Applying Frameworks to Increase Productivity for Small Application Development

Master's Thesis

Bastiaan Pierhagen

Applying Frameworks to Increase Productivity for Small Application Development

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bastiaan Pierhagen
born in Gorinchem, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Info Support
Kruisboog 42
Veenendaal, the Netherlands
`www.infosupport.com`

Applying Frameworks to Increase Productivity for Small Application Development

Author:	Bastiaan Pierhagen
Student id:	1100254
Email:	bastiaanp@infosupport.com

Abstract

The Professional Development Center of Info Support noticed that their software factory Endeavour is too extensive for developing small applications. Application developers have to write too much code manually. This document describes how we can apply frameworks to increase productivity for small application development, without losing the level of quality of the original approach. First we present an overview of the underlying technologies of frameworks. We then identify and discuss a number of candidate frameworks to target productivity bottlenecks in the original approach. With a resulting set of frameworks we build a prototype to perform a case study. With a case study and the opinion of domain experts we validate our solution. We conclude with a discussion of our results and propose some future research projects.

Thesis Committee:

Chair:	Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Andy Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Ing. Dennis Joosten, Info Support
Committee Member:	Dr. Ing. Leon Moonen, Faculty EEMCS, TU Delft
Committee Member:	Drs. Peter van Nieuwenhuizen, Faculty Computer Graphics, TU Delft

Acknowledgments

This graduation project cost me a lot of time and effort. Therefore I would like to thank a number of people who supported me during the project, and who contributed to the final result.

First of all I would like to thank my supervisor of the Delft University of Technology, Andy Zaidman, for his advice during this research project. Not only his expertise in the field of software engineering, but also his knowledge on how to write a good thesis has been very helpful. Dennis Joosten and Marco Pil of Info Support—as project supervisor and technical supervisor, respectively—have done a great job in providing a wealth of information about Endeavour and sharing ideas about how to tackle certain problems. I would like to thank Info Support for providing me with the opportunity to do this project, and all colleagues at Info Support who provided a joyful working environment. Finally I would like to thank a friend, fellow-student, and now colleague of mine, Maarten Schilt for our interesting discussions about several topics and the numerous reviews he has done on this report, which really increased the quality of this document.

On a personal note I would like to thank my friends for their support and understanding. A final thanks to my parents, Gerard and Janny, for their support and advice. They provided me with the opportunity to study, which forms the basis for my future career.

Bastiaan Pierhagen
Delft, the Netherlands
January 24, 2008

Contents

List of Figures

Chapter 1

Introduction

To develop high-quality software solutions, Info Support developed Endeavour. Endeavour is a software factory; an architectural framework which guides software developers in creating maintainable and expandable software systems, by suggesting a combination of architectural aspects and providing a number of off-the-shelf components.

Endeavour prescribes to develop software systems from a service-oriented perspective, in other words, using a service-oriented architecture, to set up software infrastructures in a modular way. Endeavour has been applied in several projects[1] and seems to work very well for somewhat larger software systems. However, when Endeavour is applied in relatively small and simple projects, developers have to perform a lot of work writing code manually, with respect to the overall size of the project; a labor intensive task.

This thesis describes a study in which we apply frameworks to increase productivity for small application development. We choose a set of frameworks, build a proof-of-concept, and finally validate our solution.

This chapter is organized as follows. In Section 1.1 we define the goals of this project. Then we present the central research question in Section 1.2 We continue by discussing the different phases of the project in Section 1.3. The succeeding section, Section 1.4 discusses the parties involved in this project. Finally, in Section 1.5, a word about the audience this document is intended for.

1.1 Project Goals

Endeavour has been applied in several projects and has proven itself for enterprise-sized SOA applications. However, when Endeavour is applied in relatively small and simple projects, developers perform a lot of work writing code, with respect to the overall size of the project. Therefore we investigate the effects of applying frameworks to increase productivity for small and simple application development, without losing the level of quality of the original approach. Comparing a framework solution with other productivity increasing technologies falls outside the scope of this thesis.

[1]For examples see `www.infosupport.com/references`.

1.2 Project Research Question

In the previous section we indicated our main goal is to research the applicability of frameworks to increase productivity in our problem domain. This domain is restricted to small and simple Endeavour applications. The application of frameworks should not result in a loss of quality when compared to the original approach. With these notions in mind, we formulated the research question below.

> *How can we apply frameworks to increase productivity for small and simple Endeavour applications without losing the level of quality of the original approach?*

This is a difficult question to answer as a whole, therefore we divided it into several sub questions. In each chapter of our thesis, we answer one sub question. The sub questions are indicated per chapter below.

Chapter 2 *What does our problem domain look like?* What are small and simple applications? What is a Service-Oriented Architecture? What is Endeavour?

Chapter 3 *What are frameworks?* What are advantages and costs of using a framework? How should a framework be designed? What are the goals of designing a framework? What are the underlying concepts of frameworks that offer them their power?

Chapter 4 *How can we map frameworks on our problem domain?* What are possible applications of frameworks in our problem domain?

Chapter 5 *Which frameworks are we going to use?* What are the actual bottlenecks in the current approach? Which bottleneck should we target? Which frameworks can target these bottlenecks?

Chapter 6 *What are the criteria our solution has to satisfy to result in a successful project?* Why are these criteria required?

Chapter 7 *What does our solution look like?* How is the solution implemented? What are the possibilities? What are the limitations?

Chapter 8 *How can the solution be applied in a practical situation?* How do we implement an application on top of our solution? What are our first impressions of the quality of our solution?

Chapter 9 *What is the quality of our solution?* Did we increase productivity with our solution? What about the other quality attributes? Which validation method should we choose?

1.3 Project Phases

In this section we discuss the different phases of the project. In every project phase we have performed a specific set of tasks. We discuss these tasks and indicate which chapters are related to which phase.

1.3.1 Assignment Definition

When we started with this project, we had some rough ideas about the assignment. In order to satisfy all involved parties, it was necessary to write a master thesis proposal. By writing such a proposal we could get a better insight in the goals of the project. As these goals might be different for all involved parties, the final version of the master thesis proposal could serve as a contract between all parties. The master thesis proposal contains an assignment definition, the project goals, some initial descriptions about domain related topics, a risk analysis, a schedule, and a description of the other project phases. These phases are described in the next sections. (Chapter 1)

1.3.2 Research Phase

During the research phase we gather knowledge about our domain. We read material about frameworks, Endeavour, SOA, etc. The relevant information is discussed in this thesis. We also start thinking about how we can apply frameworks in our domain. (Chapter 2, 3, and 4)

1.3.3 Analysis Phase

During the analysis we determine what parts of Endeavour can be optimized. We interview developers of Endeavour to identify parts that require attention. We also look at a number of applications built with Endeavour to identify specific framework application possibilities. Finally we define some solution criteria, which are used to determine the quality of our solution. (Chapter 5 and 6)

1.3.4 Implementation Phase

The results of the analysis phase will be used to design and implement a proof-of-concept. By combining a number of specific frameworks we create a solution for our general problem assignment. This proof-of-concept can be seen as a means to a goal. During the analysis phase we have identified a solution to improve the efficiency of Endeavour, which will be implemented to validate our solution in the next phase. (Chapter 7 and 8)

1.3.5 Validation Phase

During the validation phase we determine the effects of our solution on productivity and the other quality attributes. (Chapter 8 and 9)

1.4 Parties Involved

In this section we discuss the parties involved in this project. During the remainder of this thesis we refer to these parties a number of times, therefore it is important to have some context information about all parties.

1.4.1 Info Support

Info Support is an ICT consultancy agency which employees about 250 persons in the Netherlands, and about 35 in Belgium. They work for medium to large enterprises and mainly provide consultancy, software development, and training. The organization wants to present itself as a solid innovator, supporting organizations in developing innovative applications which meet their business requirements. To achieve this goal, Info Support constantly improves its software development process by adopting the latest technologies.

1.4.2 Professional Development Center (PDC)

An important part of Info Support is the professional development center (PDC). The PDC is responsible for developing and maintaining Info Support's software factory Endeavour. Endeavour provides application developers with building structures, a software development environment, design guidelines, and quality and progress monitoring possibilities. Endeavour is constantly improved by the PDC by adopting state-of-the-art technologies.

1.4.3 Software Evolution Research Lab (SWERL)

The Software Evolution Research Lab is part of the Software Engineering Research Group of the Delft University of Technology. SWERL aims to address deterioration of the structure of evolving software systems, by developing new technologies, researching new analysis techniques, and investigating restructuring techniques.

1.4.4 Fellow Student: Maarten Schilt

While working on this project, a colleague and friend of mine, Maarten Schilt, is also working at Info Support on a similar assignment. The difference between our assignments is that we will focus on frameworks to increase the efficiency of Endeavour toward the development of small applications, where Maarten focuses on model-driven development. This way we are able to discuss several topics regarding our projects. An extensive comparison between our work falls outside the scope of this project, but in the final section we briefly discuss the strengths and weaknesses of both approaches.

1.5 Audience

This document is intended for people with a basic level of experience in the field of software engineering. We try to explain the topics in such a way the reader can easily understand the material, by explaining all specific software engineering related terms and providing several code examples to illustrate certain concepts. To verify if we have done a good job, several people have reviewed this report and gave pointers to improve the quality of the document.

Chapter 2

Exploring Problem Domain

This chapter describes the problem domain in which we perform our research. By describing and analyzing the problem domain, we can narrow the scope of our research area, which enables more in-depth research possibilities. Let us recall the general research question as presented in Chapter 1:

> *How can we apply frameworks to increase productivity for small and simple Endeavour applications without losing the level of quality of the original approach?*

This question raises a number of other questions. First of all, what are frameworks? Chapter 3 provides an answer to that question. In Chapter 6 we define how we determine the quality of our solution. We also have to define what small and simple application are and how we measure an increase in productivity (Section 2.1). As we discussed earlier, Endeavour prescribes to develop applications with a service-oriented architecture (SOA). This requires an explanation of SOA (Section 2.2). Then we discuss Info Supports' software factory Endeavour (Section 2.3). Finally we discuss small and simple Endeavour applications (Section 2.4).

2.1 Small and Simple Applications

As becomes clear from our general research question, the overall goal of the project is to find a way to increase the productivity for developing small and simple applications. The productivity to improve lies in the amount of work spent on manually writing code. To measure an increase in productivity a metric representing the size of a software system is required.

An obvious choice to measure application size would be the number of source lines of code (SLOC), because this is simple to measure. However, this metric has some disadvantages [Ko07]. The number of SLOC does not say anything about the complexity or functionality of the software system, but is only limited to a special view of size, namely length. Another disadvantage is that SLOC is language dependent. Furthermore a badly designed system probably results in an excessive number of lines of code. Another drawback of SLOC is that it only takes source code into account, and not the size of the specification.

Another metric for measuring application size are function points. A *function point* (FP) is an entity to measure the size of an information system or project [BOS+04]. The number of function points is determined during a *function point analysis* (FPA). The number of function points indicates the amount of functionality offered to the user of an information system [BOS+04]. During a function point analysis the specifications that determine the information exchange of the information system with its environment are analyzed. These specifications are categorized into a number of user functions, with three levels of complexity; low, average, and high. For each user function category complexity measurements have been defined. In other words, a user function belongs to a category with a certain complexity factor. The complexity of a user function is defined as the number of function points assigned to a user function based on the amount of information processing of the user function [BOS+04]. By computing the sum of function points of all user functions and multiplying this with a technical complexity factor, we have an indication of the size of the software application. The technical complexity factor represents a complexity factor for this kind of project [Ko07].

In contrast with the SLOC metric, FP depends on complexity and functionality of the software application, but does not depend on the implementation language or the system design. A disadvantage of the FP metric is that the complexity of the user functions are estimated [Ko07]. Therefore this is a subjective process, which is difficult to compute and hard to automate. Table 2.1 gives an overview of the characteristics of both SLOC and FP software size metrics.

SLOC	FP
(+) Objective method	(–) Subjective method
(+) Process is easy to automate	(–) Process is hard to automate and compute
(–) Size depends on length	(+) Size depends on complexity and functionality
(–) Language dependent	(+) Language independent
(–) Design dependent	(+) Design independent

Table 2.1: SLOC and FP Compared

At Info Support both metrics are used. The number of function points gives an indication about the complexity and functional characteristics of the software application. For this research project we have defined that we want to analyze applications consisting out of up to 300 function points. This does not automatically imply our solution will not work with larger applications. The reason for defining this constraint is that applications consisting out of up to 300 function points have simple business logic, according to the project data collected at Info Support.

As mentioned earlier, the goal of this project is to reduce the amount of work spent on manually writing code. Reducing the number of function points is not possible, because the FP metric is design and language independent. Reducing the number of function points would actually result in a less powerful system. A reduction in the number of manual SLOC does not automatically imply an increase in productivity. The time needed to write these SLOC has to be taken into account too. Writing half the number of SLOC, in twice the amount of time as with the original approach is not a clear indication of an increase in productivity.

2.2 Service-Oriented Architecture

In this section we will introduce the concepts of service-oriented architecture. We start by discussing a service before we start with an explanation of the service-oriented architecture.

2.2.1 Service

A *service* is a logical software entity which consists of a service interface and a service implementation. The service interface consists of one or more published interfaces and can be viewed as a contract defining the input, output, and exception conditions of a service [EAA$^+$04, NS03]. The service implementation is hidden from the outside world, therefore a service has a black box nature [NS03, Plu05]. A service is invoked by another service, or by an end-user, which will both be called *clients* from here. In the next paragraphs we will further elaborate on the following aspects of services; interfaces and contracts, communication between services, service lookup and registration, and finally service state. Figure 2.1 gives an overview of the components of a service. We further explain the aspects of a service with this figure.

Interfaces and Contracts

As already mentioned, the service interface consists of a number of published interfaces, which can be viewed as a contract defining the input, output and exception conditions of a service. At Info Support these contracts are identified as the service contract and the data contract. The service contract defines the exposed functionality of a service. The data contract defines the incoming and outgoing messages of a service. By providing an interface, clients are decoupled from the actual service implementation, which leads to loose coupling. A service client accesses the interface of the service which delegates requests to the actual implementation. This approach provides the opportunity that the service implementation can be changed, without changing the client's implementation [Sta06].

Another reason for the use of interfaces is to enable the possibility to use different implementation techniques, striving for heterogeneity. Of course, when interfaces are used for this purpose, they have to be described in a technology independent way. This can be done by using an interface description language, capturing a description of the service's functionality, including pre- and post conditions and invariants, the physical location of a service, a semantic description of a service and quality-of-service information [Sta06].

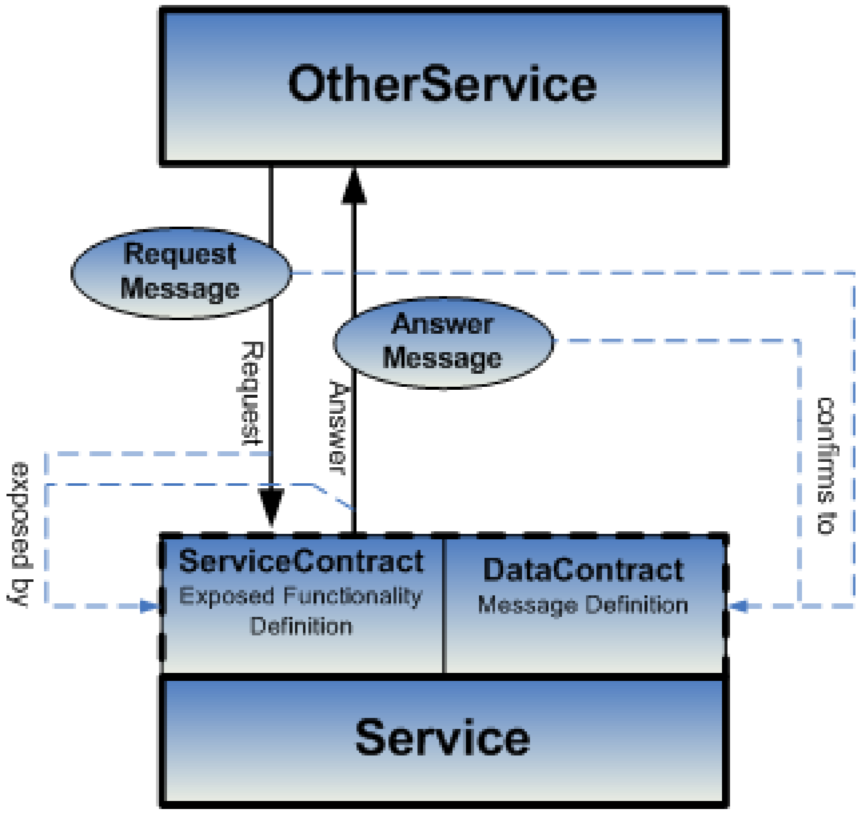


Figure 2.1: DataContract

Communication between Services

In this paragraph we will further elaborate on how services communicate. A service has to support various communication protocols, in order to provide loose coupling and flexibility in communication. When a client wants to communicate with a service, the client must exchange messages with that service. This communication should not be performed over a fixed communication line, but by applying a dynamic message routing system, so no tight coupling occurs [Sta06].

Now that we have discussed the way clients and services should communicate, we continue with the way a message should look like. In order for a client and a service to understand each other, they have to use some message format. A client and service can negotiate about a message format individually, or on the other hand, the same message format can be used for communication between all services. The latter approach is less flexible, but messages can be processed automatically [Sta06]. As we discussed earlier, the incoming and outgoing messages of a service are defined in the data contract. At Info Support the message format is mainly XML.

Service Lookup and Registration

Before a client can access a service, it must be located first. Hard-coding the location information in the client code leads to tight coupling. Therefore another approach is preferred. Services register themselves with a dispatcher. Clients query the dispatcher for available services. The *dispatcher* maintains service interface descriptions to serve as a proxy server. When the service is actually needed, it is invoked. This way a flexible service discovery is supported. Another important issue is that concrete service locations should not be used. This would only increase dependencies between clients and services. Clients should for example use a universal resource locater (URL), to reference a service. The choice of the location protocol to apply depends on the available communication protocol [Sta06].

Service State

When services and communication protocols do not maintain state information, this increases loose coupling and scalability. However, in certain circumstances state information has to be maintained. This could be the case with service affinity; services are associated with concrete entities, for example a printing service which is associated with a printer in your corridor. Another example where state information is maintained is with sessions. Sessions are used to provide information in several parts to a client for efficiency reasons, so the service must maintain which parts have already been sent [Sta06].

2.2.2 Service-Oriented Architecture

Service-oriented architecture (SOA) presents an approach for building distributed systems that deliver application functionality as services to either end-user applications or other services [EAA$^+$04]. Service providers register their services with a central repository (dispatcher), and service consumers query the repository for the services they need. Once clients have identified the right services within the repository, they can directly interact with those services [Sta06].

Figure 2.2 displays an architectural overview of a SOA system [Inf06a, Inf06b]. With this picture the configuration items of a service-oriented architecture are further explained. The components displayed in Figure 2.2 are called configuration items [Inf06b]. A *configuration item* is a logical unit of functionality and/or data [Inf06b]. The configuration items used in Endeavour are listed below.

Front-end A front-end offers functionality to an end-user through a channel, which could be the Internet, intranet, or a PDA. A front end is developed with the limitations of the channel in mind and specified for the specific tasks the end-users want to perform [Inf06a].

Service bus The service bus provides the communication channel between services. The service bus can be viewed as the dispatcher, as discussed earlier in Section 2.2.1. The dispatcher is responsible for registering services and for serving query requests from clients for specific services. All services communicate using the same message format [Inf06a].

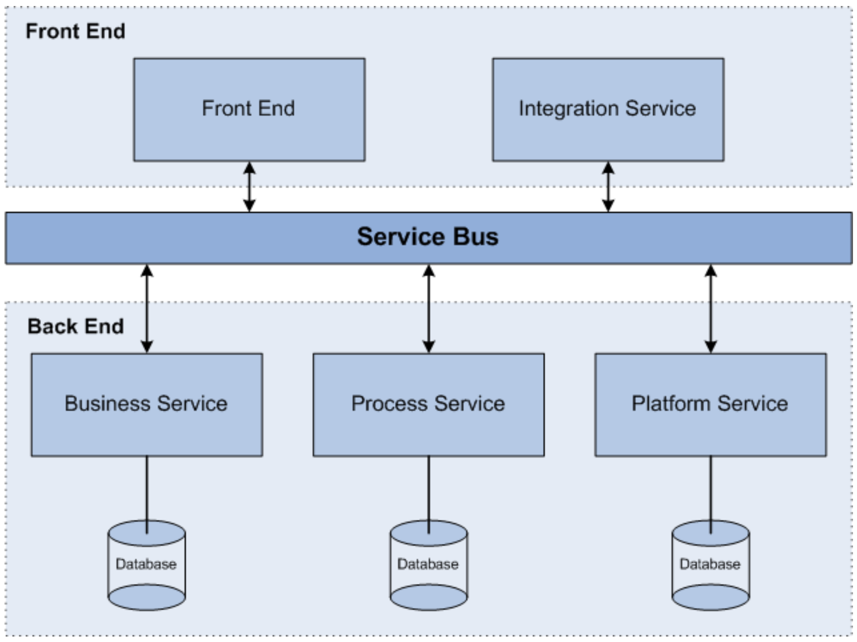


Figure 2.2: Service-Oriented Architecture

Integration Service As mentioned earlier in Section 2.2.1, the same message format can be used to communicate between services. When we want to integrate another application into the SOA, the integration service is responsible for conversion of messages to the service bus standard [Inf06a].

Business service A business service is associated with a specific business function and should be recognizable as such. The primary task of a business service is to administer data directly related to the business process and to ensure consistency and integrity [Inf06a].

Process service A process service is responsible for the automation of some business tasks and therefore uses a number of business services. By making a distinction between business and process services, responsibilities for operations on and storage of data are separated [Inf06a].

Platform service A platform service performs platform specific services, which have no direct links with business, like user account management [Inf06a].

2.3 Endeavour

Endeavour is a software factory designed and developed by Info Support. Endeavour is used to create enterprise business applications with a service-oriented architecture. A *software factory* is an architectural framework which guides software developers in creating maintainable and expandable software systems by suggesting a combination of architectural aspects. This guidance is done by providing building structures, a software development environment, design guidelines, and quality and progress monitoring possibilities. These architectural aspects are also called an architectural style, which describe a set of design rules identifying the kinds of components and connectors that may be used to compose a system, together with constraints on the way the composition is done [MR97]. Software factories try to automate part of the software development approach, to reduce cost and time to market and to improve software quality [GS03, FNO92]. According to those definitions, software factories guide a software developer and tend to automate, at least part of, his work, thereby reducing development time and improving software quality.

2.3.1 Current Application of Frameworks

Endeavour already applies a number of frameworks. In this section we identify and briefly describe these frameworks. We discuss the Info Support WebFramework, WinFramework, Logging, Exception Management, DataAccess, and Interception framework.

Info Support WebFramework

The Info Support WebFramework implements the technical reference architecture as described in [Inf06b]. With the WebFramework a graphical user interface to show in web browsers can be build. The model-view-controller pattern is applied to separate data (model), presenting the data (view), and the navigation (controller), thereby increasing the reusability of the developed pages. The framework is responsible for data transfer between different pages and for navigation between different pages [Inf07].

Info Support WinFramework

The Info Support WinFramework and WebFramework are quite similar. The main difference is that the WebFramework is used to build graphical user interfaces to show in web browsers, where the WinFramework is used to build graphical user interfaces for a Microsoft Windows environment [Inf07].

Info Support Logging

Info Support Logging is responsible for capturing information about the state of an application. This information can consist of technical failures, or exceptions, as well as run-time behavior of the application [Inf07].

Info Support Exception Management

The Exception Management framework is responsible for handling exceptions according to the specifications of the technical reference architecture, described in [Inf06b]. This framework can make use the Logging framework to log the exceptions [Inf07].

