Miguel Bispo Alves

Licenciado em Engenharia Informática

Integrated Data model and DSL modifications

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadores : João Costa Seco, Prof. Doutor,
Universidade Nova de Lisboa
Lúcio Ferrão, Chief Architect,
Outsystems

Júri:

Presidente:	Prof. Dr. Nuno Manuel Ribeiro Preguiça
Arguente:	Prof. Dr. Francisco Cipriano da Cunha Martins
Vogal:	Prof. Dr. João Costa Seco

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2013

Integrated Data model and DSL modifications

To my father Carlos,
my mother Rita,
my aunts Isabel and Regina,
and to my love Heidi.
I owe you more than I can ever say.

Acknowledgements

This dissertation is the result of the last five year dedication and hard work, aiming to achieve a dream. That would not be possible without the contribute of many people that I would like to thank. I hope I have remembered everyone.

I would like to thank to my advisors, João Costa Seco and Lucio Ferrão. They provided me guidance and support at key moments in my work. Their careful review of several versions of this manuscript improved the quality of this dissertation. Without them this work would not have been possible. To António Melo, that gave the opportunity to do this thesis in a collaboration between the Faculdade de Ciências e Tecnologias of Universidade Nova de Lisboa (FCT-UNL) and the R & D team of OutSystems. I also ankowledge FCT-UNL for giving me work conditions and partial financial support.

I am also thankful to Hugo Lourenço, Luís Lopes and many other R & D team elements for their availability to help whenever was possible along the project.

And because the last five years were not made only of hard work, thanks to André Simões, Nuno Grade, Stefan Alves, Miguel Pinheiro, Sérgio Casca, Diogo Matos, Pedro Almeida and Tiago Almeida for all the great moments we spent during our academic journey.

To all my friends that always have supported me and showed their friendship. Pedro Marques, Álvaro Paulino, Luís Fernandes, João Paixão, Bruno Lameiras, Jorge Pinto, Diogo Valente, Pedro Costa, Ana Afonso, Rafaela Proença and many others: Thank You !

A special thanks for my parents, Carlos and Rita, because any of this would not be possible without them. I want to thank them for being always there to support me if I need, for their constant love and immeasurable sacrifice. A special thanks to my father, because I know that you are proud of this, wherever you are.

I also would like to thank to my grandparents, my aunts Isabel, Regina and

Sandra, my uncle Filipe, my cousins Diogo and Martim for their support.

All errors and limitations remaining in this thesis are mine alone.

Abstract

Companies are increasingly more and more dependent on distributed web-based software systems to support their businesses. This increases the need to maintain and extend software systems with up-to-date new features. Thus, the development process to introduce new features usually needs to be swift and agile, and the supporting software evolution process needs to be safe, fast, and efficient.

However, this is usually a difficult and challenging task for a developer due to the lack of support offered by programming environments, frameworks, and database management systems. Changes needed at the code level, database model, and the actual data contained in the database must be planned and developed together and executed in a synchronized way.

Even under a careful development discipline, the impact of changing an application data model is hard to predict. The lifetime of an application comprises changes and updates designed and tested using data, which is usually far from the real, production, data. So, coding DDL and DML SQL scripts to update database schema and data, is the usual (and hard) approach taken by developers. Such manual approach is error prone and disconnected from the real data in production, because developers may not know the exact impact of their changes.

This work aims to improve the maintenance process in the context of *Agile Platform* by *Outsystems*. Our goal is to design and implement new data-model evolution features that ensure a safe support for change and a sound migration process. Our solution includes impact analysis mechanisms targeting the data model and the data itself. This provides, to developers, a safe, simple, and guided evolution process.

Keywords: DSLs, Database Refactoring, Database migrations

Resumo

No dias de hoje, as empresas estão cada vez mais dependentes de aplicações centralizadas para sustentar os seus negócios. Devido a esse facto, é necessário evoluir os sistemas de software adicionando-lhe novas funcionaliades, de forma a mante-los actualizados. Portanto, esse processo de desenvolvimento tem de ser rápido e ágil, sendo que o suporte de evolução desses sistemas tem também que ser seguro, rápido e eficiente.

No entanto, é normalmente difícil e desafiante para os programadores, pois os ambientes de programação, plataformas e sistems de suporte de bases de dados, não oferecem o devido suporte. As alterações necessárias no código, base de dados e nos dados existentes, têm que ser planeadas, desenvolvidas e aplicadas de uma forma sincronizada.

Mesmo seguindo rigorosos critérios no desenvolvimento de aplicações, é difícil de prever o impacto das mudanças efectuadas ao seu modelo de dados. O ciclo de vida de uma aplicação envolve mudanças e actualizações, desenhadas e testadas em dados que estão longe de ser os dados reais em produção.

A abordagem mais utilizada por programadores para evoluir e actualizar o modelo de dados é o desenvolvimento manual de scripts SQL. Essa abordagem é propicia a erros e não está em sintonia com os dados reais existentes em produção.

Esta dissertação tem como objectivo melhorar o processo de manutenção e desenvolvimento de aplicação no contexto da *Agile Platform* da *OutSystems*. O nosso objecto é desenhar e implementar novas funcionalidades que permitam evoluir o modelo de dados das aplicações e que esse processo seja seguro e não coloque em causa os dados em produção. O nosso modelo da solução inlcui mecanismos de

analise do impacto das mudanças ao modelo de dados das aplicações em produção e nos dados existentes nas mesmas. Pretende-se fornecer aos programadores um processo de evolução das aplicações seguro, simples e guiado.

Palavras-chave: DSLs, Refactorização de Bases de Dados , Gestão de mudança de Bases de Dados, Migrações de Bases de Dados

Contents

List of Figures

Listings

1

Introduction

This work aims to design and implement software evolution features in the *Outsystems Agile Platform*. We focus on the modification operations targeting the database layer of web applications, by adding database evolution features to the *Service Studio*, the development environment of the *Agile Platform*, in order to help the developers to change applications database model in a safer and sound way and to reduce the development teams effort.

Enterprise applications are continuously evolving to be up-to-date to contextual changes. In software maintenance in large and long-lived applications, the most challenging changes derive from the evolution of the database schema of an application, as well as the associated changes in code and data. Most web applications are centralized around a database (or the opposite), where the structure of data is represented through a database schema. Thus, changes to the database model have significant impact on the rest of the application components [Sjo93], requiring adequate adaptation to occur. During the maintenance phase of a project, the developer teams face problems when trying, to evolve the database model. Changing is difficult for several reasons. The missing access to data in production, hence it is difficult to predict the impact of those changes, the cost and the risk to execute the desired changes on the data model can be very high, since it can decrease the performance and the integrity of the application, and they must change model data and code. Besides, database models are usually shared by several applications, which requires a very broad impact analysis.

As an example, consider a developer doing maintenance tasks of an enterprise

application with a few hundred tables, some of those tables with millions of rows. He needs to evolve the database model by replacing a foreign key column from *user* foreign key to *user_master*. The process would require manually: *1)* column rename from *userId to user_masterId*; *2)* Fix the application code referencing the old attribute; *3)* Validate the new attribute type in all applications sharing the database model; *4)* Fill the new column defining a rule based on the old attribute values, assuming that is possible to convert *user* values to *user_master* values. The usual approach to accomplish such operation is to write SQL scripts, applying it on the database schema and data, change the code by hand, and deploy a new version of the application. The problem is that typically developers do not know the impact of the database change operations nor its risk on the database model of the application already in production. Due to this fact developers try to start with the more complete possible, and flexible, database model, and then change as less as possible the database model during maintenance operations. Running the scripts manually in the database can break data consistency and integrity, that may lead to system downtimes, as well as the errors within the application code. Application downtimes are not acceptable for every enterprise because it means profit loss.

While changing and evolving applications database model, developers do not know the impact of those changes on the real data in production. When an operator wants to deploy the application to the production environment, does not have any information about what changes are going to be applied to the database model. If the operator gets some error the deployment needs to be aborted, he needs to ask to the developer teams to fix those errors and the deployment of the applications needs to be postponed.

So, this problem provides fertile ground to new developments that improve the life of developers in this kind of scenarios. On this dissertation, we exercise a solution in the context of the *Agile Platform*, an integrated development environment that includes the database modelling, the application modelling and a connection to both development and production environments. Thus, the main motivation for this thesis is to implement database model evolution features within the *Outsystems Agile Platform*, to allow the evolution of the database model and its data, that alert to necessary data fixes in the production database model, and provide an efficient, safe and smooth evolution process, requiring the minimum development effort and maintaining the same data migration as before.

1.1 Problem and Goals

The process requires to modify the database by upgrading its schema, migrating the data and adapt the programs accessing to the database model to work with the new changed version. To fully support database modifications is a major and unrealistic endeavour. Thus, we need at first to identify which features are most needed, and which solutions are more effective in this context. So, in order to build a solution reducing the risk and the effort of developers when evolving applications database model we try to answer the following questions:

- Which evolution features give us a greater gain?
- In which steps should the evolution process be divided in?
- How can we measure the impact and cost of changes?
- How and to what extent should the data migration be automatic?
- How and to what extent should be code automatically adapted ?

At this moment, the *Outystems Agile Platform* does not provide mechanisms to allow the developer to automatically change the database model and to migrate data. However, it provides a tool called *LifeTime*, having full visibility of all applications across all the environments, where developers can see the exact version of each application running in the development and production environments.

The main goal of this work is to implement evolution features within *Outsystems Agile Platform*. Our approach relies on the concept of migrations. Migrations are generated when developers changes applications database model. Then, through LifeTime the system analyzes the impact of those changes in the real data in production and presents warnings to developers in *Service Studio* if the data needs to be fixed.

Our approach is to bring the necessary information related to the database, from the production environment to the development environment, making it available to developers when evolving the database model. Afterwards, taking into account the information presented developers may write upgrade rules in order to fix the bad data. The process can be performed by measuring the impact of the transformation made to the database model and by warning the user about the impact of such change within the database model and its data. For example, changing the *IsMandatory* property of an attribute may srequires all rows in the database having some value. Thus, in our approach, developers may define a default value or write an expression to fix rows having NULL value.

At this moment, developers need to run manual SQL scripts against the database model to execute changes, which is more difficult than an automatic tool to perform those changes because developers do not know the impact of the operations specified within the scripts.

For example, some operations need to be preceded by other ones in order to be executed. So, if the tools help and advise developers about the impact of their changes in applications database model, integrity and consistency errors may be avoided.

That said, this project aims:

- to enrich the *Service Studio* with new database model change features;
- to enrich the deployment support system to support the model changes and data migration;
- to build an impact analysis tool to give feedback about the data in the production environment to the developer.
- to provide to developers safe, guided and simple evolution process

1.2 Approach

In our project we will implement new evolution features to the development environment, the *Service Studio*. We anticipate that the new features introduced will provide enough information to specify the transportation of relevant data from the production environment to the development environment. Developers are able to execute the operations in the development environment having the knowledge about the impact of such changes in the production environment database model. The impact is measured depending on whether there is data in the database of the production environment. If there is no data in the production environment the developer can change safely knowing that he will not loose any information. On the other hand and in most of cases, if exists data in the production environment database the developer should receive a warning and should be able to specify a rule in order to upgrade the information.

In the first phase of this thesis, we inquired some *Outsystems* developers to find out which are the most usual changes on the database models during the lifetime of an application. The results of the interviews, help us to decide which evolution features to implement in the *Agile Platform*. We also studied techniques

as a basis to the design and implement our solution, for example, database refactoring techniques [AS06], schema modification operators [CMZ08] and data migration approaches.

We aim to build a solution that improves the database model change process in the *Agile Platform*, helping the developers during the process, by performing changes to the database model and by calculating the impact of those both on data and code.

1.3 Outline

In this chapter we introduced the motivation for our work. After that, we presented the problem we are trying to solve, the goals we expect to reach in the end of this thesis and the approach we are planning to take.

Chapter 2 presents the context of the project, by referring the architecture and functionalities of the *Agile Platform*, and where our work fits on the platform.

Chapter 3 describes the approach for the first stages of the project. We analyze the interviews we did to developers to capture the goals, risks, cost and the most frequent and potential interesting change scenarios. We describe the interviews structure, followed by a discussion about the patterns that we extracted.

In chapter 4 we provide the state of art relevant for our work. We discuss about database refactoring, database migration, query adaptation, modification languages and change patterns, referring in which systems we can find those patterns and how they implement them.

In chapter 5 we describe in detail the solution model and its features. We present our database migrations based solution and they are integrated in the *Agile Platform*.

Chapter 6 explains how the prototype for our solution model was implemented and which features we were able to implement. We also describe the limitations of implementing our solution through the other *Agile Platform* components.

Finally, in Chapter 7 we describe and discuss the work done during this dissertation, as well as the future work, that could enrich this solution and the built prototype.

2

The Agile Platform

This project is being developed in the context of the R&D team of Outsystems, particularly in the context of the *Agile Platform*, an integrated platform to develop and model enterprise web applications. In this chapter we present the platform and its components, in order to know more deeply the environment where we want to implement and test our solution.

The *Agile Platform* offers to developers a fast and incremental way to build and maintain a complete web application. In Figure 2.1 is depicted the platform high level architecture, in the perspective of the software development process.

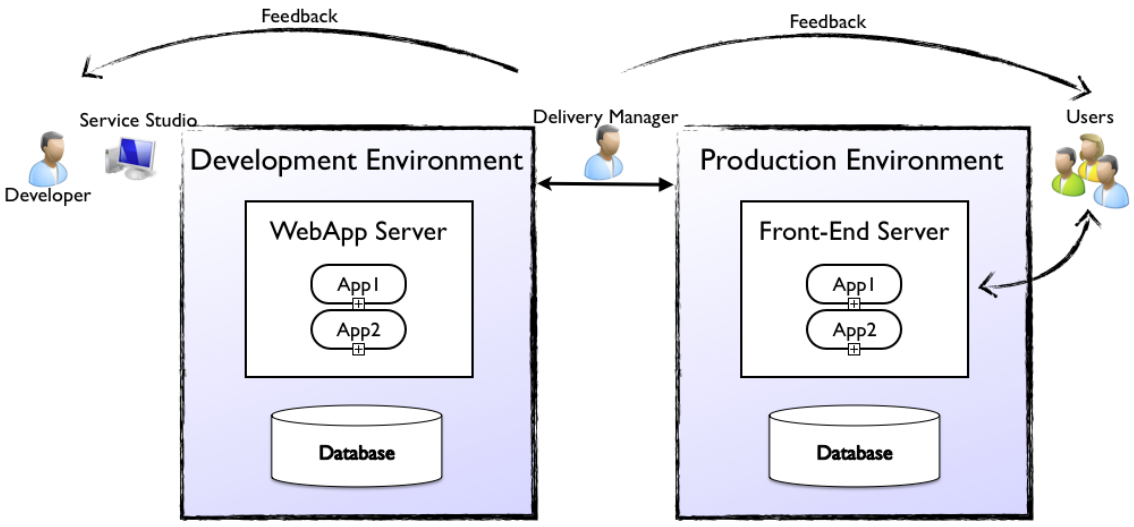

Figure 2.1: Agile Platform Architecture

The web applications are developed within *Service Studio*, which is development tool. *Service Studio* is an IDE based on a safe and easy to use visual Domain Specific Language (DSL), that covers the definition of business processes, user interfaces, business logic, and data definition and manipulation, web services, security, emails and scheduled jobs. After that, through the *1-Click Publishing* feature, applications are sent to the application server and the compiler located there generated the C#, Java, HTML, CSS, JavaScript, SQL code. This feature ends with the deploy process, an operation that updates the *eSpace* published version. Afterwards, the application is ready to be used in a web browser.

Our work is focused on equipping *Service Studio*, with new database refactoring operations directed towards the database model and to design and implement the required changes across the whole platform targeting a set of features. In the next section we describe *Service Studio*.

2.1 Service Studio

Service studio is a tool integrated in the *Outsystems Agile Platform* that supports the development of web applications available in the *Agile Platform* and allows the development of *eSpaces*. An *eSpace* is an OML (*OutSystems Markup Language*) file containing all definitions needed to develop and manage those applications, such as the application logic, database model, web pages interfaces, and security settings.

This tool implements a visual programming language, where screen, program, and process flows are represented by graphs. Hence, the applications elements are created though simple drag and drop actions, having the developer only to configure some properties.

Figure 2.2 depicts *Service Studio* interface. The most left part of the interface contains the elements the user adds to the action flows. The right part is divided by the elements tree, showing the elements within the *eSpace*. Below there is the properties panel where elements properties are edited. In the top part of the interface we have the navigation commands, as well as the *1 Click Publish* button. The middle part of the interface is the flow screen, where the user draws the application interfaces and defines the actions flow. The lower panel contains three tabs: 1) *TrueChange* where developer checks *eSpace* errors and warnings; 2) debugger where the developer checks the runtime behaviour of the application; 3) The *1-Click Publish* where the developer checks the publishing process status.

