Thesis for the degree of Master of Science

Dynamic Archive Management

a framework for dynamic database schemas

Michiel Overeem

INF/SCR-07-01

August, 2007

Supervisor	Prof. Dr. S. D. Swierstra	
Daily supervisors	Drs. Ing. P. van Diermen	DEVENT**it**
	Dr. B. J. Heeren	Universiteit Utrecht
	Drs. L. Herlaar	Universiteit Utrecht

Center for Software Technology,
Dept of Information and Computing Sciences,
Universiteit Utrecht,
Utrecht, The Netherlands.

DEVENT**it** - Developers & Inventors in IT,
Bunschoten, The Netherlands.

Universiteit Utrecht

DEVENT**it**
Developers & Inventors in IT

Abstract

Two important problems arise in the development of web-based archive management systems. The first is the tight coupling of the database schema with the application logic. Basic functions such as retrieve, update, create, and delete are implemented with knowledge of the database schema. This knowledge is often compiled into the application. Changing the schema entails changing the source code. The second problem is that the user interface is also coupled to the database schema at compile-time. Certain database fields have to be presented, so they become part of the user interface. Changing the database schema often entails that the user interface is changed.

DEVENT**it** has developed a framework that tries to solve these problems: the ADD.NET framework. The database schema is stored in a data model and this model is used to implement generic read, update, create, and delete functionality. However, this data model is still compile-time linked to the application. The application is thus static. Furthermore, the user interface is defined in a programming language, and thus also compile-time linked.

This thesis describes an extension of the ADD.NET framework: a *dynamic* ADD.NET framework. It no longer uses a compile-time linked data model, but loads the model from a database. A reload mechanism makes sure that the model is reloaded when changes are made. A template engine is built, which provides the means to use a dynamic user interface definition. Because of these extensions the application does not have to be recompiled when the database schema is modified.

An application that can react to changing database schemas is nice, but not very useful. Therefore a development environment is created: the *Dictionary Manager*. This application provides the means to update the database schema, while keeping the data model synchronized with the database. A version control system with support for branching and merging is proposed to experiment with alternative schemas.

Preface

Writing your thesis, doing research, and working on your own project is the last phase of the Master program. For me, it was obvious that it had to be practical. I like research, new techniques, and new possibilities, but only if they are applicable. My thesis project needed to be something practical, a real life situation. It had to be something that was intended to be used by a larger audience. And then this came along. Maybe not the newest techniques, or a brand new invention. But it is practical, it is a real life situation. I'm glad that it became my project...

This thesis is carried out at DEVENT**it** , for which I'm grateful to Peter van Diermen. Although the idea was yours, the project became mine :-). Thanks for the freedom to work on your idea. I really hope that we can bring it into production, building a great product.

Thanks also to Bastiaan Heeren and Lennart Herlaar who guided me on the 'academic' road. Not only during my thesis, but also during the other years (Lennart during the Bachelor years and Bastiaan during the Master). Thanks for your thoughts and ideas, focussing me on the optimal solution. Correcting my grammar must have been hard, but thanks for it!

Thanks to my parents for supporting me, for making it possible to say: "This thesis is made on a Mac!" ;-). And of course, Mirjam (my wife), thanks for supporting me, keeping me focussed and giving me a hard time whenever I had something else to do. Thanks!

Contents

Has everyone noticed that all the letters of the word *database* are typed with the left hand? Now the layout of the QWERTYUIOP typewriter keyboard was designed, among other things, to facilitate the even use of both hands. It follows, therefore, that writing about databases is not only unnatural, but a lot harder than it appears.

unknown

Chapter 1

Introduction

Many languages, techniques, and technologies such as *Cascading Style Sheets* (CSS), the *Extensible HyperText Markup Language* (XHTML), *Asynchronous JavaScript And XML* (AJAX), ASP.NET, and Java2EE have rapidly improved the process of developing web applications. The demand for web applications also increased rapidly because of the growing availability of fast Internet access. Web applications are evolving from simple, database driven websites to mission-critical web applications with a rich graphical user interface. However, along with problems that also arise in desktop application development, web applications are challenged with new problems and demands. Among those new problems are interface design issues and performance.

One of the problems that also arises in the development of desktop applications is software maintenance. The maintenance of software is an important aspect of software engineering. Already in 1983, estimates pointed out that 50% of all the development work was spent in adaption and maintenance of a system [Lientz, 1983; Roddick, 1995]. A significant amount of time is spent adapting an existing system to meet changing requirements or new insights. These changes are present in the business logic as well as in the data structures.

1.1 Changing database schemas

Many applications present data and allow the modification of this data, stored mostly in conventional relational databases. As the result of a change in the database schema, such as the removal of a column, the source code of an application should be modified or at least be reviewed. Every usage of the column should be removed, which can result in the removal of variables in a procedure which again has effect on other parts of the application. The knowledge of the database schema can be in embedded Structured Query Language (SQL) statements, but also in the surrounding code that uses the results of those SQL statements. Those embedded SQL statements can be rather simple: selection, for instance, is often done from one database table (`SELECT * FROM TABLE ...`). Advanced language constructs such as sub queries are seldom used. Although the statements might be simple, they have to be reviewed after a schema change. A sloppy written application might contain embedded SQL statements in almost every part, from the Graphical User Interface (GUI) layer to the data layer itself. This makes the adaption of an application a complicated process, that is likely to introduce new bugs.

1.2 User interface for web applications

As pointed out earlier, many traditional applications are moving towards a web-based variant. An example is the web-based e-mail application that Google provides as main interface to their e-mail service. The popularity of this service shows that people are willing to replace desktop applications with web applications. Another example are route planners. Route planners used to be stand-alone desktop applications, but nowadays many of them are available through the Internet. Disadvantages of web applications are often the responsiveness and the lack of a rich user interface. The responsiveness is influenced by the client (Internet browser) and server communication. The richness of a user interface is influenced by the techniques that can be used for developing web applications. For instance, drag-and-drop support is not a standard feature of web applications, because of the advanced techniques that are needed. However, the success of web-based applications shows that these disadvantages are considered as minor. Web applications also bring new features that were not possible with traditional desktop applications. There is no need for installation: the application just works on any modern computer (even mobile clients are often supported) with an Internet browser (unless plug-ins are needed of course). Software companies do not have to investigate how to release updates and inform their customers, because the application is deployed at one location.

Web applications challenge the software industry in new ways. Performance is one of those challenges, because the executable runs at one machine serving thousands of clients. Resources are shared among multiple clients. Some of the aspects that might influence the performance cannot be addressed by the developers, for instance, the bandwidth that clients use. Although not all variables can be influenced, it is important that the software stays usable. Another challenge that web applications bring is security. Clients can be geographically far away from the server communicating over (possible insecure) connections.

A less technical challenge that comes along with the moving towards web applications is the customization or *look and feel* of an application. Desktop applications have to worry about position and size of the different elements. The style and design of user interface elements, however, are mostly determined by the operating system. An application running in Microsoft Windows looks different than that same application running in Mac OS X. Application developers could, however, decide to design the whole interface themselves, and this is in fact not that uncommon. This results in an enormous amount of work for the developers. And this work might not be appreciated by the users, because the application does not integrate very well with the operating system.

Web applications on the other hand, do not have a standard look and feel. Non-styled web applications are presented in black and white, mixed with elements of the Internet browser and operating system. The markup language HTML has constructs to influence the style, the language CSS is created to further enhance this possibility. The lack of a standard design for web applications results in an extra design issue for web application designers with respect to desktop application designers. It could be that every web application is styled in conformance to the logo and style of the software company. However, many customers want the application to be styled according to their own company logo and style. This is not a problem if the developed software is custom-made. The software is only used by one company and can thus be styled to their demands. For commercial off-the-shelf software, this is a challenge. The software will be used by a number of different companies and they probably all want their own style and branding. A clear separation of logic and interface is thus needed. This increases the reuse of components between the different customer-specific versions.

1.3 Flexible document structures

Imagine an information management system, or better an archive management system for different types of documents. Every document has its own structure, which of course depends on the kind of information stored in such a document. An integral search engine, searching through the complete collection of documents, cannot depend on the specific document structure. If the knowledge of a specific structure is used, the engine becomes coupled to that structure. By supplying an index algorithm for each structure, the engine can search through the generic index, without direct knowledge of the specific document structure.

The document structures are represented in a database schema, every document type corresponds to a specific part of the complete schema. All document schemas connect to the generic part at a specific location. The search index is of course stored in the generic part. The knowledge of the database schema is coupled to the archive management system. The generic parts are used in, for instance, the search engine, the specific parts are used in the detail presentation of documents.

If customers want to change the document structures, the developers have to create a new version of the system. The structures are compiled into the application, because specific knowledge is used. The structures cannot be changed without changing the application, although only the parts that use the specific document structure have to be changed.

1.4 Research problem

The two identified and discussed problems, the tight coupling between database schema and application logic, and the tight coupling between database schema and interface, are both present in the development of web-based information management systems. Information management systems rely heavily on database access. Procedures and embedded SQL statements depend on the database schema, and are thus coupled to the schema. The web-based aspect makes it subject to the design problems described earlier. Often the information systems have a public side, which is accessible to anonymous visitors. As a result companies often want to design the public side in their own style and logo.

This thesis focuses on a special category of information systems, specifically information systems for archive management. Archives can contain information such as certificates of birth and death, newspapers, and historical photographs. These kind of systems typically provide functionality for searching, reporting, and maintaining the stored information. Section 1.3 sketches how the architecture of an flexible archive management system can look. However, the schema information is coupled to the application logic, and changes have to be done by the developers.

This thesis is twofold: it proposes (1) a framework to solve the two tight couplings and (2) presents a management application that allows users to modify the structure of the document types stored in the archive system. The research problem is formulated in the following questions:

1. How can a framework for *archive management systems* be constructed that removes the tight, compile-time coupling between
   a. database schema and application logic, and
   b. database schema and the user interface?
2. How can non-technical people modify a database schema used in an application built with the framework as described by 1?

The result of these questions is an archive management system that is document independent and features a development environment for constructing and maintaining document types.

The framework is implemented by extending an already existing framework. The following sections explain the context of this thesis: the company where the work is carried out and the framework that is extended. After that, the objectives are explained in more detail and an overview of the contents is given.

1.5 DEVENTit

This thesis is carried out at DEVENT**it** [DEVENTit], which is an independent software vendor. The software developed at DEVENT**it** can be categorized in three types of software: information management systems, financial software, and custom-made systems. The first type is also the context of this thesis.

A strong market position is obtained with the application platform *Atlantis*. This is a platform for several desktop applications that manage information of a particular structure such as newspapers. Also several web applications have been developed that provide the means to publish information on the Internet. The knowledge of building these applications and Atlantis is combined into a new product. This new system is the context of this thesis.

1.6 ADD.NET

In the past, the framework *ADD.NET*[1] [Kok, 2004] has been developed. This framework relieves the developers of writing code that provides the basic *create*, *retrieve*, *update*, and *delete* (CRUD) functionality. This framework is also used in constructing an archive management system.

The key data structure in this framework is the *dictionary*. The dictionary is a representation of the relational model, extended with information that is used in the interface such as help lines and error messages. The database schema of a particular application is stored in the dictionary and used in the framework as arguments of the CRUD functions.

It is important to note that the dictionary used by the framework is a data model. It is data, representing a specific databases schema in a database. To avoid confusion, the term dictionary is only used for this data. The term database schema (or only schema) is used to refer to the schema in a database (often called dictionary in database literature).

The dictionaries are maintained with another application: the ADDWin[2]. This application generates SQL statements for the creation of the database tables. Code and data files are generated from this application and loaded in the ADD.NET framework. These files contain the dictionary for a particular application. Advantage of the compile-time linking is the fact that every table and column name is available as a constant field. Instead of using literal strings, these constants can be used. This prevents spelling mistakes that are only found at run-time.

1.6.1 Example application

This section discusses an example application built using the framework. The web-based application is relatively small to keep the example concise. The application is used to store information about certificates. For instance, certificates of birth, but the exact kind of certificates does not matter for this example. The application presents the certificates and allows the modification of certificates.

[1] ADD.NET is pronounced *a d d dot net*. The *ADD* part is an acronym (which is intentionally left unspecified). The *.NET* part is a reference to the implementation in the Microsoft .NET framework [Microsoft NET].
[2] ADDWin is pronounced *a d d win*

Certificates have a date, description, and are related to multiple persons and roles. Figure 1.1 shows the schema for this information structure[3]. Every table has a column named **ID**, which acts as the primary key. Referencing columns are omitted, but can be deduced from the arrows.

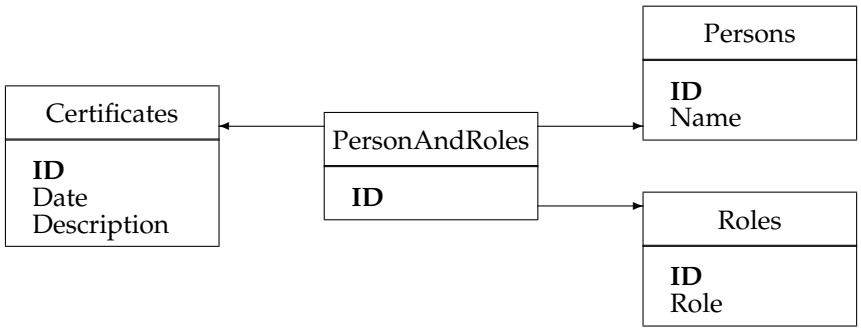

Figure 1.1: Database schema for certificates

A collection of certificates stored in the application is shown in Figure 1.2. This information can be presented in different ways: showing one certificate with all of its details or showing all certificates with a few details. The presentation itself might allow modification, but can also be read-only. Figure 1.3 shows two possible presentations. The first shows an overview of the certificates along with the persons and their roles. This presentation is read-only. The second variant shows an edit form for a single certificate. Although the row in the second presentation of Figure 1.3 is in *edit mode* and shows text fields, the user is not allowed to insert free text into these fields. Instead, when the focus is put on one of those text fields a selection list is presented with possible values from the corresponding table. This is the standard way of editing foreign key values in the ADD.NET framework.

Certificates

ID	Date	Description
1	01-01-1900	Certificate of birth for J. Doe
2	14-06-1900	Certificate of birth for K. Doe

Persons

ID	Name
1	C. Doe
2	K. Doe
3	P. Doe

PersonAndRoles

ID	Certificates_ID	Persons_ID	Roles_ID
1	1	2	3
2	1	3	1
3	2	1	2

Roles

ID	Role
1	Father
2	Daughter
3	Spouse

Figure 1.2: Example certificates data

Note that in this sample application the developer does not need to write queries, bind query results to user interface controls, or write code to react on certain user interface events. However, the developer still has to define the user interfaces, and create event handlers for the different actions such as *save*, *new*, and *delete*. The framework does take care of the validation of values and linking referencing rows together.

[3] The arrows represent foreign keys. Arrows point from the referencing table to the referenced table.

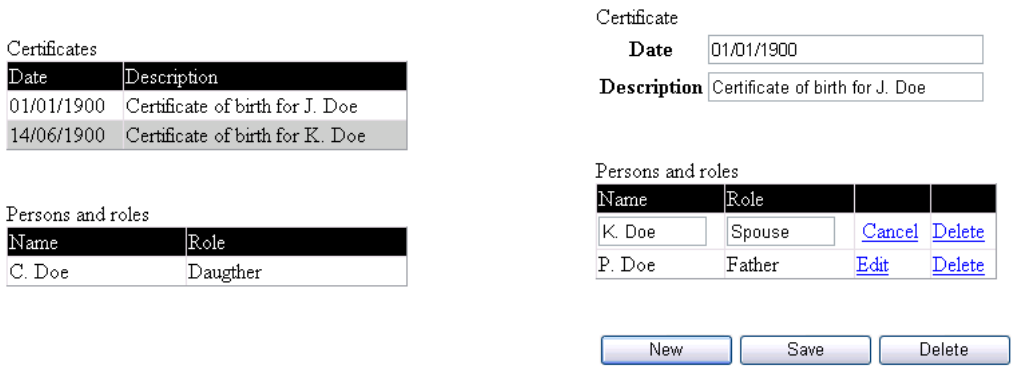

Figure 1.3: Two possible presentations for certificates

1.7 Objectives: *dynamic* ADD.NET and *Dictionary Manager*

The previous sections describe the ADD.NET framework and show an example application. This thesis extends ADD.NET and creates a *dynamic* ADD.NET framework. The extension focusses on the outer parts: the dictionary and the user interface. The inner parts of ADD.NET are taken for granted and are only discussed if they influence the new work. Among these inner parts are the parts that handle the selection, the validation, and the synchronization to the database of data. The *dynamic* ADD.NET framework solves a number of disadvantages of the ADD.NET framework.

The first disadvantage of the ADD.NET framework that is addressed, is the compile-time linking of dictionary information. The compile-time linking couples the dictionary to the application. This prevents modifications of the dictionary without recompiling the complete application. A change to the dictionary implies a new version of the application, along with the steps to deploy this new version. These steps can be prevented if the dictionary would remain mutable after deployment.

Another disadvantage in the ADD.NET framework can be found in another part: the user interface. Currently, the interfaces are defined in source code and compiled into the application. This prevents a simple modification such as the replacement of an input field without recompiling the application. The current framework does not provide a template engine or a similar solution to separate the user interface from the compiled application. The user interface is defined in the programming language itself, and it is thus not possible to modify it without recompiling.

The last extension is not a disadvantage of ADD.NET but a completely new component: the *Dictionary Manager*. Because the dictionary is compiled into the application, it is read-only to the user and the deployed application. Users cannot modify the dictionary or create their own dictionary. The dictionary is part of the source code and is hidden from the user. However, the user is in theory perfectly capable of changing some of the information such as help lines and error messages. There is also a strong believe that the user is capable of changing the actual database schema. If the dictionary would be mutable after deployment, users become capable of adding fields, removing tables, and so on. This would make an archive management system very flexible. Every customer can have its own customized dictionary version and thus its own customized application.

1.8 Structure of this thesis

The remains of the thesis have the following structure. Chapter 2 gives an informal description of the functionality through a number of use cases. After that the run-time aspect of the framework is discussed. Chapter 3 discusses the run-time loading of a dynamic dictionary, while Chapter 4 describes solutions for a flexible definition of user interfaces. After these chapters, the extension of the ADD.NET framework is ready.

The following chapters discuss the management application of the dictionary, or the dictionary development environment. Chapter 5 continues with the core of the management application, it provides the means for editing and maintaining the dictionary. It also discusses how the dictionary and the database stay synchronized. The manager is extended in Chapter 6 with a version control system for dictionaries. Chapter 7 discusses the merging of dictionaries. This is used inside the version control system and while updating a *dynamic* ADD.NET application. Finally, Chapter 8 summarizes the thesis, draws conclusions, and explains the possible future work.

Chapter 2

Use cases

This chapter gives an informal description of common tasks that the *dynamic* ADD.NET framework and the *Dictionary Manager* should support. Among those tasks are modifying the information structure and user interface. All details that do not apply to the work of this thesis are omitted. The task descriptions are based on a fictional application that is used in a fictional organization. This application uses the *dynamic* ADD.NET framework. All the use cases describe situations in the *future*, using the work described in the thesis.