Info Support DataAccess

Info Support DataAccess provides communication possibilities with a database. DataAccess is build as an additional layer of abstraction on top of ADO.NET [Inf07].

Info Support Interception

The Info Support Interception framework uses .NET attributes in order to distinguish infrastructural code, like logging, authorization, transaction management, etc, and actual business functionality. By applying this approach, infrastructural code can be separated from the business functionality [Inf07].

2.4 Small and Simple Applications with Endeavour

So far we have discussed what SOA applications look like and discussed the constraints with respect to application size and complexity. In this section we will further elaborate on the components of an application which meet our size and complexity constraint and we will define the number of expected SOA components small applications developed with Endeavour contain. For each configuration item we have estimated the number of occurrences.

Configuration Item	No of Occurrences
Front end	1
Integration Service	*
Business Service	1
Process Service	0
Platform Service	*
Service Bus	0

Table 2.2: Number of Configuration Items

Table 2.2 shows the number of expected of configuration items for small applications developed with Endeavour. These numbers are based on experiences of Info Support with previous projects. We will use these numbers at a later stage in our research project, by identifying which configuration items a candidate framework targets and estimating the impact of the productivity increase for small applications. Frameworks targeting a service bus or a process service have no impact according to our size criteria, because they do not occur in small applications.

Chapter 3

Introduction to Frameworks

According to the American Heritage Dictionary a framework is “a structure for supporting or enclosing something else, especially a skeletal support used as the basis for something being constructed” [AHD00]. A software *framework* is a set of classes that make up reusable design for an application [Joh05, Che04]. In other words, a framework can be seen as a building structure for applications.

Normally application code invokes a class library to perform an operation. A *class library* provides a collection of classes, which can be used in a number of applications [CA05, JF88]. With the application of a framework it is the other way around, a framework invokes application code, thus managing the flow of control. This approach is also known as the Hollywood principle: “Don’t call us, we’ll call you.” The official term for this design principle is *Inversion of Control*, which is seen as one of the common characteristics of a framework [Fow04a].

This chapter is structured as follows. First we begin with a discussion of the advantages and costs of using frameworks in Section 3.1 and 3.2 respectively. Then we discuss the framework design principles in Section 3.3. In Section 3.4 we continue by discussing the framework design goals. Finally, in Section 3.5, we give an overview and explanation of the underlying concepts of frameworks which offer them their power.

3.1 Advantage of Using Frameworks

In this section we will introduce a number of advantages of using a framework, which could be seen as the most important reasons for developers to make use of frameworks. We will also consider some reasons explaining the costs of using or building frameworks.

[Joh05] identifies a number of advantages of using a framework instead of the traditional approach of invoking class libraries from your application code:

- By using a framework, application developers only have to write code they need to write.
- Frameworks provide structure and consistency to an application.
- Frameworks can promote best practice through examples and documentation.

But how are frameworks able to provide these advantages to their clients? This will be further explained and illustrated in Section 3.3 and Section 3.4.

3.2 Costs of Using Frameworks

Now we have covered a number of advantages of the use of frameworks, one might think that frameworks are always the way to go. However we should keep the objectives in mind, we want to reach by using or building a framework. We will discuss the development costs of a framework and user training to use a framework, which could be reasons not to use or build a framework.

Developing a highly usable and extensible framework is a complex and expensive task [Che04, CA05]. The development process requires experts in the field of software engineering, with respect to implementation and design, whereas the engineers must also be experts in the business domain of the framework. The business domain expertise is required because certain business-specific framework layers have to be developed. Without expertise in implementation and design, an engineer would be unable to transfer the framework concepts from theory to practical applications, in creating a reusable and extensible framework. People who are experts in both disciplines, software engineering and the business domain, are hard to find [Che04].

Frameworks are used by other developers. In order to take advantage of all features a framework offers, developers who will use the framework have to be trained to perform this task. This training process can be very time consuming, which can be explained by several factors [Che04]:

- As mentioned earlier, a framework is a building structure for applications. From the viewpoint of a developer who is going to use the framework, the framework contains a lot of gaps, which will be filled by developing an application [Che04]. Before the application is completed, the framework may seem obscure, because of these gaps.

- A common property of a framework is that it manages the flow of control, as mentioned in Section 3. To manage the flow of control, a lot of wiring between components is required. This wiring is incorporated into the framework and not directly visible for the developers who make use of the framework. During the training process, this results in developers having a hard time to understand how the framework actually works under the hood.

3.3 Framework Design Principles

In this section we will discuss the overall design principles of frameworks, which are fundamental to many different framework design approaches and concepts. We will discuss the following: scenario-driven design, low barrier to entry, self-documenting API, and layered architecture [CA05], which are discussed in the next paragraphs.

3.3.1 Scenario-Driven Design

Frameworks usually contain a large set of APIs, enabling advanced scenarios that require power and expressiveness [CA05]. When developers make use of a framework it is uncommon that they use the full range of functionality a framework offers; most development revolves around a set of common scenarios, in which only a small subset of the framework's functionality is used. Therefore it is important to focus on those common scenarios during the design of the framework; a *scenario-driven design*. For this reason [CA05] proposes that framework designers first write the code that application developers have to write in main scenarios, and then design the framework model in such a way that it supports these code samples. Those scenarios should not be to granular. Reading a file is a good example of a scenario. Opening, closing or reading a line from a file are examples of too granular scenarios.

To verify if the design of the framework model results in the preferred behavior, a prototype should be developed. On this prototype we can perform a usability study, with a wide range of developers, to verify if the the APIs constructed by the framework developers are also obvious to the developers who make use of the framework. The results of the usability study give insight in how application developers would make use of the framework. When application developers are unable to implement certain scenarios, the framework APIs should be redesigned [CA05]. Because redesigning an API could have substantial impact on the design of a framework, it is important that usability studies are performed as early as possible in the development phase [CA05].

3.3.2 Low Barrier to Entry

Developers expect to learn the basics of frameworks very quickly. They usually just start by experimenting with, generally simple, parts of the framework [CA05]. Only when certain features are interesting, they take the time to understand the whole framework. Another reason to start digging deeper into the framework is when they need to implement more complex scenarios. When the framework has a badly designed API, this will leave a bad impression of the framework. Therefore it is important that a framework has a low barrier to entry, in other words, the initial encounter of a framework API should not be discouraging, to provide non expert users a good experience experimenting with the framework [CA05]. A number of requirements that APIs must meet to realize this principle have been identified below [CA05]:

- Identifying the right types and members to perform common tasks should be easy. Therefore it is important to divide functionality into different namespaces or packages.

- It should be easy to start using a framework. A framework that requires a lot of initialization is not easy to experiment with. Bad defaults for properties are an example which is likely to leave a bad impression of the framework.

- When the framework API is incorrectly used, it should be easy to fix these mistakes. In case of incorrect usage of the framework API, clear exceptions will help the developer to fix the mistakes.

3.3.3 Self-Documenting API

Many frameworks tend to be very extensive, they consist of hundreds, if not thousands, of types, members and parameters [CA05]. Therefore developers who use a framework require a lot of guidance and information in order to correctly use the framework. Reference documentation on its own cannot meet this demand [CA05]. When developers have to refer to the documentation even when they want to implement the most simple scenarios, this is too time-consuming. Furthermore, as mentioned earlier, developers like to code by trial and error and only start reading documentation when they have to implement more complex scenarios. For these reasons it is important that developers are able to make use of the framework without the implication that developers must consult the reference documentation every time they want to perform a simple task. In other words, an API of a framework should be self-documenting. We will discuss a number of principles to build a self-documenting API for a framework.

The first principle we will discuss is reserving *simple and intuitive names* for types that developers are expected to use in the most common scenarios [CA05]. The opposite of this principle is using the best names for less commonly used types. We will illustrate these principles with the example in Listing 3.1.

```
abstract class File { ... }

class NtfsFile : File { ... }
```

Listing 3.1: Bad Example of Naming (C#)

We define an abstract base class `File` with a concrete implementation `NtfsFile` [CA05]. This approach works fine with the expectation that users will understand the inheritance hierarchy of the framework, before they start using the API. As mentioned earlier, users prefer to code with trial and error. Therefore it is likely that their first attempt to perform operations on a file, will result in an attempt to instantiate a `File`, which is not possible. Therefore the example given in Listing 3.2 seems more appropriate.

```
abstract class AbstractFile { ... }

class File : AbstractFile { ... }
```

Listing 3.2: Better Example of Naming (C#)

Another recommendation with regard to naming is to use a *descriptive identifier name* that clearly states what a method does and what its parameters represent [CA05]. Therefore it is no problem to make use of verbose names when choosing identifier names. For example, use `int count` instead of `int i`. This results in more easily readable and understandable APIs.

The next principle we will discuss is about how *exceptions* should be used in a framework. Exception messages should be used to communicate usage mistakes to developers, clearly indicating how these mistakes can be fixed [CA05]. When a developer for example forgets to set a certain property, the framework should throw an exception with a message indicating that the property should be set, eventually mentioning or referring to the possible input values.

Another principle is to make use of *strong typing*. When we make use of a `Name` property, we prefer that it returns the name as a `string` instead of an `Object`. In some cases weak typing is preferred, think of property bags and other loosely typed APIs, but this should be an exception to the rule and not common practice [CA05]. Therefore it is important that when weakly typed operations are developed, the framework designers should provide some strongly typed helper methods, for frequently used properties. This is further illustrated in Listing 3.3.

Consistency with existing APIs that are already familiar to the user is another technique for designing self-documenting frameworks [CA05]. Framework designers have to estimate the target groups they will reach with their frameworks and identify the frameworks that are already used by these target groups. By implementing a new framework which is consistent with these other frameworks, users will learn the capabilities of the new framework more easily. Of course this is no excuse to use principles from frameworks which do not follow the guidelines described in this section, but you should not try to re-invent the wheel when you do not have a good reason for it.

The final principle we are going to discuss is *limiting abstractions*. When framework designers make use of complex abstractions, users of the framework have to be expert in the architectural design of the framework, in order to use it correctly [CA05]. For simple scenarios this is not acceptable.

3.3.4 Layered Architecture

We have already mentioned that it is important to perform a usability study with a wide range of developers. Requirements for a framework differ from developer to developer. Some developers use the framework to implement simple scenarios, preferring APIs optimized for productivity and simplicity. Others use it to implement very complex scenarios, preferring APIs optimized for expressiveness and power. By applying the layered architecture principle, the different requirements of the wide range of developers can be met [CA05]. By including common features in a general namespace or package, and putting more complex features in other namespaces or packages, possibly "sub" namespaces or packages, we are able to limit the exposed behavior of a framework to an appropriate level. A drawback of this approach is that when behavior of the framework is divided over multiple namespaces or packages, it is harder for developers to see all possibilities of a framework. When they need to implement more complex scenarios, they have to find out where to locate the functionality required from the framework to perform the required task.

```
abstract class File
{
    // Property bag, weakly typed.
    private IDictionary<string, object> properties;

    public IDictionary<string, object> Properties
    {
        get
        {
            return properties;
        }
    }

    // Helper method, strongly typed.
    public string Name
    {
        get
        {
            return properties["Name"] as string;
        }
    }

    public File (string name, bool readOnly, long size)
    {
            properties = new Dictionary<string, object>();

            properties.Add("Name", name);
            properties.Add("ReadOnly", readOnly);
            properties.Add("Size", size);
    }
}
```

Listing 3.3: Typing in Frameworks (C#)

3.4 Framework Design Goals

In this section we will discuss several fundamental framework design goals. [Che04] identifies modularity, reusability, extensibility, simplicity, and maintainability, which will be discussed in the paragraphs below.

3.4.1 Modularity

Modularity is defined as the division of an application into structural components, or modules, allowing application developers to use a framework in a piece-by-piece fashion [Che04]. As mentioned earlier, we should consider the limitations of the human mind and therefore solve problems by creating easily understandable solutions. Applying modularity helps us to achieve this goal. By dividing a framework in several modules or components the complexity of each of the components is reduced. When a developer requires certain functionality, it is not necessary to understand the entire framework; understanding a component of the framework is sufficient. For this reason these smaller components are easier to understand and to work with, which results in increased productivity.

A consequence of the division into multiple components is that functionality is divided over these components. When a client uses a component and a related component is changed, this could at most affect the other component's code, but the client code does not have to be altered. This way developers using certain components of a framework are shielded from changes to other components in the same framework [Che04].

3.4.2 Reusability

Frameworks provide reusability to the applications that make use of it, by sharing their classes and code, but also by sharing their design [Che04]. When an application is developed, some functionality is used multiple times within the application. In most software projects, a number of developers is working together to build the whole application. If every developer would develop an own version of the functionality which is used multiple times, this is a waste of resources, but this also leads to maintenance problems in a later stage, because all the versions of the same functionality must be maintained. When we move this functionality to a framework, we can easily reuse this code in every part of an application which wants to make use of it. Developers may not always be experienced developers, so by using a framework, they also automatically make use of many good software design approaches, such as design patterns, which are frequently applied in frameworks [Che04].

3.4.3 Extensibility

A framework provides a lot of functionality "out-of-the-box", but does not always provide all functionality required for applications built on top of the framework. Each application is unique; it has a specific set of requirements, an architecture, and an implementation. For a framework to support various applications, it is important that the framework can be extended by customization. A high degree of extensibility assures the applicability in various applications. Therefore *extensibility* is an important design goal of a framework [Che04].

3.4.4 Simplicity

An important design goal of frameworks is simplicity [Che04]. With simplicity is referred to the way frameworks simplify the design and development process by providing a building structure that relieves application developers from the responsibility to design and develop a process flow between components, because this is already incorporated into the framework [Che04].

With simplicity we also refer to the ease of use of a framework [CA05]. When we apply the design principles discussed in the previous section, we have to find the right balance between power and simplicity of the framework [CA05].

3.4.5 Maintainability

Maintainability is the ability to effectively support changes as a result of changing business requirements [Che04]. Components of frameworks are used numerous times in applications. When a change has to be made to such a component, this only requires changing this one component, instead of changing the functionality in each of the applications where it is used, when the functionality would be replicated in every application. Therefore maintainability is positively affected by code reuse.

We have already discussed a layered architecture in Section 3.3.4. By dividing different levels of business knowledge over different layers, we can minimize the dependencies between these layers [Che04]. By applying this organization, changing business requirements are more likely to affect a single layer, resulting in a change in only that layer. In other words, the code to be altered for a changed business requirement is limited to the code of a layer. For this reason layering also has a positive effect on maintainability.

3.5 Framework Design Concepts

In this chapter we will discuss a number of underlying concepts of frameworks that offer them their power. By first discussing the underlying concepts of frameworks, it is easier to recognize these concepts in practical applications of frameworks. These design concepts are general techniques that form the foundations of many frameworks. We will discuss dependency injection, the service locator, aspect-oriented programming, and configuration files.

3.5.1 Dependency Injection

Dependency injection is a form of Inversion of Control [Fow04a]. Dependency injection is a programming technique to reduce dependency among components and thereby improving their reusability [CI05]. By applying dependency injection, we decrease the level of coupling between two components, by extracting the dependency declaration definition out of any of the two components.

We will discuss three forms of dependency injection, namely constructor injection, setter injection and interface injection. These different forms of dependency injection are frequently explained with long and complex examples of how they are applied in frameworks [Fow04a, PES$^+$06]. This complexity comes from several side issues which are taken into account in the example. Think for example of configuring dependency injection through an XML file. This approach makes it harder to understand the essential ideas behind the different forms of dependency injection. Therefore we have chosen to give very small examples, which only cover relevant information for the dependency injection form we are explaining.

Constructor Injection

Listing 3.4 shows `SomeClass` which implements the `IInjectable` interface defining method `SomeOperation`. Constructor injection is illustrated in Listing 3.5. `AnotherClass` depends on an `IInjectable` which can be injected through a parameter in the constructor.

```
interface IInjectable
{
    void SomeOperation();
}

class SomeClass : IInjectable
{
  public void SomeOperation { ... }
}
```

Listing 3.4: Injection Example(C#)

```
class AnotherClass
{
    private IInjectable injectable;

    // Injecting something through the constructor.
    public AnotherClass(IInjectable injectable)
    {
      this.injectable = injectable;
    }

    public void Operation()
    {
        injectable.SomeOperation();
    }
}
```

Listing 3.5: Constructor Injection (C#)

Setter Injection

Listing 3.6 illustrates setter injection. The `IInjectable` interface and implementation of `SomeClass` remain the same as in Listing 3.4. We have adapted the implementation of `AnotherClass` in such a way that it supports setter injection. The `IInjectable` can now be injected through the `Injectable` property.

Interface Injection

Interface injection is illustrated in Listing 3.7. In order to support interface injection, we have to define an interface which defines a method, in this case `Inject`, providing the possibility to inject something into an object. The `IInjectable` interface and implementation of `SomeClass` remain the same as in Listing 3.4. `AnotherClass` now implements the `IInjector` interface, so the `IInjectable` can be injected.

```
class AnotherClass
{
    private IInjectable injectable;

    // Injecting something through a property.
    public IInjectable Injectable
    {
        set
        {
            this.injectable = value;
        }
    }

    public AnotherClass() { }

    public void Operation()
    {
        injectable.SomeOperation();
    }
}
```

Listing 3.6: Setter Injection (C#)

```
interface IInjector
{
  void Inject(IInjectable injectable)
}

class AnotherClass : IInjector
{
    private IInjectable injectable;

    public AnotherClass() { }

    // Injecting something through a specified method.
    public void Inject(IInjectable injectable)
    {
      this.injectable = injectable;
    }

    public void Operation()
    {
        injectable.SomeOperation();
    }
}
```

Listing 3.7: Interface Injection (C#)

Comparison Between Different Forms

While using constructor injection an instance of `AnotherClass` has an instance of an `IInjectable` on creation, so a valid object is created at construction time [Fow04a]. This is not the case with setter and interface injection, which can lead to an uninitialized `injectable` attribute. Therefore the latter approaches require some extra checks, to prevent runtime errors. For interface injection the implementation of an additional interface, in our case the `IInjector` interface, is required in order for an object to support the dependency injection mechanism.

3.5.2 Service Locator

Dependency injection is not the only way to remove dependencies between components. Another approach is to make use of a *service locator* [Fow04a]. The idea behind a service locator is that there is an object that knows how to get a hold of all the services an application might need [Fow04a, Sun02]. In other words, all possible dependencies are known, and an object can reach those dependencies by asking the service locator. Because the service locator is another approach to remove dependencies between components, we reuse the example which we have also used for dependency injection in Section 3.5.1.

The code in Listing 3.4 is also applicable in this example. We have `SomeClass` implementing the `IInjectable` interface. This can be seen as a service or dependency, which is required by another component, in our case, this is `AnotherClass` again, which is presented in Listing 3.8.

```
class AnotherClass
{
    private IInjectable injectable;

    public AnotherClass()
    {
      this.injectable =
        (IInjectable) ServiceLocator.GetInstance().
        Get("injectable");
    }

    public void Operation()
    {
        injectable.SomeOperation();
    }
}
```

Listing 3.8: AnotherClass using a service locator (C#)

The `ServiceLocator` is implemented as a singleton. The singleton pattern ensures only one instance of the ServiceLocator can exist. Available services have to announce their presence to the `ServiceLocator`, by calling the `Register` method in our case. This

```
class ServiceLocator
{
  private static ServiceLocator serviceLocator;
  private Dictionary<string, object> services;

  private ServiceLocator()
  {
    services = new Dictionary<string, object>();
  }

  public static ServiceLocator GetInstance()
  {
    if (serviceLocator == null)
    {
      serviceLocator = new ServiceLocator();
    }

    return serviceLocator;
  }

  public object Get(string serviceName)
  {
    return services[serviceName];
  }

  public void Register(string serviceName, object service)
  {
    services.Add(serviceName, service);
  }
}
```

Listing 3.9: Service Locator (C#)

approach removes the dependency between service and components. In our example we have not considered the physical location of a service. A component should not have any knowledge of the physical location of a service. This knowledge should be gathered by consulting the service locator, which can for example return a service URL.

The `ServiceLocator` implementation in Listing 3.9 is a very simple approach, but it presents the basic idea behind the service locator. We could decide to extend our service locator with a redundancy mechanism. For example, when lots of components make use of the same service, the service might become unavailable due to high load. Therefore we could register multiple identical services with the service locator. The service locator then monitors the service load and returns the service with the lowest load.

3.5.3 Aspect-Oriented Programming

Object-oriented programming (OOP) is a frequently applied programming paradigm. The idea behind OOP is building a software system by decomposing a problem into a number of objects and then writing code for these objects [EFB01, KLM$^+$97]. Objects encapsulate data and behavior into a single entity. One limitation of this approach is that it is hard to encapsulate certain requirements into a single entity, which could for example be a module, an object, a method, etc. [EFB01]. These requirements can be functional, like authentication or business rules, as well as non-functional, like synchronization, logging, or transaction management [EFB01]. These requirements, also called *cross-cutting concerns*, have to be spread over a number of entities in an OOP design, which violates the encapsulation principle. We will illustrate this by giving an example.

```
class Player
{
    private Location location;
    private bool isAlife;

    /* Moves player from the current location to
     * the specified target location, iff target
     * location is not occupied.
     */
    public void Move(Location target)
    {
      if (target.IsOccupied())
      {
        throw new ApplicationException( ... );
      }

      location = target;
    }
}
```

Listing 3.10: Valid Encapsulation (C#)

Listing 3.10 shows a `Player` which can `Move` to a new `Location`, if the target location is not occupied. The simple `Move` method performs the specified behavior, no more, no less.

In Listing 3.11 we have expanded the `Move` method of `Player` to meet some additional requirements. First of all, we have a business rule specifying players can only perform some behavior when they are alive. This business rule does not only apply to the `Move` method of `Player`, but to all methods performing some behavior for the `Player`. For example, think of a `Fire`, `PickUpItem`, `UseItem`, etc. method. Secondly we want to log all player behavior, another requirement which also applies to other methods. We have incorporated these requirements into the `Move` method. The result of this is code which has lost its elegance and simplicity. It actually violates encapsulation principles, the method exhibits more behavior than was specified, which lies outside the area it is responsible for. Aspect-

```
class Player
{
    private Location location;
    private bool isAlife;

    /* Moves player from the current location to
     * the specified target location, iff target
     * location is not occupied.
     */
    public void Move(Location target)
    {
        if (!isAlife)
        {
            Log.Write("Dead man walking ...");

            trow new ApplicationException( ... );
        }

        if (target.IsOccupied())
        {
            Log.Write("Target occupied ...");

            throw new ApplicationException( ... );
        }

        Log.Write("Moving ...");

        location = target;
    }
}
```

Listing 3.11: Encapsulation Violation(C#)

oriented programming can deliver a solution for this problem.

Aspect-oriented programming (AOP) is a technology to apply separation of concerns where the OOP paradigm is insufficient [aop07, EFB01, KLM+97, Fil01]. AOP treats cross-cutting concerns as first class entities called aspects [EFB01]. Cross-cutting concerns are requirements which cannot be placed in a single entity, in other words, they cross-cut the system's basic functionality [KLM+97]. An *aspect* describes the behavior and location of one or more cross-cutting concerns [Cli03]. Those aspects are defined outside the traditional source code and are, together with the source code, composed to a fully working application. We will get back on the details of the composing process later. Those aspects are defined outside the traditional source code and are, together with the source code, composed to a fully working application. We will get back on the details of the composing process later.

In order to compose a fully working application from the source code and the aspects, the composer has to know how to combine the aspects and the source code. Certain points in the source code have to be defined, where the behavior, specified in the aspects, must be executed. These points are part of the aspect and are called *pointcuts* [FK07]. A pointcut is used to specify a set of *join points* [FK07, LC03].

