ECBF: E-Commerce Back-office Framework

Designing an e-Commerce back-office system using the ECBF

Supervisors	prof. dr. M. Aiello RuG
	prof. dr. A. Telea RuG
Author	B. de Raad

Date	18 August 2008
Version	1.5 (final)

i. Abstract

The online shopping phenomenon encountered an enormous growth in the last decade. In parallel, the amount of online suppliers is increasing as well. A lot of fanatic and optimistic entrepreneurs try their luck in the e-commerce business by taking their piece of the big Internet sales pie. Competition is very high and the e-commerce businesses have to deal with fast changing sales strategies and customer needs.

In this fast changing landscape, businesses depend on their software systems when executing their sales strategies or to be the first to make profit from new opportunities in the market. Therefore, an e-commerce back-office system supporting the Internet sales needs to be agile and easy to extend to fulfil these needs. This thesis will start with a description of the path to take in order to find a suitable architecture for such a system.

After defining the needs and requirements for a suitable architecture of an e-commerce back-office system, a generic framework will be presented. This framework is designed to give the opportunity to build robust systems based on a service-oriented approach. Systems implementing this framework are easy to extend and can take advantage of existing services within the framework.

Store.nl is a company selling products on the Internet. Their back-office system implements the framework and utilizes all benefits given by it. The last part of this thesis shows remarkable positive influences by the framework when comparing the Store.nl back-office system to other back-office systems.

ii. Content

iii. Figures, Tables and Code Snippets

Figures

Tables

Code snippets

1 Introduction

1.1 e-Business

The online shopping phenomenon encountered an enormous growth in the last decade. Where in the past Internet users avoided shopping online due to security and privacy reasons, the majority of regular Internet users now starts to trust online suppliers of goods and services and many have already bought such a good or service.
In The Netherlands, research showed (CBS, November 2007) that 66% of all Internet users between 12 and 74 years old have purchased some good or service online. Most of them (47%) are even regular electronic shoppers (e-shoppers).
The total turnover of online shopping is regularly investigated by Blauw Research and Thuiswinkel.org. In the past years, the turnover increased drastically. Customers are willing to spend more money on their purchases and the amount of customers is increasing. The most recent publication showed that there had been a turnover of 1.84 billion euro's in the first six months of 2007, an increase of 38% since the previous research.

In parallel, the amount of online suppliers is increasing as well. A lot of fanatic and optimistic entrepreneurs try their luck in the e-commerce business and try to take their piece of the big Internet sales pie. Therefore, competition in this field of business is hard and just trying to sell your goods as cheap as possible is not enough anymore. After sales service is just as important and so is one's reputation and name.

Within the business of electronic shopping, many changes in selling strategies or customer needs are following each other in a high pace. With such a high competitive business field, it is of the utmost importance to be able to react to these changes as fast as possible. Therefore, a supporting back-office system needs to be flexible and easy to change to fulfil all needs. At the same time, the system must be robust and its functionalities should not suffer from this high pace of system changes.

Finding the perfect balance between fast and robust, developing a software system can be a difficult task. Especially in a fast changing business field such as the e-commerce one it is important to have a system able to react to these changes. The basis of such a system must provide the possibility to react to changing requirements and must still be trusted for its robustness and correctness.
Also, new opportunities in the business field must be taken as fast as possible to stay ahead of competition. A software system in the e-commerce business field should therefore be extensible and existing parts must be easily integrated within the new parts of the system.

1.2 Research question

Considering the scenario sketched above, one could then ask oneself:

> *What is a suitable architecture for an e-commerce back office system considering the e-business environment and requirements?*

In order to answer this question one first has to answer the following questions:

- *What are the most important requirements for a back-office system in the e-commerce business field?*
- *Which techniques must be used in architecting an e-commerce back-office system?*
- *What needs to be done to successfully implement a system based on the requirements?*

1.3 Goals

The most important goal in this thesis is to find a suitable architecture for a back office system in the e-commerce business field. In search of an answer to this question, several architecture techniques must be investigated and compared. Using the requirements for the e-commerce back-office system, the best parts of each of these techniques should be considered and used in the approach to find a suitable architecture.

The goal of the project at Store.nl is to develop a system in order to provide a basis or framework which supports several (sub) systems of the organisation. Business logic and decisions made by the management of the organisation must be integrated into the system. The system must have an evolutionary character and should be able to change and grow over time. Because of the fast changing market Store.nl is operating in, an important issue is the flexibility of the system. In practice, this means that several parts of the system are subject to change and it should therefore be possible to update the system in a fast and robust way without affecting other parts in the system.

1.4 Thesis organisation

This thesis is organised as follows: after the introduction the key understandings about electronic shopping will be explained. With this information the requirements for a back-office system in the e-commerce business field will be gathered. After that, the latest developments of several architecture techniques are being discussed. This chapter will conclude with a summary of these techniques and which parts can be used for implementing an e-commerce back-office system.
Chapter 4 contains an overview of the implemented back-office system for Store.nl and its functionalities. The chapters succeeding will elaborate on the building blocks and framework of the system.
The last part of this thesis exists of the evaluation of the back-office system. The main requirements are evaluated and an overview is given. This part will finish with a comparison of two other back-offices and again, with focus on the key requirements.

2 e-Commerce

2.1 Introduction

Electronic commerce defines a very broad domain of buying and selling goods or services over electronic systems using a computer network. This chapter focuses on selling and buying products over the Internet and especially the forces playing roles in sales strategies.
This chapter will first elaborate on the participants of e-commerce to fully understand their roles in buying and selling products. Next, we will define why e-commerce is growing in popularity. With this information we can define what makes an e-business successful and what should be the focus for a back-office system.

2.2 e-Commerce participants

In the domain of buying and selling products over the Internet there are four important participants: customers, manufacturers, vendors and information comparing parties. All of them have an important role in the e-commerce business field. The following will focus on the basic roles of the participants in order to give a global idea.

2.2.1 Customers

Customers are the most important participants in the domain. Without customers there's no market and therefore the needs of customers are important to understand. Customers want to buy the cheapest products, want the best after-sales service and want to have their products delivered at their home as fast as possible. The website they visit to buy their products should be easy to understand, easy to search, look reliable and easy to order a product.
Customers are in the position to choose between shopping online and shopping in traditional (offline) shops. With the previous stated needs, customers will make a decision between these options.

2.2.2 Manufacturers

The products sold on the Internet are manufactured by the manufacturers. Most manufacturers do not sell their products to customers but to vendors. This way they can sell many products in one transaction and they have no trouble in delivering each product to a different address.

2.2.3 Vendors

Vendors want to sell products on their website to customers. There is much competition between vendors to sell their products as cheap as possible and deliver the product as fast a possible to their customers. Vendors also have competition from traditional shops. With these shops, price is not a competition issue since online shops can safe investments on a physical shop. Availability of products, service and customer-trust however, are issues to cope with.

2.2.4 Information comparing parties

Some websites focused on comparing products on the websites of vendors and show prices and availability of the products on their own website. Customers don't have to search all vendor websites to find the cheapest or best available product but can visit a comparing website instead. The comparing websites are very important for a healthy competition between the vendors and they demand very narrow selling strategies.

2.3 The power of e-commerce

There are many properties of online shop which can not be matched by traditional physical shops. While there are also advantages for these traditional shops, online shops are getting more popular and are winning their share in the sales market. In [Fensel '03] some of the "superiorities" of online shopping were stated.
Online shops make all their products available for search. Customers are only required to input their needs with some additional mouse clicks and the desired information will appear on their screen. One step further than this individual product search can be corporative product search. By building up user profiles, similar customers can be discovered and product recommendations can be made automatically. Compared with traditional shops, user profiles are almost impossible to create and individual product search would take much more time for a customer compared to an online search.
Another very important characteristic of the Internet is its transparency. Complete market transparency could be achieved, because product information from all online shops can be obtained and compared. However, to make this process efficient this must be done automatically. Manually searching all available online shops would cost a customer a lot more than one gains from it.
With help of automatic comparing systems implemented by information comparing parties, customers can find the cheapest price of the product they want to purchase. This will automatically lead to a healthy market competition. Also other properties next to price can be used to select the preferable shop. Available stock, delivery costs or other customer's reviews are examples.
Easy access is another advantage of online shopping. Physical and time barriers have disappeared and the whole process of buying goods can be automated. In theory, it could even automate the process of searching and buying the product itself with use of special trained software agents.

2.4 e-Commerce system requirements

As shown in the previous section, providing accurate and clear information is an important aspect for customers. Their decision of buying a product depends on the information available on vendor and product comparing websites. One of the properties required by a system powering the website is to provide a means to provide information on the website about their products, prices and stock. Improving accessibility and usability of the website is also part of this information providing: the information needs to be available in a clear structure.
In [Keen '04] it is stated that e-business on demand means to enable customers to succeed in an environment with an unprecedented rate of change. Customer needs are changing in a high rate and the e-business systems should be able to alter rather easy and fast in order to comply with the company's customized strategies. Flexibility becomes an important aspect for a system to succeed in the e-business.

A system for a company selling goods on the Internet needs to have more functionality than powering an online website. To name an example: buying goods from manufacturers is a big part of the whole selling products activity. Depending on popularity or amounts of orders, decisions need to be made in when and how many products to buy or to have in stock. Again, with the changing customer needs these parts of the system will also change in the way business strategies are planned.

Over time in the last decades the diversity of products or goods sold on the Internet has grown in a fast rate. For many segments of products different requirements and business strategies exist, however, the basic functionality needed by a system remains the same. A system should therefore be extensible to support new segments but must also be able to reuse existing components to integrate into the new system parts.

2.5 Overview

This chapter described the basic principles of e-business. An overview of the participants playing roles in e-business is given in Table 1.

Participant	Description
Customers	Buy goods online on website. Have many demands and needs. Change in behaviour.
Manufacturers	Manufacture products. Want to sell products on regular basis in large quantities.
Vendors	Sell products to customers on the Internet. Have much competition. Need to act fast to react to changing customers needs.
Information comparing parties	Compare information about products from different vendors.

Table 1: e-Business participants overview

With these participants, the needs for a system supporting an online website to sell products are extracted. The needs, together with the additional requirements, resulted in the basic non-functional requirements listed in Table 2.

Requirement	Description
Flexibility	The system needs to react to changing requirements and business strategies.
Maintainability	Like flexibility, the system must be maintainable because changes in the system will not be uncommon.
Reusability	New parts of the system will often make use of existing parts and therefore it is important to use these parts again instead of implementing them again.

Table 2: e-Business system requirements overview

3 State of the art

3.1 Introduction

The software architecture of a system is the outcome of the design of a system. Sub-systems, communication protocols, design decisions and the structure of the system are reflected by or documented in the architecture. The sub-systems and components are typically specified in different views to show the relevant properties of the complete system [Sommerville '07, Buschmann '96]. An architecture is present in every software system. Even the smallest systems have one, however not always documented as such. Many different architecture styles and patterns exist and their popularity changes over time. Also, the steady growing of computation power and memory facilities of modern computers creates new opportunities for new architectural styles and patterns.
In search of a suitable architecture for an e-commerce back-office system, this chapter will give an overview of state of the art architecting styles and concepts. The focus is on architecting methodologies to tackle the non-functional requirements found in the previous chapter: flexibility, maintainability and reusability.

3.2 Service-oriented architecting

Service-oriented architecture (SOA) is a relatively new concept and a lot of discussion exists about what SOA's exactly are. The whole concept of an SOA focuses on the definition of a business infrastructure, which encapsulates a number of business processes. The goal of SOA is to decouple business applications from technical services and make the enterprise independent of a specific technical implementation or infrastructure. Before diving into how a service oriented architecture can be seen a few definitions of SOA's found in literature will be discussed first.

3.2.1 Definitions

[Papazoglou '07] defines the following: "SOA is a logical way of designing a software system to provide services to either end-user applications or to other services distributed in a network, via published and discoverable interfaces" and "SOA is a meta-architectural style that supports loosely coupled services to enable business flexibility in an interoperable, technology-agnostic manner". In [Kraftzig '05] the focus is on enterprise architectures and their specific characteristics. In their opinion, an SOA is a software architecture that is based on the key concepts of an application front-end, service, service repository and service bus. A service consists of a contract, one or more interfaces, and an implementation. Another interesting view on SOA is the remark in [Taylor '06] that SOA is an IT architecture that uses open standards such as XML and web services. This way, applications can exchange data or operating instructions independent of programming languages or communication protocols.

3.2.2 Concept

An SOA can be divided into four main components: a service repository, a service bus, application front-ends and services (see Figure 1). The application front-ends form the basic applications. These can exist of existing custom built applications like CRM or ERP packaged applications or business intelligence applications but also newly formed applications using services available in the service repository [Kraftzig '05, Arsanjani '04]. For the interconnection between the application front-ends the service bus is used for communication with or between services.

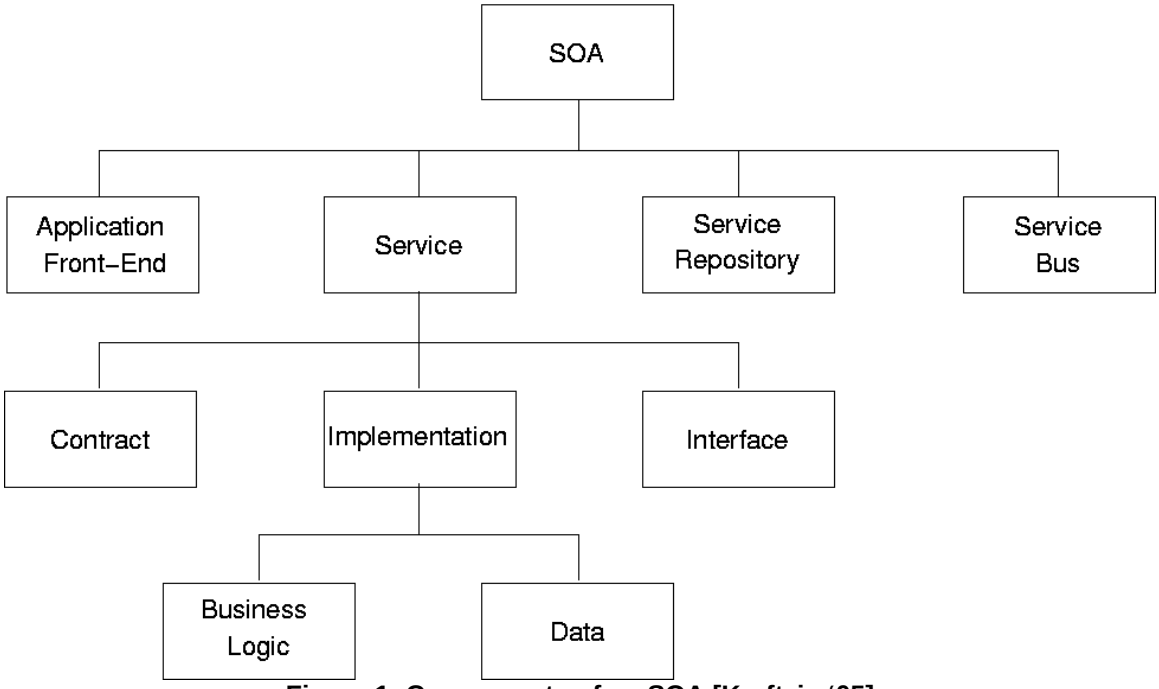


Figure 1: Components of an SOA [Kraftzig '05]

Services provide a higher-level of abstraction for organizing applications and are standardized [Singh '05]. Each service consists of an implementation that provides business logic and data, a service contract that specifies the functionality, usage, and constraints for a client (a client can either be an application front-end or another service) of the service, and a service interface that physically exposes the functionality (Figure 1).

Four types of services have been identified [Kratzig '05]:

- Basic services
- Intermediary services
- Process centric services
- Public enterprise services

Basic services are the basic elements that are used to compose an SOA. They provide usual business functionality such as a contract administration service or a billing service.

Intermediary services are used to bridge technical gaps in an architecture. They are linking services with other services or application front-ends and services in being gateways, adapters, facades, and other functionality-adding services. These services can be composed of other services. This means that functionality of the services is bundled into a flow to form a new service or act as a new single application [Arsanjani '04]. Composition refers to any form of putting services together and achieves some desired functionality [Singh '05]. Especially in e-commerce, composing new services will meet the specific needs of users.

Process-centric services are used in an SOA to control and maintain the state of the enterprise's business processes. However, the application front-ends will be rather complex in such a case because they have to control the business process (therefore, a software architect may choose to ignore such services). A modern travelling agency, for example, is offering a website where customers are able to book a vacation. The website can be very complex if it has to control the whole booking process. To make life

easier for website designers, a special booking service can be used to drastically reduce complexity of the application front-end.

Public enterprise services are services that are used to integrate enterprises (B2B) or to offer customers some facilities, such as a website service where they are able to buy products.