The functionality that is described covers the *Dictionary Manager* (Chapter 5) along with the version control system (Chapter 6) and automatic merge (Chapter 7). Furthermore, the application uses the extensions described in Chapter 3 and Chapter 4.

Throughout these use cases and the following chapters different groups of people with different roles are used. These groups are:

- **Developers** These are the software engineers that work at DEVENT**it** . They develop the applications, maintain the database schemas in the old situation, and maintain the new components.
- **Customers** They buy software from DEVENT**it** . This can be an organization (like a city archive) or a private customer, although this is a rare situation for archive management software. Customers are not always users, but they could of course. The customers have access to the installed program: the executable files and the database.
- **Users** They use the application built with the *dynamic* ADD.NET framework. They search documents, and modify them. They can also use the *Dictionary Manager*, although sometimes this special group of users is refered to as **Administrators**. Normal users might not be able to use the *Dictionary Manager*, because they are not authorized. Users do not have access to the installed program nor can they directly connect to the database.

2.1 Context

A small city archive decides to digitalize their certificates of birth[1] and publish them on the Internet. They decide to buy software from DEVENT**it** .

The archive management application is deployed and taken into production. The certificates consist of a date and a description. As said, the organization is a small city archive, with a large

[1] These are not the same certificates as used in the example of the previous chapter.

group of volunteers. A number of these volunteers have filled the database and the application is currently used to publish the certificates over the Internet (the *users*). Another group of volunteers administrate the system (the *administrators*).

The described situations are chronological in time.

2.2 Addition of a field

A number of small surrounding villages have asked to integrate their certificates in the current database. The organization has agreed, but also signaled the need for location information of certificates. Without this information, visitors are unable to tell to which village a particular certificate belongs. The location field is decided to be a free text field, without any format requirements. The field is optional.

One of the administrators first logs on to the system. He (or she) edits the current information structure and adds a text field named *location*. The default value for all existing certificates is set to the organizations location. After the modification of the structure, the administrator edits the interface definitions by adding the location field. The new field is now visible in the detail information of the certificates.

All existing certificates show the name of the organizations city. The volunteers of surrounding villages start digitalizing their certificates with the correct location.

2.3 Addition of a relation, removal of a field

The database with certificates has grown in the past time, but the organization has signaled a number of spelling variations in the location information. Some volunteers entered the location with a starting capital letter. Others use an alternative name or spelling. These variations make searching for certificates from a certain location hard. Therefore, the organization decides to remove the location as a free text field. Instead, the location is put inside a separate database table and a relation will be created between the certificates and the location table.

After some meetings, it becomes clear that the modification should happen in two phases. First the new location table along with the relation will be created. The volunteers will then use the old location field to fill the new relation. After that, the old location is redundant and will be deleted.

To begin the first phase, an administrator logs on to the system. The new location table is created and a relation between the certificates and the locations is created. This relation expresses the fact that every certificate has one location, and every location can be related to multiple certificates. After this modification, the interface needs to be updated. The administrator therefore adds an interface to modify the locations. The interface for certificates will also be modified. The old location field is now read-only and the new location field is added.

After these modifications, the volunteers start to update the existing certificates. The old location information is used to update the new location information. After some time, the organization signals that all certificates are updated and they inform the administrators. One of the administrators logs on to the system and removes the old location field. He also removes the old location field from the interfaces.

2.4 New information structure

Some time has passed again and all certificates are digitalized. However, the organization has found a large collection of old photos and decided to digitalize them along with descriptions. The initial information structure consists of a location, a period in time, and a brief description of the content on the photos.

After a lot of meetings and discussions, the information structure and interface are described. After that, one of the administrators logs on and creates the new structure. Two database tables are created, the first is a photos table with columns for begin and end date (to express a certain period, it might be impossible to exactly date a photo), and description. The second table is for the location and has only one column. The two tables are related: every photo has multiple locations, and a location can be related to multiple photos. After creating the information structure, the interfaces are defined. Again an administrator will log on to the system. He can then select the fields that should be shown in the interface and the system will generate the basic screen definitions. These definitions are then modified: the look and feel should match the already existing applications interface. The organization agrees with the new structure and interface. The photos are distributed among the volunteers and they can start the digitalization.

2.5 Modification of the user interface

The city archive has received a small fund and they decide to create a new logo and style. Therefore an advertising agency is contracted. After some time, the agency presents the new logo and style. The archive is very pleased with the new style and decides to restyle the existing applications.

The agency is contracted again and the organization hands over the existing interface definitions of the archive application along with some documentation. The agency creates new definitions that follow the new style. New definitions have to be created, because the structure of the pages is also changed, using CSS is not enough in such cases. After a while, the new definitions are ready. The administrators install the new definitions and the applications uses the new definitions.

2.6 Viewing change history

The application has been used for a number of months, and the information structures have evolved along the way. However, at some point in time the help text for the dates of the photos have been changed. The organization, however, cannot remember why and when this has been changed. Therefore, a request for this information is sent to the administrators.

One of the administrators again logs on to the system and opens the *Dictionary Manager*. The version control system allows him to browse to the change history of the photos information structure. By viewing the differences between two versions, it is possible to locate the version that introduced the new help text. It is not possible to search for a particular change, so these differences have to be reviewed manually. After some time, the version is found. Along with every change a small description and a date is recorded. He writes the information down and reports it back to the organization management.

Note that there is some indirection here, because the members of the organization board do not know how to use the system. The administrators know how to use the Dictionary Manager*, and can thus easily find requested information. Theoretically the members of the board can do it themselves, but often a small group of administrators that have good knowledge of the system handle these kind of tasks.*

2.7 Update of the application

A year is passed since the application was first deployed and a new version is released. Among the changes in this new version are changes in the certificates structure. More customers have signaled the need for a location field, this field is now part of the standard release.

The organization has reviewed the changes in this new version and they want to upgrade their system. However, they do not want to lose their own changes. The request for an upgrade is sent to the administrators, along with the request to preserve any changes. Luckily, the system preserves as much data as possible without any extra work.

The administrators start the upgrade process after carefully reading all the release information. During the upgrade process the current information structure of the organization and the information structure as released along with the application are merged into one single new structure. Conflicts are solved automatic by the application, an overview of the changes is presented.

2.8 Summary

The use cases sketch a rough functional design of the *dynamic* ADD.NET framework. They point out how the framework should be used and how it can be used. The remaining chapters describe all the details of the framework and the management application.

Chapter 3

Dynamic dictionary

In this chapter the existing ADD.NET framework is used to build the first part of the *dynamic* ADD.NET framework, which uses a mutable dictionary. The component that loads the dictionary is rewritten such that it no longer loads the dictionary from generated files, but from a database. This extension of the existing framework, should fulfill two quality requirements. The first one is that the performance of the framework should not be a bottleneck that obstructs the functionality of the application that uses the framework. The second requirement is that the dynamic dictionary should not result in errors at runtime, because of outdated data or other problems.

3.1 Introduction

The ADD.NET framework already provides the basic CRUD functionality. However, the information used to generate the correct SQL statements comes from generated files. They are only loaded once at the start of the application. The information stored in these files is immutable.

The compile-time linking of all this information is a big disadvantage of ADD.NET. Even small modifications, such as the modification of an error message, require a new development cycle: code generation, compilation, test runs, and deployment. If this compile-time linking is changed to a run-time loading of information, the framework becomes dynamic and much more flexible. It could adapt to an evolving database schema without new development cycles.

To turn the ADD.NET into a *dynamic* ADD.NET the information is no longer stored in files, but in a database. The framework could use mutable files, but using a database is chosen instead. The advantage of a database is that the dictionary can be loaded and updated partially. This is a lot harder when a file is used as storage. The database also offers features such as referential integrity: it checks that every relation between, for instance, columns and tables is correct. Another plus is the fact that bootstrapping is possible: maintaining the *dynamic* ADD.NET schema in a *dynamic* ADD.NET application.

3.1.1 Dictionary

The information that is used by the ADD.NET is stored in the dictionary. This dictionary contains every piece of information that is needed for the CRUD functionality and is the center of the framework. Basically, the dictionary is a representation of the relational model [Codd, 1970; Ramakrishnan and Gehrke, 2000]. Concepts used in the dictionary are tables, columns, and foreign keys. The columns are extended with information that is used in the interface such as help

lines and error messages. Furthermore, the types of the columns are extended with domain information which is used in the validation process. As already pointed out in Section 1.6, do not confuse the dictionary used in the framework with the dictionary of a database. The dictionary as referred to throughout this thesis is the data used in the framework. The dictionary of the database is referred to as the database schema.

The ADDWin uses a relational database as its storage for the dictionaries. The database schema used is shown in Figure 3.1. The column names printed in bold together make up the primary key of a table. The arrows represent foreign keys, pointing at the referenced table. The column Type of the table Columns contains an identifier of a supported type such as *string*, *integer*, *boolean*, *date*, or *time*.

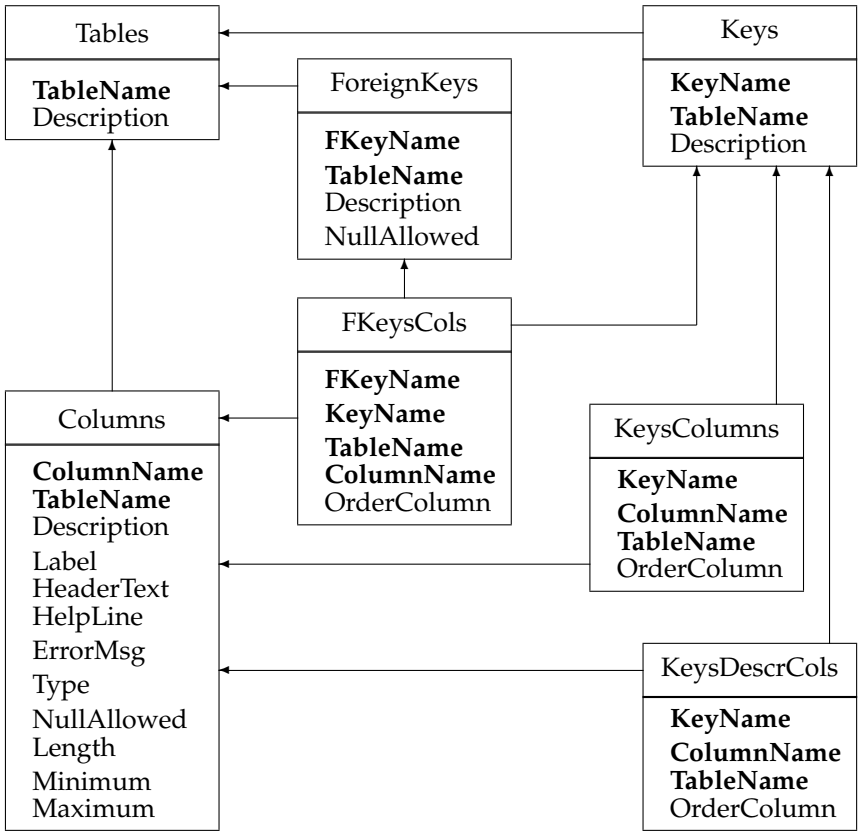

Figure 3.1: Dictionary database schema as used by ADDWin

The database schema used by ADDWin can be used in the *dynamic* ADD.NET framework, with some minor modifications that are left unspecified (these are necessary for the implementation). An extra advantage is the easy conversion of existing ADD.NET applications to *dynamic* ADD.NET applications. Existing dictionaries can easily be copied from the ADDWin database to a database used in the *dynamic* ADD.NET framework.

A constraint is applied to the dictionary which rules out an exceptional case to prevent difficulties. This constraint tells that between every two tables there can be a maximum of one relation. A second relation makes the schema more complex, but also leads to situations that require more knowledge about the relational model and database systems. This kind of situation does not occur often: preventing it is not a huge limitation of the system. However, it is possible in the

relational model. Therefore the framework has to deal with it or explicitly prevent it.

The dictionary is loaded into a data model. This model is accessible from every part of the application. How this model is loaded, however, is not important and can thus be easily changed.

ADD.NET reads the dictionary from static, generated files. They are read only once, because they are not supposed to change during the application's lifetime. In the *dynamic* ADD.NET framework, this is no longer the case. The dictionary is meant to change and the framework should reload the information after such a change. If the framework does not reload, or it reloads not fast enough, it will no longer be correct and the application may crash. It is thus very important that at any moment in time, the dictionary information loaded is consistent with the database schema.

3.2 Run-time loading of a dictionary

The dictionary stored in the database can be trusted by the *dynamic* ADD.NET application. The framework does not have to validate the dictionary information in any way. As a result, there is no need for a rollback mechanism of the dictionary that is loaded into memory. However, the framework is responsible for (re)loading the dictionary when necessary. As said earlier, the application may crash when the dictionary information is outdated.

To ensure that the correct dictionary information is used at all times, the framework can use two solutions. The first solution is a *no cache* solution: the data model does not have any state, but instead it executes a query to retrieve information from the database. The data model is a proxy between the framework and the database. This ensures that the application always uses the latest available information. The second solution is a *cache* solution: the application does cache the dictionary information, but makes sure that the information never becomes outdated. Both solution are discussed and evaluated.

3.2.1 No cache: a proxy approach

The *no cache* solution is guaranteed to always use the correct information. The framework can trust the information in the database, and because the data model is only a proxy between the framework and the database, the information returned by the data model is always correct.

However, the data model contains all the information from the dictionary. This information is the center of the framework, it depends heavily on this information. The database access increases rapidly if the data model is only a proxy. Not only does the database load increases, the data model might react slower because of the proxy mechanism. Retrieving something from a database is slower then retrieving something that is already loaded into memory. This might influence the performance of a *dynamic* ADD.NET application.

Therefore, the *no cache* solution is tested with a small example application. The example application is the certificate management application from Chapter 1, a small test environment for this extension. This application is analyzed and the results are discussed.

Two possible observation methods can be used: an *execution time* approach and a *quantitative* approach. The first approach measures the execution time of the framework and the time spent on retrieving dictionary information. These timings can then be related to each other and observations can be made about the correlation between those timings. This approach has a lot of variables that influence the results: the performance of the application platform, the performance of the machine that runs the database, and the speed of the communication between the two machines. These variables make this approach unreliable and unusable. The second approach is quantitative: it counts the number of executions and relates that number to other characteristics

of the application. For example, these characteristics can be the dictionary size and interface composition. The correlation between the two tells something about the scalability of the solution. The last approach is used to observe the no cache implementation.

The application that is observed is small: the database schema consists of four tables, a total of eleven columns, and three relations. Three different webpages are used. The first webpage shows two columns from one table. The second webpage shows three column from two tables. The last page shows five columns from all four tables.

Figure 3.2 shows the statistics collected from the *dynamic* ADD.NET framework using a no cache implementation. The second, third, and fourth column tell something about the screen composition: how many columns are used, how many tables are involved, and how many relations are there between those tables. The fifth, sixth, and seventh column show the number of queries, the number of distinct queries, and the average number of times a query is executed (the number of queries divided by the distinct number of queries). The first observation is that the amount of queries looks huge with respect to the dictionary size or the screen definition size. The number of queries increases dramatically with the increasing complexity of the screen definition.

Screen	Tables	Columns	Relations	Queries	Distinct	Average
1	1	2	0	145	20	7.25
2	2	3	1	791	45	17.58
3	4	5	3	2357	73	32.29

Figure 3.2: Query statistics from the no cache dictionary implementation

A closer look at the statistics shows that it is not only the amount of distinct queries that increases, but the queries are also executed more often. The cause of this behavior can be found in the ADD.NET framework implementation. The source is an algorithm that determines the paths between tables. This analysis heavily uses the dictionary data model: it travels from table to table by examining the foreign keys and following the references. It is clear that the no cache implementation does not work very well with the data model used in-memory by the ADD.NET framework as shown in Figure 3.3. The model is designed to be as connected as possible. However, in the no cache implementation, every link means a new query. If the primary key of a foreign key is retrieved, a query is executed. When the columns of a table are used, a query is executed. This explains the large amount of queries. The no cache implementation should use a different data model that fits better, but that requires a large amount of work: the ADD.NET framework has to be rewritten in order to use the new data model.

Another observation about the test results can be made by looking at the kind of statements. Figure 3.4 shows the statistics for this observation. Two kind of statements are executed by the framework: single value and multiple value selections. The first one retrieves a single value such as a maximum value, a help line for a specific column, or the number of columns of a table. The second kind of statement uses more resources: the result can consist of multiple rows and multiple columns. This means that those rows have to be fetched by the framework. Figure 3.4 shows the division on statement kind. The statements that return multiple values are executed far more often than the statements that return a single value. This has to do with the *HashTable* objects returned by the *Dictionary* object. These *HashTables* are often used to retrieve a particular *Table* object given a table name. However, the *HashTables* are rebuilt every time they are needed, because of the proxy implementation. This means that the resource usage increases fast: the number of multiple value statements increases faster than the number of single value statements.

Keep in mind that these statistics are from the initial load of one screen only, by one user! In a production environment, multiple users are concurrently working. Multiple pages are used, and often the pages are more complex than the one from the example.

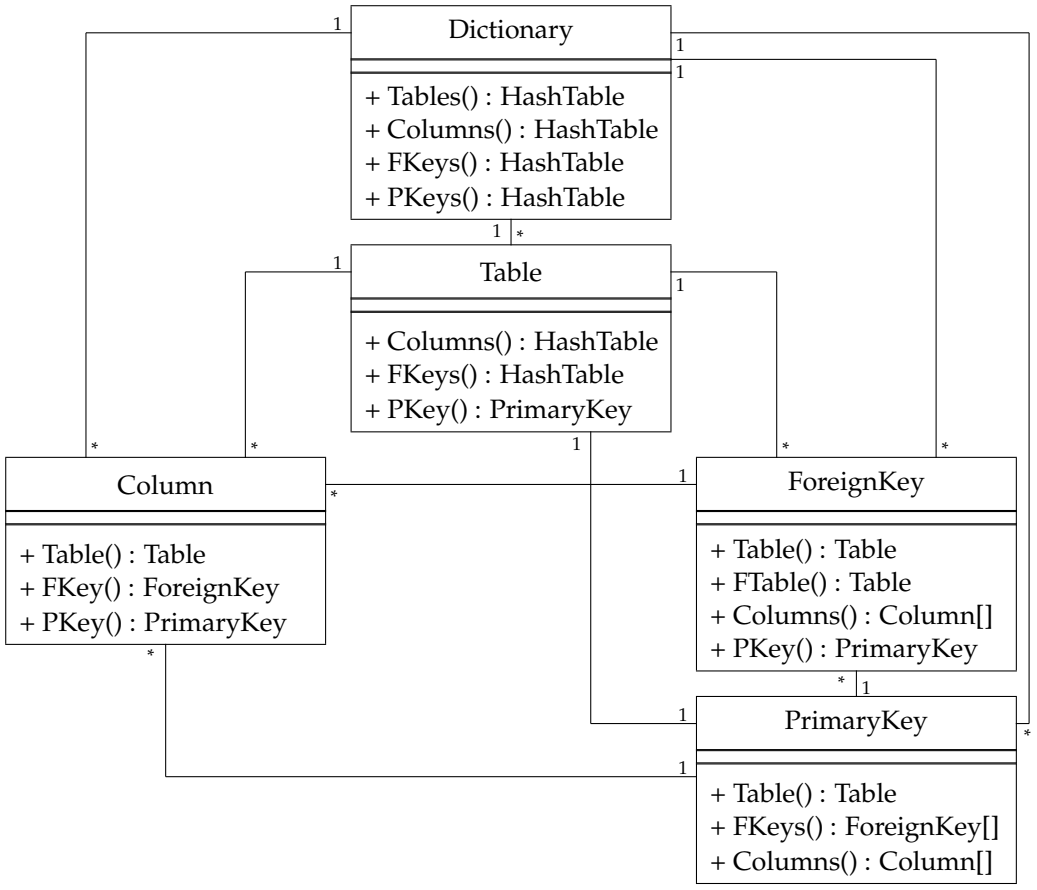

Figure 3.3: Dictionary data model (UML)

The statistics signal an increase of the database load. This increased load influences the performance of the database which in turn influences the performance of an application. It is reasonable to say that this proxy approach influences the database load in such a way that it negatively influences the performance of the applications!