A join point is a well-defined location in the OOP system, which can be modified by an aspect [FK07]. The description of this modification is called *advice* [FK07]. Join points are defined in a join point model, which can contain dynamic and static join points [LC03]. Dynamic join points are well-defined points in the execution flow of the program [LC03]. To illustrate the concept of dynamic join points, we will use the dynamic join point categories of AspectJ[1], one of the largest AOP projects. AspectJ defines three dynamic join point categories, namely execution, call and field access, which each contain a number of join point types, as indicated in Table 3.1 [LC03].

Join point category	Join point types
Execution	Method execution
	Initializer execution
	Constructor execution
	Static initializer execution
	Handler execution
	Object initialization
Call	Method call
	Constructor call
	Object pre-initialization
Field access	Field reference
	Field assignment

Table 3.1: Dynamic Join Point Categories in AspectJ

The opposite of dynamic join points are static join points, where additional attributes, properties, methods and constructors can be added to the software system [FK07]. The location of this insertion is less important than with dynamic insertion, because these can be inserted 'anywhere' in the class definition.

Figure 3.1 gives an overview of the things we have just discussed regarding aspects. By now we have a general idea about how aspects express some behavior which must be composed at a certain location in the software system, but we have not yet discussed the actual composing process.

The process in which the aspects and source code are composed into a fully working software system is called *weaving*. We distinguish two kinds of weavers, namely static weavers and dynamic weavers [FK07]. Both forms of weaving are discussed below.

With *static weaving* the program is changed, by inserting advice at join points, before run-time [Vas04, FECA04]. This can be done by inserting the advice in the source code of the program, or by inserting the advice in the intermediate code generated by the compiler, for example in the case of Java or C#, Java Bytecode or Microsoft Intermediate

[1]For more information, see `www.aspectj.org`.

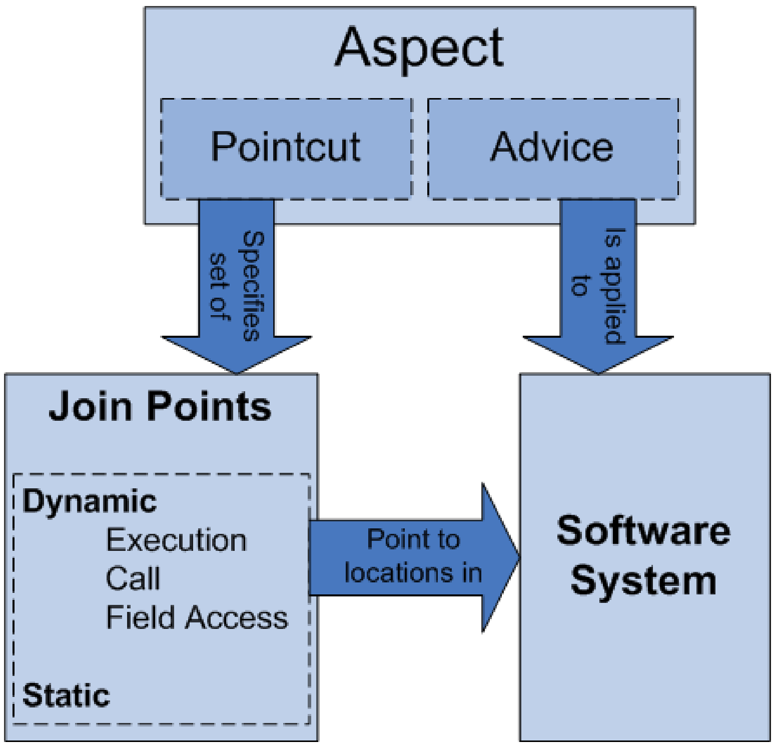


Figure 3.1: Aspect Overview

Language respectively. Static weaving is applied in among others AspectJ [KHH^{+}01], AspectC [CKFS01], and HyperJ [OT00]. Both approaches are illustrated in Figure 3.2. The first approach, inserting the advice in the source code has the advantage that the woven code must still be compiled, which diminishes the possibility to end up with a malicious program [FK07].

A disadvantage of static weaving is that it is difficult to identify aspects in woven code [FK07]. As a consequence it is very time-consuming, or even impossible, to adapt or replace aspects dynamically [FK07].

With *dynamic weaving* the program is modified at run-time, by inserting advice at join points, and providing the possibility to perform additional modifications while the program is running [Vas04, FECA04]. Dynamic weaving is applied in among others JBoss [JBo07], AspectWerkz [Bon04], and PROSE [PGA02]. Dynamic weaving is illustrated in Figure 3.3.

3.5.4 Configuration Files

Configuration files are used to store configuration information, which can be seen as initial settings for software systems. Configuration files are applied in frameworks as well as ordinary software applications. These configuration files where traditionally written in ASCII, but XML became one of the most popular standards over the last years [con07]. Examples of frameworks making use of XML configuration files are the Spring framework,

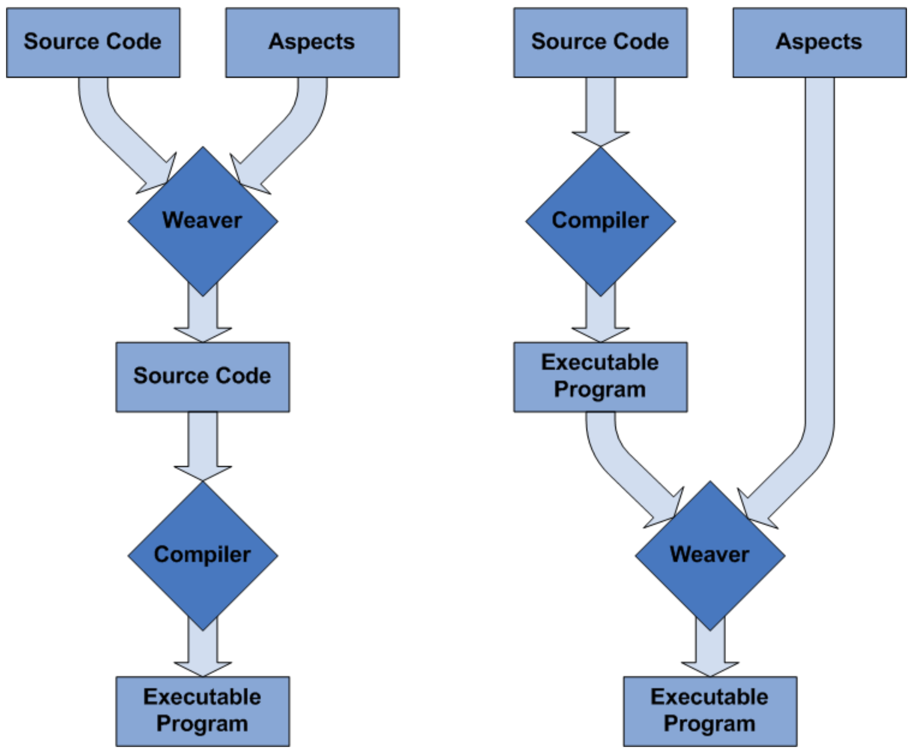


Figure 3.2: Static Weaving

Windows Presentation Foundation, and many others. We distinguish two approaches for reading these configuration files [con07]. They can be read by a software system just once, for example at load-time. Another approach is to constantly monitor the configuration file at run-time. When the contents of the configuration file change, the software system can change its setting to the new configuration.

But why do we use configuration files? In many cases it is possible to use program code to reach the same effects. But we should not simply draw the conclusion that using configuration files just causes overhead. The underlying idea behind configuration files is that certain information, which is likely to be changed when the system is for example used in different environments, or on different locations, can be changed without the need to recompile the system [Fow04a]. As mentioned earlier, frameworks are building structures, applied in various environments. Therefore configuration files are a successful mechanism to specify settings.

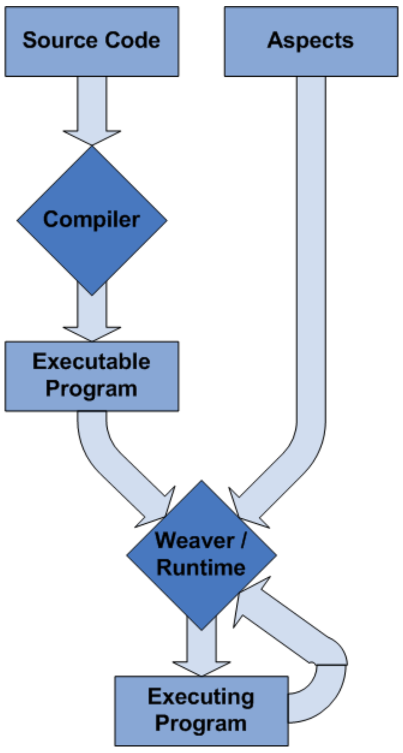


Figure 3.3: Dynamic Weaving

Chapter 4

Mapping Frameworks on the Problem Domain

In this chapter we identify a number of possibilities for applying frameworks to increase productivity in our problem domain. We continue by exploring a number of candidate frameworks for each of these applications possibilities.

4.1 Application Possibilities

In this section we identify a number of possibilities for applying frameworks in Endeavour. We have identified the applications possibilities listed below.

Application on specific layer One possibility is to apply a framework on a specific application layer, GUI, Business, or Data Access. We could for example apply a framework to increase the reusability of different windows, thereby targeting the GUI layer.

Application on data transfer between layers In stead of targeting a specific application layer, we can also decide to apply a framework between layers. A good example of a framework for such kind of application is the Hibernate framework[1], providing an object-relational mapping between database records and objects in the form of business entities. This is just one of many data mapping steps performed in SOA applications. The mapping 'pipeline' is currently executed by mapping from records, to datasets, to objects, to xml, to objects, and finally to the user interface.

Application in a vertical approach Applying a framework in a vertical approach is actually a combination of the two application possibilities identified above. In a vertical approach, the target is from the top to the bottom of the application. This approach requires a framework targeting all application layers. An example of a framework of this kind is the Spring framework[2] [AA05].

[1]For more information visit `http://www.hibernate.org`
[2]For more information visit `http://www.springframework.org`

4.2 Possibilities Explored

In this section we discuss candidate frameworks for each of the application possibilities identified in Section 4.1, satisfying the project criteria identified in Section 6. By further discussing these frameworks for each of the application possibilities, we provide a more concrete insight for each of them, as illustrated in Table 4.1.

Applications Possibility	Framework
Application on specific layer	Windows Presentation Foundation
Application in a vertical approach	Spring.NET + NHibernate + WPF
Application on data transfer between layers	NHibernate

Table 4.1: Application Possibilities with Frameworks

4.2.1 Windows Presentation Foundation

Windows Presentation Foundation[3] (WPF) is Microsoft's latest technology for creating graphical user interfaces [Nat06]. In this section we discuss a number of reasons for developing WPF.

Integrate Developers and Designers

Nowadays it is very common for applications to 'communicate' with a user by means of a graphical user interface. Designing such a graphical user interface requires specialized skills [And07]. Software developers usually do not possess these skills. Therefore GUI designers used to design a user interface, and deliver this as an image to the software developers. A software developer then recreates these images in the form of an actual user interface, using components provided by his GUI library. Not everything the GUI designer created, can be actually implemented, so the developer has to do some concessions, which have to be validated by the GUI designer again. This is a very time-consuming process, during which the GUI design and the software development process are completely disconnected [And07].

WPF provides the possibility to connect GUI design and software development, by a combination of imperative and declarative programming. The WPF library can be used by any programming language supported by the .NET platform, which is the imperative part. GUI designers are not familiar with these programming languages, because these offer more functionality than necessary for their purpose, and therefore are too complex. For this reason Microsoft introduced a new declarative programming language, named XAML, which stands for Extensible Application Markup Language [Nat06]. XAML is an XML based language in which all elements of the WPF framework can be used to create a GUI. So we have a markup language for designers to create a GUI, but designers prefer to draw

[3]For more information visit `http://msdn2.microsoft.com/en-us/library/ms754130.aspx`

GUI components and visualize the overall result of their creations. Therefore the Microsoft Expression family can be used to visually design and draw components and export them as XAML, which can be easily integrated into the application.

Composition and Customization

WPF controls are extremely composeable, you could for example create a combobox filled with animated buttons, or a menu with video clips [Nat06]. Although these particular customizations might sound horrible, you do not have to write a large amount of code to customize such controls and to incorporate them into your application.

Potential of Windows Presentation Foundation

WPF provides a rich environment for developing and designing graphical user interfaces. It couples the working environments of GUI designers and software developers. This is a powerful mechanism when somewhat more advanced user interfaces are required. For more traditional, standard user interfaces it is hard to determine if applying WPF has an impact on productivity, because [And07] and [Nat06] only illustrate the additional possibilities WPF provides with respect to the Windows Forms approach.

4.2.2 Spring.NET

The Spring.NET[4] framework provides a basis for building enterprise .NET applications and makes this process easier [PES$^+$06, WB05]. Spring.NET is based on the Spring framework for Java; the core concepts found in the Java version have also been applied to .NET. In this section we discuss a number of reasons for developing the Spring framework. Because the core concepts of both the .NET and Java version are equivalent, we use literature of both versions to discuss these reasons. We discuss the following reasons for developing the Spring framework, providing architectural structure, support for cross-cutting concerns, and providing additional building blocks.

Providing Architectural Structure

.NET is a framework which provides a wealth of functionality for architecting and building applications [PES$^+$06]. This functionality ranges from basic building blocks of primitive types and classes, to full-featured application servers and web frameworks. Forming a coherent system from these building blocks is not actively supported by the .NET environment [PES$^+$06]. This part of software development is left to the software architects and developers. Spring provides an architectural structure to solve this problem, by using Inversion of Control.

Inversion of Control Inversion of Control is the mechanism applied by Spring to form a number of building blocks into a coherent application, following a formalized approach

[4]For more information visit `http://www.springframework.net/`

[PES+06]. This approach is based on best practices that have been proven over the years in numerous applications. The idea behind Inversion of Control is providing an environment which actively supports loose coupling, therefore improving reusability and maintainability as we have discussed earlier.

Support for Cross-cutting Concerns

In Section 3.5.3 we have discussed cross-cutting concerns, which we defined as requirements which cannot be placed in a single entity, in other words, they cross-cut the system's basic functionality. Spring provides aspect-oriented programming in order to support these cross-cutting concerns [PES+06, WB05]. This is an addition to the .NET framework which supports object oriented modularization, where AOP adds the ability to 'modularize' cross-cutting concerns.

Providing Additional Building Blocks

We just discussed the two main features of Spring.NET; Aspect oriented programming and Inversion of Control [PES+06]. The Spring.NET framework also provides some additional building blocks [PES+06]. The Web library extends ASP.NET with dependency injection. The Data library supports a Data Access Layer abstraction, supporting a variety of data access providers, like ADO.NET. Spring.NET also provides an ORM integration library, to support popular object relational mapping libraries, like NHibernate. By using these additional building blocks an application developer can make use of an additional layer of abstraction on top of the .NET framework. This layer provides additional features and decouples the implementation from concrete technologies, like a Microsoft SQL Server Database or a MySQL Database Server.

Potential of Spring.NET

The two main features of Spring.NET, providing architectural structure, and support for cross-cuttings concerns are likely to have a positive effect on the productivity and maintainability. We cannot provide a more concrete estimation of the effects on productivity and maintainability of the application, because it is very hard to determine what parts of an application are affected by these features.

4.2.3 NHibernate

In this section we discuss NHibernate[5]. NHibernate is an object/relational mapping framework [BK05] and is ported from the Hibernate Core for Java to the .NET framework [nhi07]. Because NHibernate is a port of Hibernate Core, we can also use [BK05] to illustrate the ideas behind NHibernate. We first elaborate on the so-called paradigm mismatch, to explain the reasons why NHibernate is developed. Then we describe the qualities the NHibernate framework can offer.

[5]For more information visit `http://www.nhibernate.org`

Paradigm Mismatch

To explain the *paradigm mismatch*, we start by giving an example. In Listing 4.1 we have defined two business objects in C#, a `Customer` and a `BankAccount`. Both objects have a number of attributes, and a `Customer` has a number of `BankAccount` objects.

```
public class Customer
{
    private string username;
    private string name;
    private string address;
    private IList<BankAccount> bankAccounts;

    ...
}

public class BankAccount
{
    private string accountNumber;
    private string accountType;

    ...
}
```

Listing 4.1: No Paradigm Mismatch (C#)

These business objects are stored in a relational database, in the tables defined in Listing 4.2. We quickly notice that the object and relational definition look quite similar. It is quite straightforward to write code to perform basic CRUD—create, read, update, and delete—operations for a `Customer` and its `BankAccount` entities.

```
create table CUSTOMER
(
  USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
  NAME VARCHAR(40) NOT NULL,
  ADDRESS VARCHAR(100)
)

create table BANK_ACCOUNT
(
  ACCOUNT_NUMBER VARCHAR(10) NOT NULL PRIMARY KEY,
  ACCOUNT_TYPE VARCHAR(10) NOT NULL,
  USERNAME VARCHAR(15) FOREIGN KEY REFERENCES CUSTOMER
)
```

Listing 4.2: No Paradigm Mismatch (SQL)

In our example we have defined `address` as a `string` consisting of up to 100 characters. In the next example we redefine `Address` as illustrated in Listing 4.3, which we use to illustrate the paradigm mismatch.

```
public class Customer
{
    private string username;
    private string name;
    private Address address;
    private IList<BankAccount> bankAccounts;

    ...
}

public class Address
{
  private string street;
  private string number;
  private string zipcode;
  private string city;
  private string country;

  ...
}

public class BankAccount
{
    private string accountNumber;
    private string accountType;

    ...
}
```

Listing 4.3: Paradigm Mismatch (C#)

Listing 4.4 shows the database tables to store the address information separately. On obvious solution is to add an `ADDRESS` table, but in database design it is more common to add the address fields to the `CUSTOMER` table, in individual columns [BK05]. This design is likely to perform better, because it is not necessary to perform a join operation to retrieve customer and address data in a single query.

When we compare our object and database definitions, we can easily identify a mismatch. Our tables do not perfectly correspond to our business objects anymore. Although we can easily match them, by adding an `ADDRESS` table to our database definition, or by adding the separated address fields to our `Customer` object, this approach does not follow database design principles or software design principles, respectively. [BK05] identifies this as the *problem of granularity*. Database entities and business objects do not always have the same level of granularity.

```sql
create table CUSTOMER
(
  USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
  NAME VARCHAR(40) NOT NULL,
  ADDRESS_STREET VARCHAR(30),
  ADDRESS_NUMBER VARCHAR(6),
  ADDRESS_ZIPCODE VARCHAR(6),
  ADDRESS_CITY VARCHAR(20),
  ADDRESS_COUNTRY VARCHAR(20)
)

create table BANK_ACCOUNT
(
  ACCOUNT_NUMBER VARCHAR(10) NOT NULL PRIMARY KEY,
  ACCOUNT_TYPE VARCHAR(10) NOT NULL,
  USERNAME VARCHAR(15) FOREIGN KEY REFERENCES CUSTOMER
)
```

Listing 4.4: Paradigm Mismatch (SQL)

According to [BK05], the paradigm mismatch can be separated into several problems, namely the problem of granularity, the problem of subtypes, the problem of identity, the problem relating to associations, and the problem of object graph navigation. We briefly discuss and explain all these problems. For a full discussion of these topics we refer to [BK05].

Problem of Identity The problem of granularity is already covered in the example above, therefore we start discussing the *problem of identity*. The problem of identity deals with determining when two objects are identical [BK05]. When we for example take business objects written in C#, we have two equivalent ways of determining if those objects are identical, namely using the == operator, and the `Equals` method. The identity of a database row is expressed as the primary key of that row. By comparing the primary keys of database entities, we can determine whether these entities are identical. These identity determination mechanisms are not naturally equivalent [BK05].

Problem Relating to Associations The *problem relating to associations* covers the subtle differences between associations as object references in object-oriented languages and associations represented as foreign key columns in relational databases [BK05]. In an object-oriented language like C# it is possible to create many-to-many associations. Between tables in a relational database only one-to-one or one-to-many relationships can exist. If we want to represent a many-to-many relationship in a relational database, a so called link table is required [BK05]. A row of a link table represents a set of database entities which are related by referring to them using a foreign key. This link table is not represented in the object-oriented model, since there exists a built-in mechanism to represent many-to-many relationships [BK05].

Problem of Object Graph Navigation There exists a fundamental difference in the way you access business objects and database entities [BK05]. The problem regarding this issue is identified as the *problem of object graph navigation*. Let us go back to the C# example we started this section with. If we want to know the `accountNumber` of a set of `BankAccount` objects of a certain `Customer`, we could access this information by using something like `customer.BankAccounts.AccountNumber`. This way of accessing object-oriented data is called walking the object graph [BK05]. Accessing data in a relational database following the same approach, would not be a good idea since it is highly inefficient [BK05]. In order to make efficient use of a relational database it is important to minimize the number of requests to the database [BK05]. In the case of walking the object graph we reach our information by making one step in the object hierarchy at a time. It is not necessary to know on beforehand what information we want to access. To efficiently retrieve information from a relational database, we have to consider what portion of information we want to access, and determine how we should travel through this information with a query.

Potential of NHibernate

Now we have discussed a set of problems together forming the paradigm mismatch, it is also interesting to gain some insight in how big this problem actually is. According to [BK05] the main purpose of up to 30 percent of Java application code is to function as a bridge to overcome the object/relational paradigm mismatch. NHibernate provides a solution to this problem. As mentioned earlier, NHibernate is an *object/relational mapping* (ORM) framework. ORM is the automated and transparent persistence of business objects in a relational database, using metadata describing the mapping between the objects and the database entities [BK05]. NHibernate offers a number of benefits, namely an increase in productivity, improved maintainability, better performance, and vendor independence [BK05]. We discuss each of these benefits separately.

We already illustrated an increase in productivity by mentioning that up to 30 percent of Java application serves to overcome the object/relational paradigm mismatch. An ORM framework provides a solution to this problem and thereby significantly reduces development time [BK05].

The application of an ORM framework reduces the amount of manual lines of code [BK05]. This forms a basis for the improved maintainability of an application build on top of an ORM framework. A very important reason for the improved maintainability in an ORM scenario is the reduced level of change dependencies between business objects and database entities. Changing one representation does not automatically imply that the other representation is also changed, because an ORM framework offers a buffer between these two [BK05]. This buffer also guarantees that it is not necessary to make trade-offs during the design of either representation, in other words, business objects as well as database entities can be designed according to the specific design principles of each field.

Applying an ORM framework is likely to result in better performance than an hand-coded solution. While designing and developing an ORM framework a lot of time and effort is spent on optimizations of the resulting implementation [BK05]. One could compare the application of an ORM framework versus a hand-coded solution with writing assembly versus a higher level programming language. Assembly code could perform better than code generated by a compiler, but the effort spent on optimizations in the compiler are almost impossible to realize by a single developer [BK05].

Another benefit of an ORM framework is that it decreases the level of coupling of the application to the underlying SQL database and SQL dialect [BK05]. Most ORM frameworks support a number of different databases, thereby increasing the level of portability of an application. This results in a vendor independent solution.

Chapter 5

Narrowing Research Scope

In this chapter we describe the results of our productivity bottleneck analysis for Endeavour applications. The questions we answer during this phase is what specific productivity bottlenecks in Endeavour are, and what characteristics a candidate framework should have to effectively target these bottlenecks. Then we continue by ranking these bottlenecks, according to the expected increase in productivity and according to Info Support's preference. Furthermore we identify a number of candidate frameworks that target these bottlenecks, and finally propose a choice for a set of frameworks on top of which we build our prototype.

5.1 Previous Endeavour Projects

The Professional Development Center of Info Support has collected data about previous projects done with Endeavour. We have extracted some interesting numbers from these data, which are put in Table 5.1, 5.2, and 5.3. These numbers can give a more specific indication about the location where we can reach the largest increase in productivity. Each table displays some characteristics of a specific project. For information disclosure reasons, we have decided not to mention the project names. We do not think this has any negative effects on the credibility of the data or our research project.