3.2.3 Why use an SOA

Why should an organisation use an SOA for structuring their application landscape? There are a number of important reasons. One of the most important reasons is that using an SOA is an effective way to organize the whole IT-landscape of an organisation. Different services can be put in place and can be connected.

Furthermore, it makes an organisation more agile. The main elements of service-oriented architectures are: loose coupling, implementation neutrality, flexible configurability, long lifetime and granularity [Singh '05]. These properties make an application more agile and an organisation that is more agile has the ability to change its business architecture easier than an organisation that is not agile. Hence, an organisation that intends to use an SOA has the chance to become more competitive than before.

The infrastructure of an organisation becomes more flexible when SOA is used. The ability to change the business architecture also reduces the technological dependency. Another benefit of SOA is the ability to introduce new services in a relatively short time and integrate them directly into the organisation architecture. Not only new services can be added, but existing services can also be copied and adapted. Therefore, reusability is another important reason why to use SOA.

The reusability, flexibility and agility of SOA systems will cause a reduction of costs in the long term. When in a later stadium organisations need to change their architecture due to changes in business strategy, the organisation with an SOA will have much less costs to adapt to the new strategy than the organisation without an SOA.

Developing software systems is very expensive. Also, many projects fail in some way. The end product is, for example, not the result that was expected. Also, often too much time is needed to deliver a successful product. Using the SOA paradigm when developing software, results into a more efficient development process.

Finally, SOA is understandable for all stakeholders because it is about describing business processes [Gerson '06, Tews '07].

3.2.4 When to use an SOA

When should an organisation start to think about SOA? Presenting an answer that is useful for all businesses in general is impossible. Therefore, presented below are only some guidelines that may be useful to come to a conclusion with regard to the use of an SOA.

First of all, the size of an organisation is important. Nowadays, big organisations have, in most situations, a lot of applications installed. Dependencies between applications exist and can be identified by inspecting the connections between the applications. One can picture a complex web of software systems that are interconnected with each other. In such a situation, it is difficult to keep all data consistent. Also, changing some applications can easily affect other software systems.

Using an advanced SOA can be a good method to avoid these two problems, especially for large organisations. An SOA encapsulates different services. These services are used, and can sometimes be reused, as a kind of building blocks [Tews '07]. In this way, a shift is realized from a complex web of tightly coupled applications to a loosely coupled structure of application front-ends and services. Also, an important aspect of SOA is the way data management is organized. Simply speaking, basic services are the owners of the data. The only way to gain access to data is through the basic services. In this way, consistent data is guaranteed.
Nevertheless, a somewhat simpler SOA is also useful for smaller organisations because it is a method that structures one's software systems in a, in business perspective [Gerson '06], logical way.

Secondly, the context in which an organisation is operating is very important. Imagine a market in which many competitors are operating and where customers are constantly asking for new products. To survive, an organisation must be dynamic and creative. That is, a constant drive for designing new products and having short time-to-market must be two main characteristics of an organisation.
In this situation, it would be interesting to use the SOA approach for implementing an enterprise architecture. As a consequence, SOA increases the agility of the organisation's systems. It actually means that an organisation is capable of quickly reflecting changes in business processes, which is needed to support its new strategy. As a result, an organisation will become more competitive because it is reducing the time-to-market.

Finally, more organisations start to cooperate with each other and try to realize a common goal. Therefore, organisations have to look for partners to form alliances, which, for example, have a higher chance to survive in their market. The integration of the computer systems is for those alliances, or other named collaborations, a challenging task. SOA makes realizing such a task easier [Vernadat '07].

3.3 Pattern-oriented architecting

When a system has to be designed, a very common way to improve the non-functional requirements is to make use of standard proven design patterns. These patterns are a guideline while architecting the new system and can be bent to get the desired system's 'shape' right.
The first thoughts of patterns in modern software and computer systems design originated from the work of architect Christopher Alexander. 'Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.' [Alexander '79]. Modern patterns still use this definition as a basis for pattern descriptions.
Some important patterns used to solve architecting challenges for non-functional requirements like maintainability, flexibility, reusability and manageability will be stated in the following sections.

3.3.1 3-tier pattern

Data intensive systems have three characteristic properties. First of all, they operate on data with a particular structure. Second, they provide specific functionality on the data and lastly, they present the user with a view on the data and the functionality.
The 3-tier pattern separates the three characteristic properties of a data-intensive application into three layers. These are, from bottom to top: The data layer, the application logic layer and the presentation layer.
The data layer provides access to the application's data, independent of its structure, by exposing a well-defined interface to the logic layer above it. This logic layer contains all functionality relevant to the application. It can access data through the data layer and it,

in turn, exposes a well-defined interface to the presentation layer above it. The presentation layer provides the user with a view on the data and the functionality exposed through the logic layer.

The problem with data intensive systems is that data is used throughout the entire system. If such a system is implemented using a monolithic approach, then the structure of the data will be tightly interwoven with the functionality on those data and perhaps even with the view presented to the user.
Whenever the structure of the data changes, all of the functionality and views concerning those data need to be modified as well. In a similar way, changing the functionality can affect the view as well. By strictly separating data structure, functionality and presentation from one another in three, well-defined layers, changes to either of these properties will not affect another. Furthermore, since the two bottom layers expose well-defined interfaces, the functionality offered through them can be reused. This means that functionality can be implemented once and then be exposed through the interface, allowing proper reuse and preventing multiple implementations of the same functionality. Since the layers only have dependencies of other layers though their interfaces, changes to one layer do not affect the implementation of another layer, provided the relevant interfaces remain unchanged. Therefore, the maintainability and flexibility will improve using this pattern.

3.3.2 Application Server Architecture pattern

A common pattern used in systems for e-businesses is the Application Server Architecture pattern [Dyson ‘04]. Adopting this pattern usually results in one single application using a separate database and web server. The application is internally partitioned according to the preferred logical architecture.
Because of the internal partitioning of the application, communication between different ‘modules’ of the application has no need for a complex communication mechanism. Communication can be very straightforward and the application’s maintainability will improve. However, having lost the separation of different parts of a system by putting them all in one application will also lead to the loss of easily isolating bugs.

3.3.3 Peripheral Specialist Elements pattern

When taking a closer look at the Application Server Architecture pattern it becomes clear that the pattern prescribes a separate database and web server. These elements of the complete system function better without being integrated within the single core application. The Peripheral Specialist Element pattern states to separate ‘extraordinary’ functionality (like the database and web server) from the core application and implement it in specialist or dedicated elements [Dyson ‘04]. These specialized elements can be integrated with the core application, but are executed and maintained independently from it. An example can be the use of a mail server to send e-mails to business relations. This part of the system does not have to be in the core application, but can exist next to it.
By applying the Peripheral Specialist Element pattern, the system’s maintainability improves because the peripheral elements of the system can be maintained separately from the core application. This is also true for the flexibility of the system: new functionality can be added in a flexible manner by integrating a new peripheral element into the system instead of integrating it into the core application.

3.3.4 Dynamically Discoverable Elements pattern

Dynamically Discoverable Elements [Dyson ‘04] is a pattern used to make systems more flexible and maintainable. By using components that can discover new system elements, the system implementing this pattern can be extended dynamically. The main purpose of this pattern is to enable a system to introduce additional capacity without having to take

the system offline. By looking at the pattern purely from a software-side of architecting a system, this pattern can be used to dynamically add new software elements to a system without having the trouble of taking the system offline in order to integrate new elements. The component which enables to discover new parts of the whole system can be seen as the core of the system. By using such a component, the system's maintainability level increases. Parts of the system can be upgraded, removed or added without having to change other parts of the system.

3.4 Discussion

This chapter showed two different state of the art architecture styles. The first one is the service oriented architecture style. Some properties of this style have a great positive influence on the non-functional requirements needed in the e-commerce business field. An SOA is loosely coupled and is flexible configurable which makes a system implementing an SOA agile and maintainable. The services which are the central part of an SOA can be very useful when implementing applications which have a need for a high maintainability level. Services hide their implementation and can be maintained separately. At the same time, the parts of a system using the services can be maintained without knowledge about the actual implementation of these services. Besides a positive influence on the maintainability level of a system, services are very suitable for code reuse.
Since maintainability and reusability are necessary properties in an e-commerce back-office system, using a service-oriented approach in designing such a system will probably add value to the levels of maintainability and usability.

The second architecture style is the pattern oriented one. Four patterns having a positive influence on the non-functional requirements needed for a back-office system for an e-commerce business were given in this chapter. All patterns are applicable when designing an e-commerce back-office application and improve the maintainability and usability of the total system.

In the following chapters both architecture techniques are combined in order to create a framework. This framework is designed to give the possibility to create agile applications which are easy to extend and maintain. The service-oriented approach of the framework will also provide code reuse possibilities.

4 The Store.nl back-office system

4.1 Introduction

Store.nl mainly does its business through their front-end websites. The website is the first impression for potential customers and it is important to attract new customers with a good looking and easy to use website. In February 2006, Store.nl decided to design and implement a new website to satisfy these needs. A few months later, this project was outsourced to a third party. The latter company is specialised in usability optimization and has many experts and a lot of experience in designing user friendly websites. One of their methods is the use of an eye tracking system. This system is able to determine which parts of the website several test persons are looking to and which parts "catch the eye". With this information, the website can be optimized even further to reach maximum usability.
With the existence of a UNIX server and experience in building PHP-driven websites, Store.nl decided to implement its new websites in PHP. A MySQL database would store all data used on and generated by the website. Several parts of an online shop were off the shelf components implemented by the company itself and this reduced the implementation time and costs.

The Store.nl website exists of several sub-sites, each with its own domain. However, the sites are all connected by means of a tab-structure on top of each website. Figure 2 shows the top of the screen when the tab "Audio" is selected. The other domains are all accessible through the tab-structure.

Figure 2: Store.nl tab-structure

Because all sub-sites are located on one server, it is possible to have only one database for the whole website. Products and orders are all kept in this database and to distinguish the different sub-sites a navigation structure for each sub-site is stored in the database as well. Two example websites are shown in Figure 3.

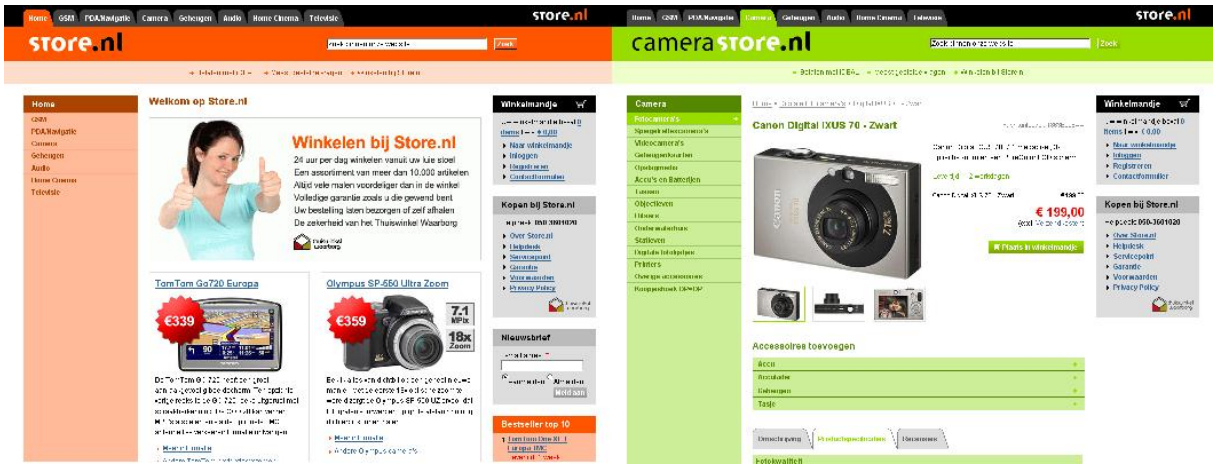


Figure 3: Store.nl sub-sites examples

An important agreement at the start of building the new website was the decision to build the complete back-office in-house. While the third-party was working on the website, Store.nl started another project in order to be able to manage the website. This ultimately means that the database used by the website is managed by another system. Because the back-office system is not implemented on a web-based platform, the database is the only overlapping part of the front and back-office systems. Figure 4 gives an architectural overview of the web-site front-end, back-office and database. This approach is a clear example of the application server architecture pattern which separates the main application from the web server and database server.

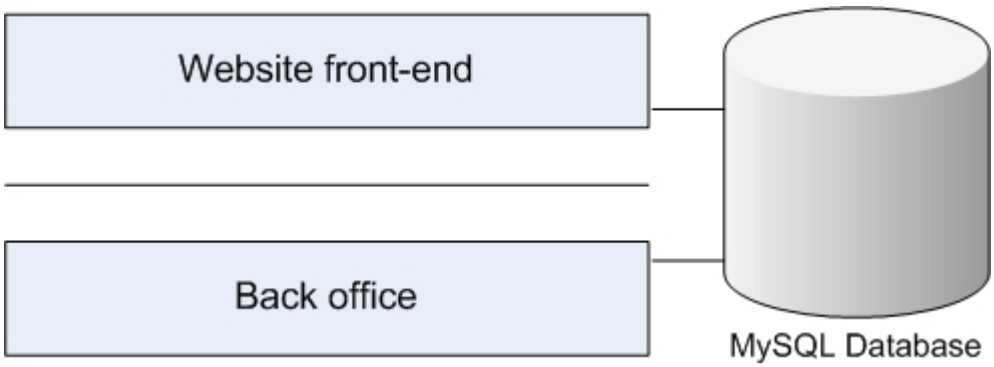


Figure 4: Front- and back-end overview

4.2 Back-Office

The back-office system is built in the Microsoft Visual Studio programming environment. The system is completely designed and implemented for Windows, but has a constant connection with an online MySQL database.
The main purpose of the back-office system is to provide functionality to the Store.nl employees like the ability to view, update and use information about customers, products, orders and more. Other purposes are connections with third party software or xml exchange with other businesses (B2B).

The Store.nl back-office system consists of many modules with different functionalities. All modules are connected in one main application and can easily be added or removed from the main application.

4.2.1 Module overview

Table 3 gives an overview of all modules existing in the back-office system at current date of writing.

Specific management modules	Customer management module
	Delivery date request management module
	Image management module
	Invoice management module
	Manufacturer management module
	Mobile contract management module
	Mobile subscription management module
	News letter management module
	Order management module
	Payment method management module
	Payments management module
	Product accessory management module
	Product category management module

	Product management module
	Product offers management module
	Product review management module
	Purchase invoice management module
	Purchase order management module
	Receipt management module
	RMA management module
	Shipping method management module
	Statistics module
	Supplier management module
	To-Do management module
Supporting modules	E-mail module
	File transfer module
	Print module
	Search module
Third party connection modules	Camtrans package delivery details export module
	Excel product and price sheet import module
	Icecat product details XML import module
	TNT package delivery details export module
	Triple deal payments (credit card, iDEAL, etc.) import module
	Unit 4 Agresso (purchase) invoice export module

Table 3: Store.nl Back-office module overview

4.2.2 Specific management modules

The modules arranged under the management modules are modules for specific tasks for all Store.nl employees. In Figure 5 the modules are represented in an architectural way. There are four main disciplines in the modules:

- Invoice management
- Order management
- Product management
- Purchase order management

These modules are the most central modules in the Store.nl back-office system. Most other modules are somehow connected to one or more of these main modules. The blue boxes completely inside the light ochre boxes are modules which exist only in the main discipline of the representing box. An example is the product category management module which has only connections with the product management module. The purpose of this module is to be able to add, remove and edit several categories and subcategories of products. The products in the database can be connected to these (sub) categories.
The blue boxes overlaying multiple disciplines are connected to more than one main discipline. An important module is the customer management module. This module is connected to both the orders and invoice management modules. Within the customer management module, information about customers can be edited or new customers can be added manually. However, most customers register themselves on the Store.nl website and the information in the customer management module is used for read-only purposes.

