An Automated Language Translation System for Classical Music Titles

M.J.C. Dingjan
R.W. Kapel
M.P. Sijm

TU Delft

muziekweb

(This page has been intentionally left blank.)

An Automated Language Translation System for Classical Music Titles

by

M.J.C. Dingjan
R.W. Kapel
M.P. Sijm

Bachelor End Project
at the Delft University of Technology,
to be presented on 4 July, 2017 at 10:00 AM.

Project duration:	April 24, 2017 – July 7, 2017	
Thesis committee:	Dr. ir. C.C.S. Liem MMus,	TU Delft, supervisor
	Ir. O.W. Visser,	TU Delft, BEP coordinator
	Dr. I. H. Vroomen,	MuziekWeb, client adviser
	C. Karreman,	MuziekWeb, client adviser

This Final Report is confidential and cannot be made public until July 4, 2017.

An electronic version of this report is available at `http://repository.tudelft.nl/`.

TU Delft muziekweb

(This page has been intentionally left blank.)

Preface

This report concludes the Bachelor End Project, the last compulsory course needed to obtain the Bachelor's degree in Computer Science at the Delft University of Technology. This project, done in the course of ten weeks, was done at Muziekweb, the biggest music collection of the Netherlands, in order to enrich the contents of their database and website. The purpose of this report is to give the reader an insight on this process and the design of the final product.

We would like to thank all employees at Muziekweb for their support and enthusiasm during this project. In special, we would like to thank the following people at Muziekweb: Casper Karreman, who was the technical supervisor of this project and gave us weekly feedback on our process; Yvo Neelemans, for giving suggestions and feedback on the product and almost hitting us with his toy drones; Ingmar Vroomen, who has created the project initially and managed the project; Hans Jacobi, who has manually translated 1000 classical music titles to kick-start our project; Thomas op de Coul, who has been our main contact point for questions about the classical music titles.

Special thanks go out to our university coach, Dr. ir. Cynthia Liem MMus from the Multimedia Computing Group, for guiding us through the project and giving us feedback on draft deliverables when needed.

M.J.C. Dingjan
R.W. Kapel
M.P. Sijm
Delft, June 2017

(This page has been intentionally left blank.)

Summary

Muziekweb is the biggest source of music that has been released in the Netherlands. At this moment, all of their data is currently in Dutch. However, Muziekweb wishes their classical music titles to be translated to other languages as well. Since over 250.000 classical music titles exist, it is too much work to be done manually. Therefore, Muziekweb requested a program which can automatically translate titles. With these translations, not only can they deliver their data to more people, it will be easier in the future to link their data to other international sources.

Research has been done on these titles on how they are structured. This knowledge is required to split up these titles in smaller parts. Some of these parts can only have a limited set of Dutch words in them. Therefore, these translations can be translated using a lookup table. Besides these parts, some parts cannot be as easily translated, since they have a cultural history behind them. Therefore, these parts need to be translated differently. To do this, a connection is set up with external databases to find translations for these parts. To be able to improve this program in the future, the program is extensible as well, either for translating to more languages or linking up with more databases.

(This page has been intentionally left blank.)

Contents

1 Introduction **1**

2 Problem Definition **3**
2.1 Problem Definition . . . 3
2.1.1 An ‘Informal’ Definition . . . 3
2.1.2 A Formal Definition . . . 4
2.1.3 The Translation Problem . . . 5
2.2 Requirements . . . 6
2.2.1 User Requirements . . . 6
2.2.2 Non-functional Requirements . . . 7

3 Problem Analysis **9**
3.1 Splitting the Problem . . . 9
3.2 Translations of Components Without Semantics . . . 10
3.2.1 Generic Names . . . 10
3.2.2 Composition of the Orchestra . . . 10
3.2.3 Numbers . . . 10
3.2.4 Keys . . . 10
3.3 Translation of Nicknames . . . 11
3.3.1 Direct Translation . . . 11
3.3.2 Translation Using External Databases . . . 12
3.4 International Music Cataloguing and Current Approaches . . . 14
3.4.1 A Global Network of Music Metadata . . . 14
3.4.2 International Standards . . . 14
3.4.3 The Effects of International Music Cataloguing . . . 15
3.4.4 Muziekweb and Music Cataloguing . . . 15
3.5 Machine Learning . . . 15
3.5.1 What Is Machine Learning? . . . 16
3.5.2 Multiple Algorithms . . . 16
3.5.3 Choosing an Algorithm . . . 16
3.5.4 Increasing the Success Rate of the Algorithm . . . 17
3.6 Conclusion . . . 17

4 Software Design and Implementation **19**
4.1 Design Goals . . . 19
4.1.1 Extensibility . . . 19
4.1.2 Scalability . . . 19
4.1.3 Fault tolerance . . . 19
4.1.4 Maintainability . . . 20
4.1.5 Reuse of Components . . . 20
4.2 Implementation of the Software Program . . . 20
4.2.1 Initializing the Application . . . 21
4.2.2 Translating Classical Music Titles . . . 21
4.2.3 Decisions about Translations using External Databases . . . 23
4.2.4 Connections to External Databases . . . 24
4.2.5 Utility Classes . . . 25

5 Testing **27**
5.1 Unit Tests . . . 27
5.2 Code Coverage Tools . . . 27
5.3 Test-Driven Development . . . 28
5.4 Regression Testing . . . 28
5.5 System Testing . . . 28
5.5.1 A Validation Script . . . 28
5.5.2 A Validator . . . 29
5.6 Performance Testing . . . 30
5.7 Manual Verification . . . 31

6 Process Evaluation **33**
6.1 Scrum . . . 33
6.2 Test-Driven Development . . . 33
6.3 Development Resources . . . 34
6.3.1 Programming Language . . . 34
6.3.2 IDE . . . 34
6.3.3 GitHub. . . . 34
6.3.4 Static Analysis Tools . . . 35
6.3.5 Sprint Tools . . . 35
6.4 Continuous Integration . . . 35
6.5 Slack . . . 36

7 Final Product Evaluation **37**
7.1 Evaluation of User Requirements . . . 37
7.2 Design Goals Evaluation . . . 37
7.2.1 Extensibility . . . 37
7.2.2 Scalability . . . 38
7.2.3 Fault tolerance . . . 38
7.2.4 Maintainability . . . 38
7.2.5 Reuse of Components . . . 38
7.3 Testing . . . 38
7.4 Validation of Test Translations . . . 39

8 Ethical Implications **41**
8.1 Web-Scraping. . . . 41
8.2 Fair Distribution of Knowledge . . . 42

9 Project Wrap-Up **43**
9.1 Code Delivery. . . . 43
9.2 Tool Elaboration . . . 43

10 Conclusion **45**

11 Recommendations **47**
11.1 Future Use . . . 47
11.1.1 Enjoy the Benefits of the Cacher . . . 47
11.1.2 Use the Lookup Tables . . . 47
11.1.3 Integrate the System . . . 47
11.1.4 Every Translation Can Count . . . 48
11.2 Future Development . . . 48
11.2.1 Improvements . . . 48
11.2.2 Extensions . . . 48

A Original Project Description **49**

B Project Plan **51**
B.1 Deadlines . . . 51
B.2 User Requirements . . . 52
B.2.1 Must haves . . . 52

B.2.2 Should haves . . . 52
B.2.3 Could haves . . . 52
B.2.4 Won't haves . . . 52
B.3 Non-functional Requirements . . . 53
B.4 Roadmap . . . 53

C Group Cooperation Contract **55**
C.1 Role Allocation . . . 55
C.2 Decision Making . . . 55
C.3 Presence and Availability . . . 56
C.4 Meetings and Schedules . . . 56
C.5 Task Completion and Responsibility . . . 56
C.5.1 Bug Fixes . . . 56
C.5.2 Document Construction . . . 56
C.5.3 Implementing Features . . . 56
C.5.4 Research and Subsequent Decision Making . . . 57
C.5.5 Sprints . . . 57

D UML Diagrams **59**

E Test Report **63**
E.1 List of Not Fully Covered Methods . . . 63
E.2 List of Classes Containing Coverage Exclusions . . . 63
E.3 Overview . . . 64

F Software Improvement Group Evaluation **65**
F.1 Evaluation of Midterm Submission . . . 65
F.1.1 How This Feedback Was Resolved . . . 65
F.2 Evaluation of Final Submission . . . 66

G Project Evaluations **67**
G.1 Evaluations by the Individual Team Members . . . 67
G.1.1 Mitchell Dingjan . . . 67
G.1.2 Rob Kapel . . . 68
G.1.3 Maarten Sijm . . . 69
G.2 Evaluations by the Client . . . 71
G.2.1 Casper Karreman . . . 71
G.2.2 Ingmar Vroomen . . . 72

H Info Sheet **73**

Glossary **75**

Acronyms **77**

Bibliography **79**

(This page has been intentionally left blank.)

1

Introduction

How are (popular) music pieces catalogued? If you would have to answer that question, you would probably come up with some kind of format including the title, artist, album, and maybe even the instruments. But how about classical music pieces, like Beethoven's Moonlight Sonata? This was one of the questions that we asked ourselves when we started this project. In general, the answer is not as easy as the one to the previous question. Besides, if you are not familiar with classical music pieces, like we were, coming up with an answer to the question is even harder.

At Muziekweb, there are several departments that are working with cataloguing music pieces on a daily basis, based on the answers that they have developed to such cataloguing questions, using standards. The classical music department, for example, manages over 250.000 catalogued classical music pieces by now. Based on the catalogued music pieces, 'readable' formats are generated and presented on their website, where each music piece has its own page. At Muziekweb, 'catalogued classical music pieces' are called 'uniform titles', which is a term that we will apply throughout the rest of this report as well. The term will precisely be defined in Section 2.1.

Aside from that question, we had a far more important question (for us) to answer: how to translate these uniform titles to different languages in an automated fashion? As Muziekweb did not have the resources to come up with an answer, they consulted the TU Delft via our supervisor, after which we accepted the challenge. During the past nine weeks, we have been working to answer exactly this question, on which you will find an answer in this report.

The goal of this project is to implement an algorithm, which can translate Dutch titles of classical music pieces to English, French, and German.

Muziekweb, which is part of the Centrale Discotheek Rotterdam (CDR), is the biggest source of music that has been released in the Netherlands. But with the rise of international streaming services like Spotify, Muziekweb also has the ambition to internationalise their website. Since the titles in their database are currently only available in Dutch, it is difficult to link these to non-Dutch sources.

To internationalise their website, Muziekweb requires translations of their Dutch uniform titles to three other languages: English, French, and German. This will already be a first big step in the right direction. As mentioned in the first paragraph, uniform titles are relatively more complex compared to titles for other types of music. Therefore, later on, a similar and simpler solution could be applied by Muziekweb for translating the sorts of catalogued music pieces other than the classical ones.

Since the Muziekweb classical music database contains over 250.000 titles, it is impossible to translate them all manually. Therefore, an algorithm needs to be designed to do this job automatically. To give us some guidance with this, Muziekweb has given us a table of 1000 classical music titles which have already been translated manually.

This report captures the various stages of this real-world software development project that can roughly be distinguished, including a research and a reflection stage. It also describes various aspects of both the process and the final product.

In order to be more specific, the following enumeration defines the goal of each chapter:

- Chapter 2 formally defines the relevant problem given by Muziekweb.
- Chapter 3 analyses the problem defined in Chapter 2. The largest part of the chapter is taken from the research report, which was a deliverable of week 2.
- Chapter 4 describes the architecture design of the final product and elaborates on the design decisions that have been made. Based on the description, it explains the implemented features of the system and the implementation details of the architecture.
- Chapter 5 explains what kind of tests have been implemented in each development cycle of the product and how they have been implemented, varying from unit test level to system test level. It also covers relevant analysis tools.
- Chapter 6 explains how the development process was organised in general, including development techniques and tools. For each explanation it then evaluates the described development process.
- Chapter 7 evaluates the final product mainly based on the requirements and design goals that were established in cooperation with the client.
- Chapter 8 reasons about two possible ethical implications that the system will have on society; these are related to web-scraping and a fair distribution of knowledge.
- Chapter 9 explains the process of how the project has been wrapped up and delivered to the client.
- Chapter 10 concludes the report by restating the conclusions of the evaluations described for the process and the final product. Based on these evaluations, a conclusion on project level is given.
- Chapter 11 provides recommendations related to starting points for future work.

In order to support these chapters, several appendices have been added. Therefore, throughout this report, we will refer several times to these appendices.

- Appendix A contains the original project description.
- Appendix B contains the project plan, which was created in the first week.
- Appendix C contains the group cooperation contract, which was created in the first week and contains the initial process rules of the project, including a definition of “done”.
- Appendix D includes the full versions of the Unified Modelling Language (UML) diagrams that were made during the project.
- Appendix E includes the test report that documents what parts of the code are not fully covered by unit tests, or excluded from coverage.
- Appendix F covers the first feedback round provided by the Software Improvement Group (SIG) and explains our adjustments that followed.
- Appendix G contains evaluations of the project both by the individual team members and the client.
- Appendix H is the info sheet of the project, which contains a description of the project, including information on its unique points, and a short paragraph on the role of each of the team members.

For technical jargon and acronyms, the Glossary and Acronyms sections can be consulted, respectively. Note that acronyms are only written in full the first time that they are introduced in the report.

2

Problem Definition

The first two weeks of the project roughly covered the stage in which we defined Muziekweb's posed problem and analysed it. During the first week, a project plan was created and reflected upon together with the client. Furthermore, in the first and the second week, a research was conducted and documented in order to analyse the problem further and already come up with possible solutions. The project plan has been added as an appendix (see Appendix B) and the research document has been processed in Chapter 3.

Section 2.1 is devoted to explaining the problem in both a 'formal' and a rather 'informal' way. From this definition, requirements are deduced in Section 2.2. How to meet these requirements and eventually solve the most difficult subproblems that can be distinguished for the solution, is analysed in Chapter 3.

