Insert here your thesis' task.

Czech Technical University in Prague

Faculty of Information Technology

Department of Software Engineering

Bachelor's thesis

Migration of relational databases using CodiScent's Projective Technologies

Christián Golian

Supervisor: Mgr. Ondřej Dvořák

5th May 2015

Acknowledgements

I would like to express my deepest thanks to my supervisor Mgr. Ondřej Dvořák for his valuable advice and frequent consultations. I would also like to thank my parents for their continuous support and encouragement.

Declaration

In Prague on 5th May 2015 ....

Citation of this thesis

Golian, Christián. *Migration of relational databases using CodiScent's Projective Technologies.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Tato práce se zabývá vývojem a implementací nové metody migrace schematu relační databáze. Nová metoda migrace a nový migrační nástroj jsou založeny na Projektivních Technologiích, které jsou vyvíjeny společností Codiscent. V teoretické části jsou definovány pojmy jako migrace schematu a migrační nástroje. Praktická část se zabývá analýzou, návrhem a implementací nového migračního nástroje MigrationGen. V závěru jsou vysloveny závěry o efektivitě této metody a o vhodnosti používaní generování kódu k migraci databáze.

Klíčová slova migrace schématu, relační databáze, Projektivní Technologie, reverzní inženýrství, generativní programování

Abstract

This thesis deals with development and implementation of a new method of schema migration. The new migration method and the new migration tool are based on Projective Technologies which are developed by the company Codiscent. The theoretical part defines terms such as schema migration and schema migration tools. The practical part deals with analysis, design and implementation of the new migration tool MigrationGen. Finally, conclusions are drawn about the effectiveness of this method and about the suitability of using code generation for database migration.

Keywords schema migration, relational databases, Projective Technologies, reverse engineering, generative programming

Contents

List of Figures

List of Tables

Introduction

Motivation

Today, source code version control is an integral part of every project. The question is not whether to use a software tool for version control, but which one to use. Its benefits are acknowledged and it is considered to be a standard in an enterprise world.

Database versioning brings all the advantages which source code version control has. It tracks evolution of the project. It backs up code. It enables simpler sharing between developers. It visualizes changes between different versions easier. It allows restoring a database at a given point in time. It is not as common and widespread as source code versioning, yet equally important.

Schema migration tools deal with relational database versioning. When relational database needs to be updated to a newer, or reverted to an older version, schema migration (or database migration) is performed. This consists of applying a sequence of scripts.

These scripts can be written manually or generated by another application. The second option is better because it saves time and cost and raises quality by reducing the possibility of a human error. There are different approaches to source code generation: from reverse engineering to generative programming. CodiScent[1] is a company which is developing tools for *generative engineering*. These tools are known as Projective Technologies and they include Generative Engineering Studio (GES) and Reverse Engineering Studio (RES).

The motivation for this thesis is to investigate if tools for schema migrations developed on top of Projective Technologies can benefit from advantages brought by code generation.

[1]`http://codiscent.com/`

Goals

The ultimate goal of this thesis is to propose a new method for database migration and to develop a new database migration tool based on it.

MigrationGen should be better (at least in some regards) than similar migration tools. Because of that, other migration tools are also examined and compared.

There are multiple measurements that can be used to compare migration tools, and that can be improved. I decided to decrease the effort needed to write the migrations. Adding a certain degree of automation to the migration tool decreases the effort needed. The complete automation is not achievable. A discussion why is a next goal of this thesis.

As a consequence, MigrationGen tends to be automated as much as possible. It is based on code generation. Code generation is implemented using Projective Technologies tools developed by company CodiScent. Given a data source and a template, these tools generate a migration script which can be then applied to the database. Thus an important goal is to introduce how Projective Technologies are applied on MigrationGen.

The last goal is to explain the reasons behind the key design decisions of MigrationGen (e.g. a suitable file format consumed by MigrationGen).

Text structure

This thesis can be split into two parts - theoretical and practical. In the theoretical part of my thesis I deal with three main topics - database migration, existing solutions and CodiScent's tools.

I start with clarifying terms. In Chapter Domain Introduction I introduce the fundamental concepts which are needed later, among others schema migration and schema migration tools.

Chapter Projective Technologies deals with tools developed by company CodiScent and their approach to software development. I introduce RES (Reverse Engineering Studio) and GES (Generative Engineering Studio), also called Projective Technologies. These will be used in the practical part of my thesis. Because the technologies are based on reverse engineering and generative programming respectively, I examine these processes.

Practical part of my thesis deals also with three topics - design of the migration tool, implementation and extensibility. The practical part consists mainly of Chapter Analysis and Design and Chapter Implementation. I explain how I decided to encode the information about schema changes in Chapter Analysis and Design. I show the design of MigrationGen and present the whole process starting with schema change and ending with applying script which modifies the schema. Finally I propose a method for schema migrations and implement it in Chapter Implementation.

Chapter Existing Solutions gives a brief overview of existing migration tools. In this chapter the migration tools are analyzed, and a review of existing methods for migration is performed. I divide the migration tools into categories and I also categorize the migration tool I developed.

The result of the practical part of my bachelor thesis is a migration tool. However, it still is a Combined Bachelor's Thesis - specifically a combination of a Design Bachelor's Thesis and an Implementation Bachelor's Thesis. I revisit and review the developed tool in Chapter Conclusion.

CHAPTER 1

Domain Introduction

In this chapter I introduce the fundamental concepts like database, relational database, database migration and database migration tools.

1.1 Database

Database is a collection of organized data. *Database management system* (DBMS) is a computer program providing the user and other applications with an interface they can use to communicate with the database. Popular DBMS include Oracle Database, Microsoft SQL Server or PostgreSQL. Change performed in the database using DBMS is called *database transaction*. DBMS should guarantee that during a database transaction a set of properties called ACID will be applied. ACID stands for:

- **A**tomicity, stating in an “all or nothing” way that if one database operation fails, entire transaction fails
- **C**onsistency, asserting that no constraints will be violated, while moving database from one state to another
- **I**solation, expressing that concurrency control should be ensured for multiple transactions
- **D**urability, declaring that all changes should be stored permanently after completing transaction

Database is organized using the database model. *Database model* is a collection of conceptual tools for describing data, data relationships, data semantics and consistency constraints [1]. Examples of database models are hierarchical database model, network model, entity-relationship model or relational model. In relational database the data is organized using the relational model of data.

1.2 Relational model

Relational model was proposed by Edgar F. Codd in 1970 in his paper *A relational model of data for large shared data banks* [2].

Relational database usually consists of one or multiple tables. Consider Table 1.1 Table people has four columns (`id`, `first_name`, `surname`, `age`) and

Table 1.1: people

`id`	`first_name`	`surname`	`age`
1	Steven	Morrissey	24
2	John	Marr	19
3	Andrew	Rourke	18
4	Michael	Joyce	19

four rows. Each row can be thought of as a relationship between `id` and values of `first_name`, `surname` and `age`. This sequence of values (`id`, `first_name`, `surname`, `age`) is called *tuple* in relational model (specifically n-tuple for n values). Set of these tuples is a *relation* which corresponds to a table in a relational database. Element of a tuple is an *attribute* and it corresponds to a column in a table. To represent information which is missing, three-valued logic is used. Attribute can take three possible values - `TRUE`, `FALSE` and `UNDEFINED`.

To distinguish between tuples in a relation, keys are used. *Superkey* allows us to uniquely identify a tuple in a relation and it is a set of attributes. *Candidate key* is a minimal superkey. *Primary key* is a candidate key which was chosen to identify a tuple in the relation. Finally, *foreign key* is a primary key of another relation. Using Table 1.1 as an example, superkey could be a triplet (`id`, `first_name`, `surname`), candidate key could be one of the columns `id`, `first_name` or `surname`, and column `id` could serve as a primary key.

Attributes, tuples and relations describe how data should be stored. To operate upon data, relational algebra and relational calculus are used.

Relational algebra is a collection of operators that take relations as their operands and return a relation as their result [3]. The fundamental operations are:

- selection (denoted by σ), which selects tuples satisfying selection condition
- projection (Π), which returns a relation containing only the set of attributes passed to it, leaving others out
- union ($\cup$), which is a union between two relations

- set difference ($-$), which returns relation containing tuples which are in one of the original relations, but not in the other
- Cartesian product ($\times$), which returns a Cartesian product of two relations
- rename (ρ), which renames attributes in a relation

Using these six operations, another operations like set intersection ($\cap$) or natural join ($\bowtie$) can be defined.

Relational calculus is an alternative to relational algebra, forming queries in a declarative way, instead of a procedural one. Codd's theorem states that relational algebra and relational calculus are equivalent in expressive power. Structured Query Language (SQL) is based on relational algebra and relational calculus and is discussed in the next section.

1.3 Query languages

SQL (originally called Sequel) was developed in the 1970s by Donald D. Chamberlin and Raymond F. Boyce [4]. SQL is a query language used to access, define and manipulate data, and it is the most commercially influential one [1].

Syntax of the language resembles natural language and it is a case insensitive language. It is a declarative language, however it has extensions adding procedural language functionality.

The language contains Data Definition Language (or Data Description Language), Data Manipulation Language (DML) and Data Control Language (DCL) statements. DDL is a language used to define data structures. In SQL CREATE, DROP and ALTER statements are used to create a table, destroy a table and add or delete a column. Example of a DDL statement is given in Figure 1.1.

Figure 1.1: DDL statement

```
CREATE TABLE people (
  id INTEGER PRIMARY KEY,
  first_name VARCHAR(50) NOT NULL,
  surname VARCHAR(50) NOT NULL,
  age INTEGER NOT NULL
);
```

DML is a language used to manipulate data. In SQL statements like SELECT or INSERT are used to retrieve values from the database, or insert values into the database. Example of a DML statement is given in Figure 1.2.

