

Marko Malinen

**Using EBI Pattern in Conjunction with Service-Oriented
Architectures**

Master's Thesis in Information Technology

February 28, 2013

University of Jyväskylä

Department of Mathematical Information Technology

Author: Marko Malinen

Contact information: marko.j.malinen@jyu.fi

Supervisor: Tuukka Puranen and Jonne Itkonen

Title: Using EBI Pattern in Conjunction with Service-Oriented Architectures

Työn nimi: EBI-mallin käyttäminen palvelukeskeisten ohjelmistoarkkitehtuurien kanssa

Project: Master's Thesis

Study line: Software Engineering

Page count: 89+10

Abstract: This thesis presents a service-oriented architecture prototype developed for route optimization system. The architecture of the prototype is based on Entity-Boundary-Interactor (EBI) pattern. The focus of this thesis is to evaluate the quality of the architecture resulting from the combination of EBI and SOA patterns. The quality is evaluated with respect to requirements of a route optimization system. The evaluation of the prototype architecture is performed by using proven software architecture evaluation methods. The architecture of the prototype fulfilled its objectives.

Keywords: EBI, SAAM, SOA, software architecture, software quality

Suomenkielinen tiivistelmä: Tässä tutkielmassa esitellään reitinoimintijärjestelmää varten toteutettu prototyyppi palvelukeskeisestä arkkitehtuurista. Arkkitehtuuri pohjautuu Entity-Boundary-Interactor-malliin (EBI). Työssä tutkitaan, pystytäänkö EBI-mallia ja palvelukeskeisen arkkitehtuurin periaatteita noudattaen toteuttamaan projektin tavoitteet täyttävä järjestelmä ja pystytäänkö syntynyttä prototyypin arkkitehtuuria soveltamaan reitinoimintijärjestelmän muuntamisessa palvelukeskeiseksi kokonaisuudeksi. Prototyypin arvioinnissa käytetään hyväksi todettuja ohjelmistoarkkitehtuurien arviointimenetelmiä. Prototyypin arkkitehtuuri täytti sille asetetut tavoitteet.

Avainsanat: EBI, SAAM, SOA, ohjelmistoarkkitehtuuri, ohjelmistojen laatu

Glossary

ADL	Architecture description language.
API	Application programming interface.
ATAM	Architecture trade-off analysis method.
ATDD	Acceptance test driven development.
CBAM	Cost-benefit analysis method.
CLR	The common language runtime.
CRUD	Create, read, update and delete.
DIP	Dependency inversion principle.
EBI	Entity-Boundary-Interactor model.
ERP	Enterprise resource planning.
FTP	File transfer protocol.
HTTP	Hypertext transfer protocol.
JSON	Javascript Object Notation.
MoMA	The Mono migration analyzer.
Mono	An open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime.
NUnit	Unit test framework.
OO	Object-oriented. A programming paradigm.
REST	Representational State Transfer.
SAAM	Software architecture analysis method.
SMTP	Simple mail transfer protocol.
SOA	Service-Oriented Architecture.
SOAP	Simple Object Access Protocol.
TDD	Test driven development.
URI	Uniform resource identifier.
URL	Uniform resource locator.
Use Case	A list of steps that describes interaction between roles to achieve a goal.

VRP	Vehicle routing problem.
WSDL	Wed services description language.
XML	Extensible Markup Language.

List of Figures

Figure 1. One vision of the Waterfall model.	7
Figure 2. Spiral model	8
Figure 3. Service locator pattern	11
Figure 4. Data Transfer Object and Assembler pattern	12
Figure 5. Dependency injection pattern	13
Figure 6. Activities and dependencies in scenario-based analysis	24
Figure 7. The Clean Architecture	28
Figure 8. Entity-Boundary-Interactor	31
Figure 9. Entity-Boundary-Interactor Module Thinking	33
Figure 10. Entity-Boundary-Interactor Request	34
Figure 11. Entity-Boundary-Interactor Response	35
Figure 12. A simplistic illustration of SOA	37
Figure 13. Development process	47
Figure 14. The implemented Data Transfer Object and Assembler pattern	58
Figure 15. Relations Between Entities	58
Figure 16. Retrieve vehicles and tasks related to the vehicle	59
Figure 17. Representation of the prototype using a simple dependency notation	63
Figure 18. The architecture before and after the separation of the services	66
Figure 19. Architectural design error	68
Figure 20. MoMA run against the Mono 2.8 (4.0 profile)	90

List of Tables

Table 1. Common design patterns.....	10
Table 2. The Properties of The Evaluation Approaches	23
Table 3. Examples HTTP method usage in REST-based application.	43
Table 4. Results of the scenario evaluation.	65
Table 5. Scenario interactions.	68

Contents

1	INTRODUCTION	1
1.1	Background	1
1.2	Research Problem and Thesis Structure	2
2	SOFTWARE DEVELOPMENT	4
2.1	General	4
2.2	Software Development Life-Cycle	5
2.3	Development Models	6
2.4	Acceptance Test Driven Development	8
2.5	Design Patterns	9
2.6	Design Pattern Examples	11
3	SOFTWARE ARCHITECTURE AND QUALITY	14
3.1	General	15
3.2	History	16
3.3	Architectural Pattern	17
3.4	Quality Attributes	18
3.5	Evaluation	19
3.6	Evaluation Methods and Techniques	21
3.7	Scenario Based Evaluation Methods	23
3.7.1	SAAM	23
3.7.2	ATAM	25
3.7.3	CBAM	26
4	EBI AND SOA PATTERNS	27
4.1	Background	27
4.2	Principles of EBI	30
4.3	Quality Attributes of EBI	32
4.4	Implementation of EBI	33
4.5	Principles of SOA	36
4.6	Quality Attributes of SOA	39
4.7	Implementation of SOA	41
5	IMPLEMENTATION	45
5.1	Objectives and Requirements	45
5.2	Development Methods and Process	46
5.2.1	Terminology	46
5.2.2	Development Cycle	47
5.3	Main Functionality	48
5.3.1	Creating and Updating Data	48
5.3.2	Retreiving and Deleting Data	49
5.4	User Stories and Components	49
5.4.1	User Stories	49

5.4.2	Interactors and Boundaries	51
5.4.3	Entities, Data Transfer Objects and Data Objects	52
5.4.4	User Interface and InMemoryDb	53
5.5	Issues.....	54
5.5.1	Updating UI.....	54
5.5.2	Initialization of Interactors	55
5.5.3	Messaging Model	57
5.5.4	Relations in DTOs	58
6	EVALUATION.....	60
6.1	Selected Evaluation Method	60
6.2	Evaluation With SAAM.....	62
6.2.1	Candidate Architecture	62
6.2.2	Scenarios	62
6.2.3	Evaluation of the Scenarios	64
6.2.4	Scenario Interactions	67
6.2.5	Weights of Scenarios and Interactions	69
6.3	Evaluation of Service-Oriented	69
6.4	Results and Observations	70
7	CONCLUSION AND FURTHER RESEARCH	72
	BIBLIOGRAPHY	74
	APPENDICES.....	83
A	Simple Architectural Dependency Notation	83
B	Design Pattern Implementation Examples	84
B.1	Service Locator Pattern.....	84
B.2	Dependency Injection Pattern	85
C	MoMA Scan Results	88
D	Issue Descriptions	88
D.1	Results.....	90
D.2	Microsoft.Practices.Unity.dll	91
D.3	Microsoft.Practices.Unity.Interception.dll.....	92

1 Introduction

I think that it's extraordinarily important that we in computer science keep fun in computing.

–Alan Perlis

Service-oriented architecture (SOA) is an architectural pattern that gives guidelines to design a software in a way that existing services are set available for service consumers. Entity-boundary-interactor (EBI) is an architectural pattern that separates the functionality of the software from so called implementation details such as the user interface and the database. This thesis examined software architectures, service-oriented architecture pattern and implements a prototype of route optimization services using design principles and the guidelines of the SOA pattern and the Entity-Boundary-Interactor architectural pattern. The goal is to evaluate the quality of the architecture resulting from the combination of EBI and SOA patterns. The quality is evaluated with respect to requirements of a route optimization system.

1.1 Background

Vehicle routing problem (VRP) is a combinatorial optimization problem. The VRP is concerned with the determination of the optimal routes used by a fleet of vehicles, based on one or more depots, to serve a set of customers (Toth and Vigo 2002). Routing problems are used, for example, in transportation companies to design efficient logistics routes. Due to differences in the operations of these companies, they can have different kinds of VRPs that needs to be solved.

Instead of focusing to a one kind of VRP, a project at the Department of Mathematical Information Technology at the University of Jyväskylä has been developing a computation core that can solve several different VRPs. The objective of the project is to develop a route planning service, which computes efficient routes for the fleet of vehicles. The customers can access the service via a web-based interface and by the ERP systems. The project has a need for an software architecture that can provide the functionality of the optimization system, as services. The motivation behind the service-based approach is that the functionalities can be

accessed without a heavy integration process.

One potential architecture for the system is the combination of service-oriented architecture (SOA) pattern and entity-boundary-interactor (EBI) pattern. SOA provides constraints and guidelines on how the services should be orchestrated whereas the EBI provides guidelines and constraints on how the system should be splitted in to modules in order to keep it maintainable and testable.

1.2 Research Problem and Thesis Structure

An service-oriented implementation of vehicle routing problem solver software is introduced in this thesis. This is done by the following principles of service-oriented architecture pattern and applying an architectural pattern called the Entity-Boundary-Interactor as top level architecture. The goal is to evaluate the quality of the architecture resulting from the combination of EBI and SOA patterns. The quality is evaluated with respect to requirements of a route optimization system.

The original vehicle routing problem solver software is a quite large construct so most of its functionalities are left off from the construct of this thesis. The prototype developed for this thesis provides only the basic data manipulation funtionalities. These functionalities include creating new data, reading and updating existing data and deleting data. One of the key elements of the construct is that it provides a well designed API that can be made available for ERP and it can be used via WWW interface.

In other words, a limited service-oriented implementation of existing vehicle routing problem solver software is introduced in this thesis. The evaluation of the construct is based on existing software architecture evaluation methods. Beside the construct this thesis also provides an answer to whether or not the presented architectural pattern should be applied to the all functionalities of the vehicle routing problem solver software.

The thesis is structured as follows. The basics of software development process and methods for building a software product are introduced in Chapter 2. In Chapter 3, theory, concept and the history of software architectures are presented. The focus is on the architectural patterns

and on the design patterns. Methods for the evaluation are also presented. The Entity-Boundary-Interactor architectural pattern and the service-oriented architecture pattern are presented in Chapter 4. In Chapter 5 the construct of this thesis is presented. The evaluation of the construct is presented in Chapter 6. The conclusion is presented in Chapter 7.

2 Software Development

Computer Science is the first engineering discipline in which the complexity of the objects created is limited solely by the skill of the creator, and not by the strength of raw materials.

–Brian Reid

In this chapter, the key aspects and concepts of the software development process and software engineering are presented briefly. The definition of the software engineering and aspects of it are described in Section 2.1. The different phases and activities of software development process are presented in Section 2.2. In Section 2.3, the key characteristics of commonly used development models are described. An agile development practice called acceptance test driven development is presented in Section 2.4. In Section 2.5, the concept of design patterns is presented. Some useful design patterns are described in detail in Section 2.6.

2.1 General

Software engineering is a discipline for professional and systematic software development rather than individual programming (Sommerville 2010). According to Sommerville (Sommerville 2010) professional software development is an activity where software is developed for specific business purposes, for inclusion in other devices, or as software products. Professional software, intended for use by someone apart from its developer, is usually developed by teams rather than individuals. It is maintained and changed throughout its life. The software engineering is defined as follows (Sommerville 2010):

Software engineering is an engineering discipline that is concerned with all aspects of software production.

It includes aspects such as *specification*, *development*, *validation*, and *evolution*. The specification includes activities such as defining the software to be produced and recognizing the constraints on its operations. The development is concerned of the designing and implementing the software. The validation is concerned of making sure that the produced software fulfills the requirements. The evolution is concerned of modification of the software in order to satisfy the requirements in a changing environment.

2.2 Software Development Life-Cycle

A *software development process* or *software development life-cycle* is a concept of how to develop a software product. The process of developing a software consists of phases such as *requirements analysis*, *design*, *implementation*, *validation*, and *evolution* or *maintainance*. **Requirement** analysis (or specification) is one of the key phases in software development life-cycle. The purpose of the requirement analysis is to identify the requirements and constraints for the software (Sommerville 2010). The requirement analysis can be challenging because software development is a dynamic process, and can therefore cause changes to the requirements while the development is still in progress (Nurmuliani, Zowghi, and Powell 2004). This phenomena is called *requirement volatility* (Curtis, Krasner, and Iscoe 1988).

Software design is a process that is usually made by using the results of requirement analysis. It is often described as a *problem-solving activity* (Curtis, Krasner, and Iscoe 1988). The purpose of the design is to describe the software to be implemented. It describes things such as the data models and architectures used by the system (Sommerville 2010). Other things involved in software design, are the design of the interfaces between system components and in some cases, algorithms. The design process can include multiple iterations before the final design is achieved.

Implementation is an activity that realizes requirements according to design (Sommerville 2010). The essential part of the implementation is programming. It also includes design activity. Depending on the used process model, implementation and design may include refinements of the specification and requirements.

Validation is an activity that verifies the product of implementation accoring to the accep-

tance criterias. Program testing is the principal validation technique. Testing can involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development.

The software development process is often divided into two parts: the development and the maintainance. The **maintainance** usually consists of making changes to the software product, such as integrating new features or fixing bugs. As the world changes, so may change the requirements of the software. This is why the software engineering should be considered as an *evolutionary* process where software is continually changed over its lifetime in response to changing requirements and customer needs (Sommerville 2010).

2.3 Development Models

Software development models are guidelines on how the software development process can be done efficiently. There are different models such as *waterfall model*, *spiral model*. Nowadays agile development methods have become popular.

Waterfall model is a software development model that comes in many variations. Two common characteristics that describe it are that *all planning is oriented towards a single delivery date*, and that *all analysis and design are done in detail, before coding and testing* (Gilb 1985). The process, so to speak, flows downward, like a cascading waterfall. This is illustrated in Figure 1. This simplified version has been criticized as an unrealistic (Gilb 1985). In 1970, Royce described this model in his article article¹ (Royce 1970) and pointed out its flaws. In the same article, he proposed enhancements to the model such as *requiring the program design to come first, creating profound documentation, doing the software twice, monitoring, planning and controlling tests, and involving the customer.*

Spiral model is an iterative and prototype-based software development model. It was defined in 1988 by Barry Boehm (Boehm 1986). A typical cycle can be divided into four phases:

1. Determine objectives, alternatives and constraints.
2. Evaluate alternatives, identify and resolve risks.

1. It should be noted that Royce did not use the term “waterfall” in his article.

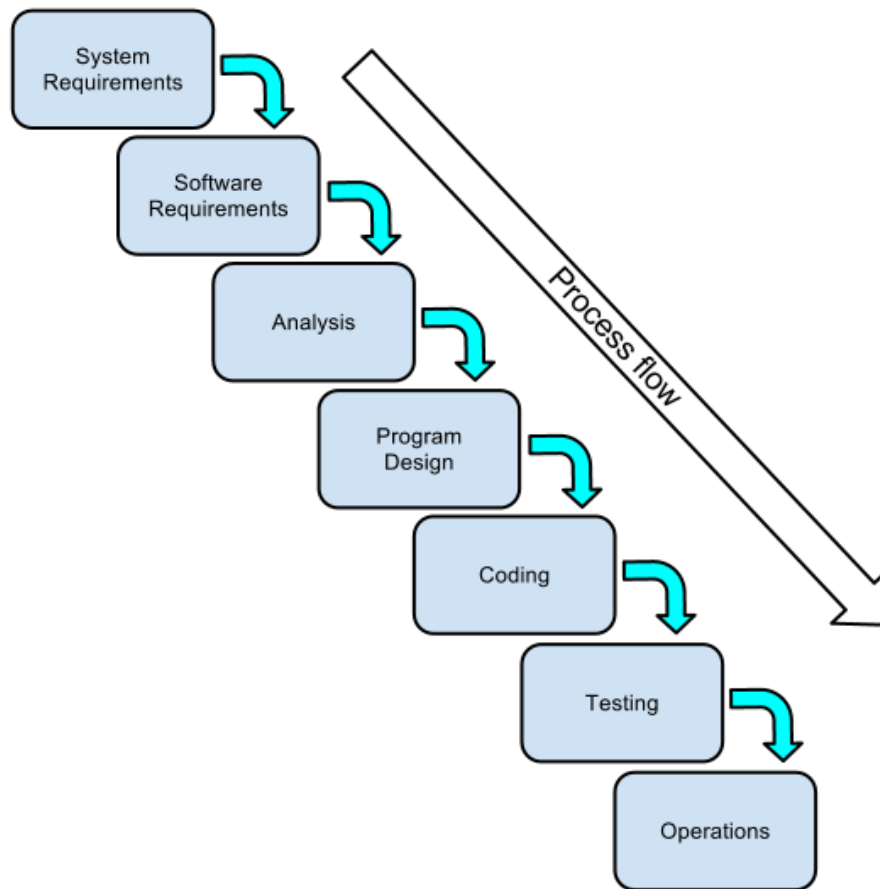


Figure 1. One vision of the Waterfall model.

3. Develop and verify next-level product.
4. Plan the phases.

The product of the cycle is an extended version of the product developed in the previous cycle. One of the most important aspects in spiral model is the risks management. Identification of technical and managerial risks, and measures to reduce them helps to keep the software development process under control. The spiral model is illustrated in Figure 2.

Agile development is a family of software development processes that share certain principles. These principles include constant communication between the developers and other stakeholders, flexibility and ability to react to changing requirements, delivering working software frequently, choosing the right people for the job, high quality design and technical excellence (*Principles Behind the Agile Manifesto* 2001; Martin 2003). The Agile Mani-

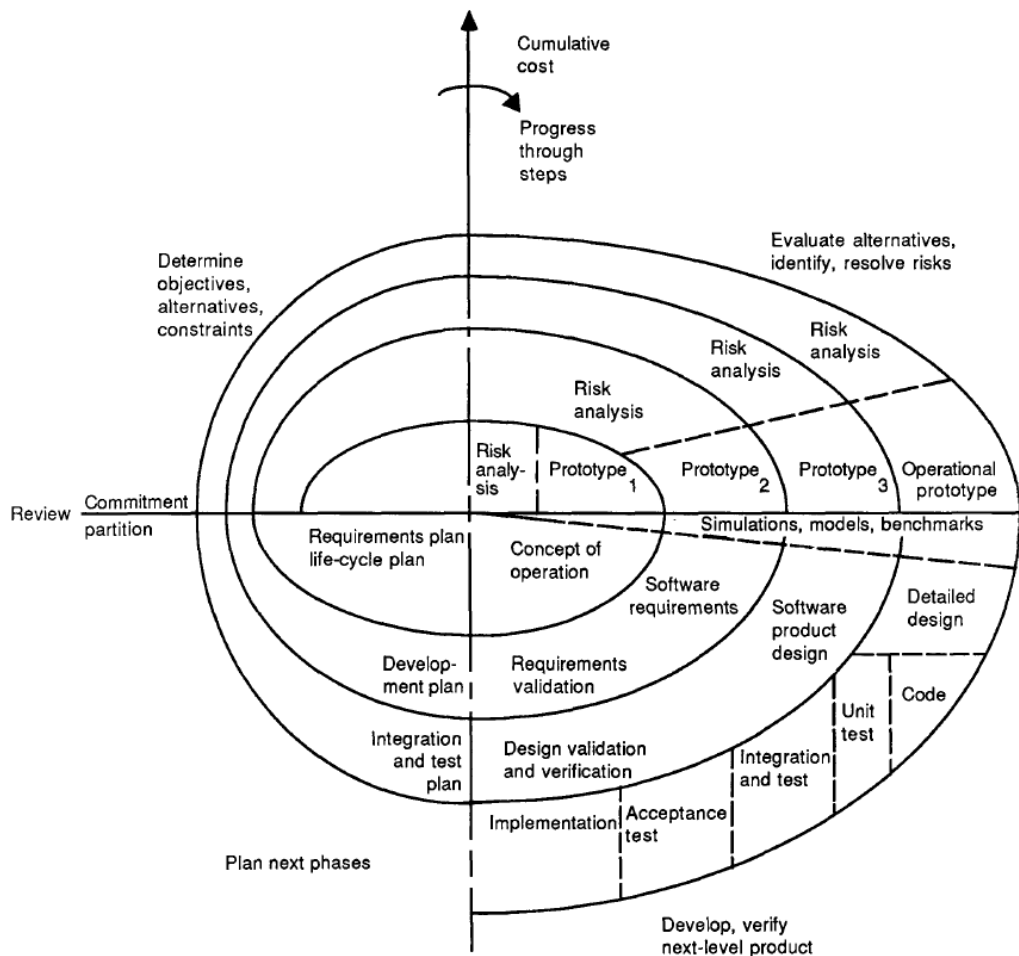


Figure 2. Spiral model (Boehm 1986).

festo, published in 2001, presented four essential values of agile development: *individuals and interaction, working software, customer collaboration and responding to change* (Manifesto for Agile Software Development 2001). There are several process models, for example *XP* and *scrum*, that are that are considered to be agile.

