FAKULTÄT FÜR !NFORMATIK

Development of a Web Application for the Worldwide Management of Fire Trucks CAN Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Thomas Bruckmayer
Matrikelnummer 0225696

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuerin: o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel
Mitwirkung: Univ.Ass. Mag.rer.soc.oec. Petra Brosch

Wien, 30.10.2009

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

_____

Wien, October 30, 2009

Danksagung

Ich möchte mich an dieser Stelle bei allen Personen bedanken die mich während meines Studiums unterstützt haben.

Darüber hinaus bedanke ich mich bei Herrn Dipl.-Ing. Oliver Hrazdera für die Möglichkeit die Diplomarbeit in einem sehr spannenden und herausfordernden Projekt bei Rosenbauer International AG anzufertigen. Bei Herrn Dipl.-Ing. Joris Gruber bedanke ich mich für die Hilfe in speziellen technischen Angelegenheiten. Ein ganz besonderes Dankeschön geht an Herrn Anton Klucsarits für das Erklären zahlreicher CAN Bus Details und das Beantworten vieler Fragen.

Für die sehr gute Betreuung und die nützlichen Anregungen bei der Erstellung dieser Diplomarbeit möchte ich mich bei Frau Mag. Petra Brosch und Frau o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel bedanken. Bei Frau Mag. Petra Brosch auch für die Unterstützung bei vielen interessanten Meetings in Linz.

Zu guter Letzt möchte ich mich bei meiner Mutter Frau Mag. Inge Bruckmayer für die finanzielle und moralische Unterstützung während meines ganzen Studiums bedanken.

Abstract

A sustainable knowledge management and sophisticated tool support are extremely important for every successfully operating company. Within an extremely wide range of tools, Web applications gain more and more importance to meet the constantly increasing requirements. Outstanding benefits are worldwide accessibility and interoperability on a very large scale. Moreover, in many cases no installation on single clients is needed, updates can be provided easily, and centralized data management on the server-side avoids costly synchronizations. These factors lead to reduced costs for the information infrastructure and support the employees to do their work. For example, a constructor of fire-fighting trucks can query the headquarter's database on the other side of the world to find out which CAN data (Controller Area Network) is needed to configure a specific vehicle.

Building such solutions is not a trivial task and therefore disciplines like Web Engineering and Internet Computing emerged. Furthermore, developers can choose from a wide range of technologies to realize their solutions. Handling these technologies leads to successful development of Web applications. This master's thesis describes in detail the solution for a specific problem in the industry, namely a Web application called CORA (CAN Bus Organization - Rosenbauer Assistant) to manage the CAN data for Rosenbauer International AG. According to the company's IT-infrastructure MSSQL Server, IIS, and ASP.NET were chosen as core technologies. On one hand the .NET framework provides the possibility to develop applications in a rather short time, also known as Rapid Application Development (RAD). On the other hand many RAD-techniques are not applicable on large enterprise solutions where the complexity has to be broken down into several layers. This work presents approaches, patterns and techniques for each layer.

The Data Access Layer is responsible for retrieving and manipulating data and uses LINQ to SQL as object-relational mapper. The Business Layer handles the communication to external systems and adds business functionality between the Data Access Layer and the Presentation Layer. Finally, the Presentation Layer presents the data in an appropriate format and handles the user interaction.

Furthermore, the database schema of the sample application has been constantly renewed and improved to cover additional requirements like multiple CAN bus systems, multilingualism, user administration, and a history of important entities. Therefore, schema evolution and data migration play an additional important role in this thesis. All these aspects are elaborated theoretically and explained practically with the help of CORA.

Kurzfassung

Ein nachhaltiges Wissensmanagement in Verbindung mit einer ausgeklügelten Werkzeugunterstützung ist extrem wichtig für jedes erfolgreich agierende Unternehmen. Unter einer Vielzahl von Werkzeugen erlangen Web-Anwendungen immer mehr an Bedeutung. Herausragende Vorteile von Web-Anwendungen sind weltweite Erreichbarkeit und Interoperabilität. Darüber hinaus ist in vielen Fällen keine Klientinstallation notwendig, Updates können leicht zur Verfügung gestellt werden und eine zentrale Verwaltung der Daten am Server vermeidet teure Datensynchronisation. Diese Faktoren führen zu einer Kostenverminderung für die Informationsinfrastruktur und unterstützen die Arbeitskräfte bei ihrer täglichen Arbeit. Zum Beispiel kann ein Konstrukteur von Feuerwehrautos die Datenbank von einem entfernten Produktionsstandort nach einer speziellen CAN Konfiguration (Controller Area Network) für ein spezifisches Fahrzeug abfragen.

Der Entwurf solcher Lösungen führte zu Disziplinen wie Web Engineering und Internet Computing. Außerdem können Entwickler aus einer Vielzahl von Technologien wählen, um deren Lösungen zu realisieren. Diese Diplomarbeit beschreibt im Detail die Lösung für ein spezielles Problem aus der Industrie, nämlich eine Web-Anwendung genannt CORA für die Verwaltung der CAN Daten für Rosenbauer International AG. Gemäß der firmeninternen IT - Infrastruktur wurden der MSSQL Server, IIS und ASP.NET als Kerntechnologien ausgewählt. Auf der einen Seite bietet das .NET Framework die Möglichkeit Anwendungen in kurzer Zeit zu entwickeln, auch als Rapid Application Development (RAD) bekannt. Auf der anderen Seite sind viele RAD-Technologien nicht im großen Rahmen für Unternehmenslösungen einsetzbar, da diese eine Aufteilung der Komplexität in einzelne Schichten verlangen. Diese Arbeit präsentiert Herangehensweisen, Entwurfsmuster und Techniken für jede einzelne Schicht.

Die Datenzugriffsschicht ist für die Abfrage und Manipulation von Daten verantwortlich und verwendet LINQ to SQL als Objekt Mapper. Die Schicht der Geschäftslogik behandelt die Kommunikation zu externen Systemen und führt die Geschäftslogik aus. Letztendlich präsentiert die Präsentationsschicht die aufbereiteten Daten dem Benutzer und verarbeitet die Benutzereingaben.

Des Weiteren wurde das Datenbankschema der Beispielanwendung ständig erneuert und erweitert, um den zusätzlichen Anforderungen wie verschiedenen CAN Bus Systemen, Mehrsprachigkeit, Benutzerverwaltung, und einer Änderungsliste der CAN Entitäten, gerecht zu werden. Deswegen spielen Schema Evolution und Datenmigration eine wichtige Rolle in dieser Arbeit. All diese Aspekte werden theoretisch im Detail behandelt und anhand der CORA praktisch erörtert.

Contents

1. Introduction

1.1. Web Engineering

It is intrinsic to human nature to search for answers to open questions and to develop supportive tools. Therefore the raise of the Internet, especially the World Wide Web, had an enormous impact on society. People are now able to find desired information instantly and share knowledge across the planet. Moreover, the industrial information management changed to a great extent as well as now companies provide their departments with new tools to handle their highly specialized knowledge. Imagine a company which produces fire-fighting technologies including vehicles. One department possesses the knowledge to equip a specific vehicle to perform a desired action, for example the configuration of the pump engine. Now another production location wants to use the already developed configuration for their pump engines. One possible solution would be a worldwide accessible database which stores this configuration. Therefore Web solutions are essential among knowledge management tools and range from simple document centric Web sites and static HTML documents to interactive Web applications. Realizing such solutions would not have been possible without the manifestation of a discipline called Web Engineering, which provides systematic approaches, concepts, methods, techniques, tools and technologies. The sources for Web Engineering can be described as in figure 1.1.

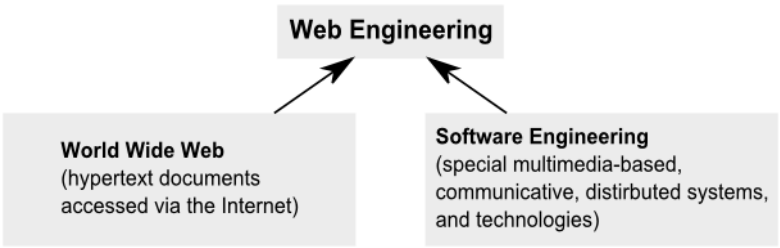

Figure 1.1.: Sources of Web Engineering according to [5]

Nevertheless Web development and traditional software development differ in a number of areas. One of the main differences is that the former uses communication technologies and generates non-sequential Web applications which can be accessed on different platforms. This leads to new challenges for the developers which may include user interfaces for mobile devices or a navigation structure which handles multiple resources for different user groups. Consequently developing Web applications for a specific problem domain is a highly complex process consisting of different phases. There exist numerous process models and frameworks like the Rational Unified Process (RUP)[1], the Microsoft Solution Frame-

[1] http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process

work (MSF)[2], or SCRUM[3] to support the development life cycle. In many cases it is not necessary to strictly follow a process model but it is inalienable to abstract some basic conditions to give the own Web application development process a certain structure. The Web application for the worldwide management of CAN data was realized through the following phases:

1. Envisioning: At the beginning of the development process a vision for solving the given problem must be generated. Therefore all the involved stakeholders have to meet and formulate requirements. Back to our CAN data management example, an existing solution allows the derivation of many requirements for the new Web solution. Nevertheless a detailed use case specification is essential.

2. Planning: After formulating requirements the solution needs to be planned and possible technologies must be evaluated.

3. Developing: The solution is implemented during the development phase. Certain milestones and review meetings guarantee that the development is on the right track.

4. Stabilizing: The functional efficiency of the developed solution can only be assured through numerous tests. In addition needed updates and bug fixes are realized in this phase.

5. Data Migration: When the quality of the implemented solution has been assured, all the existing data has to be migrated to work flawlessly with the new system.

6. Deploying: Finally the developed solution gets deployed on the productive system.

These phases are not intended to be executed in a strict sequence; in turn they are performed within iterations. For instance, some data can be migrated before stabilizing takes place in order to support quality tests.

Web Engineering incorporates a huge amount of different technologies but many aspects are technologically independent and universally valid. For example the concept of separation between layout and logic should be taken into account for every solution, no matter if Java, ASP.NET or PHP has been chosen as core technology. Choosing the right technology is a critical parameter for the Web project management. Several factors including the estimated size of the project have to be considered. In many cases the decision is limited by the company's IT-infrastructure. For the practical example described in this thesis ASP.NET was chosen as core technology by the IT-authorities.

[2]http://en.wikipedia.org/wiki/Microsoft_Solution_Framework
[3]http://en.wikipedia.org/wiki/Scrum_(development)

1.2. ASP.NET

ASP.NET is a server-side Web application technology developed by Microsoft as the successor to the Active Server Pages (ASP). The predecessor has to deal with several problems like the following.

Spaghetti Code. HTML is mixed with server-side script in classic ASP. This leads to larger source files which are harder to read and to maintain. Furthermore the performance is affected due to the server's script engine which has to switch on and off several times.

Script Languages. In classic ASP every object or variable is created as weakly typed data type which requires higher amount of memory and slows down performance. Furthermore it is impossible to develop a sophisticated integrated development environment (IDE) when the data type can be only determined and checked during runtime. This means no debugging, error checking and IntelliSense (auto completion) to support the developer.

According to [14] Microsoft was able to start from scratch with ASP.NET including the following fundamental changes.

Integration within the .NET platform. ASP.NET is completely integrated within the .NET platform and therefore the developers can choose among several supported languages (including C# and Visual Basic) and can use all the functionality of the sophisticated .NET Class Library.

Object orientation. In addition to the access of all ASP.NET objects provided by the .NET framework, a developer can use all the benefits of an object-oriented programming environment. For instance, programmers are able to create reusable classes, extend existing classes or use interfaces to standardize their code. Furthermore good examples of object-oriented thinking in ASP.NET are server-based controls. Developers can customize these objects, which are also able to react to certain events, to render any desired low-level HTML markup.

1.3. Rapid Application Development

The complexity and amount of data which has to be managed by different departments within a company is constantly increasing. As a consequence the desire for ingenious tool support and solutions with a shorter development cycle has never been higher. This applies to Web applications as well, and the industry reacts with products to support shorter development methods also known as Rapid Application Development (RAD). For example Microsoft's Visual Studio allows developing data centric Web applications in ASP.NET with the help of predefined controls and data sources in a rather short time. So the developer can use a designer with drag and drop possibilities to create a Web site that displays and modifies data without writing a single line of code. The major drawback lies in

the tight coupling of the numerous components. For example the Presentation Layer easily includes elements for data access. In the worst case SQL commands get directly inserted into the template files which should only define the Web site structure. More sophisticated RAD approaches in Visual Studio call stored procedures on the database-side or call methods to return objects. Nevertheless, the use of these predefined controls always comes along with limitations for the developer, for instance the method called by the data-source-control must provide a specific signature or supports limited return types.

The following example 1.2 demonstrates the development of a Web site which displays a simple result set.

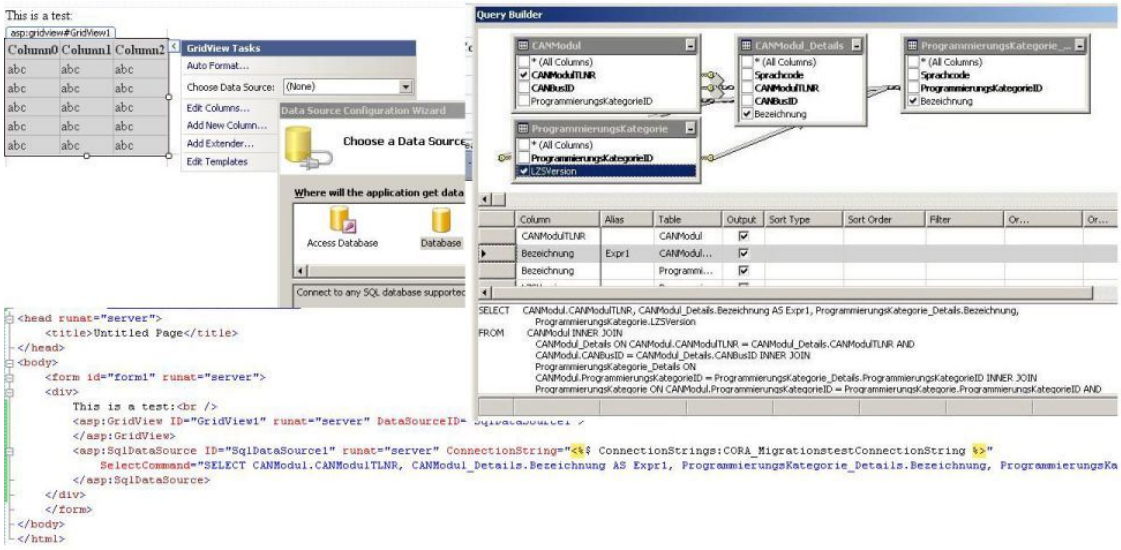

Figure 1.2.: Rapid Application Development in ASP.NET with Visual Studio

In the example above the SQL command is directly inserted in the ASPX template file which leads to several problems. First of all, it is really difficult to apply additional logic to the retrieved data. Moreover, if a similar result set is needed on another page the query has to be generated again which means a maintenance nightmare in large applications, because then every page has to be updated. In addition service orientation gains more and more importance in the company IT landscape, which basically means that before developing monolithic applications for every particular problem one should consider several services which provide certain functionality. In our example a Web service could provide the queried data for several Web applications. This is not possible when the data is directly bound to a presentation control and therefore not stored in a communicable format on a single accessible location. Last but not least a screen designer, who does not have a firm grasp of data retrieval techniques, has to deal with SQL in the template files and could produce errors.

Data source controls Almost every enterprise application has to deal with persistently stored data. Therefore data retrieval, data manipulation and data representation are targets for RAD approaches. Moreover, in many cases the data

sources are multifaceted like relational database systems or XML files. With the data source controls ASP.NET provides a method for linking a Web site to a data source. According to [14] the NET framework includes the following data source controls:

- SqlDataSource
- ObjectDataSource
- AccessDataSource
- XMLDataSource
- SiteMapDataSource
- LinqDataSource

For the CORA (CAN Bus Organization - Rosenbauer Assistant) the SqlDataSource and the LinqDataSource are not considered because the data logic is far too complex to combine parts of it with the presentation logic. The AccessDataSource and the XMLDataSource are not needed because at the moment CAN data is neither stored in Access databases nor in XML files. The SiteMapDataSource is used to connect the Web.sitemap file, which describes the navigational structure of the CORA, with the primary navigation menu. The ObjectDataSource is the only data source control which is recommended as an approach for large-scale professional Web applications because it basically supports a layered architecture [14]. Nevertheless, after evaluating the possibilities of the ObjectDataSource it became clear that this data source control was not able to satisfy the requirements of the CORA either. First of all, all objects that are intended to be updated by the control are required to have a certain appearance like a default, parameterless constructor. In addition to this the ObjectDataSource suffers from the same limitation like all the other data source controls, namely that it is not possible to explicitly handle the creation of the data object which is bound to the control. This causes a lot of problems, for instance the problem of adding extra items and the problem of handling the logic when the extra items are selected. For example, it is impossible to create a data source populated drop down-list that does not have a selected item (unless it is empty) [14]. It is inescapable to place a "Select item" on top of a drop down-list whose item selection triggers complex and resource intensive operations. For instance, a "Select CAN module" entry avoids that the entire hardware configuration is automatically loaded for the first item in the populated drop down-list when the user simply opens the Web site. There exist workarounds for many data source related problems but in the end the complete abandonment of data source controls and the direct data binding of collections of custom objects were selected as main approach. In this case all the object-oriented features can come into play and data binding is used to a reasonable extent as described in chapter 6.

1.4. Layered Architecture

Rapid Application Development enables the creation of solutions in an unprecedented short time when used correctly, but to encounter the mentioned limitations the underlying application architecture must always be the center of attraction. Otherwise the quickly created architecture cannot satisfy new requirements and the whole application has to be replaced. The term "architecture" can be described in numerous ways, including the following definition from [2]:

> The architecture of a software system consists of its structures, the decomposition into components, and their interfaces and relationships. It describes both the static and the dynamic aspects of that software system, so that it can be considered a building design and flowchart for a software product.

Moreover, not only functional requirements defined by users and other stakeholders influence the architecture but additional quality considerations like performance and scalability. In addition to this, existing architecture, patterns and technical aspects are subjects to change and therefore a sophisticated architecture should be able to deal with significant changes in the problem domain. An established method to deal with a very complex system is its decomposition into smaller, less complex components. This is the fundamental idea of a layered architecture where the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer. Furthermore, each layer usually hides its lower layers from the layers above.

In [7] the benefits of a layered system are described as the following:

- A single layer can be understood without knowing much about the other layers.
- A single layer can be substituted with an alternative implementation. For instance, if the fundamental database changes, only the Data Access Layer has to be adapted as long as it passes the appropriate information to the Business Layer. On the other hand, if another fronted for displaying data is needed, only a new Presentation Layer which uses the existing Data Access and Business Layer has to be developed.
- Between layers dependencies are minimized.
- Layers make a good place for standardization.
- Once a layer is built, it can be used for many higher-level services.

On the other hand there are some downsides in [7] described as well:

- Sometimes cascading changes are necessary. For example when a field needs to be displayed in the user interface, all the layers down to the database have to be altered as well.
- Additional layers can harm performance.

Web applications often use a three- or multi-layered architecture. For instance, in one layer the data is stored, another layer retrieves and prepares the data for further processing in the next layer which applies certain business functionality or domain logic. Finally a controller layer prepares the data for the presentation layer, which provides user interaction. Table 1.1 describes the five principal layers.

Layer	**Responsibilities**
Presentation	Provision of services, display of information (e.g., in Windows or HTML, handling of user request (mouse clicks, keyboard hits), HTTP requests, command-line invocations, batch API)
Controller	Linkage between presentation and domain layer. Prepare the data to a requested presentation, like HTML, Swing or other technologies.
Domain	Logic that is the real point of the system
Data Mapping	Linkage between domain and data source layer
Data Source	Communication with databases, messaging systems, transaction managers, other packages

Table 1.1.: Five principal layers [4], [7]

The CAN data management application called CORA uses a layered architecture as described in figure 1.3. A Microsoft SQL Server 2005 acts as data source and can be configured through Microsoft SQL Server Management Studio. The Controllers in the Data Access Layer are responsible for retrieving and manipulating data where LINQ to SQL is used to bridge the gap between the relational data stored in the database and the object-oriented world of the accessing solution. Data Transfer Objects are used to transport the data through the layers. The Business Layer adds business functionality and is primarily responsible for user administration, import- and export functionality. The Presentation Layer uses the code-behind model and custom server objects to present the data and to provide user interaction.

The development environment should reflect the architecture but this does not automatically mean that every layer must be an organizational unit of its own. For instance, Microsoft Visual Studio is one of the key development tools and the source for the Data Transfer Objects (DTOs) is allocated to the Visual Studio Project called Data Access Layer and not in a separate project called Common Layer. For sure the development environment can be reorganized for the future evolution of the CORA.

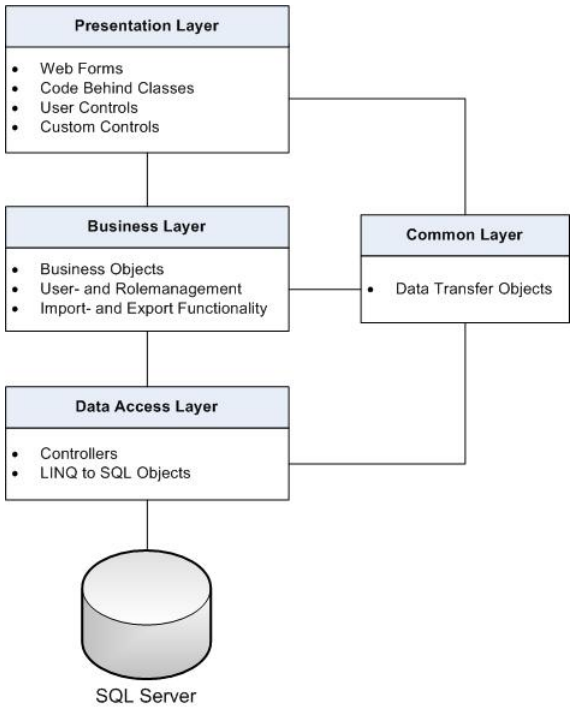

Figure 1.3.: The CORA architecture

1.5. Patterns

It is difficult to find a generally accepted definition of a pattern but [7] provides the following deceleration by Christopher Alexander.

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

One key characteristic of a pattern is that they provide a basic idea for a certain solution and have to be customized according to the given problem. Furthermore patterns are not isolated and often the usage of one particular pattern paves the way for the usage of another pattern. The boundaries between patterns are vague and the described approaches to realize the CORA are often based on numerous patterns. Nevertheless certain CORA components are often explicitly linked to widely used pattern names (regarding the highest subjective analogy) to ease communication or the finding of further resources.

1.6. Problem Statement

Managing information is a very complex task, especially when the related problem domain is very specific. For instance, the worldwide management of CAN data requires a database for storing all the information ranging from the description of a single hardware pin to the information exchange model between single CAN modules. Furthermore a solution is needed to work with the stored

data. Realizing such solutions confronts the developer with numerous recurrent challenges. Unfortunately patterns and best practice catalogs to overcome these challenges are rare to find, regarding certain technologies. For instance, in .NET and Visual Studio it is encouraged to bind data directly from the user interface to an underlying database. Especially the drag and drop facilities in Visual Studio strengthen this development approach and are well documented in numerous resources. Unfortunately this approach is bound to fail in complex enterprise scenarios where a layered architecture is needed. Resources describing the development of a layered .NET Web application by means of a detailed enterprise example are rarer to find.

1.7. Structure of the Thesis

The purpose of this thesis is to discuss architecture and design patterns of Web applications in general and especially in relation to the .NET platform. The work is based on the development of the CORA, a solution for Rosenbauer International AG to manage their CAN data worldwide. Chapter 2 introduces the company and describes how the department responsible for the CAN bus development managed their knowledge so far. The evolution from the previous data storage to the new data management is described in chapter 3. Moreover, the CORA architecture is decomposed into single layers to present patterns and practices for data access (chapter 4), for applying business logic (chapter 5) and finally for providing user interaction (chapter 6). The CORA supports multiple users including a role model to restrict certain operations. The realization of security aspects is described in chapter 7. Deploying a new solution into an existing IT-landscape is not a trivial task and therefore chapter 8 provides consideration regarding the CORA deployment. The last chapter 9 summarizes the project and points out lessons learned.

2. Worldwide Management of CAN Data

2.1. Rosenbauer International AG

The Rosenbauer Group is a worldwide manufacturer of fire-fighting vehicles with a wide range of products and services including vehicles, aerials (e.g., turntable ladders), fire-fighting components (e.g., built-in pumps) and safety equipment. Three manufacturing bases are located in North America, five in Europe and one in South East Asia. According to the company profile [25] 1800 employees achieved sales over 500 000 000 EUR in 2008. In 2002 Rosenbauer introduced the Controller–area network (CAN) to standardize the information exchange within many products and to meet the increasing complexity of the customer requirements.

2.2. What is CAN data?

When a signal lamp tells the driver of a fire-fighting vehicle that the several meters long aerial ladder is not completely retracted and can cause heavy damages while driving, some information exchange takes place. More precisely, a sender (in this case a sensor which controls the state of the ladder) sends some information over a communication channel to a receiver (in this case the warn signal lamp). To realize this information exchange standards and common interfaces are essential, for instance the receiver must use the same protocol as the sender to be able to interpret the information sent. The BOSCH company started the development of the CAN bus system for engine management at the beginning of the eighties. A bus is a system for data transfer where all communication partners are using a shared medium namely a single set of wires. As a consequence it is not necessary to wire every component with an extra cable for each function. For instance, a rear light of a fire truck needs cables for brake- and backup-light which have to be linked to the switches and lead through the whole vehicle. This means the end for obscure cable trees and results in reduced weight. Another big advantage of a CAN bus system is that every component interacts with a controller which enables self diagnosis. For example when the ignition is triggered all signal lamps are able to communicate their status and report defects to the driver. Biggest drawbacks are the required expertise and the increased complexity. Therefore a CAN bus system is basically used to realize more complex circuits. Figure 2.1 shows the CAN wiring of a fire-fighting truck. A detailed explanation concerning CAN data and further resources can be found at [29].

A lot of information and configuration is needed to operate a CAN bus system.

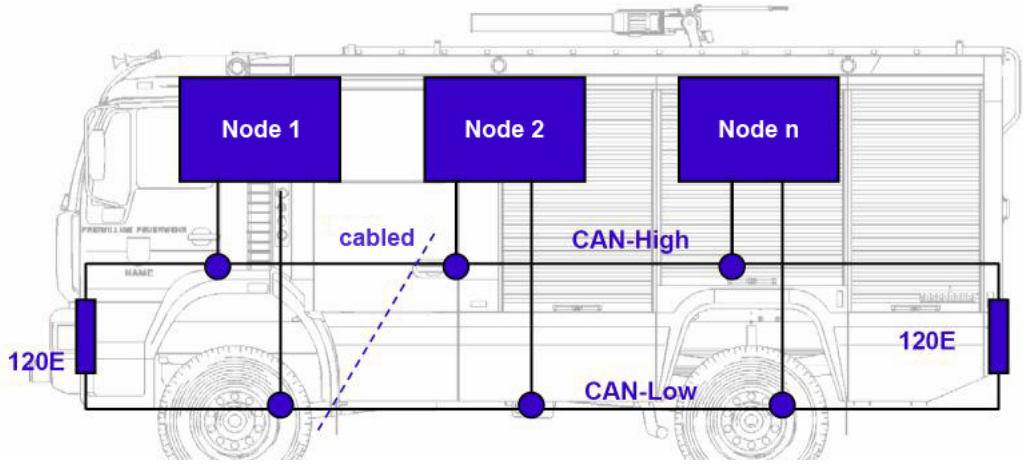

Figure 2.1.: CAN wiring of fire-fighting truck (Picture from Rosenbauer TLF AT DoKa catalog)

Programmable CAN modules with a specific hardware configuration (including connectors, pins, etc.) serve as senders and receivers and communicate with CAN messages. A single CAN message must be unequivocally identifiable and uses a carrier signal in combination with a specific byte and bit range to reach its desired destination. In order to allow an incoming message to invoke a CAN module's particular functionality, the receiver must be related to a function code which can be equipped with several parameter codes. All this information and many additional parameters are required to operate a CAN system. Sophisticated information management allows reusing a configured CAN system in another vehicle and the transfer to another production location. Consequently the management of configured CAN data is tremendously important for Rosenbauer.

2.3. How CAN data has been managed so far?

Bernhard Stadler describes the beginning of Rosenbauer's CAN data management in his work [29]. In summary, the first CAN data management solutions were file system based. More precisely every constructing engineer and every department used another tool (e.g., Excel, Word) to persist the data. The data loss caused by overwriting files was the first problem the engineering teams had to face. In addition to this, the search for specific information became harder and harder with a constantly growing amount of files. As a consequence the amount of redundant data increased as well.

The use of a centralized database system should eliminate the described problems. When all the CAN data information is integrated into a single database certain operations can be applied easily and data queried smoothly. Moreover diverse applications can access specific views of the stored data. Further benefits of database systems are consistency checks, user- and role management, transactions, synchronization and backup possibilities.

The first concepts and the resulting solution called EVI (German: Elektronisches Verwaltungs Instrument) for managing Rosenbauer's CAN data can be found in [29], including the database schema and entities definition. The IT landscape of Rosenbauer is based on the Microsoft product line and therefore the database and the information system were implemented as an Access application. The EVI abolished the problems of the file based information management but the spreading use of the CAN bus technology lead to new challenges.

