FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	CV Management and Printing
Student:	Rudolf Rovňák
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

Desing and develop a universal solution for tracking CVs using the Pharo technology:

- Perform an analysis of an existing solution provided by Tomcat GMBh.
- Perform a requirements analysis based on the former solution and requirements of Tomcat and CCMi.
- Perform an analysis and design of the Pharo backend, select an appropriate persistence solution.
- Implement the backend system based on CV templates and a simple desktop UI to access it.
- Use Pillar library to implement exports.
- Test and document your solution.

References

http://pharo.org
https://github.com/pillar-markup
https://github.com/magritte-metamodel/magritte
http://spec.st

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 12, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

CV management and printing

Rudolf Rovňák

Department of software engineering
Supervisor: Ing. Robert Pergl, Ph.D.

May 14, 2018

Acknowledgements

I would like to thank my supervisor for helping me with all the questions and problems I had.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 14, 2018 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Rovňák, Rudolf. *CV management and printing.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Táto práca sa zameriava na problém efektívnej správy a generovania životopisov v podnikovom prostredí. Text sa zaoberá implementáciou modulárneho systému pre ukladanie životopisov za použitia technológie PostgreSQL a backend aplikácie v technológii Pharo. V práci sa takisto venujem implementácii klienta pre správu dát a generovanie dokumentov za použitia knižnice Pillar a Mustache.

Klíčová slova Podniková aplikace, životopis, Pharo, Postgres, generování dokumentu

Abstract

In this thesis, I focus on the problem of effective management and printing of Curriculum Vitae (CV) in an enteprise environment. I implement a modular system for CV storage using PostgreSQL database and a backend application in Pharo. Additionally, I create a desktop client to modify and manage the data and print documents using modifiable templates using the Pillar library along with Mustache templating engine.

Keywords Enterprise application, Human Resources, curriculum vitae, Pharo, Postgres, Pillar, document generating

Contents

List of Figures

Introduction

A problem faced by many enterprises is finding the workforce needed in a shortest time possible. This process is slowed down by searching and reading through CVs. There is no effective way to search for relevant informations in documents, that are written by multiple people, each of them making his/her own structure. Because of this, a need for saving the data in a structured form emerged. This would allow for businesses to search for relevant skills and information in their applicants.

In this thesis, I decided to address this problem by designing a system for management and printing of CV's. I am creating this solution for a company Tomcat GmbH. They provided me with the know-how needed to create such a software, that will also be usable in real life situations.

There are various reasons for wanting to create a software for managing CV's. This solution allows fast and reliable searching through the data – thus allowing the company to pick the applicants with a skill required or filter applicants according to their education, work experience or every other aspect of the résumé.

The second advantage this solution brings is the ability to use the data to generate documents. A huge advantage for people working in Human Resources is the ability to have the résumés of suitable applicants in a uniform style and structure. Additionally, this allows, both the structure and the style of the document, to be fully customizable. This is useful for selecting only the relevant information, making the CV's more concise and to the point.

Goals

The goal of my bachelor thesis is to develop a SW for CV management and printing, using the methods of SW engineering. This includes:

- Performing an analysis of the existing solutions.
- Gathering the requirements from Tomcat GmbH.
- Performing an analysis for the new IS.
- Developing a complete solution using Pharo technology.
- Testing and documenting the solution.

Chapter 1

State-of-the-art

1.1 Existing solutions

As all of the companies run into a lot of CVs when looking for new employees, SW development companies developed their commercial solutions. There are also some open-source and free options. Some of the options are OpenCATS, Callibrace, Zoho Recruit and many more [1].

There are definitely some advantages to acquiring one of these out-of-the box solutions. Usually, it is cheaper than developing it in house. Also, technical support comes with some of the commercial SW. These solutions also tend to be very extensive and feature rich, stable and well-tested. Also, they may offer integration with other software usually used in an enterprise environment (such as an issue tracker).

Of course, there are also disadvantages to this approach. The extensive features can lead to complicated SW, that can be hard for people to learn and use. A lot of the time, only a small fraction of the SW potential is used, yet the company pays for all the features. Because of this, developing one's own solution can be a good alternative.

The custom-made solution can be heavily focused on the needs of particular company. This focus also brings the number of redundant features down – thus making the software smaller and easier to use. Also, this approach allows for easy extension of the said system. With the commercial SW, implementation of custom features can be very costly and even impossible.

1.2 Document templating

Document templating is also an issue for many enterprises. There are numerous approaches to templating – the simplest being the templating abilities of various office packages (Microsoft Office package, LibreOffice etc.). These solutions offer the ability to create templates and then fill them with data

– but a lot of times, it is complicated by the user interface, incompatibility between versions. Additionally, the export formats are limited and using such templates in a third-party SW is complicated or impossible.

There are commercial solutions, that offer templating capabilities for document generating – one of them being Docmosis. It uses placeholders in regular Microsoft Word / LibreOffice Writer documents. The templates are filled using JSON or XML format of the data. It can be used from many platforms and offers multiple formats of export [2].

My proposed solution is similar to this SW. The main advantage of my SW is it is build solely on open-source SW and frameworks – thus allowing company to further build upon it. The idea of Pharo also supports modification of every used component to suit the company's needs.

1.3 Used technologies

1.3.1 Pharo

Pharo is an open-source programming language and environment. It is written in Smalltalk and it is object-oriented. It offers compact syntax, hot recompilation and a big library of external packages. I have used numerous of those external packages in this thesis.

1.3.2 PostgreSQL

For the database, I have decided to use the PostgreSQL. There were multiple reasons for this choice – it is a mature, well documented implementation of a relational database. It conforms to the SQL standard. It is an open-source project and is currently under active development. It also provides an easy installation and it is multiplatform. Being one of the most popular options, it features good integration with almost any programming language. It is used in variety of large companies – for example Skype, The Federal Aviation Authority, Afilias (which manages the .org domains), LastFM and many more [3].

1.3.3 Glorp

GLORP is a complete object-relational mapper for Pharo. It supports most of the common relational databases. The name stands for *Generic Lightweight Object-Relational Persistence*. It is able to manage transactions, rollbacks, write queries using only the Pharo syntax and more.

1.3.4 Spec framework

Spec is a framework for creating UI in Pharo aimed at reusability of the created widgets. It is a powerful framework, yet simple enough to build a small UI to

create and manage records in my application.

1.3.5 Pillar

"Pillar is a markup syntax and associated tool to write and generate documentation, books and slide-based presentations" [4]. It offers ability to export documents into numerous formats, such as HTML, LaTeX and more. It also offers customization of the export via an STON configuration file and Mustache templates [4].

Pillar is mainly a standalone tool to write documents, such as LaTeX is. For the document generating in my thesis, the interesting part is the integration with Pharo. Pillar offers a document model, a parser and several export types, implemented as visitors over the document model. This is what I have used to create the documents.

1.3.6 REST

REST (Representational State Transfer) is based on HTTP [1]. It is a way to define a web services and data accessing. Every procedure is identified with its URI [2]. REST defines 4 methods for resource accessing:

- GET – used to retrieve the data,
- POST – used to create an entity,
- DELETE – used to delete the resource,
- PUT – change operation, it is similar to POST with small differences [5].

1.3.7 Teapot

To implement REST services in my application, I have used Teapot. Teapot is a simple web framework built on top of the Zinc HTTP web server [4]. I picked this framework because it is very simple and easy to use, yet powerful enough to handle all the needs of my application.

[1] Hypertext transfer protocol
[2] Uniform Resource Identifier

Chapter 2

Analysis and design

2.1 Requirements

When designing an Information system, it is important to determine the requirements for it. Requirements are usually written in natural language. It is important that they are precise, consistent, coherent, verifiable etc. Traditionally, these are divided into two categories: functional and non-functional requirements.