2.1. Problem Definition

As 'translation of uniform titles' might still sound a little vague, we will start defining 'uniform titles' and their properties by providing an 'informal' definition (see Section 2.1.1), which will be followed by a more formal one (see Section 2.1.2). If the reader is familiar with classical music pieces, then 2.1.1 can be skipped over. Finally, the problem is described (see Section 2.1.3).

2.1.1. An 'Informal' Definition

Uniform Titles

A uniform title consists of several fields, according to the standard that Muziekweb applies. In order to give you an idea, an example of the Dutch version of a uniform title is the following one:

Example 2.1. @Wals/2orkest/3op.314 "An der schönen, blauen Donau"

As you can see, Example 2.1 consists of several fields that start with an at-sign or a slash with a number attached, or are indicated by quotation marks. In general, there are also fields that start with a semicolon or a dollar-sign which are preceded and followed by a white space (' ; ' and ' $ ', respectively). Also, the number that is attached to the slash sign is a natural number between 2 and 5. From now on, we will define a field to be the part of text including the indicator tokens (e.g. '/3op.314') and a characteristic to be the part of text without the indicator tokens (e.g. 'op.314').

Each of these fields contains a characteristic of the original classical music piece. Furthermore, the different indicator tokens, which a field contains, identify a class of the characteristics. The slash with a number attached indicates the composition of the orchestra, numbers, keys, and additional information for the numbers 2, 3, 4, and 5 respectively. Besides, the quoted characteristics represent the nickname of a classical music piece, as in Example 2.1. Furthermore, a field starting with a semicolon contains part information about

a classical music piece and a field starting with a dollar sign indicates that the classical music piece is an arrangement. Finally, there is the at-sign which is prepended to the first word of the characteristic, excluding articles. By doing so, uniform titles can be sorted upon the word that is followed by the at-sign. The meaning of this field is, however, a different story, which will be explained in the next subsection.

Two Different Classes of Uniform Titles

There is a distinction to be made in uniform titles, which has to do with how classical music pieces are composed. Classical music pieces are often named based on a combination of their composition type, featured instruments, numbers, and key signatures. The composition type is called the 'generic name' and if a uniform title starts with a generic name (containing an at-sign), then the uniform title is called a 'generic title'. Clearly, Example 2.1 is a generic title, since it starts with a generic name ('Wals' is Dutch for 'Waltz'). Note that multiple generic names can occur in a generic name field, as in Examples 2.2 and 2.3, and that these are enumerated in the way they would be as in the corresponding natural language, using commas and a conjunction at the end.

Example 2.2. @Adagio en allegro/2cello, piano/3op.70/4As gr.t.

Example 2.3. @Toccata, adagio en fuga/2orgel/3BWV.564/4C gr.t.

Additionally, sometimes the composer gives the classical music piece a nickname or it is identified for some (historical or cultural) reason by some group of people by a nickname. In these cases, if the classical music piece is better known by this nickname, on a large scale, than its generic name, then the nickname is called a 'non-generic name'. The uniform title then starts with the non-generic name (containing an at-sign) and is called a non-generic title. When a title is non-generic, the composition of the orchestra field (starting with '/2') should be empty. An example of the Dutch version of a non-generic title is Example 2.4.

Example 2.4. @Tsaar Saltan/3op.57 ; De vlucht van de hommel $ Arr.

Also, note that a non-generic name does not have to be unique; it can occur in different non-generic titles, as in the titles of Example 2.5 and 2.6. Trivially, the same holds for nicknames.

Example 2.5. Le @nozze di Figaro/3KV.492 ; Ouverture

Example 2.6. Le @nozze di Figaro/3KV.492 ; Voi che sapete

However, if the nickname is not commonly accepted as the identifying name of the classical music piece, then the nickname is included in the uniform title using the nickname field (which contains quotation marks), as in Example 2.7.

Example 2.7. @Sonate/2piano/3nr.14, op.27, nr.2/4cis kl.t. "Mondschein"

Although joining these two sets (the generic titles and non-generic titles) will result in the set of all uniform titles, we have not yet covered every case that can occur. There also exist uniform titles that are, according to the definitions of generic and non-generic titles, both generic and non-generic, as they are non-generic titles and have a composition of the orchestra field ('/2'). This is occasionally done when the uniform title has too few fields in order to be 'unique enough'.

2.1.2. A Formal Definition

Classical music titles adhere to a precise standard of the Online Computer Library Center (OCLC). This standard is a big set of rules how to correctly label music titles. Their translations should also adhere to these rules.

The overall syntax of a uniform title is defined in the following way (see Definition 2.8).

Definition 2.8. <uniform title> = @<generic or non-generic name>/2<composition of the orchestra> /3<number>/4<key>/5<extra info>"<nickname>" ; <part info> $ Arr.

The following remarks are related to Definition 2.8.

Remark 2.9. A uniform title can consist of 8 different fields, where each field starts with and ends before some separation token or the start or end of the title.

Remark 2.10. Every uniform title requires at least a field for the generic name or non-generic name; the others are optional.

Remark 2.11. If a uniform title has a generic name, then it must have a field for the composition of the orchestra.

Definition 2.12 defines the semantics of each possible field.

Definition 2.12. The following semantics enumeration of fields is exhaustive:

- @<generic or non-generic name> : the composition type or nickname for which the corresponding classical music piece is known, respectively. The @-symbol is used to indicate which word in the generic name is used to alphabetise on.
- /2<composition of the orchestra> : the set of all instruments used for the relevant piece.
- /3<number> : the rank, bibliographic number, book, part and/or opus number of the piece.
- /4<key> : the key in which the piece is performed.
- /5<extra info> : the information related to the release year or version number, if needed to identify the piece.
- “<nickname>” : the nickname for which a piece is also known.
- ; <part info> : the name of the part for which the relevant piece is known, used when a piece is an element of a larger piece.
- $ Arr : the indication for when a piece is an arrangement.

Definition 2.13 and Definition 2.14 define two different classes of uniform titles.

Definition 2.13. A generic title is a uniform title that has a generic name.

Definition 2.14. A non-generic title is a uniform title that has a non-generic name.

The following remarks are related to Definition 2.13 and Definition 2.14.

Remark 2.15. There exist uniform titles that are both generic and non-generic. Such uniform titles have a non-generic name, but are not unique enough as too little information is known. In order to resolve this, the composition of the orchestra is added for the relevant classical music piece.

Remark 2.16. Because of the previous remark, perhaps trivially, the intersection of the set of generic titles and the set of non-generic titles is non-empty.

2.1.3. The Translation Problem

Now that uniform titles are clearly defined, we can clearly define the problem.

Basically, the entire uniform title has to be translated, based on the translations of its characteristics, to English, French, and German in an automated fashion. These translations should meet the rules and ‘expected’ names that are common for each relevant language. The latter implies that nicknames and non-generic names (and occasionally part information) cannot be directly translated from language to language, as in different languages these names can at least partially be different, because of historical and cultural reasons. An example of such a translation of a classical music piece from Dutch to English is Example 2.17, which is related to Example 2.1. Example 2.18 shows that names can also remain the same in a different language. The translations of such names therefore definitely require somehow some expertise in the field of classical music.

Example 2.17. An der schönen, blauen Donau $\overset{\text{English}}{\Longrightarrow}$ The blue Danube

Example 2.18. Le sacre du printemps $\overset{\text{German}}{\Longrightarrow}$ Le sacre du printemps

Trivially, the input file consists of uniform titles. But we have to be a little more specific here, as the input file can also contain the composer and additional general keywords for each related uniform title that have been assigned in Muziekweb's database. These keywords define general categories that the title belongs to, and are sometimes equal to the generic name of the title. This data might turn out to be useful for solving the problem.

As mentioned in Chapter 1, a table of 1000 manually translated uniform titles has been provided by Muziekweb at the start of the project, which can be regarded as a training set of the system. The table is provided as an Excel file. Each row contains 6 items: a unique identifier, the uniform Dutch title, a readable Dutch title, the uniform English title, the uniform French title, and the uniform German title.

2.2. Requirements

The requirements in the section have been defined in week 1 of the project. These requirements are part of the Project Plan that was a deliverable for that week, which has been added as Appendix B.

2.2.1. User Requirements

The requirements regarding functionality and service are grouped under the Functional Requirements. Within these functional requirements, four categories can be identified using the MoSCoW model[1]for prioritising requirements: Must haves, Should haves, Could haves, and Won't haves.

Two groups of users can be identified, based on their departments at Muziekweb: the automation and the classical music department. The automation department will implement the final product into the current cataloguing environment of Muziekweb and the classical music department will use the system via this integration. Both user groups are taken into account in the following groups of requirements.

Must haves

- The program must be able to translate uniform titles for classical music pieces.
- The translated form of these uniform titles must maintain the structure of the title.
- The program's input and output must be in the form of a Tab-Separated Values (TSV) file.
  - The input file must have at least three columns: Link (identification number), Uniform title, Readable title. Optionally, the input file can also have the following columns: Composer, Keywords.
  - The output file must maintain these columns and have additional columns for the translated uniform titles.
- The program must translate Dutch parts of the uniform titles to the following target languages:
  - English
  - French
  - German
- The program must translate the titles while satisfying the PICA constraints.
- "Nicknames" (i.e. names in quotation marks) that are not in Dutch, must be left in their original language.

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

  - This means that the program must recognise which language a piece of text is written in. This is done by consulting external music databases.

Should haves

- The TSV file should be tab-separated to avoid extra string escaping (since commas, semicolons and quotation marks are also used in the structure of the uniform titles).
- The program should be able to distinguish Dutch parts from Latin parts of titles.
- Latin parts should remain in Latin in any (modern) language.
- The program should have an interactive mode in which ambiguous titles can be translated directly by the user.

Could haves

- The program could be able to read the input via the standard input and write the output to the standard output.
- The system could use a learning algorithm, which can improve previously given translations.

Won't haves

- The program won't be able to translate titles of popular music pieces.
- The program won't be able to translate titles to languages other than the specified target languages.

2.2.2. Non-functional Requirements

Besides the provided functionality and services, design constraints need to be included in the requirements specification as well. These requirements do not indicate what the system should do, but instead indicate the constraints that apply to the system or the development process of the system.

- The program shall be able to run on Windows (7 or higher) and Linux.
- The program shall be implemented in C#.
- For every iteration the Scrum methodology will be applied.
- The implementation of the program shall have at least 83.5% of meaningful line test coverage (where meaningful means that the tests actually test the functionalities of the program and, for example, do not just execute the methods involved).
- The results of the translation will also be verified manually, since this cannot be automated.
- The implementation of the program shall have no warnings from static analysis tools. Any warnings that are suppressed, should be well-documented with a proper reason at the place where the warning is suppressed.
- All methods must be documented using XML documentation comments.
- The development team will use a number of tools:
  - Microsoft Visual Studio or JetBrains Rider
  - Git(Hub)
  - Travis CI
  - Waffle
  - Slack
  - Static analysis tool: StyleCop
  - Code coverage tools: Coveralls
- Tabs in Integrated Development Environments (IDEs) are done with 4 spaces (' ' or ASCII 0x20).
- Line endings are in Unix format, with a single line-feed character ('\n' or ASCII 0x0A).

(This page has been intentionally left blank.)

3

Problem Analysis

This chapter was created as a deliverable during week 1 and 2 of the project. It analyses how to meet the requirements stated in Chapter 2 and eventually recommends how to solve the most difficult subproblems that can be distinguished for the solution.

3.1. Splitting the Problem

In order to solve the given problem, defined in Chapter 2, we will classify several subproblems and try to find a solution for these. By doing so, we will be able to tackle the main problem in a structured manner. Also, we will research the cataloguing standards that are used both by Muziekweb and on an international scale, in order to better understand the concept of music cataloguing. In the following enumeration, these subproblems and research subject are represented.

- How do we translate the components of classical music titles without semantics?

  A basic subproblem in the translation of classical music titles are the translations of components without semantics. These fields do not have a (significant) semantic meaning attached, but do require to be translated correctly using a well-defined plan.

- How do we translate nicknames?

  As seen from the structure, classical titles can have a nickname (e.g. Vivaldi's opus no.8 is better known as "The Four Seasons"). However, a nickname can be different for every language. Nicknames are usually not agreed on by everyone, since they also have a semantic meaning attached to them. Furthermore, no two different pieces can have the same nickname. Translating these nicknames is therefore something that needs to be researched.

- What cataloguing standards are used on an international scale?

  Another problem arises when the titles need to be linked to other databases. As mentioned, Muziekweb uses the PICA standard. However, this is not the only standard out there. For example, other standards are the RDA standard and the MARC21 standard.

- How can we involve human experts to the system in an efficient way for ambiguous or unavailable translations of nicknames?

  Sometimes it might not be directly possible to find the correct title from these databases. For that reason, a human expert needs to be consulted. Since the experts know best, any future translations that are given should use this information. With machine learning, the translation process will be as automated as possible.

3.2. Translations of Components Without Semantics

Many fields in the title do not rely on semantics for the translation. These fields include the name (if it is based on a generic name), composition of the orchestra, number, and key. For each item, we will investigate how to translate them.

3.2.1. Generic Names

For the generic name, there is a set list of generic names that can be used with the standard of OCLC [28]. We start by looking at the translations already provided to us. Since these translations have already been verified, they will be the most reliable source. Some translations of generic names can therefore already be extracted from this list.

Some of these are very common (e.g. symphony), and more uncommon ones (e.g. choral fantasy). For the common names, Wikipedia has most of the translations already covered. Besides the common names, there are also Italian names. Many of these names are Italian, because many important composers in the Renaissance were Italian. Therefore, many of these terms are still used among all languages. Therefore, we can check whether a Dutch name is an Italian term [46]. If so, the same name can also be used for other languages.