2.4 Acceptance Test Driven Development

Acceptance test driven development (ATDD) (Hendrickson 2008; Rantanen 2012; Alleman 2008) is a development method where the customer and the development team together define the user stories and requirements. This phase is called *specification workshop* and it is the most important phase in the ATDD. Its purpose is to get answers to questions like

“why?”, “when?” and “who?” (Rantanen 2012). After that, the functionality is specified in more detail. A set of *acceptance tests* are created as a product of this activity. An acceptance test describes when a particular functionality can be defined as done by specifying the behavior and functionality that the system should fulfill (Alleman 2008).

Once the requirements are gathered and agreed on, the development team implements automated acceptance tests and starts implementing the functionalities. Whenever there occurs a change in the feature implementation, all the tests are run in order to find out the current status of the system. If new information is discovered during the process, new acceptance tests are added according to the new information. Once all the tests have passed and the required functionality is implemented, the results are demonstrated to stakeholders to make sure that the right things were achieved (Hendrickson 2008; Rantanen 2012; Alleman 2008).

The ATDD fits well within the agile development methods. It can be done in iterations that consists of four phases: *discuss the requirements, create acceptance tests according to requirements, implement functionality so it satisfies acceptance tests and deliver the results* (Hendrickson 2008; Rantanen 2012; Alleman 2008). In order to harness full potential of the ATDD, it requires the right people to participate. The participants should represent all relevant roles of the project so that as many perspectives as possible are considered.

2.5 Design Patterns

Designing a software can be difficult, especially if the goal is a reusable and flexible design. Design patterns are one tool to help the designer to choose the right design from many alternatives.

Design patterns are proven design experiences and best practices for solving similar problems across domains (Nerur and Balijepally 2007). They make it easier to reuse successful designs and architectures. Inspiration for using design patterns came from architecture. Cristopher Alexander says following about the patterns used in architecture (Alexander, Ishikawa, and Silverstein 1977):

Each pattern describes a problem which occurs over and over again in our

environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a millions times over, without ever doing it the same way twice.

The same applies to design patterns in software engineering.

In the book (Gamma et al. 1995) Gamma et al. describe structure of a design pattern in software engineering. It consists of twelve sections including the *pattern name*, description of the *problem*, description of the *solution* and the description of the *consequences* and *trade-offs* of using specified pattern. Design patterns are not classes in libraries nor are they complete solutions for implementing software systems. Each design pattern has a particular type of a problem that it is designed to be used for. Condition, when the pattern is useful, is described in the problem section. The solution describes the collaboration, responsibilities and the roles of the classes and interfaces that are used in resolving the problem. The consequence presents the results and trade-offs of the design pattern. The solution is presented in an abstract level, nothing concrete is described.

Categories		
Creational	Structural	Behavioral
Abstract Factory	Adapter	Command
Builder	Bridge	Interpreter
Factory Method	Composite	Iterator
Prototype	Decorator	Mediator
Singleton	Facade	Observer
	Proxy	Visitor

Table 1. Common design patterns.

Gamma et al. categorize their design patterns in to three groups (Gamma et al. 1995): creational, structural and behavioral patterns. Each of these categories include a set of patterns that are designed for certain type of problems. Creational design patterns are used in instantiation process, how the objects are created, composed and presented. Structural design patterns describe how objects and classes form a larger scale structures. Behavioral design patterns are concerned of dynamic relationships between objects and classes: how the ob-

jects communicate between each other, what are the responsibilities and how the control flow is handled at run-time. Some common design patterns are categorized in Table 1.

2.6 Design Pattern Examples

This section briefly introduces design patterns that were not mentioned in Gamma’s book but are relevant and useful in the context of this thesis.

Service locator is designed to reduce dependencies. This is achieved by creating a class that works as a container for implementations of services for a certain tasks. This container is called “service locator”. Figure 3 illustrates the relationship of the service locator and the service classes. Instead of binding service implementation explicitly, the requested implementation is requested from the service locator at run-time (Fowler; *Application and Design Patterns - The Service Locator*). How the service locator locates the right service, depends on the implementation. The code example below illustrates how the service locator imple-

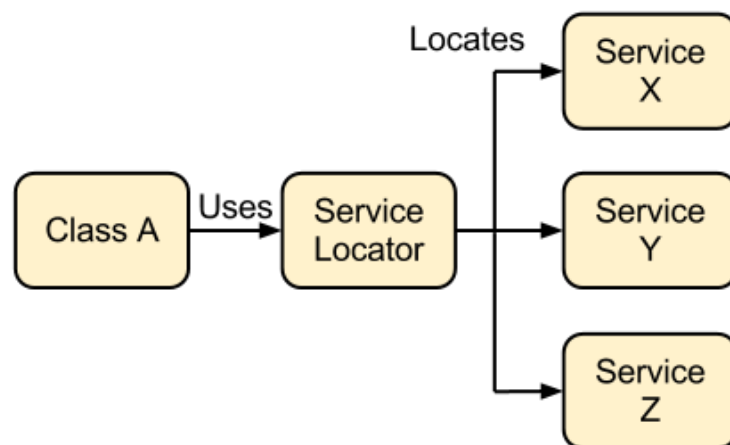


Figure 3. Service locator pattern.

mentation is used. In the example the service locator invokes the service according to the type information that is passed along the method call. Implementation details of the example service locator can be found from Appendix B.1.

```
// Initialize the service locator
```

```
IServiceLocator locator = new ServiceLocator();
```

```
//Invoke the service
```

```
IServiceZ = locator.GetService<IServiceZ>();
```

Data Transfer Object (DTO) is a design pattern that can be used to reduce the number of method calls (Fowler; *Enterprise Solution Patterns Using Microsoft .NET*). This is achieved by creating DTOs that work as data storages. The data can be transferred inside a DTO rather than making multiple method calls. This can reduce communication load between different parties. DTO is a data container so it should not have any business logic inside it.

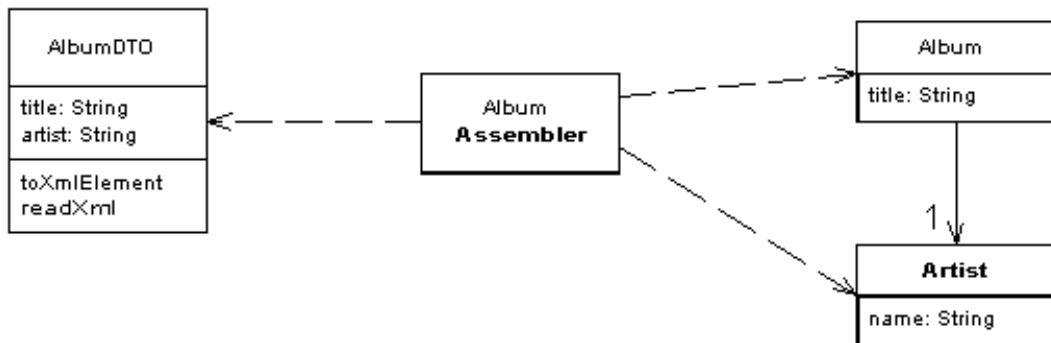


Figure 4. Data Transfer Object and Assembler pattern (Fowler).

DTO pattern is often used with *assembler* design pattern. *Assembler* pattern is a specialized instance of the *Mapper* pattern (*Enterprise Solution Patterns Using Microsoft .NET*). Its purpose is to create DTO from business objects and vice versa. DTO and assembler pattern are illustrated in Figure 4

Dependency injection is a design pattern that allows to move direct dependencies of concrete implementations. The key idea is that a class can be configured from the outside instead of configuring it from the inside. In a way the object of dependency is injected into a class

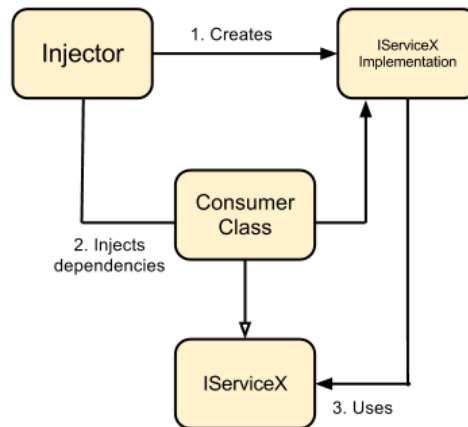


Figure 5. Dependency injection pattern.

that is dependent on it. In order to do the injection, the requirements of the destination must be known by the injector (or provider). This all can be done at run-time or compile-time (Fowler). The pattern has three key elements: *the consumer that has dependencies*, *defined component dependencies with interface contracts*, and *the injector*. The consumer has a dependency on some component that is described usually by an interface. The injector injects a concrete implementation that satisfies the contract interface into the consumer. This is illustrated in Figure 5. An example of using the dependency injection can be found from Appendix B.2.

3 Software Architecture and Quality

Architecture is the art of drawing lines.

–Uncle Bob (Robert C. Martin)

Quality is never an accident; it is always the result of high intention, sincere effort, intelligent direction and skillful execution; it represents the wise choice of many alternatives.

–Dwight David Eisenhower

Software architecture is concerned of making design decisions in order to define a structure and development guidelines for a piece of software so that it fulfills its requirements. Architecture has a great affect to the quality of software. It is usually the first thing that dictates how the requirements are achieved. That is why it is crucial to design an architecture that fits best for the purpose. Every software has an architecture whether it was deliberately designed or not. The need for comprehensive architectural design process for a software system is dependent on several things such as the size of the software system and the purpose of it. It can be a waste of time to spend weeks or months to design an architecture for a minor software system or for a system that is used only a couple of times.

Architectural patterns are proven architectural design principles under a certain type of constraints that offers guidelines for designing the architecture for the software. Early stage architectural decisions are often the hardest to change if they prove to be poor. The right kind of architecture can save money and time when software goes through changes over time. It can also expand the lifetime of the software by making it easier to do controlled changes and updates to the software.

The definition, the background and evaluation methods of the software architectures are presented in this chapter. Section 3.1 presents the definition and explains what software architectures are and what they are not. Section 3.2 describes briefly the history of software architectures. The concept of architectural pattern is presented in Section 3.3. In Section 3.4

the aspects of software quality and quality attributes are presented. The benefits of the software architecture evaluation are explained in Section 3.5. Different kind of methods for software evaluation are introduced in Section 3.6. Different kinds of scenario based evaluation methods are presented in Section 3.7.

3.1 General

The software architecture of a program or a computing system describes the essential elements of the system, interaction between those elements and relationships among them on an abstract level (Bass, Clements, and Kazman 1998). Every program or a system has an architecture whether it has been knowingly designed or not. In the book (Bass, Clements, and Kazman 1998) a software architecture is defined as follows:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

In an article (Garlan and Perry 1995) Garlan and Perry define a software architecture as follows:

The structure of the components of a program or system, their interrelationships, and principles and guidelines governing their design and evolution over time.

The *elements* or *components* are the primary building blocks of a piece of software. In practice these elements can be modules, classes or any meaningful aggregations. Architecture recognizes how these elements relate and interact with each other. In these elements essentials are their visible or public *interfaces* (Bass, Clements, and Kazman 1998). Software architecture does not describe the actual implementations behind these visible interfaces. This abstraction derives from the idea that architectural elements interact with each other through these interfaces. This is why it is not desired to add complex details in a architectural design.

As the first definition says, the architecture can comprise of more than one architecture. Each nontrivial structure or substructure can have its own architecture. The overall architectural structure of a system can be composed from these separate designs.

Software architecture does not focus on the details of the algorithms or data structures. Its concern is on global *control structures*; composition of *design elements*; *protocols* for communication; *physical distribution*; *synchronization*, and *data acces*; which *element* is responsible on what *functionality*; *scaling and performance*; and selecting among design elements (Garlan and Perry 1995).

3.2 History

The software architecture is relatively young discipline. The first occurrence of the phrase software architecture is from software engineering techniques conference organized by NATO in 1969. Before the late 1980s the meaning of software architecture was quite different from what it is nowadays. At that time the term architecture referred mostly to system architecture (Kruchten, Obbink, and Stafford 2006).

It was the beginning of 1990s when software architecture started to emerge to its own distinct discipline. The first significant article about software architectures was written by Winston W. Royce and Walker Royce in 1991 (Royce and Royce 1991). At that same year Philippe Kruchten wrote an article about combining an iterative development with architectural design (Kruchten 1991). In 1992 Dewayne Perry and Alexander Wolf published an seminar article about the study of the software architectures (Perry and Wolf 1992). The purpose of the article was to build the foundations for software architecture. The article introduced the basis for architecture description languages (ADLs). The first book on software architecture was published in 1994 (Witt, Baker, and Merritt 1994).

The late 1990s was significant in the field of the software architecture. Kazman et al. published a method for analyzing software architectures (Kazman et al. 1994). In 1995 Philippe Kruchten published an article about different view models (4+1) in software ar-

chitecture (Kruchten 1995). Many large private companies such as Siemens, Nokia and Philips (Soni, Nord, and Hofmeister 1995; Buschmann et al. 1996; Hofmeister, Nord, and Soni 2000; Jazayeri, Ran, and Linden 2000) started to pay attention to the software architectures.

In the following years many notable things happened in the field of the software architectures. In 1999 the first IFIP Conference on software architecture (WISCA1) (Donohue 1999) was held in San Antonio, Texas, USA. Software architectures were really starting to get more attention. Many books and articles concerning the best practices in the software architecture started to emerge. The IEEE standard concerning software architectures was published in 2000 (IEEE 1471 2000).

Today software architecture is an essential part in the field of software engineering. There is a large volume of literature about software architectures. Companies have dedicated employees specialized on architectures. Many technologies and frameworks have emerged that provide an architectural basis for software. For example, many modern web frameworks, such as Ruby on Rails, Grails and ASP.NET MVC, make architectural decisions for developers.

3.3 Architectural Pattern

An architectural pattern (or style) characterizes a family of systems that are related by shared structural and semantic properties (Monroe et al. 1997). *An architectural pattern* is not an architecture. A concrete software has an architecture that can be *an instance of an architectural pattern*. It also determines how **components**, **connectors** and **configurations** can be used in instances of that pattern together with a set of design constraints. In this context components are modules that contain computational logic while connectors are concerned of communication and configurations are the collection of interacting components and connectors (Abowd, Allen, and Garlan 1993). Bass et al. (Bass, Clements, and Kazman 1998) defines an architectural pattern as follows:

An architectural pattern is a description of elements and relation types together with a set of constraints on how they may be used.

Architectural patterns have a vocabulary for components and connectors (e.g. “pipe” and “filter” in *pipe and filters* pattern or “client” and “server” in *client-server* pattern) and analysis of the pattern (Garlan and Shaw 1994). The purpose of these vocabularies is to give insight on how the architectural pattern is organized and what are the responsibilities of each component and how they should interact.

An architectural pattern is one way to communicate experiences on designing software architectures in order to produce better designs. In other words, an architectural pattern promotes design reuse. These styles give an intuition to system’s high-level structure rather than reveal any concrete implementation details. Perhaps the most important thing that architectural pattern provides are the quality attributes. These quality attributes give criteria for software architects on what kind of architectural pattern should be chosen for some particular case (Bass, Clements, and Kazman 1998).

3.4 Quality Attributes

The quality of the software is measured by the degree of how well user requirements are met. The term *quality* can be defined as (IEEE P1471 2000) “*the degree to which a system, a component, or a process meets customer or user needs or expectations*”. The quality consists of *quality attributes* (QA). Quality attribute is a nonfunctional characteristic of a component or a system (Dobrica and Niemela 2002). Defining and analysing these quality attributes can be challenging because many of them can be obscure, such as safety and portability (Kazman et al. 1994, 1996). According to (Dobrica and Niemela 2002) the number of the quality attributes can vary in software evaluation literature. The ISO/IEC 9126-1 standard names six categories of characteristics which are divided into subcharacteristics. The list is as follows (ISO 2001):

- **Efficiency:** Efficiency consists of set of attributes indicate the level of performance of the software and the amount of resources used. Efficiency can be sub-characterized into *time behaviour, resource utilization and efficiency compliance*.

- **Functionality:** Functionality refers to a set of functions that satisfy required needs. It consists of *suitability, accuracy, interoperability, security and functional compliance*.
- **Maintainability:** Maintainability refers to a set of attributes that indicate the needed effort in order to make changes. Essential things are *analyzability, changeability, stability, testability and maintainability compliance*.
- **Portability:** Portability is the ability of the software to be transferred from one platform to another. Attributes that are related to it are *adaptability, installability, co-existence, replaceability and portability compliance*.
- **Reliability:** Reliability refers to the ability of the software to perform under required conditions for a required period of time. It has four following sub-characteristics: *maturity, fault tolerance, recoverability and reliability compliance*.
- **Usability:** Usability concerns about the amount effort that is required in order to use the software. It has five sub-characteristics: *understandability, learnability, operability, attractiveness and usability compliance*.

When evaluating a software the evaluator should decide what kind of quality attributes are relevant in that particular case. Not all of the listed attributes should be a part of the evaluation process. The set of the quality attributes should always be chosen according to case specific objectives. Things that may affect on the set of evaluated quality attributes are the architectural pattern, the environment where the software operates and functional objectives of the software.

3.5 Evaluation

The architecture of the software system has always a set of constraints that has its benefits and tradeoffs. By knowing what kind of effects the chosen architecture has on the system, architects can make better design decisions. An architecture evaluation should be a standard part of every architecture-based development methodology (Bass, Clements, and Kazman 1998).

The quality of the software is a property that can not be added in the last phases of the development. By evaluating architecture early in a project, design decisions can be made in

order to fulfill the quality requirements. Early evaluation is usually the most cost-effective. The earlier the problems are discovered, the cheaper it is to fix them (Abowd et al. 1997). Evaluations should be made multiple times during the software life-cycle.