3.2.2 Cache: controlling the database access

Another solution to deal with the dynamic dictionary is to use some form of caching. The big advantage of using caching over the *no cache* solution is the decrease in database load. As a result, the database performance does not influence the application performance anymore. Because the data model now caches the dictionary information, retrieving the dictionary information no longer depends on database access. The information is only retrieved once in a certain interval, and not on every request.

The *dynamic* ADD.NET framework can use caching in different ways. The first solution loads the dictionary at the start of an application. The framework is notified when the dictionary is changed (this is a *push* approach). The dictionary cache is then invalidated and the data model reloaded. Although this solution is straightforward, there are some drawbacks to this approach. The framework for one is no longer responsible for the reloading of the dictionary, but the appli-

Screen	Tables	Columns	Relations	Single value	Multi value
1	1	2	0	25	120
2	2	3	1	59	732
3	4	5	3	98	2259

Figure 3.4: Query type statistics from the no-cache dictionary implementation

cation (or person) that changes the dictionary. Whenever the dictionary is changed, a notification should be sent. Such a notification is easily forgotten, but the consequences are severe. The application uses an incorrect dictionary, and cannot function anymore. Problems stay, even when the notification is not forgotten: the notifier can crash just before the notification is sent. This leaves the framework in an incorrect state.

A second possibility is the checking for updates on a regular interval, a *pulling* approach. This approach makes the framework responsible for the updating of the dictionary cache. It no longer depends on a notifier. However, deciding on the interval between checks is not trivial. The dictionary might change every day. If the interval in that case is set to one day, the application could use incorrect information for a day, in a worst case scenario. The dictionary might be changed at 14:00 hour, but the reloading does not happen until 13:59 hour: the application uses inconsistent information and cannot operate for the next 24 hours (minus a minute). On the other hand, setting the interval to one minute (or even less) could introduce a performance problem.

The last possibility is to reload the dictionary when a generated statement fails. The framework monitors the executed statements and if a statements fails, it might be that the dictionary information is wrong. This approach leads to an unstable application and is therefore discarded.

3.2.3 No cache versus cache

The first solution, *no cache*, has the advantage of always being right. Because the database is to be trusted, the information is always correct. However, the big disadvantage of this solution is that it increases the load on the database. Modern DataBase Management Systems (DBMSs) are fast and perform well, however, they can become a bottleneck. The *dynamic* ADD.NET framework is the basis of an archive management system. The core task of such a system is searching and maintaining documents. This already puts a high load on the database. If the framework would increase this load beyond reasonable limits, it would conflict with the core functionality of the archive management system. Another disadvantage of this solution is the dependency on the database performance. If this performance decreases, the performance of the *dynamic* ADD.NET application will decrease too.

The first solution faces another problem: concurrent reading and updating of dictionary information. The first solution always reads the dictionary partially, small pieces are retrieved when needed. If a collection of read operations are interleaved with an update of the dictionary, the resulting information used in the application is inconsistent: it is a combination of information before the update and after the update. A solution to this problem involves a less direct proxy implementation, and instead retrieves the dictionary as a whole. Before every page load, the dictionary is retrieved and used. It does not involves caching, but it can make sure that the retrieval does not interfere with an update of the dictionary. Section 5.3 discusses concurrency problems faced while modifying the dictionary in more detail.

The second solution, *caching*, has the possibility that outdated information is used: if the reloading does not happen fast enough, the framework uses an incorrect dictionary. However, the load on the database can be balanced in this solution. Choosing the right interval decreases the proba-

bility that the dictionary information is outdated. It can be decreased even further by combining the two cache solutions: use an explicit notification and use checking on a regular interval as a fallback.

The interval setting allows a flexible use of the periodical checking. With this setting, the load on the database can be balanced. An application with a stable dictionary might do with a large interval, whereas an unstable, experimental dictionary might use a small interval. The notification system allows applications to make sure that they always have the latest version, but because the framework checks for updates, it can catch any problems in the notification system.

3.3 Summary

The *dynamic* ADD.NET framework is a modest extension of the ADD.NET framework: it reads the dictionary information from another source, a database. Because of this extension, the framework is already more dynamic, it can react on changing dictionaries. How these changes are made is discussed further one. Second, the user interface that uses this framework is still defined in a programming language, and cannot react on a changing dictionary. The next chapter discusses this problem.

Chapter 4

User interface

The GUI of an applications is important: it exposes features to the users of an application. If these features are hard to use, then the application becomes hard to use, and it does not matter how well these features are implemented. Designing GUIs of web applications give rise to new aspects, as explained in Chapter 1. Designing them is harder, because a default (or inherited) design does not exist. The interface is not as rich as the interface of desktop applications: the Internet browser environment limits the possibilities. There is also a demand for customization. Often customers want to use a web application in their own style, this becomes more important if the application has an interface for anonymous visitors.

The ADD.NET framework only uses custom-made webpages. They are built in a programming language and are not mutable by customers. This limits the flexibility of the application and it also causes problems when customized webpages are necessary for different customers.

This chapter describes the extension of the ADD.NET framework with a user interface that is not compile-time linked. These interfaces are used to edit the data that is described by the dynamic dictionary, as described in Chapter 3. The interface uses information from the dictionary such as help lines and error messages.

4.1 Introduction

In the ADD.NET framework every webpage that is used to edit data is custom-made. Although the framework and its algorithms are generic, every webpage is tightly coupled to the dictionary. This is a result of the fact that every control[1] is bound to a specific field. If two customers want the exact same webpage in a different layout, two different files are created. Of course much of those files can be shared, but those two files have to be maintained. This gives a huge administrative task, because different files have to be maintained although they perform the same task. Every layout change has to be done by the developers. This limits the flexibility of the application for customers and causes an increased work load for the developers.

The dynamic dictionary as described in Chapter 3 is of little use when the webpages to edit the data are still defined at compile-time. The interface cannot react to changes in the dictionary, so the source code has to change and this of course leads to a new version. A dynamic graphical user interface is necessary to make the dynamic dictionary useful.

There are many different solutions and techniques to create a flexible interface system for webpages. Template engines such as Mawl [Atkins et al., 1999] and Velocity [The Apache Velocity

[1] A control is a single user interface component such as a text field, a button, or an image.

Project] try to separate the business logic from the GUI by using a custom template language. The language acts as the glue by allowing plain HTML, CSS and special markers to variables (and sometimes even functions). These engines are very sophisticated and complex, because they are meant as a general purpose engine.

Another possible solution to separate presentation from logic is the language XForms [The Forms Working Groups]. This W3C recommendation aims at replacing the current HTML forms with a more powerful alternative. It decouples the interface from the application.

This chapter analyzes both approaches: the XForms standard and a custom-made template engine.

4.2 XForms

Interaction on webpages is done with forms. A webpage developer creates a form, adds controls such as text fields and check boxes, and adds a submit button. The submit button causes all input to be submitted to a URL by integrating the input with the HTTP request. The input values are submitted as a list of name-value pairs without structure. Although these forms are widely used by all web applications, they do have a number of drawbacks. First of all, the controls are aimed at presentation. A developer has to choose a type of control such as a plain text field, a list of choices, or a password field. This type is independent of the model that the user has in mind. The second drawback is the type-ignorance of the controls. For instance, to edit a number one has to use a plain text field and validate the input with client script or server code (often both). The third drawback is the binding of data to the controls which has to be written by hand. The application generates the HTML form with the values from the database on every load. Once new data is submitted the application should analyze, validate, and process the submitted data.

XForms is a new, XML-based, forms language that promises to solve all the drawbacks of plain HTML forms. The language combines XPath and XML schema to offer a clear separation of data and presentation. An XForms definition consists of a *model* and a *form*. The model describes the structure of the data, in the form of an XML schema. Fields can be constrained with aspects such as types and quantities. The model can also supply initial data. How this data is submitted is also described by this model. The form is the actual interface definition: it defines which controls to use. The controls can be placed inside the webpage by interleaving them with normal HTML, CSS, and JavaScript. This way the appearance of the controls can be influenced. The glue between the controls and the model are XPath expressions that bind specific data to a control.

An XForms definition is processed by a processor (such as an Internet browser) and presented in a platform specific manner. Controls might look different across different platforms, such as browsers or mobile devices. On a browser, the XForms definition looks like a normal HTML form. The XForms processor makes sure that the data is only submitted when it is valid according to the XML schema. There are no round-trips to the server necessary to validate data, this is all done in the XForms processor. This saves in the bandwidth of course, and gives a smooth interface. The server receives the submitted data in a structured way, as depicted by the model. The minimization of round-trips gives an AJAX-feel[2] to the webpage without extra work, which clearly is an advantage.

The solution provided with XForms gives a clean separation between the server (the web application) and the user interface. The webpage using XForms lives inside the XForms processor and the complete editing cycle is done inside the processor. This makes it easy to reuse forms or to change the web application at some point.

To integrate XForms with the *dynamic* ADD.NET framework, a few things have to be done. First,

[2] A webpage using AJAX updates the page partially, retrieving only that information necessary to respond to the user. This minimizes the 'flashing' of webpages and gives a smooth interface.

there should be an XML schema that describes a particular information structure. This schema can be deduced from the dictionary. Furthermore, server components that provide initial data (the current document for instance) and receive modified data should be available.

The XForms solution offers a clear separation of GUI and application logic. Because of the separate processor, no parsing has to be done. The users can create an XForms definition and direct it to the right URL. However, an enormous drawback is the complexity of XForms. It is not aimed at non-technical users and it is not tailored for the needs of the *dynamic* ADD.NET framework. The creation of an XForms definition is far from simple. It requires a substantial knowledge of XForms, XPath, and JavaScript. This knowledge is not only required for creating visual appealing effects and other extras, but the knowledge is also necessary for the core of the webpage: editing the data. Another problem is the clear mismatch between XML and relational data. In XML elements are nested to indicate relations. However, sometimes relational data cannot be converted to a fully nested XML document. Instead some attributes reference other elements in the document. One has to follow these references to find related elements. This is not a straightforward task and requires substantial knowledge of XML, XML schema, and XPath.

Although the XForms language is not new, XForms processors are not widely available. All the current popular Internet browsers require plug-ins and extensions. This makes the language less usable, because it requires a non-standard Internet browser environment. Fortunately, a 100% pure JavaScript solution [FormFaces] exists. It is not as fast as a processor built into the browser, but it removes the requirement of external tools.

4.3 Template engine

The template approach leaves the path of using standard specifications. However, this also means total freedom of implementing required features. The approach itself is straightforward: normal HTML pages are extended with custom placeholders. Every placeholder is translated to a specific control. It is matched to a field (or group of fields) from the dictionary and rendered in the appropriate way.

The template definitions are limited to pure XML. Although parsers do exist for HTML, XML gives the advantage of being better structured and more standardized. HTML and XML look very similar, but are not the same. HTML is a less structured language, allowing more variations in syntax. XML on the other hand is more strict. Better support exists because of this. Everything around the placeholders remains untouched and is directly written to the output. A template can contain XHTML, CSS, and JavaScript along with the framework placeholders. The placeholders are replaced with the components already available in the ADD.NET framework. This replacement is done before data is loaded or other events occur. The framework does not see a difference between the normal applications and the templates, and is thus reused without modification.

A template can use two kind of placeholders: fields and tabular grouping of fields. The only required attribute of a field is an id. This id is a field from the dictionary. This way a control is bound to a field. There are more things that can be influenced. For instance, the type of the field. By default every field is an input field. By supplying a different field type, a different aspect of the field can be presented: aspects such as descriptions, help messages, and error messages. The tabular grouping of fields is a grouping element with a number of children: field elements.

Besides the placeholders, the template engine defines one attribute that can be used on normal HTML elements: the *action* attribute. This attribute can be added to elements such as hyperlinks, buttons, and images. The system will then add a piece of JavaScript that will cause an event to fire. The specified action will then be executed. This way the CRUD actions can be linked to specific user interface events. Appendix A gives a more detailed overview of the template system.

The template approach is tailored to the requirements of the framework. This means that superfluous features can be left out. The template language can be fitted for the non-technical users. The mix of XHTML, CSS, and JavaScript with custom placeholders makes it possible to use existing webpage editors. Users can create the webpage in their favorite editor and save it as a normal webpage. After that, they can open the template in a text editor and insert the placeholders. They have to make sure that the webpage is valid XHTML, but a lot of editors support this.

The end-user gains a lot with the template approach. They can use a standard webpage editor and add the placeholders after the styling of the page. The XML format of the templates will add a little more complexity, but modern editors are capable of creating XHTML pages so it should not be too much of a burden. The system tries to use as much default values as possible, giving a good result without too much work. The use of defaults should of course not interfere with the flexibility of the system.

4.4 Conclusion

Although the XForms standard looks appealing, the complexity and lack of built-in support makes it unfeasible. The non-technical background of users steers the dynamic graphical user interface towards a non-standard custom-made solution. A custom-made solution makes it possible to adapt the solution to the needs of the users. Complicated aspects can then be left out, or can be abstracted over. Standard solutions often have a lot of configuration, in custom-made solutions this configuration can be captured in conventions, making it easier for the users. These aspects give a custom-made solution a better focus on specifying *what* and *where*, without the *how*.

The template approach also offers more existing code reuse. The ADD.NET framework has a large control collection, built to specific needs. These controls cannot be reused in the XForms solution. The template approach, however, can use them. The template engine is built on top of the existing framework: the framework does not know that there is a template or even a template engine. This means that existing webpages can be used in combination with the new templates.

4.5 Building on top of the templates

The template engine is a flexible approach. However, it requires users to create XML files. It was already discussed that users can work with existing editors to create a webpage and then manually enter the placeholders. However, the system could do more for them. The placeholders can easily be generated by the framework. A solution would be to let the user view the dictionary and generate placeholder-XML for a custom selection. Then the user only has to copy and paste the XML into the file. Even better would be to integrate the template part with an editor, because the copy and paste is not necessary then. However, that solution depends on a specific editor and the possibilities that the editor offers for integration. This approach tries to make the construction of a template a better experience for the user. Thus the template engine and the definition of templates is left untouched, but the construction process is supported by more tools.

Another possibility that is not researched is the creation of an abstraction on top of the template definition. This requires, for instance, the design of a Domain Specific Language (DSL) that describes a webpage. The DSL is then compiled into a webpage. The advantage of this approach is that HTML concepts that are not used or too complex can be left out. However, it also obstructs any familiarity that end-users might have with HTML, CSS, or JavaScript. It also prevents the usage of standardized webpage tools. However, the template language can also be made more powerful, for instance, by adding constructs that promote the reuse of (parts of) templates.

4.6 Summary

This chapter discusses the extension of the ADD.NET framework with a template engine. This engine allows users to define how a webpage should look. Users can place input controls, but they can also define buttons that trigger a certain function like saving the changes. This solution is a simple system that focusses only on the definition of the interface. Extensions are discussed that further improve the template engine in order to increase the user friendliness.

Although the template engine offers a flexible user interface, it does not offer the specification of business logic. The logic is built into the template engine and framework. If more logic is necessary, the framework has to be extended. However, the template engine can be used for CRUD-pages.

Chapter 5

Dictionary management

While the previous chapters describe how the new framework should use the dynamic dictionary, they do not discuss the management of the dictionary. This chapter goes into the details of the dictionary management. How can a user influence the dictionary? How are the dictionary and the database synchronized? This chapter presents an abstraction of the dictionary for users, discusses concurrency problems while managing the dictionary, and explains the synchronization of the dictionary with the database.

5.1 Introduction

The information structure stored in the dictionary is meant to be dynamic. If users want an extra field, table, or relation, they can add it themselves. Chapter 3 describes how the *dynamic* ADD.NET loads and refreshes information from a dynamic dictionary. The user, however, cannot influence or modify the dictionary in any way. Of course it is possible through a standard database tool, but remember that the users do not have a technical background. Therefore, a management applications is built: the *Dictionary Manager*. This application provides a user with a friendly way of editing the dictionary. The typical user that will perform these tasks is not likely to have a background in database systems and the relational model. This gives extra requirements on the application, making usability and user friendliness very important.

The application should guide the user while he is editing the dictionary. He should be prevented from doing things that are wrong and he should be warned for things that he might not want. The user should for instance be warned in cases that a dictionary modification causes the loss of data.

The dictionary structure is a representation of the relation model [Codd, 1970; Ramakrishnan and Gehrke, 2000]. As pointed out earlier: the users are not educated in database modeling. Therefore, the low-level details that are of no use to the user should be hidden. The dictionary does provide additional information such as error messages and help texts. However, the current structure only has room for one text per column. Multilingual applications are thus not supported. This extension is marked as future work.

5.2 Abstraction of the relational model

The chosen abstraction hides the detailed information such as foreign keys, foreign key columns, primary keys, and primary key description columns (see Figure 3.1). These things are not impor-

tant for the user and worse: they could confuse them. The abstraction simplifies the dictionary by presenting only three concepts: entities, fields, and relations. This abstraction is the result of the experience of DEVENT**it** in communicating data structures to non-technical customers.

Entities correspond with tables: every table is an entity and vice versa. Fields correspond with columns in the same way. Relations are the foreign keys between tables, but the direction of foreign keys is not shown in this abstraction. Instead, the abstraction presents the multiplicity of the relation. There are three possible multiplicities:

1 to many* or *many to 1 In this relation one row of the first table can be related to many rows of the second table. However, one row of the second table only relates to one row of the first table. This can be expressed in a database schema by creating a foreign key from the second table to the first table. An example is the relation between a *house* and a *window*. A house can have many windows, but a window can only belong to one house. The opposite is when many rows of the first table are related to one row in the second table, and a row from the second table can only be related to one row in the first table. An example is the relation between a *person* and a *place of birth*. A person can only have one place of birth, but of course multiple persons have the same place of birth.

many to many This relation expresses that one row from the first table can have a relation with many rows in the second table. One row from the second table can also have a relation with many rows in the second table. An example is the relation between a *person* and a *house*. A person can have many houses, but a house can also be owned by multiple persons. An extra table is needed to express this relationship in a database schema, because a single foreign key is not enough. The extra table will have a foreign key to the first table and a foreign key to the second table. This *link-table* relates the rows from the first table with the rows from the second table.