2.4. Requirements for a Web-based CAN Management System

- Consider multiple CAN bus systems: The EVI was only designed for a single CAN bus and therefore a new Access database was needed for every CAN bus. This again raises the problem of data redundancy because some entities like a connector can be used on multiple CAN bus systems. Furthermore a lot of different database files are a lot harder to maintain.
- Consider multiple production locations: Every production location needs to manage its own CAN bus data but should be able to retrieve information from other locations as well.
- Worldwide accessibility: Although Microsoft Access solutions can be operated in a distributed environment they cannot compete with the accessibility offered by Web technologies. The new CAN management solution must be accessible from all over the world with different devices and multiple platforms. Therefore most established browsers have to be supported.
- User- and role management: Additional roles are needed for the worldwide CAN data management as described in section 7.1.
- Internationalization: The CAN data itself and the user interface must support multiple languages. It must be able to easily add new languages to the system without changing the database schema or a lot of the programming code.
- History support: For specific entities all the applied changes have to be traceable.
- Additional input-, import-, export possibilities for remaining file based data: Some information is still kept in comma separated value files (csv files) and should be integrated into the new database system as well.

In addition improved search functionality and certain changes for specific CAN bus entities were requested. With these significant new requirements the evolution of the old database schema and new technologies were out of the question. The whole CAN bus data management of Rosenbauer International AG needed to be revolutionized.

3. Data Management

Before the revolution in Rosenbauer's CAN data management was able to take place in form of the CORA, the fundamental database and the data itself had to be updated to fit into the new system. But database changes are not limited to introductions of new solutions. On the contrary it is likely that database systems (DBS) evolve according to the evolvement of real world systems. For instance, with the ongoing development of the CAN bus system it is inescapable to make changes to the database. According to [33] the evolvement of a database system can be classified along two independent dimensions: the level of abstraction and the transformation mode. The former refers to the data modeling levels and the latter to the handling of their changes. According to this classification one can distinguish between:

Instance evolution and instance versioning. Although the database content is frequently modified by diverse database operations a consistent state must be always guaranteed. Most of the traditional DBS offer mechanisms for instance evolution like transactions, which ensure that data gets only changed in meaningful units. For example when a new CAN node is inserted to the database all the corresponding identifiers are created as well. If the system crashes during these operations the database will be rolled back to the previous state. On the other hand if all operations are successfully completed the original data becomes inaccessible. Therefore instance versioning often has to be implemented manually.

Schema Evolution and schema versioning. The database schema has to change accordingly to the development of information which has to be stored persistently. Schema Evolution deals with the management of modifications at the schema level and the resulting changes at the database level. For example the CORA completely supports internationalization and therefore all the language specific information has to be stored in an extra table. In the case of CAN data management, schema evolution plays the most important role and will therefore be described in more detail in the next section. Schema versioning is not supported and therefore old instances must be converted to conform the accurate schema.

Data model evolution and data model versioning. The structure and behavior of database schemas are defined on the data model level. It is highly unlikely that Rosenbauer will switch to a database system that does not support previous schemas and therefore this topic will not be covered in this paper. Please refer to [33] for further resources.

3.1. Schema Evolution

A database system is able to respond to changes in the real world by allowing the database schema to evolve. This ability can be referred as Schema Evolution. Moreover, as mentioned above, Schema Evolution deals with the management of modifications. The reasons for modifications can be manifold, reaching from changes in the real world, to requirement changes or mistakes during the schema design. Requirement changes were the main reason for the schema evolution from the EVI to the CORA but improvements to existing entities were realized as well. There exist three principle lines for approaching Schema Evolution according to [1].

1. Previous states are not considered: The first approach does not retain the pre-modification state. Therefore each database schema change is applied irreversibly without taking possible consequences to the data into account. This approach is only applicable during design phase.

2. Previous states are not considered but data is converted to correspond to the new schema. Again the evolution of the database is not controlled and the data in its old structure will be lost.

3. State of the schema before modification is conserved.

There are two ways to realize the third approach. The first one is the so called historical approach. Every change in the database model creates a new version and each version is kept along with the related data. The versions are stored independently. Only the current version is allowed to be changed. The parallel approach on the other hand enables a common storage of all schema versions and common operations on the same data collection but is very complex to implement. In case of the CORA the second principle was used. The database schema had been renewed and the existing data migrated.

3.2. Database Schema

The database schema reflects the real-world entities, elaborated during the requirement specifications and the database design. Data types and relationships between entities are fundamental aspects of every database schema. For complex schemas it is highly recommended to use a model in combination with a tool to visualize the schema. Although a database schema basically describes the structure of a database, it directly impacts the Data Access Layer (DAL) of database accessing applications as well. As a consequence the schema creation possibilities of the DAL technologies (e.g., object-relational mapper frameworks) should be evaluated in the schema planning phase as well. As described in chapter 4 LINQ to SQL is used as object-relational mapper and offers schema generation possibilities. However, when data is inserted in the created database it cannot be altered with the LINQ to SQL designer anymore. As a consequence the database schema has to be designed separately from the application model with established tools.

For the design of the CORA database schema Microsoft Visio for Enterprise Architects was used. The drafted model was then synchronized with the Microsoft SQL server. For sure one can omit Visio and design the database directly in Microsoft SQL Server Management Studio with the help of database diagrams. The major disadvantages are the limited presentation possibilities and the on the fly manipulation of database components. At the time this thesis was written, there was no Visio for Enterprise Architects version available that had installed Visual Studio 2008 instead of Visual Studio 2005. Modifying the windows registry provides a workaround, detailed instructions can be found at [6]. Visio allows the generation of scripts and the direct execution of database commands as well. The synchronization works in both directions so it is possible to apply changes in Visio and in SQL Server Management Studio. The evaluation of the CORA database schema creation results in the following best practice:

1. Generation of the initial schema with Visio into a script.
2. Loading and execution of the script in SQL Server Management Studio.
3. Applying all the changes in SQL Server Management Studio and update the model in Visio.

The reasons for that approach are simple. With the help of a standardized script the database schema creation is decoupled from the original tool. With the generated script the initial state of the database can be created without touching Visio again. Applying changes directly in SQL Server Management Studio has the advantage that developers can use all the security mechanism and the advanced features of the SQL server. For example several tables can be altered at once with the help of T-SQL and it is impossible to apply incompatible data types. In Visio model validation is only triggered automatically before the update process. If modifications are not compatible cause of existing data the update process will fail and the model has to be adapted manually.

3.2.1. Modification based on new requirements

The existing database schema had to be modified to meet new requirements described in Requirements for a Web-based CAN Management System (2.4).

Multiple CAN bus systems. As described in figure 3.1 two entities called CANBus and CANBusInfo represent a CAN bus. The former entity encapsulates the data which describes a specific CAN bus (e.g., id and name) and uses the latter to describe general characteristics like the protocol, bits and frequency. The separation into two entities can be easily explained because several CAN buses can have the same characteristics. The entire CAN bus specific entities can reference the CAN bus using a foreign key. Some entities are uniquely identified because of their CAN bus affiliation and therefore the referenced CANBusID becomes part of their primary key. For instance, a CAN module with a specific item number (e.g., in German Teilenummer) has a composite primary key in contrast to a signal with an auto-generated identifier.

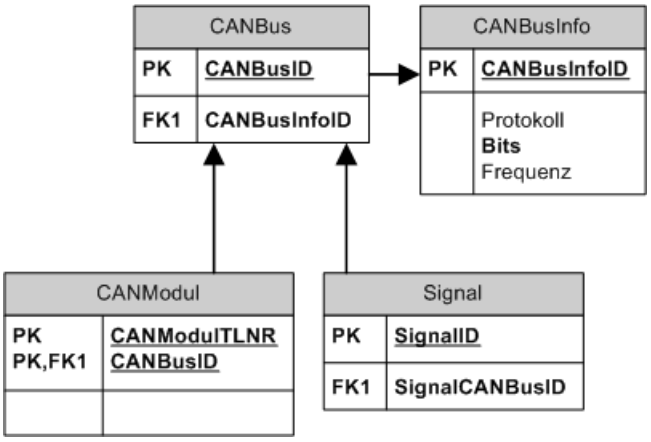

Figure 3.1.: Database schema for multiple CAN Bus support

Multiple production locations. As shown in figure 3.2 the introduction of a new entity called Standort (German for location) assigns a CAN bus clearly to a specific production location.

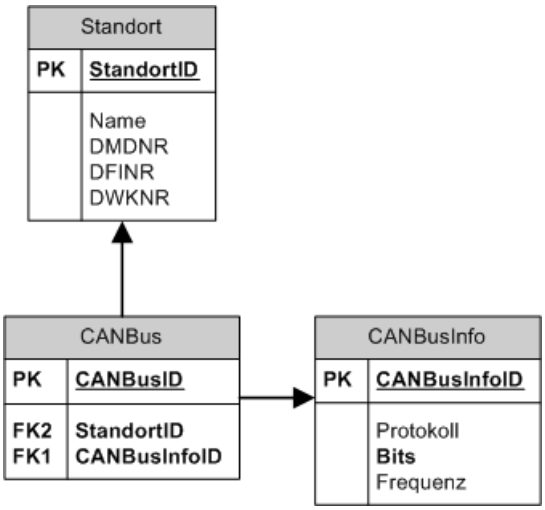

Figure 3.2.: Database schema for multiple production locations

User- and role- management ASP.NET provides built-in security mechanisms including user- and role- management. Using these features would result in automatically generated database entities. However the security requirements are too specific and therefore the built-in security mechanisms have been extended, as described in chapter 7. According to the requirements a user (German: Benutzer) must be assigned a specific role (German: Rolle) for a certain location. The combination of these three entities generates a new right (German: Recht). In the case of a CAN bus administrator not all the CAN buses belonging to a specific location are automatically manageable. On the contrary, the design enables the selection of single CAN buses for a CAN bus administrator. Figure 3.3 describes the designed entities.

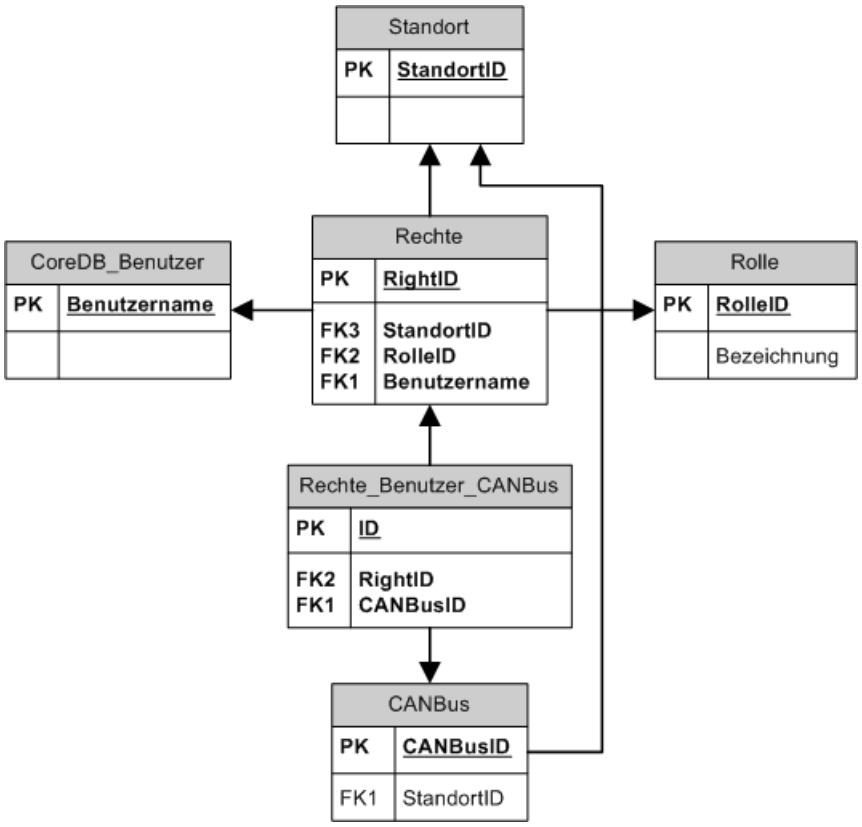

Figure 3.3.: Database schema for role management

Internationalization. A properly-designed database schema supports future language additions with no changes to the schema. An internationalized catalog schema is described in [31]. According to this approach every internationalized entity has an additional entity (a "details entity") to store localized data. As a consequence adding or removing language specific data can be achieved with a simple insert or delete operation. In the EVI database every language is represented through an extra column but the CORA database design described in figure 3.4 allows a flexible language management.

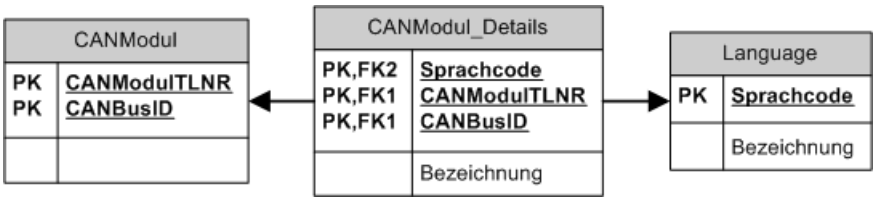

Figure 3.4.: New internationalized schema

History support Many different users are performing operations on the same entity and therefore every create-, update- and delete- operation is logged to a history entity. This entity includes all the original fields and adds the following information:

- Time stamp: Describes when the operation took place. The field HistoryCreated indicates when the history entry was created
- User: The field HistoryBenutzer contains the user who performed the operation.
- Comment: The HistoryComment field describes the operation.

Consequently the database provides a very slim variant of instance versioning.

3.2.2. Improvements

Even highly elaborated database schemes are likely to evolute after some time, not only because of new requirements but also because of reviewing the database schema again during a new development life cycle. So while reviewing the entity relationship diagram of the original Access database the following improvements had been made to the database schema.

Separation of Function- and Parameter code. The table dbo_FunktionCode contains twenty columns to describe a parameter for a function code on a specific position. This parameter itself is a code with a function code greater than 9000 and without input- or output characteristics. So function- and parameter codes are inserted into the table dbo_FunktionCode but the latter does not need the Eingang and Ausgang columns. In addition to this the table references itself to assign a parameter- to a function code. Moreover, if more than ten parameters are needed the database schema has to be modified. The introduction of an extra entity called Parameter puts things right. Now function- and parameter codes can be described exactly by the really needed fields and a connection table realizes the many to many relationship. The position field allows a variable amount of parameters. The improvement to the database schema concerning the function code is described in 3.5.

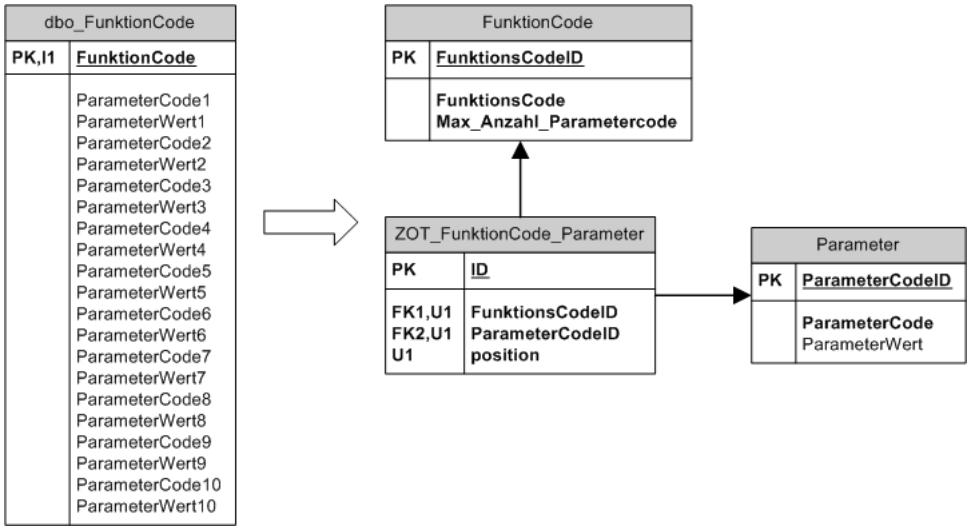

Figure 3.5.: Schema Evolution function code

Introducing the CAN message entity (German: CAN Nachricht). A CAN message is basically represented by an identifier, a certain byte- and bit range and a carrier signal. In the original database schema a CAN message entity does not exist. The byte- and bit range is specified by a single column called Bits and separated by a delimiter. So the value 2.3 indicates that byte 2 and bit 3 are used. Furthermore every table which represents an association between a function code and signal has to specify the Identifier and the Bits column. The introduction of a CAN message entity makes the association between the function codes and signals clearer and supports object-oriented thinking. In the remodeled database schema a CAN message uses an identifier and a signal. The Bits column is split into two atomic fields namely BytePos and BitPos which eases query possibilities. Finally the CAN message is associated with the function code. The Modus flag combines the two tables for read and write operations into one single association table. The improvement to the database schema concerning the CAN message is described in 3.6.

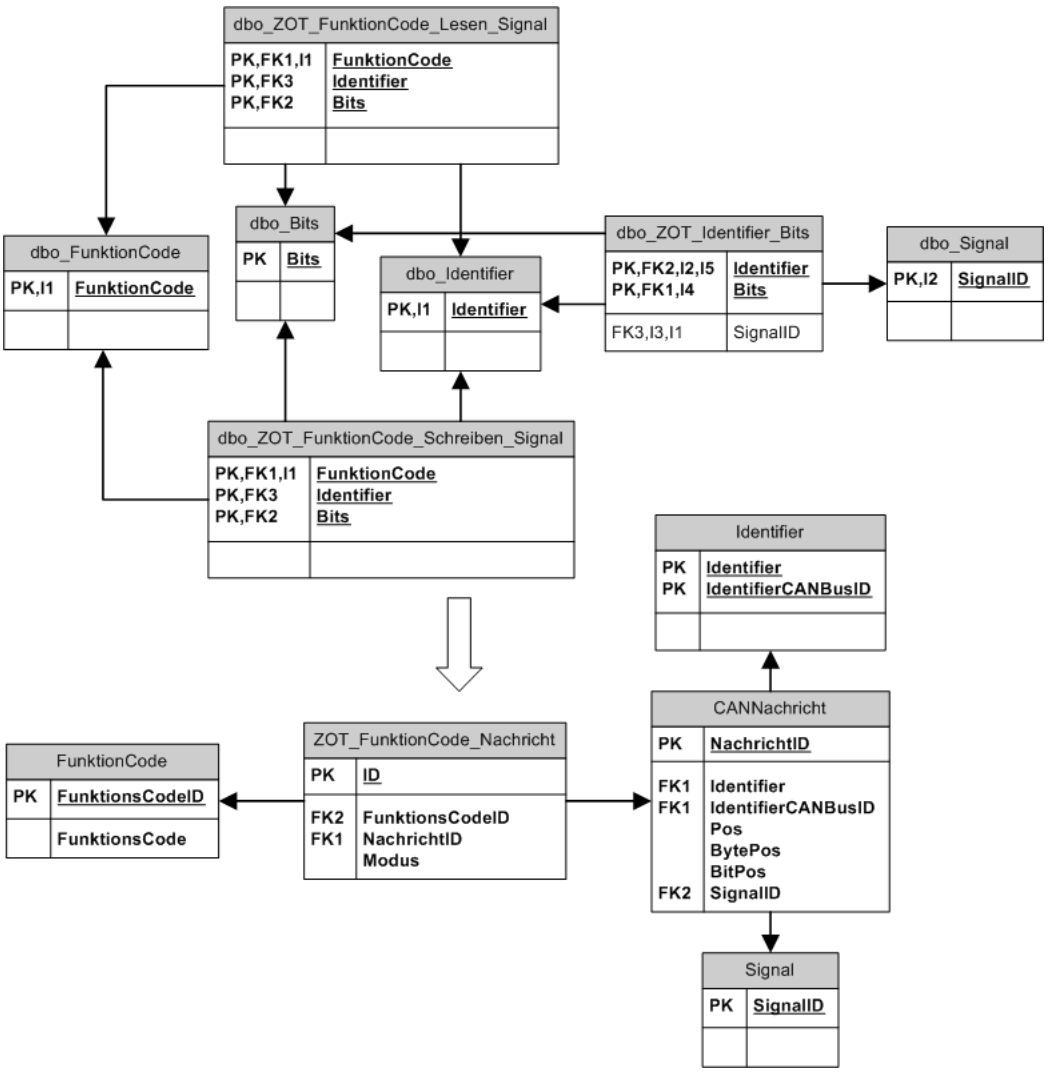

Figure 3.6.: Schema Evolution CAN message

Introducing the measurement entity(German: Maßeinheit). In the original database model a signal has six columns describing metric- and inch measurement characteristics. If a measurement applies for metric and inch all the values have to be inserted. If another measurement system is needed several extra columns are needed. The introduction of a measurement (German: Maßeinheit) entity allows to specify a measurement system for every measurement range. For instance, the min value "on" and the max value "off" apply to metric and inch and therefore the signal only has to reference these measurements twice instead of inserting the value twice. If an additional measurement system is needed only one foreign key column has to be added to the signal. Moreover a many to many association table could connect a signal with an arbitrary number of measurement systems. The improvement to the database schema concerning the measurement units is described in 3.7.

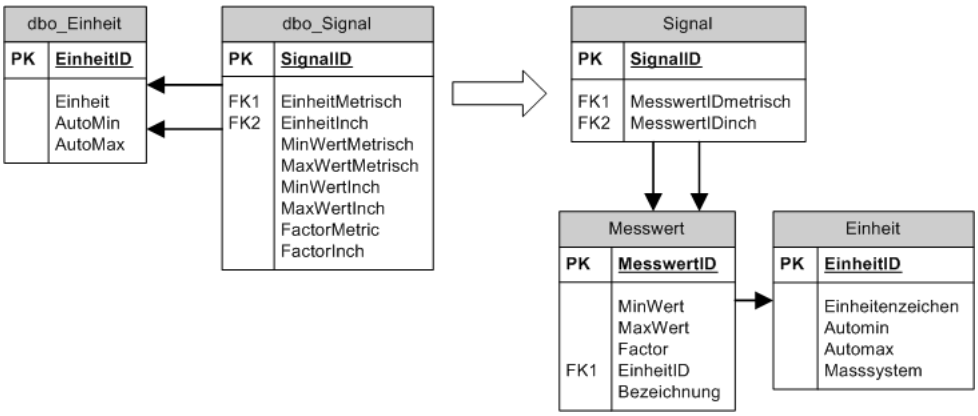

Figure 3.7.: Schema Evolution measurement units

3.2.3. Dealing with further Schema Evolution

As described earlier, two tools supported the development of the CORA database schema, namely Microsoft Visio for Enterprise Architects and Microsoft SQL Server Management Studio. The third tool called LINQ to SQL designer, which bridges the gap between the relational database and the object-oriented application, is described in detail in chapter 4. In summary, at the moment the CORA database schema is presented by three different tools.

1. Microsoft Visio for Enterprise Architects
2. Microsoft SQL Server Management Studio
3. LINQ to SQL designer

The schema modification possibilities of these tools differ in many ways. According to [32] it is possible to create the initial version of the database schema with the LINQ to SQL designer, but once the database is populated with live data, it can no longer be changed by simply re-creating the entire database.

Therefore the LINQ to SQL designer only supports the model-first or model-driven approach, but not ongoing development. Consequently two tools remain as a starting point to realize changes in the database schema. As described in the beginning of this chapter, the development of the CORA database schema has proven that Visio is great for the initial creation of the database and that the Management Studio is more suitable for changes. On the grounds that SQL Server Management Studio is the primary tool for managing the schema of the database engine possible synchronization errors caused by applied changes in Visio can be excluded. Changes applied in the Server Management Studio may only result in unsatisfactory model presentation in Visio after the synchronization, but do not harm any stored data. Furthermore T-SQL, also known as Transact-SQL, a proprietary extension to SQL for the Microsoft SQL Server products, enables complex schema modification possibilities. For instance, by now the CORA database schema identifies a language by an ISO 639 two-letter lowercase culture code. So "en" stands for English and "de" for German. Consequently the CORA makes no difference between American English and British English. This was an explicit requirement to prevent additional translation work. The column to store the ISO 639 language code has the exact length of two. A possible scenario would be the extension of the language field to three, to meet the needs of ISO 639-2 (e.g., "eng" for English), or to five to allow the storage of language codes according to ISO 639 in combination with ISO 3166 (two-letter uppercase subculture code associated with a country or region like "en-US"). According to the internationalization pattern 3.2.1 the language specific data of every entity is stored in an extra table. In the case of the CORA this means that 24 detail tables have to be updated. This is a time consuming task, regardless if the changes are applied with the help of a database diagram representation or with the dialogs of the object explorer. A faster way to realize the desired modifications provides the usage of the following T-SQL script. The fact that the language code is part of a composite primary key adds additional complexity. In order to be able to alter the table column, the primary key constraint has to be dropped first and then added again.

Listing 3.1: Modify database schema with T-SQL

```
1  declare @tblName varchar(200)
2  declare @colName varchar(200)
3  declare @dataType varchar(200)
4  declare @charMaxLength varchar(200)
5  declare cur cursor for SELECT TABLE_NAME, COLUMN_NAME, DATA_TYPE,
       CHARACTER_MAXIMUM_LENGTH FROM information_schema.columns WHERE TABLE_NAME
       LIKE '%_Details'
6  open cur
7
8  fetch Next from cur into @tblName, @colName, @dataType, @charMaxLength
9  while @@fetch_status = 0
10 begin
11 if(@colName = 'Sprachcode')
12 begin
13   declare @con_name varchar(500)
14   declare @con_colname varchar(500)
15   declare @sql varchar(500)
16   declare @cur_pk_size int
17   SET @sql = 'alter␣table␣"'+ @tblName+'"␣add␣constraint␣'
18   -- retrieve and save the primarykey constraint
19   declare cur_pk cursor for SELECT K.CONSTRAINT_NAME, K.COLUMN_NAME FROM
         INFORMATION_SCHEMA.KEY_COLUMN_USAGE K INNER JOIN INFORMATION_SCHEMA.
```

```
        TABLE_CONSTRAINTS C ON K.CONSTRAINT_NAME = C.CONSTRAINT_NAME WHERE C.
        CONSTRAINT_TYPE = 'PRIMARY KEY' AND K.TABLE_NAME = @tblName
20    select @cur_pk_size = COUNT(*) FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE K
        INNER JOIN INFORMATION_SCHEMA.TABLE_CONSTRAINTS C ON K.CONSTRAINT_NAME = C
        .CONSTRAINT_NAME WHERE C.CONSTRAINT_TYPE = 'PRIMARY KEY' AND K.TABLE_NAME
        = @tblName
21    open cur_pk
22    Declare @counter int
23    set @counter=0
24    fetch next from cur_pk into @con_name, @con_colname
25    while @@fetch_status = 0
26    begin
27      if(@counter=0)
28      begin
29        SET @sql = @sql+'"'+@con_name+'" primary key ('
30      end
31      if(@counter+1=@cur_pk_size)
32        SET @sql = @sql+'"'+@con_colname+'")'
33      else
34        SET @sql = @sql+'"'+@con_colname+'",'
35      set @counter=@counter+1
36      fetch next from cur_pk into @con_name, @con_colname
37    end
38    CLOSE cur_pk
39    DEALLOCATE cur_pk
40    exec('ALTER TABLE "' +  @tblName + '" DROP CONSTRAINT "'+@con_name+'"')
41    --alter the desired column
42    exec('ALTER TABLE "' +  @tblName + '" ALTER COLUMN Sprachcode char(5) not null
        ')
43    --add primary key again
44    exec(@sql)
45    --print @sql
46  end --ende vom if
47  fetch next from cur into @tblName, @colName, @dataType, @charMaxLength
48  end
49  CLOSE cur
50  DEALLOCATE cur
```

After the modification of the database schema the related applications have to be updated as well. In case of a layered architecture only the Data Access Layer has to be updated. Unfortunately LINQ to SQL does not provide any synchronization mechanism and therefore the changes to the LINQ to SQL entity model have to be done manually. If a lot of tables have been altered (like in the example with the language code) and the LINQ to SQL entity model represents nearly exactly the SQL database schema and the few differences are well documented, the entire entity model can be recreated from the database schema.

The SQL Server Management Studio database diagrams can present the Schema Evolution caused by the evolution of the language code but Microsoft Visio provides a richer set of model presentation possibilities. The following steps reverse engineer the modified database schema.

1. Open a new database drawing (e.g., Database Model Diagram Metric).
2. Select Database → Reverse Engineer... and create or choose an existing ODBC data source. Worth mentioning is the fact that it is possible to create an ODBC data source at this point but modifications and deletion must be performed in the Microsoft Windows system settings.
3. Select the objects which should be imported.
4. Arrange the object on the diagram (relationship are drawn automatically).