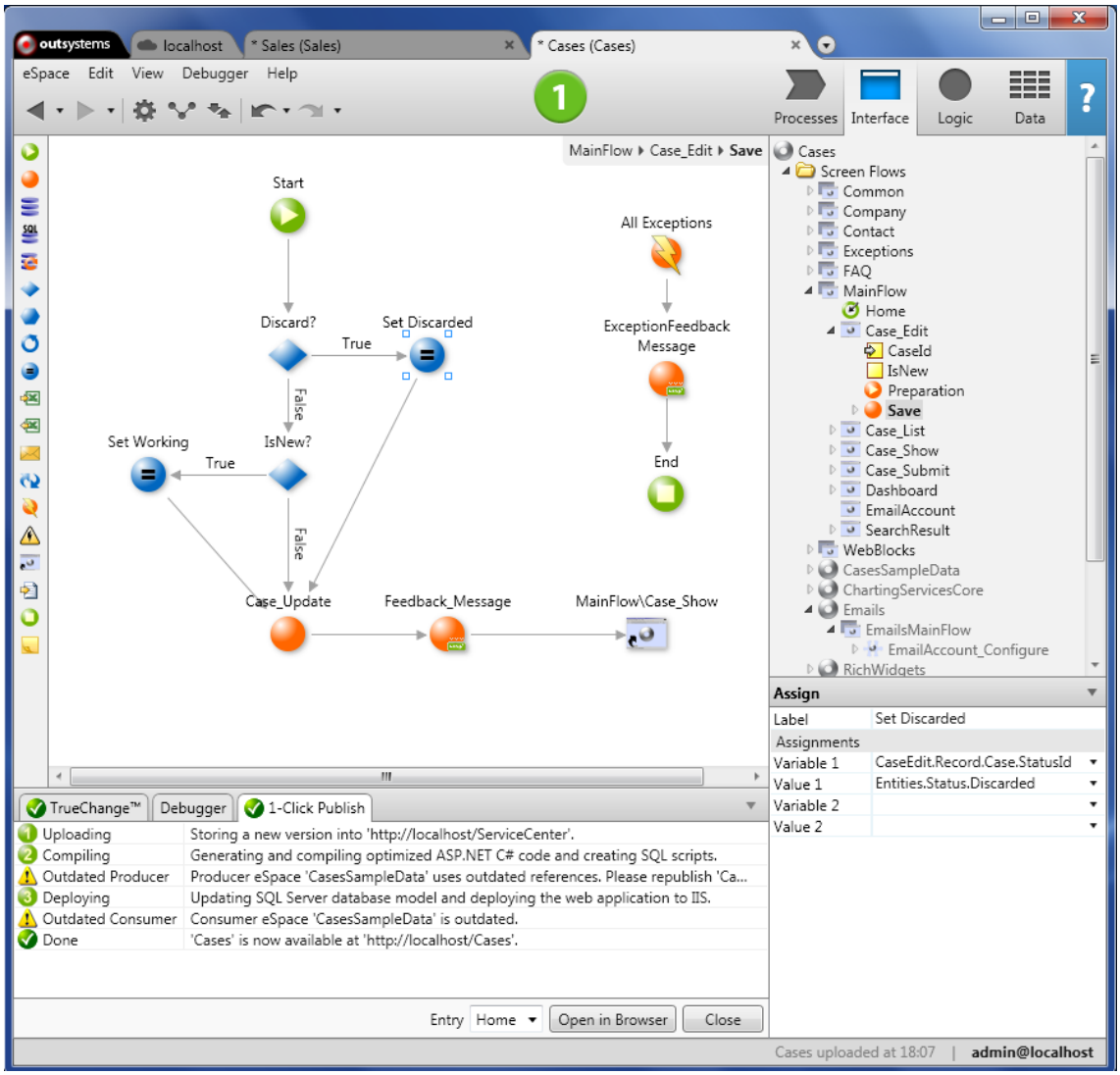

Figure 2.2: Service Studio - Outsystems applications development environment.

2.1.1 Data Model

In *Service Studio* we define the logic, interface and database model for the applications. Our work focus on the database model and in this section we explain how the developer can build it within *Service Studio.* Figure 2.3 depicts a *Service Studio* screen showing the created Entity Diagram for an application.

As we observe in Figure 2.3, the application has defined an Entity-relationship diagram [Che76]. We have two types of Entities: *Entity* and *Static Entity*. An *Entity* is an element which allows the developer to keep business information in a persistent way. Entities are used to represent and manage the database model. A *Static Entity* is an entity that has static data associated to it, The static data is then managed in design time and can be used directly in the business logic design.

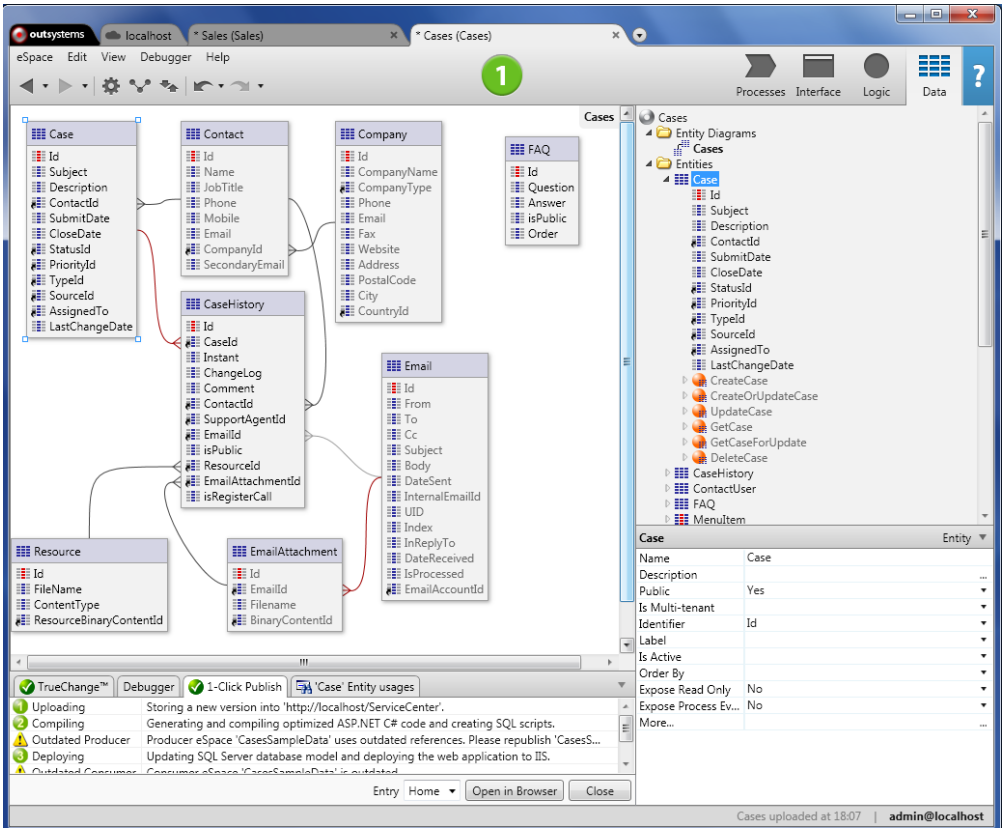

Figure 2.3: Service Studio - Application Data Model

Good examples of static data implementation are constants and enumerations. In the right side of the tool we can add new Entities and in order to visualise them in the diagram, the users need to drag and drop the Entity they want. To edit an *Entity* the user clicks on the respective one, and then edits the properties associated to it. To add new attributes to an Entity the user selects the Entity and then he adds new attributes to it. To check where the *Entity* is being used the user needs to select the *Find Usages* feature. The information is presented below within the tab "*Application_Name* Entity usage".

The structures defined by the tables in the database model are used not only in the database queries, but also in the application logic and interface.

2.2 Programming language

A Domain Specific languages (DSL) is a programming language that is focused, through appropriate notations and abstractions, on a particular problem domain. Due to its focus on a specific domain, a DSL allows a very efficient application development process, since the development is not focused on implementation details. Examples of its usage are programming languages for robots, graphic environments definition or physics simulations. Some web developing tool like *Agile platform*, implement a DSL. The tool interacts with the system components though simple constructions, making easier the communication with data repositories, the manipulation of its data, and the interface with the user.

The main components of the language implemented by *Service Studio* are: 1) *Web Flows*, to connect the application web pages and define the possible end-user interaction sequence; 2) *Web Screens* and *Web Blocks* , to define the application web interface; 3) *Action Flows*, to implement system behaviour; 4) *Entities*, to build the database model.

A *web flow* is represented by an directed graph, containing an initial node followed by an infinite number of nodes. Each node represent a *web Screen* and the edges the transitions between *web Screens*. The *web screens* represent the application web pages.

An *action flow* is also a directed graph, representing each node an operation of the language. An edge defines the next action to perform. Figure 2.4 depicts an example of an *Action flow*.

In order to handle the events triggered by user interaction with the application, the language allows the creation of *Actions*. An *Action* is formed by: 1) Input and Output parameters; 2) Local variables; 3) Own *Action flow*. Below we briefly

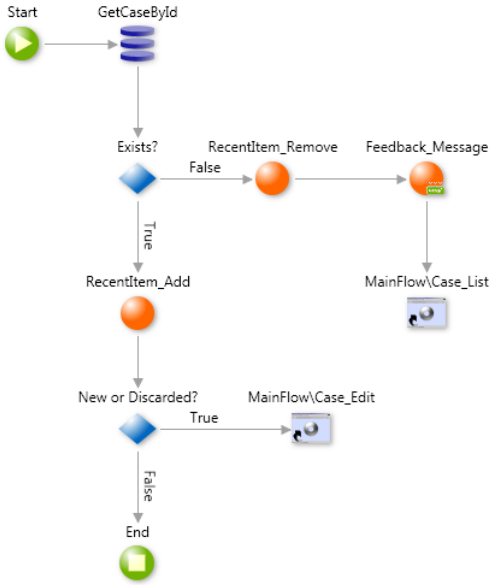

Figure 2.4: Example of an *Action flow*

describe some constructs supported by the language:

- **If & Switch** - Control the execution flow by evaluating expressions.
- **Assign** - Assign a value to a variable.
- **Foreach** - Performs a single or a collection of actions for each element of a list.
- **Simple Query** - Executes a database query over one or more *Entities* within the *eSpace*. *Service Studio* provides a graphical interface to define the query elements, which permits the user to build database queries without coding SQL.
- **Advanced Query** - Similar to the previous construct, however in this case the user needs to specify the SQL code for the query.
- **Destination** - Deviates the execution flow to a web page created within the application (*web screen*).
- **Execution Action** - Executes any action defined in the *eSpace*. The user has available various action types, like *Entity Actions* or *User Actions*.

Figure 2.4 representing the *action flow* is possible to see some of the language constructs being used.

By adding new data migration constructs to the *Service Studio* language, we offer to developers functionalities to reduce the difficulty, effort and problems for the maintenance process.

After building the applications, those are ready to compile and publish to *Platform Server*. In the next section we describe the application lifecycle.

2.3 Applications Lifecycle

In *Outsystems* application lifecycle consists in two environments: Development and Production. Considering applications lifecycle depicted in Figure 2.5 we briefly describe the main steps:

1. Developers use *Service Studio* to implement and model their web applications. To test those applications it is necessary to publish them to the development environment.

2. After the testing phase, if the application is ready to go into production the *delivery manager* is responsible for that.

3. The users use the applications published to the production environment.

Service Center is a web application within the web server connecting *Service Studio* and the development environment. In the last version of the *Agile Platform*, was integrated a tool called *LifeTime*[1]. The tool is similar to *Service Center*, however it connects both development and production *Service Centers*.

2.3.1 Deploying an application

When the implementation process is concluded, the application is ready to be deployed to the development environment through the *1Click-Publish* feature. This operation involves four main steps:

- Save - Saves the *eSpace*
- Upload - *Uploads* the *eSpace (OML)* to the server
- Compile - The compiler receives the *eSpace* and generates C#, Java, HTML, CSS, SQL and more files. After that, the files are compiled into assembly/byte code files.

[1] http://www.outsystems.com/help/lifetime/7.0/

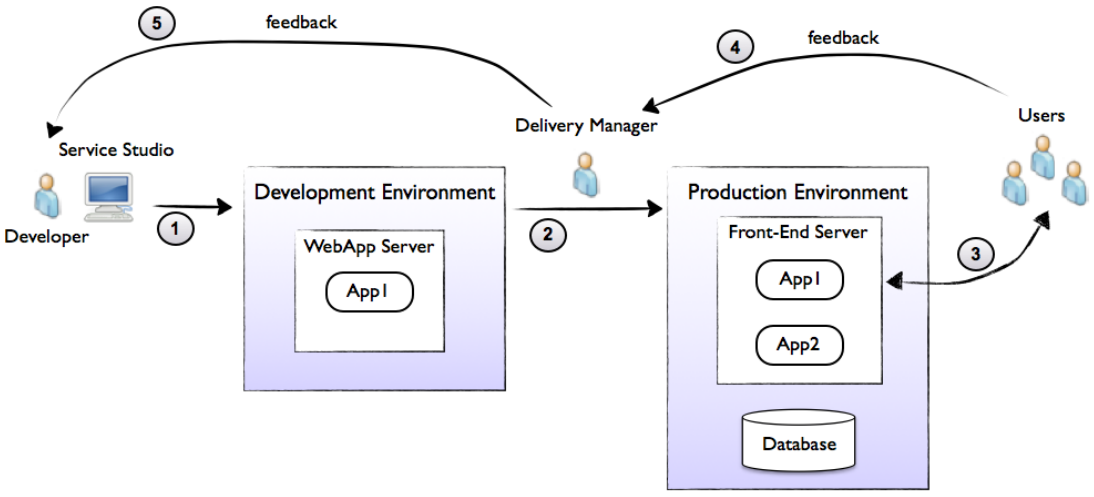

Figure 2.5: *Ousystems applications lifecycle*

- Deploy - Updates the last published *eSpace* version.

In this chapter we described the *Agile Platform* components and services to support the development and execution of *Outsystems* web applications. We present next the approach taken for the first phase of this project and the related work.

3

Related Work

In order to better understand the context of our problem, it is necessary to discuss concepts and the related work previously done. In this chapter we will provide an overview of this subject's state of the art.

3.1 Background

Relational Data Model The relational model is the primary model used by commercial data-processing applications, and uses relational algebra as its underlying theory. We will describe the structure of the relation model briefly. Therefore, to know more about relational algebra and the relational model we refer to [SKS10].

The Relation model is consists in a collection of *tables* or *relations*, each one with a unique name. Those relations consist in a set of attributes, each with a unique name within the relation. Each attribute in a relation defines the set of allowed values of a given attribute, being called *domain* of the attribute.

keys aim to distinguish various Entities. There are the following key types:

- *Primary Key* is a candidate key to identify the tuples within the relation.
- *Foreign Key* - If we have two relations R_1 and R_2, a foreign key of R_2 is the primary key of R_2 referenced in R_1

Diagrams to represent this type of database models have the following rules:

- Relations are represented by boxes
- Relation attributes are listed on those boxes
- Attributes belonging to the primary key of relation are listed first
- Foreign key dependencies are drawn as arrows from the referencing relation to the referenced relation.

We can see an example of a diagram in Figure 3.1.

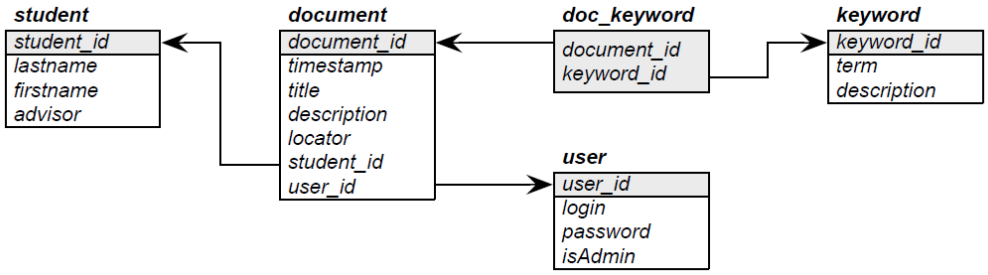

Figure 3.1: Relational Schema Diagram Example

SQL language [SKS10] is the standard language for relational databases. Each relational database management system (RDBMS) uses this as its front-end. The language is composed by :

1. Data Definition Language (DDL) - Constructs supported by a database that enable the insertion, delete or modification of structures (relational tables or classes) within the database.
2. Data Manipulation Language - Constructs supported by a database that enables the access of data within it, including insertion, delete, search, and update of data.

We will refer next the important SQL commands for our work. For a complete list and more deeply explanation about the SQL language we refer [SKS10]:

- The SELECT command specifies the structure of the result of the query. It is one of the most used SQL commands.
- The FROM command do specify the data sources for a given query. It can be constituted by one or more tables, which can be linked by some operators like the JOIN construct.

- The clauses WHERE, ORDER BY, HAVING or GROUP BY are used to filter the results of a given query.
- The INSERT and UPDATE commands. The first one allows to insert new data into the database, whereas the second one allow to update the existent data within the database.
- The CREATE or ALTER commands that allows to create and alter tables in the database schema.

These commands fit in our work since we want to have modification and creation clauses to allow us to perform the changes both in the data model and its data. Consequently, we present an example of a query updating a field of a table gathering the data from another table using the SELECT command:

```
1  UPDATE Customer SET
2    TotalAccountBalance =
3      (SELECT SUM(balance) FROM Account
4        WHERE Accound.CustomerId = Customer.CustomerId)
```

The most usual and used way to modify and migrate the schema and data is through SQL scripts. Due to the difficulty (because it is a manual approach) of that method arises the need to build tools to help the developers to perform changes on the data models and its data in a more automatic way. In the next section we will speak about Database refactoring, explaining the process and strategies to refactor database models.

3.2 Database Refactoring

In [AS06], a Database Refactoring is defined as a simple change to a database, that improves not only its design, but retains its behavioural and informational semantics. Refactoring a database is conceptually more difficult than performing code refactorings [FBB+99], since code refactorings need to maintain behavioural semantics, whereas database refactoring must also maintain informational semantics [AS06].

The process of refactoring a database gets even more complicated by the amount of coupling resulting from the database architecture. We can define coupling as a measure of the dependence between two item, thus, the more highly coupled two things are, the greater is the possibility that a change in one will require a change in the another one. In Figure 3.2 we can see both database architectures. The

single-application database architecture consists in only one application interacting with the database, whereas the second architecture is much more complicated because there are many external sources interacting with the database, some of them beyond our control.

