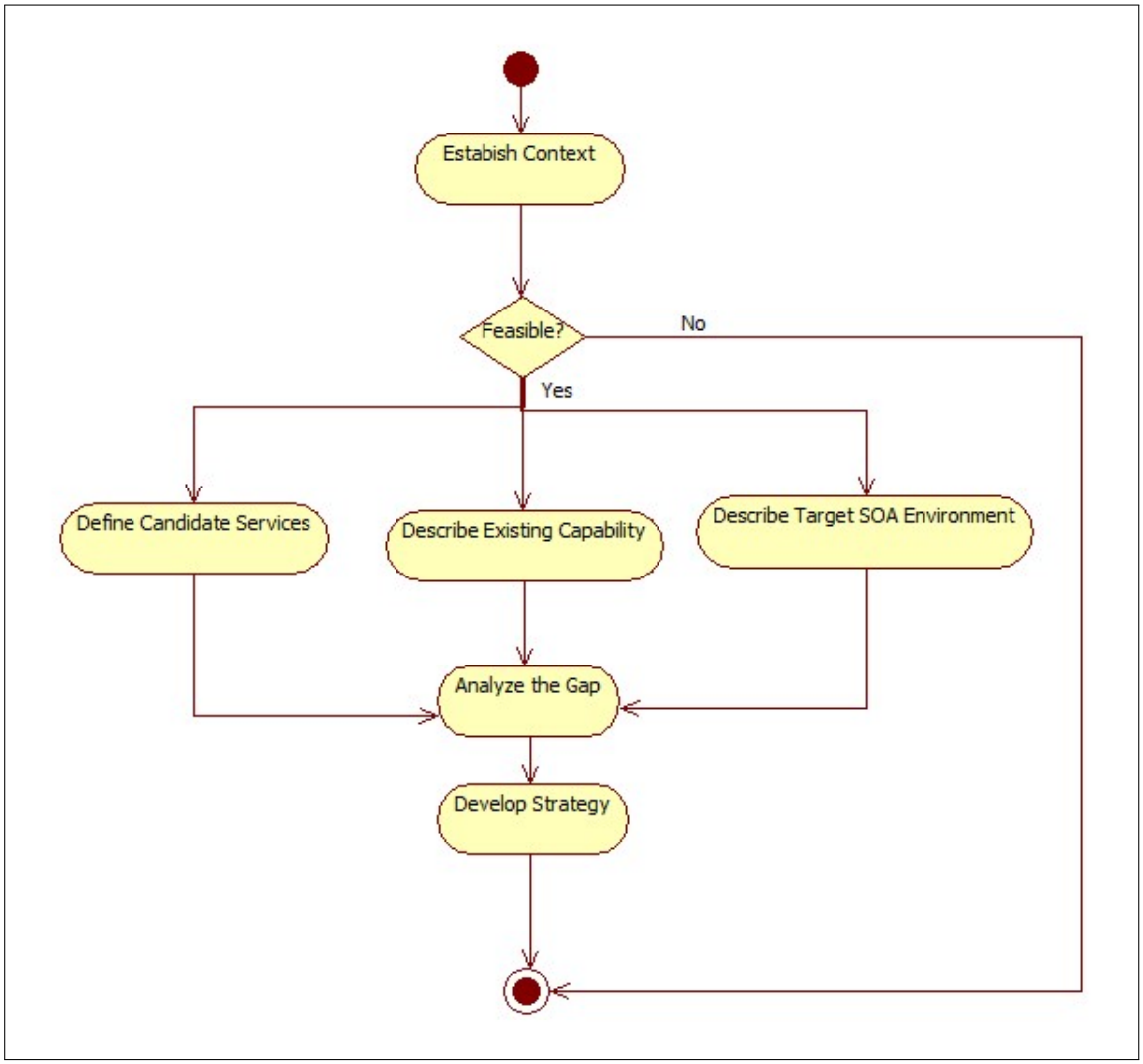

Figure 2.1: SMART work flow. Adopted from [10]

However the numeration assigned to each process may suggest that the processes are strictly sequential, it is not true. Moreover the processes should be conducted in an iterative manner and the order of activities can be changed to align to changes better and better support project specification [50]. Consequently, iterative approach forces to reconsider previous steps if information gathered in current steps influences it. Reconsideration refers also to artifacts obtained as results of the processes. SMART–MP contains one more element which has significant impact on the Process of migration but was not mentioned before. This element is a decision point that is localized after Establishing Context. The decision point provides initial answer for the question: *"Is the project feasible to create?"*.The answer for this question is based on artifacts acquired from the first step and relays on availability of stakeholders and a level of both

current and target system understanding.

SMART – SMF Application of SMART Service Migration Feasibility determinates wherever the project is feasibly for migration toward SOA. This version of SMART is meant for companies that are new in the field of Service Orientation or migration of projects that are characterized by a high risk of migration.

SMART – ESP In opposite to SMF, Enterprise Service Portfolio (ESP) is designed for enterprises that decided to migrate their systems to SOA. ESP supports creation of service portfolio. The portfolio is based on services selected from all the systems across the organization.

SMART ENV The Environment variant is something between ESP and SMF versions. This variant is the best for organization that chosen SOA as their target environment but they are still not completely aware of implications, costs and risks of migration.

SMART System SMART System is designed from a perspective of migration of a current system to a complete Service Oriented environment. SMART System supports identification and creation of services. This variant establishes a complete Service Oriented infrastructure[10].

2.1.2 Advantages and drawbacks of SMART

The SMART migration technique is a technique that examines the migrated system in details. This high level of granularity of the examination brings both advantages and drawbacks.

Advantages

1. The technique is systematic
2. SMART examines the migrated application from different points of view, including risks, costs and migration feasibility
3. The technique produces many artifacts that help in understanding the migrated system
4. SMART has a few variants that are tailored to different needs

Drawbacks

1. Application of the technique requires a lot of time
2. The technique is heavy. It produces a lot of documents
3. The technique does not consider properties of architecture of migrated system

2.1.3 Black–Box: Wrapping

In opposite to White–Box technique, Black–Box technique does not affect the code of the migrated system. The system is considered as a whole entity and analysis of a legacy system is limited only to analysis of inputs and outputs. This section presents briefly an overview of an example of a Black–Box technique, namely Wrapping [23].The description presents two parts of the approach. The first part describes basic elements. The second presents main processes

Elements

Wrapping technique consists of several element that work on Finite State Automation and legacy screens. See picture 2.2 for wrapper schema. The elements are following:

Terminal emulator – replaces server terminal that normally has an access to the legacy system and transfers requests from a GUI based application or requests called via legacy system's API. In other words, terminal emulator provides an interface to access legacy functionality. The implementation depends of terminal type. There are two main types of terminals [23]:

1. Stream Oriented Terminal –communicates with the system via two–directional channel. The terminal sends data just after they are typed. Received data are computed by the legacy system and send back as a screen represented by a matrix of characters.
2. Block Oriented Terminal –in opposite to stream terminal, Block Oriented Terminal uses screens represented by a set of fields with their localisation. The legacy system computes the result and sends back a screen containing a new set of fields with corresponding localisations.

States identifier – Given an input and a start state, the system can reach a finite amount of states. Each state has own screen that is returned by the system as an output. The responsibility of State Identifier is to match Screen Template to the screen returned by the system. To increase matching probability, the legacy screen should be described by the biggest possible amount of data that can be obtained using reverse engineering on the legacy screen.

Repository – stores Finite State Automaton (FSA) files with corresponding Screen Templates, Automation Variables and actions executed by wrapper after reaching this particular state. These actions include [23]:

1. Init –sets Automation Variables based on Service Request Message
2. Set Input Field –sets a value of an input field
3. Get Output Field –gets a value of an output field from the legacy screen
4. Submit Command –triggers transition from one state to another
5. Set –Update Automation Variable –modifies values of Automation Variables based on output of variables of legacy screen and other Automation Variables
6. Build Response –creates a service response based on actual values of Automation Variables

Automation Engine – Automation Engine is a core of the approach. The engine is a component that receives web service request and returns an output. Execution of Engine functionality starts once components receive URI of invoked functionality and corresponding input data. Component execution can be divided into three parts [23]:

1. Start Activity –starts when the engine receives request message. Just after the start a corresponding XML description file is acquired from the Repository and the legacy system is invoked. The first screen returned by the legacy system is obtained from Terminal and a current state is obtained from State Identifier.
2. Interpretation Activity –Interpretation Activities are a set of activities repeated in a loop until the final state is not reached. The set includes execution of current state activities, update of variable buffer called Automaton Variables, transfer of a new data to Terminal Emulator and identification of a new state.
3. Final Activity –when the last state has been reached, engine create an output message based on Automation Variables values.

Processes

In order to achieve fully migrated system, three steps should be executed in a sequence. The steps are following:

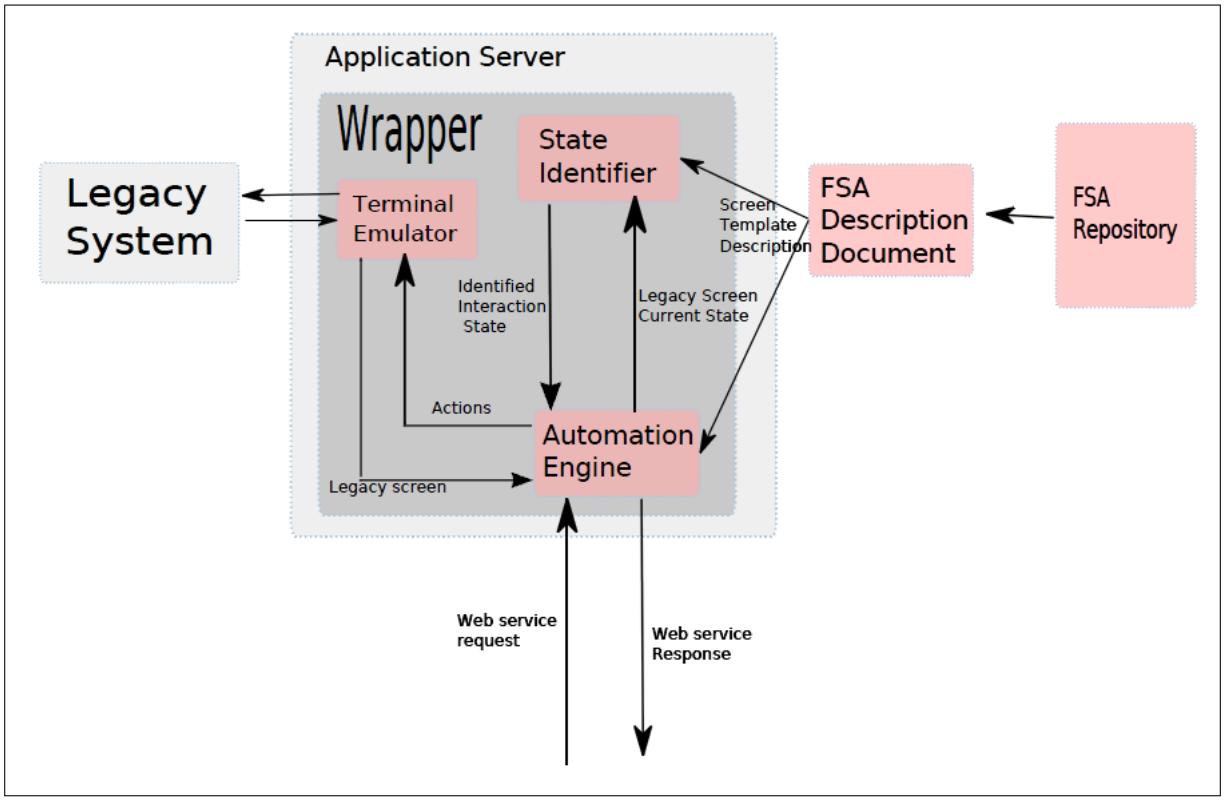

Figure 2.2: Wrapper Schema. Adopted from [23]

Selecting candidate services – This is the first process of the wrapping technique. The process is meant to identify and allocate potential services in legacy system. However the approach notices meaning of introduction of a structural process in order to support service allocation, it does not provide own solution, but rather refers to work of Sneed [76] or SMART [49][10]

Wrapping the selected use case – The second process is a core part of the technique. This process aims at wrapping identified use cases into service. It is also the most complex process which is composed of three sub–processes [23]:

1. Service Identification –aims at selection of use cases that can be transformed to services with respect to previously described characteristic (see 4.3.1)and braking down complex use cases into more elementary [23]. Complex services are not removed from the list services. They became process services that utilise lower level services to perform tasks.

2. Reverse Engineering –during the second sub–activity, the system is analysed in terms of interaction with user in order to gain broad and in depth coverage of main and alternative scenarios. As the result, information about request

/ response messages and corresponding screens are gathered and following outputs are generated [23]:

(a) Finite State Automation – contains all identified states, transitions between the states and transition triggers. The automation is not deterministic because given the same input; user may have a few available paths to follow.

(b) Screen Templates – contain sets of screens that correspond to each state from FSA. Each screen contains features like labels and their localisation. This information is necessary for automatic identification.

(c) Interface –more precisely service contract with all the input and output variables (see 4.2.4).

3. Wrapper Design – the last process makes use of previously created Interaction Model. Designing is meant to make a model interpretable by Automation Engine. In order to achieve interpretability, the model has to be enhanced by an additional information regarding activities executed by Engine on Automation Variables or screen fields. The additional data may contain localization of a field, commands, input /output fields etc.

Deploying and validating the wrapped use case The last process is nothing more than establishing infrastructure for services and testing whether they fulfill requirements [23].

1. Deployment –includes all the information required to export and publish services. The process is dependent on selected technology, for instance: selection of Web Service requires creation of WSDL document [23]

2. Validation –since a significant part of the second process uses automation for creation of Finite State Automation, validation gains in value and becomes a mandatory activity. The Validation aims at identification of failures. The failures may manifest as a lack of screen or an a received message is different than expected. Validation process can be performed in following manner [23]:

   (a) Regressive testing – tests take as an input data used during Reverse Engineering and check whether expected output is delivered by a legacy system. The tests require coverage on scenario level, what means that every scenario of each use case have to be executed at least once [23].

   (b) Path testing – the second approach requires analysis of FSA in order to design tests covering each independent path. This means that all nodes and edges of the FSA need to be visited at least once. Path Coverage is a high level testing that requires a lot of effort to identify all possible paths.

Advantages and drawbacks of Wrapping

Wrapping is a technique that does not examine code of the migrated application. In turn, this approach requires a documentation that is up to date. The migrated system also must be developed as a set of separated artifacts.

Advantages

1. The technique is systematic
2. Execution of the technique is semi automatic
3. There is no need to have code of the migrated application

Drawbacks

1. Documentation of the migrated system is a must. The documentation must be up to date.
2. A full list of use cases is needed. The use cases are described in documentation that may be missing or not maintained
3. The technique bases on inputs and outputs of the system. It is hard to define all possible combinations of input-output pairs.
4. The technique does not consider properties of architecture of migrated system
5. Wrapping requires very complex and time consuming testing at the end of migration.
6. The application must be developed as a set of components that can be separately wrapped.

2.1.4 Gray– Box: Taxonomy analysis

Gray–Box is a compromise between white and Black–Box techniques. It takes analysis the migrated system on both code and system level. The approach presented in this section derives from clustering analysis. The clustering analysis is applied in order to allocate fully functionally and reusable services. Clustering analysis is a sequence of following processes 2.1.4:

Service Identification –Service Identification underlines all the processes associated with migration toward Service Oriented Architecture. In context of the presented approach, the identification is based on analysis of documentation, especially requirement documentation. The identification consists of two steps [84]:

1. Sub domain identification –includes analysis of selected sub–domains in order to define boundaries of the whole legacy system.

2. Further analysis –carries the burden from strict analysis to modelling, especially on modelling of the whole system as UML diagram that is afterwards transferred to third party software. The diagrams are expressed in XML [84]. The model serves as an input for identification of potentially reusable components that latterly may be provided as services. Those services are called *logical services* and they are spanned between concept of service itself and the technical realisation separated from legacy code.

Legacy System Understanding –The second process is intended to estimate potential migration chances and model the system. According to Zhuopeng Zhang[84],a system is suitable to migrate toward SOA when the system contains reusable business functionality that in comparison to the whole system is reasonably maintainable. Moreover, the functionality should bring additional value after exposing it as a service. Having estimation done, a model of the system should be elaborated. Elaboration of the model is based on a re–engineering process and analysis of existing documentation including technical documentation, graphs, interfaces and other existing models. Approach description includes different treatment of procedural software and object–oriented (OO) software. Re–engineering of procedural–application in opposite to OO applications involves usage of additional tools in order to obtain candidate classes and intermediate object–oriented model. In turn, Intermediate Model incorporates UML representation, event driving programming models and object oriented programming language [84]. Object–Oriented approach tries to define system on different levels of abstraction by creation of UML class diagram and control flow graph.

Agglomerative Clustering Analysis – Agglomerative Clustering requires a set of data for reorganization purpose. This set of data in case of migration involves functions, procedures and classes that are further clustered according to feature similarity. The similarity is a key concept in legacy system restructuring[84]. The technique defines features that are used to measure the similarity and determinate whether two entities belong to the same cluster. A feature reflects found identifiers–name. A name can identify any non local entity or a property of the entity including property, name or method name [84].

Then, the algorithm is executed (see [84] for more information about the algorithm).The algorithm produces a dendrogram that describes dependencies

between all the provided data. In order to allocate services, architects have to set cutting points manually and divide the diagram into sub–trees. The sub–tree become services. During service selection, architects have to take into consideration other information retrieved during previous steps as well as the amount of dependencies in order to receive coarse grained and loosely coupled services (see 4.3.1).

Service Packaging and Integration This is the last step of presented Gray–Box technique. The step accomplishes the process of migration from legacy system towards Service–Oriented Architecture. The last step is composed from a sequence of three sub–steps [84]:

1. Refinement –the services obtained as a result of all the previous steps are loosely coupled, but they still contain some dependencies. Refinement increases independence of those services and improve their quality by removal of all the *"dead"* code and unrelated interactions of interfaces. During this process also an interface for service communication is established [84].

2. Component integration and packaging –component integration takes previously refined services and connects them with a *"glue code"* that serves as an adapter in order to transform ingoing and outgoing invocation. Additionally, some differences between domain and retrieved model may appear. The differences should be filled up with new services.

3. Interface design –this step is meant to create interfaces[84],but in fact the output is a contract that specifies exact input and output for each operation of the service. Also an interface –a representation of the contract–is created.

Application of clustering seems to be very promising solution because it base on very basic human activity–classification [4]. Generally, clustering is meant to reorganize groups of entities by taking into consideration their similarity and based on that divide them into more homogenous groups[4]. The similarity can vary as a domain of analysis and can be understood as for instance similarity of properties which is measured with metrics. Method applied in this approach bases on an improved agglomerative hierarchical clustering [4],This technique emphasizes functional aspects [4]that are essential in SOA. The technique is presented as a set of processes and an algorithm.

2.1.5 Advantages and drawback of Taxonomy Analysis

Taxonomy Analysis analyses code of the migrated application in order to identify services and relationships between them.

Advantages

1. Identifies relations between services
2. The technique is systematic
3. Execution of the technique can be performed semi automatic
4. Provides a lot of information about the migrated system

Drawbacks

1. The technique does not consider properties of architecture of migrated system
2. Requires documentation - the documentation of an old system may be missing or not maintained

2.2 Other approaches

The approaches described in the previous section present complex and generic techniques meant to migrate a legacy system to SOA. Their description did not go into details because the techniques need complex tailoring for each particular migration. The next group of techniques migrates systems toward SOA emphasising architecture of the migrated system. This section presents three different approaches utilising benefits brought by architectural patterns. The first approach is a try of standardisation of migration. In fact, the approach does not refer only to migration but also to other activities changing the application. The fact that this is a try of standardisation makes this technique worth to describe. The next example presents modification of architectural pattern (PCBMER) in order to adjust it to SOA. The last approach presents how to create SOA based on Peer–to–Peer architectural pattern

2.2.1 Architectural–Driven Modernisation

Architectural Driven Modernisation is [58] *"the process of understanding and evolving existing software assets for the purpose of software improvement; modifications; interoperability; re–factoring; restructuring; reuse; porting; migration; translation into another language; and enterprise application integration."* The process describes modernisation of an architecture on three levels of abstraction [79]:

1. Technical Architecture Modernisation –includes migration between both platform and language or just platform or language. The migration may be caused by technology obsolesce, lack of further modernisation and maintenance possibilities or other technological factors.

2. Application and Data Architecture Modernisation –is modernisation on a higher level of abstraction that includes changes to architecture of the system and may require changes in structure of the stored data. The modernisation includes also Technical Architecture Modernisation. The modernisation may be needed, because the application due to incremental changes may become monolithic and impossible to maintain block of code.

3. Business Architecture Modernisation –is the most complex and the most risky type of modernisation. The risk is strengthening by lack of any standardisation in the domain of migration of legacy assets. The complexity is a result of the scope of the modernisation process. The result includes both previously described modernisations.

Analysis of the modernisation processes within the context of Service Oriented Architecture allows making an assumption that the final version of the standard will align to migration towards SOA due to mapping between SOA structure and modernisation scope. While modernisation on a technology level corresponds introduction of SOA infrastructure into the application, modernisation on application and data architecture matches changes required during identification and specification of services.

Substandards – However Architecture–Driven Modernisation is presented as one standard, in fact it is a set of sub–standards that include different aspects of modernisation. There are following standards [64]:

1. ADM: Knowledge Discovery Meta–Model (KDM) –establishes an initial meta–model that describes the structure of the system and data model, but does not describe the system below procedural level.
   Status ow work: Adopted in 2006

2. ADM: Abstract Syntax Tree Meta–Model (ASTM) –starts where KDM finishes, namely ASTM describes the system below procedural level what guarantees a full representation of the system.
   Status ow work: Rolled out in 2009

3. ADM: Pattern Recognition –tires to identify structural aspects of the system by recognising both pattern and anti–patterns in order to define requirements and opportunities for refactoring and transformation of the system.
   Status ow work: Work started in 2009

4. ADM: Structured Metrics Package (SMM) –uses KDM in order to derive a set of various metrics describing different attributes of the system. Metric coverage includes technical, functional and architectural aspects of the system.
   Status ow work: Rolled out in 2009

5. ADM: Visualization –illustrates meta–data collected in KDM. The visualisation includes charts, graph, views, metrics and UML models.
   Status ow work: No target date set

6. ADM: Refactoring –includes modernisation of an application by structuring, rationalising and modularising.
   Status ow work: No target date set

7. ADM: Transformation –brides existing application with the target system by defining possible mapping and transformations. It is worth to underline that the standard will target at usage of Model–Driven Architecture (MDA) paradigm.
   Status ow work: No target date set

Those specifications cover many different aspects of modernisation but in context of this research mainly Pattern Recognition is very interesting. The fact that Pattern Recognition is becoming a part of standard emphasises contribution of pattern during migration. The status of work is taken from [64]. The document is dated February 2009 but it still available on the web page.

2.2.2 PCBMER

Presentation–Controller–Bean–Mediator–Entity–Resources (PCBMER) is an architectural pattern elaborated by Lech Maciaszek. The pattern is dedicated for highly adaptive systems [53]. The pattern is based upon Holistic theory. The theory says that a holon in a recursive entity that can contain other holons inside. The theory is also enriched by Holarchy (Hierarchy of holons) [53]. The architecture consists of six following layers:

1. Presentation –is a top level layer and represents User Interface (UI). The interface provides information contained in beans from underlying Bean layer.
   Depends on: Bean Layer, Controller Layer

2. Controller –corresponds to Controller from MVC or PAC architectural pattern. A controller encapsulates an enterprise business logic which is invoked by Presentation Layer
   Depends on: Bean Layer, Entity Layer, Mediator Layer

3. Bean –represents information meant to be displayed on UI. A single bean is an object that consists of entities from Entity layer as well as additional properties.
   Depends on: Independent

4. Mediator –mediates between Resource Layer and Entity Layer. The purpose of the layer is to manage business transactions, enforce business rules, instantiate business objects in Entity Layer and manage the memory cache of the application.
   Depends on: Resource Layer, Entity Layer

5. Entity –represents all the business objects within the application.
   Depends on: Independent

6. Resources –mediates between application and data sources. The source can be a database, service or other external entity containing required information.
   Depends on: External Resources

The pattern fits well to for business solution and as the business solutions become a subject of integration. The integration presented by L. Maciaszek aims at integration with Service Oriented Architecture. The pattern is modernised in order make the integration possible. The modernisation of the pattern is named PCBMER–U (Utility) and introduces three new elements into the pattern(see figure 2.3).

1. Broker –is a message broker

2. Orchestration –encapsulates and executes a new business logic (see 4.3.2 for more information)

3. Service Registry –is responsible for service discovering and depends only on mediator.

The modification seems to have some contradiction. The first is that according to description, Registry depends on *"utility's business logic in Mediator"* [52] what in fact is quite interesting because Mediator Layer does not depend on Controller Layer, consequently Mediator layer is not "aware" of Controller layer. Registry may not discover higher level services located in Controller. The next potential contradiction is where Orchestration is introduced. According to figure 3 in [52] Controller depends on Orchestration what seems to be against role of Orchestration, because in fact Orchestration / Process Services depend on underlying services what in this case may mean that Orchestration depends on services allocated in Controller layer. The last contradiction refers to the figure 3 and the description. Description says that Broker and Orchestration depend on logic in Controller Layer while the dependency in the picture is opposite. The same situation refers to Mediator Layer and Registry.

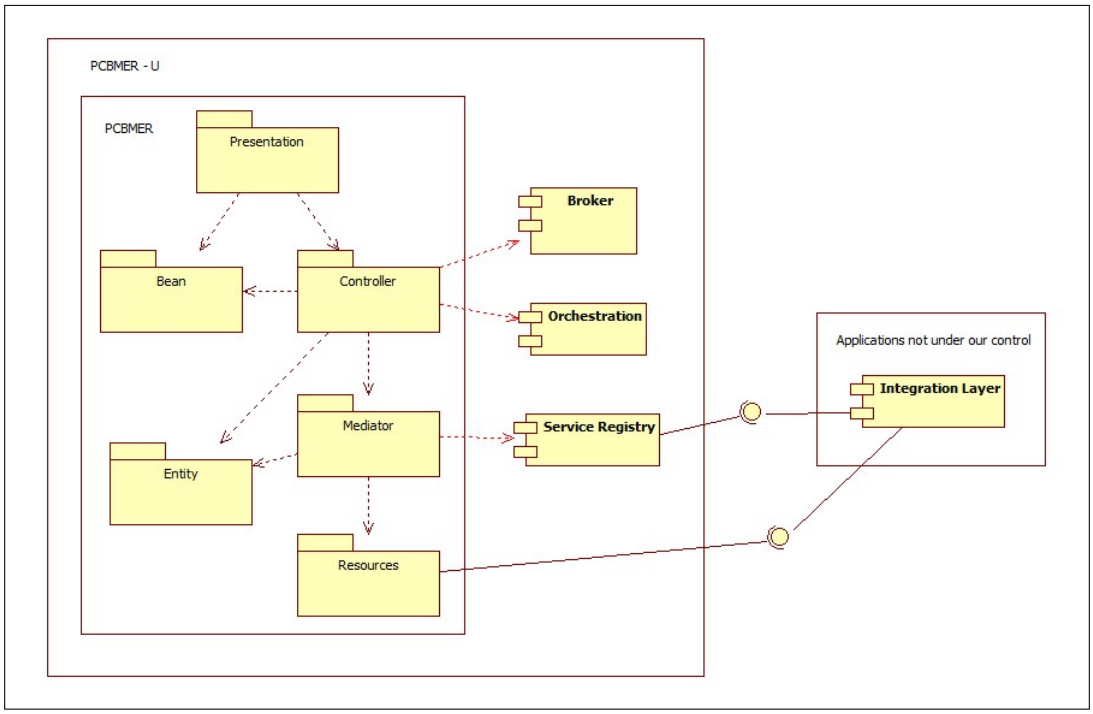

Figure 2.3: Relation between PCBMER and PCBMER–U, adopted (figure 3) from [52], potentially inconsistent dependencies are marked in red

2.2.3 Peer–to–Peer

Available literature provides also an example of mixture of both Service Oriented and non–SOA patterns. The approach proposes a union of services and Peer–to–Peer infrastructure. The infrastructure is build upon Gradient Topology and Distributed Hash Table. Gradient Topology supports efficient Peer discoverability [29],Distributed Hash Table (DHD) provides efficient routing algorithm that enables consumption of services [69] and can be successfully employed as a mean of service discoverability and registration [50]. The infrastructure has also one more important element of the Peer–to–Peer pattern, namely a set of servers used as start up points for other peers (see 14 for more P2P related information). Gradient Topology supports also service registration through decentralized Service Registry [69].In order to register, a service (peer) has to discover an instance of service registry via gradient topology. Then the service registers itself and uploads own proxy via DHT. After this sequence of steps a service can be discovered by other services.

2.3 Summary

This chapter presents briefly main Service Oriented Migration approaches. The first three techniques are the most mature and have the more detailed documentation. SMART (White–Box) technique analyses the code in details. Wrapping (Black–Box) wraps the legacy system into services and does not integrate into code of the migrated legacy system. Taxonomy Analysis (Grey–Box) analysis the code of the migrated system. As the result of analysis a dendogram is created. The diagram is further converted into sub–threes that are implemented as services. Remaining three approaches differ in scope of migration. ADM migrates whole systems by migrating their infrastructure, architecture and business layer. In turn PCBMER is migrated to SOA using additional layers. The last technique applies architectural pattern on more abstract level. The whole communication within system is organised as a Peer–to–Peer network. Study of architectural patterns becomes a part of standard. Migration of the system will base upon white or black or grey technique.

Chapter 3

Architectural Patterns

The goal of this chapter is to select an architectural pattern that will be migrated to Service Oriented Architecture (SOA). Selection of a proper pattern is very important. The selection must eliminate patterns that are not feasible for migration, the patterns that are already migrated to SOA and the patterns that are already part of SOA. Additionally the selected pattern should be popular. It means that the pattern should be often implemented in real systems. Guidelines for a rarely used pattern have reduced application. Since there is a number of architectural patterns that differ from each other and the thesis is not meant to migrate all of them, a systematic approach for selection of the pattern for migration is needed. The selection bases on a number of steps. Each step is a filter that is meant to reduce number of candidate patterns or prepare the patterns for the next filter. Filters are lined so the output of one filter is the input for another. The filters (in order of their application) are following:

1. Selection of literature sources
   The idea of architectural pattern along with example patterns is described in a great number of publications. This filter selects several literature sources that will be the base for identification of architectural patterns.
   *Input:*Literature about architectural patterns
   Output: Selected literature sources

2. Selection of architectural patterns from the literature sources
   Literature sources selected by the first filter contain different types of patters (see 3.1.1 for types of patterns), but only architectural patterns are in scope of the work. This filter removes all the non-architectural patterns.
   Input: Selected literature sources
   Output: List of architectural patterns

3. Removal of patterns that exist in only one source
   This is the first filter that removes the less popular architectural patterns. All the patterns that exist in only one literature source are removed. This assures that the most unusual and rare patterns are not taken in the further consideration.

Input: List of architectural patterns
Output: Architectural patterns that exit in more than one literature source

4. Assignment of architectural patterns to selected category
   In opposite to the previous filters, this filter does not reduce amount of architectural patterns. It organizes previously identified patterns. The output of this filter is a list of architectural patterns assigned to categories. Both identified and chosen categories are described.
   Input: Architectural pattern that exits in more than one literature source
   *Output:*Categories of architectural patterns and architectural patterns assigned to those categories

5. Selection of representatives for categories
   At this point, architectural patterns are grouped into several categories. Each category contains several patterns. This filter reduces number of candidate architectural pattern by selection of patterns that represent particular categories. Representatives of categories are patterns that have the simplest structure. In addition, application of the patterns is similar to application of other patterns in its category.
   Input: Categories of architectural patterns and architectural patterns assigned to categories
   Output: A list of patterns that represent their categories

6. Removal of rarely interacting patterns
   This filter bases on mutual interaction of patters (see 3.3). Mutual interaction of patterns is important because systems often base on several architectural patterns - pattern language. The choice of applied patterns needs to be carefully considered. Proper choice of patterns strengthens the system. Wrong choice of patterns that are applied together can bring some side effects like performance drop or limited testability. Patterns that bring side effects when they are applied with other patterns are less often implemented.
   Input: A list of patterns representing categories
   Output: A list of representatives that does not contain the less mutually interacting architectural patterns.

7. Prefeasibility study
   The filter finds literature about architectural patterns in context of SOA. The filter will removes patterns not fulfilling at least one of the following criteria:

   (a) A pattern has to exist in literature in context of SOA

   (b) A pattern cannot be already migrated to SOA using guidelines

(c) A pattern cannot be recommended as not suitable for migration to SOA

(d) A pattern cannot be already a part of SOA

Input: Representatives of categories without the less mutually interacting architectural patterns.
*Output:*Architectural patterns that are feasible for migration to SOA.

8. Final Selection
   This is the last filter. The filter selects a pattern that will be further migrated to SOA. The selection bases on literature gathered during prefeasibility study.
   Input: Architectural patterns feasible for migration to SOA
   Output: Architectural pattern that will be migrated to SOA

Beside the main flow of the filters, the chapter presents some background information. The information helps to understand steps of the procedure and motivation behind them.

This chapter is organized in following manner:

1. *Patterns* – presents basic information about patterns including description of types of patterns in Software Engineering. This section addresses also the problem of distinguishment of architectural patterns from design patterns. The section provides an initial list of patterns that bases on three sources: a series of books and two articles. Finally, the list of patterns is reduced by removing the patterns that appear in only one source. This list is further categorised in the next section.

   Covers filters: Selection of literature sources, Extraction of architectural patterns from the literature sources, Removal of patterns that exist in only one source.

2. *Categorisation* – the first section provides nineteen patterns. This amount is too large to choose from, thus it still has to be downsized. This section describes patterns from the last section by providing their overview, description of elements, description of relationships between elements and a picture presenting example application of the pattern. Afterwards, several existing ways of pattern categorisation are presented. One of the methods is selected and previously selected patterns are assigned to categories according to the selected method. Finally, the patterns inside each category are compared to each other and representatives of the categories are chosen. The representatives are further reconsidered.

   Covers filters: Assignment of architectural patterns to selected category, Selection of representatives for categories.

3. *Mutual Interaction* – this section describes pattern language and its role during system design. Next, result of study of forty-seven real documentations of systems is presented. The results of the study present frequency of usage of particular patterns and common pairs of patterns. Finally, the list of representatives of the categories is confronted with results presenting occurrences of pairs of patterns.

   Covers filters: Removal of rarely interacting patterns.

4. *Pattern selection* – this section presents selection of the pattern for migration. The selection requires a research helping to determinate which pattern is the most feasible to migrate. The assumption is that a pattern is feasible to migrate if there are any works describing attempts of migration of the pattern in context of SOA. The more works and the more elaborated they are, the higher feasibility is. Additionally, if the pattern occurs to be migrated to SOA or the pattern is already a part of SOA then this pattern will be discarded because the solution already exists. Finally, a detailed description of the selected architectural pattern is presented.

   Covers filters: Prefeasibility study, Final selection.