The evaluation process can also increase understanding of the architecture of the software system. Before the architecture is reviewed, reviewees are required to document it. This documentation can then be used as top-level architectural description. The evaluation process may raise questions that needs answer. Questions create conversations that offer explanations for architectural choices that were made. Given answers may prove valuable during the life cycle of the software (Abowd et al. 1997).

The architectural evaluation can also give a better understanding of software requirements. By examining how well an architecture fulfills its requirements it awakens questions and discussions about the requirements. Requirements may have conflicts with each other. By uncovering the conflicts it is possible to make tradeoffs between the requirements by prioritization (Abowd et al. 1997).

As can be seen, the evaluation has a lot of benefits that can make a difference how the whole software project turns out. The lack of evaluation may cause financial cost and, therefore, be the reason for possible failure. In order to perform a meaningful evaluation for an architecture, there must be a set of preconditions that the architecture should fulfill. Bass et al. defines six preconditions for evaluation (Bass, Clements, and Kazman 1998): *clearly articulated goals and requirements for the architecture, controlled scope, cost-effectiveness, key personel availability, compenent evaluation team and managed expectations.*

In order to make decission if architecture is suitable or not, the recognition of the quality attributes is required. The chosen architecture can enable high performance but suffer from the lack of modifiability. Whether this is desired quality for an architecture depends on the requirements.

The evaluation should be focused on a small set of high-priority goals. If the number of goals is more than five, it may indicate that the expectations are unrealistic.

Expenses of the evaluation should not be greater than gained benefits. For a small project it

is not meaningful to spend months for the evaluation nor is it wise to do the evaluation in one day for a large scale architecture.

In order to perform a good quality evaluation, all relevant stakeholders should participate to evaluation process. People that do the evaluation should also be motivated. If the people do not consider it beneficial, the quality of the evaluation suffers. Evaluators should also be fluent in architecture and architectural issues. Everyone should also have a mutual understanding of the expectations and goals set for the architecture.

3.6 Evaluation Methods and Techniques

Architecture evaluation methods can be divided into two category: *questioning techniques* and *measuring techniques* (Abowd et al. 1997; Dobrica and Niemela 2002). The key idea of questioning technique is to produce qualitative questions that can reveal possible defects in an architecture. In practice, *a questioning technique* can be a scenario, questionnaire, or checklist. The following listing explains briefly what these techniques are (Abowd et al. 1997):

- **Scenario:** Scenario is a short description of some specific usage of software or some modification that is to be made. The purpose of the scenario is to provide information on how the architecture responds to it. That is to say, whether it should be modified before it can fulfill the scenario. It is beneficial to create scenarios for each group of stakeholders. A stakeholder can be *an end user, a developer or a system administrator*. Scenarios are always system or software specific and they are developed as a part of evaluation process.
- **Questionnaire:** A questionnaire is a list of questions that apply to all architectures. Therefore, they are general and relatively open. These questions can be for example *“how the architecture is documented?”* or *“who designed the architecture?”*
- **Checklist:** Checklist contains more detailed questions than a questionnaire. Checklists are composed by using the experience and knowledge drawn from previously evaluated systems. Whereas questionnaires are relatively general, checklists are usually highly domain-specific.

Several methods have been developed for questioning. These include *SAAM*, *ATAM*, *CBAM*, *ALMA* and *FAAM*. All mentioned techniques are so called scenario-based software architecture evaluation methods. An overview of each of these methods is described in (Hammer, Ionita, and Obbink 2002).

Measuring techniques include quantitative measurements of the architecture. They can be regarded as more mature than questioning techniques. This is because measuring techniques provide answers rather than generate questions. These techniques include metrics, simulations, prototypes and experiences (Abowd et al. 1997; Dobrica and Niemela 2002). Metrics measure some observable aspects of the software. These can be, for example, lines of code or cyclomatic complexity. Some useful metrics measure *cohesion* and *coupling*. One way to capture *cohesion* and *coupling* is to examine relationships and connections between modules (or classes) and functions (or methods) (Briand, Morasca, and Basili 1993). It is desirable that a system has low coupling and high cohesion. Cohesion can be defined as follows (Briand, Morasca, and Basili 1993):

Cohesion is the extent to which a module only contains data declarations and subprograms which are conceptually related to each other.

Cohesion can be extracted in many ways but one good example of how the cohesion can be calculated is presented in (Briand, Morasca, and Basili 1993).

While the cohesion describes how well conceptually related functions are grouped together, *coupling* captures their dispersion by looking dependencies between different modules (Briand, Morasca, and Basili 1993).

Prototypes and simulations may help to figure out what kind of architecture should be chosen. A prototype or candidate model for architecture consist of unimplemented function or method stubs. Simulations can be performance models. These simulations and prototypes may provide an answers to questioning techniques.

The mentioned evaluation techniques are categorized and summarized in Table 2. Generality column implies to focus of the technique. In other words, can it be applied to all cases or is it perhaps a domain specific. Level means the level of detail or how much we should

know about the architecture before that technique can be used for evaluate it. The phase says when the technique should be applied during the software life cycle. The target tells what the technique evaluates. *Artifact* means that the focus is on the architecture and its properties. *Artifact process* means that the evaluation focus on the role played by the architectures in the development process. More detailed descriptions about the meanings of the dimensions can be found from (Abowd et al. 1997).

Method	Generality	Detail	Phase	Target
Questionnaire	general	coarse	early	artifact process
Checklist	domain-specific	varies	middle	artifact process
Scenarios	system-specific	medium	middle	artifact
Metrics	general or domain-specific	fine	middle	artifact
Prototype, Simulation, Experiment	domain-specific	varies	early	artifact

Table 2. The Properties of The Evaluation Approaches (Abowd et al. 1997).

3.7 Scenario Based Evaluation Methods

In this section, three different scenario-based architectural evaluation methods are presented. The methods are *SAAM*, *ATAM*, and *CBAM*.

3.7.1 SAAM

SAAM is a scenario-based software analysis method that forces designers to consider the future of the software system rather than the present state. SAAM has five different steps: *describe candidate architecture; develop scenarios; evaluate each scenario; reveal scenario interaction; and, weight scenarios and scenario interactions* (Kazman et al. 1994, 1996). The dependencies between the steps are illustrated in Figure 6.

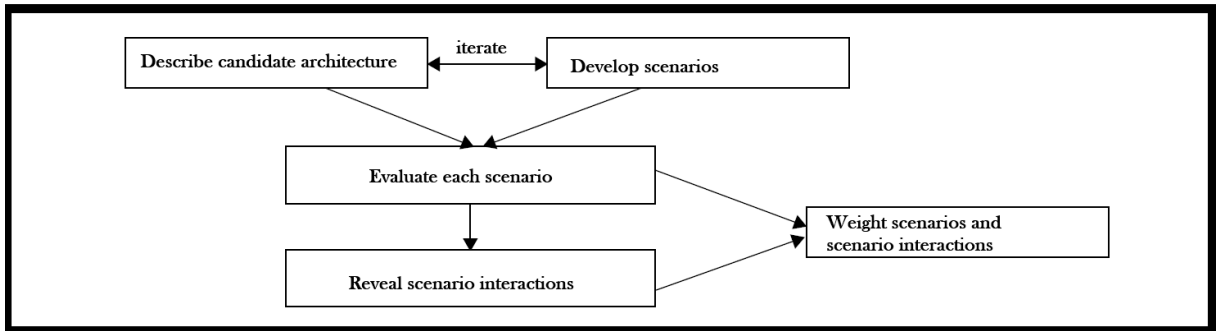


Figure 6. Activities and dependencies in scenario-based analysis (Kazman et al. 1996).

A more detailed descriptions of each of the steps are presented in the following listing (Kazman et al. 1994, 1996):

- **Describe candidate architecture:** Before the analysis can start, there must be a representation of the current (candidate) architecture that describes what it means to be a component or connector. It should indicate computation and data components and relationships between them. Representation can be a static or dynamic and there can be many representations. Dynamic representation can be a natural-language specification of the overall behavior.
- **Develop scenarios:** Scenarios should include activities that system must support and changes that system is expected to undergo. Those activities should be relevant to different stakeholders.
- **Evaluate each scenario:** If the architecture can execute the task and activities of the scenario, then the scenario is *direct*, otherwise it is *indirect*. The interest is on *indirect* scenarios because they represent changes that the system is required to undergo in order to execute that particular scenario. For each indirect scenario there should be a list of the required changes, cost and time estimates. By the end of this step, there should be a summary table of all direct and indirect scenarios. For each indirect scenario there should be description of the impact the scenario has on the architecture.
- **Reveal scenario interaction:** If two or more indirect scenarios require changes to the same component they are said to interact. A high degree of interaction may indicate a poorly isolated functionality. These interactions point potential trouble spots of the design. They also measure how the architecture supports an appropriate separation of

concerns.

- **Weight scenarios and scenario interactions:** Each scenario and scenario interaction should be weighted in terms of relative importance and use those results to determine an overall ranking. This step involves all stakeholders in the system. The weighting reflects the relative importance of the quality factors that scenario manifests.

3.7.2 ATAM

Architecture trade-off analysis method (ATAM) (Kazman, Klein, and Clements 2000) is a scenario-based architecture evaluation method that is based on SAAM. The focus is on the evaluation of the quality attributes. The method also explores the interactions, interdependencies of the quality attributes, and the tradeoffs between the quality attributes. It gives answer to how well the software satisfies the predefined quality goals.

The ATAM evaluation sessions consists of four phases: *presentation phase*, *investigation phase*, *testing phase*, and *reporting phase*. A more detailed descriptions of the phases are presented in the following listing (Kazman, Klein, and Clements 2000):

- **Presentation phase:** This phase begins by describing all the participants involved in the evaluation session. After that, the business goals and primary architectural quality drivers are described. Lastly the software architect presents the architecture of the software and describes how it addresses the business goals that were set.
- **Investigation phase:** In this phase the architectural approaches are presented. After that, the requirements of the system are mapped to appropriate architectural property. Lastly, the architectural approaches are analyzed and rated.
- **Testing phase:** This phase begins by creating a large group of scenarios and prioritizing them. After that, the prioritized scenarios are used as an input for reiterations of architecture approach analysis step.
- **Reporting phase:** In this phase, the evaluation team summarizes and presents the results of the three previous phases to the stakeholders.

3.7.3 CBAM

Cost-benefit analysis method (CBAM) (Kazman, Asundi, and Klein 2001) is an architecture evaluation method that brings together the architecting process and the economics of the organization. CBAM adds costs as quality attributes and treats them equally with other quality attributes. The method is build upon the ATAM. The CBAM evaluation sessions of two phases: *tread*, and *detailed examination*. The first phase is necessary only if there are many architectural strategies to be discussed. Only a few are selected to the second phase. If there is only a few architectural strategies, the evaluation session starts right from the second phase. Both phases consists of six steps that are described in the following (Kazman, Asundi, and Klein 2001):

1. **Choose scenario of concern an their associated architectural strategies:** Choose scenarios that concern most the stakeholders and address architectural strategies for chosen scenarios.
2. **Assess quality attribute benefits:** Benefits of the elicited quality attributes are formed by using the expertise of managers who best understand the business implications.
3. **Quantify the benefits of the different architectural strategies:** The architectural strategies are elicited by using the expertise of software architects.
4. **Quantify the architectural strategies costs and schedule implications:** The cost and schedule are elicited with business managers and architects.
5. **Calculate desirability:** The evaluation team counts the desirability level for each architectural approach based on the ratio “benefit divided by the cost”. More detailed formula can be found from (Kazman, Asundi, and Klein 2001).
6. **Make decissions:** The cost-benefitally best architectural strategies are chosen.

4 EBI and SOA Patterns

Divide et impera – Divide and Conquer.

–Julius Caesar

The Entity-Boundary-Interactor (EBI) architectural pattern and the service-oriented architecture pattern (SOA) are presented in this chapter. The background of the EBI pattern is presented in Section 4.1. In section 4.2 a general principles of the EBI pattern are presented. The quality attributes of the EBI are presented in Section 4.3. The implementation details of the EBI are presented in Section 4.4. In Section 4.5 the service-oriented architecture pattern is defined and the principles of SOA are presented. Section 4.6 introduces the SOA related quality attributes and what kind of architectural trade-offs comes with them. Section 4.7 presents implementation details of the SOA pattern.

4.1 Background

The clean architecture is a term used by Robert C. Martin in his talks and and writings concerning software architectures (Martin, 2012, 2011a). In his blog post (Martin 2012) Martin mentions four architectural patterns: *hexagonal architecture* (Cockburn), *onion architecture* (Palermo), *DCI* (Coplien and Bjørnvig 2011), and *BCE (or EBI)* (Jacobson 1992). They all share a common objective, which is *the separation of concerns*. Previously mentioned architectures achieve this by dividing the software into layers. They all share five common aspects, which are presented in the following listing (Martin 2012).

- *Independent of frameworks*. The architecture does not depend on any external libraries or frameworks. Frameworks and libraries can be used as tools rather than inseparable part of the architecture.
- *Testable*. The business rules can be tested without any external elements.
- *Independent of UI*. The UI can be changed without any affect to the architecture be-

cause the UI not bound to it.

- *Independent of Database.* The database can be changed without any affect to the architecture because the business rules are not bound to the database.
- *Independent of any external agency.*

The clean architecture concept is illustrated in Figure 7.

In this thesis, the term *clean architecture* refers to an architecture that follows the rule that makes these architectures work: *the dependency rule*. This means that source code dependencies can only point inwards. In Figure 7 it means that inner circles does not know

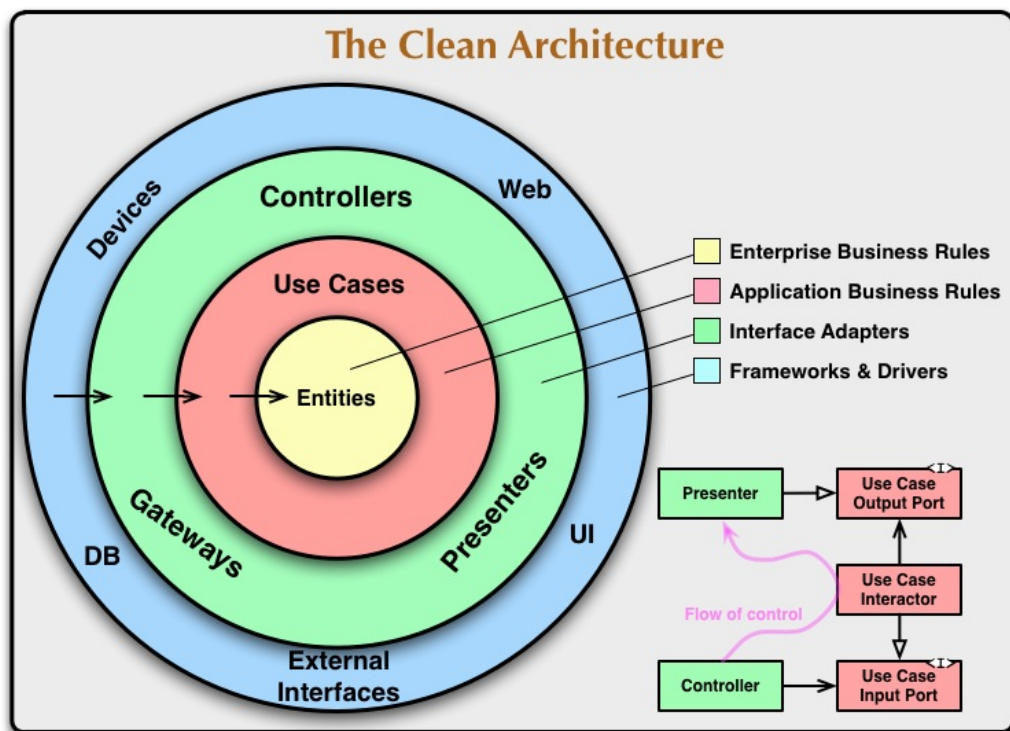


Figure 7. The Clean Architecture (Martin 2012).

anything about the outer circles. Nothing on the outer circle should have any impact to the inner circles.

There are four different layers¹ in Figure 7: *entities*, *use cases*, *interface adapters*, and

1. There is nothing that limits the number of the circles to four. The important thing is that the dependency rule is not violated.

frameworks and drivers. The most inner layer consist of *entities*. An entity encapsulates enterprise-wide business rules. An enterprise-wide business rule is an application agnostic rule. Put differently, a rule that is so general that it can be used in multiple applicatios without any changes to it. In practice, it can be an object with methods. or it can be a set of data structures and functions (Martin 2012).

The application business rules are encapsulated into use cases. The use case orchestrate the flow of data to and from the entities. Enterprise wide business rules of the entities are used to achieve the goals of the use case. Changes in this layer should not cause changes to the entities. Changes should be expected only if use cases change (Martin 2012).

The interface adapters layer is a set of adapters that convert the input data of the application into proper format for the use cases and interactors. The adapters also convert the output data of the application into a format that is proper for external agencies, such as the database or the web (Martin 2012).

The frameworks and drivers layer consists of external agencies. In other words, this layer contains the code that is needed to attach the UI, the database or some web framework into the application (Martin 2012).

The boundaries between the layers should be crossed in a way that the dependency rule is not violated. This means that the inner layers are not allowed to to direct calls into upper layers. That is to say, the source code dependencies should always points inward. At the lower right of Figure 7 is an example how the boundaries are crossed. The work flow arrow shows how the controller in the interface adapters layer communicates with the use case interactor of the use cases layer. Then the use case interactor communicates with the presenter of the interface adapters layer. It should be noted that all the arrows from the interface adapters layer (dependencies) points towards the interactor in the use cases interactor. The use case interactor has no dependencies with the interface adapters. This is achieved by using *the dependency inversion principle* (DIP) (Martin 2012). The DIP states two things (Martin 2003):

1. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

2. *Abstractions should not depend upon details. Details should depend upon abstractions.*

In Figure 7 the DIP manifests as dependencies on the interfaces rather than concrete classes. The controller access the interactor via *use case input port interface*. It knows nothing about the underlying implementation. The same pattern is found when the interactor needs to communicate with the presenter. Instead of making direct call to the concrete presenter class, the interactor calls some thing that implements the *use case output port interface*. The dependency rule is not violated.

The data that crosses the boundaries should be simple data structures (Martin 2012). The data can be in a struct or wrapped into DTO, or it can be arguments in function calls. The important thing is that the entities are not passed across the boundaries as such. This would violate the dependency rule. Outer circles should pass data in the form that is most convinient for the inner circle.

4.2 Principles of EBI

Entity-Boundary-Interactor (EBI) pattern is inspired by Robert C. Martin's speech '*Keynote: Architecture the lost years*' at Ruby Midwest conference on 2011 (Martin). The EBI is based on the ideas of Ivar Jacobson that he presented in his book '*Object-Oriented Software Engineering: A Use Case Driven Approach*' (Jacobson 1992). In the EBI pattern, the architecture is build upon *use cases*. In this context a use case is a sequence of transactions that the user performs in a dialog with the software system (Jacobson 1992). The software that is build according to the EBI patterns consists of three basic elements: *entities*, *boundaries*, and *interactors*. Interactors are objects that are invoked when the user uses the software. In other words they implement use cases. This is called as *use case driven design* (Jacobson 1992). The whole system is controlled from what the user wish to do with the system. If the functionality of the software system requires changes, that is, the use case changes, the developers see directly which interactor needs to be changed.

The essential parts or building blocks of the EBI pattern are presented in the following list and the top level architecture can be seen in the Figure 8.