Another example where T-SQL becomes extremely handy is the implementation of the history pattern 3.2.1. According to this pattern every table is associated with an extra history table to persist old versions of stored data with additional meta data. Of course these tables are not created by hand, the 63 history tables of the CORA database schema were created with the help of a script generated by the following T-SQL script:

Listing 3.2: Create history tables

```
declare @tblName nvarchar(200)
declare @table_type nvarchar(200)

declare cur cursor for SELECT TABLE_NAME, table_type FROM information_schema.
    tables WHERE table_type = 'BASE␣TABLE' order by table_name
open cur
fetch Next from cur into @tblName, @table_type
while @@fetch_status = 0
begin
    declare @sql nvarchar(max)
    SET @sql = 'GO' + char(10) + 'CREATE␣TABLE␣[dbo].['+@tblName+'_History](
␣␣[HistoryID]␣[int]␣IDENTITY(1,1)␣NOT␣NULL,
␣␣[HistoryCreated]␣[datetime]␣NOT␣NULL,
␣␣[HistoryComment]␣[nvarchar](250)␣COLLATE␣Latin1_General_CI_AS␣NULL,
␣␣[HistoryBenutzer]␣[nvarchar](100)␣COLLATE␣Latin1_General_CI_AS␣NOT␣NULL,';
  SET @sql = @sql +char(10);
  declare @colName nvarchar(200)
  declare @dataType nvarchar(200)
  declare @charMaxLength nvarchar(200)
  declare @nullable nvarchar(200)
  declare cur_col cursor for SELECT COLUMN_NAME, DATA_TYPE, isnull(
      CHARACTER_MAXIMUM_LENGTH,-1), IS_NULLABLE FROM information_schema.columns
      WHERE TABLE_NAME = @tblName
  open cur_col
  fetch Next from cur_col into @colName, @dataType, @charMaxLength, @nullable
  while @@fetch_status = 0
  begin
    if(@charMaxLength=-1)
      SET @charMaxLength = ''
    else
      SET @charMaxLength = '('+@charMaxLength+')'
    if(@nullable = 'YES')
      SET @nullable = 'NULL'
    else
      SET @nullable = 'NOT␣NULL'
    SET @sql = @sql + '␣[' + @colName + ']␣['+ @dataType + ']␣' + @charMaxLength
        +'␣'+@nullable+','+char(10)
  fetch next from cur_col into @colName, @dataType, @charMaxLength, @nullable
  end
  CLOSE cur_col
  DEALLOCATE cur_col

  SET @sql = @sql + 'CONSTRAINT␣['+@tblName+'_History_PK]␣PRIMARY␣KEY␣CLUSTERED
␣␣␣␣(
␣␣␣␣␣␣[HistoryID]␣ASC
␣␣␣␣)WITH␣(PAD_INDEX␣␣=␣OFF,␣IGNORE_DUP_KEY␣=␣OFF)␣ON␣[PRIMARY]
␣␣)␣ON␣[PRIMARY]'
  SET @sql = @sql +';'+char(10);
  print @sql
fetch next from cur into @tblName, @table_type
end
CLOSE cur
DEALLOCATE cur
```

The evolution of the CORA database schema will constantly go on, but when the design of the database schema is completed so far to meet the actual requirements, the database is ready to store the old data.

3.3. Data Migration

According to [11] the term database migration is defined as the following:

> Database migration is the process of mapping a database application from a source DBS to a target DBS. The migration process consists of a set of conversion operations or conversion techniques that are applied to the source database application and result in a target database application.

Furthermore [11] distinguishes two parts in a database application, namely the database interaction part and the computation part. The former is usually formulated in SQL or other high-level data definition and manipulation language but embedded in the application code (In case of a layered architecture in a separate layer). The computation part is programmed in a specific programming language and applies the application logic. In general data migration approaches benefit from this separation because the database queries are isolated. In case of the update from the EVI to the CORA this separation does not play an important role because the data access logic was completely rewritten.

The CAN data existing in several Microsoft Access databases had to be migrated into a single database on Microsoft SQL Server 2005 with a new database schema. During the development of the old CAN data management application (EVI) a conversion to SQL server had already been taken into account as shown in figure 3.8. As a consequence the data had been separated from the application logic into a separate Microsoft Access file.

Figure 3.8.: Outlook for the EVI development in [29]

Converting an Access database to a SQL Server is straight forward by using the built in Upsizing Wizard. Consequently the challenges of migrating data between two different database technologies can be avoided. Moreover, the use of T-SQL scripts to migrate the upsized data to the final single target database seemed to be reasonable at a first glance, but the following challenges substantiate the need for a more sophisticated migration solution.

3.3.1. Challenge of combining multiple independent databases.

The fact that the EVI uses a single database for each CAN bus bears the challenge of migrating redundant data. For instance, a specific connector exists on

"CAN bus A" and on "CAN bus B". In the old CAN bus management solution the database storing the CAN data of "CAN bus A" and the database storing the CAN data of "CAN bus B" include both the same connector. Simple copying of the data of both databases into the new CAN bus combining database would lead to the double insertion of the same connector. This problem only affects CAN bus independent data, or to put it differently, entities that exist on multiple CAN bus systems, for instance connectors, hardware pins or programming categories. Entities that are assigned to a specific CAN bus, like a configured CAN module, are equipped with an additional attribute to mark the CAN bus affiliation. CAN bus independent entities with an auto-generated identity provide the biggest challenge because an auto-generated identity column is only valid for the particular table. Back to the connector example, if the primary key column is an auto-generated sequential number, it cannot be used to verify if a connector had already been migrated from another database. Consequently another attribute has to be chosen to identify a connector, for instance a custom text description. In addition the auto-generated identities are often used as foreign key references and it is tremendously important not to lose any associations between tables during migration. In this case it is necessary to look up the manually chosen identity column of the associated entity and to map it to the newly generated auto id.

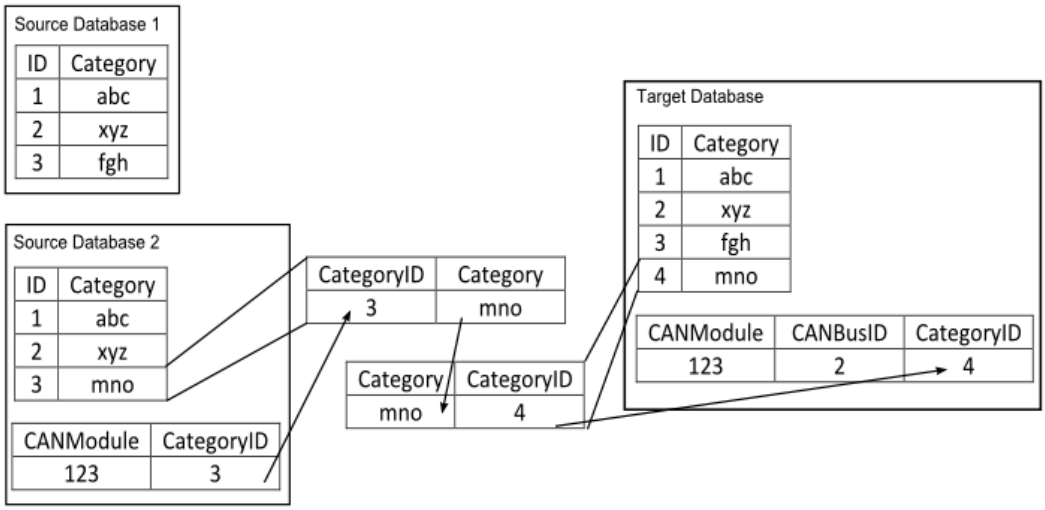

Figure 3.9.: Mapping solution for auto generated IDs

In figure 3.9 the Category entity is CAN bus independent and has an auto-generated primary key. Consequently not the ID column is used as unique identifier for the migration process but the more descriptive Category column. Therefore, four categories are migrated to the combined table and they all get a new auto-generated identity. When a CANModule entity from the source database 2 is migrated, it is necessary to retrieve the new auto-generated identifier by means of the manually chosen identifier column. Unfortunately, without explicit database constraints, the uniqueness of the manually chosen identifier column cannot be guaranteed. If a value of the Category column appears more often in the table, the mapping is bound to fail. These entities have to be migrated manually. Check routines must ensure that these entities are pointed out and do not affect the mi-

gration process. The following T-SQL statement finds multiple entries according to a specific column, in this case the column describing signals.

Listing 3.3: Find multiple entries

```
1 SELECT DISTINCT  t1.SignalID, t1.SignalBeschrDeutsch FROM dbo.dbo_Signal t1, dbo
      .dbo_Signal t2 WHERE t1.SignalBeschrDeutsch = t2.SignalBeschrDeutsch AND t1.
      SignalID  <> t2.SignalID order by t1.SignalBeschrDeutsch
```

Summarizing the complexity of the data migration, only the use of T-SQL is inefficient for the given circumstances, and the development of a separate object-oriented migration application is inescapable. Especially the possibility to express tables and columns with objects and properties endorse the following migration approach:

1. Migrate the Access database to SQL Server using the Upsizing Wizard.
2. Retrieve all the data from the source database into an object-oriented context.
3. Create new objects considering the source data and the already inserted data in the target database.
4. Insert the newly created objects into the target database.
5. Iterate the approach with the next Access database.

Bridging the gap between the relational- and the object-oriented world seems to be the biggest problem with this migration approach, also known as impedance mismatch and will be discussed in more detail in the chapter Data Access Layer (4). In the case of data migration between the EVI and the CORA the object structure is simple, because one table is reflected exactly by one object. In summary the in .NET 3.5 integrated object-relational mapper LINQ to SQL solves the problem. The whole migration approach is described in figure 3.10.

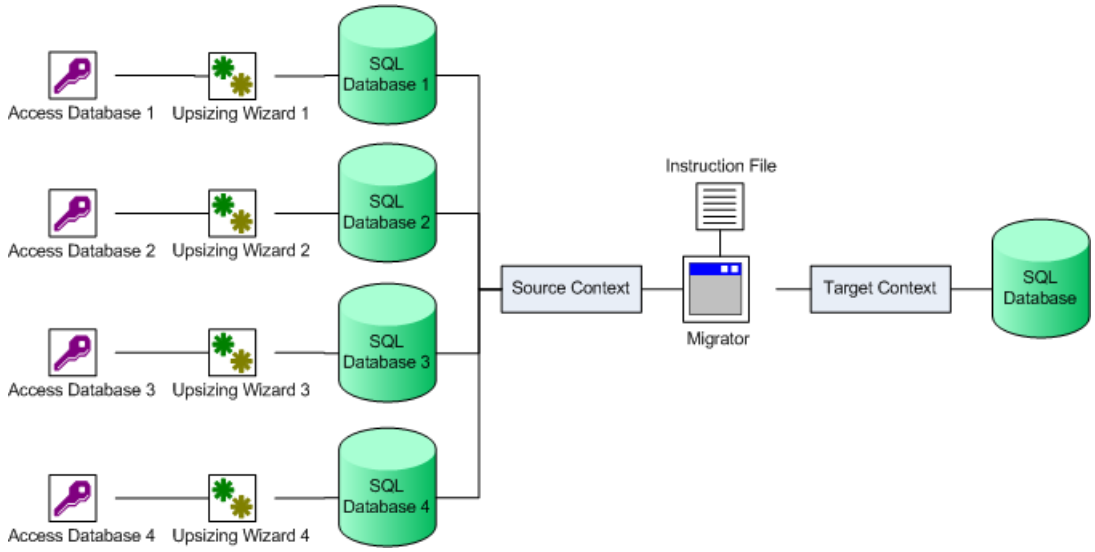

Figure 3.10.: Data migration approach

The migration application is a C# command line application organized in different classes, each of them responsible for the migration of specific tables. For instance, there exists a CANModulMigrator, a FunktionCodeMigrator or a SignalMigrator. All these classes inherit from a superclass Migrator which provides initial settings, the source- and the target data context and mapping collections. Furthermore the migration application is designed to support incremental migration. This means that single tables can be marked for migration in the instruction file and the source database can be changed in a configuration file. In other words a table by table migration is possible. This supports manual interference and keeps the cost of system resources down. Nevertheless several source databases require a migration process that takes transactional behavior into account. In many cases it is inescapable that data gets immediately written to the target database. For example, regarding the internationalization pattern in combination with an auto-generated primary key on the database side, the table for language specific data and the table for language unspecific data must represent the same entity with the help of a unique identifier. Before the primary key can be inserted in language specific data tables it has to be generated through the insertion of data into the language unspecific table. Imagine the migration process crushes exactly after the language specific data has been inserted. Then an entity without the language specific data exists on the target database. To prevent this awful scenario the migration of every single source table has to be defined as transaction. This means if something goes wrong, the state of the target database will be rolled back. After solving the errors, the migration of the table can be started again without leaving the database in an inconsistent state. Running the table migration within a transactional context is more expensive regarding resources, but there exist no applicable alternatives. One possibility would be to log every single entity migration. In case of an unexpected interruption of the migration process the log files can be consulted to check if all the data of the last touched entity has been migrated. Even if no entity has been incompletely migrated there remains the problem of the incompletely processed table, because without a transactional triggered rollback the so far migrated data remains in the target tables. In this case one could either modify the statements for retrieving the source entities to start with the right database row, or start from the beginning including validation logic which prevents an entity from getting migrated twice. This validation logic could be complex regarding the mentioned changes of primary keys during the migration. Consequently every table migration is considered as transaction. Transactions in LINQ to SQL are described in more detail in the chapter Data Access Layer (4). Regarding the data migration process it is important to know that putting the LINQ to SQL datacontext of the source and target database inside the same transaction scope requires that the "Distributed Transaction Coordinator service" is running. If not the following code listing will cause an exception that the Microsoft Distributed Transaction Coordinator (MSDTC) service is unavailable on the database server.

Listing 3.4: Transaction spanning two DataContexts

```
using (TransactionScope scope = new TransactionScope())
{
  List<dbo_CANModul> srcEntities = (from src in sourceContext.dbo_CANModuls
      select src).ToList();
  List<CANModul> targetEntities = (from target in targetContect.CANModuls select
       target).ToList();
}
```

The Distributed Transaction Coordinator can be turned on easily in the administrative tools but distributed transactions require a lot of resources. During the migration process no changes are made to the source database and therefore there is no need to make the retrieval of the source entities part of the transaction.

Listing 3.5: Transaction for one DataContext

```
List<dbo_CANModul> srcEntities = (from src in sourceContext.dbo_CANModuls select
     src).ToList();
using (TransactionScope scope = new TransactionScope())
{
  List<CANModul> targetEntities = (from target in targetContect.CANModuls select
       target).ToList();
}
```

In most cases the migration process of a single table does not exceed the standard timeout for transactions. On the other hand the migration of tables with many entries, or in cases that require many additional entries, the timeout has to be increased explicitly. For example the EVI only stored the first bit and the last bit of occupied ranges in certain relation tables. It was the job of the EVI to perform additional counting to calculate the occupied bytes and bits. The CORA stores the full byte and bit assignments, so that no additional application logic to the database queries is required to determine an occupied byte and bit range. The following code listing demonstrates a transaction scope to successfully migrate a table with a manually increased timeout.

Listing 3.6: Transaction with increased timeout

```
TransactionOptions options = new TransactionOptions();
options.Timeout = TimeSpan.FromMinutes(10);
using (TransactionScope scope = new TransactionScope(TransactionScopeOption.
    Required, options))
{
}
```

In summary an object-oriented data migration approach overcomes all the described challenges. Several migration objects provide functions to migrate the source tables to meet the new database schema.

4. Data Access Layer

The persistently stored data must be accessible and modifiable for one or more applications. Therefore the application has to be aware of the database's physical structure. In case of a relational database the data is presented in form of many related tables and columns. Regarding a layered architecture the functionality for allowing an application to interact with the database is placed in a separate layer. As a consequence many different applications can build on the same layer and code only has to be adapted at a central point if the database changes, without affecting the application logic. This means a huge maintenance benefit compared to monolithic applications where data access aspects pervade the whole application. Long story short, the role of the Data Access Layer is to handle the communication between applications and the database.

A sophisticated Data Access Layer exists of numerous components, and several patterns provide guidelines for designing them to face common data access challenges. The first challenge lies in establishing a connection to the database. Then commands in a for the database understandable language must be executed in order to perform data retrieval and data manipulation operations. The Structured Query Language (SQL) is jointly responsible for the success of relational databases and used by many applications to retrieve and manipulate data. It is recommended to separate SQL access from the domain logic and place it in separate classes which form a Gateway to the database tables. By decoupling the persistent storage implementation from the rest of your application a huge maintenance benefit is gained and interoperability is strengthened. If data access logic does not pervade the whole application changes can be applied on a central point or the whole database can be exchanged more easily without affecting code of other layers. Moreover, an approved Data Access Layer can be used by many different applications. According to [7] the gateway pattern promotes to use the following patterns.

Table Data Gateway. A Table Data Gateway is an object that handles all the rows of a table. Therefore it contains finder methods to retrieve data with the help of Record Sets (an in-memory representation of tabular data) and methods for insert-, update- and delete operations.

Row Data Gateway. A Row Data Gateway is an object that looks exactly like a row in the database and often uses a Finder Object to query data. Therefore every database table is represented by a Row Data Gateway and a Finder Object. In case of the CORA this could be a CANModule Gateway which implements fields for the item number, the programming category and additional methods for insert-,

update- and delete operations. The CANModule Finder could provide methods for returning all CAN modules without a hardware configuration.

Active Record. The next design question concerns the extent of functionality provided by the objects that represent database data. A so called Active Record carries data and behavior and can be considered as an object that wraps a database row, encapsulates database access and even adds domain logic to the data.

Data Mapper. The biggest challenge lies in bridging the database's relational world with the application's object-oriented world, also known as object-relational mismatch. Building applications according to the object paradigm means creating objects that include data and behavior. On the contrary, the relational paradigm is based on storing data in table rows. At this point object-relational mapper come into play as described in 4.1.

Data Access Object. According to [3] a Data Access Object is described as the following:

> The Data Access Object (also known simply as DAO) implements the access mechanism required to work with the data source. Regardless of what type of data source is used, the DAO always provides a uniform API to its clients. The business component that needs data access uses the simpler interface exposed by the DAO for its clients. The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this allows you to change a DAO implementation without changing the DAO client's implementation. Essentially, the DAO acts as an adapter between the component and the data source.

The CORA Controllers which are introduced in chapter 4.3 resemble the DAO pattern because they provide all the functionality for accessing data. Although not every function is exposed as an interface to the accessing Managers of the CORA Business Layer (described in chapter 5) at the moment, the Managers can simply call the method of the particular Controllers without needing any additional information concerning the data access. The structure of the Data Access Object pattern is described in figure 4.1.

In case of the CORA the Client is a Business Object of the Business Layer and uses a Controller class (DataAccessObject) which accesses a Microsoft SQL Database Server as underlying DataSource. Furthermore the Controller uses the ResultSet, which is generated by the database engine in response to LINQ to SQL generated queries, and creates a collection of DataTransferObjects. These objects transport the data to the Business Layer which applies business logics and forwards the data to the Presentation Layer.

The sequence diagram depicted in figure 4.2 shows the process and the interaction of various components of the Data Access Object pattern for retrieving

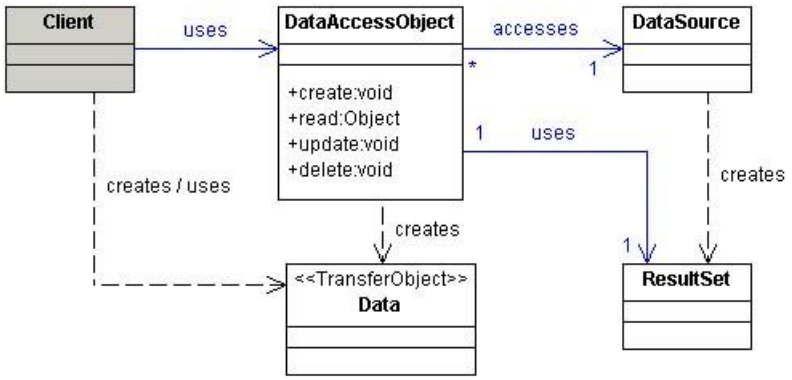

Figure 4.1.: Structure of the Data Access Object pattern [3]

data from a data source. In the constructor of the Business Objects the required DataAccessObjects are initialized. The DataAccessObjects use a LINQ to SQL DataContext, and a LINQ to SQL query (cf. section 4.2) that populates a TransferObject (cf. section 4.4), to take care of all the further described actions.

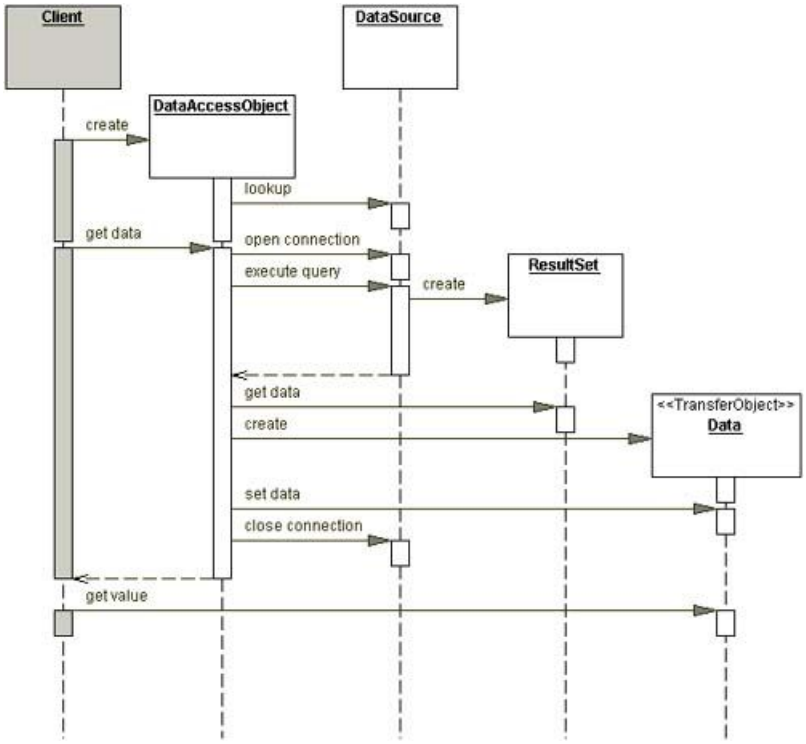

Figure 4.2.: Data Access Object sequence diagram [3]

Taking all the ideas of the mentioned patterns into account, the main components of the CORA Data Access Layer are:

- Classes generated by the object-relational mapper LINQ to SQL as described in section 4.2.
- Controllers which provide access to the database and perform data retrieval and data manipulation operations. With the help of LINQ to SQL one

Controller handles one CAN bus entity which consists of multiple tables as described in section 4.3.

- Data Transfer Objects which transport retrieved data to other layers or deliver user changed data to the DAL as described in section 4.4.

Figure 4.3 describes the basic architecture of the CORA Data Access Layer.

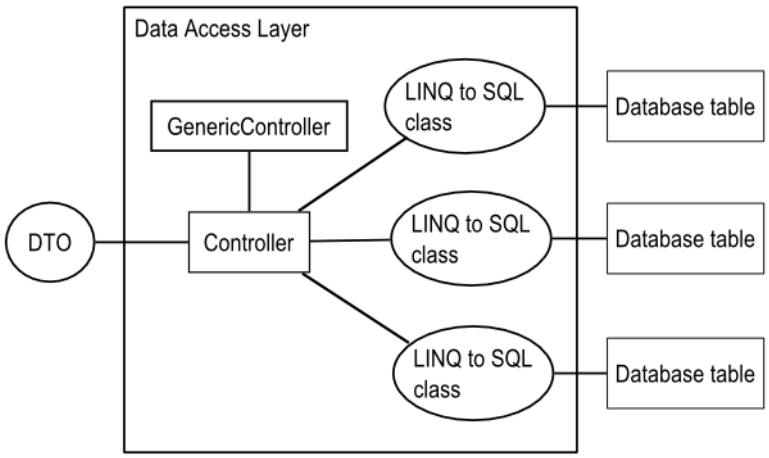

Figure 4.3.: Data mapping inside the Data Access Layer

4.1. Object-Relational Mapping

According to [7] a mapper can be defined as the following:

> An object that sets up a communication between two independent objects.

A data mapper (cf. figure 4.4) is a more concrete example of a mapper and can be defined as the following:

> A layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.

An object-relational mapper (ORM) is a special occurrence of a data mapper and can be described as a technique for converting data between relational databases and object-oriented programming languages; or to put it differently, to support automatic retrieval and persistence of data between the object-oriented- and the relational world. In many cases the mapping is realized by a model which maps the application objects to columns and tables in a database. This data mapping can be described in different formats, for example in XML. A sophisticated ORM framework has to fulfill the following criteria [4].

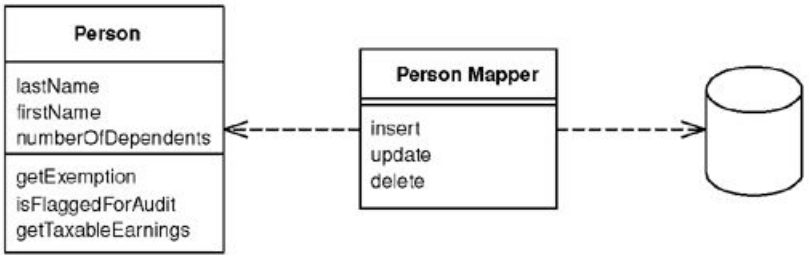

Figure 4.4.: Data mapper [7]

- An API to perform basic create-, read-, update- and delete (CRUD) operations. For instance, a developer should be able to store a collection of objects persistently in the database with a single function call and without worrying about executing numerous insert statements manually.
- A language or API to query the database and generate application objects out of the retrieved result set.
- Provide a possibility for specifying mapping meta data.
- Support advanced database features like transactions.
- Consider performance.

There exist numerous object-relational mappers for different Web development technologies with different possibilities. Some of them only map directly the database schema and other more powerful solutions are able to persist object models that use advanced object-oriented features like inheritance. Therefore a well elaborated evaluation of all the possible ORM solutions is extremely important.

Entity Framework

- Required .NET Version: 3.5 Service Pack 1
- Three layers: Source schema, conceptual layer, and the mapping layer in between.
- LINQ as integrated query language.

LINQ to SQL

- Required .NET Version: 3.5
- One-to-one mapping of tables to objects.
- LINQ as integrated query language.

NHibernate

- Required .NET Version: 1.1 or 2.0

- Open source and distributed under the GNU Lesser General Public License.
- Long established in the development community.
- No LINQ support.

Genome

- Required .NET Version: 2.0
- Supports complex mappings.
- Supports multiple databases.
- Additional license costs.

The Microsoft Entity Framework had not been officially released at the start of the CORA project and after a consultation with Rosenbauer's IT officials LINQ to SQL was chosen as object-relational mapper.

4.2. LINQ to SQL

A quick introduction to LINQ to SQL can be found at Scott Guthrie's Web log [10]. A more detailed insight is provided by [15]. Moreover, the following list of criteria from [36] substantiated the choice for LINQ to SQL.

Object identity. In a relational database every entry is uniquely identifiable with the help of a primary key. When an existing attribute is used one speaks of a meaningful or natural key. If an extra column only for identification is added one speaks of a meaningless or surrogate key. Natural key can affect the readability of the database schema in many-to-many relationships. Primary keys are not limited to a single column. For instance, in the CORA database schema every entity which is associated to a specific CAN bus and does not use a surrogate primary key has a composite key (or compound key), existing of the natural key and the CAN bus ID. Furthermore surrogate keys are often automatically generated in form of a sequence on the database side. Certainly the database ID field has to be stored in the object which represents the database entity to maintain identity between an in-memory object and database rows.

In LINQ to SQL, the so called DataContext manages the connection to the database and the object identity. The DataContext can be described as source of all entities mapped over a database connection. Therefore if the same query is executed two times, you receive a reference to the same object in memory every time as described in the following code listing [21].

Listing 4.1: Retrieving reference to same object

```
1 Customer cust1 =
2     (from cust in db.Customers
3      where cust.CustomerID == "BONAP"
4      select cust).First();
5
```

```
6  Customer cust2 =
7      (from ord in db.Orders
8       where ord.Customer.CustomerID == "BONAP"
9       select ord).First().Customer;
```

Furthermore, in case of an auto-generated ID on the database side, the value of the identity field can only be determined after the insert operation took place and the corresponding object has to be updated on the object-oriented application side. The by the LINQ to SQL designer generated entity classes can handle composite primary keys and get automatically synced in case of auto-generated identifier. This is important for the creation of language specific entities. In the CORA database schema carrier signals for CAN messages have an auto-generated ID. If a new Signal is inserted into the database the corresponding details object automatically gets the generated SignalID assigned as described in the following code listing.

Listing 4.2: Auto synchronization for ID field

```
1  using (TransactionScope scope = new TransactionScope())
2  {
3      Signal signal = new Signal();
4      DataContext.Signals.InsertOnSubmit(signal);
5      //Submit needed to get auto-generated ID
6      DataContext.SubmitChanges();
7      Signal_Detail signalDetailGer = new Signal_Detail();
8      //Due to synchronization the auto-generated SignalID can be assigned to the
           language specific entity
9      signalDetailGer.SignalID = signal.SignalID;
10     signalDetailGer.Sprachcode = "en";
11     signalDetailGer.Bezeichnung = "This␣is␣a␣testsignal";
12 }
```

Relationships. In the relational model relationships among tables are realized through column references. For a 1:n or 1:1 relation the primary key of one table is used as a foreign key in the referenced table. If primary key and foreign key contain the same value the datasets are referenced. The LINQ to SQL designer automatically recognizes the relationships for the given tables and generates the entity classes accordingly. Figure 4.5 demonstrates a 1:n relation in the LINQ to SQL designer. The automatically generated entity classes provide a property to return the related entities, stored in a collection of the type EntitySet<T>.

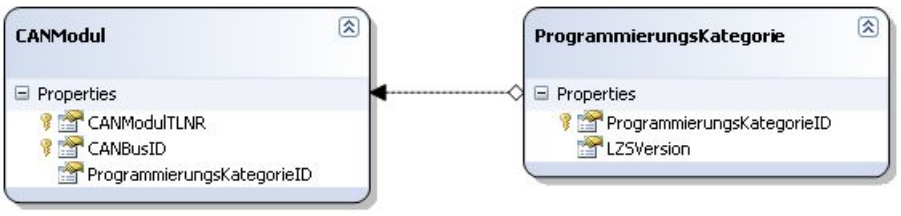

Figure 4.5.: A 1:1 or 1:n relationship displayed by the LINQ to SQL designer

As described in figure 4.6 it is easily possible to realize m:n relations with the help of an additional table. The generated LINQ to SQL classes do not provide built-in support for many to many relationships (like generated properties or

methods which return the desired result), but LINQ eases the creation of complex join queries thanks to the marvelous integration into Visual Studio.