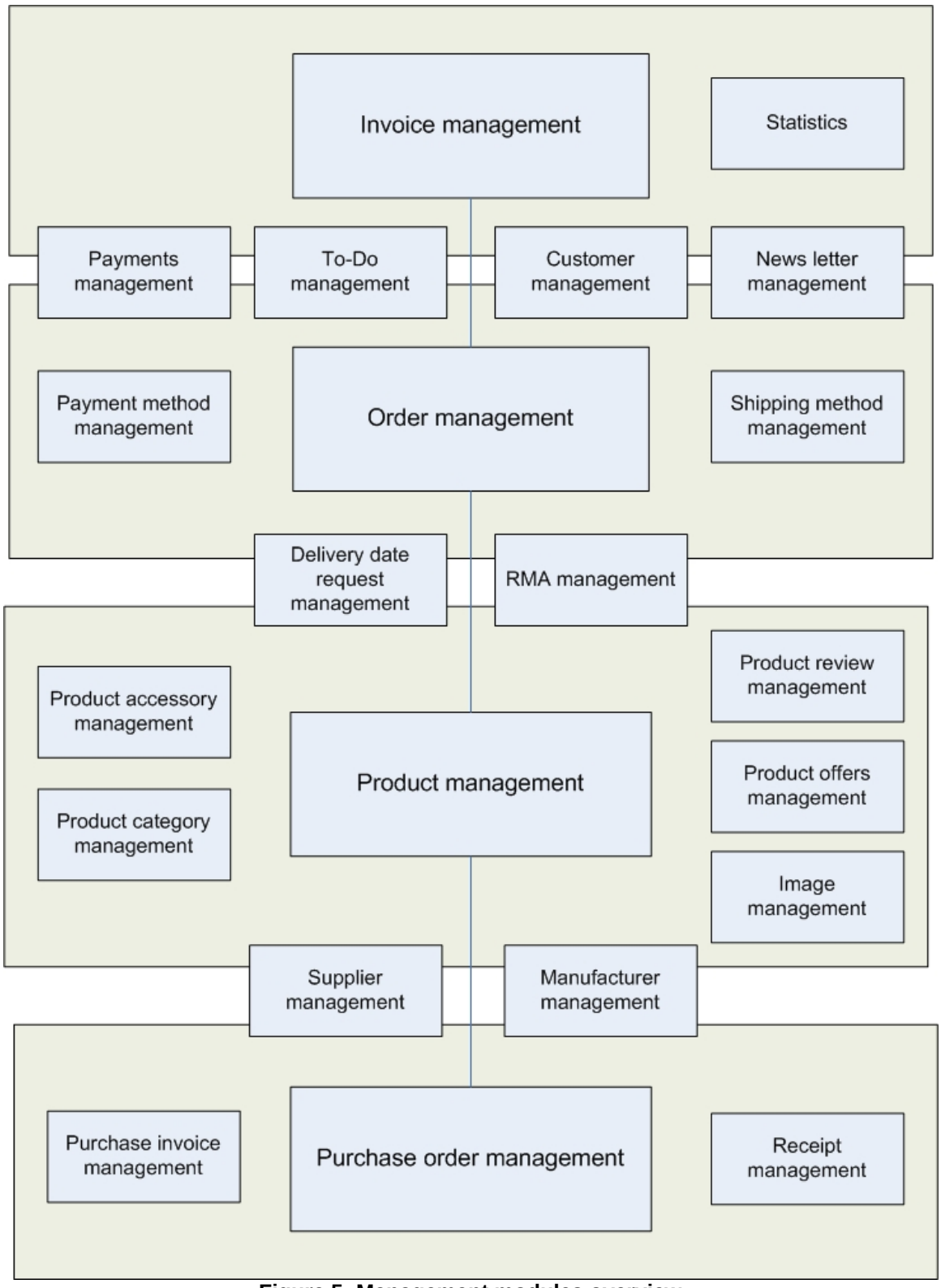


Figure 5: Management modules overview

4.2.3 Supporting modules

The supporting modules in the Store.nl back-office system are modules connected to most of the other modules within the system. For example the e-mail module is used in many modules for different purposes. The invoice module uses the e-mail module to send e-mails to customers about and with their invoices; the order management sends e-mails to customers about the status of their ordered products. In other modules like the purchase order management module e-mails have to be sent to suppliers to order new products.
The search module is the only visible module for the end-users of the back-office system. In this module all kinds of search results can be sought and many criteria can be filled in to use in the searching process. Depending on the search result chosen, double clicking on the result brings the user to the relevant module.

4.2.4 Third party connection modules

As in many systems, converting information from the system to other systems brings a lot of time-consuming work with it. Therefore, automated connections between these systems are very important. The Store.nl back-office system has several connections with third party systems. Some of them are connected by means of xml documents while others have direct connections with a database or connector.
An example is the product and price sheet excel import module. This module is used to import excel documents and process information about products. These sheets are supplied by suppliers or manufacturers and are updated on a regular basis. Newly added products or removed products can be identified by the module and changed prices are directly calculated in the Store.nl database using the latest margin used for this product.
Together with the Icecat product details xml import module the total offer of products by Store.nl is completely automated and is always up-to-date.

4.3 E-Commerce Back-office Framework

The modules listed in Table 3 are connected to each other using an application framework. This framework supports several kinds of possibilities to offer functionality of different modules to other modules using the framework. This way, a complete system can be built of independent modules without losing connection between the modules.
Another important building block for the modules listed above is the possibility to connect to some kind of database. The Store.nl back-office makes use of a data access framework to support the connection between the modules and the database. In the next chapters the back-office frameworks will be elaborated and discussed.

Together, the application framework and the data access framework form a complete foundation for building a modular system for e-commerce back-office systems. The collaborative framework is called: the e-commerce back-office framework (ECBF).

5 Data access framework

5.1 Introduction

An essential part of the complete back-office system is the connection with the stored data. In order to be able to view, edit and delete information, about customers for example, the system must have a means to access this stored information.
To realise such a connection a data access framework has been designed. This framework can be used in any visual studio solution to access several kinds of databases. The Store.nl back-office system implements this framework and makes use of the supported connection to a MySQL database.
The data access framework consists of three elements: Microsoft Datasets, data access layers and database adapters. Figure 6 gives the architectural overview of how the elements are related to each other.

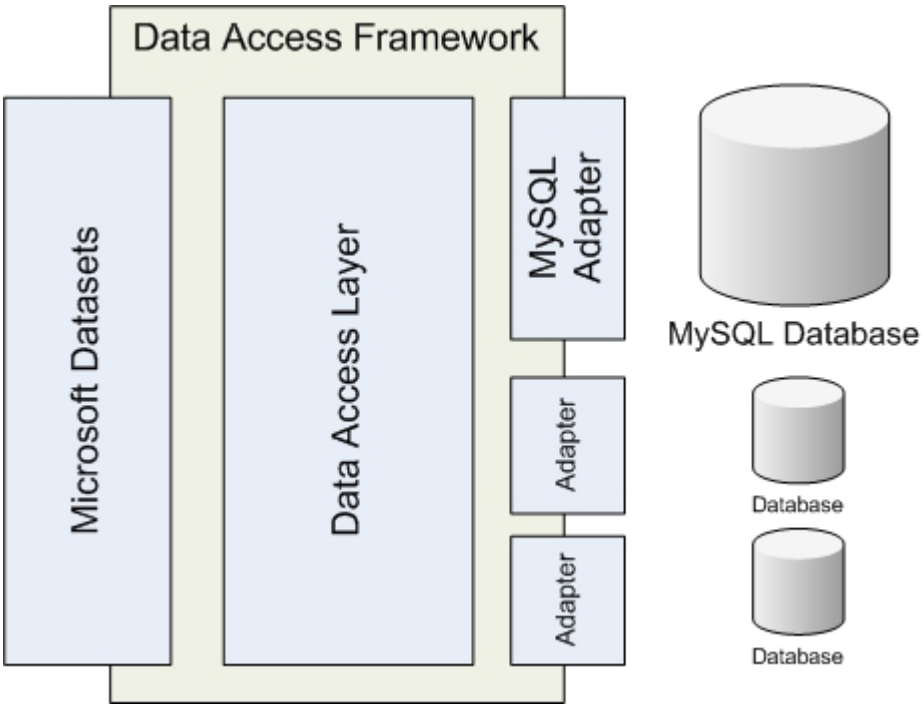


Figure 6: Data access framework overview

The datasets are used to define and store data tables and rows. These tables reflect the actual tables in the database and can be fully or partially loaded by the framework.
The data access layers contain all source code handling the connection between the system and the database. Most common is the definition of the queries. The data access layers are the only parts of the system containing query definitions. An advantage of this restriction is the ability to relatively easily change the kind of database the system is using for data storage. In this case, the queries are the only part to change because of the slight differences between structured query languages.
For each supported kind of database, a database adapter must be available. These adapters translate the database types to the C# types and make use of the correct database connectors.

5.2 Transactions

Data consistency is a key subject in most systems using a database for information storage. One possible tool to obtain a higher data consistency is to make use of database transactions.
Database transactions are atomic operations executed on a database. This means that a series of operations either all occur, or all do not occur. With the use of transactions it is therefore possible to guarantee that some series of operations to perform on a database will never occur partially.

The data access framework supports the use of transactions. Throughout the complete system implementing the framework, transactions are available in a generic approach. At any point, a new transaction can be started or committed and in the framework this transaction is called a scope-transaction.
The generic part is the use of sub-transactions. The framework remembers the current scope-transactions for each thread and keeps them alive. See Code snippet 1 for the details of the creation of a new scope-transaction. If a scope-transaction is started while there is another scope-transaction being alive for that thread, the new scope-transaction is set to be a sub-transaction. In this case the actual transaction used for the connection to the database is the transaction set in the main scope-transaction started before in the current thread.
The lookup for an existing scope-transaction uses a static dictionary to store the scope-transactions for each thread with the thread itself as key.

```
public ScopeTransaction()
{
    if (transactions.ContainsKey(Thread.CurrentThread))
    {
        subTransaction = true;
        return;
    }
    AbstractDatabase.DatabaseType databaseType =
        (AbstractDatabase.DatabaseType)new Properties.Settings().DatabaseType;
    connection = AbstractDatabase.NewConnection(databaseType);
    connection.Open();
    transaction = connection.BeginTransaction(IsolationLevel.ReadUncommitted);
    transactions.Add(Thread.CurrentThread, this);
}
```

Code snippet 1: Creation of new ScopeTransaction

The difference between a normal scope-transaction and a sub-transaction is that the commit of a sub-transaction does not immediately affect the data in the database. The changes committed while running a sub-transaction truly affect the data in the database at the moment the main scope-transaction is committed.

```
public void Commit()
{
    if (subTransaction)
    {
        subCommitted = true;
        return;
    }
    transaction.Commit();
    transaction = null;
}
```

Code snippet 2: Commit a transaction

In Code snippet 2 it becomes clear that if the scope-transaction is committed but is part of another scope-transaction (i.e. *subTransaction* is set to true), the transaction is not committed yet. The moment the scope-transaction without a parent scope-transaction is committed the transaction is really committed to the database.
ScopeTransaction implements the IDisposable interface and when handling the disposal of the *ScopeTransaction* it will be removed from the static dictionary.

5.3 Entities

Working with Visual Studio gives the possibility to make use of datasets. As mentioned earlier, these datasets are used by the data access framework to reflect the tables and their relations in the database.
In the back-office system, several data entities were defined. All these entities exist in one single library and all modules of the system reference this library. This way, entities are to be defined only once and small database changes will only affect this library. Another advantage, which will become clear in the application framework chapter, is the common knowledge of the same datasets between the modules. Communication between modules can carry information about datasets as long as the modules both know the definition.

Entities can consist of only one table without any relations to other tables, or consist of multiple tables with many relations between these tables. For example, some *orders* entity defines two tables: *orders* and *order products* (Figure 7). There is a '*one on n*' relation between these tables and this relation is set to have cascading influence on the rows in the tables. This means that autonumber fields are automatically handled even when inserting *orders* and *order products* at the same time, but also the deletion of *orders* will immediately cause the corresponding *order products* to be deleted as well.

Figure 7: Order - order product relation

Processing data can be done very fast because all changes affect in-memory datasets only. Multiple deletions, additions and changes to the datasets can be made and afterwards stored in the database with just one single transaction.

The design of entities is one of the first steps to take when building a system for an e-business driven organisation. Business scenarios are proposed at the design period of a new system [Keen '04]. They describe the processes happening in the organisation. With these processes, business objects can be identified and modelled, which will result in appropriate entities. A process needs some kind of data to use or modify and entities should reflect this relevant data used in the process.
Designing well structured entities will empower the application to be adjusted to the unprecedented rate of change of the e-commerce business field and creates opportunities for an agile system. With the definition of correct boundaries or overlaps between the entities (although these overlaps should be minimized) the opportunities for creating stand-alone or loosely coupled pieces of code are well preserved.

5.4 Concurrency

The Store.nl back-office system is designed to aid the full staff in handling order-, customer- and product-related data. As a result, different people can be working with the same data at the same time. Therefore, maintaining consistent data becomes a big issue and handling concurrency becomes a high priority.
A common used approach to avoid data inconsistencies is to add a lock to one or more records in the database. A lock makes sure no other writes or reads can be done to the locked records. However, for the Store.nl back-office system, this is not a feasible option. Visitors on the website must be able to view information that they have requested and can not be treated with a "sorry this information is currently locked" message. Therefore a less restrictive way of maintaining consistent data is needed.
The data access framework is able to handle multiple read and write requests without losing data consistency. To avoid unnecessary locks for the online customers and users of the system, the framework maintains the isolation level "read uncommitted". This means that a dirty read is possible, meaning that no shared locks are issued and no exclusive locks are honoured. It also means that two different users can start to edit the same record(s) in the database. Before updating new data to the database, the framework makes sure, that the actual data in the database is the same as the data originally gathered by the user. When the data is changed by some other user in the mean time, the system will throw a database concurrency exception and denies the update to the database.
The obvious disadvantage of this kind of concurrency handling is the risk for the user of the system to start over updating some data after getting a concurrency exception. However, the advantages of maintaining consistent data and no record locks, is more important than this disadvantage.

5.5 Results

As mentioned before, the data access framework is designed to support several kinds of databases. In theory, any structured query language is supported as long as there is an available connector library. Currently, three different databases are tested and show positive results: a MySQL database, an MS Access database and an MS SQL database.

The use of generic transactions works particularly well in a highly database-driven system like the Store.nl back-office system. Many parts of the system use transactions for their responsible part of the system. There are many cases in which a combination of two or more of these transactions must be committed to the database. If one of the transactions fails, none of them should be committed and this is undoubtedly realised by the framework with the use of sub-transactions. The next chapter will elaborate further about this property of the data access framework when discussing the use of services.
Important to mention is the fact that the Store.nl back-office makes use of only one database. This means that there is no need for a distributed transaction coordinator. The

transactions are therefore completely controlled by the data access framework and sub-transactions can be handled properly.

The use of datasets to work in-memory with data from the database results into few unnecessary database queries. Because data added, edited or deleted in the back-office system is not immediately updated to the database but kept in memory in datasets instead, the queries are minimized. If a user of the back-office system is busy editing multiple records and finally decides to put back the original data, no update queries are required into the database.

6 Application framework

6.1 Introduction

Together with the data access framework, the application framework is one of the most important building blocks of the Store.nl back-office system. The application framework gives the possibility to create a main application. Mutually, in a main application several modules can be added to the framework and can be used in the main application. Within the application framework, communication between the different modules and main application in the back-office system is made possible by means of sending text messages through a broker. The broker makes use of a service repository storing all available services in the modules.
Additional features the application framework provides are main forms and settings forms. Main forms can get a special treatment in the main application by setting them directly visible in the main application. Settings forms from all modules can be collected and added to a single settings compilation.

The remainder of this chapter describes the technical details of implementing the framework with a main application or modules and the possibilities using the mainforms, settings and services. Finally, the actual implementation of the framework for the Store.nl back-office will be discussed.

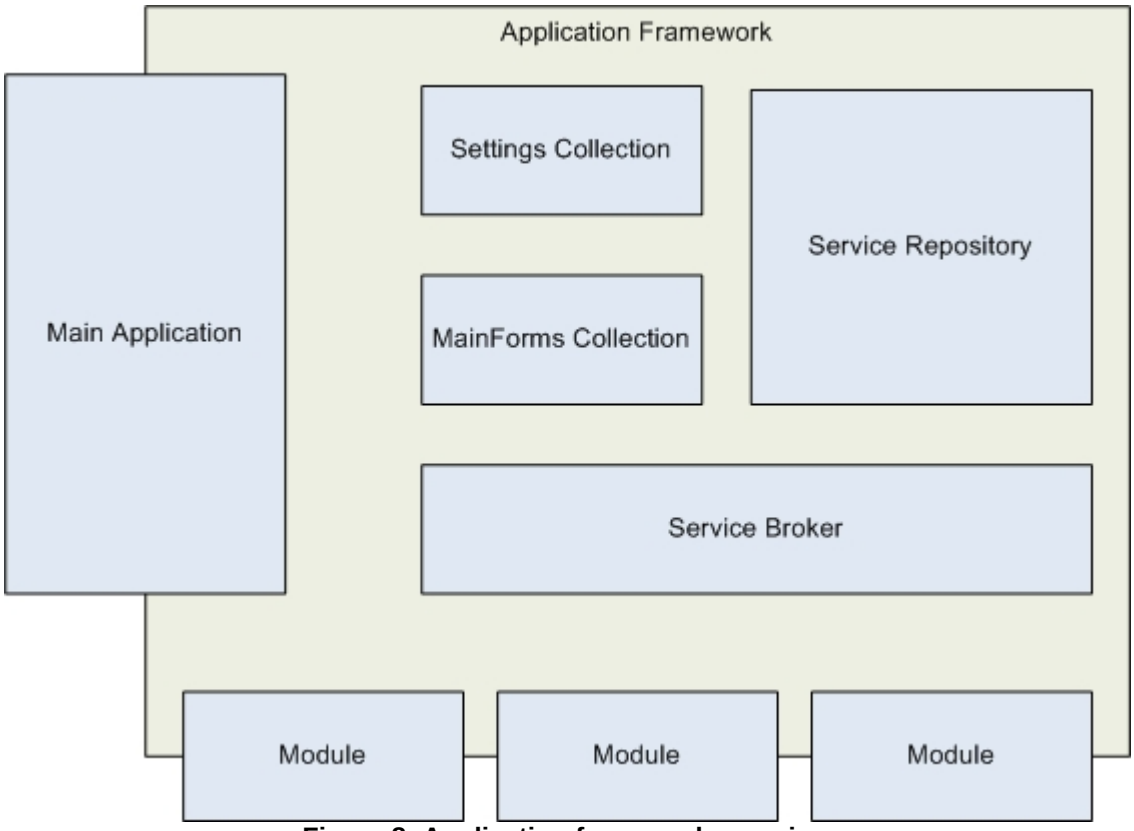


Figure 8: Application framework overview

6.2 Main components overview

Figure 8 shows the main components of the application framework. The components completely inside the area of the application framework box are the actual parts of the framework. The settings collection, mainforms collection and service repository are collections of objects which can be used by the main application or modules implementing the framework. The service broker is the means to access these collections.
The boxes half inside the application framework area are the parts of the system implementing the framework. Modules implementing the framework are stand-alone libraries and have access to other modules implementing the framework through the broker. The main application is the part of the system which actually starts when running the system.