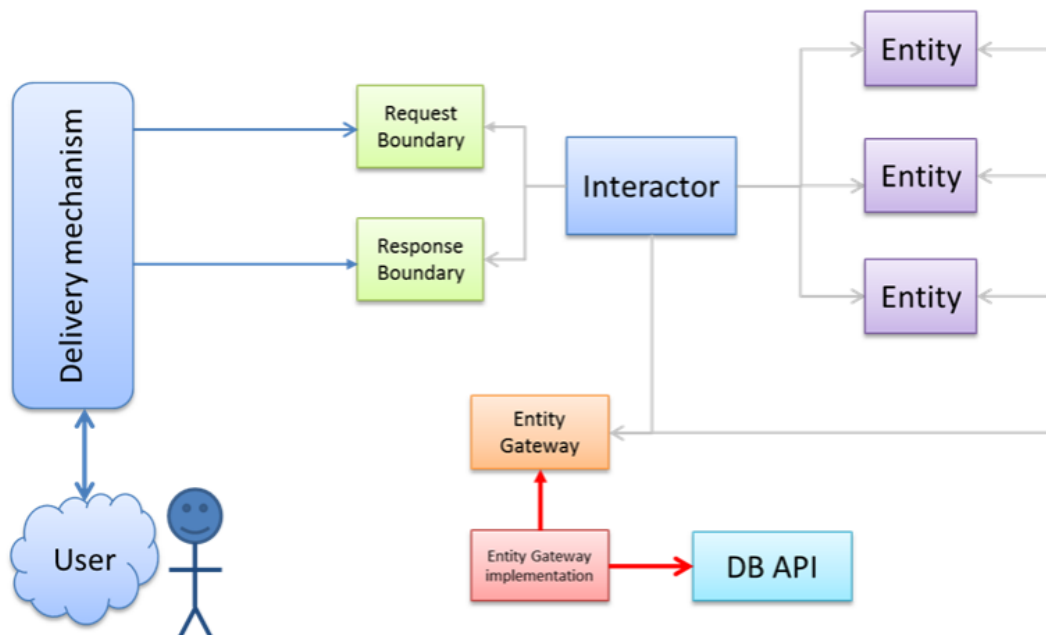


Figure 8. Entity-Boundary-Interactor Pattern.

- **Entity:** Entities represent business objects that have application independent business rules. This means that no matter where they exist they always encapsulate the same business rules. All the application agnostic business rules should be located in the entities. Entities in the EBI pattern represents the entities layer described in Section 4.1.
- **Interactor:** Interactor is a object that implements a use case. They have application specific business rules. Application specific business rules are rules that apply to some specific application for instance a book store application. This means that these business rules can not be a part of some framework or library that might be used in the application. This means that the interactors do not live in any framework or library; they are a essential part of the core functionality of the some specific application. If we look at Figure 4.1, these interactors are located in the use cases layer.
- **Boundary:** Boundaries are interfaces. In the figure 8 there are three different boundaries. The *request boundary* is an access point to services that the interactor implements. Where as the *reponse boundary* is interactors way to send the possible results of the request that was made. The third boundary in the figure 8 is called *Entity gateway*. It is an access point to the system that stores entities. It can, for example, be

a database. In the context of the clean architectures, described in Section 4.1, these boundaries represents the boundary between the interface adapters layer and the use cases layer. These boundaries enable compliance with the DIP.

- **Delivery mechanism:** Delivery mechanism is something that delivers request to the application and takes reponse messages from the application. For instance, it can be a web framework or it can be a desktop application or some testing framework like the NUnit. Put differently, it is just a detail that has nothing to do with the functionality of the application. In the context of the clean architectures, the delivery mechanism is located on the most outer layer in Figure 7.

4.3 Quality Attributes of EBI

In his blog post ‘*Screaming Architecture*’ (Martin 2011b) Martin points out the reasons why a good architecture is build around use cases and why the application should be decoupled from frameworks and other external agencies. Decoupling the architecture from external agencies enables architect to describe structures that support the use cases. This should be architects first concern, not the frameworks that should be used. The EBI pattern enables this by *separating the details*, such as UI and database from the application (Martin, 2011a). This increases the *flexibility* of the software.

The top level architecture should *reflect the intent of the application* or as Martin says the architecture should “scream” its intent. This manifests in the EBI pattern as a naming convention. The interactor should be named according to the use case it implements (Martin). For example, if the use case is called ‘*List customers*’ the interactor class that implements this use case should be called `ListCustomers` or equivalent. The use case driven design makes the architecture more *traceable* in the situations where the architecture requires changes.

By decoupling the application from external agencies, the architect can defer decissions such as what framework or database should be used. In the EBI pattern, all dependencies point inward, which means that the application has no external dependencies (Martin). This gives flexibility to the architecture. It will also make the application easier to test. Use cases can

be tested without the presence of the UI, web server or the database. The database can be replaced with some lightweight module that mimics the database. The UI can be replaced with some testing framework such as the NUnit. This will shorten the time required to get the results of the unit tests. In other words, decoupling in the EBI pattern improves the *testability* of the architecture.

4.4 Implementation of EBI

The implementation details concerning the EBI pattern are presented in this section. These details include the architectural structure and messaging model.

As mentioned in Section 4.2, all the dependencies in the EBI pattern point inward. This is presented in Figure 9. The figure shows how the interactors and entities can be hidden behind the boundaries. The delivery mechanism uses request boundary that is implemented

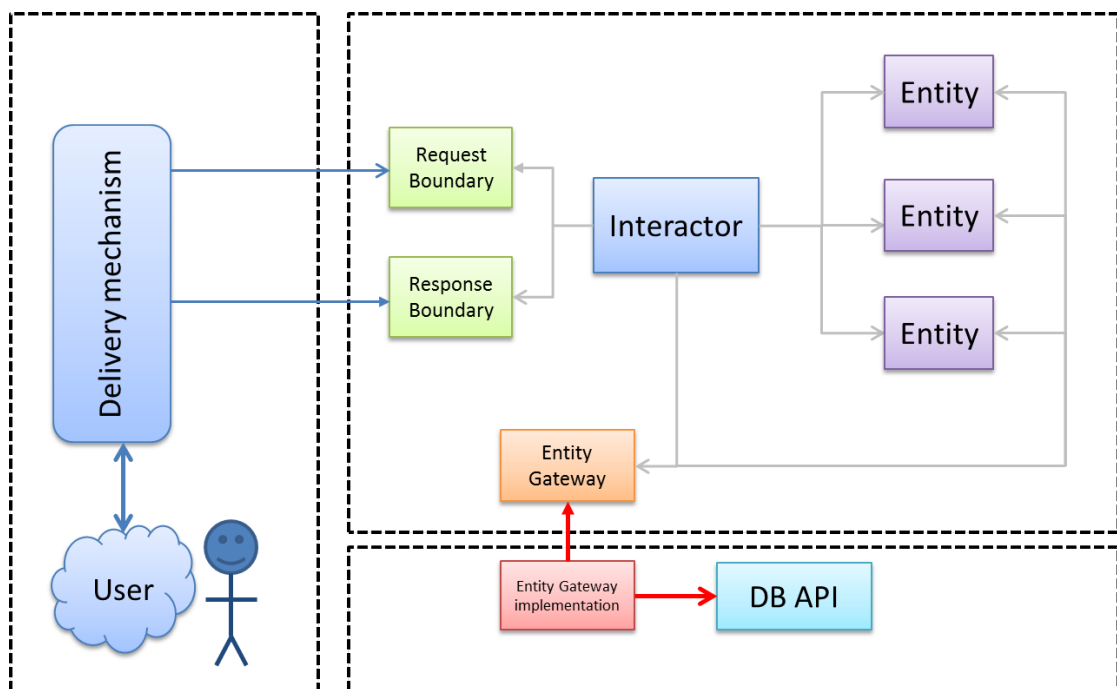


Figure 9. Entity-Boundary-Interactor Module Thinking.

by the interactor. The response boundary is implemented by the delivery mechanism. That is to say, the delivery mechanism has dependencies on boundaries but nothing in the core system depends on the delivery mechanism. This same idea applies to the entity gateway

boundary. The interactor uses the entity gateway to access preserved data from some system that implements the entity gateway boundary.

The whole system can be thought as a set of modules. The delivery mechanism can be a module to the core application and the system that implements the entity gateway can be a module. The application itself is completely decoupled from the implementations of the request and gateway boundaries. Each one of these can exist, for example, on its own dynamically linked library file. This whole idea is illustrated by dashed boxes in the Figure 4.4. The delivery mechanism is on the right, the application itself is on the top left and the entity gateway implementer on the bottom left.

There are two kinds of messaging boundaries in the EBI pattern: *request boundaries* and *response boundaries*. *Request boundary* is an interface that hides the actual implementation of the interactor from the delivery mechanism. It operates as an access point on which the

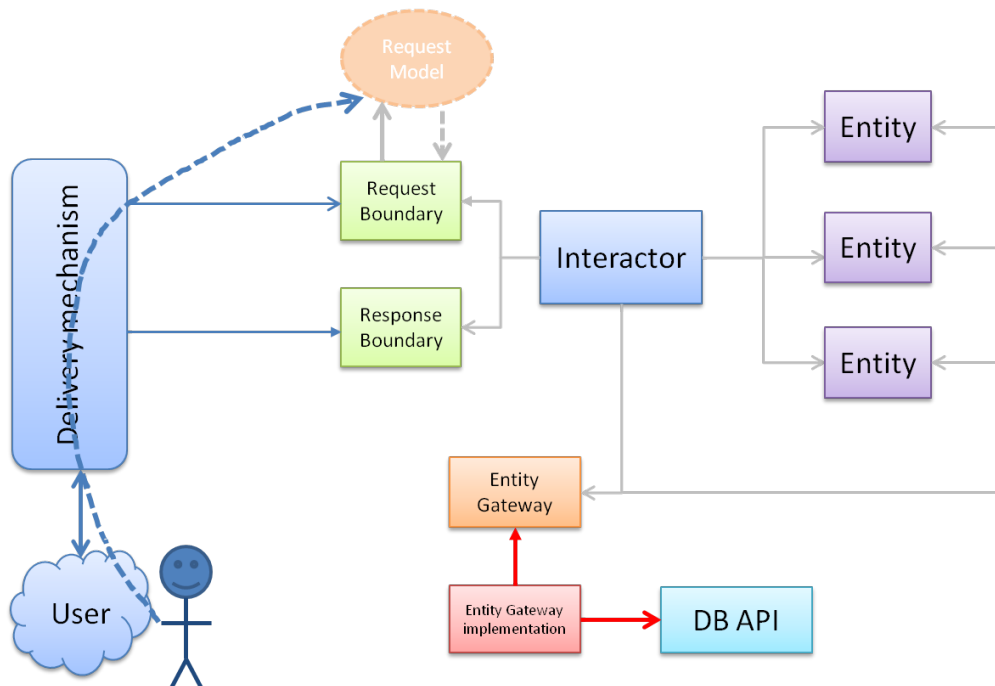


Figure 10. Entity-Boundary-Interactor Request.

user can use to make service requests. The request message or model itself can be, for example, a some kind of simple data structure. It should not contain any logic and its only purpose is to store necessary data so that interactors can reason the meaning of the message.

A dictionary that contains only key-value pairs is a good example of this type of structure. The request action is illustrated in Figure 10.

Response boundary is an interface through which the interactor can interact with the delivery mechanism. In other words, the interactor sends all the response messages through the response boundary. The implementation of this boundary is responsibility of the implementor of the delivery mechanism. Like the request model, the response model should be a sim-

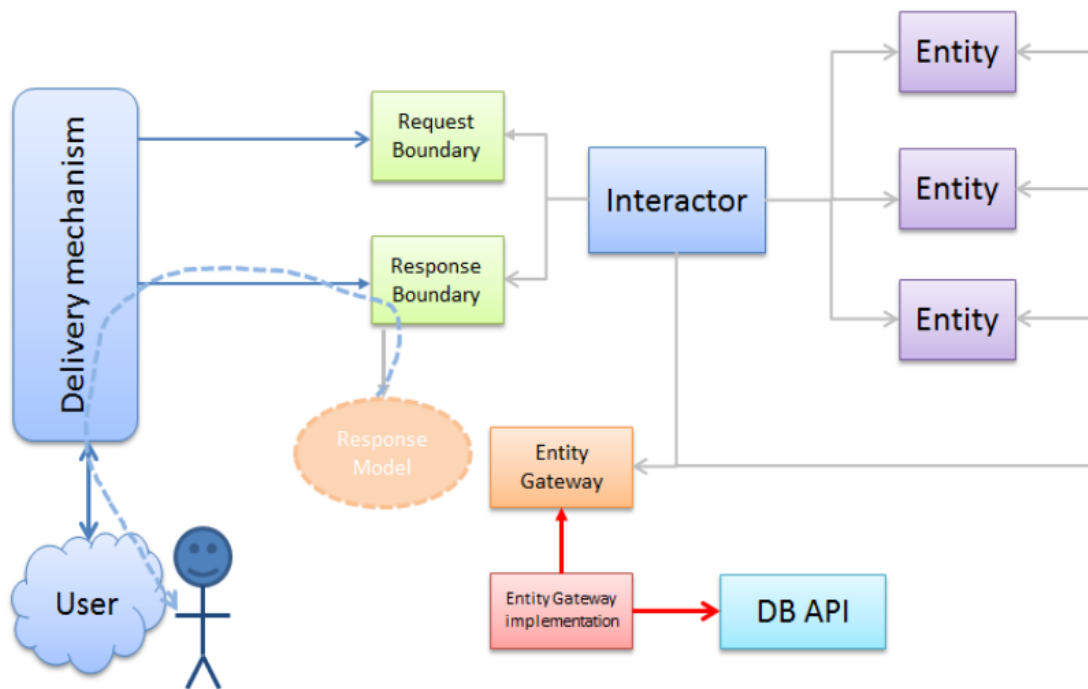


Figure 11. Entity-Boundary-Interactor Response.

ple data structure that only contain the relevant data. The response action is illustrated in Figure 11.

4.5 Principles of SOA

Many of today's distributed systems are implemented using the *service-oriented architecture* (SOA) approach. SOA has many definitions in the literature. Some definitions claim that it has something to do with a particular technology such as web services. In reality SOA is an architectural pattern, a set of principles, and it is not bound into some particular technology. Erl et al. (Erl et al. 2012) define SOA as follows:

Service-oriented architecture is a technology architectural pattern for service-oriented solutions with distinct characteristics in support of realizing service-orientation and the strategic goals associated with service-oriented computing.

Another good definition for SOA is (Spratt and Wilkes 2004):

The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface.

The first definition defines that SOA is an architectural pattern for realizing service-oriented systems. As a architectural pattern, it sets constraints and gives guidelines for reaching the service-oriented architectural design (see Section 3.3).

In the latter definition, the focus is on the concept of *services* and on how the *services* are made available. It refers to a fact that SOA is a guideline for designing a software that provides *services* to *a service consumers*. SOA itself does not say how the services should be implemented. It just gives guidelines how to offer services to *a service consumers*. Party that provides *the service* is called *a service provider*. A *consumer* and a *provider* are roles. Former invokes the service and latter provides (implements) it. Jointly they are referred to as *service participants*. A *service* is some task done by *a service provider* in order to produce desired result for *a service consumer*.

The latter definition says that services can be invoked, published and discovered through *an interface*. This *interface* should also be segregated from the implementation. This refers to a

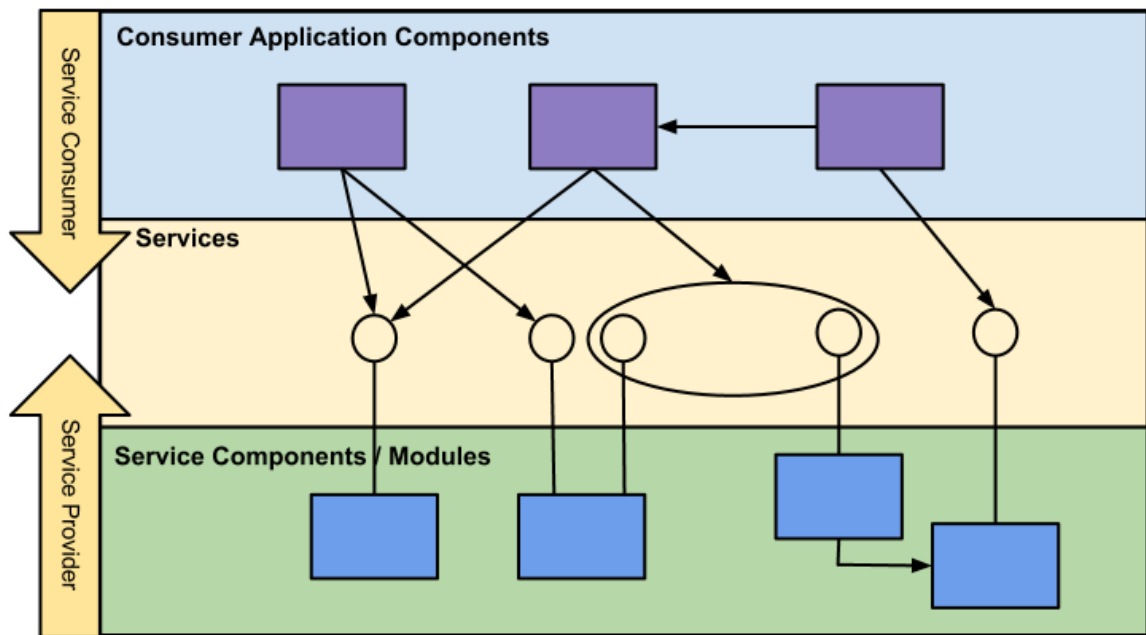


Figure 12. A simplistic illustration of SOA.

design principle that *an interface* is in some sense a layer that separates *the consumer* from *the provider* so that *the consumer* has no interest on what kind of provider implements the service on the other side of that interface and vice versa. The consumer's only concern is that it gets the desired result from the service. This service interface is in a way a promise to *the consumer* on which *the provider*, that implements the service, is bound. Figure 12 presents a simple illustration of the SOA.

A service is a way to access one or more utilities produced by *the service providers*. *The service consumer* can access these utilities by predefined interface that is described in the service description. Brown et al. (Brown, Johnston, and Kelly 2002) define *a service* as follows:

A service is generally implemented as a coarse-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model.

O'Brien et al. (O'Brien, Merson, and Bass 2007) give the following definition for *a service*:

A service is a distributed component with the following characteristics: is self-contained; has a published interface that abstracts the underlying logic; is location transparent; can be implemented in different languages or platforms and still interoperate; is discoverable and dynamically bound.

In the first definition parts “generally implemented as a coarse-grained” and “interacts with applications and other services” refers to a principle that *a service* can be composed from multiple services so that it offers one meaningful service in a perspective of *the consumer*.

SOA comes with a set of principles that are to be followed when implementing a software system in an SOA pattern. What these principles are varies in the literature. Different kinds of proposals of SOA principles can be found from following references (Brown, Johnston, and Kelly 2002; Erl 2005; Balzer 2004; The Microsoft Windows Communication Foundation team). Most of the proposals have same kind of principles and guidelines. The following listing is constructed according to those principles that are found from most of the references.

- **Service abstraction:** Service should be abstracted from the implementation (service provider), i.e., hides logic from consumers. This promotes reuse, growth, and interoperability.
- **Service statelessness:** All necessary information should be passed to the service when it is invoked. This means that the service provider does not make any hidden assumptions that the consumer should be aware of.
- **Service reusability:** Service should be reusable in a way that other services can be composed by using the functionalities of existing ones.
- **Published service interfaces:** The internal logic of the service should not be exposed. Only the functionality should be published.
- **Formal contracts:** There should be clear specifications about the obligations for service providers and consumers concerning the service.