Figure 1.2: DML statement

```
INSERT INTO people (id, first_name, surname, age)
VALUES (5, 'Craig', 'Gannon', 16);
```

In addition to DDL and DML SQL contains commands to add integrity constraints, define views, begin and end transactions. It also contains commands to add and remove access rights which constitute the DCL.

1.4 Schema migration

Database schema describes structure or a logical design of a database. It contains definitions of tables, fields, relationships and other elements. Database schema does not generally change, tables in database change more often, as they are updated with new records [1]. However, in some environments - like developing software, we have to consider the possibility of change.

Series of changes (like creating a table, adding a column, renaming a column) brings database schema from one version to another. We have two possibilities how to migrate from the one version to the other. The first consists of deleting the old database and creating a new one. This option is not used often, because it is associated with data loss. We have to insert data from the old database to the new one. In case that data in the old database was generated, we have to regenerate it and regenerating takes time. Second option is used much more in software development and it is the migration of the database.

The process of updating or reverting database schema from older version to newer (or newer version to older) is called *schema migration* (also database migration or database change management)

There are another definitions of database migration - one is migrating from one type of database to another. This definition is used in paper by Maatuk, Ali and Rossiter[5]. Their paper proposes method for migration of relational database to an object-based/XML database. Another definition defines schema migration as moving database from one physical location to another.

In this thesis I deal only with the first definition which is transforming relational database schema from one version to another.

1.5 Migration tools

In an effort to automatize the whole process, schema migration tools are developed. These tools vary in their functionality. Some of them just store the migration scripts and a number identifying last applied SQL script. Some of them connect to the database and enable to apply these scripts. Some tools are integrated into web application frameworks like Django[2] or Ruby on Rails[3]. Schema migration tools are the main topic of Chapter 4.

1.6 Projective Technologies

Projective Technologies is a term used by company CodiScent to describe their tools Generative Engineering Studio (or GES) and Reverse Engineering Studio (RES). GES and RES are based on the principles of generative programming and reverse engineering. CodiScent uses these tools in their generative engineering approach. In this approach the majority of the application code is generated using templates. This approach saves time and effort and improves quality of the code.

This topic is dealt with in more detail in Chapter 3.

1.7 XML

Extended Markup Language (XML) is a markup language which is used to describe XML documents. The language was designed to be usable, easy to write and easily readable by humans. The specifications for this language are provided by World Wide Web Consortium[4].

The constructs which XML uses are similar to another markup language - HTML. The key ones are tags, elements and attributes. Figure 1.3 contains some of these constructs. It depicts a transformation which adds column `first_name` to the table `people`.

The elements such as `transformation` or `table_name` are discussed in detail in Section 5.2.1.1.

1.8 XSD

XML schema is a formal description of the elements an XML file should contain. This description can be used then to verify if the actual XML file contains these elements. The process of verification is called *validation.*

The description includes specifications dictating what types of data can the elements contain, what order should the elements be in or which elements are optional and which are mandatory.

The description can be written in several languages. Schema specification for my migration tool was written in XML Schema Definition (XSD). XSD is a standard also developed by World Wide Web Consortium.

[2]`https://www.djangoproject.com/`
[3]`http://rubyonrails.org/`
[4]`www.w3.org`

Figure 1.3: A simple transformation written in XML

```
<transformation>
  <id>1</id>
  <transformation_type>ADD_COLUMN</transformation_type>
  <table_name>people</table_name>
  <field>
    <field_name>first_name</field_name>
    <field_type>VARCHAR</field_type>
    <field_dimension>50</field_dimension>
  </field>
</transformation>
```

There are several advantages which XSD has over other XML schema languages. They include support for namespaces and data types.

1.9 Summary

In this chapter fundamental concepts were explained and defined. The most important one is probably database schema migration, as this is the main subject of this thesis. But other important concepts were mentioned too like database, database management system or relational model. It was essential that I explained them here, as they are mentioned in the next chapters extensively.

Chapter 2

Goals Revisited

In my bachelor thesis I want to explore the topic of schema migration and to develop a new migration tool. Goals of this thesis are:

- *G1: Examine Projective Technologies*
- *G2: Develop a method for schema migration based on Projective Technologies*
- *G3: Using this method, implement MigrationGen - a new migration tool*
- *G4: Compare MigrationGen with the others*
- *G5: Demonstrate the possibility of extending MigrationGen*

Requirements for the tool developed in *G3* can be split into functional and non-functional. The functional requirements for this tool are:

- *F1: The system should be able to apply a migration directly in a connected database*
- *F2: The system should be able to revert a migration (if it is possible)*
- *F3: The system should be able to generate a migration script automatically*

The non-functional requirements for this tool are:

- *N1: The system should support multiple databases*
- *N2: In future, it should be easy to add a plugin which would bring support to another database*

The realization of the goals begins with Chapter Projective Technologies, where goal *G1* is accomplished. Chapter Analysis and Design deals with goals *G2* and *G5*.

Chapter Implementation deals with *G3* and of course with all the functional (*F1*, *F2*, *F3*) and the non-functional requirements (*N1*, *N2*).

Migration tools are compared in Chapter Existing Solutions and goal *G4* is fulfilled.

Chapter 3

Projective Technologies

Generating code saves time and cost and raises quality by reducing the possibility of a human error. Company CodiScent is developing tools enabling code generation. Currently, CodiScent stands behind *Reverse Engineering Studio* and *Generative Engineering Studio*, also called *Projective Technologies.* I used Projective Technologies in MigrationGen and they make the schema migration tool unique. Projective Technologies are covered in this chapter. These tools are based on principles of *reverse engineering* and *generative programming* and that is the reason why these principles are explained in this chapter too.

3.1 CodiScent

CodiScent is a consulting and software development company. The company has been evolving for the last 15 years and within last two years started to offer their services. The founder, CEO and chief scientist of CodiScent is software entrepreneur Zeev Chared. Leaders of the company have experience in delivering software solutions to different branches of business.

CodiScent approaches software development in a non-traditional way. The approach is *generative software engineering* and *reverse engineering.* Generative software engineering is based on generating code. Generating code is used extensively today, from web pages to plug-ins in different integrated development environments (IDE). To generate code we need input data, template and finally generator. According to CodiScent, one of the problems is the use of imperative code. This directs the process of reading the data with formatting and outputting the resulting text in their templates[6]. Imperative code makes the templates unnecessarily complex, and most importantly, it has to be written manually.

Another problem is the lack of modularity. The generating tools are usually coupled with one particular language or integrated into IDE and their applicability is narrow. Problem also lies in the fact that usually only a small percentage of code is generated which does not make generating worth the effort.

The approach used by CodiScent does not encounter these problems. Imperative code is not used in the templates. The applicability of the tools is not narrow, they serve from generating C# code to generating SQL scripts. Often 80% of the application code is generated in half the time[6].

3.2 Projective Technologies

As mentioned earlier, Projective Technologies consist of Reverse Engineering Studio and Generative Engineering Studio. Both RES and GES are still being developed.

CodiScent presents several possible use cases for Projective Technologies on their website[5], including:

- Accelerated software development
- Reengineering
- Modernization of an application
- Data transformation and data management
- Infrastructure development

I am going to start by explaining the concept of generative programming. Then GES is going to be introduced. Before describing RES, reverse engineering is discussed.

3.3 Generative Programming

Generative programming is defined as "*a software development paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge. The generated products may also contain non-software artifacts, such as test plans, manuals, tutorials, maintenance and troubleshooting guidelines, and so on*" [7].

In other words, the goal of generative programming is to replace the manual work in software development with automated generation.

Other definitions of generative programming mention thinking in terms of *software systems families.* This means that components from similar systems and processes which were used in developing similar systems should be reused.

[5]`http://codiscent.com/?page_id=335`

Another important concept interrelated to generative programming is *domain engineering. Domain* describes the functionality program should have and the requirements program should meet. Domain engineering is about producing new programs using knowledge about the domain. In other words, domain engineering is the process of producing new software systems using knowledge about similar computer systems from the same application domain.

There are three phases in domain engineering: *domain analysis*, *domain design* and *domain implementation.* In domain analysis information about the domain is extracted, domain requirements are gathered and domain model is produced. In the domain design phase the architecture of the system is produced using the domain model. Finally, in domain implementation the tools for generating the system are produced.

Important part of domain engineering is the choice of domain-specific language. The formal definition of a *domain-specific language* is "*A domain-specific language (DSL) is a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain*"[8]. Example of well-known DSL is HyperText Markup Language (HTML) and its domain are web pages.

Another example of a DSL would be the templating language CodiScent created. Among other tools, it is used in GES, and GES is discussed in the next section.

3.4 Generative Engineering Studio

Generative Engineering Studio serves as an Integrated Development Environment for CodiScent's generative engineering approach. GES was designed for developing and managing projects - models, templates and source files included. It consists of several components. I am going to introduce them one by one in this section.

Projector Template Generator (PTG) is the fundamental part of GES. Given a source file and a template, it can generate code, data or text. Different types of files can be passed as a source - XML, Excel, UML, SQL or plain text files.

The templates are written in a language developed by CodiScent. Example of one template is given in Figure 3.1. This template was used to generate SQL script which added column to a table. The template has only two lines, other templates I created are also similarly short. This can be seen as a sign that less code than normally was needed. Furthermore, one can notice that the imperative code is missing - this is the responsibility of GES.

Data source for this template was an XML file containing elements such as `table_name` or `field_name`. GES then parsed this file and created a model - hierarchy tree, similar to the structure of the XML.

Variables in the template start and end with a dollar sign. The square brackets are used to create loops. The language also includes certain special characters for formatting output. The values of the variables are assigned from the data source and *projected* through the template. One of the possible outputs generated from the template can be seen in Figure 3.2.

Figure 3.1: Template for adding column

```
[[ALTER TABLE $table_name$ ADD]
  [$field_name$ $field_type$[($field_dimension$)]^, \n]];
```

Figure 3.2: One of the possible outputs generated from Figure 3.1

```
ALTER TABLE people ADD
  city VARCHAR(50);
```

Relational Metadata Inference Transformer (RMIT) and *Configurable Graphical Interface Factory* (CGIF) are tools which are used to map the data source to the model.