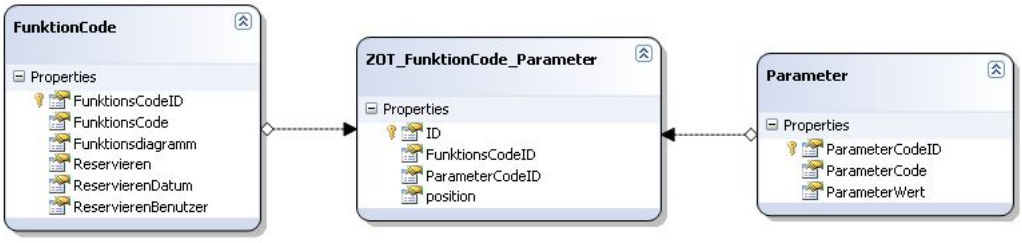

Figure 4.6.: An n:m relationship displayed by the LINQ to SQL designer

Embedded Value. Objects are often finer grained than tables in the database. As a consequence a technique is required to map an object into several fields of another object's table. A good example is provided in [7] where an employment object links to a data range object. In the object-oriented world this distinction makes sense but not necessarily in the relational world. Here the start- and the end date of the employment are directly stored in an employments table. LINQ to SQL provides the possibility to specify a storage column for every corresponding class but mapping a LINQ to SQL entity class to multiple tables is not supported [17].

Inheritance. The traditional relational model does not provide inheritance and therefore additional strategies are needed to model hierarchies as described in [7]. The CORA database schema does not use any inheritance but LINQ to SQL supports Single Table Inheritance and possible implementation approaches can be found in [15].

Lazy Loading. The Lazy Load pattern is described in [7] as an object that does not contain all the needed data but knows how to get it. By default LINQ to SQL only executes queries when the results are processed, also called deferred query execution. If immediate query execution is desired the result must be stored into an in-memory collection for example with the extension method toList().

Listing 4.3: Demonstration of Lazy Loading

```
1  var query = from signal in DataContext.Signals select signal;
2  //query is executed for the first time
3  foreach(var signal in query)
4  {
5      Console.WriteLine(signal.SignalID);
6  }
7  //query is executed again
8  foreach (var signal in query)
9  {
10     Console.WriteLine(signal.SignalID);
11 }
12
13 //query is executed immediately
14 var query2 = (from signal in DataContext.Signals select signal).ToList();
15 foreach (var signal in query2)
16 {
```

```
17      Console.WriteLine(signal.SignalID);
18  }
19  foreach (var signal in query2)
20  {
21      Console.WriteLine(signal.SignalID);
22  }
```

Connection Pooling. Creating connections is expensive in many environments and therefore requesting and releasing existing connections from a pool can improve performance in many situations. LINQ to SQL supports Connection Pooling and provides settings like the number of active connections that are managed by the pooling infrastructure.

Transactions. The challenge of combining multiple databases during the migration process (cf. section 3.3.1) introduced the LINQ to SQL capabilities to handle transactions. Before going into detail regarding the TransactionScope object it is important to know that LINQ to SQL provides a default concurrency implementation due to the object tracking functionality of the DataContext. A good example is an insert scenario which does not require any field generation on the database server side. For instance, the generation of a new CAN node triggers the generation of multiple identifier for the CAN messages. If either the generation of the node or of the identifiers fails nothing should be committed to the database. No additional programming effort is needed to guarantee this consistent state because methods like InsertOnSubmit() and InsertAllOnSubmit() only mark objects for insertion but the actual database commands are submitted when the method SubmitChanges() is called. As a consequence the CRUD methods of the Controller (classes for data access described in section 4.3) provide a parameter to indicate if SubmitChanges() should be called.

Listing 4.4: CRUD method with submit parameter

```
1  public List<TDTO> Insert(List<TDTO> dtobjects, bool submitChanges)
2  {
3  }
```

Therefore the class which is responsible for inserting a CAN node only creates the node object and marks it for insertion by passing a false boolean. The class which creates the corresponding CAN message identifier completes the insert operation by calling SubmitChanges(). This is only possible because neither the CAN node nor the CAN identifier require an auto generated column on the server side. The former uses a custom integer less or equal 64 as a node number, the latter is generated according to the node number. If an automatically created sequence by the database server is used the TransactionScope object provides transactional benefits. Moreover, using the TransactionScope does not require to explicitly begin the transaction or to roll it back. The only requirement is to complete the transaction by calling the Complete() method.

Listing 4.5: Transaction with TransactionScope

```
1  using (System.Transactions.TransactionScope scope =
2  new System.Transactions.TransactionScope())
3  {
4  context.SubmitChanges();
```

```
5  scope.Complete();
6  }
```

Furthermore nested transactions (if a method call inside a TransactionScope invokes a method which implements a TransactionScope as well) are flawlessly supported and the behavior can be specified by numerous parameters. These are the transactional mechanism used by the CORA so far, more detailed information can be found at [15] and [20].

Reuse LINQ queries. One possible solution to reuse LINQ queries is to provide functions which return IQueryable collections as show in code listing 4.6.

Listing 4.6: Returning IQueryable to enable reuse

```
1  public IQueryable<CANModulSWHWBelegungDTO> GetHWComponentsQuery()
2  {
3    StringBuilder langCondition = new StringBuilder();
4    langCondition.AppendFormat("Sprachcode ==\"{0}\"", settings.Language);
5    var query = from hb in DataContext.CANModulSW_HWBelegungs
6
7                join hbd in DataContext.CANModulSW_HWBelegung_Details.Where(
                     langCondition.ToString())
8                on hb.HWBelegungID equals hbd.HWBelegungID into dj
9                from hbdX in dj.DefaultIfEmpty()
10
11               join steckerDetails in DataContext.Stecker_Details.Where(
                     langCondition.ToString())
12               on hb.SteckerID equals steckerDetails.SteckerID into sd
13               from sdX in sd.DefaultIfEmpty()
14
15               join steckerFunktionDetails in DataContext.SteckerFunktion_Details
                     .Where(langCondition.ToString())
16               on hb.SteckerFunktionID equals steckerFunktionDetails.
                     SteckerFunktionID into sfd
17               from sfdx in sfd.DefaultIfEmpty()
18
19               join hwpb in DataContext.HardwarePinBezs
20               on hb.HardwarePinBezID equals hwpb.HardwarePinBezID into hwpbJR
21               from hwpbj in hwpbJR.DefaultIfEmpty()
22
23               join hwpbd in DataContext.HardwarePinBez_Details.Where(
                     langCondition.ToString())
24               on hb.HardwarePinBezID equals hwpbd.HardwarePinBezID into
                     hwpbetails
25               from hwpbdX in hwpbetails.DefaultIfEmpty()
26
27               join pf1 in DataContext.PinFunktion_Details.Where(langCondition.
                     ToString())
28               on hb.PinFunktionID_1 equals pf1.PinFunktionID into pfjoined
29               from pf1x in pfjoined.DefaultIfEmpty()
30
31               join pf2 in DataContext.PinFunktion_Details.Where(langCondition.
                     ToString())
32               on hb.PinFunktionID_2 equals pf2.PinFunktionID into pf2joined
33               from pf2x in pf2joined.DefaultIfEmpty()
34
35               join sf in DataContext.SteckerFunktions
36               on hb.SteckerFunktionID equals sf.SteckerFunktionID into sfjoined
37               from sfx in sfjoined.DefaultIfEmpty()
38
39               join defaultfc in DataContext.FunktionCodes
40               on hb.DefaultFunktionCodeID equals defaultfc.FunktionsCodeID into
                     defaultfcjoined
41               from defaultfcX in defaultfcjoined.DefaultIfEmpty()
42
43               join defaultfcd in DataContext.FunktionCode_Details.Where(x => x.
                     Sprachcode == settings.Language)
```

```
44                on hb.DefaultFunktionCodeID equals defaultfcd.FunktionsCodeID into
                       defaultfcdjoined
45                from defaultfcdX in defaultfcdjoined.DefaultIfEmpty()
46
47                join konfigfc in DataContext.FunktionCodes
48                on hb.KonfigFunktionCodeID equals konfigfc.FunktionsCodeID into
                      konfigfcjoined
49                from konfigfcX in konfigfcjoined.DefaultIfEmpty()
50
51                join konfigfcd in DataContext.FunktionCode_Details.Where(x => x.
                      Sprachcode == settings.Language)
52                on hb.KonfigFunktionCodeID equals konfigfcd.FunktionsCodeID into
                      konfigfcdjoined
53                from konfigfcdX in konfigfcdjoined.DefaultIfEmpty()
54
55                select new CANModulSWHWBelegungDTO
56                {
57                    HWBelegungID = hb.HWBelegungID,
58                    CANModulSWTLNR = hb.CANModulSWTLNR,
59                    CANModulSWCANBusID = hb.CANModulSWCANBusID,
60                    HardwarePin = hb.HardwarePin,
61                    SoftwarePin = hb.SoftwarePin,
62                    SteckerBez = sdX.Bezeichnung,
63                    SteckerFunktionBez = sfdx.Bezeichnung,
64                    HardwarePinBezeichnung = hwpbdX.Bezeichnung,
65                    Pinfunktion1Bez = pf1x.Bezeichnung,
66                    Pinfunktion2Bez = pf2x.Bezeichnung,
67                    SteckerFunktionID = hb.SteckerFunktionID,
68                    SteckerFunktionIndex = sfx.SteckerFunktionIndex,
69                    AnaEingang = sfx.AnaEingang,
70                    AnaAusgang = sfx.AnaAusgang,
71                    DigEingang = sfx.DigEingang,
72                    DigAusgang = sfx.DigAusgang,
73                    Sprachcode = hbdX.Sprachcode,
74                    KonfigFunktionCodeID = konfigfcX.FunktionsCodeID,
75                    KonfigFunktionCodeBez = konfigfcdX.Bezeichnung,
76                    DefaultFunktionCodeID = hb.DefaultFunktionCodeID,
77                    DefaultFunktionCode = defaultfcX.FunktionsCode,
78                    DefaultFunktionCodeBez = defaultfcdX.Bezeichnung,
79                    PinFunktionID_1 = hb.PinFunktionID_1,
80                    PinFunktionID_2 = hb.PinFunktionID_2,
81                    HardwarePinBezID = hb.HardwarePinBezID,
82                    AusgabeInCSV = hwpbj.AusgabeInCSV,
83                    SteckerID = hb.SteckerID,
84                    Bemerkung = hbdX.Bemerkung,
85                    AdditionalKonfigFunktionCode = hb.AdditionalKonfigFunktionCode
86
87                };
88    return query;
89  }
90
91  Now another function can retrieve this query and add additional filter criteria
        or search expression as described in~\ref{list:RetrieveQuery}.
92
93  \begin{lstlisting}[commentstyle=\scriptsize\ttfamily,caption=Retrieve and enrich
        existing query,label=lst:RetrieveQuery]{}
94  public List<CANModulSWHWBelegungDTO> GetHWComponentsExport(
        CANModulSWHWBelegungDTO search, string sortExp, string sortDirection)
95  {
96      var query = GetHWComponentsQuery();
97      query = from hwconfig in query
98              where hwconfig.CANModulSWTLNR == search.CANModulSWTLNR && hwconfig.
                    CANModulSWCANBusID == settings.CANBusID
99              select hwconfig;
100     if (sortExp != null && sortDirection != null) query = query.OrderBy(
            CreateMultipleOrder(sortExp, sortDirection));
101     List<CANModulSWHWBelegungDTO> result = new List<CANModulSWHWBelegungDTO>(
            query);
102     return result;
103 }
```

Combining LINQ queries. According to the CORA requirements, one single user interaction can require data from additional tables without losing existing search parameters. A good example is the listing of CAN messages for a specific CAN module. Among other search criteria the user can specify if only the CAN node or the associated function codes as well, should be taken into account to reference the CAN messages. One possibility is to create every search query explicitly, but a nicer solution is to modularly extend the existing query with additional tables which is fully supported by LINQ to SQL. The first query ensures that only CAN message identifier for a specific CAN bus are used.

Listing 4.7: Query identifier for a specific CAN bus

```
1 var identQuery = from ident in DataContext.Identifiers.Where(x => x.
      IdentifierCANBusID == settings.CANBusID) select ident;
```

This query is used in a join condition with the base query for retrieving CAN messages.

Listing 4.8: Combine Identifier query with CAN message query

```
1 var query = from cn in DataContext.CANNachrichts.Where(x => x.IdentifierCANBusID
       == settings.CANBusID)
2             join ident in identQuery
3             on new { cn.Identifier, cn.IdentifierCANBusID } equals new {
                  Identifier = ident.Identifier1, IdentifierCANBusID = ident.
                  IdentifierCANBusID }
```

With this technique LINQ queries can be combined in any imaginable way with all the language benefits of C# to realize validation. In the next code listing it is verified if function codes should be taken into account. If this is the case, all the for the CAN module approved function codes are retrieved for a specific CAN bus.

Listing 4.9: Query associated function codes

```
1 if (searchCmsw.ConsiderCodesWriting)
2 {
3 var cmsw_fc_query = from cmsw_fc in DataContext.ZOT_CANModulSW_FunktionCodes.
      Where(x => x.CANModulSWTLNR == searchCmsw.CANModulSWTLNR && x.
      CANModulSWCANBusID == settings.CANBusID) select cmsw_fc;
4 if (searchCmsw.FunktionCodeTLNRFreigabe.HasValue)
5 {
6   cmsw_fc_query = cmsw_fc_query.Where(x => x.FunktionCodeTLNRFreigabe ==
        searchCmsw.FunktionCodeTLNRFreigabe);
7 }
```

In the end, the original query and the query which specifies the function code relation are joined.

Listing 4.10: Combine the query to retrieve the can messages

```
1 query = from tempResult in query
2         join fc_cn in fc_cn_query
3         on tempResult.NachrichtID equals fc_cn.NachrichtID
4         select new CANNachrichtDTO
5         {
6         }
```

Grouping data with LINQ to SQL. Grouping data is required to present entities that are represented by multiple rows in a database. A good example is the CAN message entity which uses a specific byte- and bit range on a certain

CAN identifier. It depends on the length of the carrier signal how many bits are occupied. For instance, a CAN message which uses a signal with a length of 16 is represented by 16 rows on the database level, each row indicating which byte and bit is occupied. The database representation of a CAN message is shown in figure 4.7.

	NachrichtID	Identifier	IdentifierCANBusID	Pos	BytePos	BitPos	SignalID
1	8508	257	4	0	4	0	1741
2	8509	257	4	1	4	1	1741
3	8510	257	4	2	4	2	1741
4	8511	257	4	3	4	3	1741
5	8512	257	4	4	4	4	1741
6	8513	257	4	5	4	5	1741
7	8514	257	4	6	4	6	1741
8	8515	257	4	7	4	7	1741
9	8516	257	4	8	5	0	1741
10	8517	257	4	9	5	1	1741
11	8518	257	4	10	5	2	1741
12	8519	257	4	11	5	3	1741
13	8520	257	4	12	5	4	1741
14	8521	257	4	13	5	5	1741
15	8522	257	4	14	5	6	1741
16	8523	257	4	15	5	7	1741

Figure 4.7.: CAN message representation on database level

The old EVI database schema does not store every occupied byte and bit allocation persistently in some cases. For example only the start- and the end bit are stored in the relation with function codes. Additional calculation based on the signal length is needed to determine if a certain bit position is already occupied. Furthermore there does not exist an object to represent the byte and bit assignments. The CORA uses the CAN message object (CANNachrichtDTO) in combination with a ByteBitRepresentation class. Listing 4.11 show the simplified version (without joins to language specific data and additional signal characteristics) of the query specification to retrieve CAN messages.

Listing 4.11: Retrieve CAN messages

```
1  var query = from cn in DataContext.CANNachrichts
2              join ident in identSet
3              on new { cn.Identifier, cn.IdentifierCANBusID } equals new {
                   Identifier = ident.Identifier1, IdentifierCANBusID = ident.
                   IdentifierCANBusID }
4              join signal in DataContext.Signals
5              on cn.SignalID equals signal.SignalID
6              select new
7              {
8                 NachrichtID = cn.NachrichtID,
9                 Identifier = cn.Identifier,
10                Pos = cn.Pos,
11                BytePos = cn.BytePos,
12                BitPos = cn.BitPos,
13                SignalID = cn.SignalID,
14             };
```

Listing 4.12 shows how grouping and a sub query enrich the original defined query to create the desired transfer objects. In LINQ to SQL it is possible to perform grouping by multiple criteria with the help of anonymous types.

Listing 4.12: Group the retrieved CAN messages

```
var groupedQuery = from n in query
                   group n by new { n.SignalID, n.Identifier } into g
                   select
                       new CANNachrichtDTO
                       {
                           NachrichtIDsc = g.Select(x => x.NachrichtID).ToList()
                               ,
                           Identifier = g.Key.Identifier,
                           BytePos = g.First().BytePos,
                           BitPos = g.First().BitPos,
                           ByteBitRep = (from bbr in g
                                         select new ByteBitRepresentation()
                                         {
                                             Pos = bbr.Pos,
                                             BytePos = bbr.BytePos,
                                             BitPos = bbr.BitPos
                                         }).ToList()
```

Dynamic LINQ to SQL. One of the benefits of LINQ to SQL is that it enables the creation of type-safe queries in a .NET language like C#. As a consequence the developer is supported with compile-time checking of the LINQ queries, and full IntelliSense and refactoring support over the code. Writing type-safe queries is great for many scenarios, but there are cases where developers want the flexibility to dynamically construct queries on the fly. For example the Query By Example technique (described later in the chapter 4.5) allows the dynamically construction of queries based on an input object. In this case a StringBuilder is responsible for the concatenation of the WHERE clause. To enable this scenario the Dynamic Query Library[1] provides extension methods which take string expressions as parameter. Another example is dynamic sorting of data because a CORA user can customize the ordering of the desired results with the help of the user interface. The code representing the Dynamic Query Library is placed in the Dynamic.cs source file under the lib folder of the CORA Data Access Layer.

Impact of LINQ to SQL on the application architecture. As pointed out before, the complexity of the CORA database schema (3.2.1) strongly encourages the creation of an explicit Data Access Layer and not to follow the RAD way (1.3) and use LINQ to SQL directly in your Presentation- or Business Logic Layer. The whole chapter 4 describes how to use LINQ to SQL in a three layered architecture but a few things have to be considered using LINQ to SQL the RAD way (through the whole application) as described in [15]. In this case LINQ to SQL acts as a kind of a minimal Data Access Layer. The classes generated by the LINQ to SQL Designer or the SqlMetal tool are the data entities that form the data object model. Moreover, there is no data access code in these entities. The SQL code that performs data retrieval or data manipulation operations is generated by the LINQ to SQL DataContext based on queries written in C# or VB. Therefore the actual interaction with the database is always realized with LINQ to SQL. Furthermore using LINQ to SQL in a RAD way allows fine-grained customization and very often each page requires a custom database query. When working with a DAL one can provide these methods in the Data Access Objects. If a method

[1] http://msdn.microsoft.com/en-us/vcsharp/bb894665.aspx

already exists to return the desired object, and one specific Web site does not require displaying all the fields, it is highly likely that the existing method is still used and simply some fields are omitted at the output. This can affect performance. More precisely, if there is any DAL that has code generic enough to satisfy the needs of every page, it may be at the cost of performance [15]. In case of enterprise applications the benefits of an explicit DAL (e.g., separation of concerns) usually exceed the performance impacts by far. Moreover, it is easily possible to add some RAD approaches at some points in a multi layered application but it is extremely costly the other way round.

4.3. Generic Controller Pattern

Within the Data Access Layer of the CORA the Controllers are the Data Access Objects mentioned in [3] and based on the architectural approach described in "An Example of a Multi Tier Architecture for LINQ to SQL" project, developed by developers of the open source community and formerly hosted on MSDN Code Gallery[2] and now on Codeplex[3].

The GenericController design approach addresses the problem of managing the LINQ to SQL DataContext and the reuse of queries among Data Access Objects. Regarding a Web application it is a recommended approach to maintain the DataContext for each request, also known as scoped DataContext. Therefore the DataContext has to be stored in a collection that is valid for the current HTTP request. The HttpContext.Current.Items collection exactly offers this possibility. The following code listing describes the management of the Datacontext.

Listing 4.13: Managing the DataContext during a request

```
1  private TDataContext _dataContext;
2  protected TDataContext DataContext
3  {
4      get
5      {
6          //We are in a Web app, use a request scope
7          if (HttpContext.Current != null)
8          {
9              TDataContext _dataContext = (TDataContext)HttpContext.Current.Items[
                   "_dataContext"];
10             if (_dataContext == null)
11             {
12                 //Create a new DataContext
13                 _dataContext = Activator.CreateInstance<TDataContext>();
14                 HttpContext.Current.Items.Add("_dataContext", _dataContext);
15             }
16             if (logging)
17             {
18                 string linq_log_file = System.Configuration.ConfigurationManager
                       .AppSettings["DataAccessLayer_linq_logfile"];
19                 _dataContext.Log = new HelperTools.FileLogger(linq_log_file,
                       false);
20             }
21             return _dataContext;
22         }
23         else
24         {
25             //If this is not a Web app then just create a datacontext
```

[2]http://code.msdn.microsoft.com/
[3]http://www.codeplex.com/

```
26              //which will have the same lifespan as the app itself
27              //This is only really to support unit tests and should not
28              //be used in any production code. A better way to use this
29              //code with unit tests is to mock the HttpContext
30              if (_dataContext == null)
31              {
32                  _dataContext = Activator.CreateInstance<TDataContext>();
33              }
34              return _dataContext;
35          }
36      }
37  }
```

Worth mentioning is the fact that the DataContext is generated via reflections according to the passed TDataContext type. Therefore the class which manages the DataContext is not limited to a specific source database. Using generic parameters to enable reuse on a very large scale is the basic idea behind the Generic Controller Pattern. Every single Controller takes care of the data access logic for one CAN entity and inherits from the GenericController. In the original version the GenericController is defined by the following code listing.

Listing 4.14: Class definition of the GenericController

```
1  public class GenericController<TEntity, TDataContext> where TDataContext :
       DataContext where TEntity : class
```

In addition to the TDataContext the particular LINQ to SQL entity is passed to implement CRUD operations. The following code listing demonstrates a method which returns an entity according to a specified ID.

Listing 4.15: Generic database retrieval

```
1  protected static string TableName
2  {
3    get
4    {
5        var att = EntityType.GetCustomAttributes(typeof(TableAttribute), false).
             FirstOrDefault();
6        return att == null ? "" : ((TableAttribute)att).Name;
7    }
8  }
9
10 private static PropertyInfo _primaryKey;
11 protected static PropertyInfo PrimaryKey
12 {
13     get
14     {
15         if (_primaryKey == null)
16         {
17             foreach (PropertyInfo pi in Columns)
18             {
19                 foreach (ColumnAttribute col in pi.GetCustomAttributes(typeof(
                       ColumnAttribute), false))
20                 {
21                     if (col.IsPrimaryKey)
22                         _primaryKey = pi;
23                 }
24             }
25         }
26         return _primaryKey;
27     }
28 }
29
30 public static TEntity GetEntity(object id)
31 {
32     string query = string.Format("Select␣*␣from␣{0}␣where␣{1}␣=␣{2}", new object
           [] { TableName, PrimaryKey.Name, id });
```

```
33      return DataContext.ExecuteQuery<TEntity>(query).FirstOrDefault();
34  }
```

A general method which is responsible for the insertion of all entities can be easily implemented as well.

Listing 4.16: Generic database operation

```
1  public static void Insert(TEntity entity, bool submitChanges)
2  {
3      EntityTable.InsertOnSubmit(entity);
4      if (submitChanges)
5          DataContext.SubmitChanges();
6  }
```

So by simply implementing a CategoryController or a ProductController the functionality for retrieving an entity according to an ID field and to insert a new entity is provided. The class CategoryController does not need any further code than specified in the following code listing.

Listing 4.17: Controller definition

```
1  public class CategoryController : GenericController<Category,
       NorthwindDataContext>
2  {
3  }
4  }
```

The overall rule for the Generic Controller Pattern is to implement general data logic in the GenericController and entity specific logic in the particular Controller for each single entity by adding functions, or by overwriting inherited functions. The GenericController of the CORA exists of nearly 2500 lines of code, requires additional generic parameter types and implements a generic interface which describes the available functionality and therefore allows method invocation via reflection.

Listing 4.18: Definition of the Generic Controller of the CORA DAL

```
1  public class GenericController<TDTO, TEntity, TEntityDetails, TEntityHistory,
       TDetailsHistory, TDataContext> : IController<TDTO, TEntity, TEntityHistory>
2          where TDTO : class, IDTO
3          where TEntity : class
4          where TEntityDetails : class
5          where TEntityHistory : class
6          where TDetailsHistory : class
7          where TDataContext : DataContext
```

- TDTO: The type of object which transports data to other layers (described in 4.4).
- TEntity: The type of the LINQ to SQL generated object representing the language specific data.
- TEntityDetails: The type of the LINQ to SQL generated object representing the language unspecific data.
- TEntityHistory: The type of the LINQ to SQL generated object representing the history table for language unspecific data.

- TDetailsHistory: The type of the LINQ to SQL generated object representing the history table for language specific data.
- TDataContext: The type of the LINQ to SQL Datacontext.

For simple CAN bus entities like a hardware pin or a connector (German: Stecker) the generic methods of the GenericController suffice to implement the data access logic. For example if specific connectors are requested, language specific and language unspecific data is combined into a single object and sent to the other layers. If a user modifies an existing connector or inserts a new one, the single object is sent to the SteckerController and mapped to the LINQ to SQL entities which insert the data into the particular tables. In addition the history of each connector can be retrieved as well. All these operations are instantly available due to the generic logic of the parent class and therefore the class SteckerController only requires the following lines of code.

Listing 4.19: Definition of a CORA controller

```
1  public class SteckerController : GenericController<SteckerDTO, Stecker,
       Stecker_Detail, Stecker_History, Stecker_Details_History,
       CANBusDatenDataContext>
2  {
3      public SteckerController(SettingsDTO settings)
4          : base(settings)
5      {
6      }
7
8      public SteckerController()
9      {
10     }
11 }
```

The SettingsDTO provides initial settings for the Data Access Layer like the current selected language of the Web application, the current user, the selected headquarter and the selected CAN bus. This eases the parameter management and provides information because an additional parameter can easily be added to the SettingsDTO class and there is no need to update numerous method signatures. Furthermore the Controller knows on which CAN bus the user is currently working on and therefore there is no need to set the CANBusID on every entity explicitly.

More complex CAN bus entities like CAN modules, signals and function codes provide much more functionality within their specific Controllers. For example the generic default retrieval methods are insufficient if the entity is composed of more than the two (language specific and language unspecific) LINQ to SQL entities. In this case some methods of the GenericController have to be overwritten. In many cases the overwritten methods in the child controllers use generic methods of the common parent controller. The following example demonstrates the benefits of an object-oriented approach like the Generic Controller Pattern. The deletion of a production location (German: Standort) sounds like a complex process because all the reference data like CAN bus systems and user settings have to be deleted as well. With the help of the Generic Controller Pattern only a few lines of code are required. In the StandortController the Delete method is overwritten to delete the referenced data first and finally the generic delete method is called (base.Delete(dtObject)) to delete the production location.

Listing 4.20: Generic overwritten method

```
public override void Delete(StandortDTO dtObject)
{
    using (TransactionScope scope = new TransactionScope())
    {
      canbusCon.Delete(dtObject);
      benutzerCon.DeleteRights(dtObject);
      base.Delete(dtObject);
      scope.Complete();
    }
}
```

Moreover, validation-, logging- and security aspects can be managed at a central point in the GenericController, which results in huge maintenance benefits. CAN bus entities which do not need certain functionality of the GenericController yet pass default classes. For example the signal cycle is represented by the ZyklusController but does not incorporate any language specific data. So the class is defined like the following.

Listing 4.21: Controller for a simple CAN entity

```
public class ZyklusController : GenericController<DefaultDTO, Zyklus,
    DefaultDetails, Zyklus_History, DefaultDetailsHistory,
    CANBusDatenDataContext>
{
    public ZyklusController(SettingsDTO settings)
        : base(settings)
    {
    }

    public ZyklusController()
    {
    }

}
```

The GenericController performs operations according to the passed types. In case of DefaultDetails no LINQ to SQL entity and table for language specific data is taken into account for CRUD operations.

4.4. Data Transportation

Data has to be transported through the layers. There are different architectural options regarding data transportation due to the choice between a generic data transfer structure and a specific data transfer object. Figure 4.8 shows the different options according to [27].

1. The first option is a generic, relational data object (e.g., an ADO.NET DataSet) offered by the Data Access Layer. This generic data object is forwarded from the business logic to the user interface.

2. The Data Access Layer offers a generic, relational data object (e.g., an ADO.NET DataSet), which is packed into a typed data object within the Data Access Layer or the Business Layer.

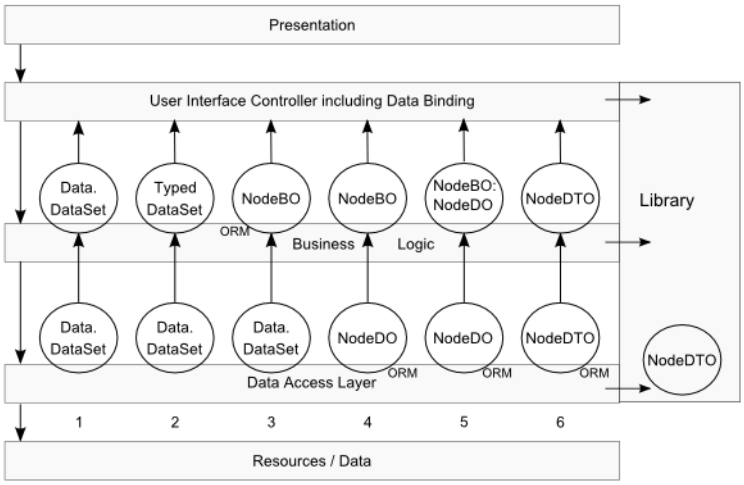

Figure 4.8.: Options for data transport according to [27]

3. The Data Access Layer offers a generic, relational data object (e.g., an ADO.NET DataSet) out of which the business logic creates a domain specific business object. In this case the object-relational mapping takes place within the Business Application Layer.