Functional requirements are related to functions of the IS. Non-functional requirements describe the characteristics of the system, a lot of times implicit. Non-functional requirements include security, performance, cost, accessibility, disaster recovery and basically all of the requirements not directly related to the functions of the IS [6].

Logically, the requirements are divided as Must have – these are the requirements, that are crucial to achieve an operational software. In this document, I also define Nice-to-have requirements – the SW is fully functional and working without fulfilling them. Meeting these requirements is not mandatory and depends on the agreement of the two parties [6].

2.1.1 Functional requirements

After the initial consultation with the representative of Tomcat, I have collected these functional requirements for the CV Management software:

- **F1:** The User Interface allows the user to create new records, browse and modify the existing records, delete records.
- **F2:** The UI also allows searching through the existing records.
- **F3:** The SW is able to generate documents (CVs) from the stored data.
- **F4:** Documents are generated using modifiable templates.

- **F5:** The SW should be able to generate the documents at least in one file format - preferably PDF. Other formats are nice-to-have.

The specific data to be stored about each person were defined as the standard informations contained in a CV. Additionally, I was provided with some examples of the existing résumés. I used these as a point of reference and modelled the data structure accordingly.

2.1.2 Non-functional requirements

For the non-functional requirements, we agreed on these:

- **N1:** For data storage, the system ought to use a database (the DB technology is left for my consideration).
- **N2:** The UI is implemented as a desktop client.
- **N3:** The IS is not to be integrated with any other existing SW.
- **N4:** The SW should be easily extended.

Other non-functional requirements, that are not explicitly stated are considered as nice-to-have, as they could be too costly or time-consuming for the sake of bachelor thesis.

2.2 Use-case analysis

"Use case analysis is the most common technique used to identify the requirements of the system, information that is then used to define processes and later to design classes that will ultimately fulfill the use case" [7].

The basic use-case diagram in Figure 2.1. shows the overview of the functions of the application (or the UI, which is basically the only way for user to interact with the whole application). There is only one role – that is the user of the application.

Other tools to further understand the processes in the company and design an application accordingly are the Sequence diagrams and Activity diagrams.

2.2.1 Activity diagram

Activity diagrams help to understand the expected behaviour of the application. It describes the dynamic aspects of the system, from one activity to another. The activity diagram that shows a record creation is in Figure 2.2.

The activity starts with the user filling up the form with the information. After clicking the submit button, the inputs are checked. If everything is OK, the request is created and sent to the server part of the application. On the server, the request is parsed and the desired object is created. The server has

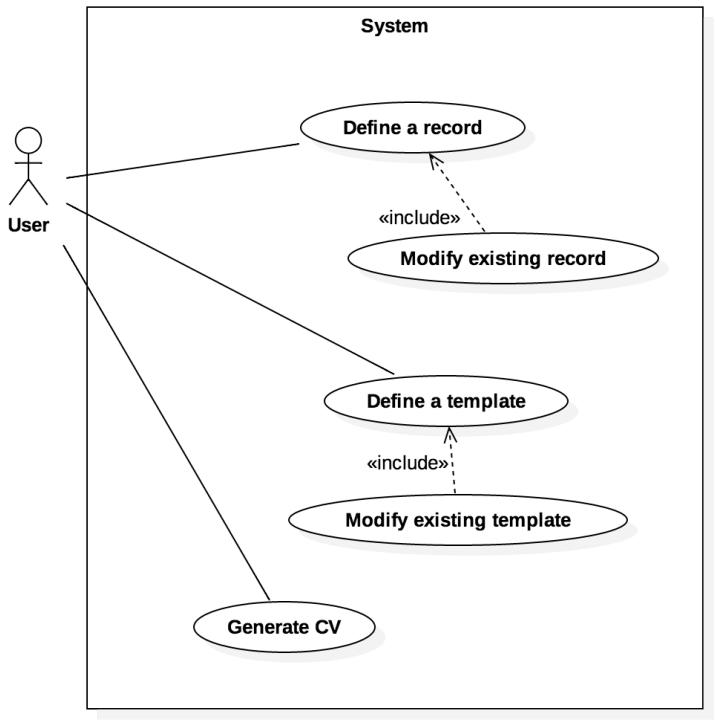

Figure 2.1: Use-case diagram

a session with the database open. This newly created object is registered with the session – that means, that the SQL statement is created from the object and is sent to the database in an enclosing transaction. The DB performs the action. After that, the server retrieves the created record and returns it as a response to the request of the client. The client parses the response, creates an object and the UI displays the newly created record, allowing user to further add and modify the record.

2.3 Application architecture

The very first step when designing an enterprise application is choosing a suitable architecture. There is a lot of ambiguity in defining the word "architecture" in SW development. In [8, p. 1], author states: "*Architecture is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions, that are hard to change.*"

Let's look at two parts this definition consists of. The most significant part from the first part is the *highest-level breakdown*. Enterprise applications

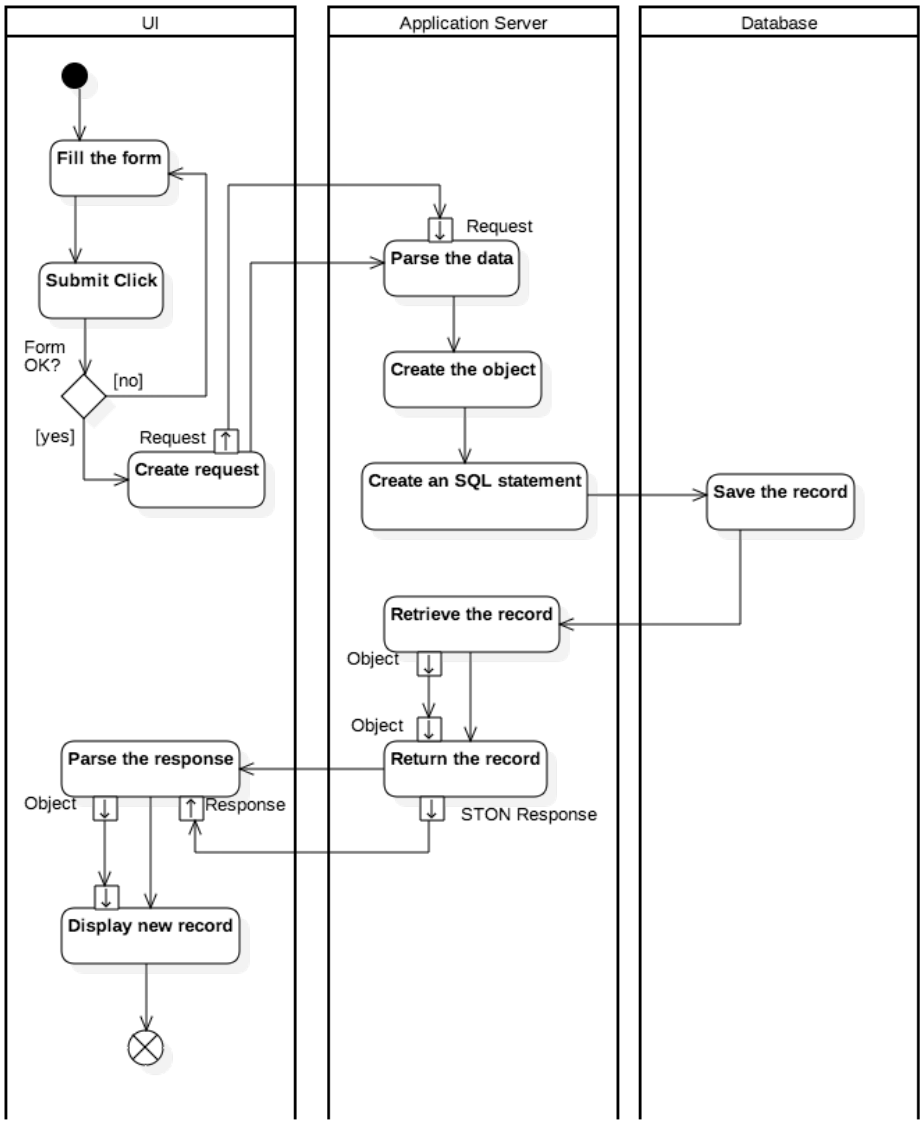

Figure 2.2: Activity diagram for creating a record

are not big monoblocks anymore – they consist of multiple "building blocks", that function separately, but also closely integrate. The way they function and communicate is the subject of the architecture. This corelates to the second part of the definition. Overall logic of the whole system is very hard to change later on in the development.