A command-line version of GES was released recently. There are two ways of running it. The interface can be run with name of the solution and names of the templates to be generated. The other option is to run it with name of the source file and text of the template. This command-line interface lies at the heart of MigrationGen, and this topic is discussed more in Chapter 5.

3.5 Reverse Engineering

Usually, reverse engineering is defined as "*the process of extracting the knowledge or design blueprints from anything man-made*" [9].

The term reverse engineering of software has multiple meanings. Firstly, reverse engineering is used in security. It is used by both parties - attackers and defenders. Creators of malicious software use reverse engineering to locate weak points in programs and operating systems. Developers of anti-virus software reverse engineer viruses and try to extract information about them.

Secondly, reverse engineering is used in software development. Reverse engineering is used to explore the system to compensate for the lack of documentation. It is used to review third-party code and it is used to analyze product developed by competition.

Another definition which is still abstract, but more applicable to software, is "*reverse engineering is the process of analyzing a subject system to*

- *identify the system's components and their interrelationships*
- *create representations of the system in another form or at a higher level of abstraction*

"[10]. CodiScent's reverse engineering approach fits this description.

3.6 Reverse Engineering Studio

Reverse Engineering Studio is another IDE developed by CodiScent. There are many similarities with GES. It is also a commercial software and it is still under development.

The advantage GES has over RES is the command-line version. RES does not have one yet and this makes it very uncomfortable to use from an external program.

The workflow of RES is also similar to that of GES. However, RES uses templates not to generate code, but to extract information and save it to a model. The syntax used in the templates is similar to the one Projective Template Generator uses.

One of the typical uses for RES is detecting changes made to the database schema. The template is used to extract information about tables and columns the schema contains. The database schema is then represented as a model. The extracted information can be then easily compared to the older versions of the schema.

This way MigrationGen which I developed in the practical part of this thesis, could be extended. This possibility is discussed in Chapter 7.1.

3.7 Summary

In this chapter Projective Technologies, developed by company CodiScent, were introduced and goal *G1* from Chapter 2 was resolved. They consist of Reverse Engineering Studio and Generative Engineering Studio.

GES serves as an IDE for CodiScent's generative engineering approach. In this approach most of the code is generated using templates. Using templates for code generation is not new, but CodiScent's approach is different. The main difference is the lack of imperative code. This makes the templates easier to read, develop and maintain. RES is also an IDE, similarly to GES, but it uses the templates to extract information from the data source and save it to a model.

Both RES and GES are still under development, with the command-line version of GES appearing only recently[6]. Both of these tools are Windows-only.

[6]December 2014

RES and GES are based on the principles of reverse engineering and generative programming respectively. These principles were also explained in this chapter.

Chapter 4

Existing Solutions

I defined schema migration tools at the end of Section 1.5 in Chapter 1. This chapter focuses entirely on them. Firstly, I am going to start with analyzing the simpler ones and progress later to the more complex ones. Secondly, I want to compare them with my own application and to compare the functionality of these applications with my own.

4.1 Open DBDiff

Open DBDiff is an open-source database schema comparison tool for SQL Server 2005/2008[11]. The project is still in Beta phase, with the latest version (0.9) released on 26th March 2014.

Open DBDiff compares two database schemas and provides a script to transform one schema into another. The script can be then applied and schema updated. One of the disadvantages is that it supports only SQL Server. Nonetheless, the main disadvantage is that it does not manage these scripts.

Multiple alternatives to Open DBDiff exist. They range from open-source ones like CompareZilla[7] to commercial ones like Redgate SQL Compare[8]. The full list can be found on Open DBDiff web site[11].

To sum up, Open DBDiff is suitable for one time schema migration, but not for more complex schema evolution.

4.2 Flyway

Flyway is an open-source database migration tool, favoring simplicity and convention over configuration[12]. Creator and project lead is Axel Fontaine.

The most important term here is probably *metadata table*. Flyway creates one metadata table for the database. Every time a change is made to the

[7]`http://sourceforge.net/projects/comparezilla/`

[8]`http://www.red-gate.com/products/sql-development/sql-compare/`

database schema, an SQL script is applied. Information about this migration is added to the metadata table as a new record. Every row of the table contains information, such as name of the script, version number, when was it applied and by whom it was applied. Example of a similar table is given in Table 4.1.[9]

Table 4.1: Flyway: schema_version

version	description	script	date	author
1	Initial Setup	initial_setup.sql	25.02.2015	axel
2	First Changes	first_changes.sql	26.02.2015	axel
3	Minor Refactoring	minor_refactor.sql	27.02.2015	axel

After running Flyway, it will try to locate the metadata table first. If it does not exist, it will create it. Then, classpath of the application we want to migrate is scanned for migrations. These are then divided into two categories:

- *applied migrations* are migrations with their version number lower or equal to current migration
- *pending migrations* are migrations which are available, but were not applied

Flyway can be configured either from command line, or by editing a configuration file. It can be run using a command-line interface. It can also be integrated into a project using Maven[10], or its alternatives Gradle[11] and Sbt[12]. An Application Programming Interface (API), written in Java, is also available. Using one of the options mentioned above, pending migrations can be applied. The metadata table will be updated accordingly. Migrations are usually inputted as SQL scripts, but they can also be written as Java classes implementing `JdbcMigration` interface from Flyway API.

Flyway is a simple and easy-to-use tool, supporting multiple DBMS including Oracle, SQL Server, PostgreSQL, MySQL or SQLite. It does bring order into the management of the migrations and applying of them. The possibility of writing migrations in Java can be thought of as an advantage. It does not bring any abstraction, given that the statements are still in SQL, but it brings integration. Another advantage over Open DBDiff is support of multiple DBMS. However, it does not provide a solution for detecting changes and this means that the whole process of schema evolution is hard to automate.

[9]Table inspired by `http://flywaydb.org/getstarted/how.html`
[10]`http://maven.apache.org/`
[11]`https://gradle.org/`
[12]`http://www.scala-sbt.org/`

Figure 4.1: Liquibase: ChangeSet creating table from Table 1.1.

```
<changeSet id="1" author="chgolian">
  <createTable tableName="people">
    <column name="id" type="integer">
      <constraints primaryKey="true" nullable="false"/>
    </column>
    <column name="first_name" type="varchar(50)"/>
    <column name="surname" type="varchar(50)"/>
    <column name="age" type="integer"/>
  </createTable>
  <rollback>
    drop table people
  </rollback>
</changeSet>
```

4.3 Liquibase

Liquibase is an open-source tool similar to Flyway. The main developer is Nathan Voxland.

While Flyway uses a metadata table, the root of all changes in Liquibase is *databaseChangeLog* file.This file contains information about changes to be done to the database encoded in *changeSets*. Every changeSet is identified by combination of attributes `author`, `filename` and `id`. Into these changeSets preconditions can be inserted which are checked before executing every changeSet. These preconditions serve multiple purposes such as enforcing assumptions or documenting them. Example of such precondition could be a check if the change is being run on a specific DBMS or if the change is being run as a specific user.

DatabaseChangeLog supports XML, JSON, YAML data serialization standard and even SQL. Example of a changeSet written in XML is given in Figure 4.1.

There exists a commercial version of Liquibase, called Datical[13]. It provides additional functionality such as support for stored procedures and triggers. Liquibase also supports most of the popular RDBMS, less common ones have been moved into extensions and these have to be installed additionally.

The advantage that Liquibase has over Flyway is that it enables to define rollback of a change. However, the migrations cannot be written in Java, unlike Flyway. Because they are written in XML, the developer does not have to care about specific SQL dialect, although they can also be written in SQL. Similarly to Flyway, the application does not provide a solution to

[13]`http://www.datical.com/`

Figure 4.2: migrate4j: Migration creating table from Table 1.1

```
public class Migration_1 implements Migration {

  public void up() {
    createTable(
      table("people",
        column("id", INTEGER, primarykey(), notnull()),
        column("first_name", VARCHAR, length(50)),
        column("surname", VARCHAR, length(50)),
        column("age", INTEGER)));

  }

  public void down() {
    dropTable("people");
  }
}
```

detect changes and generate changeSets, and because of this schema evolution is hard to automate. Possible solution would be to use Open DBDiff to generate a synchronization script each time schema is changed, and to use Flyway/Liquibase to manage, store and apply these scripts.

4.4 migrate4j

Migrate4j is an open-source database migration tool. The initial intent of migrate4j was to make a Java version of Ruby's `db:migrate`.Database migration in Ruby is discussed in Section 4.5.

Migrations in migrate4j are written in Java. Every migration is represented by a class implementing `com.eroi.migrate.Migration` interface. The interface defines two methods: `up` and `down`. First of the mentioned methods represents applying a change, second represents rollback. The migrations are named `Migration_1`, `Migration_2` and so forth. Example of one such migration is given in Figure 4.2. When using migrate4j for a specific database, table called `version` with a single column `version` has to be created. This table is then used to set database to a state after specific migration.

Unfortunately, migrate4j supports only Derby, H2, MySQL and Sybase. Authors of the documentation admit that this is a very small number of supported DBMS. Migrate4j, similarly to Liquibase, brings developer abstraction. Migrations are written in Java, similarly to Flyway. In contrast to Flyway, they provide methods and objects like `createTable()` and `table()` like in

Figure 4.2. However, the small number of supported DBMS does not make it a serious competition for other migration tools.

4.5 Ruby on Rails

4.5.1 Ruby

Ruby is a programming language which presents itself as a language of careful balance[13]. It is a language which is multi-paradigm, with features like dynamic and duck typing, garbage collection and exception handling. Ruby appeared in 1995 and its author is Yukihiro Matsumoto.