1 to 1 This last relation is easy: one row from the first table can be related to only one row from the second table and vice versa. An example is the relation between a *person* and a *social security number*. However, there is no straightforward way to express this in a database schema. The easiest way is to join the two tables into a single table. Relations with this multiplicity are not possible in the *Dictionary Manager*.

The examples given can also be found in Figure 5.1, which combines the examples into one database schema.

Figure 5.1: Three of the four multiplicity's

Although an abstraction is used, not all entities and not all fields are shown in the interface. The entities (or actually link-tables) created for many to many relationships are hidden. They only exist for the relation and are therefore useless to the user. However, in some cases the relation has attributes, for instance, the serial number. In that case the link-table is important, because the attribute is a column in that table. To solve this a user can explicitly create his own link-table and create two relations from that table to the other tables. In all other, simple cases, the user can

directly create a many to many relation and forget about the link-table. Fields that are hidden are the fields created to store foreign values and the fields that are part of the primary key. Every entity has at least one field: the **ID** field. This field is created by the application and is used as primary key: it identifies the row. This field is also used to create foreign keys. However, the users will never see these ID values. Although the user cannot see the primary key, he might want to add another unique identification to a table. Maybe there is a column (or a collection of columns) that should be unique. This is not discussed any further in this thesis, but instead is marked as future work.

This abstraction presents the dictionary in a simplified manner. Low level details like primary keys, and foreign keys are hidden or shown in a higher level manner. However, hiding this information is probably not desirable for developers using the framework. A solution for this would be the possibility to switch between a *simplified* mode and an *advanced* mode.

5.3 Concurrent schema and edit modifications

Editing the dictionary structure has consequences for the database. An important one is the fact that updates based on an old schema fail against the new schema: concurrent data editing and schema editing is likely to fail.

One possible solution would be to introduce locking (or *pessimistic concurrency control*). If someone starts editing the data, the schema is locked and if someone starts editing the schema, all data is locked. The BSCW application [Horstmann and Bentley, 1997] uses these kind of locks to control the concurrent editing of documents. The web-based nature of the archive management system, however, makes the application of a locking scheme hard. There are two options: using implicit locks and using explicit locks.

Implicit locks are applied whenever the user opens a document. The document is then locked. This lock prevents other users from editing the same document. The lock is released when the user leaves the document. It is the leaving that introduces difficulties: this leaving is subtle in a web application. The user can close his Internet browser or navigate to another page. These events are not signaled at the moment that they happen, they are only signaled at the moment that the users session expires. This expiration happens after a time-out. However, this time-out can be minutes after the actual closing of the document. As a result, documents are locked far longer than necessary. Explicit locks can make this easier: the user has to lock a document by clicking a button and he can release the lock by saving or clicking another button. This can decrease the duration of a lock, but the user has to remember that he should release the lock. A hybrid solution is also possible: applying and releasing locks implicit, but also allow the explicit form.

The locks on documents influence the lock on the database schema and vice versa. If a single document is locked, the schema has to be locked too. If the schema is locked, all documents are locked. It is likely that there are thousands of documents that belong to a specific schema. This makes the chance that one of those documents is locked big. The other way around, a lock on the schema, prevents the users of editing any of those thousands of documents. The locking scheme is to restrictive for an application of which the schema is changed regularly.

Another solution is *optimistic concurrency control*. This solution does not apply locks to the scheme or data. Instead when an update is committed the system checks for intermediate changes. If these are signaled, the user is notified. As a result, all the changes are lost.[1] This solution is less user friendly, because of possible loss of data. Not using locks, however, gives the system a higher availability.

The approach taken in the *dynamic* ADD.NET framework and in the *Dictionary Manager* is the

[1] In some situations the changes can be merged, which enhances the user experience of the system

optimistic concurrency control. This allows users to not think about locks and releasing them. Otherwise the complete document collection of a type can be locked while someone is reviewing the schema for modifications. Some adjustments to increase the user friendliness can be made, but they are not discussed further in this thesis.

5.4 Synchronization of dictionary and database

The dictionary should remain synchronized with the database. Otherwise the framework contains incorrect dictionary information and generated statements would fail. Therefore, modifications done by the user should result in Data Definition Language (DDL) statements that update the database schema. These statements should capture the exact modifications.

The synchronization does not happen after every action the user performs. Instead, the editing takes place in the same way as one edits a document. The document is opened in a word processor and the user performs different tasks. He might change the style of a piece of text, add some text and modify another piece. After these different actions, the user saves the document. In the same way, a user opens the dictionary, edits the dictionary in different steps, and saves the dictionary. In one edit session, the user might add a table, modify a field, and remove a relation. After an edit session is closed (the user saves the dictionary), the framework has to generate the correct update statements. Note that not every edit action corresponds to a single DDL statement. There are nine possible interface actions: addition, modification, and removal of tables, columns, and relations. Some of these actions can be translated into several DDL statements. For instance, the addition of a relation is one edit action. But the database schema is synchronized by adding a column (the referencing column) and one constraint (the actual foreign key).

Of course the analogy with a word processor does not hold completely. When writing a document users often save the document to make sure their modifications are stored. They do not finish their edit session before they save the document, but they save intermediate versions. With the dictionary every save triggers the synchronization of the dictionary with the database. Intermediate saves are thus more than only storing the dictionary in order to prevent data loss.

The DDL statements create tables, add columns, or remove relations. However, these statements have to be executed in the right order. For instance: a user adds an entity with some fields, and a relation from the new entity to an existing entity. The framework should first create the entity, then add the fields to that entity, and finally the relation can be created. If this order is not followed, a statement fails and the whole modification has to be undone. The ordering is determined on two levels. The action orders the statements locally, every action has specific knowledge about the order of the statements. The actions themselves have to be ordered too: they have to be executed in a certain order. The edit action determines that the addition of a relation is first the creation of a column and then the creation of a constraint. Globally, the synchronization determines that first a table has to be created, before the relation is created.

If the synchronization is only done after an edit session, the ordering is not necessary. That is because the user already gives the order, the application just has to execute the actions in the same order. However, the same synchronization process is also used later on in the version control system. For instance, if the system has to apply a certain set of differences to a dictionary in automated way there are now user actions to follow. The order in that case has to be determined by the application.

The result of an edit session is a collection of edit actions. This collection contains every edit action done by the user. This collection has to be ordered and then the actions have to be executed in this order. Or better, the statements generated by the actions have to be executed. This synchronizes the dictionary with the database.

Before talking about the ordering of the collection, it is important to look at the elements in the

collection: the kind of actions. The actions can be classified by the subject of the action: entities, fields, and relations. Every concept can be the subject of three different operations: addition, modification, and removal. This gives a total of nine types of actions. These actions are represented by functions. These functions take a number of arguments, for instance the function that creates a table takes that table as its argument. Nine different functions are distinguished:

$AT(t)$	addition of a table t
$MT(t)$	modification of a table t
$RT(t)$	removal of a table t
$AC(c, t)$	addition of a column c to table t
$MC(c, t)$	modification of a column c in table t
$RC(c, t)$	removal of a column c from table t
$AR(c_1, t_1, c_2, t_2)$	addition of a relation between column c_1 in table t_1 and column c_2 in table t_2
$MR(c_1, t_1, c_2, t_2)$	modification of a relation between column c_1 in table t_1 and column c_2 in table t_2
$RR(c_1, t_1, c_2, t_2)$	removal of a relation between column c_1 in table t_1 and column c_2 in table t_2

These functions are the elements of the collection that results from an edit session, identified in the following text by A. However, the application enforces rules upon this collection of actions. These rules constraint the composition of the collection.

Another situation has to be explained first. Relations are a concept that hides details from the user, as explained earlier in this chapter. A relation actually consists of columns, constraints, and sometimes even a table in the database schema. These columns and tables are not shown in the *Dictionary Manager*, instead they are hidden. A many to 1 relation consists of a column and a constraint, just like the 1 to many relation. The many to many relation, however, consists of a table (the link-table) with three columns, and two constraints. These things are hidden from the user: they only see the relation. As a consequence, a user cannot perform actions on the columns created for a relation. These are not visible, and there cannot be operated upon. The same holds for tables. Every table has at least one column, the **ID** column. This column is hidden from the user and he thus cannot modify or remove it. If the table is created, that column is created and if the column is removed if the table is removed.

There is only one action that performs an operation on an instance of a table, column, or relation. All edit actions in one edit session are combined such that the result is one action. For instance, if a column is modified in three separate actions, the result is one single statement that combines these three actions. If a table is added and removed in the same session, the result is nothing, because the two actions cancel each other out. Note the subtle notion of *instances*. The different instances are not solely identified by their name. The following sections make this clear. The following table shows how two actions are combined:

first action	second action	
addition	modification	Resulting action is a single addition.
addition	removal	*nothing*, the removal cancels the addition.
addition	addition	*not possible*, an instance can only be added once.
modification	modification	Resulting action is a combined modification.
modification	removal	The modification is lost, removal is executed.
modification	addition	*not possible*, the modification implies the existence of the particular instance, addition of an existing instance is not possible.
removal	addition	This combination is possible, but the actions do not interfere with each other, the new instance is different from the old instance. The second addition is not a restore! Both actions are executed, because they work on different instances.

first action	second action	
removal	modification	*not possible*, something cannot be modified after removing it.
removal	removal	*not possible*, things can only be removed once.

The result *not possible* shows that a certain combination of actions is not possible in the application, in other words: the application prevents the second action if the first action is done. For instance the third combination: two addition actions. If a user adds a particular table he cannot add it again, hence the *not possible*. However, the resulting action that is part of A is the first action: the addition. Another example: the eight combination. If a user removes a column as the first action, he cannot modify it as the second action. The column is no longer shown in the interface and consequent actions cannot be done, hence the *not possible*. Again the resulting action that is part of A is the removal. The result *nothing* means that there will not be a statement in A that corresponds to one of the actions. If a table is created and removed in the same edit session, nothing has to be done to synchronize the dictionary with the database schema.

The two actions take place chronologically. If there are more than two actions, the third and following actions are combined with the resulting action of the previous combination. The result *nothing* combines with everything: the result is the other action. At the end of an edit session, there is a maximum of one statement per table, field, or relation.

This can be formalized into the following rules. These rules explain the composition of A. The first rule states that for all tables, there cannot be a composition of A such that two actions working on the same table are present.

$$\forall t \quad : \neg(AT(t) \in A \wedge RT(t) \in A)$$
$$\forall t \quad : \neg(AT(t) \in A \wedge MT(t) \in A)$$
$$\forall t \quad : \neg(MT(t) \in A \wedge RT(t) \in A)$$

$$\forall c, t \quad : \neg(AC(c,t) \in A \wedge RC(c,t) \in A)$$
$$\forall c, t \quad : \neg(AC(c,t) \in A \wedge MC(c,t) \in A)$$
$$\forall c, t \quad : \neg(MC(c,t) \in A \wedge RC(c,t) \in A)$$

$$\forall c_1, c_2, t_1, t_2 : \neg(AR(c_1,t_1,c_2,t_2) \in A \wedge RR(c_1,t_1,c_2,t_2) \in A)$$
$$\forall c_1, c_2, t_1, t_2 : \neg(AR(c_1,t_1,c_2,t_2) \in A \wedge MR(c_1,t_1,c_2,t_2) \in A)$$
$$\forall c_1, c_2, t_1, t_2 : \neg(MC(c_1,t_1,c_2,t_2) \in A \wedge RR(c_1,t_1,c_2,t_2) \in A)$$

If a table or column is removed, another action cannot add or modify something that is related to that table or column. For instance, the removal of a table cannot co-exist with an action that adds a column to the same table.

$$\forall c, t \quad : \neg(RT(t) \in A \wedge AC(c,t) \in A)$$
$$\forall c, t \quad : \neg(RT(t) \in A \wedge MC(c,t) \in A)$$

$$\forall c_1, c_2, t_1, t_2 : \neg(RT(t_1) \in A \wedge AR(c_1,t_1,c_2,t_2) \in A)$$
$$\forall c_1, c_2, t_1, t_2 : \neg(RT(t_1) \in A \wedge AR(c_2,t_2,c_1,t_1) \in A)$$

$$\forall c_1, c_2, t_1, t_2 : \neg(RT(t_1) \in A \wedge MR(c1,t_1,c_2,t_2) \in A)$$
$$\forall c_1, c_2, t_1, t_2 : \neg(RT(t_1) \in A \wedge MR(c_2,t_2,c_1,t_1) \in A)$$

Not only the actions involving removal are constrained, also the actions involving addition are constrained. It is impossible to remove a relation if one of the tables involved is created in this edit session. The relation cannot exist before the table is created. The same holds for newly created columns.

$$\forall c, t \quad : \neg(AT(t) \in A \wedge MC(c,t) \in A)$$
$$\forall c, t \quad : \neg(AT(t) \in A \wedge RC(c,t) \in A)$$

$$\forall c_1, c_2, t_1, t_2 : \neg(AT(t_1) \in A \wedge MR(c1, t_1, c_2, t_2) \in A)$$
$$\forall c_1, c_2, t_1, t_2 : \neg(AT(t_1) \in A \wedge MR(c_2, t_2, c_1, t_1) \in A)$$

$$\forall c_1, c_2, t_1, t_2 : \neg(AT(t_1) \in A \wedge RR(c1, t_1, c_2, t_2) \in A)$$
$$\forall c_1, c_2, t_1, t_2 : \neg(AT(t_1) \in A \wedge RR(c_2, t_2, c_1, t_1) \in A)$$

The given rules describe the composition of A. Some compositions are not possible, meaning that a user cannot combine certain actions. However, by ending the edit session the actions become available again, because A is empty again. In some cases it might be easier for the user to end the current edit session, by saving it, and start a new session.

Now that is defined what the composition of A looks like, the actions can be ordered. The actions can be ordered in different ways. The ADDWin application, which generates actions to create tables, columns, and relations, orders the actions on table dependency. This means that the statements of table a are generated before the statements of table b if b has a foreign key to table a. The type of actions and composition of the action collection is ignored in this ordering. However, this ordering scheme cannot deal with the situation where two tables both have a foreign key to each other. The advantage of this ordering is the readability of the output. Statements are grouped per table and can therefore easily be found.

The second ordering does look at the type of the action and ignores the table dependencies. A partial order can be defined on the functions that are explained earlier. The ordering is described in the following rules, which are discussed groupwise. These rules only constrain the order, if there are no constraints the order does not matter.

The second way of ordering is used in the synchronization of the dictionary with the database. The ordering is given below, grouped per action type. The rules formulate that a specific action should be executed before another action if the particular condition holds, denoted with the operator $\sqsubset$. This ordering is a partial order: it is irreflexive, transitive, and asymmetric:

- Irreflexive: $\forall a \in A \quad : \neg(a \sqsubset a)$
- Transitive: $\forall a, b, c \in A : a \sqsubset b \wedge b \sqsubset c \rightarrow a \sqsubset c$
- Asymmetric: $\forall a, b \in A \quad : a \sqsubset b \rightarrow \neg(b \sqsubset a)$

The following list specifies per action what the ordering rules are:

- $AT(t)$

  The following actions specify which actions should be executed after the addition of a table. Remember that there is only one action that operates directly on this table (addition, modification, removal). Therefore, the order of AT and MT is left unspecified: that situation will never happen. Another situation that is already ignored is the possibility of a deleted column that is part of the newly created table.

  $$\forall c, t \quad : AT(t) \in A \wedge AC(c,t) \in A \quad : AT(t) \sqsubset AC(c,t)$$
  $$\forall c_1, c_2, t_1, t_2 : AT(t_1) \in A \wedge AR(c_1, t_1, c_2, t_2) \in A \quad : AT(t_1) \sqsubset AR(c_1, t_1, c_2, t_2)$$
  $$\forall c_1, c_2, t_1, t_2 : AT(t_1) \in A \wedge AR(c_2, t_2, c_1, t_1) \in A \quad : AT(t_1) \sqsubset AR(c_2, t_2, c_1, t_1)$$

  The rules state that columns and relations can only be added to the table after the addition of that particular table.

- $MT(t)$

  The second group defines the order between the modification of a table t and other actions. One situation that is not discussed here, is the situation where two tables are modified together and swap name. This situation is not handled in the ordering, because it is impossible to do so. Instead that situation is identified and should be dealt with as a specific case using temporary names.

$$\begin{array}{lll}
\forall c,t & : MT(t) \in A \wedge AC(c,t) \in A & : MT(t) \sqsubset AC(c,t) \\
\forall c,t & : MT(t) \in A \wedge MC(c,t) \in A & : MT(t) \sqsubset MC(c,t) \\
\forall c,t & : MT(t) \in A \wedge RC(c,t) \in A & : MT(t) \sqsubset RC(c,t) \\
\forall c_1,c_2,t_1,t_2 & : MT(t_1) \in A \wedge AR(c_1,t_1,c_2,t_2) \in A & : MT(t_1) \sqsubset AR(c_1,t_1,c_2,t_2) \\
\forall c_1,c_2,t_1,t_2 & : MT(t_1) \in A \wedge AR(c_2,t_2,c_1,t_1) \in A & : MT(t_1) \sqsubset AR(c_2,t_2,c_1,t_1) \\
\forall c_1,c_2,t_1,t_2 & : MT(t_1) \in A \wedge MR(c_1,t_1,c_2,t_2) \in A & : MT(t_1) \sqsubset MR(c_1,t_1,c_2,t_2) \\
\forall c_1,c_2,t_1,t_2 & : MT(t_1) \in A \wedge MR(c_2,t_2,c_1,t_1) \in A & : MT(t_1) \sqsubset MR(c_2,t_2,c_1,t_1) \\
\forall c_1,c_2,t_1,t_2 & : MT(t_1) \in A \wedge RR(c_1,t_1,c_2,t_2) \in A & : MT(t_1) \sqsubset RR(c_1,t_1,c_2,t_2) \\
\forall c_1,c_2,t_1,t_2 & : MT(t_1) \in A \wedge RR(c_2,t_2,c_1,t_1) \in A & : MT(t_1) \sqsubset RR(c_2,t_2,c_1,t_1)
\end{array}$$

  An arbitrary choice is made in the fact that the modification actions on tables are executed before the specified actions. The actions that are listed use the new, updated information of the table. However, they could also use the old information, but then the actions have to be executed before the table modification actions.

- $RT(t)$

  There is only one rule that constrains the removal of a table:

$$\begin{array}{ll}
\forall t_1,t_2 & : RT(t_1) \in A \wedge AT(t_2) \in A \wedge name(t_1) = name(t_2) \\
 & : RT(t_1) \sqsubset AT(t_2)
\end{array}$$

  The condition is somewhat special: t_1 and t_2 cannot be the same table. If the addition is done before the removal, there would be no action in A, if the removal is done before the addition the two tables are different instances (as explained in the table with action combinations). However, the new table can use the name of a removed table. In that case the old table has to be removed first.

- $AC(c,t)$

  No actions interfere with the addition of a new column.

- $MC(c,t)$

  No actions interfere with the modification of a column.

- $RC(c,t)$

  Removing a column should be done before the removal of the table to which the column belongs:

$$\forall c,t \quad : RC(c,t) \in A \wedge RT(t) \in A \quad : RC(c,t) \sqsubset RT(t)$$

  Note that most modern DBMSs allow the deletion of a table with columns still in it (the columns are removed automatically along with the table). For completeness, assumed is that the system removes all columns first and then the table.