6.3 Store.nl back-office main application

To create a new system with the application framework, the framework must be implemented by some main windows application. The Store.nl back-office system has one main window which has several views for the included modules. Examples are adding all main modules to tab pages and a list of modules to open in a new window when double clicked. Figure 9 shows two screenshots of the Store.nl back-office main application window. The first figure is an example of all main forms collected and added to a tab control. The second adds them all to a list on the left side of the main window.

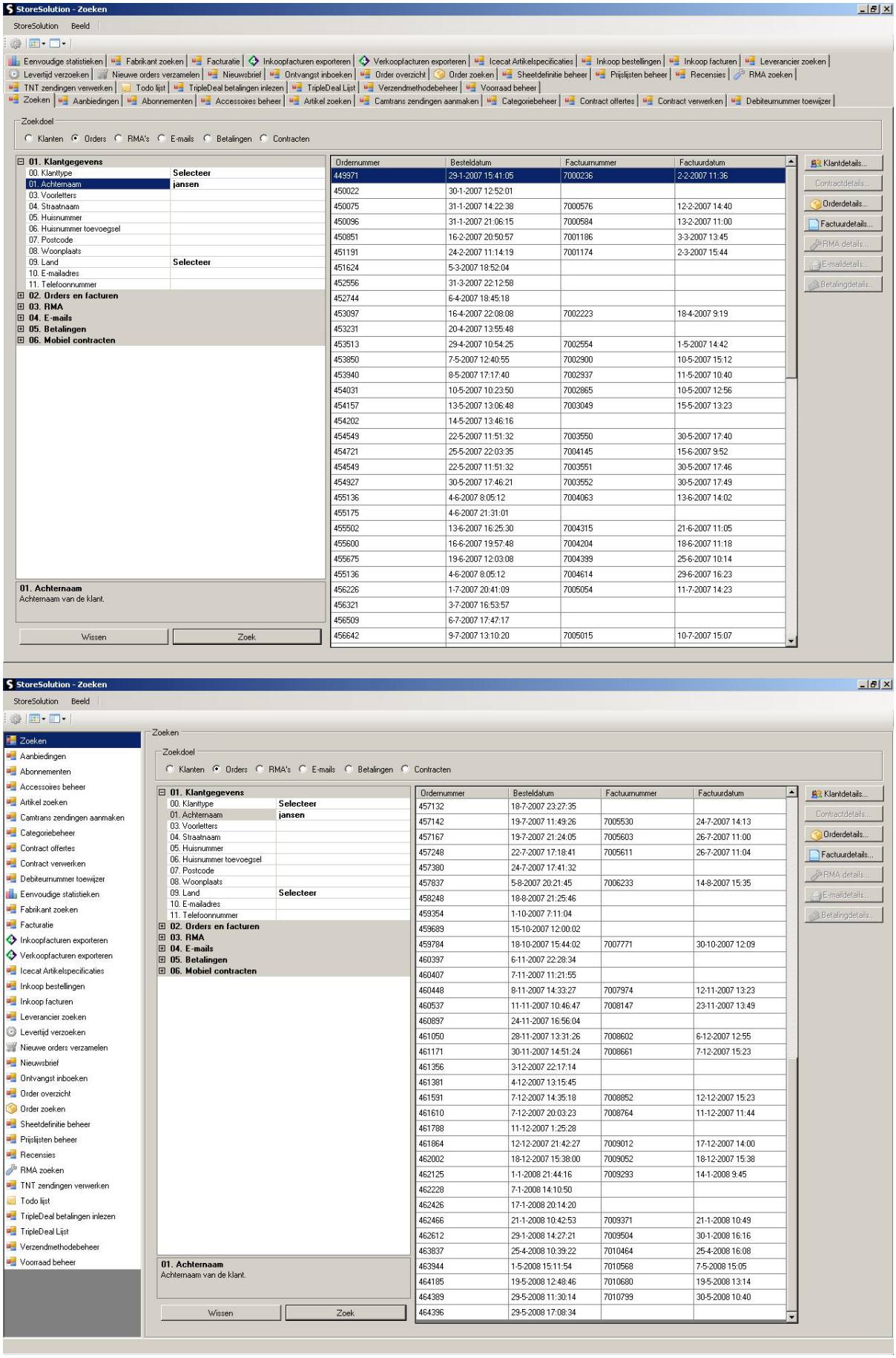


Figure 9: Store.nl back-office main application examples

6.4 Loading modules

With a main application implementing the framework it is possible to load modules into the application. While loading the module assemblies, the framework analyses all available attributes in the assembly and adds known attributes to a repository. In Visual Studio external assembly files can be loaded into an *Assembly* class. Adding external functionality to a program giving possibilities to analyze and modify the structure or behaviour of the complete program at runtime is called reflection. The *Type* enumerator on the *Assembly* class allows us to get all available types in the assembly which includes classes. If the type is a class, available constructors, methods and properties can be obtained. If there are custom attributes defined, these are on their turn available within the constructors, methods and properties. This final step brings the framework to analyzing the attributes and registers the needed information into the service repository.
Currently, there are three possible attributes to define for the application framework: *MainForm* attributes, *SettingsForm* attributes and *Service* attributes.

MainForm attributes can be used to define the main windows of the application. These kind of windows will be shown by the application (in the way the main application implemented the use of *MainForms*) when it has started. Figure 9 shows the *MainForms* in the tab pages and list. A main application should at least have one *MainForm* in order to be able to make use of the modules. Without a *MainForm*, other windows cannot be reached.
A *SettingsForm* can be defined to make some kind of (saveable) settings available in a module. *SettingsForms* are collected by the application framework and made available in one universal settings form in the main application. Typical settings are printer or log file settings.
The last possible attribute is the *Service* attribute. Constructors, methods and properties in the modules can be given this service attribute. Each constructor, method and property with this attribute will be collected by the framework and made available to all other modules.

6.5 Services

While the main application implementing the framework loads all available modules into the application, the framework searches available services in the modules by analyzing the attributes and service interface descriptions in the modules. Services can be constructors, returning the complete object they are constructing, methods, returning nothing or some object and properties, returning themselves. The parameters defined in the service interface descriptions which need to be sent to the constructors and methods are available in the framework and must match while using the services in other modules.

A main advantage of the use of services is the properties of services that come with them. Modules using services from the service repository don't have to worry about implementation or dependencies of the information the service is sharing. With the loosely coupled property the use of a service requires only the knowledge to invoke the service together with the information about the expected return type.
Services can be reused in many modules. This avoids code duplication and saves programming time. If changes have to be made to a particular part of the system, services only have to worry about their service contract and not about the implementation. With many different copies of code to be modified, it is inevitable to monitor and test all dependencies in the rest of the system to keep the system running properly. Obviously, this would take much more time. In the Store.nl back-office system there are many examples of avoiding code duplication. Examples are the calculation of a certain product purchase price according to several discounts and cancelling an order with products at a supplier, taking into account possible pre-reservations for customers

and possible stock depletion predictions. In practice, it often happens that a certain action must be performed at multiple places in the system. This way of exploiting services makes the system more flexible and the quality of the system can be kept at a high level.

The framework only supports synchronous service communication. The result is that the communication in the back-office system has the following characteristics [Papazoglou '07]:

- A module invoking a service requires an immediate response
- The module and service work in a back-and-forth conversational way

The main application with its modules is running on the same Windows computer and synchronous service communication is sufficient with respect to speed. However, in the future, asynchronous communication should be added to offer new possibilities in the service interface descriptions. This could become convenient when multiple computers are going to work with the same services over a network.

6.6 Composite services

Composite services can be more complex and coarse grained than single stand-alone services. The communication between the services requires more error handling. Nonetheless, the use of composite services appears to be of great use in the Store.nl back-office system.
Services interacting with the Store.nl database can exploit an important characteristic of the data access framework. The availability of sub-transactions is particularly valuable with the use of composite services. The following example illustrates the use of a combination of two transactional services interacting with the database.

Imagine a service updating the available stock of a product. This service requires a product id, a location id and the amount of products to add to the stock as input. Inside this service, two data tables in the database are updated. The actual stock data table is updated to transit the updates about the amount of available products in the system. The other data table is the product table, to update a status flag about stock information. Both updates are committed in one transaction. Now imagine another service taking care of registering newly received products from some supplier. Like the previous service, it requires a product id, location id and an amount of products received. Additionally, it also needs an id from a purchase order the products were ordered with. Within this service, several tables in the database are affected. The purchase order table must register the received products and update some status, but also new records must be added to register the actual (physical) products and their receipts. Again, all tables are updated by the service within one transaction.
Now, using both services described above, it is possible to register new products in the Store.nl back-office system. Obviously, both services must succeed before the changes may be made to database. Therefore, another service is created. Starting a new transaction and calling both services within, the transactions created by the services will be demoted by the database framework to sub-transactions. Without the knowledge of the services, both of the services will commit their changes to the database within the composite services' transaction. Would one sub-transaction fail, both will be rolled back and no updates are made to the database. Only if both sub-transactions succeed the changes are committed to the database.

6.7 Service broker

Within the application framework, multiple modules can be added to a single main application. One of the properties of the modules is the ability to offer services to other

modules. As stated previously, the communication between the services is regulated by a service broker. The service broker is part of the application framework and is available to all modules implementing the framework.
The broker used in the application framework is designed around a message based conversation architecture. Four possible actions are supported by the broker:

- Construct a new service object
- Invoke a service object
- Set a property on a service object
- Get a property from a service object

6.7.1 Construct a new service object

Available services like *mainForms*, *settings* and *services* can be accessed through the broker by identifying the service package, the service name, the service constructor and optional service parameters. With these identifiers, a new *AFServiceObject* is created by the broker and returned to the sender. This *AFServiceObject* contains information about the location which can be used when invoking the object by the broker.

6.7.2 Invoke a service object

To invoke a service object an *AFServiceObject* is needed along with a method name and optionally any parameters. The named method is then invoked by the service broker and an object is returned to the service requester.

6.7.3 Set a property on a service object

To set a property on a service object an *AFServiceObject* is needed along with a property name and a value to set the property with. The value must match the target type to succeed.

6.7.4 Get a property from a service object

To get a property from a service object an *AFServiceObject* is needed along with a property name. The service broker will retrieve the property from the service object and returns the retrieved value as an object.

6.8 Example of framework usage

To give a better overview of the application framework, an example usage of the framework will be discussed. All basic implementations and steps to take in order to create an application with modules will be shown. For the example some of the existing modules from the Store.nl back-office system are used. The main application example is the Store.nl back-office main application.

6.8.1 Implementing a main application

For each usage of the application framework, a main application must be created. The main application is a normal windows project in the Visual Studio environment. The application framework must be referenced to the project to make use of its functionality.
Loading available modules and parsing all available services and forms need to be initialised by the main application. Code snippet 3 gives an example of how to load modules (DLL and EXE files) into the framework. The most important line is *AFLoader.AddAssembly(file);* This triggers the actual loading of the module.

```
private void AssemblyThread(object o)
{
    // Retrieve the module directory.
    string moduleDirectory = (string)o;

    // Retrieve all dll files in the module directory.
    string[] exeFiles = Directory.GetFiles(moduleDirectory, "*.exe");
    string[] dllFiles = Directory.GetFiles(moduleDirectory + ModulePath + "\\", "*.dll");
    string[] files = new string[exeFiles.Length + dllFiles.Length];
    exeFiles.CopyTo(files, 0);
    dllFiles.CopyTo(files, exeFiles.Length);

    // Initialise and show the ProgressForm.
    progressForm.SetText("Modules laden");
    progressForm.SetMinimum(0);
    progressForm.SetValue(0);
    progressForm.SetMaximum(files.Length);

    // Load the assemblies from their files.
    foreach (string file in files)
    {
        progressForm.SetMessage("Bezig met het laden van module: " + Path.GetFileName(file));
        try
        {
            AFLoader.AddAssembly(file);
        }
        catch (Exception exception)
        {
            ExceptionMessageBox.Show("Er is een fout opgetreden tijdens het laden van de module:\n
        }
        progressForm.SetValue(progressForm.GetValue() + 1);
    }

    // Signal that the assembly thread is about to end.
    AssemblyThreadEnd(progressForm);
}
```

Code snippet 3: Loading assemblies into the application framework service repository

After loading the modules into the framework, it is necessary to give a representation of the *MainForms*. These objects are loaded from the modules together with the services and are available to the main application. Code snippet 4 shows how the *MainForms* are initialized by the framework. An extra event handler is set to each main form to be able to intercept termination of the forms. This way it is possible to cancel the termination of the whole system at the main application level.

```
private void InitialiseMainForms()
{
    // Create the MainForms.
    this.mainForms = AFMainForms.CreateInstances();

    // Set an event handler for every MainForm.
    foreach (Form mainForm in this.mainForms)
        mainForm.FormClosing += new FormClosingEventHandler(MainForm_FormClosing_Intercept);
}
```

Code snippet 4: Initialize *MainForms* from the application framework

With the *MainForms* loaded into an array in the main application, one possible way of showing the forms is the use of a tab control. Each main form is displayed on a different tab page. Code snippet 5 shows the way this is done in the Store.nl back-office main application.

The main tab control (tcMain) has been added to the main application's main window in design mode. After adding the forms to the tab control some extra actions on the forms are performed. All menu strips, status strips and tool strips are extracted from the *MainForms* and added to the main application. The relevant strips are added to and are shown in the main application when a tab page gets the focus.

```
public void SetMainForms(Form[] mainForms)
{
    foreach (Form mainForm in mainForms)
    {
        mainForm.TopLevel = false;
        mainForm.Dock = DockStyle.Fill;
        mainForm.FormBorderStyle = FormBorderStyle.None;

        // Create a new tabpage and add the MainForm to the tabpage.
        TabPage tabPage = new TabPage(mainForm.Text);
        tabPage.Tag = mainForm;
        tabPage.Controls.Add(mainForm);

        ilTabPages.Images.Add(mainForm.Icon);
        tabPage.ImageIndex = ilTabPages.Images.Count - 1;

        //mainForm.Show();

        // Add the new page to the tab control.
        tcMain.TabPages.Add(tabPage);
    }

    ExtractStrips();
    HideStrips();
    ModifyMainForms();

    UpdateSelectedMainForm();
}
```

Code snippet 5: Add a *MainForms* collection into a Tab Control

6.8.2 Framework windows

Standard functionality available in the framework is the offer of two different dialog windows. One of them gives an overview of all loaded modules and their services including the types of parameters expected and the return types (see Figure 10). The other window is the settings window (Figure 11). This window gives the opportunity to group all *SettingsForms* into one collection of settings.
Initially, this window contains no functionality and is an empty window. When loading the modules into the framework, the *SettingsForms* are extracted from the modules and added to this window. Each *SettingsForm* is added onto the list on the left side of the window and its contents are added on the right side of the window.