4.5.2 Rails

Ruby on Rails is a web application development framework and it is written in Ruby. Initially released in 2005, it became a popular tool to build web applications even for companies like Github[14] or Scribd[15]. The reason for its popularity is that Rails philosophy is based on two principles: Don't Repeat Yourself (DRY) and Convention Over Configuration (COC). DRY is a principle which states that: "*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*". COC is self-explanatory. Set of conventions should be preferred over complicated editing of configuration files.

4.5.3 ActiveRecord

Relevant for this thesis is one library of Ruby on Rails: ActiveRecord. ActiveRecord is named after *Active Record* design pattern described by Martin Fowler in his book *Patterns of Enterprise Application Architecture*[14]. Active Record design pattern provides a solution for *object-relational mapping (ORM)*. Object-relational mapping is a technique used in object-oriented programming to access relational database records comfortably. In Active Record design pattern one database table corresponds to one class, one object instance corresponds to one row in a table. The class provides methods enabling CRUD (Create, Retrieve, Update, Delete) operations on a database. Class Base in module ActiveRecord presents an API implementing these methods.

Migration in ActiveRecord (and in Ruby on Rails) is represented as a class inheriting from `ActiveRecord::Migration` class. Then method `change` has to be overridden. The `change` method currently supports limited number of migration definitions such as `add_column`, `add_foreign_key` or `create_table`. The reason for this limited support is that ActiveRecord knows how to reverse this transformations. Reversal for migrations that ActiveRecord does

[14]`https://github.com/`
[15]`https://www.scribd.com/`

not know how to reverse is possible using the `reversible` method. In the old version of ActiveRecord the methods in `ActiveRecord::Migration` were called `up` and `down` and this is where migrate4j developers got its idea from.

Migrations are run from command line using tool called `rake` which is similar to Unix utility `Make`. After each migration schema is dumped into an SQL file or a Ruby file called `schema.rb`. ActiveRecord::Migration supports PostgreSQL, SQLite, Oracle, Microsoft SQL Server and MySQL.

Overall, migrations in Ruby on Rails are well-thought out. The abstraction is supplied by ActiveRecord API, so there is no need for knowledge of SQL. The management of migrations is made simple with the command-line utility and schema dump serving as a reference.

4.6 Django

4.6.1 Python

Python is, similarly to Ruby, a high-level multi-paradigm language. Its many features include dynamic typing, garbage collection and support for functional programming. It appeared in 1991 with Guido Van Rossum as its author.

4.6.2 Django

Django is an open-source web development framework written in Python. Initially released in 2005, and originally named after jazz guitarist Django Reinhardt, its motto is: "*The web framework for perfectionists with deadlines*". Its design philosophy includes DRY, an addition is a core Python "*Explicit is better than implicit*" principle. Another reason for its popularity is that Django documentation counts among the best ones from open-source projects. Mozilla[16] uses Django for its web site.

4.6.3 Migrations in Django

Initially, a schema migration tool South[17] was used for migrations in Django. Project South is now end of lifed. However, the integrated version of Django migrations is created by people from the original South project and is close to the original project. Due to this, I decided to analyze both migrations in South and integrated migrations.

South was a tool which was installed using one of package management systems like `pip`. Migrations were ran using command line. The tool had two ways of migrating: automatic and manual. The manual way was very similar to other migration tools. Migrations were just classes with two methods: `forwards` and `backwards`. One can see that this is very similar to `up` and

[16]`https://www.mozilla.org`
[17]`http://south.aeracode.org/`

`down` in Ruby on Rails (or migrate4j). From these methods methods from database API were called to interact with the database.

The automatic way was more interesting. Every time South was ran, it detected changes being made to the model and created the migration. To elaborate the migration workflow:

1. If a field was added to the model, migration script (in Python) was created (either automatically by South or manually by developer)
2. This script contained a class with methods `forwards` and `backwards`
3. From method `forwards` method from database API (`db.add_column`) was called with specific parameters
4. Migration then could be applied using Django command line utility `manage.py`

South supported PostgreSQL, MySQL, SQLite, Microsoft SQL Server and Oracle. There was also beta support for Firebird.

From version 1.7 migrations are firmly integrated into Django. There are still two ways of migrating. Migration is still represented as a Python class, but methods `forwards` and `backwards` disappeared. Instead the class has two lists now: `dependencies` and `operations`. List `dependencies` contains other apps on which our migration could depend. Project in Django is divided into separate apps which can communicate with each other. `operations` is a list of `Operation` classes. These classes define single transformations to the model like `migrations.addField`. All DBMS which Django supports are supported.

To sum up, migrations can be done automatically in Django, but there is a possibility to write them manually. I consider the automatic way to be a big advantage, although it does not seem to give Django an edge against Rails. All the popular DBMS are supported. Django even offers a solution for migration of data, but that is not the topic of this thesis.

4.7 Summary

In this chapter I introduced and analyzed several migration tools - goal *G4* from Chapter 2 is resolved here.

I started with relatively simple Open DBDiff and ended with migration tools integrated into web frameworks Rails and Django. I decided to divide them into these categories:

- **Category 1**: Tools only generating migration scripts automatically
- **Category 2**: Tools only managing migration scripts
- **Category 3**: Tools both generating migration scripts automatically and managing them

Generating migration scripts is a harder task than managing them. The reason is that changes in the database schema have to be somehow detected. The advantage of these tools which generate scripts, is that they save effort. However, the most important quality to software developers seems to be simplicity. For this reason Flyway is more popular than Liquibase.

There are two methods how migration scripts can be generated. The first one is comparing two versions of schema and producing resulting script. The second involves some sort of a model which represents the schema. When original schema is changed, changes are stored in the model. This model is then used for generating the migration script.

Open DBDiff takes two versions of database schema compares them and provides a synchronization script. Because of that, it belongs to Category 1. Flyway, Liquibase and migrate4j are very useful for managing scripts and applying them. The migrations have to be written manually (either in SQL, Java or XML) and because of that, they belong to Category 2. Django detects changes in the model part of the MVC (Model-View-Controller) architectural pattern automatically. It generates migrations and provides a solution to managing them. This makes it a representative of Category 3.

MigrationGen, my own application, currently belongs to Category 1. The advantage other migration tools have over MigrationGen is that MigrationGen is Windows-only. Migrations in MigrationGen have to be defined in XML files. It enables to apply a few of them or all of them. MigrationGen could belong to Category 3 - the reason why is explained in the next chapter.

Chapter **5**

Analysis and Design

The main topic of this chapter is the design of my own migration tool - MigrationGen. In this chapter I want to explain the reasons behind the key decisions made during this phase.

One of the major decisions was to consider what types of schema changes should be supported by MigrationGen. It is very hard to encompass more complex changes like defining views or triggers. The problem lies in deciding which changes should be categorized as simple and which ones as complex.

The next major question was how the migration itself should be represented. Multiple formats are available, all of them have pros and cons.

Finally, I want to present the overall process of migrating using MigrationGen.

5.1 Analysis

The functional and non-functional requirements for MigrationGen are described in Chapter 2. Obviously, the new migration method should meet these requirements.

The requirement *F1* can be met easily. Most of the programming languages used today have some sort of API which can be used to communicate with the database. I decided to write the migration tool using Java and Java uses `java.sql` package to do this.

The requirement *F2* is a bit harder to fulfill. The previous requirement can be summed up as applying an SQL script to the database. A problem is encountered here - only a few types of changes are reversible. One of the reasons behind this problem is that sometimes information needed for the reversal is missing. An example of this would be an attempt to create a database table after it was dropped. However, this can be solved by storing the information.

Another reason is that a revert of a certain change could violate certain database constraints. This is much more serious and makes some changes irreversible. Hence, reversal is possible only for a few types of the changes.

To satisfy requirement *F3* several decisions have to be made. Firstly, for the migrations to be generated automatically, there has to be a generator. I used one part of GES, Projective Template Generator (PTG), to do this. Secondly, the given generator must be instructed to create migrations. In a context of PTG, these instructions are contained in so called templates. Thus, the necessary templates for PTG must be written. The templates are written using the templating language CodiScent developed. Thirdly, the migrations have to be structured in a reasonable way, so the generator can cope with them. Because of this, I decided that the XML format is going to be used, and I created an XML Schema using which they can be written.

The non-functional requirement *N1* is fulfilled easily. To fulfill it, more templates have to be written in different SQL dialects.

Finally, when all the previous functional and non-functional requirements are met, *N2* is met automatically. All of the decisions behind the design and the technical aspect of the design is discussed in the next few sections.

5.2 Design

5.2.1 XML Schema of a migration

I decided to use the XML format to describe the migrations. An introduction to this format was given in Chapter 1. The following subsection describes the structure of the XML migrations in detail.

5.2.1.1 XML structure

The XSD should serve as a *blueprint* for a migration written by developer. In other words every migration written by developer should contain at least all the mandatory elements specified in the XSD and it should contain them in the right order. Structure of the XML Schema I defined can be seen in Figures 5.1, 5.2 and 5.3.

Every migration consists of a series of *transformations.* These transformations are changes that were made to the database schema. An XML element `transformation_set`, depicted in Figure 5.1, contains one or more `transformation` elements.

A `transformation` element contains information about the type of transformation. This is used later when MigrationGen matches it with a correct template. It also contains name of the table, and it contains an id, so it can be later sorted. The `id` element is also important in case the migration consists of multiple XML files. Diagram can be seen in Figure 5.2.

Every transformation can also contain a `field` element. This element is not mandatory, because there exist transformations which operate only on tables e.g. DELETE TABLE or CLEAR TABLE statements. Every field must contain the name of a column which is to be changed. Other optional elements include information if the field is a primary key or not or if the field is mandatory or not. All of these elements can be seen in Figure 5.3.

Only certain types of transformations are supported, but that is discussed in Section 5.2.2.