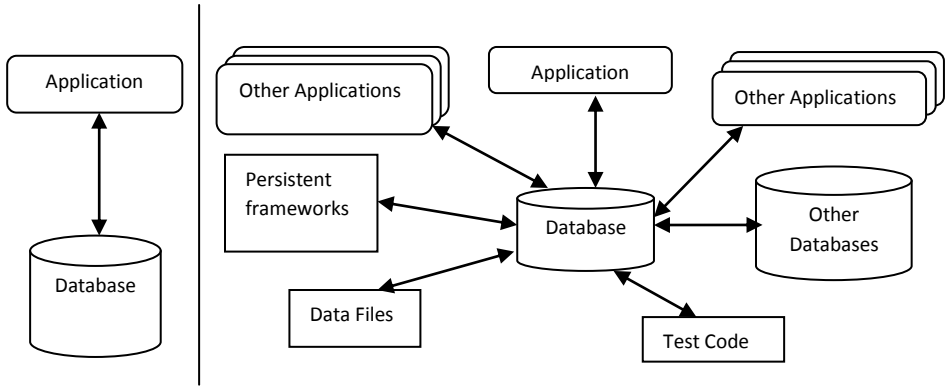

Figure 3.2: Database architectures

3.2.1 Database Smells

"code smell" concept was introduced by Fowler in [FBB+99], defining it as categories of problems in the code that lead us to the need of refactoring. Similarly, the concept of *database smells* was defined in in [AS06], indicating the possible need to refactor the database. The smells identified by the authors were the following:

- **Multipurpose column** - a column being used for many purposes can possibly mean extra code exists to ensure the use of the data in the right way.
- **Multipurpose table** - when a table is being used to represent for example several types of entities, which means that we possibly could have some NULL values. As an example, if we have a *Customer* table to represent people and companies, and to represent a company we need a legal name, while a person has first,middle and last name, in some rows we will have NULL values.
- **Redundant data** - This can lead to inconsistency issues, once we have the data stored in too many places.

- **Tables with too many columns** - When a table has lots of columns, it indicates a lack of cohesion, then we need to normalize the structure.
- **Tables with too many rows** - Large tables are an indicative of poor performance, since the search is time-consuming in a table with millions of rows.
- ***Smart* Columns** - Smart Columns are the ones in which different positions within the data represent different meanings. We need to parse to discover more granular information.
- **Fear of change** - If we have fear to change the database it is because we are afraid of breaking something, and it is a sign that we really need to refactor our database schema. This is one of the main goals of our work, we want to reduce the fear of change of the developers when facing a change scenario.

3.2.2 Process of Database Refactoring

In this section we describe the process implementing safely a refactoring within a production environment [AS06]. The process of a database refactoring starts when the developers has the need to fix an issue in the application or when he his faced with a new requirement to implement. The order we should follow defined by [AS06] is:

- Verify if the refactoring is appropriate;
- Choose the most appropriate refactoring;
- Deprecate the original schema;
- Test before, during and after;
- Modify the schema;
- Migrate Source Data;
- Modify the programs accessing the database;
- Run regression tests;
- Version Control;
- Announce the refactoring.

We will describe the steps we are more interested in our work, however if the reader wants to know more deeply all the steps we suggest [AS06].

Verify and choose the most appropriate refactoring Before performing a database refactoring we should reflect if it is really necessary perform that refactoring or if it is the right one to perform. Moreover, we should think if the change it is worth of the effort.

Deprecate the original schema The life cycle of a database refactoring consists in three phases: Implementation, Transition and, Completed. While the refactoring is not completed and could exist applications accessing the old data we should maintain in the database the data as deprecated data.

Modify the schema This step is one of our main goals. We want to modify a schema in a more user friendly and automatic way. In [AS06] is proposed a manual way to perform the changes in the schema, which we believe is the most used nowadays due to the interviews we did in Chapter 4. Two characteristics we should take in account is the **Simplicity** of performing the changes, by not creating scripts difficult to maintain, and the **Correctness**, once we want that the database schema evolves in a defined manner.

Migrate the source data We may want not only change the schema, but all the cases when we are in a production environment we need to take in account the data. Thus, we can have DML scripts to migrate the data.

Modify the programs accessing the database Changes on the database schema, may require the adaption of the external programs that access the changed portion of the schema. When many programs are accessing to the database schema, we run the risk that some of the application were not changed to work with the new version. So, we should assign to someone (usually a team) the task of update the external programs.

3.2.3 Database Refactoring Strategies

[AS06] refers strategies we could follow to accomplish the refactorings. We will use the ones we think are more relevant to our work, letting to the reader the reference to know more deeply the description of each one.

- Smaller changes are easier to apply;
- Identify uniquely individual refactoring;
- Implement large change by small one;

- Have a database configuration table;
- Choose a sufficient deprecation period.

3.2.4 Database Refactoring Categories

Refactoring were divided by categories in [AS06], those being:

- *Structural Refactorings* - Changing the table structure of the database schema
- *Data Quality Refactorings* - Intended to improve the quality of the information within the database. Improve consistency and usage of the values.
- *Referential Integrity Refactorings* - Changes that ensure that a referenced row exists in another table.
- *Architectural Refactorings* - To improve the overall manner in which the external programs interact with the database.

In the Appendix we list all refactorings associated to each category. We do not explain all of them, besides we analyze the change pattern we are more interested for our work and that we identified in Chapter 4. In the next section we will discuss about Schema modification language[CMZ08], an approach to evolve a schema with a set of defined operators.

3.3 Schema Modification Language

A Schema modification Operator (SMO) is a function that receives a schema with a database as input and produces as output the modified version of that schema and a migrated version of the database. These operators tie together schema and data transformations and they carry enough information to enable the automatic query mapping. The SMOs together represent the SMO language and are shown in 3.3.

As an example we can consider the operation *"JOIN TABLE R,S into T"*. This operation creates a table T that results from joining the tables R and S. Considering sequences of SMOs we have the following characteristics:

- Depending the order in which operator is used the result can be different;
- Each operator acts in isolation on its input to produce the output;
- Different sequences of SMOs can produce the same result on the same schema.

SMO Syntax	Input rel.	Output rel.	Forward DEDs	Backward DEDs
CREATE TABLE R($\bar{\mathbf{A}}$)	-	R($\bar{\mathbf{A}}$)	-	-
DROP TABLE R	R($\bar{\mathbf{A}}$)	-	-	-
RENAME TABLE R INTO T	R($\bar{\mathbf{A}}$)	T($\bar{\mathbf{A}}$)	$R(\bar{x}) \rightarrow T(\bar{x})$	$T(\bar{x}) \rightarrow R(\bar{x})$
COPY TABLE R INTO T	$R_{V_i}(\bar{\mathbf{A}})$	$R_{V_{i+1}}(\bar{\mathbf{A}})$, T($\bar{\mathbf{A}}$)	$R_{V_i}(\bar{x}) \rightarrow R_{V_{i+1}}(\bar{x})$ $R_{V_i}(\bar{x}) \rightarrow T(\bar{x})$	$R_{V_{i+1}}(\bar{x}) \rightarrow R_{V_i}(\bar{x})$ $T(\bar{x}) \rightarrow R_{V_i}(\bar{x})$
MERGE TABLE R, S INTO T	R($\bar{\mathbf{A}}$), S($\bar{\mathbf{A}}$)	T($\bar{\mathbf{A}}$)	$R(\bar{x}) \rightarrow T(\bar{x})$; $S(\bar{x}) \rightarrow T(\bar{x})$	$T(\bar{x}) \rightarrow R(\bar{x}) \vee S(\bar{x})$
PARTITION TABLE R INTO S WITH *cond*, T	R($\bar{\mathbf{A}}$)	S($\bar{\mathbf{A}}$), T($\bar{\mathbf{A}}$)	$R(\bar{x})$, $cond \rightarrow S(\bar{x})$ $R(\bar{x})$, $\neg cond \rightarrow T(\bar{x})$	$S(\bar{x}) \rightarrow R(\bar{x})$,*cond* $T(\bar{x}) \rightarrow R(\bar{x})$,$\neg cond$
DECOMPOSE TABLE R INTO S($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$), T($\bar{\mathbf{A}}$,$\bar{\mathbf{C}}$)	R($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$,$\bar{\mathbf{C}}$)	S($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$), T($\bar{\mathbf{A}}$,$\bar{\mathbf{C}}$)	$R(\bar{x},\bar{y},\bar{z}) \rightarrow S(\bar{x},\bar{y})$ $R(\bar{x},\bar{y},\bar{z}) \rightarrow T(\bar{x},\bar{z})$	$S(\bar{x},\bar{y}) \rightarrow \exists \bar{z}\ R(\bar{x},\bar{y},\bar{z})$ $T(\bar{x},\bar{z}) \rightarrow \exists \bar{y}\ R(\bar{x},\bar{y},\bar{z})$
JOIN TABLE R, S INTO T WHERE *cond*	R($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$), S($\bar{\mathbf{A}}$,$\bar{\mathbf{C}}$)	T($\bar{\mathbf{A}}$,$\bar{\mathbf{B}}$,$\bar{\mathbf{C}}$)	$R(\bar{x},\bar{y})$, $S(\bar{x},\bar{z})$, $cond \rightarrow T(\bar{x},\bar{y},\bar{z})$	$T(\bar{x},\bar{y},\bar{z}) \rightarrow R(\bar{x},\bar{y})$,$S(\bar{x},\bar{z})$,*cond*
ADD COLUMN C [AS $const\|func(\bar{\mathbf{A}})$] INTO R	R($\bar{\mathbf{A}}$)	R($\bar{\mathbf{A}}$,C)	$R(\bar{x}) \rightarrow R(\bar{x}, const\|func(\bar{x}))$	$R(\bar{x},C) \rightarrow R(\bar{x})$
DROP COLUMN C FROM R	R($\bar{\mathbf{A}}$,C)	R($\bar{\mathbf{A}}$)	$R(\bar{x},z) \rightarrow R(\bar{x})$	$R(\bar{x}) \rightarrow \exists z\ R(\bar{x},z)$
RENAME COLUMN B IN R TO C	$R_{V_i}(\bar{\mathbf{A}}$,B)	$R_{V_{i+1}}(\bar{\mathbf{A}}$,C)	$R_{V_i}(\bar{x},y) \rightarrow R_{V_{i+1}}(\bar{x},y)$	$R_{V_{i+1}}(\bar{x},y) \rightarrow R_{V_i}(\bar{x},y)$
NOP	-	-	-	-

Figure 3.3: Schema Modification Operators

[CMDZ10] extends the SMO language by introducing new six operators called ICMO (Integrity Constraints Modification Operators), used to perform the evolution of integrity constraints. Following we can analyze an example integrating both types of operators, and then in Figure 3.5 is presented the SMO and ICMO syntax.

```
1  ALTER TABLE exon DROP PRIMARY KEY pk1;
2  DROP COLUMN rank FROM exon;
3  ALTER TABLE exon ADD PRIMARY KEY pk2(id) ENFORCE;
```

In Figure 3.4 we have the table *exon*. The evolution example starts by dropping the primary key of the table *exon*, dropping the *rank* column too after that. The last step adds a new primary key for the same table, but the new key is only composed by the *id* field.

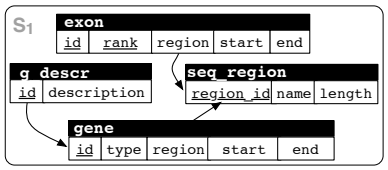

Figure 3.4: Schema Example

Forcing Information Preservation for SMOs Some technical challenges for the data migration and query rewriting problems were raised by the operators that were not information-preserving. The authors defined that an operator O is information-preserving if and only if:

- Is invertible
- The old and new schema are equivalent, i.e. have the same information.

With the introduction of ICMOs, each SMO operator can be information-preserving by adding the correct Integrity Constraints, whereby any information loss will be

Schema Modification Operators (SMO) Syntax
CREATE TABLE R(a,b,c)
DROP TABLE R
RENAME TABLE R INTO T
COPY TABLE R INTO T
MERGE TABLE R, S INTO T
PARTITION TABLE R INTO S WITH $cond$, T
DECOMPOSE TABLE R INTO S(a,b), T(a,c)
JOIN TABLE R,S INTO T WHERE $cond$
ADD COLUMN d [AS $const \vert func(a,b,c)$] INTO R
DROP COLUMN c FROM R
RENAME COLUMN b IN R TO d
Integrity Constraints Modification Operators (ICMO) Syntax
ALTER TABLE R ADD PRIMARY KEY $pk1(a,b)$ <policy>
ALTER TABLE R ADD FOREIGN KEY $fk1(c,d)$ REFERENCES $T(a,b)$ <policy>
ALTER TABLE R ADD VALUE CONSTRAINT $vc1$ AS $R.e =$ "0" <policy>
ALTER TABLE R DROP PRIMARY KEY $pk1$
ALTER TABLE R DROP FOREIGN KEY $fk1$
ALTER TABLE R DROP VALUE CONSTRAINT $vc1$

Figure 3.5: SMO Language: SMO+ICMO

because of an ICMO and not because of a SMO. With this approach the management of structural changes and alterations of information capacity can be separated.

To better understand this approach we have the Listing 3.1. Constraining the values of table *gene* with operator 2, guarantees that the JOIN operator will be information preserving, and consequently any loss of tuples will be imputed to operator 2 and not to operator 3.

Listing 3.1: Information-Preserving Example

```
1 RENAME COLUMN type IN gene TO biotype;
2 ALTER TABLE gene ADD FOREIGN KEY fk2 (id)
3      REFERENCES g_descr(id) ENFORCE;
4 JOIN TABLE gene,g_descr INTO gene
5      WHERE gene.id = g_descr.id;
```

3.3.1 SMO Invertibility

PRISM [CMDZ10] deals with invertibility within the operational SMO language, having each SMO one or more inverses. The invertibility of each operator is characterized by the existence of a perfect/*quasi* inverse and uniqueness of the inverse. As we can see in Figure 3.6 JOIN TABLE and DECOMPOSE TABLE represents each other's inverse, in the case of the information preserving step, while if the forward step is not information preserving they represent a quasi inverse.

As we referred before some SMO can have multiple inverses. PRISM uses integrity constraints or interaction with the DBA to disambiguate the inverse. If

SMO	unique	perfect	Inverse(s)
CREATE TABLE	yes	yes	DROP TABLE
DROP TABLE	no	no	CREATE TABLE COPY TABLE NOP
RENAME TABLE	yes	yes	RENAME TABLE
COPY TABLE	no	no	DROP TABLE MERGE TABLE JOIN TABLE
MERGE TABLE	no	no	PARTITION TABLE COPY TABLE RENAME TABLE
PARTITION TABLE	yes	yes	MERGE TABLE
JOIN TABLE	yes	yes/no	DECOMPOSE TABLE
DECOMPOSE TABLE	yes	yes/no	JOIN TABLE
ADD COLUMN	yes	yes	DROP COLUMN
DROP COLUMN	no	no	ADD COLUMN, NOP
RENAME COLUMN	yes	yes	RENAME COLUMN
NOP	yes	yes	NOP

Figure 3.6: Schema Modification Operators Inverses

the integrity constraints do not carry enough information the DBA can define a unique inverse for all the queries or he can manage each query independently and choose different inverses for different queries.

3.4 Data Migration

After the changes were made to the database structure, arises the need to migrate and maintain the data consistent, hence the data needs to be migrated in a manner it can work with the new version of the schema. There are several ways to perform the migration of the data of a database. The migration mechanisms proposed by [AS06] are manual SQL scripts, while in [CMZ08, CMDZ10] the migration of the data is done through logical mappings called DEDs (Disjunctive Embedded Dependencies).

In [VWV11] is presented a DSL for the coupled evolution associated to data models and its data. After creating the WebDSL language [GHKV08], the authors modeled an evolution model, defining three types of migrations:

1. Schema Modification - Operators only requiring schema modification;
2. Conservative Migrations - To change the schema and re arrange the data;
3. Lossy Migrations - Exists data loss.

Ruby on Rails[1] supports migration of databases for an evolving web application. Those applications use an ORM to persists data in a relational database, and the developer needs to specify the migrations himself.

A tool implementing the migration mechanisms referred in [AS06] is Liquibase[2], which stores the changes in XML files and after that applying it to the databases.

3.5 Access Program Adaptation

We can have databases shared by many applications and databases accessed only by an application. In both cases it is important to adapt the code of those programs, for example, queries working with an old schema version need to be adapted to work with the refactored version of the schema.

[CMZ08, CMDZ10] solve that issue by using the chase and backchase algorithm [DNR08] to rewrite the queries to work with the new version of the schema, using the algorithm the DEDs referred in the previous section to rewrite the queries.

3.6 Change Patterns

The goal of this section is to analyze the motivations, tradeoffs, impact on application code and how to perform data migration in the change patterns we identify with the interviews in chapter 4. Our is based in [AS06].

Move Column *Move Column* consists on moving a column from a table to another one.

The motivation to achieve this refactoring is that we may want to normalize the table, or to perform a refactoring afterwards, or denormalize if the column is inserted in a join only due to its existence in the wrong place. Reorganizing the tables structure is another motivation.

As potential tradeoffs, reducing the data redundancy may decrease the performance if additional joins are required by the applications to obtain the data. If we improve the performance we will increase data redundancy.

Updating the schema we need to take care about the value of the column and if referential integrity constraints exists.

[1] http://guides.rubyonrails.org/migrations.html
[2] Available on-line: http://www.liquibase.org/

Change ID column type This change is usually related with integrity constraints, since by changing them deals with the consistency of the database. In some point, because of a new requirement, the foreign key that was pointing, for example, to *user* table is now pointing to the *UserMaster* table.

According to our research, we did not find any tools implementing a similar pattern, however it was realized in Chapter 4 that this is a common pattern, within *Outsystems* context.

Adding a new constraint Constraints enforce data dependencies at the database level [AS06], preventing persistent data to be invalid.

Adding foreign key constraints can decrease the performance of the database, since the foreign key table will be always verified when an update is done to the data. Furthermore, we need to take into account the order of insert or delete operations. Liquibase implements this operation by defining a change set specifying the attributes required. [CMDZ10] allows the user to perform the refactoring through the SMO operator *"ALTER TABLE R ADD FOREIGN KEY fk1(c,d) REFERENCES T(a,b) <policy>"*, having the user to specify the relation R to be changed and the how the foreign key is formed. The last parameter *policy* can be as following defined by the authors:

- CHECK: The system verifies if the database instance satisfies the new constraint and, if not, the ICMO operation is rolled back.
- ENFORCE: The system removes all tuples violating a new integrity constraint. That removed tuples are kept in a temporary table named *violation tables*.

Master/Detail table This operation consists on vertically split an existing table into one or more tables. Consider as an example, a table *Client* with the attributes *Id, Name, Address, Phone, Phone_2, and Email*. After, we realized we want to have more than two phone numbers. Thus, we need to evolve our model by creating a detail table, possibly with the name *Contacts*, to accomplish such functionality. This operation implies splitting the table *Client* and migrate the existent data within *Phone*,*Phone_2* and *email* attributes into the *Contacts* table. Besides, we need to take into account the applications using those fields. Using SMO operators [CMZ08, CMDZ10] it is possible to use the operation using the *DECOMPOSE Table* operator. *Liquibase* does not implement the *Split Table* refactoring. A possible approach is to manually code SQL scripts as referred in [AS06].