For the uncommon names, for which a translation is not available on Wikipedia, it is not as easy. Not a single source has all the information on generic names in other languages. Therefore, we utilise multiple resources, such as:

- Doing a simple translation using Google Translate, and then verifying on language-specific websites for music terms, such as
  - `https://dictionary.onmusic.org/` for English.
  - `http://www.musiklehre.at/fachwortlexikon/` for German.
  - `http://www.larousse.fr/archives/musique` for French.
- Asking a human expert, whether there is another, more common name for the generic name, for which a translation can be found.

3.2.2. Composition of the Orchestra

For the composition of the orchestra, there is also a set list of instruments that can be used with the standard of the OCLC [28]. For translating the instruments, many translations have already been made by others [16, 49]. However, not all instruments listed in the standard can be found here. Therefore, Wikipedia will be used as another source for the translations.

If neither of these gives a translation, we have to use a translator as Google Translate, to translate the instruments. Since this is not guaranteed to give the correct translation, it needs to be verified as well on language-specific websites, as mentioned above.

3.2.3. Numbers

The numbers are relatively easy in comparison to the other fields. This field can only contain a rank, bibliographic number and/or opus number. From the translated titles given to us, we have determined that a rank number is preceded by “nr.” and opus number is preceded by “op.”. The translations for “nr.” to English, French, and German are “no.”, “no.”, and “Nr.” respectively. The translation for “op.” stay the same for English and French, but in German, it is translated to “Op.”. In Table 3.1, the translations of the number field are shown in the first two rows.

3.2.4. Keys

Finally, we have the key in which the piece is performed. These will be based on the translations given to us, and other sources found online [48, 50]. A short summary of these sources is given in the next few paragraphs.

In Dutch and English, the chromatic scale is used, with pitches ranging from “c” to “b”. In German, they use a slightly different scale. They have the same pitch notations for “c” to “a”. For “b flat” and “b” however, “b” and “h” are used respectively. In French, the same scale

is used. However, instead of using letters, they use the syllables from solmisation (e.g. “do”, “ré”, “mi”).

To indicate semitones, German and Dutch use the same system. For a “flat”, the letters “es” are added after the pitch. However, if the pitch is a vowel, only an “s” is added. As mentioned before, in German there is a slight variation in which letters are used. Therefore, these cannot be directly translated from Dutch to German. For a “sharp”, the letters “is” are added after the pitch, even if it is a vowel. In French and English, a word is added after the pitch. In French, they use the words “dièse” and “bémol” for “sharp” and “flat” respectively.

Finally, a pitch is also either in “major” or “minor”. These translations are consistent over all different pitches. For Dutch, German, and French, the translations of major are “gr.t.”, “-Dur” and “majeur” respectively. In German and Dutch, the first letter of the pitch is also capitalised to indicate this. And the translations for minor are “kl.t.”, “-Moll”, and “mineur”.

In Table 3.1, the translations for the pitch are shown in the middle part, followed by two rows of translations for major and minor. In the final 2 rows, an example is given to show the change to upper case for pitches in “major”.

Dutch	English	French	German
nr.	no.	no.	Nr.
op.	op.	op.	Op.
c	c	do	c
cis	c sharp	do dièse	cis
des	d flat	ré bémol	des
d	d	ré	d
dis	d sharp	ré dièse	dis
es	e flat	mi bémol	es
e	e	mi	e
f	f	fa	f
fis	f sharp	fa dièse	fis
ges	g flat	sol bémol	ges
g	g	sol	g
gis	g sharp	sol dièse	gis
as	a flat	la bémol	as
a	a	la	a
ais	a sharp	la dièse	b
bes	b flat	si bémol	b
b	b	si	h
kl.t.	minor	mineur	-Moll
gr.t.	major	majeur	-Dur
a kl. t.	a minor	la mineur	a-Moll
A gr. t.	A major	la majeur	A-Dur

Table 3.1: Translations of the number and key fields.

3.3. Translation of Nicknames

As mentioned in the problem statement, translating the nicknames of classical music is not trivial. In this section, we will investigate whether it is better to use direct translation or to translate using external databases.

3.3.1. Direct Translation

The nicknames can be seen as text strings containing natural language and can thus be translated using common “direct translation” approaches. We will look into four of these approaches: interlingua representation, transfer method, statistical machine translation, and neural networks. Direct word-by-word translation will not be covered, since natural language consists of too much complexity and ambiguity [6] for this method to work.

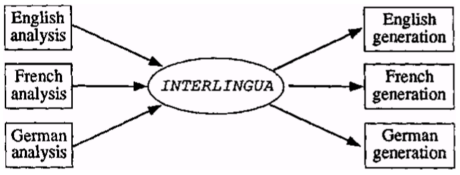

Figure 3.1: Schematic overview of machine translation using interlingua representation [17].

Interlingua Method

The interlingua method [17, Ch.4] uses a universal intermediate representation for all languages. Source languages are first analysed and converted into this interlingua representation before being converted into the target languages. This is shown in Figure 3.1. Since natural languages are diverse, this interlingua representation can be very complex and difficult to maintain.

Transfer Method

The transfer method [17, Ch.4] is similar to the interlingua method. However, the transfer method does not use one intermediate representation, but one per pair of languages in the system. This is shown in Figure 3.2. The obvious downside of this is that there has to be a quadratic amount of language transfer modules. However, the system that has to be built for this project only needs translation from Dutch to three other languages, reducing the number of modules needed from twenty to seven. Still, the amount of work needed to create these modules would result in having to translate all nicknames by hand.

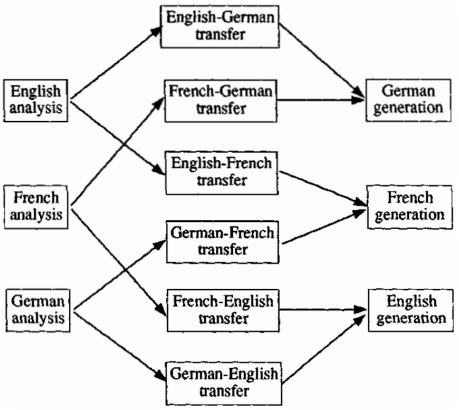

Figure 3.2: Schematic overview of machine translation using the transfer method [17].

Statistical Machine Translation

Statistical Machine Translation (SMT) [4] is an approach that is completely different from the previous ones. This method uses already existing translations of sentences from the source language to the target language to generate word translation probabilities. These can then be used to translate new sentences. An advantage of SMT is that manually constructing a language model is not needed, since this is generated based on the training set. One of the disadvantages is the amount of memory needed to store the result of training, since parameters have to be calculated for every word pair between the source and the target language. This is again a quadratic amount, but this language model does not have to be made by hand, like with the transfer method.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) [7] are a more recent improvement on the slightly older SMT approach. RNNs often consist of an encoder and a decoder. The encoder extracts a fixed-length representation from a variable-length input sentence, and the decoder generates a correct translation from this representation. These models are often smaller than one gigabyte, while most SMT models often require tens of gigabytes of memory [7]. However, RNNs mostly work best on sentences that are about 10-20 words in length [7], which is longer than the average nickname for a classical music piece.

Summary

We have seen four methods to translate natural language strings. Interlingua and transfer method require manual generation of language models, which would be about as much work as translating all nicknames manually. SMT and RNN use training sets containing full sentences, while nicknames consist mostly only of a few nouns. This provides less context for translation which makes these approaches unlikely to give accurate results.

Another reason why current natural language translation approaches cannot be used for nicknames, is the significant difference in how classical music pieces are commonly known in different regions. For example, the nickname “An der schönen blauen Donau” is commonly known in English as “The Blue Danube” [47], while a translation is more likely to give a result like “By the Beautiful Blue Danube”. The following section will look into methods that make use of already existing translations for nicknames.

3.3.2. Translation Using External Databases

Most classical music pieces that have a nickname, already have a translation that is not equal to a translation that would be given by a direct translation. There are databases containing these translations available on the Internet. This section will investigate some of these databases. First, we will look into common sources of knowledge: Wikipedia, and its database-structured equivalent: DBpedia. Then, we will examine some open-data online music databases: Discogs, AllMusic, and MusicBrainz.