5. Summary – a summary of the chapter

3.1 Patterns

Software Engineering is a domain that covers practical as well as theoretical aspects of software development. Software Engineering is also a term that appeared for the first time during NATO conference in 1969 year [57]. Since then, the domain evolved and adapted new engineering solutions. One of such adapted solution is the concept of patterns. This section is meant to provide basic information about patterns in Software Engineering emphasising Architectural Patterns. The section provides also a list of Architectural Patterns. This section is broken into following subsections:

1. *Patterns in Software Engineering*– presents the origin of Patterns in Software Engineering and describes groups of those patterns.

2. *Definition of Architectural Patterns*– presents definition of architectural patterns.

3. *Sources of Architectural Patterns*– presents sources of Architectural Patterns used during selection of pattern for migration.

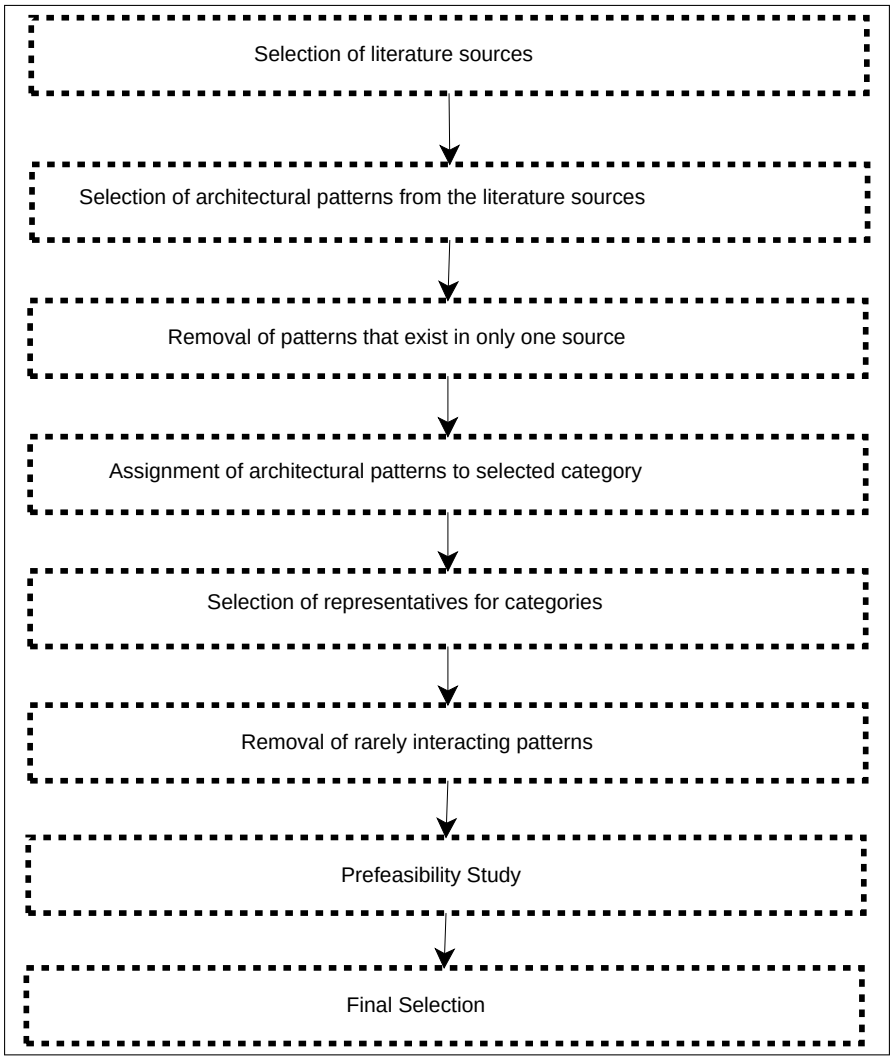

Figure 3.1: Procedure of selection of Pattern for migration

4. *Identified Architectural Patterns*– presents a list of identified Architectural Pattern. The identified patterns are selected from literature presented in section *Sources of Architectural Patterns*. The list is further reduced to the patterns existing in two out of three sources.

3.1.1 Patterns in Software Engineering

Pattern is not a brand new idea. The first person who defined a pattern as [56] : *"A recurring solution to a common problem in a given context and system of forces."* was Christopher Alexander [24] [56] – an architect born in Vienna. He published (1977) in his book more than two hundreds of architectural solutions, which included both very general and abstract tasks like city planning as well

as very detailed tasks like room planning. The Alexander's idea inspired Kent Beck and Ward Cunningham. They presented their observations and adaptations of patterns to Software Engineering during Object Oriented Programming, System and Language (OOPSLA) conference in 1987 [56]. However , the origin of patters are patterns used in architecture, the patterns in Software Engineering do not refer only to architecture of system. The general classification presented in *"Patterns in the field of software engineering"*[56] groups patterns into three main categories that in turn are divided into seven subcategories (see figure 3.2. The main categories are[56] :

1. Implementation patterns – present implementation of the system from different points of view. Those patterns are further broken into four more categories.
   (a) Reference Architectural Patterns – this category contains patterns that are dedicated to a particular domain [56].
   (b) Design Patterns – in general, design patterns define, explain and solve design problems in an object oriented system [56].
   (c) Analysis Patterns – those patterns are rather a group of concepts that represent a common modeling in business [56].
   (d) Architectural Patterns – architectural patterns provide predefined subsystems with relations between them and responsibilities of those subsystems [56].
2. Process and improvement patterns – this type of patterns present solutions for tasks that are common during development and improvement activities. The category is further broken into two subcategories:
   (a) Process Patterns – the patterns show and guide how to conduct specific tasks in the development of process [56].
   (b) Software process Improvement Patterns – define improvements for already introduced processes [56].
3. Configuration management patterns – contains only one subcategory.
   (a) Software configuration management patterns – the patterns describe processes that are associated with whole lifecycle of a product(see [56] for more detailed information about types of patterns).

3.1.2 Definition of Architectural Patterns

Despite the fact that the classification [56] of patterns in Software Engineering clearly separates Architectural Patterns from Design Patterns, it is not always

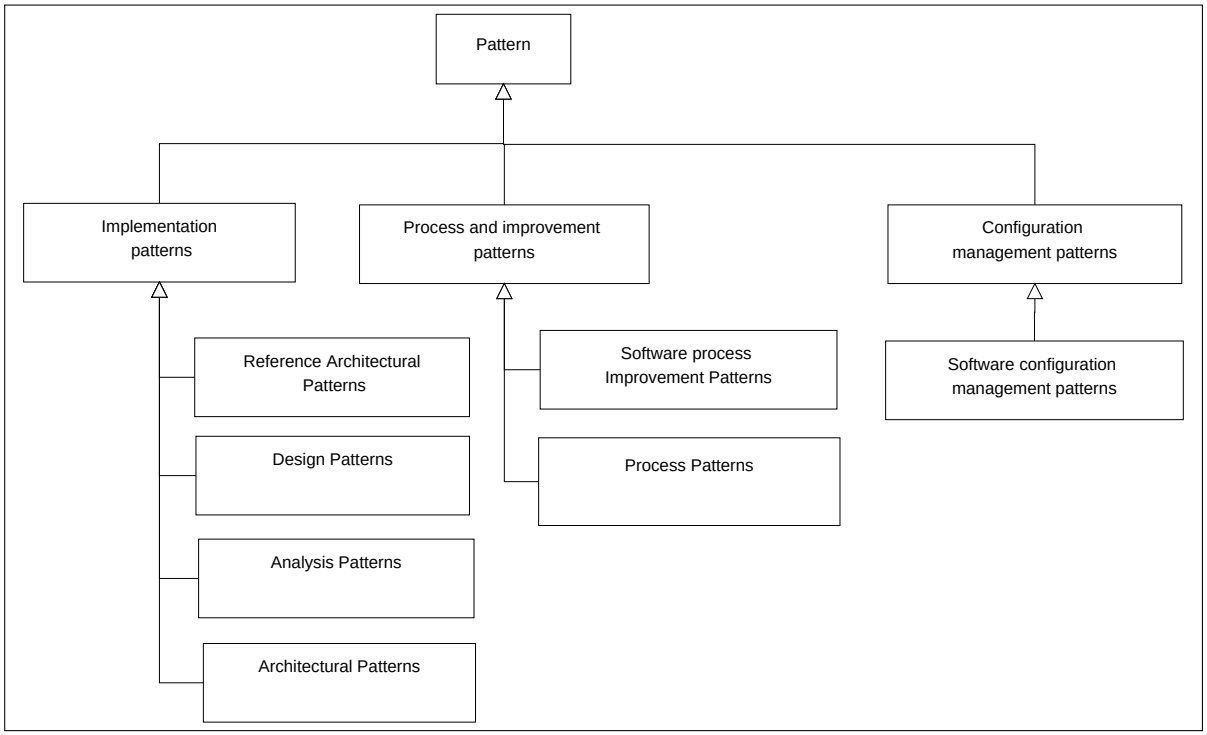

Figure 3.2: Patterns in Software Engineering

obvious and both terms are used interchangeably. In order to cope with this problem, a clear definition of both types of patterns is needed. Scientific articles, books and encyclopedias provide many different definitions of architectural and design patterns; therefore, to get better understanding of those terms several definitions will be analysed.

Frank Buschmann, author of a series of books about patterns and architectures describes architectural patterns in following manner [22]:

> *"Architectural patterns express fundamental structural organisation schemas for software systems. They provide a set of predefined subsystems, specify they responsibilities and include rules and guidelines for organising the relationships between them."*

Other people who contributed to researches in the domain of architectures and patterns are Paris Avgeriou and Uwe Zdun. They define architectural patterns as follows [43]:

> *"Architectural patterns provide solutions to recurring problems at the architecture design level. These patterns not only document 'how' solution solves the problem at hand but also 'why' it is solved, i.e. the rationale behind this specific solution."*

Both presented definitions present architectural pattern as a pattern applied on level of architecture. In turn definition of Design Pattern given in [35] says that:

> *"Design Pattern describes a commonly–recurring structure of communicating components that solve a general design problem in a particular context."*

Other definition [22] says that:

> *"Design patterns are medium–scale patterns. They are smaller in scale than architectural patterns, but are at a higher level than the programming language–specific idioms. The application of a design pattern has no effect on the fundamental structure of a software system, but may have a strong influence on the architecture of a subsystem."*

Architectural patterns are the patterns that have an impact on architecture of the whole system while design patterns may have an impact only on subsystems. Even having the definitions, classification of a specified pattern to a particular category is not so obvious. In some specific circumstances a design pattern can be implemented as an architectural pattern. This can happen when a system is very specialized and subordinated to a specific task that imposes solutions like design patterns. In this case, elements of a design pattern can grow to the size of subsystems and determinate architecture. In order to provide a better explanation of this concept, an example imaginary system ("DataProvider") will be considered.

The "DataProvider" system provides information about individuals to requesting and authorized institutions. Information provided by the system can be general like personal data or detailed like pictures and fingerprints. Database of the system contains thousands of records and it is distributed over the country. One of the most important aspects of the system is availability of the system and its resources. The system is queried twenty-four hours, seven days a week. The system will be designed as a distributed application. The system has to be stable and has to have high availability. The description of requirements of the system matches one of the design patterns, namely lazy acquisition design pattern. Lazy acquisition design pattern [45] has two very important properties what makes it useful to apply here:

1. Availability – all downloadable data are available, but the data is not acquired in advance. This approach saves resources of the system what is especially important when the system has to process a lot of data (especially multimedia).

2. Stability – all resources are acquired only when needed so the system does not have to overuse memory and other resources. Overview of the design pattern is presented on the figure 3.3

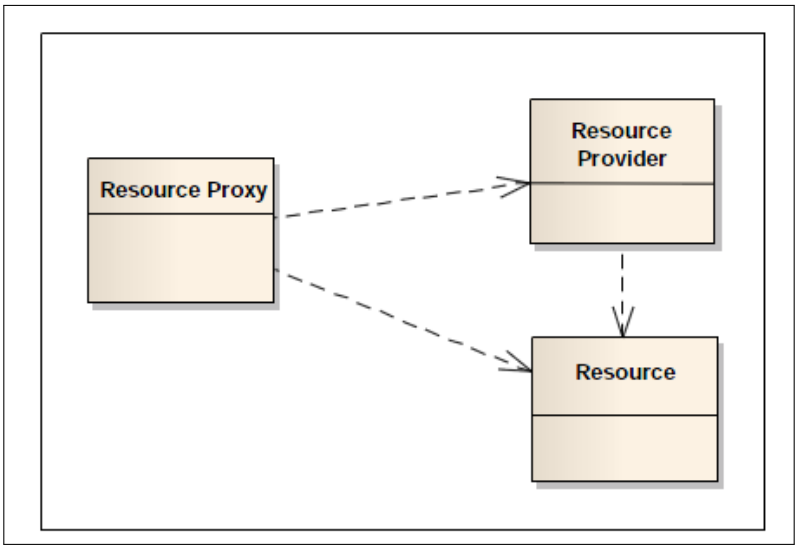

Figure 3.3: Lazy Acquisition design pattern

Role of particular elements:

1. Resource proxy – simulates requested resource by providing an interface that is identical with interface of the requested resource. provider.
2. Resource provider –provides resource to resource proxy.
3. Resource –contains resources that are obtained through the resource proxy.

It is possible to make a whole system implemented using this pattern. The elements of design pattern have to be converted into subsystems. Additionally, an access interface is needed. Resource provider becomes a subsystem responsible for providing resources. It is invoked by access interface. This subsystem invokes Resource Proxy subsystem. Resource Proxy is responsible for converting obtained resources into proxies. The system also lazy initiates the proxies if it is needed. Resource subsystem is responsible for providing required resources. The subsystem maintains also connections to database and executes other database related operations. The overview of architecture of this system is determined by a design pattern what in this case makes it an architectural pattern because the pattern does not affect only particular subsystem as definition of Design Pattern says. It defines the subsystems and interactions between them (see figure 3.4). Application of the design pattern matches to previously stated definition of Architectural Pattern.

Applying design patters as architectural pattern is not the only problem in pattern distinguishing. How close are those two types of patterns can be seen in literature [34], where Model View Controller (MVC) pattern is described as

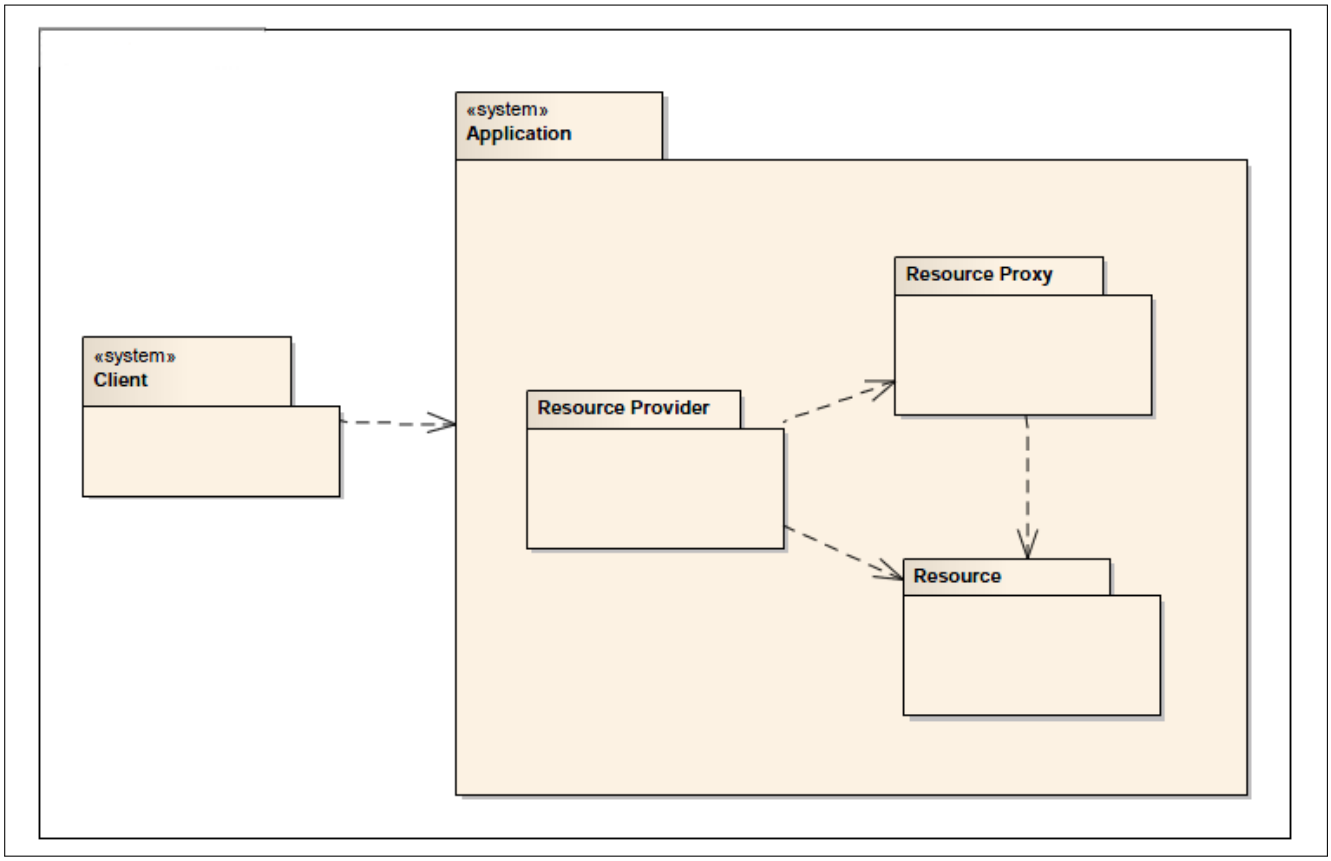

Figure 3.4: Lazy Acquisition as architectural pattern

a composed design pattern, while Bushman for instance presents MVC as an architectural pattern [22]

3.1.3 Sources of Architectural Patterns

Filter 1: Selection of Literature Sources
The idea of Architectural Patterns in Software Engineering is relatively old and popular thus there are a number of publications about this particular type of pattern. Unfortunately, there is no standardized set of architectural patterns. Different authors present their own definitions of architectural patterns and propose their own categorisation of those patterns. The process of selection of an architectural pattern for migration to SOA requires a number of patterns to revise in order to select the most appropriate pattern. Due to a significant number of publications, it is impossible to identify all Architectural Patterns in Software Engineering. The initial list of patterns has to be created based on a sized down list of publication. The initial list of patterns bases on three sources:

1. The first and the most elaborated source of architectural patterns is a series of books by Frank Buschmann. The series consists of five volumes [22][71][45][20][21]. The first volume provides general information about Architectural Patterns and Design Pattern. It provides also some descriptions of example patterns. The next three volumes describe patterns for specific domains like concurrency or resource management. The last volume summarizes all the presented knowledge and introduces pattern languages, which provide motivation for usage of many different patterns together. The series provides comprehensive description of a number of patterns. The number to large to consider during this study, therefore some selection has to be conducted. The selection is based on description of the patterns. It chooses only patterns that are clearly named as architectural. The selected patterns are presented in the next section.

2. The second source is an article written by Nail B.Harrison and Paris Avgeriou. The document *"Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation"*[38] presents results of analysis of forty seven documentations of systems. The document includes information about pattern grouping, number of applications of particular patterns as well as found mutual interaction between patterns and amount of those interactions. The document was selected because it describes architectural patterns in real systems.

3. The last source presents results of architectural pattern categorisation elaborated by Paris Avgeriouis [8].The main purpose of the document is to group patterns into pattern languages. In order to conduct grouping into languages, the author presents definitions and example usage of architectural patterns.

3.1.4 Identified Architectural Patterns

Filter 2: Extraction of architectural patterns
Filter 3: Removal of once appearing architectural patterns
Analysis of the selected literature sources identifies thirty two patterns. The patterns are listed in the table 3.1.

Pattern name	F. Buschmann	Neil B. Harrison	P. Avgeriou
Layers	✓	✓	✓
Pipe and filters	✓	✓	✓
Blackboard	✓	✓	✓

Continued on Next Page...

Pattern name	F. Buschmann	Neil B. Harrison	P. Avgeriou
Broker	✓	✓	✓
Model View Controler	✓	✓	✓
Presentation –Abstraction –Controller	✓	✓	✓
Microcernel	✓	✓	✓
Reflection	✓		✓
Interceptor	✓	✓	✓
Reactor	✓		
Proactor	✓		
Half–sync / half–async	✓	✓	
Leader Follower	✓		
Shared repository	✓	✓	✓
Domain object	✓		
Messaging	✓		✓
Message channel	✓		
Client – Server	✓	✓	✓
Explicit invocation	✓	✓	✓
Plug – in		✓	
Peer–to–Peer	✓	✓	✓
C2		✓	✓
Publisher–Subscriber		✓	✓
State transition		✓	
Interpreter			✓
Active repository		✓	✓
Remote Procedure Call		✓	✓
Implicit Invocation	✓	✓	✓
Indirection Layer			✓
Batch Sequential			✓
Virtual Machine			✓
Rule–Based System			✓

Table 3.1: Summary of architectural patterns investigation

The check marks set in the table 3.1 mark the existence of a particular pattern in a specific source. Gray rows mark patterns that were mentioned in only one

source. Those patterns are not taken into consideration during next steps. The table contains also Publisher-Subscriber. Publisher–Subscriber is also rejected because it is a known design pattern [22][34] while the table should contain only architectural patterns. This exception is important because it confirms previously addressed problem that says that it can be very hard to distinguish both architectural and design patterns from each other. Moreover, authors of the second source claim that all architectural patterns were checked before in terms of their adherence to domain of architectural patterns. This finding means that other sources may also present a design pattern as an architectural pattern.

3.2 Categorisation

In the previous section, nineteen architectural patterns out of thirty two were chosen. Analysis of feasibility of migration of this number of architectural patterns would require a lot of effort, consequently, the number of pattern will be reduced during the next selection. The selection is based on categorisation of patterns. The patterns are grouped and the most general pattern is chosen as a representative. This approach reduces considerably amount of patterns to analyse. The *most general* means that a pattern can be compared to other patterns because they have similar structure and purpose. Additionally, its structure is the simplest. The representatives of each category reconsidered in next sections. This section presents process of selection of representatives of categories. The section is organized in following manner:

1. *Description of selected Architectural Patterns* – the subsection provides descriptions of patterns selected in the previous section. Each description has four elements: overview, elements and relationships between elements. Additionally a picture presenting an example usage of a pattern is provided. Since P. Avgerious in opposite to authors of other sources does not provide descriptions of patterns in his work, the descriptions and pictures base on remaining two sources of patterns.

2. *Methods of patterns categorisation* – this subsection presents shortly identified ways of pattern categorisation. Since creation of own categorisation method is not in scope of the work, one of presented approach is chosen.

3. *Allocation of pattern to categories* – the purpose of this subsection is to allocate patterns described in the first subsection to categories of categorisation method selected in the second subsection.

4. *Selection of representatives* – this section provides comparison of patterns within categories. As the result of comparison, representatives of categories are selected. The pattern for migration will be selected from those representatives.

3.2.1 Description of selected Architectural Patterns

This section presents descriptions of previously selected architectural patterns. The description includes short overview of the pattern, elements of the pattern, relationship between the elements and a picture presenting an example usage of the pattern. Figure 3.5 presents notation used in figures presented example usage of architectural patterns.

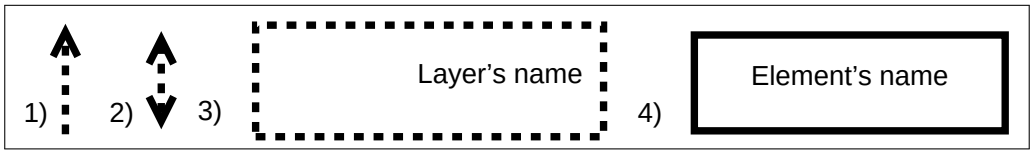

Figure 3.5: Notation used in example application of patterns

Description of the notation

1. *Unidirectional relationship*– the relationship is represented by a dashed line with an arrow on one end. An arrow on the top of the line points direction of the relationship. An element having end of line without an arrow can start interaction with the element pointed by the arrow.

2. *Bidirectional relationship*– the relationship is represented by a dashed line with arrows on both ends. In this case, elements on both sides of line can start an interaction.

3. *Layer*– it is represented by a dash-lined rectangle. This is a grouping element having abstract meaning depending on description of the pattern. It can be linked using both types of relationships. It can contain other layers and elements. A layer is identified by a name.

4. *Element*– it is represented by a solid lined rectangle. An element has a specific meaning dependent on particular pattern. An element is identified by a name.

Architectural Patterns The architectural patterns are listed in the order they appear in the table 3.1. Their description is following: tab:PatternSummary

1. Layers
   Overview – This pattern imposes encapsulation of the similar functionalities in containing layers what helps to separate low-level functionalities from high level.
   Elements

Layer – A layer encapsulates common logic on the same level of abstraction. Each layer provides an interface allowing another layer to cal its functionality.
Relationships – A layer can use other layer only if they are connected. There are uni and bi directional relations between layers. This pattern is characterized by lack of direct calls of layers to non-neighbor layers. If the layer number one wants to call the method from the layer number three and there is no direct connection between them, the layer number one has to call the layer number two in order to call the layer number three. A layer can invoke only a layer that is bellow or above this layer. Connection limited to only upper and lower layer increases maintainability but reduces performance (see figure 3.6).

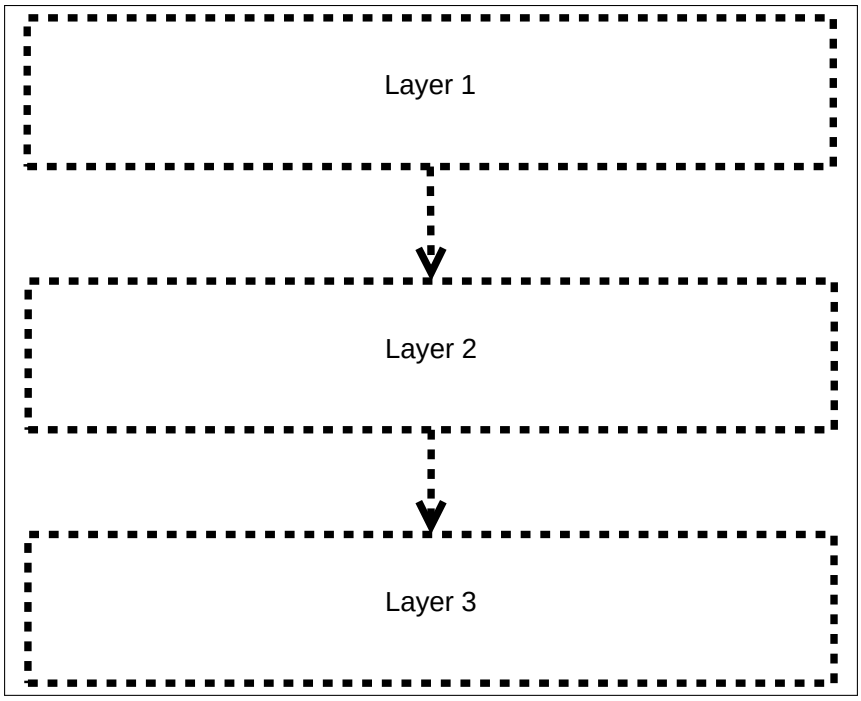

Figure 3.6: Example usage of Layer pattern

2. Pipe and filters
Overview – The pattern is characterized by a good and efficient way of data processing but the structure of the pattern reduces its reliability because if one of the filters fails, the whole process fails as well.
Elements
Filter – processes data. A filter can push data to a pipe or pull it out from a pipe.

Pipe – forwards processed data to next filter.
Relationships – Each pipe connects two filters and each filter has to be connected to at least one pipe. There is no possibility to connect a pipe to a pipe or a filter to a filter directly. Additionally, pipes offer only unidirectional connections (see figure 3.7)

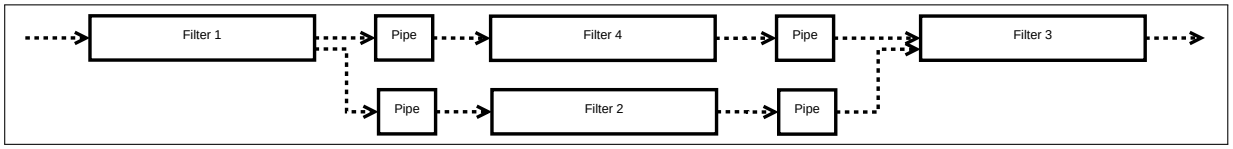

Figure 3.7: Example usage of Pipes and Filters pattern

3. Blackboard
Overview – the pattern is very useful when no deterministic solution is known. The pattern improves solution-finding process that bases on small deterministic solutions.
Elements
Blackboard – contains all currently gained knowledge or solutions. The blackboard serves also as a place for future solutions / knowledge
Knowledge source – provides more advanced or more complete solution based on own data and blackboard's information.
Moderator – moderates knowledge improvement. The element determines the order in which every source of knowledge has to access the blackboard.
Relationships – Moderator is in relationship with Knowledge Source and Blackboard. Knowledge Source can operate only on Blackboard. Blackboard does not invoke other elements of the pattern (see figure 3.8).

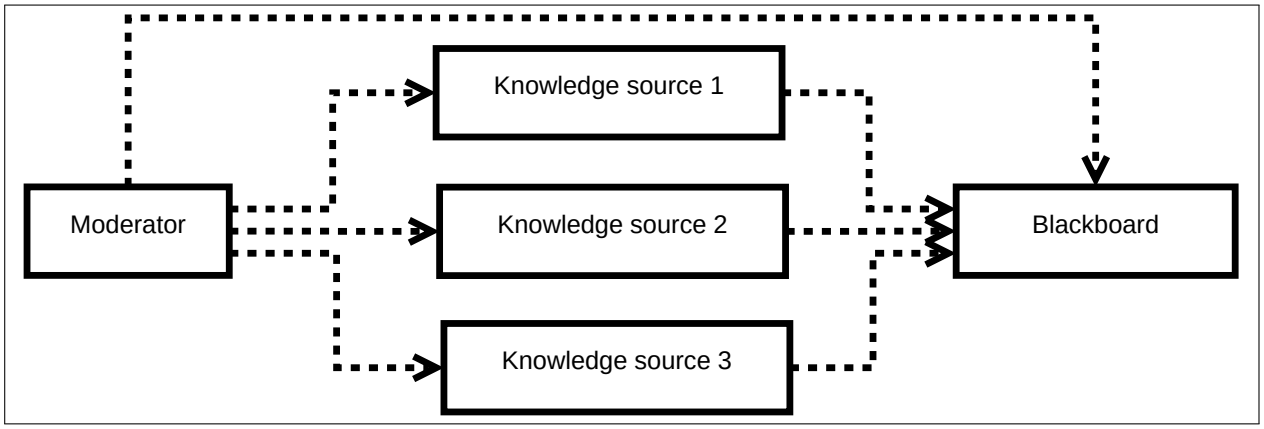

Figure 3.8: Example usage of Blackboard pattern

4. Broker
   Overview – Broker is an architectural pattern that organises communication within distributed system.
   Elements
   Client – is a user application that sends requests to the broker.
   Broker – has to maintain all requests, provides security and loadbalancing.
   Node – is a part of the system that encapsulates logic of the system.
   Relationships – client as a requesting element connects to a broker. The client does not know location of nodes. Locations of nodes are known only to the broker. The nodes do not know location of the clients, but they know location of the broker. This knowledge is required because each node has to register itself in the broker. Connections between broker and nodes are dynamic, it means that any node can connect and disconnect in runtime (see figure 3.9).

5. Model View Controller (MVC)
   Overview – the pattern is dedicated for systems that emphasise role of information displayed to users. Due to clear separation of presentation part, the pattern supports development and simplifies maintenance of this sort of systems.
   Elements
   Model – represents represents data of the system and functionalities to manipulate them
   View – displays information
   Controller – handles requests from view and translates them to request to model.
   Relationships – Model cooperates with Controller because it can update

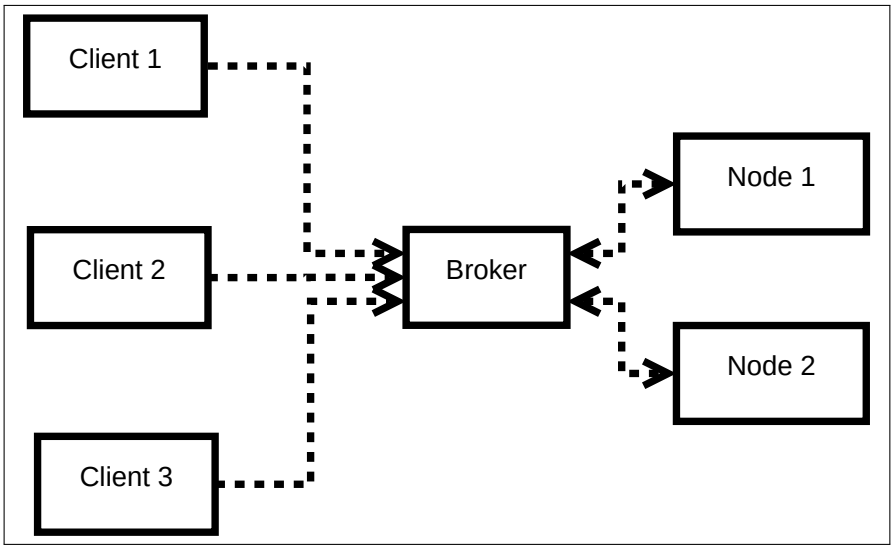

Figure 3.9: Example usage of Broker pattern

registered Controllers. It also cooperates with View because Model can notify View about changes of state of Model. View interacts with Controller because View invokes Controller. It can also retrieve data from Model.Controller interacts with both Controller and Model (see figure 3.10)

6. Presentation Abstraction Controller
   Overview– the purpose of the pattern is similar to the purpose of MVC pattern. This pattern represents different approach to the problem of interaction of a user with an application.
   Elements
   Agent– each agent consist of three elements: Presentation which is meant to provide information to a user, Abstraction manipulating data of the system and Controller handling communication between Presentation and Abstraction. Agents are further divided into agents representing particular functionalities and agents coordinating the system.
   Layer of Abstraction– an application consists of several Layers of Abstraction. Each layer contains Agent on the same level of abstraction. The top layer contains only one agent which coordinates the whole application while the "leaves agents" in the lowest layer represent particular functionalities. A layer invokes other layer on behalf of an agent.
   Relationships– Each Agent can communicate directly with other agents in the same layer of abstraction. Layers of Abstraction communicate only with layers bellow (see figure 3.11).