4. Already the Data Access Layer provides a domain specific data object. This object is transformed within the Business Layer into a domain specific business object. This transformation is based on copying the data between the objects, also referred as object mapping.

5. To avoid the copying process the domain specific business objects inherit from the domain specific data objects.

6. The last displayed alternative puts the data transfer objects in a separate library which is used by every layer. In this case the business object is instantiated in the Data Access Layer and passed to the Presentation Layer over the Business Layer. The business object is a plain data object which is managed by manager objects on both layers.

The CORA architecture is based on the last option. Furthermore, the architecture considers the needs of service orientation which demand the exchange of automatically processable information. As a consequence the only concern of data carrying, layer pervading objects is data transportation, and therefore they do not contain any domain or database access logic. Furthermore, to support disconnected environments and future architectural trends like Service Oriented Architecture it is recommended to avoid exposing LINQ to SQL generated entities through the DAL. On the contrary the DAL should return simple Data Transfer Objects (DTO) whose sole purpose is to carry data across service boundaries [28]. Although the CORA Web application does not operate in a disconnected environment at the moment, the objects which carry assembled data from the LINQ to SQL entities are referred to as Data Transfer Objects (DTO). Figure 4.9 provides a definition for a DTO.

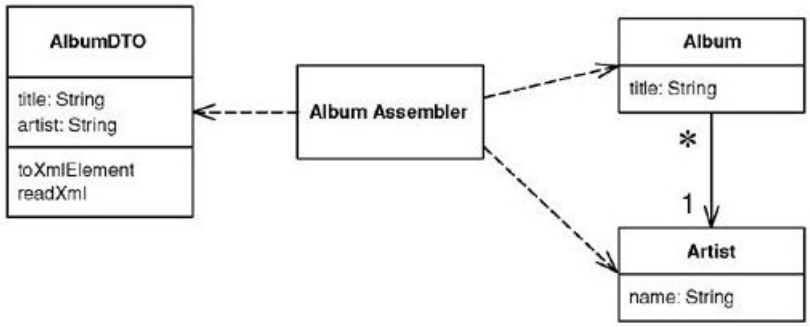

Figure 4.9.: Data Transfer Object according to [7]

The CORA Data Transfer Objects are plain C# objects. They do not provide serialization methods to XML but they make use of the ISerializable interface which allows an object to control its own serialization and deserialization. In the first released version of the CORA the DTOs are generated manually with the help of auto-implemented properties (or automatic properties). The compiler automatically creates a private field which can only be accessed through the property's get and set accessors which result in less code for a property deceleration. The major drawback is that no additional logic can be added to the property. In addition to the properties to store queried data a CORA DTO must implement the interface IDTO which instructs the implementation of a GetID and SetID method. Furthermore the interfaces IMultiLangDTO and IHistoryDTO are available for implementation. In addition the property names which store the ID, the CAN bus ID and the language code, are exposed by constants. This allows the comfortable invocation via reflection. For example the StandortDTO looks like the following.

Listing 4.22: Sample DTO

```
1  [Serializable]
2  public class StandortDTO : IDTO, IMultiLangDTO, IHistoryDTO
3  {
4      public const string IDFieldname = "StandortID";
5      public const string CANBusIDFieldName = null;
6      public const string SprachcodeFieldName = "Sprachcode";
7
8      public object GetID() { return StandortID; }
9      public void SetID(object id) { StandortID = int.Parse(id.ToString()); }
10
11     public int StandortID { get; set; }
12     public string Name { get; set; }
13     public string DMDNR { get; set; }
14     public string DFINR { get; set; }
15     public string DWKNR { get; set; }
16     [LanguageDependent]
17     public string Bezeichnung { get; set; }
18     [LanguageDependent]
19     public string Sprachcode { get; set; }
20
21     public DateTime HistoryCreated { get; set; }
22     public string HistoryBenutzer { get; set; }
23     public string HistoryComment { get; set; }
24 }
```

The fact that LINQ to SQL supports the automatic population of custom objects eliminates any programming code overhead for copying the retrieved result set into the DTOs. The following code demonstrates a query to get the entire

CAN bus systems for a specific production location.

Listing 4.23: Populating a DTO with LINQ query

```
1  public List<CANBusDTO> GetForStandort(StandortDTO standort)
2  {
3      var query = from cb in DataContext.CANBus
4                  where cb.StandortID == standort.StandortID
5                  join cbd in DataContext.CANBus_Details.Where(x => x.Sprachcode
                       == settings.Language)
6                  on cb.CANBusID equals cbd.CANBusID
7                  into cbdJ
8                  from cbdjX in cbdJ.DefaultIfEmpty()
9                  join cbi in DataContext.CANBusInfos
10                 on cb.CANBusInfoID equals cbi.CANBusInfoID
11                 join cbid in DataContext.CANBusInfo_Details.Where(x => x.
                       Sprachcode == settings.Language)
12                 on cbi.CANBusInfoID equals cbid.CANBusInfoID
13                 into cbiJ
14                 from cbiJX in cbiJ.DefaultIfEmpty()
15                 select new CANBusDTO
16                 {
17                     CANBusID = cb.CANBusID,
18                     Bezeichnung = cbdjX.Bezeichnung,
19                     Bits = cbi.Bits,
20                     Frequenz = cbi.Frequenz,
21                     Protokoll = cbi.Protokoll,
22                     CANBusInfoID = cb.CANBusInfoID,
23                 };
24     List<CANBusDTO> result = new List<CANBusDTO>(query);
25     return result;
26 }
```

The custom object population is not limited to LINQ to SQL statements which get translated into T-SQL queries. Custom SQL queries executed by the LINQ to SQL framework populate custom objects as well. The following code returns a collection of DTOs of the type TDTO queried by a custom SQL string.

Listing 4.24: Populate DTO with custom SQL statement using LINQ to SQL

```
1  \begin{lstlisting}
2  List<TDTO> resultList = DataContext.ExecuteQuery<TDTO>(query.ToString()).ToList<
       TDTO>();
```

Unfortunately the custom object population provided by the LINQ to SQL framework is limited to flat objects. So it is not possible to define common properties in an abstract base class or an interface. So repeating fields like HistoryComment have to be implemented in every single DTO.

Mapping a DTO to the corresponding LINQ to SQL entities (or the other way round) is achieved via Reflection (the process by which a program can read its own metadata [13]) and a strict naming convention. If a property of a DTO is named exactly like the property of the LINQ to SQL generated class the value can be copied automatically. For sure additional validation logics have to be taken into account as well. For example not only the name of the property has to match but also the return type. Code listing 4.25 demonstrates the mapping of the language specific entity and the language unspecific entity to an assembled DTO.

Listing 4.25: Map LINQ to SQL entities to a DTO

```
1  private TDTO EntitiesToDTO(TDTO dtObject, TEntity ent, TEntityDetails entDet)
2  {
```

```
3     foreach (PropertyInfo piDTO in dtObject.GetType().GetProperties())
4     {
5         if (ent != null)
6         {
7             foreach (PropertyInfo piEntity in GetType<TEntity>().GetProperties())
8             {
9                 PropertyInfo piDTOent = GetType<TDTO>().GetProperty(piEntity.Name)
                      ;
10                {
11                    if (piDTOent != null && CheckProperty<TEntity>(ent, piEntity)
                          && piEntity.PropertyType.Name.Equals(piDTOent.PropertyType
                          .Name))
12                    {
13                        piDTOent.SetValue(dtObject, piEntity.GetValue(ent, null),
                              null);
14                    }
15                }
16            }
17        }
18        if (entDet != null)
19        {
20            foreach (PropertyInfo piEntityDetails in GetType<TEntityDetails>().
                  GetProperties())
21            {
22                PropertyInfo piDTOdet = GetType<TDTO>().GetProperty(
                      piEntityDetails.Name);
23                {
24                    if (piDTOdet != null && CheckProperty<TEntityDetails>(entDet,
                          piEntityDetails) && piEntityDetails.PropertyType.Name.
                          Equals(piDTOdet.PropertyType.Name))
25                    {
26                        piDTOdet.SetValue(dtObject, piEntityDetails.GetValue(
                              entDet, null), null);
27                    }
28                }
29            }
30        }
31
32    }
33    return dtObject;
34 }
```

Furthermore it is possible to add additional information to properties with the help of Attributes, which are a mechanism for adding metadata, or to put it differently, an object that represents data which is intended to be associated with an element of the program [13]. With Custom Attributes as described in 4.26 the properties of a DTO can be exactly mapped to the corresponding LINQ to SQL generated class. Moreover the LanguageDependent attribute has been designed for properties which contain language specific data.

Listing 4.26: Custom Attribute

```
1  public class FieldRelations : Attribute
2  {
3      public string EntityName { get; set; }
4      public string PropertyName { get; set; }
5
6      public FieldRelations(string entityname, string propertyName)
7      {
8          EntityName = entityname;
9          PropertyName = propertyName;
10
11     }
12 }
```

In summary the Data Access Layer of the CORA uses Metadata Mapping twice. Firstly for mapping the fields in the database to the fields of in-memory objects

(LINT to SQL entities) and secondly for mapping the fields of in-memory objects to assembled Data Transfer Objects and vice versa.

4.5. Advanced Data Retrieval and Manipulation

In a layered architecture the implementation of the information exchange between the single layers is a crucial architectural decision. The main characteristic of data-centric Web applications like the CORA is the high amount of requirements concerning data storage and manipulation. The focus lies on retrieving stored data, presenting it to the user, and applying all made modifications. In many cases data is represented in the tabular relational form. A Record Set is an in-memory representation of tabular data and looks like the result of a database query. In ASP.NET the typed DataSet gained widespread popularity but mainly because it fully supports data-binding and has great integration with Visual Studio's IDE. Therefore built-in wizards can automatically create typed DataSets. Another approach is to store retrieved data in custom objects. There exist numerous debates when to use typed Datasets and when custom objects. Most jump to the conclusion that typed DataSets support Rapid Application Development for smaller solutions because they can be created automatically through the IDE. Custom objects on the other hand are fully object-oriented and to favor in terms of enterprise applications. In case of the CORA it was obvious that custom objects like the automatically generated LINQ to SQL classes should act as the primary representer of tabular data. Moreover, due to the fact that these entities are generated as partial classes it is easy to add additional code which is not lost on recreation. At the moment the business logic does not require that the objects, which represent the persistently stored real world data, are transformed to particular Business Objects. On the contrary the business logic can be applied directly on collections of the data carrying objects as described in 5. Furthermore the Presentation Layer can use these object collections to provide user interaction. Due to the fact the data representing objects are used through the whole application, the LINQ to SQL generated classes are insufficient mainly because of the following two reasons.

Many object-relational mappers allow to map an object to multiple tables. For example the Entity Framework allows a way of describing the data structure (the schema) on a higher level of abstraction with the help of the Entity Data Model. This model supports entities which contain information out of multiple database tables and therefore there are no additional joins required, as described in the example provided by [23]. On the other hand LINQ to SQL only supports one-to-one table mapping, but the generated classes can be mapped to a database view that joins multiple related tables. For example a CAN bus signal is represented by the following database tables as shown in figure 4.10.

Basically a database view is a virtual table created by stored queries. The SQL Server Management Studio supports the creation of views with the Query Designer which enables to build a query graphically including links between associated tables.

The class representing the created view can be easily generated by the LINQ

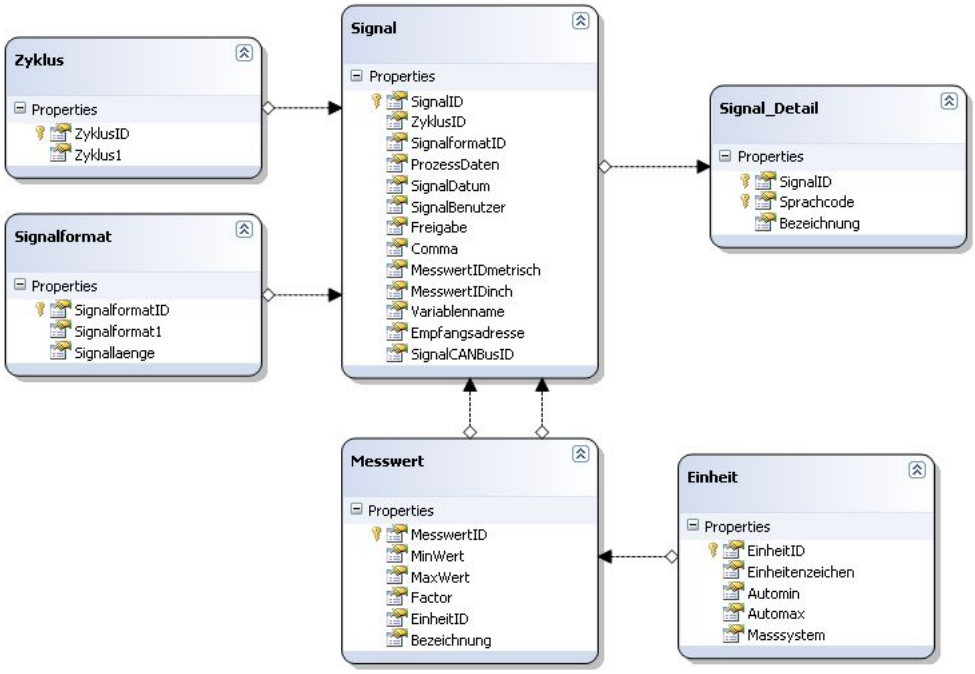

Figure 4.10.: Tables that represent a CAN bus signal

to SQL Designer. Querying the view is similar to query a table. In listing 4.27 the assembled CAN Signal is first bound directly to a GridView. The next code lines demonstrate how easily the result of multiple tables can be combined into a single object.

Listing 4.27: Querying a databse with with LINQ to SQL

```
1  DataClasses1DataContext dc = new DataClasses1DataContext();
2  //bin directly to a GridView
3  var query1 = from signals in dc.vSignals select signals;
4  GridView1.DataSource = query1.Take(10);
5  GridView1.DataBind();
6
7  //create custom objects
8  var query2 = from signals in dc.vSignals where signals.SignalID > 500
9               select new SignalDTO
10              {
11                  SignalID = signals.SignalID,
12                  Desription = signals.Bezeichnung
13              };
14 query2 = query2.Take(10);
15 List<SignalDTO> signalsList = query2.ToList();
16 //this collection of custom objects (SignalDTOs) can be send across layers and
       systems
17 //for instance to a PresentationLayer which retrieves the list
18 GridView1.DataSource = signalsList;
19 GridView1.DataBind();
```

As indicated by the name, a database view can only be accessed by reading operations. It is not possible to perform update operations against a view in order to automatically update the underlying tables. A common approach is to use Stored Procedures to update the database. The fact that LINQ to SQL queries are automatically translated to cached parameterized T-SQL queries reduces the concerns from a performance and security point of view. Nevertheless, it is easy to use Stored Procedures with LINQ to SQL. In listing 4.28 the simplified version

of a Stored Procedure which updates a CAN signal is shown.

Listing 4.28: Create Stored Procedure

```
1  CREATE PROCEDURE UpdateSignal
2    @SignalID int,
3    @Comma int,
4    @Bezeichnung nvarchar(100),
5    @Sprachcode char(2)
6  AS
7  BEGIN
8    UPDATE dbo.Signal SET Comma =  @Comma WHERE SignalID = @SignalID;
9    Update dbo.Signal_Details SET Bezeichnung = @Bezeichnung WHERE SignalID =
         @SignalID AND Sprachcode = @Sprachcode
10   /* more tables and fields.... */
11 END
12 GO
```

After adding the Stored Procedure to the DataContext with the drag and drop capabilities of the LINQ to SQL Designer, the invocation only requires one line of code.

Listing 4.29: Calling a SPROC with LINQ to SQL

```
1  DataClasses1DataContext dc = new DataClasses1DataContext();
2  dc.UpdateSignal(1, 2, "SPROC␣test", "en");
```

It is even possible to customize LINQ to SQL entities to use custom Stored Procedures for their built in CRUD operations. In this case the developer only has to assign the object properties and in the minute when the DataContext calls SubmitChanges() all the associated Stored Procedures are automatically invoked. A quick example can be found at [9]. Using Stored Procedures to retrieve data does not work that flawlessly with LINQ to SQL. When the Stored Procedure returns multiple result sets, uses a temporary table, or executes dynamic string concatenating queries, the LINQ to SQL Designer cannot figure out the return type. For instance, it is not possible to pass the sort criteria as a string parameter to complete the query. All the sort possibilities must be hard coded in a control structure, like case or if statements.

Views and Stored Procedures are a wide spread approach to realize data retrieval and manipulation. Nevertheless one cannot find a single View or a Stored Procedure in the CORA database (maybe only for testing purposes) at the moment. The maintenance benefits of the object-oriented LINQ to SQL queries in combination with the excellent integration into Visual Studio got the upper hand in the end. Therefore even complex queries including multiple joins are written as LINQ to SQL queries and provided for reuse by methods as demonstrated in section 4.2. Articles about the Entity Framework like [23] point out that database views are simply too complex to be generated and maintained by developers in a cost-effective way. Moreover, databases are often used by many departmental applications, and having each individual application create several views in the database would pollute the database schema and create significant maintenance workload for the database administrators. In addition to this views are limited to the expressiveness of the relational world.

The fact that views have not been used so far by the CORA does not imply that they should not be used in the future. The same applies for stored procedures.

There are a lot of discussions going on in which case stored procedures should be favored over retrieving mechanisms offered by object-relational mapping frameworks. The result of these discussions should not be exclusive. On the contrary the object-relational mapper should support multiple retrieving strategies.

Query by Example Originally Query by Example (QBE) refers to a database query language for relational databases, devised by Moshé M. Zloof at IBM Research during the mid 1970s. Nowadays the term is often used to describe a general technique influenced by Zloof's work whereby only items with search values are used to filter the results [35]. The queries are automatically generated according to these items. The Controllers of the CORA Data Access Layer often expect DTOs as input parameters for the LINQ to SQL query generation. Listing 4.30 shows a simplified version of the GenericController methods which creates a query according to the property values of the passed object.

Listing 4.30: Query by example query generation

```
1  public StringBuilder CreateLinqConditionsBasedOnQueryDTO(Object input,
       QuerySettings querySettings)
2  {
3      StringBuilder result = new StringBuilder();
4      Type type = input.GetType();
5      foreach (PropertyInfo propInfo in type.GetProperties())
6      {
7          object value = propInfo.GetValue(input, null);
8          if (value != null)
9          {
10             Type propType = propInfo.PropertyType;
11             PropertyInfo piHasValue = input.GetType().GetProperty(propInfo.Name
                   + "HasValue");
12             switch (propType.Name)
13             {
14                 case "Int32":
15                     if (piHasValue != null)
16                     {
17                         if (bool.Parse(piHasValue.GetValue(input, null).ToString
                               ()) == true)
18                         {
19                             if (result.Length > 0) result.Append("␣" +
                                   querySettings.ConjunctionOperator + "␣␣");
20                             result.AppendFormat(prefix + propInfo.Name + "␣=␣{0}
                                   ", value);
21                         }
22                     }
23                     else
24                     {
25                         if ((int)value != -1 && (int)value != -0)
26                         {
27                             if (result.Length > 0) result.Append("␣" +
                                   querySettings.ConjunctionOperator + "␣");
28                             result.AppendFormat(prefix + propInfo.Name + "␣={0}"
                                   , value);
29                         }
30                     }
31                     break;
32                 case "Int64":
33                     break;
34                 case "String":
35                     break;
36                 case "Nullable`1":
37                     if (result.Length > 0) result.Append("␣" + querySettings.
                           ConjunctionOperator + "␣");
38                     result.AppendFormat(prefix + propInfo.Name + "␣=␣{0}", value
                           );
39                     break;
```

```
40                 }
41             }
42         }
43         return result;
44     }
```

The first parameter is the Query by example object, the second a custom object which specifies additional parameter for the query generation, like which conjunction operator (e.g., "AND" or "OR") should be used to connect the single search conditions. At the beginning the PropertyType is determined to delegate the query creation accordingly. Then a check is performed if a "HasValue" property exists to indicate if a search criterion is set for a particular property. Otherwise the check is preformed against the standard value. If the property contains a value that passes all the validation logic, the value is used for the query creation.

Realizing entity history. One of the CORA requirements covers a history or a change log for the most important CAN entities as described in section 2.4. There are two possible implementation strategies:

1. Single table approach: All entries are stored in the same table with additional assigned fields which indicate the expiration of the validity for every entry.

2. History tables approach: An additional table which backups every single entry is created for every table.

The initial implementation of the first approach is not very laborious. One field indicates the creation of the row and a second field contains the time stamp of the first modification, or null if it is the current entry. This modification can be an update- or delete operation. Before an operation takes place the database row is copied to provide the traceability for all changes. A simple query against null still ensures acceptable performance for retrieving the current entities. The biggest drawback of this approach is that every table gets polluted with obsolescent (only for change tracking relevant) data. In case of an enterprise application like the CORA this can result in tremendously big tables. Moreover, every application that accesses this database has to take the history pattern into account as well. On the other hand the history tables approach allows a clean separation of current- and history data. As a consequence not every single query has to be concerned about the retrieval of the current entry, and the additional table implementation guarantees transparency for application which do not need change tracking aspects. In general, data which represents old version of CAN entities is only touched when really needed. The creation of the history tables is not that cumbersome with the help of tools or scripts as described in 3.2.3. The major drawback lies in changes to the database schema after the initial creation because every affected history table has to be modified as well. For environments where database schemes are changing frequently the modification of the history tables should be automated as well.

The next point to consider is the strategy to backup every single database row before any kind of modification is applied. Any kind of modification refers to

CRUD operations without taking "read" into account. Again there exist two approaches, whereas the former is completely realized by the database engine itself and the latter is realized on the application side:

1. The use of database triggers.
2. The Data Access Objects control the creation of the history entries.

Using database triggers to perform backup operations is a wide spread approach. Due to the fact that the triggers operate completely on the database side it is not possible to pass any parameters and therefore all the information must be available in the database schema. This is not a big problem for simple scenarios like basic change tracking realized with history entries including three fields as described in section 3.2.1. The time stamp of the last performed operation and the responsible user can be stored in the table on which the operation is applied. The history table needs an additional field describing the operation which leads to the creation of the history entry. If history support is limited to create-, update- and delete operations the description can be hard coded within the trigger generation or referenced from a table which stores history comments. Listing 4.31 shows the code for creating a trigger that backups an inserted entry into the history table.

Listing 4.31: Create Trigger for Insert History

```
1  CREATE TRIGGER CANModul_Insert
2     ON  dbo.CANModul
3     AFTER INSERT
4  AS
5  BEGIN
6    DECLARE @CANModulTLNR VARCHAR(100);
7    DECLARE @CANBusID int;
8    DECLARE @ProgrammierungsKategorieID int;
9    DECLARE @Modified datetime;
10   DECLARE @User VARCHAR(100);
11
12   SET @CANModulTLNR = (SELECT CANModulTLNR FROM Inserted);
13   SET @CANBusID = (SELECT CANBusID FROM Inserted);
14   SET @ProgrammierungsKategorieID = (SELECT ProgrammierungsKategorieID FROM
         Inserted);
15   SET @Modified = (SELECT Modified FROM Inserted);
16   SET @User = (SELECT [User] FROM Inserted);
17
18   INSERT INTO dbo.CANModul_History (HistoryCreated, HistoryComment,
         HistoryBenutzer, CANModulTLNR, CANBusID, ProgrammierungsKategorieID)
19   VALUES (@Modified, 'CREATED', @User,  @CANModulTLNR, @CANBusID,
         @ProgrammierungsKategorieID);
20 END
```

With the help of the automatically created virtual tables (Inserted, Updated, Deleted) it is possible to create triggers for all data manipulating operations. For sure the data contained in the language specific tables must be backed up as well. When retrieving the history of an entity for a specific time it makes sense to join language unspecific and language specific data. Therefore the time stamp which stores the date of a single operation should be the same in both tables describing the entity. This can be easily realized by querying the history table of the language unspecific data first and then make the history entry for the language specific table.

The second approach allots the Data Access Objects to take care of creating a history for each database row. In this case custom queries can realize any imaginable creation of history entries. As described in section 4.3 the GenericController pattern provides the possibility for child controllers to pass generic parameters. Therefore every child controller can pass the objects which represent the history tables to the GenericController and generic methods ensure that the history entries are created for each CAN entity as described in listing 4.32.

Listing 4.32: Entity history managed by generic DAO

```
1  public void AddHistoryEntry(TEntity ent, string comment, DateTime dateTime)
2  {
3      TEntityHistory history = (TEntityHistory)Activator.CreateInstance(GetType<
           TEntityHistory>());
4      CopyEntityProperties<TEntity, TEntityHistory>(ent, history);
5      AddHistoryGeneric<TEntityHistory>(history, comment, dateTime);
6  }
7
8  public void AddHistoryDetailsEntry(TEntityDetails det, string comment, DateTime
       dateTime)
9  {
10     TDetailsHistory history = (TDetailsHistory)Activator.CreateInstance(GetType<
           TDetailsHistory>());
11     CopyEntityProperties<TEntityDetails, TDetailsHistory>(det, history);
12     AddHistoryGeneric<TDetailsHistory>(history, comment, dateTime);
13 }
14
15 private void AddHistoryGeneric<T>(T history, string comment, DateTime dateTime)
       where T : class
16 {
17     PropertyInfo piCreated = history.GetType().GetProperty(HistoryHelperDTO.
           CreatedField);
18     piCreated.SetValue(history, dateTime, null);
19     PropertyInfo piComment = history.GetType().GetProperty(HistoryHelperDTO.
           CommentField);
20     piComment.SetValue(history, comment, null);
21     PropertyInfo piBenutzer = history.GetType().GetProperty(HistoryHelperDTO.
           BenutzerField);
22     piBenutzer.SetValue(history, settings.User, null);
23     GetTable<T>().InsertOnSubmit(history);
24 }
```

The biggest difference between the two approaches is that different environments take care of the history management. When using triggers on the database side the history support is automatically enabled for every application that is able to perform database operations with Microsoft SQL Server, regardless the technology and the actual implementation. On the other hand creating and managing triggers in a database environment is often more cumbersome than using fully object-oriented features provided by accessing applications. Due to the GenericController pattern, the fact that the CORA is the only application that interacts with the database at the moment, and the limited resources, triggers are not implemented and the Data Access Layer takes care of managing the history for CAN entities.

4.6. Creation of Data Access Layer Components

Model Driven Development (MDD) is a generic term for techniques which focus on models to create solutions. In many software engineering projects models are primarily used for documentation purposes but for the MDD approach models

are equal to code. Moreover models are used to generate code. One core benefit of models lies in the higher level of abstraction. For instance, a software architect is able to model the software architecture with the help of platform independent models which are automatically transformed to platform specific models and finally to code. This approach is called Model Drive Architecture which can be considered as part of MDD but defines interoperability as primarily goal. On the contrary general MDD aims to generate usable building blocks.

The CORA is designed as an ASP.NET Web application and until this time there is no need to transfer the architecture to other technologies and therefore MDA has not been applied so far. On the other hand MDD was used to generate the database schema with the help of Microsoft Visio as described in 3.2. Furthermore the in Visual Studio integrated LINQ to SQL Designer provides a visual model to create the LINQ to SQL entities and model the data mappings. Unfortunately this model does not provide the possibility to model the Data Transfer Objects and the Controllers. Additional tools like Enterprise Architect from Sparx Systems[4] or Scultpure by Dawaliasoft [5] can be considered to enable richer model driven possibilities for the ongoing CORA development. Another possibility is the creation of the DTOs source files with the help of Reflection.

[4]http://www.sparxsystems.com/
[5]http://www.dawliasoft.com/

5. Business Application Layer

The Business Application Layer (BAL) contains the components which realize the business logic for certain requirements of a company. In other words this layer coordinates the application, processes user commands, makes logical decisions, evaluations and performs calculations [34]. For example the calculation of a CAN message identifier is based on a strict formula and implemented in a class of the Business Application Layer. Moreover the BAL coordinates the communication between the Presentation Layer and the Data Access Layer. According to [7] there exist three general approaches for realizing domain logic:

1. Transaction Script
2. Domain Model
3. Table Module

Transaction Script. This approach defines a Transaction Script basically as a procedure that takes the input from the Presentation Layer, applies validation and calculations and finally stores the processed data in the databases. During this process invocations of operations from other systems can take place as well. Furthermore the Transaction Script retrieves data from the databases, again applies business logics, and represents the output to the user. The basic characteristic of a Transaction Script is the simple procedural model without taking object-oriented ideas into account, which is only suitable for a simple problem domain and leads to problems with more complex scenarios. For instance, the business logic for the hardware configuration of a raw CAN module without software can be realized with a Transaction Script. The business logic for the hardware configuration of a CAN module with software must be implemented as well, and therefore another Transactions Script is needed. Both processes will have common and shared validation logic. Although it is possible to factor out common subroutines, the biggest disadvantage of the Transaction Script pattern for realizing business logic is duplicated code because several transactions perform similar operations.