- $AR(c_1, t_1, c_2, t_2)$

  There are no actions that have to be executed after the addition of a relation.

- $MR(c_1, t_1, c_2, t_2)$

  There are no actions that have to be executed after the modification of a relation.

- $RR(c_1, t_1, c_2, t_2)$

  Before a table can be removed, every relation that involves that particular table as to be removed.

$$\forall c_1, c_2, t_1, t_2 : RR(c_1, t_1, c_2, t_2) \in A \wedge RT(t_1) \in A \quad : RR(c_1, t_1, c_2, t_2) \sqsubset RT(t_1)$$
$$\forall c_1, c_2, t_1, t_2 : RR(c_2, t_2, c_1, t_1) \in A \wedge RT(t_2) \in A \quad : RR(c_2, t_2, c_1, t_1) \sqsubset RT(t_2)$$

This ordering specified by the rules above has a high level of granularity. For the sake of laziness, these rules can be generalized, making the order more strict (the algorithm can then order on action type, and does not have to inspect actions). The generalization is done by removing the conditions of the rules given. The resulting rules are:

$$\begin{aligned}
&\forall c_2, t_1, t_2 &&: AT(t_1) \sqsubset AC(c_2, t_2)\\
&\forall c_2, c_3, t_1, t_2, t_3 &&: AT(t_1) \sqsubset AR(c_2, t_2, c_3, t_3)\\
&\forall c_2, t_1, t_2 &&: MT(t_1) \sqsubset AC(c_2, t_2)\\
&\forall c_2, t_1, t_2 &&: MT(t_1) \sqsubset MC(c_2, t_2)\\
&\forall c_2, t_1, t_2 &&: MT(t_1) \sqsubset RC(c_2, t_2)\\
&\forall c_2, c_3, t_1, t_2, t_3 &&: MT(t_1) \sqsubset AR(c_2, t_2, c_3, t_3)\\
&\forall c_2, c_3, t_1, t_2, t_3 &&: MT(t_1) \sqsubset MR(c_2, t_2, c_3, t_3)\\
&\forall c_2, c_3, t_1, t_2, t_3 &&: MT(t_1) \sqsubset RR(c_2, t_2, c_3, t_3)\\
&\forall t_1, t_2 &&: RT(t_1) \sqsubset AT(t_2)\\
&\forall c_2, t_1, t_2 &&: RC(c_2, t_2) \sqsubset RT(t_1)\\
&\forall c_2, c_3, t_1, t_2, t_3 &&: RR(c_2, t_2, c_3, t_3) \sqsubset RT(t_1)
\end{aligned}$$

These rules are visualized in Figure 5.2. Orderings that are redundant because of transitivity are omitted. One of the possible orderings is MT, MC, MR, RR, RC, RT, AT, AC, and finally AR.

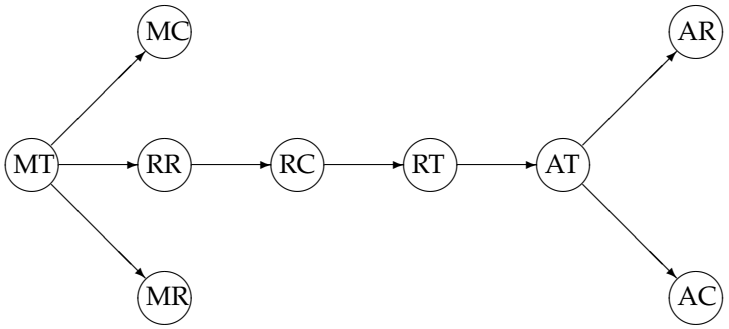

Figure 5.2: Coarse statement-type ordering of DDL statements

5.4.1 Data aware schema modifications

The DDL statements that are generated by the manager to synchronize dictionary and database are data ignorant. The fact that they are data *definition* language statements means that they only work on the definition level of a database. Another part of SQL consists of *Data Manipulation Language* (DML) statements. These statements alter the data stored in the schema.

The data unawareness of DDL statements, does not mean that the synchronization cannot be data aware. The important part of an archive management system is the data. It is thus very important that data is preserved as good as possible.

The addition of a table has of course nothing to do with data awareness. A whole new concept is added to the database and there is not yet data to be aware of. Modification of a table on the other hand has to do with data. Nobody wants to lose data stored in a table, just because that particular table is modified. Lets first look at the modification itself, and what it means if a table is modified. The only aspect of a table that can be modified (at least through the *Dictionary Manager*) is the name of a table. This renaming of a table is often supported by the DBMS by providing a special statement. This statement itself is data preserving. However, even if this statement is not available, the renaming is possible. It only has to be done in several steps. A new table has to be created that only differs from the original table on the name, the new name has to be used. Every row is copied to this new table. The relations directed at the original table have to be redirected to the new table (this has to be done by adding the relations to the new table and removing them from the original table). After that, the original table can be removed. The modification now results in more work, but it can be done with data preserving. The removal of a table again has to do with data. However, nothing can be preserved if a table is removed. And this is desired, otherwise why would someone remove a table?

The addition of a column is not similar to the addition of a table. Existing rows (data) are extended with a new column. However, these rows should be preserved of course, the new column should be added to every existing row. Again all DBMSs provide statements to add a column to a table already containing data. A few exceptional cases do exists, however. If a new column is marked as required, the system has to know a value for these rows. This can be a default value, equal for all rows, but it can also be the result of some procedure. The addition of a required column is therefore often done in several steps. First the column is added as optional. Then the existing rows can be modified by supplying a value for the new column. After that, the row can be modified by marking it required. The last two steps are of course not needed if an optional column is added to a table.

The modification of a column is similar to the modification of a table. However, more types of modifications are possible: renaming, data type conversion, and from required to optional (and vice versa). Again, the DBMS often provides direct statements for this. But it can also be done without these statements: a new column is added, data is copied, and the old column is removed. If the name is not changed, a temporary column in between is needed. In a few situations, more input is required. If the type of a column is changed, the system should know how to convert the values of one type to the other. If the column is changed from optional to required, the same problem arises as when a required column is added: values for existing rows have to be supplied. In many cases, the user should supply these values. In that case a warning should be given that it is not possible to change from optional to required, because not all rows have a value.

The removal of a column is again similar to the removal of a table: a part of the schema is removed, and so is the data stored in that part. This data cannot be preserved (but there are other solutions, given in Chapter 6).

The addition of a relation can be easy: the relation is created but none of the rows are related to each other until the users relate them. However, if the relation is required, the same problems arise as with a required column. The existing rows suddenly have to be related to each other. Again, the relation has to be created in several steps: first adding the relation as optional. Then

the users can relate rows to each other, and then when all the rows are related, the relation can be made required. Note that this causes a removal and addition of the relation again. The required part of a relation is not expressed in the foreign key (or constraint) added to the database schema. Instead, the column involved in the relation is marked as required. Thus if a relation is changed from optional to required, it is the column that is modified.

The modification of a relation is allowed in the interface, but it is translated to a removal of the old relation and addition of the new relation. The removal already indicates that the data is not preserved. However, there are other options. As explained before, the relations as shown in the *Dictionary Manager* are actually foreign keys and columns in the database schema. The foreign keys can be removed separated from the columns, this removes the constraint, but it preserves the actual data. This means that the system can remove the constraint, transform the data, and then add the new constraint. This transformation can be done, but it is not always fully data preserving. The kind of modification determines if it can be done data preserving. The first type is the change from optional to required. This is explained together with the addition of required relations: it can preserve data. The second type is the change of the multiplicity of the relation, and unfortunately that cannot always be done data preserving.

There are six possible changes, from each of the three multiplicities (1 to many, many to 1, many to many) to the other ones. If a 1 to many or many to 1 relation is changed into a many to many relation, the relation is *widened*. This means that the related rows in the old relation can still exists in the new relation. The other way around, however, is *narrowing* of a relation. This means that not all related rows can exist in the new relation. This choice, however, cannot be made by the system. The user should tell the system which related rows survive the narrowing. The other changes, from 1 to many into a many to 1 or vice versa are also narrowing. Although the multiplicities are similar, they cannot express the same. Thus, the widening of a relation is data preserving, but the narrowing is not. Changing from 1 to many into many to 1 and then back again, is not the *identity* function: data is lost by narrowing!

The last statement type, the removal of a relation is similar to the removal of tables or columns: a part of the schema is removed and thus data is lost.

Because the data awareness is built into the edit actions, the ordering as described in the previous section can be left untouched. It does not matter that a single action generates multiple DDL and DML statements, as long as the actions are executed in order.

Preserving data with single actions might be complex in some cases, other problems arise when the user is *refactoring* the database schema. For instance, consider the situation that was outlined in Section 2.3. In that particular use case, the organization did the refactoring in two phases. However, one can see that by executing the right statements, this can be done in one automated step (except the reconciliation of different spelling variations). However, the complex part of this use case is the fact that this refactoring is hard to signal in the application. The application does not allow that a column is moved to another table. This means that the column move has to be done by removing the column in one table and adding it in another table. These are two different edit actions, and the application cannot unambiguously detect that the user meant it to be one action: moving the columns.

The combination of multiple edit actions into a new big edit action is not discussed in this thesis. Instead the refactoring of the dictionary is marked as future work. The extension could be the addition of a collection of refactorings, that each consist of multiple edit actions. A refactoring should also contain the knowledge of how to perform the refactoring in a data preserving manner.

The synchronization of the dictionary with the database schema should not lose any data if it is possible to keep or transform the data into the new schema. Actions that involve the loss of data are the removal of tables, columns, and relations, along with the narrowing of relations through modification. All other actions do not involve data loss.

5.5 Dictionary validation

The *Dictionary Manager* guides the user that modifies the schema. This means that the interface should prevent things that *are* wrong and should warn about situations that *might* be wrong. The dictionary information is validated on a few aspects.

The first validation is the uniqueness of table, column, and relation names. This validation is easy, but should also take into account that the database might contain tables that are created by other applications. Some names might also be reserved by the DBMS. All these exceptions should be reported to the user and creating an entity or field with a reserved name should be prevented.

Another validation is the presence of unrelated tables in a schema. A user might remove relations, but forget to remove the unused tables. This is no error, but it leads to garbage in the structure and thus in the database. The manager should warn the user about these unrelated tables. Every table should be reachable from every other table by walking over the relations. If table(s) are not reachable, the schema is split into parts.

A last validation is the addition of required elements to the schema that cannot be done, because of existing data. For instance the change of an optional column to a required column. This can only be done if the data already conforms to this situation. If it is not possible to add the required element, the user is warned and should first correct the data.

5.6 Summary

The *Dictionary Manager* as described in this chapter offers a clear view on the dictionary and allows non-technical users to modify the database. It abstracts from various details of the relational model to make it comprehensible. The manager makes sure that the dictionary and the database is kept synchronized by translating edit operations into DDL statements. The edit operations are ordered in such a way that the synchronization can be done without errors. The DDL statements are then executed in this order. Validation checks should prevent that the dictionary or the database is left in an unusable or incorrect state.

The *Dictionary Manager* provides means to modify the database schema. The DDL statements that are executed work on the data definition level, not on the data level. This chapter explains how various edit actions can still preserve existing data, but also identifies those actions that cause data loss. More advanced scenarios are discussed, which can only be solved by using predefined refactorings. These are not explored or discussed in this thesis.

All of these procedures might crash in the middle and would leave the database in an incorrect state. The database transaction mechanism is not sufficient, because it often does not allow the mixing of DDL and DML statements. Therefore, a custom transaction mechanism should be built and used in the *Dictionary Manager*. This mechanism is not discussed in this thesis, but instead marked as future work.

Chapter 6

Version control system

The previous chapters described a dynamic extension of ADD.NET, a template engine built on top of the framework and the *Dictionary Manager* (an application for dictionary management). This *Dictionary Manager* can be seen as a development environment for dictionaries. The previous chapter describes how it synchronizes the dictionary with the database. It offers an environment in which users can modify the dictionary, changes are validated and a correct state is preserved.

Modern development environments often integrate a version control system. Such a system keeps track of the history of edit sessions (can be per file or per project). This chapter discusses how the *Dictionary Manager* can be enhanced by extending it with a version control system. This gives users insight in the change history, but also allows them to undo certain edit sessions.

6.1 Introduction

The database schema for an arbitrary application is likely to change [Lientz, 1983; Roddick, 1995]. By providing a dynamic dictionary, and thus a dynamic database schema, this probability increases. The schema in an arbitrary *dynamic* ADD.NET framework will change, because the resources are available. Changes in an arbitrary software system are often regulated by storing the source code of an application in a version control system [Pilato et al., 2004; Rochkind, 1975; Tichy, 1982]. Such a system regulates the concurrent editing of source files, but it also keeps a record of the change history. This gives insight to developers of how a particular source file has changed in a certain period of time. It aids the writing of release notes. But the versioning of source code does not only give insight in the past, it also makes it possible to revert to older versions. It creates an *undo* function.

The database schema is, much like ordinary source files, part of an application. It seems natural to control the database schema with a version control system. The database schema is an integral part of the DBMS. Possibilities to control a database schema are limited and not as easy as other source files. One possibility is to store the SQL script that creates the schema inside the version control system. The file has to be kept synchronized with the actual database schema used in the DBMS. A drawback of this approach is that the version control system cannot give a difference between two versions in terms of DDL statements. This minimizes the use of a version control system for database schemas.

The source code of the *dynamic* ADD.NET framework and the applications built on top of it are stored in a version control system. The dictionary information seems to be part of the applications data and might be viewed as equal to the actual documents stored in an archive management system. But the dictionary information is not only part of the application data, it is also part of the

application itself! The dictionary describes the database and feeds the interface with information, which both are important parts of the application. This pleads for storing the dictionary in a version control system.

The dictionary is stored in a database, as described in Chapter 5. As explained before, storing a database schema in a version control system is hard and an unnatural fit. The dictionary in this case is not so much a database schema, it is data stored inside a database. Of course it represents the database schema, but it is the data that should be versioned. This situation can hardly be solved with a conventional, file-based version control system. If a file-based system is used, the dictionary should be stored in some kind of file format. The database file(s) can be used for this, but they are often binary which makes it nearly impossible to generate a meaningful difference. Another solution is to export the dictionary to a special file format, but that places the version control system outside the *Dictionary Manager*. Users have to install a version control system, reverting a version has to do with reverting the file stored in the system and then importing that file into the *Dictionary Manager*. Summarizing, it is not an easy task to integrate a conventional version control system with the *Dictionary Manager*. Therefore, the *Dictionary Manager* is extended with a custom version control system. In this way the *Dictionary Manager* stays the central management application for the dictionary.

Version control should give insight in the change history of the dictionary. But it also should give the possibility to revert certain changes. Of course, all this functionality should be presented in a clear and user friendly manner.

6.2 Related version control systems

This is not the first system that approaches the problem of controlling different versions of a database schema. Different approaches exists: one approach is by allowing applications to view and maintain certain information through older schemas. This approach is often seen in Object-Oriented Database Systems (OODBMS) [Chou and Kim, 1988; Kim and Chou, 1988], but also in Temporal databases [Castro et al., 1997; Wei and Elmasri, 2000]. In many cases the database system allows the modification of schemas, and the using of older schemas to access data. In a survey written by Roddick [Roddick, 1995] two levels of freedom are distinguished.

The first level, *schema evolution*, provides a basic level of freedom. It allows the modification of a schema in a populated database. The guarantee is given that existing data will not be lost for those modifications that do not remove parts of the schema. In some situations data loss is inevitable. The data itself is no longer accessible through the old schema, but is restructured according to the new schema. Commercial relational DBMSs such as Microsoft SQL Server and Oracle support these modifications through the DDL. Applications that are built on top of a DBMS that only provides schema evolution are subject of the software maintenance that was pointed out earlier. The source code of the application has to be reviewed after schema modifications.

The second level, *schema versioning*, is in fact a collection of levels. Databases that provide schema versioning do not all provide the same level of versioning. A database, however, should at least provide a history of schemas and the ability to view data through different schemas. Some systems also provide a history of the actual data. Schema versions can be labeled and used as stable points. Schema versioning allows multiple versions that are *active* at the same time. The data is transformed such that it can be viewed through another version of the schema. The versions are *views* upon the actual data.

A related project is the analysis on schema evolution to aid the software maintenance. One example is visualizing schema evolution impact on applications [Karahasanovic and Sjøberg, 2001]. This project shows in an object-oriented system where the schema changes affect the application. However, it does not provide an automatic evolution of the application. A second example is

the quantification of evolution [Sjøberg, 1993]. This project aims at the guidance and analysis of schema evolution. With the results of this analysis, developers can be directed at possible change locations.

6.3 Version storage approaches

The implementation of a version control system should take care of storing the history, the changes made. The system should provide an interface to that history, but also to the individual versions. The storage can be done in different ways: two extremes are storing every version as a separate copy and only storing the changes (*deltas*). Choosing between the two approaches is a decision between costs: time costs or space costs. Time costs refer to the costs of building an individual version. If every version is stored as separate copy, these costs are zero, because the version does not have to be *materialized*. However, when only the deltas are stored, a version has to be calculated from those deltas. The space costs refer to the storage that is needed by the version control system. Storage needs increase rapidly if every version is stored as a separate copy. This is very important when the deltas are small and the actual file is big (as is the case with binary files). Storing deltas is less storage consuming: only the changes between two versions are stored. The storing of deltas gives little advantage if these changes are big. Obviously, there is a trade-off in this decision. Using less storage means using more time and vice versa. A good analysis of this trade-off can be found in work by Yu and Rosenkrantz [Yu and Rosenkrantz, 1988].

Dadam et al. [Dadam et al., 1984] give a good overview of the different approaches for version control. Five different forms of version control systems are distinguished.

- The first approach stores the first version as a complete version. The following versions are stored as a delta. Every delta only contains the differences made in that particular version. In order to materialize version n, the deltas 1 to n have to be applied on the complete version 0. All deltas have to be applied in the right order to materialize a version.
- The second approach stores the deltas between a version n and the first, complete version 0. This means that in order to materialize a version n, only the delta of that version has to be applied to the version 0. This means that the materialization of a particular version is fast. However, the deltas do become bigger with every new version. It also is hard to show the differences between two versions (that are not equal to version 0). There is no delta between those two, it has to be calculated.
- The third option again stores deltas. However, these deltas are not with respect to the first version or the previous version, but with respect to the following version. The last version is accessible without any penalties. The older versions are more time consuming to materialize with every new version.
- The fourth option stores deltas with respect to the last version. Again the last version is accessible without penalties. The other versions are all one delta away from that version. However, with every new version all deltas have to be recomputed. This is of course very time consuming and is a big disadvantage of this approach.
- The fifth approach that Dadam discusses is the storage of separate copies for every version. They see the storage usage of this approach as an insuperable disadvantage.

Approach two and four have the advantage of fast materialization of particular versions. Approach one is slow, and it becomes slower with every new version. Approach three is also slow, but it becomes slower when older versions are materialized, not when the current version is

materialized. Approach four is slow in creating new versions, because all deltas need to be recomputed. Five is fast in materialization, but needs the most storage capacity.