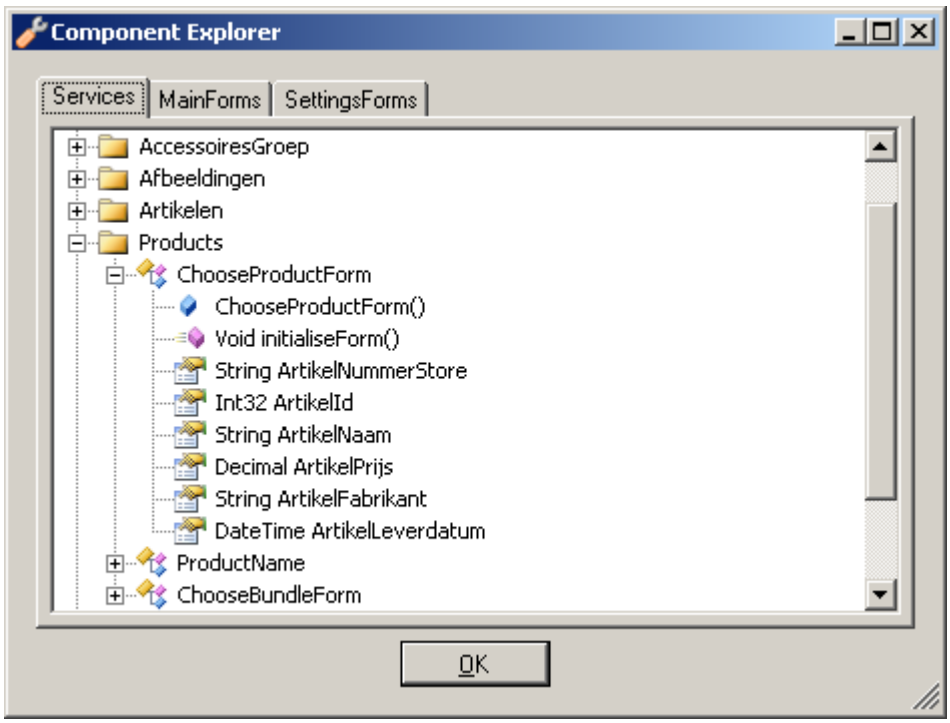


Figure 10: Application framework component explorer example

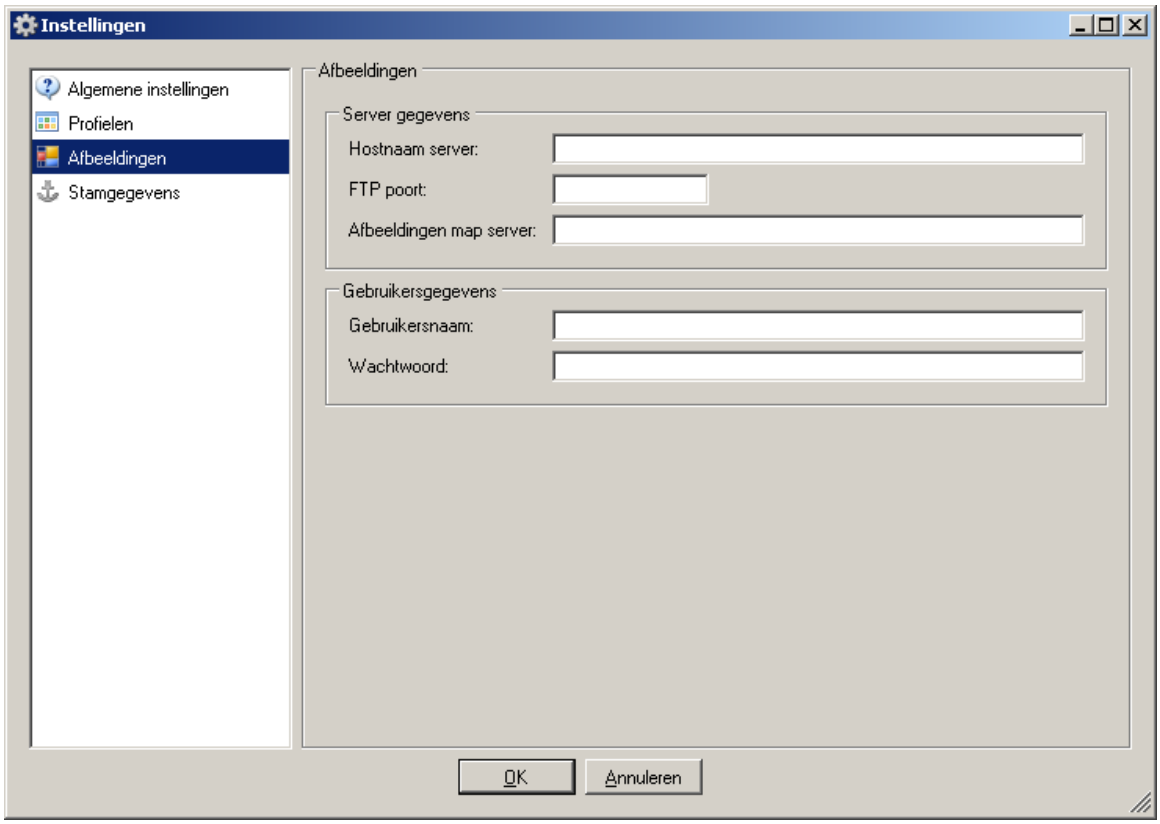


Figure 11: Application framework settings form example

6.8.3 Implementing a module

Just like implementing a main application, implementing a module requires starting a new windows project in the Visual Studio programming environment. The application framework library must be referenced. After finishing the module, it must be built and added to the modules directory referenced by a main application.

Marking a module as a main form needs a trivial action. Code snippet 6 shows an example of the attribute to be added above the declaration of the class. The application framework will use the default empty constructor to create an instance of the form when needed.

```
[AFMainForm("ArtikelZoeken", "Biedt functionaliteit om artikelen te zoeken.")]
public partial class ArtikelZoekenForm : Form
{
```

Code snippet 6: Mark a form as an application framework *MainForm*

Within a module, the declaration of services is shown in Code snippet 7. In this case there exists an *ArtikelService* object. Constructors need to be defined explicitly because multiple constructors can exist. Services that should be available within the framework are also addressed with a special application framework attribute.

```
namespace Artikelen.Services
{
    [AFServiceClass("Artikelen","ArtikelService")]
    public class ArtikelService
    {

        [AFServiceConstructor]
        public ArtikelService()
        {
        }

        [AFServiceMethod]
        public int GetSupplierId(int productId)
        {
```

Code snippet 7: Mark an object as a service class

6.8.4 Using a service

Now we've seen how we can mark some object's services and constructors as services within the application framework. The last part needed to make full use of the application framework's possibilities is to use a service in another module using the application framework's service broker.
The broker in the example showed in Code snippet 8 is used to construct a new instance of service which is actually a form. Next, a method is invoked by the broker on the service object and finally the service object is after a cast to a form object shown to the user.

```
private void ArtikelDetails(int? id)
{
    this.Cursor = Cursors.WaitCursor;

    try
    {
        AFServiceObject artikelDetailForm = AFBroker.Construct("Artikelen", "ArtikelDetailsForm", "ArtikelDetailsForm");
        AFBroker.Invoke(artikelDetailForm, "initialiseForm", id);
        ((Form)artikelDetailForm.Instance).Show();
    }
    catch (Exception exception)
    {
        ExceptionMessageBox.Show("Er is een fout opgetreden tijdens het openen van het artikeldetails scherm.", Text, exc
    }

    this.Cursor = Cursors.Default;
}
```

Code snippet 8: Use the application framework service broker to invoke a service

6.9 Results

The use of the service broker can result in a very clear and ordered way of storing services. Each service can be put into a package and the naming of services can be done without adhering to the actual object name the service is representing. This is not the only advantage. A property of the broker is that it provides late binding and different modules only bind on runtime in stead of on design time. Therefore, no dependencies are required between the different modules in the programming environment. A disadvantage is that the compiler cannot check for any errors made in calling the services.
Without the references between all modules in the programming environment, the system can scale without dropping speed and increasing memory usage while developing. Multiple developers can work on different modules without any need for source merging. Since all layers (GUI layer, data access layer, business logic layer) used in a module are split within the modules themselves, these layers don't overlap with other modules either.

The application framework provides the ability to save many database queries by caching data into a service. This is a property being used in the Store.nl back-office: a single service contains datasets each filled with the complete contents of the corresponding table in the database. When the information from these datasets is used for the first time in any module, the data is loaded from the database and kept into memory for as long as the application is running. Example tables are: categories, manufacturers, shipping methods, payment methods and many more. All these tables are not likely to change much. Only sporadic these tables are edited and updated. Within the application framework, a settings service is made available to refresh the data on demand if necessary and of course when the application is restarted all datasets are flushed to be read again from the database.

7 Qualitative evaluation

7.1 Introduction

This chapter describes the evaluation of the Store.nl back-office system by analyzing key requirements stated in the beginning of this thesis.

The main goals described in the introduction chapter of this thesis stated that the back-office system should be flexible and must support changing requirements and business strategies. Maintainability is therefore an important requirement. If the system has a high maintainability level, it is possible to update and change it rather easy. With a low maintainability level, changes and updates are hard to perform which is certainly not a desired situation.
A similar main goal is to create the ability to act fast in a fast changing business area. New subsystems must be added when the need rises and must be integrated in the back-office system. Many subsystems are a derivative or have common needs of other parts of the system and with the availability of services reuse becomes a topic to investigate. Therefore, the Store.nl back-office system's reusability level is also addressed in this chapter.

7.2 Maintainability

Given the main goals of the Store.nl back-office system, a measurement of in what ways the system adheres to these goals is necessary. To measure the maintainability level of the system the following properties are investigated:

- Coupling with other parts of the system
- Complexity of the source code
- Size of the system
- Algorithms and data structures used in the system
- Programming language used to develop the system
- Documentation

7.2.1 Coupling with other parts of the system

The Store.nl back-office system is a modular system. Although different modules can communicate and connect to each other, they are independent subsystems. Nonetheless, the functionality of the modules could be limited when other connected modules are not available. Considering this the Store.nl back-office system has a medium coupling level.

7.2.2 Complexity of the source code

Throughout the whole back-office system, the source code in the modules is strictly separated into several layers. The presentation (or GUI) layer contains code related to events and actions on the back-office system windows. This code is very straightforward and easy to read. The next layer is the business logic layer (BLL) and contains code for calculations, (simple) algorithms and data manipulation. This layer is the most complex layer and therefore important to investigate in order to find out the complexity of the complete system. The last layer, the data access layer (DAL) contains all queries and all communication protocols with the database. Like the presentation layer, the code in this layer is straightforward. The methods are short and are usually limited to query definitions and communication with the data access framework.

For each entity defined in the system a separate DAL and BLL exist. The functionality of these layers is explicitly focused on the data defined in the relevant entity. By means of this extra separation the source code becomes easier to understand and less complex. Overall, the Store.nl back-office system has complex pieces of source code but is strictly separated into several layers. The readability level of the source code is high and the total complexity of the source code can be called medium.

7.2.3 Size of the system

The size of the complete Store.nl back-office system is rather large. A simple count of the lines of code gives a result of approximately 430.000 lines, comments excluded. However, this does include automatic generated code for presentation forms and datasets. Given this size of the system it is fair to conclude that this is not good for the maintainability of the system. However, the modules of the system can be maintained without knowledge of other parts of the system. The size of a separate module becomes more important than the size of the whole system. Modules have around 15.000 lines of code each on average. Again this includes comments and automatic generated code. Average modules in the Store.nl back-office system will not restrict the maintainability level of the complete system.

7.2.4 Algorithms and data structures used in the system

The most common complex data structure used in the Store.nl back-office system is the dataset. These datasets contain reflections of the database tables and relations between them. Most of the functionality in the back-office system consists of data manipulation. Pure algorithms are not present in the system. However, an example of what could be seen as a form of algorithm is the calculation of invoice sequences. This calculation is done in the invoice module and contains a very comprehensive structure. Many variables are considered when determining in which order customer orders have to be invoiced. Besides the invoice-algorithm and the datasets there are no complex algorithms and data structures used in the Store.nl back-office system and the maintainability level of the system will not decrease because of this property.

7.2.5 Programming language used to develop the system

The programming language used to develop the Store.nl back-office system is C#. The language is widely used and has a lot in common with another very popular programming language: Java. There is much documentation and there are many examples about C# in books, e-books and websites.
The language used for the Store.nl back-office system will contribute to the maintainability of the system.

7.2.6 Documentation

Documentation is a very important aspect when maintaining a system. However, the documentation for the source code of the Store.nl back-office system is mediocre. Important design decisions are documented in high detail, but, other straightforward methods and classes lack any documentation.
Next to the documentation directly in the source code several documents should exist like a technical and functional design. Unfortunately, the amount of documentation for the Store.nl back-office system is rather low. Some modules have functional and technical documentation, but the larger part hasn't. In contrast, a very well maintained database design including all fields and relations exists. This design is included in the appendix of this thesis. The documentation can be a great limitation to the maintainability level of the complete Store.nl back-office system.

With the above properties considered it is now possible to determine the level of maintainability of the Store.nl back-office system. We have seen properties contributing

to the maintainability level but also properties against a high maintainability level. Table 4 summarizes them.

Property	High	Medium	Low
Coupling		X	
Complexity of source code		X	
System size	X		
Algorithms and data structures	X		
Programming language	X		
Documentation			X

Table 4: Overview maintainability properties Store.nl back-office

With almost all properties contributing mediocre or much, it is safe to conclude that the Store.nl back-office system is maintainable.

7.3 Reusability

Like with investigating the maintainability level of the system, several properties have to be taken a look at in order to decide whether or not the Store.nl back-office system has a high reusability level. The properties taken into account are:

- Modularity
- Program understanding
- Programming style guidelines
- Coupling with other parts of the system
- Documentation
- Self descriptiveness
- Good exception handling
- Information hiding
- Appropriate use of packaging
- Few external dependencies

Maintainability and reusability have many properties in common. Some properties listed above are also part of the maintainability level determination. This is the case with documentation and coupling with other parts of the system and partly with program understanding (this is close to complexity of the source code). The remainder of the properties will be further elaborated.

7.3.1 Modularity

Being a modular system is probably the most important property for having a high level of reusability in a system. Since the Store.nl back-office system makes use of the e-commerce back-office framework, the complete system is modular in principle. Each module existing in the Store.nl back-office system is an independent project without any references to other modules. Data exchange with other modules is limited to communication through the service broker in the application framework.
Modules can be added and removed from the system on demand and could also be added to other applications implementing the framework. The modularity of the Store.nl back-office is very high and will add a high value to the reusability of the system.

7.3.2 Programming style guidelines

Programming styles are important to create consistent code. When reuse comes into view, this helps understanding the source code. Programming in visual studio allows the programmer to enforce source code formatting styles. Indentation, the use of white

space and alignment can be set and the editor will enforce these settings when typing source code.
Another style which is harder to enforce is the use of lowerCamelCase. In the Store.nl back-office system this type of casing is being used. This means that compound words in variables are capitalized with exception of the first word. Together with the casing, the variables should always have a clear description about their meaning and type (e.g. customerAddressDataSet).
The appropriate and consistent use of programming styles will increase the Store.nl back-office system's readability and therefore its reusability.

7.3.3 Self descriptiveness

Self descriptiveness is very closely related to understandability and makes it easier to understand a system. Comments in the source code or well chosen class and method names will raise the ease to understand a system. Within the Store.nl back-office system we tried to add comments to important design and implementation decisions. The code is rather self descriptive.

7.3.4 Exception handling

The Store.nl back-office system has an advanced exception handling structure. On multiple levels, exceptions can be thrown and handled accordantly. In case no possible automatic 'fix' is possible, the user gets an on screen message showing the kind of error which occurred. Behind the scenes however, the exception description, stack trace and user/computer information is stored to a local database. This way, developers can track possible programming flaws after publishing a module. A separate utility allows grouping the errors for example by username, computer name or kind of exception. With this system, errors can be observed and will add value to the reusability of the system.

7.3.5 Information hiding

The application framework the Store.nl back-office system is implementing creates opportunities for information hiding through the service broker. The service interfaces in the modules clearly define the operations and data available for other components and modules. When some module or part of module is needed elsewhere, in another module for example, the service interface is the only information required. The implementation is completely hidden for the developer reusing the module's functionality. An additional advantage of this approach is the ability to change the implementation of the service without affecting other parts of the system.
In the Store.nl back-office system many service constructions are used especially between modules (although the services can be used within a module as well). The information hiding in the system can be called good and adds value to the overall reusability level of the system.

7.3.6 Packaging

Another property of the application framework is the ability to define packages to create some kind of hierarchy into (the functionality of) the services (see the Application framework chapter for more details on packaging). Packages create a better overview of the functionality of the complete system and create opportunities for developers to make use of existing functionality of the system. The reusability level rises when making appropriate use of packaging.

7.3.7 External dependencies

When an application has a lot of dependencies on other external libraries the reuse of such an application requires the use of these external libraries as well. This is not good for the reusability level of a system. Within the Store.nl back-office system, only a few

external dependencies exist. Examples are libraries for communication with a MySQL database and communication with MS Office documents. However, these libraries used in the system are all available for free and easy to use. The complete system's reusability level will not lower because of external dependencies.