Other Patterns There are simpler patterns that we are not discussing, but they are completely identified. Although, because the patterns are already implemented on usual database management systems we will not give then a special emphasis. Those patterns are for example:

- Rename a column,
- Rename a table,
- Adding new attribute,
- Adding new table.

4

Preliminary Analysis

The early stages of the project were dedicated to learn about the *Agile Platform*. At first we performed tutorials available on *OutSystems Academy*, in order to understand how applications are developed in the *Agile Platform*. We wanted to understand developers difficulties when developing web applications in *Service Studio* focusing on the data model. Afterwards, we studied concepts, techniques and solutions that guided our decisions along the solution model design and implementation phase. Our research was focused on database refactoring, database migrations and database schema evolution.

After that, in order to understand problems and difficulties faced by developers when evolving database models we interviewed experienced developers and project managers frequently evolving applications database model, in the context of the *OutSystems Agile Platform*. Interviews intended to capture the most frequent scenarios faced by developers when changing and evolving the database model. In order to introduce new customer requirements to applications or to improve applications design and performance, developers may have different approaches to accomplish those tasks. Thus, we also aimed to understand what developers do to keep applications data model consistent and their approach to evolve it.

4.1 Interviews Structure

Interviews were split in two parts. The first part was composed by questions aiming to capture the problems, difficulties and most common scenarios faced by the interviewee when evolving applications database model. In the second part our goal was to understand if the scenarios captured from the related work, are common in the context of the *Agile Platform*.

4.1.1 First Part

The questions composing the first part of the interviews are:

1. Which are the most frequent scenarios that require changing the database model?
    - Goals, Driver, Processes, Risk, Human Effort both for Development and Operations
2. Which are the most costly scenarios that require changing the database model?
3. How do you handle high risk database changes?

On this part, we forced the interviewee to think on scenarios faced during previous projects, in order to understand, which are the most frequent scenarios, when changing applications database model. We asked also, what is their approach and how much effort was required to accomplish the requirements requiring database changes. The second question aims to know, how costly were the scenarios they faced. The third question aimed to know, how the interviewee handles high risk database changes, for example when applications already contain data in production.

4.1.2 Second part

The questions composing the second part of the interviews are:

- Can you prioritize the following scenarios by saying if they are Frequent or Not Frequent?
    1. Convert primary key to auto-number.
    2. Convert compound key to a simple key.

3. Convert column between two different type of entity identifiers. e.g.: *ProductId* → *CustomerId*
4. Split column. e.g.: convert *Name* to *FirstName*, *LastName*.
5. Merge column e.g.: convert *FirstName*, *LastName* into *Name*
6. Move column to entities that have a one to one relation. e.g.: moving binary data column to an extension entity, or moving back data from an extension entity to the main entity
7. Move data from a master entity to a detail entity. e.g.: *email* attribute from a *person* entity to a *person_contact* detail entity
8. Move data from a detail entity to a master entity. e.g.: Last modified attribute

Chart 4.1 depicts the answers collected on the second part of interviews , and also shows the frequency of the proposed database change scenarios.

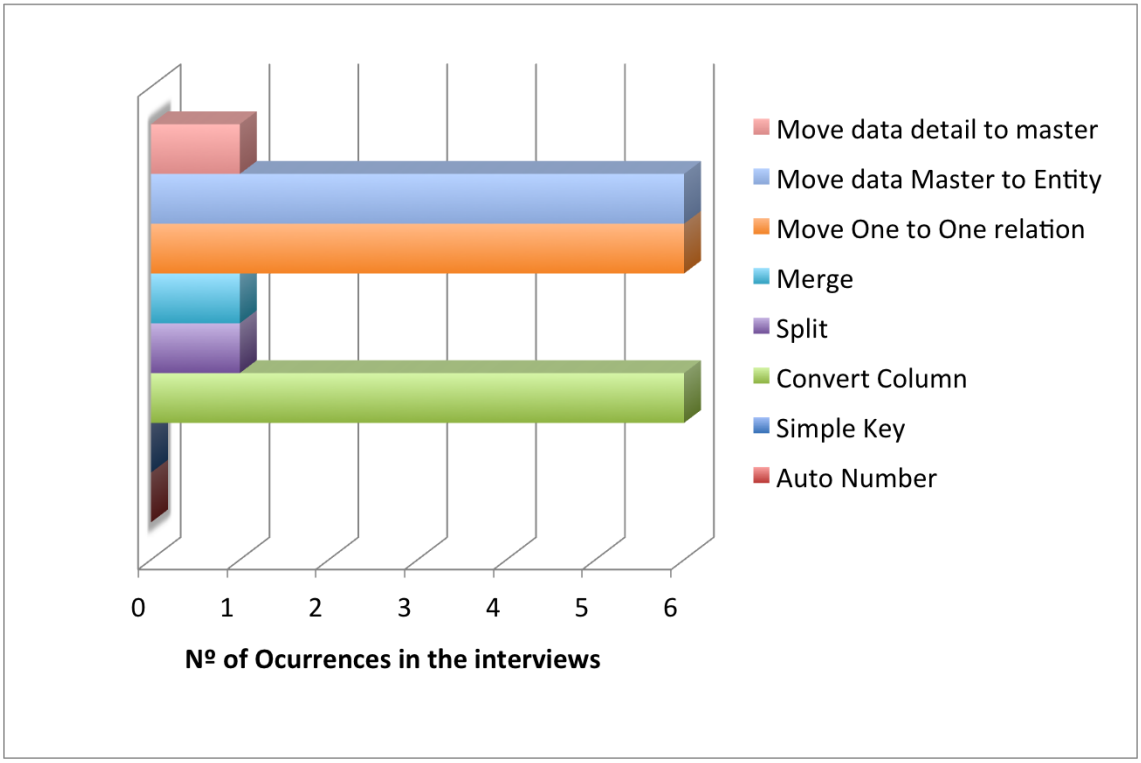

Figure 4.1: Interviews second part results

4.2 Interviews Notes

Regarding the interviews, during the first part we did not follow strictly the questionnaire, once the respondents were free to relate their thoughts and experience

when evolving applications. They described the most difficult problems in the development and maintenance process of applications, focusing on the database layer.

As a common observation, respondents referred that in the early stages of applications development, they aim to have the database model as clean as possible, because when later they need to evolve applications and introduce new features requiring database model changes the process may be simpler. Writing SQL scripts to update and evolve the database is the usual approach, as referred by the respondents during interviews.

The most challenging problems when changing applications database structure, arise when applications already contain real data in production, as reported in two interviews. It was also referred, that the evolution process is usually painful for the developers, since loosing data or having application downtimes are not acceptable options.

In addiction to the scenarios presented during interviews, respondents also reported other common scenarios as for example:

- Changes in attribute type (Similar to the Change foreign key), reported in three interviews;
- Change the attribute property *isMandatory*;
- Adding new constraint - When the developers want to add a new constraint to the model, they insert manually the constraint in the database management system, publishing after that in the *eSpace*.

In order to migrate data and change the database model, writing SQL scripts is the most frequent approach as reported in an interview. Because of that, was also reported that having an impact analysis tool analyzing the real data in production would be very useful, since developers may know, if the transformations performed in the database model on the development environment are compatible with the real data in production. A tool analyzing the real data in production, would improve extensively the maintenance process.

In the first part of the interviews, was difficult to developers to identify scenarios they faced on previous projects, while in the second part they properly understood the scenarios and they gave us useful feedback.

Discussion Analyzing interviews results, we conclude that developers face the problems we referred before when evolving applications database model. We also conclude that does not exist an automatic, smooth and safe way to evolve

the database models, data and code. Thus, implementing automatic features to evolve database models and its data, it is very useful to improve applications development and maintenance processes, keeping applications compatible along the different environments.

4.3 Change Operations Identified

This section describes change operations identified during interviews. We also analyze which are the most interesting and useful to provide to developers, and how they can accomplish those tasks with tools available at the moment. We also discuss which are the most frequent, costly and risky. Analyzing Figure 4.1, we can conclude that scenarios usually faced by developers are similar.

Change Foreign Key In the sixth interview, this scenarios was defined as frequent. Changing the foreign key type is a very specific scenario in the context of *Service Studio*, but can bring problems related to integrity constraints.

Adding a new constraint It was referred in sixth and fourth interviews that adding a new constraint to the database model is a very common scenario. The usual approach is to add the constraint manually in the DBMS, before publishing the application, in order to avoid integrity errors.

Change attribute type Analyzing the interviews this is a very common scenario. Thus, it is a change pattern that deserves our attention. This operation consists on changing the type an attribute/column. When changing an attribute type, the real data in production may not keep compatible. That is an issue we aim solve with our solution.

Master/Detail table Analyzing only the table 4.1, we realize that this is not a frequent scenario. However, the respondents referred that besides is not one of the most frequent scenarios, they face sometimes this kind of refactoring. For example, consider a table *Client* with the attributes *Id, Name, Address, Phone, Phone 2, and Email*. After that, we realized that we want to have more than two phone numbers. Thus, we need to evolve our model by creating a detail table, possibly with the name *Contacts*, to accomplish that requirement. In conclusion, although interviews statistics we will consider this as a possible operation to support.

In this chapter the approach taken on the early stages of this dissertation was described. We started by getting familiar with the *Agile Platform* studying after that concepts, techniques and solutions that guided our decisions along the solution model design and implementation phase.

After that we interviewed experienced developers and project managers in the context of the *Agile Platform*. With that, we extracted common scenarios faced by the developers when evolving applications and which scenarios carry higher risk. This research was aimed to identify and analyze scenarios captured during interviews and identified in the previous study of the state of the art, in order to select the most interesting scenarios to support in our work.

In the next chapter we present the solution model to achieve our goals, in the context of the *Agile Platform*.

5

Model Solution

In this chapter we present our solution to introduce database evolution features in the *Agile Platform*. As presented in chapter 1, our goal is to provide a safe and guided process for developers to change applications data model. We focus on applications already in the maintenance phase, i.e., applications that were already deployed to production. The main problem to developers is to evolve those applications that already deal with real and sensible data which cannot be lost, and whose migrations cannot be directly tested in development environments. Thus, our solution allows developers to maintain the synchronization between both development and production environments. One of the problems that usually arises when developers are changing the database schema in the development environment, is that they do not have access to the real data in production environment. Thus, is difficult to developers to know what is the impact of changing the database structure on the application data. Having information about the state of the real data in production, helps the developers to know the real risk of their changes in the applications database structure. Figure 5.1 shows the flow from applications development phase until applications are deployed into production. The developer changes the application in *Service Studio* and publishes it in the development environment. The system is responsible to deploy the application when it is ready. The system deploys the application in the production environment through *LifeTime*. In this case, we have two different scenarios: either the application is successfully deployed into production or the process is aborted due to database conflicts between application versions.

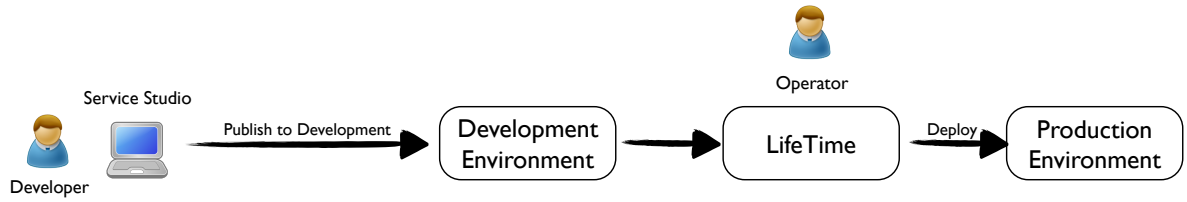

Figure 5.1: Application Development and Deployment process

The second case when there are problems in the data migration is critical because the deployment process is postponed and the operator needs to inform the development team about the errors in the migration script and wait for it to be fixed. The common approach followed by developers is to write SQL scripts to fix the database conflicts problems and to also to migrate data if needed.

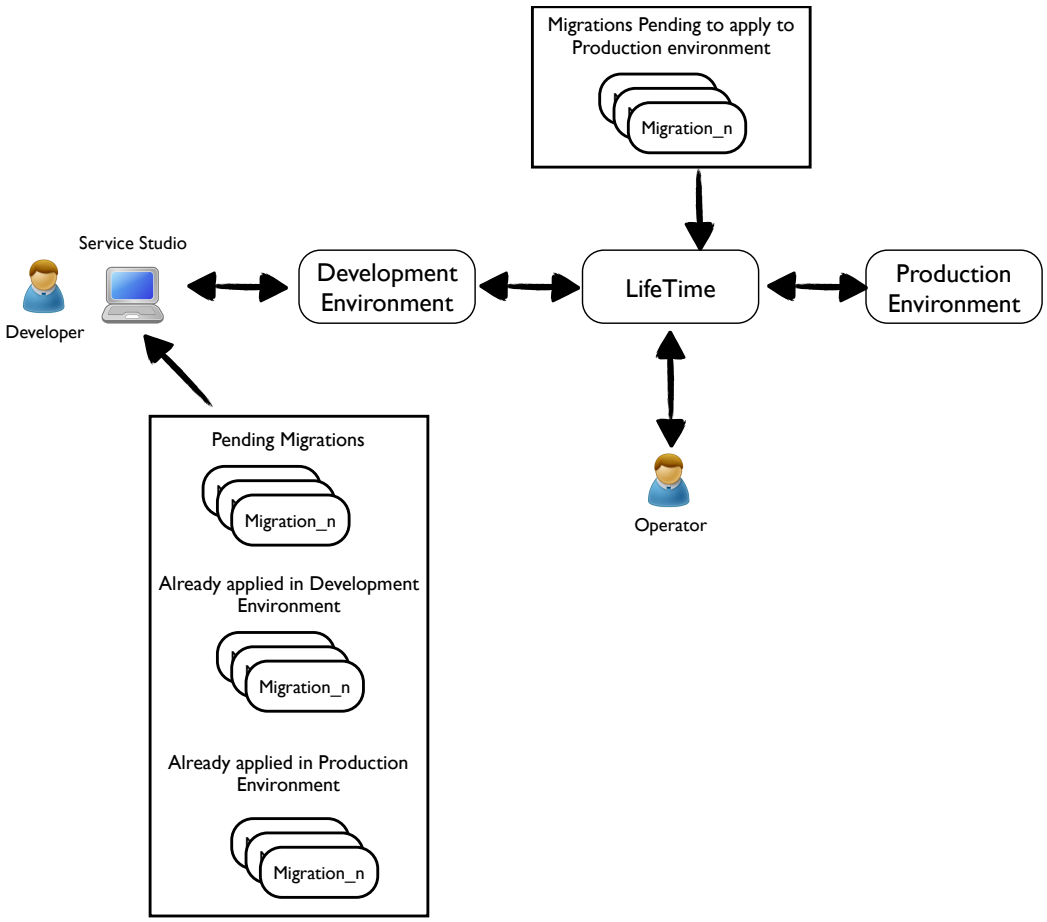

Figure 5.2: Migrations within *Agile Platform*

To overcome these limitations and to provide an easier and safer way to developers to change and evolve the database model we introduce database migrations

in the *Agile Platform*. Migrations are a mechanism to capture transformations in a database model and schema in a structured and organized way (we explain migrations in detail in the next sections). Figure 5.2 depicts how migrations are integrated. To distinguish the migrations already applied on the different environments we divide them into categories according to their application stage:

- **Pending migrations** - Generated in *Service Studio* when the developer changes the database model;
- **Already applied in Development Environment** - Changes to the database model already published in the development environment;
- **Already applied in Production Environment** - Changes to the database model already deployed into production.

Whenever developers change the database model or schema, a new migration is generated and stays pending until it is published to the development environment. The generated migrations contain information about who changed, when and what was changed. After that, the development environment contacts *LifeTime* that communicates with the production environment to get information about the impact of the migration in the database data already in production. This is possible because the *LifeTime* allows developers to have information about the version of each application running in development and production environment and also to access to an instant snapshot of any applications inconsistencies between environments. Thus, after developers change the database model in the *Service Studio*, the system analyzes on demand the impact of those changes in the data model already in production. As the production data analysis may take some time to complete, developers continue changing and evolving their applications while the system is analysing the impact of their transformations in the database model in production. After a while, the developer receives a warning that requires a migration rule to fix the incompatible data in production.

Thus, developers have on demand what is the impact on the database model in production, of the changes they made in the development environment. In the next sections we present the migrations model in more detail.

5.1 Migrations Model

As we presented before, our solution is based in the concept of migrations. Migrations are a mechanism to capture transformations in a database model and

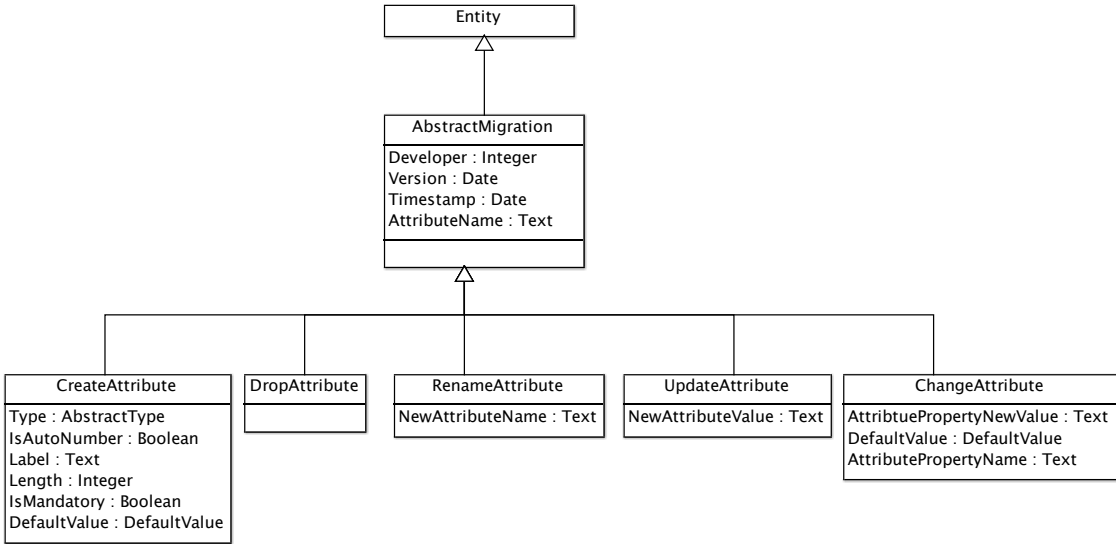

Figure 5.3: Class Diagram.

schema in a structured and organized way. We describe the implementation of migrations within *Service Studio* in the next chapter. It is important to refer that migrations are not editable by the developer, except the ones requiring updates. Migrations represent operations made by the developer to the database model to evolve it. So, the developer modifies the database structure and migrations are generated automatically creating a sequence of operations. Then, they are applied to the database model according to their order. To represent the modifications made by the developers in the database model, we have defined the following kinds of operations, represented in the Figure 5.3:

- **Create Attribute** : captures the creation of an entity attribute;
- **Drop Attribute** : captures the dropping of an entity attribute;
- **Rename Attribute** : captures the renaming of an entity attribute;
- **Change Attribute** : captures a change in a property of an entity attribute;
- **Update Attribute** : defines a new value for the entity attribute.