They disregard approach one, four and five from further investigation, because of the disadvantages they identified. While further investigating the two remaining solutions, they identify another solution: a hybrid form of approach one, two, three, or four with five. This means that a complete version is stored every x deltas. This increases the need for storage capacity, but also saves calculations in order to materialize certain versions. Approach three has one advantage over two: old history can be purged without a penalty, in approach two this means that deltas have to be recomputed.

Kim and Chou chose the delta approach in their work on OODBMSs [Chou and Kim, 1988; Kim and Chou, 1988]. Their ORION database system is aimed at supporting CAD design. The application built on top of this system was multi-user. Therefore, the versioning of the designs was required. In their system, the schema and its data uses a significant amount of storage, and it is undesirable to store separate copies for every version.

The first source code version control system, SCCS [Rochkind, 1975], uses deltas. Again, the most important reason is the limitation of storage capacity. To materialize a particular version, all deltas have to be applied in order. Another well known version control system, RCS [Tichy, 1982], also uses the storage of deltas. However, the approach used in both systems is different. In SCCS, the deltas are stored as merged deltas. This means that all the deltas are stored in one file. Every entry in that file tells something about a specific line. It specifies in which versions the line is included and in which versions it is excluded. This file has to be scanned and applied to the root version. In RCS, every delta is a separate file and contains the instruction that an editor should execute to materialize a particular version.

The discussed approaches all had to deal with a limited amount of storage capacity. However, computational resources are limited too. Therefore, a hybrid approach can be used, as discussed by Dadam et al. [Dadam et al., 1984]. With a hybrid approach, the trade-off can be chosen to accommodate specific situations, for instance, by storing a complete version after each x number of deltas. This saves storage, but keeps the number of deltas to apply small.

The ADDWin application does its own versioning on the schema. It keeps track of all modifications done on a schema and stores them as separate copies (every edit session is a version). This gives insight in the history of a schema. The application lacks the possibility of viewing differences between two arbitrary versions. Storage capacity nowadays is no longer a real limitation. In addition, the storage of the database schema is negligible with respect to the core functionality of archive management systems. The framework will be used for those management systems. Their core functionality is the management of documents and multimedia. The probability that the storage of the database schema (and its versions) will influence the storage capacity is very small. To give a hint about the size of the dictionary, a few statistics were collected from the ADDWin database:

number of projects:	32	
number of versions:	2044	average 63 versions per project
number of tables:	86655	average 43 tables per project per version
number of columns:	619869	average 7 columns per table
size of the database on disk:	approximately 2.5 gigabyte	

Every project is a different application, and a different schema. This database stores every version of every database schema in use, since approximate 1996. This means that every schema has around six versions per year. This is including development of new applications and new schemas.

While using separate copies means an increase of storage capacity, using deltas means also more calculations to materialize versions. The computational resources are more limited than the storage capacity. The framework is web-based, the application thus serves multiple clients. The

resources have to be shared among multiple clients. The materializing of the current version is therefore important, it is the version that is used the most in the framework. The other versions are only materialized for reviewing purposes.

It is clear that both approaches have their advantages and downsides. The following sections explore the two approaches. The solutions are discussed based on two features: the version control system and the undo/redo functionality. The undo/redo functionality is first introduced, then the concept of *branches* is explained. After that the two solutions are discussed.

6.4 Undo and redo of dictionary versions

Undo is a common feature found in many applications nowadays. Text editors, word processors, image manipulation tools, they all provide some means to undo the last action. Often an action is done without knowing what the precise result is, users are likely to make mistakes that they want to undo later. Therefore, the *Dictionary Manager* provides an undo/redo functionality to support users. It gives an easy way of reverting any changes done in the past. The redo feature is actually the undo of an undo action.

The undo/redo functionality offered in the *Dictionary Manager* is different from the one seen in text editors and other applications. Often the undo only works on actions done after the last save of the current document. The undo offered in the *Dictionary Manager* undoes a complete edit session: all actions between two saves. Users can step through the different versions of dictionary. The undo function cannot remove version history: the redo would become impossible.

To undo a certain dictionary modification, two things have to be undone: the dictionary data modifications and the database schema modifications. The first is data inside a database, the second is the schema of a database. However, the second can also be reconstructed from the first.

After an undo or redo, the dictionary should still be synchronized with the database schema, just as described in Chapter 5. It should also preserve as much data as possible.

6.5 Branches of dictionaries

When a version is undone and the current version is edited, a new version is created. But where is this version placed? It cannot be stored as the latest version, because the versions in between are not the actual predecessors of the new version. Instead the version that was edited gets multiple successors. This is called *branching* in version control systems. The versions in the system, connected by the predecessor-successor relation, together can be viewed as a tree. Every version that has multiple successors is the root of those branches.

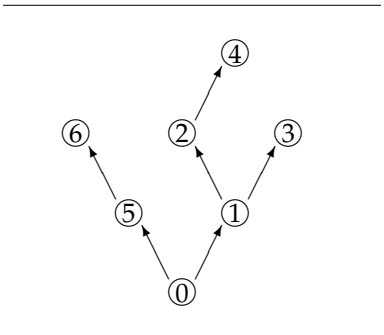

Figure 6.1: Branches

Branches are a normal concept in version control systems. They present alternatives that are equally important, for instance two alternative experiments that are worked on side-by-side or two versions that are maintained side-by-side. But how to deal with branches in a situation were there cannot be worked with two different active versions? There can only be one active dictionary, because it has to correspond with the database schema and the DBMS only allows one database schema. However, this does not mean that branches cannot be useful in this situation. The fact that there can only be one

active schema, means that there can only be one active branch. The switch between branches is changing the active schema and synchronizing the database again.

Using branches can be useful to explore alternatives. It can also be useful to combine two branches into one version: the two branches join again and become a single branch again. This is done by *merging* two versions into a new version. This is often seen in version control systems. The next chapter discusses the merging of dictionaries.

6.6 Separate copies

The first version control solution uses the implementation of the ADDWin application and applies it in the *dynamic* ADD.NET framework. It offers an easy access to separate versions: they can be retrieved without any computational effort.

A *current* version is introduced. This is a pointer to one of the copies stored in the version control system. The current version always corresponds with the database schema. The undo function is implemented as a change of the current version: a straightforward task. The same holds for a series of undo steps, the current version is changed to a previous version. To redo certain versions the current version is redirected again, but now to newer versions.

Updating the pointer only restores the dictionary, the data in the database. The database schema has to be restored too. Luckily, the DDL statements that bring the database schema from one version to another are already calculated by the synchronization algorithm. These statements are stored along with every version, such that they can be executed when a version has to be redone: the synchronization process only calculates the statements to *do* a certain version. To implement the undo function, the synchronization process is extended. For each action, the opposite statements are calculated too: the statement to delete a table is stored when a table is created.

Branching in this approach is fully supported: the undo and redo steps can be executed to switch between branches. Switching between branches is nothing more than a series of undo and redo actions. See for an example situation Figure 6.1. The active version is changed from 4 to 3, this means that the series of actions to synchronize the database become: *undo* 4 - *undo* 2 - *redo* 3. If the common ancestor is even further away in history (in other words, more versions are between the active version and a common ancestor with another branch), the series of actions becomes longer.

This implementation gives a straightforward undo/redo functionality. The synchronization process is extended and reused: the DDL statements can be executed to keep the database schema corresponding with the current version of the dictionary.

6.7 Dictionary deltas

The second approach uses the first solution described by Dadam: only the differences between two version are stored (the delta). The delta contains nothing more than what is changed in an edit session. It has to contain which tables and columns are new, which are modified, and what is deleted. This means that the database schema for the dictionary, as seen in Figure 3.1, cannot be used without modifications. The storage of new and modified data is rather easy: by storing partial dictionaries the current dictionary can be materialized by combining them, keeping the newest data. However, how to specify that certain information is removed? This can be done by extending every table in the schema with a new column: *removed*. This column is a boolean: false if the particular row is new or modified and true if the row is removed in the edit session. However, storing partial dictionaries also implies that the referential integrity of the DBMS can no longer be used, the different deltas are not connected.

The undo/redo function in this system is somewhat different then the one described in the previous solution. In the previous system one could only undo all versions between the current active version and the new active version. A version in the middle could not be undone, because of dependencies between the different versions. Imagine the following situation: a dictionary has seven versions (marked one to seven). The current version is seven. A user decides to undo version three (this is allowed in this imaginary example). This means that the delta between two and three needs to be removed from seven. After that, he decides to undo version seven. The current version becomes version six, but the delta from two to three needs to be removed from six. If the user decides to redo version three, the delta needs to be added to six again. What happens is that the current version is no longer a pointer to one of the versions in history, but a separate copy that represents the current version. Deltas are calculated to create this special copy. To support this kind of undo/redo functionality, the delta-based approach stores deltas.

This approach, selecting deltas, is also known as *cherry-picking* in literature on version control systems. Some version control systems support this better than others, a product that is said to support this well is Darcs [Darcs]. It allows the selection of deltas with a minimum number of dependencies.

The undo/redo function is implemented by extending every delta with an on/off switch. This switch expresses that a delta can be active or inactive. All active deltas are combined into the current version. If a delta is switched off it is no longer part of the current version. Switching it on again makes it again part of the current version.

Switching off a delta somewhere in the middle does not mean that all succeeding deltas have to be switched off too. It can be more flexible: sometimes not all succeeding deltas have to be switched off. There is a dependency between deltas, some deltas need other deltas. For instance, if a delta x creates a relation between a new table and a table created in delta x-1. In that case delta x depends on delta x-1. Switching off delta x-1 makes delta x impossible. These dependencies specify which deltas might be switched off (or on). The switching happens in groups: a delta can only be switched off if every delta that depends on the particular delta is switched off too. This is the same behaviour as the previous undo implementation. However, if a delta in the middle has no dependencies, it can be switched off without switching off the others.

The dependencies between deltas can be analyzed by using the partial ordering specified in the previous chapter. Every rule in the ordering scheme can be translated to a dependency between deltas. If, for instance, a table is added in delta a and a relation is created in delta b using that table, the ordering scheme tells that the table has to be created before the relation. Translating this to dependencies: delta b is depended on delta a. If there are actions in delta a that have to come before delta b, as described in the ordering schema, delta b depends on delta a. And thus a cannot be switched off without switching off b.

The deltas are stored with respect to previous versions, and not the succeeding version. This has to do with the current version. If the current version is stored as a separate copy, that copy has to be updated after every delta switching. This is now done in the framework, the part that loads the dictionary has to materialize it from the active deltas. The framework already stores the result in a cache. Instead of storing the current version in persistent memory, as a separate copy, the current version is stored in volatile memory. Storing it in as a separate copy in the database does not gain anything, the cache of the framework can even be shared such that other applications can use the materialized dictionary. Other versions then the current have to be materialized too of course. However, this is only for reviewing purposes and is less important than the current version. But these other versions can even be stored in a cache too, building a cache of materialized dictionaries.

The alternative undo/redo implementation also leads to a new branching implementation. Deltas are combined into a current version, but what if there are multiple branches? The new system therefore needs a *current branch*: only the deltas in the current branch are combined into the current version.

This implementation encourages a modular design of the database schema, if deltas are built in such way that the dependencies are small. This enables users to switch off certain parts. This undo/redo function along with branching makes it possible to design a schema by experimenting.

6.8 Data aware version control

The two different solutions for version control both have their advantages and disadvantages. Both solutions also have a common disadvantage: the fact that the data is not versioned. This means that a user can review previous versions of the database schema, but cannot see how the data looked at that point. A solution would be to backup the data just before a new version is created. This data can then be linked to the old version. Another solution is to just use the data aware synchronization as described in the previous chapter.

6.8.1 Backup facility

The first solution achieves versioning of data by storing the data in a backup table, whenever it is influenced by the creation of new version. For instance, the data stored in a column just before a column is removed. Of course not all the columns have to be stored, instead only the column that is removed along with a row identification has to be stored. Whenever a relation is removed, the rows related by that relation have to be stored in a backup table. This can be done by using the row identifications. The backup facility is also used when a column is modified. All actions that cause something to change in the data trigger the backup process.

The rollback approach means that the data stored in those backup tables is restored whenever a version is undone. The algorithm is the reverse of the algorithm that creates the backup table: columns that were removed are restored, modified data is copied back. This mechanism, however, does not allow the viewing of data through an older schema: it can only restore the data.

A disadvantage of this approach is that it cannot handle new data. For instance, the conversion of a column to another data type. The column data is changed after the conversion, and the conversion is undone. However, the backup facility does not convert the data back again, instead it restores the data from before the conversion. Modifications are thus lost.

6.8.2 Data transformations

The second solution is to reuse the algorithm used while synchronizing the dictionary and the database schema. In contrast of the rollback approach, this approach preserves data changes done in version x when reverting to version x-1. This approach does not have a backup facility. Therefore, it cannot rebuild a removed column, because that action results in data loss. These problems occur in situations where actions result in data loss. The previous chapter identified those situations: the removal of tables, columns, and relations, and the narrowing of relations through modification.

6.8.3 Hybrid solution to data awareness

Both proposed solutions have their drawbacks. The backup facility cannot deal with data changes done in a newer version, instead it restores the data from a previous version causing data loss.

The transformation approach cannot deal with columns or tables that are removed, no transformation can bring them back.

Remember the level on which the *Dictionary Manager* operates: on the level of a schema. As a result, the manager does not facilitate the versioning of data for backups and restores. It is not the intention to create some form of automated backups. Therefore, it seems more natural to transform the data to the new situation (for both the creation of new versions and the restoring of old versions). The transformation mechanism could be extended or mixed with the rollback mechanism in those situations where transformations are not enough. Restoring columns for instance is something that can only be done with a rollback of that column.

The situations in which the rollback mechanism should complete the transformation are already given: removal operations and the narrowing of relations through modification of an existing relation. If one of those actions is undone, the rollback mechanism restores the removed parts and the relational data before the narrowing. However, if one of the restored parts is required, new rows (rows added after that particular version) have to be extended somehow to fit in this new schema. This can be done with default values, or by restoring required elements as optional elements. The latter solution means that the undo function is more automated: it succeeds, but with warnings. The first solution asks for human intervention while undoing some version, and it can be that there is no single default value. Therefore, the latter approach is taken: required elements are restored as optional.

6.9 Conclusion

This chapter discusses two solutions for version control. The first solution stores every dictionary as a separate copy: versions are directly accessible. By introducing a current version the undo/redo functionality is implemented in straightforward manner. The second solution used deltas. This approach uses less storage capacity, but versions are not directly accessible anymore. The undo/redo functionality, however, becomes more flexible.

The undo/redo in the first solution has one disadvantage: it can only undo a complete series of versions. It is to difficult to undo only one step somewhere in the middle. The second solution solved this problem. The better solution depends on the trade-off that has to be made: flexible undo/redo and more computational resources or more storage capacity, less computational resources and a less flexible undo/redo.

The flexibility of the second solution makes it a better choice: it supports the features in the first solution plus more. It also is the better environment for experimenting with different schemas: parts can be switched off and on if the schema is built in a modular fashion. If different parts are added in different sessions, they can easily be turned off. The *Dictionary Manager* benefits greatly from the second, delta-based solution. It gives more flexibility and freedom to users.

One disadvantage is shared by both approaches: only undo and redo on a complete edit session can be done. This is the result of synchronizing per edit session, instead of per edit action. A solution is to synchronize per edit action, however, every edit action then becomes a new version. The history then consists of many more versions. Another approach is to store actions as a special kind of version. The edit sessions become stable versions, whereas the actions are intermediate versions. Synchronization can then be postponed to a stable version.

The supporting of versions and undo/redo functionality enhances the user friendliness of the *dynamic* ADD.NET framework. It allows users to experiment: bad decisions can be reverted. It also allows developers to use the framework as a development environment, supporting common features from other environments. The experimental phase should not be done on a production database, because the preserving of data is not guaranteed. Instead a combination of data transformations and data restores is used.

Chapter 7

Branch merging

The previous chapter left one subject untouched: how to merge different branches. Sometimes alternatives can be joined into a single dictionary. Automating this process makes it more user friendly.

Another situation where this problem arises is when a *dynamic* ADD.NET application is upgraded. Both DEVENT**it** and the customers can alter a dictionary. These concurrent modifications can be seen as two branches, although they are stored in separate databases. How to upgrade the application without loosing any modifications? This chapter discusses the merging of different dictionaries into a new version.

7.1 Introduction

The version control system as described in Chapter 6 offers a powerful control system to the users. They can review the history of modifications and undo certain steps in history. This mechanism gives insight into the evolution of the dictionary and enables to return to previous points in the history of the schema. The version control system allows branching, but the previous chapter left the merging of two different branches untouched. Merging is required to combine two different branches into a new version. This situation arises when, for instances, two experiments are done in parallel. One branch experiments with a certain part of the dictionary, while another branch experiments with a different part. If both experiments result in a permanent modification, the two parts have to be combined. This can of course be done by hand, but that is tedious job and is likely to introduce errors. An automated process is capable of merging the two branches without introducing errors.

There is another situation where the merging of two dictionaries is needed: the updating of a deployed application. Any *dynamic* ADD.NET application probably ships with some default dictionary (or multiple). They are developed for the application. However, if the *Dictionary Manager* is used, the dictionary is likely to change. If the application is updated, the dictionary might get an update too, it is part of the application so it could be that errors are corrected. So now there is a situation where both changes from customers and from DEVENT**it** have to be merged into a new version. The changes made by DEVENT**it** are deployed with an update of the application, the changes of customers are stored in the version control system. It is not possible to choose between the two, the DEVENT**it** changes might be necessary for the updated application while the customer changes are important too of course. The changes made by the users are also valuable and cannot be discarded. An automated merge process should fix this situation by merging the changes on both sides into a new version. The changes of the customers can be seen as one branch, while the updated dictionary from DEVENT**it** is another branch.

Merging the both versions into a new dictionary seems like the perfect solution to this problem. However, the situation where the dictionary is modified by both the customer and DEVENT**it** can also be prevented. To prevent the situation, dictionaries have to be marked as immutable. These dictionaries cannot be touched, only DEVENT**it** can change them. If customers want to change that particular dictionary anyway, they can copy the dictionary locally and alter the copy. If the application is updated, the changes are only applied to the original dictionary and not to the copy. The customers have to manually merge the changes of the original with the copy. Because the one situation is now prevented, a merging algorithm is not required anymore. Downside of this solution is that branches cannot be merged anymore. This means that the version control system can allow branching, but merging them is not possible anymore. Branching is then only useful for experimenting with two different approaches in parallel. The approaches solve the same problem, because only one of them can be used in the end. This solution is far from optimal, it limits the user and the features in the *Dictionary Manager*. Therefore, the remaining part of this chapter discusses automated merging: the possibilities and an algorithm that can be used for merging dictionaries.

7.2 Automated merge

Merging different versions of a file is a common action in software development [Mens, 2002]. The development of large software systems can benefit greatly if developers can develop in parallel. Parallel development on the same file can be done, if the communication is adequate. However, the two modifications have to be joined into one file again at some point. Therefore, merging is required. One of the version control systems that allows this is Subversion [Pilato et al., 2004]. It does not have a strict locking scheme. Developers have a private copy of the current version and edit that copy independently of other developers. At a certain moment in time these modifications need to be incorporated with the file stored on the server. This incorporation is done by merging the differences into a new version. Other developers can then do the same by merging their differences with the last version.