Wikipedia

Let us start examining a source that everybody knows: Wikipedia. Most classical music pieces are listed on Wikipedia in multiple languages. Take for example again "The Blue Danube" [47]. On the left side of the web page, there are links to this article in different language. By going to the French version, we immediately find "Le Beau Danube bleu" to be the nickname of this piece in French. An advantage of using Wikipedia as a source for translation is that it is kept up-to-date by millions of users. A downside is that extra text analysis has to be done in order to extract the translated nicknames from the Wikipedia pages.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?piece ?len ?lde ?lfr ?lnl WHERE {
    ?piece rdfs:label ?len.
    ?piece rdfs:label ?lde.
    ?piece rdfs:label ?lfr.
    ?piece rdfs:label ?lnl.
    FILTER (?len="The Blue Danube"@en &&
        LANG(?lde)='de' && LANG(?lfr)='fr' && LANG(?lnl)='nl')
}
```

Listing 3.1: A SPARQL query that will fetch the nicknames in English, German, French, and Dutch for The Blue Danube on DBpedia.

DBpedia

An online database that has more structure and is thus easier to parse for computers, is DBpedia. When looking on the English page for The Blue Danube [11], we see under the entry `rdfs:label` its common title in different languages. It is easy to write a query to get the common title in our four target languages. An example of this is shown in Listing 3.1. Running this query directly gives us that the French nickname for this piece is "Le Beau Danube bleu", without having to do a text analysis.

Discogs

The first music database we will look into is Discogs. Whereas the previous two databases were general-purpose databases, Discogs is designed to store meta-data of music releases. Almost half a million classical music releases can be found in this database [13]. However, Discogs only maintains English data found about individual releases on vinyl, CD, or other hardware media (called discographies). Discogs does not maintain general information per original classical music piece, so we will not be able to fetch nicknames in different languages from Discogs.

AllMusic

Another online source of information about music releases is AllMusic. In contrast to Discogs, AllMusic does have information pages per classical music piece. However, it does not have a multilingual database. Again looking at The Blue Danube, we find that the original title of this piece is "An der Schöne, Blaue Donau" [1], but it does not even state that this title is in German. Just like Discogs, we cannot use this database for the translation of nicknames.

MusicBrainz

MusicBrainz is a more promising music database. It is an open-data collection of music titles, releases and other meta-data, including aliases [24]. This list of aliases also contains translations in other languages. However, this list of aliases is more often incomplete than DBpedia which we can already see for The Blue Danube: the list only contains translations to German and English. But even though MusicBrainz has a smaller database, we can still use it to get translations of nicknames.

Summary

In conclusion, we decide to use DBpedia and MusicBrainz as external databases to retrieve translations of nicknames of classical music pieces. These databases have one common advantage. These databases contain translations for nicknames that are easily found when searching for a classical music piece. This makes these databases easily queriable for computer programs.

3.4. International Music Cataloguing and Current Approaches

Section 3.1 already highlighted several major cataloguing standards that are used on an international scale. In this section, we will analyse these approaches in-depth. By doing so, we will gain an understanding of similarities and any struggles in cataloguing. Also, we will analyse any effects that are introduced by international music cataloguing. But before going into the details, we will first provide an introduction to how music (meta)data distribution is internationally organised today.

3.4.1. A Global Network of Music Metadata

The distribution of music (meta)data is informally organised using a global network, consisting of smaller networks. An important sub-graph in this graph is OCLC's WorldCat. WorldCat describes itself as "the world's largest network of library content and services" [30]. Even as we are searching for related literature via the TU Delft Library, we are using WorldCat's services. Their libraries provide (free) online access to their resources and grow daily as anyone can add information. Additionally, there are other important sub-graphs out there; we have already seen Discogs, AllMusic, MusicBrainz, DBpedia, and Wikipedia in Section 3.3.2. Other open-access networks are, for example, FreeDB, Rate Your Music, and Open Library. Because most of these instances also use each other's data, these networks are often intertwined.

When analysing the relatively smaller networks, chances are that we will detect Muziekweb, which is also an open-access organisation. The collection of Muziekweb consists of more than 500.000 CDs, 300.000 LPs and 30.000 music DVDs [25]. Especially for the Netherlands, Muziekweb is an important music data network. However, as we have seen in Chapter 1, Muziekweb wants to internationalise more and therefore grow new or relatively stronger connections with other networks in the graph.

From an end user perspective, the existence of an international network is important. One could reason that end users are likely to consult their national network first (e.g. for ease), in order to satisfy their need for music data. However, if that will not help the users much further, chances are that they will consult an international network.

3.4.2. International Standards

As we have seen in Section 3.4.1, there are a lot of instances around the world that have a database containing music metadata. In order to facilitate co-operations within this network, the introduction of international music cataloguing standards is useful. As they have been introduced, we will now analyse the most important ones.

Resource Description and Access

The standard 'Resource Description and Access' (RDA) has evolved from another standard, the Anglo-American Cataloguing Rules, 2nd Edition Revised (AACR2) [18]. It was initially created over the course of 2005-2009, to replace AACR2, which was published in 1978. The RDA Steering Committee (RSC) describes their services as: "a package of data elements, guidelines, and instructions for creating library and cultural heritage resource meta-data that are well-formed according to international models for user-focused linked data applications" [33]. It therefore tries to meet the needs of international, cultural heritage, and linked data communities. RDA was first adopted by the Library of Congress in 2013 [20], after which e.g. Library and Archives Canada, British Library, Deutsche Nationalbibliothek, and National Library of Australia followed. Along with the package, an 'RDA Toolkit' can be purchased, which "is an integrated, browser-based, online product that allows users to

interact with a collection of cataloguing-related documents and resources" [34]. In April-May 2017, RSC announced that the toolkit was translated from English into Catalan, Chinese, Finnish, French, German, Italian, and Spanish [36, 37]. They are still working on translating the toolkit into other languages, like Dutch, Arabic, Greek, Hebrew, and Swedish [35]. Therefore, based on developments on implementing RDA in countries all over the world [42] and recent developments, RDA is really making its way to a global standard.

Machine-Readable Cataloguing in the 21st Century

The standard Machine-Readable Cataloguing 21 (MARC21) was initially designed in 1999 for the 21st century to make the original MARC record format more accessible on an international scale, which worked out [3]. As an indication, on January 1, 2017, 2.2 million out of 386.2 million records were not encoded using the MARC format [29]. It is the result of combining the American and Canadian MARC formats. The MARC21 records are encoded using the MARC-8 encoding, which is the total collection of characters that are encoded by its component sets [19]. It also allows the use of Unicode. According to the RDA Music Implementation Task Force, 'best practises' for music cataloguing nowadays start by using both RDA and MARC21 [23]. The standards can be intertwined, using, when advised, a mapping from RDA to MARC or the MARC21 encoding.

Bibliographic Framework

In 2012 the new standard Bibliographic Framework (BIBFRAME) was announced, by the Library of Congress, in order to replace the MARC standards (thus including MARC21). Furthermore, the goal was to: "(1) enable far more integration of existing bibliographic resources and (2) create a roadmap for moving forward toward refinement, redevelopment or development of alternative approaches" [22]. The latest version of BIBFRAME, version 2.0, was published in 2016 by the Library of Congress [21].

BIBFRAME is different in the sense that it is using RDA for its model and is able to translate MARC21 to a Linked Data model [22]. Although the idea might sound good and progress is made, there are still some issues that require to be fixed [26].

3.4.3. The Effects of International Music Cataloguing

Along with the international standards of music cataloguing, comes the desire to translate or transliterate parts of the metadata to different languages. Some people are sceptic about this, and, looking at current developments, strongly advice to take absolute care when mapping music meta-data from one language to the other [15]. They argue that if there is not enough precision in the mapping or the right use of technical terms, a part of the semantic value of the data gets lost.

3.4.4. Muziekweb and Music Cataloguing

Moving back from the bigger picture to Muziekweb, we notice that Muziekweb is using cataloguing rules that are based on a former standard introduced by the Online Computer Library Center (OCLC). Back then, they were cataloguing rules defined by PICA, of which all shares by 2007 had been sold to OCLC. There are currently discussions going on at Muziekweb on whether to change to the RDA standard. But doing so now will cost a lot of time, since their entire database is based on a different standard. During our project, we will therefore at least take into account the standard of OCLC. Now that we have researched the international standards, we have improved our understanding of what is out there on an international scale.

3.5. Machine Learning

As mentioned in Section 3.3.2, there are multiple databases that can be used to aid in the translation process. By relying on just one database, there will be titles that cannot be translated, due to insufficient data. Therefore, using multiple databases seems wise. This leads to a new problem. If both databases have a different translation for the same title, which should be picked? To solve this problem, we will have a look at machine learning methods.

3.5.1. What Is Machine Learning?

Machine learning is a method of data analysis that automates analytic model building [41]. In other words, in the process of machine learning, the computer is given a data set. From the items in the data set, features are extracted for each item. With the predefined labels, a model can be described to determine how to label items based on this features. Then, when a new, unlabelled item is given to the computer, it predicts what label to give based on the model.

There are 3 main types of machine learning [32]. First up is unsupervised learning. With unsupervised learning, no specific predictions are made. Unsupervised learning algorithms are used to group items as efficiently as possible. Besides that, we have reinforcement learning. Reinforcement learning is an algorithm that learns by trial and error. The final one is supervised learning. The gist of supervised learning is predicting a dependant variable, using independent variables. In other words, by giving a certain input to the system, the system should produce an outcome.

3.5.2. Multiple Algorithms

For our categorisation problem, a machine learning algorithm from supervised learning would be most suitable. From the features, we can collect for each music piece (e.g. composer, artist, and release date) we can determine which database is most reliable for the piece.

There are multiple algorithms that could work for our problem. We take a look at a few promising algorithms.

Support Vector Machine Model

The first one to look at is the Support Vector Machine (SVM) model. The SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible [9]. The further away a new data entry is located from this gap, the more likely it is that the algorithm is correct.

There are multiple advantages to the SVM model [2]. One of them is that it is flexible. This is useful for our problem, since the databases do not contain information on all the same fields. Furthermore, the model puts more stress on similarity. When pieces are very similar, it is likely that the nicknames have been given by the same person. Therefore, if they have given correct nicknames before, they are more likely to give more correct nicknames.

Decision Tree

Another promising algorithm is the Decision Tree. A Decision Tree is a series of if-then statements to conclude which label to add [39]. This makes up a tree structure, with at each node a new statement. The if-then statements can also be used for continuous features, by applying thresholds. These thresholds are determined by the algorithm.

The Decision Tree has advantages as well [12]. One of the main advantages is that a Decision Tree is very easy to explain. Since the tree structure can be retrieved, every decision made by the algorithm can be retraced manually. Additionally, a Decision Tree does not require a lot of pre-processing of the data.

3.5.3. Choosing an Algorithm

To make a decision between these algorithms, we need some way to test them. The best way is by measuring the recall and precision of these systems on test data. The recall of the system is how many of the positive cases were caught. For example, if we have a recall of 80%, it means that 8 out of 10 correct nicknames in database A were labelled as correct by the algorithm. Precision is how many of our positive predictions were correct. In other words, if we have a precision of 80%, it means that out of 10 predictions for which database A is said to have the correct, only 8 were actually correct nicknames in that database. This can be done for each database.

To properly test which algorithm is the best, our translations that have been given to us, need to be separated into training data and testing data. The training data is used to define

the thresholds for the algorithm. Afterwards, we use the testing data to determine precision and recall.

3.5.4. Increasing the Success Rate of the Algorithm

It is important to have the correct nicknames for other languages; if they are not the same, it is difficult to find the right reference on other databases. Therefore, we will look at active learning. Active learning algorithms are able to query users on items which the algorithm is unsure about [40]. The principle behind active learning is that an expert knows the best how to label items. Every time the algorithm has unlabelled objects left which it is unsure about, the expert is queried to aid in labelling. This is then incorporated in the algorithm, so that it can retry to label the unlabelled items. This is repeated until everything is labelled.

The advantage of using this method is acquiring a more precise result instead of not using it; by requiring information from an expert, a correct title can be guaranteed. However, the process is significantly slower than regular machine learning. As an expert, you need way more time to label an item, than a computer does. Therefore, since this is in an automation problem, it is important not to query the expert too often. Only the cases which the algorithm is most unsure about should be queried. Therefore, a certainty percentage can be calculated during the labelling. This is then used to determine the most uncertain translations.

3.6. Conclusion

In order to conclude our research, we will briefly restate the major subproblems classified in Section 3.1 and for each of them restate the best solutions that we have found in the rest of Chapter 3.