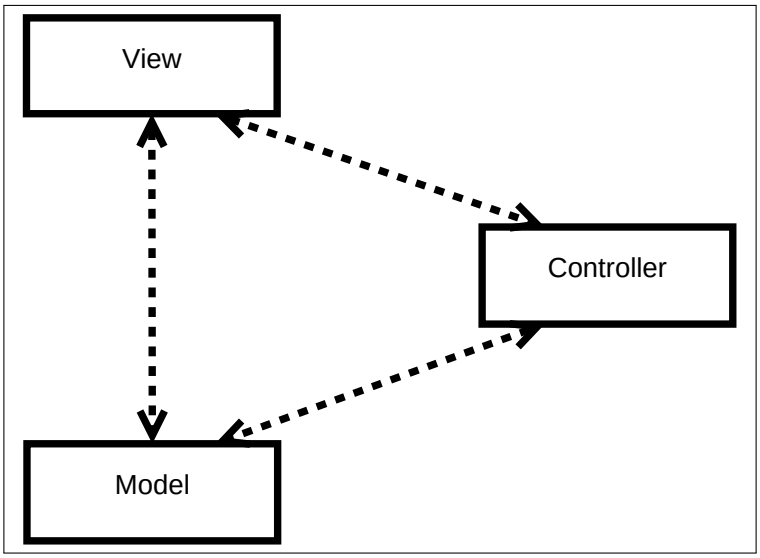

Figure 3.10: Example usage of MVC pattern

7. Microkernel
   Overview – This architectural pattern provides a solution for problems that need systems characterised by high adaptivity.
   Elements
   Microkernel – It is called also Message Bus or just Kernel. Kernel receives messages through external API from clients, forwards them to specific service and returns the result to calling client.
   Service – It encapsulates business logic. Can invoke other services (acts as a client)
   Client – Sends request to Kernel.
   Relationships – a service does not know location of other services and cannot invoke other service directly. A service can invoke only Kernel. Kernel knows location of all the services and can invoke them(see figure 3.12).

8. Reflection
   Overview – This very unusual pattern enables rapid evolution of system. This evolution is ensured by a "self – aware" meta–layer
   Elements
   Meta Layer – consists of Meta Objects
   Meta Object – contains information about structure of the system and its behavior including knowledge about operations that should be invoked if an element from Base Layer invokes an action.
   Base Layer – contains elements of the system.
   Element – each element calls Meta Layer in order to execute an action.

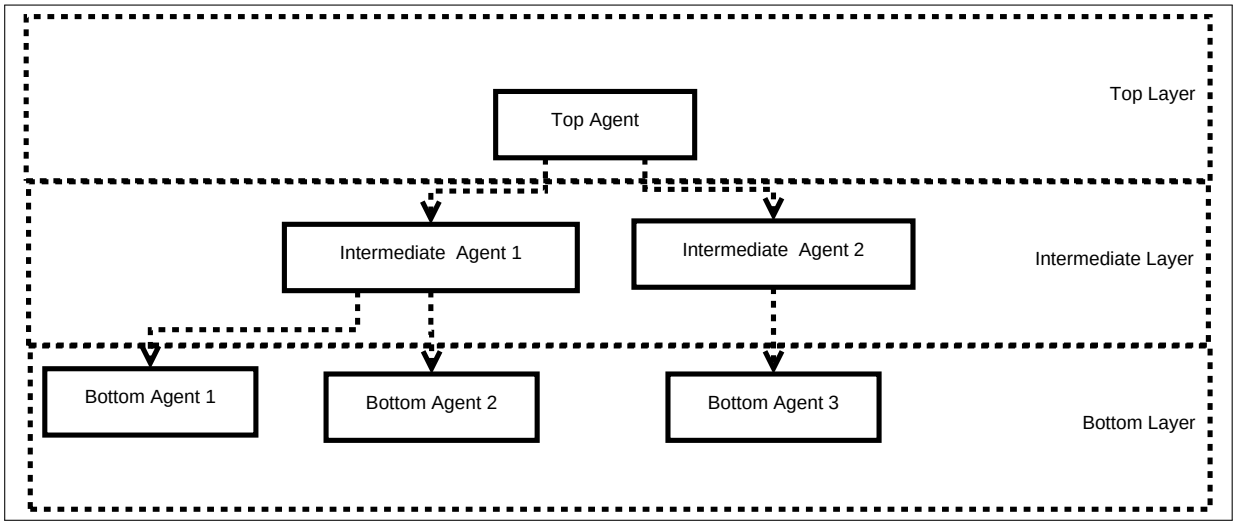

Figure 3.11: Example usage of PAC pattern

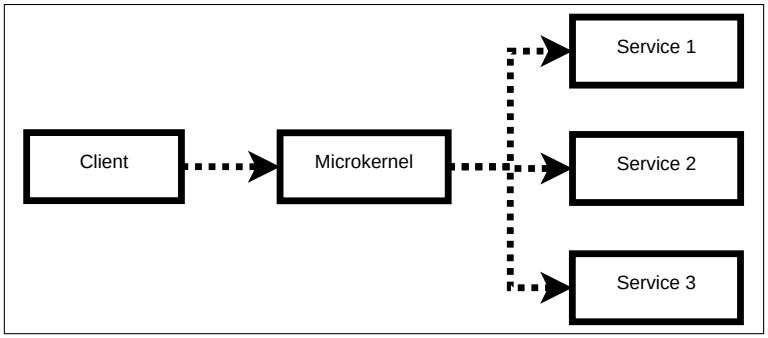

Figure 3.12: Example usage of Microkernel pattern

Meta Object Protocol – enables modification of Meta Layer
Relationships – Meta Object Protocol is a part of Meta Layer and can perform actions only on Meta Objects. Meta Objects can invoke other Meta Objects and Elements from Base Layer but they cannot invoke Meta Object Protocol. Elements of the System can invoke other elements of the system or Meta Objects from Meta Layer (see figure 3.13).

9. Interceptor
Overview– the pattern is meant for systems that evolve rapidly. The Build–in mechanism allows leaving core functionality unchanged and implementing only extensions.
Elements
Client– invokes business functionality
Interceptor Manager– intercepts invocation from the client and starts reg-

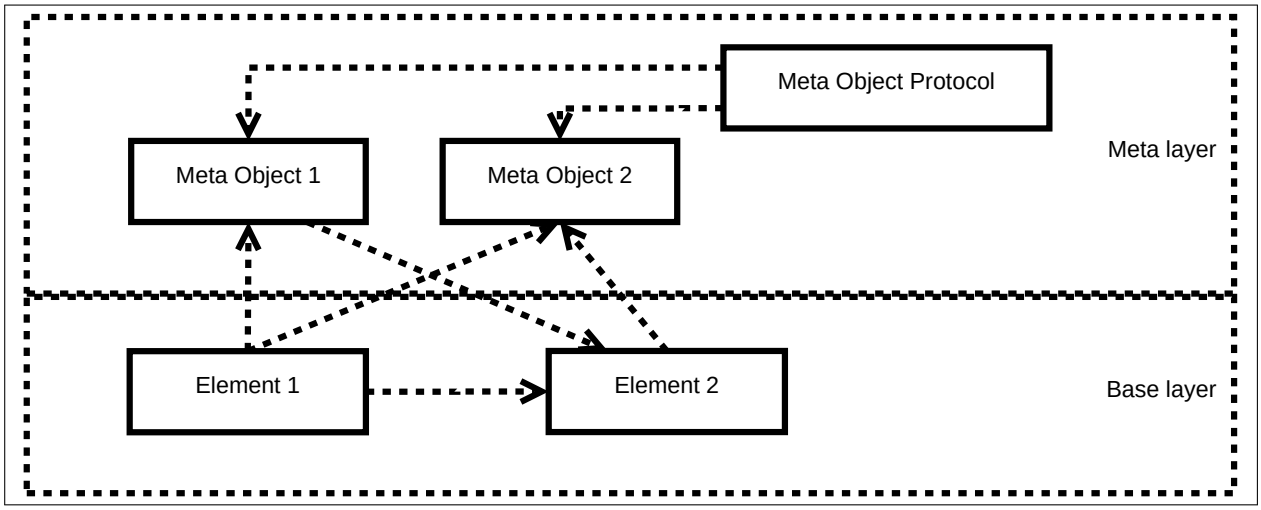

Figure 3.13: Example usage of Reflection pattern

istered interceptors
Invoked Component– contains business logic
Preaction Interceptor– this interceptor is invoked before "Invoked Component" is invoked
Postaction Interceptor– this interceptor is invoked just after action invoked by Client is accomplished but before Client gets answer
Relationships– Client invokes Interceptor Manager, which invokes preaction interceptors. Then, the manager executes an action on the component and invokes postaction interceptors when the action is finished (see figure 3.14).

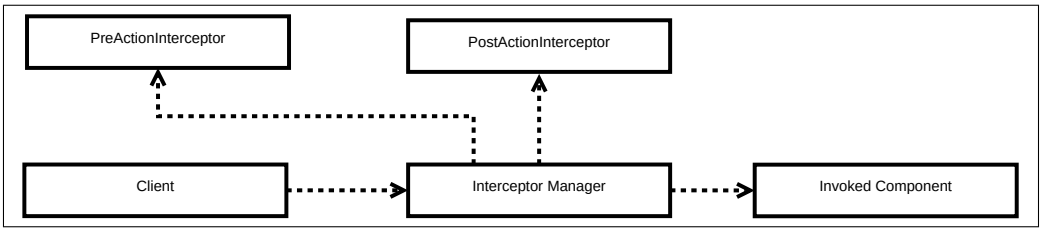

Figure 3.14: Example usage of Interceptor pattern

10. Half–sync/half–async
Overview– supports control over synchronous and asynchronous operations in concurrent systems without reducing performance of the system.
Elements
Synchronous Service Layer– contains high–level elements executing syn-

chronous logic of the system
Synchronous Service– represents logic of the system executed in a synchronous manner
Asynchronous Service Layer– contain low–level elements executing asynchronous logic of the system
Asynchronous Service– represents logic of the system executed in an asynchronous manner
Queueing Layer– contains element mediating between Synchronous and Asynchronous layers
Queue– a mediating element from Queueing Layer
External Event Source– generates event processed by services from Asynchronous Layer
Relationships– - Services from each layer can cooperate with other services from the same layer. They can also invoke Queue that serves as a message store. External Data Source can invoke only services from Asynchronous Layer (see figure 3.15).

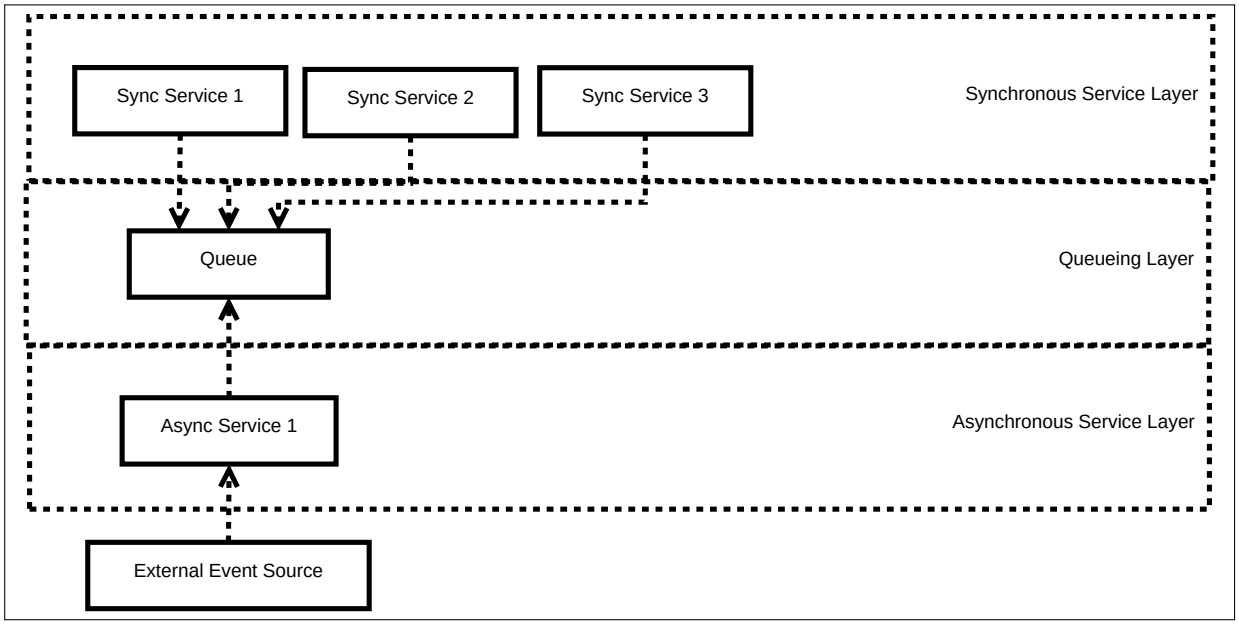

Figure 3.15: Example usage of Half–Sync Half–Async pattern

11. Shared Repository
Overview– provides solution for problem of passing large amount of data between elements of a system. The pattern reduces additional overhead and performance drop.
Elements
Repository – is a core element of the pattern. It is meant to store, maintain

data and provide them to clients
Client – uses repository to store data in it. Client also obtains data from the repository.
Relationships – all the client elements are aware of the repository and can perform operations on it. The repository knows nothing about the clients and cannot invoke them (see figure 3.16).

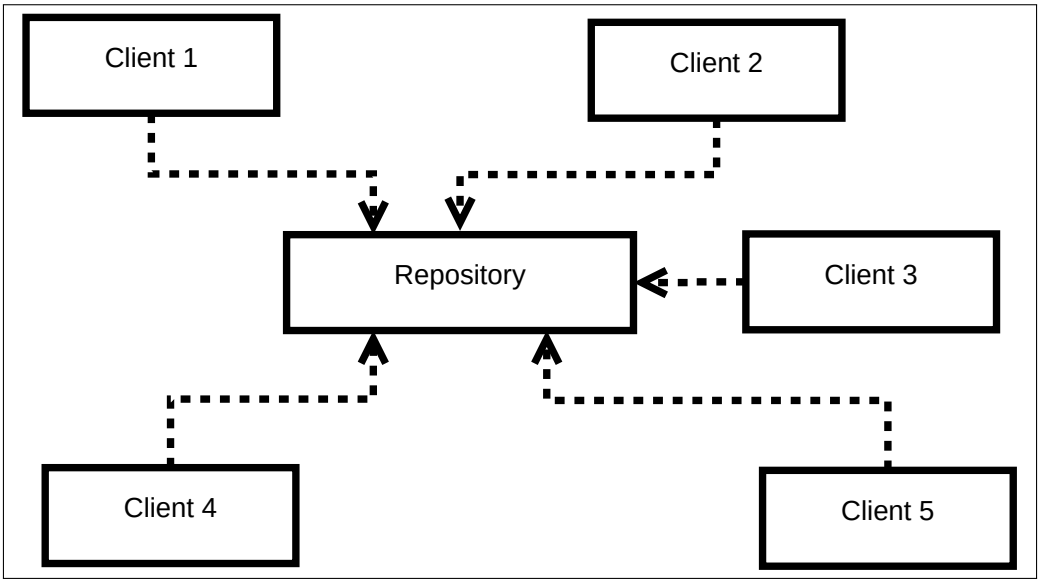

Figure 3.16: Example usage of Shared Repository pattern

12. Messaging
Overview – supports low coupling between elements of the system. This pattern is also characterised by high maintainability and lower performance.
Elements
Message Bus – gets requests from services and invoke requested services.
Service – presents business logic of the system.
Relationships – Services are not aware of the location of other services.Only Message Bus is known to services. Message Bus is aware of all the registered services (see figure 3.17).

13. Client Server
Overview – - the pattern is dedicated for distributed architectures. The pattern clearly separates frontend of the application from its logic.
Elements
Client– it is a frontend of the application. This element allows a user to

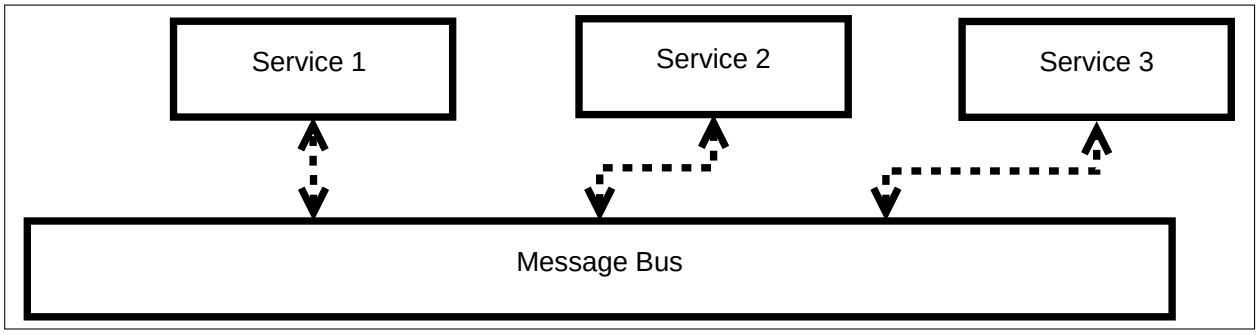

Figure 3.17: Example usage of Messaging pattern

perform operations on the system
Server– provides logic of the system.
Relationships – - a user invoking operation on the clients invokes servers indirectly. The invocation is transparent. Clients can invoke servers but servers cannot invoke clients (see figure 3.18).

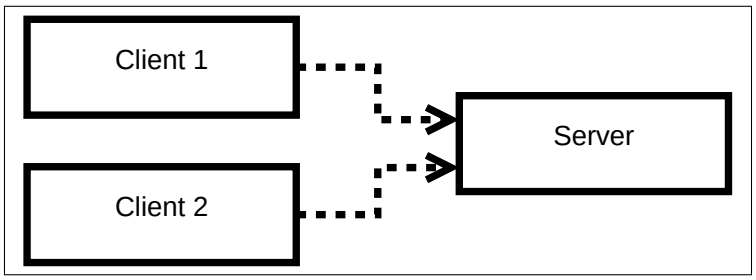

Figure 3.18: Example usage of Client Server pattern

14. Explicit invocation
Overview– the pattern determinates way of communication between elements of the system in distributed environment. This pattern is in fact a variation of Client-Server with one assumption. The assumption says that connections between Client and Server are known during design time.
Elements
Client– is a frontend of an application.
Client Logic– represents logic of the client.
Client Broker– wraps connection to a supplier
Supplier– application containing logic
Supplier Logic– logic of the application
Supplier Broker– wraps connection to a client

Relationships – Client is a container for logic and Client Broker. Logic of the client can invoke the client broker but broker cannot invoke the logic. Supplier is a server proving requires operations. A supplier contains Supplier Logic and Supplier Broker. In opposite to Client, Broker invokes Logic and Logic cannot invoke Broker. The connection between Client and Supplier is bidirectional because an asynchronous connection is assumed. The connection can be also unidirectional, what removes connection from Supplier to Client (see figure 3.19).

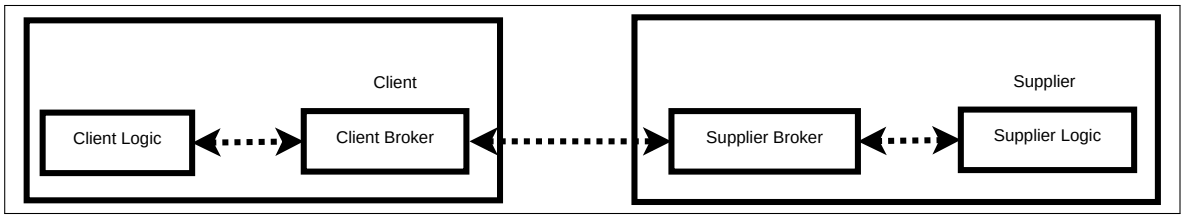

Figure 3.19: Example usage of Explicit Invocation pattern

15. Peer —to —Peer
    Overview– the whole pattern consists of only one type of elements: peer. The pattern creates a network build with peers. The number of peers is not specified because peers can join and leave the network dynamically
    Elements
    Peer – the whole network consists of only one type of element–Peer. Peer is a server and a client in the same time. This means that each "peer" requests and consumes services. To start up each peer has to find at least one other peer to create a network. This problem can be solved by introducing few dedicated public peers that are known to other peers.
    Relationships – there is no limitations in connections between peers (see figure 3.20).

16. C2
    Overview – this patter is another possible solution for system interacting with users.
    Elements
    Component – contains logic of the system
    Connector – forwards messages from one Component to another. Connections between Components and Connectors are asynchronous. Connectors have to be aware of all the Components "above" and "below" them.
    Relationships–All requests are asynchronous so the components have to know only components above in hierarchy to return the result. All requests

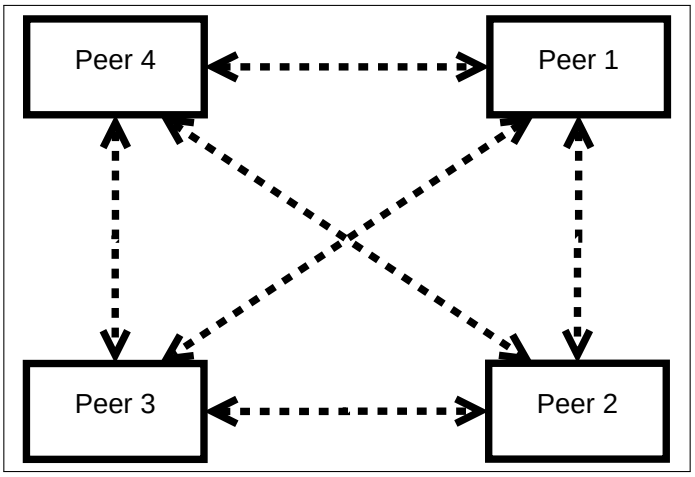

Figure 3.20: Example usage of Peer–to–Peer pattern

to lower components are forwarded by connecting layers and components do not care which exactly sub–agent will execute it. Connectors must know components in order to return result of asynchronous operations.

17. Active Repository
    Overview– this pattern is a variation of Shared Repository that provides a notification mechanism to Repository. Client elements do not have to invoke Repository to check if there is something meant for them. Instead, Repository can notify all registered Clients
    Elements
    Repository – is a core element of the pattern. It is meant to store, maintain data and provide it to clients.
    Client – uses repository to store data in it and to obtain data from it.
    Relationships – all the client elements are aware of the repository and can perform operations on it. Since Repository can notify Clients, all the connections between Clients and Repository are bidirectional

18. Remote Procedure Call
    Overview – The pattern assumes that all the remote procedures should be treated, as they were local. User should not know that a procedure is executed somewhere in the network. The procedures are mainly provider–dependent and can change over the years, therefore the description of the elements and relationships bellow refer to just an example implementation of the pattern
    Elements
    Client– is a frontend of the application. This element allows user to perform

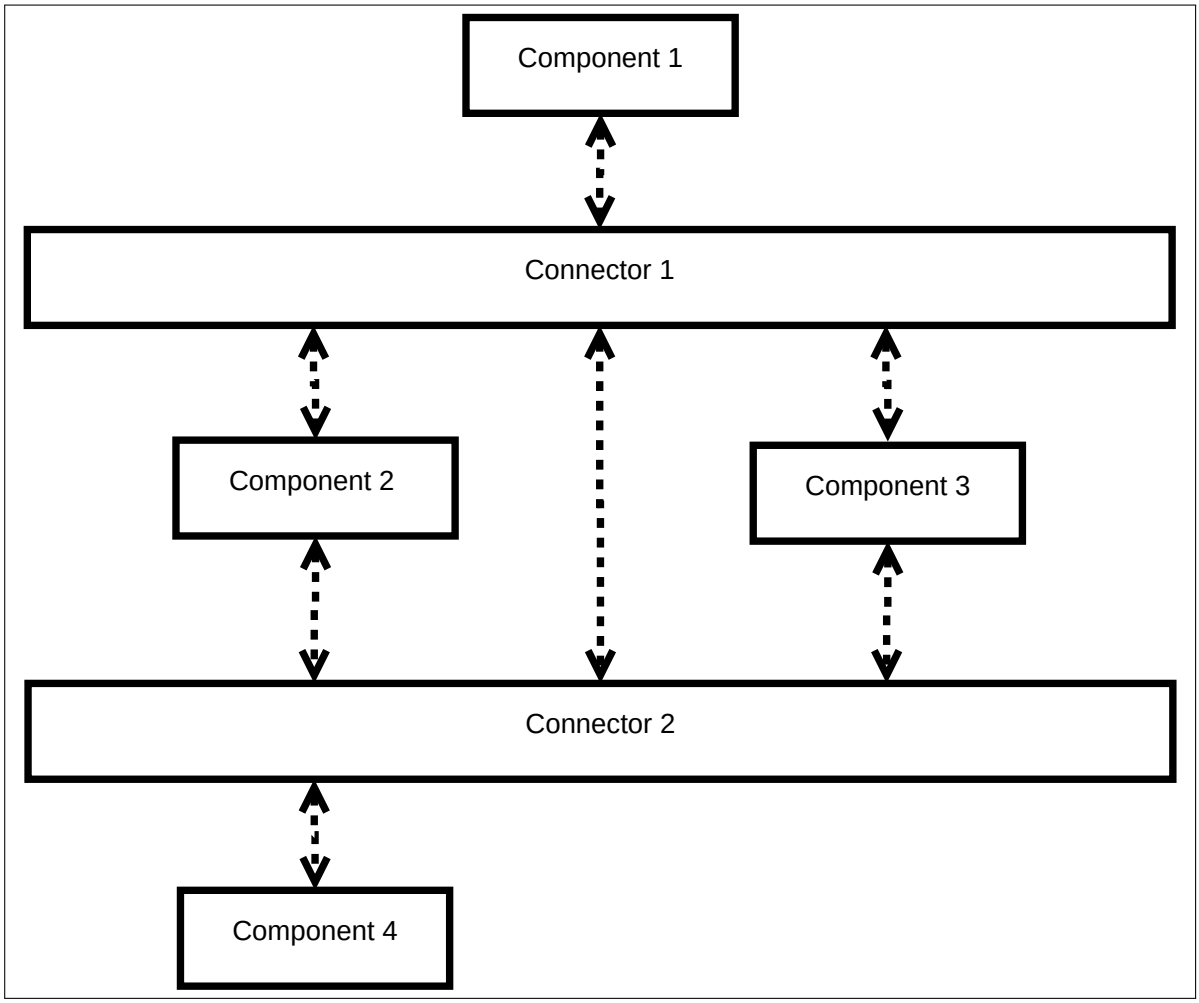

Figure 3.21: Example usage of C2 pattern

operation on the system. Client encapsulates own logic and a server proxy.
Server– provides logic of the system.
Logic– - logic of a client
Proxy– wraps functionality of the server and allows logic of a client to invoke functionality of the server as a functionality of own local object.
Relationships – a user invoking operation on the client invokes servers without knowing it. Clients invoke servers, but the servers cannot invoke clients. Client contains its logic and proxy of a server. The logic can invoke proxy. Proxy can invoke server (see figure 3.23). Other invocations are not possible.

19. Implicit Invocation

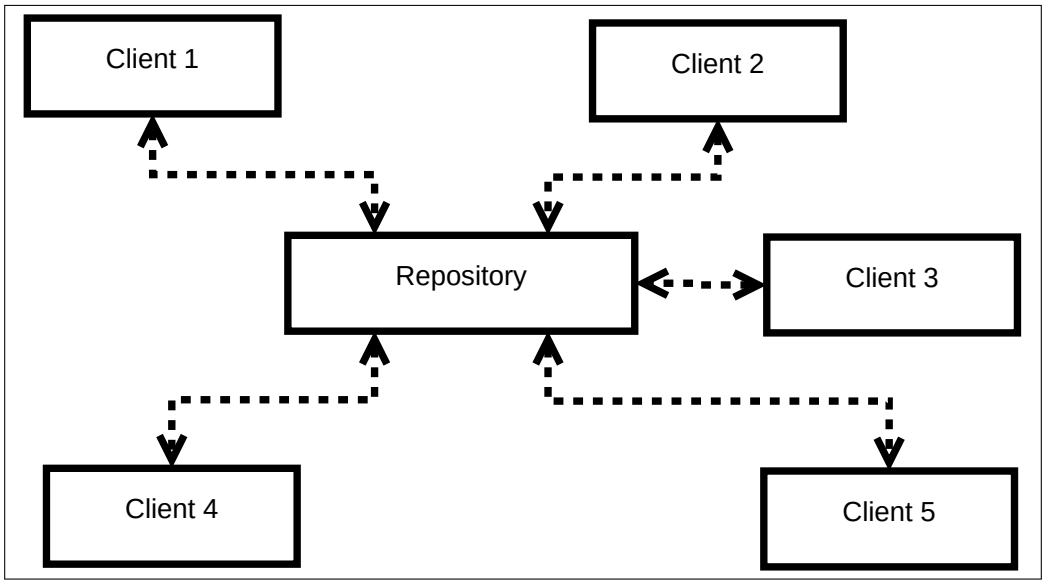

Figure 3.22: Example usage of Active Repository pattern

Overview– the pattern is useful especially in distributed environment where parts of the system do not know where remaining parts are localized or other elements can become a part of the system via dynamic connection.
Elements
Client – is a frontend of the application
Implicit Mechanism – knows where the servers are and it is able to invoke them.
Server – provides functionality of the system
Relationships– client invokes Implicit Mechanism that is meant to invoke server or servers. The Implicit Mechanism maintains also dynamically connecting servers (see figure 3.24).

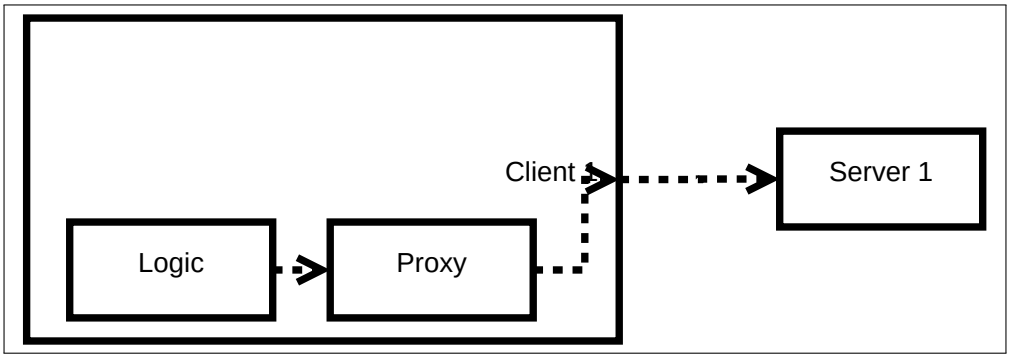

Figure 3.23: Example usage of Remote Procedure Call pattern

Figure 3.24: Example usage of Implicit Invocation pattern

3.2.2 Methods of patterns categorisation

Literature presents several different methods of pattern categorisations [8][38] which are meant to underline specific aspects of patterns like their structure, properties or applications. The reason why particular aspects are underlined may also vary in addition, may include for instance justification for architecture selection e.g. highly adaptive architectures adaptation. Source literature for architectural pattern section provides several different ways of pattern categorisation. Buschmann [21] presents following approaches:

1. Domain –This categorisation groups all patterns that are often used in a specific domain. An example domain can be for instance: scientific application. Most of those scientific applications need a lot of disc space to share common data and a high performance algorithm to do calculations. In many cases, algorithms are distributed over many computers that have to work together. This set of requirements and constrains can impose a set of most adequate patterns.

2. Partition –Groups all the patterns that present a system as a stack of tiers. For instance, there can be a business tier and an integration tier. Integration tier contains all patterns that enable easy integration and adaptation. Business tier groups patterns that improve maintainability and extendability.

3. Intent –This category groups patterns in very narrow and high–specialized groups. Each group consists of patterns that solve the same problem or help to reduce it. For instance Intent may enclose a group of patterns that solve a problem of system distribution or a group that collects only patterns that enables easy exchange of data.

Presented categorisations of patterns start from very general idea of a domain and go deeper through grouping of pattern having structure presented as a stack of tiers. Finally, patterns having the same application are put together.

The second source [38] provides only one categorisation, namely categorisation by domain of application. This approach is in fact a clarification of the domain

category presented by F. Bushamnn. Author distinguishes seven main domains and allocates patterns to each of them. If allocation fails then pattern is treated as 'Other'. The domains are following:

1. Embedded Systems –Systems dedicated for special machines. The machines may have also a dedicated hardware what may set additional limitations.
2. Dataflow and Production –This category enclose systems designed for process monitoring and logistic.
3. Information and Enterprise Systems –Enterprise systems that have to manage and present data.
4. Web–based Systems –This category includes all the systems that are designed for web interaction.
5. Case and Related Developer Tools –This category contains all the applications that help developers.
6. Games –The category contains patterns applied in computer games.
7. Scientific Approaches –Include all the patterns that do support calculations for scientific purpose.

However the descriptions are not to detailed, the names of domains explain little bit more. An interesting fact is that there is no category that in opposite to "Others" could group solutions that do not match to only one category. Those patterns could be assigned to category "MIXED". The categorisation by domains assumes that each pattern can belong to only one domain. An example of mix between domains can be a mix of Web based and Enterprise system because such systems are not unusual.

The last literature source presents different perception of pattern categorisation. Author presents categorisation by *View*, which in fact is a mixture of concept of Intent presented by F.Bushman[21] and an approach based on structure of patterns. P. Avgeriou and U. Zdun distinguish following categories [8]:

1. Layered View – In this approach architecture is presented as a complex entity that can be decomposed into interacting parts. A special emphasis is laid on description of particular parts and their mutual interaction. The view also requires strong decoupling of parts of pattern and takes into account attributes of the pattern like scalability and modifiability.
2. Data Flow View – The next view gathers data flow related patterns. The patterns are composed from elements performing transformations, carrying data streams and connections between them. Essential attributes are modifiability and integrity.

3. Data–Centered View – presents system as a data store. The store contains one or more independent components to store data and one or more components that request data. Description of patterns has to provide information about a method of data storage, access, update and distribution as well as a method of communication with and among data storing elements. The patterns have also to point out whether they are active or passive. The description of a pattern belonging to Data-Centered View includes its scalability, modifiability, reusability and integrability.

4. Adaptation View – presents a system from adaptivity point of view. System must have two main parts: static and dynamic. The static part is a core of the system. Dynamic parts can be added or removed from the system in runtime. Patterns introduced in this category have to enable architecture to evolve and keep communication between particular elements when a system evolves.

5. Language Extension View – divides system into two parts. The first part is native for environment in opposite to the second, which is not. Patterns underline how non–native parts of the system are integrated and translated into environment.

6. User Interaction View – underlines perspective of users. Patterns that belong to this category have presentation that is strongly separated from the rest of the system. Those patterns take special care about data, application logic and their associations with presentation, therefore an impact on usability, modifiability and reusability is also considered.

7. Component Interaction View – underlines communication between particular components of a system and presents the system itself as a set of independent components. The communication technique can be synchronous as well as asynchronous. The important fact is that whichever communication type is chosen it has to be indirect. Important aspects are modifiability and integrability.

8. Distribution View – lays a special emphasis on physical localisation and relations of components over network. A special emphasis is laid on decoupling of distributed components. Important aspects are performance, modifiability, location and transparency.

Categorisation by View presented by P. Avgeriou and U. Zdun provides method covering not only elements of architecture and connection between those elements but also other properties like usability of application are considered This way of categorisation of patterns will be further used as a selected to categorisation.

3.2.3 Allocation of patterns to categories

Filter 4: Assignment of architectural patterns to selected category
This subsection presents allocation of patterns identified in subsection 3.1.4 to views described in the previous subsection. Since eighteen out of nineteen identified patterns exist in the document that provides selected approach to categorization (see second source [8] in the Source of Patterns subsection), the categorisation is almost automatic. The table 3.2 presents results of allocation.