Many merge tools are based on textual merge techniques. They work line based: comparing files line for line. Obvious, they cannot use any knowledge about the contents of the file, the domain. Semantics and structures that are present in the content cannot be used, because the merge tool is unaware of their presence. Because of the domain-ignorance, these tools are limited in situations with a specific domain or where structured data is used. Merging source code with a textual merge algorithm can lead to syntactically incorrect files. These kind of errors need to be solved by hand, making the merge process less automatic and more cumbersome for developers.

Syntactic merging [Mens, 2002] does not merge line based, but uses the syntax of the given files. Instead of merging two collections of lines, the files are parsed into some representation and those models are used to merge. This can solve more conflicts, because the cases that result in a syntactically incorrect file are discarded. This kind of merging can of course only be used if the syntax of the two files is known and a parser is available.

Merge algorithms can be divided on more characteristics: the input. Two categories can be distinguished: two-way and three-way merging. In a two-way merge algorithm, only the last versions of the two branches are used. These two versions are merged in the best possible way. Conflict solving in these algorithms is not precise: often it is not obvious what happened. There is no way to tell if a line that differs is modified in one file or in both.

In a three-way merge, the common ancestor (the root version for both branches) is used. Instead of comparing the two different versions to each other, both versions are compared to the common ancestor. Those two comparisons are then used to find a result that incorporates both differences. By comparing the two different versions against this ancestor, more changes can be correctly merged. Because the ancestor is available, it can be seen in which branch a part was

changed. This type of algorithm is more powerful than the two-way merge and gives better results. Sometimes, the common ancestor is not available and in those situations two-way merging is the only alternative available.

The disadvantages of textual merge techniques makes it obvious that the dictionary merging needs a different approach. The dictionary is structured data from a specific domain, not just a text file. The merging is done by non-technical users: the conflicts should be solved with a minimal amount of human intervention. The common ancestor is available in the *dynamic* ADD.NET framework: three-way merging can be used. The following section discusses a three-way merging algorithm for XML documents in more detail.

7.3 Three-way merge for XML documents

The three-way XML document merge algorithm is described by Lindholm [Lindholm, 2004]. The algorithm described by Lindholm takes three document as input: documents $\mathbf{T}_0$, $\mathbf{T}_x$, and $\mathbf{T}_y$. Document $\mathbf{T}_0$ is the base document, this is the common ancestor of the other documents. Document $\mathbf{T}_x$ is the last version of the dictionary stored in the application deployed on the customers system. The third document, $\mathbf{T}_y$, is the version created by the developers. $\mathbf{T}_x$ and $\mathbf{T}_y$ are two branches, created from $\mathbf{T}_0$. The result of the merge is $\mathbf{T}_m$. The merge algorithm works only on XML files, but every tree-like data structure can be represented in an XML document. To use this algorithm, the dictionaries have to be transformed into XML files.

The documents are each represented as a collection of *parent-child-successor* (PCS) relations. The relation $pcs(r, p, s)$ expresses that r is the parent of both p and s and that p immediately precedes s in the child list of r. Three special characters are used: $\perp$, $\vdash$, and $\dashv$. The symbol $\perp$ is used to specify the root node, it stands for the empty parent. To express that a certain node is not preceded or succeeded, $\dashv$ and $\vdash$ are used. The first, $\dashv$, represents the first child node of every node, it is a special node that is always present. The last, $\vdash$, is used to represent the last child node of every node. Every node thus has at least two children: $\dashv$ and $\vdash$. The relations $pcs(\perp, \dashv, a)$ and $pcs(\perp, a, \vdash)$ express that the node a has no parent and no sibling nodes. The advantage of this representation is that nodes are described relative to their context, and not absolute: as seen from the root node. The relative description of nodes makes it easier to analyze the movement of nodes, nodes are often moved relative to their context. The content of a node is represented as $c(n, x)$: x is the content of node n.

Lindholm identifies rules, or guidelines, that direct the merge process. They define pre-conditions for the merge:

1. Nodes can be identified and compared. This is needed to identify certain elements in the XML document and reconcile them between the input documents.
2. $\mathbf{T}_m$ is defined by requiring it to contain certain node contexts. Nodes have a context consisting of parent, predecessor and successor. This is necessary for the PCS relations.
3. Some node contexts appear in $\mathbf{T}_m$ to prevent merging of changes originated from different trees.

The rules also describe key behavior of the algorithm:

4. Changes to the content of a node in one branch should occur in $\mathbf{T}_m$. These are the low-level changes, they should always appear in the merge result.
5. Nodes that are inserted in $\mathbf{T}_x$ or $\mathbf{T}_y$ should occur in $\mathbf{T}_m$. Newly added nodes should always be part of the merge result.

6. $\mathbf{T}_m$ should not contain nodes that are removed in $\mathbf{T}_x$ or $\mathbf{T}_y$. Removed nodes should never be part of the merge result.

7. Nodes that are moved in $\mathbf{T}_x$ or $\mathbf{T}_y$ should appear moved in $\mathbf{T}_m$. The context of a node is used to merge this move.

8. Not all changes can be merged. There are several conflict situations.

These rules are the foundation of the algorithm: based on them decisions are made. The last rule is explained after discussing an example.

7.3.1 Example merge situation

An initial dictionary with *Persons*. Every person as a *Number*. The customers increase the length of this field to 15. DEVENT**it** moves the field to a new table *Phonenumbers*, which has a foreign key to persons. Let $\mathbf{T}_0$ be the common ancestor of $\mathbf{T}_u$, the current version of the user, and $\mathbf{T}_d$, the current of version of DEVENT**it** . Figure 7.1 shows the simplified XML of these dictionaries. These XML documents can be represented with PCS relations:

$$
\begin{aligned}
\mathbf{T}_0 = \{\ & c(a_0, \{type : element, name :\text{DICTIONARY}, attributes\{\}\}),\\
& c(b_0, \{type : element, name :\text{TABLES}, attributes\{\text{NAME: } Persons\}\}),\\
& c(c_0, \{type : element, name :\text{COLUMNS}, attributes\{\text{NAME: } Number,\text{LENGTH: } 10\}\}),\\
& pcs(\bot_0, \dashv, a_0), pcs(\bot_0, a_0, \vdash), pcs(a_0, \dashv, b_0), pcs(a_0, b_0, \vdash),\\
& pcs(b_0, \dashv, c_0), pcs(b_0, c_0, \vdash), pcs(c_0, \dashv, \vdash)\\
\}&
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{T}_u = \{\ & c(a_u, \{type : element, name :\text{DICTIONARY}, attributes\{\}\}),\\
& c(b_u, \{type : element, name :\text{TABLES}, attributes\{\text{NAME: } Persons\}\}),\\
& c(c_u, \{type : element, name :\text{COLUMNS}, attributes\{\text{NAME: } Number,\text{LENGTH: } 15\}\}),\\
& pcs(\bot_u, \dashv, a_u), pcs(\bot_u, a_u, \vdash), pcs(a_u, \dashv, b_u), pcs(a_u, b_u, \vdash),\\
& pcs(b_u, \dashv, c_u), pcs(b_u, c_u, \vdash), pcs(c_u, \dashv, \vdash)\\
\}&
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{T}_d = \{\ & c(a_d, \{type : element, name :\text{DICTIONARY}, attributes\{\}\}),\\
& c(b_d, \{type : element, name :\text{TABLES}, attributes\{\text{NAME: } Persons\}\}),\\
& c(d_d, \{type : element, name :\text{TABLES}, attributes\{\text{NAME: } Phonenumbers\}\}),\\
& c(c_d, \{type : element, name :\text{COLUMNS}, attributes\{\text{NAME: } Number,\text{LENGTH: } 10\}\}),\\
& c(e_d, \{type : element, name :\text{RELATIONS}, attributes\{\text{TABLE1: } Persons,\text{TABLE2: } Phonumbers\\
& \qquad\qquad ,\text{MULTIPLICITY: } 1tomany\}\}),\\
& pcs(\bot_d, \dashv, a_d), pcs(\bot_d, a_d, \vdash), pcs(a_d, \dashv, b_d), pcs(a_d, b_d, d_d), pcs(a_d, d_d, e_d), pcs(a_d, e_d, \vdash),\\
& pcs(b_d, \dashv, \vdash), pcs(d_d, \dashv, c_d), pcs(d_d, c_d, \vdash), pcs(c_d, \dashv, \vdash), pcs(e_d, \dashv, \vdash)\\
\}&
\end{aligned}
$$

Rule number four dictates that the update of node c in $\mathbf{T}_u$ should occur in $\mathbf{T}_m$. Nodes d_d and e_d should also occur in $\mathbf{T}_m$, because they are new (rule number five). The nodes should occur in the same context as the one they occur in in $\mathbf{T}_d$. Because the context of d and e should be retained (rule two) and because of the movement of c (rule seven), c should occur as a child of d in $\mathbf{T}_m$. The result of the merge is shown in Figure 7.2 and in the following definition:

$\mathbf{T}_0$ =
```
<DICTIONARY>
    <TABLES NAME="Persons">
        <COLUMNS ... LENGTH="10" NAME="Number" />
    </TABLES>
</DICTIONARY>
```

$\mathbf{T}_u$ =
```
<DICTIONARY>
    <TABLES NAME="Persons">
        <COLUMNS ... LENGTH="15" NAME="Number" />
    </TABLES>
</DICTIONARY>
```

$\mathbf{T}_d$ =
```
<DICTIONARY>
    <TABLES NAME="Persons" />
    <TABLES NAME="Phonenumbers">
        <COLUMNS ... LENGTH="10" NAME="Number" />
    </TABLES>
    <RELATIONS table1="Persons" table2="Phonumbers" multiplicity="1tomany" />
</DICTIONARY>
```

Figure 7.1: The three dictionary versions: 0, u, and d

$$
\begin{aligned}
\mathbf{T}_m = \{\ & c(a_d, \{type : element, name : \text{DICTIONARY}, attributes\{\}\}),\\
& c(b_d, \{type : element, name : \text{TABLES}, attributes\{\text{NAME: } Persons\}\}),\\
& c(d_d, \{type : element, name : \text{TABLES}, attributes\{\text{NAME: } Phonenumbers\}\}),\\
& c(c_d, \{type : element, name : \text{COLUMNS}, attributes\{\text{NAME: } Number, \text{LENGTH: } 15\}\}),\\
& c(e_m, \{type : element, name : \text{RELATIONS}, attributes\{\text{TABLE1: } Persons, \text{TABLE2: } Phonumbers\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{,MULTIPLICITY: } 1tomany\}\}),\\
& pcs(\bot_m, \dashv, a_m), pcs(\bot_m, a_m, \vdash), pcs(a_m, \dashv, b_m), pcs(a_m, b_m, d_m), pcs(a_m, d_m, e_m), pcs(a_m, e_m, \vdash),\\
& pcs(b_m, \dashv, \vdash), pcs(d_m, \dashv, c_m), pcs(d_m, c_m, \vdash), pcs(c_m, \dashv, \vdash), pcs(e_m, \dashv, \vdash)\\
\}&
\end{aligned}
$$

$\mathbf{T}_m$ =
```
<DICTIONARY>
    <TABLES NAME="Persons" />
    <TABLES NAME="Phonenumbers">
        <COLUMNS ... LENGTH="15" NAME="Number" />
    </TABLES>
    <RELATIONS table1="Persons" table2="Phonumbers" multiplicity="1tomany" />
</DICTIONARY>
```

Figure 7.2: The result of merging versions u and d

7.3.2 Conflict resolving

It is clear that in the result of the merge, $\mathbf{T}_m$, the phonenumber field is moved to a new table and the length is increased. Edits from both $\mathbf{T}_u$ and $\mathbf{T}_d$ are combined into a new dictionary. This example is one of the many examples that can be handled correctly. However, there are also situations that cannot be merged unambiguously into a new version. In these situations, human intervention is necessary. Lindholm identifies three kind of conflicts:

- **update - update** In this situation a node n is updated in both branches. Such an update often cannot be merged. For instance the phonenumber column from the example. If both branches update the length of that column to different values, merging becomes impossible. A choice has to be made about which modification is kept.

- **position - position** Another conflict situation is when a node is repositioned in both branches. In the case of dictionaries, this can only happen when columns are moved to other tables. The XML merge algorithm also sees a conflict when the order of child nodes is different, but this is not an issue in the dictionaries. The ordering of the columns or tables is not important.

- **delete - edit** Last conflict situation arises when a part of the dictionary is removed while the other branch updates that part. Such cases are hard to deal with in an automated manner.

These conflict situations can be solved by human intervention or by using extra rules, heuristics. Possible heuristics that can be used are for instance heuristics that always choose the DEVENT**it** modification. In some cases this heuristic might be correct, in order cases not. The users cannot make this decision, because they lack the knowledge about the DEVENT**it** modifications. These modifications might be used in the business logic of the updated applications. However, the process needs to be automated otherwise it becomes an exhaustive task. Therefore, the delta between $\mathbf{T}_0$ and the DEVENT**it** branch is annotated. These annotations mark every change as mandatory or optional. When a conflict arises, these annotations can be used to solve them without human intervention. When the change is optional, the users version is kept. When the change is mandatory, the DEVENT**it** version is kept. Afterwards, a summary of the conflicts and their solutions is presented. This makes it possible to update an application without human intervention. Of course the developers have to annotate the delta, but they only have to do it once for all customers.

Although not all situations can be merged in an automated fashion, conflict situations can be solved with the application of some basic rules. Annotating changes results in more work for developers, but the merge process gains a lot from it. Conflict situations can be solved without human intervention, making the merge process completely automated.

After the merge process, the dictionary and the database schema need to be synchronized again. This can be done by analyzing the changes between the merge result and the previous current version. This delta is then used to synchronize the dictionary with the database. The delta is split into actions, the actions are ordered and executed. As a result, the merge version is part of the history and can thus be undone.

However, in the case of an application update, some dictionary changes might be done to support the applications logic. Therefore, the new version can be marked as a special kind of version, a stable point. The dictionary then cannot be undone to a version before that stable point. Because of the new application version, the dictionary starts all over with the current version as root. Existing branches are locked and can only be reviewed. This of course means that customers have to plan their upgrade carefully, it might be that the new version is not backwards compatible.

7.4 Summary

Two situations exists that need a merge algorithm for dictionaries: branches in the version control system and the update of a *dynamic* ADD.NET application. The merge process cannot depend on human intervention, because the users (probably) do not have the required knowledge.

One solution is to prevent concurrent modifications on a dictionary by both DEVENT**it** and customers. Merging different branches is then impossible. This cripples the version control system and the flexibility of the application. Dictionaries have to be copied in order to be mutable by customers, causing multiple dictionaries to exist, some of them not used. Modifications done by DEVENT**it** have to be merged in the dictionary by hand.

The second solution is an automated three-way merge algorithm for XML documents. This algorithm shows that it is possible to merge two branches without human intervention. Using heuristics and annotations makes it possible to solve conflicts without intervention.

With this algorithm, the *dynamic* ADD.NET framework features a strong and powerful version control system. It is user friendly, but still flexible. Deployment of new versions can be done in an automated way.

Chapter 8

Conclusion and future work

This thesis describes a framework that can use a dynamic description of database schema, and it is no longer tied to a specific static schema. Furthermore a template engine is added, such that webpages for CRUD functions can be defined. After that a development environment for dictionaries is described, featuring synchronization with a database schema, version control, and a merge algorithm.

This last chapter reviews and discusses the work done in this thesis.

8.1 Conclusion

The central research problem of this thesis, as stated in the introduction, is:

1. How can a framework for *archive management systems* be constructed that removes the tight, compile-time coupling between
   a. database schema and application logic, and
   b. database schema and the user interface?
2. How can non-technical people modify a database schema used in an application built with the framework as described by 1?

Two tight couplings have to be solved by the framework: the one between database schema and application, and the one between database schema and interface. The first coupling was almost solved by the ADD.NET framework. However, this framework is static: the dictionary is linked at compile-time. Chapter 3 describes the solution for this problem: an extension of the framework with a loading mechanism for dynamic dictionaries. These dictionaries are stored in a database for convenience. The framework implements both a reloading mechanism and a notify system. This combination makes sure that the framework does not use a out-dated, incorrect dictionary for the CRUD functions.

The second coupling is solved by the template engine as described in Chapter 4 and Appendix A. The templates define webpages that provide CRUD functionality on single documents from an archive management system. The templates consist of pure XHTML, JavaScript, and CSS which is extended with custom placeholders. These placeholders are used to insert input controls. The templates are basic, plain text files. This can be an advantage: they can be edited and controlled completely separated from the framework. However, the disadvantages outweigh the advantages in this case. The templates are not connected to the dictionary, making it impossible to detect which templates are in use, which templates might be out-dated because they are based on an old dictionary, and which templates are incorrect because they use wrong information.

The second part of the research problem calls for an application to manage the dictionary that is used in the run-time of the framework. This management application allows non-technical users to view and modify the dictionary, without breaking the application or leaving the database in an incorrect state. This is all provided by the proposed application, the *Dictionary Manager*. Chapter 5 describes an abstraction of the relational model, as used in the application, along with the central synchronization algorithm of the application. The synchronization of the dictionary with the database schema is explained, along with how to preserve data in the best possible way. Chapter 6 extends the basis by adding a delta-based version control system. This version control system gives the user insight in past modifications. It allows the reviewing of certain steps. It also supports undo and redo. Users can revert modifications done in the past, by switching off deltas. This gives users the possibility to undo certain actions, without the necessity to undo everything that was done after those actions. The version control system does allow branching, but only one dictionary can be the active one, because it needs to resemble the database schema. Chapter 7 describes a merge algorithm that allows the merging of two branches into one new version. Based on XML documents, this algorithm (along with some heuristics for conflict resolving) does not require any human intervention. The same algorithm can be used to update an application and merge dictionary changes made by DEVENT**it** into the dictionary of the customer.

Although this sounds very positive and successful, there is only so much one can do in twenty-two weeks. Much, however, needs to be validated with an implementation. The current implementation covers Chapters 3, 4, and 5. Furthermore, the first solution to version control, the separate copies as described in Chapter 6, is implemented. The parts that are left are the delta approach to versioning, which is far more flexible, branching and merging: important features! Appendices A and B give a more detailed description of the implementation. The first one gives a short overview of the template definition, while the second one shows the *Dictionary Manager*.

The framework and the management application together make a strong foundation for a generic, document independent, archive management system. Often the documents in archive management systems are defined by the developers, and any modification has to be done by them. However, the system as proposed in this thesis allows users to define their own documents, in an evolving manner. They can modify the structures such that they reflect their requirements, without the need to consult the developers.

One of the problems that kept coming back in this thesis was the user friendliness of the framework. Normal database schema management tools are aimed at developers. They offer a technical view along with detailed possibilities such as the direct input of SQL. The advantage of this is that one can assume a certain level of technical knowledge. The *dynamic* ADD.NET framework is aimed at applications for non-technical users. As a result, certain subjects that are easy for developers become difficult and need extra work. This is why the dictionary has an abstraction to high-level concepts.