With that in mind, there are a lot of problems, that share the same con-

cept, therefore could be solved similarly. That is when the term architectural pattern comes into place. "*An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context.*"[9]

Some of the most well-known patterns include[9]:

- Multi-layer pattern,
- Pipe-filter pattern,
- Model-view-controller pattern,
- Event bus pattern.

Each of these patterns are suitable for different kind of problems. For the CV management SW, a split into 3 layers, each of them having its own responsibility, comes to mind. Those are:

- data storing,
- user interface,
- the link between the two former (the logic of the application).

This abstraction is perfectly described by the Three-tier pattern.

2.3.1 Three tier pattern

Three tier pattern (also three layer pattern) divides the applications into 3 parts, according to their responsibilities:

- **Data tier** is responsible for data operation, persistence.
- **Application tier** is the middle layer; it contains business logic, performs operations.
- **Presentation tier** is the part visible to the user. It presents the data; provides user input for higher tiers. [10]

2.4 Persistence

Persistence is a very important term in IT. In a nutshell, it means, that the data created in an application survive even after the process, that created them has ended. For the end user it is normal to expect, that when they save their work in an application, they can come back to it and find it in a state they left it. It can be a document, they write and save to their local disk,

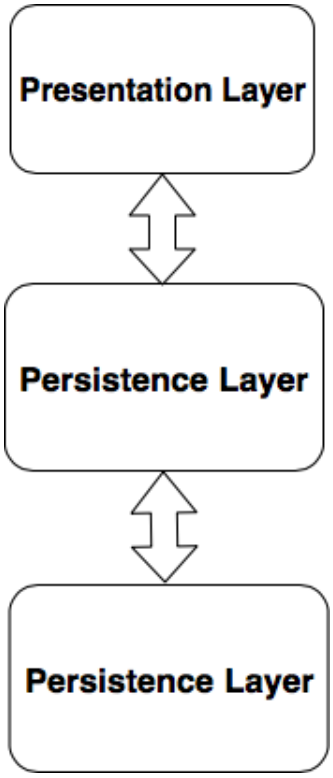

Figure 2.3: Visualisation of the 3 tier architecture

a file they upload to the cloud or simply a web application, that stores the informations about their account etc.

From the various use-cases, it is evident that there are a lot of approaches to storing data. They are all suitable for different purposes, offer different advantages and disadvantages.

2.4.1 Requirements

When considering a persistence solution, it is best to first think about the requirements. What do I expect from the solution? What is the nature of the data? How will my application access the data? What about security? What are the requirements for performance – will only single user work with the system at a given time, or will it be multi-user? How to achieve consistency in the latter? The questions keep coming up the more one thinks about it, so let's try to answer some of these and define the requirements.

I will start with the more obvious answers. My application will definitely not be a single-user one – therefore, I need a solution that supports multi-user access. When I consider the possible future development of this project – for example a web application for applicants to fill out – there could be several,

maybe even dozens of users working simultaneously. Some of them may access the same data, which may cause problem if I used a very simple, naive solution like saving the data to files.

The number of users is closely tied to another aspect of the storage – performance. This means the ability to perform the requests of the users in a timely manner. These days, the speed of the storage solutions is often the bottleneck of the whole application. On the other hand, in the context of my specific system, the expected load is not nearly as big to cause big problems with standard solutions, that I discuss in next sections.

The data storage also needs to be resistent to data corruption in case of unexpected events. The loss of power, network outages, or simply an error in the application code cannot cause the data to become corrupted. In this day and age, information is a very (sometimes the most) important asset of a company. Thus, when considering a data storage, data consistency has to be a priority.

2.4.2 SQL vs. NoSQL

There are a lot of different database technologies these days. There are the classic relational databases (also called SQL databases) and non relational databases (sometimes called NoSQL, also meaning Not Only SQL). Both of these approaches offer very different properties.

The traditional relational databases are the oldest form of DB storage technologies. As the first concept of relational DB dates back to 1970 [11]. these systems came a long way. They are the most mature, widely used and well documented. There are a lot of implementations, some of them being free (PostgreSQL, MySQL), while others are commercial (Oracle).

The main disadvantage I see with the concept of relational DB is the data structure. The data (on the logical level) are stored in tables – in other words, rows and columns. This can sometimes be limiting, depending on the domain.

On the other side, there are numerous so-called No-SQL databases. This term encompasses a wide variety of technologies. Some of them include key-value stores, graph stores or wide-column stores. Compared to relational DB, NoSQL can provide better performance in some cases. They can also work better with object-oriented programming and they are usually easy to scale, as opposed to a monolithic architecture of relational databases [12].

I have considered all these things when deciding on a database technology for my use. Finally, I decided on a relational DB, more specifically PostgreSQL. Some kind of document-oriented database could seem like a better way to go, but I think the advantages it provides would not be used in my case. The scalability and performance are not a big issues for a system in one company. On the other hand, there would be a lot of added work. The documentation and community around these technologies is not very extensive. Furthermore, Smalltalk (and Pharo) are not very widely used, so the support

for these exotic DBs is also not very good – that would lead to a lot of extra work, or it could even be impossible and I would have to decide on a different programming language to write the backend of my application in.

2.4.3 Data structure

In the process of designing a database schema, the best way to describe it is an Entity Relationship Model (ERD). This model describes the entities in the domain and relationships between them. There are a lot of notations currently used – to name some of them, I can mention Chen notation, UML, Bachman or Crow's foot notation. Basically, all of these are different ways to capture the same thing – that is the structure of DB schema, on a conceptual or a logical level.

2.4.3.1 Crow's foot notation

For the data modelling in this thesis, I have decided to use the Crow's foot notation for ERD. This is one of the most widely used types of the notation amongst Software Engineers. It is generally really easy to understand, even for the non-technical audience. As not everyone is familiar with this particular notation, I have provided a small overview of the relationships cardinality notation (Figure 2.4) to better understand diagrams in the next sections. Other aspect of Crow's foot are self explanatory.

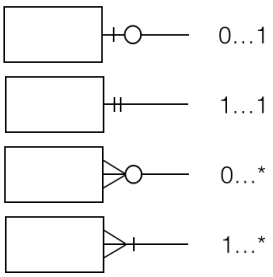

Figure 2.4: Simple overview of Crow's foot notation

There are also three levels on which we can look on the data model, being the Conceptual, Logical and Physical models. I will now demonstrate the transitions from one to another on my domain model.

2.4.4 Conceptual model

"The conceptual model is an abstract form of logical model and it shows all entities at high level without worrying about the detailed structure such as attributes (columns) and its types." [13]

The conceptual model for the CV management persistence is shown in the Figure 2.5.