View Name	Pattern Name
Layered View	Layers
	Indirection Layer
Data Flow View	Batch Sequential
	Pipes and Filters
Data–Centred View	Shared Repository
	Active Repository
	Blackboard
Adaptation View	Microkernel
	Reflection
	Interceptor
Language Extension View	Interpreter
	Virtual Machine
	Rule –Based System
User Interaction View	Model –View –Controller
	Presentation –Abstraction –Controller
	C2
Component Interaction View	Explicit Invocation
	Implicit Invocation
	Client –Server
	Peer –to –Peesr
	Publisher –Subscriber
Distribution View	Broker
	Remote Procedure Call
	Message Queuing

Table 3.2: Architectural Patterns – View organisation. Adopted from [8]

The table 3.2 contains also names of pattern marked grey. Those patterns are assigned to particular categories ("views") but they were rejected during Pattern Identification subsection. This table clearly shows that Language Extension View is discarded due to lack of assigned patterns. The pattern that is not present in the table 3.2 but exists in the Pattern Summary table 3.1 is Half sync / half

async. This missing pattern will be further considered as a Component Interaction Pattern because the pattern describes ways of communication between elements of the system. It separates synchronous operations from asynchronous. Existence of only one non–assigned pattern shows that selected categorisation method fits well to list of identified patterns.

3.2.4 Selection of representatives

Filter 5:Selection of representatives for categories
This subsection provides list of representatives of patterns assigned to categories in previous subsection as well as motivation behind selection of those representatives.This criterion of selection is structure (elements and relations among them) of the pattern including purpose and behavior of elements of the pattern. The more generic structure a pattern has, the more likely it is to become a representative of the category. The structure is generic when it is similar to structure of other patterns in the same category and has the lowest complexity. A category may have more than one representative when structure of considered patterns is very different. All the patterns rejected during previous steps are not taken and will not be taken into consideration during pattern selection. The selection of representatives shown below provides a list of candidates (patterns from category), motivation behind each selection and selected representative(s). Summary of representatives allocated to particular categories is provided in table 3.3

Layered View
Candidates: Layers
Selection: This category is composed of only one pattern so this pattern automatically becomes the representative.
Representative: Layers

Data Flow View
Candidates: Pipes–and–Filters
Selection: This category is composed of only one pattern so this pattern automatically becomes the representative.
Representative: Pipes–and–Filters

Data Centred View
*Candidates:*Shared Repository, Active Repository, Blackboard
Selection: Consideration of all of the candidates in terms of criterion for selection of representatives results in conclusion that Blackboard pattern is not a good representative because this pattern introduce a moderator, which does not exists in remaining two patterns. However, amount of elements does not decide about pattern rejection, but a behavior of this particular element does. Moderator introduces additional data access mechanism that does not exist in remain

two patterns. Shared and Active Repositories are different in only one, but very important detail. Active Repository provides notification service what makes it more specialized then Shared Repository, consequently Shared Repository is more general, therefore it automatically becomes the representative of this view.
Representative: Shared Repository

Adaptation View
Candidates: Microkernel, Reflection, Interceptor
Selection: Microkernel is a pattern that strongly influences architecture of system. Everything has to be organized around the kernel of the system. Communication through this element is the only way to get to other elements of the pattern. Interceptor has also a main part –Interceptor Manager, which serves as a main connection to remaining parts of the system. Additionally, Interceptor provides functionality allowing invocation of other elements before the target element is reached. This makes the pattern less abstract than Microkernel. The last candidate -Reflection is a very specific pattern. This pattern has the structure of the system build inside. The pattern is aware of the elements and relations between them. There is no way to compare structure of Reflection to Microkernel. This category has two representatives –Microkernel and Reflection.
Representative: Microkernel, Reflection

Language Extension View
Candidates: none
Selection: The category does not have any pattern assigned so consequently there is no candidates and no representatives.
Representative: none

User Interaction View
Candidates: Model–View–Controller, Presentation–Abstraction–Controller, C2
Selection: User Interaction View consists of three patterns. The first in row is Model-View-Controller. The pattern was mentioned before as an example of categorisation of an architectural pattern as a design pattern. The second is Presentation Abstraction Controller (PAC). Names of both patterns underline similar approach to the problem of user interaction. Elements of PAC pattern can be mapped to some degree to elements of MVC but in case PAC those elements are not the main elements. The main elements of PAC pattern are agents containing presentation, abstraction and controller parts. Those parts could be theoretically replaced by MVC parts, this makes the pattern sort of a MVC agent network, therefore, MVC pattern is more general than PAC. The last pattern in this view is C2 pattern. C2 is not as similar as PAC and MVC. C2 defines components that serve as elements in MVC or PAC, but in opposite to MVC or PAC they do not have specialized function. Even if an artificial division of C2 elements into Model / View / Controller components is made, there remain connectors that do

exist neither in MVC nor in PAC. The connectors specify more precisely way of communication between elements. Summarizing, MVC pattern is the most general pattern within the category and becomes representative of User Interaction View.
Representative: Model–View–Controller

Component Interaction View
Candidates: Explicit Invocation, Implicit Invocation, Client–Server, Peer–to –Peer, Half–Sync / Half–Async
Selection: Pattern descriptions presented in section 3.2.1 underlines similarities between Explicit invocation, Implicit Invocation and Client–Server. Client–Server defines division of a system into two parts. Explicit and Implicit Invocations also make this division but they also introduce new elements determining communication details between Client–Server parts. They are just variants of Client–Server pattern, therefore Client-Server is more general in terms of previously stated criteria. Peer–to–Peer pattern describes interaction between elements that are clients and servers in the same time. In fact, this pattern crates a network of modified Client-Server entities. The problem appears when a comparison between Client-Server and Half–sync/half–async has to be made. The problem is that those two patterns describe different model of connection between components. Responsibilities of elements those two patterns are incomparable. Parts of Half–Sync / Half–async pattern in opposite to Client–Server take care about two different aspects of communication, therefore those two patterns cannot compared to each other. The Component Interaction view has two representatives.
Representative: Half–Sync / Half–Async, Client–Server

Distribution View
Candidates: Broker,Remote Procedure Call, Message Queueing
Selection: Distribution View contains three architectural patterns. The first in row is Broker. The description of Broker pattern underlines similarities between this pattern and previous category. The structure and functionality of this pattern matches description of Client –Server pattern. In fact, Broker is just a more complex version of Client–Server; therefore, this pattern will not be compared with patterns from Distribution View and shall be moved to Component Interaction View. Since this pattern is a more complex version of Client–Server, Broker is not a representative of Component Interaction View. Remote Procedure Call does not have fixed structure. It is rather an idea how to keep connections between different parts of distributed systems masked. The patterns are incomparable; consequently, this view has also two representatives–both are different in terms of their structure and functionality.
Representative: Remote Procedure Call, Message Queueing

View Name	Pattern Name	Representative
Layered View	Layers	Layers
Data Flow View	Pipe and Filters	Pipe and Filters
Data –Centered View	Shared Repository	Shared Repository
	Active Repository	
	Blackboard	
Adaptation View	Microkernel	Microkernel
	Interceptor	
	Reflection	Reflection
User Interaction View	Model –View –Controller	Model –View –Controller
	Presentation –Abstraction –Controller	
	C2	
Component Interaction View	Half –Sync / Half –Async	Half –Sync / Half –Async
	Explicit Invocation Implicit Invocation Client Server Pear –to –Pear	Client Server
Distribution View	Broker	
	Remote Procedure Call	Remote Procedure call
	Message Queuing	Message Queuing

Table 3.3: Architectural Patterns – categories with their representatives

3.3 Mutual Interaction

The Problem of pattern interaction is an implication of pattern organisation and complexity of systems. This problem cannot be neglected during pattern selection. The problem derives from the fact that combination of patterns affects system and have an impact on non-functional properties. A way in which patterns interact has to be carefully considered. It may strongly increase quality of a system, but it also may lead to a disaster. This section presents patterns that are applied together in real systems. The section is organized as follows:

1. Definition of Pattern Language – the subsection provides basic definition of pattern language, including criteria required to be fulfilled by a pattern language

2. Pattern Language in real systems – presents shortly results of studies about number of architectural patterns in systems

3. Popularity of architectural patterns in real systems – presents occurrences of particular patterns in real systems.

4. Representatives of categories in real systems – this section presents application of filter removing patterns that do not exist in pairs with other architectural patterns. The guidelines elaborated in this thesis shall be useful. This also means that the guidelines should be as much applicable as possible. What in turn means that the migrated pattern should be popular - often used. Not often used patterns reduce usability of the guidelines because they cannot be applied frequently.

3.3.1 Definition of Pattern Language

Robert S. Hanmer and Kristin F. Kocan present [37] broad studies on mutual interaction of patterns. They provide following definition of pattern language: *Pattern languages are collections of patterns that can be used to build something larger than any individual pattern can be used to build.* They present two criteria which define *complete pattern language*:

1. *"Morphologically complete (i.e., complete enough that an entire solution can be seen from only the patterns present in the language)"*

2. *"Functionally complete (i.e., the language contains patterns that resolve all the new forces introduced by the use of the languages patterns). "*

According to Robert S. Hanmer and Kristin F. Kocan [8]definition of pattern language should contain following parts:

1. Abstract - provides high-level understanding of the pattern language. This part is usually a few sentences long.

2. Map - The map is a non-cyclic graph presenting patterns as nodes and relations between them as arcs. The arcs are directional. Arrowhead point the pattern that is influenced by the pattern on the opposite side of the arc. Each pattern presented on the map should contribute to the patterns that are in relation with this pattern.

3. Description - the description presents order of application of all the patterns creating the pattern language.

4. Patterns

   (a) Name - the name of the pattern

   (b) Problem - the problem that needs to be solved

   (c) Context - in fact, this is a cause of the problem

(d) Forces - presents factors that should be taken into consideration while applying the pattern.

(e) Solution - the pattern is a solution

(f) Result Context - this context presents results of application of the solution. What is gained or introduced.

3.3.2 Pattern language in real systems

An additional study in the domain of pattern languages was conducted by Harrison B. Neil and Paris Avgeriou [38].During the study, documentation of real systems was analysed [16]. One of the outcomes of the studies was amount of identified patterns in the documentation. Result of analysis is presented below in table 3.4.

Number of patterns found	**Number of system**
1	10
2	22
3	9
4	4
5	0
6	1
7	0
8	1

Table 3.4: Identified amount of patterns. Adopted from [38]

Analysis of documentations stated that in ten systems only one architectural pattern was found. This number may be over the top, because it is possible that the reviewers of the documentations did not recognise all the patterns or just did not know the applied patterns. Moreover, if reviewers were not sure whether a pattern is an architectural pattern, the pattern was rejected as well. Nevertheless, 78% of systems were built using more than one architectural pattern.

3.3.3 Popularity of architectural patterns in real systems

The next outcome of the investigation is popularity of particular patterns expressed as a number of occurrences of an architectural pattern in documentations. The summary of pattern "popularity" presents table 3.5

Pattern name	Occurrences
Layers	18
Shared Repository	13
Pipes and Filters	10
Client–Server	10
Broker	9
Model– View– Controller	7
Presentation–Abstraction–Control	7
Explicit Invocation	4
Plug–in	4
Blackboard	4
Microkernel	3
Peer to Peer	3
C2	3
Publish–Subscriber	3
State Transition	3
Interpreter	2
Half Sync, Half Async	2
Active Repository	2
Interceptor	2
Remote Procedure Call	1
Implicit Invocation	1

Table 3.5: Popularity of particular patterns. Adopted from [38]

There is a strong correlation between table 3.5 and table 3.1, namely, most popular patterns from table 3.5, like Layer or Pipe And Filters exist in all source documents. Less popular patterns like *Plug–in* or *State Transmission* did not pass the first selection or do not exist there at all. Paris Avgeriou and Uwe Zdun [8]] present one more very interesting outcome, a table 3.5presenting which exactly patterns interact with others.

Pattern name	Occurrences
Layers – Broker	6
Layers – Shared Repository	3
Pipes Filters – Blackboard	3
Client Server – Presentation Abstraction Control	3
Layers – Presentation Abstraction Control	3

Continued on Next Page...

Pattern name	Occurrences
Layers – Model View Controller	3
Broker – Client-Server	2
Shared Repository – Presentation Abstraction Control	2
Layers – Microkernel	2
Shared Repository – Model View Controller	2
Client Server – Peer to Peer	2
Shared Repository – Peer to Peer	2
Shared Repository – C2	2
Peer to Peer – C2	2
Layers – Interpreter	2
Layers – Client Server	2
Pipes and Filters – Client Server	2
Pipes and Filters – Shared Repository	2
Client Server – Blackboard	2
Broker – Shared Repository	2
Broker – Half Sync/Half Async	2
Shared Repository – Half Sync/Half Async	2
Client Server – Half Sync/Half Async	2

Table 3.6: Popularity of pairs of architectural patterns. Adopted from [38]

However, there are many pairs of patterns, not all are included. Reviewers excluded sixty seven pairs that appeared only once.

3.3.4 Representatives of categories in real systems

Filter 6: Removal of rarely interacting patterns

Table presenting categories 3.3 with their representatives provides ten representative patterns as follows: Layers, Pipes and Filters, Shared Repository, Microkernel, Reflection, Model View Controller, Half Sync/Half Async, Client Server, Remote Procedure Call and Message Queueing. Additionally. Table 3.6, which is presenting popularity of particular patterns presents thirteen different patterns combined into pairs. The patters are as follows: Layers, Broker, Shared Repository, Pipes and Filters, Blackboard, Client Server, Presentation Abstraction Controller, Model View Controller, Microkernel, Peer to Peer, C2, Interpreter, Half Sync/Half Async. In order to minimize the list of potential patterns

to patterns that can cooperate with other architectural patterns, the lists should be intersected. The outcome of intersection removes a few patterns from list of representatives and provides the final list of patterns that are considered as potential candidates for migration. The list will be further reconsidered in terms of feasibility of migration toward SOA. The list of final patterns presents following entities:

1. Layers
2. Pipes and Filters
3. Shared Repository
4. Microkernel
5. Model–View–Controller
6. Half–Sync / Half–Async
7. Client–Server

In the result of intersection, three patterns were rejected:

1. Reflection
2. Remote Procedure Call
3. Message Queueing

3.4 Pattern Selection

The patterns of the final set are the patterns from intersection of list of pattern representatives and patterns from list of mutually interacting patterns. This section confronts final list of patterns with available literature in order to remove patterns already migrated and provide information about migration feasibility.

1. Prefeasibility Study – presents identified works related to final list of patterns in the context of SOA
2. Pattern for migration – provides final pattern for migration with justification of the choice.

3.4.1 Prefeasibility Study

Filter 6: Prefeasibility study
Selection of a pattern proper for migration is not limited to pointing which pattern will be migrated but it also should consider whether migration has chances to be successful. The final group of patterns should be reconsidered from the feasibility point of view. The consideration includes review of available literature. The review is meant to search for attempts of migration and their results.

Layers

The pattern already exists in context of SOA. This pattern does not have clearly distinguished subsystems. It is rather focused on separation of different part of the system based on their level of abstraction or functionality. Selection of this pattern for migration is not promising because the pattern is rather a concept. The elements of the pattern are very abstract and rather descriptive.

Pipes–and–Filters

The pattern in the context of SOA is considered as a good pattern for Enterprise Application Integration (EAI) [70]]. The description provided by authors presents Filters as separate services that are incorporated into orchestration process (see 4.3.2). The description compares two methods of business integration. In fact is a kind of migration but not from Pipe and Filters architecture. The final system utilizes Pipes and Filters as an element of the final architecture. Summarizing: The pattern is already a part of SOA. It is used in Enterprise Application Integration.

Shared Repository

No information about Shared Repository and its integration with / migration to SOA was found.

Microkernel

No information about Microkernel and its integration with / migration to SOA was found.

Model–View–Controller

Model View Controller in relation to SOA exists in literature in several contexts. The pattern does not appear as a part of SOA in opposite to for instance Pipes And Filters. Elements of SOA are parts of Model View Controller pattern, namely, services replace controllers in for instance the Art Examination Management System [73].The system was not a result of migration, but it was created

from the scratch with usage of Java Struts framework. The framework provides build-in support for MVC pattern. Other approach for MVC-SOA integration proposes changes in MVC pattern. More explicitly, it adapts MMVC pattern to SOA architecture [67] and focuses on data transfer part of the desktop application. The solution proposes introduction of an additional broker component and publisher-subscriber design pattern for communication. The other approach suggests collaboration of services with model part of MVC [28]. Model simply binds and invokes previously found in registry services. The services become external resources like database. Studies on MVC and SOA include also integration from GUI point of view [78].

Client–Server

Client–Server is an architectural pattern that due to its nature is used in distributed systems therefore a special interest from SOA related researches might be expected. The studies exist and in fact are in advances phase. They do not consider only possibilities of migration but provide also a prototype framework automating migration from Client-Server legacy application into SOA application [36]. The framework solves problem of migration of VB.Net application by wrapping their code into web services but other languages are also considered as a feature work. Other studies consider performance of SOA applications based on Client–Server pattern and compare it to SOA applications based on Peer–to–Peer pattern [26].

Half–Sync / Half–Async

No information about Half–Sync / Half–Async and its integration with / migration to SOA was found.

3.4.2 Pattern for migration

Filter 7: Final selection
Considering information gathered during prefeasibility study, three patterns have increased probability of successful migration. The increased feasibility is a result of already conducted studies related to those patterns and SOA. However, the studies present the patterns as worthy further consideration, only one can be chosen. Client–Serverpattern is rejected due to advances phase of studies related to SOA and migration, while Pipe–and–Filters appears in the context of Enterprise Application Integration only. Additionally, Pipes–and–Filters is considered only in one study. Model View Controller is the pattern that is analysed from many perspectives but does not exist in the context of migration. The pattern is also widely supported by many frameworks which are not restricted to one language but rather are a bias of existing languages including Java (Spring, Struts)

C#(ASP.Net), PHP (Zend) or Ruby on Rails (MVC Framework), therefore this pattern is a good candidate for migration.

3.5 Summary

The chapter presented several architectural patterns and problem with definition of an architectural pattern. Next, a list of architectural patterns was elaborated. The patterns from the list were further investigated. Different ways of pattern categorisation were presented. One of the categories was chosen and previously found pattern were assigned to this category. Additionally each category got at least one representative pattern. The architectural patterns were also considered in the context of their mutual interaction. Information obtained during selection of representative of particular categories as well as information about mutual interaction allowed selecting a final set of patterns that was considered as a set of candidate patterns for migration. A prefeasibility study was conducted and Model View Controller was selected the pattern for migration.

Chapter 4

Service Oriented Architecture

This chapter presents an overview of Service Oriented Architecture (SOA) along with benefits and challenges and the target architecture that bases on SOA architectural patterns. Section is organised in following manner:

1. Definition of Service Oriented Architecture – presents definition of SOA from both business and architectural point of view.
2. Elements of SOA – architecture is composed of a set of elements. This section briefly presents elements of SOA. This knowledge is required to understand architectural point of view.
3. SOA – business point of view – the point of view presents SOA as a business process. The perspective underlines activities and services. The activities are required to create SOA, while services are abstract entities providing business values.
4. SOA – architectural point of view – this point of view presents three different approaches to realize Service Oriented Architecture.
5. SOA Vendors – one of the basic properties of SOA says that SOA is vendor independent. Since there is no unified definition describing SOA, each vendor presents own vision. Those visions are not necessarily similar to visions of other vendors. This section presents overview of SOA delivered by a few leading vendors and an overview of tooling provided by the leading vendor.
6. Architectural Patterns in SOA – this section presents identified SOA patterns. The patterns support different elements of SOA. Based on the patterns the target architecture is created.
7. Benefits of SOA – the section presents briefly possible motivation for choosing SOA. The benefits present different perspectives on benefits of SOA. The section contains also several SOA related challenges.
8. SOA Manifesto – presents SOA Manifesto
9. Summary – a section summarizing the chapter.

4.1 Definition of Service Oriented Architecture

There are many definitions of SOA in literature. The definitions differ in terms of scope and details. In general, the definitions underline business or technical aspects of SA. An example of business-focused definition is provided by Thomas Erl [31]:

"SOA establishes an architectural model that aims to enhance the efficiency, agility, and productivity of an enterprise by positioning services as the primary means through which solution logic is represented in support of the realisation of strategic goals associated with service–oriented computing."

Other business focussed definition says that SOA is [13]: *"A set of business, process, organizational, governance, and technical methods to reduce or eliminate frustrations with IT and to quantifiably measure the business value of IT while creating an agile business environment for competitive advantage."*

While business view on SOA is important and cannot be neglected, the thesis focuses on architectural aspect of Service Oriented Architecture. *"Architecture"* or more precisely on *"Software Architecture"*. Software Architecture [39]:*"is not a project plan that describes activities and staffing for designing the architecture or developing the product. Instead it is a structural plan that describes the elements of the system, how they fit together and how they work together to fulfil the system's requirements. It is used as a blueprint during the development process, and it is also used to negotiate system requirements(...)"*

The definition of SOA from architectural point of view must align to the definition of Software architecture presented above. A strict definition of Service Oriented Architecture, which includes aspects of the Soft- ware Architecture is presented by Krafzig [46]:

A Service–Oriented Architecture (SOA) is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus. A service consists of a contract, one or more interfaces, and an implementation.

The definition of SOA presented by Krafzig is the definition used in this thesis. Each time the term SOA is recalled, it refers to this definition unless it is stated explicitly that other definition is applied. The definition as presented by Krafzig includes structural aspects of an architecture using several other terms. The explanation those terms is presented in the section below.

4.2 Elements of SOA

This section describes briefly elements of service oriented architecture.

1. Main elements – This section focuses on elements presented in the defition provided by Krafzig[47].
2. Other elements – The second section presents other elements of SOA that are not mentioned in the definition but they were identified during researches.
3. Types of Services – While the first section describes the idea behind services, this section presents briefly types of services.
4. Structure of a service – this section presents structure of a service

4.2.1 Main elements

Service — derives from business context of an organisation. Service can be described as [59] *"an asset that corresponds to real–world business activities or recognisable business functions."* Service does not only provide functionality, it also consumes or consumes and provides functionality. A service should be discoverable (available through discovery mechanism like for instance service registry) and bound in run-time, however static bound is also possible . A service can be also be characterized by some properties [42]] such as performance, capacity, and reliability. Service is a complex entity and consists of Service Contract, Interface and Implementation [46].
Service repository [46][31] —contains descriptions of services and provides all information required to access them. The information includes localization, provider, technical constrains, terms of usage, fees, service subscription and user registration if the services are public. Repositories can be stored in databases to store contract of descriptions for each service version. The database may store contracts, description of each version of a service and some administrative data. The term repository is also interchanged in literature with registry. In fact both terms are very similar and differ only in one aspect. Repository is a term used during design time, while registry is associated with runtime usage [31]

Enterprise Service bus — Service Bus is also referred as Enterprise Service Bus (ESB). The bus is a middleware concept that enables interaction between different applications [25].(Enterprise) Service Bus can be implemented as for instance Enterprise Application Integration middleware [25] [13], brokering technology [13] or a platform specific component like WebShare application Server [13] [60]. In fact, ESB is a communication mechanism that performs following activities [59]:

1. Accepts information requests from service requesters.
2. Consults a metadata registry to determine how to assemble the requested information using known data services.
3. Retrieves the data from one or more data service providers.
4. Applies data transform rules using a data transformation service that removes duplicate data and converts the data that have been collected into the format specified by the service contract.
5. Returns the results to the service requester.

Frontend —corresponds to an upper layer of a traditional application like Web page or a rich client[46]. Frontends are used to initialize a process and retrieve a result. The initiation does not have to be invoked by end user. It can be a long living process which invokes specific events in some circumstances like periodical events.

4.2.2 Other elements

The definition of SOA[46] does not mention about one more element of Service Oriented Architecture, namely Service Inventory. This element is not obligatory but it simplifies maintenance of large amount of services.

Service Inventory [31] [46] – inventory is an independent element of the architecture and independently manages collection of services. The collections contain services important from point of view of the company.

4.2.3 Types of services

Services can be divided into following groups:

1. Application Frontend – in general it does not have to be a service in terms of previously stated definition - it is rather a client of an application but it can also be other services like services from other company or a batch service executing tasks periodically [46] .
2. Basic Service[46] [31] – represents basic and stateless element of a domain. A basic service can be Data Centric Service or Logic Centric Services.
   (a) Data Centric Services – are responsible for maintaining only one main data entity. The maintenance does not mean that a service executes only CRUD operations - a service has to care also about locking mechanism and transactional management. To fulfill all his responsibilities, the service has to maintain database connection.

(b) Logic Centric Services– correspond to algorithms and business rules libraries in monolithic systems.

The definitions clearly show that there are two types of basic services but in practice, it is very hard to create pure Data Centric Service or pure Logic Centric Service.

3. Intermediary Service [46] [31] – are stateless mediators that serve as bridges between services. They may be categorized into Gatways, Adapters, Facades and functionality adding services. Because they connect very specific services, reusability of Intermediary services is much lower than reusability of Basic Services.

4. Public Enterprise Service [46] – Enterprise level public services are very rare entities. They make available their functionality outside a company in opposite to other types of services. They have to be loosely coupled and well described on business level. Those issues raise security problems, which may occur and generate more problems than services within a company.

5. Process Centric Service [31] – encapsulates and executes a part of process-logic of a company. This type of service is the most complex. Process Centric Service is less reusable and does not contribute too much to functional infrastructure of SOA due to its complexity and narrowed usage [46]. Nevertheless, its process control is essential for the organization [34]. In spite of its complex structure and lowered reusability, Process Centric Service bring some benefits, which are especially important in applications that have to guarantee simultaneous access for many users. Those benefits are:

   (a) Process encapsulation [46] [31][59] [13]– Process Centric Service hides and encapsulates hermetically structure of the process. This supports designing of architecture may be expressed as a set of Process Centric Services implementing processes. Encapsulation improves testability of the system and allows different teams to work on different parts of the system simultaneously.

   (b) Load balancing simplification [46]– clear separation of processes supports parallel executing what is very important for applications that are characterised by short response time.

   (c) Support for multi-channel application [46] –multi-channel applications are [86] applications enabling users to access them by using a variety of devices like personal computer, mobile phones or any other device that can connect to the web. Each of these devices is called 'a channel'. In fact, multi-channel application is a set of interfaces for the same

logic. Application of SOA to multi-channel application simplifies reuse of logic of application by encapsulation of process logic in Process Centric Services.

(d) Separation of process logic [46] [31] [59] –process logic separation is a precondition for efficient process management.

Architects that consider usage of Process Centric Services have to carefully separate process from process control activities. Clear separation simplifies maintenance and reorganization, which due to possible changes of business process may appear more frequent.

From all of the service types mentioned before, only Basic Service is a mandatory part of SOA. All the remaining Services are meant to help during development and maintenance [46].

The relationship between SOA and its elements is described in figure 4.1

4.2.4 Structure of a service

A service is not a simple entity. It is composed of following elements [46]:

Service contract – may be described as a 'technical interface' which provides technical constrains, requirements and any other information that is meant to be public [31]. A contract may consist also from a set of technical documents describing properties of the service. Those documents can be definition of service described with Web Service Definition Language (WSDL), XML Schema definition or Web Service (WS) policy. The purpose [46] of documents like WSDL definition is to provide technology independence by describing further level of abstraction. The most general goal of a contract is to set boundaries of the service [59].

Implementation — is a realisation of the contract. The realization is provided by business logic and appropriate data.

Interface —exposes functionality of service to clients connected via network. An interface consists of two parts. The first is a description, which is also a part of a service contract. The second is a physical representation in the shape of a service stub that is a part of service's client.

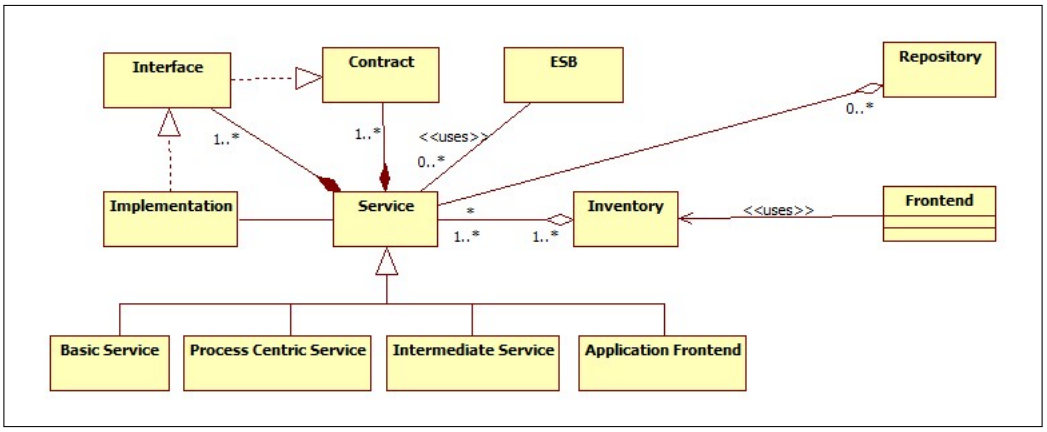

Figure 4.1: Relationship between SOA elements

4.3 SOA – business point of view

One of the goals of SOA is to align the business. It is very important to establish business context and define boundaries of the system by a management team[6].There is no SOA without proper analysis. The need of proper business analysis in order to separate services and their relationships was notified and resulted in organized methodologies like Service Oriented Modeling Analysis (SOMA) [6],[11] and supporting Service Oriented Modeling Framework (SOMF) [11]. There are also other approaches like presented by T.Erl [19] or N. Bieberstein[13]. This section presents SOA from business perspective.

1. Properties of Services – presents basic properties of services.
2. Activities – describes activities executed during designing SOA.

4.3.1 Properties of Services

Companies that decided to introduce Service Oriented Architecture to their systems choose also specific approach for analysis and design. Analysis selects the most suitable candidates for services. Each candidate is characterized by following properties [13]:reuse, encapsulation, coupling, cohesion and granularity.

Reuse – services should be designed keeping high reuse in mind. The importance of service reuse is explained in details in benefits section (4.7), which fully motivate **high reuse** of a service.

Encapsulation [13] – hides implementation information from customers of a service by making services accessible only via their interfaces. There are consumers who consume the service. The consumption shall not depend on underling

implementation details, because the consumer is interested only in the result of invoked operations. There are also providers who have to design services as technically independent as possible. This independency allows consuming service's operations by any potential consumer. Each service should have possible **highest encapsulation**.

Coupling – ddescribes dependencies between services. This term in case of services is much broader then coupling between components. Coupling refers also to [13]:

1. localisation –the localization of a service should not determinate behavior and accessibility of a service. To achieve low localization coupling, services should be accessible via configurable component like Gateway [59], Enterprise Service Bus (ESB)[59] or Repository [31]

2. technology – the implementation technology of the service cannot determine possible consumers. To eliminate technology coupling open -standards for both transport and messaging like [13]SOAP,XML, TCP/IP are recommended.

There is also coupling between [13], message formats, languages, network protocols and interface contracts but they are consequences and special cases of previously described cases. The aim for architects and designers is to keep services **loosely coupled**.

Cohesion – cohesion refers to relations between operations offered by a service. If operations relate to each other by sharing the same data and/or references to object then the cohesion is high. A low cohesion means that operations are not related to each other or the types of arguments of operations are primitives. The higher cohesion a service has, the more flexible and reusable is. Consequently, a good service is characterised by a **high cohesion**.

Granularity – granularity describes size of a service. The size is considered in two dimensions [13].The firs perspective is the size that can be fine or coarse. The second is a purpose, which can be business or technical. Literature points size of a service as factor determining granularity but it also does not provide any measure which can conclude that a service characterized by a particular size is fine/coarse grained. In practise [13], coarse-grained services are mainly invokers, while fine grained are invoked . Other definition says that a coarse-grained service is a service that provides significant business capability [49]. This definition also does not define how to measure this signification.

4.3.2 Activities

Activities are building blocks of methodologies [13].Each activity derives from a method principle, constrains or good practices applied in this particular approach. This section discusses a set of common activities required to create Service Oriented Architecture [13]. The activities are presented in sequence so the first description matches the first activity that should be executed and so on.

Service Identification – The goal is to identify services by using supporting techniques like Input Analysis, Top–down Analysis, Bottom -up Synthesis or Taxonomy Building. However all those techniques can be used separately and guarantee useful output, the best result is achieved by mixing the approaches. This step creates a baseline for further analysis and modelling. The supporting techniques can be defined as follows:

1. Input Analysis [13] requires documentation or business models of the systems, therefore it is applicable to legacy system analysis. According to [13], Input Analysis may be driven answers to following questions:
   (a) *Are the business drivers and goals for the SOA project articulated and quantifiable in terms of key business performance indicators?*
   (b) *Have the business processes to be realized been named and described at a level of detail that is sufficient for architectural decision–making at the IT level?*
   (c) *Have the existing and future non–functional requirements been documented with any unresolved pain points?*
2. Top–identifies high-level business concerns in order to capture and model requirements [13]. There is no *"best solution"* for this type of analysis, however it can be implemented using even very basic elicitation techniques like interview. The main benefit of this technique is that it does not go into details on object level. Object level analysis may lead to impossible to manage "net" of classes and their relationship.
3. Taxonomy Building uses semantic of word used within the company taking into consideration both technical and business aspects[13].As the result, a dictionary is elaborated. The dictionary describes all the processes and dependencies between them and this is a good starting point for service identification. Creation of services depends on terms in the elaborated dictionary and relations between those terms. The most frequent used words identify entities, rules, processes or others elements of the system that should be encapsulated in a service. Successful realization of taxonomy building establishes also unified vocabulary that improves communication between teams.

4. Bottom –Up Synthesis During this type of service identification, the system is decomposed into single processes and components. Decomposition has to be performed with potential reuse of components in mind. The next step is synthesis that is executed after the system is broken down. As the result of synthesis, new services are created by analysis of processes and previously separated components. This technique is applicable for existing systems.

The best result is gained when both Top –Down and Bottom –Up approaches are combined together [13].

Service Categorisation – assigns previously identified services to predefined categories. According to [13] services can be assigned to following categories:

1. role in business model: process, business function, business rule validation, application utility, infrastructure
2. type of consumer: customer, partner internal service
3. implementation strategy: external, composed, adapted

Categorisation helps to better understand and describe identified services. Understanding of properties of those services is essential during next steps.

Service Specification – defines technical interface of a service by specification of its contract. The contract shall contain information about a set of operations and their parameters, preconditions, postconditions and invocation syntax of the operation [13] This syntax describes types and structure of exchanged information. The contract has to keep high cohesion (see 4.3.1 for more information about cohesion).

Service Orchestration – Service Orchestration is an activity connecting technology and business. Service Orchestration creates meaningful processes from services. This is a key activity that decides whether the system will success [82].The value of Services Orchestration does not derive only from the value of its concept but also from the way of process performance. Orchestrationbrings a significant value for the organisation and it is does not require deep technical knowledge because it is all about modelling [82] without any single line of code. However Orchestration is widely known, it is often confused with choreography. Choreography is very similar but it operates on different level of abstraction. Orchestration in opposite to choreography is meant to create executable and internal processes which communicate with both internal and external services [66], while choreography defines how multiple parties of a system exchange messages [59]. In other words, choreography specifies orchestration in a global model [12].

Service Realisation – When services are identified and specified, they have to be realized. However, realization of a particular service is the final activity; it has to be avoided when it is possible. This may sound strange or irrational, but it is fully justified. Realisation should be avoided because of reusability. Reusability prohibits development of a new service where an existing service can be reused. A new service is needed only when there is no existing service that can perform required operation. This new service can be build from previously separated reusable components if there is such possibility or it can be implemented from scratch.

4.4 SOA – architectural point of view

Overview of Architecture of SOA can be presented on three levels of complexity. (see figure 4.2).Each level consists of several layers of abstraction [46].