On the first row the development year, the number of function points, and the number of hours spent on the project are given. The rest of the specified values are:

PART	A project is divided into four parts, namely GUI, business, data, and conversion/batch.
MAN SLOC	Manual source lines of code.
GEN SLOC	Generated source lines of code.
%MAN/SLOC	Percentage of 'man sloc' of the amount of 'man sloc' + 'gen sloc'.
%MAN/TSLOC	Percentage of 'man sloc' of the total amount of 'man sloc' + 'man sloc' of all four parts.

Project A - 2004 - 80FP - 700Hours				
Part	Man SLOC	Gen SLOC	%Man /SLOC	%Man /TSLOC
GUI	11207	0	100%	≈57%
Business	2425	0	100%	≈13%
Data	5925	0	100%	≈30%
Conversion/Batch	1	0	100%	≈0%
TSLOC	19558	0		

Table 5.1: Endeavour - Project Data A

Project B - 2003 - 132FP - 860Hours				
Part	Man SLOC	Gen SLOC	%Man /SLOC	%Man /TSLOC
GUI	9878	0	100%	≈59%
Business	3368	0	100%	≈20%
Data	2294	0	100%	≈14%
Conversion/Batch	1328	0	100%	≈8%
TSLOC	16868	0		

Table 5.2: Endeavour - Project Data B

Project C - 2006 - 220FP - 1594Hours				
Part	Man SLOC	Gen SLOC	%Man /SLOC	%Man /TSLOC
GUI	18374	4989	≈79%	≈40%
Business	7051	0	≈100%	≈15%
Data	931	14471	≈6%	≈2%
Conversion/Batch	334	0	≈100%	≈1%
TSLOC	26690	19460		

Table 5.3: Endeavour - Project Data C

Table 5.1, 5.2, and 5.3 all indicate that a large part—57%, 59%, and 40%, respectively—of the manual SLOC are located in the GUI part of the projects. Therefore it is *likely* that we can gain the largest increase in productivity in the GUI. We emphasize likely, because we have not analyzed the source code of the projects ourselves. For example, when most lines of code have a level of complexity which prohibits that it can be captured in a more generic design, we have little room to optimize. Such an assumption seems unlikely, because in Project C in Table 5.3, a large part of the GUI consists of generated code.

A drawback of the choice of targeting the GUI with the application of frameworks is that the Professional Development Center (PDC) of Info Support is also planning to target the GUI with productivity increasing measures. When we target the GUI, just like the PDC is about to, this would result in performing the same research in parallel, possibly a waste of time.

5.2 Identifying Bottlenecks

To get a more concrete idea about which productivity bottlenecks exist in Endeavour applications, we took two possibilities into consideration.

1. A discussion with Endeavour application developers with more than three years of experience in developing Endeavour applications or Endeavour itself.

2. Performing an analysis on existing Endeavour applications, by inspecting the code base of some Endeavour applications.

For the latter approach, performing an analysis on existing applications by inspecting their code bases, we have identified two drawbacks. Firstly, it is very hard to identify a number of bottlenecks on our own, because a high level of experience with Endeavour applications is required, which we do not have. Secondly, a code-analysis is very time-consuming, because the analysis has to be performed at the source code level. This could possibly be automated, with a so called similarity analysis [BYM$^+$98] for example. The result of a similarity analysis shows the amount of parts of the software system with similar functionality, in other words bottlenecks. These results have to be analyzed by hand, in order to understand the problem and to identify candidate frameworks that target these bottlenecks. Therefore the second alternative is still likely to consume more time than our other approach.

For these reasons we decided to focus on the first approach, namely a discussion with Endeavour application developers. We planned to use the other possibility as a contingency plan, when the discussion does not result in the desired outcome. There was no need to use the contingency plan, because the discussion resulted in sufficient information to work with.

We organized a discussion with two PDC members. As mentioned earlier, our expertise with Endeavour is quite limited, therefore we clearly indicated this at the start of the discussion. Then we discussed the goal of the discussion, namely identifying a number of possible productivity bottlenecks in Endeavour. As discussed in Section 1.4.2, the PDC is responsible for continuously improving Endeavour. For this reason we only indicated a few applications possibilities, as discussed in Section 4.1, to start up the discussion. This kick-off resulted in an ongoing discussion and the ideas simply piled up.

This section describes the productivity bottlenecks we identified during the discussion with Endeavour experts. We have identified three bottlenecks; in GUI development, data management, and infrastructural bottlenecks. We discuss each of these bottlenecks in more detail in the paragraphs below.

5.2.1 GUI Development

As we have seen in Section 5.1 the GUI application layer contains a large amount—ranging from 40 up to almost 60 percent—of manual lines of code. Therefore we mentioned that it is likely that we could increase productivity in this layer of the application.

During the interview we tried to get a more detailed insight in the possibilities to increase productivity in this application layer. Developing enterprise business applications involves designing and developing lots of screens with similar functionality. For example, it is quite common to have a screen in which a user can enter some data, and when finished, save it. Another example is a screen in which a user can search a collection of data with some search criteria, which finally shows the data satisfying the search criteria. It is not difficult to come up with some more examples.

In the current development approach these screens are developed separately. This means that large parts of similar code are rewritten for similar screens, which affects productivity and maintainability. Therefore we propose to search for frameworks which provide possibilities to design and develop a number of generic screens, which can be reused several times. We formed some initial ideas about the desired capabilities of such a framework, as presented below.

- The framework should provide a possibility to bind data objects to a screen. A screen to enter data should not only support a `Customer` but also a `Car` or any other object. This provides the opportunity to easily reuse screens.

- Practical experience at Info Support indicates that developers tend to throw a general solution away when it slightly deviates from their desired solution, because it seems more work to change the existing solution than building a new solution. Therefore it is important that a provided solution is easily customizable. When a developer wants to change an existing, general solution, it should only be required to specify the differences with regard to the existing solution.

5.2.2 Infrastructural Bottlenecks

In applications it is common to implement some infrastructural requirements, like logging, transaction management, authorization, exception handling, etc. As we have seen in Section 3.5.3 these infrastructural requirements should not be captured in an object-oriented design, because this cannot capture these cross-cutting concerns in an appropriate way. Therefore we propose to investigate the possibility of applying an AOP framework. A desired capability of such a framework is indicated below.

- The framework should provide a means to apply infrastructural requirements on a method level. Applying such requirements should have a minimal impact on the source code of an application.

5.2.3 Data Management

Applications in our problem domain usually make use of a database. In Section 4.2.3 we have discussed that mapping business objects on database entities is not just a one-on-one mapping process. The mapping requires a large amount of manual lines of code. This asks for a framework to target that problem.

Another problem with large collections of data in business applications is how to effectively perform operations on it. What do we mean with this? For example, when we want to select a subset of data we can easily query the database. The next step is to map these results on object entities. To perform an operation—for instance to verify if a set of objects satisfies some business rules—on these object entities, we could write a `foreach` loop with an `if` statement in order to perform this operation. However, we would prefer to try a more data-centered approach, by simply selecting the objects which satisfy our search criteria. Below we have given some desired capabilities of a framework that satisfies the demands that we have just discussed.

- The framework should provide a solution for the object/relational mapping paradigm mismatch.
- Frameworks promise to take a lot of work out of the developer's hands. Although this might seem a good property at first, this could also be a drawback. When a framework does not provide the possibility to change the default behavior, in other words offers customizability, the developer cannot apply the framework in every situation.
- If a new version of the framework works with another database format, for example some tables are changed, it should provide the possibility to migrate the previous version of the database to the new format. Persistent database data cannot just be thrown away.
- The framework should support a means to query objects in a more data-centered approach.

5.3 Ranking Bottlenecks to Target

In this section we plan to rank the bottlenecks identified in the previous section, based on the following two criteria:

1. *Expected productivity increase.*
2. *Info Support's preference.*

Ranking our bottlenecks is the first step in making a choice for a specific combination of frameworks. We discuss the given criteria in the next two paragraphs.

5.3.1 Expected Productivity Increase

As we mentioned earlier, the GUI layer of small and simple Endeavour applications contains about 40 up to 60 percent of manual lines of code. So it is likely that we can gain a significant increase in productivity by applying a framework targeting the GUI layer.

Estimating an increase in productivity for applying a framework to solve infrastructural bottlenecks is more difficult. We do not have figures indicating the number of manual lines of code which have to be written for infrastructural aspects. By applying AOP we could treat infrastructural aspects, which could be characterized as cross-cutting concerns, as first class citizens. This improves maintainability, but we have no research showing an increase in productivity.

To determine the productivity increase of applying a framework targeting data management, we look back at Section 4.2.3 where we discussed NHibernate. [BK05] claims that the main purpose of up to 30 percent of application code is to function as a bridge to overcome the object/relational paradigm mismatch. The data layer of small and simple Endeavour applications contains about 2 up to 30 percent of manual lines of code, as discussed in Section 5.1. Keeping these numbers in mind, we do expect to reach an increase in productivity by applying a framework targeting data management bottlenecks.

In Table 5.4 we have ordered the expected productivity increase for each identified bottleneck from high to low.

Productivity rank	Bottleneck
1	GUI development
2	Data management
3	Infrastructural bottlenecks

Table 5.4: Productivity Increase for Each Bottleneck

5.3.2 Info Support's Preference

Info Support has investigated the possibilities of applying a framework to target the GUI development bottlenecks in a way we discussed in Section 5.2.1. Their investigation resulted in the conclusion that a framework with the desired capabilities is not yet available. Therefore they are developing a prototype of a framework to investigate what the possibilities in this area are. This prototype would be an interesting framework candidate for our project assignment. Unfortunately the prototype will not be finished in time, and therefore we cannot use it during our project assignment.

To prevent the situation where we perform the same research in parallel, Info Support suggests to investigate the effects on productivity of applying frameworks targeting infrastructural bottlenecks and data management. In Table 5.5 we have ordered the bottlenecks according to the ranking of Info Support from high to low.

Preference rank	Bottleneck
1	Data management
2	Infrastructural bottlenecks
3	GUI development

Table 5.5: Info Support's Preference for Each Bottleneck

5.4 Identifying Candidate Frameworks

In this section we identify a number of candidate frameworks to target the bottlenecks identified in Section 5.2. We searched the web for these frameworks. In Chapter 2 we already defined that a candidate framework must be applicable in a .NET environment. We also mentioned that the GUI of our application is web-based in Section 2.4. All candidate frameworks satisfy these criteria.

To get some initial ideas about the quality of the framework, we took a quick look at the documentation of the candidate frameworks. Our impression of the frameworks documentation is indicated in the tables in the column DOC, ranging from + (good) to - (bad). We also formed an idea about the popularity of a framework by looking at the number of downloads per month of the framework in 2007. We have used [sds07] as source of information for these download statistics. In many cases we have used small pieces of text from the websites of the frameworks to give an impression about the capabilities of the framework.

5.4.1 GUI Development

Table 5.6 gives an overview of candidate frameworks targeting GUI development. In the next paragraphs we give a brief description of each framework.

Windows Presentation Foundation

Windows Presentation Foundation (WPF) is Microsoft's latest technology for creating graphical user interfaces [Nat06]. In Section 4.2.1 we discussed two reasons for using WPF, namely integrate developers and designers, and composition and customization.

Monorail

MonoRail is a MVC Web Framework inspired by Action Pack. MonoRail differs from the standard WebForms way of development as it enforces separation of concerns; controllers just handle application flow, models represent the data, and the view is just concerned about presentation logic. Consequently, you write less code and end up with a more maintainable application [mon07].

PixelDragonsMVC.NET

PixelDragonsMVC.NET is an open source, easy-to-use MVC framework for use with ASP.NET 2.0. It has built-in support for NHibernate and Log4Net and

Framework	Downloads	Doc	URL
Windows Presentation Foundation	Unknown	+	`http://msdn2.microsoft.com/en-us/library/ms754130.aspx`
Monorail	Unknown	+	`www.castleproject.org/monorail`
PixelDragonsMVC.NET	75/month	+/-	`www.codeproject.com/aspnet/PixelDragonsMVCdotNET.asp`
Maverick.NET	450/month	+	`mavnet.sourceforge.net`
Microsoft UIP	Unknown	+	`msdn2.microsoft.com/en-us/library/ms998252.aspx`

Table 5.6: Candidate Frameworks Targeting GUI Development

was produced by Pixel Dragons Ltd. to make creating ASP.NET 2.0 applications as easy as possible for us and you [pdm07].

Maverick.NET

Maverick.NET is a .NET port of Maverick, a Model-View-Controller (a.k.a. "Model 2") framework for web publishing. It is a minimalist framework which focuses solely on MVC logic, allowing you to generate presentation using a variety of templating and transformation technologies [mav07].

Microsoft UIP

The User Interface Process Application Block provides a simple yet extensible framework for developing user interface processes. It is designed to abstract the control flow and state management out of the user interface layer into a user interface process layer. This enables you to write generic code for the control flow and state management of different types of applications (for example, Web applications and Windows-based applications) and helps you write applications that manage users' tasks in complex scenarios (for example, suspending and resuming stateful tasks). This leads to simpler development and maintenance of complex applications. The User Interface Process Application Block can easily be used as a building block in your own .NET application [uip07].

5.4.2 Infrastructural Bottlenecks

Table 5.7 gives an overview of candidate frameworks targeting infrastructural bottlenecks. The table contains two extra columns with respect to the previous table. AOP indicates whether the framework provides aspect oriented programming. DI indicates whether the framework provides dependency injection. The next paragraphs give a brief description of each framework.

Spring.NET

The Spring.NET framework provides a basis for building enterprise .NET applications and makes this process easier [PES$^+$06, WB05]. Spring.NET is based on the Spring framework for Java; the core concepts found in the Java version have also been applied to .NET. In Section 4.2.2 we discussed the following reasons for developing the Spring framework, providing architectural structure, support for cross-cutting concerns, and providing additional building blocks.

PicoContainer.NET

PicoContainer is a highly embeddable container for components that honor the dependency injection pattern. Despite it being very compact in size (the

Framework	Downloads	Doc	AOP	DI	URL
Spring.NET	3500/month	+	Yes	Yes	`www.springframework.net`
PicoContainer.NET	Unknown	+	No	Yes	`www.picocontainer.org`
ObjectBuilder	Unknown	-	No	Yes	`www.codeplex.com/ObjectBuilder`
Aspect#	Unknown	+/-	Yes	No	`www.castleproject.org/aspectsharp`
LOOM.NET	Unknown	+/-	Yes	No	`www.dcl.hpi.uni-potsdam.de/research/loom`

Table 5.7: Candidate Frameworks Targeting Infrastructural Bottlenecks

core is 128K and it has no mandatory dependencies outside the JDK), PicoContainer supports different dependency injection types (Constructor, Setter, Annotated Field and Method) and offers multiple lifecycle and monitoring strategies. PicoContainer has originally been implemented in Java but is also available for other platforms and languages [pic07].

ObjectBuilder

ObjectBuilder is a framework for building dependency injection systems, originally written by the Microsoft patterns and practices group. We are currently looking for your participation in deciding where ObjectBuilder should go [obj07].

Aspect#

Aspect# is an AOP (Aspect Oriented Programming) framework for the CLI (.Net and Mono). It relies on DynamicProxy and offers a built-in language to declare and configure aspects, and is compliant with AopAlliance. It became part of the Castle Project in June 2005. Aspect# promotes:

Separation of Concerns(SoC) Code reuse Decomposition

And it offers, plus the basic AOP capabilities, Mixins and a Ruby like configuration language making the framework flexible and easy to use [asp07].

LOOM.NET

The LOOM .NET project aims to investigate and promote the usage of AOP in the context of the Microsoft .NET framework. We have been developing two AOP tools (so called weavers) that all implement different approaches to weaving: A runtime weaver creating weaved objects on the fly and a static weaver [loo07].

5.4.3 Data Management

Table 5.8 gives an overview of candidate frameworks targeting data management. We specifically searched for ORM frameworks, because they solve the paradigm mismatch as discussed in Section 4.2.3. In the next paragraphs we give a brief description of each candidate framework.

NHibernate

NHibernate is a port of Hibernate Core for Java to the .NET Framework. It handles persisting plain .NET objects to and from an underlying relational

Framework	Downloads	Doc	URL
NHibernate	18000/month	+	`www.nhibernate.org`
OPF3	Unknonwn	+	`www.opf3.com`
ADO.NET Entity with LINQ	Unknown	-	`msdn2.microsoft.com/en-us/library/aa697427(vs.80).aspx`
ObjectMapper.NET	250/month	+	`www.objectmapper.net`

Table 5.8: Candidate Frameworks Targeting Data Management

database. Given an XML description of your entities and relationships, NHibernate automatically generates SQL for loading and storing the objects. Optionally, you can describe your mapping metadata with attributes in your source code.

NHibernate supports transparent persistence, your object classes do not have to follow a restrictive programming model. Persistent classes do not need to implement any interface or inherit from a special base class. This makes it possible to design the business logic using plain .NET (CLR) objects and object-oriented idiom [nhi07].

OPF3

Object Persistent Framework 3 (or as we call it OPF3) is an Object Relational Mapper (ORM) Framework for .NET 2.0. It has been developed for over 2 years specifically for .NET 2.0. You only have to annotate your business objects and OPF3 arranges the communication with the underlying database, by providing the basic CRUD—create, read, update, delete—operations. [opf07].

ADO.NET Entity with LINQ

A primary goal of the upcoming version of ADO.NET is to raise the level of abstraction for data programming, thus helping to eliminate the impedance mismatch between data models and between languages that application developers would otherwise have to deal with. Two innovations that make this move possible are Language-Integrated Query and the ADO.NET Entity Framework. The Entity Framework exists as a new part of the ADO.NET family of technologies. ADO.NET will LINQ-enable many data access components: LINQ to SQL, LINQ to DataSet and LINQ to Entities [ane07].

The ADO.NET framework only partly uses the power of LINQ. With LINQ, the data-access code is abstracted from the underlying storage facility. This means you can use the same code constructs for querying relational databases, XML, and in-memory objects, and you can easily join information across different source domains [LG07]. In other words, LINQ is a higher level query language, which supports various underlying data-sources.

ObjectMapper.NET

Object relational mappers aim to bridge the gap between object oriented programming and relational database models. The ObjectMapper .NET offers state-of-the-art object mapping functionality, yet it also provides advanced features that set it apart from other o/r mappers. Tailored to achieve maximum performance and development ease-of-use this technology will give developers the ability to build business components with unprecedented productivity and optimal runtime performance [omn07].

5.5 Choosing Frameworks

In this section we choose a combination of frameworks from the frameworks discussed in the previous section. These choices are based on the rankings, reflecting expected productivity increase and Info Support's preference, provided in Section 5.3. We also take the popularity, maturity and quality of the documentation of the framework into consideration. Furthermore we consider the ease of integration of a set of frameworks. The frameworks also have to fit in our problem domain, as discussed in Chapter 2. We first discuss a choice for a framework for each bottleneck, and propose a combination of a set of frameworks in the final section.

Data Management

The data management frameworks we have seen in Section 5.4.3, contain two serious alternatives, namely the NHibernate and ADO.NET Entity framework. Considering the ADO.NET Entity framework as a possible candidate might be surprising, because it only is a Community Technology Preview (CTP). We come back on this later on. When we compare OPF3 and ObjectMapper.NET with NHibernate, they do not provide any additional features, and they are far less mature than NHibernate.

The NHibernate framework is the most mature framework of our selection, with about 18000 downloads per month. Our first impression of the documentation of the NHiberate framework is quite good, and several books have been written about the framework. The ADO.NET Entity framework is only a CTP, therefore there are hardly any experiences with this framework and the documentation, as far as it is available, is still in a preliminary stage. However in combination with LINQ this framework promises very powerful possibilities, therefore we considered it as an alternative. In Table 5.9 we give an overview of the differences between the NHibernate and ADO.NET Entity framework.

NHibernate	ADO.NET Entity
(+) Mature	(–) CTP (Community Technology Preview)
(+) Sufficient documentation	(–) Hardly any documentation
(+) Easy integration with Spring.NET	(–) Unknown compatibility with Spring.NET
(+) Numerous positive experiences	(–) Hardly any experiences

Table 5.9: Comparing NHibernate and ADO.NET Entity

Although the idea behind the ADO.NET Entity framework seems promising, we have to realize that it is only a CTP, with hardly any documentation. Building a prototype on top of this framework is likely to result in a lot of additional work, when we compare it to building a prototype with NHibernate. Another drawback of building a prototype with a framework which is only a CTP is that the interfaces of the framework are not steady, a new release can contain totally different interfaces, which results in incompatible versions. Therefore we propose to use the NHibernate framework in our prototype.

Infrastructural Bottlenecks

As discussed in Section 5.2.2, a framework targeting infrastructural bottlenecks should provide a means to apply infrastructural requirements on a method level. An aspect-oriented programming framework is specifically built for this purpose. Therefore we identified a number of candidates for targeting infrastructural bottlenecks in Section 5.4.2.

The Spring.NET framework has sufficient documentation, in digital form as well as actual books. Our impression of the documentation of Spring.NET and PicoContainer.NET is quite good. However, the Spring.NET framework offers aspect-oriented programming and dependency injection, where the other candidate frameworks from Section 5.4.2 only provide one of the two. Furthermore the Spring.NET framework is the most mature of our selection of candidates, with about 3500 downloads per month. Another feature of Spring.NET is that is has build-in support for NHibernate. For these reasons we have decided to make use of Spring.NET in our prototype.

GUI Development

In this section we make a choice for a GUI development framework. We identified five possible candidates in Section 5.4.1. Our impression of the documentation of the candidates ranges from average to good. PixelDragonsMVC.NET is an interesting candidate, because it provides a basis for a complete vertical approach, as discussed in Section 4.2. However, with about 75 downloads per month, it does not seem to attract a lot of attention.

When we look at the desired capabilities for GUI development frameworks, as identified in Section 5.2.1, our candidates do not provide the possibility to bind generic data-objects to screens. As previous research of Info Support indicated, a framework with the desired capabilities is not yet available. Therefore we do not choose a framework for GUI development.

5.6 Choice of Frameworks

We decided to focus our efforts on building the back-end of an application, by applying the Spring.NET and NHibernate framework. The Spring.NET framework has built-in support for NHibernate, and our impression of the quality of the documentation and level of maturity of both frameworks are good. Another reason for this choice is the fact that Info Support is currently developing a GUI development framework, and our choice prevents the possibility of performing the same research in parallel.

Chapter 6

Solution Criteria

In this chapter we discuss some criteria to which are used to determine the quality of our solution. These criteria are used during the validation phase in Chapter 9. In this chapter we discuss each of these criteria in more detail. We identified the following criteria:

- Increase productivity
- Increase maintainability
- Ensure scalability
- Ensure customizability
- Applicable in a SOA
- Targets small and simple applications
- Applicable in a .NET environment

The last three criteria might seem quite obvious, as they are actually domain-related restrictions. The applicability in a .NET environment is quite easy to validate, so this does not require much attention. However, the other two criteria are less obvious. As we are not domain experts, we think it is useful to validate whether our solution actually satisfies these criteria.

6.1 Increase Productivity

Developing software at Info Support is still a labor intensive task; a lot of time is spent on writing code manually. With our solution we aim to *reduce the amount of manually written code* by applying frameworks. This should lead to an increase in productivity. However, as we discussed in Section 2.1, a reduction of the manually written lines of code does not automatically imply an increase in productivity, because we should also take the *time needed* to write the reduced amount of code into account. We could for example apply a framework, which is very hard to configure. Although you only have to write a few lines of code, you might spent the same amount of time writing these lines of code as with the original approach. In this case we should not draw the conclusion that we have reached an increase in productivity, because we have only sacrificed the ease of development for a reduction of the source lines of code. We have to be very careful in drawing conclusions regarding an increase in productivity.