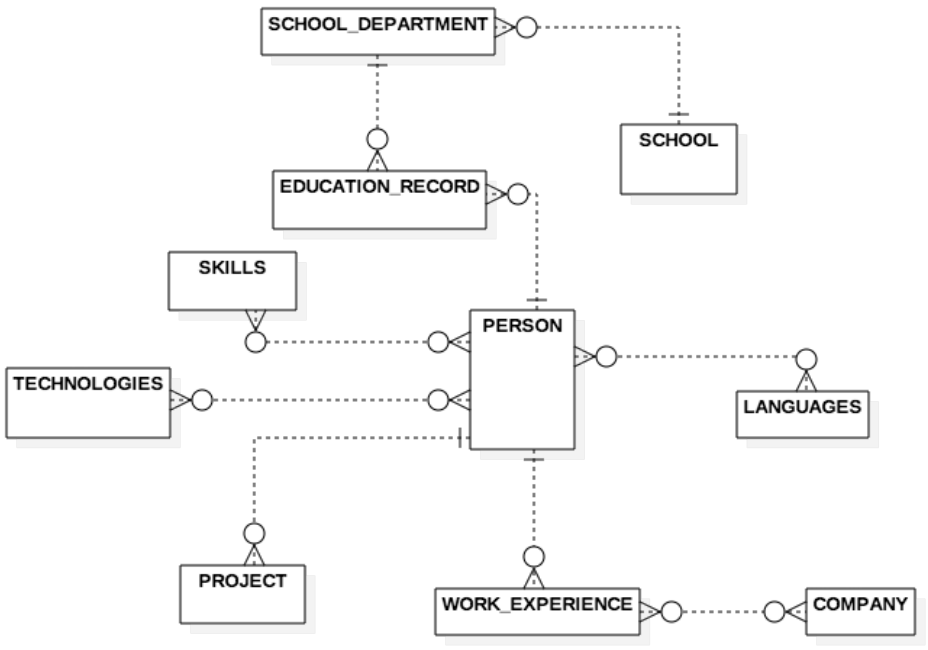

Figure 2.5: Conceptual model for the CV management DB schema

From the diagram, it is apparent, that the main entity is the PERSON entity. It can be considered an envelope that contains all the information needed in the CV. The entities WORK_EXPERIENCE and EDUCATION_RECORD are used to describe information you would most likely expect in a CV. They are quite self-explanatory, they represent records about education or a work experience in a CV. The cardinality is 1:M – there is no point in keeping records without the actual CV. Also, there cannot be multiple people with the exact same records.

After that, there are a few entities describing parts of a CV, that are not so common. They are a result of consultations with Tomcat. The first example is the PROJECT entity – describing any kind of project a person worked on. There are also SKILLS and TECHNOLOGIES – these entities describe various skills a person has. The last entity is used to describe the language capabilities of a person.

2.4.5 Logical model

The Logical model serves as a basis for the Physical model. It describes the

entities attributes Important thing is to keep the Logical model still platform independent – therefore not using specific data types etc. [13]. The Logical model is displayed in the Figure 2.6.

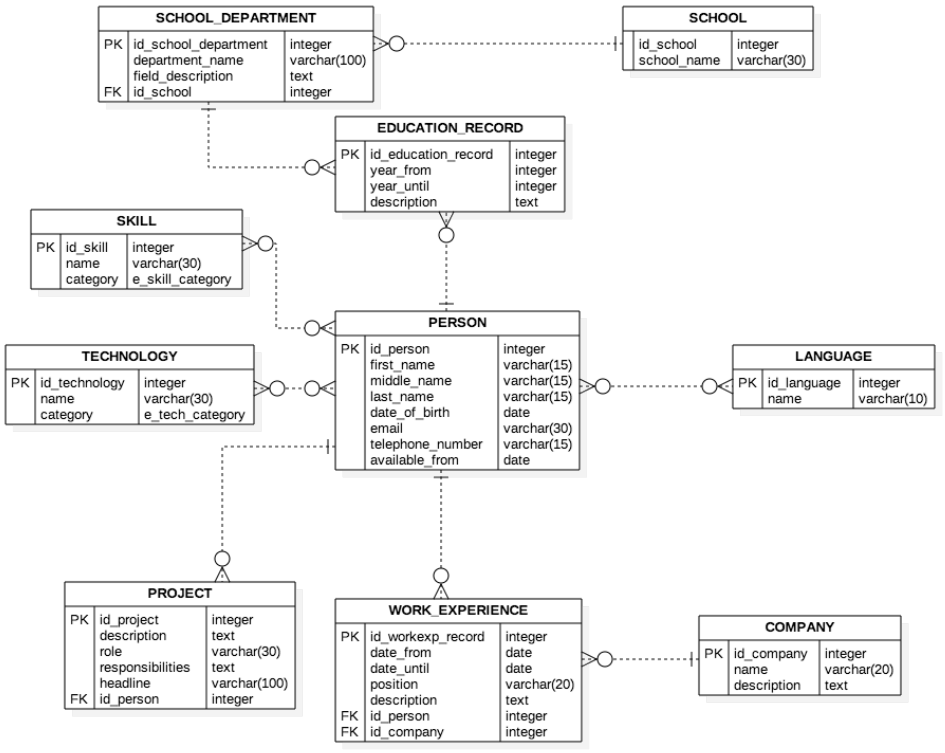

Figure 2.6: Logical model for the CV management DB schema

2.4.6 Physical model

The physical model is an actual representation of the database schema. This model is platform specific, therefore the data types are specific for the particular DB implementation. Additionally, the M:N relations are decomposed into link tables – thus allowing to generate a script to create a schema [13].

2.5 Application tier

The application tier is the "core" of the application. It contains all the business logic, it performs all the operations on the data. Internally, this tier can be divided into two parts. The actual implementation will be described in the

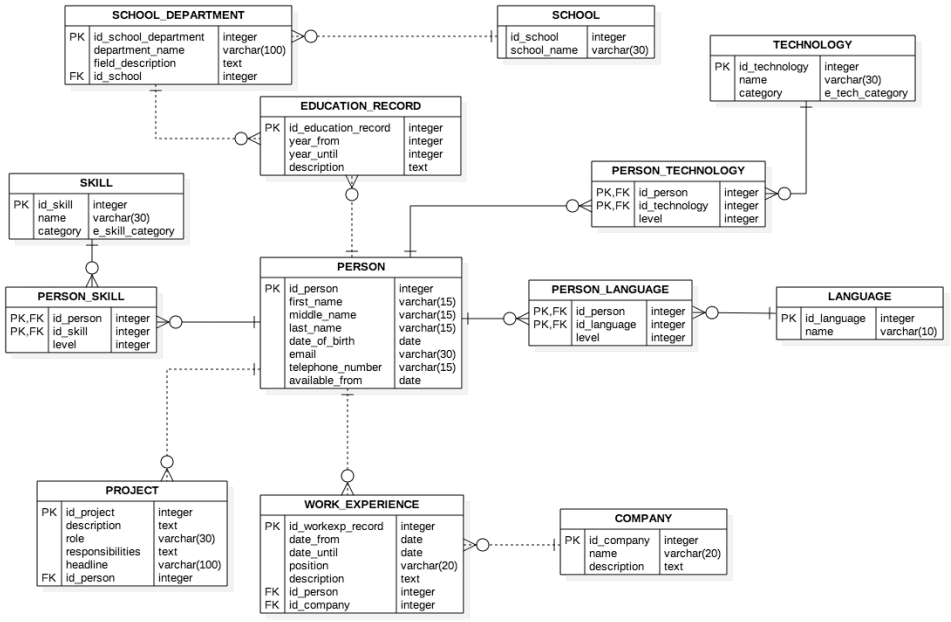

Figure 2.7: Physical model for the CV management DB schema

Chapter 3 – in this chapter, I will describe the architectural patterns, that I have used in this tier of the system.