4.4.1 Fundamental SOA

Fundamental SOA is the first and the most abstract level. It distinguishes only Enterprise and Basic layers [46]. Those two layers are sufficient to provide basic SOA functionality. Enterprise layer contains application frontends (clients) (see 4.2.3 for more information about frontends). Frontends provide access to the application itself. Basic layer contains basic services (see 4.2.3 for more information about basic services). The structure built on only two layers may appear to be insufficient, but it is sufficient to provide basic functionality and it is a good starting point for further development of more elaborated and complex structure. Due to clear separation of clients from basic services, the architecture supports maintain activities and, in a long run prevents data redundancy. The redundancy may exist when a company converts existing application into Service Oriented Architecture.

4.4.2 Networked SOA

Fundamental SOA may be sufficient for small projects but it is hard to maintain when a project grows in scope or when an application has to be integrated with another system. *Networked SOA*is an upgraded version of *Fundamental SOA* [46]. The upgrade relies on introduction of one additional layer-Intermediary layer. This additional layer is placed between Basic and Enterprise layer and is meant to provide mechanisms facilitating technical and conceptual integration by introducing Intermediary Services. Those services serve as gateways, facades and adapters. The names of services of Intermediary Layer derive from design patterns and behavior of those services correspond the patterns. The role of particular types of services is following:

1. Adapter – plays a role of message converter. An adapter service receives a message in one format and transforms it to another format. Adapters have very strict role what in turn reduces their potential reuse in both current and other project.

2. Facade – hides underlying complexity and provides unified API which supports usage of underlying basic services. A goo d example is facade utilization to hide operation on distributed databases [46].The unified API serves also as a view, which clarifies usage of a part of the system by hiding unnecessary (from service customer point of view) functionality. This type of service serves as an additional layer of abstraction. This reduces reusability of the service. Consequently, Facade is rather project scope service and should be considered in this way.

3. Gateway –Gateway or Technology Gateway is a kind of an additional interface, which in opposite to a traditional interface (like interface in programming language) serves as an interface between two different technologies. At first glance, it seems that Gateways are unnecessary or redundant especially when the application is designed from scratch. This is not true because SOA by definition assumes usage of different technologies from different vendors. Taking advantage of Gateways removes the need of introduction of old technology to new system while the new system is integrated with legacy system. As a result, Gateway simplifies achieving of technology independence.

4.4.3 Process–Enabled SOA

The last and the most complex SOA structure is Process Enabled SOA [46]. In fact it is just an enriched version of Networked SOA. The difference between those two versions is that Process Enabled SOA has forth –Process –layer. The layer contains only Process Centric Services which are in one respect very special. Process Centric Services are not stateless as the others are. They manage the state in order to handle long living processes or to share the state between services. This is especially important when an application uses asynchronous communication. Utilising of Process Centric Services lighten Application Frontends and brings them to role of User Interface, what increases cohesion of single services.

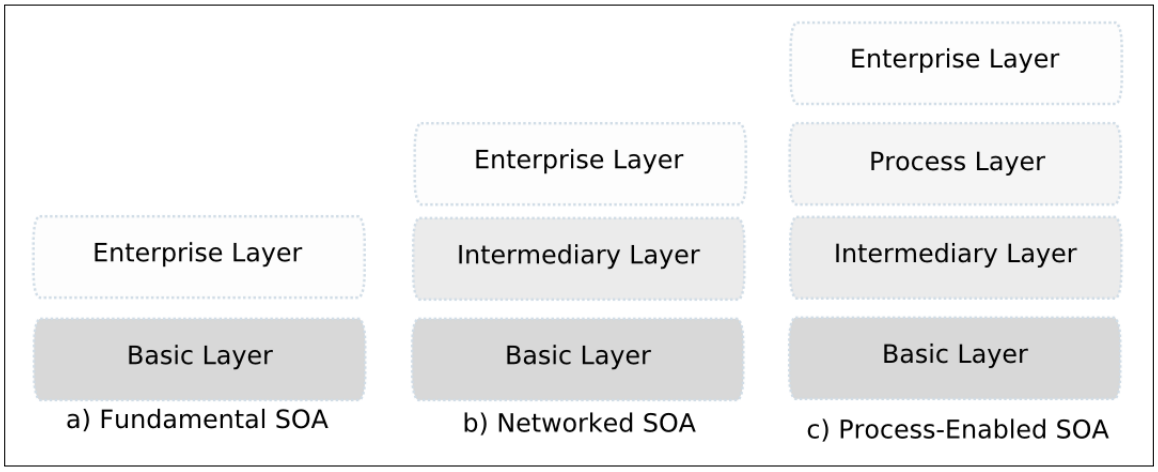

Figure 4.2: Service Oriented Architecture on three levels of abstraction. Adopted from [46]

Despite all the benefits associated with more complex concepts of SOA, it should be kept in mind that an architect should not introduce additional complexity if the system does not require it. The system should be enough complex to support desired objectives and not more.

4.5 SOA Vendors

Introduction of SOA to a company is not an easy task. This difficultly has been noticed by leading software developing companies, what resulted in a number of supporting frameworks and developer's suits. The solutions that have been proposed differ in scope of support but also in understanding of SOA, what (paradoxically) contributes to companies introducing SOA because SOA is vendor independent and the diversity allows an architect to create more complex (and cheaper) solutions. List of SOA vendors is relatively long (see [15]) and is still growing. Some vendors provide sophisticated solutions that allow creating application from concept to detailed plan, while other vendors provide pluggins to IDEs. According to [5][80] IBM is one of the leading vendor in supporting SOA and has the highest market share, therefore, it is worth to outline briefly SOA IBM concept and supporting tools.

4.5.1 IBM –Layers of abstraction

However basic concepts of Service Oriented Architecture like service, ESB or orchestration are very similar to presented in 4.1, the structure of SOA is slightly different. It is said that: *"Any problem in computer science can be solved by*

adding a layer of indirection" and IBM leverages this solution very well. Consideration of a system on different layers of abstraction helps to underline both business and technical aspects. According to IBM, a system should be considered on five layers of abstraction during SOA analysis. Starting from the most abstract enterprise level and ending on very precise object level (see figure 4.3). The layers and their role are following[13]:

Enterprise layer – describes an enterprise on business model level. The model consists of processes, services (as activities) and enterprise data as well as relationship between those elements.

Process layer – Process layer consists of main processes which handle main business concern by composition and decomposition [83]. Composition creates business services from underlying service layer, whereas decomposition decomposes exiting process into services.

Service layer – Service layer contains services that expose technical and business functionality of an application [13]. The services are independent and carry individual tasks. Those services are also building blocks used by Process Layer.

Component layer – On component layer, the system is considered as a set of physical components - building blocks of services. This layer is very important because components that exist in more than one place are identified [13]. In result those components become potential candidates for becoming services or parts of a service.

Object layer – Object layer is the lowest modelled layer during analysis and design activities. However the name refers to objects -instances of classes - the layer contains description of objects understood as classes characterised by operations, attributes and relationship between them [13].

4.5.2 IBM – SOA Foundation Suite

SOA Foundation Suite by IBM is a set of SOA supporting tools. Very important aspects of IBM solution are modularity which enables usage of only a subset of all the tools and scalability that simplifies architecture modernization when a company grows [60].The tools support different aspects of SOA like Service Design, Creation, Governance, Integration, Connectivity and Collaboration as well as Service Security. A short characteristic of the tools supporting different aspects of SOA is following:

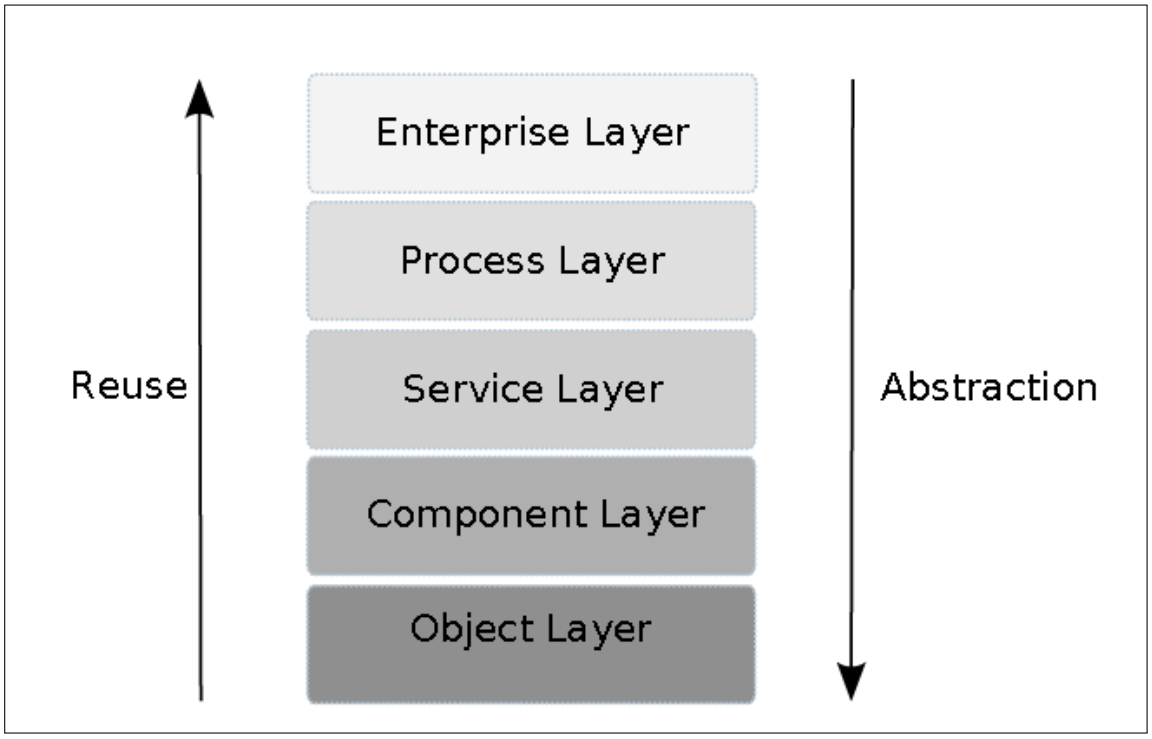

Figure 4.3: SOA layers of abstraction according to IBM. Adpoted from [13]

Service Design IBM offers support of Service Designing activities with two tools. The first is IBM WebShare Business Modeller which simplifies business process modelling based on business requirements. The second is IBM Rational Software Architect (RSA) that enables to model services with usage of UML. An important property of RSA tool is that it can import models of WebShare Business Modeller for further and more detailed modelling with UML.

Service Creation – As the previous aspect, Service Creation is also supported by two tools, but in a different way. The first tool -Rational Application Developer -supports Service Creation with both top-down and bottom-up approaches. The tool provides also a wizard that supports quick creation and publication of Web Services and their clients. The next supporting tool is also a backbone of the whole suite-Application Server. Application Server as other application servers allows deploying and executing services. It is also enriched with WebSphere Process Server, WebSphere ESB, WebSphere Service Registry and Repository, WebSphere Portal, and WebSphere Services Business Fabric [60].

Service Governance – IBM provides two tools supporting Service Governance, namely WebShare Service Repository and Registry (WSRR) hosted on

Application Server and WebShare DataPower SOA Appliance. WSRR serves as a source of information about Services for ESB (see 3.2 for service repository), while WebShare DataPower SOA Appliance is an endpoint which is invoked every time a consumer wants to consume a service, the tool can also enforce policy and routing service request because it has an access to the registry. WebShare DataPower SOA Appliance acts as ESB. In fact it is second out of three (ESB, DataPower, Message Broker) Service Buses provided by IBM.

Service Integration – provides two more tools supporting Service Integration. The first tool is WebShare Integration Developer which simplifies process orchestration with Business Process Execution Language (BPEL). WebShare Process Server (WSPS) is the second tool. It is integrated with Application Server. This tool enables deployment of previously created processes and mediates ESB flow. WSPS enables communication of those processes with WSRR in order to acquire latest data. The communication is also established with WebShare Business Service Fabric, which provides latest service endpoints -dynamic binding.

Service Connectivity – The third type of Enterprise Service Bus is IBM WebShare Message Broker, which is meant to simplify Service communication between different companies and different technologies applied for web service creation.

Service Collaboration – IBM provides a tool simplifying creation of web-based portals enabling a rule-based access to company's resources. This tool is IBM WebSphere Portal. Another supporting tool is IBM WebSphere Portlet Factory facilitating creation of applications for portals.

Service Security – Security is a very important aspect of an application and SOA-applications are not exceptions here. The tool-IBM Tivoli Federated Identity Manager - is introduced to support this aspect of services. The facility provides [60] security infrastructure services, which include secure token services, authorization services, authentication services and directory services for the information technology (IT) infrastructure. In other words, the tools enabling message-based authentication and authorization as well as creation of trust relationships between elements of architecture.

4.6 Architectural Patterns in SOA

A term "pattern" in the context of Service Oriented Architecture appears very often with an adjective "design", but it is not always clear whether "design" refers to a design pattern in terms of previously stated definitions (see section 3.1.2) or rather refers to design activities including design of whole system (architectural

pattern), design of a single component (typical design pattern) or maybe design of a process (process pattern)(see section 3.1.1). The distinction grows in importance when architectural knowledge is taken into consideration. Architectural Knowledge is meant to describe an architecture and rationale behind decision made during architecture designing [9]. The decisions do not refer only to rationale of fine-grained components but also to a "big picture" – system architecture. Architecture requires more detailed studies, due to a large-scale impact on system. This subsection describes SOA architectural patterns in following manner.

1. SOA patterns– presents a problem with SOA architectural patterns and presents results of pattern investigation.

2. The target architecture – presents the target architecture that is a result of combination of patterns found in section 4.6.1

.

4.6.1 SOA patterns

The book *"SOA –Design Patterns"* [32] is the largest and the most complete source of pattern in SOA domain[54]. Patterns presented in the book belong to design patterns, but their descriptions question this adherence. The problem rises from definition of pattern and design pattern used by T.Erl in this book.

According to T.Erl pattern [32] *"provides a proven solution to a common problem individually documented in a consistent format and usually as part of a larger collection"* what corresponds to the definition of pattern presented in section 3.1.1 .

Design pattern is defined as follows [32]: *"Patterns in the IT world that revolve around the design of automated systems are referred to as design patterns."*.This definition can be also applied to other types of patterns. All the pattern described in section 3.1.1 "revolve" around design of automated system, not only design patterns.

SOA patterns are described also in *"Patterns: Service–Oriented Architecture and Web Services"*[30] but they refer to Enterprise Application Integration. Integration is out scope of this thesis.

Selection of architectural patterns

The target architecture will be build on architectural patterns. Usage of the book *"SOA –Design Patterns"* [32] as a source of architectural patterns requires further studies.

Analysis of patterns presented by T. Erl (see [32]) results in division of presented pattern into four categories. The categories are following

1. Architectural Patterns – describe general solution for a whole architecture or a part of architecture. Those patterns refer to structure and communication within the system.

2. Design Patterns – provide solution for a fine-grained problems that has a local impact on the architecture

3. Process patterns – describe patterns for processes that should be applied in order to gain particular outcome.

4. SOA Concepts – describe issues that already are part of Service Oriented Architecture

5. Technical Issues – describe both Design and Architectural patterns which are build-in exiting frameworks and supported by external technologies.

Table 4.1 presents summary of pattern assigned to particular categories.

Pattern Type	**Amount**
Architectural	14
Design	28
Process	12
Technology issue	9
SOA concept	10
Total	73

Table 4.1: Summary of pattern types

Examples of pattern that were not classified as architectural (see table 4.2 for summary):

1. Canonical Expression –assumes up-front analysis in order to standardise naming conventions, which is later applied to service contracts. A good example of Canonical Expression is CRUD.
   Motivation: Canonical Expression was classified as a *Process pattern*, because it defines a process – analysis.

2. Service Encapsulation –defines a service as a containing logic entity. The logic can be encapsulated in a new service as well as become a part of an existing service.
   Motivation: Encapsulation of logic in a service is a basic **concept of SOA**,therefore there is nothing what could be considered as a pattern. See Service Definition 4.2.1

3. Multi–Channel Endpoint –is a Service Inventory Endpoint variation that allows consuming services using different channels. The channel can be for instance a PC or a mobile phone with an access to Internet.
   Motivation: the pattern does not provide anything what may strongly affect architecture. The problem can be also solved by creation of a set of single channel service. This is a matter of a **design**.

4. Version Information – the pattern advices introduction of a service version to service contracts.
   Motivation: The solution is: *"Use version annotation"*. Annotation is an XML element therefore this is a **technical** issue

Pattern Name	Type
Canonical Expression	Process
Service Encapsulation	SOA
Multi–Channel Endpoint	Design
Version information	Others

Table 4.2: Examples of rejected SOA patterns

Description of SOA architectural patterns

This subsection presents briefly description of identified architectural pattern in SOA. Description of each architectural is followed by "Rationale". Rationale motivates selection of the pattern as an architectural. Additionally two patterns were classified as duplicates. Canonical Schema appears to be Schema Centralization and Rules Centralisation is in fact Validation Abstraction.

1. Service Layers –Services delivered by different teams may contain inconsistencies and redundant functionalities. The pattern divides service inventory into logical layers containing services with similar functionality. Each layer contains one specific and abstract concern. The pattern enforces usage of at least two layers-one contains basic service, the other contains composed services.
   Rationale: The pattern corresponds to *Layers* architectural pattern.

2. Canonical Protocol –Service can be provided by different vendors. The vendors can use different languages and communication protocols. However the difference in language is not so important, the differences in communication protocols are. Different communication protocols limit number of potential consumers and introduce the need of protocol bridging. Application of the pattern enforces usage of one communication protocol as a main

mean of communication. Pattern application eliminates bridging services in the system what improves performance and maintainability. In order to increase benefits associated with the pattern, architects should consider usage of widely supported and vendor independent communication protocols like SOAP.

Rationale: Enforcing usage of one specific protocol by every single service in an inventory has a large impact on communication within the system.

3. Canonical Schema –Many services perform operation on the same data that are modeled differently by different vendors or teams. Those differences increase development efforts and make design more complex. Additionally, the transformation between different models introduces performance drop. The pattern standardizes data model for information within the inventory. The motivation behind the pattern is to avoid translation between different definitions of the same data and eliminate data definition redundancy across the system.
   Rationale: Forces services to use predefined schemas. Service cannot define own 'implementation'. The pattern influences every service in the system.

4. Utility Abstraction –Services, even those that do not provide redundant functionality may perform some common operation. Those operations are redundant. The redundancy requires additional effort for testing and maintenance. Utility Abstraction defines one layer containing common utility services. The layer contains mainly services without business concern, therefore in opposite to business-based services, utility services are developed mainly by architects and developers what raises problems related to content of such service. Teams need good communication in order to avoid redundant services.

   Rationale: Enriches structure of Service Layers, what also changes flow of messages in the system.

5. Entity Abstraction –Many business processes uses the same business entities. It is very likely that the functionality is implemented several times in different services. This raises problems with maintenance and testability. Entity Abstraction introduces another layer into layered structure. The layer contains services performing operation on business entities.
   Rationale: See Utility Abstraction rationale.

6. Process Abstraction –Grouping of business process services together with non-business process services makes harder maintenance of both types of the services. It is harder to change processes and maintain functionality

provided by other services.This pattern introduces enterprise wide process services as an additional layer. The ser- vices are mainly stateful and manipulate services from all the underlying layers.

Rationale: See Utility Abstraction rationale.

7. Policy Centralisation–Services may require to process individual polices. Policies may refer to security, transaction requirements and Quality-of-Services. Replication of policies across inventories brings redundancy and problems with synchronization of changes to the policies. Application of the pattern separates policies from services.
   Rationale: Forces services to use predefined WS–Policy files. WS–Policy describes service polices like security.

8. Rules Centralisation –The same business rules are applied in many different services. This may lead to redundancy and problem with maintenance of the changes. Business rules are stored and accessible through dedicated entities like business rules services.
   Rationale: However the pattern simplifies maintainability, the structure of process–centric services become more complex due to a need of additional services invocation in order to validate data against business rules and constrains.

9. Canonical Resources –Modern systems require access to external resources like databases, central directories or transaction management frameworks. The resources are accessed by many subsystems of the system. The access logic becomes a part of the subsystem. Canonical Resources identifies and standardises resources across the company. A term resource refers to databases, activity management systems, state deferral mechanism or other resource like for instance public services. Separation of access to external resources grows in importance when an application is not small, because in small application access to external resources is rather centralised.
   Rationale: The pattern has significant impact on the architecture because each service requiring access to external resources has to invoke those dedicated services.

10. State Repository –Long living processes occupy system resources for a long time. The pattern introduces a state repository that is meant to maintain state of stateful services. Stateful services delegate their state to special services in order to free system resources. The state is not transferred automatically. The transfer occurs when special criteria are met. The criteria can be for instance an extending waiting for a response. This solution occurs to be very useful when an asynchronous communication is applied.
    Rationale: Application of the pattern requires additional effort related to

state transfer. Each stateful service needs an access to state repository and the state has to be converted into maintainable data like XML files.

11. UI Mediator –Business services bind into process many independent services. The execution time of those services may vary and it can take a while to provide result requested by the user. Application of the pattern introduces an additional layer that processes information about progress of invoked operation. The information is further acquired by frontends and presented to users.
    Rationale: See Utility Abstraction rationale.

12. Inventory Endpoint–An inventory may contain several services providing functionality that might be used outside the inventory. Services accessing the inventory from outside should not know structure of the inventory and its functionality. The access also needs to be secured. Inventory Endpoint establishes an entry point for external service consumers. Each customer accessing service repository, access the endpoint and receives information from the endpoint. Capabilities of endpoints can be extended by for instance direct / brokered authentication. A single Inventory can have more than one endpoint.
    Rationale: : Inventory Endpoint enforces communication through gate-like services. Requests coming from service consumers and going back have to go through an endpoint.

Pattern Name
Service Layers
Canonical Protocol
Canonical Schema
Utility Abstraction
Entity Abstraction
Process Abstraction
Policy Centralisation
Canonical Resources
State Repository
UI Mediator
Inventory Endpoint
Rules Centralisation
Validation Abstraction
Schema Centralisation

Table 4.3: Selected SOA architectural patterns. Grey rows contain patterns identified as duplicates

4.6.2 SOA– The target architecture

The target architecture (see figure 4.4) is expressed using identified SOA architectural patterns and relations between them. Motivation behind location of particular patterns the target architecture is following:

1. Schema – is an XML document which describes vocabulary to express business data [82]. The schema is divided into domains. The reason why each layer depends on the schema is that each information in the system has to be described according to the schema, consequently each service includes a part of the schema layer.

2. Policy – provides a model and a syntax to describe and communicate policies [13]. The policies are defined in one place and as the schemas are applied to particular services, therefore each layer has to have an access to the policies.

3. Frontend – contains service clients therefore it is a top layer. Frontends may communicate with UI–Mediator if continuous feedback to user is required. This layer does not exist in patterns presented by T.Erl[32] but legacy systems have some user interface that also have to be migrated.

4. Endpoints Layer – contains entry point for each inventory of basic services. If a service from other inventory or a client application wants to consume a service then the consumption has to be conducted via an endpoint. Endpoints define also possible operations therefore they serve also as Facade.

5. Process Abstraction – consists of process services. Process services depends on basic services.

6. Basic Services consist of services that provided by Entity Abstraction(Entity), Utility Abstraction(Utility), Rules Centralisation(Rules), Canonical Resource(Resources) and State Repository(State). Names in round brackets correspond to names in figure.

4.7 Benefits of SOA

SOA requires many experienced specialists including architects, programmers, requirement engineers and project managers. SOA requires also a lot of time what does not support well projects with a short time-to-market. Nevertheless Service Oriented Architecture is introduced to companies that operate on a variety of business domains including banking [46](Halifax Bank Of Scotland), post

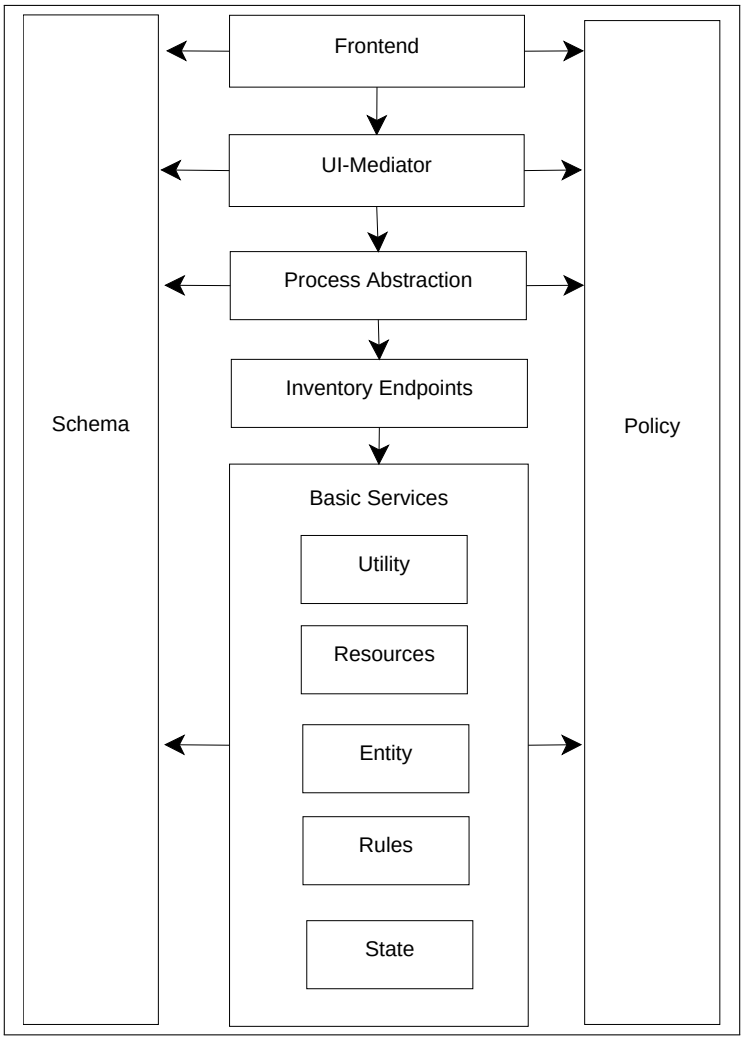

Figure 4.4: SOA target architecture

[46](Deutsche Post AG) or insurance companies [46] (Winterthur). The choice of SOA is justified by a number of benefits. Literature provides a variety of SOA benefits and advantages that include business and personal perspective. On account of a huge list of potential profits, only the most significant and characteristic are presented in this section. For more information about SOA benefits see [46][31][62].

Business perspective

The main benefit of SOA is agility [46][31]. This word and its variants became very popular in last few years. There is a number of publications having "agility in name". We can find Agile Software Development, Agile Architecture, Agile

Project Management or Agile Testing. World is going to be Agile to the limits of technical capabilities, therefore short clarification what is Agile in context of SOA is needed. Taking into account business perspective of SOA, agility can be expressed as a sum of following factors:

1. Reuse [46] [31] [59] – reuse is a kind of very high level goal that people tried to achieve in many ways, including development of Object Orientation or Aspect Programming. High reuse reduces problem of inconsistency in business data. SOA reuse is even more powerful because it is supported by technology independency that (theoretically) eliminates technical barriers. The most important consequence of reusing of a huge amount of code is reduction of risk of failure. The most important consequence of reusing of a huge amount of code is reduction of risk of failure.

2. Independence from technology [46] [31] [59] –Technology dependency is a major barrier for system evolution and development, especially in enterprise-level applications that grow with the enterprise. In result of the grow and technological dependencies, applications become a heterogeneous and composed systems that were separate applications working for specific departments. SOA does not try to convert system to homogenous system. Keeping system independent from a technology allows to introduce new solutions if they bring additional value to the company. This also reduces cost of improvement / development of the system by enabling selection of the cheapest vendor on the market.

3. More efficient development process [46] [31] [59] – SOA in nature aligns business processes. Identification of business processes becomes identification of services and relationships among them. Services are self contained by nature, thereupon it is much easier to test them and integrate into complete processes. Moreover, development teams are smaller because their tasks are limited to a single service or several services that are in a relationship.

4. Cost saving [46] [31] [59]–Cost of a project is one of the most important factors that may cause project to fail or reduce its scope. Each possible saving is a benefit that cannot be snubbed. Saving gained by introduction of SOA is a consequence of previously mentioned advantages. High reuse reduces costs of maintenance and project itself. Technology independence allows choosing the cheapest vendor available on the market. Efficient development process shortens development time and reduces effort required for communication between individual teams.

Personal perspective

Business does not consist only of a set of procedures, goals and company logo. Business is people, people who spend their time trying to achieve company's goal

by performing some predefined procedures, therefore a consideration how SOA driven project affects them is also needed. Technology and big projects are the challenge that have to be faced. Business level benefits can be translated to personal perspective in following manner [46]:

CEO – work of execution officer is supported by a clear strategy what enables short-term planning and reduction of budged for maintenance.

Project Manager – an underlying infrastructure does not play so significant role anymore and Manager can focus on functional aspects of a project that is also supported by shorted iterations / project. The shortage can be achieved by parallel development of decoupled services and reduction of testing.

Architect – Service Oriented Architecture is a kind of challenge that makes work more interesting and helps develop creative thinking. Service driven approach eliminates also monolithicall infrastructure and supports creation of loosely coupled components what increases stability of the system.

Developer – Fact that services are self contained entities reduces dependencies. This simplifies refactoring and testing. It is also easier to define requirements for small components with strict boundaries.

Challenges

Although benefits brought by Service Oriented Architecture seems to be tempting but as there is no smoke without fire, there are no pure benefits without challenges to be faced. Introduction of SOA is a challenge itself, especially when this approach was not used in projects before. Each novelty causes resistance of people, especially when it requires additional overheads as SOA on initial phase. The overhead is associated mainly with transferring existing infrastructure to SOA [46].As a result, company has to reconsider relationship with vendors of standard software and software infrastructure.

However decoupling brings a lot of benefits, it appears as non trivial task that requires experienced requirement engineers and architects. Architects have also to design their systems with service oriented structure in mind. A special attention has to be guaranteed for the most significant part of SOA -reuse. Designing highly reusable components is very challenging task requiring open-minded thinking and of course previously mentioned experience which grows in importance because of lack of clear standardization. Although this deficit is slowly eliminated by SOA vendors who propose own perspectives (see 4.5)

Summarizing, introduction of SOA requires positive attitude of IT staff because it is not an easy task and requires some overhead. It challenges their experience and ability of creative thinking, but in long range it brings a lot of non-trivial benefits which enhance quality of a pro duct and associated processes.

Summarising, introduction of SOA requires positive attitude of IT staff because it is not an easy task and requires some overhead. It challenges their experience and ability of creative thinking, but in long range it brings a lot of non–trivial benefits which enhance quality of a product and associated processes.

4.8 SOA Manifesto

Service Oriented Architecture is not standardised and different vendors may have slightly different understanding of this specific type of architecture, but the main idea remain the same. Diversity of implementation provided by different SOA vendors is an advantage of SOA.

Nevertheless lack of clear explanation what exactly SOA is, leads to many different misconceptions and misunderstanding like presented in [47] [14]. This problem was notified by leading SOA experts and resulted in SOA Manifesto announced in October the 4th 2009 during SOA Symposium in Rotterdam. The manifesto is following [1]:

Service orientation is a paradigm that frames what you do. Service–oriented architecture SOA is a type of architecture that results from applying service orientation.

We have been applying service orientation to help organizations consistently deliver sustainable business value, with increased agility and cost effectiveness, in line with changing business needs.

Through our work we have come to prioritize:

Business value *over technical strategy*
Strategic goals *over project–specific benefits*
Intrinsic interoperability *over custom integration*
Shared services *over specific–purpose implementations*
Flexibility *over optimization*
Evolutionary refinement *over pursuit of initial perfection*

That is, while we value the items on the right, we value the items on the left more.

However, the manifesto tries to clarify the concept of SOA, it has been criticised. The main objection is that the manifesto is ambiguous, consequently some authors like Joe McKendrick [55] try to explain and motivate Manifesto statements. Nevertheless the manifesto is a good omen that brings some clarification

into the domain which according to Martin Fowler [33] is *"semantics–free concept that can join 'components' and 'architecture'."*

4.9 Summary

Service Oriented Architecture is not standardised. There are many vendors proposing their understanding of SOA along with tools support this understanding. Service Oriented Architecture has several benefits that may be presented from personal and business perspective. There are also some challenges that have to be faced.

Definition of Service Architecture revolves around services, frontends, service repositories and service bus.

This section identifies 14 architectural patterns that were extracted from 73 found patterns. Based on identified architectural patterns, the target architecture was elaborated.

Chapter 5

Guidelines

This chapter presents elaborated guidelines in order of their application. The guidelines apply white-box approach (see section 2.1.1 for example white box approach). The presented white-box approach defines process which requires context identification, service identification and identification of the target architecture. While the target architecture is already identified (see figure 4.4). The context analysis and service identification is possible because the documentation and source code is available. Each guideline is meant to introduce one SOA architectural pattern into target architecture. The chapter is organized into following sections:

1. Pattern Languages– the section presents motivation behind chose of pattern languages as a mean of transformation between two architectural. Additionally, the section describes migrated and the target architectures are pattern languages.

2. Guidelines– Guidelines for migration are meant to simplify migration to SOA. The guidelines are implementation of translation between migrated and target pattern language. Order of application of each guideline reflect the description section of the target pattern language. Each guidelines introduces one SOA pattern (see section 4.6.1 for full list of identified SOA patterns) into architecture. The target architecture (see figure 4.4 for target architecture) is a result of application of all the guidelines. This section presents how guidelines were created and what determines order of their application. Guidelines are described by *"How to"* section and figure presenting state before application of the guideline and after the application.

3. Project for migration – the section presents briefly a project selected for an example migration along with selection criteria and their application. Migration of the project illustrates how to apply the guidelines.

4. Application of the guidelines– the section presents an example application of guidelines and technical issues that were identified during migration.

5. Discussion– the section presents a discussion about guidelines and the target architecture that is created as result of their application.

5.1 Pattern Languages

Previous chapters provide two architectures. First that is the migrated architecture provided by MVC architectural pattern. The second is the target architecture that is a result of application of several SOA architectural patterns. The target architecture is in fact a pattern language because [37]:

Pattern languages are collections of patterns that can be used to build something larger than any individual pattern can be used to build.

Since the target architecture may be expressed as a pattern language, it is worth to use this fact and convert transformation between two architectures into translation between two pattern languages.

This section presents the MVC architectural pattern and the target architecture described as pattern languages. The description follows template of description of pattern language presented in section 3.3.1

5.1.1 MVC Pattern Language

This section presents MVC architectural pattern (see 3.2.1 for description) as a pattern language.

Abstract

This pattern language is dedicated for application interacting with users. The pattern language simplifies interaction through clear separation of GUI from logic of the system. Application of the pattern language supports maintenance and development of multiple user interfaces.

Map

This pattern language is one element pattern language only. The map of this pattern language is the map of only one node without arcs. The structure of the pattern is presented in figure 3.10

Description

The pattern language is one element only. The order of application is reduced to the only one step.

Patterns

Name Model View Controller

Problem GUI of modern systems is very complex and prone to changes. The changes include modification of existing views, elements of the views and creation of new views. Many applications offer also dynamic change of 'look and feel'. Nowadays system offer interaction via many different devices. Interaction via keyboard and mouse becomes not sufficient. Users want to access their application using mobile phones or tablets. All those types of devices need a separate and dedicated user interface

Context Applications providing a rich and flexible user interface.