6.2 Increase Maintainability

Anticipating is an important software design goal [AM04]. Using a framework should not result in an application with a lower level of maintainability. The source code of an application built on top of a framework should be *easy to understand*, thereby supporting the maintenance process. When a software application is modified, the application has to be tested again. Testing is also part of the maintenance process. *Constructing for verification* is another software design goal. Another factor influencing maintainability of the resulting application is the *stability of the framework*. With stability we mean the size and impact of changes of the underlying framework that have a direct effect on the application. When frameworks are unstable, it is hard to incorporate a new version of the framework into your application. During the validation phase we discuss the effects on maintainability in the light of the characteristics mentioned above.

6.3 Ensure Scalability

Although we developed our solution for small and simple applications, we tried to keep the option open to scale it to an enterprise service-oriented application. Scaling our solution to an enterprise level, means adding additional services to the software system. During the validation phase we discuss to what extent we can scale our current solution, and what the restrictions of the current approach are in this area.

6.4 Ensure Customizability

By applying frameworks, we can use a lot of functionality out-of-the-box. This functionality is likely to result in an increase in productivity when applying frameworks. However, this functionality also has certain restrictions. For example, when the framework has slightly different functionality than required, we should be able to adapt or extend the functionality in such a way that we can still use the framework in our application. In other words, we should have the ability to customize the framework's functionality in such a way that it fits our requirements. During the validation phase we discuss customizability limitations of applying frameworks.

6.5 Applicable in a SOA

Applications are designed and developed according to the service-oriented architecture design principles of Endeavour. The application of a framework should not prohibit the possibility to develop an application according to those design principles. During the validation we discuss possible limitations that occur when using our chosen frameworks. We also discuss whether our prototype actually conforms to a service-oriented architecture.

6.6 Target Small and Simple Applications

As we already discussed in Section 2, our goal is to increase productivity in building small and simple applications, where a small application consists of up to 300 function points. We defined how small and simple applications built with Endeavour look like and we discussed the number of each type of configuration items which are most likely to occur. When choosing frameworks for further exploration, we keep our target applications in mind. Choosing a framework which only improves productivity in large applications does not fit into our research area. During the validation phase we verify whether we have actually targeted these kind of applications.

6.7 Applicable in a .NET Environment

Applications in Endeavour are developed on top of the .NET framework. For this reason we only explore frameworks with .NET support. Therefore our solution is applicable in a .NET environment. Although this is a criterion for our solution, there is no further need to validate this criterion.

Chapter 7

Solution: MyFramework

In this chapter we discuss the solution, called MyFramework, we developed during the implementation phase. MyFramework can be seen as an additional layer on top of the Spring.NET and NHibernate frameworks in order to improve the ease of use of these frameworks in our problem domain—small and simple service-oriented applications. We do not discuss every single detail of our implementation, but decided to take a closer look at the most interesting aspects. We also discuss how we incorporated the Spring.NET and NHibernate frameworks in our solution. Finally we discuss the limitations of our approach.

Before we continue, we would like to point out that MyFramework is a prototype which is built for demonstration purposes only. We did not have the intention to develop a fully functional framework, but just a means to experiment with the power of frameworks in our problem domain.

7.1 Implementation

This section presents the infrastructure MyFramework provides to an application built on top of it. We start with the discussion of the data contract. Then we continue with object-relational mapping, for which we use the NHibernate framework. Finally we discuss some infrastructural components, like authorization, logging, and exception management.

7.1.1 Data Contract

As we discussed in Section 2.4, service-oriented applications use a data contract to define messages which can be used to request information from a service. We illustrate the idea behind these messages in Listing 7.1.

Let us take a look at the message in Listing 7.1, which is an example of a message in the traditional approach. The `msgVehicle` message is an answer to a request from another service. In Section 2.4 we have discussed the service- and datacontract. The functionality a service offers is defined in a servicecontract. The messages which are used to communicate with a service are defined in the datacontract. Think of messages for modifying, deleting, or searching certain entities. We do not discuss any details of these messages at this point, but we come back on this later on.

```
public class msgVehicle
{
    private VehicleCollection vehicles;

    public VehicleCollection Vehicles
    {
        get {   return vehicles; }
        set {   vehicles = value; }
    }

    public msgVehicle()
    {
        vehicles = new VehicleCollection();
    }
}
```

Listing 7.1: Message Example (C#)

To make more optimal use of the power of our underlying frameworks, we have exploited the general characteristics of the messages in a data contract, and have defined a generic message format. The generic message format is incorporated in MyFramework and is illustrated in Figure 7.1.

In Listing 7.2 the code of `msgBusinessObject` is displayed. We used a `List` instead of an `IList`. Although using an `IList` is better practice, according to the software design principle of separating interface and implementation [Fow04b], due to a bug in the Microsoft XML serializer an `IList` cannot be serialized [xai07]. Therefore we chose to use a `List`.

The `msgBusinessObject` is a generic message, which is designed to contain a `List` of `IBusinessObject` instances. This solves the problem of implementing a specific message for each business object. As we discussed earlier, the `msgBusinessObject` is actually only a message to answer to a request of another service. We also discuss an example of a request message, as presented in Listing 7.4.

Listing 7.3 shows the implementation of the `MessageTypes` class. It contains an enumerator for `SaveMessageType`, which can be used to indicate whether an insert or an update operation should be performed on the business objects in list of the message. The `msgSaveBusinessObject` of Listing 7.4 simply inherits from `msgBusinessObject` and adds a `SaveMessageType` to the message. The other messages, `msgDeleteBusinessObject` and `msgSearchBusinessObject` are designed in a similar way. By designing the messages in this way, we created a generally applicable data contract for SOA applications. This design already saves an application developer some time in writing code manually, as illustrated in Figure 7.2. As we mentioned earlier, this design also makes it possible to use the underlying frameworks more effectively. We illustrate this in the next paragraph.

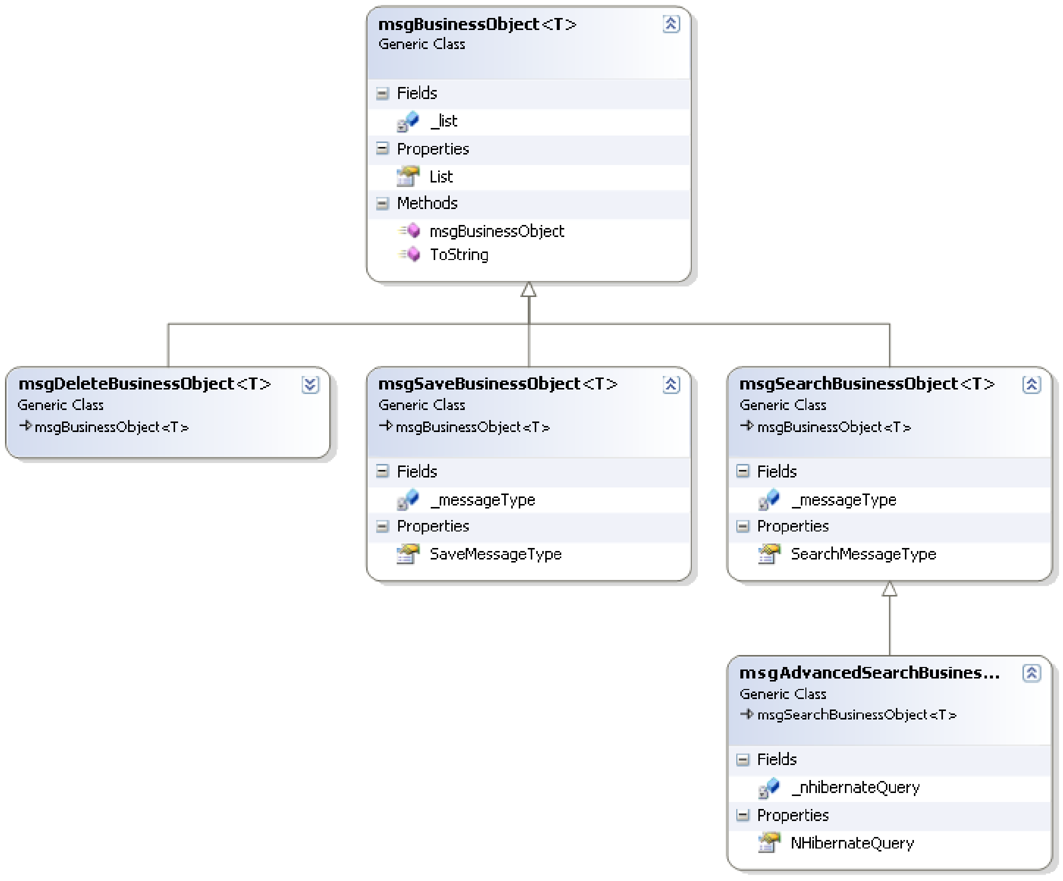


Figure 7.1: MyFramework DataContract Class Diagram

7.1.2 Object Relational Mapping

In this section we discuss how we incorporated object relational mapping, by means of the NHibernate framework, into our solution. We started by defining a generic interface supporting the basic CRUD— create-retrieve-update-delete—operations, as illustrated in Listing 7.5. A class implementing this interface is able to perform the defined operations on business objects. We implemented this interface for NHibernate, and named it `NHibernateBusinessObjectDAO`. We do not discuss an example of the actual retrieval of a `BusinessObject` from the database, because we need application specific NHibernate mapping files in which we define how database entities map to business objects. Therefore we discuss `BusinessObject` retrieval and the associated mapping file in Chapter 8.

Problem with Detached Objects

With the `NHibernateBusinessObjectDAO` we can retrieve and store business objects from and in the database. Retrieving the objects did not cause any problems. We simply serialize

```
public class msgBusinessObject<T> where T : IBusinessObject
{
    private List<T> _list;

    public List<T> List
    {
        get { return _list; }
        set { _list = value; }
    }

    public msgBusinessObject()
    {
        _list = new List<T>();
    }
}
```

Listing 7.2: MyFramework msgBusinessObject Class (C#)

these to their associated XML message format, and send them to another service. In this service the message is deserialized and the business object is recreated. The service can modify the business object, and send an update message to our business service. The business service in turn deserializes the message, and recreates the modified business object. When we perform an update operation with the recreated business object, NHibernate complains that the business object is unsaved or transient, in other words, NHibernate thinks the business object is not yet present in the database or is not meant to be saved.

Why does this problem occur? To understand this, we first discuss some details about how NHibernate deals with business objects. When we retrieve a business object from the database, NHibernate associates this object with a session [nhr07]. When we modify this object and then perform an update operation, NHibernate simply performs an update as specified. Unfortunately in our solution we do not perform an update operation on the business object associated with the session, but on a recreated business object. In this case [nhr07] proposes to reassociate the recreated business object with the session or using an update method which updates unassociated objects, but both approaches do not perform the desired functionality in our solution. In order to tackle this problem, we decided to create a simple but effective solution; a data mapper. Before we perform an update operation on a business object, we first read the corresponding business object from the database. Then we copy all properties from our recreated business object to the business object which is still associated with the NHibernate session. When this operation is finished, we let NHibernate update the business object. This solves the problem.

Advanced Search Criteria

By defining generic messages, as we discussed in Section 7.1.1, we provide other services the functionality to perform save, delete, and search operations on the business objects of

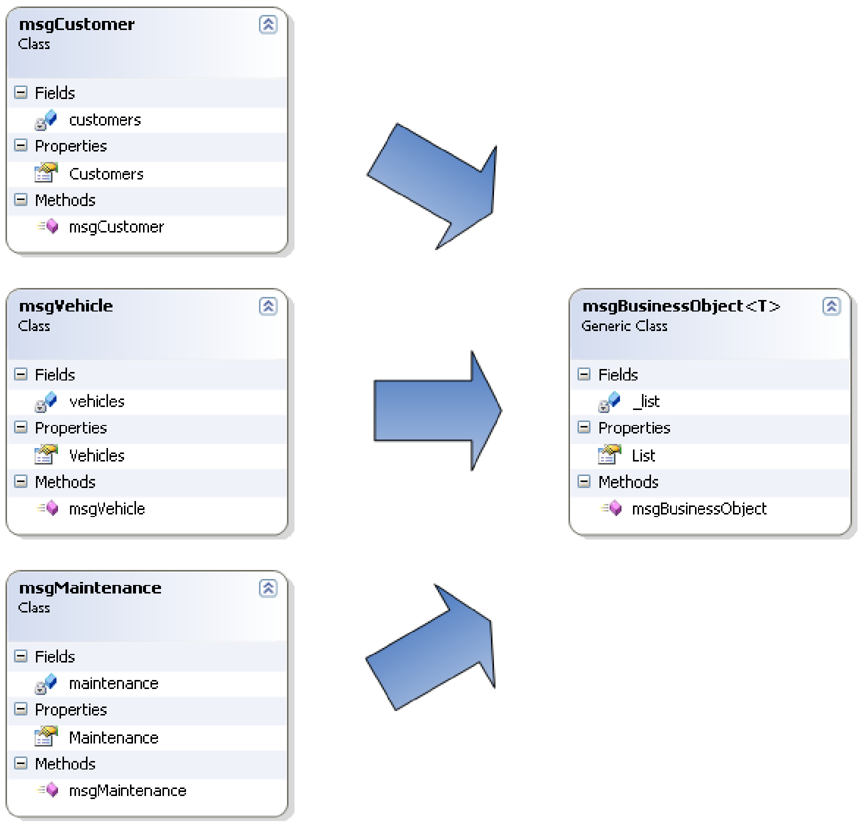


Figure 7.2: From Multiple Messages to One Generic Message

our business service. The `msgSearchBusinessObject` we implemented only provides the possibility to find a business object with a certain identifier or find all business objects of some type. Therefore the functionality of this implementation is very limited. To extend these possibilities, we implemented a generic message providing the functionality to add additional search criteria.

These additional search criteria must be specified in the Hibernate Query Language (HQL). HQL is a fully object-oriented query language, understanding notions like inheritance, polymorphism, and association [nhr07]. HQL can be used to query all business entities which have a NHibernate mapping file, by transforming a HQL query into an ordinary SQL query [hql07]. We discuss a concrete example of a HQL query in the Chapter 8.

7.1.3 Infrastructural Components

We equipped MyFramework with components for logging, authorization, and exception management which is illustrated in Figure 7.3. These components are designed and implemented as far as necessary. In our case we just want to use these components to demonstrate

```
    public class MessageTypes
    {
        public enum SaveMessageType
        {
            Insert,
            Update
        }

        ...
    }
```

Listing 7.3: MyFramework MessageTypes Class (C#)

```
public class msgSaveBusinessObject<T> : msgBusinessObject<T>
    where T : BusinessObject
{
    private MessageTypes.SaveMessageType _messageType;

    public MessageTypes.SaveMessageType SaveMessageType
    {
        get { return _messageType; }
        set { _messageType = value; }
    }
}
```

Listing 7.4: MyFramework msgSaveBusinessObject Class (C#)

dependency injection and aspect-oriented programming in Chapter 8.

Because the implemented functionality is for demonstration purposes only, the logging component simply writes time stamped information to a text file, which is sufficient to see whether information is actually logged. The authorization component does not perform any checks on whether someone is authorized or not, because authorization itself lies outside the scope of this project. The authorization component simply calls the logging component with a standard message to indicate when it is called. For the exception management component we chose a similar approach. When an exception is thrown, its message is simply forwarded to the logging component.

To use these infrastructural components with Spring.NET's AOP engine, we implemented a number of Spring.NET advices. Listing 7.6 shows the implementation of our authorization advice. This advice should be invoked before a method is executed. Line 6 of Listing 7.6 shows the retrieval of an instance of the authorization component. Which authorization component this is, and how this is configured is application specific, so we discuss this in Chapter 8. When a user is not authorized, an exception is thrown. In the other case, when a user is authorized, we simply use our logging component to log some relevant information. For our logging and exception management component we also implemented

```
public interface IBusinessObjectDAO<T> where T : BusinessObject
{
    // Find a BusinessObject by means of its identifier.
    T FindById(int id);

    // Find all BusinessObject entities of a specific type.
    IList<T> FindAll();

    // Find all BusinessObject entities of a specific type,
    // matching the query string.
    IList<T> FindAll(string query);

    // Insert a BusinessObject.
    T Save(T t);

    // Update a BusinessObject.
    T Update(T t);

    // Delete a BusinessObject.
    void Delete(T t);
}
```

Listing 7.5: MyFramework IBusinessObjectDAO Interface (C#)

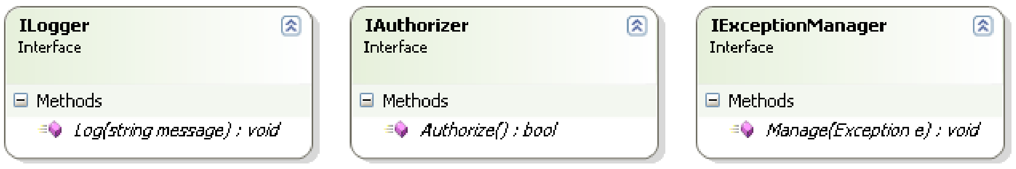


Figure 7.3: MyFramework Infrastructural Components

similar advices. In Chapter 8 we discuss the actual usage of the advice.

7.2 Limitations

In this section we discuss some limitations of the implementation of MyFramework. The first limitation has to do with our solution for the problem of detached objects. As a result of the data mapper, we cannot use the functionality of the NHibernate framework to resolve concurrency issues. To solve this problem, we have to adapt or rewrite the concurrency functionality of the NHibernate framework, to fit into our solution. We did not spent any time on implementing this solution since we only build a prototype, and concurrency mechanisms lie outside the scope of our solution.

Another limitation is the result of using HQL. We have constructed our generic messages in such a way, that the front-end is responsible for creating an HQL query. This

```
public class AuthorizationBeforeAdvice :
    Spring.Aop.IMethodBeforeAdvice
{
    public void Before( ... )
    {
        if (!((IAuthorizer)ContextRegistry.
            GetContext()["myAuthorizer"]).Authorize())
        {
            throw new ApplicationException( ... );
        }

        // Log that the user is authorized.
    }
}
```

Listing 7.6: MyFramework AuthorizationBeforeAdvice (C#)

results in a form of technology coupling between the front- and back-end. We think it is a better approach that the front-end is not aware of any implementation specific details, in this case the use of the NHibernate framework. Therefore we propose a more technology independent language like LINQ. Using LINQ in MyFramework would require an LINQ to HQL transformation. Building such a component does not fit in the scope of this project.

A SOA application works with a service- and datacontract, as mentioned earlier. These contracts define the exposed functionality of a service. The datacontract defines the incoming and outgoing messages. MyFramework offers the possibility to execute advanced search queries, by means of HQL. The datacontract however introduces limitations on the number of possible queries. We cannot simply execute any possible query, because the number of answer messages is limited. A result of a query must fit in the defined message. This limitation can be assigned to the design principles of a SOA. Therefore we will come back on this limitation during the validation phase, because providing advanced search queries might strike at the roots of SOA.

Chapter 8

Case Study: Garage Case

In this chapter we discuss our case study. We implemented part of the Garage Case designed by Info Support's PDC on top of MyFramework. The Garage Case is designed for demonstration purposes of the latest Endeavour technologies. With the Garage Case the PDC can validate their design and show best practices to other developers. Therefore we decided to use part of the Garage Case, with a size of approximately 50 function points, in our case study. The chosen part of the Garage Case fits in our problem domain, since it is a small and simple service-oriented application.

The general idea behind the the Garage Case is that it is used by a garage, to store and access information about customers, their vehicles, and the maintenance which is performed on these vehicles.

In Section 8.1 we discuss the implementation of the Garage Case on top of MyFramework. We do not discuss every single detail of our implementation, but decided to take a look at the most interesting parts. Section 8.2 describes some experiments with our solution to get a better impression about its quality. In Section 8.3 we evaluate the results of our case study in the light of the solution criteria presented in Chapter 6. Finally, in Section 8.4 we draw conclusions.

8.1 Implementation

In this section we discuss the parts an application developer has to implement in order to end up with a working application. In this context an application is considered working when it can perform the basic CRUD operations on the business entities in the database. We make use of MyFramework from Chapter 7 to implement our prototype. In the next paragraphs we discuss the parts which are required for a working application, namely a database, the business objects, the NHibernate mapping files, a service contract, and the front-end.

8.1.1 Database

For our prototype we used a Microsoft SQL Server database. The database model we used is displayed in Figure 8.1. The names of our tables are actually in Dutch, but for convenience we translated them to English in this model. The database contains nine tables, which consist out of five to thirteen columns. Because we use NHibernate, it is not necessary to work with stored procedures.

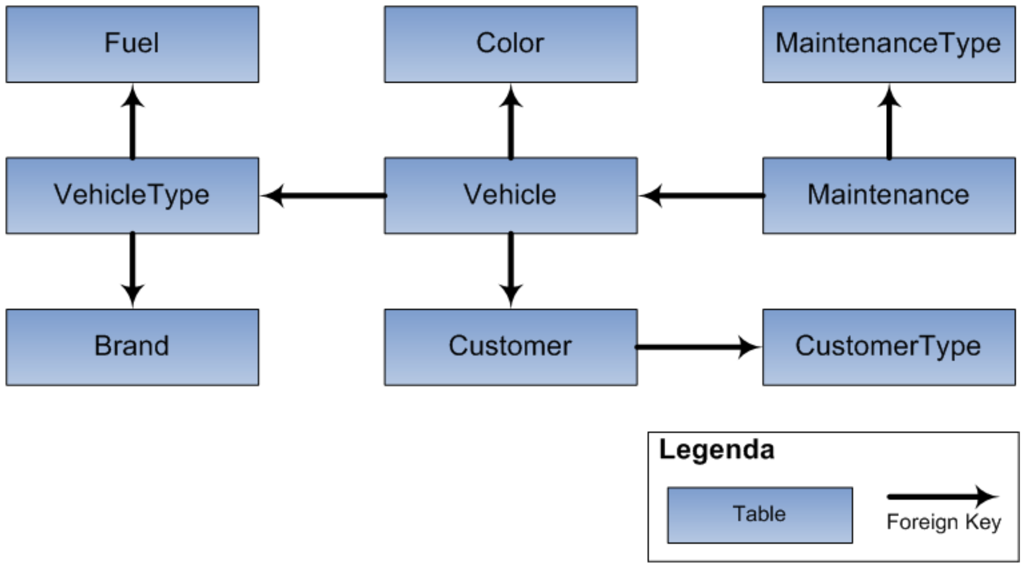


Figure 8.1: Garage Case Database Model

8.1.2 Business Objects

After creating the database tables, we implemented the business objects. Listing 8.1 shows the implementation of our `Vehicle` business object. All business objects must extend the MyFramework `BusinessObject` class in order to be recognized. The `Vehicle` implementation is quite straightforward; it contains a property for each attribute that NHibernate has to map. The NHibernate framework requires that the properties are declared virtual, in order to create proxies of the business objects [nhr07]. To fill the business objects with information from our database, we write NHibernate mapping files, which we discuss in the next paragraph.