The presented operations capture the transformations on the database model and represent specific kinds of migrations extending the generic migration class (Figure 5.3). In our model a generic migration has the following properties:

- **Developer**: captures who changed the database model;

- **Version**: A version of a migration is a value based on the version of the migration being created and an instant. Versions have an absolute order. With that we distinguish which are the migrations already applied in development and in production. Also we know which ones are pending to publish. The development and production versions of the application are both stored in the server. The development server version is captured at the time of publication of the application in the development environment. The production server version is captured when the application is deployed into production.

- **Timestamp**: when the migration was defined;

- **Attribute Name**: represents the name of the attribute related to the migration.

Migrations intend to capture the common changes in the database model such as adding or deleting an attribute, renaming and changing a specific property of an attribute. The classes extending the generic migration and that represent each kind of operations are described in the next paragraphs.

Create Attribute The operation of creating an attribute is captured by this kind of migration and that includes all the underlying attribute properties, namely:

- **Type:** The data type of the attribute. Ex: Text, Integer, Decimal, Boolean, Date, etc;

- **IsAutoNumber:** Boolean property of an attribute. If set to *True* the number is automatically generated and set at runtime;

- **Label:** The text used when the attribute is displayed in the widgets;

- **Length:** Integer specifying the size of the attribute;

- **IsMandatory:** Boolean property that if set to *True*, the element must have a value specified;

- **DefaultValue:** The attribute default value. Must be the same data type specified in the property ***Type***

These properties represent an attribute within *Service Studio* and that is the reason why they are cloned to the migration. When the attribute is created this kind

of migration is generated and the property value cloned into the object. The developer also can write upgrade rules in order to define which value the attribute should have after it was created. Upgrade Rules details are explained later on this chapter.

Drop Attribute When an attribute is dropped a migration of this type is generated in order to capture that action.

Rename Attribute This operation captures the renaming of an attribute. It has associated the new name of the attribute.

Change Attribute This operation is generated when the developer changes an attribute property. In the ***Create Attribute*** operation we capture all the properties related to the attribute but here we just capture the property that was changed and its new value because we only need that information to perform the modifications in the database. Also for a question of traceability to know what was changed The additional information associated to this property is:

- Property Name : The attribute property changed,
- New Property value: The new assigned value,
- Old Property value: The old value associated to the property.

The real data in production may not continue compatible with the database model in the development environment after developers alter it. Thus, developers also can write upgrade rules when changing an attribute. For example, change the *IsMandatory* property to *Yes* may not be compatible with the data in the database in the production environment. To solve that, the developer writes a value or an expression that incompatible rows (on this case, rows having the NULL value) should have. As previously referred, upgrade rules are described later on this chapter.

Update Attribute This operation allows developers to update the value of an attribute. It includes as a property the new value for the attribute. It can be an expression as we explain later in this chapter or just a default value. This kind of operation is useful when developers want to migrate just data from an existing attribute to another one.

This representation intends to capture who is responsible for the modification in the database model and when the changes happened. Also intends to separate

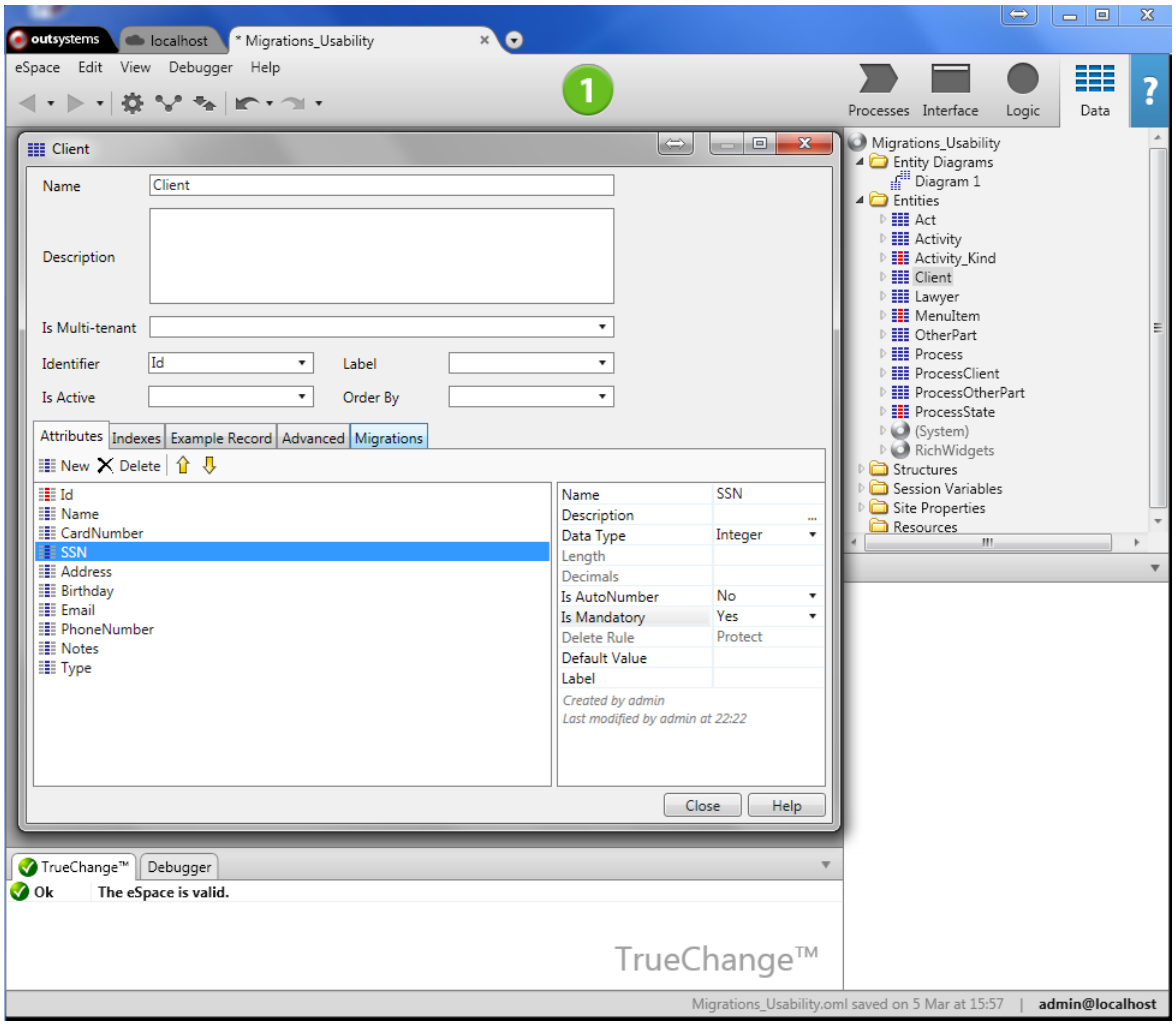

Figure 5.4: Property IsMandatory changed from No to Yes.

the migrations by version, i.e., if the migrations are aggregated to a specific version of the application, they are then separated by migrations already applied in the development version and migrations already applied in the production version. The advantage to this approach is to help the developers and also the operators to know which changes were made and in which version and environment. Figure 5.4 shows the attributes tab for an entity *Client* inside *Service Studio*. The developer wants to change the property *IsMandatory* of the *SSN* attribute show in figure 5.4. Changing the property value to *Yes* (meaning that the attribute cannot have NULL values) a migration is automatically generated. Changing to the tab Migrations in the Entity editor we see the generated migration and all its information. Figure 5.5 shows the migrations generated for the entity *Client*.

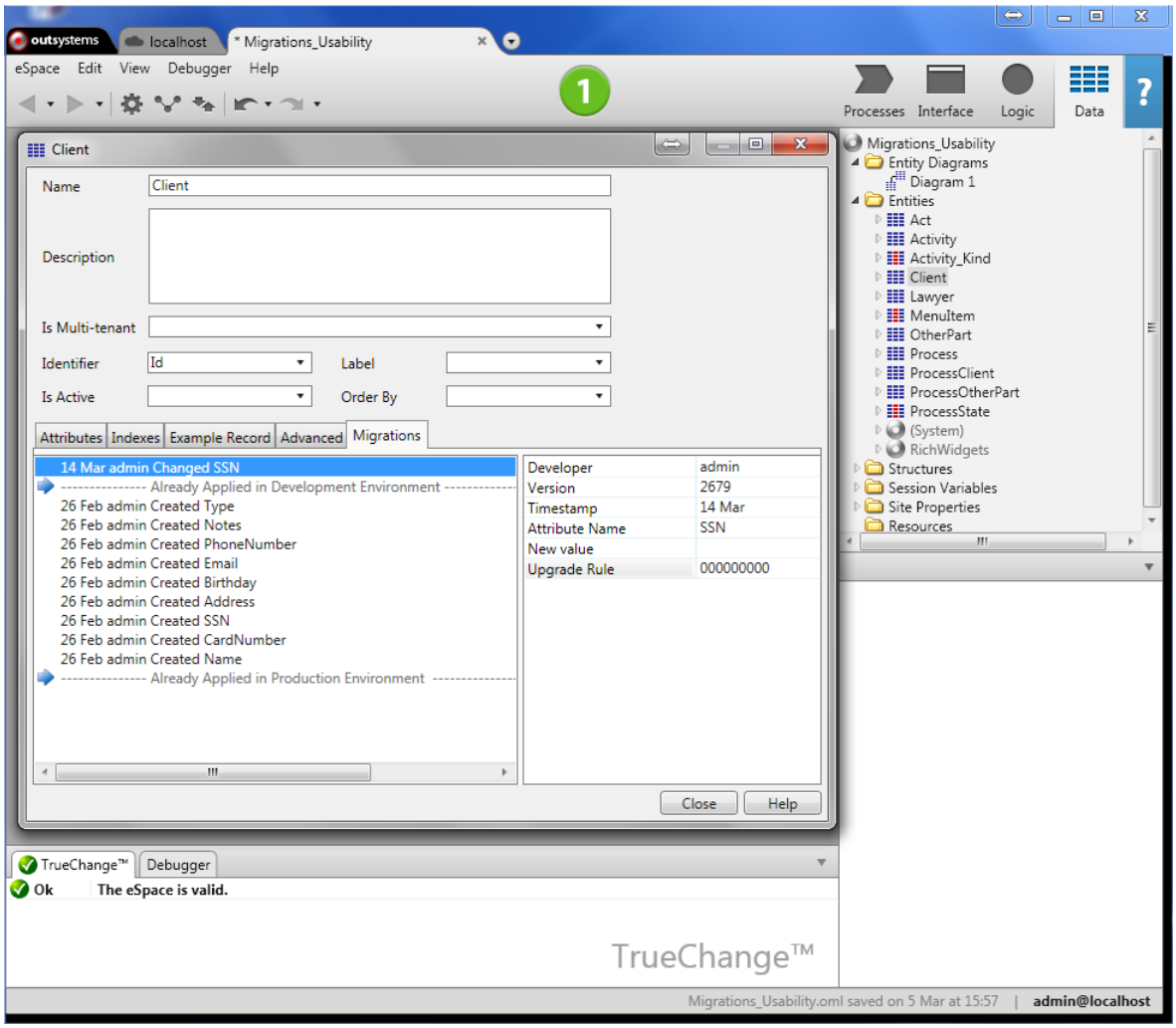

Figure 5.5: Generated Migration after the property was changed.

5.1.1 Merging Migrations

To reduce the number of pending migrations, merge migrations whenever is possible for each change made by the developer by merging them. We also increase applications performance by not having unnecessary operations updating and changing the database schema [SBB04]. Merging two migrations means comparing two consecutive migrations related to the same entity attribute and if the conditions to merge are valid we have two scenarios: a) One migration is generated having the properties of both migrations; b) Both migrations are removed. We now illustrate the merging of migrations by means of examples. Figure 5.6 depicts the first scenario. The attribute *PhoneNumber* was created and after that its type was set to integer. This generates two migrations that are compared after the second migration is generated. If the conditions related to the merge operation are valid, a new migration replaces the first one in the model with the new value of the type property , in this case *Integer*.

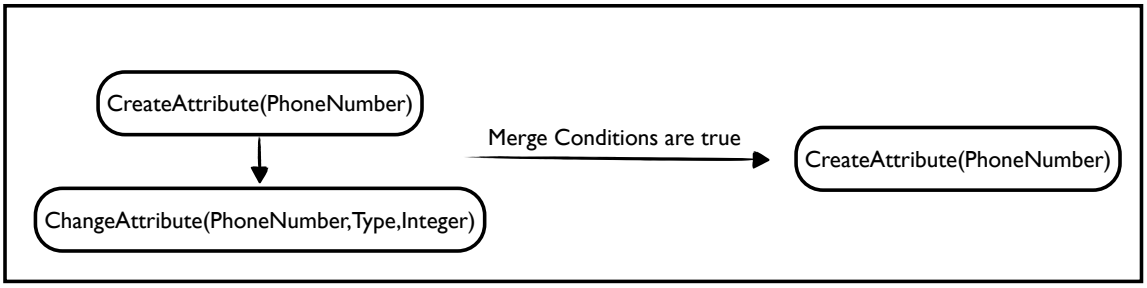

Figure 5.6: Merge Example

Figure 5.7 depicts the second scenario where both migrations are removed. As we see, the *PhoneNumber* attribute is created and later removed from the database model. Assuming that the conditions to merge are valid, after dropping the attribute *PhoneNumber* the migrations are compared. Because the attribute does not exist anymore in the database both migrations are removed from the model since they are no longer necessary.

Figure 5.8 represents what happens if two consecutive kinds of operations related to the same entity attribute are compared. According to the matrix the operations in the left column come at first, followed by an operation of a different kind on the top of the matrix.

Merge Conditions In order to merge migrations we take into account the **developer** currently modifying the database, a time interval with the duration of 1

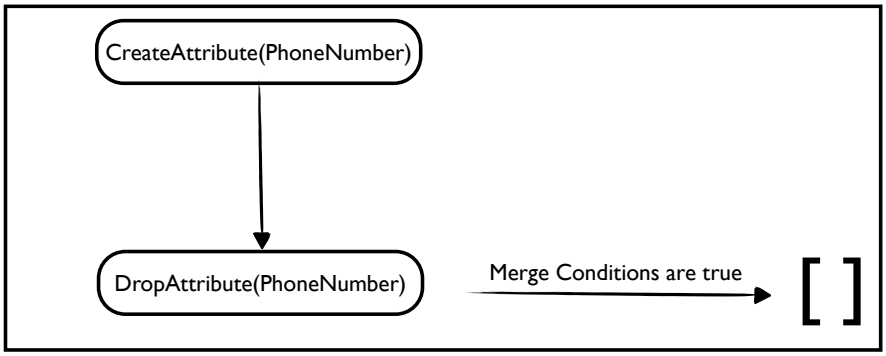

Figure 5.7: Merge Example

1st \ 2nd	CreateAttribute	DropAttribute	RenameAttribute	ChangeAttribute	UpdateAttribute
CreateAttribute	-	CreateAtribute	CreateAtribute	CreateAtribute	Not Merge
DropAttribute	Imp	-	Imp	Imp	Imp
RenameAttribute	Imp	None	-	Not Merge	Not Merge
ChangeAttribute	Imp	None	ChangeAttribute	-	Not Merge
UpdateAttribute	Not Merge	Possible	Not merge	Possible	-

Imp - Impossible to merge the kind of migrations

Possible - It is possible to merge if the attribute related to migrations is not used in a previously upgrade rule.

➡The other cells contain the remaining migrations after the merge process.

Figure 5.8: Merge between two consecutive migrations.

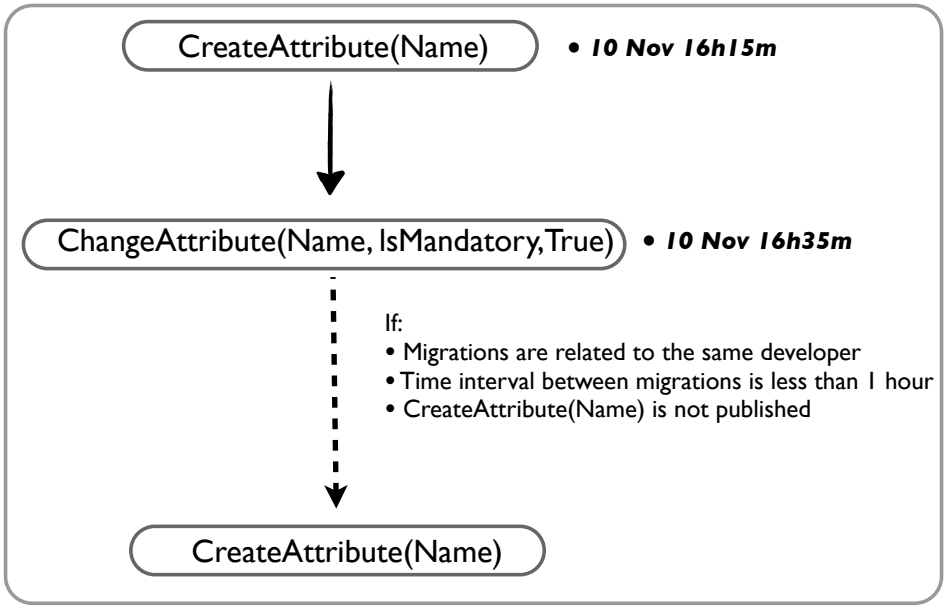

Figure 5.9: Merge Conditions Example

hour and if the migration that could possibly merge with the new change is published in the development environment or not. Also only migrations related to the same attribute are merged.