Forces

1. The same information is presented on different types of devices
2. The view should be easy to modify and maintain
3. Changes to user interface should be available in runtime, for instance change of look and feel
4. Changes to user interface should not affect core functionality of the system

Solution Introduce MVC architectural pattern. For more information about the pattern see refsec:PatternDescription.

Result Context Introduction of the pattern simplifies maintenance and of interface. Clear separation of view from logic of the system supports also development of new interface for different types of devices.

5.1.2 SOA Pattern Language

Abstract

This pattern language presents architecture that is created using SOA architectural patterns. This pattern language is a high level solution for implementation of SOA. Description of patterns presented in pattern section is a more detailed version of description presented in section 4.6.1

Map

Figure 4.4 presents the map of the target pattern language.

Description

Order of application of the patterns presented in the "Map" section is determined by impact of each pattern on the final architecture. The first patterns set the frame of the architecture; the next patterns add new elements to the pattern language.

The pattern that has the largest impact on the architecture **Service Layers** pattern. This pattern defines two basic layers. The first (lower) layer contains basic services while the second contains process services. At this point, a way of communication between layers is needed, so the **Canonical Protocol** is applied. Those two patterns define together two layers of services and the mean of communication between them. **Canonical Schema** along with **Policy Centralisation** define application wide documents: schemas and policies. Early introduction of those patterns is important because all the services will share the same documents.

At this point, the main elements (layers,communication protocol, schema and policies) are established. Application of the next patterns introduce basic services into architecture. Service belonging to **Utility Abstraction** are the first to identify and migrate because they provide functionality that is used by other services. The next are services defined by **Canonical Resources** Those services enable usage of external resources. Next **Entity Abstraction** and **Rules Centralisation** should be applied. Those two types of services provide entity–related operations.

Inventory Endpoints is a pattern that introduces an additional layer about basic services. Endpoints are facades. They are introduced when basic services are defined.

Process Abstraction introduces process services. This type of services requires basic services to execute processes. Process services are stateful, it means that they maintain their state for the whole live of the process. This time does not have to be short. **State Repository** can be introduced when it is known what services need state deferring mechanism.

The last elements that remained are GUI related. **Frontends** are the first to be migrated. Accomplishment of migration of frontends makes possible identification of all the places where continuous feedback to user is required and introduction of **UI–Mediator**.

Patterns

1. **Name** Service Layers

   Problem Services delivered by different teams may contain inconsistencies and redundant functionalities.

Context An application is composed of many services that are developed by different teams.

Forces

(a) Many services provide similar functionality.
(b) Functionality of each service is different.
(c) High cohesion of services has to be kept.

Solution The pattern divides service inventory into logical layers containing services with similar functionality. Each layer contains one specific and abstract concern. The pattern enforces usage of at least two layers-one contains basic service, the other contains composed services.

Result Context The application is divided into several (at least two) service layers that support maintainability and further development of this application.

2. **Name** Canonical Protocol

Problem Service can be provided by different vendors. The vendors can use different languages and communication protocols. However the difference in language is not so important, the differences in communication protocols are. Different communication protocols limit number of potential consumers and introduce the need of protocol bridging.

Context An application is composed of services that are delivered by different vendors.

Forces

(a) Services need to communicate with each other
(b) A main way of communication needs to be established
(c) A way of communication cannot limit functionality of services

Solution Application of the pattern enforces usage of one communication protocol as a main mean of communication. Pattern application eliminates bridging services in the system what improves performance and maintainability. In order to increase benefits associated with the pattern, architects should consider usage of widely supported and vendor independent communication protocols like SOAP

Result Context The system has one main way of communication. Each service of the system provide interface for this way of communication.

3. **Name** Canonical Schema

Problem Many services perform operation on the same data that are modeled differently by different vendors or teams. Those differences increase development efforts and make design more complex. Additionally, the transformation between different models introduces performance drop.

Context Application uses communication protocols

Forces

(a) Schema of the data may change over time.

(b) The schema is used in many services from different inventories.

(c) Schema needs to be easily extended.

Solution The pattern standardizes data model for information within the inventory. The motivation behind the pattern is to avoid translation between different definitions of the same data and eliminate data definition redundancy across the system

Result Context System has a centralized schema that easily accessible and maintainable.

4. **Name** Policy Centralisation

Problem Services may require to process individual polices. Policies may refer to security, transaction requirements and Quality-of-Services. Replication of policies across inventories brings redundancy and problems with synchronization of changes to the policies.

Context The system is composed of services that need to process policies.

Forces

(a) Number of services using policies is not fixed

(b) The policies change over time

Solution Application of the pattern separates policies from services.

Result Context The policies are separated from services. It is easier to keep all the policies up to date. Update of policy affects all the services using this policy in the same time.

5. **Name** Utility Abstraction

Problem Services, even those that do not provide redundant functionality may perform some common operation. Those operations are redundant. The redundancy requires additional effort for testing and maintenance.

Context An application contains some common functionality that used in code in several places.

Forces

(a) Common functionalities are used by code in several places

(b) Those functionalities do not exist in requirements - they are rather helpers for architects and developers

(c) Different vendors may not be aware that other vendors implemented the same common functionality

Solution Utility Abstraction defines one layer containing common utility services. The layer contains mainly services without business concern, therefore in opposite to business-based services, utility services are developed mainly by architects and developers what raises problems related to content of such service. Teams need good communication in order to avoid redundant services.

Result Context The application has clearly separated and implemented common functionalities that are further wrapped into services.

6. **Name** Canonical Resources

Problem Modern systems require access to external resources like databases, central directories or transaction management frameworks. The resources are accessed by many subsystems of the system. The access logic becomes a part of the subsystem.

Context An application requires access to external resources

Forces

(a) There are many services that require access to external resources

Solution Canonical Resources identifies and standardizes resources across the company. A term resource refers to databases, activity management systems, state deferral mechanism or other resource like for instance public services. Separation of access to external resources grows in importance when an application is not small, because in small application access to external resources is rather centralized.

Result Context The system has standardized and centralized access to external resources.

7. **Name** Entity Abstraction

Problem Many business processes uses the same business entities. It is very likely that the functionality is implemented several times in different services. This raises problems with maintenance and testability.

Context An application is composed of business services performing operations on the same entities.

Forces

(a) Functionality provided by services cannot be implemented in other services

(b) Business entities do not change too frequently

Solution Entity Abstraction introduces another layer into layered structure. The layer contains services performing operation on business entities.

Result Context The application has some services performing operations on business entities.

8. **Name** Rules Centralization

Problem The same business rules are applied in many different services. This may lead to redundancy and problem with maintenance of the changes.

Context A System uses many rules across services.

Forces

(a) Rules may change over time

(b) New rules may be added

(c) Outdated rules may be removed

(d) One rule may be used by many services

Solution Business rules are stored and accessible through dedicated entities like business rules services.

Result Context The application has unified access and storage of business rules. The rules are easy to update and access.

9. **Name** Inventory Endpoint

Problem An inventory may contain several services providing functionality that might be used outside the inventory. Services accessing the inventory from outside should not know structure of the inventory and its functionality. The access also needs to be secured.

Context An application exposes some of its services to external services.

Forces

(a) The services that can potentially access the inventory are not known.

(b) Structure of the inventory needs to be hidden.

(c) The functionality of the inventory needs to be limited.

(d) Access to the inventory needs to be secured.

Solution Inventory Endpoint establishes an entry point for external service consumers. Each customer accessing service repository, access the endpoint and receives information from the endpoint. Capabilities of endpoints can be extended by for instance direct / brokered authentication. A single Inventory can have more than one endpoint.

Result Context Application of Inventory Endpoints introduces an additional layer of abstraction that guards access to inventories. Endpoint serve as facades to inventories and may provide security related functionality.

10. **Name** Process Abstraction

    Problem Grouping of business process services together with non-business process services makes harder maintenance of both types of the services. It is harder to change processes and maintain functionality provided by other services.

    Context An application contains business process.

    Forces

    (a) Business process cannot be changed.

    (b) Development of business services needs to be separated from development of non-business services.

    Solution This pattern introduces enterprise wide process services as an additional layer. The ser- vices are mainly stateful and manipulate services from all the underlying layers.

    Result Context The system has clearly separated layer of process services. Those services maintain only the process. Other functionality is implemented in other services.

11. **Name** State Repository

    Problem Long living processes occupy system resources for a long time.

    Context An application contains long living processes.

    Forces

    (a) A long living process may need store and restore state few times

    (b) State cannot be lost–restored state must be the same as stored

Solution The pattern introduces a state repository that is meant to maintain state of stateful services. Stateful services delegate their state to special services in order to free system resources. The state is not transferred automatically. The transfer occurs when special criteria are met. The criteria can be for instance an extending waiting for a response. This solution occurs to be very useful when an asynchronous communication is applied.

Result Context State of stateful services can be stored when special criteria are met. Afterwards, when it is needed the state can be restored.

12. **Name** UI Mediator

Problem Business services bind into process many independent services. The execution time of those services may vary and it can take a while to provide result requested by the user.

Context An application is interacting with users.

Forces

(a) Services creating the process can be dynamically changed.

(b) Time needed for execution of an operation varies and depends on load and geographical location of a service.

Solution Application of the pattern introduces an additional layer that processes information about progress of invoked operation. The information is further acquired by frontends and presented to users.

Result Context An application that is able to provide continuously feedback about progress of invoked operation.

5.2 Guidelines

The guidelines for migrated system should use existing code and documentation. Application of Top– Down approach bases more on available documentation because a good understanding of business domain and associated business processes is needed to divide the whole application into smaller parts. Each guideline introduced a SOA architectural pattern in order described by SOA pattern language (see section sec:SOApatternLanguage). Application of a subsequent guideline removes some code from migrated system and migrates it to the target system.

System's services are created based on an available code. The documentation is used to consult if a question arises. The description below presents guidelines with a "How to" section. Additionally a figure presenting current step of migration is provided. Each figure contains green elements. The element between + and = represents introduced SOA architectural pattern. The second green element presents change to architecture.

5.2.1 Description of Guidelines

1. Convert MVC into layers
   Applied SOA pattern: Service Layers

   How to: Service layering simplifies service maintenance. The main idea behind the pattern is to divide services having similar characteristic into separate layers (see figure 5.1). The characteristic of services is described by its functionality, technical aspects, complexity or other properties that allow grouping services. Structure of considered pattern has to be analyzed in terms of its components and connections between them. Elements of the architectural pattern are converted into layers. Original dependencies between elements of MVC should be removed. Layers allow dependencies only in one direction (top–down). Introduction of this pattern simplifies further migration because the pattern frames architecture.

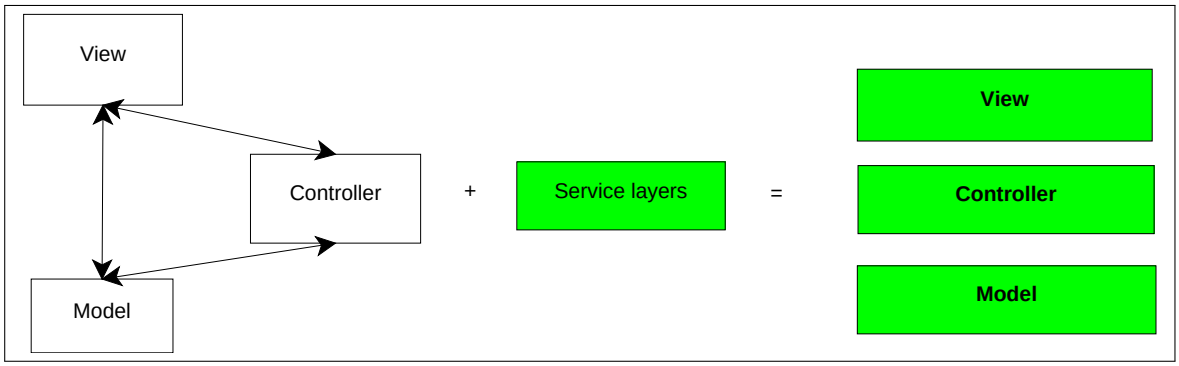

Figure 5.1: Migration step 1

2. Choose main communication protocol
   Applied SOA pattern: Canonical Protocol

   How to: Canonical Protocol recommends usage of one major communication Protocol (see figure 5.2).. In fact the choice is not limited to only one protocol, because communication is transport plus message description

standard. The communication can be conducted for instance over RPC or HTTP with SOAP as a message standard. A pair of HTTP and SOAP messages is widely used in Web Services [32] but it is one of the biggest obstacles during Web Service adaptation [59].However reliable message specification and SOAP reliability headers increase messaging reliability, it still does not guarantee message delivery, message status notification, duplicate elimination and message ordering [59].Therefore SOAP over HTTP should be avoided in mission critical system. Instead protocols like WebShare MQ, Java Messaging Service (JMS), RosettaNet or Electronic Data Interchange (EDI) should be applied [59].

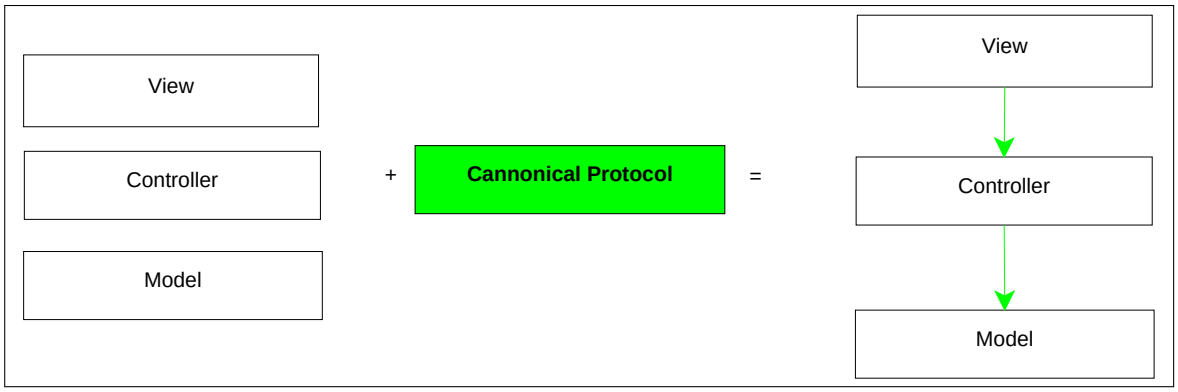

Figure 5.2: Migration step 2

3. Unify used schemas
   Applied SOA pattern: Canonical Schema

   How to: At this point only a schema document is needed. The document will be filed up with definitions when new services will be identified and implemented. The definitions present in schema are further used by other services. One important thing is that while next types of information are defined, they should base on types that are currently defined. In other words, it is highly recommended to reuse exiting definition of information and it does not generate additional dependencies because Schema describes information types in XML. Reuse of existing definitions helps to avoid redundant definitions and standardises domain vocabulary across the company. Finally, schema should be divided into many smaller documents that describe a set of related information. The documents should be broken into namespaces. A naming conversion should be also established. See figure 5.3 for general idea.

4. Unify policies
   Applied SOA pattern: Policy Centralisation

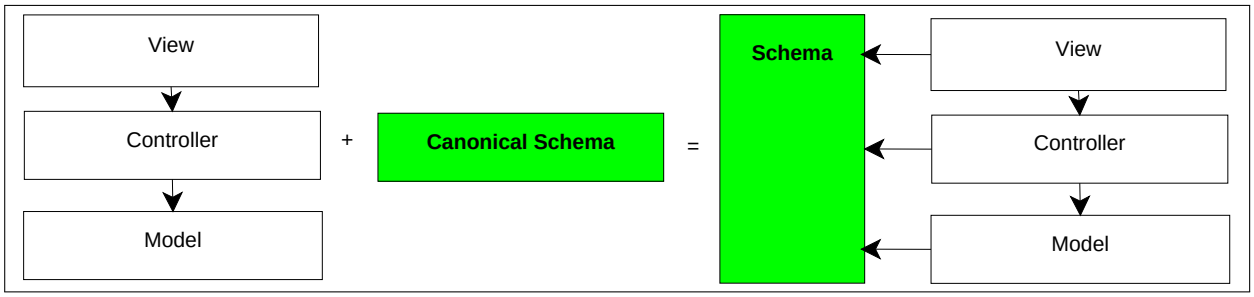

Figure 5.3: Migration step 3

How to: Services may implement policies. The policies can be common for a group of services. The common parts should be shared (see figure 5.4). Policies and security in SOA is a very wide and complex domain, that includes different messaging types, security description files (WSPolicy), encoding and many others. This guideline identifies moment during migration when this aspect should be considered. Information about security related issues should be investigated in existing documentation. Implementation is a project specific task.

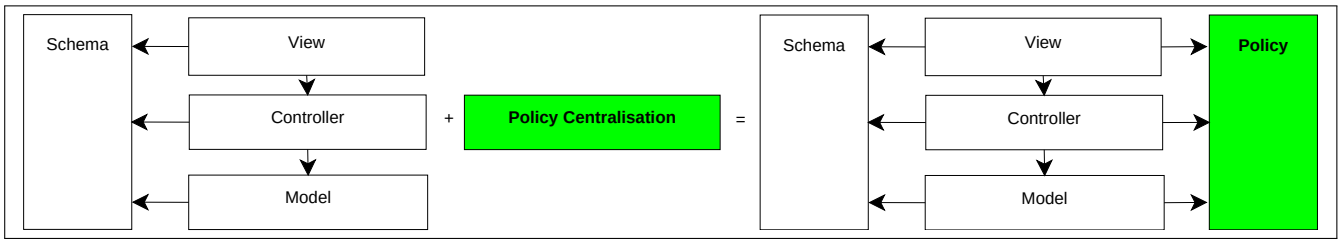

Figure 5.4: Migration step 4

5. Identify and wrap into services all coarse grained utilities
   Applied SOA pattern: Utility Abstraction

How to: Unlike Core Business logic services, utility services usually are not designed by people responsible for business process planning . Utility Services are basic services and they contain general purpose functionalities like for instance event logging facilities. They are created mainly by architects and developers [32]in order to simplify and support current architecture. Separation of utility functionality is difficult. The difficulties derive from role of utilities. They can be used anywhere. Starting from Graphical User Interface (for instance Java SWING has a lot of utility functionalities) through advanced text parsers supporting Core Business logic to Logging components that may log events from all elements of the system. The next

problem is granularity of utility functionalities. While Logging is coarse grained, SWING utilities support fine grained issues like *"boolean isRectangleContainingRectangle(Rectangle a, Rectangle b)"* method. Such fine utilities should not be taken into consideration during Utility Abstraction designing. Garrulity itself is rather hard to define, thus an architect or a developer shall decide whether granularity shall be defined based on size of code of the utility or other criterion is needed. Model of MVC is the place where utility code should be found (see 5.5).

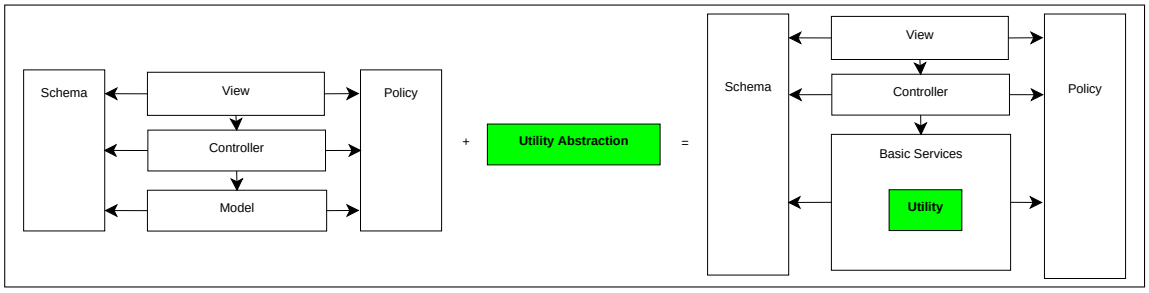

Figure 5.5: Migration step 5

6. Identify and encapsulate access to any external resource
   Applied SOA pattern: Canonical Resources

   How to: Canonical Resource wraps an access to any external resource used by application. Term "external resource" refers to databases, state deferral mechanisms (mechanisms that allow to store temporary state of a service), activity management extensions (like context and transaction management frameworks)[32]etc. It does not include resources like CPU or memory if they belong to the host machine. Model should be checked in terms of having any connection to databases, outside company services or other components storing and maintaining data like for instance service grids. Depending on resource granularity (for instance database size) and complexity of operations executed on the resource, a new service may provide an access to the resource or to operations on resource related to particular entities (see 5.6).

7. Identify and wrap into services all entity related code
   Applied SOA pattern: Entity Abstraction

   How to: Entity–related Core logic of migrated system should be investigated in components elements that manipulate entities. The manipulation includes mainly computations and entity-related tasks. Entity related services can be understood as services providing utilities for entities and should be found in Model (see 5.7).

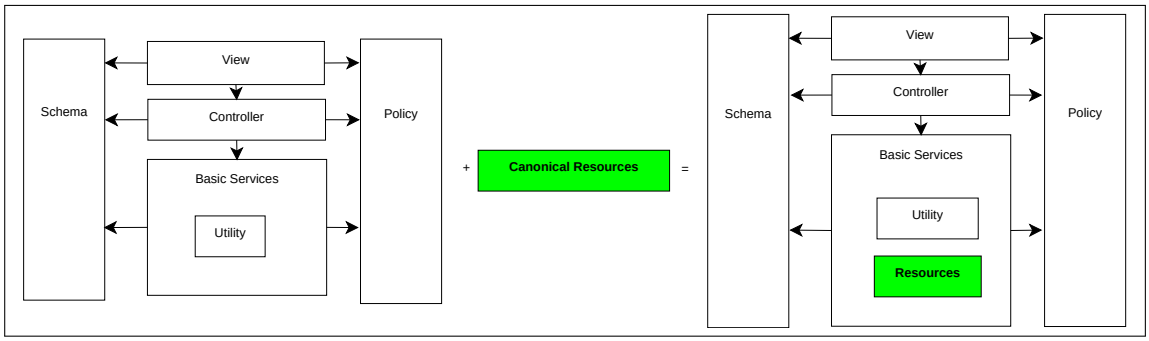

Figure 5.6: Migration step 6

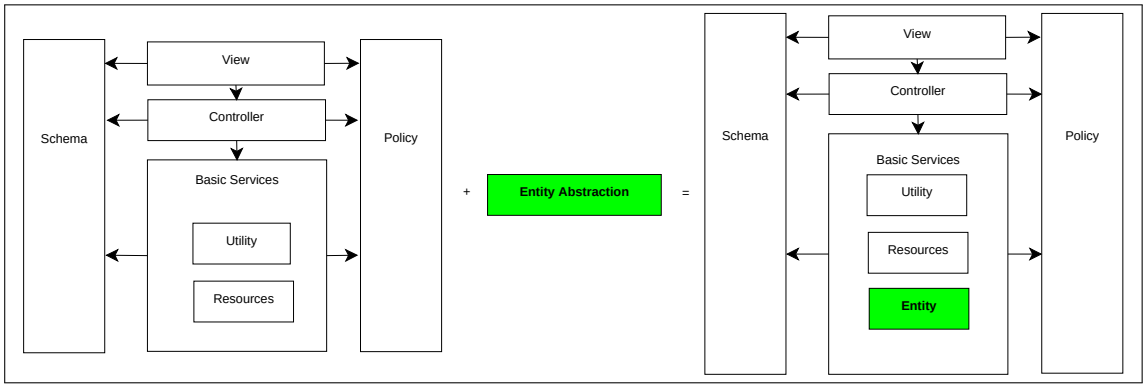

Figure 5.7: Migration step 7

8. Identify and wrap into services business rules
   Applied SOA pattern: Rules Centralisation
   How to: Business Rule can be both very simple like value ¿ 0 or a complex task that relays on many factors. It may be difficult to identify business rules in code, because complex rules may be confused with process control flow. Consequently other source (not only code and code comments) of rules should be chosen. An alternative source of rules and constrains is a documentation (if exists). Firstly, business rules should be divided into three groups:

   (a) First group contains rules that validate arguments of operations for instance *value >0*.

   (b) The second group consist of business rules defining processes, for instance: *" (normal flow) ... if value >0 then add else send message ... (normal flow)"*. Another example: *"after adding an entity a new documentation has to be generated"*

   (c) The third group contains rule describing requirements on level of objects and relations between them, for instance"A supervisor may su-

pervise up to ten students in one semester". This group may partially overlap with the first group.

The first group of rules, due to its simplicity and framework support (for instance JSF allows to add simple validation rules to controls) should be implemented on a client side. The second group should be included during creation of process services. The third group of requirements can be implemented in logic of the system and in database. The code of Model should be check in order to identify all the code validating business rules (see figure 5.8).

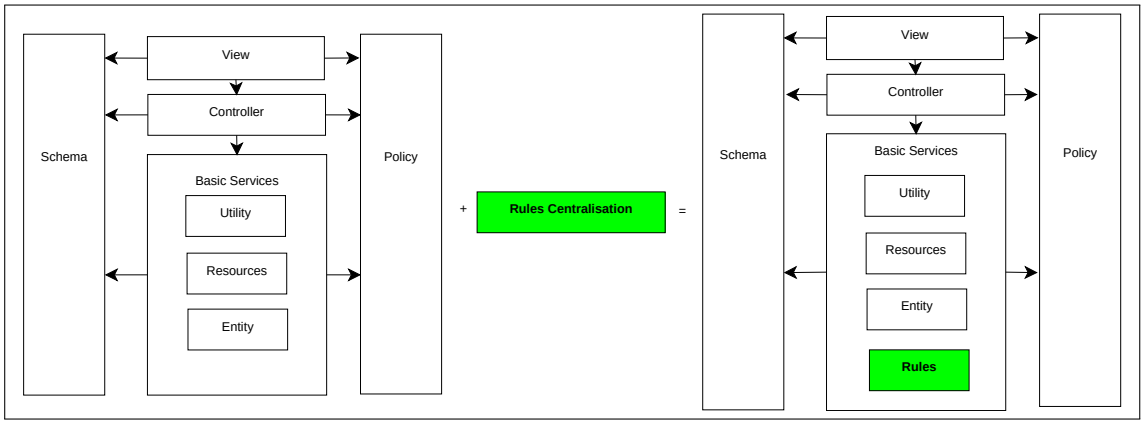

Figure 5.8: Migration step 8

9. Provide inventory endpoints to basic services *Applied SOA pattern:* Inventory Endpoint

   How to: Inventory Endpoint establishes an additional layer of abstraction composed of a set of services that serve as facades. The layer contains both services that limit access to the system (check permissions etc) and services that serve as facades to particular inventories like Canonical Resource. In general, each inventory should have at least one endpoint (facade). Operations provided by endpoints should be defined by architects. The operations can be composed of only operations from underlying services (see figure 5.9).

10. Identify all business processes within the legacy application
    Applied SOA pattern: Process Abstraction

    How to: Business Process logic describes flow of processes within company; therefore all controlling components from Controller should be investigated in order to identify processes. The next source of process-related knowledge is an available documentation. At the moment, all non-controlling

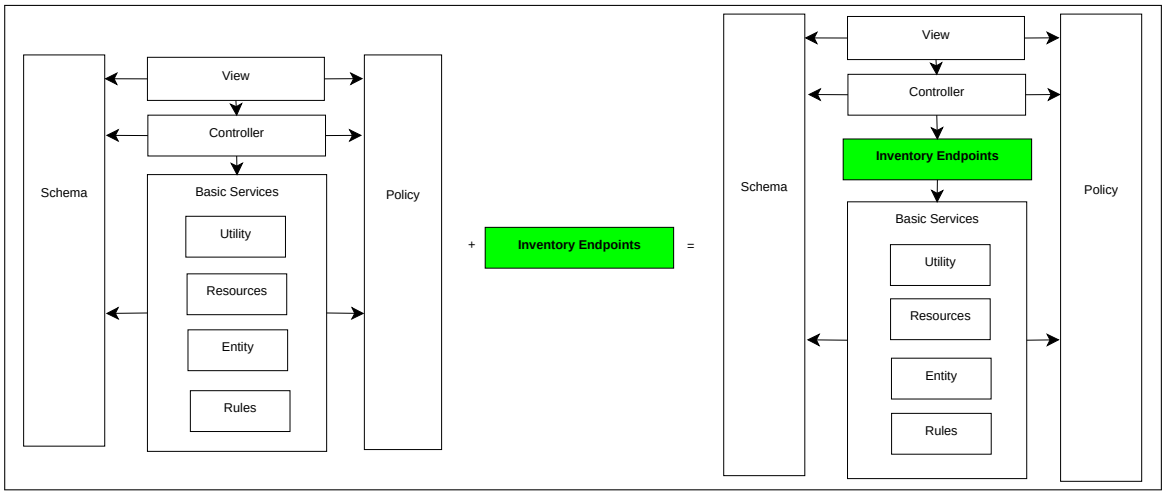

Figure 5.9: Migration step 9

code should be removed from controlling components. If the controlling components still contain additional code, the code should be moved either to elements defined previously like Canonical Resource or elements that will be described like Frontend, State Repository, UI mediator or Inventory Endpoints. The location of the removed code shall base on function of the code and purpose of the listed elements of the architecture (see figure 5.10).

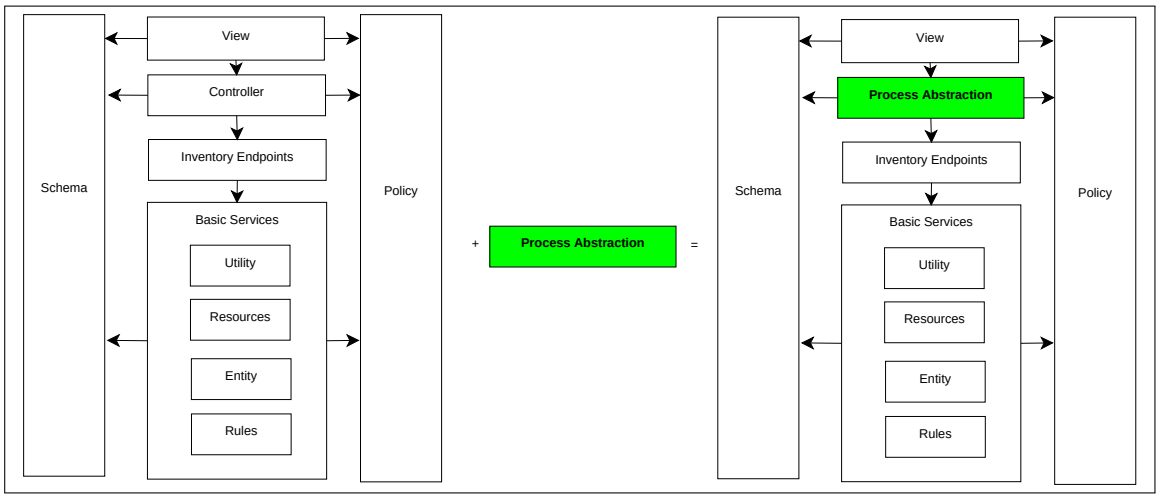

Figure 5.10: Migration step 10

11. Identify all statefull services and decide if their state can be deferred
    Applied SOA pattern: State Repository

How to: Reduces memory load of a stateful service by deferring their states to a specially designed repositories like databases. State deferring requires additional effort related to memory load controlling as well as state storing and recovering, but it may occur crucial for performance. Access to deferring mechanism should be enabled via Canonical Resource [32]. Literature provides two possible ways for state deferring. First, (Partial State Deferral) describes storing of state of services in databases, while second (Service Grid) introduces Grids - an alternative for database state storage

(a) *Partial State Deferral* [32]assumes deferring of only a part of the service's state. The deferred state will be no longer used in the process. Implementation of State deferring mechanisms should be considered when all the processes are identified (see figure 5.11) This will allow the process service to remain statefull and reduce memory load in the same time. In order to fully leverage state deferring pattern, parts of the process that uses the same data should be executed in a sequence if it is possible. For instance a process invokes firstly only services manipulating on one part of a message, then a partial result is stored in database and process manipulates the next part of the message.

(b) *Service Grid* [32]proposes an alternative for state storage. In opposite to a typical database-based, Service Grid proposes state storage in a special dedicated grid of services. The grid contains many services that are synchronized in order to provide security. Besides all the benefits related to state deferring and alternative storage solution, Service Grid has one significant drawback. It is vendor dependent. Grid has to be integrated with SOA infrastructure - typically infrastructure provided by vendor of the grid.

Introduction of State Deferring mechanism is always associated with performance reduction, thereupon the pros and cons of the mechanism should be carefully considered. Nevertheless application of State Repository may pay off when a process is very long or asynchronous. In case of asynchronous processes, final states can be stored and returned once (a few results) instead of a few single returns.

12. Identify current points of access to the migrated system
Applied SOA pattern: Frontends

How to: Available literature presents this part of migration as a complex activity that is not limited only to "separate clients" task. According to [2],Presentation Layer is a complex entity that is broken into three separated elements:

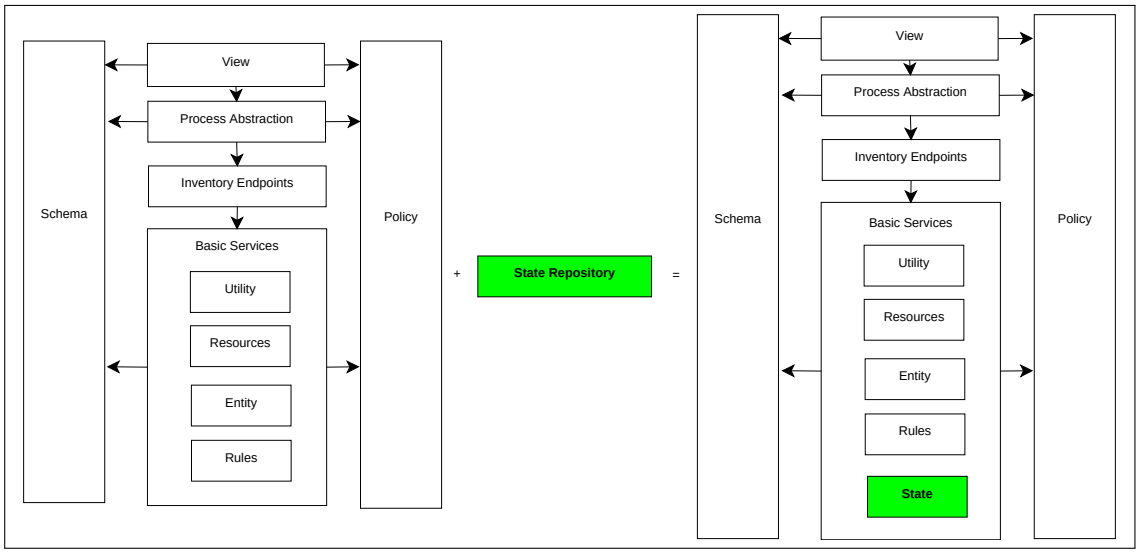

Figure 5.11: Migration step 11

(a) Presentation Service Provider – in opposite to ordinary service, presentation provider does not provide business functionality. The service provides presentation components that can be further integrated with user interface. The components are also connected to corresponding business services.

(b) Presentation Service Consumer – invokes Presentation Service Providers and uses provided presentation components to create a complex User Interface (UI). If elements of UI are coherent, a user is not even aware that different parts of the screen derive from different applications.