In addition to former listing there are some constraints that can improve the SOA-based software system if they are applied. First of these constraints is *idempotent request*. In the context of computer science the term *idempotent* refers to an operation which can be done

multiple times with same input and the result is always the same as in the first time. If the service is *idempotent* it will increase the reliability of it because the same request can be repeated in the case of failure without the fear that *the service* would produce unexpected results.

Another constraint is *technology neutrality*. This means that services should be available to consumers in spite of the platform of the consumer application. This restricts of the use of a programming language specific APIs as a practice of making services available for consumers. Service invocations should be based on standardized formats and vendor independent technologies.

4.6 Quality Attributes of SOA

Before choosing the SOA approach for implementing a software systems, architect should be aware of SOAs quality attributes and what kind of trade-offs are related to those. O'Brien et al. list nine SOA quality attributes in their article (O'Brien, Merson, and Bass 2007). The quality attributes are: *interoperability, performance, security, reliability, availability, modifiability, testability, usability, and scalability*.

Interoperability is communication between different entities that share specific information and use it according to an agreed operational semantics (Brownsword et al. 2004). It is considered as the most prominent benefit of the SOA.

Performance has multiple meanings depending on the context. Usually it refers to the response time, throughput, or timeliness. Performance is a drawback of the SOA because of its distributed nature. Service providers and service consumers are usually located on different machines, which can cause latency in communication. On the other hand, SOA enables *location transparency*, for example, services deployed in multiple locations that can improve the total throughput and availability of a system.

Security is a thing that should always be taken under a cautious consideration when designing a software, especially when dealing with systems that use web services.

Reliability is the ability of a system to keep operating over time without failure (Clements,

Kazman, and Klein). Reliability of messaging between end points and the reliability of services are essential in SOA. Usually the SOA platform is responsible for providing reliability in messaging. The reliability of services refers to preserving data integrity in the context of transaction management during failures.

Availability concerns both service providers and consumers. Ideally the service is always available when customer tries to invoke it. If not, functional requirements can not be achieved. It is in providers interest that the service they produce is always available for consumers. In order to offer the service in the most reliable way, services should be designed as *stateless*. Statelessness enables services to be replicated easily because there are no internal states to maintain. The replicas do not need to know what another instance is doing.

Modifiability (Clements, Kazman, and Klein) is the ability to make changes to a system quickly and cost-effectively. Loose-coupling is one of the key aspects of the SOA. This is achieved by implementing services as self-contained components that can be accessed via cohesive interfaces. This enables low costs when making changes to implementations and therefore increases the modifiability of the whole system. Once the interfaces have been decided it can be challenging to change them because once published they might be used by many applications.

Testability is the degree to which a software artifact (i.e., a software system, software module, requirements or design document) supports testing in a given test context (*Wikipedia - Software testability*). Testing a SOA-based system can be challenging for multiple reasons. Access to the source code and log files or outputs in general can be restricted if there are external service providers. If services are discovered at run-time or different providers may be used in some cases and platforms may vary. These kind of issues may cause extra effort in order to make system testable.

Usability is a measure of the quality of a user's experience in interacting with information or service (O'Brien, Merson, and Bass 2007). In a distributed SOA-based system, remote calls can take a relatively long time. The ability avoid these kind of delays can be challenging in SOA-based systems.

Scalability refers to the ability of the system to function well in cases where it is changed

in size or volume, for example, system has to serve larger amount of consumers without any major performance degradation. There are multiple strategies to gain better scalability: *horizontal scaling* (add load-balanced servers), *vertical scaling* (increase the capacity of a server), *stateless services* or *service scope*.

4.7 Implementation of SOA

Services are essential building blocks of an SOA styled software system. This section introduces the concept of web services as an implementation option of services.

Web services are often associated to SOA. Even though web services are not mandatory in SOA they to enable some principles of SOA such as *technology independent* end point (consumer application). The W3C defines *a web service* as follows (Web Services Glossary):

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

The problem with this definition is that it assumes SOAP-messages are used when interacting with *the web service*. According to (Hao 2003) *a web service* can be thought as an SOA with two following constraints:

1. Interfaces must be based on Internet protocols such as HTTP, FTP, and SMTP.
2. Except for binary data attachment, messages must be in XML.

This definition does not force SOAP to be the only way to interact with web services. Following subsections cover to possible ways to produce web services: REST and SOAP.

REST is an architectural pattern for networked systems that is derived from several of the network-based architectural patterns (Fielding 2000). It is not a standard specification or technology. The term *REST* stands for *Representational State Transfer*. It was described by Roy Fielding in his Ph.D. dissertation (Fielding 2000). One of the key concepts of REST is *a resource*. Any information that can be named is *a resource*. Every resource has an URI (Uniform resource identifier) that is a unique identification of it. *A resource* can have zero or multiple representations. *A representation* is the current or intended state of *a resource*.

REST is potential architectural pattern for implementing an SOA-based software system. REST has common constraints with an SOA and it also promotes many of the SOAs quality attributes. When these constraints are applied as a whole, the emphasize scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems (Fielding 2000).

REST promotes quality attributes such as *reliability*, *modifiability* and *scalability*. *Reliability* is achieved by the stateless nature of REST. REST presume that the client provides all necessary information for monitoring system in the request in order to monitoring system to understand it. This makes it easier to recover from errors. REST server components can quickly free resources because there are no internal states to maintain and therefore improves scalability (Fielding 2000)

Modifiability is achieved by the uniform interface constraint. The uniform interface decouples server side implementations from clients. This allows implementation to be changed without any need to make changes on client side (Fielding 2000).

Web services can be implemented by using REST. The HTTP protocol can be used as uniform interface for services. Its benefits are simplicity and platform independence. Let us assume that there should be a REST interface for a web service that offers basic CRUD functionality for client data. Different HTTP methods can be used for accessing and manipulating the representations of the client data. This is illustrated in Table 3.

In the examples, representations are identified with URLs. For example the client can create a new representation of a customer by sending a HTTP PUT request to the server with specified

Method usage		
Method	Action	URL example
GET	Retreive representation	https://api.example.com/customers/5
POST	Create a representation	https://api.example.com/customers/
PUT	Sets a state of a representation	https://api.example.com/customers/5
DELETE	Delete a representation	https://api.example.com/customers/5

Table 3. Examples HTTP method usage in REST-based application.

URL and appropriate data. Later this representation can be retrieved by sending a HTTP GET request to the server with the URL that has all the necessary information in order for customer service to derive the right representation. In the table above, URL has an id number attached to it for identifying right representation.

SOAP is an acronym standing for *Simple Object Accessing Protocol*. It is is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment (Gudgin et al. 2007). It was originally developed by Microsoft in March 1998 and later in co-operation of multiple organizations (*Exclusive .NET Developer's Journal "Indigo" Interview with Microsoft's Don Box*). Its advantages are language and platform independence. It is also simple and extensible. In 2003 it became a W3C recommendation.

SOAP message is based on XML and works like an envelope that carries its letters. It has to contain following elements: *an envelope* that identifies the XML document as a SOAP message; *a header* element, *a body* element and *a fault element*. A simple SOAP message is illustrated in the following example (*SOAP Tutorial - SOAP syntax*):

```
<?xml version="1.0"?>
  <soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle=
      "http://www.w3.org/2001/12/soap-encoding">
    <soap:Header>
    ...
```

```
</soap:Header>

<soap:Body>
...
  <soap:Fault>
...
  </soap:Fault>
</soap:Body>
</soap:Envelope>
```

The transportation of SOAP messages is done by using existing protocols, such as HTTP and SMTP. In the context of web services SOAP has a strong relation to a XML format called *web services description language* (WSDL), developed by Microsoft and IBM in 2000. WSDL describes web service's interface. In practice a WSDL file describes available services for a client, what the web service offers, what kind of message exchange should be practiced and what kind of data is expected (*Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*; Curbera et al. 2002).

5 Implementation

In most people's vocabularies, design means veneer. It's interior decorating. It's the fabric of the curtains of the sofa. But to me, nothing could be further from the meaning of design. Design is the fundamental soul of a human-made creation that ends up expressing itself in successive outer layers of the product or service.

–Steve Jobs

In this chapter the construct of this thesis is presented. The objectives and requirements for the construct are presented in Section 5.1. The development process and essential terminology is introduced in Section 5.2. Section 5.3 gives a brief introduction on how the construct operates. In section 5.4 the actual implementation is presented. Implementation issues are presented in Section 5.5.

5.1 Objectives and Requirements

The objective of this thesis is to create a prototype software that is implemented by using the EBI architecture pattern and following the principles of SOA pattern. The prototype should give an insight whether this implementation model will work on real-life scenarios. For the sake of simplicity, the prototype implementation should provide basic data manipulation services such as creating new data records, reading data, updating data, and deleting data. The data itself consist of entities such as *vehicles*, *tasks* and *task events*. These entities were chosen because the optimization system uses similar entities.

In addition to the functional requirements, the construct has also non-functional requirements. Based on workshops by the members of the project group, three of the most important non-functional requirements were elicited. The requirements were *scalability*, *extensibility*, and *testability*. The following listing describes these requirements more specifically.

- **Scalability:** In this context, the scalability refers to the systems ability to serve multi-

ple customers concurrently.

- **Extensibility:** The architecture of the system should enable developers to add new functionalities to the system in a way that the backward compatibility is preserved. In this context, the backward compatibility means that the API concerning the old functionality should be left unmodified.
- **Testability:** The architecture of the system should enable automatic regression tests. It should take twelve hours or less to run all the tests to ensure daily verification of the system functionality.

5.2 Development Methods and Process

This section presents the essential terminology used in this chapter. The development process is also introduced.

5.2.1 Terminology

This section gives a short definitions for key terms used in this chapter. The terms and definitions for them are the following:

- **Epic:** An arbitrary collection of user stories used to group related functionality under a common narrative.
- **User Story:** A simple way to express requirements. It is usually one or two sentences long description stating *who* does *what* and *why*. The user story represents the requirement and acts as a promise of a future conversation between the customer and the developer (Alleman 2008).
- **Acceptance Criteria (AC):** A set of input/action output/response pairs which defines how the system functions properly in a given usage situation.

It should be noticed that the terms and definitions above are specific to the project that this thesis relates. Other definitions can be found for these terms in literature or other terms might be used to describe same things elsewhere.

5.2.2 Development Cycle

The construct of this thesis was developed by using the ATDD process described in Section 2.4. A typical development cycle consist of choosing *the user story* to work with, defining *acceptance criteria* for it, creating *unit test stubs*, defining *boundaries* (interfaces) and *interactor stubs*, writing unit tests according to acceptance criteria and implementing interactors. First developers choose a user story from an epic and start writing acceptance criteria for it. After the first draft, ACs are validated with the quality assurance team. If the ACs are missing something relevant or something needs to be edited, this phase gets iterated. If the ACs are accepted the second phase can begin. In the second phase, the developers create

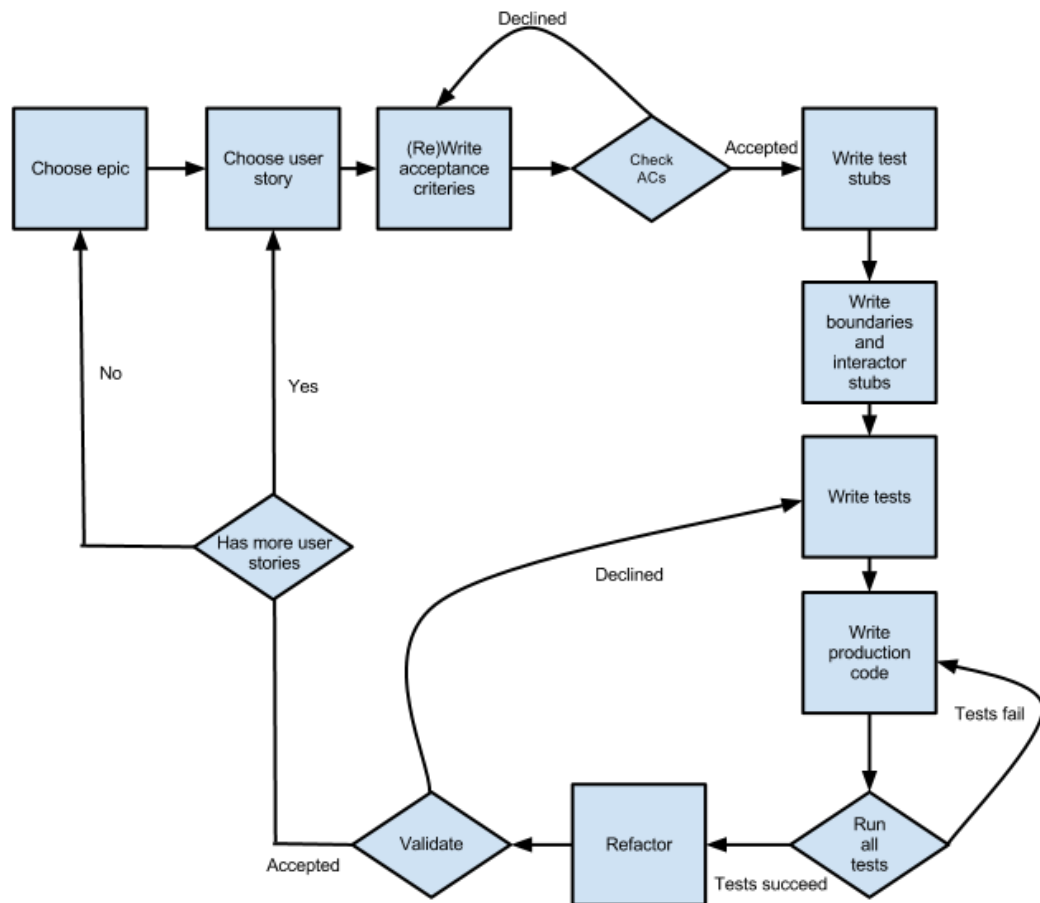


Figure 13. Development process.

unit test stubs according to acceptance criteria. These stubs are not yet implemented. Only the test method signatures are created. At this point all the unit test are in a failing state.

The reason the stubs are created before the unit tests are written is because two or more developers might be working with the same boundaries or interactors at the same time. It is important to agree names of the methods and classes in order to avoid creating two or more methods or classes that do same things as the one developed by another developer.

Third phase consists of defining and creating the boundary interfaces and stubs of classes (interactors) that will implement the boundaries. At this point boundaries might not be in their final state. Changes may occur when the actual implementation starts.

After the boundaries and class stubs are created the fourth phase can start. This phase consist of writing the unit tests according to the acceptance criteria. After that, tests should still be in failing state.

The fifth phase is the actual implementation of user stories. This phase is iterated as long as the necessary classes are implemented and all the unit tests are in passing state. When developers think that the user story is implemented, the person in charge of the quality checks that the implementation fullfills the acceptance criteria. If not, developers will correct these shortcomings. After the possible corrections, or if all the implementation fullfilled all ACs, developers choose another user story and start the same process again. This process is illustrated in Figure 13.

5.3 Main Functionality

This section describes briefly how the construct works. The basic CRUD functionalities are presented in the following subsections.

5.3.1 Creating and Updating Data

New data entities can be created by sending a request DTO to the interactor. Interactor then checks that the data is valid. If the data in DTO contains an id value greater than zero, it assumes that the request is actually an update request for an entity with that id. If the id does not exist the interactors send an error response to the client. If the id is zero, the interactor reasons that client want's to use "create" functionality so it creates a new entity from the

received data and sends an response to the client that contains an id for the newly created entity.

5.3.2 Retrieving and Deleting Data

Data can be retrieved by sending a request that contains a list of ids of the entities. If each of the requested ids is found from the database, the application sends a list of those entities to the client. But if there is at least one mismatch, none is send back. The client receives an error message that indicates the cause of the failure.

Deletion of data works same way as retrieving it, by sending a list of ids to the interactor. The response message is although different. In a case where everything goes as planned, that is to say each id has its correspondense in the database, a simple notification response is sent to the client that indicates a success of the requested operation.

5.4 User Stories and Components

This section presents the actual implementation of the construct. First the user stories are briefly introduced. After that interactors, boundaries and entities created for this construct are presented. Also the test user interface and test database are introduced in this section.

5.4.1 User Stories

The functionality requirements in this theses included one epic with eight user stories (US). The theme of the epic was basic data manipulation. User stories are presented briefly in the following listing.

1. **As a service user, I want to a add set of vehicles.** The system should be able to create new vehicle entities based on the received set of VehicleDTOs. This also includes valiadation of received DTO data. The system should notify the user with an error message if the received data was not valid.

2. **As a service user, I want to remove a set of vehicles** The system should be able to set the state of the vehicle as inactive based on the received list of vehicle ids. The system should notify the user with an error message if the received data was not valid.
3. **As a service user, I want to set vehicle properties.** The system should be able to alter the data of existing vehicle entity according to the received VehicleDTO. The system should validate the received DTO and notify the user if property values were not valid or the vehicle did not exist.
4. **As a service user, I want to set task properties.** The system should be able to alter the data of existing task entity according to the received TaskDTO. The system should validate the received DTO and notify the user if property values were not valid or the task did not exist.
5. **As a service user, I want to add a set of tasks.** The system should be able to create new task entities based on the received set of TaskDTOs. This also includes validation of the received DTO data. The system should notify the user with an error message if the received data was not valid.
6. **As a service user, I want to remove a set of tasks.** The system should be able to set the state of the task as inactive based on the received list of task ids. The system should notify the user with an error message if the received data was not valid.
7. **As a service user, I want to unassign a set of tasks.** The system should be able to unassign a set of tasks of the vehicle according to received list of task ids of the tasks that are assigned to the vehicle. The system should also check that the tasks are assigned to the vehicle before they can be unassigned. If the list contains an id of a task that is not assigned to the vehicle, an error message should be sent to the user.
8. **As a service user, I want to set task event sequence for the vehicle.** The system should be able to add a sequence of task events to the vehicle according to the received list of task sequence ids. The system should make sure that the sequence is valid. If it is not, an error message should be sent to the user.

5.4.2 Interactors and Boundaries

The following listing presents interactors with request and response boundaries that are related to that specific interactor. The number after the name of the interactor in the following listing indicates the number of the user story presented in Section 5.4.1 The asterisk symbol indicates that the interactor can relate into multiple user stories.

- **AddTaskSequenceInteractor (8)**. This interactor implements `IAddTasksToVehicleRequest` and is responsible for adding valid task event sequences for a vehicle. If vehicle has a sequence, it can be replaced by a new one or it can be edited by inserting another sequence to it. The interactor uses `IAddTasksToVehicleResponse` as a response channel.
- **AddTasksInteractor (4, 5)**. The interactor is responsible for adding tasks and task events into the database. It implements `IAddTasksRequest` interface and uses `IAddTasksResponse` for sending responses back to the user. Initially there were a separate interactor for adding task events into the database. It turned out that one interactor is enough because tasks and task events can not exist without each other. This is why tasks and task events are added at the same time. This also reflects to the DTO classes. The `TaskDto` class is composite class that contains two `TaskEventDto` objects.
- **AddVehiclesInteractor (1, 3)**. The interactor is responsible for adding a vehicle or multiple vehicles into database. It implements `IAddVehiclesRequest` interface and uses `IAddVehiclesResponse` for sending responses back to the user.
- **DeleteVehiclesInteractor (2)**. The interactor implements `IDeleteVehiclesRequest` interface and is responsible for removing vehicle entities from the database. The interactor uses `IDeleteVehiclesResponse` as a response channel.
- **GetTaskEventsInteractor (*)**. The interactor is used in multiple use cases. It is especially used in unit tests where it is important to retrieve saved task events to be sure that they were saved correctly. This interactor implements `IGetTaskEventsRequest` and uses `IGetTaskEventsResponse` interface to send response messages.
- **GetTasksInteractor (*)**. As above, this interactor is used in multiple use cases. It can

be used to retrieve saved task entities from the database. This interactor implements `IGetTasksRequest` and uses `IGetTasksResponse` interface to send response messages.