- How do we translate the components of classical music titles without semantics?

  Based on Section 3.2, it depends; we cannot apply the same translation method for all of these components.

  - For generic names, we distinguish between common and uncommon names, where common names are translated using already provided translated data or data from Wikipedia and uncommon names are translated using a combination of multiple sources (a predictor and a verifier) or by consulting a human expert.
  - For composition of the orchestra, we apply a similar approach as for the generic names.
  - For numbers, we will apply the rules for ‘nr.’ and ‘op.’ that are common for each language.
  - For keys, we will apply the specified rules that are common for each language.

- How do we translate nicknames?

  Based on Section 3.3, we will use DBpedia and MusicBrainz as external databases to retrieve translations of nicknames for the classical music pieces.

- What cataloguing standards are used on an international scale?

  Based on Section 3.4, we have seen that the most dominant international standards are RDA and MARC21. Also, as developments on BIBFRAME are still in progress, this could become an international standard as well.

- How can we involve human experts to the system in an efficient way for ambiguous or unavailable translations of nicknames?

  Based on Section 3.5, we will use a machine learning algorithm from supervised learning; we will either choose for the Support Vector Machine Model or the Decision Tree. The choice will be based on several tests that evaluate their precision and recall outcomes. Trivially, we will choose the algorithm with the highest precision and recall percentages.

By merging the solutions that were found based on this division, we will be able to conquer the main problem.

(This page has been intentionally left blank.)

4

Software Design and Implementation

In this chapter, the reasoning behind the design and implementation of the software product will be discussed, based on the results of the analysis of the problem in Chapter 3. In Section 4.1, the most important design goals are listed. In Section 4.2, an outline of the architecture is provided by dividing the software program into subsystems. The implementation will be described in detail and is supported by simplified versions of the UML diagrams that were made during the project. The full UML diagrams are given in Appendix D.

4.1. Design Goals

The design of the software program has been done with some design goals in mind. The following subsections will indicate which ones and give examples of how these design goals can work out in the implementation. In Section 4.2, occasionally a reference is made back to one of these design goals to show that they have been used in the design of the software program.

4.1.1. Extensibility

The software program has to be designed in such a way that it can easily be extended. For example, the target languages for translating titles are currently English, French, and German. However, for future use it might be interesting to add additional languages to the system. Another example is the use of external databases. If, in the future, a decision is made that new external databases should be used, it should be easy to implement this addition. The design of the system should allow for such extensions.

4.1.2. Scalability

Since the system should be able to translate over 250.000 uniform titles, it should be capable of processing such a large amount of data. Therefore, during the design of the system, we should take this into account. So, for instance, when using external databases in order to facilitate the translations, these resources should be able of processing the relatively large amount of data. Also, the software program should not unnecessarily claim a large amount of RAM for each title that is translated.

4.1.3. Fault tolerance

During the translation of the entire database, the program should not crash to avoid loss of data. Since all uniform titles have been entered in Muziekweb's database manually, they can contain mistakes. These mistakes should be caught so that the software program will not crash.

4.1.4. Maintainability

When the translation of 250.000 uniform titles has been done, the software program will stay in use at Muziekweb, since new classical music pieces are being catalogued every day. These titles also have to be translated. If anything would change in the translation process of uniform titles, the software program should be easy to maintain for the employees at Muziekweb.

4.1.5. Reuse of Components

The architecture of the software program should allow reuse of components. If not, there might be duplicate pieces of code, which is difficult to maintain. Improving one occurrence of the duplicate code would leave the bugs in the other occurrences. Therefore, components should be reused as much as possible.

4.2. Implementation of the Software Program

The software program has been implemented in C#. In C#, so-called 'projects' are added to a 'solution', where the project contains the code and the solution is a container of projects. For this software program, a project called 'Translate' was created in a new solution.

The software product needs to be divided into several subsystems. There should be a part that handles any input/output files or other options that are given to the program via the command line, which is discussed in Section 4.2.1. Another part will be dedicated to the actual translation of classical music titles, shown in Section 4.2.2. To decide which translation coming from external databases should be used, another subsystem is needed, which is shown in Section 4.2.3. Another subsystem makes the actual connection to external databases to translate nicknames and non-generic names, discussed in Section 4.2.4. Any component that does not fit inside any of these subsystems are listed in Section 4.2.5, these components can be used in any subsystem. An overview of the connections between the subsystems is shown in Figure 4.1.

Each subsystem is accompanied by a simplified version of the UML diagram for that subsystem. In the UML diagrams, abstract classes or methods have their name in *italics*.
Symbols are used to denote access levels on fields and methods: `+` for `public`, `~` for `internal`, `v` for `protected`, and `–` for `private`.
Not all types are fully written out to save space. The following list of abbreviations is used:

- `str` - stands for `string`.
- `List` - can be of type `IEnumerable`, `IReadOnlyList` or `IList`.
- `Dict` - stands for `Dictionary`.
- `Lang` - stands for `Language`, an enumeration in the `Utilities` subsystem.

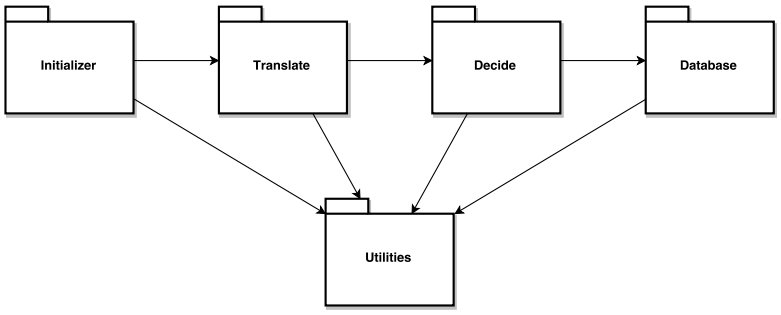

Figure 4.1: Connections between the subsystems in the software product.

4.2.1. Initializing the Application

Since the software program is a Command-Line Application (CLA), the user is allowed to specify arguments to the program that indicate how the user wishes to use the program. The first subsystem, called `Initializer`, will read these arguments and kick-start the rest of the software program. This subsystem consists of three classes: `EntryPoint`, `Program` and `FileHandler`. A simplified version of the UML diagram for this subsystem can be found in Figure 4.2.

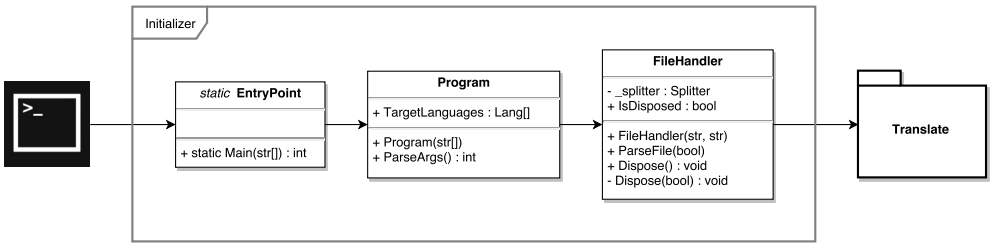

Figure 4.2: Simplified version of the UML diagram for the `Initializer` subsystem.

The `EntryPoint` class contains the Main method. The only thing that it does is passing the arguments to a new instance of the `Program` class. This makes sure that the static context is left behind as soon as possible.

The `Program` class reads the options given on the command line. It validates these options and prints the list of available options if one of the given options is invalid or when the user asks for help. There are four options, which can be passed to the `FileHandler`:

- `--help`, which shows the list of options and exits the software program immediately.
- `--input=FILE`, used to specify the location of the input file.
- `--output=FILE`, used to specify the location of the output file.
- `--timestamps`, which adds a timestamp for each translated title to the output.

The `FileHandler` class makes sure that the input of the program is read, line by line. As mentioned in Section 2.1.3, each line of the input file will contain a uniform title and other information belonging to that title. This information is passed to the `Splitter` in the `Translate` subsystem. When the `Splitter` returns the translations, the `FileHandler` appends these to the input line and writes this to the desired output location. When no input file is specified, it will read from the standard input. When no output file is specified, it will read from the standard output. When all titles have been processed, the software program exits.

4.2.2. Translating Classical Music Titles

The subsystem that parses classical music titles and translates the several parts to the target languages, is called `Translate`. A simplified version of the UML diagram for this subsystem can be found in Figure 4.3.

The main component, the `Splitter`, accepts uniform titles as input and pre-processes them. It splits the titles into parts according to the syntax of a title as described in Section 2.1. Each part is then passed on to the designated translation components. The results of the translated parts are then joined together again and returned as result. Resources that can be used to translate multiple titles are shared between translations, taking the design goal scalability (Section 4.1.2) into account. The `Splitter` is also open for extension (design goal in Section 4.1.1); new translators can be hooked onto the `Splitter` without much effort.

The translation of separate parts of the uniform titles is delegated to separate translator classes that all implement an interface called `ITranslate`. This interface defines a method that takes a Dutch term and any characteristics that belong to the uniform title and returns the corresponding English, French, and German term in a `Dictionary`. Because all translators use the same interface, each uniform title can be processed by looping over all

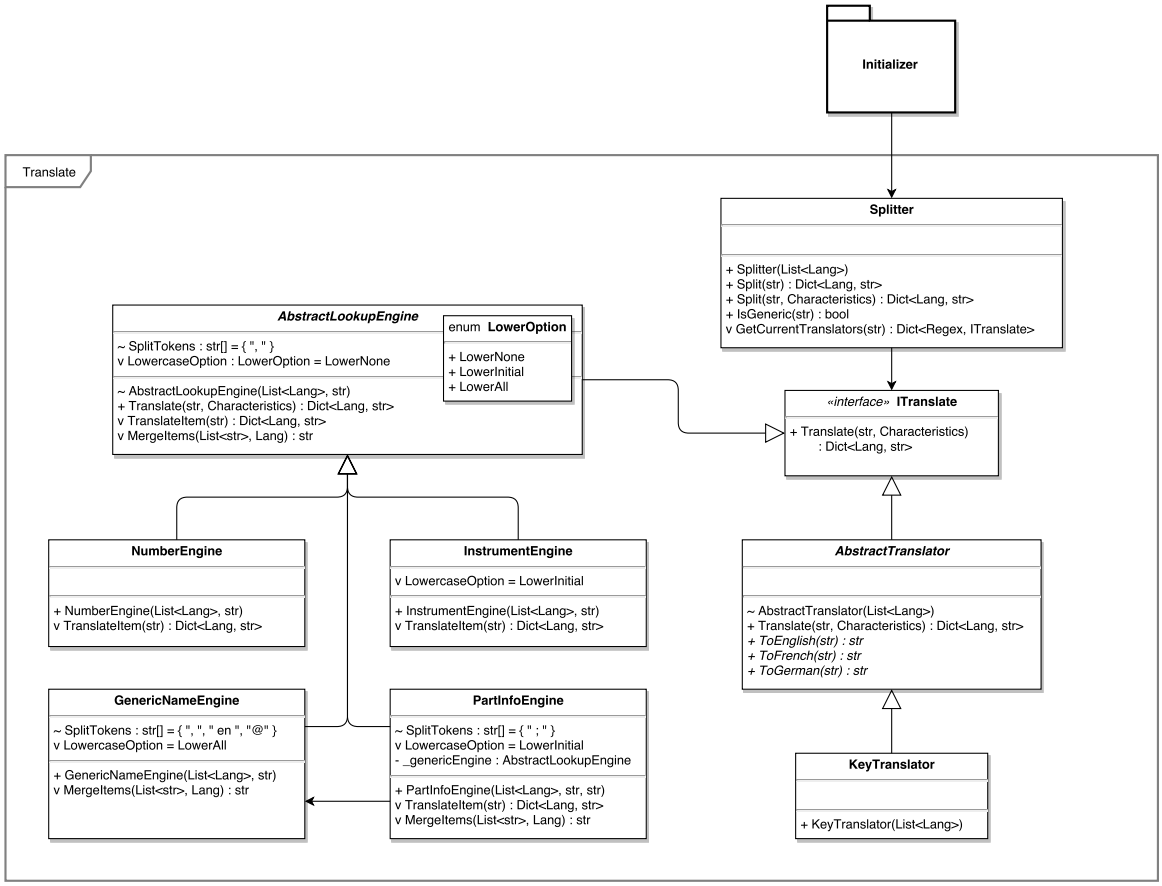

Figure 4.3: Simplified version of the UML diagram for the `Translate` subsystem.

translators. If any of the translators throws an exception, the Splitter will set the output of that specific translator equal to the input for all target languages, meaning that the rest of the title can still be translated. The exception that was thrown by the translator is printed in the console. This construction contributes to the fault tolerance of the system (see Section 4.1.3).

The `Translate` subsystem also contains the simpler translators that implement the interface `ITranslate` using pre-definable structures to generate the translations. The translators that rely on external databases can be found in the `Decide` subsystem, see section 4.2.3.

The `AbstractLookupEngine` defines functionality for reading in a lookup table in TSV format to a `Dictionary` in memory and using it to look up items in this table. This functionality can be used for translating generic names, instruments, and numbers because these fields consist of lists of predefined items. Since these items are all in separate files and not hard-coded, they can easily be updated in the future, if needed. This contributes to the maintainability (see Section 4.1.4) of the system.

The translation of part info fields is less trivial. When the field consists of *only* a generic name, it can be automatically translated, reusing the existing component that translates generic names (see Section 4.1.5). There are also some terms that are specific to the part info, that can be translated using its own lookup table. All other terms that can occur in the part info, are left untouched.

The subclasses of the `AbstractLookupEngine` (these are named `GenericNameEngine`, `InstrumentEngine`, `NumberEngine`, and `PartInfoEngine`) define more specific functionality other than being a lookup engine. The differences between the lookup engines lie in concatenation (“ and ” vs. “, ” vs. “ ; ”), instrument numbers (since a number of instruments is always denoted in square brackets, if the number of instruments is multiple), capitalisation

and opus numbers.

The `AbstractTranslator` defines functionality for merging translations to the target languages into a result `Dictionary`. The translations to different languages are done in methods that should be implemented in the subclasses.

The subclass `KeyTranslator` translates the keys to the target languages, in separate methods for English, French, and German. These translations are returned as a `Dictionary` using the functionality from its parent, `AbstractTranslator`.

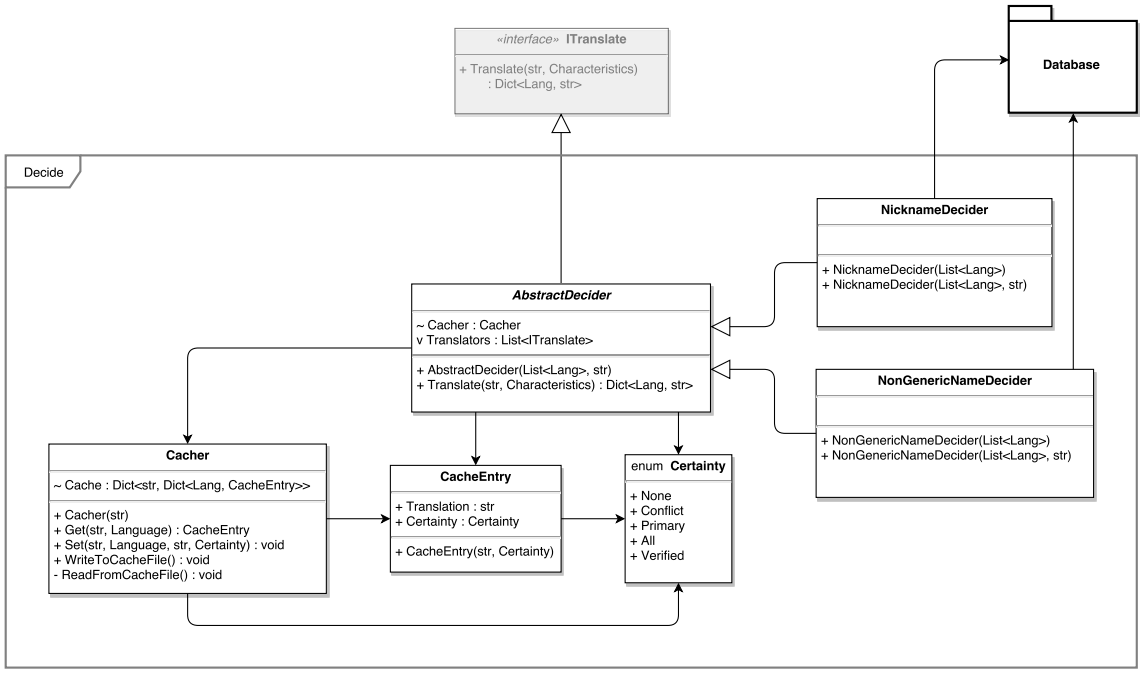

Figure 4.4: Simplified version of the UML diagram for the `Decide` subsystem.

4.2.3. Decisions about Translations using External Databases

Since there are multiple external databases used in the process of translating nicknames and non-generic names, a subsystem called `Decide` is created to decide which translation will eventually be returned as result. A list of translators, one for every external database, is used to prioritise the external translations. This list can easily be extended in the future if any new external databases should be used for translation (making it open for extension, as was the design goal in Section 4.1.1). After translating a title, a level of certainty is assigned to this translation. A simplified version of the UML diagram for this subsystem can be found in Figure 4.4.

This decision process is implemented in a class called `AbstractDecider`. It has two subclasses: `NicknameDecider` and `NonGenericNameDecider`. They both have the ability to define different lists of external database translators. The `NonGenericNameDecider` has a small difference: it wraps its translators in an `AtSignHandler` (see Section 4.2.5) using the decorator pattern.

This subsystem also contains a class called `Cacher` that can maintain a cache file to reduce calls to external databases, because one name can be in multiple uniform titles (see Examples 2.5 and 2.6). The `Decider` classes will, for every incoming name, check if it is not already in the cache file. If the cache file already contains the name, the cache entry is directly returned without consulting external databases. The certainty level of each translation (assigned by the `Decider` classes) is also written to the cache.

These cache files can be manually checked. If they contain translations that are incorrect, the incorrect entries can be manually corrected, so that future translations of the same name will be correct.

The one who checks the cache file will also see the certainty value. This value can be used to help determine whether it is needed to check this translation. Lower certainty values might have more need for manual verification than higher certainty values. When a translation is corrected, the certainty can be changed to the highest level, to remember that this translation has been manually modified.

The `Decide` package also contains two auxiliary classes. A data class `CacheEntry` is used to link a translation with its certainty level. An enumeration `Certainty` is used to determine the certainty levels. It has five possible values, in order increasing of certainty:

0. `None` - No translations have been found, and the input is returned as default.
1. `Conflicting` - Separate external databases have found different translations. The translation of the first translator is returned.
2. `Primary` - Only the first translator in the list has found a translation.
3. `All` - All translators agree on the translation.
4. `Verified` - This translation has been manually verified. This level can only be set in the cache during manual verification.

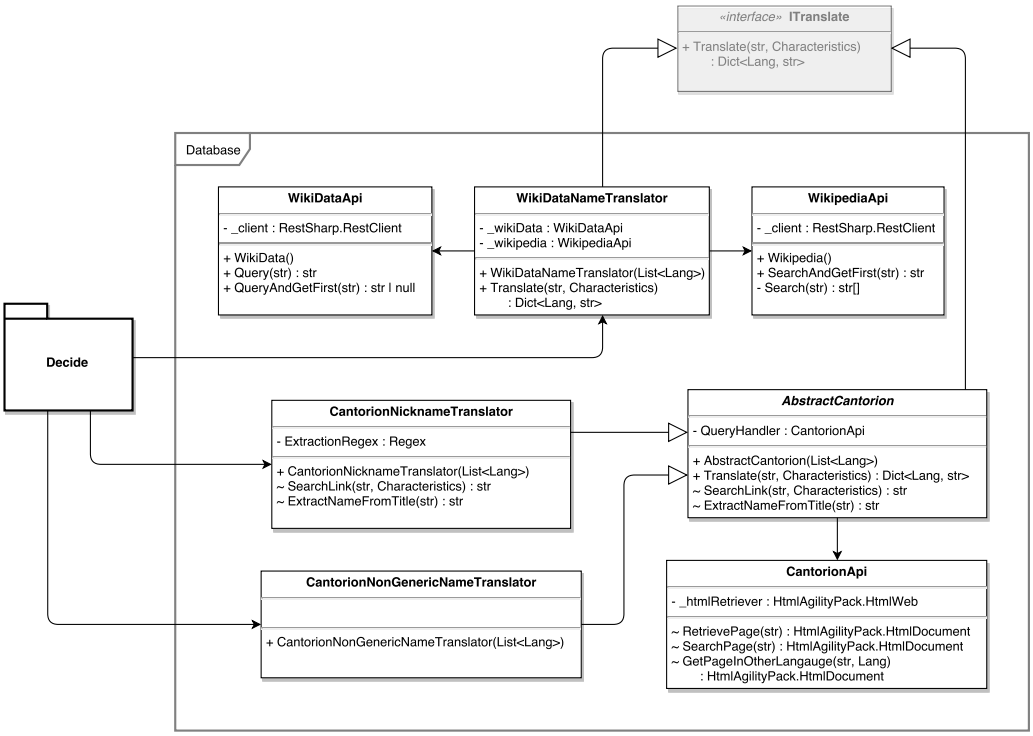

Figure 4.5: Simplified version of the UML diagram for the `Database` subsystem.

4.2.4. Connections to External Databases

The `Database` subsystem handles the connection to external databases. For each external database, this subsystem contains a class that can be used to make calls to the Application Programming Interface (API) of these databases. Next to these classes, this subsystem contains classes that make use of these APIs to translate nicknames and non-generic names. A simplified version of the UML diagram for this subsystem can be found in Figure 4.5.