For a question of traceability we decided not to merge migrations when different developers are involved. If some problems occur after the database model was changed we know who and when changed the database model and to which attribute are related the transformations. Merging migrations related to different developers could also be a possible approach. However, it would not be possible to track the responsible for each migration, because we do not know who was responsible for the modification.

Another parameter involved in the merging of migrations is the time interval between the new operation in the database model and the older one that could possibly merge. Taking into account the time interval, avoids the generation of migrations for each change that developers do every minute for the database schema. For example, considering a *CreateAttribute* operation followed in a short time interval by some *ChangeAttribute* migrations related to the same attribute. Because migrations were generated in a short time interval, instead of creating the attribute and then to change it, the attribute can be created with the value of the properties changed afterwards. Thus, we defined that one hour would be a fair time interval on which migrations could merge. For instance in the scenario that the developer created a new attribute and 20 minutes after he decides to

drop it. In this case would be generated two migrations, one when creating the attribute and the other one when dropping the same attribute. With our approach both migrations are deleted from the data model, thus avoiding two unnecessary operations in the database.

Taking the same scenario of the last paragraph, when the developer drops the attribute and the time interval since the creation of the same attribute is less than one hour, if the migration representing the creation of the attribute was already **published** on a different environment (Development or Production) it is not possible to perform the merge and we keep both migrations. The next time the application will be published the attribute is deleted. Thus, is not possible to merge migrations belonging to different environments because the changes were before applied to the database. Also, for a question of traceability to know which changes are applied in the different environments.

5.1.2 Commutativity of migrations

We also explore in the merge of migrations the commutativity of the operations. Two migrations are commutative if independently of their order, they will produce the same result. The commutativity of migrations is useful for the merge process in order to know if the previous operations are suitable to merge. The next two examples exemplify how the commutativity of migrations works. On the first one, are created the attributes *Name*, *PhoneNumber* and *Email*. After that, the *Email* attribute is updated with the value that it should contain and finally the attribute *PhoneNumber* is dropped. The *DropAttribute(PhoneNumber)* is compared with all the previous operations until it can merge with other migration or until it can not commutate with a previous operation. In this case, the *CreateAttribute(PhoneNumber)* and *DropAttribute(PhoneNumber)* are merged and both removed from the model.

$$
\begin{aligned}
&CreateAttribute(Name);\\
&CreateAttribute(Email);\\
&CreateAttribute(PhoneNumber);\\
&UpdateAttribute(Email, Name + \text{"@}outsystems.com\text{"});\\
&DropAttribute(PhoneNumber)
\end{aligned}
$$

In the second scenario two attributes are created: *FirstName* and *LastName*. Then, an expression is defined to update the value of both attributes, using an existing attribute *Name*, and later we drop the attribute *Name*. Below we see the sequence of the operations:

$CreateAttribute(FirstName)$;
$CreateAttribute(LastName)$;
$UpdateAttribute(FirstName, split(Name)[0])$;
$UpdateAttribute(LastName, split(Name)[1])$;
$DropAttribute(Name)$

On this case, the *DropAttribute(Name)* operation is not commutative with the previous operation because the *Name* attribute needs to exist in the database model, due to the fact that is used in the expression of the *UpdateAttribute* operation immediately before. So, the operations are only commutative if does not exist any previous migration using the same attribute in some expression. On this specific example two expressions are using the attribute *Name*, thus, the attribute needs to exist in the time interval before it is dropped.

5.1.3 Impact Analysis on Production Environment

As we referred before, one of the problems we want to solve with our solution is the feedback that the user has about the database model and its data already in production. The analysis is intended to be on demand, i.e., the developer changes the model in *Service Studio* and the platform performs the impact analysis in the production environment, in order to give the feedback about the impact of such change in the existent model in the production environment. While the system is analysing the model in the production environment the developer continues to work and to change the application as he wants. After that and if the changes that he made have impact on the real data in production, the developer receives a warning explaining the conflicts between the new version of the model and the one in production.

Figure 5.10 shows the relevant component of the *Agile Platform* involved on this process. It also shows the flow of the impact analysis process.

Development Environment - After the developer changes the model, the *Service Studio* contacts de *Service Center* of the development environment in order to get the impact of the change in the production environment. If the relevant information is cached in the *Service Center* it will be returned to the *Service Studio* and if that is the case a warning is presented to the developer. If the information is not cached the *Service Center* contacts the *LifeTime* component to get the information about what is in production.

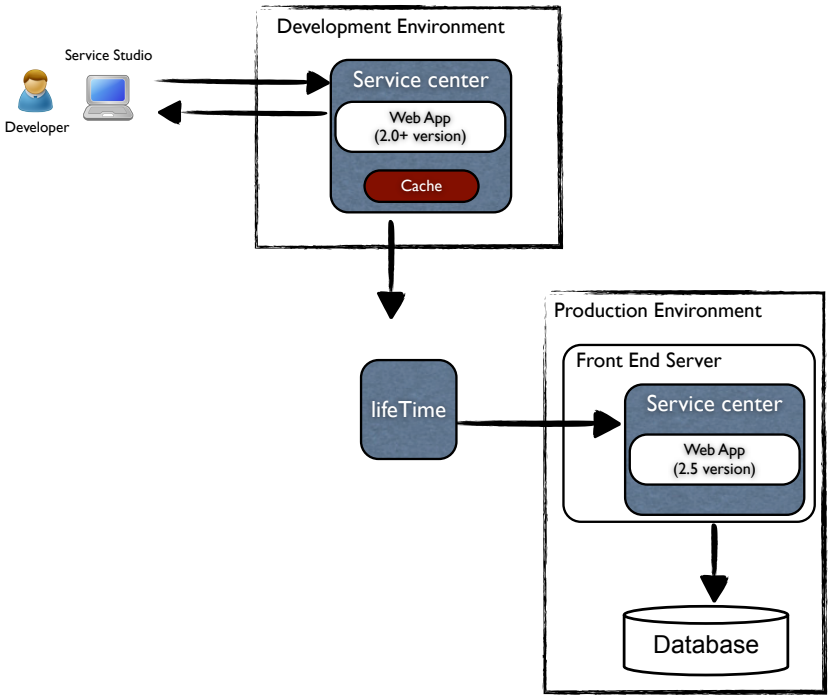

Figure 5.10: Relevant architecture components for Impact Analysis

LifeTime - Receives the information from the development environment *Service Center* and after that contacts the production environment *Service Center* in order to get impact analysis about the database model in production. After receiving the answer from the *Service Center* in the production environment, this component returns that to the *Service Center* in the development environment.

Production Environment - The *Service Center* of the production environment receives the information from *LifeTime* about the changes made by developer in *Service Studio*. Then, the database is analysed in order to know the impact of the change. For example, if the developer changed the property *IsMandatory* of an attribute, the relevant information that *LifeTime* gets is if exist NULL values associated to that attribute. If the developer changed the type of an attribute from Text to Integer, and if exist values not compatible with the Integer type, is returned to *LifeTime* that exist values related to the attribute involved in the change that are not compatible with the new type of the attribute.

Production Warnings After changing the database model and the impact analysis of the changes in the read data in production is finished, the developer may or may not receive a warning to fix incompatible data existent in the model in production. Figure 5.11 shows a *Service Studio* screen before the developer changes

the database model. For example, if in Figure 5.11 the developer changes the property *IsMandatory* of the SSN attribute, the system begins the impact analysis process that we explained before. If the change does not affect any data in the production environment the state of the eSpace continues valid. In this case if does not exist any NULL values related to the SSN attribute no warning is presented to the developer. On the other hand, if exist NULL values associated to the attribute SSN, a **Production Warning** is generated. Figure 5.12 shows the case when the developer gets that type of warning.

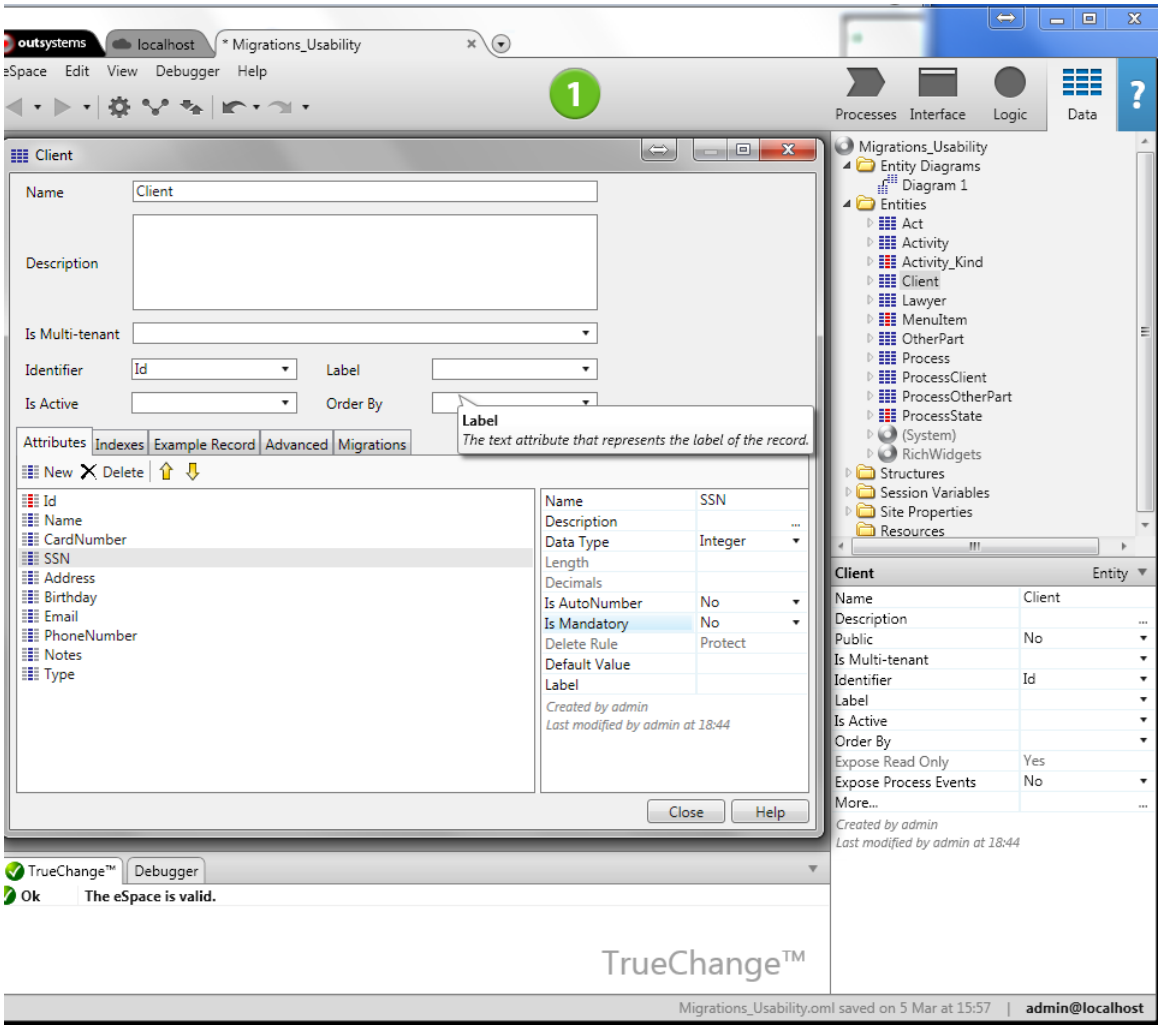

Figure 5.11: Before changing the attribute

Upgrade Rules When the developer gets a Production Warning means that something in the database needs to be fixed. In our solution the developer can write an expression or define a value that is used to fix the incompatible data by

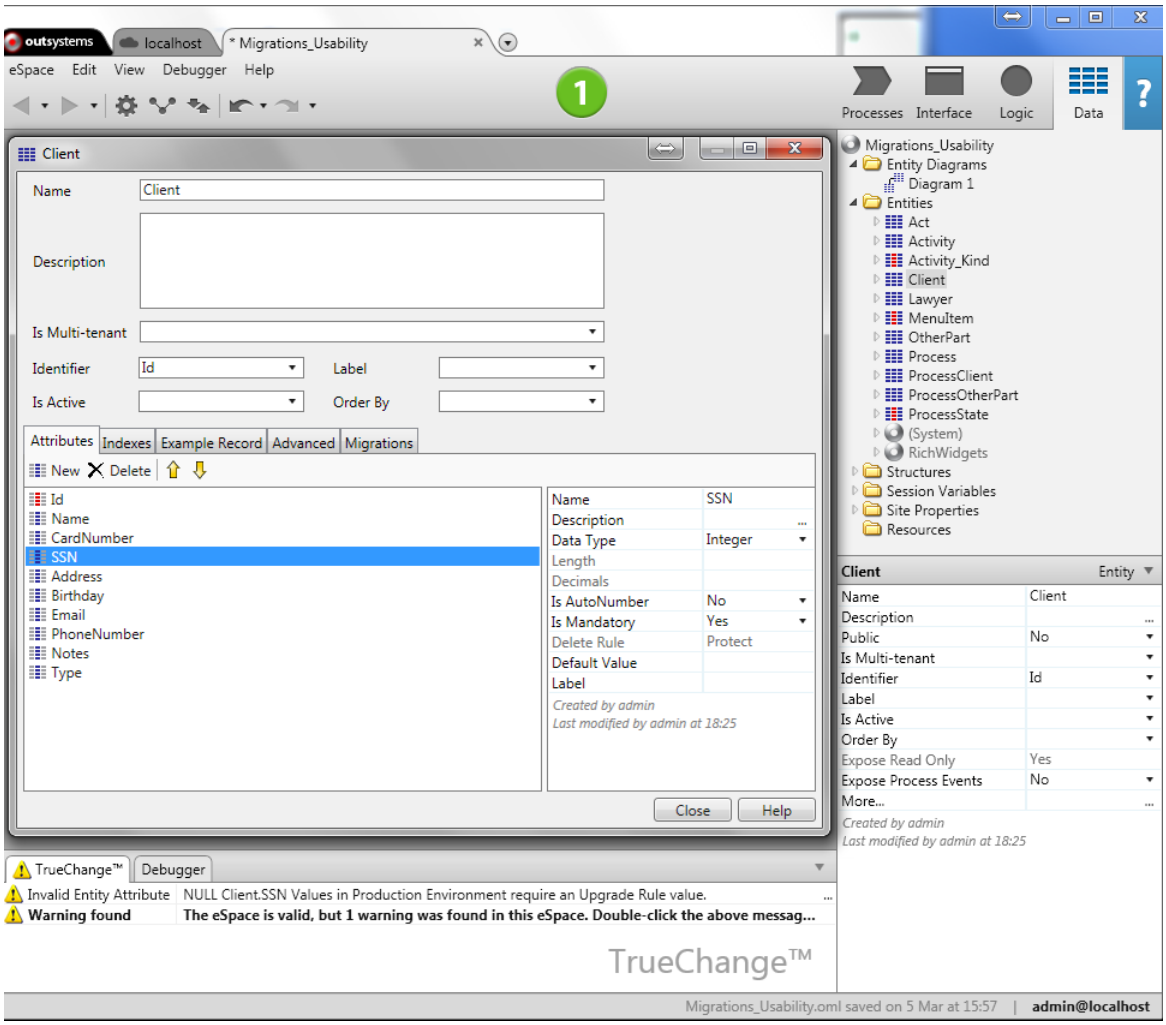

Figure 5.12: Production Warning Example

default. In figure 5.12 is depicted an example where the developer specified a default value to fix the rows where the attribute value is NULL, since the developer changes the IsMandatory property of the attribute SSN to Yes and exist NULL values in the production environment database for that field. The developer has also the possibility to write an expression using values from other attributes or some function available in *Service Studio* expressions. The environment of those expressions are:

- The same elements available to a *Simple query* scope, i.e. , local variables, Entities used in a query and built-in functions;
- Attributes created and available by previous migrations until the moment where the developer specifies the upgrade rule.

5.2 Deploy into Production

When the applications are ready to be deployed into production, an operator is responsible for that task. With our solution, before deploying the application the developer has available a list with all the database model changes made by the developers. The visible migrations are the migrations in the version to be deployed that are older than the version in production, so the operator cannot see the pending migrations. This information is helpful for the operator because in the case that something goes wrong with the database model in the deployment process, the operator sees who is responsible for that and also knows which migration failed. The operator is not from *Outsystems* and is important to know who was responsible for the database modification because if something goes wrong during the deployment process the operator can easily understand who and why the changed was made. In Figure 5.13 are depicted the *Agile Platform* components involved on this process. The operator in *LifeTime* deploys the applications into production. Then, is contacted the *Service Center* of the production environment. This component is responsible to communicate with the *Deployment Controller Service*, that on its turn starts the compilation process and updates the database.