- **GetVehiclesInteractor (*)**. The interactor implements `IGetVehiclesRequest` and uses `IGetVehiclesResponse` interface to send response messages. Its purpose is to retrieve saved vehicle entities from the database. In unit tests this interactor is used to test that sent vehicle entities are saved correctly.
- **RemoveTaskSequenceInteractor (6, 7)**. The interactor is responsible for removing a task event sequence or a part of it from a vehicle. It implements `IRemoveTaskSequenceRequest` interface and uses `IRemoveTaskSequenceRequest` to send response messages. As the numbers indicate this interactor implements two different user stories. During the development process it was noticed that the interactor of the user story number 6 can be integrated into this interactor.

5.4.3 Entities, Data Transfer Objects and Data Objects

In the following listing entities, data transfer objects (DTO) and data objects are presented. In general DTOs are used when making request and sending responses. Data objects are used when using store interfaces, for example, retrieving or saving data into database.

- **Entities**. There are three kind of entity classes in this construct; `Vehicle`, `Task` and `TaskEvent`. The `Vehicle` entity represents a vehicle that can have different kind of tasks. `Task` and `TaskEvent` entities can not exist without each other. Each `Task` is composed from two `TaskEvent` entities; pick and delivery entities. When `Task` entities are assigned to a certain `Vehicle` the `TaskEvent` entities form so called *task event sequence*. A task event sequence is a set of `TaskEvent` entities that have an order. This sequence specifies what the `Vehicle` should do and in what order.
- **Data transfer objects and data objects**. Four different kind of *data transfer* objects exists in this construct; `VehicleDto`, `TaskDto`, `TaskEventDto` and `StatusDto`. The `TaskDto` consists of two `TaskEventDto` objects. Task events are never sent without the tasks that they are part of. The `StatusDto` is always sent from core the application to the UI. It can be sent as a separate message or as a part of

VehicleDto and TaskDto.

- *Data objects* are data transfer objects that are used in the communication between the database and the software. Data objects use different kind of naming convention than the data transfer objects described in the previous paragraph. Instead of the “*Dto*” postfix, they use “*Data*” postfix. There are three kind of data objects in this construct; VehicleData, TaskData and TaskEventData.

It should be noticed that even though it seems that each of the entity classes have corresponding data transfer object and data object, it is not required. Entity classes can be changed over time or even removed. Changes in the entity classes do not have any affect on DTOs or data classes because entities are only visible to the core application. In other words, the entities are in a way decoupled from data transfer objects and data objects. The DTOs and data classes are visible to everyone so when these classes are once made they should always remain the same to enhance backwards compatibility.

5.4.4 User Interface and InMemoryDb

The user interface (UI) in the construct was implemented with NUnit test framework. A simple UI implemented all the response boundaries. Its only purpose is to emulate a real UI in the test cases.

In order to test full functionality of the construct, there must be a plug-in that implements the so called *store interfaces*. Store interfaces are a way to retrieve and save data. In this thesis the plug-in that implements interfaces mentioned before is called InMemoryDb. It keeps data in simple data structures, hence the name “*In Memory*”. This implementation is used in the test cases.

5.5 Issues

This section presents the major issues and problems encountered during the development process. The chosen solutions and reasons behind the choices made are explained.

5.5.1 Updating UI

One of the issues encountered was how to keep user interface (UI) up-to-date when creating new entities. The construct allows to create multiple entities with one request. The problem occurs when the UI should get ids for created entities. The biggest question is how to map ids to entities on UI. Three different options were considered: first, the UI receives ids from return value of request method, second, the UI receives ids in some kind of data structure with response DTO, and third, the UI receives ids and all other data related to newly created entities.

One of the core principles of the EBI pattern is that there are different boundaries for request and response messages. This means that after the request is made the UI starts to listen the response channel. If the ids are send as a return value of request method the UI should also listen possible return values from the request channel. This is confusing and it also violates the design guidelines of the EBI pattern (see Chapter 4). This is why this option was discarded.

If the ids are retrieved by response message, the design follows EBI principles. There is still a problem to be solved: how to map retrieved ids with the entities in the UI. One solution is to send the saved entities back to UI with ids. This allows the UI to discard the old entities and retrieve all the necessary data from service. This is in some sense a “refresh method” as the old entities in the UI are erased and replaced every time. The drawback of this method is that the amount of response data grows significantly. The increased amount of transferred data was the reason why this option was discarded.

As said in the paragraph above, the problem with retrieving only ids, instead of all data related to entities, is that there must exist some kind of rules how to map these new ids with entities in the UI. One way is to expect that the UI keeps its entities in a data structure that maintains some order between the entities. Now the response message can include a

data structure that contains ids in the same order that the request message. The UI can now assume that the id in the first cell of retrieved data structure is for the entity that lies in the first cell of the corresponding data structure of the UI. One of the benefits of this approach is that response messages include less data than in the method discussed in the paragraph above. The obvious drawback of this method is that it requires more complex logic than the “refresh method”. This method was chosen because the benefits of the minimization of transferred data was considered to be greater than the drawback of extra complexity in the implementation logic.

5.5.2 Initialization of Interactors

One of the issues is how to initialize interactors without creating extra dependencies. All the interactors of this construct use store and response boundaries. Store boundaries are used by interactors to save and retrieve entities from the database. Response boundaries are a way to send responses back to the UI. In the first version of the construct, the implementations for these boundaries were done explicitly in the constructor of the interactor. This approach worked fine but it was complex. Another problem in the described solution is that the implementing classes are defined statically. It would be better if the implementing classes for boundaries or interfaces could be selected dynamically at run-time, for example, with a given dependency interface, or by configuration file. The implementation could be chosen according to the purpose of the task in hand. For instance, it can be pragmatic to use different implementations for testing and the production code.

One technique to decouple the boundaries from the implementing classes is called *inversion of control* (IoC). More about the inversion of control can be read from (Fowler). There are several ways to implement IoC such as *factory pattern*, *service locator pattern* (see Section 2.6) and *dependency injection* (DI) (see Section 2.6). In this case, the dependency injection approach was chosen. The main reason for this was that there was available a lightweight dependency injection container called *the Unity Container* or just *Unity* (*Unity Container* 2012). With the Unity the boundary instantiation can be done at run time by defining implementing classes for boundaries in `App.config` file. The following code illustrates the instantiation without a dependency injection container.

```

// ..
private readonly IVehicleStore vehicleStore;
private readonly IAddVehiclesResponse response;

public AddVehiclesInteractor(IVehicleStore vehicleStore,
    IAddVehiclesResponse res)
{
    vehicleStore = vehicleStore;
    response = res;
}
// ...

```

With a dependency injection container, the above example can be written as follows.

```

// ..
private readonly IVehicleStore vehicleStore;
private readonly IAddVehiclesResponse response;

public AddVehiclesInteractor(IAddVehiclesResponse res)
{
    var container = new UnityContainer();
    container.LoadConfiguration();
    vehicleStore = container.Resolve<IVehicleStore>();
    response = res;
}
// ...

```

Observe that there are less parameters in the method signature because the `App.config` file defines which implementations should be bound into `IVehicleStore` boundary. Now the implementing class can be changed by changing only the `App.config` file. The imple-

menting class is changed in all the places where the Unity is used. This makes the application more maintainable as the implementation of the `IVehicleStore` is resolved at run-time.

5.5.3 Messaging Model

One of the issues was to decide what kind of request and response models should be used in messaging. Method calls from a client application to a core application can be a performance bottleneck if the number of calls increases dramatically. That is why in the construct of this thesis the number of calls from delivery mechanism side (see Chapter 4) to core application are minimized by using so called dump data structures as messages.

In the first version of the construct the dump data structure was a *.NET* `List` object that contained `Dictionary` objects with strings as keys and values. The problem with this solution was that it required a lot of parsing and type casting in the interactors. The code was also quite contained large amounts of boilerplate code and hard to maintain.

Better solution was to use the *Data Transfer Object (DTO)* (see Section 2.6) design pattern. The purpose of the DTO pattern is to reduce the number of method calls by transferring more data on one call. This is achieved by creating a Data transfer object that holds the data. The construct of this thesis uses several different DTOs such as `VehicleDto`, `TaskDto`, `TaskEventDto` and `StatusDto`.

The data transfer objects are created by using the *Assembler* design pattern. The assembler is an object that creates data transfer objects from business objects from entity objects and vice versa. This separates parsing logic from interactors and makes code more readable. The drawback of using the Assembler pattern is that if an entity or DTO needs changes the changes must also be done to the Assembler as well. The implemented DTO and Assembler pattern is illustrated in Figure 14.

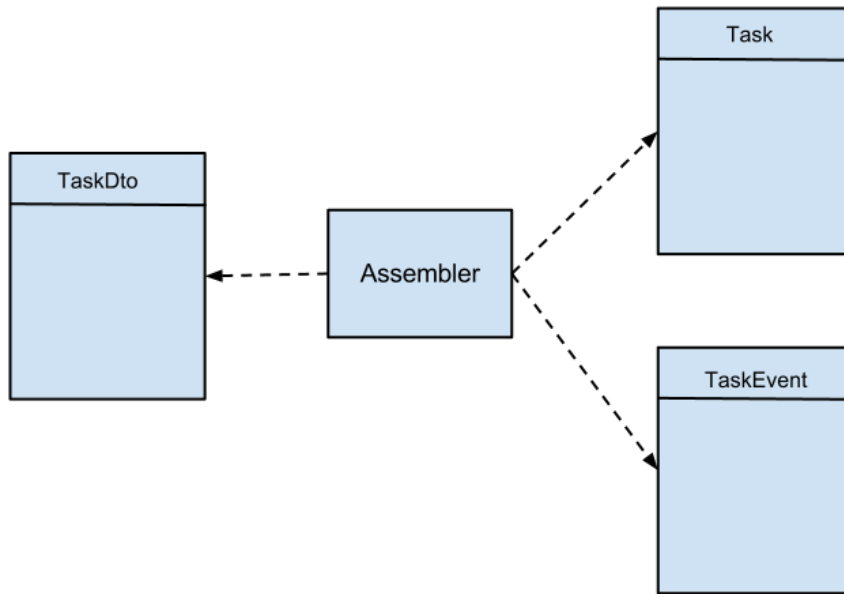


Figure 14. The implemented Data Transfer Object and Assembler pattern.

5.5.4 Relations in DTOs

As mentioned in Section 5.5.3, the number of method calls can be kept low by using the DTO pattern. In some cases the data transfer object can be fairly large and complex even though it does not contain business logic. This issue occurred in the first version of this construct. A `Vehicle` entity can have none or multiple `Task` entities. The `Task` entity always contains two `TaskEvent` entities; pick up and delivery task events. This relation is illustrated in Figure 15.

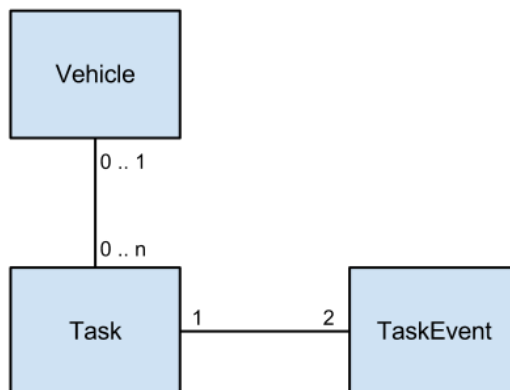


Figure 15. Relations Between Entities.

This kind of relation means that the DTO complex. This observation is based on empirical prototyping. The Assembler of the first version of the construct used `System.Data.DataSet` object to build the DTO. The implementation was able to preserve the same relation between the entities as shown in Figure 15. Even though it did preserve the relations between the DTOs, it was considered to be too complex to be easily maintained.

The idea of keeping the relation between entities was abandoned in the final version of the construct. In the final version, the relation can be retrieved by the UI by making multiple method calls to several different interactors. The `Vehicle` entity contains an array of `Task`

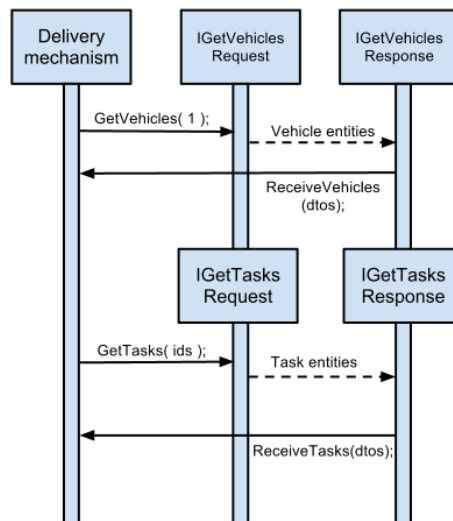


Figure 16. Retrieve vehicles and tasks related to the vehicle.

entity `ids`. By calling the `GetVehiclesInteractor` via `IGetVehiclesRequest` boundary, the UI gets `ids` of the tasks related to vehicle retrieved. With these `ids` the UI can make another call to `GetTasksInteractor` via `IGetTasksRequest` boundary. The retrieved `TaskDto` objects contain two `TaskEventDto` objects. Thus instead of making one method call, two is now needed. This process is presented in Figure 16.

The benefits of this approach is that the DTOs are simpler and therefore the object building logic in the Assembler is simpler. This is also lighter because if the user of the UI wants simply to view a list of vehicles, it does not necessarily require information about the tasks and task events. List of related task `ids` suffices. If the user wants to navigate to task view the UI makes another method call to retrieve task and task event data.

6 Evaluation

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.

–Winston Churchill

This chapter evaluates the implemented prototype architecture described in Chapter 5. Evaluation is done from three viewpoints: *purity of EBI pattern in the construct, how well the SOA principles were followed, and how well the implementation fulfills the project specific requirements.*

6.1 Selected Evaluation Method

As mentioned in Chapter 1, the goal of this thesis is to create a prototype software that is implemented by using the EBI architecture pattern and following the principles of the SOA pattern. As mentioned earlier, the developed system is a prototype. Therefore the system is in the middle of the software life-cycle. According to Table 2 that leaves three possible evaluation methods: checklist, scenarios and metrics. It should be noted that some early phase evaluation was made before this particular construct was under development. The early phase evaluation was a prototype of the construct of this thesis. As a result, it gave more insight for developers on how to implement the real prototype using the EBI pattern.

Again referring to the fact that the construct is a prototype indicates that it is used as a reference when making decision on which kind of architectural approach should be chosen when implementing the system. One of the architectural goals was to build an architecture that requires minimum maintenance efforts and responds well to possible changes such as adding new services. Checklist method must be excluded because there is no pre-existing domain-specific checklist for evaluation.

Metrics offer a good way to measure cohesion and coupling levels of the architecture as described in Section 3.6. Properly used it might reveal unwanted dependencies between the

architectural components and even point places of high complexity. Those could indicate potential challenges concerning the maintainance and expansion efforts. The problem with the metrics in this case is that results indicate the current state of the architecture. It does not necessarily tell what happens if a new service is added to the system, how much effort that might cause or is it even possible to do. It is difficult to measure these kind of quality attributes because there are no clear universal formulas to count values for attributes like flexibility or maintainability.

Scenarios offer a method to evaluate how the architecture responds to possible changes in the future. They are also better in reviewing ill-defined quality attributes such as those mentioned. Scenarios put these attributes in more specific circumstances by capturing system use context (Kazman et al. 1996). Based on presented arguments the evaluation in this thesis is done by using scenarios.

The precise evaluation method of choice is called *scenario-based architecture analysis method* (SAAM). It was introduced by Kazman et al. (Kazman et al. 1994) in 1993. It has been proved to be useful in the evaluation of the quality attributes such as *modifiability*, *portability* etc. It has been argued that *architecture trade-off analysis method* (ATAM) (Kazman, Klein, and Clements 2000) is improved version of SAAM because it captures attributes, such as *performance* and *reliability* in more detail (Hammer, Ionita, and Obbink 2002). Even though SAAM has some weaknesses such as it does not give clear quality metrics, it is a quite simple method and it does not require as much detailed technical knowledge as ATAM. Therefore it is suitable in this particular case. A more detailed analysis of the strenghts, weaknesses, and other attributes of different scenario-based evaluation methods can be found from (Hammer, Ionita, and Obbink 2002).

6.2 Evaluation With SAAM

The evaluation of the architecture of the developed system prototype is presented in this section. The evaluation was done by using the *software architecture analysis method* (SAAM).

6.2.1 Candidate Architecture

The architectural description was made in five iterations. The first version included all classes and interfaces that the prototype has. It was too complicated to be used as a representation of an architecture. In the second version the `InMemoryDb` and `UiMock` was left out because they are not part of the core application. Third iteration combined all interfaces related to vehicles as two interfaces: `IVehicleRequest` and `IVehicleResponse`. The same thing was made for task sequences, tasks and task events. As a result of the fourth iteration, the DTOs and the data entities were combined into groups: `DTO` and `DataEntities`. In the fifth iteration `StatusDto`, `StatusMessage` and `StatusCode` were left out because they are considered trivial in respect to systems functionality.

The fifth and the last iteration combined all interactors related to vehicles as `VehiclesCRUD` and all interactors related to tasks, task events or task sequences as `TasksCRUD`. The final version is illustrated in Figure 17. This static and relatively simple architectural representation captures relationships between the components as the Section 3.7.1 suggests (see Appendix A).

6.2.2 Scenarios

Scenarios used in this thesis were defined by the project group. The group consisted of both business and technology oriented people. Almost 30 different scenarios were elicited. Seven of the most important scenarios were chosen for the final evaluation. Those seven scenarios were grouped in to two groups: *phase one* and *phase two*. The ones in the phase one are scenarios that were used in the evaluation of the concrete construct developed. The scenarios of the phase two were used in the evaluation of the potential architecture, not the prototype architecture developed in this thesis.