This subsystem consists of three classes that make use of APIs of external databases: `WikipediaApi`, `WikiDataApi`, and `CantorionApi`. In the Problem Analysis (Section 3.3.2), it was decided that DBpedia and MusicBrainz would be used as external databases. Instead

of DBpedia, WikiData is used, because it is updated more frequently than DBpedia. Both databases contain the data from Wikipedia, but using WikiData will yield more up-to-date results. Also, we swapped MusicBrainz for Cantorion. MusicBrainz has turned out to have no search API for Dutch titles, rendering it useless. Cantorion is relatively lesser-known and thus has less data than MusicBrainz, but can still serve as provider for translations.

The `WikiDataNameTranslator` class translates nicknames and non-generic names using Wikipedia and WikiData. The `WikipediaApi` class is used to send queries for the incoming names to the public search API of Wikipedia. This is done because pages on WikiData are indexed case-sensitively on the Wikipedia page names. Sending a query to the Wikipedia search API helps make sure that the correct identifier is used for the WikiData pages. The `WikiDataApi` class uses this identifier to retrieve the same page in other languages by issuing a SPARQL query to the public WikiData endpoint, as shown in 3.3.2 (this query is slightly altered, since the structure of DBpedia is slightly different than that of WikiData).

The `CantorionApi` class connects to the Cantorion website. This website does not have a public search API, thus it fetches the HyperText Markup Language (HTML) pages from the website instead. The `AbstractCantorion` class uses this class in two phases. Firstly, the search engine of the website is used to select the correct music piece. Secondly, the webpage for this music page is accessed. If this page contains a section called “Other titles”, this section is scanned for translations. If this section does not exist, an attempt is made to access the same page in a different language, by changing the subdomain from `nl.cantorion.org` to `en`, `fr`, and `de`.

The `AbstractCantorion` class is abstract, because different behaviour is needed to translate nicknames (done in the `CantorionNicknameTranslator`) or non-generic names (in the `CantorionNonGenericNameTranslator`). Since Cantorion is a website built to provide sheet music for classical music, it knows the difference between generic titles and non-generic titles. Therefore, when a classical music piece has a nickname, Cantorion will also put this nickname in quotation marks. The result is that the `CantorionNicknameTranslator` has to extract this nickname from the quotation marks. Since most of the functionality for querying Cantorion resides in the abstract superclass, this contributes to the reuse of components (see the design goal in Section 4.1.5).

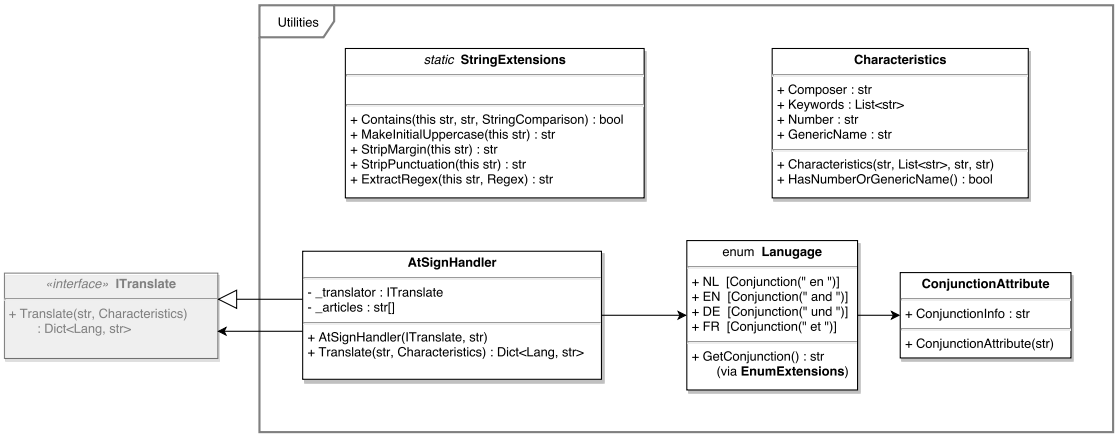

Figure 4.6: Simplified version of the UML diagram for the `Utilities` subsystem.

4.2.5. Utility Classes

The `Utilities` subsystem contains all classes that do not specifically belong to one subsystem because they are used in multiple places in the program. A simplified version of the UML diagram for this subsystem can be found in Figure 4.6.

The `AtSignHandler` class implements the `ITranslate` interface, but is not a translator by itself. Instead, it makes sure that other translators do not have to worry about `@`-signs by means of the decorator pattern. This class is used in the `Decider` classes for non-generic

names and in the `Splitter` for generic names (taking into account the design goal of Section 4.1.5, Reuse of components).

After removing the `@`-sign from the input, this class passes the item to another translator. When that translator returned its result, the `AtSignHandler` inserts an `@`-sign into this result, taking articles of any language into account, using a list of articles in multiple languages that Muziekweb provided for us.

The `Language` enumeration is a list of languages used throughout the program. Instead of using "magic strings" to encode which language is used, an item from this enumeration (`NL`, `EN`, `FR`, or `DE`) can be used instead. Using strings would be error-prone, since it is possible to make typos or change capitalisation. The compiler cannot warn for this type of mistakes, but it can warn you when an invalid enumeration element is used. The `Language` enumeration improves the maintainability of the program (see Section 4.1.4).

The items of the `Language` enumeration also have an attribute that indicates the conjunction word for that language. This attribute is implemented in the `ConjunctionAttribute` class. The conjunctions for Dutch, English, French, German are "en", "and", "et", "und" respectively. These conjunction words are used in the `GenericNameEngine`, because generic names can be concatenated using these words (as shown in Examples 2.2 and 2.3).

As each uniform title can have characteristics that can be used for translation, these characteristics should be passed around to the translators. The characteristics are stored in instances of a data class called `Characteristics` to encapsulate them. This class defines four optional fields: one for the composer, one for a list of keywords, one for the generic name and one for the number field of the uniform title. These characteristics objects are created in the `FileHandler` class (since composer and keywords can be found in the input file, see Section 2.1.3). They are optionally updated in the `Splitter`, adding the fields for the generic name and number if they exist. The characteristics are used in the `WikiDataNameTranslator` and `AbstractCantorion` classes to improve the queries to external databases.

Finally, this subsystem contains a `StringExtensions` class. This class defines a number of methods that extend the native `String` class with functionality that is not available in C#. These methods are defined as static methods, but their signatures indicate that these methods can be called as instance methods on an instance of `String`. The method signature, shown in Example 4.1, has the first parameter annotated with `this` to make this possible. The method in this example can be called as in Example 4.2.

Example 4.1. `public static string MakeInitialUppercase(this string input)`

Example 4.2. `"apple pie".MakeInitialUppercase()` will return `"Apple pie"`

Since these extension methods are used throughout the code, they contribute to the design goal in Section 4.1.5, Reuse of components.

5 Testing

The output of the algorithm, consisting of uniform title translations, requires to be correct, meeting all kinds of rules and a standard. Testing has been an important process that has been part of each development cycle, which eventually guarantees the correctness of the output. The level on which we tested the software has varied from unit level to an entire final product level and this chapter is dedicated to explaining each of these testing approaches.

Note that the test processes described in this chapter will not yet be evaluated here, as this is done on the entire project level in Chapter 6.

5.1. Unit Tests

One of the first things we did in week 1 of the project was setting up a test environment. After some research, we decided to use NUnit for unit testing, as NUnit seemed to have become an industrial standard for unit testing in C# [43]. The initial test environment was set up in the same week and from then on we really started implementing features. The unit tests have been added to a separate project called 'TranslateTest' within the solution. A parallel class hierarchy has been used for unit testing the software. 'TranslateTest' contains a test class for each class and each test class contains tests for each component of the relevant class.

The unit tests are created in such a way that test code duplication is minimised as much as possible and test code is as meaningful as possible. This has been achieved by using parameterised tests wherever possible, applying a parallel class hierarchy, and using mocking features. Parameterised tests have been applied quite frequently, as, for example, the different kind of translations of a uniform title (fields) that are expected to be outputted can easily be covered using the same test code. Paramaterised tests also make the test in some way open for extension. For example, if you want to cover another case for a method and there already is a parameterised test, then all you have to do is add a specification of the expected input and output for the case. By applying a parallel class hierarchy in the tests, functional components of abstract classes can be tested using abstract test classes, instead of duplicating these tests in all of their subclasses, for example. Mocking features have been used to verify method calls by the method under test at unit level, for instance.

5.2. Code Coverage Tools

As soon as the test environment had been set up for unit tests, we wanted a code coverage report generating tool. The reports of such a tool would support in providing insight about what has been tested well and what has not been yet.

In Section 2.2.2, which is part of the Project Plan (see Appendix B) created in week 1, we mentioned that by using the services of Coveralls we would already have a good way of integrating code coverage analysis into our development cycles, as it supports C#. However, you have to pay for the services once you want to add a private repository [10]. Therefore, in week 2, we decided to use the services of Codecov instead, as Codecov does allow adding 1 private repository without any charges and supports code coverage analysis for C# [8].

All that was needed next is an actual code coverage generating tool so that we can generate the reports whenever we build the solution and send these to Codecov for analysis. Besides an analysis, Codecov also provides an online fancy overview of the code coverage status. We researched this and decided to use a tool called 'OpenCover' which does exactly that [31] and integrated it via NuGet. As a side note, NuGet is a package manager for the Microsoft development platform (including .NET) [27].

As building is done in an automated fashion via continuous integration, which will be discussed in Section 6.4, after each commit we are aware of code coverage changes. We have also integrated code coverage status reporting bots into our communication tool and GitHub for ease, but this will be discussed in Chapter 6.

As a consequence, aside from code coverage change reports on commit level, we have also received them on pull request level. This is because Codecov reports the code coverage differences when merging a branch into the development branch, for instance.

The resulting code coverage number of the final product is included in Chapter 7 as part of the final product evaluation. A testing report has been added as an appendix (see Appendix E), elaborating on the code coverage results.

5.3. Test-Driven Development

Whenever we added a feature, we tried to do this in a test-driven way. Doing so encourages the developer to think about how to implement the feature. Also, it gave the developer the purpose to, aside from implementing the feature itself, implement the feature against a set of predefined tests.

5.4. Regression Testing

Once a few small features were implemented, including their tests, and the steps were taken described in the previous two sections, we were fully able of applying regression testing to the development cycle. Whenever we introduced some change to an existing component, then by running its tests again, these changes were reflected in the tests. I.e. if the (regression) test would fail, we knew that the regression test either required maintenance or that a defect had been revealed. Especially the latter was very useful, as it was a way of showing the presence of bugs.

5.5. System Testing

As mentioned in Chapter 1, Muziekweb initially wanted to translate just over 250.000 uniform titles, which is the number of uniform titles that they had in their database. Before the project started, 1.000 of these titles were already translated manually by experts. These 1.000 translations can perfectly be used as a training set of the program.

5.5.1. A Validation Script

In week 3, a (Bash) script has been created for comparing the manually translated uniform titles with the automated ones. The script accepts four arguments: show help, verbose mode, interactive mode, and colour mode.

Trivially, the help option shows the possible parameters including an explanation. Besides, verbose mode compares the entire input file against the manual translations at once, while interactive mode does the same thing but expects the user to press enter after each translation. Finally, colour mode is an optional parameter for verbose mode and interactive mode. The output of verbose and interactive mode is a diff, which can be coloured by the optional colour mode parameter. The characters in red represent the original characters from the manual translation and characters in green represent the characters from the automated translations. When all translations are finished, the ratio of perfectly translated uniform titles compared to the total amount is printed. As an illustration, Figure 5.1 is a screenshot of an example diff of a uniform title using colour mode (and interactive mode).

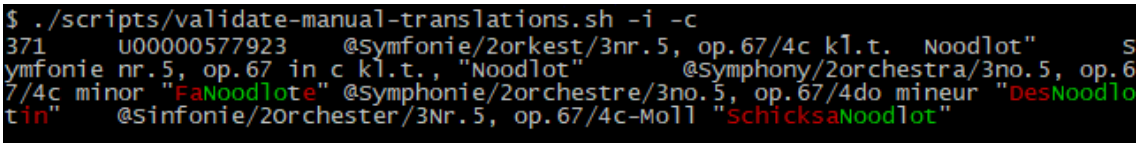

Figure 5.1: Example of a validated uniform title via the validation script.

As a consequence, from week 3 on, we were able to monitor our progress in terms of uniform title translation. Not only was this fun to include in our meetings with the client and coach, but in general also very insightful on where progress could be made and where progress was made. This way of testing offered us another way of finding defects.

The resulting number from the script of the last sprint is included in Chapter 7 as part of the final product evaluation.

5.5.2. A Validator

During our midterm meeting in week 6, the idea was posed to not only validate the entire uniform title, like we did with the validation script, but also the different fields of each uniform title. By doing so, we would relatively easier get a more precise overview of our progress on field level.

We have decided to add a new project called 'Validator' to our solution. The project contains the code to use our current system and validate on both field level and uniform title level. The Validator project is not part of the system, but only there for testing purposes.

The 'Validator' is very similar to the validation script of Section 5.5.1 in terms of how it works, but does not have a colour mode. It indicates what is expected and what is the actual translation, which are the manual translation and the automated translation, respectively, including the line number of the relevant translation in the input file.

The program accepts one or more of the following arguments: 'genericname', 'instruments', 'number', 'key', 'nickname', 'partinfo', 'nongenericname', 'all', and 'quick'. If no argument is supplied, each possible argument with an explanation is printed. These arguments represent the 7 fields that require a translation of the fields defined in Section 2.1.2 and two extra arguments.

The first 7 arguments of the previous argument listing return the differences for the relevant field of each uniform title, if it contains the field. Besides, argument 'all' returns the differences of each entire uniform title. Finally, 'quick' returns the differences of all fields that do not require external databases for the translation. The argument is called 'quick' because involving external databases increases the time required for a translation significantly, trivially. For field arguments, aside from printing the ratio of perfectly translated relevant fields when compared to the total amount, it also prints the amount of uniform titles that have been skipped as they do not contain the relevant field. As an illustration, Figure 5.2 is a screenshot of a part of the output of the Validator when running 'quick'; only the output of fields 'key' and 'part info' are presented.

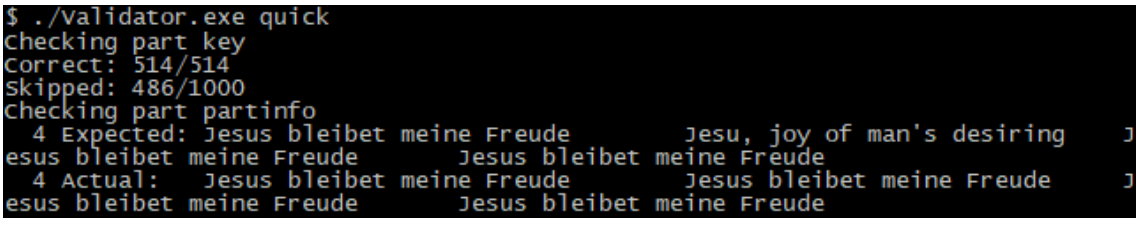

Figure 5.2: Example output part of running the Validator using argument 'quick'.