8.1.3 NHibernate: Mapping Files

For each business object we define a mapping file. Listing 8.2 shows the NHibernate mapping file for the `Vehicle` business object from Listing 8.1. We briefly discuss the contents of the mapping file. The third line defines the mapped class, in this case `Vehicle`, and the assembly in which this class is located. For each property of our business object we define a mapping, by indicating the name of the property and the name of the database column. When the property is used to reference an object, NHibernate simply uses the corresponding mapping file of the object to fill it. An example of this is displayed on line 21 of Listing 8.2 as a `many-to-one` relation.

```
public class Vehicle : MyFramework.BOL.BusinessObject
{
    private string _name;
    private int _mileage;
    private string _licenseNumber;
    ...

    public virtual string Name { ... }
    public virtual int Mileage { ... }
    public virtual string LicenseNumber { ... }
    ...
}
```

Listing 8.1: Garage Case Vehicle Class (C#)

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
    <class name="GarageManagementSystem.BOL.Vehicle,
            GarageManagementSystem.BOL" table="Vehicle">
        <id name="Id" unsaved-value="-1">
            <column name="VehicleId"/>
            <generator class="native"/>
        </id>
        <timestamp name="LastModificationDateTime"
            column="LastModificationDateTime"/>
        <property name="Name">
            <column name="Name" length="50" not-null="true"/>
        </property>
        <property name="Mileage">
            <column name="Mileage" not-null="true"/>
        </property>
        <property name="LicenseNumber">
            <column name="LicenseNumber" not-null="true"/>
        </property>
        ...
        <many-to-one name="Color" column="ColorId" lazy="false"/>
        <many-to-one name="VehicleType" column="VehicleTypeId"
            lazy="false"/>
        ...
    </class>
</hibernate-mapping>
```

Listing 8.2: Vehicle NHibernate Mapping File

8.1.4 Service Contract

Listing 8.3 shows the interface of the service contract. As discussed in Section 2.4 a service contract defines the functionality which is exposed to other services. Implementing the service contract is merely forwarding a message to the responsible business object controller, and returning the response message.

When we look at the service contract of Listing 8.3, one might notice that we use explicit types—like `Vehicle`, `Customer`, `Maintenance`, etc.—instead of the generic type `T`. Although it would seem a better idea to use the generic type, we did not have a choice. As we discussed in Section 2.4, all messages are serialized to XML. T is not serializable by default. A solution to this problem would be to make all business objects serializable. Although this is a solution for the problem, we did not use it, because there also is another reason for our choice of using explicit types. This reason can be brought back to the design principles of a SOA application, which enforce to write a service contract which explicitly defines the exposed functionality, and does not expose more functionality than required. By using explicit types we can restrict the functionality of a service to a desired level, in contrast with the use of T, where we provide to much freedom.

```
public interface IBSGarageManagementSystem_v1
{
    // Consult business objects.
    msgBusinessObject<Vehicle> ConsultVehicle
        (msgSearchBusinessObject<Vehicle> message);
    msgBusinessObject<VehicleType> ConsultVehicleType
        (msgSearchBusinessObject<VehicleType> message);
    ...

    // Consult reference data.
    msgBusinessObject<Fuel> ConsultFuelReferenceData();
    msgBusinessObject<Color> ConsultColorReferenceData();
    ...

    // Advanced consultation of data.
    msgBusinessObject<Fuel> AdvancedConsultFuel
        (msgAdvancedSearchBusinessObject<Fuel> message);
    ...

    // Save business objects.
    void SaveVehicle
        (msgSaveBusinessObject<Vehicle> message);
    void SaveVehicleType
        (msgSaveBusinessObject<VehicleType> message);
    ...
}
```

Listing 8.3: Garage Case Service Contract (C#)

8.1.5 Front-end

The last part we need is the front-end. As already discussed in Chapter 5 we do not use frameworks to target the front-end. For this reason it does not have any additional value for this project to develop a front-end ourselves. Therefore we used a front-end of the Garage Case which was designed by Maarten Schilt. More about his work can be found in Section 1.4.4. By making some small changes to his front-end, we were able to use it with our back-end.

8.2 Experimenting with the Solution

As we have seen in the previous section, we only have to create a database, the business objects, the NHibernate mapping files, the service contract, and a front-end to end up with a working application. These parts are relatively easy to create; they do not contain complex structures which have to be designed and developed by an application developer. In this section we want to go a step further. We perform some experiments with our solution by adding additional functionality. These experiments help us to get a better impression of the quality of our solution in the light of the criteria discussed in Chapter 6. We discuss our impression in Section 8.3.

8.2.1 Applying AOP

So far we did not concern ourselves with any infrastructural arrangements, like authorization, logging, or exception management. In this paragraph we discuss how we take care of these infrastructural arrangements, by means of the AOP capabilities of the Spring.NET framework. Listing 8.4 shows how we configure the aspects by specifying an advice and a pointcut. We briefly discuss the `logging` aspect. On Line 11 is specified that the advice is an instance of `LoggingAroundAdvice` which is located in MyFramework. We discussed this advice in Section 7.1.3. To define where the advice is applied, we make use of the Spring `NameMatchMethodPointcutAdvisor` as indicated on Line 9. In the `MappedNames` property we define a list of regular expressions of method names to be advised. All methods starting with "Consult" or "Save" will be advised. The authorization and exceptions aspects look very similar, so we do not discuss these.

We configured our aspects, but we still have to indicate which object we want to target with these aspects. Therefore we configure `BSGarageManagementSystemController` as target, as indicated on Line 35. In the property `InterceptorNames` we define that all previously configured aspects—`logging`, `authorization`, and `exceptions`— have to be used.

8.2.2 Using HQL

In order to perform more advanced searches, we make use of the Hibernate Query Language (HQL), as discussed in Chapter 7. In this Section we discuss an example of the use of HQL in our prototype. Listing 8.5 shows a concrete example of a query for retrieving all possible

```xml
<?xml version="1.0" encoding="utf-8" ?>
...

<spring>
    ...
    <objects xmlns="http://www.springframework.net">
        <!--Logging around aspect.-->
        <object id="logging" type="Spring.Aop.Support.
            NameMatchMethodPointcutAdvisor, Spring.Aop">
            <property name="Advice">
                <object type="... .LoggingAroundAdvice, ..."/>
            </property>
            <property name="MappedNames">
                <list>
                    <value>Consult*</value>
                    <value>Save*</value>
                </list>
            </property>
        </object>

        <!--Authorization before aspect.-->
        <object id="authorization" type="...">
            <!--Similar to configuration of logging.-->
        </object>

        <!--Exception management aspect.-->
        <object id="exceptions" type="MyFramework.Exceptions.
            Advice.ExceptionThrowsAdvice, MyFramework.Exceptions.
            Advice"/>

        <!--Applying a set of aspects to a certain instance.-->
        <object id="bsGarageManagementSystemController"
            type="Spring.Aop.Framework.ProxyFactoryObject">
            <property name="Target">
                <object type="GarageManagementSystem. ...
                    BSGarageManagementSystemController, ..."/>
            </property>
            <property name="InterceptorNames">
                <list>
                    <value>authorization</value>
                    <value>logging</value>
                    <value>exceptions</value>
                </list>
            </property>
        </object>
    </objects>
</spring>
```

Listing 8.4: Configuring Spring.NET AOP

fuel types for a "Corsa" vehicle. For a more in-depth explanation about the HQL language, we refer to [nhr07]. We create a `msgAdvancedSearchBusinessObject<Fuel>`, which contains the HQL query. We then send the request to our service, which responds with a `msgBusinessObject<Fuel>` with a list containing the results.

```
IBSGarageManagementSystem_v1 service = ... ;
msgAdvancedSearchBusinessObject<Fuel> message =
    new msgAdvancedSearchBusinessObject<Fuel>();
message.NHibernateQuery = "select distinct fuel from Fuel as " +
    "fuel, VehicleType type where type.Name = 'Corsa' and " +
    "fuel.Id = type.Fuel.Id";
msgBusinessObject<Fuel> result =
    service.AdvancedConsult<Fuel>(message);
```

Listing 8.5: Advanced Search with HQL (C#)

8.2.3 Expanding our Controller

MyFramework contains a controller which is responsible for reading incoming messages and forwarding them to the responsible manager and returning a message containing the results. The functionality of the default controller is limited. In this section we discuss the effects of the scenario in which additional functionality is required from the controller.

We start by sketching the idea behind the additional functionality. In some applications data is divided into ordinary data, and reference data. The ordinary data is frequently modified by the user of an application. With reference data we refer to data which is more or less static—it does not change frequently—, like a list of colors, or a list of fuel types. This type of data is usually sent in one message from the back-end to the front-end, where it could be cached. The front-end can for example retrieve all fuel types, by creating a `msgSearchBusinessObject<Fuel>` message and indicating that it requires all possibilities. However, creating such a message for reference data is not necessary, since we always require a complete collection of available reference data of a certain type. Therefore we create a special message to retrieve reference data of some type. This requires the modification of the default controller of MyFramework.

Listing 8.6 shows the required modification. We simply extend the interface of the default controller of MyFramework, by adding the required functionality. A function for consulting reference data, which does not require an input message. The final step is to implement the interface, which results in a newly created controller which satisfies the additional requirement.

8.2.4 Using Dependency Injection

As we discussed in Section 8.2.1 we make use of infrastructural components of MyFramework. To make use of these components, we use AOP and the advice defined in MyFramework. To prevent tight coupling we do not explicitly define which component we use, but

```
public interface IBSGarageManagementSystemController :
    IMyBusinessObjectController
{
    msgBusinessObject<T> ConsultReferenceData<T>()
        where T : BusinessObject;
}
```

Listing 8.6: Expanding the Default MyFramework Controller (C#)

use the dependency injection engine of the Spring.NET framework. This approach results in the possibility to configure for example which logging component we want to use, as illustrated on Line 7 of Listing 8.7. Every logging component, which confirms to the `ILogging` interface of Figure 7.3 can be injected into our advice. Configuring the logging component itself, for example specifying a log file path, can also be performed with the configuration file of Listing 8.7, as shown on Line 9.

```
 1 <?xml version="1.0" encoding="utf-8" ?>
 2 ...
 3
 4 <spring>
 5     ...
 6     <objects xmlns="http://www.springframework.net">
 7       <object id="myLogger" type="MyFramework.Logging.MyLogger,
 8         MyFramework.Logging">
 9         <property name="LogFilePath"
10             value="C:\\temp\\LogFile.txt"/>
11       </object>
12     </objects>
13 </spring>
```

Listing 8.7: Configuring Spring.NET Dependency Injection

8.2.5 Modifying the Database

In this section we discuss two scenarios of modifying the database. First we discuss what the effects of adding an additional column to an existing table are. Secondly we discuss the effects of adding a new table to the database. In both cases we describe the required work to end up with a working application after modifying the database. However, we only discuss the effects on the back-end of the application, since any modifications to the front-end lie outside the scope of this project.

Adding an Additional Column to an Existing Table

We start by adding an additional column to the database in Microsoft SQL Server, for example adding a `Chassisnumber` to the `Vehicle` table. To make use of this additional information, we have to modify the corresponding business object, like the one discussed in Section 8.1.2, by adding an attribute and a property for the `Chassisnumber`. The next step is to alter the mapping file, by adding a mapping from the database to the business object `Chassisnumber`.

Adding a New Table

Adding a new table to the database is slightly more complex than the previous scenario we discussed. In this case we add a table to the database, add an additional business object, and create a corresponding mapping file. When the newly created business object is referenced by other business objects, we have to modify these other business objects as well, in a similar way as described in the previous paragraph.

8.2.6 Alter the Object Model

In this section we discuss the modifications required when we decide to change our object model. In this scenario we do not modify the database, this enables us to discuss the effects of a paradigm mismatch as discussed in Section 4.2.3. We alter the object model in the way indicated in Figure 8.2. In the original `Customer` class we incorporated the contact information, but we decided to move this to a separate class, namely `ContactInformation`. This decision results in an object model which does not match the database anymore, in other words, a paradigm mismatch.

To solve this problem we modify the mapping file as illustrated in Listing 8.8. In the original mapping file the `ContactInformation` component did not exist. This modification solves the paradigm mismatch.

8.3 Evaluation

The case study we just discussed gives us a first impression of the quality of our solution to develop small and simple Endeavour applications. In Chapter 6 we identified a number of criteria to validate our solution. In this section we discuss the solution according to these criteria.

8.3.1 Productivity

By building a small and simple Endeaovur application on top of MyFramework, we only have to create the database, the business objects, the mapping files, the service contract, and the front-end, which are sufficient to end up with a working application. This suggests quite an increase in productivity, but let us take a closer look before drawing any overhasty conclusions.

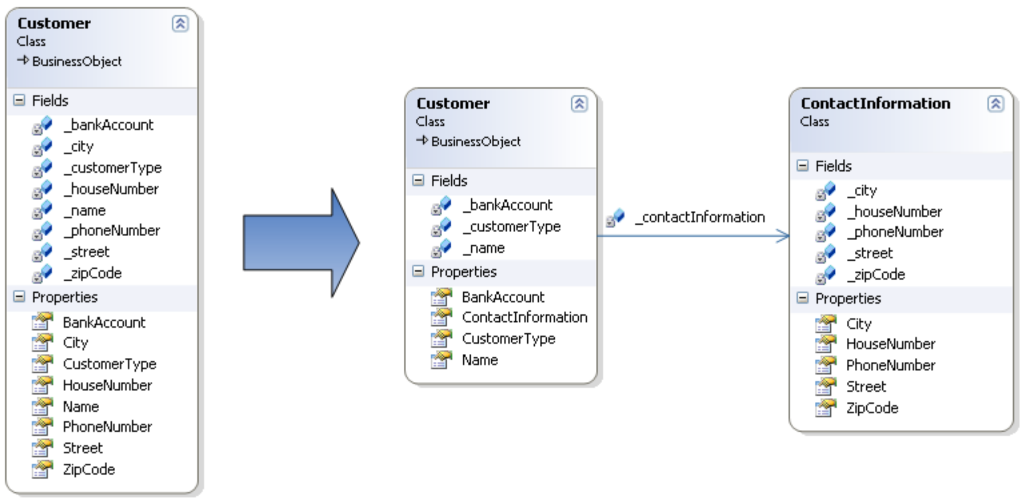


Figure 8.2: Modifying the Object Model

In the original approach a developer had to write approximately 3000 manual source lines of code to implement the part of the back-end of the Garage Case we also developed. For our implementation of the Garage Case on top of MyFramework we wrote approximately 750 source lines of code. This amount consists of the C# code, as well as the XML code in the various configuration files. We do not take the front-end into consideration, since we did not apply any frameworks in that application part.

As we mentioned in Section 6.1 a reduction of the manual lines of code does not automatically imply an increase in productivity. Therefore we also take the time required to develop the solution into account. The time required to develop a solution is strongly related to the complexity of the underlying technology. There are no indications that we increased the complexity of the code, because most parts an application developer has to write look very similar to the original ones. However, we do think that productivity on the short term is negatively affected by the learning curve due to the application of new technologies, namely the Spring.NET and NHibernate framework. Taking all of this into consideration, we conclude that our solution provides a significant increase in productivity *in the test case we described in this chapter*.

8.3.2 Maintainability

In this section we take a closer look at our solution from a maintainability perspective, by using the case study and the experiments described earlier in this chapter.

Move Responsibility to Frameworks

The Garage Case is built on top of MyFramework, which makes use of the NHibernate and Spring.NET framework. This way we move some responsibility of the application to frameworks, which provide standardized functionality. The functionality offered by the