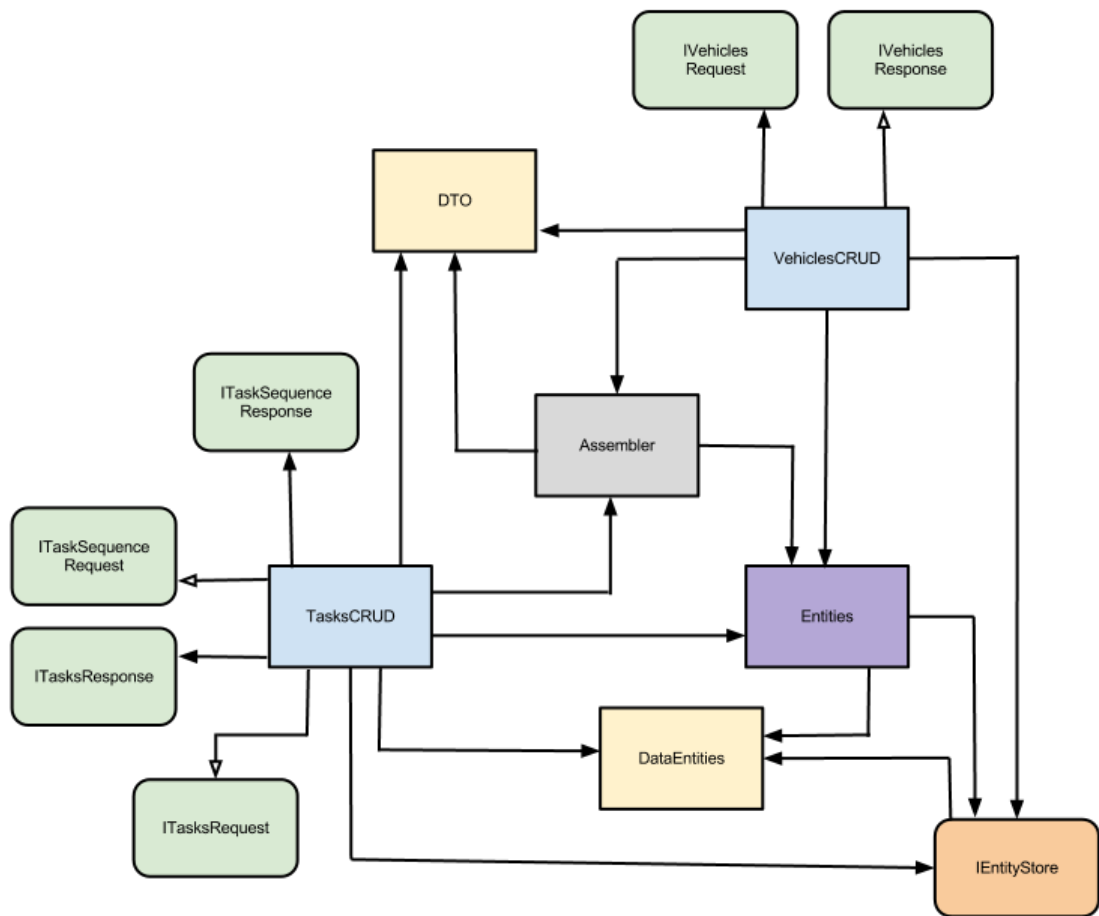


Figure 17. Architectural representation of the prototype using a simple dependency notation.

Scenarios: phase one

- *Extend the Vehicle class.* The Vehicle class is extended by adding the height property into the class.
- *Install the system to Linux.* By default, the system runs on Windows but it may have to be installed to Linux.
- *Integrate an ERP system into the system.* ERP system is integrated into the system so that it can use the API of the system.
- *Add optimization functionality into the system.* Optimization functionality is added

into the system so that it is available for consumers to use.

Scenarios: phase two

- *The amount of the map data grows significantly in a short period of time.* The amount of the map data that is to be stored into the database, grows significantly in a short period of time.
- *Separate the optimization core from the overall system.* The optimization core is separated from the overall system as a stand-alone module, which provides optimization services.
- *Separate the map module from the overall system.* The map module is separated from the overall system as a stand-alone module, which provides road data services.

6.2.3 Evaluation of the Scenarios

The evaluation of the scenarios revealed four *indirect*, two *direct*, and one scenario that requires further investigation. The direct ones included following scenarios: *integrate ERP system into the system*, and *the significant growth of map data in a short period of time*. The integration of an ERP system does not require any changes to the architecture. The ERP system can use the provided API directly if it is implemented with .NET. If the ERP is not implemented with .NET, the API can be used by creating a new component between the ERP and the system that works as an interface adapter. According to the principles of the SOA pattern, services should be made available in spite of the platform of the consumer application (see Section 4.5). One way to achieve technology neutrality in this case, is to use an open source web services framework called Service stack (ServiceStack 2012).

Two of the indirect scenarios are from phase one and two of them are from phase two. The extension of the `Vehicle` class causes changes to related DTO classes and to `Assembler` class. The addition of the `Optimize` interactor requires request and response boundaries so they must be created.

RESULTS OF SCENARIO EVALUATION		
Scenario	Direct/ Indirect	Required Changes
Extend the Vehicle class	Indirect	Modify Vehicle, VehicleDto, VehicleData and Assembler classes.
Install the system to Linux	Direct/ (Indirect?)	May require removal of the dependencies with the Unity.
Integrate ERP system into the system	Direct	-
Add optimization functionality into the system	Indirect	Create the request and the response boundaries for the Optimize interactor.
The amount of the map data grows significantly in a short period of time	Direct	-
Separate the optimization core from the system	Indirect	Modification of the Optimize interactor and creation of boundaries for the interaction between the separated optimization core and the system.
Separate the map module from the system	Indirect	Modification of the Optimize interactor and creation of boundaries for the interaction between the separated map module and the system.

Table 4. Results of the scenario evaluation.

The separation of the optimization core from the system requires modifications to the `Optimize` interactor. The application-specific computation logic must be removed from it to the separate optimization module. After the changes, the `Optimize` interactor works as a message passer between the application and the optimization module. The optimization module requires its own boundaries. The separation of the map module from the system requires similar changes as the separation of the optimization core. Figure 18 represents the architecture before and after the separation of the modules.

The developed architecture prototype was analysed with MoMA (Xamarin Inc.) in order to find out if it can be used with Mono. An open source, cross-platform, implementation of C# and the CLR that is binary compatible with Microsoft.NET (Xamarin Inc.). Put differently, it enables C# programs to be run in Linux environment (Xamarin Inc.). The analysis

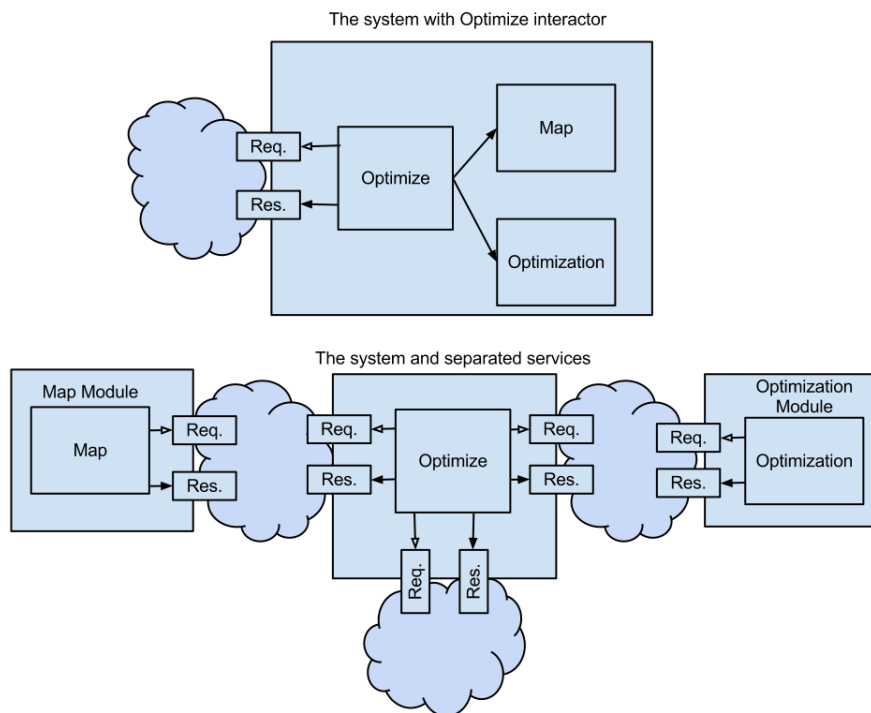


Figure 18. The architecture before and after the separation of the services.

revealed potential compatibility issues with the Unity framework. The results are presented in Appendix C. It should be noted that the issues should be verified in order to find out if they are relevant. As it is said in (Xamarin Inc.): “While MoMA can help show potential issues,

there are many complex factors that cannot be covered by a simple tool. MoMA may fail to point out areas that will cause problems, and may point out areas which will not actually be an issue.” Therefore these issues require further research such as trying to run the system on Mono. The results of the evaluation are summarized in Table 4.

6.2.4 Scenario Interactions

Table 5 show the number of changes required in each class. The total number of changes was quite low which indicates a good architectural design. Changes in the `VehicleDto` and into the `VehicleData` classes were expected. The only change those classes require is the addition of the same properties that are added to the `Vehicle` class. The deployment into Linux environment may cause some minor changes to the interactors because the dependencies with the Unity framework must be removed. This also means that the Unity must be replaced with other dependency injection framework. The use of dependency injection framework inside the interactor also indicates a design error because interactors should not have any dependencies to any third party libraries or frameworks as mentioned in Section 4.2. Therefore, the direct use of dependency injection frameworks should be removed from interactors. The injection should be done in a some kind of interactor initializer class.

The number of changes required in the `Optimize` interactor was also expected. In the EBI pattern, the interactors are bound to the boundaries. Therefore if a new interactor is added to the application, it requires at least two boundaries to be added. One for the request messages and another one for the response messages. This is counted as a single change. Three changes come from the phase two scenarios: *separate the optimization core from the system* and *separate the map module from the system*. Two of them are caused by the boundaries. The `Optimize` interactor requires boundaries in order to communicate with separated module. The fourth change is caused by the changes in the application specific logic of the interactor. Instead of optimizing its job is to operate as a message passer between the application and the separated modules.

The number of required changes to the `Assembler` class an to the `Vehicle` class was unexpected. The job of the `Assembler` class is to convert DTOs and data classes to entities

SCENARIO INTERACTIONS	
Class/Module	Number of Changes
Optimize interactor	4
Vehicle	3
VehicleDto, VehicleData, Assembler	1 each
All the interactors (11 total)	1 each possibly

Table 5. Scenario interactions.

and vice versa. Therefore, the expected number of changes is two instead of one. The changes expected to the `Vehicle` class were the addition of properties and minor changes to the validation logic. Still, the number of changes is three. This indicates a design error. The third required change is result from the `ParseVehicle` method of the `Vehicle` class. The method takes a `VehicleData` object as an argument. The `Vehicle` class is an entity and therefore it should know nothing about the `VehicleData` class. This violates the principles of EBI and clean architectures as described in Chapter 4. The design error illustrated in Figure 19.

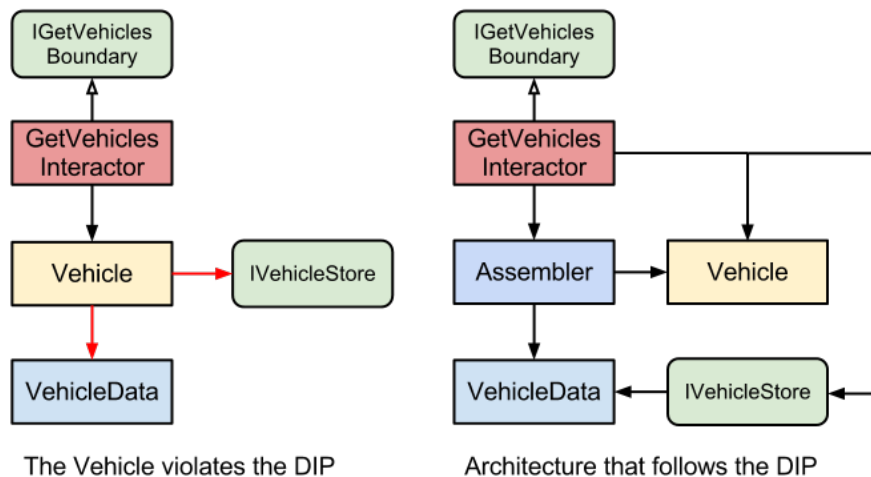


Figure 19. Architectural design error.

Red arrows on the left-hand side of the figure presents dependencies that should not exist. The right-hand side show how the dependencies should be.

6.2.5 Weights of Scenarios and Interactions

The required amount of time and work to make changes to the architecture is low. The extension of the `Vehicle` class scenario requires that the properties to be added to the `Vehicle` are added to `VehicleData` and to `VehicleDTO` as well. The changes required to the validation logic depends on the nature of the properties. The dependency between the `Vehicle` and the `VehicleData` can be removed by moving the `ParseVehicle` method to the `Assembler` class. These changes can be made in a matter of days.

The amount of time and work required to the separation of the optimization core and the map module depends on how fast the developers are able to remove dependencies between the modules. This is the most troublesome part of the two scenarios. After the separation of the modules, the boundaries must be defined and implemented by the interactors. This can be done in a couple of days.

The possible removal and substitution of the Unity framework requires research in order to find a proper framework as a replacement. Depending on the framework, the work required should be a week at maximum.

6.3 Evaluation of Service-Oriented Architecture

One of the architectural goals of the prototype architecture of this thesis was to follow the principles of SOA. The principles of SOA were presented in Section 4.5. The principles of *service abstraction* and *published service interfaces* were achieved by hiding the interactors and entities behind the boundaries.

All the interactors are stateless. They are also *idempotent* (see Section 4.5) except the interactors that are responsible of creating new data representations. This means that the principle of *statelessness* fulfills.

The principle of *formal contracts* is not fulfilled in the prototype architecture. No contracts were defined in the scope of this thesis.

It should also be noticed that the use of the services provided by the prototype is bound to

the .NET Framework. In order to make services available in a technology neutral way, the prototype requires an extra layer over the boundaries. One potential option is the *Service stack* (ServiceStack 2012). The Service stack was tested with the prototype and it proved to be a workable technology. The Service stack integration is not in the scope of this thesis.

6.4 Results and Observations

The prototype architecture fulfills most of the requirements that were set. The modular structure supports the service-oriented design. The modular structure also supports the scalability and the extensibility requirements described in Section 5.1. Multiple customers can be served by replicating services. The concrete implementation for this does not exist yet and therefore requires more research. Addition of the service to the application has no impact on the existing boundaries (or API). As said in Chapter 4, no changes to the boundaries or to the interactors should be expected unless the use cases change.

The modular structure of the EBI pattern and the fact that the application is decoupled from external agencies such as the UI, the database and web server supports testability. The functionality of the prototype can be tested any of the previously mentioned parts. This makes testing faster. It is also easy to automate the tests.

One of the disadvantages of the EBI is that the extensions of the functionality requires quite a lot of work. The extension of functionality in a form of adding a new interactor requires three things. Firstly, creation of the interactor class. Secondly, creation of the boundaries. And thirdly, possible creation of DTO classes. Without a careful design, the amount of source code can grow drastically.

The DTOs used in the prototype require rethinking. This is because the changes to entities caused changes to the DTOs. One possible solution is that the properties in DTOs could be stored into a *hash map* or equivalent data structure as key-value pairs. If new attribute is added to the entity, the corresponding (if there is any) DTO class would not require any structural changes because the new attribute can be stored into the same structure as key-value pair. The use of previously mentioned data structure as an attribute storage increases the flexibility and backward compatibility of the system. This is because if the data structure

of the DTO contains old and new attributes, the DTO can be redirected to the interactor that knows how to handle new attributes. If the DTO contains only old attributes, the interactor that was originally created to handle that kind of messages can take the message.

In principle, the combination of EBI and SOA patterns has potential conflicts such as the EBI promotes testability, whereas the SOA makes testing difficult. The developed prototype showed that a good testability can be achieved by combining EBI and SOA patterns. Another potential conflict rises from the design principles. In the EBI pattern, the data is always in format most convenient to the inner circle. That is to say, the outer circles must adapt to the inner circles. In the SOA, services are designed in a way that they are in the most meaningful form to consumers. In the developed prototype, the EBI patterns dominates which means that client applications must adapt to the prototype.

EBI and SOA patterns have several common characteristics such as: *statelessness*, *service interfaces*, and *service abstraction*. In SOA pattern, the statelessness is one of the essential design principles as it is in the EBI pattern. In the prototype, the statelessness manifests as a use of DTOs. Interactors receive all the necessary data from the DTOs. Therefore, it is not necessary to maintain any state information between the service calls. In addition, both patterns separate the implementation from interfaces. The consumer is only aware of the interface (or boundary) it uses, not the actual implementation of the service.

The architecture of the prototype followed most of the design principles of EBI and SOA patterns. The combination of these two architectural patterns proved to be functional, though, some design errors were found. All the entity classes have a direct dependency with corresponding data classes. This violates the design principle of lower layer should not know anything about the upper layers (see Chapter 4). The possible compatibility issues with the Mono requires further research. Another design error was detected in interactor initialization as interactors had a dependency on the Unity framework.

7 Conclusion and Further Research

The construct of this thesis is an prototype application with basic data manipulation functionality related to the a vehicle routing application. The prototype application is designed according to the EBI pattern and guidelines of SOA pattern. The purpose of the prototype was to test if an service-oriented application can be builded by using the EBI pattern so that the requirements are achieved.

Entity-Boundary-Interactor (EBI) is an architectural pattern that decouples the application from the external agencies such as the user interface and the database. The design decisions concerning the UI and the database can be therefore deferred. One of the most important principles of the EBI pattern is the dependency inversion principle (DIP). It says that low and high level modules should depend on abstractions rather than details.

Service-oriented architecture (SOA) is a architectural pattern that gives guidelines and sets constraints about how to design an application in the form of services. A service is a way to access one or more utilities produced by the service providers. The service consumer can access these utilities by predefined and published interface.

The prototype was developed by using the acceptance test driven development (ATDD) process. The developed prototype fulfilled most of the requirements. The architecture proved to be flexible, extensible and testable. The architecture was evaluated using the simple architecture evaluation method (SAAM). Some design errors were discovered during the evaluation concerning the design principles of EBI pattern. These design errors can be fixed with minor changes to the architecture.

In order to get the maximum potential from the combination of EBI and SOA, a further research is required. SOA is essentially a architectural pattern for implementing distributed systems. Therefore, research must be made on how the EBI functions in distributed environment. The modular structure of the EBI pattern implies that it should work on distributed environment. The possible compatibility issues with the Mono requires further research.

The services of the prototype can be accessed by using the API provided. Because the prototype is developed using the .NET framework, the API is not technology independent. In order to make the API technology neutral, potential technologies for enabling the technology neutrality should be examined. One of the most promising technology is the ServiceStack.

This thesis presented two architectural patterns that were used in the development of the prototype application: EBI and SOA. The combination of those architectural patterns proved to be functional, though some design flaws were detected. In general, SOA and ABI patterns can be concluded to work well together.

Bibliography

Abowd, G., L. Bass, P. Clements, R. Kazman, L. Northop, and A. Zaremski. 1997. “Recommended Best Industrial Practice for Software Architecture Evaluation”.

Abowd, Gregory, Robert Allen, and David Garlan. 1993. “Using style to understand descriptions of software architecture”. *SIGSOFT Softw. Eng. Notes* (New York, NY, USA) 18, number 5 (): 9–20. ISSN: 0163-5948.

Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Alleman, Glen. 2008. *Acceptance TDD Explained*.
<http://www.methodsandtools.com/archive/archive.php?id=72>.
Accessed 5.2.2013.

Application and Design Patterns - The Service Locator.
<http://msdn.microsoft.com/en-us/library/ff648968.aspx>.
Accessed 11.2.2013.

Balzer, Yvonne. 2004. *Improve your SOA project plans*.
<http://www.ibm.com/developerworks/webservices/library/ws-improvesoa/>.
Accessed 10.1.2013.

Bass, L., P. Clements, and R. Kazman. 1998. *Software architecture in practice*. Boston, MA, USA:
Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-19930-0.

Boehm, B. 1986. “A spiral model of software development and enhancement”. *SIGSOFT Softw. Eng. Notes* (New York, NY, USA) 11, number 4 (): 14–24. ISSN: 0163-5948.

Briand, L.C., S. Morasca, and V.R. Basili. 1993. “Measuring and assessing maintainability at the end of high level design”. In *Software Maintenance ,1993. CSM-93, Proceedings, Conference on*, 88–87.

Brown, Alan, Simon Johnston, and Kevin Kelly. 2002.

Using service-oriented architecture and component-based development to build Web service applications.

<http://public.dhe.ibm.com/software/dw/rational/pdf/2169.pdf>.