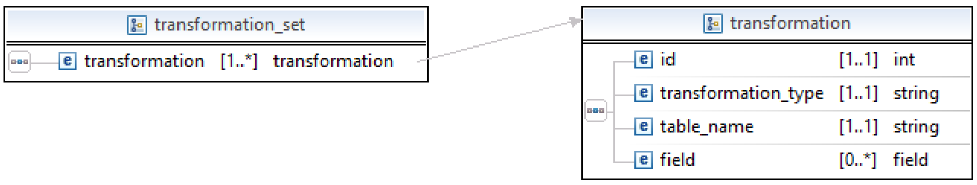


Figure 5.1: `transformation set` element

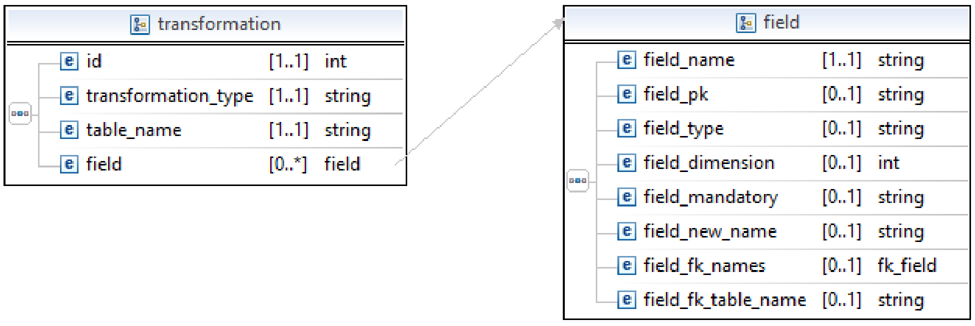

transformation
id [1..1] int
transformation_type [1..1] string
table_name [1..1] string
field [0..*] field
field
field_name [1..1] string
field_pk [0..1] string
field_type [0..1] string
field_dimension [0..1] int
field_mandatory [0..1] string
field_new_name [0..1] string
field_fk_names [0..1] fk_field
field_fk_table_name [0..1] string

Figure 5.2: `transformation` element

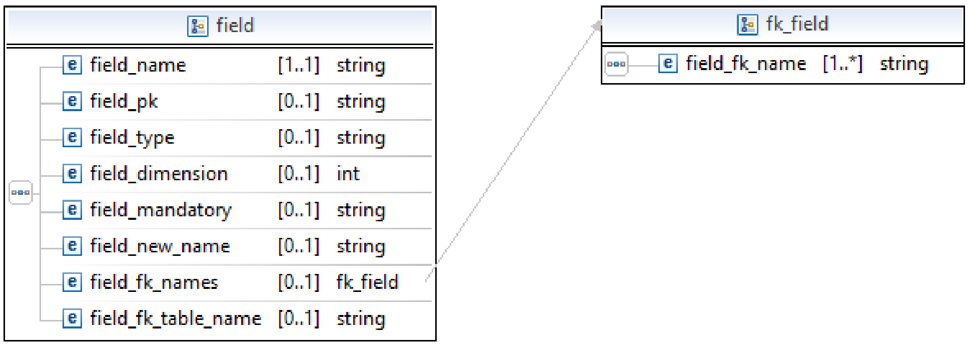

field
field_name [1..1] string
field_pk [0..1] string
field_type [0..1] string
field_dimension [0..1] int
field_mandatory [0..1] string
field_new_name [0..1] string
field_fk_names [0..1] fk_field
field_fk_table_name [0..1] string
fk_field
field_fk_name [1..*] string

Figure 5.3: `field` element

5.2.2 Schema changes

Schema migration was defined in Chapter 1. The incentive to the process of migration was a series of changes that were made to the database schema. When designing MigrationGen I encountered a problem - which types of changes should my tool support? The changes made to the database schema vary from adding a column to defining a view. Obviously, with every new type of change the migration tool would support, it would become more and more harder to make the tool automated. I decided to look into the *taxonomy* of schema changes in this section.

Marks and Steritt in their paper *A metadata driven approach to performing complex heterogeneous database schema migrations* identified 11 kinds of change to the database schema[15]. They divide these changes into two categories: simple and complex. Difference between these two lies in the fact that simple changes can be performed using a single SQL statement. The simple changes consist of *adding a table*, *deleting a table*, *renaming a table*, *adding a column*, *removing a column* and *renaming a column.* According to the criteria mentioned earlier, there are the five types of complex change: *manipulating data in place*, *changing type of columns*, *updating foreign keys*, *manipulating with large objects* and *merging and splitting tables.*

After consideration, I decided that my migration tool will support these types of changes to the database schema:

- Adding a column
- Altering a column
- Dropping a column
- Renaming a column
- Creating a table
- Clearing a table
- Deleting a table

Another types of changes could be added, but it would have two negative consequences. Firstly, a new template would have had to be written for every new type of a change. Secondly, the XML Schema Definition would have had to be extended. New elements would have had to be defined. This would result in the XSD becoming more and more complex. Furthermore, I believe that these seven changes are more than enough to demonstrate the basic functionality of the migration tool.

5.2.3 Templates

The developer writes the migration using the XSD and the migration is passed to MigrationGen. Every migration contains one or more changes, which are represented by `transformation` elements.

For every type of change defined in Section 5.2.2 a template was written. Most of these templates were small, about 2-3 lines in length.

The largest template is for creating a table. This template consists of three SQL statements - one CREATE TABLE and two ALTER TABLE statements. The CREATE TABLE statement creates all the necessary columns. The two ALTER TABLE statements add all the necessary primary and foreign keys. Nevertheless, 11 lines of code still makes it reasonably sized.

The templates are stored in one of the MigrationGen subdirectories. Then, MigrationGen parses the XML file containing the migration and every time a `transformation` element would be encountered, it is matched with a correct template. Both the element and the template are passed to the GES and using the template the resulting script is generated.

5.3 Summary

This chapter covered the design of MigrationGen. A new method of migration based on Projective Technologies was designed here - goal *G2* from Chapter 2 was resolved.

Several functional and non-functional requirements were put on this tool - they can be found in Chapter 2. One by one, I tried to fulfill them and I discuss this process in Section 5.1.

To automatize the migration process, the migrations are written by developer in one or multiple XML files. These files can be created from an XML schema file I designed. This is examined in Section 5.2.1.

Not all types of changes to the database schema are supported. I picked only seven types of them and I discuss the reason why in Section 5.2.2.

The migration files are then along with a connection string passed to MigrationGen. When applying or reverting a migration, the application always matches the individual change with a correct template. The templates are discussed in Section 5.2.3.

Both of these are then passed to the GES command-line tool which generates the SQL script. This script can be then applied to the database.

The migration method is depicted in Figure 5.4. Four main phases are depicted in the figure. Firstly, a DDL script is generated from the database. Secondly, RES extracts information from this script about the schema and saves it into a model Thirdly, GES is used to produce the migration script Finally, the migration script is applied to the database.

The big advantage of this method is that all of these four steps can be automated. A multitude of tools enables to generate the DDL of a database

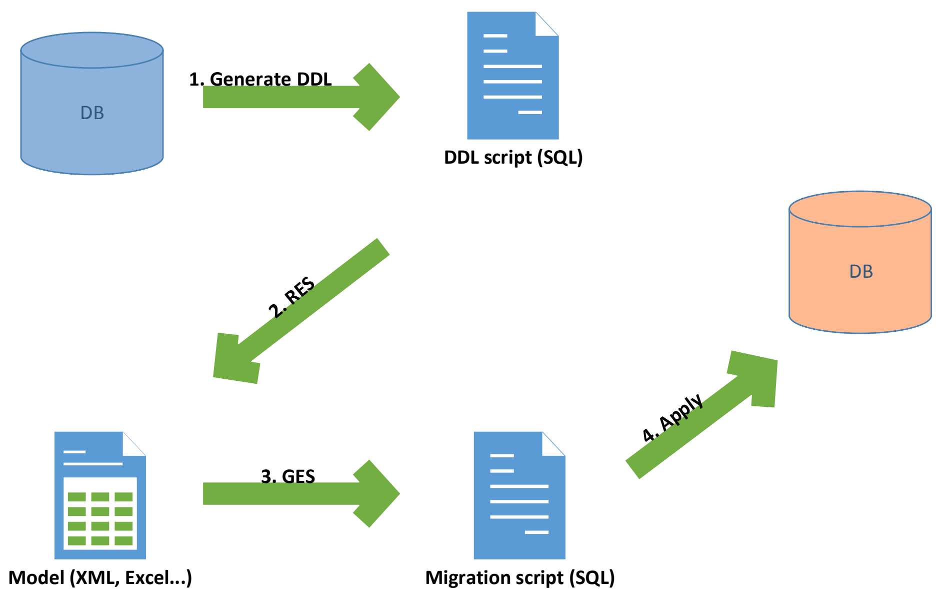


Figure 5.4: Schema migration using Projective Technologies

schema. Extracting information and projecting output can be automated using Projective Technologies. Finally, applying of migration script can be done using various APIs - one of them for example is JDBC for Java.

As mentioned in Chapter 3, RES is still under development. I discussed only the last two steps of this method in this chapter because of this. However, it is possible to *extend* this application - this was defined in Chapter 2 as goal *G5*.

In the next chapter, this migration method is implemented.

Chapter 6

Implementation

This chapter describes the process of implementing MigrationGen.

I decided to implement it using three-layered architecture. Therefore, I start with a brief theoretical introduction where I introduce the multitier architecture. Then, using UML diagrams I show how it was implemented during the development of MigrationGen.

All three layers - data, business and presentation are described in detail in individual sections. The underlying technologies under each of these layers are also discussed.

Finally, the workflow of the application which was introduced in Chapter 5 is restated again. However, this time it is done more explicitly.

6.1 Multitier architecture

6.1.1 Theory

The multitier architecture (also known as the n-tier architecture) is a type of architecture in which the individual tiers are separated. Usually, the number of tiers is three - *a presentation tier*, *a logic (also called business or application) tier* and *a data tier*.