2.5.1 Object-Relational Mapping

The first responsibility of the application tier is the communication with the DB. There is a problem though – we can observe two ways to look at data. On the DB side, there is a relational schema, therefore tables. On the other side, there is an application server written in Smalltalk – Object Oriented programming language.

To overcome this difference, I have used something called Object-Relational mapping (ORM). There are a lot of frameworks, that are able to simplify this process tremendously. This is usually far better solution than trying to write this layer yourself.

In order to map the relational schema onto objects, it is necessary to create an appropriate class model. The classes themselves are very simple. The mapping is described in Chapter 3.

Figure 2.8 shows the class diagram of the object design for the ORM mapping. It is not a coincidence, that it is very similar to the Physical schema of the database, which it basically mirrors. The interesting thing (from the object point of view) are the link entities – they were necessary because of

the ORM mapping framework I chose. I describe this in more detail again in Chapter 3.

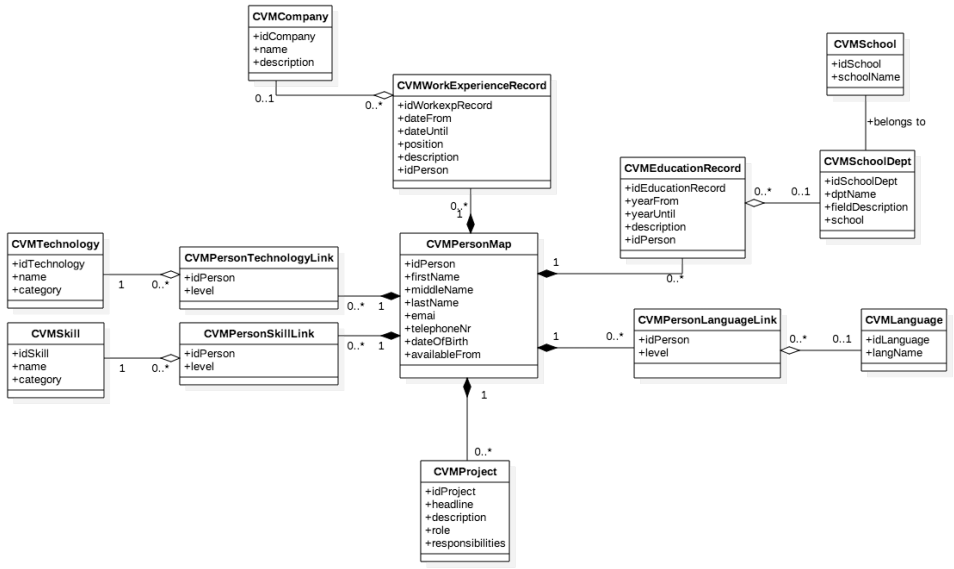

Figure 2.8: Class diagram for the ORM mapping

2.5.2 REST API

The second responsibility of the application server is communication with the UI – getting the input from the interface, processing it and deciding, which data to display. Again, there are a lot of technologies that solve this. One of the most used approaches is a communication via REST API.

Main advantages of this approach of API is its simplicity – it can be reused by many frontend applications. Additionally, the change of persistence will not affect them – the endpoints will stay the same.

2.6 User interface

User interface (UI) allows the user to communicate and control the SW. In the last decades, the Information systems are becoming more and more complicated. The main goal of a good UI is to make all the features available in an easy manner, without requiring a special knowledge from the user.

There are two main categories of the UI. Historically, CLI (Command line interface) was heavily used. It was mainly caused by the limited graphical capabilities of the computers. Although, this way of controlling programs still

has its advantages, even today. Command line users only utilize the keyboard which can lead to faster speeds. The CLI is also less demanding on computer's resources, it is less demanding to develop and is very easily extensible [14]. One of the examples of SW, which relies heavily on CLI is for example Git.

On the other hand, GUI (Graphical User Interface) has been gaining a lot of popularity in the past years. The biggest advantage is the visual intuitiveness – the users are much more likely to pick up, how to use the SW. It also enables users to multi-task. The GUI is more difficult and time consuming to develop. It also requires more system resources as there are a lot more elements that need loading (although this is becoming less of an issue thanks to the performance of today's PCs) [14].

2.6.1 Requirements

The best way to determine, which route to take is to look at the requirements and use-cases of the application.

For my application, I have decided to go with the GUI. In this particular case, the choice is apparent – the application is designed for an enterprise environment and it is primarily meant to be used by employees. It will be far easier for these users to understand the program via GUI. This alone is a reason good enough to choose graphical interface, but let's look at other reasons why this choice is better.

The big part of the use-case is creating new records and managing the existing ones. This requires to gather a lot of input data – this is easily done by forms, whereas it would be a big challenge with CLI. Also, the displaying of the data will be a lot more user friendly and intuitive.

There is a place for a CLI in this use-case too. It could be a good overlook for the future development to implement input of the existing documents. This interface would allow for easy scripting (access from Bash, for example). The application could parse multiple records and create records from these. I have decide not to implement this functionality, as it could be too time consuming. Additionally, there is a big problem with formats used for CVs – PDF, DOC/DOCX etc. Those are either too difficult to process or have a proprietary structure, that makes this problem more complicated.

2.6.2 Structure

As I described in Section 3.4, the framework I have used to implement the UI is focused on building reusable widgets. This leads to a object model, that relies heavily on composition. The simple class diagram to describe the class structure of the UI implementation is shown in Figure 2.9.

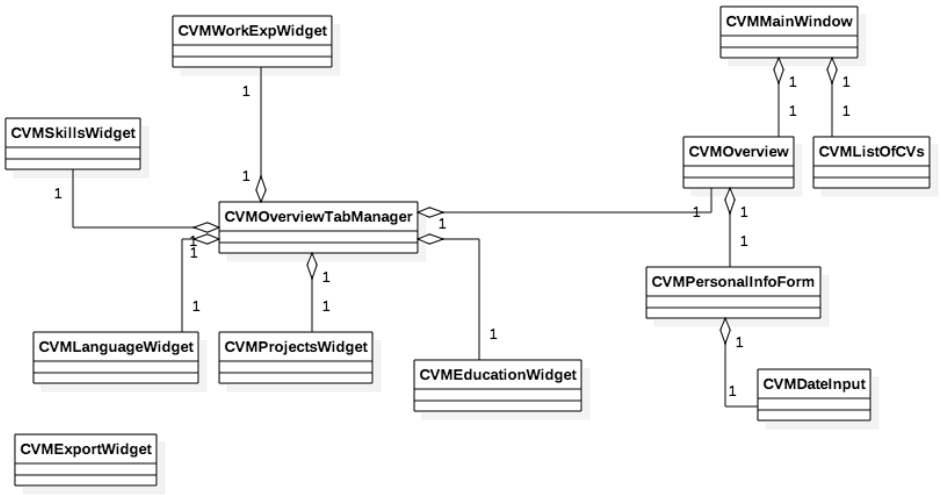

Figure 2.9: Class diagram for the UI structure

Chapter 3

Implementation

3.1 Database

3.1.1 DB Logic

PostgreSQL offers a very powerful tool to implement logic into a DB – PL/pgSQL. It is a procedural extension of SQL, which allows user-defined procedure and functions to be implemented and stored in the database.