Chapter 2 describes several use cases that show sequences of actions executed by the user. They form a small and informal functional description of the framework. The use cases can also be used to review the described framework. The first two use cases describe rather simple examples of the *Dictionary Manager* usage. They only use basic information. The third use case is more complex, it creates a whole new dictionary. This is of course possible. The generation of a basic template definition is not described in the thesis, but is straightforward. The fourth use case describes how the template engine can be used. The fifth and sixth use case make use of the version control system: reviewing history and using the merge algorithm.

8.2 Future work

There are a number of topics that require more exploration. Some are extensions, other topics are side steps of this thesis. First of all, the implementation. As said before, not all the work

is covered by an implementation. Furthermore, the work done in this thesis will be embedded in a much larger system: a web-based archive management system. This embedding has to be implemented. The system will then become the system that is projected in this thesis: a flexible and customizable archive management system that provides functionalities to manage, search, and present documents in any structure. However, focusing at the *dynamic* ADD.NET framework and the *Dictionary Manager*, there is also some work left.

8.2.1 Graphical user interface

One of the things that became clear when working on the version control system is the fact that the templates are a disjoint component of the system. Although they are important, they are not integrated with the other parts: there is no link between a dictionary and its templates. This makes the system flexible: templates can be written and used without configuration whatsoever. However, it also makes it hard to manage them. If the dictionary is modified and a new version is created, there is no way to tell which templates are influenced. Furthermore, the system could benefit from an impact analysis on the templates. Before a new dictionary version is committed, the system analyses the templates and shows what the impact of the modifications in the dictionary is.

The templates are not controlled by the version control system: undoing and redoing needs to be followed by a manual replacement of the templates. This makes it difficult to step through different dictionary versions: one has to remember to place the right templates, otherwise the application cannot function.

A possible solution would be to not use XML files, but to normalize the contents into a more structured model: maybe a database schema. The templates could then be related to the dictionary, and it would be clear which templates belong to which dictionary version. Templates could be stored in the version control system and the undo/redo facilities would manage the templates too. Furthermore, it might even be possible to update the templates more automated: react on field renames or type changes. The templates are then synchronized with the dictionary without human intervention. Chapter 4 gives other possibilities: integration with editors or a DSL on top of the templates.

Without another form of the templates, versioning them is still possible. The system could use different locations for every version. Updating the dictionary would then mean that the templates are copied to a new location. The templates can then be modified, without losing the previous templates.

8.2.2 *Dictionary Manager*

The *Dictionary Manager* itself also has room for improvements. They were already mentioned in Chapter 5: refactorings, multilingual support, and a transaction mechanism.

The refactorings would allow more complex edit operations to become data preserving. Moving a column to a new table, along with a relation to the old table, could become a data preserving operation if the *Dictionary Manager* would allow such an action. Possible other refactorings exist too, and adding them increases the usability of the *Dictionary Manager*. Refactorings are defined as actions that change the schema, while keeping the semantics the same. Adding a column is therefore not a refactoring, it extends the schema and changes the semantics.

The dictionary has only room for one set of texts per field. It is thus not possible to create multilingual applications based on the *dynamic* ADD.NET framework. The database schema has to be extended in order to support multiple languages.

The synchronization of the dictionary with the database happens by executing multiple DDL

and DML statements. Modern DBMSs do not allow the combination of those statements in one transaction. If the synchronization would fail somewhere in the middle, the application is left in an incorrect state. To prevent this, the framework should use a transaction mechanism. This mechanism can rollback partial synchronizations, preventing an inconsistent state.

8.2.3 Development environment

Another road that has not been explored is the creation of a development environment out of the *Dictionary Manager*. The thesis hinted at such an environment, but it takes more than described in this thesis. The *Dictionary Manager* in its current state is obvious embedded in the context of a *dynamic* ADD.NET application. It enables users to modify a particular dictionary in a safe way. The features are similar to a development environment, but the use of the manager without an application is not beneficial.

However, one could extend the manager into a stand-alone, web-based database schema manager. One of the drawbacks of the ADDWin application is the fact that it does not generate an update script for databases in production. They have to be written by hand after an update. The *Dictionary Manager* does have this possibility: it updates databases in production. In order to use the manager as a development environment, it should be possible to work off-line or disconnected. Instead of synchronizing the dictionary with the database after every change, an update script is generated. Such a script can update a database in production, without the need of installing the manager.

The manager in off-line modus takes the role of a central repository of all schemas in use in different application, much like the ADDWin application. However, it has also all advantages of web-based application: a central installation and update point and reachable from every client. The features such as branching and undo/ redo (by means of switching deltas on and off) especially comes in very handy, because it encourages experimentations. Developers can modify the dictionary in a modular fashion and can then easily combine different features into a single dictionary.

Projects and products that aim at the same goal are LiquiBase [LiquiBase] and the migrations in Ruby On Rails [Ruby On Rails]. The first one, LiquiBase, offers a management environment in which changes are stored, from which migration scripts can be generated, and which guides the migration process. The second one is a possibility to describe database schema changes in terms of migration steps (actually just deltas). Every step is described as an *up* and *down*, making it possible to undo and redo. The first one supports branching and merging, the second lacks that kind of support.

Bibliography

D. L. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A Domain-Specific Language for Form-Based Services. *IEEE Transactions on Software Engineering*, 25(3):334–346, 1999.

C. D. Castro, F. Grandi, and M. R. Scalas. Schema versioning for multitemporal relational databases. *Information Systems*, 22(5):249–290, 1997.

H.-T. Chou and W. Kim. Versions and change notification in an object-oriented database system. In *DAC '88: Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 275–281, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13 (6):377–387, June 1970.

P. Dadam, V. Y. Lum, and H.-D. Werner. Integration of time versions into a relational database system. In *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 509–522, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

Darcs.
http://www.darcs.net/.

DEVENTit.
http://www.deventit.nl.

FormFaces.
http://www.formfaces.com/.

T. Horstmann and R. Bentley. Distributed authoring on the Web with the BSCW shared workspace system. *StandardView*, 5(1):9–16, 1997.

A. Karahasanovic and D. I. K. Sjøberg. Visualizing impacts of database schema changes - a controlled experiment. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments*, page 358, Washington, DC, USA, 2001. IEEE Computer Society.

W. Kim and H.-T. Chou. Versions of schema for object-oriented databases. In *VLDB '88: Proceedings of the 14th International Conference on Very Large Data Bases*, pages 148–159, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

V. Kok. ADD.NET, *Bachelor's thesis, Hogeschool Utrecht*, 2004.

B. P. Lientz. Issues in software maintenance. *ACM Computing Surveys*, 15(3):271–278, 1983.

T. Lindholm. A three-way merge for XML documents. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 1–10, New York, NY, USA, 2004. ACM Press.

LiquiBase.
http://www.liquibase.org/.

T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

Microsoft NET.
http://www.microsoft.com/net/.

M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004. ISBN 0596004486.

R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000. ISBN 0072465352.

M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4): 364–370, 1975.

J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

Ruby On Rails.
http://www.rubyonrails.org/.

D. I. K. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.

The Apache Velocity Project.
http://velocity.apache.org/.

The Forms Working Groups.
http://www.w3.org/MarkUp/Forms/.

W. F. Tichy. Design, implementation, and evaluation of a revision control system. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 58–67, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

H.-C. Wei and R. Elmasri. Schema versioning and database conversion techniques for bi-temporal databases. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):23–52, 2000.

XHTML 1.0 in XML schema.
http://www.w3.org/TR/xhtml1-schema/#xhtml1-strict.

L. Yu and D. J. Rosenkrantz. Minimizing time-space cost for database version control. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 294–301, New York, NY, USA, 1988. ACM Press.

Appendix A

Template engine

This appendix is a short manual on the template engine of the *dynamic* ADD.NET framework. It explains the working and how it can be used. The example given in Chapter 1 is used.

A.1 Introduction

The templates that are used within the *dynamic* ADD.NET framework are described with an XML schema definition. This schema is used to validate templates. These templates can thus also be validated by a standard XML editor.

The basis of the template schema is the XHTML schema [XHTML 1.0 in XML schema]. The schema definition is a W3C note, and can only be used informative. However, it gives a good approximation of the XHTML that modern browsers accept. It is not guaranteed that the output of the system is XHTML valid according to the W3C validators, because they use the Doctype definition for validation. Because of the XML schema extension mechanism and the relative small extension of the templates, updating to a new XHTML schema definition is no problem.

A.2 Fields and fieldgroups

The schema starts with defining the possible locations of the custom placeholders, shown in Figure A.1. The placeholders may occur at the *block* level, the same level for elements such as `div`, `p`, and `h1`. This level is used for content elements. It also ensures that the placeholders can only be used inside the body of the webpage and not in the header.

Two types of placeholders are defined: *field* and *fieldgroup*, shown in Figure A.2. The first matches a single column, non-repeatable. This can for instance be used to present the certificate date as shown in Figure A.3. The second element is used to present tabular data in a repeatable way. A number of field elements can be used to specify the columns of a table. An example usage is the list of persons and their role (again from the certificate example) as shown in Figure A.4. This tabular presentation allows the insertion of new rows and the editing and removal of existing rows.

Attributes influence the presentation and functionality of the elements. Three attributes are highlighted.

```
<xs:redefine schemaLocation="xhtml.xsd">
  <xs:group name="block">
    <xs:choice>
      <xs:group ref="block"/>
      <xs:element ref="groupfield"/>
      <xs:element ref="field"/>
    </xs:choice>
  </xs:group>
</xs:redefine>
```

Figure A.1: Definition of possible placeholder-locations

```
<xs:element name="field">
  <xs:complexType>
    <xs:attribute name="id" type="xs:string" use="required"/>
    <xs:attribute name="type" type="fieldtype" default="input"/>
    <xs:attribute name="control" type="controltype"/>
  </xs:complexType>
</xs:element>

<xs:element name="fieldgroup">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="field"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

Figure A.2: Definition of placeholders

```
<div>
    <field id="ADD_CERTIFICATES_DATE" type="header"/>:
    <field id="ADD_CERTIFICATES_DATE" />
</div>
```

Figure A.3: Example of the field element

```
<div>
    <fieldgroup id="ADD_PERSONSANDROLES">
        <field id="ADD_PERSONS_NAME"/>
        <field id="ADD_ROLES_ROLE"/>
    </fieldgroup>
</div>
```

Figure A.4: Example of the fieldgroup element

- The two types of placeholders both share one attribute, the *id* attribute. This attribute binds a placeholder to a column (field) or table (fieldgroup). The group element does not require an id, because the child elements already depict the contents of the control. However, if no child elements are specified, the id is used to create a default set of columns for the grouping element.
- The field element can have more attributes, and they can influence the presentation of the control. The *type* allows the user to specify a different usage of the placeholder. The default for every field is input, but a user might want to display other information such as the header text, the help message or the error message. Note that this attribute is not valid for field elements nested in a fieldgroup element.
- The *control* attribute allows the specification of a different control type than the default. Every field has a type such as string, date, number, time or boolean. These types all have a default input control such as text field, date field or check box. However, sometimes a non-default control is needed. A field of type string might only contain three values. The user can then specify that the control type should be a list of choices and supply the valid choices.

A.3 Actions

Defining fields is not enough for an webpage that offers CRUD functionality. A number of actions can be executed. A person editing data should be able to save his modifications, he should be able to create a new blank document, or delete a document. These actions should be specified somewhere in the template. They can for instance be executed when a button or an hyperlink is clicked. The template engine allows this by supplying a custom attribute *action*. This attribute can be added to standard HTML elements such as `input`, `a`, or `img` as shown in Figure A.5. The system will transform this attribute to a piece of JavaScript that will cause the appropriate action to be called.

```
<input type="button" value="save document" action="save"/>

<a href="#" action="new" title="new document">new</a>

<img src="delete.png" alt="delete document" action="delete" />
```

Figure A.5: Example of the action attribute

Appendix B

Dictionary Manager

The most prominent part of the implementation is the *Dictionary Manager*. This web application is only linked to the framework because they work with the same dictionary. Other than that, it is stand-alone. Of course without an application that uses the dictionary, the *Dictionary Manager* is pretty useless (in its current state, Chapter 8 explains how it could be useful in the future). This appendix gives an introduction to the application. The current implementation of the *Dictionary Manager* features the edit page for the current dictionary, version control using separate copies, and a history overview with undo and redo.

Note: the texts in the application are in a mixture of Dutch and English. The application will be Dutch at first, but some texts are in English because of this thesis. As becomes obvious: the current implementation is a prototype, evidence that it can be done. The implementation is far from ready. However, it gives an impression of what it will be.

The entry page gives an overview of the present dictionaries, as shown in Figure B.1. In the final Atlantis product one might see here a list of present document types such as photos and certificates. The two icons next to the name and description give access to the history overview and the edit page.

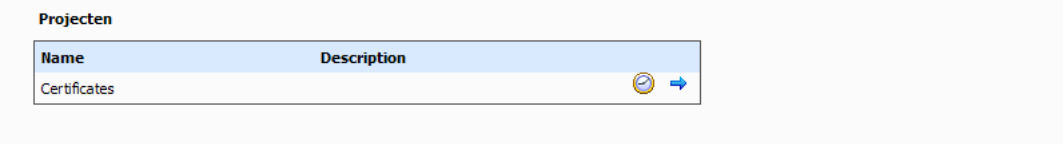

Figure B.1: Entry page: overview of present dictionaries

The edit page shows the current version of a dictionary, see Figure B.2. It shows what the version number is, and when it was created. The left side shows an overview of the entities that are present in the dictionary. Selecting one shows the details on the right side. The name of the entity and its description appear first. Then a list of the fields of the selected entity is presented. As explained in the thesis some fields are hidden. The collection of fields can be modified by using the icons: the document creates a new field, the pencil edits a field, and the trash can removes a field. Creating a new field and editing an existing field are done through the same form (shown in Figure B.3). It shows the different attributes of a field along with an input field that resembles the possible values.

Below the fields is a list with relations. A relation is represented by the name of another entity, and two check-boxes. The two boxes represent the multiplicity of the relation. The first one

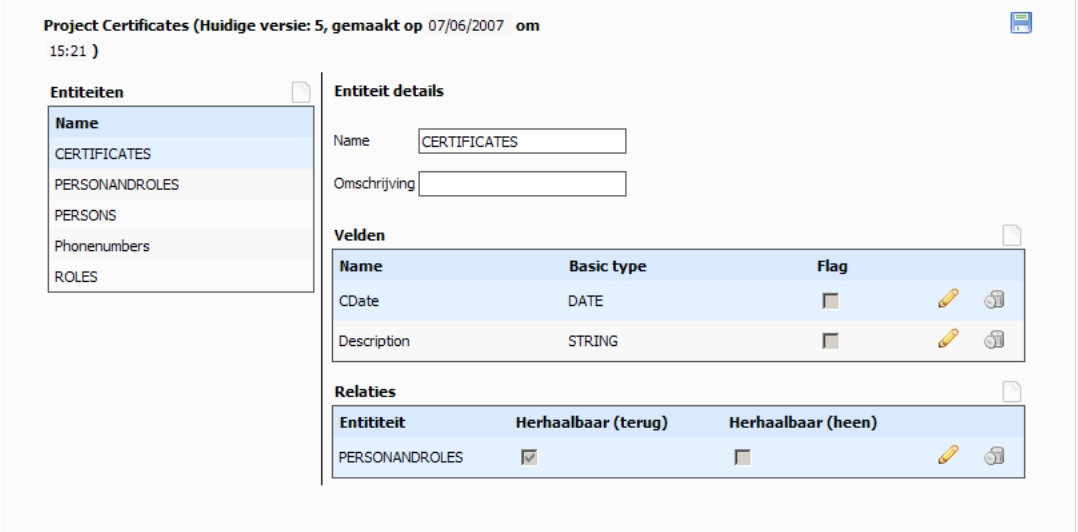

Figure B.2: Details of the current version in editable form

shows if it is a many to something relation. And the other shows if it is a something to many relation. What is missing from this form is the possibility to mark a relation as required.

Saving the dictionary causes the synchronization algorithm as described in Chapter 5 to be executed. The abstraction is translated to the dictionary model and the changes are collected. After that part, the dictionary is stored as a new copy along with the version number and creation date.

The history overview of a dictionary (see Figure B.4) shows a list of versions: the number, the date and time, and a description of what is changed in that version. The tick mark icon in the first column identifies the current version. The icons in the last four columns are for other purposes. The first icon shows the details of that particular version in terms of DDL and DML statements. It is there solely for debugging purposes. The second icon, the capital *T*, gives a textual representation of a particular dictionary version, again solely for debugging purposes. The third icon allows to mark another version as the current version. If that version is older than the current version, versions are undone. If that version is newer, versions have to be redone. The last icon

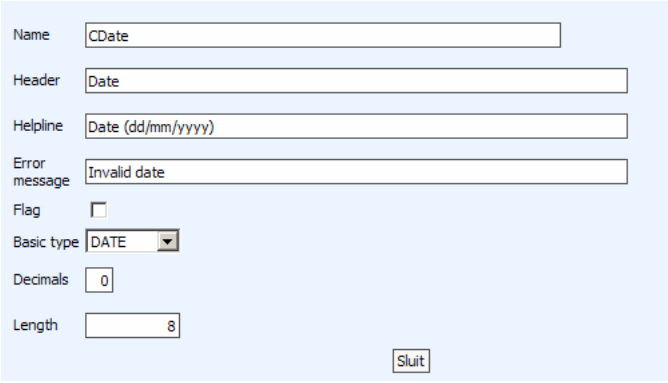

Figure B.3: Form with details of a single field

shows the dictionary in the form of the edit page, but if the version is different from the current version it is read only. The two lists and the button below the version table are used for the diff viewing. The diff is currently a line based diff that uses the textual representation of a dictionary.

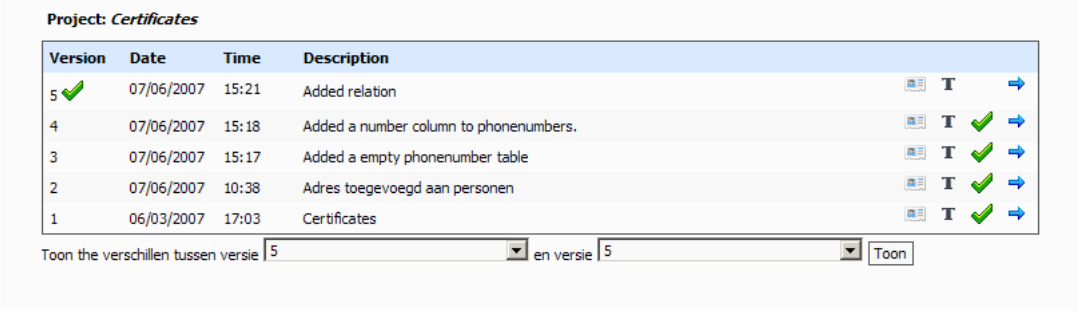

Figure B.4: History overview of a dictionary

The history overview currently does not support branching, and uses separate copies. The optimal solution as described in Chapter 6 uses deltas and allows branching. That approach needs a completly different user interface. One possibility is to present the history as a tree (showing branches in a intuitive manner). The deltas can then be presented as nodes in the tree, shown along with a check box. The user can easily switch deltas on and off in that way.