Property	High	Medium	Low
Modularity	X		
Programming understanding		X	
Programming style guidelines	X		
Coupling		X	
Self descriptiveness		X	
Exception handling	X		
Information hiding	X		
Documentation			X
Packaging	X		
External dependencies		X	

Table 5: Overview reusability properties Store.nl back-office

Taking all properties into account it can concluded that the reusability level of the Store.nl back-office system is quite high. Table 5 shows an overview of the properties and their amount of positive influence on the reusability level of the complete system.

8 Other back-office systems

8.1 Introduction

This chapter will focus on a comparison between the Store.nl back-office system and other back-office systems.

To get a better idea of the possibilities given by the Store.nl back-office system and assisting e-commerce back-office framework, other back-office systems are investigated and compared with the Store.nl back-office system. Two packages containing similar functionality are considered. One of them is a commercial package called Office for Business (OFB) and the other is an open source package OSCommerce (OSC).
The metrics used for the comparison are the level of maintainability and the level of reusability. In chapter 7 these levels are discussed for the Store.nl back-office system. To compare the OFB and OSC systems to the Store.nl back-office system, this chapter first describes these levels for the compared systems. After that the complete overview and comparison is given.
Before defining the levels of maintainability and reusability, a short overview of the complete packages will be given to get a better understanding of the packages and the technologies used. Together, some notable properties of the systems will be compared to properties of the Store.nl back-office system.

8.2 Office for Business

Office for Business (OFB) is a commercial software package [OFB]. Like the Store.nl back-office system OFB is a back-office system offering functionality to aid in selling products. Their main audience is the retail trade business, especially computer retail business. The complete solution contains customer relationship management (CRM) modules, web shop modules, and back-office modules. Additional (add-on) modules like (extended) product management and accountancy program connectors are also available.

For this research a version published in 2004 is being used. All main components of the back-office modules were available together with the CRM modules. In addition, an extended product management module is being used. The web modules were not available yet.

8.2.1 Technology

The OFB back-office makes use of an MS Access database to store all information and runs on one central server. The programming language used is VBA which is the standard programming language when using MS Access.

8.2.2 Modules

The functionality in OFB is divided into six categories: general functionality, product and product configuration related, product sales, product purchase, management information and settings. See Figure 12 for the OFB main window displaying the categories in a tab-structure.

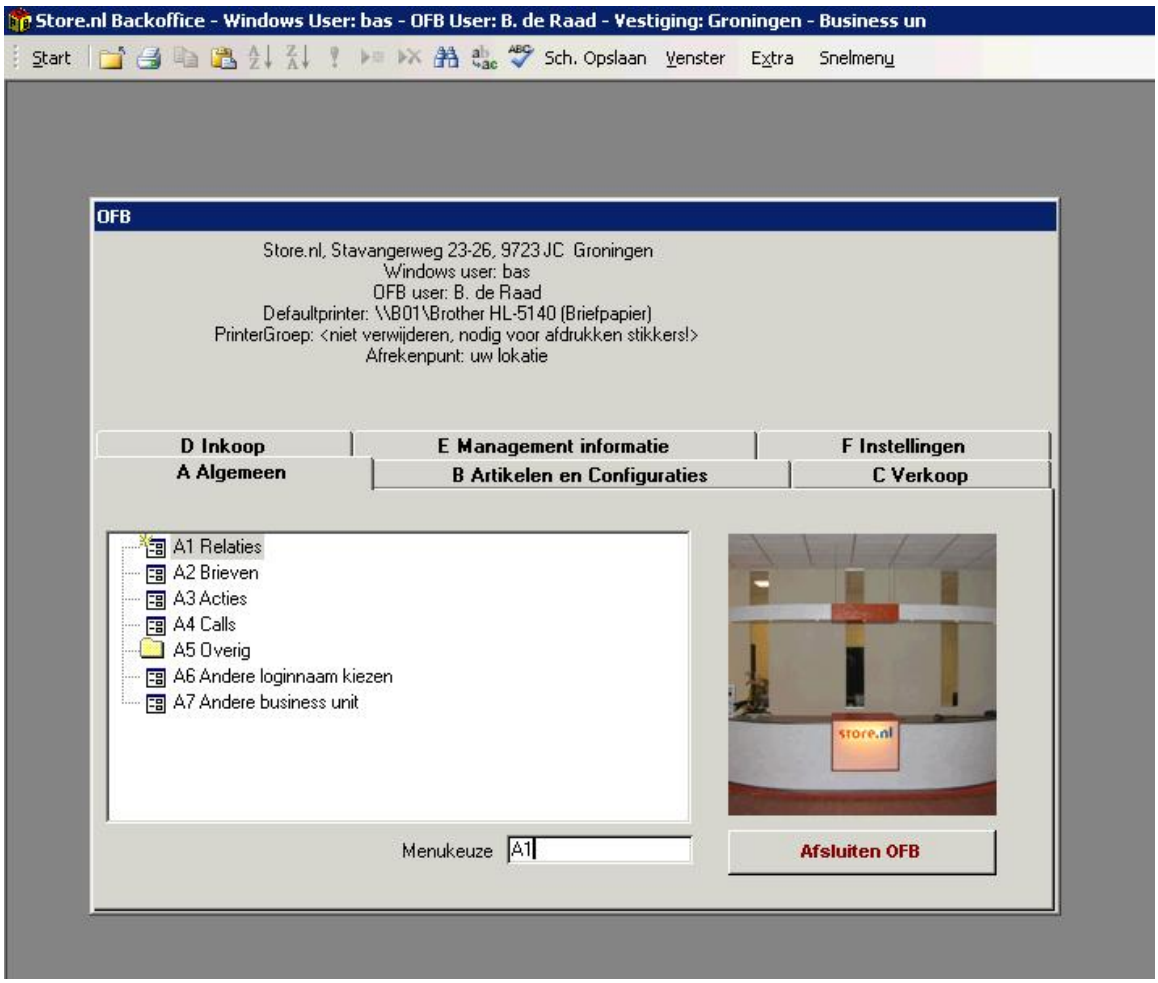


Figure 12: OFB main window

8.2.3 User accounts

Users of the OFB back-office system need to connect to a central server using a remote control program which was in this case the default remote desktop connection software coming with Windows Server 2003. This approach has serious shortcomings when the amount of users grows. The more users working with OFB, the faster the server must be and this has limitations. Windows Server 2003 is limited to 75 users and 4GB of memory (RAM). The average use of memory by the OFB back-office program is 50MB which will result in 3.75GB memory usage for 75 users. However, the operating system itself and to start a remote session requires memory as well. Considering these memory calculations it is reasonable to conclude that the amount of users is at least limited to 75 users but probably somewhere between 50 and 75 users.

Compared to the amount of users the Store.nl back-office system can handle this is rather low. In theory, the amount of users working with the Store.nl back-office system is unlimited. Each user has its own copy of the application running on their local computer. Memory and processing power will therefore not be a problem. The only overlapping part of the system is the central database which runs on a separate server. In case one database server is not able to handle the amount of transactions, clustering[1] multiple databases will tackle this problem.

[1] http://www.mysql.com/products/database/cluster/mysql-cluster-datasheet.pdf shows a promotional MySQL cluster datasheet.

Working with the OFB back-office requires much processing power, especially when working with large amounts of database records. The additional extended product management module provides the ability to read and process spread sheets with product information like prices and stock. Updating the database with information from large spread sheets takes several minutes to complete with a processor usage of 100 percent. A test running this action on three different user accounts resulted into a hardly responding server and other users were unable to use the back-office system.

On the Store.nl back-office system we can run the same test in the product and price sheet import module. This module has the same functionality as the product management module in the OFB back-office. The results are clear: multiple users can use this module at the same time without affecting other user's performances. Since the Store.nl back-office runs on the client's computer, all processing power needed comes from this client computer. Only at the final step when a user wants to save the modified data, some update queries are sent to the database. For the database server this is a trivial action to take considering the amounts of products Store.nl is working with.
One remark however, is that the data access framework maintains the isolation level "read uncommitted" (see chapter 5.4 for more details). This means that multiple users trying to update the same products will fail for all but one user. In practice, this is rarely occurring since there is no use in doing the same job twice at the same time.

8.2.4 Website synchronization

OFB has some support for connecting an online website to the local Access database, but just in a straightforward way. The information flow is one-way, i.e. from website to back-office.[1] The information must be sent by the website through e-mail with an attachment in XML format. Next, OFB offers the possibility to connect to MS Outlook to read the XML attachments into the database.
Because there is no support for exporting information from the OFB database to the website this must be implemented with some external synchronizer. Another possibility is copying all information by hand but this is a very unpleasant and time-consuming option.

In the Store.nl back-office system, this problem won't occur: both the website and back-office operate on the same database. Synchronization is therefore unnecessary. The advantages of this architecture are:

- No error-prone synchronization solution.
- Real-time order information available for back-office users.
- Real-time order and stock information available for customers on the website.

8.2.5 Maintainability

To give a clear comparison with the Store.nl back-office system's maintainability level a comparison of all investigated properties for this system and the maintainability properties of the OFB back-office system needs to be made. However, for some properties a very accurate description can not be given, since the OFB back-office system has no open source code. The missing information will not be added to the final conclusion in the comparison.

[1] This is based on a version of OFB published in 2004. Later versions offer web-module extensions which contain a two-way flow of information between back-office and website.

Coupling with other parts of the system
The total OFB system consists of a back-office system, add-ons for the back-office system and connection applications with other systems like a website and an administration application. The coupling with a website is an important connection in the total system and is intertwined with the back-office system. The back-office system will work without the connection, however, no automated data will be added to the database and the system will not be very useful. The add-ons are stand-alone applications and the back-office system will run properly without them. The add-ons can be maintained separately which is good for the maintainability level of the complete system. The back-office system (which is the largest part of the complete system) and its connection applications have to be maintained all together and this can lower the maintainability level of the complete system.

Complexity of the source code
There is no source code available to investigate the complexity. However, the fact that the system has been built using VBA in MS Access is familiar. The complete system consists of one single database file which includes all source code of the back-office system (with the exception of add-ons). Even if the source code is superbly classified internally, the complexity will still be rather high because there are no clear boundaries between several (sub) systems.

Size of the system
The amount of lines of source code can not be identified since the source code is not open. Regarding the amounts of megabytes of the system (62MB) and the functionality available in the back-office system, the amount of lines of code will probably be of the order of amount of lines of code the Store.nl back-office system contains. Including the product and price management module the size can be called big and will not contribute to the maintainability level of the OFB back-office system.

Algorithms and data structures used in the system
Since the source code of the OFB back-office system is not open to view, it is impossible to define the algorithms and data structures.

Programming language used to develop the system
The OFB back-office system is built in MS Access using the VBA programming language. The programming language is rather out-dated but is nevertheless widely used in systems. Many documentation and examples for the language can be found on the Internet and the system's maintainability level will profit from this.

Documentation
For every screen and functionality available in the OFB back-office system, documentation exists. This is very good for the maintainability level of the complete system. However, there is no way to tell if there is any technical documentation available or of what quality this technical documentation is. The system's maintainability level depends on technical documentation as well as functional documentation so if the documentation will increase or decrease the system's maintainability level can not be concluded.

Property	High	Medium	Low	N/A
Coupling		X		
Complexity of source code			X	
System size		X		
Algorithms and data structures				X
Programming language	X			
Documentation				X

Table 6: Overview maintainability properties OFB back-office

Table 6 gives an overview of all properties investigated in this section. The properties which could not be investigated are added to an extra column N/A (not available). The total level of maintainability is medium.

8.2.6 Reusability

Like the maintainability level, the reusability level of the OFB back-office system is compared with the reusability level of the Store.nl back-office system by investigating the reusability properties for the OFB back-office system. Again, for some properties no accurate description can be given since the OFB back-office system has no open source code and therefore these results will not be added to the final conclusion.
Some of the properties for the reusability level are also present for the maintainability level and will only be named in the overview of the comparison.

Modularity
The functionality in OFB is divided into several categories (see Figure 12). The OFB back-office system consists of only one file which includes all this functionality. We have no information about how the source code is organized but it is clear that it is not possible to reuse one single part of the complete system without extracting it from the system first. It is safe to conclude that the OFB back-office system is not modular and this will not aid the reusability level.

Programming style guidelines
Since the source code of the OFB back-office system is not open to view, it is impossible to define the programming style guidelines.

Self descriptiveness
Self descriptiveness is very closely related to understandability and makes it easier to understand a system. Comments in the source code or well chosen class and method names will raise the ease to understand a system. However, since the source code of the OFB back-office system is not open to view, it is impossible to define the self descriptiveness of the system.

Good exception handling
Since the source code of the OFB back-office system is not open to view, it's not possible to actually view how exceptions are handled. However, in practice it can be concluded that exceptions are handled very well. After each exception the system gives the opportunity to save the exception or send it to the creator of the system.

Information hiding
Since the source code of the OFB back-office system is not open to view, it is impossible to define if any information hiding is applied to the back-office system.

Appropriate use of packaging
Since the source code of the OFB back-office system is not open to view, it is impossible to define if the internal structure of the source code makes use of packaging.

Few external dependencies
When an application has many dependencies on other external libraries the reuse of such an application requires the use of these external libraries as well. This is not good for the reusability level of a system. The OFB back-office system has just a few external dependencies which is good for the reusability level.
The system makes use of MS Outlook to send e-mails to customers or other relations and the complete system depends on MS Access to run at all. Reusing any of the OFB back-office system implies that at least MS Access is needed as well. This is not good for the reusability level. Considering all this, the reusability level is influenced mediocre.

Property	High	Medium	Low	N/A
Modularity			X	
Programming understanding			X	
Programming style guidelines				X
Coupling		X		
Self descriptiveness				X
Exception handling	X			
Information hiding				X
Documentation				X
Packaging				X
External dependencies		X		

Table 7: Overview reusability properties OFB back-office

Table 7 gives an overview of all properties investigated in this section. The properties which could not be investigated are added to an extra column N/A (not available). The total level of reusability is between medium and low.

8.3 OS Commerce

A popular open source project to implement an e-commerce web shop is the OSCommerce (Open Source Commerce) project [OSCommerce]. Over 170.000 store owners and developers globally make use of this project. This kind of popularity has resulted into a stable and robust open source solution and there are many add-ons available for many purposes. The source code is available for free under the GNU General Public License [GNU].

Store.nl has used the OSCommerce solution in the past to sell their products to online customers. Although this solution was a good starting point for Store.nl, it does have several shortcomings for companies that want to grow and operate more professionally. However out of scope of this research, the options to change the layout and design of the visible website are very limited. This has been one of the reasons Store.nl decided to design and implement a dedicated website for their own. What does lie within the scope of this research is the back-office solution coming with OSCommerce. All basic modules to operate an online store are available such as product and order management or several possibilities in managing payment and shipping methods. These basic modules are designed to aid in operating and maintaining relatively small e-commerce web shops. Figure 13 shows the back-office main window which is the starting point to access all functionality of the system.

Figure 13: OSCommerce back-office main menu

8.3.1 Technology

OSCommerce is a completely web-based solution. The implementation language used is PHP and data is stored in a MySQL database.

8.3.2 Single task culture

Performing actions on the content of the web shop must be done one by one. Examples are adding and updating products in the database or making connections between products for selling strategies. With growing catalogues of products and additional information this ‘single task culture’ becomes a time consuming job.
In the Store.nl back-office system, the opposite is true: time consuming jobs are automated as much as possible. For adding and updating products, there exists a module to read input from files delivered by manufacturers. Newly added products and updated prices can be added and updated in one transaction. For creating connections between products the Store.nl back-office uses extra tables to connect products category-based. This means connections do not have to be created one by one, but can be connected in groups. This saves much time and has the same result on the website.

Another example of the single task culture in OSCommerce is the processing of customer orders. With just a couple of orders each day, OSCommerce has an acceptable system of processing orders available in the back-office. Each order can be edited and processed with just a few clicks. However, when more and more orders need to be processed each day, this becomes a time consuming job as well. Even more important is the impact each order has on the main stock. The overview fades away as the amount of orders grow, especially with orders containing multiple products.
Again, in the Store.nl back-office system there is no need to process orders one by one (although this is still possible). An advanced algorithm to process all orders at the same time can be run any time. This algorithm calculates the order of processing customer

orders depending on stock, date of order, date of payment, amount of products per order and price level.

8.3.3 Payment and shipping methods

To reach a wide variety of store owners in many different countries, OSCommerce provides a generic way of adding shipping and payment modules. By implementing a standard interface, the back-office allows to add as many shipping and payment methods as one would like. This way third party shippers and payment collectors can be integrated into the OSCommerce website and back-office. The downside of this approach is the lack of communication between shipping and payment modules. In practice, these two depend on each other. Payment methods available depend on the kind of shipping method chosen before and vice versa. An example is to get the products delivered by shipping company x and to choose to pay at delivery. However, due to security protocol, shipping company x does not accept payments when delivering packages while another shipping company y does.
In the Store.nl back-office it is possible to add new shipping and payment methods as well. Supplementary, the possibility is given to link shipping methods to payment methods. This way, the dependability between the methods can be enforced.
However, the properties of a shipping or payment method are needed in many places throughout the back-office system. Modules working with these different properties have to be modified in the source code when a new shipping or payment module is added. This is not good for the flexibility of the system, but a solution to this problem brings much other developing with it.