This approach adds load to a DB server and adds more layers of complexity to the SW. I have decided to keep the database as simple as possible. Therefore, the only responsibility of the DB server is data storage. In the future, this procedural extension could be used to implement some triggers or procedures that could be used for logging, for example.

3.2 ORM

To be able to work with a relational DB within a programming language, there is a need to use an ORM framework. In Pharo (and Smalltalk), there are multiple choices. They are different in their level of abstraction from the actual relational database.

When I started with the development, I had decided to use a Garage database driver. As I progressed, I have decided to change it entirely. I rewrote the whole communication with the DB using GLORP framework. I found the comparison between this two approaches very interesting, therefore I will now describe them separately and then compare them.

3.2.1 Garage driver

Garage is a simple database driver for Pharo. It provides API to connect to most common relational databases.

The main caveat with this approach is, that it still relies heavily on SQL. The driver provides support for prepared statements, simple binding of results etc. Using it would mean writing a lot of declarative code. Using the SQL in the code would make it a little more complicated to extend the data model. From a security standpoint, the programmer has to be aware of the possible SQL-injection attacks.

Because of these reasons, I did more research and then decided to rewrite this part of an application using different approach.

3.2.2 GLORP

The first step is to create a simple class, that will serve as a model of the data from the DB. It is necessary to also include accessors (getters and setters) for all of the classes concerning Glorp.

"Glorp models all the involved concepts (such as tables, columns, classes, etc.) as first class objects, and then links instances of those objects in a DescriptorSystem. It is the core of a Glorp system, it holds all the Glorp metadata […]" [15]

In practice, this means I have described the DB schema, the object model and the mapping between them in a subclass of `DescriptorSystem`. I will now present this in practice.

3.2.3 GLORP in use

The first step is to define a subclass of `DescriptorSystem`:

```
DescriptorSystem subclass: #CVMGlorpDescriptor
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'CV-management-db-connection'
```

Now, I will take a table from the data model (let's say the SCHOOL_DEPARTMENT table) and create a description for it. This takes place in a method `tableForSCHOOL_DEPARTMENT`. The thing I would like to point out here is that I have used a strict rules for notation:

- when referencing tables, I have used all caps, underscore as a delimiter,
- when referencing classes, I have used camel case.

From that, it is apparent that the mentioned method describes a table in a database schema, although it does it in a Pharo syntax. The methods for other tables follow the same naming schema.

Therefore, the example code to describe the table is as follows:

```
CVMGlorpDescriptor>>tableForSCHOOL_DEPARTMENT: aTable
    | schoolId |
    aTable
        createFieldNamed: 'department_name'
        type: (platform varchar: 100);
        createFieldNamed: 'field_description'
        type: (platform varchar).
    (aTable
        createFieldNamed: 'id_school_department'
        type: platform sequence) bePrimaryKey.
    schoolid := aTable
        createFieldNamed: 'idSchool'
        type: platform integer.
    aTable
        addForeignKeyFrom: schoolId
        to: ((self tableNamed: 'SCHOOL') fieldNamed: 'id_school')
```

From the code, it is apparent that the whole structure of the table is described – including primary keys, foreign keys and platform specific data types.

The second step is creating a class model. In this example, I am creating a class model for the class `CVMSchoolDept`, therefore the corresponding method will be named `classModelForCVMSchoolDept`. Again, just from the naming scheme, it is apparent that this is an object model. The code to define a class model is very simple:

```
CVMGlorpDescriptor>>classModelForCVMSchoolDept: aClassModel
    aClassModel
        newAttributeNamed: #dptName;
        newAttributeNamed: #fieldDescription;
        newAttributeNamed: #school type: CVMSchool;
        newAttributeNamed: #idSchoolDpt
```

The one interesting thing in this example is the attribute `school` – it defines its type to another class. This helps the framework to determine relations. Thus, every instance of the school department class will have an instance of the corresponding school as its instance variable.

Now that I have created the description for a DB table, a model for a class, all that is left is to connect these two things. This is done via descriptor – the naming of the methods follows the same pattern. Thus, for the example, the code for mapping is:

```
descriptorForCVMSchoolDept: aDescriptor
    | table schoolTable |
    table := self tableNamed: 'SCHOOL_DEPARTMENT'.
```

```
aDescriptor table: table.
schoolTable := self tableNamed: 'SCHOOL'.
aDescriptor table: schoolTable.
(aDescriptor newMapping: DirectMapping)
    from: #dptName
    to: (table fieldNamed: 'department_name').
(aDescriptor newMapping: DirectMapping)
from: #fieldDescription
to: (table fieldNamed: 'field_description').
(aDescriptor newMapping: DirectMapping)
    from: #idSchoolDpt
    to: (table fieldNamed: 'id_school_department').
(aDescriptor newMapping: OneToOneMapping)
   attributeName: #school;
   join: (Join
       from: (table fieldNamed: 'id_school')
       to: (schoolTable fieldNamed: 'id_school'))
```

There are multiple ways to map the values in a descriptor. The ones I have used are:

- `DirectMapping` – this is used to map values (numbers, strings, dates…)
- `OneToOneMapping` – this mapping is used to map a single reference to another object. The only information necessary is the name of the attribute to map – Glorp determines the instance from the ClassModel, foreign keys from the table definitions etc.
- `OneToManyMapping` is used to map a collection of object to a parent object.
- `ManyToManyMapping` is similar to the previous one. The difference is that Glorp expects a link table (as this is the way M:N relationships are implemented in a relational DB). [15]

3.2.4 Transactions

As a way to preserve atomicity of the data, relational databases use the transactions. They allow user to rollback the changes or commit them in batches. In Glorp, this is ensured via *UnitOfWork*, which keeps track of all the persisted objects and changes made to them. It works at the object level, therefore it enables the changes to be in synchronization with the database transactions.

The *session* object provides a way to control units of work, specifically with messages `beginUnitOfWork, commitUnitOfWork, rollbackUnitOfWork`. I have also used the message `inUnitOfWorkDo:`, which takes a block as an argument. This message is self-explanatory – it takes the block and executes it

in one unit of work. I used it to implement simple operations like inserting or deleting new records, just because it is more concise and it makes the code shorter.

In the tests, I used the message `commitUnitOfWorkAndContinue`. It takes the current unit of work, commits it and creates a new one. This helped with tests, where I accessed the DB multiple times and verified the results after each operation.

3.2.5 CRUD operations

CRUD (Create, Read, Update, Delete) operations are very simple with Glorp. Again, it is solely on the object level – synchronizing with the DB is done via units of work by the framework.

Reading instances from the database can be done in 2 ways, depending on the expected result. Those are:

- `readOneOf:` returns a single instance,
- `readManyOf:` return a collection.

The criteria for the query are specified in the block after the `where:` part of the message (which is optional, just as the WHERE clause is optional in an SQL SELECT statement). The one thing to be aware of here is that the block passed to the `read:where:` message is not a regular Pharo block – it is translated into a WHERE clause by Glorp [15].

Creating a new record in a database works simply – the newly created object is passed as an argument of a `register:` message. Deleting works the same way, only the message sent to the session is `delete:`.

Updating the records in the DB is the simplest of all the operations. Once the object is registered, all of the changes are handled automatically within units of work by Glorp.

There are other aspects of Glorp and the way it handles ORM mapping. It would be too time and space consuming to describe them in this thesis. The best resource to find more information is [15].

3.3 Application server

For the application server part of the SW, I have decided to use Teapot.

One of the main responsibility of the server is to send data to the client. As the server works on HTTP, there needs to be a way to transfer objects in a text form. There are multiple approaches used these days. One of the most popular (and most used with the REST API) is JSON format. I have decided to go one step further and use another format – STON.