As a consequence, from week 6 on, we were able to monitor our progress in terms of field translations more closely. Instead of running the validation script during our meetings, we used the Validator both on field and uniform title level for showing our progress. This way of testing made the validation process even more powerful in terms of detecting possible defects.

5.6. Performance Testing

As the end of the project came closer, over the period of week 8 and 9 multiple test runs have been performed. During such a test run an input file of 10.000 uniform titles has been generated and given to the system.

The main reason for running the first performance tests was related to checking whether the system can actually handle such a relatively large amount of uniform titles. A few times during the first run, an `ArgumentNullException` was thrown. These defects were afterwards immediately resolved, but detecting them would have been quite hard based on the testing methods that have been described in the previous sections. One of them, for example, was related to an erroneous Cantorion webpage, which required us to better take into account such possible erroneous results.

As described in Section 4.2.3, a cacher has been developed. Such a test run would be the perfect opportunity to test its effect on the performance of the system over time. Therefore, two other test runs were done. One run was done with our final product, and the other run had the cacher removed. A graph has been created with the results of these runs, presented by Figure 5.3.

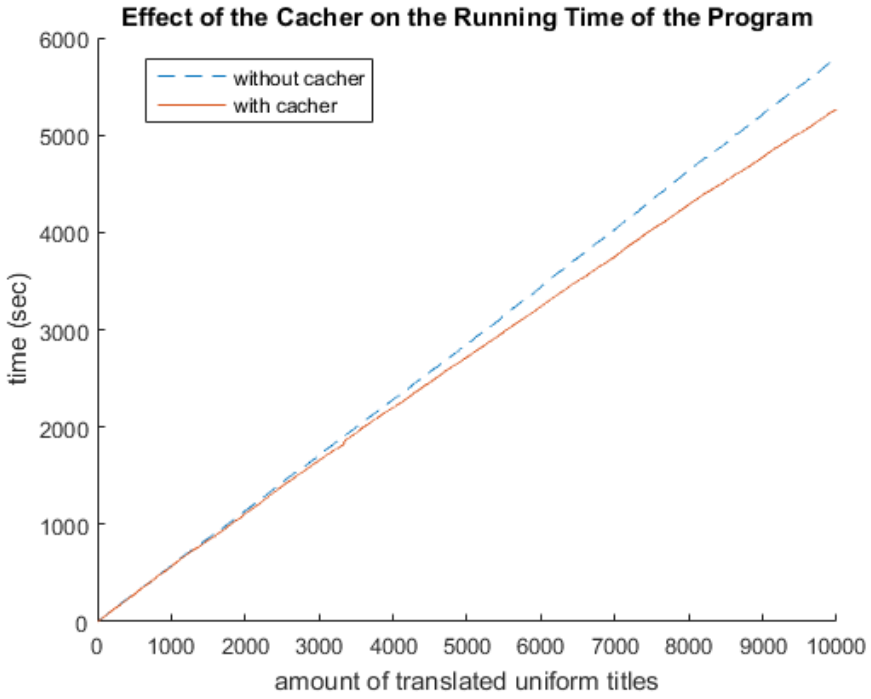

Figure 5.3: Effect of the Cacher on the Running Time of the Program

From Figure 5.3 it can be deduced that without the cacher, the running time of the program fits a linear curve. The fitted curve is not drawn, as the original line is very close to the linear fit; the R^2 value equals 0.9999. When introducing the cacher, initially, the line follows the same curve as the line without the cacher, which makes sense as the cacher will only have effect on the running time when items are actually cached. After some time, the line leaves this linear curve and the translation speed relatively slowly increases over time. When we introduce the cacher, the resulting line fits a power curve with an exponent of 0.982, meaning that it is sublinear, with an R^2 value that equals 0.9997. The results therefore reflect the aim and the effects of the cacher. However, we do need to be objective here. These two runs were performed once; doing the same experiment multiple times would provide even more certainty about the effects. It is possible that other processes that were running at the same time on the laptop could have influenced the runs. Unfortunately, there was not enough time to perform multiple experiments.

5.7. Manual Verification

In week 9, together with the client, we decided that it would be useful to randomly pick 100 uniform titles as a test subset from the set of approximately 250.000 uniform titles (excluding the 1.000 uniform titles that were used as training set), translate these and hand them over to the experts at the classical music department of Muziekweb. They would then manually verify whether the translations are correct and only mark the fields that are wrong using a scale that we came up with. We provided them an easy way to indicate the scale and any feedback. The scale was enumerated in the following way: 0 meant the translation was wrong and could hardly be automated as it required human expertise, 1 meant that the translation was wrong and could be automated, 2 meant the translation was alright but not entirely correct, and 3 meant that the translation was perfect. The experts did not need to indicate the “3”, since many fields were translated almost flawlessly, parallel to the Dutch proverb “geen bericht is goed bericht” (no message is a good message).

The outcome of this manual verification test will be presented in Section 7.4.

(This page has been intentionally left blank.)

6

Process Evaluation

In this chapter, the development process will be evaluated. In Section 6.1, the agile development method Scrum is discussed, and how it affected the process. In Section 6.2, the test-driven development is brought up, followed by providing the pros and cons of this method. In Section 6.3, the choices of development resources are explained, and whether they provided to be useful or not. After that, in Section 6.4, continuous integration is taken a look at, closing off with the advantages of continuous integration which were encountered. Finally, in Section 6.5, the main tool of communication is explained.

6.1. Scrum

During the development process, the Scrum method was used. This agile development method consisted of weekly development cycles called sprints, with a working version of the program being delivered at the end of each sprint. This way, Muziekweb could provide useful feedback on the system. Scrum was also used by Muziekweb, so they recommended using Scrum. Since Scrum was extensively used during the Bachelor programme, this did not cause any troubles.

At the end of each sprint, a meeting was held with Muziekweb to provide them information on what had been done during each sprint. In these meetings, feedback was also provided on the system, so that the input of the Muziekweb was able to get resolved as well in the product.

In the end, the Scrum method worked out pretty well. A total of 7 sprints were done, normally lasting 1 week each. However, two of the sprints covered two weeks. Sprint 4 covered two weeks, due to having multiple days off in that week. Instead of having three days for a singular sprint, these 3 days were added to another sprint. Sprint 7 was also extended, since the code was due in a weekend. Since regular sprints ended on Wednesdays, it would be unwise to make another sprint for the final two days. Therefore, the final sprint was extended to last a week and a half. Each sprint resolved around implementing a new feature of the program, with bug fixes of the previous sprint being done on the side. Now and then, not all tasks were able to be finished at the end of each week, but those were mostly tasks that did not involve the main functionality implemented that sprint. Due to Scrum, Muziekweb could provide valuable feedback on the new functionality. If they wished any changes, those changes were easily added to the next sprint, so that Muziekweb could see the differences quickly again.

6.2. Test-Driven Development

Developing a good system, included providing well-tested code. For this, several different techniques were used. The one that was mostly used was Test-Driven Development (TDD). The principle of TDD was writing tests before actually implementing a feature. The reason for using this technique was ensuring code is thought out before implementation.

During the development, TDD was used, but not for every feature. The reason for not using TDD all the time was that some development tasks could not be thought out completely beforehand, such as covering all corner cases. Since these depended on how the classes were structured, these tests were written afterwards. However, other results could be thought out beforehand, such as the translation of the key from a uniform titles. Since the translations for the keys were known beforehand, TDD could be used to implement the tests before writing the code.

6.3. Development Resources

For making the development process go smoothly, several choices have been made on which resources to be used. In this section, the choices that needed to be made are elucidated, followed by these choices being discussed, whether they provided to be useful or not. Note that one of the team members used Linux while the others worked on Microsoft Windows, so the resources should work on both operating systems. Firstly, in Section 6.3.1, the programming language to be used is looked at. Secondly, in Section 6.3.2, the IDEs used are given. Thirdly, in Section 6.3.3, the internet hosting service that was used is shown. After that, in Section 6.3.4 the used static analysis tools are explained. Finally, in Section 6.3.5 the tools for managing sprints are enlightened.

6.3.1. Programming Language

For choosing which language to use, Muziekweb did not ask for any specific language. Their only requirement was, was that the program needed to work in combination with their current systems, which were mostly written in C#. So the choice was either writing the code in C#, or writing it in another language and making an adapter to C#. The only downside to using C# was that there was no prior experience using it. To make the integration of the program as easy as possible, and sparing the effort on writing an adapter, it was decided to write the program in C#, even though the language was never used before. Not only would that make the integration easy, but it would also make any future improvements of the program more easy for the current developers at Muziekweb.

Since there was no prior experience with C#, some research into the language had to be done. However, since the language is very similar to Java, it was not very difficult to learn it. One unfortunate problem was that Mono, an implementation of the C# environment for Linux, is not officially supported by Microsoft. As such, some features of the .NET framework on Windows are not able to be implemented in Mono. Therefore, some tools were not able to be used. This cost extra effort in finding and setting up tools that worked on both Windows and Linux. Although setting up the tools took quite a while, once all the tools were up and running, coding went pretty smoothly in C#.

6.3.2. IDE

Among the developers, two different IDEs were used: JetBrains Rider and Microsoft Visual Studio. Rider is an IDE that has the features from other JetBrains IDEs and the ReSharper plugin for Visual Studio (see Section 6.3.4) combined.

Whereas Rider runs on both Windows and Linux, Visual Studio only runs on Windows. Since Muziekweb works with Visual Studio, the program needed to be built using Visual Studio to make sure it could be built at Muziekweb as well. Even though Rider was the favourite IDE for most of the team members, Visual Studio was still used by one of the team members (and occasionally by a second one as well) to make sure Muziekweb could eventually maintain the software product.

6.3.3. GitHub

To host the project, GitHub was used to store out project. GitHub was a web-based version of Git, which was most often used to store code. Muziekweb created a private repository on here to manage the project. Since several tools had to be integrated with GitHub, the repository ownership was transferred to us. At the end of the project, these rights were transferred back to Muziekweb.

6.3.4. Static Analysis Tools

To ensure the code quality remained high, three static analysis tools have been used. The first tool that has been used, was FxCop. FxCop is a compiler plugin that analysed the code for writing robust and maintainable code [14]. This is done according to the Design Guidelines set out by Microsoft. Besides that, StyleCop has also been used as a static analysis tool. StyleCop analyses the code and enforces a set of common style rules [44]. As a final static analysis tool, ReSharper has been used. ReSharper is an extension developed by JetBrains, which not only warns you when there is a problem in the code, but also suggests fixes to solve them automatically [38]. For a more detailed description of these tools, see Section 9.2.

The usage of these tools provided a more consistent style in coding. Many common errors that were made, were easily detected before committing new code. This made the coding style among us very consistent. However, even though StyleCop provided a lot of rules on style, it did not provide style rules on everything. Therefore, some rules in style, such as an empty line at the end of each class, had to be manually checked for. Additionally, some warnings from the tools were eventually suppressed. Certain warnings were conflicting, if the warning would have been fixed, another one would have popped up. Therefore, certain warnings were suppressed, after a discussion whether it was the correct thing to do.

6.3.5. Sprint Tools

For managing the progress of sprints, the Waffle tool was used. “Waffle is an automated project management tool powered by your GitHub issues and pull requests” [45]. Waffle keeps track of sprint issues that need to be done. The issues can be categorised, be given an estimated time effort, and assigned to persons to work on the issue. Waffle integrates with GitHub, so the status of the sprint issues were automatically updated, based on commits and pull requests.

For an issue to be considered done, a definition of done was defined at the start of the project (see Appendix C, Section C.5).

Waffle turned out to work pretty smoothly with the development progress. Most of the times, the items correctly updated to the right status. However, for Waffle to recognise that an issue needs to be moved, specific messages need to be used in the Pull Request. Since the messages contained a typo now and then, some issues were in the wrong status for a while.

6.4. Continuous Integration

Whenever multiple developers are writing and uploading code several times a day and needs to be integrated frequently, it is important that everything is working. This can be done with Continuous Integration (CI). With CI, a build of the code is automated, which can then be checked on standards whether everything is correctly implemented.

For the development of the system, Travis is used as a CI service. Travis is chosen as it is free of charge for students, and it had already been used before. Travis has been integrated with the static analysis tools. Whenever Travis makes a build, it checks whether StyleCop gives any warnings, and fails the build if it does.

However, since Travis is a Linux based platform. Because of this, Travis can only run StyleCop. Also, because only works with Mono, it could not test the build on the official .NET framework. To ensure that it would work on both systems, an extra CI was used: Buildkite. Buildkite provided an agent on any machine, which was able to execute any script [5]. These agents ran on our own local systems, including both Linux and Windows. This way, it was ensured that the program works on both platforms.

In the end, the CI helped us figure out whether the newly added code was actually running correctly, independent of the machine. Several occasions occurred where everything was fine on a local machine, but problems happened in the Travis build. This helped with integrating the code so that the problems did not pile up in the end.

6.5. Slack

As the main communication tool, Slack has been used. Slack allows for different channels to be made, so each topic can have its own channel in which it can be placed. There were two different kinds of channels which were used.

Firstly, we got the conversation channels, where we either discussed programming related issues, announce updates on the projects and communicating with Muziekweb when not present at the office of Muziekweb.

Besides that, we also had channels which were mostly used by our development tools. Since Slack allows incoming webhooks to be set up, they could announce messages in a dedicated channel as well. Therefore, there were also channels for GitHub, Travis, Waffle, CodeCov and Buildkite. These would share messages whenever something happened within these tools. For example, GitHub would announce it whenever a commit was pushed, and BuildKite would announce whether a build passed or failed.

7

Final Product Evaluation

In this chapter, we will evaluate the final product. In Section 7.1, the user requirements are checked, whether they have been satisfied or not. In Section 7.2, the design goals are evaluated. After that, in Section 7.3, the results from testing are shown. Finally, in Section 7.4, the results from validating the translations of the program are presented.

7.1. Evaluation of User Requirements

In this section, the user requirements set at the start of the development process are discussed. These are the requirements from the MoSCoW method, which can be found in Section 2.2. The results of this evaluation are presented in Table 7.2, at the end of this chapter. The "won't haves" are not in included in this table, since they were never intended to be implemented. For each requirement, it is first stated whether the requirement is met or not. It the requirement is met, it is explained how this requirement was met. Otherwise, arguments are given by the requirement has not been met.