Domain Model. A Domain Model considers object-oriented principles and describes various Business Objects and their relationships instead of having one routine dealing with all the logic for a single user action. This enables developers to handle increasingly complex logic in a well-organized way. Within a Domain Model multiple objects can interact until the desired result is achieved. For instance, a hardware configuration object can be called by multiple objects representing the different kinds of CAN modules or two separate configuration objects can inherit the common validation logic from a parent object. According

to [3] these Business Objects encapsulate and manage business data, or to put it differently they implement a well-defined business domain concept and include intrinsic business logic and business rules that apply to that domain concept. Moreover, there exist primarily two strategies for implementing Business Objects:

1. With "regular" objects offered by the used object-oriented environment, POJOs for JAVA or POCOS for .NET.

2. With Entity Beans or Composite Entities, which provide a persistent storage mechanism.

Table Module. The third choice for structuring domain logic is the Table Module which resembles a Domain Model. The significant difference is that the Domain Model uses an extra instance of a Business Object to apply business logic to each entity, and the Table Module only uses one single instance and works with a collection of entities. For example the configuration of CAN modules leads to the instantiation of multiple CAN module Business Objects according to the Domain Model pattern. In this case different parameters can be passed to the multiple instances and they can therefore perform customized operations in any imaginable way. According to the Table Module pattern one instance of a Business Object managing a collection of CAN modules is created to apply the desired business logic. This example illustrates that a Table Module can be placed between a Transaction Script and a Domain Model when it comes to organize the domain logic. As the name indicates a Table Module organizes the domain logic around the database tables. This provides more structure and eases the search and removal for duplication than straight procedures. However Table Modules cannot tap the full object oriented potential which comes along with a finer grained logical structure provided by a Domain Model. According to [7] the biggest advantage of a Table Module is the integration into the rest of the application architecture. For example an environment like .NET and Visual Studio provide a lot of tools to work with Record Sets. The previous chapter 4 described the manifold possibilities for creating generic result sets with LINQ to SQL. These result sets can be easily processed by Table Modules.

The choice between the three patterns is basically influenced by the complexity of the domain logic. The initial effort for developing a Domain Model is rather high and therefore Transaction Scripts are more suitable when dealing with simple domain logic. On the other hand complex business logic needs object-oriented possibilities to be realized in a cost efficient way. Figure 5.1 depicts the relationship between complexity and effort for different domain logic styles.

Stored Procedures. Database engines provide mechanisms like Stored Procedures for applying business logic as well. For example the calculation of the CAN message identifier could be easily realized with a Stored Procedure. The advantage of this approach are modularity, portability and in general a good performance. The major drawbacks are that most database environments do not offer gut structuring mechanisms and it requires a lot of effort to realize more complex business

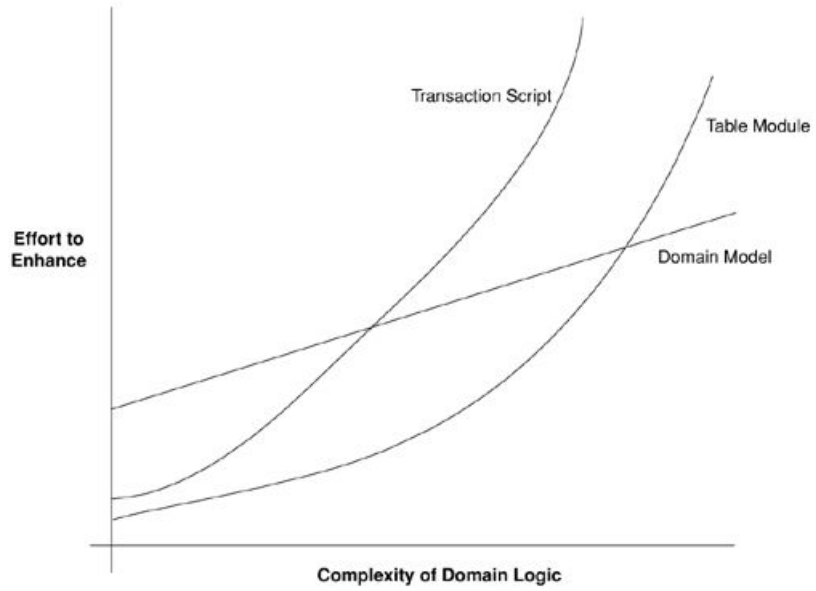

Figure 5.1.: Relationship between complexity and effort [7]

logic without fully object-oriented methods.

For the CORA the Table Module pattern is the most suitable approach because the business logic is rather simple compared to the data access and the presentation. Moreover the table centric approach fits perfectly with the design of the Data Access Layer and the Generic Controller Pattern. The Table Modules are represented by the Manager classes within the Business Application Layer of the CORA. Every Manager takes care of a set of CAN bus entities. For example there exists a CANModulManager, a FunktionCodeManager and a SignalManager. If the complexity of the business logic increases over time a shift to a Domain Model is extremely easy to realize because the Data Access Layer provides all the needed functionality. As a consequence the additional Business Objects like a CANModulBO can receive all the data from the corresponding Data Transfer Objects and add business functionality with the help of additional methods. The Data Transfer Objects can be mapped to the Business Objects or inheritance can be used to realize the data exchange. More information concerning the data transfer between the objects provided by the Data Access Layer and business logic realizing objects can be found in [26].

5.1. Communication

Besides executing business requirements, the Business Application Layer has to coordinate the communication between the Data Access Layer and the Presentation Layer in both directions. The Managers of the BAL follow the same generic strategy as the Controllers of the DAL and therefore inherit from a GenericManager. Listing 5.1 shows the class definition of the GenericManager. The corresponding Controller and the DTO are specified as generic parameters. Moreover, the type of the CAN entity and the related history are provided for simple sce-

narios when an assembled DTO is not needed.

Listing 5.1: Generic Manager class definition

```
public class GenericManager<TController, TEntity, TDTO, THistory> :
    IGenericManager<TDTO>
    where TDTO : class
    where TEntity : class
    where THistory : class
    where TController : IController<TDTO, TEntity, THistory>, new()
    {
      protected SettingsDTO settings;
      protected TController controller;

      public GenericManager(SettingsDTO settings)
      {
        this.settings = settings;
        controller = (TController)Activator.CreateInstance(typeof(TController));
      }

      public GenericManager()
      {
      }
}
```

The GenericController provides methods to call the CRUD methods of the particular Controller. In many cases the objects are just forwarded and no business logic is applied, but the CORA architecture considers scalability. These methods shown in 5.2 can be enriched with additional business logic during the future development.

Listing 5.2: Basic methods provided by the GenericManager

```
public List<TDTO> Insert(List<TDTO> insert)
{
    return controller.Insert(insert);
}

public List<TDTO> GetList(TDTO search, QuerySettings querySettings)
{
    return controller.GetList(search, querySettings);
}

public void Update(List<TDTO> update)
{
    controller.Update(update);
}

public void Delete(List<TDTO> delete)
{
    controller.Delete(delete);
}
```

For sure this is only a small excerpt of the provided methods because the Managers prepare the method invocations for the Controllers. This means that in many cases a Controller only has to specify one method signature, for example GetList(search, querySettings). If the caller in the Presentation Layer does not need to specify custom settings for the query, the Manager provides an additional method with only one parameter and will simply call the corresponding method with a second default parameter. This explains the numerous method definitions within the Managers. Moreover, simple CAN bus entities like connectors and hardware pins are assembled into a common Manager class, like a HardwareManager.

In addition to the Managers, which manage the business logic for CAN entities, a DTOManager takes care of mapping information entered by the user in the Presentation Layer to the DTOs. The class takes a collection of values from a presentation control and matches the names to the properties via Reflection.

5.2. Interfaces

The BAL is not only responsible for managing the communication within the application but also to external systems. The CORA interacts with the Rosenbauer Service Tool (RST) which finally configures the CAN bus systems on the vehicles with the help of the meta-data delivered by the CORA. This interaction is based on the exchange of comma-separated value files (CSV files). As a consequence the CORA Business Application Layer provides objects to realize the import- and export functionality.

5.2.1. Import

The name space import contains all the classes which are relevant for import scenarios. An abstract base class called Importer provides basic information, like a standard import location, for every specific import implementation. The MultipleFileImporter is another base class which provides shared functionality for classes that are designed to import multiple files. For instance, the function for retrieving data from the files and an in-memory collection for storing the retrieved data. The import process includes the following stages:

1. Locate the input files.
2. Open the input files.
3. Retrieve data according to a given structure.
4. Store the retrieved data in an in-memory collection.
5. Apply business logic.
6. Save the retrieved data persistently.

Regarding the import process of CSV files it is important to map the values of the single columns of the import file, indicated by a delimiter (in this case a comma), to the corresponding in-memory objects. This mapping can be realized with the help of a key value collection like a generic Dictionary in a .NET environment.

5.2.2. Export

The export mechanism is realized with a similar object-oriented approach. Every class responsible for an export use case inherits from a common base class called Exporter. This common base class provides general properties for storing the file path, the file name, the file extension, the delimiter and references to

other objects. A fundamental use case of the CORA is the export of the settings for a specific CAN module (or CAN controller). Therefore the export file needs aggregated information out of numerous database tables. Retrieving this information is achieved with the help of the Data Access Layer. Fine grained methods create the export file. One method is responsible for the creation of the head line, another for the creation of the CAN signals, and additional methods create the entries for the hardware configuration, the function codes including parameter codes, and the error codes. If the complexity of the export logic increases the single ControllerSettingsExporter class can be divided into multiple objects. Again the mapping to columns is crucial, especially in an internationalized environment where one language column can be added or removed easily. Therefore every export column is represented through an ExportColumn object for enabling a comfortable composition of the export files. The columns of an export file are managed by the ExportColumnManager. Moreover, two approaches are implemented considering different export formats and depending if the export process results in a single or in multiple files.

1. Single file: For a single export file the output is directly sent to the client with the help of HttpContext.Current.Response. In this case the user receives a file which can be stored locally.

2. Multiple files: The content for multiple export files is written with the help of a FileStream and a StreamWriter. In addition to this it is possible to archive the created files in a zip archive using SharpZipLib[1].

During the development of the CORA it was impossible to change the communication format from CSV to XML because legacy systems are only able to understand CSV at the moment. However the CORA can deal easily with the shift from CSV to XML because the export data is represented by in-memory collections which simply have to be transformed into the desired output format.

[1] http://www.sharpdevelop.net/OpenSource/SharpZipLib/Default.aspx

6. Presentation Layer

The with business logic enriched data has to be presented to the user for further processing or display. The rise of Web-browser-based user interfaces has changed the landscape of enterprise applications significantly because they are often preferred to rich-clients which run the presentation on the client. As a Microsoft Access application the EVI provided a rich-client presentation. Therefore the users are used to fast responsiveness and mighty user interface elements like combo boxes. Nevertheless, scaling the management of CAN data up to a worldwide extent (including multiple production locations and CAN bus systems) requires the benefits which come along with Web applications. First of all universal access is guaranteed because any connected device can access the Web application with a URL. Moreover no extra installation of client software is required and a central Web server eases the update process tremendously. In addition, a common user interface and huge support in the software engineering industry favor Web applications as preferred presentation mechanism.

As for the other layers, there exist numerous patterns for the Presentation Layer and again many of them can be found in [7]. In general a request has to be processed on the server-side and an appropriate response has to be presented to the client. In many cases the response is a dynamically generated HTML document. Investigating the request-response principle, a first separation into two parts makes sense. The first part interprets the request and the second part takes care of the response formatting. This separation of concerns is a fundamental approach but often neglected when it comes to presentation of information.

6.1. Separation of Concerns

In fact the separation is an old idea but first surfaced in user interfaces with the introduction of the Model View Controller (MVC) pattern which acts as a basis for many other patterns regarding presentation. Basically the Model View Controller pattern states that nonpresentation logic should be factored out by dividing the user interface interaction into three distinct roles as shown in figure 6.1.

There exist numerous different ways for implementing a MVC pattern but the distinction between the three roles must be visible.

- Model: The model is an object that represents information about the domain, including data and in many cases business logic as well.
- View: The view represents the display of the model in the user interface. It is possible that multiple views exist for a single model for different display purposes.

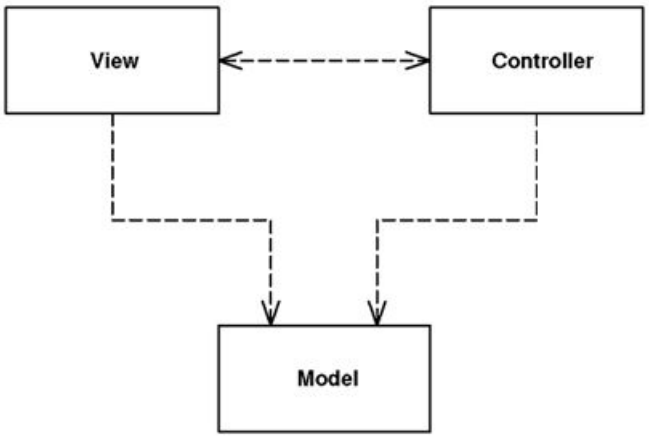

Figure 6.1.: Model View Controller [7]

- Controller: The controller handles the user input, manipulates the model and causes the view to update appropriately.

According to [7] the MVC pattern comprises two principal separations. First separating the presentation from the model and then separating the controller from the view. These two separations implicate many advantages. First of all, software developer can specialize in working with a model (including business logic and database interaction) or in designing a sophisticated view and providing a good user interface. Moreover, when the model is clearly separated, entirely different views can be created to display the same model. For example the hardware configuration of a CAN module can be presented by a rich-client, a Web browser, a remote API or even a command-line-interface. Finally a model free of any visual objects is usually easier to test.

In general the separation between presentation and model is easier to achieve than to assert independent controllers. Furthermore the term "controller" is used in multiple ways (e.g., not to be confused with the Controllers which act as DAO within the DAL) and therefore the term input controller is more significant for the controller in the MVC pattern. The following sequence diagram depicted in figure 6.2 illustrates how the model, view and input controller work together with a Web server.

An input controller handles an incoming request and derives information. Then the controller triggers particular model objects to gather information to build the model for the response. When the model is finished the control is returned to the input controller which investigates the result and decides which view is most suitable for displaying the response. Finally the controller passes the control to the view and the HTTP response is generated. As pointed out before, there exist multiple occurrences and different implementations and frameworks for the Model View Controller pattern. Therefore a detailed evaluation is important to choose a suitable architecture for the Presentation Layer for a certain domain.

Implementing Model View Controller in ASP.NET. The code-behind feature of the Microsoft Visual Studio .NET development system makes it easy to sepa-

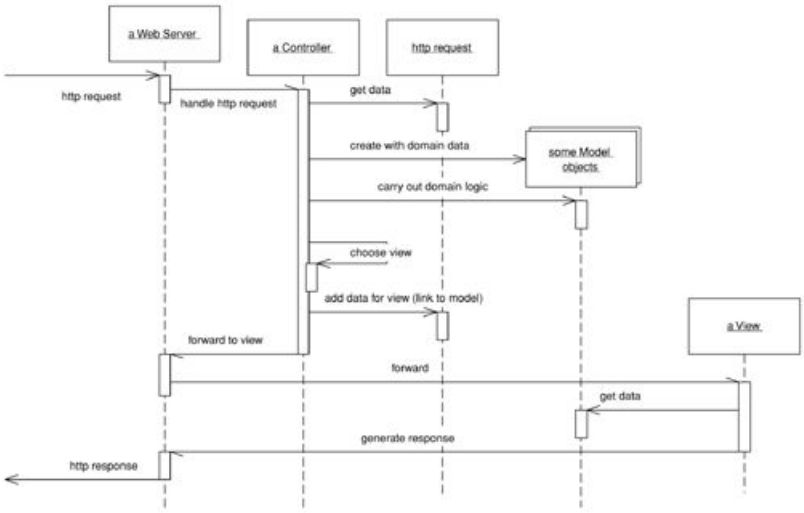

Figure 6.2.: MVC in action together with a Web server [7]

rate the presentation (view) code from the model-controller code. Furthermore, each ASP.NET page has a mechanism that allows methods, that are called from the page, to be implemented in a separate class [18].

6.1.1. Page Controller

A Page Controller handles a request for a specific page or action on a Web site [7]. Having one module on the Web server which acts as the controller for each page on the Web site does not work out exactly, because a link may redirect to a different page, but the controller is associated with each action. The role of the Page Controller can be described as the following. The controller receives a page request, extracts any relevant data, invokes any updates to the model, and forwards the request to the view. The view in turn depends on the model for retrieval of data to be displayed. Defining a separate page controller isolates the model from the specifics of the Web request - for example, session management, the use of query strings or hidden form fields, or passing of parameters to the page [22]. Figure 6.3 depicts the structure and the relation to the view and the model.

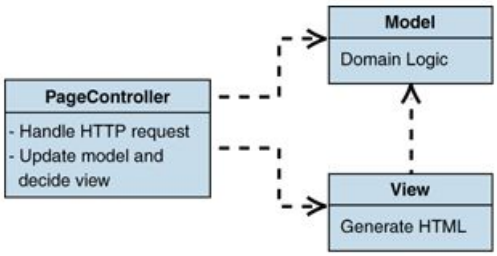

Figure 6.3.: Page Controller structure [22]

To avoid significant code duplication, which results of the creation of a separate controller for each Web page (or action), a BaseController class which incorporates

common functions should be considered. Each individual Page Controller can inherit this common functionality from the BaseController. In addition to inheriting from a common base class, the definition of helper classes (which can be called by the controllers to perform common functions) leads to more structured code. Figure 6.4 illustrates the approach.

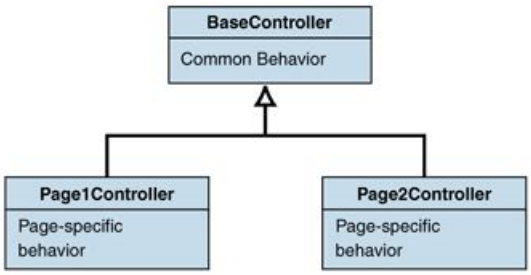

Figure 6.4.: Using BaseController to eliminate code duplication [22]

Many Web application frameworks provide a default implementation of the Page Controller but make it very easy for the developer to combine view-related code with controller-related code. Therefore many approaches favor the Front Controller patterns where a controller handles all requests for a Web site. In this case a single controller and a hierarchy of commands solve the decentralization problem present in the Page Controller. Nevertheless, the code-behind classes of the ASP.NET page framework provide an excellent mechanism for achieving separation and for implementing a Page Controller.

Implementing Page Controller in ASP.NET. The concepts of the Page Controller pattern are implemented in ASP.NET by default. The underlying mechanism of capturing an event on the client, transmitting it to the server, and calling the appropriate method is automatic and invisible to the implementer [19]. One crucial component of this underlying mechanism is the ASP.NET Page Life Cycle which covers among other things: initialization, instantiating controls, restoring and maintaining state. Within each stage of the life cycle of a page events are raised to run custom code. More information concerning the ASP.NET Page Life Cycle can be found at [16], [14] or [26]. Figure 6.5 depicts the structure of the code-behind pages implementation regarding PageController in ASP.NET.

Another approach would be the use of the ASP.NET MVC framework [1] instead of ASP.net Web Form model. The biggest advantages of this approach are that ViewState, PostBack events and code behind classes are more or less replaced by classes which explicitly define a controller and a view. One of the bigger drawbacks using this framework is that controls that take advantage of PostBacks or ViewState will not work. There was no explicit need to use the ASP.NET MVC framework for the CORA at the moment but the framework can be useful for the future development.

[1] http://www.asp.net/mvc/

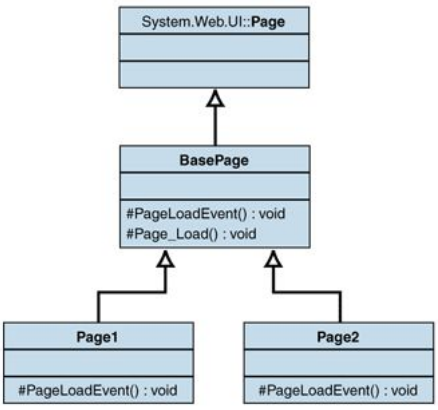

Figure 6.5.: Structure of the code-behind pages implementation [19]

6.1.2. Template View

A Template View or Template Method renders information into HTML by embedding markers in an HTML page as described in figure 6.6.

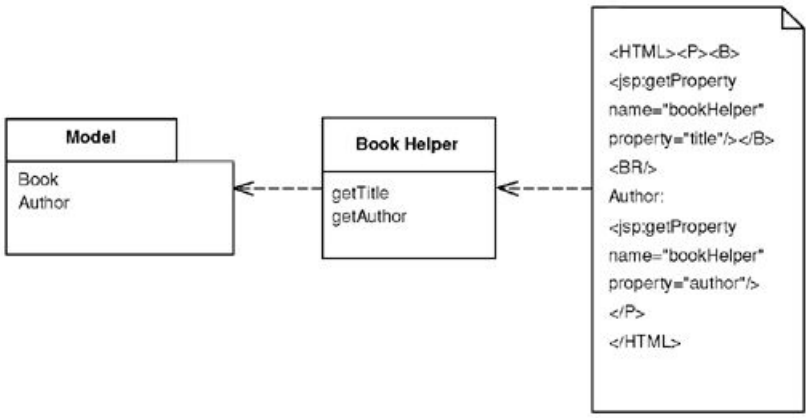

Figure 6.6.: Template View structure [7]

Dynamic Web pages cannot be created with regular HTML editor because their appearance varies for each result. A recommended approach to compose a dynamic Web page is to create a static page with placeholders or markers that can be resolved into calls to gather dynamic information [7]. Markers can be placed in the HTML in several ways, including HTML-like tags or special text markers in the body text. Many environments provide markers or even objects which render to HTML. In addition they allow embedding arbitrary programming logic into the template. This should be avoided regarding the separation of concerns principle and the accompanying disadvantages.

6.1.3. Separation of Concerns regarding the CORA

As [7] points out, realizing ASP.NET Web applications with Visual Studio works great with Page Controller in combination with Template View. The CORA architecture of the Presentation Layer continuous the generic approach of the previous

layers and defines a GenericBasePage which extends System.UI.Page and acts as a common BaseController for multiple Web pages.

Listing 6.1: Definition of GenericBasePage

```
1 public class GenericBasePage<TManager, TDTO> : Page
2     where TManager : IGenericManager<TDTO>, new()
3     where TDTO : class, IDTO, new()
4 {
5 }
```

As shown in listing 6.2 a ChildPage passes the type of the primarily associated Manager (a TableModule of the Business Application Layer) and the type of the primarily associated Data Transfer Object to realize common basic operations in the GenericBasePage. If a page is designed to manage a specific CAN entity "primarily associated" means the corresponding objects of other layers. For sure many pages use multiple Manager and DTOs but they have to be handled in the ChildPage.

Listing 6.2: Definition of ChildPage

```
1 public partial class CANModulDetails : GenericBasePage<CANModulManager,
      CANModulDTO>
2 {
3 }
```

The fundamental task of the GenericBasePage is to initialize and process the following parameters which are relevant for the whole application and not only for specific Web pages like:

- The authenticated user
- The application language
- The production location
- The CAN bus

Preferred default settings are stored in the database for every CORA user, like the preferred language, the preferred production location and the preferred CAN bus. Therefore the GenericBasePage verifies if the user has already applied any particular settings, for example has decided to work on a specific CAN bus, otherwise the preferred default settings are applied. While working with the CORA, the application settings have to be managed for the whole session, which is realized with the help of the SessionManager. Moreover, the inheritance level of the BasePage pattern can be increased for certain scenarios. For example all the pages which provide the possibility to copy a particular CAN bus entity like a CAN module or a function code to another CAN bus, inherit from CopyBasePage which provides common validation logic. The class CopyBasePage in turn inherits from a class called PopupBasePage which provides initial settings for pop-ups.

6.2. Data Presentation

The aim of displaying data is to convey information, and therefore a clearly understandable structure and a supportive layout is required. In order to provide

a smooth shift from the EVI to the CORA, the existing navigational structure was considered and retained to a reasonable extent. A detailed description concerning structure and layout of the old Microsoft Access application can be found at [29]. A first draft of a CORA Web page for managing a CAN bus entity is shown in figure 6.7.

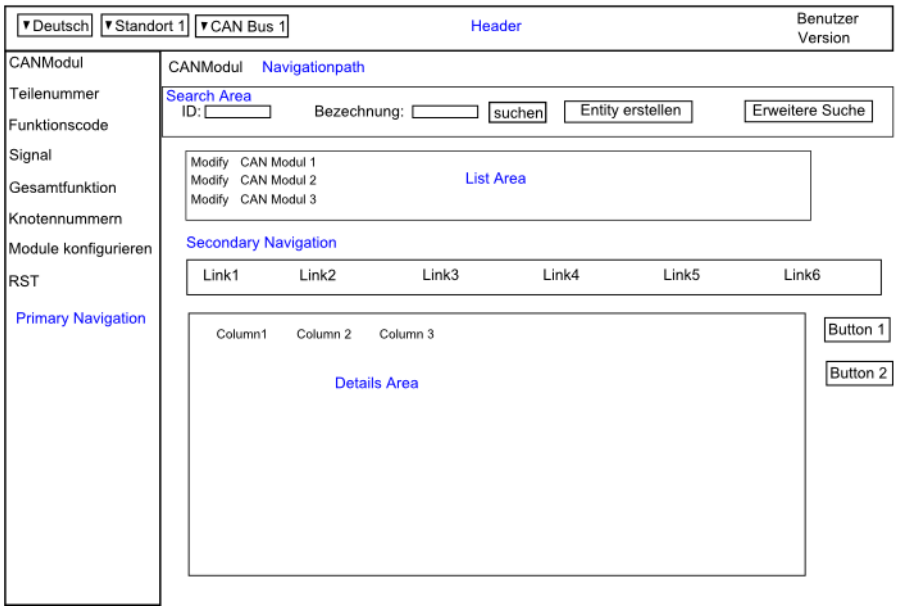

Figure 6.7.: First draft of the CORA interface

Such a screen sketch can be quickly realized with any graphical software and provides an assailable basis for further discussion to understand the UI needs of the different stakeholders. Figure 6.8 shows a CORA Web page for managing CAN module with software. The fundamental elements are described in the following enumeration whereas the enumeration numbers refer to the numbers on the screen.

1. The three drop-down menus provide the mentioned general settings for the application. Furthermore the current user and role are displayed.

2. The left-sided vertical menu acts as primary navigation for the different CORA Web sites. It adapts accordingly to the role of the authenticated user. As a consequence more menu items are provided for a system administrator than for a CAN bus administrator. The selected item is emphasized.

3. The bread crump navigation menu helps the users to keep track of their location within the CORA Web pages.

4. The filter controls allow filtering the subsequent entity list accordingly to multiple parameters. Moreover it is possible to execute several operations for these entities.

5. The entity list displays the CAN bus entities accordingly to the applied filters. Due to the possibilities offered by the DataPresenter the list can

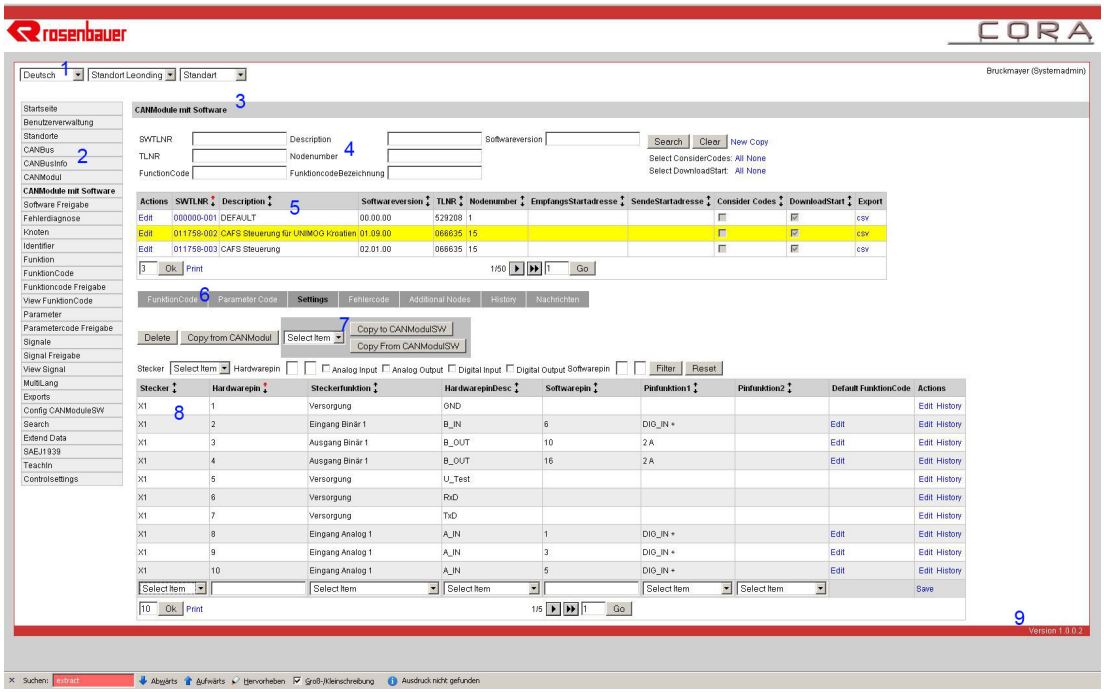

Figure 6.8.: CORA screenshot

be sorted, browsed and the number of the per page displayed entities customized. Furthermore it is possible to edit, delete, create and select items. The details of the selected entity are displayed underneath.

6. The horizontal secondary navigation menu enables the user to navigate through the details of a selected entity. For example the function codes or the hardware configuration for a specific CAN module can be viewed and modified.

7. As for the corresponding entity, certain operations can be performed for the detail items of a selected CAN bus entity. For example all hardware configuration entries can be deleted for a selected CAN module. In addition, filters can be applied to customize the listing of the detail items.

8. The second DataPresenter displays the detail entries and again sorting, browsing, editing, deleting and inserting functionality is offered.

9. The footer only displays the current version of the CORA at the moment.

The consistent layout for the pages of the CORA Web application is realized with the help of Master Pages and Nested Master Pages which define the look and feel and standard behavior for all of the pages (or a group of pages). The individual content pages contain the content which is intended to be displayed and when a user request a content page, it merges with the master page to produce output that combines the layout of the master page with the content from the content page. The Site.master defines the global layout of the CORA and subgroups are defined by nested master pages like Search.master for all the sites

which offer advanced search functionality.