3.3.1 STON

STON is short for *Smalltalk object notation*. It is a text-based data exchanged format, backwards compatible with more popular JSON. On top of JSON, STON allows symbols (globally unique strings) – allowing for simpler and more readable map keys [4, p. 119].

The usage is very simple – the Teapot offers the ability to configure the default output to be the STON. The STON implementation in Pharo provides users with reader and writer classes – thus allowing easy serialization of objects and easy reading from strings containing serialized objects.

3.3.2 GET Methods

The GET methods are used to retrieve a resource from the server. In my application, these can be divided into multiple categories.

The first category are the simplest methods, that return the collections of all the records currently saved. The URI for these is simple - it is a simple plural noun to describe the entity.

The second category contains methods to retrieve one particular instance. These are handled in two ways, depending on the entity – either by an ID (primary key from the DB) or by name. The approach with an ID is self-explanatory. The request contains an ID of the entity to retrieve, the response is the found entity, or *nil* in case it is not found. The approach for text-based filtering is used for entities such as `CVMTechnology`, `CVMSkill` and some others. I chose this because of the way user input is collected – the user inputs the record by name. Therefore, at the time of saving the record, it is not known, if the records exist. The searching is case insensitive.

The last category is used to retrieve the context for document generating (for more details on this, see section 3.5.2). The query contains the desired format and ID of the cv to generate. It returns a dictionary with the generated elements.

3.3.3 POST Methods

There are POST endpoints created for every entity in the domain model. The data needed to create the object are retrieved from the HTTP request from the client.

A few endpoints, implemented as POST could be considered as a modification of an existing record. Those are the endpoints that serve to create a link entity – from a logical view, that is a modification of the existing record. From an implementation standpoint though, it is a creation of a new instance, new record that is linked to an existing entity, therefore I have decide to keep it as a POST method.

To make working with these endpoints easy and to make the client code compact, the responses of these endpoints vary. The simpler return the newly

created record. The ones that deal with creating link entities typically return the new collection of the records of said type, corresponding to the record with an ID provided in the request.

3.3.4 DELETE Methods

The DELETE methods work according to the primary keys of the entities. There are endpoints for every entity in the model. They are very similar to GET methods.

3.4 User interface

User interface is one of the most important parts of an application – it is the part, that the user will see and work with. If it is not well arranged and comprehensible, it can destroy the user experience. I have created a simple UI, more specifically a desktop client using Spec framework.

The main window allows the user to create new CV or manage the existing ones. It offers an easy to navigate tab view, that spans into the view for a particular record. There, a user can modify a CV. Everything is neatly organized into tabs. The one detail making the work with the UI faster is the implementation of switching the active element using the TAB key on the keyboard.

3.4.1 Communication with the server

The UI communicates with the application server via HTTP requests. It calls the endpoints that are published by the server.

To concentrate the code used to communicate with the server, I created a helper class `CVMApiConnect`. This class has a server configuration (mainly the URL, port) set in the `initialize` method. For sending requests, it is possible to always create a new instance of this class.

Every method in this class works in the similar fashion. It uses the `ZnClient` to build the request. It the retrieves the response, parses it using `STONReader` and returns the object returned by the server. When writing the code in the UI, this creates the illusion of direct connection to the server (or database). It is also easier to extend and debug it, as it is concentrated in one place.

3.5 Document generating

One of the main features of the developed application is the ability to generate CVs using modifiable templates. For this, I have decide to use the Pillar library along with the templating engine Mustache.

3.5.1 Pillar

The whole document in Pillar is represented by the `PRDocument` class. When building a document, I have created an instance of this class and added the elements into it. At the same time, I create a Dictionary, containing each of the created elements of the document under its unique key (for the template use, for more details, see subsection 3.5.3. To understand this model, these are the elements used:

- `PRHeader` represents a header in a document. In my SW, I made sure to always specify a level of the header.
- `PRText` is the most simple (and probably the most used) element. It is a simple string to be inserted into the document.
- `PRSection` represents a section of the document. I have used it to distinguish parts of the document.
- `PRUnorderedList` and `PRListItem` are pretty self explanatory – they represent a list (unordered) and the items of the list. I used this to display the personal information of the person at the beginning of the document.
- `PRMailLink` creates a mailto link in the document. This allows user to mail the applicant just by clicking the e-mail address in the generated document.
- `PRTable`, `PRTableRow` and `PRTableCell` are used to create tables.

3.5.2 Mustache

To further understand the exporting capability of my SW, it is also important to take a look at the Mustache templating, as my model relies heavily on templates. The SW is able to generate a very simple documents without the templates, although to achieve the best results, it is recommended to use a template to generate a document.

Mustache is a logic-free templating format, designed mainly to be a templating engine for HTML, although it is useful in other areas too.

Templates using Mustache are very simple to understand. It uses the expression `{{{}}}` to delimit a variable, or a *tag* inside a template. When the template is loaded and evaluated, the tags are replaced by the values given in the context.

In Pharo, the simplest way to provide a context for a template is to create a Dictionary. The keys of this dictionary are the names of the tags used in the template, and the values are the elements of the document to be inserted in the place of this tags.

This allows user to modify the structure of the document in an easy manner. The user can choose, which elements he wants to pick in the generated document and what order and structure they will have.