The presentation tier *presents* information to the user. It also provides an interface which the user can use to communicate with other layers.

The business tier contains functionality of the application. It processes the input passed from presentation tier and performs various evaluations and calculations.

The data tier should provide an API to store and retrieve the information. There are multiple ways how this can be done - one of them is storing the information in a database.

The terms *tier* and *layer* are often used as synonyms. However, the general consensus seems to be that tier is used for a physical separation, while layer

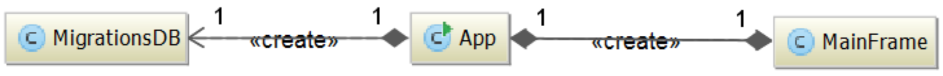


Figure 6.1: Important classes and their interaction in MigrationGen

for a logical one. In my application the separation of layers is a logical one, thus the term *multi-layered architecture* is correct.

6.1.2 Implementation

The three layers of three-layered architecture in MigrationGen are represented by three packages - `data`, `logic` and `presentation`.

The important classes in these packages are respectively `MigrationsDB`, `App` and `MainFrame`. Their class diagram can be seen in Figure 6.1.

The `MigrationsDB` class provides the interaction with the H2 embedded database using JDBC, the `MainFrame` class provides the GUI for MigrationGen. The `App` class serves in a sense as a *controller* from the MVC design pattern. It processes the input from the user and connects the model and the view. The process of generating the migration scripts is also done here.

6.2 Data layer

The main function of the data layer in MigrationGen is to perform CRUD operations on an embedded database. The data layer is represented by package `data` which contains one class - `MigrationsDB`. Methods of this class communicate with the database. The embedded database contains one table which stores information about the migrations that were added to the application.

The table `migrations_table` contains five columns - `id`, `migration_name`, `migration_folder`, `connection_string` and `latest_transformation_id`.

Most of these names are self-explanatory, the only interesting one is the last one. This column contains id of the latest transformation successfully applied. Because of this, when the migration is updated with a new set of XML files, only the latest ones are applied. When the migration is reverted the id is adequately updated.

This table is similar to the one which is used by the migration tool Flyway. Flyway is analyzed in Section 4.2 in Chapter Existing Solutions.

6.2.1 JDBC

Java Database Connectivity is a technology developed by Oracle Corporation and it is an API providing access to relational databases. The connection

Figure 6.2: Updating table `migrations_table`

```
Statement updateStmt = connection.createStatement();
String updateSql = "UPDATE " + DB_TABLE_NAME
  + " SET migration_name='" + migration_name
  + "', migration_folder='" + folder
  + "', connection_string='" + connectionString
  + "', latest_transformation_id=" + latestTransformationId
  + " WHERE migration_id=" + id;
updateStmt.execute(updateSql);
```

to the database is made using a JDBC Driver. Then, SQL statements can be created and executed and any results they return can be retrieved. One important thing is that the connection is in *auto-commit mode* by default.

The code snippet from Figure 6.2 comes from the method `executeUpdate` and this method is called whenever a migration is applied and the id of the latest transformation needs to be updated. It shows the typical workflow using JDBC. A `java.sql.Statement` object is created and one of its methods is called. What this snippet does not show is that a `java.sql.Connection` object has to be created first and closed later.

6.2.2 H2

H2 is a fast open-source Java SQL DBMS and it complies with the JDBC API. However, its most important feature is the embedded mode of operation.

The first version of MigrationGen used serialization to store all the information needed about the migrations. However, this was only a temporary solution that needed to be replaced as soon as possible. The advantage that the embedded database has is the availability. It was for this reason that H2 was chosen to store the migrations.

6.3 Business layer

The business layer is represented in MigrationGen by a multitude of classes contained in package `logic`. As mentioned before, the class `App` serves as a controller between the data and the presentation layer. Another important class is `Migrations` which calls GES.

However, it is different from the controller from the MVC design pattern. It contains the object representation of a migration - class `Migration`. According to the MVC, this should be stored in the *model* part. Another rule of the MVC which it breaks is that in the MVC the controller should be “thin” and

Figure 6.3: Code snippet from method `applyMigrations`

```
writeTransformation(newFilename, node);
String templateText = readTemplateText(node, false);
String ges_output = runGES(newFilename, templateText);
try {
  applyToDB(ges_output, connection);
} catch (SQLException e) {
  handleError(e.toString());
  cleanUp(newFilename);
  return id;
}
cleanUp(newFilename);
return id;
```

the model "fat". Actually, the package `logic` contains both the model and controller from the MVC.

The class `Migrations` is basically an API which I created for reading from and writing to XML files and running the GES command-line interface.

`applyTransformations` and `revertTransformations` are method from this class essential to the whole application.

These methods are similar to each other. The parameters to the both of them are a `Migration` object and an `id` of a transformation to be applied/reverted last. Both of them return id of the last successfully applied or reverted transformation.

From all the XML files which are passed along with the migration an `ArrayList` from `java.util` package is created. This list contains XML nodes which are the individual transformations. This list is then iterated over in a `for` loop and code snippet from Figure 6.3 is executed each time.

In this snippet, every XML node is written into a separate temporary file. Then, a correct template is found which matches the `transformation_type` element. The file and the template are then passed to the command-line interface which produces the SQL script. This script is then applied to the database, the temporary file is deleted and the `id` of the latest applied transformation is returned.

6.3.1 GES

There are two possibilities how the GES command-line interface can be run. The first is running the whole solution with a list of templates. The second is running one source file with one template. In MigrationGen the second possibility is used because I generate the scripts on the fly.

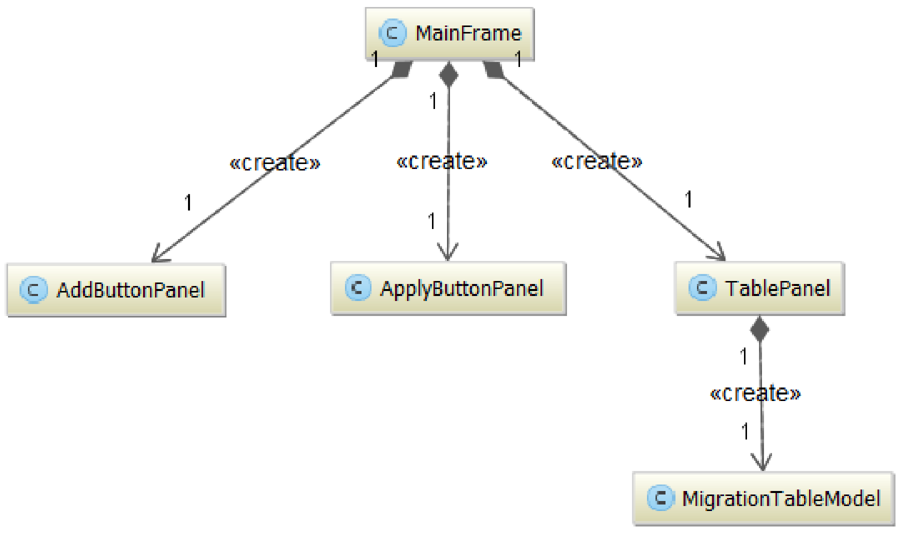


Figure 6.4: Presentation layer

The interface is run as a separate process from the method `runGES`. The output is then collected and returned from this method as a `String` object.

The fact that GES is Windows-only, restricted my application to this platform.

6.4 Presentation layer

The presentation layer of MigrationGen is contained in package `presentation`. The `Swing` toolkit was used to create the GUI.

The most important class in `presentation` package is `MainFrame` which extends the `JFrame` class.

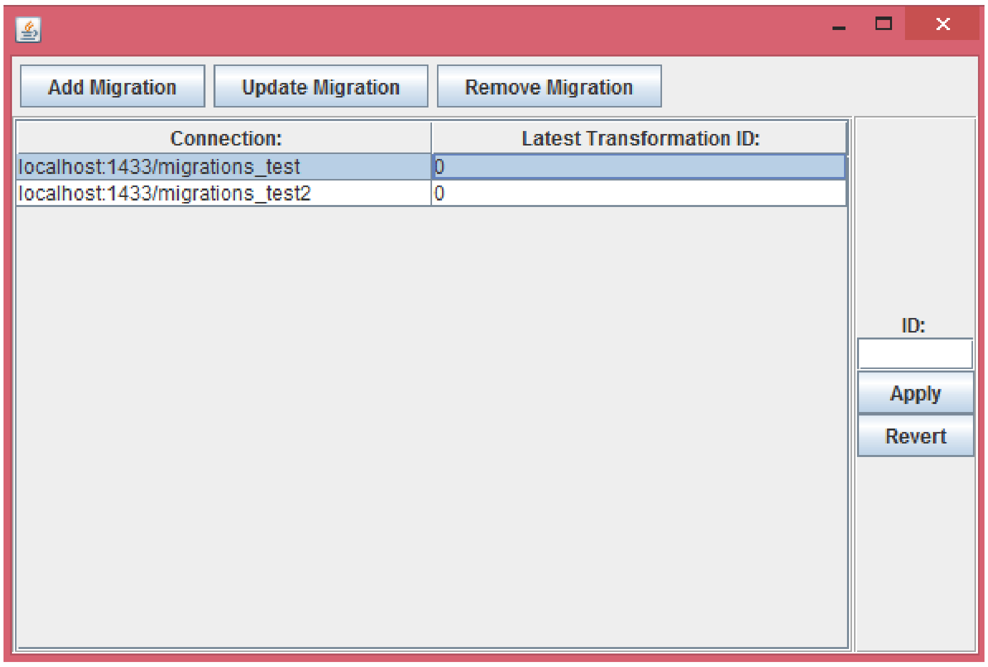
The button panel for adding, removing and updating migrations is contained in class `AddButtonPanel`. Similarly, the panel containing buttons for applying and reverting migrations is contained in class `ApplyButtonPanel`. Both of these class extend the `JPanel` class from Swing.