Comparing the user interface of the EVI and the CORA, one of the most significant differences is the handling of the segregation between input possibilities and the simple display of information. The EVI defines distinct Access Forms, with basically the same layout, for editing and viewing scenarios. Furthermore form objects which provide input possibilities for the user submit the changes instantly to the database. Moreover the primary navigation of the main menu form is divided into two columns, whereas the left sided column offers access to parts of the application which are accessible for default users and the navigational elements of the right sided column are reserved for administrative users. The CORA takes a slightly different approach. The user can transform objects which serve display purposes into objects which accept user input if needed. Moreover the presentation objects are adjustable according to the role of the authenticated user. Therefore the primary navigation is not divided but will hide some menu elements if the user does not possess sufficient rights. In addition some presentation controls are enabled or disabled according to the role of the user.

Investigating the graphical user interface of the EVI and the CORA it becomes obvious that the preferred way to manage CAN bus entities is the display of data in an extremely flexible tabular form. Furthermore it must be possible to retrieve a subset of specific CAN data, according to a user defined amount of displayed items, to enable efficient working. Therefore paging and sorting of data has to be fully supported by the CORA user interface. The ASP.NET environment provides predefined controls for displaying data. Table 6.1 shows a comparison of the different data controls. Due to the rich out of the box functionality, and the through templates improved control over the rendering process, the in ASP.NET 3.5 introduced ListView control was chosen as primary control for displaying data.

Control	Paging	Data Grouping	Flexible Layout	Update, Delete	Insert	Sorting
ListView	x	x	x	x	x	x
GridView	x			x		x
DataList		x	x			
Repeater			x			

Table 6.1.: Comparing ASP.NET data presentation controls according to [8]

Paging. In addition to the ListView, a new control called DataPager was introduced with ASP.NET 3.5. Unfortunately this new control is very limited because it only works with the ListView control and both controls have to have Viewstate enabled at the moment. Moreover, the DataPager has no Paging Events and no SelectedPageIndex property and it is not possible to overwrite the paging behavior manually [30]. Furthermore the CORA data presentation control has to support server-side paging. Retrieving all configuration entries and trimming the result on the client for displaying purposes would harm performance significantly. In this

case only the result specified by the number of displayed items has to be retrieved. In fact the DataPager offers built-in server-side paging in combination with data source controls but not for custom collections returned by business objects. All these limitations lead to the development of a custom pager control for the CORA called EviPager. Worth mentioning at this point is the fact that all custom server controls of the CORA still have the "EVI" prefix, because the name of the EVI successor was not defined at the moment of their creating. Renaming should take place within the next refactoring iteration. The EviPager provides all the relevant information for the paging process like the current page and the customized page size in an object-oriented way and can be easily extended. Furthermore the layout and the images for the buttons can be arbitrarily adapted. Any user interaction triggers an event which has to be captured for further processing.

Sorting. Although the ListView offers built-in sorting possibilities in combination with data source controls, the requirements are the same as for the paging process. Therefore an independent expendable object which provides all the necessary sorting information for the Data Access Objects is the best solution. For this purpose the classes SortableTableHeaderRow and HeaderField were generated. Any presentation control simply has to provide a collection of Headerfields and the sorting possibilities are automatically realized.

Presentation control. Due to the previous mentioned data presentation requirements including paging and sorting it is obvious that the final CORA data presentation control consists of numerous objects. Furthermore these objects have to be integrated into a common template because editing every tabular presentation control of the whole Web application would end in a maintenance nightmare. For example future requirements demand an additional headline above the SortableTableHeaderRow, changing every presentation template would be extremely inefficient. The first idea was to load common templates dynamically with the help of an ItemPlaceHolder within the LayoutTemplate of the ListView. This raises some problems as described in [24] and even becomes impracticable if the template contains custom user- and server controls. The solution comes in form of the DataPresenter, a custom user control which serves as a container control for all the objects which make up the final presentation as shown in figure 6.9.

Figure 6.9.: CORA presentation control

The DataPresenter is an ASP.NET user control which consists of two files, a

template- and a code file. Listing 6.3 shows an abstract of the template file combining the object for sorting, the ListView for displaying the data, and the control for paging.

Listing 6.3: Abstract of the DataPresenter template file

```
<esc:CustomTable ID="tblWrapper" runat="server" CellPadding="0">
    <esc:SortableTableHeaderRow ID="sorterRow" runat="server" OnSorterCommand="
        SorterRow_Command"
        CssClass="Header" />
    <esc:CustomTableRow>
        <esc:CustomTableCell>
          <esc:EviListView ID="lvContent" runat="server">
               <LayoutTemplate>
                    <tbody id="itemPlaceholder" runat="server" />
               </LayoutTemplate>
          </esc:EviListView>
        </esc:CustomTableCell>
    </esc:CustomTableRow>
    <asp:TableFooterRow ID="tfrFooter" runat="server" CssClass="Footer">
        <asp:TableCell ID="tcFooter">
            <asp:Table ID="tblFooterElements" runat="server" CssClass="
                FooterElements">
                <asp:TableRow ID="trFooterElements" runat="server">
                    <asp:TableCell ID="tcPageSize" runat="server" CssClass="
                        PageSize" Wrap="false">
                        <asp:TextBox ID="tbPagesize" runat="server" Width="25"
                            />
                        <asp:Button ID="btnPagesize" runat="server" OnClick="
                            btnPagesize_Click" Text="<%$ Resources:Site, Ok %>"
                            />
                    </asp:TableCell>
                    <asp:TableCell ID="tcPrint" runat="server" CssClass="Print">
                        <asp:HyperLink ID="hlPrint" runat="server" Text="<%$
                            Resources:Site, Print %>" />
                    </asp:TableCell>
                    <asp:TableCell ID="tcPager" runat="server" CssClass="Pager"
                        Wrap="false">
                        <esc:EviPager ID="pager" runat="server" OnPagerCommand="
                            Pager_Command" />
                    </asp:TableCell>
                </asp:TableRow>
            </asp:Table>
        </asp:TableCell>
    </asp:TableFooterRow>
</esc:CustomTable>
```

The ListView is the heart of the DataPresenter, but as the tag esc:EviListView indicates it is an extended ListView and not the default implementation offered by the ASP.NET environment. The extension provides any imaginable flexibility but first of all, without using a predefined data source control a custom implementation for accessing user input is needed. For this purpose the EviListView provides a NewValues collection and numerous methods gather the data while iterating over all the controls specified within the ListView templates.

The code-behind file of the DataPresenter carries a lot of presentation logic and provides properties of the type ITemplate and the PersistenceMode.InnerProperty which specifies that the property persists in the ASP.NET server control as a nested tag. As a consequence the DataPresenter offers the possibility to specify a template declaratively and map it to the templates of the ListView. This mapping adds additional flexibility. For example the default ListView has different templates for scenarios which require similar controls. Viewing an item only needs a control for simple textual representation but editing and inserting

both require a control which takes user input, like a TextBox. Moreover, editing and inserting require the same controls in many cases and adding an extra template would cause additional maintenance costs. Therefore the DataPresenter introduces a ModifyTemplate which is mapped editing and inserting templates of the ListView. The few elements that differ within the ModifyTemplate like an edit and insert button for triggering the desired operations are automatically treated by additional routines with the help of strict naming conventions. For instance, a button with the id "lbtnUpdate" is only rendered if the DataPresenter is in edit mode. In case that the insert operation requires totally different controls than the editing operation, separate editing and inserting templates for both scenarios are provided as well. Code listing 6.4 depicts the simplified version of the declarative definition of a DataPresenter for displaying the hardware configuration of a CAN module.

Listing 6.4: Defining a DataPresenter

```
1  <esc:DataPresenter OnContentUpdating="dpSettings_Updating" OnContentInserting="
       dpSettings_Inserting">
2    <ItemTemplate>
3        <asp:HiddenField ID="hfHWBelegungID" runat="server" Value='<%# Bind("
             HWBelegungID") %>' />
4        <tr class="<%# Container.DisplayIndex % 2 == 0 ? "dp-item" : "dp-alternate-
             item" %>">
5            <td>
6                <asp:Literal ID="ltSteckerBez" runat="server" Text='<%#Eval("
                     SteckerBez") %>' />
7            </td>
8            <td>
9                <asp:Literal ID="ltHardwarePin" runat="server" Text='<%#Eval("
                     HardwarePin") %>' />
10           </td>
11           <td>
12               <asp:Literal ID="ltSteckerFunktionBez" runat="server" Text='<%#
                     Eval("SteckerFunktionBez") %>' />
13           </td>
14           <td>
15               <asp:Literal ID="ltSoftwarePin" runat="server" Text='<%#Eval("
                     SoftwarePin") %>' />
16           </td>
17           <td>
18               <asp:LinkButton ID="lbtnEdit" ValidationGroup="vgEdit" CommandName
                     ="Edit" runat="server"
19                   Text="<%$ Resources:Site, Edit %>" />
20           </td>
21       </tr>
22   </ItemTemplate>
23   <ModifyTemplate>
24       <asp:HiddenField ID="hfHWBelegungID" runat="server" Value='<%# Bind("
             HWBelegungID") %>' />
25       <tr class="<%# Container.DisplayIndex % 2 == 0 ? "dp-item" : "dp-alternate-
             item" %>">
26           <td>
27               <asp:DropDownList ID="ddlSteckerID" runat="server" />
28               <asp:RequiredFieldValidator ID="rfvSteckerID" runat="server"
                     ControlToValidate="ddlSteckerID"
29                   InitialValue="<%$ Resources:Site, DdlInitVal %>" Display="
                         dynamic" Text="*" ValidationGroup="vgModify" />
30           </td>
31           <td>
32               <asp:TextBox ID="tbHardwarePin" runat="server" Text='<%#Bind("
                     HardwarePin") %>' />
33           </td>
34           <td>
35               <asp:DropDownList ID="ddlSteckerFunktionID" runat="server" />
36           </td>
37           <td>
```

```
38                <asp:TextBox ID="tbSoftwarePin" runat="server" Text='<%#Bind("
                      SoftwarePin")␣%>' />
39            </td>
40            <td>
41                <table class="actions">
42                    <tr>
43                        <td>
44                            <asp:LinkButton ID="lbtnInsert" CommandName="Insert"
                                  runat="server" ValidationGroup="vgModify"
45                                Text="<%$␣Resources:Site,␣Save␣%>" Visible="false"
                                     />
46                        </td>
47                        <td>
48                            <asp:LinkButton ID="lbtnUpdate" CommandName="Update"
                                  runat="server" ValidationGroup="vgModify"
49                                Text="<%$␣Resources:Site,␣Update␣%>" />
50                        </td>
51                        <td>
52                            <asp:LinkButton ID="lbtnCancel" CommandName="Cancel"
                                  runat="server" Text="<%$␣Resources:Site,␣Cancel␣%>
                                  " />
53                        </td>
54                        <td>
55                            <asp:LinkButton ID="lbtnDelete" CommandName="delete"
                                  runat="server" Text="<%$␣Resources:Site,␣Delete␣%>
                                  " />
56                        </td>
57                    </tr>
58                </table>
59            </td>
60        </tr>
61    </ModifyTemplate>
62 </esc:DataPresenter>
```

Outstanding are the multiple event handler definitions like OnContentUpdating and OnContentInserting. This enables the DataPresenter to handle multiple events. Moreover, a technique called Event Bubbling allows a child control to propagate events up its containment hierarchy. With the extension of the ListView Event Bubbling offers multiple points for realizing certain logic. For instance, when the user clicks the link button to update some values of a CAN module the extended ListView catches the event with overriding the parent method OnItemUpdating, retrieves all the updated values, and raises another event which is handled by the DataPresenter above in the hierarchy. At this point additional common logic concerning the update process can be applied and another event is triggered which is finally handled by the page containing the DataPresenter. Here the updated values are passed to the corresponding Business Object. Figure 6.10 illustrates the update scenario.

If the next control in the hierarchy does not handle the Event it will be simply passed on, so one could easily omit the handling routine of the DataPresenter if no logic has to be applied at the moment. In addition to this Event Bubbling enables to bundle events. For example it is not required that every page defines an own method for handling the event raised by the SortableTableHeaderRow when a GUI element for sorting is pressed and an own method for every button click of the paging object. In both scenarios a data reload with the actual parameters is required. Therefore both events are handled within the DataPresenter and raise a common single event to reload the content of the page.

Transforming the DataPresenter. A lot of flexibility and transformation possibilities are gained due to the numerous templates and the component based

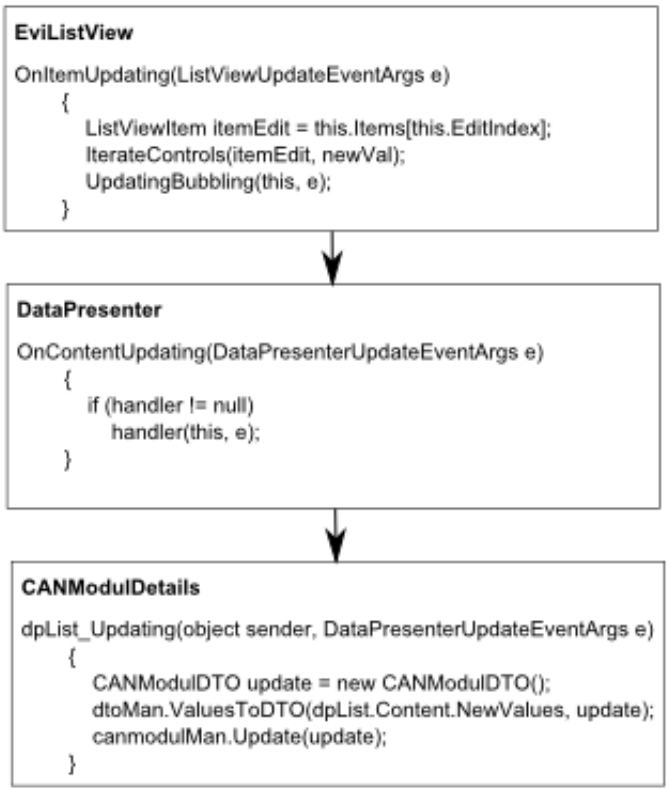

Figure 6.10.: Event Bubbling

structure of the DataPresenter. For example the DataPresenter can be set into insert mode which causes the contained and extended ListView to specify the location of the InsertItemTemplate and to render the appropriate HTML. Furthermore a select mode is available which does not only affect the ListView to display an additional column with select check boxes, but also enables controls like a button for submitting the desired select action. Moreover, the possibility to alter columns at will allows reusing a DataPresenter for different use cases which demand a similar result set. For instance, the use case "show reserved function codes of all users" requires one extra column compared to the use case "show reserved function codes of the current user", namely a column which displays the user who reserved the function code. In general the greatest common subset of shared columns defined in the template files is displayed by default and the other columns are set to "invisible". Within the code behind file, subroutines which are executed accordingly to the use cases add the additional column objects and the DataPresenter automatically changes the column status to visible.

Rendering HTML. Every ASP.NET Server Control can render itself into a presentation format which is sent back to the client in response to a Web request. In most cases this presentation format is HTML. One possibility to customize the rendering process is to overwrite the Render method of the base class System.Web.UI.Control. Another possibility is to write custom Control Adapters or to use existing solutions like the CSS Friendly Control Adapters[2]. The CORA uses

[2]http://www.asp.net/CssAdapters/

the CSS Friendly Control Adapters to render the menus with more CSS suitable HTML than provided by the default rendering process.

Internationalization. How internationalization affects the persistent storage of information is described in previous chapters. Among other things, internationalization for the Presentation Layer means to provide the graphical user interface elements in different languages. A prerequisite for internationalization of Web applications are independent resources which contain the language specific information. Moreover text, graphics and videos which should be easily replaceable should not be compiled within the application. The .NET framework provides an infrastructure for the management of resource files in different languages. The separation results from different filenames or sub folders according to RFC 1766. For the CORA again only the language code, but not the country or region code is important. Furthermore the CORA uses only Explicit Localization and not Implicit Localization because many resource entries are used on multiple pages.

6.3. User Interaction

The advantages of Web application are apparent but the user acceptance of a new enterprise application depends whether a comfortable interaction is possible. With HTML and additional client-sided technologies it is possible to generate an UI which does not have to fear the competition with the UI of rich-clients. Nevertheless, in Microsoft Access a so called combo box allows dynamic filtering of a drop down list accordingly to user input. This control is widely used in the EVI. Unfortunately there is no default representation of a combo box in HTML but there exist several solutions to provide a comfortable handling of large drop-down lists. Some browsers like Firefox already support the search for an entry within a drop down list by simple fast typing the sequence of the letters. Unfortunately at the moment the Internet Explorer, which is Rosenbauer's standard browser, only recognizes the first letter. Even if a user types the sequence of the letters very fast, the Internet Explorer always jumps to the word beginning with the letter. Moreover with a fully functional combo box users are able to correct the search input. For example they can delete a single letter, or add an additional one. This functionality is provided by the ListSearchExtender of the ASP.NET Ajax Control Toolkit[3], which is an open-source project built on top of the Microsoft ASP.NET AJAX framework and provides a rich control collection. Figure 6.11 demonstrates a ListSearch control to search a drop-down list displaying multiple CAN modules.

In order to provide a sophisticated interaction for the user the logical, temporal, and physical separation between the client and the server has to be considered. The rich server-based programming model provided by ASP.NET easily entraps developers to disregard the client-side which affects performance. For instance, it is totally unnecessary to send data to the server only to validate if the integer input lies within a predefined range. This validation can easily be realized with client-side scripts. Fortunately many ASP.NET validation controls provide client-side validation and there are numerous built-in possibilities for adding client-side

[3] http://www.asp.net/ajax/AjaxControlToolkit/Samples/

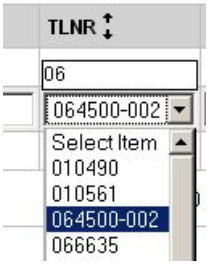

Figure 6.11.: Searchable drop-down list with an Ajax control

script to a Web page. The CORA takes a step forward and provides two classes to enrich client-side user interaction:

- ScriptInitiator
- ScriptManager

The ScriptInitiator is used for the dynamic creation of client-side code, for example JavaScript functions. Due to the fact that StringBuilders are used, this code can be arbitrarily composed as show in code listing 6.5.

Listing 6.5: Dynamically Creation of client-side code

```
1  public StringBuilder EnableDisableControls(List<Control> controls, CheckBox
       cbFlag)
2  {
3      StringBuilder sb = new StringBuilder();
4      sb.AppendLine("function␣EnableDisableControls" + cbFlag.ClientID + "()");
5      sb.AppendLine("{");
6      sb.AppendLine("var␣disabled␣=␣true;");
7      sb.AppendLine("if(document.getElementById('" + cbFlag.ClientID + "').checked
           )");
8      sb.AppendLine("{");
9      sb.AppendLine("disabled␣=␣false;");
10     sb.AppendLine("}");
11     foreach (Control con in controls)
12     {
13         sb.AppendLine("document.getElementById('" + con.ClientID + "').disabled=
               disabled;");
14     }
15     sb.AppendLine("}");
16     return sb;
17 }
```

The ScriptManager is responsible for managing the client-side script for a Web page. The class takes client code in form of a string and stores it within a collection. Finally the client script is rendered into the HTML response. Another approach is to use external files with client-side code to use the full debugging potential of Visual studio.

Input validation plays a tremendously important role for the CORA in order to guarantee the consistency of the complex CAN data. The default ASP.NET validation controls are suitable for most scenarios and a CustomValidator enables user defined validation logic on the server- and client-side. Moreover, the ExistsValidator inherits from the BaseValidator and provides multiple methods to validate if a particular entity already exists on a specific CAN bus. All validation controls can

be dynamically enabled or disabled. This becomes handy when copying multiple entities between two different CAN bus systems. In this case an exclusion of an entity from the copy process automatically results in the deactivation of all validation controls regarding this entity, without affecting the validation of the other entities.

7. Security

Sophisticated security mechanisms are an essential part of enterprise Web applications to prevent unauthorized actions. These include identifying users, granting or denying access to sensitive resources, and protecting data. The application's environment tremendously affects the security strategy and as a consequence threat modeling gained importance within Web Engineering. According to [14] threat modeling is a structured way of analyzing the application's environment for possible threats and ranks them according to certain criteria. In addition to this, Web applications have to face the following security topics according to [27].

- Authentication: A fundamental security design decision addresses the question if the application is available for anonymous users or if an authentication is inescapable. This raises the question if the users have to provide the information required for the login explicitly, or if the operation system can pass the login to the Web server.
- Access control: Which resources or pages are allowed to be accessed by which users?
- Application identity: The next question concerns the user context in which the Web application is acting on the server and if the identity of the authorized user or a dedicated identity is used.
- Accounts management: The management of the user accounts and user groups is another fundamental security decision.

ASP.NET as the underlying framework for the CORA provides built-in functionality for implementing security aspects. In [27] the security mechanisms of a typical ASP.NET Web application are described as the following:

- Access restriction based on IP-addresses: The Web server can check for each request whether the client is allowed to send requests at all.
- The next step addresses the browser authentication.
- After passing the IIS Web server, the request is forwarded to the ASP.NET framework. At this point the user identity has to be clarified. It is possible to forward the identity of the authenticated user or an independent identity.
- By means of the passed identity the framework checks the access rights, including database access or access to other application servers.
- Moreover, with the help of additional connection strings the identity can be modified again.

7.1. User and Role Management

For controlling authentication and access to certain resources the allocation of users to certain groups is a well established approach. ASP.NET supports a user group based security system, but the term "role" is preferred over the term "user groups". For instance, it is possible to define different roles declaratively in the web.config file, or one can use the Microsoft Active Directory, the user database of a windows system (SAM), or a Microsoft SQL server database. For the last mentioned approach a tool for generating all the needed tables exists and even a default Web interface is available for managing the roles.

The CORA distinguishes between the following roles:

- System administrator: This role stands above the affiliation to a specific production location and manages other users and their role memberships.
- Production location administrator: Users of this role are able to manage production locations and for sure perform all CAN bus related operations.
- CAN bus administrator: A CAN bus administrator can create, edit and delete different CAN bus systems and even copy CAN entities to another CAN bus. It is possible to assign the role CAN bus administrator only for a particular CAN bus system. If a CAN bus administrator switches to a CAN bus which is not explicitly allocated to him, only working rights are granted.
- Working: A working user can manage many CAN bus entities but not the CAN bus itself. For instance, the role is allowed to create a CAN module, or to configure hardware settings and to manage signals.
- Reading: A reading role is able to view a CAN bus but is not able to apply any changes.

The CORA Microsoft SQL database is used for storing these roles as mentioned in section 3.2.1. All the tables are manually designed because the default, tool generated role management does not meet the requirements considering the different production locations and CAN bus systems, because a single user can be a CAN bus administrator for a specific production location and a CAN bus worker for another. Furthermore many built in functionality like password management is not needed because the authentication of the windows system is used as described later on.

7.2. Authentication

ASP.NET distinguishes between two forms of authentication namely forms authentication and Windows authentication. The former provides a flexible approach and custom login pages whereas the latter uses existing windows accounts. As an enterprise application the CORA uses Windows authentication, and therefore no

login user interface is required, and no additional precautions against the interception of network traffic is needed. A drawback of using Windows authentication is that it is limited to Windows users. Moreover, it is important to know that Windows authentication is not built into ASP.NET but the responsibility lies by the IIS. Here several authentication strategies can be used including Integrated Windows authentication where user name and password are not transmitted, but the identity of an already logged in Windows user is passed automatically as a token. In this case the authentication takes place transparently and no user intervention is needed. Detailed instructions concerning the configuration can be found in [14].

7.3. Implementing CORA Security

In the case of the CORA it is important that an authenticated user only accesses resources or performs actions according to the assigned role. As a consequence the enterprise application has to take the following security requirements into account:

- Some Web pages are only accessible for certain roles. Therefore the primary navigation menu has to be adapted and access denied to certain resources.
- The user interface has to be transformed accordingly to the given role to deny certain operations. This includes simple enabling and disabling of buttons or the complete rendering prevention of insert elements of a DataPresenter.
- Certainly deactivating only user interface elements is insufficient, whole code blocks have to be prevented from execution.

First of all the user has to be automatically authenticated with Integrated Windows Authentication. The GenericBasePage checks if the request is authenticated and the EviRoleProvider takes care of the role management. Therefore the class inherits from RoleProvider to use the built in ASP.NET security mechanism including the Role Management Providers, which separate the functionality of role management from the data store that contains role information. Furthermore, the EviRoleProvider overrides the GetRolesForUser method to return the role names that the specified user is associated with as a string array. The first mentioned security requirement, namely the access restriction to certain Web pages can easily be realized with a configuration in the web.config file. Listing 7.1 depicts a role based Web site restriction.

Listing 7.1: Role based Web site restriction

```
1 <location path="Sections/Exports/Exports.aspx">
2     <system.web>
3         <authorization>
4             <allow roles="Systemadmin, Standortadmin, CANBus-Admin"/>
5             <deny users="*"/>
6         </authorization>
7     </system.web>
8 </location>
```

In this case only a system administrator, a production location administrator, or a CAN bus administrator is allowed to access a Web site which offers export possibilities, like the export of a configured CAN bus to the Rosenbauer Service Tool. The menu entry in the primary navigation is hidden as well. The fact that the role determination requires additional information from the application, namely the current selected production location and the current selected CAN bus affects the security implementation. When the user changes the production location with the help of the drop down-list in the header, the Web site has to adapt accordingly for the newly assigned role. This is not a problem if only GUI elements have to be transformed, but additional handling is required when the access to the current page is now denied. The CORA prevents this scenario by simply deactivating the drop down-list for switching a production location on Web sites which access is limited to certain roles and therefore have an authorization entry in the web.config.

For instructing a presentation control to render HTML accordingly to a security role, the SecurityManager has been introduced. This class takes a "minimum role" which indicates that the role of the authenticated user has to satisfy at least this security level for passing the validation.

Listing 7.2: SecurityManager access control

```
1  public bool GrantAccess(EviEnumerations.Rollen minimumRole)
2  {
3      bool grant = false;
4      switch (minimumRole)
5      {
6          case EviEnumerations.Rollen.Working:
7              grant = GrantAccessMinWorking();
8              break;
9          case EviEnumerations.Rollen.CANBusAdmin:
10             grant = GrantAccessMinCANBusAdmin();
11             break;
12         case EviEnumerations.Rollen.Standortadmin:
13             grant = GrantAccessMinStandortadmin();
14             break;
15         case EviEnumerations.Rollen.Systemadmin:
16             grant = GrantAccessMinSystemadmin();
17             break;
18     }
19     return grant;
20
21 }
22
23 private bool GrantAccessMinStandortadmin()
24 {
25     bool grant = false;
26     if (RoledID == (int)EviEnumerations.Rollen.Standortadmin) grant = true;
27     if (RoledID == (int)EviEnumerations.Rollen.Systemadmin) grant = true;
28     return grant;
29 }
30
31 private bool GrantAccessMinWorking()
32 {
33     bool grant = false;
34     if (RoledID == (int)EviEnumerations.Rollen.Working) grant = true;
35     if (RoledID == (int)EviEnumerations.Rollen.CANBusAdmin) grant = true;
36     if (RoledID == (int)EviEnumerations.Rollen.Standortadmin) grant = true;
37     if (RoledID == (int)EviEnumerations.Rollen.Systemadmin) grant = true;
38     return grant;
39 }
```

A SecurityManager instance is allocated to each DataPresenter. For example, the DataPresenter for managing the production locations receives a SecurityManager with a production location administrator as a minimum role, and the DataPresenter for managing CAN modules receives a SecurityManager with a working role. In both cases the GrantAccess method will indicate when the edit button should be rendered as enabled or disabled. For security purposes most pages and controls are equipped with an ApplySecurity method which is invoked shortly before the rendering process.

With the help of the PrincipalPermissionAttribute it is verified that users running certain code have been authenticated and belong to a specified role. For instance, if a method is declaratively protected by the PrincipalPermissionAttribute and users manage to find a workaround to invoke this method, or if one GUI element is accidentally enabled, a security exception is thrown. Code listing 7.3 depicts an example where only a system administrator is allowed to run the code to insert a new production location.

Listing 7.3: PrincipalPermission code execution

```
1  [PrincipalPermission(SecurityAction.Demand, Role = "Systemadmin")]
2  protected void dpList_Inserting(object sender, ListViewInsertEventArgs e)
3  {
4      StandortDTO insert = new StandortDTO();
5      dtoMan.ValuesToDTO(dpList.Content.NewValues, insert);
6      CANBusDTO insrtCanbus = new CANBusDTO();
7      insrtCanbus.Bezeichnung = Resources.Entity.Template;
8      standortMan.InsertStandort(insert, insrtCanbus);
9      base.InitStandortCon();
10 }
```

8. Deployment

Deploying an enterprise application is a complex process and many methodologies like the Rational Unified Process[1] provide detailed guidance. In the same way additional steps are required in order to make the CORA available for all departments which manage the CAN bus systems of Rosenbauer. First of all the existing IT-infrastructure has to be analyzed considering the CORA deployment. An Internet Information Services (IIS) Web server which runs as virtual server on an ESX[2] farm is used. Furthermore the server runs in combination with a Microsoft Office SharePoint Server (MOSS) and is protected through a Microsoft Internet Security and Acceleration Server and an additional firewall. After analyzing the infrastructure the following steps for deploying the CORA remain:

- Publishing and configuration of the database: The database schema of the development environment has to be published to the production environment.
- Publishing the CORA Web application: The source code including all the additional resources has to be deployed on the Web server.
- Configuration of the Web server: The Web server has to be configured in order to provide the deployed Web application according to the requirements.

Publishing the database. Microsoft SQL Server 2005 is used for the development environment as well as for the production environment. For publishing the database, the database schema of the development database is exported into a file and executed on the server. This file includes all the necessary instructions for creating the database. In addition, the database has to be filled with initial data, like users, roles, default production locations and CAN bus systems. Unfortunately the SQL Server Management Studio does not support the export of the database schema or the database content into a SQL file but Visual Studio 2008 provides this functionality with the SQL Database Publishing Wizard. Therefore the CORA database could be easily created. Regarding maintenance after the initial creation, all changes are first made and tested in the development environment and then applied to the database of the production environment. In this case database management tools are used instead of a file based information exchange.