5.3 Deprecated Data

With our solution the data is not permanently deleted from the database, because data may be later necessary to recover the application and usually enterprises need to keep the data for a given time interval. When an attribute is dropped from

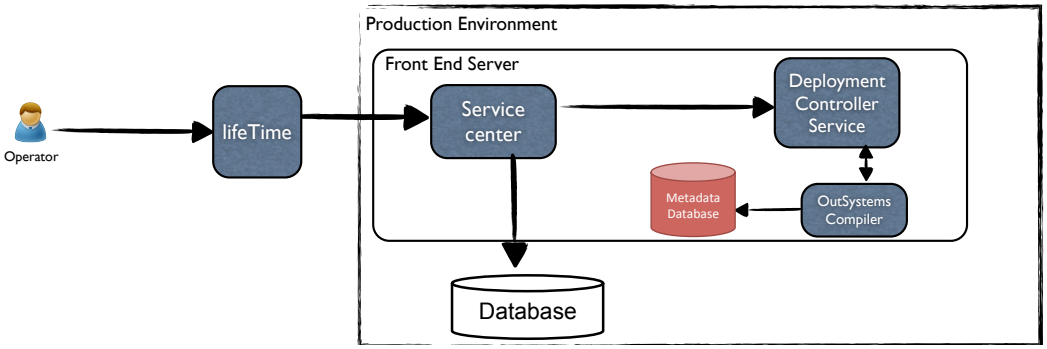

Figure 5.13: Relevant Architecture component to deploy the application into production

the database, instead of deleting the data related to the attribute, it is marked as deprecated data and saved for a time interval defined in the server. For example, a *Name* attribute is deleted from an Entity *Client*, as depicted in figure 5.14, a new table is created in the database containing the attribute value and the reference to the entity *Client*.

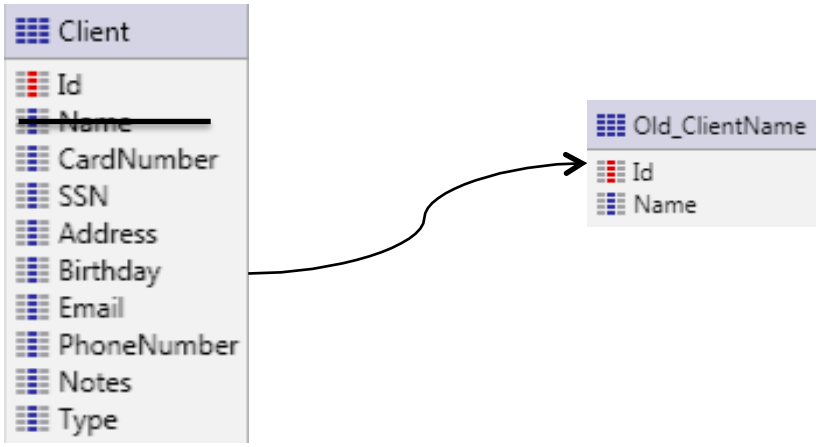

Figure 5.14: Deprecated Data

6

Implementation

In this chapter, we describe the design and implementation of a prototype in *Service Studio* for the solution model previously presented. At first, is described how migrations are integrated in the application model. After that, is explained how migrations are generated and migrations are applied throughout the different environments, i.e., the development and production environments. Then, it is described how the merging process is implemented within *Service Studio* and after that, how production warnings are presented to developers and what they can do to fix it.

When a web application is created with *Service Studio* it generates Microsoft .NET [Mic13] or JAVA [Sun13] code, depending on the target application server. In this work, we defined changes in the generation of .NET code. Our approach can also be adapted to the JAVA language.

6.1 Migrations Object Model

In order to capture the transformations made by developers to the database schema, the objects definition of the platform was extended to support migrations. Listing 6.1 shows how objects are defined. A migration is a child of an *Entity*. An *AbstractMigration* represents a generic kind of migration with properties shared by all the sub migrations extending the generic one. The objects are defined in XML, being then generated to C# [HWG03, HTWG10] classes representing them.

To represent operations referred in Chapter 5 four kinds of objects were defined, extending the *AbstractMigration*: ***CreateAttribute*** captures the creation of an entity attribute. The values of its properties are cloned from the entity attribute itself and whenever the attribute is changed, the migration properties are also updated. Here we may have two situations: *a)* If the attribute is changed and the conditions to merge the generated migrations are valid, the properties in the *CreateAttribute* migration are updated with the new attribute properties values; *b)* On the other hand, if the conditions to merge are not valid, a ***ChangeAttribute*** migration is generated. This kind of migration saves the attribute property that was set, its value and also allows the developer to specify an upgrade rule to fix bad data, if that is the case. This option is also available when creating the attribute. To capture the renaming of an entity attribute is defined the ***RenameAttribute*** migration and this kind of migration saves the attribute new name. With the ***UpdateAttribute*** object we allow the developer to specify an upgrade rule (explain later on this Chapter) for a certain attribute. Finally, the ***DropAttribute*** object captures the action of dropping an entity attribute. In the next section is shown how migrations are generated in *Service Studio*.

Listing 6.1: Migrations Object Model Example

```
<AbstractReferenceableEntity name="Entity" ...>
      ...
      <Children>
        <Child type="AbstractMigration" ... />
      </Children>
  </AbstractReferenceableEntity>

  <AbstractObject name="AbstractMigration" isAbstract="true" >
    <Properties>
        <Property name="Developer" ...  />
        <Property name="Version" ...  />
        <Property name="Timestamp" ...  />
        <Property name="HiddenTimestamp" ... />
        <Property name="AttributeName" ... />
    </Properties>
  </AbstractObject>

<AbstractMigration name="CreateAttribute" >
  ...
    <Properties>
      <Property name="Type" ... />
      <Property name="IsAutoNumber" ... />
      <Property name="Label" ... />
```

```
24          <Property name="Length" ... />
25          <Property name="IsMandatory" ... />
26          <Property name="DefaultValue" ... />
27        </Properties>
28    </AbstractMigration>
29
30    <AbstractMigration name="RenameAttribute">
31      <Properties>
32          <Property name="NewAttributeName" />
33        </Properties>
34    </AbstractMigration>
35
36    <AbstractMigration name="DropAttribute">
37        ...
38    </AbstractMigration>
39
40    <AbstractMigration name="UpdateAttribute" >
41      <Properties>
42        <Property name="NewAttributeValue" />
43        </Properties>
44    </AbstractMigration>
45
46    <AbstractMigration name="ChangeAttribute">
47      <Properties>
48        <Property name="AttributePropertyNewValue" />
49        <Property name="DefaultValue" />
50      </Properties>
51    </AbstractMigration>
```

6.2 Generating Migrations

As referred before in Chapter 5, when developers change the database model, migrations are automatically generated in order to represent those changes. In this section is shown by mean of examples how the developer sees generated migrations while changing the database model. In *Service Studio*, is available an entity editor where developers can edit the existing entities in the model. To show the developers interactivity with *Service Studio* we use the database model created during the first stages of the thesis (Chapter 4) and already published. Figure 6.1 depicts *Service Studio* interface with the entity editor open. As we can see, the developer can add, delete, and edit each attribute properties. The figure shows the existing attributes for an entity *Client*, and we want to add the new attributes,

PhoneNumber and *Email*, to it. Those changes are represented in the second *Service Studio* screen in Figure 6.1

After that, the developer decided that the attribute *Notes* is no longer needed and drops it. Also the attribute *CardNumber IsMandatory* property was set to *Yes* meaning that the attribute cannot allow NULL values and for simplicity in this case, defined a default value "123456789" for the NULL rows in the database. Figure 6.2 depicts the final state of entity *Client* after the developer transformations. Thus, on this case four migrations are generated to represent the changes made by the developer to the entity *Client*.

Figure 6.3 represents the entity *Client* editor in migrations tab. As depicted, the last migrations generated represent the most recent entity changes and they are in the top of the list as pending to publish to the development environment. The most relevant aspect to retain on this section is that changes to each entity generate automatically migrations and those migrations are set to pending until they are published. The next section shows and explains how migrations are separated by the different environment, i.e., the development and production environments.

6.3 Migrations through different environments

After migrations are generated its state is pending to publish. While they are not published, the database remains with the same state. Migration have three possible states:

- Pending;
- Already applied in development;
- Already applied in production.

As we see in Figure 6.3, the last changes made by the developer to the entity *Client* stay as pending until they are published in the development environment. In this case, the entity *Client* was already published at least one time, since existing migrations applied in the development environment as depicted in Figure 6.3. With this kind of separation, developers control what is already applied in the different environments and what is changed and pending to publish to the development environment. It is important to refer that when published, migrations next state is *"Already applied in development"* and after that when deployed into production its state change to *"Already applied in production"*. This represent

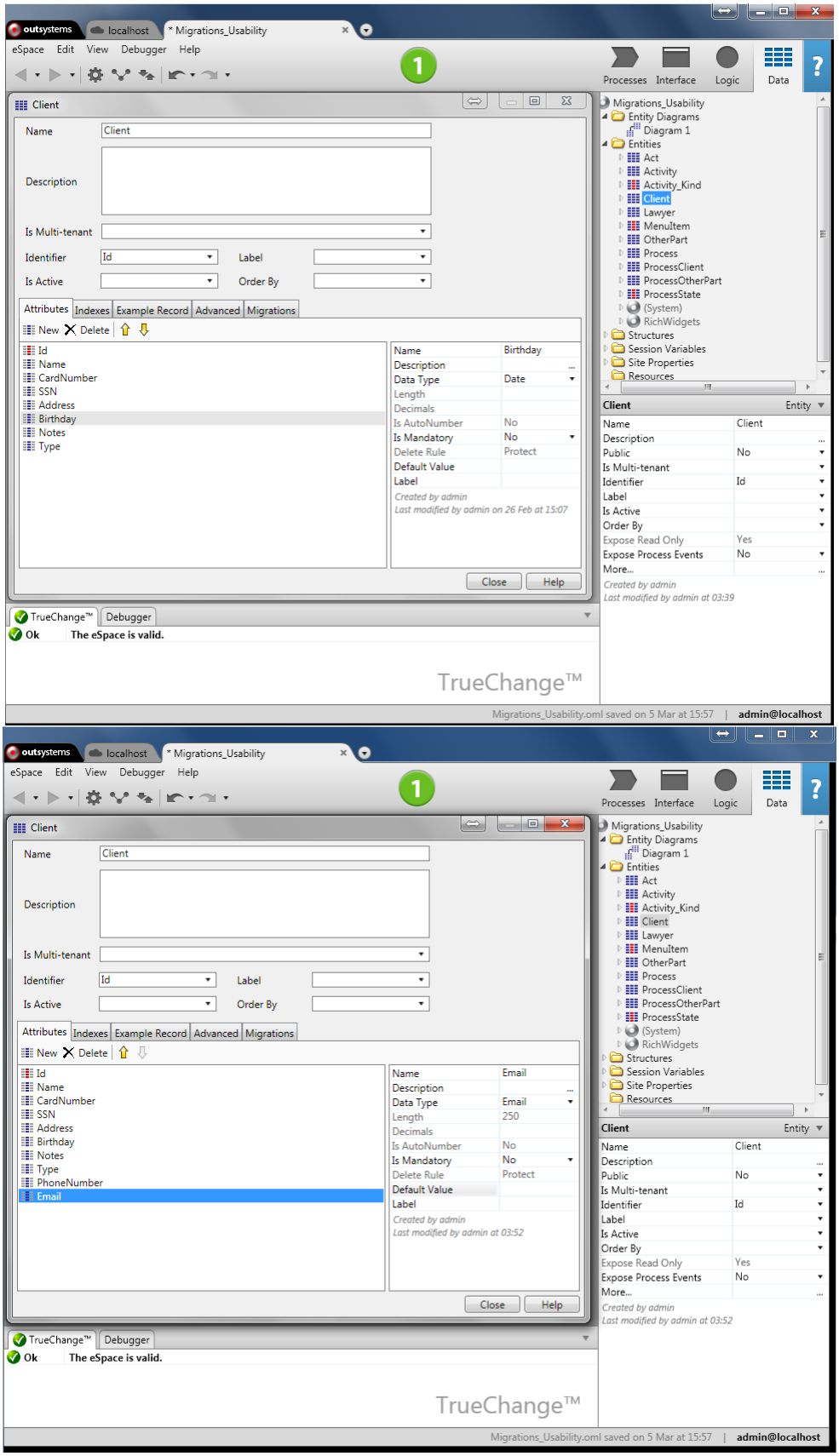

Figure 6.1: Service Studio: Entity Editor.

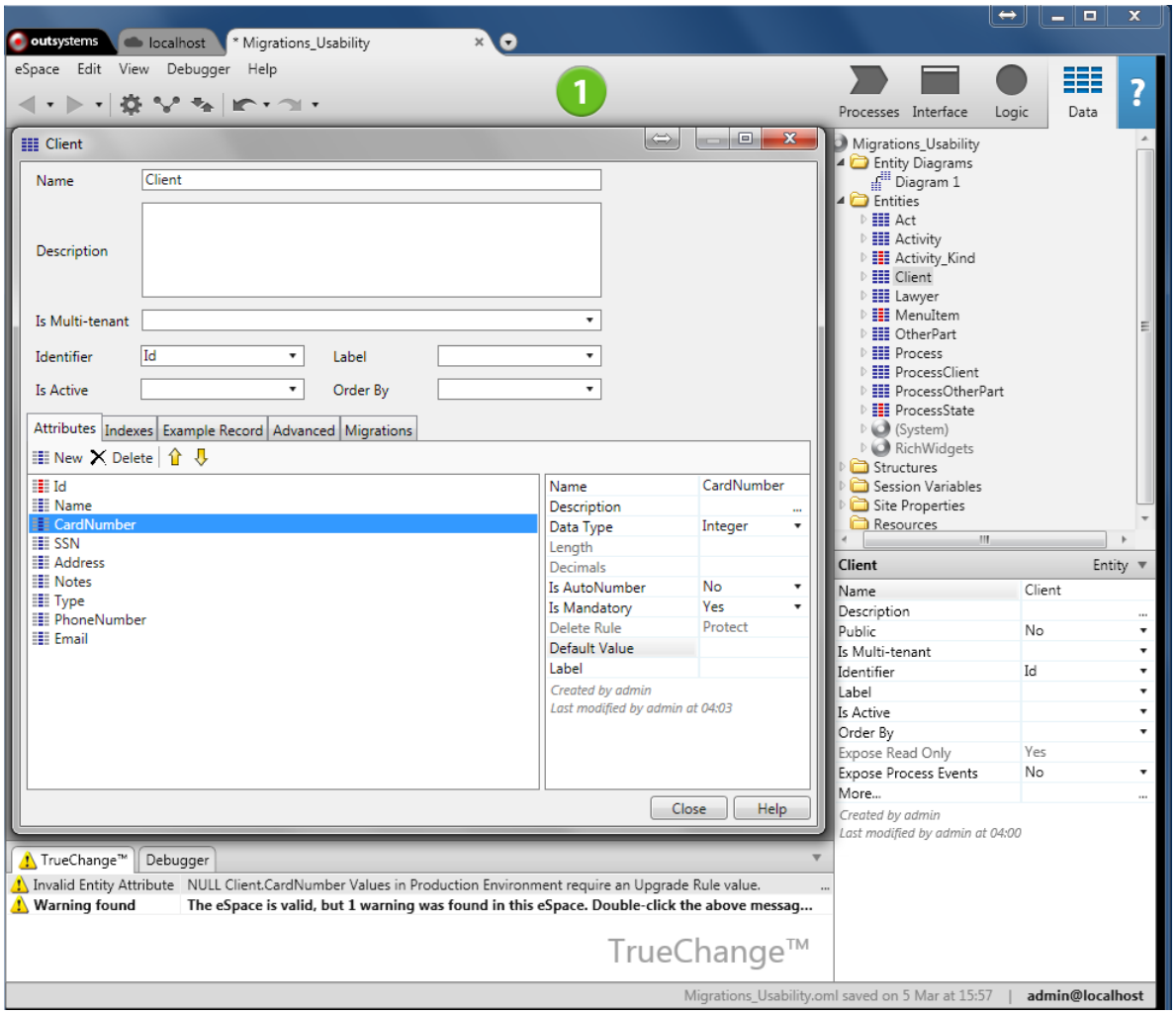

Figure 6.2: Service Studio: Entity Editor with final state of entity *Client*.

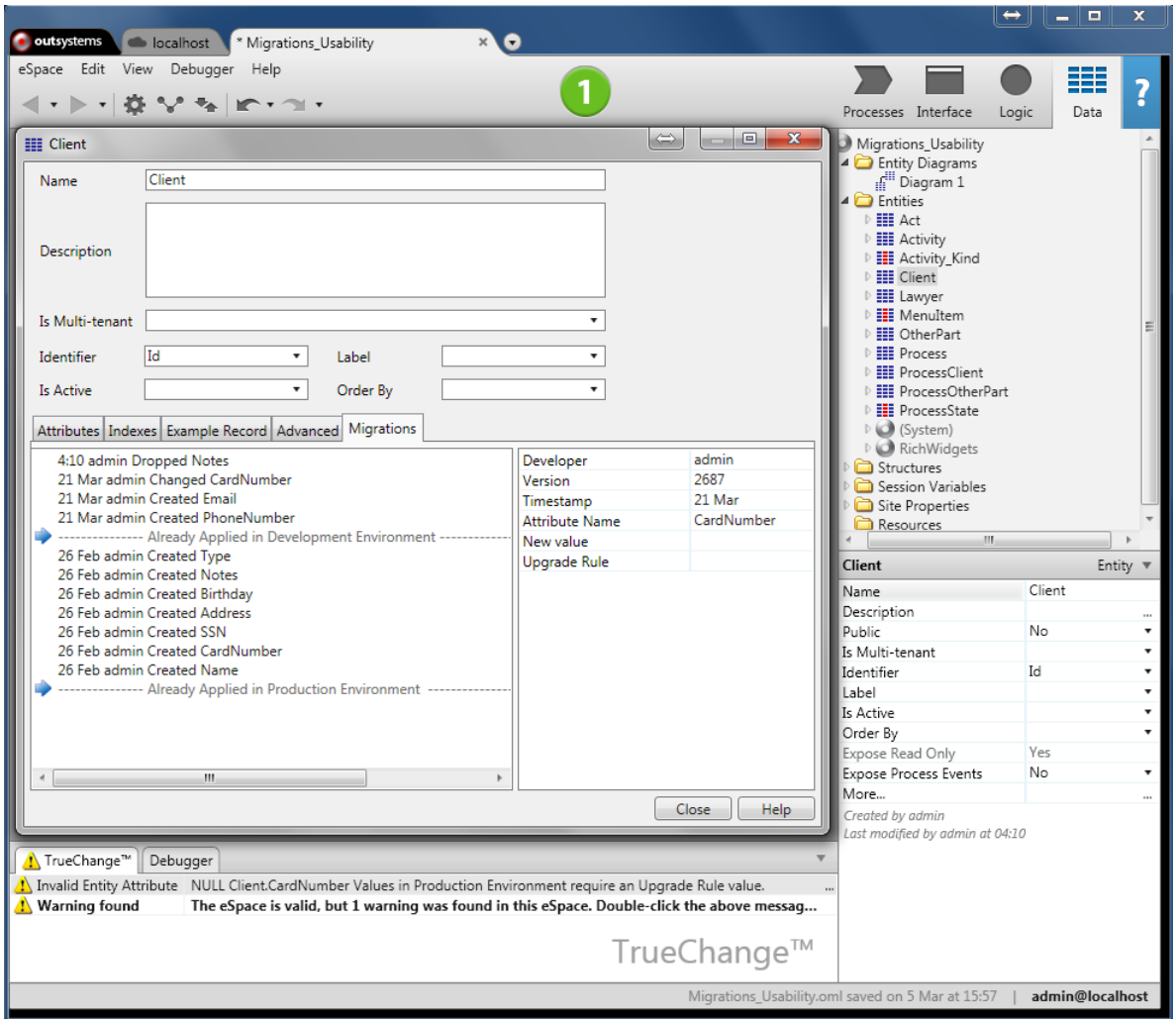

Figure 6.3: Service Studio: Generated migration for the changes on entity *Client*.

the flow of an application between the different environments. In order to know for each application stored in the server which are the running development and production versions of the application was created a Dictionary in the server class that stores for each *ESpace* published in *Service Center* the respective development and production versions.