```xml
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
  <class name="GarageManagementSystem.BOL.Customer,
    GarageManagementSystem.BOL" table="Customer">
    <id name="Id" unsaved-value="-1">
      <column name="CustomerId"/>
      <generator class="native"/>
    </id>
    <property name="Name">
      <column name="Name" length="120" not-null="true"/>
    </property>
    ...
    <property name="BankAccount">
      <column name="BankAccount" length="10" not-null="true"/>
    </property>
    <component name="ContactInformation"
        class="ContactInformation">
        <property name="Street">
          <column name="Street" length="120" not-null="true"/>
        </property>
        <property name="HouseNumber">
          <column name="HouseNumber" not-null="true"/>
        </property>
        ...
      </component>
    <many-to-one name="CustomerType" column="CustomerTypeId"
        lazy="false"/>
  </class>
</hibernate-mapping>
```

Listing 8.8: New Hibernate Mapping File for Customer

frameworks is not maintained by the application developer, which results in a positive effect on maintainability of an application.

Alter the Object Model

As we discussed in Section 8.2.6, changing the object model for whatever reason, does not automatically imply that the database also needs to be changed. This advantage is a result of the application of the NHibernate framework. So it is not necessary to have a database model which is very similar to our object model, a paradigm mismatch does not require an application developer to search for another solution. As a result, we can make use of a standardized solution, even in more complex scenarios.

Modifying the Database

In Section 8.2.5 we discussed the effects of modifying the database. Adding a column or a table results in adding or modifying the corresponding mapping files and business objects. Although these changes require manual effort, the effort is clearly reduced when we compare it to the traditional approach. The application developer does not have to change any code in the data access layer of the application.

Applying AOP

To configure the infrastructural arrangements, we make use of the AOP capabilities of the Spring.NET framework. This approach might have a positive effect on productivity, but we think the effect on maintainability is significantly larger. By using AOP we remove a lot of duplicate code from the source base of our application. We define infrastructural arrangements at a more appropriate level at a central place, which results in an improved insight of the application developer in the effects of such arrangements.

Configuration with XML Files

Configuring an application with XML files provides a central location for defining configurable properties. This certainly has a positive effect on the understandability of an application. These positive effects are however related to the size of a configuration file; larger configuration files tend to become less understandable.

Testability and Understandability

We did not perform any automated tests on our solution, therefore our insight in the testability of our solution is limited. In our solution we applied AOP. When AOP is applied in more complex scenarios, it is very hard to predict where the effects of certain advice will occur [Mao]. As a result, the understandability of an application declines. This is likely to cause testability problems. We do not foresee any other problems at this point.

8.3.3 Customizability

In this section we discuss our solution from the viewpoint of customizability. We discuss some experiments we presented earlier in this chapter.

Expanding Controller

When designing and developing MyFramework, we tried to provide the possibility for an application developer to expand the default provided functionality. To illustrate this, we expanded the MyFramework controller to support reference data, as discussed in Section 8.2.3. This is a quite straightforward job, which does not require the redevelopment of large parts of the framework. For other default parts which require customization we expect a similar result.

Dependency Injection

As discussed in Section 3.5.1, dependency injection is used to reduce dependency among components and thereby improving their reusability. By applying dependency injection, we can for example easily configure which logging component we want to use. This is a very powerful feature to perform customization in an application, by merely changing an XML configuration file.

8.3.4 Other Solution Criteria

The solution criteria we did not discuss so far are scalability, applicability in a SOA, and target small and simple applications. Although these criteria are relevant for our project, we do not evaluate these ourselves, but validate them in Chapter 9. We explain this decision for each criterion below.

Scalability Since we did not have time available to perform any experiments on expanding our prototype to an enterprise-size application, we do not give an opinion about the scalability of our solution at this point.

Applicability in a SOA As we mentioned in Chapter 6, we are not domain experts. Therefore it is very hard to form an opinion regarding this criterion. In Section 7.2 we identified some limitations of our approach which violate SOA design principles, namely application of HQL. However, this limitation is not caused by the applied framework, but by a lack of time prohibiting us from implementing a solution.

Target small and simple applications The same reason which holds for "Applicability in a SOA" also holds for this criterion. We are not domain experts, therefore we do not give an opinion regarding this criterion at this point.

8.4 Conclusions

In this section we draw the overall conclusions about the results of this case study. To give a more concrete insight of our impression of the quality of our solution for this case study, we decided to assign a score to each criterion, ranging from 1 (bad) to 5 (good). These scores are accompanied by a short explanation.

Productivity - Score: 4 The application of MyFramework results in a significant increase in productivity in this test case. We saw a reduction of the amount of manual lines of code in this test case. However, we have to keep in mind that the application of the Spring.NET and NHibernate framework causes application developers which first have to learn to work with these frameworks, before reaching an optimal increase in productivity.

Maintainability - Score: 3.5 By using functionality from frameworks, an application developer does not have to implement or maintain this functionality. Nor modifications of the object model nor of the database cause maintainability issues. The application of AOP and configuring the application with XML files also positively effects maintainability. However, we have to keep in mind that when AOP is applied in more complex scenarios, we should expect testability issues as discussed earlier.

Customizability - Score: 3 Expanding default functionality does not cause any problems, therefore we think we provide an acceptable level of customizability. Dependency injection also contributes to the customizability of the solution. We do think this criterion requires some additional attention, because the customizability of the solution determines for a large part to what extent it is applicable in a production environment. We get back on this topic in Chapter 9.

Scalability - Score: ? We do not provide a score for this criterion, since we have insufficient information to form an opinion about the criterion at this point.

Applicability in a SOA - Score: ? Not all functionality a framework provides is directly applicable in a SOA for domain related restrictions, as is the case with HQL. We did not encounter any more problems regarding this criterion, but since we are not domain experts, it is hard to form a good impression regarding this criterion.

Target small and simple applications - Score: ? In this test case we have applied our solution by building part of the Garage Case, a small and simple application. Although we did not encounter any problems, we do not give a score for this criterion at this point, but come back on this topic in Chapter 9.

Chapter 9

Validation: Experts' Opinion

In Chapter 8 we discussed our case study and evaluated the result. To perform an objective validation however, we require an external view on the project; we make use of the experts' opinion. We organized a meeting with four members of the PDC of Info Support. This meeting took about one-and-a-half hour. The goal of this meeting is for them to give their opinion about our solution in the light of the criteria as presented in Chapter 6. During the meeting we demonstrated the capabilities of our solution by giving a demonstration of MyFramework in combination with the Garage Case as presented in Chapter 7 and Chapter 8. During this demonstration the participants were free to ask questions about any specific detail or underlying idea. This resulted in an interesting discussion. Afterward we asked the participants to fill in the questionnaire of Appendix A. To check whether we interpret their answers correctly, we provided them with a copy of this chapter and incorporated their final remarks.

In this chapter we describe the outcome of the experts' opinion in Section 9.1. In Section 9.2 we draw conclusions. Finally, we present a retrospective on our validation approach in Section 9.3.

9.1 Experts' Opinion

In this Section we present the outcome of the experts' opinion by discussing the results of the questionnaire. The criteria of Chapter 6 formed the basis for the questionnaire, so we divide this section in six paragraphs, each discussing one criterion. During our evaluation in Section 8.4 we rated our solution for each criterion. In the questionnaire we also asked the participants to give such a score—ranging from 1 (bad) to 5 (good)—for each criterion. Table 9.1 gives an overview of the scores of the participants. In the next paragraphs we discuss the explanations for the scores awarded to each criterion.

9.1.1 Productivity

The participants concluded that our solution increases productivity in developing small and simple Endeavour applications. The traditional approach of programming an entire application manually would take more time than building an application on top of MyFrame-

Person / Criterion	A	B	C	D	Average
Productivity	3	4	4	4	3.75
Maintainability	3	3	5	4	3.75
Target small and simple applications	3	4	4	4	3.75
Scalability	3	3	3	3	3
Customizability	2	3	4	3	3
Applicability in a SOA	3	3	3	2	2.75

Table 9.1: Questionnaire Results: Score per criterion

work, which makes use of the Spring.NET and NHibernate framework. In the original approach an application developer had to write a lot of functionality, which is now captured in MyFramework. One of the participants described MyFramework as an out-of-the-box architecture. An application developer has to implement certain application specific parts— like a database, business objects, etc.— and the framework "wires" these together into a coherent application.

During the demonstration we also discussed the complexity of the solution, with respect to the original approach. The participants argued that the structure of our solution looks very similar to the original structure. Some parts have been moved to the framework, which could result in an application developer who does not understand everything in an application. This could be a drawback in the case of software testing for example. But moving parts to a framework also has an advantage; the application developer can develop an application at a somewhat higher level of abstraction. According to the participants it seems the solution reduces the complexity of the application code.

The provided functionality of a framework is however not always sufficient. Therefore it is important to have the possibility to change or expand the default provided functionality. Therefore the customizability also affects the productivity. In Section 9.1.5 we discuss this customizability criterion.

9.1.2 Maintainability

During maintenance it is quite likely that an application developer has to add, change, or remove business objects. With our solution this still requires some manual code, but the amount of manual code is significantly reduced when compared to the original approach. This certainly is a positive effect on maintainability. The application developer has to perform less changes, in less application layers.

The participants also liked the idea of AOP in combination with dependency injection to target infrastructural arrangements. Specifying aspects in configuration files provides an application developer with a central location to apply infrastructural arrangements—like logging, authorization, etc. By means of dependency injection the application developer can decide which component should be used for a certain task. This is also specified in a configuration file. When an application developer wants to add or change any aspects or components, this merely requires the modification of a configuration file, instead of exploring the source code to find the right line of code to modify.

Although AOP can result in positive effects on maintainability, it also has a drawback. When AOP is applied in more complex scenarios, it is likely to have negative effects on testability. It can be very hard to predict what advice affects which parts of the application.

As we discussed earlier, by applying MyFramework, we move responsibility from the application to the framework. This results in a positive effect of the maintainability of the application. However, the framework must be maintained as well. This is of course not the responsibility of the application developer, but of the framework developers. Although this might seem promising, the application really depends on the functionality of the framework. What happens when the framework is discontinued? What are the effects of exchanging the discontinued framework for a similar framework in application? These questions must be taken into account before making a decision to make use of a certain framework, because they certainly affect maintainability.

9.1.3 Target Small and Simple Applications

During our research we scoped our domain to small and simple Endeavour applications. Small and simple applications are mainly data-driven. This solution exploits that property, by providing predefined messages and a generic database layer, and is therefore very applicable on these kind of applications. We discuss the effects of this scoping on the scalability of our solution in Section 9.1.4.

9.1.4 Scalability

The scalability of our solution determines to what extent the solution is applicable to develop enterprise-sized business applications. Although we scoped our domain to target small and simple Endeavour applications with our solution, it is interesting to discuss the scalability limitations of our solution for possible future work.

The predefined generic messages in our solution are restricted to contain one collection of business objects of the same type. In enterprise-sized applications this causes problems, since it should be possible to construct messages which contain a combination of different types of business objects. This requires the modification of MyFramework. A participant proposed to use a configuration file specifically for messages. Such a file would specify which business objects a message contains. An interesting idea to look into.

In our solution we apply NHibernate to perform ORM. In a small and simple application like the Garage Case, this did not cause any performance related problems. But to what extent is NHibernate applicable in an enterprise-sized application? Before applying NHibernate in such an environment, we must first research the performance effects of NHibernate.

9.1.5 Customizability

As we discussed earlier, the customizability of the solution also affects the increase in productivity. When it is not possible to add or modify default provided functionality, the application of the framework might not have a positive effect on productivity. Although we encountered just a few limitations in the light of this criterion during our case study, we should not simply assume that we can build any small and simple application we like without complications.

9.1.6 Applicability in a SOA

When the participants determine the applicability of our solution in a general SOA environment, they tend to think of enterprise-size applications. Therefore the limitations in the light of this criterion are partly discussed in Section 9.1.4.

During the implementation of MyFramework, we already identified the limitation of applying HQL, as discussed in Section 7.2. We proposed to use a more general query language, like for example LINQ. During the demonstration this resulted in a discussion whether this would be a good solution. The participants claimed, that according to SOA design principles, services should be platform-independent. Therefore a service should not expose functionality which requires a query as input. They prefer an approach in which a collection of constraints per property indicate some search pattern. An application developer is free to choose to transform this search pattern into for example a HQL query inside the service.

9.2 Conclusions

The overall impression of the participants is that our solution is a good start. Of course, it is not ready for production work, but just a means to experiment with the power of frameworks in our problem domain, as we stated in Chapter 7.

Applying a framework provides structure and consistency to an application. This has positive effects on both productivity and maintainability. Decisions to capture certain parts of an application in generic parts in a framework did result in an increase in productivity. However, we should be careful with such decisions, because they also affect scalability, which was a result of exploiting characteristics of small and simple applications.

9.3 Retrospective

In the introduction of this chapter we discussed the validation method, namely the experts' opinion. In this section we discuss an alternative for the experts' opinion; the empirical study. We then discuss why we chose for the experts' opinion, although the empirical study is likely to give more interesting results.

9.3.1 Empirical Study

To perform a validation of our solution, we could have performed an empirical study. In this case we develop some part of an application—in or domain—according to the original approach. We also build part of an application with our solution. Then we compare the effort needed in both approaches. Good candidates for this comparison are some function points from the Garage Case. We form two teams, team A and B. Then we choose two function points from the Garage Case, FP1 and FP2. During the first session we ask team A to implement FP1 with the original approach, and team B implements FP1 with our solution. During the second session team A implements FP2 with our solution, and team B follows the original approach for FP2. Finally we analyze the results of this experiment and use them as the basis for the final validation.

9.3.2 Method Choice

We identified two reasons for using an experts' opinion in favor of an empirical study. First of all, we expected that an empirical study would take more time. We would have to prepare four assignments, hold two sessions of about four hours and evaluate the results. For the experts' opinion we could reuse the work we described in Chapter 7 and Chapter 8. The results of this approach are more concrete and take less time to evaluate. Due to a lack of time, we were forced to choose another validation method.

Secondly we were allowed to make use of more experienced participants in the case of an experts' opinion. For the experiment of the empirical study, we could use a number of students. For the experts' opinion we were allowed to use members of the PDC. However, since the PDC members only have limited time available, they could not participate in the experiment of our empirical study.

Therefore we decided to use the experts' opinion validation method. This is a limitation, because when we would have performed the empirical study with participants of questionnaire, or even better, a larger group of developers of Info Support, we could make a more thorough analysis of the effects of our solution. We think the experts' opinion limits the insights of the participants because it is a more controlled environment. During the experiment all kinds of interesting things could have happened, which could have resulted in the discovery of several limitations of our solution we have not yet discovered. Therefore we propose an empirical study as future work. We come back on this in Section 10.4.

Chapter 10

Conclusions and Future Work

10.1 Discussion

In this thesis we proposed a solution to improve productivity in developing small and simple Endeavour applications by applying frameworks. In this section we discuss the strengths and weaknesses of our solution. Then we discuss an alternative approach, namely Model Driven Development (MDD). We continue by comparing the framework and MDD approach, and finish with a hypothesis about the effects of a combination of both approaches. This can be considered a basis for future research.

10.1.1 Our Approach: Frameworks

To get an idea about the effects of our solution we performed a case study with the Garage Case as discussed in Chapter 8. This case study resulted in a reduction to around 25% of the original amount of manual lines of code. An impressive result which can be explained by the fact that we have moved a large part of functionality from the application to the framework. The validation showed that we did not increase the complexity of the application code. Most parts that an application developer still has to write manually, look very similar to the original approach. However, we have to keep in mind that we have applied new technologies, namely the Spring.NET and NHibernate framework, which results in a learning curve for the application developer.

When we look at our solution from a maintainability perspective, we see that the code base of an application is significantly smaller. Because we moved responsibility from the application to the framework, in some cases entire application layers, an application developer has to modify a smaller amount of code, in less application layers. However, this also has a drawback. As a result of the moved responsibility, an application developer does not have a total understanding of an application, this decreases his understanding of an application.

During experiments with the prototype, we have not encountered significant customizability limitations. We cannot simply assume those limitations do not exist. Therefore we propose an alternative validation method in the future work section, which might result in the discovery of limitations of our solution.

Our solution targets small and simple applications. Scaling the solution to support enterprise-sized SOA applications brought up some restrictions. An example of such a restriction is that the solution only supports messages containing a single business object, as we discussed in Section 9.1.4.

10.1.2 Alternative Approach: MDD