A very simple template for an HTML document using Mustache looks as follows:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>{{{title}}}</title>
  </head>
  <body>
    <div class="personal-info">
      {{{personal_information}}}
    </div>
    <div class="education">
      {{{education}}}
    </div>
  </body>
</html>
```

In this case, the templates is very simplified. This simple approach allows user to modify the styling and structure of the document the way they would on a particular one, only the data are dynamically added later.

3.5.3 Structure of the elements

In order to create the document and evaluate the template, it is necessary to create the structure of the elements. I will now describe some of the models I have decided to use in order to generate the document from the data stored in the DB.

Let's look at the project record. In the object model, projects are represented as a collection of `CVMProject` inside a `CVMPersonMap` instance. Therefore, when processing these records, I have decided to first, generate a header for the section. Then, inside a new section, I incrementally add each of the project records.

3.5.4 Exporting

So far, I have only discussed building the document using the Pillar classes. To export the document into formats I have chosen, there is one more step to take. As I have mentioned in section 3.5.1, Pillar offers ability to export documents from this data model.

Therefore, when generating a context for a template, after creating the document model (which is common for all the formats), I use the writers

(specifically `PRHTMLWriter` and `PRLaTeXWriter`) to transform each element of the said dictionary. This is then used as a context for the template.

This approach brings a lot of possibilities for further develop the generating capabilities. It is very easy to:

- add more possible tags in the templates (even creating redundancies), in order to let user specify the structure of the document in more detail,
- add more export formats – the ones offered by Pillar, possibly even writing new visitors over the `PRDocument`,
- add new data structures, or specify multiple ways to export the same element of the document.

3.6 Testing

When developing a program, it is impossible to avoid errors. Some of them are easy to eliminate – for example syntactical errors (violating the rules of the programming language). These errors lead to the program not building (therefore, not running). In most cases today, the compiler/interpreter is a big help. It locates the problem and sometimes even provides a suggestion for a solution.

The errors, that are far more difficult to find are the semantic ones. In this case, the program can be built and runs, but it does not behave as expected. The main caveat in recognizing semantic errors is that there is no way to determine, what the programmer meant – therefore we have to test the SW in order to assure the expected behaviour.

"A test is the act of exercising software with test cases. A test has two distinct goals: to find failures or to demonstrate correct execution." [16]

3.6.1 Tests classification

There are a lot of ways to look at tests, depending on the way the SW is tested or the focus of the test. From the scope of the performed tests, we distinguish these types of tests:

- **Unit testing** tests individual SW components (classes, functions etc.) Sometimes, it requires creating a driver code and is often performed in debugger.
- **Integration testing** focuses on multiple components and their communication.
- **System testing** tests a complete, deliverable product. [17]

According to the type of performed tests, there are two main types:

- **Functional testing** is performed without the regard to the source code. These tests are modelled after the requirements and specifications. It is also called *black-box testing*.

- **Structural testing** is based on the source code and the data structures used in it. It is also known as *white-box testing*. [17]

Additionally, the term *regression tests* encompasses all of the categories above. Regression testing is testing of parts and features of the application, that have been successfully tested in the previous versions of the SW. It is necessary to perform these to make sure the additions did not break something.

3.6.2 SUnit

SUnit is a framework for test creation and deployment. The main focus is on unit tests, but it can be also used for integration and functional testing as well. The tests written using this framework are self-checking and it allows to organize the tests.

All of the tests in SUnit are written as a subclasses of an abstract class `TestCase`. The tests are run as a new instance of the user-defined subclass of `TestCase`, first running the `setUp` method, the test itself and finishing with the method `tearDown`.

3.6.3 SUnit in use

For a practical demonstration, I will describe the use of SUnit to test my implementation of an ORM mapping. It nicely shows the basic usage of the possibilities provided by the `TestCase` class. Although tests for communication with the DB are not generally considered as Unit tests, they can be easily written as such, therefore the use of this framework is in place.

For starters, I created a subclass of `TestCase` called `TestCVMDbAccessor`. If I started writing the tests as-is, I would face the problem of initializing the connection to the database in every test, as well as properly closing the connection. That is where the `setUp` method comes in – it serves perfectly to initialize this connection, while `tearDown` closes it.

Writing and performing the tests also benefit from the superclass. The `TestCase` provides the `assert` message. It expects a boolean argument. When the value is true, the test passes. It fails when the value is false. Of course, there can be multiple `assert` calls in one tests.

I created a suite of tests to test the communication with the database and all the endpoints published by the application server.

3.6.4 Code review

Code analysis is a technique, where the developer (or other programmer) reads the written code in order to find logical errors in an application. I performed this kind of tests during the whole implementation phase.

Conclusion

The goal of this bachelor thesis was to design and implement a software for managing data about CV and to implement a document generating function, that would use a simple templates to personalize the output of the application.

In this thesis, I analysed the existing commercial and open-source solutions for this problem. After that, I gathered the requirements from Tomcat GmbH and designed the architecture of the application according to their needs. I designed a persistence solution, an application server and a simple UI to access all the features. Additionally, I have implemented a simple document generating function using Pillar and Mustache templates. I tested the solution, mainly using the SUnit framework for Smalltalk.

The output of this thesis is a standalone three-tier application. It is ready to be further extended, mainly extending the data model. This is made simpler thanks to the use of the Glorp ORM framework I used. Furthermore, there is a potential to extend the templating possibilities and formats of the export.

Thanks to the application architecture, it is easy to add different presentation layers. Therefore a web interface, CLI or a mobile application could be developed, using the REST API provided by my solution.

Bibliography

[1] Ingwersen, H. The Top 8 Free/Open Source Applicant Tracking Software Solutions [online]. September 2016, [cit. 2018-04-16]. Available from: https://blog.capterra.com/top-8-freeopen-source-applicant-tracking-software-solutions/

[2] Introducing Docmosis [online]. 2018, [cit. 2018-05-14]. Available from: https://www.docmosis.com/how-it-works/introducing-docmosis.html

[3] 2ndQuadrant. Who uses PostgreSQL? [online]. 2018, [cit. 2018-05-05]. Available from: https://www.2ndquadrant.com/en/postgresql/who-uses-postgresql/

[4] Cassou, D.; Ducasse, S.; et al. *Enterprise Pharo: a Web Perspective*. Square Brackets Associates, october 2016, ISBN 978-1-326-65097-1, 267 pp.

[5] Malý, M. REST: architektura pro webové API [online]. august 2009, [cit. 2018-05-05]. Available from: https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/

[6] Software requirements [online]. 2018, [cit. 2018-05-04]. Available from: https://www.tutorialspoint.com/software_engineering/software_requirements.htm

[7] Taylor, A. Analysis, Design, and Development Techniques with J2EE [online]. june 2003, [cit. 2018-05-05]. Available from: http://www.informit.com/articles/article.aspx?p=31942&seqNum=5

[8] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003, ISBN 0-321-12742-0.

[9] Mallawaarachchi, V. 10 Common Software Architectural Patterns in a nutshell [Online]. September 2017, [cit. 2018-04-10]. Available from: https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013

[10] Three-tier architecture. May 2015, [cit. 2018-04-10]. Available from: https://managementmania.com/en/three-tier-architecture

[11] Pokorný, J.; Valenta, M. *Databázové systémy*. Česká technika – nakladatelství ČVUT, 2013, ISBN 978-80-01-06212-9, 274 pp.

[12] mongoDB. NoSQL Databases Explained [online]. 2018, [cit. 2018-05-05]. Available from: https://www.mongodb.com/nosql-explained

[13] Rayan, J. C. Data modelling using ERD with Crow Foot Notation [online]. February 2015, [cit. 2018-05-05]. Available from: https://www.codeproject.com/Articles/878359/Data-modelling-using-ERD-with-Crow-Foot-Notation

[14] Command line vs. GUI [online]. december 2017, [cit. 2018-05-07]. Available from: https://www.computerhope.com/issues/ch000619.htm

[15] Maringolo, E.; Pratt, N.; et al. Object-Relational Persistence with Glorp. may 2017, [cit. 2018-05-08]. Available from: https://files.pharo.org/books-pdfs/booklet-Glorp/2017-05-02-Glorp-Spiral.pdf

[16] Jorgensen, P. C. *Software Testing: A Craftman's Approach Second Edition*. CRC Press LLC, 2002, ISBN 0-8493-0809-7, 359 pp.

[17] Whittaker, J. A. What Is Software Testing? And Why Is It So Hard? *IEEE Software*, January/February 2000: pp. 70–79.

Appendix A

Acronyms

CLI Command Line Interface

CV Curriculum vitae

DB Database

GUI Graphical User Interface

OOP Object Oriented Programming

ORM Object Relational Mapping

SW Software

UI User Interface

IS Information System

Appendix B

Installation instructions

The first step to run the SW is to have a functioning PostgreSQL database running. The scripts to create the schema are enclosed – just run them.

To run the application, the easiest way is to load the enclosed Pharo image. Then, in the playground, run these two commands:

```
CVMApiServer new start.
CVMMainWindows start
```

The other way is to file-in the created SW. For it to work, it is necessary to have a functional Glorp, Teapot and Pillar. The script to install these frameworks is also enclosed – just run it in the Playground.

Appendix C

Content of the enclosed SD card

- `readme.txt` ..... the file with CD contents description
- `db-ddl-scripts` ..... directory with the database scripts
- `src` ..... directory with the implementation
  - `CV-management-api-server.st` ..... file with the API server implementation
  - `CV-management-db-connection.st` ..... file with the DB connection implementation
  - `CV-management-generating.st` ... file with the document generating implementation
  - `CV-management-ui` ..... file with the UI implementation
- `pharo-image` ..... image for Pharo with functioning code
- `frameworks-install.txt` . instructions to install the needed frameworks
- `thesis` ..... directory with the thesis source and PDF
  - `thesis-src` ..... directory with the thesis source codes
  - `thesis.pdf` ..... thesis in the PDF format