8.3.4 Maintainability

To compare the OSC back-office system's maintainability level with the Store.nl back-office system's maintainability level we describe the properties of a maintainable system in this paragraph.

Coupling with other parts of the system
There are no clear boundaries inside the OSC back-office system between different parts or modules except for the add-ons. The total coupling of the system is therefore high which is not good for the maintainability level. The add-ons can be maintained separately. However, some knowledge is still required from the OSC back-office system.

Complexity of the source code
The source code of the OSC back-office system consists of files containing the mark-up of the pages (which can be seen as a GUI layer), files containing general and specific functionality, and class files containing objects used in the system. The latter two both contain the business logic of the system. This separation is good and helps in maintaining the total system.
However, there is no separation for data access control at all. Database queries can be found in both the GUI files and business logic files and even within these files the queries are not separated from other functionality.
Overall, the maintainability level of the OSC system is mediocre influenced considering both properties.

Size of the system
The total number of lines of code is approximately 68.000 lines. This does not include comments or add-ons. The OSC back-office system has a tight coupling and so the complete system is affected when it is maintained. The amount of lines is large too maintain all at the same time so this will not be positive for the system's maintainability level.

Algorithms and data structures used in the system
The data structures found in the source code are not complex and straightforward. Default PHP structures are well documented on the Internet and custom data structures are no more than classes containing some attribute fields.
There are very few algorithms used in the OSC back-office system. The most complex one is the calculation of the total shopping cart content of online customers including taxes and optional product additions.

Programming language used to develop the system
The OSC back-office system is completely developed in PHP. This is currently the most popular and most used language in building web-based scripts and applications. There are lots of examples and documentation available on the Internet which is good for the maintainability level of the system.

Documentation
Together with the OSC back-office system, an extensive functionality document is supplied. This gives a basic overview of all functionality and defines how to apply simple changes to the back-office. Furthermore, possible exceptions are described and many tips about administrating the system.
The source code contains some comments, but not many. Besides the 68.000 lines of code, 2500 lines of comments are available in the source files. Most of the comments are legal notices and these will not help in maintaining the system.
There is a clear database design available which includes all tables and fields used in the database and all relations between the fields.

Property	High	Medium	Low
Coupling		X	
Complexity of source code		X	
System size			X
Algorithms and data structures	X		
Programming language	X		
Documentation		X	

Table 8: Overview maintainability properties OSC back-office

Table 8 gives an overview of the levels of maintainability of the OSC back-office for each property. It can be concluded that the total level of maintainability is medium.

8.3.5 Reusability

To compare the OSC back-office system's reusability level with the Store.nl back-office system's reusability level the properties of a reusable system are described in this paragraph. Some of the properties for reusability level are also present for the maintainability level and will only be named in the overview of the comparison.

Modularity
The OSC back-office system's functionalities are divided into several categories. Most functions offered by the back-office exist in one or more separate files. These files could be reused in another system. There are also many files which are included in multiple other files and these files break down the modularity of the system. When reusing some parts of the system, these files also need to be included. Also, business logic, data access and user interface are not divided in a structured way. Reusing functionality will mostly add other (unwanted) functions or user interfaces as well. The overall modularity is being identified as medium.

Programming style guidelines
When reuse comes into view, adhering to programming style guidelines helps understanding the source code. In the OSC back-office system, the source code is pretty consistent in its style. Although there no specification is available about the programming style guidelines, clear regularities are sometimes present in the source code. This structured source code helps understanding the source when reusing parts of the system.

Self descriptiveness
Source files in the OSC back-office system have good names which name the category or function they implement. However, the contents of the source files are not always straightforward in understanding. Many includes are sometimes misleading and could lead to unexpected behaviour when inspecting the source code. Overall, the source is not very self descriptive, which is not good when reusing parts of the system.

Good exception handling
The OSC back-office system has not many specific handlers in place to intercept exceptions and deal appropriately with them. When an exception occurs, the PHP engine will simply dump the error text to the client instead.
In the functional document many kinds of possible errors (exceptions) were described with additional information about the possible origin and solution of the error. Both considered, reuse of parts of the system will not be completely straightforward when exceptions occur, however some causes and solutions are documented.

Information hiding
The use of several classes in the OSC back-office system makes information hiding possible in the system. The classes can be reused and the contents of the methods and functions are hidden from the component using the class. Despite this contributing property there are also many parts of the system completely transparent to all other components.

Appropriate use of packaging
The OSC back-office system does not make any use of packaging which is not good for the reusability level of the system.

Few external dependencies
The OSC back-office system does not have external dependencies. All functionality is present in the system itself and does not rely on other external applications. When reusing parts of the system this is a positive property.

Property	High	Medium	Low
Modularity		X	
Programming understanding		X	
Programming style guidelines	X		
Coupling		X	
Self descriptiveness			X
Exception handling		X	
Information hiding		X	
Documentation		X	
Packaging			X
External dependencies	X		

Table 9: Overview reusability properties OSC back-office

Table 9 gives an overview of the levels of usability of the OSC back-office for each property. The total level of reusability is medium.

8.4 Comparison overview

In sections 8.2 and 8.3, many properties of the OFB back-office and the OSCommerce back-office are compared with the Store.nl back-office. In Table 10 the most important general properties are listed and compared between the three back-office systems.

	OSC	**OFB**	**Store.nl BO**
Database type	MySQL	MS Access	Any SQL language
Real-time website information	Yes	No	Yes
Transaction support	Yes	N/A	Yes
Explicit service orientation	No	No	Yes
Interoperability	Low	Low	Medium
Implementation language	PHP	VBA	.Net C#
Desktop/Web based	Web	Desktop	Desktop
Extensible	Yes (Add-ons)	No	Yes (Modules)
Manage multiple stock locations	No	Yes	Yes

Table 10: General properties comparison overview

The following gives an overview of the differences in maintainability and reusability levels between the Store.nl back-office system, OFB back-office system and OSC back-office system. These properties are the most important requirements when successfully implementing an e-commerce back-office system.

Property	**Store.nl BO**	**OFB**	**OSC**
Coupling	Medium	Medium	Medium
Complexity of source code	Medium	Low	Medium
System size	High	Medium	Low
Algorithms and data structures	High	N/A	High
Programming language	High	High	High
Documentation	Low	N/A	Medium

Table 11: Maintainability level comparison overview

Table 11 shows an overview of the investigated properties of all three back-offices regarding maintainability. The properties coupling and programming language have the same impact on all back-offices. The coupling with other parts of the system has a medium impact on the maintainability level. Although all back-offices are implemented using different programming languages, these have no negative influence on the maintainability levels at all.
Some minor differences exist in the complexity of the source code. For the Store.nl and OSC back-office this has a medium impact on the maintainability level and for the OFB back-office it has a negative impact. Related to the complexity of the source code the algorithms and data structures are being used. The OFB back-office does not disclose its source code but concerning both OSC and Store.nl there are no complex algorithms and data structures used which could impact the maintainability level.
Remarkable is the outcome of the system size property. The Store.nl back-office is largest system but has the least negative impact on the maintainability level. Due to the architecture of the Store.nl back-office system all sub systems can be maintained independently from each other. This breaks down the complexity of the system and results into a high maintainability level.

Concluded can be that the Store.nl back-office system has the highest overall maintainability level based on the properties investigated. It must be noted that some

information about the OFB back-office system is not available and the results could be different when this would have been available.

Property	Store.nl BO	OFB	OSC
Modularity	High	Low	Medium
Programming understanding	Medium	Low	Medium
Programming style guidelines	High	N/A	High
Coupling	Medium	Medium	Medium
Self descriptiveness	Medium	N/A	Low
Exception handling	High	High	Medium
Information hiding	High	N/A	Medium
Documentation	Low	N/A	Medium
Packaging	High	N/A	Low
External dependencies	Medium	Medium	High

Table 12: Reusability level comparison overview

Table 12 shows an overview of all properties investigated of all three back-offices regarding reusability. An important property is modularity. The Store.nl back-office has a high modularity while the OFB and OSC back-offices have a low and medium modularity respectively. Together with coupling (which is medium for all three back-offices), modularity has the most influence on the level of reusability.
Programming understanding is medium for the Store.nl and OSC commerce and low for the OFB back-office. When reusing parts of these systems, the first two mentioned are more understandable and easier to reuse.
The programming style guidelines are clear in the OSC and Store.nl back-office which will add value to the reusability level. This property can not be compared with the OFB back-office.
The quality of exception handling is high in the Store.nl and OFB back-office and medium in the OSC back-office. If parts of the systems are reused, the latter back-office could miss some additional exception information compared to the first two back-offices.
All three back-offices have few external dependencies. The OSC back-office has the least of them and this will add value when reuse of parts of the system is required.
The Store.nl back-office has not much technical documentation available. The OSC back-office has more documentation available and has a medium influence when reusing parts of the system. The OFB back-office has no documentation available.

The Store.nl back-office system has most properties influencing the total reusability level positively and only one property influencing it negatively. Therefore, we can conclude that the Store.nl back-office system has the highest reusability level out of the three back-office systems compared. Again, some information is missing about the OFB back-office system, but for the most important properties of being a reusable system, modularity and coupling, there is information available for all systems.

9 Conclusions

In this thesis the technological needs of e-businesses has been focused on. These businesses manage to profit on the Internet sales market by selling goods and services. In order to do so, a supporting system is needed and therefore the following research question has been set up:

What is a suitable architecture for an e-commerce back office system considering the e-business environment and requirements?

To answer this question one has to first answer some other questions. Firstly, must be concluded what the most important requirements for a back-office system in the e-commerce business field are. Analyzing the e-business field, it can be concluded that the non-functional requirements: flexibility, maintainability and reusability are the most important requirements in successfully implementing an e-commerce back-office system.

Secondly, the latest state of the art architecting techniques has been searched in order to make a decision which techniques should be used in architecting an e-commerce back-office system. Using the requirements maintainability and reusability, two suitable techniques have been selected: service-oriented architecture and pattern-oriented architecture. Using both techniques the question: “What needs to be done to successfully implement a system based on the requirements?” can be answered. This has resulted into a framework to support an e-commerce back-office system.

The e-commerce back-office framework (ECBF) has a service-oriented approach in adding or removing functionality to an application implementing the framework. Using this approach, different sub-systems can be integrated into one main application and can be developed independently from each other. Communication between the different sub systems has been realized by collecting all available services in the modules and by adding them to a service repository.

Store.nl uses a back-office system implementing the ECBF. To give a clear conclusion about the levels of maintainability and reusability of the system several properties indicating these levels have been defined. Additionally, two other back-office systems were being compared with the same properties.
The back-office system implementing the ECBF has a high level of maintainability while two other comparable back-office systems have a medium level of maintainability. The same conclusion holds for the level of reusability of the system implementing the ECBF: the level is high while the other systems have a medium or low level or reusability.

With the latter conclusions, the main research question can be answered. Considering the e-business environment and requirements, an e-commerce back-office system should be designed using the ECBF. The service-oriented approach of the framework provides solutions for the collected requirements and shows positive influences on them. Compared to two other back-office systems with comparable functionality and purpose, the back-office implementing the ECBF has the highest levels of maintainability and reusability.

9.1 Future work

In the current state of the e-commerce back-office framework, there are three possible attributes to declare: *MainForm*, *SettingsForm* and *Service* attributes. A possible addition which would add value to the framework is to introduce a web service attribute. Code marked with this attribute should be a service exposed to the outside of the framework available over a network.

The data access framework inside the ECBF has the ability to change the database type it is communicating with, by just changing a setting. However, the queries specified in the data access layers of the modules should possibly be changed as well because of the slight differences between the structured query languages. A possible addition to the framework would be the introduction of *NHybernate*. This package has its own query language defined and gives the ability to translate it into many other languages. Introducing this would make the data access framework more flexible and changing the database type would be even easier than it is now.

10 Abbreviations used

B2B	Business to business
BLL	Business logic layer
CBS	Centraal bureau voor de statistiek (Dutch)
CRM	Customer relationship management
DAL	Data access layer
DLL	Dynamic-link library
E-Commerce	Electronic commerce
EXE	Executable
GUI	Graphical user interface
MS Access	Microsoft access
MS SQL	Microsoft structured query language
MySQL	My structured query language
OFB	Office for business
OSC	Open source commerce
PHP	Hypertext pre-processor
RAM	Random access memory
RMA	Return material authorization
SOA	Service-oriented architecture
VBA	Visual basic for applications
XML	Extensible markup language

11 References

[Alexander ‘79] Alexander C., *The Timeless Way of Building*, Oxford University Press, 1979

[Arsanjani ‘04] Arsanjani A., *Service-oriented modeling and architectures*. IBM, November 2004. http://www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/

[Bichler ‘98] Bichler, Martin, Segev, Arie, Ahao J. Leon, *Component-based E-Commerce: Assessment of Current Practices and Future Directions*, ACM Sigmod Record: Special Section on Electronic Commerce, Vol. 27, No. 4, December 1998, pp. 7-14

[Blauw Research] Blauw Research & Thuiswinkel.org, Thuiswinkel marktmonitor 2007-1, http://www.thuiswinkelawards.nl/onderdeel/thuiswinkelbiz/persberichten.asp?navid=5&id=6733

[Buschmann ‘96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns*, Whiley, 06/1996

[CBS] Centraal Bureau voor de Statistiek, Internetaankopen 2007, http://www.cbs.nl/NR/rdonlyres/F0D0B910-52ED-4C61-8231-44B01DC5E39D/0/pb07n072.pdf

[Chappell ‘04] Chappell, David A., *Enterprise Service Bus*, O’Reilly, 2004

[Dyson ‘04] Dyson P., Longshaw A., *Architecting Enterprise Solutions: Patterns for High-Capability Internet-Based Systems*, John Wiley & Sons, Ltd., 2004

[Fensel ‘03] Fensel Dieter, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer, 2003

[Fowler ‘03] Fowler Martin, *Patterns of Enterprise Application Architecture*, Pearson Education Inc, 2003

[Gaedke ‘99] Gaedke, Martin, Turowski, Klaus, *Framework for Maintaining Evolution of E-Commerce Applications in the Web*, J.-C. Rault (Ed.): 12th International Conference Software & Systems Engineering and their Applications (ICSSEA'99). Vol. 5, Paris, pp. 18.4.1-18.4.10, 1999

[Gerson ‘06] Gerson G., *Service Oriented Architecture and Business Processes*, AIIM E-Doc Magazine, 06/2006

[GNU] GNU.org, GNU General Public License, http://www.gnu.org/licenses/gpl-3.0.txt

[Keen ‘04] Keen M., Acharya A., Bishop S., Hopkins A., Milinski S., Nott C., Robinson R., Adams J., Verschueren P., *Patterns: Implementing an SOA Using an Enterprise Service Bus*, IBM Internation Technical Support Organisation, 07/2004

[Kraftzig ‘05] Kraftzig D., Banke K., Slama D., *Enterprise SOA: Service-Oriented Architecture Best Practices*, Prentice Hall, 2005

[OFB] OFB Software website, http://www.ofb.nl

[OSCommerce] OSCommerce website, http://www.oscommerce.com

[Papazoglou '07] Papazoglou, Michael P., *Web Services: Principles and Technology*, Pearson Prentice Hall, 2007

[Papazoglou '06] Papazoglou, Michael P., Ribbers Pieter M.A., *e-Business: Organizational and technical foundations*, John Wiley & Sons, Ltd., 2006

[Poulin '94] Poulin, Jeffrey S., *Measuring Software Reusability*, Proceedings of the Third International Conference on Software Reuse, Rio de Janeiro, Brazil, 1-4 November 1994

[Singh '05] Singh Munindar P., Huhns Michael N., *Service-Oriented Computing Semantics, Processes, Agents*, John Wiley & Sons, Ltd., 2005