Finally, the table which shows the individual migrations is contained in class `TablePanel`.

The class diagram of classes contained in this package is depicted in Figure 6.4. The picture of MigrationGen's GUI can be seen in Figure 6.5.

6.4.1 User Interface Description

The button **Add Migration** enables to add a folder containing the migration. The folder should contain a JDBC connection string to the database and XMLs



Connection:	Latest Transformation ID:
localhost:1433/migrations_test	0
localhost:1433/migrations_test2	0

Figure 6.5: A screenshot of MigrationGen

containing the transformations. The specific structure which is required is presented in Appendix B. It also adds a new record to the H2 database.

Update Migration button should be used if new XMLs were added to an existing migrations. It reloads the folder and scans for new XMLs.

Remove Migration button is used to remove the selected migration. It deletes the migration from the table and from the H2 database.

Apply and **Revert** buttons enable to apply or revert the selected migration. By default, the last transformation applied is the transformation with the highest id. This can be changed by specifying the id in the **id** field.

6.5 Testing

MigrationGen was tested on different levels and in different phases of the development.

Firstly, there was *unit testing* JUnit framework was used to write unit tests to verify the behavior of MigrationGen. The behavior is contained in package `logic` and `data` and classes from these packages were tested. These tests can be found in package `cz.cvut.fit.migrationgen.test`.

Secondly, there was *usability testing*. UX testing was done by the Centre for Conceptual Modeling team. However, MigrationGen serves mostly as a *proof of concept* and not as an application which is to be deployed instantly.

Figure 6.6: Running times of transformation CREATE TABLE in milliseconds

transformation type	I.	II.	III.
CREATE TABLE (1 row)	1176	667	631
CREATE TABLE (10 rows)	648	730	705
CREATE TABLE (50 rows)	772	808	768
CREATE TABLE (100 rows)	957	943	942

Thirdly, there was *stress testing*. I wanted to verify if the size of the transformation influences running time of GES. I decide to go for CREATE TABLE transformation. The results can be seen in Figure 6.6.

The table depicted there shows running times of the transformation CREATE TABLE in milliseconds. The purpose of this experiment was to find out if size of the specification has any influence on running time. Four different XMLs were created. They created tables with 1, 10, 50 and 100 rows respectively. Each transformation ran three times.

6.6 Summary

In Chapter 5 a new method for migrating databases was designed. The implementation of this method and implementation of the application MigrationGen was described in this chapter. Thus, *G3* from Chapter 2 was resolved.

The migration method this application uses is:

1. The developer writes the migration in an XML file complying with the XSD from Chapter 5
2. The migration is then passed along with a connection string to the database in a folder to MigrationGen
3. Every transformation contained in the XML MigrationGen is matched with a correct template
4. The transformation is then written to a temporary file which is along with the template passed to the GES command-line interface.
5. The command-line interface generates an SQL script
6. The script is then applied to the database

The workflow of MigrationGen is depicted in Figure 6.7.

The migration tool MigrationGen was implemented using three-layered architecture. The three layers of this architecture are data, business and presentation. In MigrationGen the separation of layers is done using three packages.

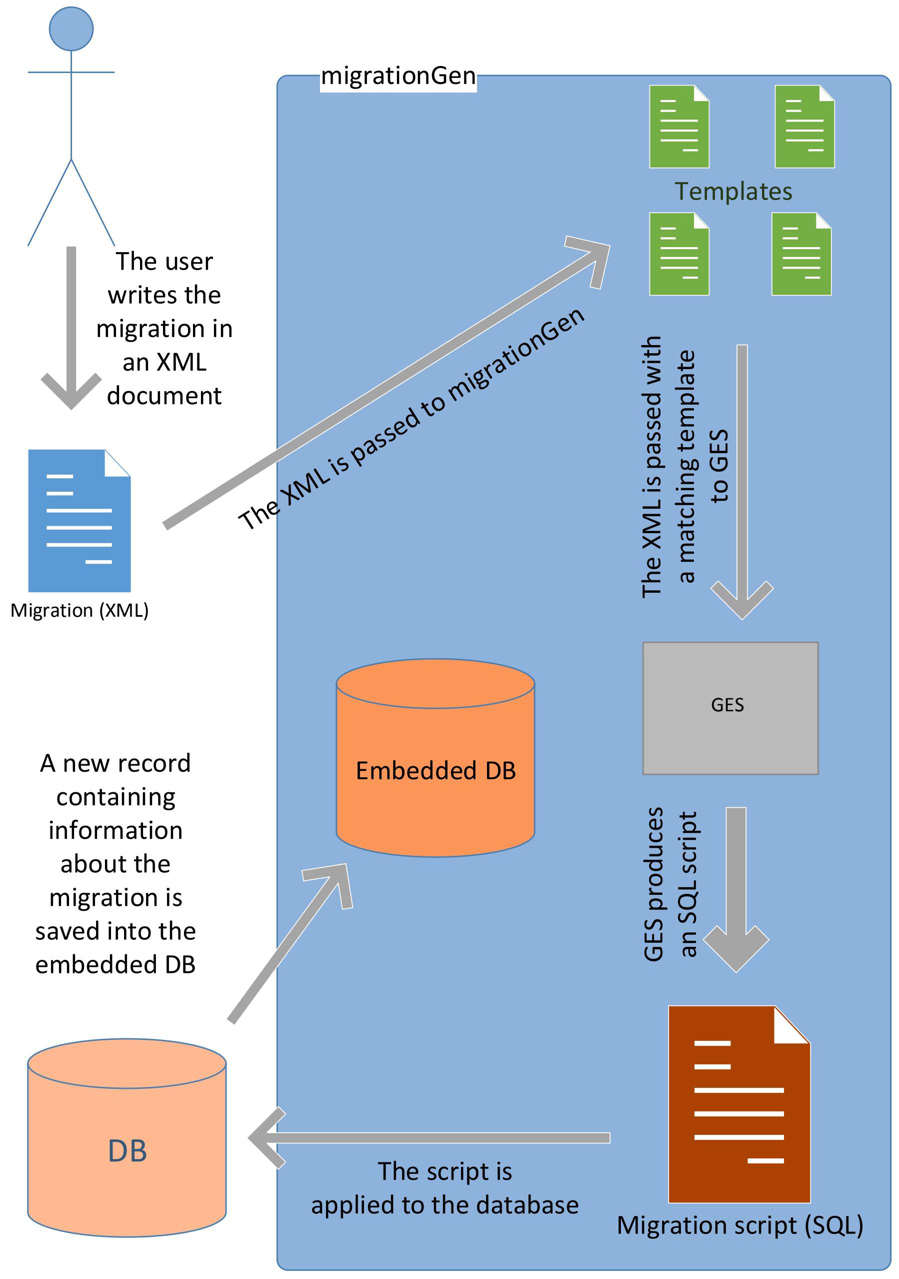

migrationGen
Templates
The user writes the migration in an XML document
Migration (XML)
The XML is passed to migrationGen
The XML is passed with a matching template to GES
GES
GES produces an SQL script
Migration script (SQL)
The script is applied to the database
DB
A new record containing information about the migration is saved into the embedded DB
Embedded DB

Figure 6.7: MigrationGen workflow

The data layer in MigrationGen provides interaction with the embedded database H2. The database is used to store information about migrations such as the id of the latest transformation applied.

Probably the most important method `applyMigrations` (and its equivalent for reverting migrations `revertMigrations` is contained in the business layer. They do the actual work of MigrationGen, as described in the numbered list above.

The presentation layer provides an graphical interface the user can use to add or remove migrations and to apply or revert them.

Chapter 7

Evaluation

In this chapter I want to revisit and review goals which were set in Chapter 2.

7.1 Review

The goals defined were:

- *G1: Examine Projective Technologies*

  Both RES and GES are described in Chapter Projective Technologies. RES is still under development and because of this only GES was used in the implementation. The workflow of GES makes generating code easy. Both of these tools are for Windows which restricts them and the tools that use them.

- *G2: Develop a method for schema migration based on Projective Technologies*

  The process is described in detail in Chapter Analysis and Design. The advantage of this method lies in the fact the user does not have to know SQL to write the scripts. SQL is contained in the templates and the scripts can be generated by GES from XML files. Because of this, some effort is saved. However, the XML format is verbose and thus the XMLs containing the migrations can easily get large.

- *G3: Using this method, implement MigrationGen - a new migration tool*

  The implementation is described in Chapter Implementation. Only a part of the migration method was implemented due to the fact that RES is still under development. Thus, migrations have to be written manually and this makes the process of migrating using MigrationGen clumsy.

- *G4: Compare MigrationGen with the others*

  This was done in Chapter Existing Solutions. MigrationGen is more automated, however currently it supports only two DBMS. It surpasses some migration tools but it is not as powerful as the migration tool which is integrated into Django.

- *G5: Demonstrate the possibility of extending MigrationGen*

  This was demonstrated in Section 5.3 at the end of Chapter Analysis and Design. If the migration method was implemented fully, it would make MigrationGen a substantial tool.

The functional requirements for MigrationGen were:

- *F1: The system should be able to apply a migration directly to a connected database*

  MigrationGen enables to do this.

- *F2: The system should be able to revert a migration (if it is possible)*

  In present version of MigrationGen it is possible for some types of transformations. It could be possible for more transformations if there was some form of model which would store information about the schema. However, not even the best migration tools are able to do this because some transformations are irreversible.

- *F3: The system should be able to generate a migration script automatically*

  The script is generated using GES which is provided with an XML and a template. This feature sets MigrationGen apart from other migration tools.

The non-functional requirements for this tool are:

- *N1: The system should support multiple databases*

  MigrationGen supports SQL Server and PostgreSQL. It can be extended to support other DBMS as discussed in the next point.

- *N2: In future, it should be easy to add a plugin which would bring support to another database*

  This can be done. However, new templates would have to be probably written, given that SQL dialects differ.