As we discussed in the introduction, Maarten performed a similar research, but in stead of frameworks he applied MDD. When we look at the productivity increase of this alternative approach, we see a reduction of the manual lines of code to nearly 1% of the original amount [Sch07]. In this approach an application developer does not write any C# code in the first stage of the project, but creates an XML configuration file from which the application is generated. In the case a developer wants to change the generated application, he can simply change the generated code.

The generated code is very similar to the original code. Therefore the manual maintenance in this approach is very similar to the original approach. An application developer must put himself through a large code base to perform maintenance. In some maintenance cases the solution provides the possibility to regenerate an application. This only requires an application developer to change the XML configuration file.

Closely related to maintainability is customizability, which is identified by [Sch07] as flexibility. The MDD solution has a number of customizability limitations. First of all the object model must be identical to the database model. Other issues are the result of the fact that not all manual code and generated code can be separated, so manually modified parts of an application could be lost on regeneration.

When we look at the scalability of this solution, we see similar restrictions as with our approach.

10.1.3 Comparison

In this paragraph we try to give a more concrete insight in the effects on both solutions. We discuss productivity and the maintainability and customizability of both approaches and compare them with each other and the original approach.

The validation of both approaches showed, that neither solution increased the complexity of the code. Therefore we *estimate*—we emphasize estimate, because this assumption is based on a simplification—the development time to be proportional to the amount of manual lines of code, which results in Figure 10.1. This graph indicates the time required, in other words, the number of manual lines of code required, to develop a working application—an application to perform the basic CRUD operations on entities in a database—for the original approach, the framework approach, and the MDD approach. Although this graph is based on the outcome of only one case study, we think that it indicates some interesting assumptions about a productivity increase of both solutions.

In Figure 10.2 we have made a sketch of a graph which indicates the time required for maintenance and customizability issues according to their complexity. With the original approach, every change requires manual modification of some code. The framework solu-

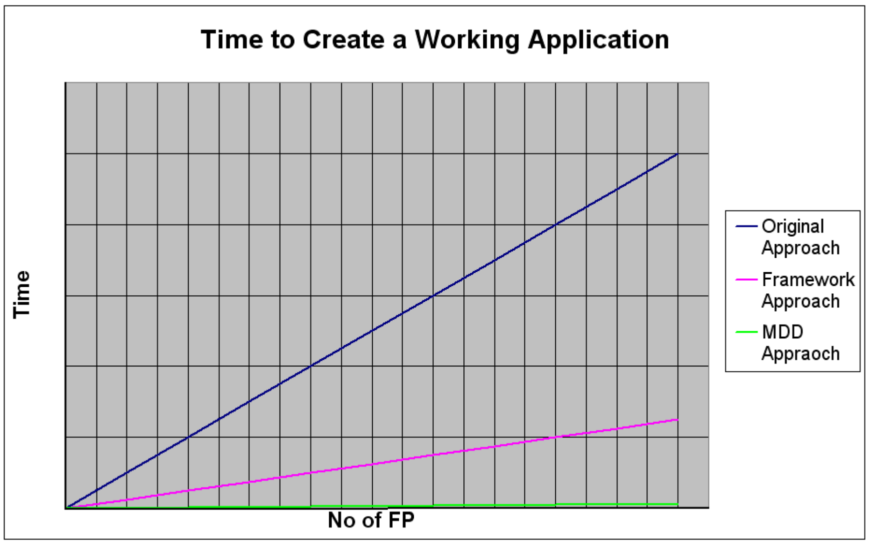


Figure 10.1: Graph Indicating Time to Create a Working Application

tion reduces the amount of code to be modified, but still requires a significant amount of manual work. The MDD approach provides an environment in which some modifications only require a few small changes to the configuration file. However, as the code base of an MDD generated application is almost identical to the original code base, more complex maintenance and customizability scenarios do not differ from the original approach. In this case the framework solution provides a better basis.

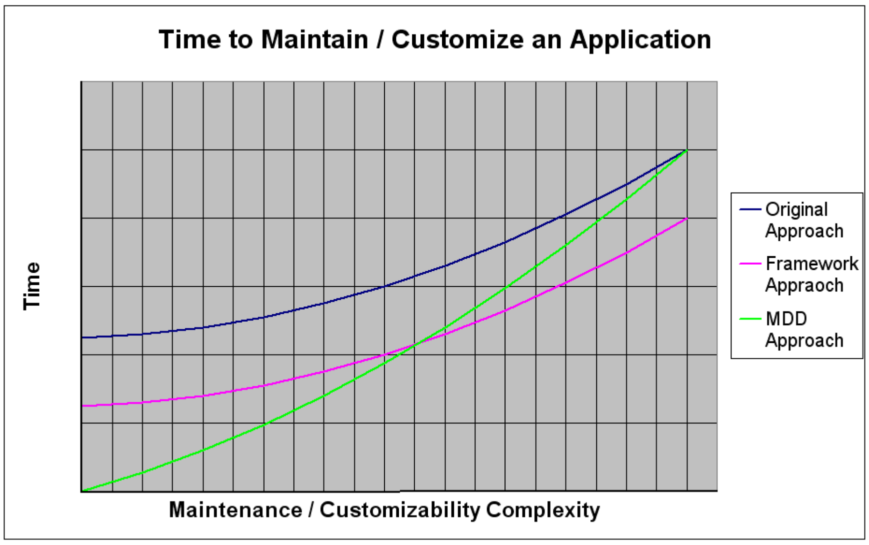


Figure 10.2: Graph Indicating Time to Maintain / Customize an Application

10.1.4 Combining Both Approaches

We have briefly discussed the effects of both solutions—MDD and framework—separately. We think a combination of both solutions results in a more powerful solution. Let us discuss why. Generation on top of a framework reduces the amount of generated lines of code. This has a big influence on productivity, because the manually developed part can now be generated. The maintainability of the resulting application will be higher, because we moved a large part of functionality to the framework. The same holds for a number of customizability issues which are a result of the MDD approach. By generating on top of a framework, we can overcome some of them. Therefore *we think a combination of both solutions results in one approach with the strengths of both.*

10.2 Conclusions

In this section we answer the central research question as given in the introduction:

> *How can we apply frameworks to increase productivity for small and simple Endeavour applications without losing the level of quality of the original approach?*

During this project we researched the applicability of frameworks to increase productivity for developing small and simple Endeavour applications. We chose two frameworks to target the back-end of an application, namely the NHibernate and Spring.NET framework. We have built MyFramework as an additional layer on top of the Spring.NET and NHibernate framework in order to improve the ease of use of these frameworks in our problem domain. MyFramework is a prototype and therefore should not be considered a fully functional framework. MyFramework contains a set of components which can be used as a basis for small and simple Endeavour applications.

In order to get a more concrete insight in the effects of applying MyFramework, we performed a case study. During this case study we built an application on top of MyFramework. This case study and the experts' opinion proved that our solution led to a *significant increase in productivity in our domain*. We significantly reduced the amount of manual lines of code and have no indications that we increased the overall complexity of the code which is still written manually. An application developer does not have to concern himself with low-level data access code, but simply uses the functionality which is provided by our solution. As a result an application developer works on a higher level of abstraction. This advantage, could also be a disadvantage, since the application developer does not have a full understanding of the entire application, which negatively affects the understandability of an application. Therefore we conclude that we did not only reduce the manual amount of code, but also the overall development time. However, we do think the productivity on the short term is negatively affected by the learning curve due to the application of new technologies, namely the Spring.NET and NHibernate framework.

The effects of our solution on the maintainability of an application are mainly positive. When an application requires maintenance, a smaller amount of lines of code needs to be changed, and the code which must be modified is located in less application layers. This is the result of moving responsibility—in some cases entire application layers—to the framework. To target infrastructural arrangements—like logging, authorization, etc.— we applied AOP in combination with dependency injection. These infrastructural arrangements can be applied by configuring them as advice in a central configuration file. Dependency injection enables an application developer to determine which component should be responsible for a specific task, by specifying this in a configuration file. Although AOP results in positive effects on maintainability, it also has a drawback: When AOP is applied in more complex scenarios, it is likely to have negative effects on testability and understandability, because it can be hard to determine what parts of an application are targeted by a pointcut.

The customizability of our solution also affects productivity. We performed some experiments during the case study. For small and simple Endeavour applications we did not encounter customizability limitations. However, we should not simply assume that we can build any small and simple Endeavour application without complications.

In our assignment we scoped our domain to small and simple Endeavour applications. These small and simple applications are mainly data driven. Our solution exploits that property, by providing some generic components. This scoping has some implications for the scalability of our solution. Simple applications have simple business logic. Broadening the scope requires future research to work on a solutions which is applicable in larger and more complex applications. When we want to scale our application to an enterprise-sized SOA application our current solution is not sufficient. For example, our messages now simply contain one type of business object. In an enterprise-sized SOA application the application developer wants to determine the contents of a message.

Another limitation can be brought back to the SOA environment. We made use of the Hibernate Query Language, which provides an object querying mechanism, to enable the front-end to specify search criteria. This worked fine, although we would prefer a more technology independent language. The application of HQL resulted in a violation of the platform independent design principle of a SOA application.

10.3 Contributions

In this section we discuss a number of contributions which are the result of our research for this graduation project.

A study of the effects of applying a framework to increase productivity We identified a set of frameworks to build a prototype to increase productivity for small and simple SOA application development. We used this solution to perform a case study to determine the effects of applying a framework in a practical situation. We also performed some experiments to get an impression about other quality criteria, like maintainability, customizability, and scalability. Finally we validated our solution by means of an experts' opinion with members of Info Support's PDC.

An overview of the underlying concepts of frameworks Numerous frameworks exist, but a lot of them use similar underlying concepts. This thesis gives an overview of the underlying concepts of frameworks, which form the basic elements of these powerful building structures.

Provided a basis for future research During the discussion in Section 10.1 we proposed to research a combination of frameworks and MDD to increase productivity in developing applications. This thesis and the thesis of Maarten [Sch07] can be used as a basis for future research in the effects of applying a combination of frameworks and model driven development to increase productivity.

10.4 Future work

In this section we discuss a number of possibilities for future work. During the discussion in Section 10.1 we formed an hypothesis about the combination of frameworks and MDD to one solution which might result in one solution containing the strengths of both separate solutions. We think this hypothesis is an interesting basis for future work.

Another interesting research area is to determine the effects of applying frameworks in enterprise-sized applications. This would result in more attention to the scalability criterion. In our current solution we use messages which can only contain one type of business object. This has negative effects on the scalability of a SOA applications, since enterprise-sized SOA applications are likely to contain more complex messages. A way to solve this problem might be to create an additional configuration file which determines which business objects a message contains.

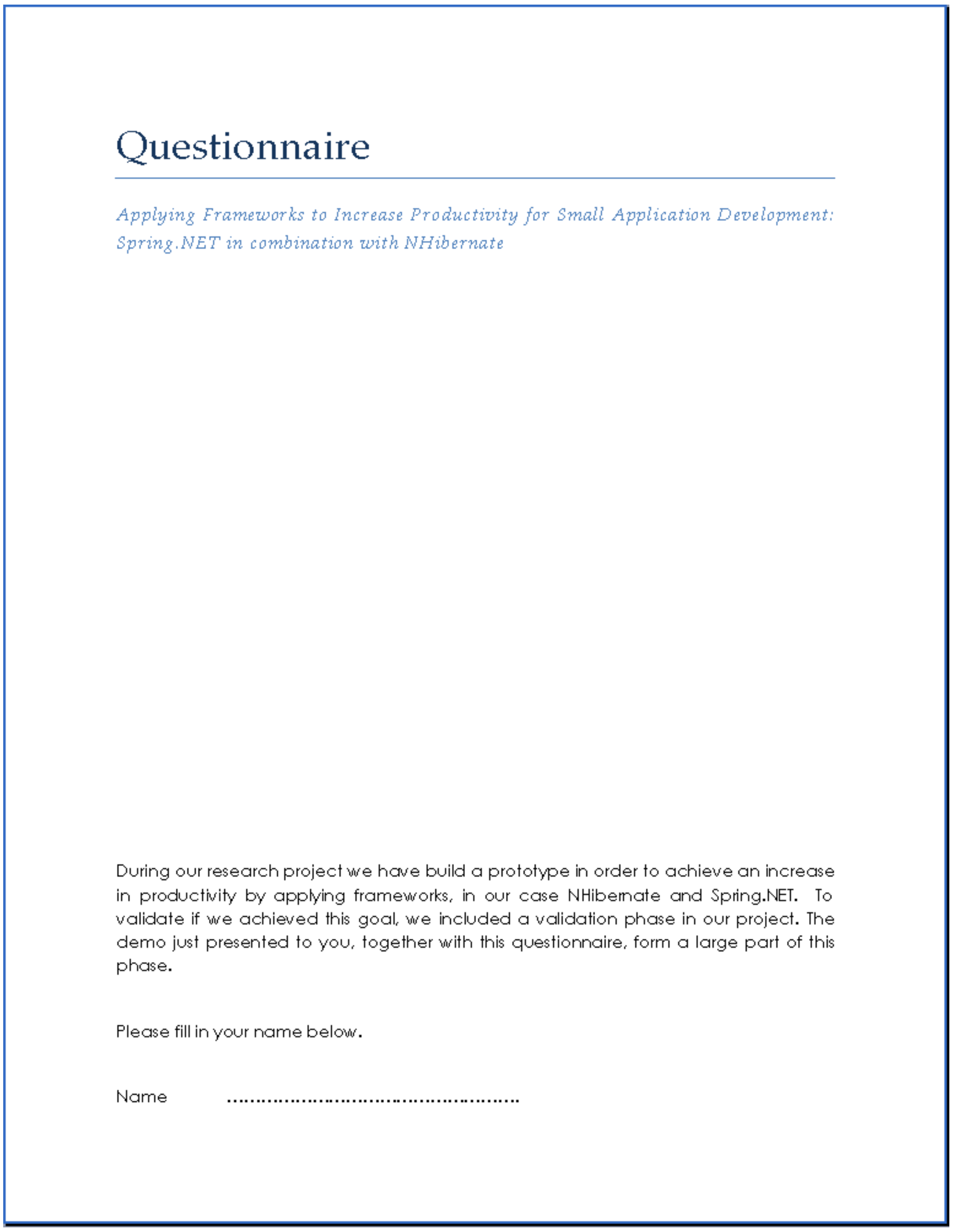
During this thesis we identified the Spring.NET and NHibernate framework as best candidates from our selection. For future research we propose a new selection round, preferably with more criteria and possibly some experiments with candidate frameworks. We think new frameworks could provide other, possibly better ways to solve the same problems.

In Section 9.3 we discussed another validation method, instead of the experts' opinion, namely an empirical study. We think an empirical study with an experiment in which a number of developers could really use our solution in a practical situation is likely to result in a better insight in the limitations and possibilities of our solution. This can be explained by the fact that during the experts' opinion we only presented our case study to a group of people, which results in a more controlled environment than in the case of an empirical study. However, due to time constraints we were unable to perform such an empirical study.

Appendix A

Questionnaire

This appendix presents the questionnaire we used during the validation phase of Chapter 9.



Questionnaire

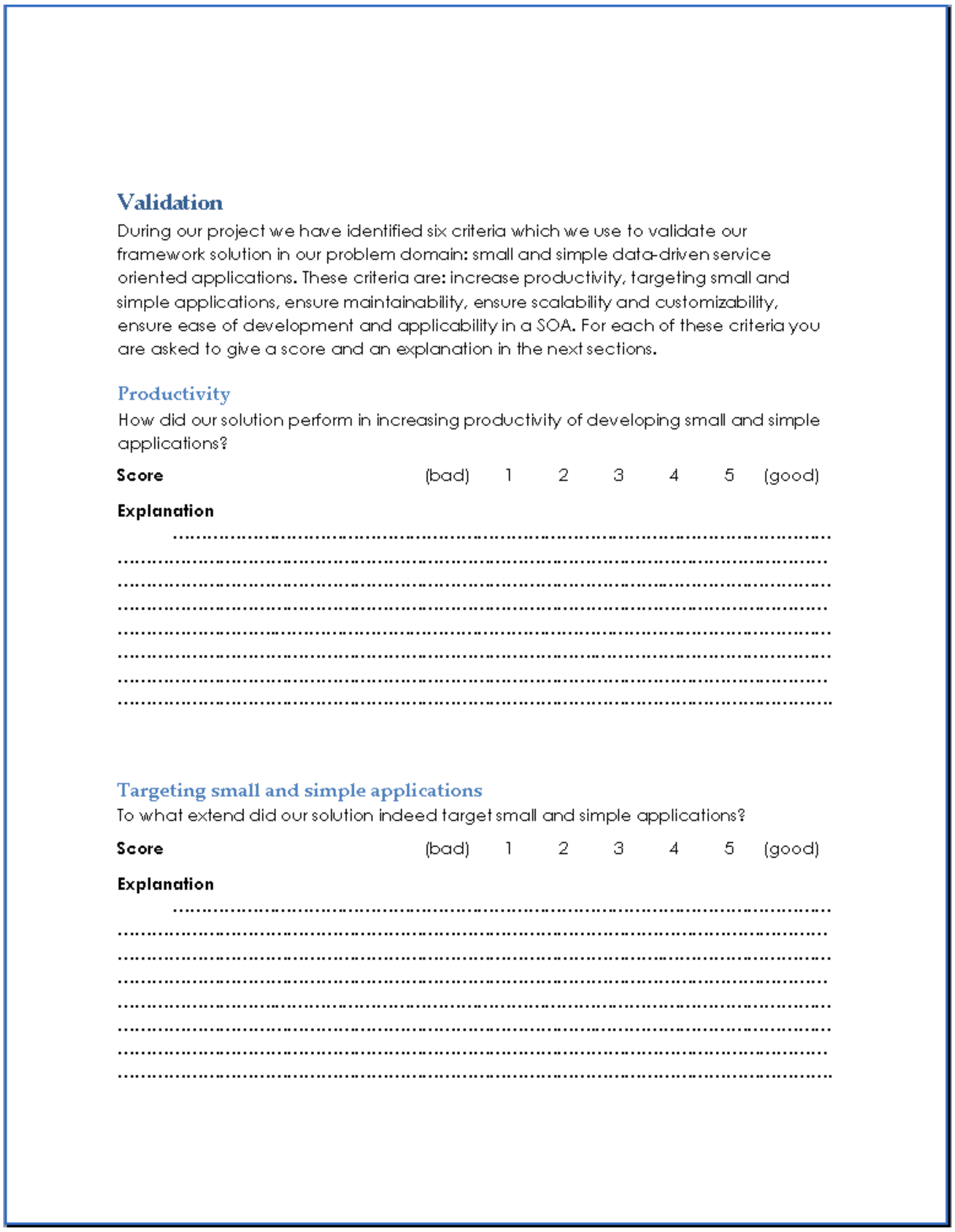
Applying Frameworks to Increase Productivity for Small Application Development: Spring.NET in combination with NHibernate

During our research project we have build a prototype in order to achieve an increase in productivity by applying frameworks, in our case NHibernate and Spring.NET. To validate if we achieved this goal, we included a validation phase in our project. The demo just presented to you, together with this questionnaire, form a large part of this phase.

Please fill in your name below.

Name ....

Figure A.1: Questionnaire - Page 1



Validation

During our project we have identified six criteria which we use to validate our framework solution in our problem domain: small and simple data-driven service oriented applications. These criteria are: increase productivity, targeting small and simple applications, ensure maintainability, ensure scalability and customizability, ensure ease of development and applicability in a SOA. For each of these criteria you are asked to give a score and an explanation in the next sections.

Productivity

How did our solution perform in increasing productivity of developing small and simple applications?

Score (bad) 1 2 3 4 5 (good)

Explanation

……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………

Targeting small and simple applications

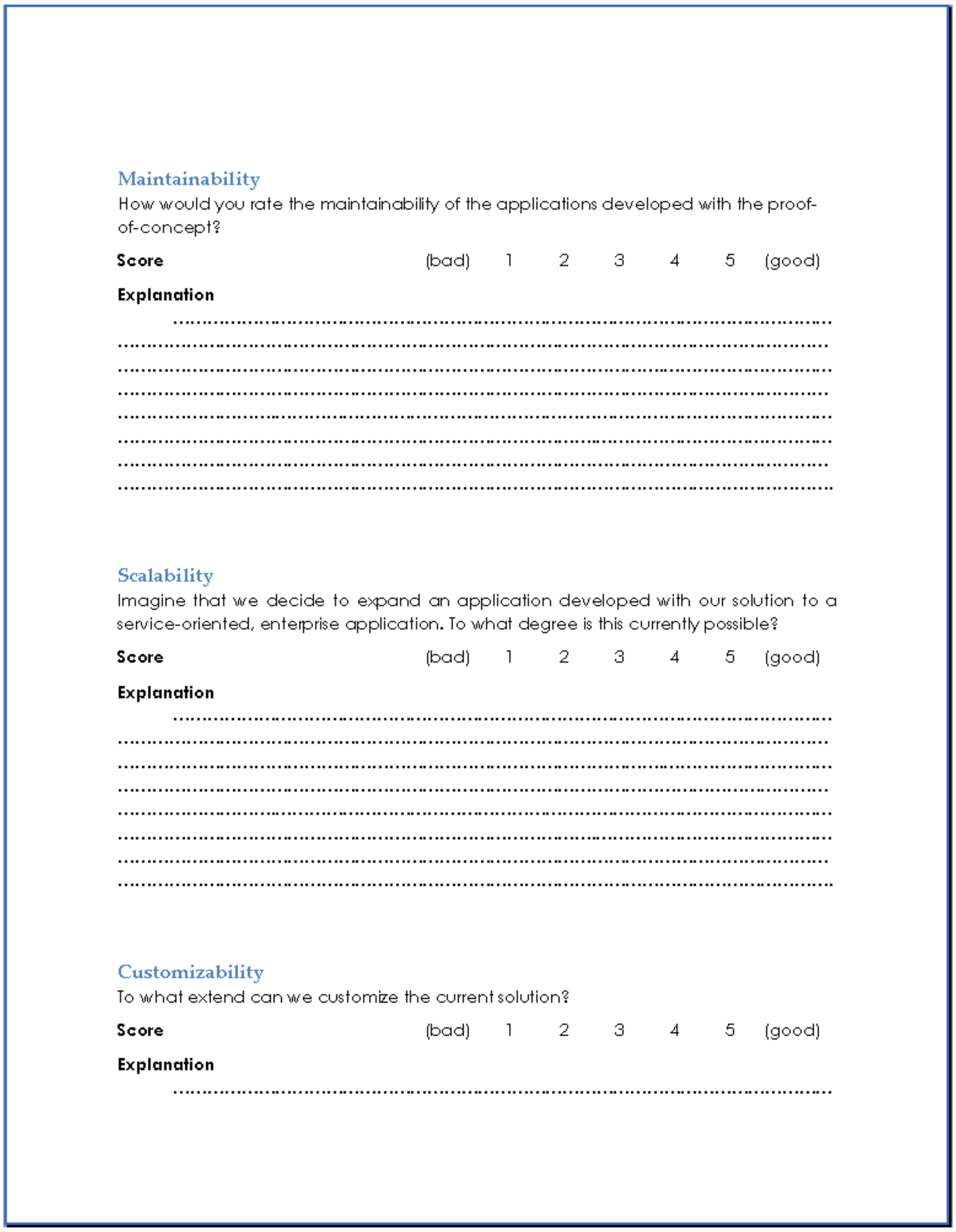
To what extend did our solution indeed target small and simple applications?

Score (bad) 1 2 3 4 5 (good)

Explanation

……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………
……………………………………………………………………………………………………

Figure A.2: Questionnaire - Page 2



Maintainability

How would you rate the maintainability of the applications developed with the proof-of-concept?

Score (bad) 1 2 3 4 5 (good)

Explanation

....
....
....
....
....
....
....
....

Scalability

Imagine that we decide to expand an application developed with our solution to a service-oriented, enterprise application. To what degree is this currently possible?

Score (bad) 1 2 3 4 5 (good)

Explanation

....
....
....
....
....
....
....
....

Customizability

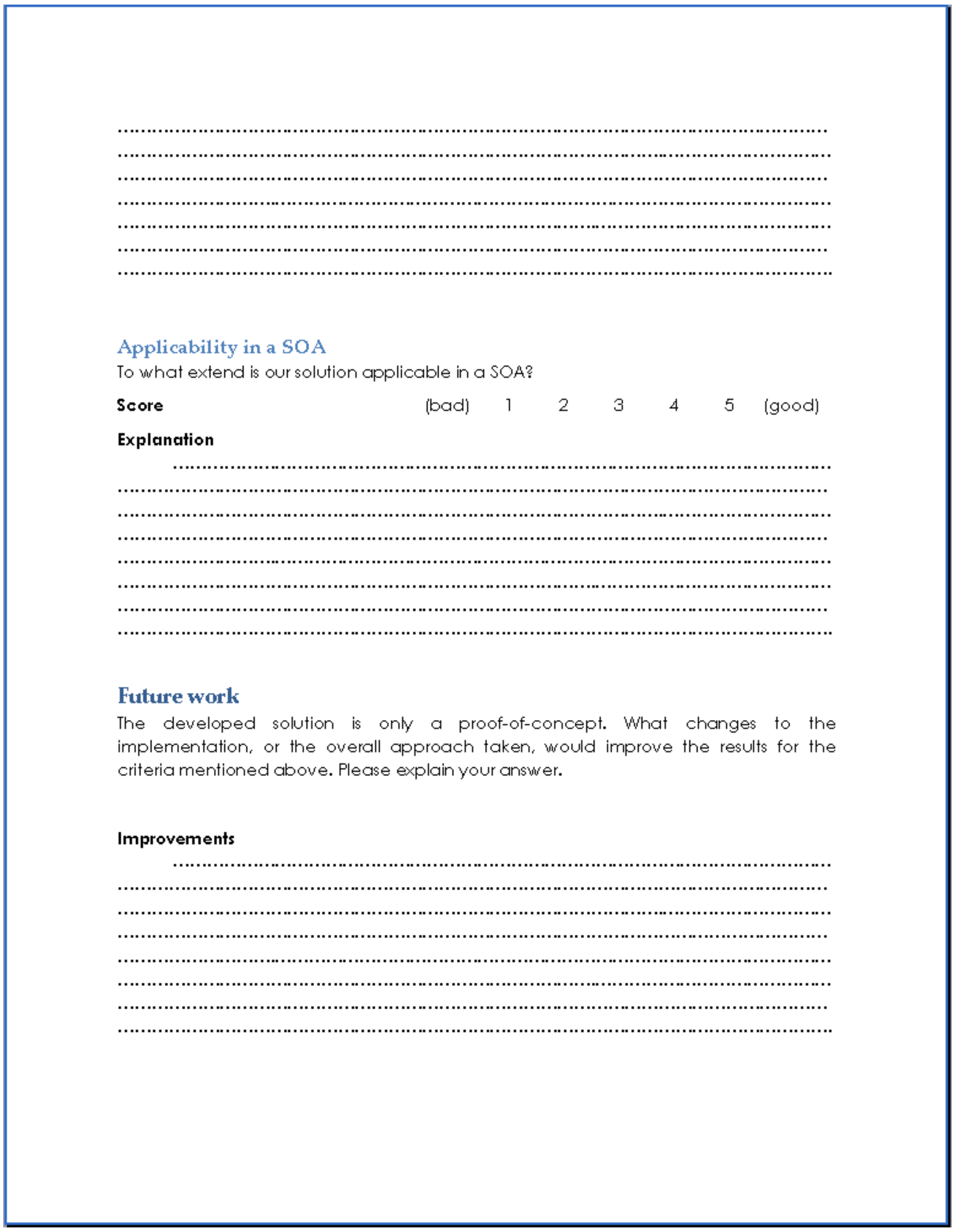
To what extend can we customize the current solution?

Score (bad) 1 2 3 4 5 (good)

Explanation

....

Figure A.3: Questionnaire - Page 3



……
……
……
……
……
……
……

Applicability in a SOA

To what extend is our solution applicable in a SOA?

Score (bad) 1 2 3 4 5 (good)

Explanation

…………………………………………………………………………………………………
……
……
……
……
……
……
……

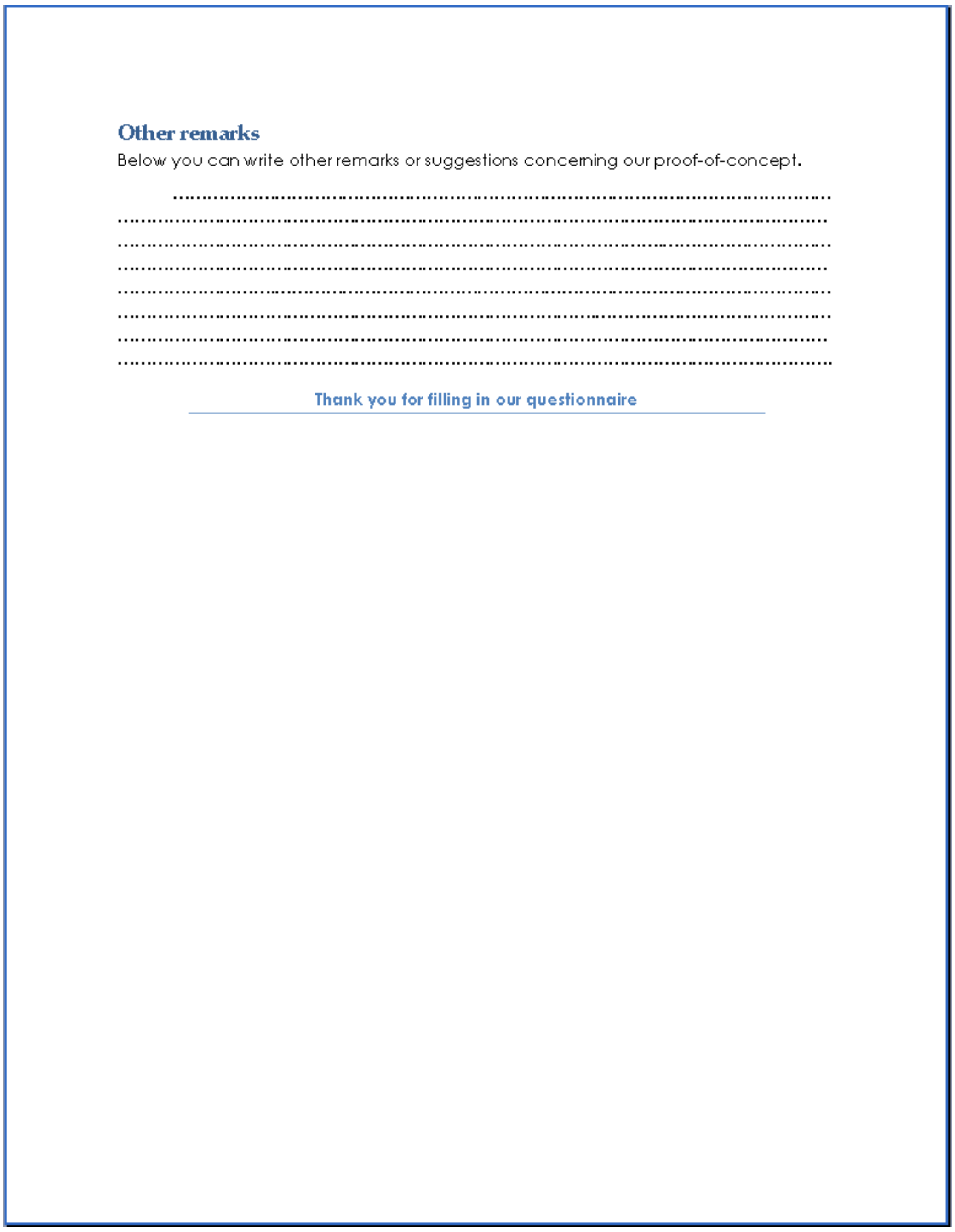
Future work

The developed solution is only a proof-of-concept. What changes to the implementation, or the overall approach taken, would improve the results for the criteria mentioned above. Please explain your answer.

Improvements

…………………………………………………………………………………………………
……
……
……
……
……
……
……

Figure A.4: Questionnaire - Page 4



Other remarks

Below you can write other remarks or suggestions concerning our proof-of-concept.

………
……
……
……
……
……
……
……

Thank you for filling in our questionnaire

Figure A.5: Questionnaire - Page 5

Bibliography

[AA05] John Arthur and Shiva Azadegan. Spring framework for rapid open source J2EE web application development: A case study. In *Proceedings of the Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks*, pages 90–95. IEEE, 2005.

[AHD00] *The American Heritage Dictionary of the English Language*. Houghton Mifflin, 4th edition, Sep 2000.

[AM04] Alain Abran and James W. Moore. *Guide to the Software Engineering Body of Knowledge*. IEEE, 2004.

[And07] Chris Anderson. *Essential Windows Presentation Foundation*. Addison-Wesley Professional, Apr 2007.

[ane07] The ADO.NET Entity framework overview. Website, Aug 2007. `http://msdn2.microsoft.com/en-us/library/aa697427(vs.80).aspx`.

[aop07] Aspect-oriented programming. Website, Apr 2007. `http://en.wikipedia.org/wiki/Aspect-oriented_programming`.

[asp07] Aspect#. Website, Aug 2007. `http://www.castleproject.org/aspectsharp/`.

[BK05] Christian Bauer and Gavin King. *Hibernate in Action*. Manning, 2005.

[Bon04] J. Bonér. What are the key issues for commercial aop use - how does aspectwerkz address them? 2004.

[BOS$^+$04] M.A. Barth, Drs. J. Onvlee, Ing. M.K. Spaan, Drs. A.W.F. Timp, and E.A.J. van Vliet. *Definities en telrichtlijnen voor de toepassing van functiepuntanalyse, Nesma Functional Size Measurement Method Conform ISO/IEC 24570 versie 2.2*. NESMA - Nederlandse Software Metrieken Gebruikers Associatie, 2004.

[BYM$^+$98] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo SantAnna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM 98: Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE Computer Society, 1998.

[CA05] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines - Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Microsoft .NET Development Series. Addison-Wesley Professional, Sep 2005.

[Che04] Xin Chen. *Developing Application Frameworks in .NET*. Apress, 1st edition, Apr 2004.

[CI05] Shigeru Chiba and Rei Ishikawa. *Aspect-Oriented Programming Beyond Dependency Injection*, volume 3586 of *Lecture Notes in Computer Science*, pages 12–143. Springer Berlin/Heidelberg, 2005.

[CKFS01] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98, 2001.

[Cli03] Marc Clifton. Aspect oriented programming / aspect oriented software design. Website, Apr 2003. `http://www.codeproject.com/gen/design/aop.asp`.

[con07] Configuration file. Website, May 2007. `http://en.wikipedia.org/wiki/Configuration_file`.

[EAA$^+$04] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pl Krogdahl, Dr Min Luo, and Tony Newling. *Patterns: Service-Oriented Architecture and Web Services*. IBM, 2004.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44:29–32, Oct 2001.

[FECA04] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit. *Aspect-oriented Software Development*. Addison-Wesley, 2004.

[Fil01] Robert E. Filman. What is aspect-oriented programming, revisited. Technical report, Research Institute for Advanced Computer Science (RIACS), May 2001.

[FK07] Michal Forgáč and Ján Kollár. Static and dynamic approaches to weaving. In *5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics*, Jan 2007.

[FNO92] Christer Fernström, Kjell-Hakan Närfelt, and Lennart Ohlsson. Software factory principles, architecture, and experiments. *IEEE Software*, 9(2):36–44, Mar 1992.

[Fow04a] Martin Fowler. Inversion of control containers and the dependency injection pattern. Website, 2004. `http://martinfowler.com/articles/injection.html`.

[Fow04b] Martin Fowler. Module assembly. *IEEE Software*, 21(2):65–67, Mar 2004.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. ACM Press.

[hql07] Hibernate query language. Website, Nov 2007. `http://www.roseindia.net/hibernate/hibernatequerylanguagehql.shtml`.

[Inf06a] Info Support b.v. Endeavour logische referentie architectuur versie 3.0. Technical report, Info Support b.v., Jul 2006.

[Inf06b] Info Support b.v. Endeavour technische referentie architectuur versie 2.5. Technical report, Info Support b.v., Feb 2006.

[Inf07] InfoSupport b.v. Endeavour digital coach. Website, 2007.

[JBo07] JBoss. Jboss aspect-oriented programming. Website, May 2007. `http://labs.jboss.com/jbossaop/`.

[JF88] Ralph E. Johnson and Brain Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, Jun-Jul 1988.

[Joh05] Rod Johnson. J2EE development frameworks. *Computer*, 38(1):107–110, Jan 2005.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001. `http://citeseer.ist.psu.edu/kiczales01overview.html`.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer, Jun 1997.

[Ko07] Ümit Karaka and Sencer Sultanoǒlu. Software size estimating. Website, May 2007. `http://yunus.hacettepe.edu.tr/~sencer/size.html`.

[LC03] Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[LG07] Ting Liang and Kit George. Unleash the power of query in visual studio ”orcas”. Website, Jun 2007. http://msdn.microsoft.com/msdnmag/issues/07/06/VBLINQ/default.aspx.

[loo07] LOOM.NET. Website, Aug 2007. http://www.dcl.hpi.uni-potsdam.de/research/loom/.

[Mao] Aop-based testability improvement for component-based software), author = Chengying Mao, journal = Computer Software and Applications Conference, year = 2007, month = Jul, pages = 547–552, volume = 2.

[mav07] Maverick.NET. Website, Aug 2007. http://mavnet.sourceforge.net/.

[mon07] Monorail. Website, Aug 2007. http://www.castleproject.org/monorail/index.html.

[MR97] Nenad Medvidovic and David S. Rosenblum. Domains of concern in software architectures and architecture description languages. In *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages*, 1997. http://citeseer.ist.psu.edu/medvidovic97domains.html.

[Nat06] Adam Nathan. *Windows Presentation Foundation Unleashed*. Sams, Dec 2006.

[nhi07] NHibernate for .NET. Website, May 2007. http://www.hibernate.org.

[nhr07] *NHibernate Reference Documentation - Version 1.2.0*, 2007.

[NS03] Yefim V. Natis and Roy W. Schulte. Introduction to service-oriented architecture. Technical Report SPA-19-5971, Gartner, Apr 2003.

[obj07] Objectbuilder dependency injection framework. Website, Aug 2007. http://www.codeplex.com/ObjectBuilder.

[omn07] The objectmapper .NET project. Website, Aug 2007. http://www.objectmapper.net.

[opf07] OPF3. Website, Aug 2007. http://www.opf3.com/.

[OT00] Harold Ossher and Peri Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. *Proceedings of the 22nd international conference on Software engineering*, pages 734–737, 2000.

[pdm07] PixelDragonsMVC.NET - an open source mvc framework. Website, Aug 2007. http://www.codeproject.com/aspnet/PixelDragonsMVCdotNET.asp.

[PES$^+$06] Mark Pollack, Rick Evans, Aleksander Seovic, Fedorico Spinazzi, Rob Harrop, Griffin Caprio, Choy Rim, and The Spring Java Team. *Spring .NET Reference Documentation Version 1.1*, Dec 2006. http://www.springframework.net/documentation.html.

[PGA02] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. ACM, April 2002.

[pic07] PicoContainer - Overview - What is PicoContainer? Website, Aug 2007. `http://www.picocontainer.org/`.

[Plu05] Daryl C. Plummer. Defining 'service' is key to implementing a service-oriented architecture. Technical Report G00125996, Gartner, Mar 2005.

[Sch07] Maarten Schilt. Applying model-driven development to reduce programming efforts for small application development. Master's thesis, TU Delft, Dec 2007.

[sds07] Sourceforge.net downlaod statistics. Website, Sep 2007. `http://www.sourceforge.net`.

[Sta06] Michael Stal. Using architectural patterns and blueprints for service-oriented architecture. *IEEE Software*, 23(2):54–61, 2006.

[Sun02] Sun Microsystems. Core J2EE patterns - service locator. Website, 2002.

[uip07] User interface process application block for .net. Website, Aug 2007. `http://msdn2.microsoft.com/en-us/library/ms998252.aspx`.

[Vas04] Alexandre Vasseur. Dynamic AOP and runtime weaving for java - how does aspectwerkz address it? In *Proceedings of the 2004 Dynamic Aspect Workshop (DAW04)*, pages 135–145, 2004.

[WB05] Craig Walls and Ryan Breidenbach. *Spring in Action*. Manning, 2005.

[xai07] Xmlserializer and ilist¡t¿. Website, Mar 2007. `http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=1376732&SiteID=1`.

Index