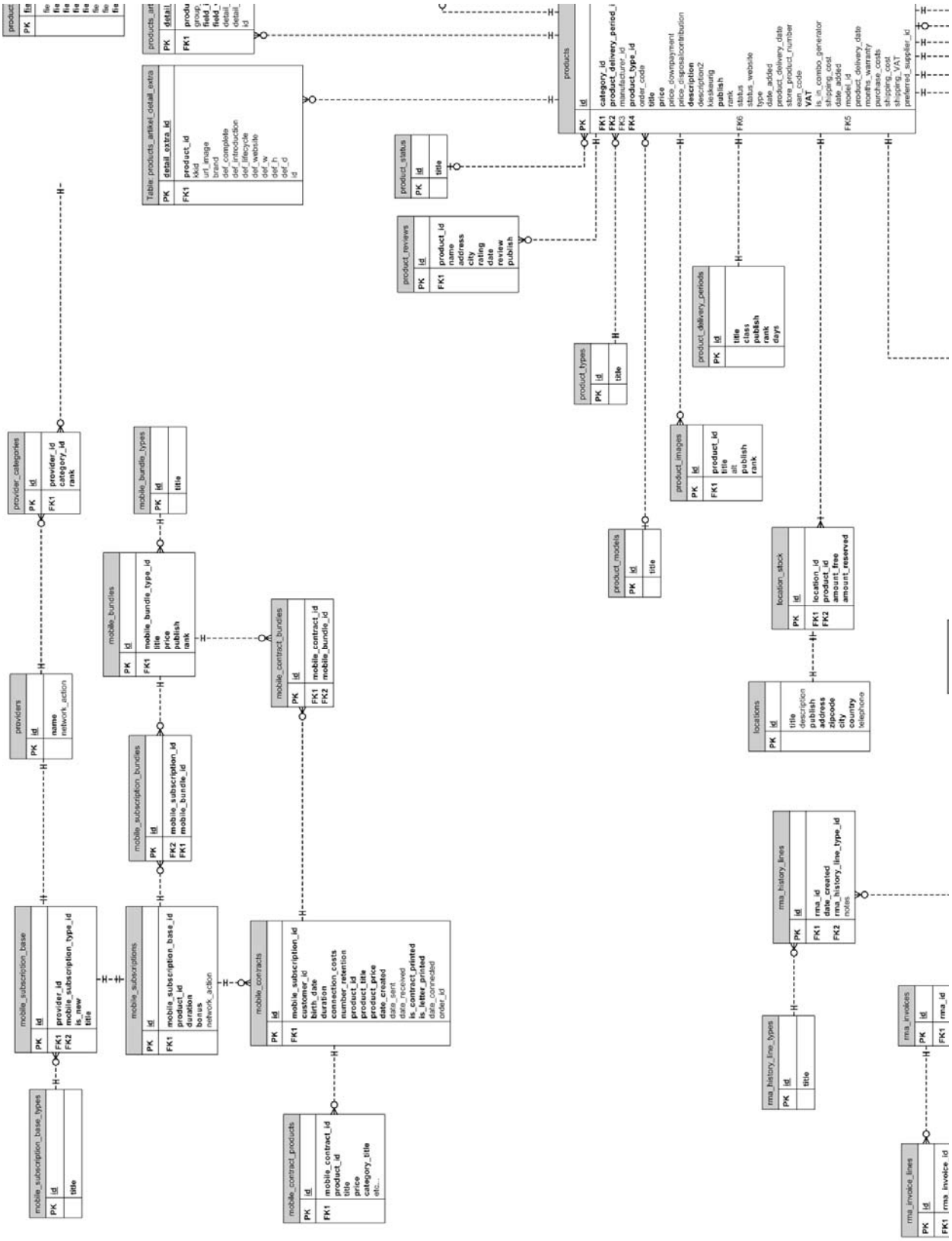
[Sommerville '07] Sommerville I., *Software Engineering 8th edition*, Pearson Education Limited, 2007

[Taylor '06] Taylor H., *Compliance in the Era of Service-Oriented Architecture*, AIIM E-Doc Magazine, 06/2006

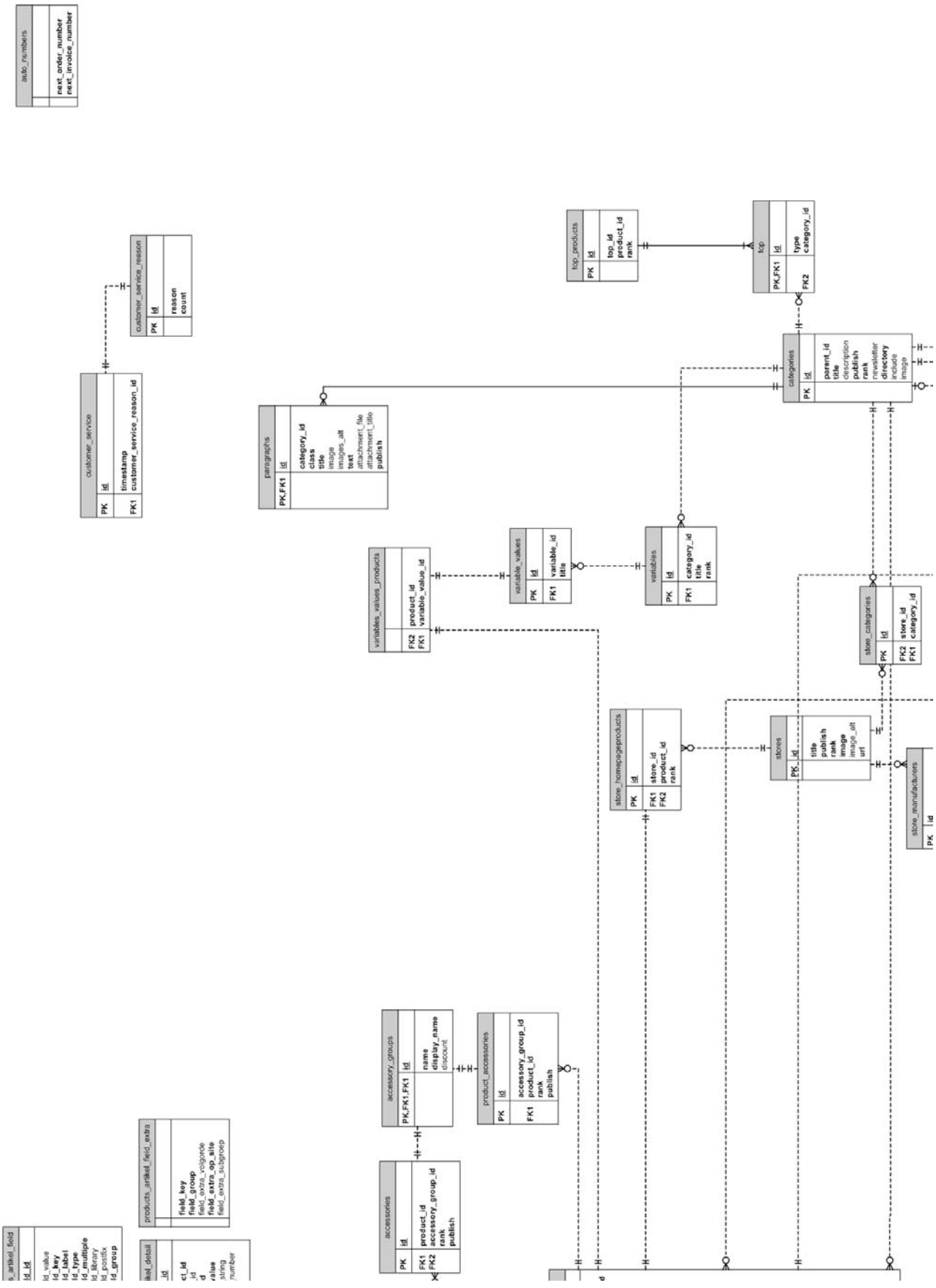
[Tews '07] Tews, Richard, *Beyond IT: The Business value of SOA*, AIIM E-Doc Magazine, 10/2007

[Vernadat '07] Vernadat F.B., *Interoperable enterprise systems: Principles, concepts and methods*, Elsevier Annual Reviews in Control 31 pp137-145, 05/2007

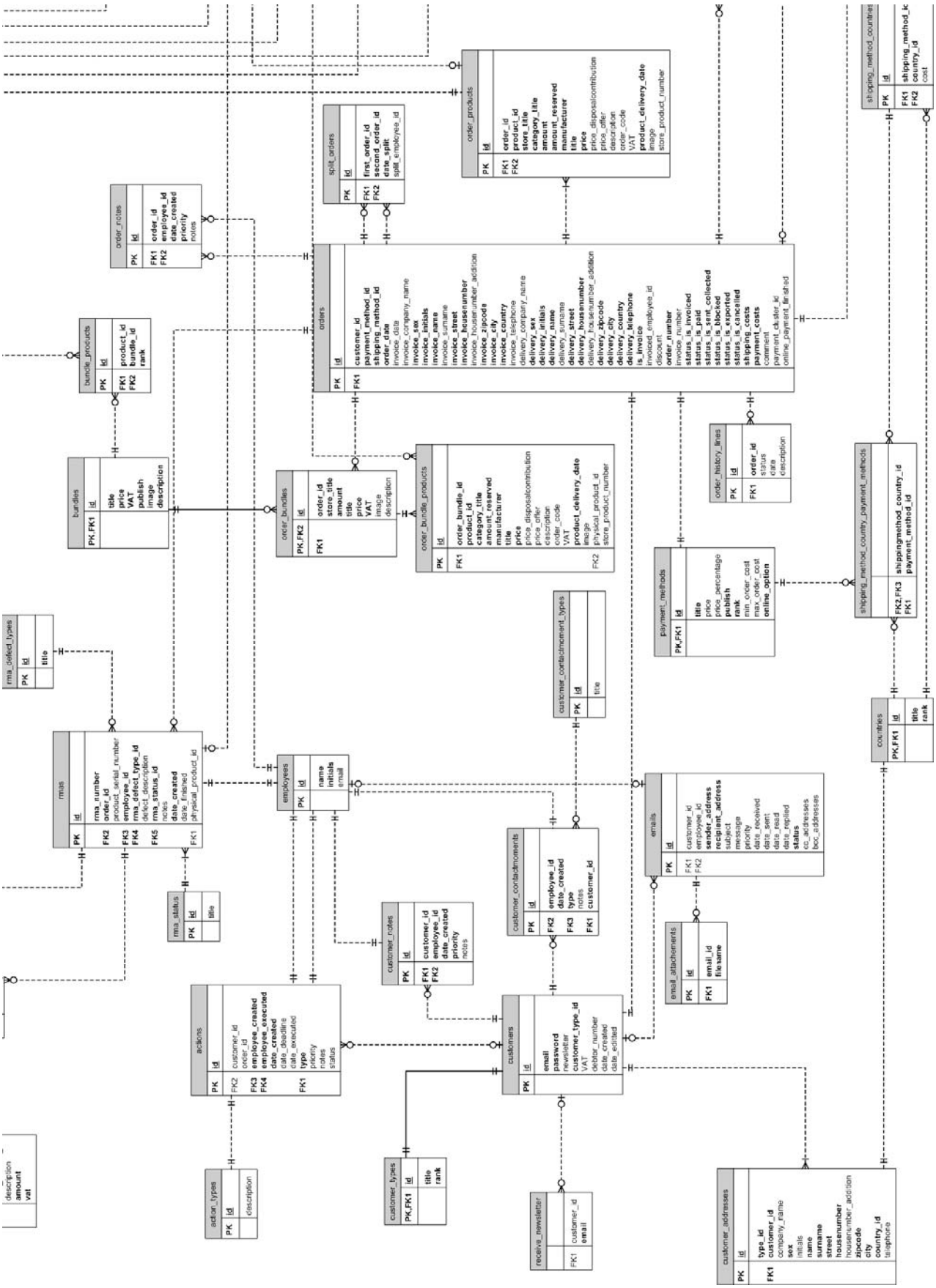
Appendix A – Store.nl back-office database design



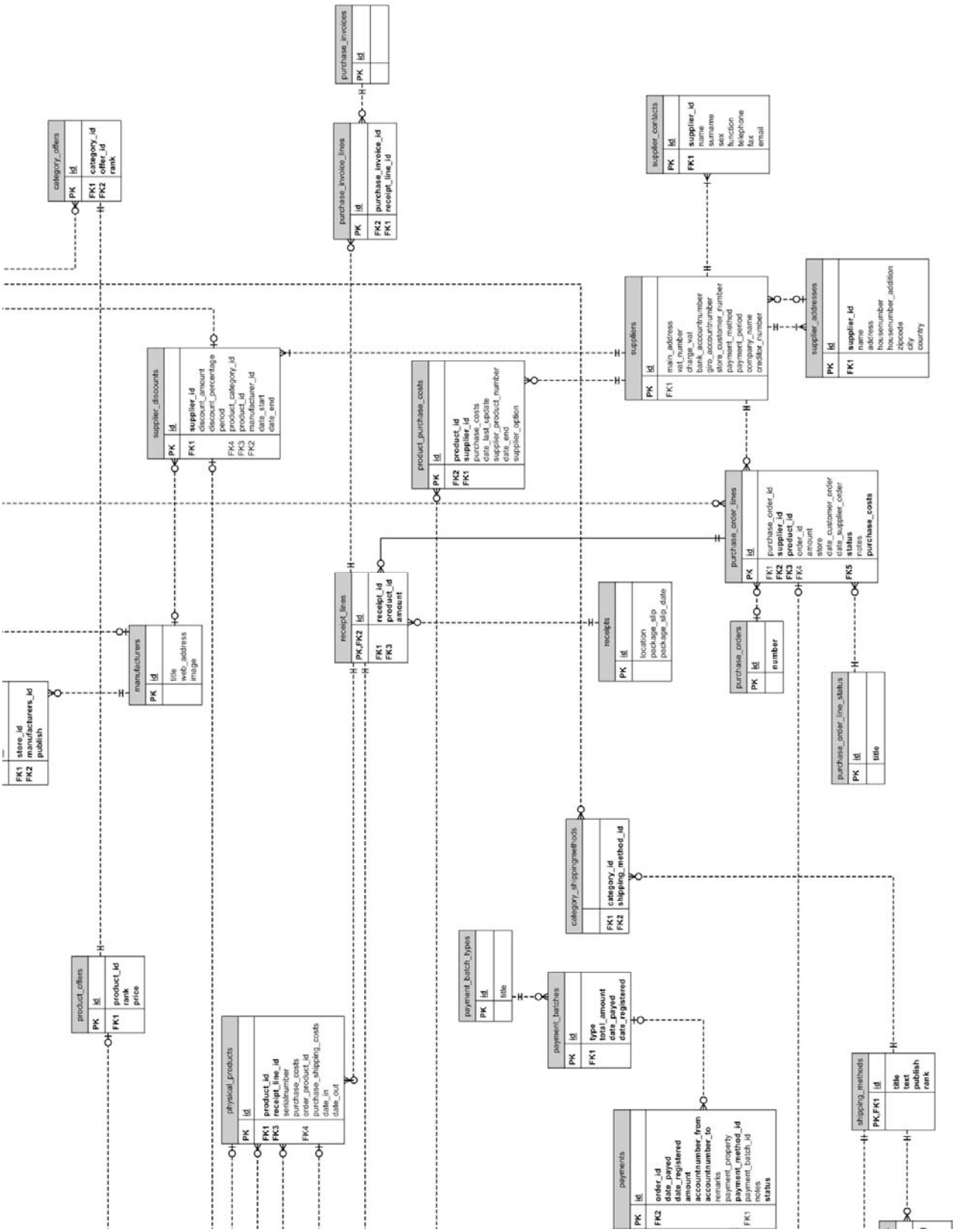
(top left)


auto_numbers
next_order_number
next_invoice_number
customer_service_reason
PK id
reason
count
customer_service
PK id
timestamp
FK1 customer_service_reason_id
top_products
PK id
top_id
product_id
rank
top
PK,FK1 id
type
category_id
FK2
paragraphs
PK,FK1 id
category_id
class
title
image
images_alt
text
attachment_file
attachment_title
publish
categories
PK id
parent_id
title
description
publish
rank
newsletter
directory
include
image
variables_values_products
FK2 product_id
FK1 variable_value_id
variable_values
PK id
FK1 variable_id
title
variables
PK id
FK1 category_id
title
rank
store_categories
PK id
FK2 store_id
FK1 category_id
store_homepageproducts
PK id
FK1 store_id
FK2 product_id
rank
stores
PK id
title
publish
rank
image
image_alt
url
store_manufacturers
PK id
accessory_groups
PK,FK1,FK1 id
name
display_name
discount
product_accessories
PK id
FK1 accessory_group_id
product_id
rank
publish
accessories
PK id
FK1 product_id
FK2 accessory_group_id
rank
publish
products_anked_field_extra
field_key
field_group
field_extra_volgorde
field_extra_op_site
field_extra_subgroep
ikel_detail
id
ct_id
id
d
alue
string
number
s_anked_field
id_id
id_value
id_key
id_label
id_type
id_multiple
id_library
id_prefix
id_group

(top right)


orders
order_products
order_notes
split_orders
bundle_products
bundles
order_bundles
order_bundle_products
order_history_lines
payment_methods
shipping_method_country_payment_methods
shipping_method_countries
countries
rma_defect_types
rmas
rma_status
employees
customer_contactmoments_types
customer_notes
customer_contactmoments
emails
email_attachements
actions
action_types
customers
customer_types
receive_newsletter
customer_addresses

(bottom left)



(bottom right)