(c) Presentation Service Registry – registers Presentation Service Providers and provides all the information as previously described Service Repository does.

There is also SOAUI Composition Framework [81] supporting creation of Presentation Layer that covers GUI in similar scope. However the solution gives additional reusability on potentially not reusable abstraction level, it requires additional effort associated with introduction of the framework. Application of the framework reduces also performance of application what is an implication of additional layers of abstraction. Application of this relatively heavy presentation is fully justified in the field of large business application where UI-related user experience can be a major evaluation factor [81]. In case of MVC, frontends are migrated from View (see figure 5.12).

13. Identify all the places in user interface where a continuous feedback from application to end user is provided

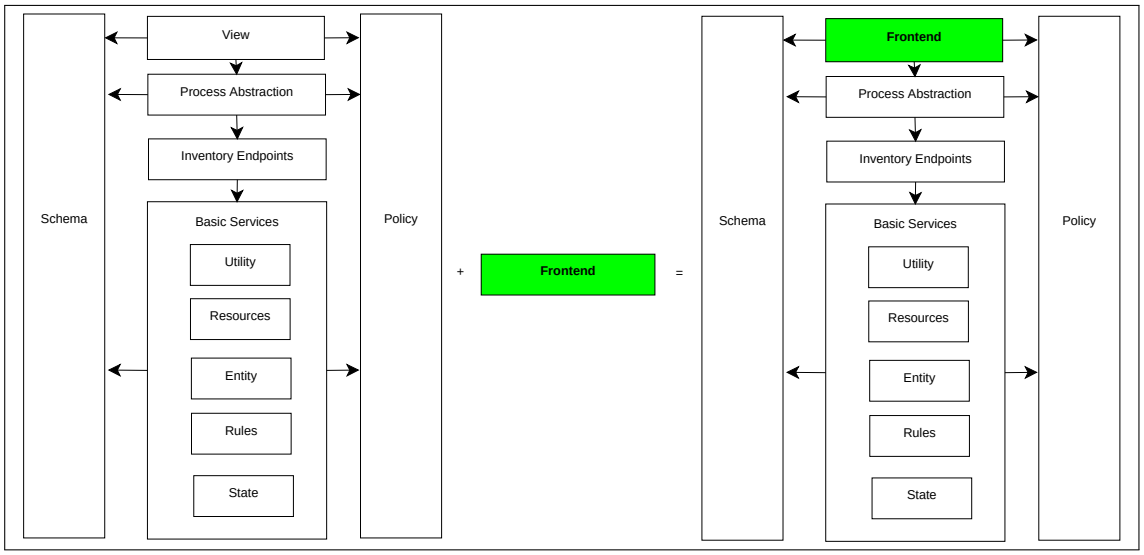

Figure 5.12: Migration step 12

Applied SOA pattern: UI mediator

How to: Provide continuous feedback to users. The feedback includes information about progress of an ongoing process, responding to exception as well as displaying forms in order to acquire additional information from the user [32]. Literature provides two possible realizations of UI mediator pattern [32]:

(a) A mediator service – requires a permanent binding of client application to mediator service for process duration. This kind of mediator service contains own logic that decides when and how often interact with client.

(b) A mediator service agent – a service agent is an implementation of event driven logic. Events are lunched for instance after each step of the process like service invocation. The pattern does not specify how an event can be lunched and intercepted by the agent. The pattern states that an agent has to be stateful in order to continuously notify client application. UI–Mediator related code can be found in already migrated Frontends and Controllers of the migrated system. It is also worth to revise documentation of the migrated system.

5.3 Project for migration

The previous section presents guidelines for migration in order to their application. The guidelines are theory. There is a need of illustration how to apply them

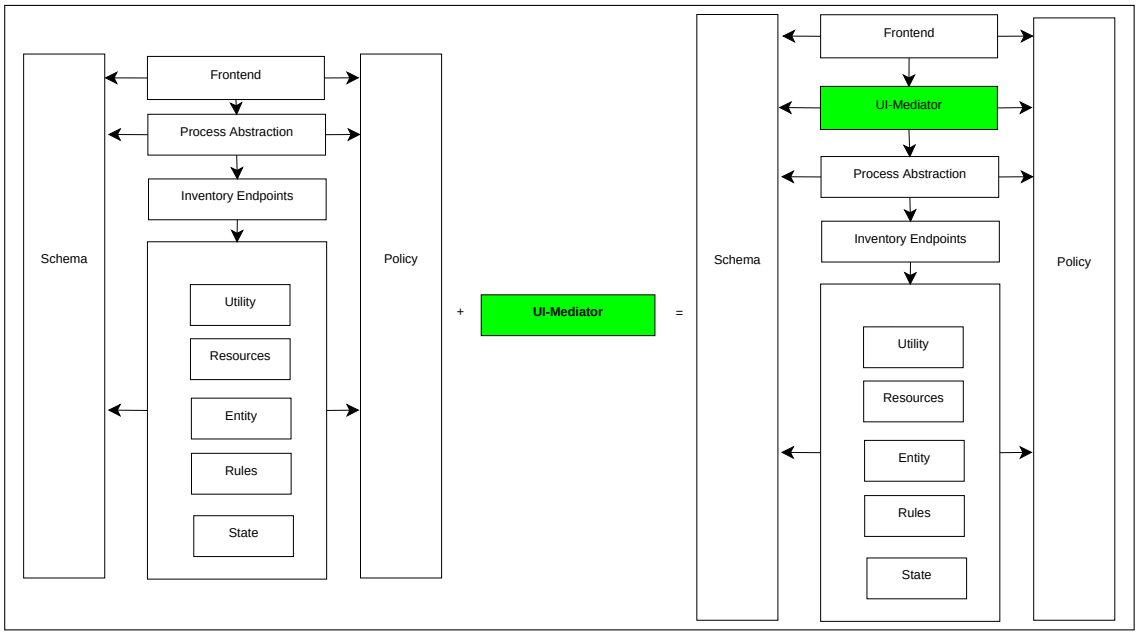

Figure 5.13: Migration step 13

in practise. Application of the guidelines along with comments supplements the theoretical description. This section presents briefly criteria applied in order to select the project for migration along with description of the project.

5.3.1 Selection Criteria

Selection of the example project is conducted according following criteria:

1. The system must have documentation with models including at least overview of architecture and flows of business processes.
2. The system must implement MVC architectural pattern
3. The system cannot be build using frameworks supporting MVC

The first criterion is motivated by a need of reverse engineering if the documentation is missing. This is the first criterion because the guidelines require documentation. All projects that do not have documentation are rejected. However reverse engineering is very useful when documentation is missing and it even was employed in migration toward services [61], it is still associated with an additional effort. Results of Reverse Engineering may also not be satisfactionary and do not motivate decisions taken during design time of the project. Reverse engineering presented in [61] serves also to reconstruct architecture. There is a possibility of identification of architectural patterns when the process is finished, but again there is no possibility to clearly state whether the identified pattern

was implemented intentionally or it is just an artificial product of Reverse Engineering.

The second criterion is quite obvious having selected pattern for migration in mind.

The third criterion rejects projects implemented with frameworks implemented MVC like Zend, JSF, Spring MVC or ASP.NET MVC. This is a serious constrain taking into consideration a number and popularity of those frameworks but the requirement is dictated by the nature of the framework. A framework "frames" a project by providing architecture, Application Programming Interface (API) and preimplemented functionalities. A framework is meant to simplify work by execution of own (preimplemented) code *"behind the scene"*. The code of application using framework is more transparent than code that does not use frameworks. Complexity of the code using frameworks is also lower because framework provides some core implementation. Unfortunately, it is hard to migrate a project that uses a framework because functionality provided by frameworks would have to be reimplemented. Additionally, even frameworks supporting the same architectural pattern provide different scope of the support. The guidelines that do not exclude framework would need to be tailored to a particular framework.

If more than one projects fulfils the criteria, an additional analysis of available documentation of the projects is carried out. The project with the most detailed and descriptive documentation will be selected

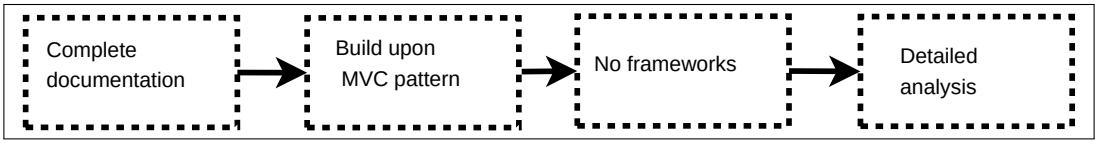

Figure 5.14: Overview of the criteria

5.3.2 Source of projects

Selection of the system to migrate is not an easy task taking previously stated criteria into consideration. At the very beginning, open source systems were considered as potential candidates. Unfortunately, found project did not have any formal documentation- only comments in code (if any) and/or Application Programming Interface (API). They also did not clearly support any architectural pattern - lack of documentation. Theoretically, architectural metrics (see [65] for more information) could be used to identify architectural patterns but themetrics do not prove that the pattern was used intentionally. Additionally, usage of technologies like Java Server Faces (supports MVC) also reduce scope of the search because those frameworks violate previously stated criteria. Due to all the

complications related to open source projects; a new source of candidate systems has to be chosen.

Student's projects occurred to have more documentation; consequently they are better choice than open source systems. Analysis of works of students does not require assumptions - at least theoretically. Process of design of the system is documented with motivation behind decisions taken during design time.

There are many student projects available but majority of them do not fulfil the first criteria. The projects do not have documentation except comments in code. This is caused by the aim of the projects. They are dedicated to show how to apply particular languages or frameworks. Projects developed by students from last years of studies are more elaborated. They have documentation and good quality of code. Unfortunately (for selection process) most of them are implemented with cooperation of companies. Code of those project cannot be published or given to people not participating in it due to Non Disclosure Agreements (NDA).

There were only twelve projects that were available and could be given for research studies.

5.3.3 Application of the criteria

1. The system must have documentation with models including at least overview of architecture and flows of business processes. All twelve projects have documentation having at least following documents:

   (a) Vision– describes overview of the system including description of problem, stakeholders, product features and constrains.

   (b) Software Architecture Document– describes goals and limitations, usecases view, logical view, view of processes, view of deployment, view of implementations, view of data.

   (c) Rules and Limitations– specify business rules and domain constrains.

   (d) Glossary – glossary used in system.

   (e) Usecases– detailed description of usecases.

   *Beginning number of projects:*12
   *Result number of projects:*12

2. The system must implement MVC architectural pattern
   Architectural pattern applied in system is present in seven projects.

   Architectural pattern applied in system is present in seven projects. Five of them do not mention any architectural pattern. Only systems based upon MVC are further take into consideration. The patterns mentioned in the descriptions are:

(a) Pipe-and-Filters
Found amount: 1
This pattern is used in system processing images.

(b) Layers
Found amount: 2
Used in desktop applications.

(c) MVC
Found amount: 5
Used in desktop and web applications.

*Beginning number of projects:*12
*Result number of projects:*5

3. The system cannot be build using frameworks supporting MVC
Three out of five systems were web applications. Those applications were build using web frameworks for JAVA. Found frameworks are following:

   (a) Spring 1.0 MVC –this framework was applied in two systems.

   (b) JSF 1.0 –applied in one system.

   *Beginning number of projects:*5
   *Result number of projects:*2

Application of the criteria results in two potential systems for migration. A more detailed analysis is needed in order to choose the system for migration. Both of them are written in Java and have documentation. They are also using MVC architectural pattern.

Analysis of documentation of one of the project questions usage of MVC pattern in the project because:

1. Document describing "Project of the architecture" in point three says that: *"Do implementacji aplikacji klienta przyjęta zostanie architektura MVC (Model View Controller)."* MVC architecture is used to implement client application.

2. The "Project of the architecture" document shows that system is composed of database server and client

3. Document describing "Design of classes" presents following figure described as "Architecture of the system":

Concluding, the authors of the system claims that the system is build using MVC pattern but the diagram presenting architecture is Layers, not MVC. Even the figure contains names like Layer. The diagram contains Top Layer and Middleware Layer. The architecture contains no Model, View or Controller names.

Consequently, this project cannot be taken as an example project for migration..
*Beginning number of projects:*2
*Result number of projects:*1

5.3.4 Description of the selected project

The selected student's project is meant to support maintenance of IT assets. Name of the system is "TRWAM". The system is a desktop application written in Java with SWING user interface. The system performs create, read, update and delete operations (CRUD). The description below presents roughly the system. The information derives from available documentation.

Vision

According to the vision, the system is developed in order to replace an existing system. The new system has to have the same functionality as the old, has to preserve standards of data persistence and enable printing of reports. The system supports following main activities:

1. Maintenance of database of assets
   The system must provide functionality of creating, reading, updating and deleting information about assets. The asset related information includes for instance name of the asset and provider of the asset. Search functionality is also required.

2. Support in maintenance of assets
   The system must provide functionality of generation of documents based on content of database and applied criteria. The documents are associated with for instance process of lending (lending form) and removing (protocol of removal) of assets.

3. Automation of activities associated with maintenance of assets
   This functionality includes automation of calculations of fees, fines and amortizations. The system should also inform about violation of rule and other incorrectnesses.

4. Creation of reports and listings
   It includes creation of reports and listing based on actual state of database.

Software Architecture Documentation

The overview of the architecture of TRWAM project is presented in the figure 5.16: Content (according to available documentation) of particular elements of the architecture is following:

1. View – contains packages representing graphical part of the system. Package *Assets (SrodkiTrwale)* is responsible for proper representation of assets on Graphical Usage Interface. Package *Commons (Wspolne)* contains classes responsible for common parts of Graphical User Interface.

2. Controller – here also two packages are available. Package *Assets (SrodkiTrwale)* contains classes associated with service of assets. Package *Commons (Wspolne)* contains classes common for the whole application like classes managing printing.

3. Model – represents classes responsible for persistence of data in database. Package *Assets (SrodkiTrwale)* contains classes responsible for persistence of assets. Package *Commons (Wspolne)* contains classes used by other classes. Package *Data (Dane)* contains classes persisted in database.

4. javax.swing – this is a core package of Java language containing classes responsible for displaying GUI.

5. java.sql – this is a core package of Java language containing classes responsible for establishing connection to a database. The package allows sending data to previously connected database.

6. javax.print – this is a core package of Java language. The package provides print-relate functionality that allows discovering and selecting printers, set page format and submit documents to selected printer.

Rules and Limitations The rules and limitations presented in TRWAM project are divided into two types:

1. Business rules – document contain sixteen business rules. An example rule: An asset can be lent only if the manager of unit gave permission.

2. Domain constrains – document presents only five domain constrains. An example constraint: System can use both PLN and Euro currency.

Glossary

The glossary defines basic vocabulary used to describe domain of the system.

Usecases

The system shall implement following usecases:

1. Add a vendor information
2. Add an asset
3. Add / remove spare parts for internal filing system
4. Generate asset's documentation
5. Revise asset's information
6. Remove a vendor information
7. Remove an asset
8. Lend an asset
9. Search an asset vendor information
10. Search for an asset
11. Change asset's configuration
12. Return an asset

Overview of available usecasees is presented on figure 5.17

5.3.5 Implementation

TRWAM is a desktop application. Users invokes functionality of the system using Swing GUI (see figure 5.18).

The system has several use cases, only one asset related CRUD operation was completely implemented - add an asset. The code has only several classes and no unit testing. The only comments that exists in the code are the comments auto generated by Integrated Development Environment (IDE). The project is not large, but it is sufficient to show how to implement guidelines.

5.4 Application of the guidelines

1. Convert MVC into Layers
   Applied SOA pattern: Service Layers

   Application: MVC architectural pattern consists of only three components and dependencies between them. Components of MVC are translated into

layers. View becomes View Layer, Controller becomes Controller Layer and Model becomes Model Layer. Connections between particular components also have to be reorganized in order to minimize dependencies (coupling) between layers. Dependency reduction can be achieved by removing registration mechanism from Model. This change removes notification functionality what not always is intended.

Lack of the mechanism eliminates Model ⇾ View and Model ⇾ Controller dependencies. This change also eliminates transitive dependencies. Additionally the View ⇾ Model dependency needs to be removed In case of the migrated project.

Output projects: Frontend, ProncessAbstractionService and Basic Ser- vices. The Frontend project is meant to contain all GUI related code. Pro- cessAbstractionService is meant to contain BPEL. Basic Services project is a temporary project that is meant to contain code that cannot be copied to any of the existing services. This project will be deleted at the end. The code from this project will be further transformed into services.

2. Chose main communication protocol
   Applied SOA pattern: Canonical Protocol

   Application Project meant for adaptation is not a mission critical system. The system is also used only within university by very limited group of people and any external user (non-university IP) does not have any access to the application. Therefore a standard HTTP plus SOAP messaging is applied for project migration. Besides, technologies like RosettaNet or WebShare tie architects to vendors, while SOAP and HTTP are vendor independent.

   Output: Communication protocol: SOAP over http

3. Unify used schemas
   Applied SOA pattern: Canonical Schema

   Application: In order to simplify schema maintenance following name notation is used:

   (a) name of a namespace have to refer to the name of lower layer when namespace contains definitions of information used in communication between two layers. For instance information used in communication between Process Abstraction and Canonical Resource will be placed in a namespace referring to Canonical Resource.

   (b) schema is divided into a set of sub–schemas. Each of sub–schemas defines one “main” complex type and a set of related types. In case of AssetDAO, the main complex type is Asset and related types are: Com-

puter (extends Asset), Part (composes Asset), AssetNumber (identifies Asset). The name of such schema is defined as <namespace><nameOfComplexInformationDefinition>.xsd. For instance resourceAsset.xsd.

Output: XML schema. The schema is located in ProcessAbstractionService. The document will contain definition of arguments of service's methods and their return types.

4. Unify Policies
   Applied SOA pattern: Policy Centralisation
   Application: Available documentation does not require special security protocols. Application of policies is a complex task. See [44] for more information.

   Output: No new elements are introduced in case of this project.

5. Identify and wrap into services all coarse grained utilities
   Applied SOA pattern: Utility Abstraction

   Application: Presentation Layer has only GUI related code, while Controller except process flow code has some code responsible for maintenance of temporary assets. Code of Model also do not provide any "utility" code. The project does not have utility related code, therefore application of this guideline in case of TRWAM will not introduce changes to the target architecture.

   Output: No new services are implemented in case of this project. If a Utility Abstraction related code would exist; it would be wrapped into a basic service.

6. Identify and encapsulate access to any external resource
   Applied SOA pattern: Canonical Resources

   Application: In case of MVC, the only place where an access to Resource is allowed is Model. Therefore any resource-related parts of Model should be identified. Analysis of TRWAM project shows only two resource related classes. The first, AbstractDAO defines general CRUD operations. The second is AssetDAO (SrodkiTrwaleDAO) which executes asset related operations on database. AssetDAO becomes a service providing asset-related database operations.

   Output: Two projects are created. The first is crAssetService providing asset related operations like addAsset (adds a new asset) and getCurrentMaxNumber (returns max id of an asset). There is also removeAsset operation but it is only a stub that takes crComputer as an argument. This

stub was needed to make crComputer available. This is a technology related issue (see 5.4.1 for more details). The second project is crCommons. This is a shared library that is included to canonical resource project. The project provides database related operations like executeResultQuery or getConnection.

7. Identify and wrap into services all entity related code
   Applied SOA pattern: Entity Abstraction

   Application: MVC pattern has two components containing business logic. The first is Model which should be investigated at the begging and controller which potentially can contain core logic but it should not. TRWAM Project is a Create-Read-Update-Delete (CRUD) based application. TRWAM's Model contains only DAO classes and corresponding entities. Controller does not contain only work flow controlling code. It also has a code responsible for generation of a new id of assets. This generation activity should not be a part of controller, because controller delegates tasks. Existence of a non-work flow related code attests to braking structure of the pattern. Asset number generation should be separated into an asset-related Core Logic service.

   Output: A new project is created. The service is named eaAssetService. The service provides operation Service providing following operation "generate identifiers" The operation takes collection of services and starting number as arguments.

8. Identify and wrap into services business rules
   Applied SOA pattern: Rule Centralisation

   Application: Available TRWAM documentation (SPECYFIKACJA_REGU__I_OGRANI.doc) contains sixteen business rules and five domain constrains.

   (a) Group 1: OGR/2, OGR/3
   OGR/2: An asset has to belong to one category: time depreciating or others

   (b) Group 2:
   Rules defining processes are described in form of use cases.

   (c) Group 3:REG/01, REG/02,REG/03,REG/04,REG/05,REG/06,REG/07,REG/08, REG/09, REG/10, REG/11,REG/12,REG/13, REG/14, REG/15,REG/16, OGR/1, OGR/4
   REG/02: An asset can be lend if executives allowed.
   REG/03: Lending person creates a lend note during asset lending.
   OGR/4: Numbers of assets are unique. Each asset may have more than one asset number.

Group 3 may be encapsulated in services but there is no implementation referring to those rules in code, consequently no rules validation services will be created during migration .

Output: No services are implemented in case of this project. If a Rule Centralisation related code would exist; it would be wrapped into a basic service as it is done in the previous guideline.

9. Provide inventory endpoints to basic services
   Applied SOA pattern: Inventory Endpoint

   Application: Inventories in TRWAM project consists of only one service due to a small size of the project. Introduction of an additional service (inventory endpoint) that will be a facade to the existing services is simply redundant. Such introduction would unnecessary complex the process and introduce overhead.

   Output: No new services are implemented in case of this project. Implementation of an inventory endpoint in case of this system makes no sense because inventories have only one service. The endpoint service would have exactly the same interface.

10. Identify all processes within the legacy application
    Applied SOA pattern: Process Abstraction

    Application: MVC architectural pattern has one component that takes care about Business Process Logic within application - Controller. In case of TRWAM project, Controller is not implemented properly what was noticed in Entity Abstraction part. This is not the only problem of Controller in TRWAM project. TRWAM's Controller contains also code responsible for maintenance of temporary assets. Temporary asset is an asset that was defined by a user but was not persisted yet. Due to wrong Controller implementation, the flow of process has been distorted. The process is artificially divided into two separated processes. The fist process creates identification numbers for temporary assets. The number is based on actual maximum value of asset primary key and requires additional computing. Computation of a new number for temporary assets does not make sense at all. The assets simple may not be persisted and all the computation effort is wasted. The second process is meant to persist temporary assets. A proper work flow of the process should be following:

    (a) Create an asset.

(b) Assign a temporary identifier to created asset. Format of the id of the temporary asset is not specified. It does not matter for temporary assets.

(c) if asset is meant to be persisted, generate a proper (unique) identity number, otherwise do not generate a number.

(d) Persist selected assets.

The rationale for having Business Process Logic "clean" from non- flow related code is usage of Business Process Execution Language (BPEL) to orchestrate processes. BPEL allows only to design a process flow and the language syntax does not allow introducing any code (like Java, C#) within process. It is a graphical/xml language that bases on invocation of previously defined services.

Output: Two addition documents are created as a result of application of this steps. Those documents are:

(a) WSDL –defines available request and response SOAP messages. Links XSD schema and sets some technology related parameters like PortTypes and bindings.

(b) BPEL –this document defines the process, invoked services and their operations.

11. Identify all statefull services and decide if their state can be deferred
Applied SOA pattern: State Repository

Application: Considered migration example have no asynchronous or long living processes.

Output: No new services are implemented in case of this project because there are no long living processes. The implementation also does not contain any process that consumes a lot of resources. An implementation of a state repository would be similar to implementation of other basic services.

12. Identify current points of access to the migrated system
Applied SOA pattern: Frontends

*Application:*Considered project is neither large nor User Interface sensitive. Only a few people use current application and they got used to current interface. Introduction of a complex presentation layer will not pay off at all, therefore a simple migration of View Layer towards Frontend Layer is sufficient.

Output: During this step, GUI was migrated to Frontend Project. The Project contains the original GUI and some Business Objects that were created based on XML schema.

13. Identify all the places in user interface where a continuous feedback from application to end user is provided
Applied SOA pattern: UI mediator

Application: Available documentation and implementation does not require any functionality providing continuous feedback to end users.

Output: There is no implementation regarding this pattern.

5.4.1 Results

The guidelines presented previously were a base for migration of the student's project. Purpose of the this migration was didactic. It was meant to illustrate an example application of the guidelines. The migration was conducted according to the sequence listed above. As a result of migration, an asset storing process was created. This is also the only fully implemented process in the migrated project (see BPEL 5.19, deploy model 5.20). The migrated process consists of following steps:

1. Create Asset object via UI
2. Retrieve the highest stored Asset id in the database
3. Generate unique identifier for the entity
4. Store the entity

The migration process identified few technology related issues affecting the migration. The issues were following:

1. Encapsulation – The project had very poor implementation that even did not include encapsulation of properties of classes.All the properties were accessible via <objectName>.<property>.Lack of access methods (get /set) did not allow migrate classes as there were. Additional implementation of access methods was enforced by openESB. In fact this is very minor issue because today Integrated Development Environments (IDE) allows generating access methods automatically. The IDEs allow also to identify places in the code where a property is used explicitly. Both features helped to refactor code in required scope.

2. Redundant operations – Connection to a service requires creation of a service client in the application. Netbeans supports this activity by generation of stubs of services, ports, connections, operations and messages stubs automatically. IDE will generate only stubs of classes that are inputs for operations or that are building blocks of operation input message. So far so good, but the problem arises in special circumstances as it was with migrated project, namely one of the classes extended the other (CrComputer extends CrAsset) and this sub class (CrComputer) was neither a part of any existing operation of Canonical Resource service nor a part of any message accepted by this service. So when a user wanted to pass CrComputer object through an operation which takes CrAsset as an argument (what is possible thanks to polymorphism), it was impossible because a stub for CrComputer was not created; consequently a user has no possibility to use CrComputer stub at all. Creation of CrComputer stub manually does not work either because the IDE recognizes changes and before run, it rebuilds generated classes, besides there is no guaranty that autogenerated client does not depend on any "behind the scene" configuration files. The only possible solution was to create a "doing nothing" operation that takes CrComputer as an input- this operation ensures that the missing stub is generated properly.

3. Dependency refresh – IDE generates and refreshes automatically dependencies created during project development, at least in theory. It appeared several times that a part of removed dependency was left somewhere in a WSDL document, configuration file or somewhere else. Each case caused hard to find problems.

4. Schema inclusion – XML schema allows to include other schemas from the same or different namespace and use imported elements/complex types as own. The problem arises when a schema that imports other schemas is used in a WSDL document, this in turn is an input for automatic client

generation. OpenESB cannot parse such schemas and prompts information that included file was not found. The file exists, moreover it was included via Nebeans build-in menu. The way around is to copy content of included schemas to a current schema but it is neither elegant nor easy to maintain, because each change to "base" schema requires copping of its content to the current schema.

5. Deprecated methods – since the migrated application was a few years old, it contained a few deprecated methods related mainly to operation on dates. During project building a user is informed that there are some deprecated methods that may cause an error and this is only a warning, but while the project starts a method that calls a service and contains deprecated methods is simply skipped without any information. This is quite serious problem during migration, because it may occur that a significant part of API is deprecated. The problem can be eliminated relatively easy when an official API is used because the API provides not only information that a method is deprecated; it also provides information which method/class should be used instead. Usage of non-official API does not guarantee such information.

6. Schema namespaces – produces the most serious technology related problem, which was observed during the migration and it is related to variable coping. Normally coping can be done in two ways.

   (a) The output variable of one activity is copied/casted to input variable of the second activity. It corresponds to Variable1 =Variable2.

   (b) Each element of Variable2 to copied to Variable1 manually. It corresponds to Variable1.element1= Variable2.element1 etc.

   The problem appears when two variables (Variable2 extends Variable1) are casted (1st way) but they are from two different namespaces . It should work, but usually it does not [40]. According to official forum [41]: *"Our type cast feature isn't perfect yet. There are quite many restriction for using it."* All the problems should be resolved in future versions (The problem occurs with Netbeans 6.7.1). There are at least three possible workarounds. There are at least three possible workarounds. The first is to create a version of operation for each Variable1 extension and invoke directly appropriate operation; the second is to add a flag that will allow BPEL process to invoke a proper operation. The last is to avoid different namespaces. The first solution was tested and it works fine.

Migration was finally accomplished according to the guideline, it occurred to be more difficult task that it was expected. The difficulties did not arrive from guidelines adaptation problems but they were technology related. The technology,

namely vendor implementation, played more significant role than it was expected. This fact underlines a role of vendor selection and in the same time it questions vendor diversity as SOA advantage. Additionally, application of some guidelines did not bring any result because there was no related code (see figure 5.21 for the result architecture).

5.5 Discussion

Application of guidelines migrates systems that base on MVC architectural pattern to Service Oriented Architecture. This section presents the target architecture and the guidelines in context of previously presented information.

5.5.1 Processes

According to section 4.3.2, creation of SOA is a result of sequence of following activities: Service Identification, Service Categorisation, Service Specification, Service Orchestration and Service Realisation. The guidelines presented in this section match this process with small modifications. The modifications are caused by the fact that the original process is meant to elaborate system based on SOA from scratch while the guidelines migrate an existing system to SOA. The process of migration can be described as following activities:

1. Context Establishment
    (a) Service Layers
    (b) Schema Centralisation
    (c) Policy Centralisation
2. Service Categorisation + Service Identification
    (a) Entity Abstraction
    (b) Rules Centralisation
    (c) Canonical Resources
    (d) Utility Abstraction
    (e) Inventory Endpoint
3. Service Orchestration
    (a) Process Abstraction
    (b) State Repository
4. Frontend Migration

   (a) Frontends
   (b) UI–Mediator

The first activity is meant to frame the architecture, thus patterns introduced during this activity have the largest impact on the target architecture. Result of every other activity is incorporated into frames crated during the first activity.

Service Categorisation and Service Identification are listed together because the guidelines try to identify services belonging to particular category. For instance, the code is analysed in order to identify utility related code which is further converted into services.

Service Orchestration is conducted when all the basic services are identified.

The last activity is migration of frontends. Implementation of frontends is not mentioned in the list of activities from section 4.3.2. Frontends request access to application. They do not have to be created with implementation of a new system, but the system that is already implemented have some user interface. This interface should also be migrated.

The activities listed in this section do not contain Service Specification and Service Realisation. Those two activities are performed simultaneously with the second and third activity from the list above.

5.5.2 Structure

Section 4.4 presents three types of architecture. The target architecture crated as a result of application of guidelines corresponds to Process–Enabled SOA. Mapping between layers of Process–Enabled SOA and the target architecture is following:

1. Enterprise Layer – contains application frontends
   (a) Frontends
   (b) UI-Mediator
2. Process Layer – contains orchestration related services
   (a) Process Abstraction
3. Intermediary layer – facilitates technical and conceptual integration
   (a) Inventory Endpoints
4. Basic Layer – provides core functionality

(a) Entity Abstraction
(b) Rules Centralisation
(c) Canonical Resources
(d) State Repository
(e) Utility Abstraction

The list does not contain Service Layers, Schema Centralisation and Policy Centralisation. Those three patterns set frame of the architecture. Service Layers define the layers presented above while the remaining two support in maintenance of application wide technical elements: schemas and policies.

5.5.3 Advantages and Drawbacks

Advantages

1. *The target architecture represents Process–Enabled SOA*
   Process–Enabled SOA is the most complex architecture. This architecture can be used also as a target architecture for systems that are not result of migration.

2. *The guidelines use properties of migrated system – namely applied architectural pattern*
   Elaborated guidelines supports migration of systems based on MVC architectural pattern

3. *Documentation is not required but it is very helpful* Fact that the documentation of the migrated system is not required is an advantage because the systems that are migrated are maintained after they are released. During maintenance, the changes that are introduced to code are not always reflected in documentation.

4. *The migration is systematic*
   People dealing with migration of systems based on MVC have now an elaborated list of steps that should be executed.

Drawbacks

1. *Migrated System cannot use frameworks supporting MVC pattern.* Modern systems often use frameworks that provide a lot of out–of–box code. The code makes implementation simple but more difficult to migrate.

2. *The guidelines have limited application – only systems build on MVC pattern can be migrated*
   The guidelines were elaborated only for MVC architectural pattern. They are not applicable as they are for other architectural patterns.

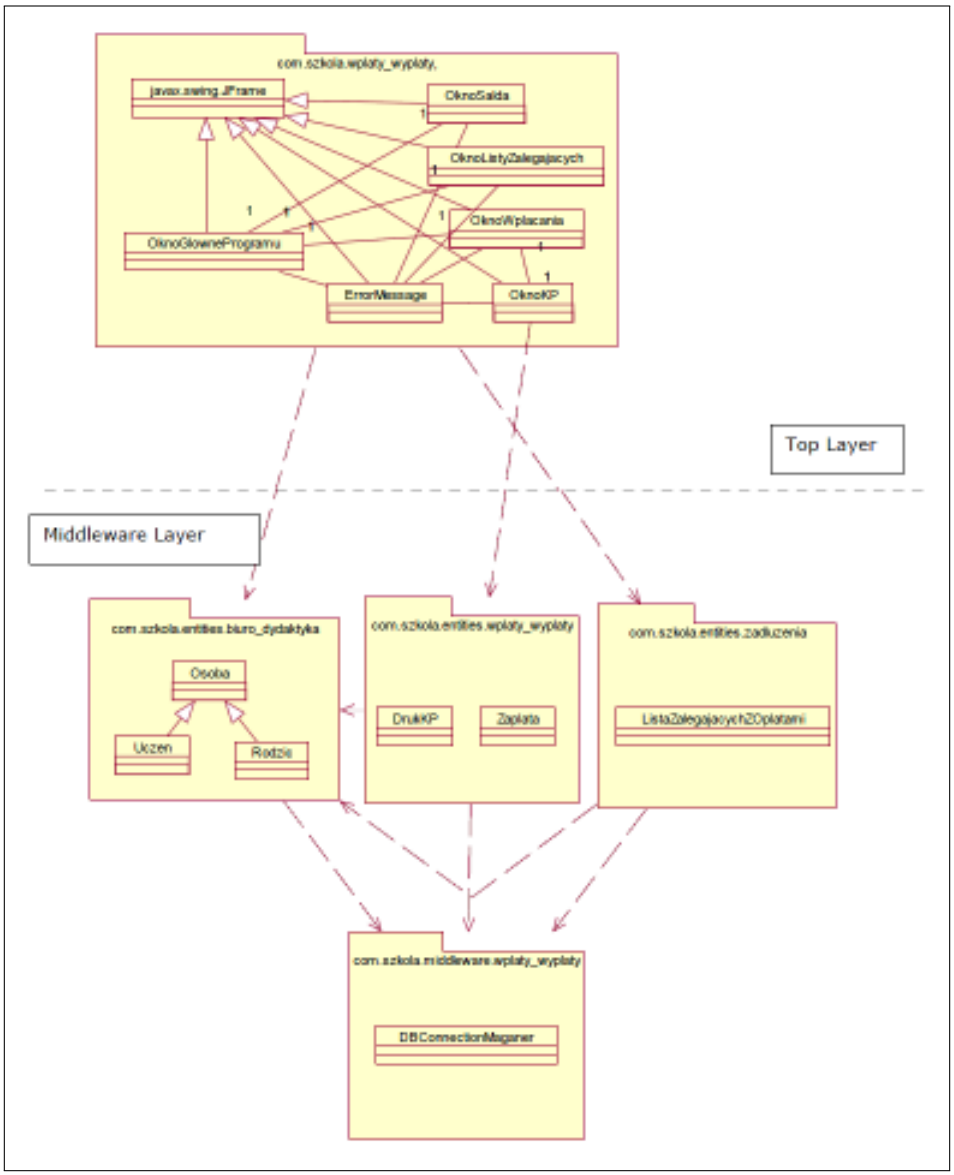

Figure 5.15: Architecture described as MVC in one of projects.