[1] http://de.wikipedia.org/wiki/Rational_Unified_Process
[2] http://www.vmware.com/

Publishing the CORA Web application. In order to deploy an ASP.NET Web application the source files have to be compiled. There exist two possibilities regarding the compilation.

1. Compile at runtime: The Web application gets compiled for the first request or again in case of applied modifications. In this case the source of the development environment can simply be copied to the production environment, also referred as XCopy deployment. This enables instant modification within the production environment.

2. Compile at development time: A precompilation within the development environment provides a faster access of the Web application for the first request.

The CORA was precompiled and published with the built in Publish Wizard offered in Visual Studio 2008. A lot of settings are available for the publishing process but in the initial case the output folder of the CORA contained a sub folder called bin which includes all the required libraries. Other resources like the language files for the user interface, images, CSS and the template files remain in an editable format. This enables the possibility to apply modification to certain resources within the production environment. Therefore a strict deployment and maintenance policy is needed which states exactly what kind of modifications have to be applied in which environment in order to avoid inconsistencies. At the moment all modifications of the CORA Web application are first applied in the development environment and then deployed to the server. To keep track of the changes, the Version of the CORA is set in the AssemblyInfo.cs and updated accordingly. Furthermore, configuration files like the Web.config differ in both environments and must be handled carefully during the deployment process. In case of the CORA all the application settings are stored in the Web.config file and there exist two separate versions, one for the development environment and one for the production environment.

Web Deployment Projects. In addition to the default deployment possibilities offered in Visual Studio Microsoft provides an additional Visual Studio project template called Web Deployment Project with a GUI for further deployment options including [27]:

- Automatic precompilation through the translation process.
- More influence how the assemblies are assembled.
- Part of the web.config file can be automatically exchanged.

Web server configuration. The IIS provides tons of configuration options, some of them are described in [14]. For the CORA the following settings are essentially important:

- Enabled Integrated Windows Authentication.

- Access to the database server.
- Access to file system for accessing import files generating export files.

Therefore the Trust Level has to be set to FULL with the help of the IIS Manager.

9. Conclusion

Providing enterprise solutions for a certain problem domain is a complex process which is most likely to succeed with the help of sophisticated patterns and a tailored architecture. This master's thesis describes one possible approach for realizing a solution with the help of existing patterns and architectural guidelines by means of a real life example called CORA. The CORA enterprise solution enables the worldwide management of CAN data of fire-fighting vehicles. The flexible three layered architecture, which is commonly used in software systems, defines the structure of this thesis. But flexibility adds additional complexity and therefore technologies and frameworks gain more and more importance. In case of the CORA the primarily used technologies are Microsoft SQL Server, LINQ to SQL and ASP.NET. Consequently, on one hand the development of the CORA covers long established and well documented technologies and on the other hand brand new technologies which call for documented application in enterprise solutions. For instance at the beginning of the Data Access Layer development it was hard to find detailed literature dealing with the use of LINQ to SQL in a layered architecture. Furthermore, the Rapid Application development approach plays an important role in the .NET development community and advanced architectural principles are not that well documented. This thesis presents technologies and patterns for each architectural layer and how to link them together.

9.1. Lessons Learned

During the development of the CORA the author gained a lot of experience including positive and negative aspects. On the positive side one has to mention the fact that the described and newly introduced technologies like LINQ to SQL for ASP.NET enable a single developer to realize complex enterprise solutions. After some inquiries basically every sophisticated pattern could be applied, and every problem solved. On the other hand it was harder to start with .NET in the enterprise development world as with comparable technologies like JAVA from a subjective point of view. The reasons for this might be the popular and wide spread Rapid Application Development approach within the .NET world. This made it harder to find the answers to particular questions. For instance, the question for an appropriate architecture concerning the communication exchange between the layers (regarding the new LINQ to SQL technology) invoked a longer discussion including the reasons for neglecting standard data source- and presentation controls. In the end it was inescapable to try different approaches because detailed information of more complex enterprise solutions was hard to find. Another example is the design of the component based presentation control. The idea of composing a presentation control out of numerous single components enables a lot of flexibility and obviously inspired the development of the newly

introduced controls in ASP.NET 3.5 as well. Nevertheless, it was hard to find existing patterns considering a custom container control which was indispensable for maintenance and for applying generic logic. But the fact that every architectural problem was solved immediately and not deferred, according to the broken windows theory[1], contributes to the success of the CORA. Another lesson learned is the impact of an existing predecessor solution on the development of the successor solution. It makes sense to keep an eye on the existing solution if it has satisfied the company requirements for years, but one always has to evaluate the design principles again. For instance, it seemed reasonable to take the existing model of representing a function code signal including the identifier, byte- and bit position from the already existing solution at a first glance. While implementing this model, it quickly became obvious that it is inefficient to store the byte- and bit position in the same field as a decimal number, separated by a delimiter which is even culture specific. The creation of an additional CAN message entity and separating the information into atomic fields solves a lot of problems and in this case the existing predecessor solution led to a false track. In a nutshell, the author was able to learn a lot of lessons which can be summarized as follows:

- Model the "real world" according to object-oriented principles.
- If time allows, evaluate even proven concepts again.
- Solve any crucial architectural design problem immediately and not in the next iterative cycle.
- For each scenario, try to identify aspects which can be solved in a generic way and those which require tailored handling.

9.2. Future Work

Already during the development it became clear that the CORA offers a lot of new possibilities and needs ongoing development to unfold its full potential. Apart from the evolvement of the problem domain, the existing CORA offers plenty of room for improvements.

To begin with the Data Access Layer, LINQ to SQL seems to reach the limit as an object-relational mapper. Regarding the GenericController pattern, an advanced object-relational mapper enables to pass an already automatically assembled entity to the GenericController, instead of multiple single LINQ to SQL entities which represent a single table and have to be assembled manually.

Within the Business Application Layer the file-based communication to external systems could be enhanced with a more convenient transport format like XML. For sure, the communication partners and legacy systems have to be updated as well.

[1] http://en.wikipedia.org/wiki/Broken_windows_theory

For future work regarding the Presentation Layer the feedback provided by CORA users will contribute to further improvements like additional search functionality and extra validation logic. Moreover, they will give precious feedback about the page design.

Furthermore, from an architectural point of view the expansion from the layered CORA to a more service oriented approach should be considered. As [12] states enterprise software is tightly coupled with the internal organization, processes and business model of the enterprise. The process of managing CAN data will increase over time and maybe even additional applications are required for technological support. Service-oriented architecture (SOA) promotes the usage of loosely coupled services which provide certain functionality instead of monolithic applications. For instance, one could extract the logic for creating CAN message identifiers out of the Business Application Layer and create a service which could be used by any other application. Web services are one possible implementation technology for SOA. The .NET framework supports the usage and the allocation of Web services. Depending on the development of the Rosenbauer IT landscape it could be advantageous to provide the functionality of the CORA Business Application Layer with the help of multiple Web services. In this case another department which uses a different technology for developing solutions (for example JAVA) could use the existing implementation of the business logic offered by the CORA Web services within their applications. Last but not least the ongoing development of ASP.NET and hence the resulting potential improvements for the CORA have to be observed.

A. Database Diagram

On the following two pages a simplified version of the CORA database schema is described. For a better overview the language specific tables and the tables which contain the history entries are omitted.

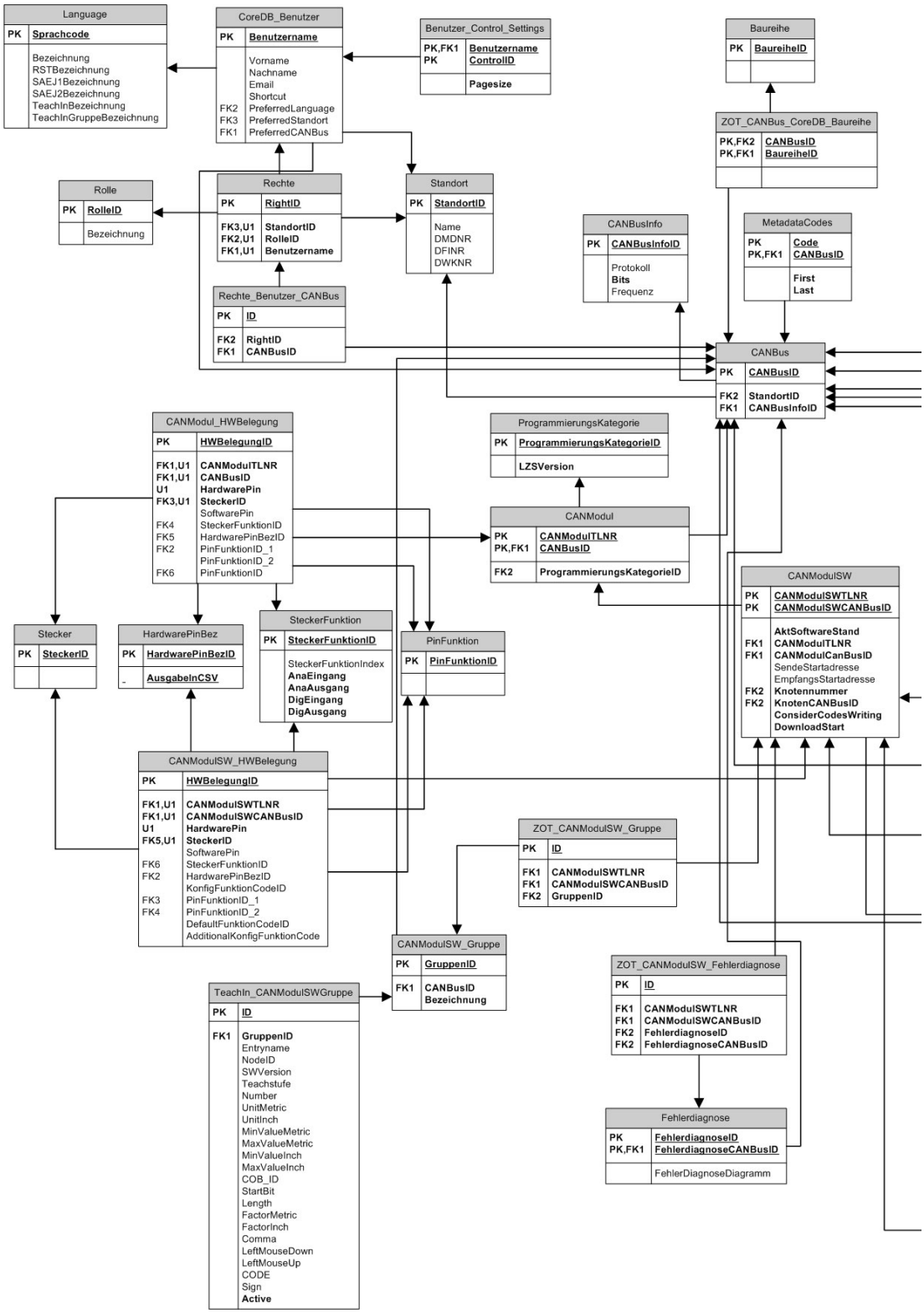

Figure A.1.: CORA Database Schema simplified version 1/2

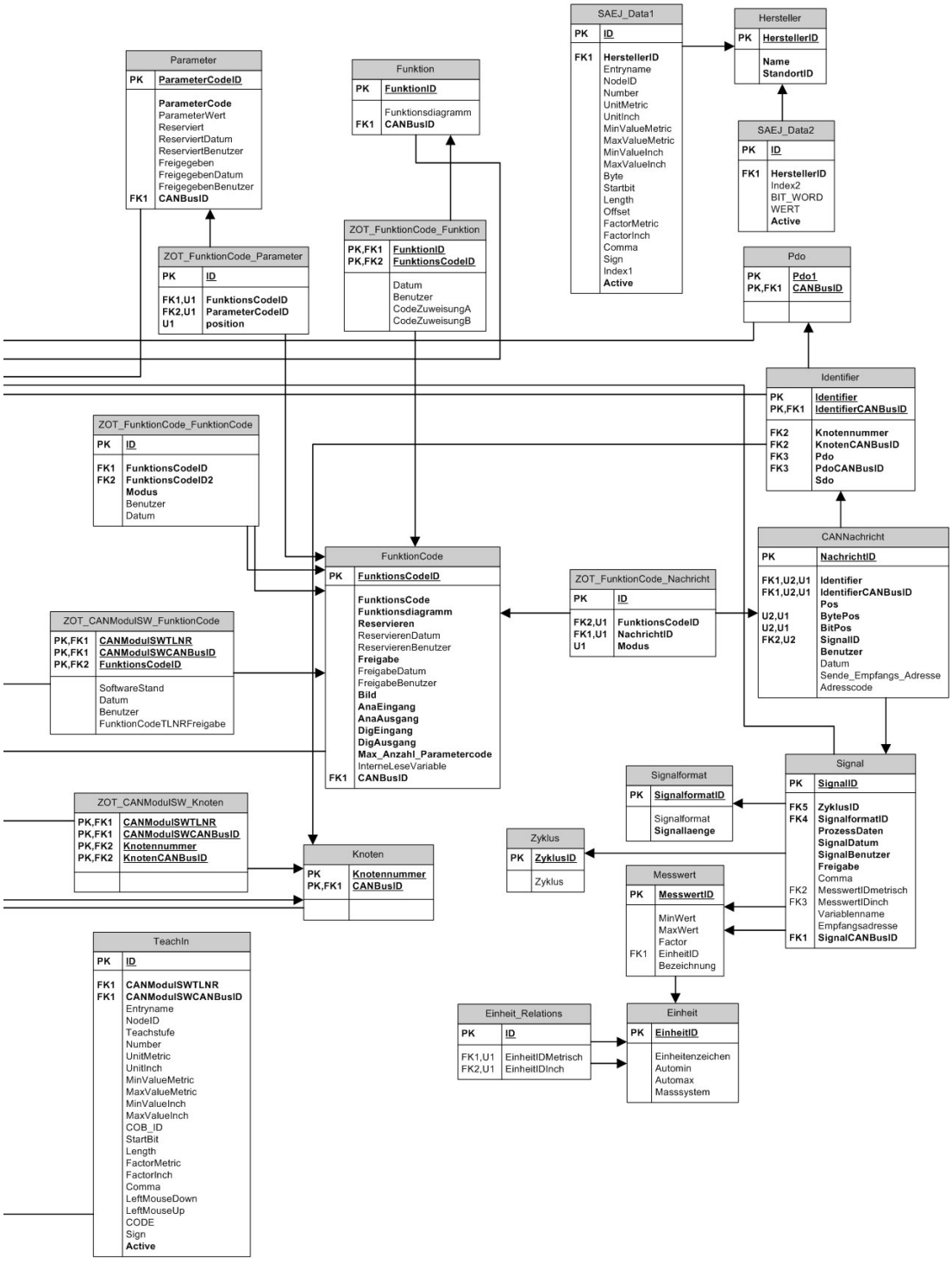

Figure A.2.: CORA Database Schema simplified version 2/2

B. Screenshots Frontend

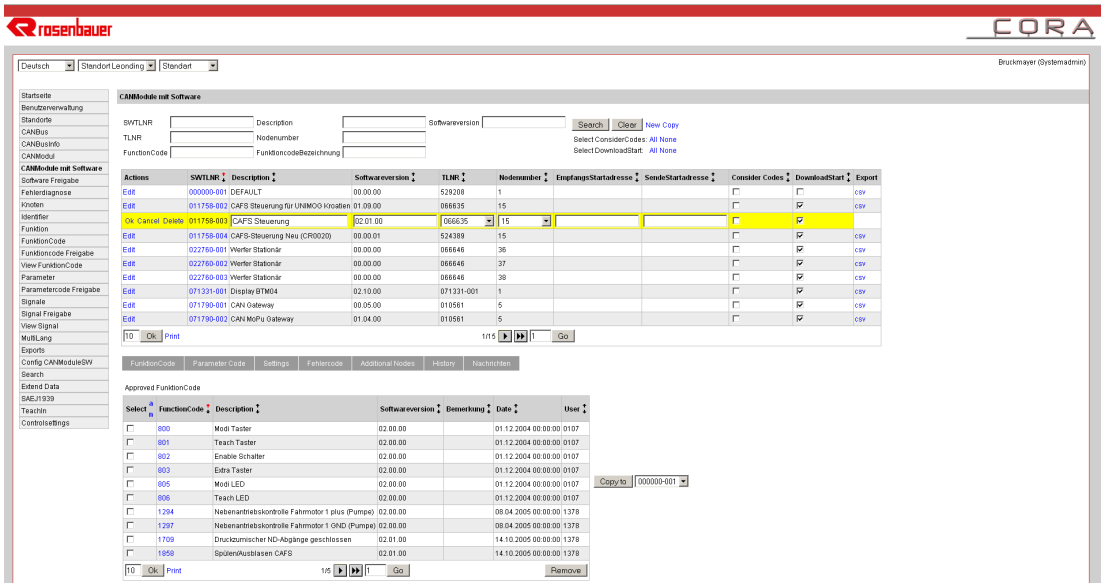

Figure B.1.: Edit CAN module

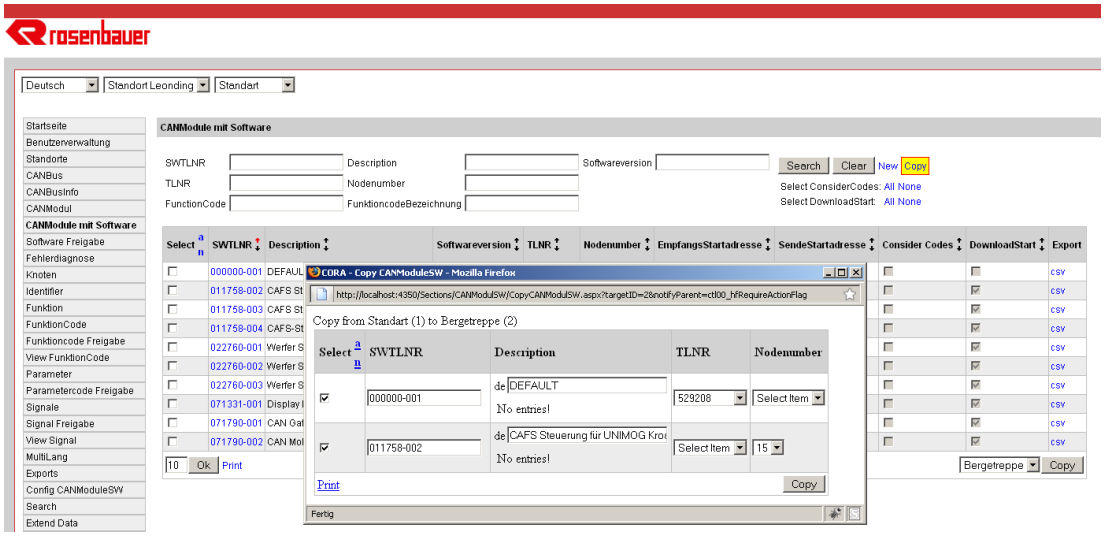

Figure B.2.: Copy CAN module

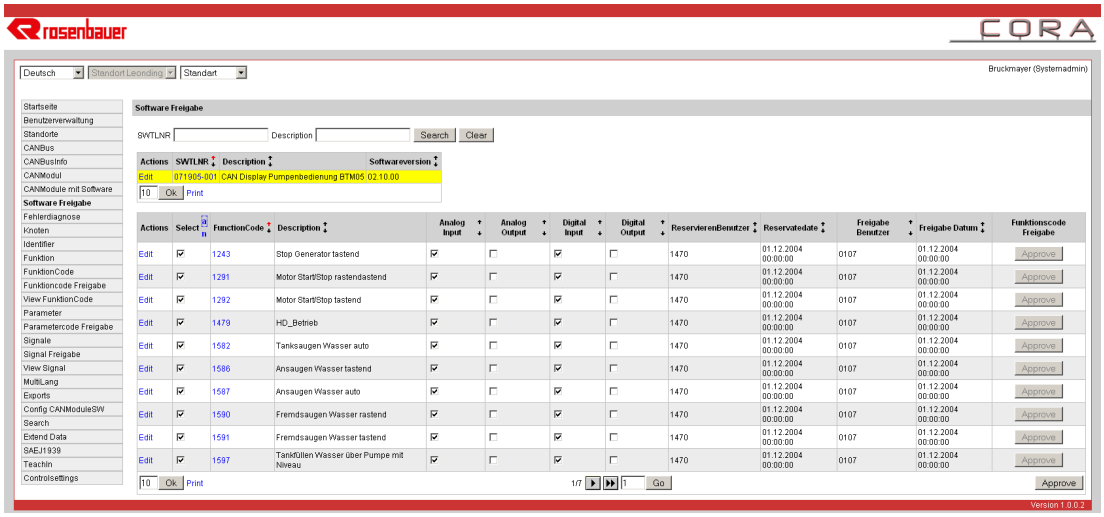

Figure B.3.: Approve codes

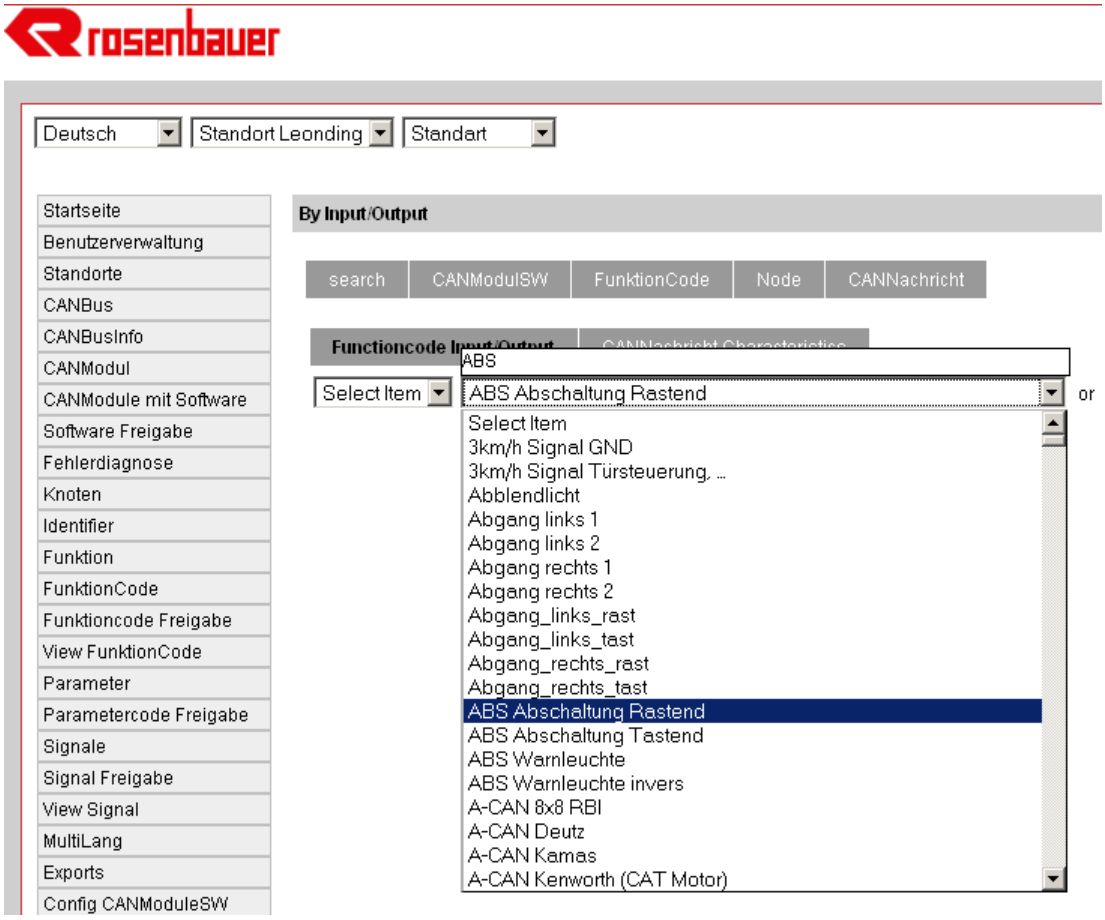

Figure B.4.: Select code

List of Figures

List of Tables

Listings

Bibliography

[1] Andany, J., Léonard, M., and Palisser, C. Management Of Schema Evolution In Databases. *17th International Conference on Very Large Data Bases (VLDB)* (1991), 161–170.

[2] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*. Addison Wesley, 1998.

[3] Deepak, A., Crupi, J., and Malks, D. *Core J2EE^{TM} Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2003.

[4] Demolsky, M. State of the Art in Java Enterprise Web Application Development. Master's thesis, TU Wien, 2006.

[5] Dumke, R., Lother, M., Wille, C., and Zbrog, F. *Web Engineering*. Pearson Studium, 2003.

[6] Foster, J. Visual Studio 2008 and Visio for Enterprise Architects. Web site. `http://www.gravitycube.net/blog/post/Visual-Studio-2008-and-Visio-for-Enterprise-Architects.aspx`; Last visit: 2009-07-31.

[7] Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., and Stafford, R. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[8] Ghanem, A. Comparing ListView with GridView,DataList and Repeater. Web site. `http://weblogs.asp.net/anasghanem/archive/2008/09/06/comparing-listview-with-gridview-datalist-and-repeater.aspx`; Last visit: 2009-09-16.

[9] Guthrie, S. LINQ to SQL (Part 7 - Updating our Database using Stored Procedures). Web site. `http://weblogs.asp.net/scottgu/archive/2007/08/23/linq-to-sql-part-7-updating-our-database-using-stored-procedures.aspx`; Last visit: 2009-09-04.

[10] Guthrie, S. Using LINQ to SQL (Part 1) . Web site. `http://weblogs.asp.net/scottgu/archive/2007/05/19/using-linq-to-sql-part-1.aspx`; Last visit: 2009-08-24.

[11] Huemer, C., Kappel, G., and Vieweg, S. Migration in Object-oriented Database Systems-A Practical Approach. *Software-Practice and Experience 25(10)* (1995), 1065–1096.

[12] Krafzig, G., Banke, K., and Slama, D. *Enterprise SOA*. Prentice Hall, 2004.

[13] Liberty, J. *Programming C#*. O'Reilly, 2001.

[14] MacDonald, M., and Szpuszta, M. *Pro ASP.NET 3.5 in C#*. Apress, 2007.

[15] Marquerie, F., Eichert, S., and Wooley, J. *LINQ in Action*. Manning, 2008.

[16] MSDN authors. ASP.NET Page Life Cycle Overview. Web site. `http://msdn.microsoft.com/en-us/library/ms178472.aspx`; Last visit: 2009-09-16.

[17] MSDN authors. How to: Create LINQ to SQL Classes Mapped to Tables and Views (O/R Designer). Web site. `http://msdn.microsoft.com/en-us/library/bb384396.aspx`; Last visit: 2009-08-31.

[18] MSDN authors. Implementing Model-View-Controller in ASP.NET. Web site. `http://msdn.microsoft.com/en-us/library/ms998540.aspx`; Last visit: 2009-09-16.

[19] MSDN authors. Implementing Page Controller in ASP.NET. Web site. `http://msdn.microsoft.com/en-us/library/ms998548.aspx`; Last visit: 2009-09-16.

[20] MSDN authors. Introducing System Transactions. Web site. `http://msdn.microsoft.com/en-us/library/ms973865.aspx`; Last visit: 2009-09-04.

[21] MSDN authors. Object Identity (LINQ to SQL). Web site. `http://msdn.microsoft.com/en-us/library/bb399376.aspx`; Last visit: 2009-08-24.

[22] MSDN authors. Page Controller. Web site. `http://msdn.microsoft.com/en-us/library/ms978764.aspx`; Last visit: 2009-09-16.

[23] MSDN authors. The ADO.NET Entity Framework Overview. Web site. `http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx`; Last visit: 2009-09-04.

[24] Ormond, M. Dynamically Loading ListView Templates. Web site. `http://blogs.msdn.com/mikeormond/archive/2008/07/26/dynamically-loading-listview-templates.aspx`; Last visit: 2009-09-20.

[25] Rosenbauer. Rosenbauer Web site. Web site. `http://www.rosenbauer.com`; Last visit: 2009-07-21.

[26] Schwichtenberg, H. *ASP.NET 2.0 mit Visual C# 2005*. Microsoft Press, 2006.

[27] Schwichtenberg, H. *ASP.NET 3.5 mit Visual Basic 2008*. Microsoft Press, 2009.

[28] Sneed, A. Flexible Data Access With LINQ To SQL And The Entity Framework. Web site. `http://msdn.microsoft.com/en-us/library/bb384396.aspx`; Last visit: 2009-08-31.

[29] Stadler, B. CAN Bus Informationssystem für Feuerwehrfahrzeuge. Master's thesis, FH Oberösterreich, Automatisierungstechnik Wels, 2005.

[30] Stahl, R. ListView and DataPager in ASP.NET 3.5. Web site. `http://www.west-wind.com/WebLog/posts/127340.aspx`; Last visit: 2009-09-20.

[31] Sun Microsystems. Java BluePrints. Web site. `http://java.sun.com/blueprints`; Last visit: 2009-07-30.

[32] Sych, O. T4 Toolbox: LINQ to SQL schema generator. Web site. `http://www.olegsych.com/2009/05/t4-toolbox-linq-to-sql-schema-generator`; Last visit: 2009-08-24.

[33] Vieweg, S., Kappel, G., and Tjoa, A. Change Management in Object-Oriented Database Systems. Tech. rep., Institute of Computer Science. Department of Information Systems. University of Linz, Austria, 1994.

[34] Wikipedia authors. Business logic. Web site. `http://en.wikipedia.org/wiki/Business_logic`; Last visit: 2009-09-04.

[35] Wikipedia authors. Query by Example. Web site. `http://en.wikipedia.org/wiki/Query_by_Example`; Last visit: 2009-09-04.

[36] Zöchbauer, G. O/R-Mapping in .NET Das ADO.NET Entity Framework. Master's thesis, Fachhochschule Hagenberg, 2008.