Brownsword, Lisa, David Carney, David Fisher, Grace Lewis, Edwin Morris, Patrick Place, James Smith, Lutz Wrage, and B. Meyers. 2004. *Current Perspectives on Interoperability (CMU/SEI-2004-TR-009)*.

<http://www.sei.cmu.edu/library/abstracts/reports/04tr009.cfm>.

Accessed 11.1.2013.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-oriented software architecture: a system of patterns*. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0-471-95869-7.

Clements, P., R. Kazman, and M. Klein. *Evaluating software architectures: methods and case studies*. 2002.

Cockburn, Alistair. *The Pattern: Ports and Adapters*.

<http://alistair.cockburn.us/Hexagonal+architecture>.

Accessed 15.2.2013.

Coplien, J.O., and G. Bjørnvig. 2011. *Lean Architecture: for Agile Software Development*. Wiley. ISBN: 9780470970133.

Curbera, F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S.

Weerawarana. 2002. "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI". *Internet Computing, IEEE* 6, number 2 (): 86–93. ISSN: 1089-7801.

Curtis, Bill, Herb Krasner, and Neil Iscoe. 1988. "A Field Study of the Software Design Process for Large Systems". *Communications of the ACM* 31:1268–1287.

Dobrica, L., and E. Niemela. 2002. "A survey on software architecture analysis methods". *Software Engineering, IEEE Transactions on* 28, number 7 (): 638–653. ISSN: 0098-5589.

Donohue, P. 1999. "Software Architecture – 1st IFIP Conf. Software Architecture".

Enterprise Solution Patterns Using Microsoft .NET.

<http://msdn.microsoft.com/en-us/library/ff647095.aspx>.

Accessed 15.1.2013.

Erl, T., B. Carlyle, C. Pautasso, and R. Balasubramanian. 2012. *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*. The Prentice Hall Service Technology Series from Thomas Erl. Pearson Education. ISBN: 9780132869911.

Erl, Thomas. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0131858580.

Exclusive .NET Developer's Journal "Indigo" Interview with Microsoft's Don Box.

<http://dotnet.sys-con.com/node/45908>.

Accessed 14.1.2013.

Fielding, Roy Thomas. 2000. "Architectural styles and the design of network-based software architectures". AAI9980887. PhD thesis.

Fowler, M. *Catalog of Patterns of Enterprise Application Architecture.*

<http://www.martinfowler.com/eaCatalog/index.html>.

Accessed 15.1.2013.

———. *Software Design - Inversion of Control.*

<http://martinfowler.com/bliki/InversionOfControl.html>.

Accessed 19.12.2012.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA:

Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-63361-2.

Garlan, David, and Dewayne Perry. 1995. "Introduction to the Special Issue on Software Architecture". *IEEE Transactions on Software Engineering* 21, number 4 ().

Garlan, David, and Mary Shaw. 1994. *An Introduction to Software Architecture*. Technical report. Pittsburgh, PA, USA.

Gilb, Tom. 1985. "Evolutionary Delivery versus the "waterfall model"".

SIGSOFT Softw. Eng. Notes (New York, NY, USA) 10, number 3 (): 49–61. ISSN: 0163-5948.

Gudgin, Hadley, Mendelsohn, Moreau, Nielsen, Karmarkar, and Lafon. 2007. *SOAP Version 1.2*.

<http://www.w3.org/TR/2007/REC-soap12-part1-20070427/>.

Accessed 19.10.2012.

Hammer, D. K., M. Ionita, and H. Obbink. 2002. "Scenario-Based Architecture Evaluation Methods: An Overview".

Hao, He. 2003. *What Is Service-Oriented Architecture*.

<http://www.xml.com/pub/a/ws/2003/09/30/soa.html>.

Accessed 10.1.2013.

Hendrickson, Elisabeth. 2008. *Driving Development with Tests: ATDD and TDD*.

[http://testobsessed.com/wp-content/uploads/2011/](http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf)

[04/atddexample.pdf](http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf).

Accessed 5.2.2013.

Hofmeister, Christine, Robert Nord, and Dilip Soni. 2000.

Applied software architecture. Boston, MA, USA:

Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-32571-3.

IEEE 1471:2000, Recommended Practice for Architectural

Description of Software-Intensive Systems. 2000. IEEE Press.

Recommended practice for architectural description. 2000. IEEE Press.

ISO. 2001. *ISO/IEC 9126-1:2001, Software engineering – Product quality – Part 1: Quality model*. Technical report. International Organization for Standardization.

Jacobson, Ivar. 1992. *Object Oriented Software Engineering: A Use Case Driven Approach*.

1st edition. Addison-Wesley Professional. ISBN: 0201544350.

Jazayeri, M., A. Ran, and F. van der Linden. 2000. "ARES Conceptual Framework for Software Architecture". In, 1–29. Addison-Wesley.

Kazman, R., G. Abowd, L. Bass, and P. Clements. 1996. "Scenario-based analysis of software architecture". *Software, IEEE* 13, number 6 (): 47–55.

Kazman, R., Jai Asundi, and M. Klein. 2001. "Quantifying the costs and benefits of architectural decisions". In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, 297–306.

Kazman, R., Len Bass, Mike Webb, and Gregory Abowd. 1994. "SAAM: a method for analyzing the properties of software architectures". In *Proceedings of the 16th international conference on Software engineering*, 81–90. ICSE '94. Sorrento, Italy: IEEE Computer Society Press. ISBN: 0-8186-5855-X.

Kazman, R., M. Klein, and P. Clements. 2000. *ATAM: Method for Architecture Evaluation*. Technical report. CMU/SEI.

Kruchten, P. 1991. "Un Processus de Développement de Logiciel Itératif et Centré sur l'Architecture [An Iterative Software Development Process Centered on Architecture]".

———. 1995. "The 4+1 View Model of Architecture". *IEEE Softw.* (Los Alamitos, CA, USA) 12, number 6 (): 42–50. ISSN: 0740-7459.

Kruchten, P., H. Obbink, and J. Stafford. 2006. "The Past, Present, and Future for Software Architecture". *Software, IEEE* 23, number 2 ().

Manifesto for Agile Software Development. 2001.

<http://agilemanifesto.org/>.

Accessed 17.1.2013.

Martin, Robert C. *Keynote: Architecture the Lost Years*.

<http://www.confreaks.com/videos/759-rubymidwest2011-keynote-architecture-the-lost-years>.

Accessed 6.11.2012.

———. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0135974445.

———. 2011a. *Clean Architecture*.

<http://blog.8thlight.com/uncle-bob/2011/11/22/Clean-Architecture.html>.

Accessed 6.2.2013.

Martin, Robert C. 2011b. *Screaming Architecture*.

<http://blog.8thlight.com/uncle-bob/2011/09/30/Screaming-Architecture.html>.

Accessed 6.2.2013.

———. 2012. *The Clean Architecture*.

<http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html>.

Accessed 6.2.2013.

Monroe, Robert T., Andrew Kompanek, Ralph Melton, and David Garlan. 1997. “Architectural Styles, Design Patterns, and Objects”. *IEEE Softw.* (Los Alamitos, CA, USA) 14, number 1 (): 43–52. ISSN: 0740-7459.

Nerur, Sridhar, and VenuGopal Balijepally. 2007. “Theoretical reflections on agile development methodologies”. *Commun. ACM* (New York, NY, USA) 50, number 3 (): 79–83. ISSN: 0001-0782.

Nurmuliani, N., D. Zowghi, and S. Powell. 2004. “Analysis of requirements volatility during software development life cycle”. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, 28–37.

O’Brien, Liam, Paulo Merson, and Len Bass. 2007. “Quality Attributes for Service-Oriented Architectures”. In *Proceedings of the International Workshop on Systems Development in SOA Environments. SDSOA ’07*. Washington, DC, USA: IEEE Computer Society. ISBN: 0-7695-2960-7.

Palermo, Jeffrey. *The Onion Architecture*.

<http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>.

Accessed 15.2.2013.

Perry, Dewayne E., and Alexander L. Wolf. 1992. “Foundations for the study of software architecture”. *SIGSOFT Softw. Eng. Notes* (New York, NY, USA) 17, number 4 (): 40–52. ISSN: 0163-5948.

Principles Behind the Agile Manifesto. 2001.

<http://www.agilemanifesto.org/principles.html>.

Accessed 9.2.2013.

Rantanen, Juha. 2012. *Acceptance Test Driven Development ATDD*.

http://reaktor.fi/en/our_expertise/acceptance_test_driven_development_atdd/.

Accessed 5.2.2013.

Royce, W.E., and W. Royce. 1991. "Software architecture: Integrating Process and Technology". 14:2–15.

Royce, Winston W. 1970. "Managing the development of large software systems: concepts and techniques". Reprinted in Proc. Int'l Conf. Software Engineering (ICSE) 1989, ACM Press, pp. 328-338 ().

ServiceStack. 2012. *Service Stack*.

<http://www.servicestack.net/>.

Accessed 8.2.2013.

SOAP Tutorial - SOAP syntax.

http://www.w3schools.com/soap/soap_syntax.asp.

Accessed 14.1.2013.

Sommerville, Ian. 2010. *Software Engineering*. 9. Addison-Wesley. ISBN: 978-0-13-703515-1.

Soni, Dilip, Robert L. Nord, and Christine Hofmeister. 1995. "

Software architecture in industrial applications". In

Proceedings of the 17th international conference on Software engineering, 196–207. ICSE '95. Seattle, Washington, United States: ACM. ISBN: 0-89791-708-1.

Sprott, David, and Lawrence Wilkes. 2004. *Understanding Service-Oriented Architecture*.

<http://msdn.microsoft.com/en-us/library/aa480021.aspx>.

Accessed 7.1.2013.

The Microsoft Windows Communication Foundation team. *Principles of Service Oriented Design*.

<http://msdn.microsoft.com/en-us/library/bb972954.aspx>.

Accessed 10.1.2013.

Toth, Paolo, and Daniele Vigo. 2002. "Models, relaxations and exact approaches for the capacitated vehicle routing problem". *Discrete Appl. Math.* (Amsterdam, The Netherlands, The Netherlands) 123, **numbers** 1-3 (): 487–512. ISSN: 0166-218X.

Unity Container. 2012.

<http://msdn.microsoft.com/en-us/library/dd203101.aspx>.

Accessed 19.12.2012.

Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language.

<http://www.w3.org/TR/2007/REC-wsd120-20070626/>.

Accessed 14.1.2013.

Web Services Glossary.

<http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/>.

Accessed 10.1.2013.

Wikipedia - Software testability.

http://en.wikipedia.org/wiki/Software_testability.

Accessed 11.1.2013.

Witt, B., F.T. Baker, and E. Merritt. 1994. *Software Architecture and Design: Principles, Models, and Methods*. Van Nostrand Reinhold.

Xamarin Inc. *MoMA - Issue Descriptions*.

http://www.mono-project.com/MoMA_-_Issue_Descriptions.

Accessed 12.2.2013.

———. *Mono - Cross platform, open source .NET development framework*.

http://mono-project.com/Main_Page.

Accessed 12.2.2013.

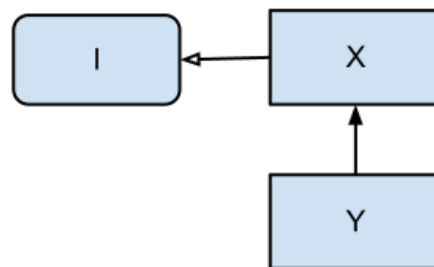
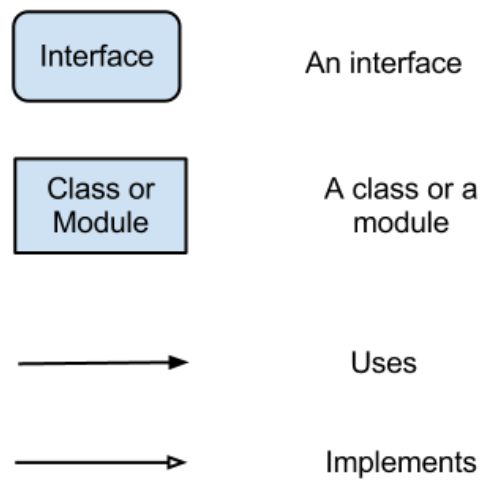
Xamarin Inc. *Mono - MoMA*.

<http://mono-project.com/MoMA>.

Accessed 12.2.2013.

Appendices

A Simple Architectural Dependency Notation



Example: X implements I and Y uses X.

B Design Pattern Implementation Examples

B.1 Service Locator Pattern

```
public interface IServiceLocator { T GetService<T>(); }
public interface IServiceX { /* implementation */ }
public interface IServiceY { /* implementation */ }
public interface IServiceZ { /* implementation */ }

class ServiceX : IServiceX { /* implementation */ }
class ServiceY : IServiceY { /* implementation */ }
class ServiceZ : IServiceZ { /* implementation */ }

class ServiceLocator : IServiceLocator
{
    private readonly IDictionary<object, object> services;

    public ServiceLocator()
    {
        services = new Dictionary<object, object>();

        this.services.Add( typeof( IServiceX ),
                        new ServiceX() );
        this.services.Add( typeof( IServiceY ),
                        new ServiceY() );
        this.services.Add( typeof( IServiceZ ),
                        new ServiceZ() );
    }

    public T GetService<T>()
    {
        try
```

```

    {
        return (T)services[typeof( T)];
    }
    catch (KeyNotFoundException)
    {
        throw new ApplicationException(
            "The requested service is not registered"
        );
    }
}
}

```

B.2 Dependency Injection Pattern

An example implementation of the dependency injection pattern using the Unity.

The container section in the XML file registers `RelationalDb` as an implementation for `IDatabase`. The `ExampleApplication` class is a consumer that has dependency on `IDatabase`. The `UnityContainer` injects an concrete implementation into the `ExampleApplication`.

```

public interface IDatabase
{
    ICollection<object> GetListOfEntities(string entity);
    object GetEntityById(int id);
    bool AddEntity(object entity);
}

public class RelationalDb : IDatabase
{
    public ICollection<object> GetListOfEntities(string entity)
    { /* implementation */ }
}

```

```

public object GetEntityById(int id)
{ /* implementation */ }

public bool AddEntity(object entity)
{ /* implementation */ }

// ...
}

public class ExampleApplication
{
    private readonly IDatabase db;

    public ExampleApplication()
    {
        IUnityContainer container =
            new UnityContainer();
        // reads the xml file and
        // loads IDatabase implementation
        container.LoadConfiguration();
        // injects the implementation
        this.db = container.Resolve<IDatabase>();
    }
    // ...
}

// an example xml file for Unity
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="unity"
            type="Microsoft.Practices.Unity.Configuration.

```

```
        UnityConfigurationSection,
        Microsoft.Practices.Unity.Configuration"/>
</configSections>
<unity xmlns=
"http://schemas.microsoft.com/practices/2010/unity">
  <container>
    <register type="IDatabase, ExampleApplication"
      mapTo="RelationalDb, ExampleApplication" />
  </container>
</unity>
</configuration>
```

C MoMA Scan Results

Mono migration analyzer (MoMA) is a program for scanning .NET applications. It checks if the application uses anything not supported by Mono. While MoMA can help show potential issues, there are many complex factors that cannot be covered by a simple tool. MoMA may fail to point out areas that will cause problems, and may point out areas which will not actually be an issue (Xamarin Inc.).

D Issue Descriptions

There are 4 types of issues that MoMA will detect and report (Xamarin Inc.). The issues are described in the following listing (Xamarin Inc.).

- **Missing Methods:** This is the most severe type of issue. These methods are methods that are not implemented in Mono in any way, not even as stubs. If you try to compile your application that uses these methods with Mono, you will get an error like:

```
myfile.cs(22,16): error CS0117: 'xxxxxxxxxxxxxxxxxxx' does not contain a definition for 'xxxxxxxxxxxxxxxxxxx'
```

If you compile your application with MS's compiler, your application will run on Mono until it tries to use the missing method. It will then exit the whole application with an error like:

```
System.MissingMethodException: Method not found: xxxxxxxxxxxxxxxxxxxxxx
```

- **MonoTodo:** Methods marked with “MonoTodo” may or may not cause problems for your application. Sometimes a method may be marked with this to remind a developer that some small part is not implemented or to clean it up later. Other times, the method may not be implemented at all and simply will not perform any function. This is generally done to make an application compile and run, even if it missing some functionality.

The detail report may list a specific reason why the method is marked with “MonoTodo”. Going forward, it has been requested that any developer who uses

“MonoTodo” provide a reason that can be used for this report. However, numerous pre-existing tags do not have this reason.

- **NotImplementedException:** These can be tricky to determine if they are a problem or not. In many cases the methods are not implemented at all, and simply throw a NotImplementedException as soon as they are called. In other cases, the method may only throw the exception under certain circumstances, while most calls work as expected. We would be very interested in any feedback about these issues. If certain methods provide a lot of false positives, we need to come up with a solution to discount them.
- **P/Invokes:** P/Invokes (platform invokes) are used to call functions that are written in unmanaged languages, often times provided by the platform itself (user32.dll, shell32.dll). However, these can also be calls into your own unmanaged libraries. Mono can handle these calls when the unmanaged library is available for the platform you are using, however many times the whole purpose of using Mono is to run on many platforms.

D.1 Results

The results of the MoMA scan are represented in Figure 20. Scan was ran against the Mono 2.8 (4.0 profile).












Assembly	Version	Missing	Not Implemented	Todo	P/Invoke
 Microsoft.Practices.ServiceLocation.dll	1.0.0.0	0	0	0	0
 Microsoft.Practices.Unity.Configuration.dll	2.1.505.0	0	0	0	0
  Microsoft.Practices.Unity.dll	2.1.505.0	0	0	21	0
  NFleet.dll	1.0.0.0	0	0	1	0
 nunit.framework.dll	2.5.9.10348	0	0	0	0
 NFleet.Test.dll	1.0.0.0	0	0	0	0
 Microsoft.Practices.Unity.Interception.Configuration.dll	2.1.505.0	0	0	0	0
  Microsoft.Practices.Unity.Interception.dll	2.1.505.0	0	0	15	0
Totals		0	0	37	0

Figure 20. MoMA run against the Mono 2.8 (4.0 profile)

Calling Method	Method with [MonoTodo]	Reason
bool Equals (Object)	bool Type.op_Inequality (Type, Type)	Implement it properly once 4.0 impl details are known.

D.2 Microsoft.Practices.Unity.dll

Calling Method	Method with [MonoTodo]	Reason
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
DynamicMethod CreateBuilderMethod (Type, string)	void DynamicMethod..ctor (string, Type, Type[], bool)	Visibility is not restricted
void EmitLoadBuildKey ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void EmitLoadTypeOnStack (Type)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void EmitResolveDependency (Type, string)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void EmitResolveDependency (Type, string)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void EmitClearCurrentOperation ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void CreatePreamble ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void CreatePostamble ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void PreBuildUp (IBuilderContext)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported

D.3 Microsoft.Practices.Unity.Interception.dll

Calling Method	Method with [MonoTodo]	Reason
MethodBuilder CreateDelegateImplementation ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation ()	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateMethodOverride (MethodBuilder)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateMethodOverride (MethodBuilder)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
void ImplementAddInterceptionBehavior (TypeBuilder, FieldInfo)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation (MethodInfo)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation (MethodInfo)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation (MethodInfo)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation (MethodInfo)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateDelegateImplementation (MethodInfo)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateMethodOverride (MethodBuilder)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported
MethodBuilder CreateMethodOverride (MethodBuilder)	void ILGenerator.EmitCall (OpCode, MethodInfo, Type[])	vararg methods are not supported