6.4 Merging Migrations

After a migration is generated in *Service Studio*, we always check if it is possible to merge with other migrations associated to the same entity. This process follows the proposed approach on the model solution. When the developer changes an entity attribute and before adding the migration to the model, the system checks if the condition to merge are valid. Listing 6.2 presents a pseudo code showing the merge algorithm. The algorithm searches in the list of migration associated to an entity, if some previously created migration is related to the same attribute than the new one generated. In that case, the algorithm checks if the conditions to merge are valid:

- If both migrations are generated in the time interval of one hour (as explained in Chapter 5 we defined this time interval. The code can be adapted if we define another time interval as acceptable);
- If the developer that changed the attribute is the same for both migrations;
- And if the oldest migration is not yet published in the development environment.

If the conditions are valid the old migration is returned and after that its properties updated according to newest change.

Listing 6.2: Merge Algorithm

```
1
2  if (entity.Migrations.Count > 0) {
3
4    foreach (AbstractMigration m in Migrations) {
5
6      if (m.AttributeName == entityAttribute.Name) {
7
8        if (
9
10       (DateTime.Now.Month==Convert.ToDateTime(m.HiddenTimestamp).Month)
```

```
11        && m.Developer == entityAttribute.LastModifiedBy &&
12        (DateTime.Now.Day==Convert.ToDateTime(m.HiddenTimestamp).Day) &&
13        (DateTime.Now.TimeOfDay.Subtract(m.Timestamp).TimeOfDay)).Hour< 1
14        && !m.IsPublished ) {
15
16          mig = m;
17        }
18      }
19    }
20  }
21  return mig;
```

6.5 Production Warnings

Due to access limitations to the real data in a production environment, on this prototype we simulate the behaviour of the application when changes made by the developer to the database model have impact on the real data. Thus, the prototype focus on the developer experience rather than the connection to the production environment. Figure 6.4 depicts *Service Studio* after the developer changed an entity *Client* and change the *IsMandatory* property of the field *SNN* to *Yes*. After that, and because that kind of transformation has impact on the data in production, a *Production Warning* is generated immediately. The description of the warning contain the information about what is the type of warning, to which entity and entity attribute is related. It is important to refer that developers can publish applications even if they have warnings. The warning just advises the developer to fix something than will create incompatibilities. Developers are free to keep applications with warning in early stages of the development process and later to fix the warnings.

Upgrade Rules To fix the *Production Warnings* developers can write expressions that we call upgrade rules. When creating, changing or updating an entity attribute developers have the possibility to write those rules. In Chapter 5 we referred what the environment when writing an upgrade rule contains and here we show an example how developers write upgrade rules within *Service Studio*. Upgrade rules are not only useful to fix bad data, once developers can write expressions to move data from a specific column to another. Figure 6.5 depicts the scenario where the developer creates a new attribute *Name* and existing in the model are the attributes *FirstName* and *LastName*. The developer wants to migrate the data from those attribute and to have in the future only the

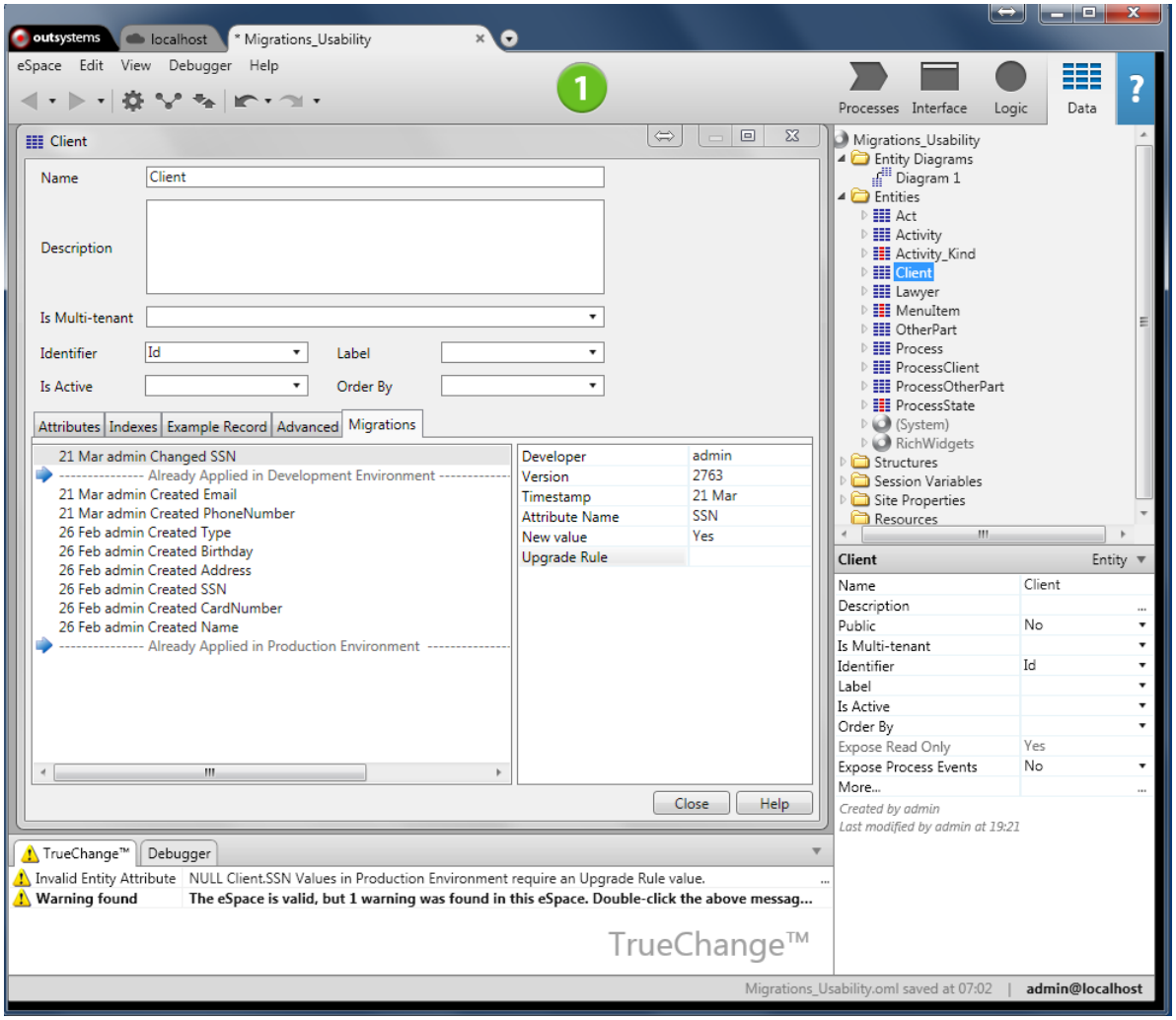

Figure 6.4: Service Studio: Production Warning.

attribute *Name*. To accomplish that the developer specifies the following rule : $FirstName + "" + LastName$. Figure 6.5 shows a warning in the $TrueChange^{tm}$ tab. To solve that the developer just needs to also specify an upgrade rule to fix the bad data related to the *SSN* attribute in the production environment.

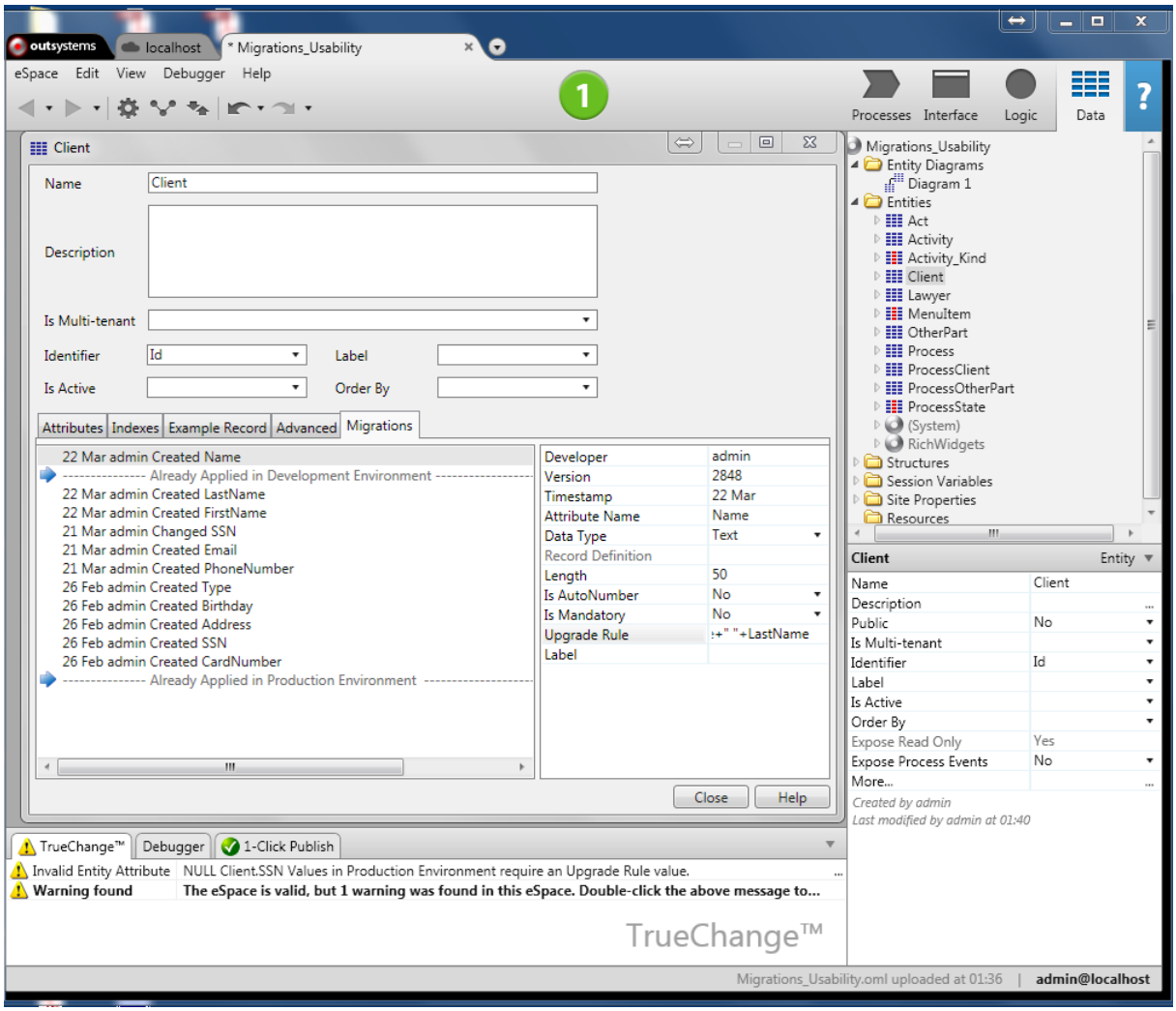

Figure 6.5: Service Studio: Specifying upgrade rule for an entity attribute

7

Final Remarks

This thesis is integrated in the Research and Development (R & D) team of the *OutSystems* company. The thesis was divided in two different phases.

The early stages of the project were dedicated to learn about the OutSystems product, the *Agile Platform*. Our focus was to know and understand difficulties, problems and scenarios faced by developers when evolving web applications in the context of the *Agile Platform*. After that, we interviewed experienced developers and project managers in order to capture frequent database change patterns and to know the most common scenarios when developers are evolving applications database model. After that, we studied and researched about database refactoring, database migrations, database change patterns and also tools or programming languages implementing these concepts.

The first phase occupied 40% of the time available and was already in collaboration with the company and also with the support from the University.

After studying methods and solutions related to our problem, the second stage of the project consisted on designing a solution model based in the concept of migrations and implementing a prototype in *Service Studio*. Both project stages followed the OutSystems agile methodology based on SCRUM agile methodologies, for control and organisation of projects.

In the second phase, we started by designing iteratively the solution model. At first, we focused on the developer experience, in order to define how to integrate migrations in the development of applications with *Service Studio*, without

compromising the simplicity of using the tool. Then, we defined the other components and features of the solution model, such as the object model, properties for each kind of migration, the merge process, migrations commutativity and also how the impact analysis of the real data in production should be done. After that, we built a prototype in *Service Studio*, following the concepts of the solution model.

The *Agile Platform* is formed by a set of more than 70 projects developed in Microsoft Visual Studio with a code base of more than 1 million lines. Due to the platform complexity, dependencies and time constraints, we focused on defining the solution model and to implement a prototype focusing in the developer experience. Thus, not all the features composing the solution model were implemented through all the components of the *Agile Platform*. So, in order to be fully integrated with the *Agile Platform* the prototype needs more work time and iterations. Nevertheless, the implemented prototype reproduces the key features of the solution model, such as the object model, migrations generation, merging process and we also simulate the impact analysis by producing the warning according to the changes in the database model and allowing the developer to write the upgrade rules to fix those warnings. The prototype helps the *OutSystems* product management to have a concrete vision of our solution usability and functionality, reducing the integration risk through all the *Agile Platform* components. To fully implement our solution throughout the *Agile Platform* components, would cost about three months work and require an experienced team in the *Agile Platform*.

7.1 Future Work

As for the future, we aim to solve limitations and improve our solution. To improve our solution expressiveness could be defined more kinds of migrations capturing modifications to the database model and schema. The merge process could be addressed differently, i.e., the conditions and the constraints to merge migrations could be different. For example, merging migrations created by different developers or migrations belonging to different environments. Also for future work, our prototype could be extended in order to fully integrate our solution in the *Agile Platform*. For example, translating migrations to SQL queries, implementing the impact analyzis mechanisms to connect the development environment to *LifeTime* and the production environment and also implementing the cache mechanism, are features to be implemented in the future of this work.

Bibliography

[AS06] Scott W. Ambler and Pramodkumar J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[Che76] Peter Pin-Shan Chen. The entity-relationship model toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[CMDZ10] Carlo A. Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++. *Proc. VLDB Endow.*, 4(2):117–128, November 2010.

[CMZ08] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008.

[DNR08] Alin Deutsch, Alan Nash, and Jeff Remmel. The chase revisited. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '08, pages 149–158, New York, NY, USA, 2008. ACM.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, July 1999.

[GHKV08] Danny M. Groenewegen, Zef Hemel, Lennart C.L. Kats, and Eelco Visser. Webdsl: a domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN conference on*

Object-oriented programming systems languages and applications, OOPSLA Companion '08, pages 779–780, New York, NY, USA, 2008. ACM.

[HTWG10] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *C# Programming Language*. Addison-Wesley Professional, 4th edition, 2010.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Mic13] Microsoft. Msdn .net development website. http://msdn.microsoft.com, 2013.

[SBB04] Dennis Shasha, Philippe Bonnet, and Nancy Hartline Bercich. Database tuning principles, experiments, and troubleshooting techniques. *SIGMOD Rec.*, 33(2):115–116, June 2004.

[Sjo93] D Sjoberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35 – 44, 1993.

[SKS10] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, http://books.google.pt/books?id=re4YQAAACAAJ, 2010.

[Sun13] Sun. Java 2ee. java.sun.com/javaee, 2013.

[VWV11] Sander Daniël Vermolen, Guido Wachsmuth, and Eelco Visser. Generating database migrations for evolving web applications. *SIGPLAN Not.*, 47(3):83–92, October 2011.

8

Appendix

Table 8.1: Structural Refactorings

Database Refactorings
Drop Column
Drop Table
Drop View
Introduce Calculated Column
Introduce Surrogate Key
Merge Columns
Merge Tables
Move Column
Rename Column
Rename Table
Rename View
Replace LOB With Table
Replace Column
Replace One-to-Many With Associative Table
Replace Surrogate Key with Natural Key
Split Column
Split Table

Table 8.2: Data Quality Refactorings

Add Lookup Table
Apply Standard Codes
Apply Standard Type
Consolidate Key Strategy
Drop Column Constraint
Drop Default Value
Drop Non-Nullable Constraint
Introduce Column Constraint
Introduce Common Format
Introduce Default Value
Make Column Non-Nullable
Move Data
Replace Type Code With Property Flags

Table 8.3: Referential Integrity Refactorings

Add Foreign Key Constraint
Add Trigger for Calculated Column
Drop Foreign Key Constraint
Introduce Cascading Delete
Introduce Hard Delete
Introduce Soft Delete
Introduce Trigger for History

Table 8.4: Architectural Refactorings

Add CRUD Methods
Add Mirror Table
Add Read Method
Encapsulate Table With View
Introduce Calculation Method
Introduce Index
Introduce Read Only Table
Migrate Method From Database
Migrate Method to Database
Replace Method(s) With View
Replace View With Method(s)
Use Official Data Source