Conclusion

The ultimate goal of this thesis was to develop a new method of schema migration and to implement it. The migration method and the new migration tool - MigrationGen are the main contributions of this thesis.

Before designing this method (and implementing it) I had to spend some time on the terminology. I had to define precisely the term schema migration which has multiple meanings and they vary greatly.

The method was based on Projective Technologies supplied by the company CodiScent and implemented as MigrationGen. Projective Technologies consist of two parts: RES and GES. I introduced them along with the principles which they are based on - reverse engineering and generative programming.

Only one half of the migration method was implemented. This was caused by the fact that RES is still under development and unstable. Thus it was only evaluated how it can be integrated into MigrationGen in the future. The implemented half enables to generate the migration script from an XML file and apply it to the database. The XML format was chosen to represent the migrations for the reason that multiple files can be easily produced using one XML schema definition.

Only a limited number of schema changes is supported. More schema changes could be added, but that would increase the complexity of the XSD I wrote. New templates would also have to be written. However, the seven types of changes supported suffice to demonstrate the possibilities MigrationGen offers.

The migration tool MigrationGen was written in the Java programming language and the Java API JDBC is used to communicate with the database and apply the migration scripts.

Several non-standard requirements were put on MigrationGen. One of them was that it should be automated as much as possible. Another requirement demanded that MigrationGen should be open to extensibility. One by one, I solved or showed that it is possible to solve these requirements.

Finally, I researched other existing migration tools and compared them with MigrationGen.

MigrationGen differs from other migration tools. It is not bound to a specific DBMS and more importantly, it *generates* the scripts.

However, the necessity to write the migration files manually and their verbosity makes migrating using MigrationGen clumsy. The benefits which code generation brings do not outweigh these disadvantages. Generation of code is more suitable for tasks where a larger amount of heterogeneous code is generated.

Bibliography

[1] Silberschatz, A.; Korth, H. F.; Sudarshan, S. *Database System Concepts*. New York: McGraw-Hill, 2011.

[2] Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, volume 13, no. 6, 1970.

[3] Date, C. *An Introduction to Database Systems*. Reading, Mass.: Addison-Wesley, 2003.

[4] Chamberlin, D. D.; Boyce, R. F. SEQUEL: A structured English query language. *SIGFIDET '74 Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, , no. 1, 1974.

[5] Maatuk, A.; Ali, A.; Rossiter, N. Relational Database Migration: A Perspective. In *Database and Expert Systems Applications*, *Lecture Notes in Computer Science*, volume 5181, edited by S. Bhowmick; J. Kng; R. Wagner, Springer Berlin Heidelberg, 2008, ISBN 978-3-540-85653-5, pp. 676–683, doi:10.1007/978-3-540-85654-2_58. Available from: `http://dx.doi.org/10.1007/978-3-540-85654-2_58`

[6] About Generative Software Engineering. 2014. Available from: `http://codiscent.com/?page_id=325`

[7] Czarnecki, K.; Eisenecker, U. W. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, ISBN 0-201-30977-7.

[8] van Deursen, A.; Klint, P.; Visser, J. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, volume 35, no. 6, June 2000: pp. 26–36, ISSN 0362-1340, doi:10.1145/352029.352035. Available from: `http://doi.acm.org/10.1145/352029.352035`

[9] Eilam, E. *Reversing: Secrets of Reserve Engineering.* Indianapolis: Wiley Publishing, 2005.

[10] Chikofsky, E. J.; Cross, J. H. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, volume 7, no. 1, 1990.

[11] Open DBDiff. 2008. Available from: `https://opendbiff.codeplex.com/`

[12] Documentation - Flyway: Database Migrations Made Easy. 2010. Available from: `http://flywaydb.org/documentation/`

[13] About Ruby. 2001. Available from: `https://www.ruby-lang.org/en/about/`

[14] Fowler, M. *Patterns of Enterprise Application Architecture.* Boston: Addison-Wesley, 2003.

[15] Marks, R. M.; Sterritt, R. A Metadata Driven Approach to Performing Complex Heterogeneous Database Schema Migrations. *Innov. Syst. Softw. Eng.*, volume 9, no. 3, 2013.

Appendix **A**

Acronyms

API Application Programming Interface

CRUD Create, retrieve, update and delete

DBMS Database Management System

GUI Graphical user interface

GES Generative Engineering Studio

HTML HyperText Markup Language

JDBC Java Database Connectivity

MVC Model-view-controller

RES Reverse Engineering Studio

SQL Structured Query Language

UML Unified Modeling Language

XML Extensible Markup Language

XSD XML Schema Definition

Appendix **B**

Practical examples

MigrationGen supports at the moment PostgreSQL and SQLServer. I decided to give some practical examples here.

The directory which is passed to MigrationGen must have a structure similar to the one in Figure B.1.

Figure B.1: Structure of a directory passed to MigrationGen

```
Migration
├── conn.txt
├── migration1.xml
└── migration2.xml
```

The `conn.txt` file contains a JDBC connection string to the database. The XML files contain the individual transformations and should follow one after another.

Creating a table

This example creates a table in the PostgreSQL database `migrations_test`. The `conn.txt` file looks similarly to Figure B.2:

Figure B.2: JDBC connection string to a PostgreSQL database

```
jdbc:postgresql://localhost:5432/migrations?user=admin&password=
admin
```

The table `people` contains three columns: `first_name`, `last_name` and `id`. The XML containing this transformation is shown in Figure B.3. The CDATA section between elements `field_fk_table_name` and `field_fk_name` is mandatory in this case because it would cause an error in GES.

The SQL script which GES produces is shown in Figure B.4.

Figure B.3: Single transformation creating a table

```
<?xml version="1.0" encoding="UTF-8"?>
<tset>
 <transformation>
  <id>1</id>
  <transformation_type>create_table</transformation_type>
  <table_name>people</table_name>
  <field>
   <field_name>first_name</field_name>
   <field_type>VARCHAR</field_type>
   <field_dimension>50</field_dimension>
  </field>
  <field>
   <field_name>last_name</field_name>
   <field_type>VARCHAR</field_type>
   <field_dimension>50</field_dimension>
   <field_mandatory>YES</field_mandatory>
   <field_fk_table_name><![CDATA[ ]]></field_fk_table_name>
   <field_fk_name><![CDATA[ ]]></field_fk_name>
  </field>
  <field>
   <field_name>id</field_name>
   <field_type>INTEGER</field_type>
   <field_mandatory>YES</field_mandatory>
   <field_pk>YES</field_pk>
  </field>
 </transformation>
</tset>
```

Figure B.4: SQL script which GES produces from Figure B.3

```
CREATE TABLE people( first_name VARCHAR(50),
     id integer NOT NULL,
     last_name VARCHAR(50) NOT NULL );
ALTER TABLE people ADD CONSTRAINT pk_id_people PRIMARY KEY (id);
```

Creating a table with a foreign key

This migration creates two tables: `addresses` and `people`. The table `people` contains a foreign key from `addresses`. One transformation from this migration is shown in Figure B.5. The output is presented in Figure B.6.

Figure B.5: Single transformation creating a table with a foreign key

```
<transformation>
 <id>2</id>
 <transformation_type>create_table</transformation_type>
 <table_name>people</table_name>
 <field>
  <field_name>first_name</field_name>
  <field_type>VARCHAR</field_type>
  <field_dimension>50</field_dimension>
 </field>
 <field>
  <field_name>last_name</field_name>
  <field_type>VARCHAR</field_type>
  <field_dimension>50</field_dimension>
 </field>
 <field>
  <field_name>address_id</field_name>
  <field_type>INTEGER</field_type>
  <field_mandatory>YES</field_mandatory>
  <field_fk_name>id</field_fk_name>
  <field_fk_table_name>addresses</field_fk_table_name>
 </field>
 <field>
  <field_name>id</field_name>
  <field_type>integer</field_type>
  <field_mandatory>YES</field_mandatory>
  <field_pk>YES</field_pk>
 </field>
</transformation>
```

Altering a table

The last example shows how deleting and adding a column is done. Both of the transformations can be seen in Figure B.7 and the resulting SQL script is shown in Figure B.8.

Figure B.6: SQL script which GES produces from Figure B.5

```
CREATE TABLE people( address_id INTEGER NOT NULL,
     first_name VARCHAR(50),
     id integer NOT NULL,
     last_name VARCHAR(50) );
ALTER TABLE people ADD CONSTRAINT pk_id_people PRIMARY KEY (id);
ALTER TABLE people ADD  CONSTRAINT fk_address_id FOREIGN KEY
(address_id) REFERENCES addresses(id);
```

Figure B.7: Transformations dropping and adding columns to the table

```
<transformation>
 <id>2</id>
 <transformation_type>drop_column</transformation_type>
 <table_name>people</table_name>
 <field>
  <field_name>to_be_removed</field_name>
 </field>
</transformation>
<transformation>
 <id>3</id>
 <transformation_type>add_column</transformation_type>
 <table_name>people</table_name>
 <field>
  <field_name>city</field_name>
  <field_type>VARCHAR</field_type>
  <field_dimension>50</field_dimension>
 </field>
</transformation>
```

Figure B.8: SQL output which GES produces from Figure B.7

```
ALTER TABLE people DROP COLUMN to_be_removed;
ALTER TABLE people ADD city VARCHAR(50);
```

Figure B.9: Running times

transformation type	**running time (ms)**
creating a table	900
creating two tables	1831
altering a table	1450
adding a column	919
altering a column	1135
dropping a column	911

Performance

The running times of the three migrations mentioned and of other types of transformations, are shown in Figure B.9.

Appendix C

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
migrationGen ........................... the directory with executables
src ........................................ the directory of source codes
    migrationGen_src ........................... implementation sources
    thesis ............. the directory of LaTeX source codes of the thesis
text ........................................... the thesis text directory
    BP_Golian_Christian.pdf ............... the thesis text in PDF format
```