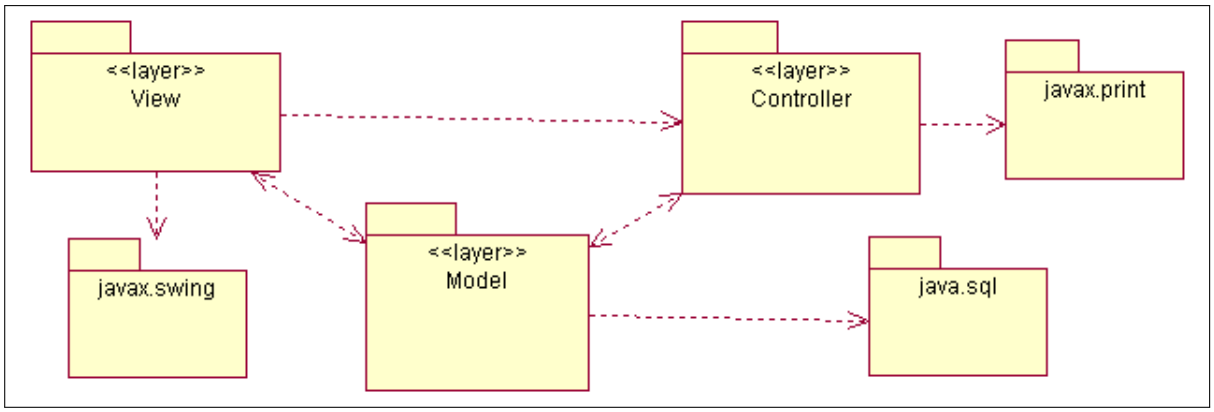

Figure 5.16: Architecture of TRWAM system

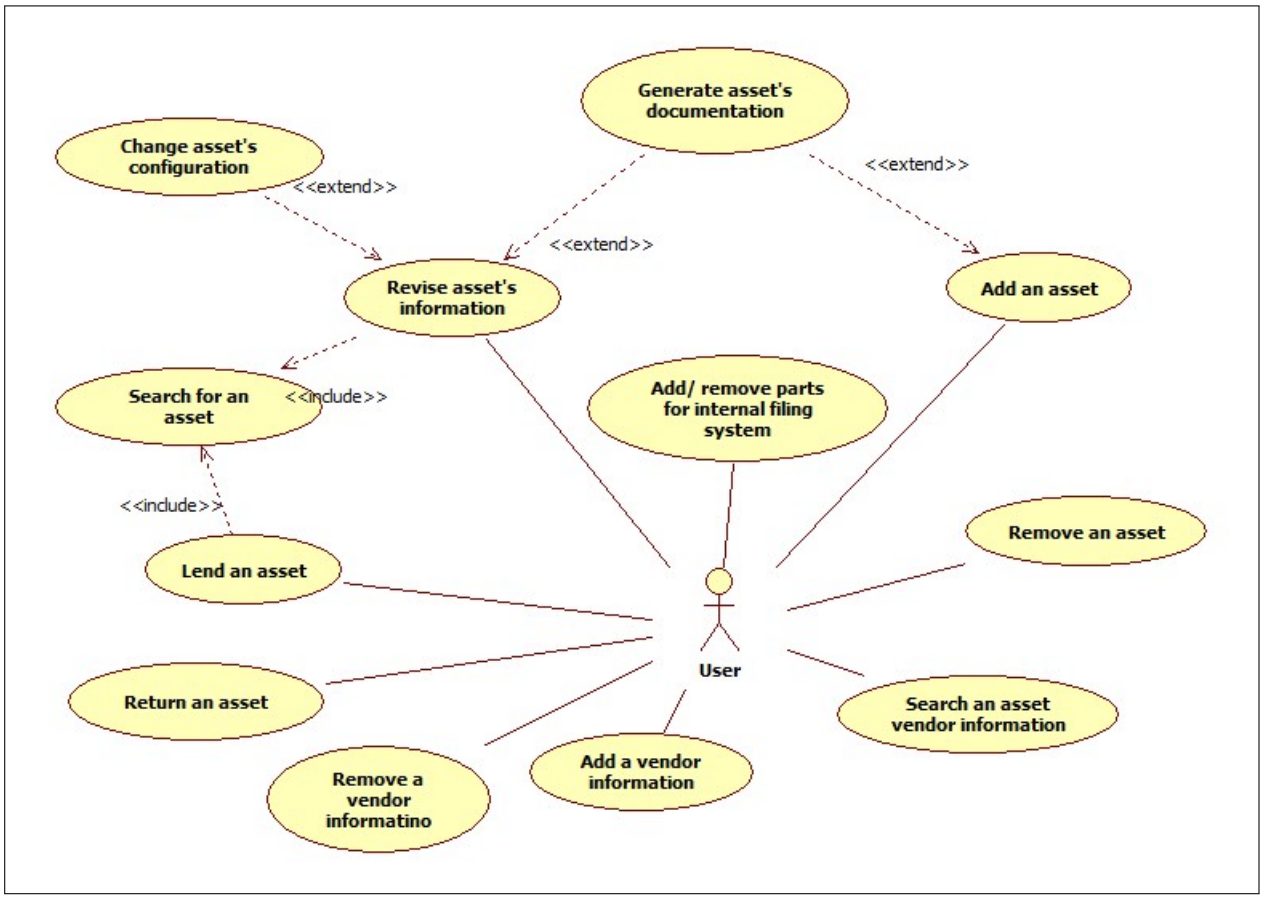

Figure 5.17: TRWAM – all the use cases

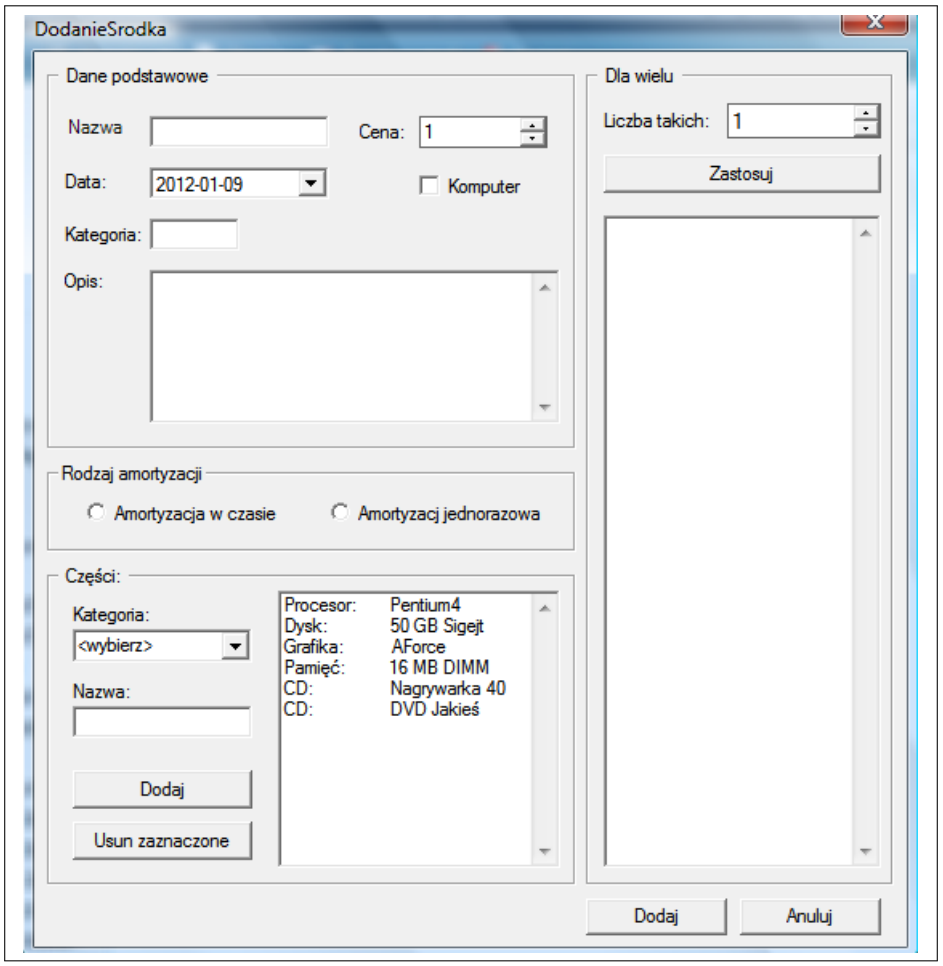

Figure 5.18: GUI of TRWAM

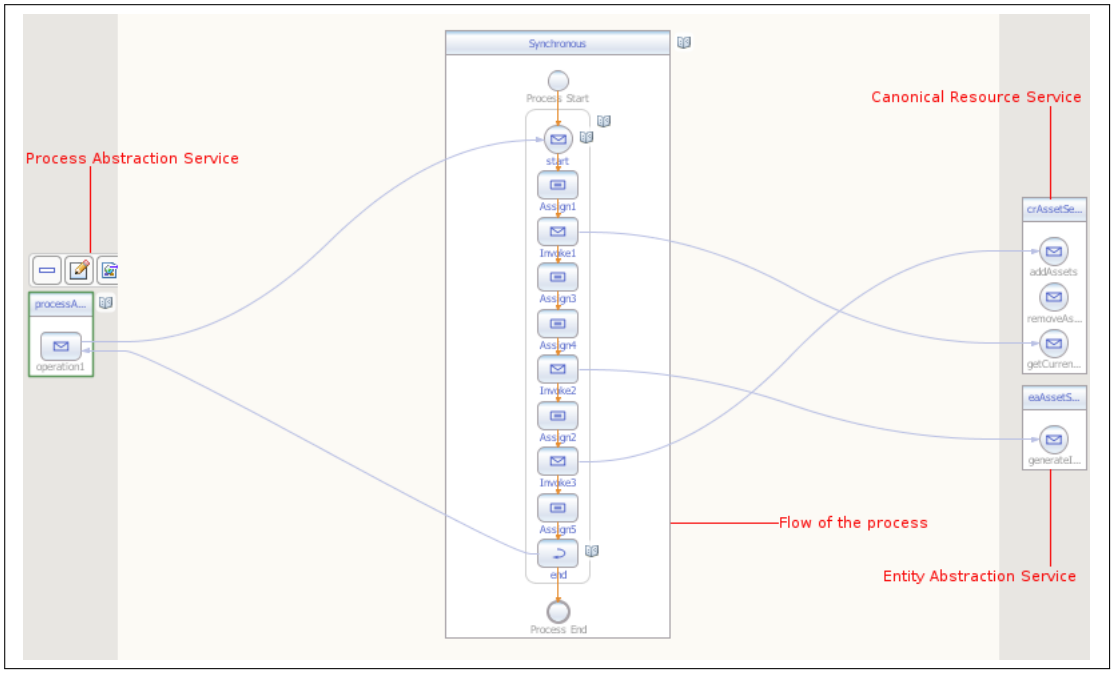

Figure 5.19: Flow of implemented add assets process (BPEL). Flow does not contain UI related tasks, because it is not a matter of BPEL modelling

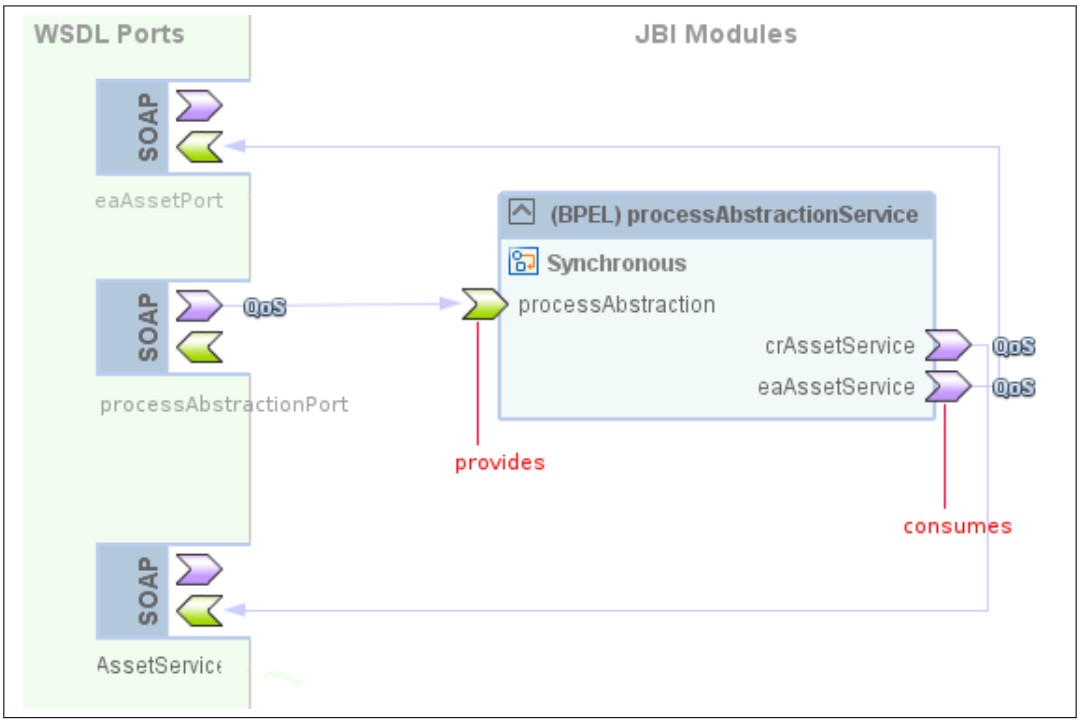

Figure 5.20: Composite Application – a deploy model of add asset process

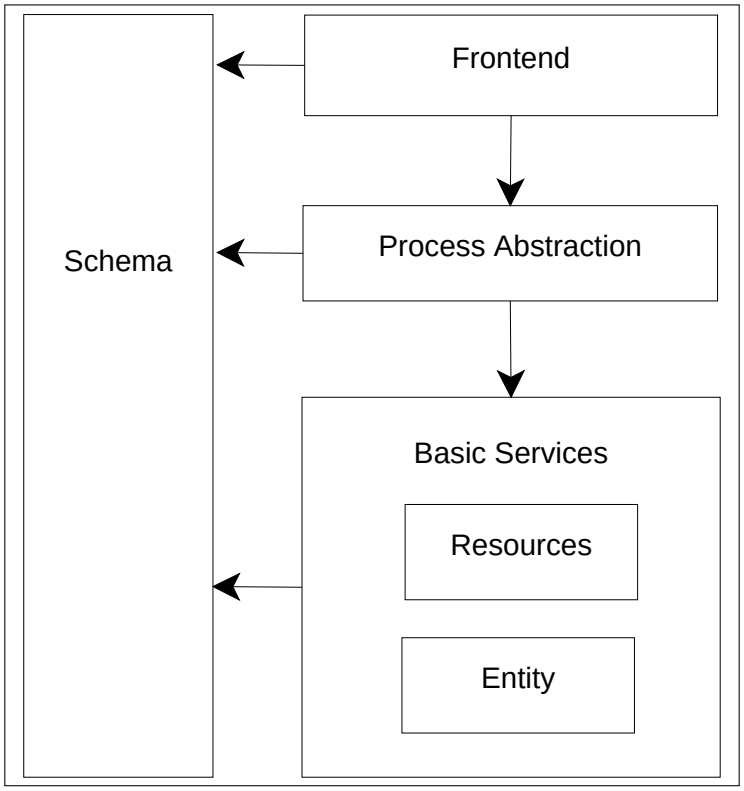

Figure 5.21: Architecture of migrated project

Chapter 6

Conclusion

6.1 Conclusion

This chapter summarises outcomes of the work. The section is broken into three section

1. Answers to Research Questions – provides answers to research questions asked in the first chapter
2. Future Works – addresses works that could derive from outcome of this study
3. Summary – provides summary of contribution brought by the thesis

6.2 Answers to Research Questions

RQ 1 What are the existing techniques of migration toward SOA?

According to Z.Zhang and H.Yang there are three types of migration to SOA [84]: White–Box, Grey–Box and Black–Box. Each type is characterised by different properties. The list below presents example techniques classified to those three types of migration(see chapter number 2).

1. White box–This technique needs deep insight view into existing code and documentation of the migrated system. Example: SMART[49]
2. Gray box–This approach is a mix of white and black box techniques. Application of the technique requires analysis of code of the application and the system with subsystems. Example: Taxonomy Analysis[84].
3. Black box–The technique treats the migrated system as a black box. Migration according this technique bases upon analysis of system's input and output. Example: Wrapping[23]

There are also some attempts of migration toward SOA based on architectural pattern applied in the migrated systems. Some patterns like Pipe–and–Filters are already part of SOA[70]. Other patterns like Client–Server have prototypes of frameworks automating migration to SOA [36]. There are also some attempts of migration of MVC[73][67][28][78], PCBMER[52] or Peer–to–Peer[69].

In addition to those works, a standard (Architectural–Driven Modernisation) of modernisation of systems was identified. The standard considers migration of systems as a form of modernisation. The standard consists of seven substandards. At the time being, only two substandards are defined. The third substandard (issued in 2009) is especially important in context of this work. The standard is named *"Pattern Recognition"* and is meant to identify patterns and anti–patterns in order to identify requirements and opportunities for transformation(see section 2.2.1).

RQ.2 What are drawbacks and advantages of identified techniques of migration toward SOA?

The presented techniques of migration have different approach to migration. Their advantages and drawbacks are consequences of the way how they treat the migrated system. Some of advantages and drawbacks were identified while establishing context of the work (see section 1.1) while others were identified during studying related works (see sections 2.1.2, 2.1.3 and 2.1.5). Here only the main advantages and drawbacks are mentioned. See respective sections for full list of advantages and drawbacks. The sections for particular techniques are mentioned below.

SMART was classified as a White–Box technique. It generates a lot of documentation but it also gives a deep insight view into the system (see 2.1.2 for more information).

Taxonomy Analysis was classified as a Gray–Box technique. The main advantage of the technique is the fact that relationships between services are identified during identification of services. Application of the technique also gives a deep understanding of the system and relationship between its element. The drawback is that the technique is executed both on code and documentation of the system. The problem here is with documentation that may not be maintained and the result of the analysis many be misleading(see 2.1.3 for more information).

Wrapping is the last colour technique. This Black–Box technique is very specific because it does not require analysis of the code of the system. It treats the system as a black box. Unfortunately, application of the technique requires well

elaborated and consequently maintained documentation in order to identify all input–output pairs and use cases to cover(see 2.1.5 for more information).

In addition to those drawback. The presented techniques do not use architectural pattern built into migrated system.

RQ.3 What process employ in order to select the pattern for migration?

There are many architectural pattern. Selection of the pattern for migration needs to be structured and conducted in a way that allows to remove successively patterns that do not fulfil current criteria. The selection of the final pattern is conducted according to following steps:

1. Selection of literature sources
2. Selection of architectural patterns from the literature sources
3. Removal of patterns that exist in only one source
4. Assignment of architectural patterns to selected category
5. Selection of representatives for categories
6. Removal of rarely interacting patterns
7. Prefeasibility study
8. Final Selection

More detailed information about all the steps is presented in chapter 3

RQ.4 Which pattern should be chosen for migration?

Model–View –Controller is the pattern for migration. The pattern was selected because it passed process of selection. MVC was mentioned in revised publication and occurred to be the most generic from within its category. The pattern is also applied with other patterns (see table 3.6). Finally, the pattern was found to be in scope of researches related to migration to SOA (see section 3.4.1)

RQ.5 What elements should be the building blocks of the target architecture?

The target architecture consists of identified SOA architectural pattern. The full list of pattern is presented in table 4.3. Detailed description of the patterns is presented in section 4.6.1

RQ.6 How the target architecture should look like?

The target architecture is a SOA architecture that consist of identified SOA architectural patterns. The architecture is presented in figure 4.4. Motivation behind the structure of architecture is presented in section 4.6.2. The elaborated target architecture divides the system into layers. The layers revolve around interaction with users, applied business processes and basic services. Additionally the architecture presents schemas and policies as parts of the system that should be accessible by services from all layers of the target architecture.

RQ.7 How to translate the selected architectural pattern into the target architecture?

Following steps are proposed for migration from MVC to SOA.

1. Convert MVC into layers
2. Choose main communication protocol
3. Unify used schemas
4. Unify policies
5. Identify and wrap into services all coarse grained utilities
6. Identify and encapsulate access to any external resource
7. Identify and wrap into services all entity related code
8. Identify and wrap into services business rules
9. Provide inventory endpoints to basic services
10. Identify all business processes within the legacy application
11. Identify all statefull services and decide if their state can be deferred
12. Identify current points of access to the migrated system
13. Identify all the places in user interface where a continuous feedback from application to end user is provided

What is important here, each step of migration introduces into the architecture one SOA architectural pattern. Detailed description of guidelines is presented in section 5.2.1. Section 5.4 presents example application of the guidelines.

RQ.8 What are the drawback and advantages of the new technique ?

Information about advantages and drawbacks is presented in section 5.5.3. The most important advantage is that the guidelines consider architectural patterns applied in migrated architecture. The most significant drawback is limitation of application. The guidelines are applicable only for systems that do not use functions provided by MVC supporting frameworks.

The elaborated guidelines are implementation of White–Box technique. The most important advantage of the guidelines is solution for the drawback that that exist in identified White–Box, Grey–Box and Black–Box techniques, namely *"The technique does not consider architectural patterns that are applied in architecture of migrated systems"*.

6.3 Future works

The example system that was used for illustrating application of the guidelines was small. This was caused by problem of availability of candidate projects. The elaborated guidelines should be applied for migration of other systems in order to validate them and propose improvements.

The elaborated guidelines provide a migration technique only for MVC architectural pattern, but the target architecture can be classified as Process–Enabled SOA (see 5.5.2). This architecture could also be applicable for other migrations. Future researches could modify the guidelines in order to adjust them other architectural patterns. At the beginning for other patterns belonging to the same category as MVC (see table 3.3 for categories and patterns).

Finally, elaborated technique of migration is a list of subsequent steps that introduce additional elements into architecture. The technique is systematic and it known how the architecture changes after each guideline. This property can be used in future in order to implement a framework supporting the migration.

6.4 Summary

The contribution the thesis is the list of guidelines that are applicable for migration of systems based on MVC architectural pattern toward Service Oriented Architecture. The target architecture and the migrated architecture are based on architectural pattern. Pattern encapsulate proven solutions, thus they increase quality of architecture. Additionally, the target architecture, the next outcome of the thesis is expressed as a pattern language that can be used to describe also other target systems.

References

[1] http://www.soa-manifesto.org/.

[2] Stefan Link Fabian Jakobs Ludwig Neer Sebastian Abeck. Architecture of and migration to soas presentation layer. Technical report, Cooperation and Management, Universitt Karlsruhe (TH), 76128 Karlsruhe 2 CAS GmbH, 76131 Karlsruhe, 2006.

[3] Mohammed A. Aboulsamh. Towards a model-driven approach for planning a standard-based migration of enterprise applications to soa. In *SERVICES '09: Proceedings of the 2009 Congress on Services - I*, pages 471–474, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Mark S. Aldenderfer and Roger K. Blashfield. *Cluster Analysis*. Sage University Paper Series on Quantitative Applications in the Social Sciences. Sage Publications Inc., Beverly Hills, 1984.

[5] Firstservis Pty Ltd. Alexander Roussekov, Senior SOA Architect. Evaluation of soa vendors a technical white paper from firstservis. Technical White Paper.

[6] Lianjun An and Jun-Jang Jeng. Business-driven soa solution development. In *e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on*, pages 439 –444, 24-26 2007.

[7] Francesca Arcelli, Christian Tosi, and Marco Zanoni. Can design pattern detection be useful for legacy systemmigration towards soa? In *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 63–68, New York, NY, USA, 2008. ACM.

[8] P. Avgeriou and U. Zdun. Architectural patterns revisited - a pattern language. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, Irsee, Germany, July 2005.

[9] Paris Avgeriou, Patricia Lago, and Philippe Kruchten. Towards using architectural knowledge. *SIGSOFT Softw. Eng. Notes*, 34(2):27–30, 2009.

[10] Sriram Balasubramaniam, Grace A. Lewis, Ed Morris, Soumya Simanta, and Dennis Smith. Smart: Application of a method for migration of legacy systems to soa environments. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 678–690, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Michael Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. Wiley Publishing, 2008.

[12] A.K. Bhattacharjee and R.K. Shyamasundar. Choreography = orchestration with scripts + conversations. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 824 –827, 23-26 2008.

[13] Norbert Bieberstein, Sanjay Bose, Marc Fiammante, Keith Jones, and Rawn Shah. *Service-Oriented Architecture Compass: Business Value, Planning, and Enterprise Roadmap*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[14] Kevin Bierhoff, Mark Grechanik, and Edy S. Liongosari. Architectural mismatch in service-oriented architectures. In *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*, page 4, Washington, DC, USA, 2007. IEEE Computer Society.

[15] bitpipe. http://www.bitpipe.com/olist/SOA.html.

[16] Grady Boch. Handbook of software architecture: Gallery.

[17] Scott Bollig and Dan Xiao. Throwing off the shackles of a legacy system. *Computer*, 31:104–106,109, 1998.

[18] Hongyu Pei Breivold, Stig Larsson, and Rikard Land. Migrating industrial systems towards software product lines: Experiences and observations through case studies. In *SEAA '08: Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 232–239, Washington, DC, USA, 2008. IEEE Computer Society.

[19] Paul Brown. *Implementing soa: total architecture in practice*. Addison-Wesley Professional, 2008.

[20] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, May 2007.

[21] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. Wiley, June 2007.

[22] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition, August 1996.

[23] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *J. Syst. Softw.*, 81(4):463–480, 2008.

[24] Robert J. Cloutier and Dinesh Verma. Applying the concept of patterns to systems architecture. *Syst. Eng.*, 10(2):138–154, 2007.

[25] Fred A. Cummins. *Building the Agile Enterprise: With SOA, BPM and MBM*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[26] P. De, P. Chodhury, and S. Choudhury. A framework for performance analysis of client/server based soa and p2p soa. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 79 –83, 2010.

[27] Andrea De Lucia, Rita Francese, Giuseppe Scanniello, Genoveffa Tortora, and Nicola Vitiello. A strategy and an eclipse based environment for the migration of legacy systems to multi-tier web-based architectures. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 438–447, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Suyin Dong, Min Huang, Ying Liu, Jingyang Wang, and Xiaohong Wang. Design of competitive intelligence consciousness and skill cultivating platform for undergraduate students based on soa and mvc. In *Database Technology and Applications (DBTA), 2010 2nd International Workshop on*, pages 1 –4, 2010.

[29] J. Dowling, J. Sacha, and S. Haridi. Improving ice service selection in a p2p system using the gradient topology. In *Self-Adaptive and Self-Organizing Systems, 2007. SASO '07. First International Conference on*, pages 285 – 288, 9-11 2007.

[30] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pœl Krogdahl, Min Luo, and Tony Newling. *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, July 2004.

[31] Thomas Erl. *Soa: principles of service design*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[32] Thomas Erl. *SOA Design Patterns*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[33] Martin Fowler. http://www.martinfowler.com/bliki/ServiceOrientedAmbiguity.html, 01 2005.

[34] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[36] He Guo, Chunyan Guo, Feng Chen, and Hongji Yang. Wrapping client-server application to web services for internet computing. In *Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on*, pages 366 – 370, 2005.

[37] Robert S. Hanmer and Kristin F. Kocan. Documenting architectures with patterns. *Bell Labs Technical Journal*, 9(1):143–163, 2004.

[38] Neil B. Harrison and Paris Avgeriou. Analysis of architecture pattern usage in legacy system architecture documentation. In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 147–156, Washington, DC, USA, 2008. IEEE Computer Society.

[39] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Professional, November 1999.

[40] http://jsexton0.blogspot.com/2009/01/openesb-bpel-assignment tips.html.

[41] http://openesb-users.794670.n2.nabble.com/Use-of-xsd-anyType-in-BPEL-assign-operations td1514431.html.

[42] S. Jones. Toward an acceptable definition of service [service-oriented architecture]. *Software, IEEE*, 22(3):87 – 93, may-june 2005.

[43] Ahmad Waqas Kamal, Paris Avgeriou, and Uwe Zdun. Modeling architectural pattern variants. In Till Schmmer, editor, *EuroPLoP*, volume 610 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[44] Ramarao Kanneganti and Prasad Chodavarapu. *Soa security*. Manning Publications Co., Greenwich, CT, USA, 2008.

[45] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. Wiley, June 2004.

[46] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[47] G.A. Lewis, E. Morris, S. Simanta, and L. Wrage. Common misconceptions about service-oriented architecture. In *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2007. ICCBSS '07. Sixth International IEEE Conference on*, pages 123 –130, feb. 2007.

[48] Grace Lewis, Edwin Morris, and Dennis Smith. Analyzing the reuse potential of migrating legacy components to a service-oriented architecture. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 15–23, Washington, DC, USA, 2006. IEEE Computer Society.

[49] Grace Lewis, Edwin Morris, Dennis Smith, and Liam O'Brien. Service-oriented migration and reuse technique (smart). In *STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 222–229, Washington, DC, USA, 2005. IEEE Computer Society.

[50] Quanhao Lin, Ruonan Rao, and Minglu Li. Dwsdm: A web services discovery mechanism based on a distributed hash table. In *Grid and Cooperative Computing Workshops, 2006. GCCW '06. Fifth International Conference on*, pages 176 –180, oct. 2006.

[51] Giuseppe A. Di Lucca, Anna Rita Fasolino, Patrizia Guerra, and Silvia Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 122–129, Washington, DC, USA, 1997. IEEE Computer Society.

[52] Leszek Maciaszek. *Adaptive Integration of Enterprise and B2B Applications*, volume 10 of *Communications in Computer and Information Science*, chapter 1. Springer Berlin Heidelberg, 2008.

[53] Leszek A. Maciaszek. Modeling and engineering adaptive complex systems. In *ER '07: Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling*, pages 31–38, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[54] C. Mauro, J.M. Leimeister, and H. Krcmar. Service oriented device integration - an analysis of soa design patterns. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1 –10, 5-8 2010.

[55] Joe McKendrick. http://www.soamag.com/I34/1109-1.php, 12 209.

[56] F. Medina-Dominguez, M.-I. Sanchez-Segura, A. Mora-Soto, and A. Amescua. Patterns in the field of software engineering. pages 248 – 53, Piscataway, NJ, USA, 2009//. software engineering products;product patterns;multimodel software development;Wiki;knowledge repository;software reuse;.

[57] J.F. Naveda, M.J. Lutz, T.B. Hillburn, L.M. Northrup, and M.C. Stinson. The role of software engineering in undergraduate education. volume vol.2, pages 750 vol.2 –, Champaign, IL, USA, 1997//. software engineering;undergraduate education;computer science;Software Engineering Institute;Association for Computing Machinery;IEEE Computer Society;elective noncredit paracticums;existing courses;required course sequence;elective course;.

[58] P. Newcomb. Architecture-driven modernization (adm). In *Reverse Engineering, 12th Working Conference on*, pages 237 – 237, 7-11 2005.

[59] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services (Independent Technology Guides)*. Addison-Wesley Professional, 2004.

[60] Tinny Ng, Jane Fung, Laura Chan, and Vivian Mak. *Understanding IBM SOA Foundation Suite: Learning Visually with Examples*. IBM Press, 2009.

[61] L. O'Brien, D. Smith, and G. Lewis. Supporting migration to services using software architecture reconstruction. In *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pages 81 –91, 0-0 2005.

[62] P. Offermann, M. Hoffmann, and U. Bub. Benefits of soa: Evaluation of an implemented scenario against alternative architectures. pages 352 –359, sept. 2009.

[63] M.R. Olsem. Enabling technology for migrating legacy systems to client-server systems. *Software Maintenance, IEEE International Conference on*, 0:304, 1997.

[64] OMG. http://adm.omg.org, 2 2009.

[65] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, and A. Inkeri Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pages 325–332, Beijing, China, 2000.

[66] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46 – 52, oct. 2003.

[67] Xiaohong Qiu. Building desktop applications with web services in a message-based mvc paradigm. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 765 – 768, 6-9 2004.

[68] Michael Rosen, Boris Lublinsky, Kevin T. Smith, and Marc J. Balcer. *Applied SOA: Service-Oriented Architecture and Design Strategies*. Wiley Publishing, 2008.

[69] J. Sacha, B. Biskupski, D. Dahlem, R. Cunningham, J. Dowling, and R. Meier. A service-oriented peer-to-peer architecture for a digital ecosystem. In *Digital EcoSystems and Technologies Conference, 2007. DEST '07. Inaugural IEEE-IES*, pages 205 –210, 21-23 2007.

[70] T. Scheibler, F. Leymann, and D. Roller. Executing pipes-and-filters with workflows. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 143 –148, May 2010.

[71] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 1 edition, September 2000.

[72] Miguel A. Serrano. Evolutionary migration of legacy systems to an object-based distributed environment. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 86, Washington, DC, USA, 1999. IEEE Computer Society.

[73] Liu Shan and Liu Shi. The design and implementation of art examination management system based on soa. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pages 1 –3, 2009.

[74] Paulo Pinheiro Da Silva, Alberto H. F. Laender, Rodolfo S F. Resende, and Paulo B. Golgher. Capples - a capacity planning and performance analysis method for the migration of legacy systems, 1999.

[75] Dennis Smith. Migration of legacy assets to service-oriented architecture environments. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 174–175, Washington, DC, USA, 2007. IEEE Computer Society.

[76] Harry M. Sneed. Integrating legacy software into a service oriented architecture. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 3–14, Washington, DC, USA, 2006. IEEE Computer Society.

[77] Harry M. Sneed. Wrapping legacy software for reuse in a soa. In F. Lehner, H. Nösekabel, and P. Kleinschmidt, editors, *Multikonferenz Wirtschaftsinformatik 2006*, volume 2 of *XML4BPM Track*, pages 345–360. GITO-Verlag Berlin, 2006.

[78] Wei-Tek Tsai, Qian Huang, J. Elston, and Yinong Chen. Service-oriented user interface modeling and composition. In *e-Business Engineering, 2008. ICEBE '08. IEEE International Conference on*, pages 21 –28, 2008.

[79] William Ulrich Vitaly Khusidman. Architecture-driven modernization: Transforming the enterprise draft v.5. white paper.

[80] Ken Vollmer. The forrester wave integration-centric business process management suites, q4 2008, 10 2008.

[81] Tsai Wei-Tek, Huang Qian, Elston Jay, and Chen Yinong. Service-oriented user interface modeling and composition. In *ICEBE '08: Proceedings of the 2008 IEEE International Conference on e-Business Engineering*, pages 21–28, Washington, DC, USA, 2008. IEEE Computer Society.

[82] Dan Woods and Thomas Mattern. *Enterprise SOA: Designing IT for Business Innovation*. O'Reilly Media, Inc., 2006.

[83] Liang-Jie Zhang and Jia Zhang. Componentization of business process layer in the soa reference architecture. In *Services Computing, 2009. SCC '09. IEEE International Conference on*, pages 316 –323, 21-25 2009.

[84] Z. Zhang and H. Yang. Incubating services in legacy systems for architectural migration. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 196 – 203, nov.-3 dec. 2004.