All non-functional requirements have also been met, with some exceptions. For static analysis tools, FxCop and ReSharper have been used next to StyleCop (see Section 6.3.4. As a code coverage tool, Codecov has been used instead of Coveralls (see Section 5.2).

7.2. Design Goals Evaluation

In this section, the design goals (presented in Section 4.1) are evaluated. The design goals are extensibility, scalability, fault tolerance, maintainability, and reuse of components. For each of the design goals, arguments are given why the goals are met. These goals are important for the developers at Muziekweb, since they need to continue using the code that was written.

7.2.1. Extensibility

Extensibility was the design goal with the highest priority. The two factors that were focused on were the target languages and databases to be used.

The target languages are easily extensible, since the external databases also contain the translations for other languages, however, these results are not yet used. The only things that need to be adjusted are lookup tables. The lookup tables only contain the translations for three languages (English, French, and German). To make the program work with other languages as well, extra columns have to be added to each lookup table. But besides that, new languages are quite easily added.

Besides this, we got the databases. Currently, only two databases are used (WikiData and Cantorion). However, if more databases are wished to be added, this can be quite easily done. There exists an interface in the program which the new database class must inherit from. The only thing that needs to be changed then, is adding it to the `Decider` class. One thing to mention is that the certainty for translations needs to be adjusted a bit, since we do not now how reliable new databases will be. Currently, there is a primary database which

is more heavily relied upon. However, if a more reliable database is added, this needs to be adjusted. But this is quite easily done by switching the order of the databases in the decider.

7.2.2. Scalability

The program eventually needs to work with a large number of titles. Therefore, precautions need to be made. The biggest factor that was focused on is the number of calls made to external databases. For the translation of nicknames and non-generic names, a query is made to an external database. However, if this database is called every time for each title, the program will be very slow. Additionally, the same name can appear in multiple titles as well. Therefore, a cacher has been implemented. This cacher stores the found translations in a local file. Whenever a nickname or non-generic name needs to be translated, it will first check this local file, instead of making the call to the external database.

To see how much RAM would be used during the execution of the program, a test run of 10.000 titles was done in the second to last week. During the entire run, the RAM used approximately 80MB. Since the computer on which this program will be run has gigabytes of RAM, this will not pose any problems.

7.2.3. Fault tolerance

To ensure that our program can deal with any errors in the database, an extra safety measure is implemented. If a title cannot be translated for any reason (such as an error in the title) and an exception is thrown, this will be caught and the program continues with the next title. This way, the program will not crash whenever it encounters an error in the database, making a bulk run most likely to finish.

7.2.4. Maintainability

There are several ways to keep the code maintainable, such as reducing code duplication (more info on that in Section 7.2.5), keeping methods short, and writing tests. Since there are so many factors to this, it is difficult and tedious to assess them one by one. However, SIG has provided us with an analysis of our code, ranking our maintainability. This analysis can be found in Appendix F. Halfway through the project, they ranked the program with 4.5 out of 5 stars, only having comments on lengthy methods. With the lengthy methods split up, our code had proven to be very maintainable at that time.

At the end of the project, a second submission to SIG was made. Their feedback can also be found in Appendix F. SIG stated that we had correctly reduced the method lengths by splitting up functionalities into separate methods. With a final ranking of 5 out of 5 stars, we have finished this project with a highly maintainable code base.

7.2.5. Reuse of Components

To make sure components are reused, many abstract ancestors have been made for similar classes. Any code that would be in both subclasses, is instead placed in the ancestor. This causes there to be as little code duplication as possible. Since the amount of code duplication is at a minimum, this also increases maintainability.

Additionally, some lookup engines can be used among different parts. Since the part info can just consist of a generic name, it means that the `GenericNameEngine` can also be used for that, instead of making a completely new lookup engine. Furthermore, a `StringExtensions` class was created. This class extends the `String` class, which is reused among multiple classes as well.

7.3. Testing

Since we maintained a high focus on testing, our total code coverage ended up at being 99%. The only class that was not fully tested, was `AbstractCantorion`. This class contained a branch that was added as a safety measure that cannot be reached in practice unless a mistake is made on Cantorion's website. Even though it is currently not possible to trigger that branch, it is still left in the code, in case Cantorion's website changes in the future. The full test results can be found in Appendix E.

7.4. Validation of Test Translations

At the beginning of the project, a file with already manually translated titles were already provided. These consisted of the 1000 most popular titles, and the translations of them to English, French, and German. To validate whether the designed program translates correctly, the results of the program are compared to the file with manual translations.

To check how well the translations are, comparisons are made for each part of the title, such as instruments, keys, and nicknames. If a title does not contain any of these fields, the comparison is "skipped", which has been explained in 5.5.2. The results of this comparison are found in Table 7.1.

Part	#Titles with part	#Correctly translated	Percentage correct
Generic Name	619	619	100%
Non-Generic Name	381	262	69%
Instruments	619	619	100%
Number	777	777	100%
Key	514	514	100%
Nickname	216	131	60%
Part Info	178	164	92%

Table 7.1: The results from comparing the program's translation to the manual translation. The first column shows how many titles out of a 1000 have that part. The second column is how many of those are correctly translated.

As you can see in the results, some parts are perfectly translated. The ones that are not perfectly translated are "Non-Generic Names", "Nicknames", and "Part Info". Since these the first two parts utilise an external database, they are not perfectly translated. The "Part Info" itself can only be manually translated, since the structure of this part is arbitrary. This is due to the database not being perfect, either with missing entries, or slightly different spellings and capitalisation of translations. This leads to a relatively low number of correctly translated, even though the correct translation is found.

Besides comparing the translations of our program with the manually translated titles, we also took another 100 sample titles and translated them. These 100 titles were then verified manually by the experts of Muziekweb.

From the 100 titles, 88 were correctly translated. However, since this verification is more lenient than the verification done on the 1000 titles, more translations are labelled as correctly translated. The verification in this part will label titles as correctly translated, even if the spelling or capitalisation is slightly off. From the twelve titles that were not correctly translated, eleven of those titles had an indication that they could not be translated automatically. This means, that the translation requires human expertise. Therefore, 99 out of a 100 titles are correctly translated by the program or need manual translation.

Requirement	Implemented?	Explanation
The program must be able to translate uniform titles for classical music pieces.	Yes	The program can accept two kinds of classical titles: generic titles or non-generic titles. These titles can be split up, and individual segments are translated.
The translated form of these uniform titles must maintain the structure of the title.	Yes	After splitting up the title and translating the individual parts, the translations are substituted in the original title. This way, the original structure of the title is maintained.
The program's input and output must be in the form of a TSV file.	Yes	The program reads out the input based on tab characters, being able to read a file with any amount of columns. The columns used are title, composer, and keywords. Any other columns can be present, but will not be utilised. The output has the same columns as the input file, but with an added column for each language which has been translated to, containing the translated title.
The program must translate Dutch parts of the uniform titles to the following target languages: English, French and German	Yes	The program has the functionality to translate to the three given languages. The program is structured in such a way, that additional languages can easily be added, in case extension is wanted.
The program must translate the titles while satisfying the PICA constraints.	Yes	The titles are split based on the PICA constraints, and substitutes the translated parts back in the original PICA-compliant title. Therefore, the translated titles are also PICA-compliant.
"Nicknames" (i.e. names in quotes) that are not in Dutch, must be left in their original language.	No	After more research, not all non-Dutch titles are the same in all other languages. Therefore, only titles that are Gregorian are not translated, since these are guaranteed of not needing a translation. A title is considered Gregorian if it has `__@gregoriaans` as one of its keywords.
The TSV file should be tab-separated to avoid extra string escaping (since commas, semicolons and quotation marks are also used in the structure of the uniform titles).	Yes	The given Excel files have been converted to TSV files, which made it possible to avoid extra escaping of specific strings.
The program should be able to distinguish Dutch parts from Latin parts of titles.	No	The program is not able to explicitly distinguish between Dutch and Latin. However, with the databases that are used, most Latin titles will remain Latin as well. Also, titles that are Gregorian are not translated.
Latin parts should remain in Latin in any (modern) language.	No	Although many Latin parts will remain in Latin, it is not entirely sure whether they will all stay in Latin (since external databases are used as well).
The program should have an interactive mode in which ambiguous titles can be translated directly by the user.	No	Instead of having an interactive mode, a cacher is used to store the nicknames and non-generic names with a certainty attached to them. This file can be manually updated to the correct translation wherever needed. This feature is implemented for the classical department of Muziekweb, so they can easily verify whether translations are correct or not, but do not have to sit next to the computer all the time.
The program could be able to read the input via the standard input and write the output to the standard output.	Yes	The input and output files can be specified before running the program. As a default, if no file name is given for input and/or output, it will use the standard input and/or output.
The system could use a learning algorithm, which can improve previously given translations.	No	Instead of a learning algorithm, a cacher is implemented storing the certainty of each translation. With more databases being implemented, this certainty could increase or decrease. When more databases are implemented, it can determine a new certainty and possibly overwrite previously generated translations.

Table 7.2: Table containing the user requirements for the program.

8

Ethical Implications

In this chapter, the ethical implications of our system are looked at. In Section 8.1, the ethical implications of one of the translation methods is looked at: web-scraping. Secondly, in Section 8.2, the role of Muziekweb as a fair distributor of knowledge is explained.

8.1. Web-Scraping

The program that is developed, has one specific feature that might involve ethical implications: web scraping. For gathering translations of nicknames and non-generic names, WikiData and Cantorion are used to gather this data. WikiData does have an API for this, but Cantorion does not. Therefore, the translations gotten from Cantorion are scraped. Since many titles contain a nickname and/or a non-generic name, many calls to Cantorion need to be made. This is not the intention of Cantorion.

So is it ethically allowed to scrape data from a website like this? On the one hand, this seems like no big deal at all. After all, all the data is available on the Web, which is accessible to everyone. Therefore, one should be allowed to gather as much data from a website as they want.

However, this is not a person gathering the data, it is a program. With the number of calls the program does to this website, it is clear that it is not humanly possible to do that. Doing so many calls, will demand a lot from their servers, so it will decrease their performance.

Furthermore, Cantorion does not know what is done with the data. It is officially their data, and Cantorion wants to ensure that other parties do not start proclaiming the data is theirs. Since it is difficult to decipher what happens with the scraped data, one might say that therefore scraping is not the right way to go.

However, we do take these disadvantages of web scraping in consideration. One big reason for implementing the cacher, is reducing the load on the Cantorion servers. Every title needs to be only looked up once, after which it is stored in the local cache file. This way, the translation of the entire database can be done in multiple sessions, in case not all titles can be translated in one go. All that have been translated already can be found in the cache file. Therefore, executing the program again will not make the same calls to the Cantorion server again. Furthermore, we have stated in the program that the translations from Cantorion are only used to verify other found translations. We do not proclaim the data is ours, and Muziekweb should not present these translations anywhere without crediting Cantorion.

8.2. Fair Distribution of Knowledge

On the other hand, we have the ethical values of Muziekweb themselves. The goal of translating the titles is not to make a profit. After all, Muziekweb is a non-profit organisation. Their goal is to have their data accessible to as many people as possible. This goes in line with the concept of fair distribution of knowledge. Beforehand, the knowledge of Muziekweb was only comprehensible by people who can read Dutch. This is only a small fraction of the people in the world. With these translations, the data of Muziekweb is also accessible to people who do not speak Dutch, but do speak English, French or German, making the data more fairly distributed.

9

Project Wrap-Up

Before finishing up the project, several steps need to be taken. These steps are described in this chapter. First, in Section 9.1, the process around delivering the code is documented. Then, in Section 9.2, more instructions are given about the tools that were used during the project.

9.1. Code Delivery

To finish up the project, one step that had to be made was delivering the code. The first step in this was handing back the GitHub repository rights to Muziekweb. The rights were temporarily given to us, so we could authorise tools to run on our code. However, since it is still the repository of Muziekweb, rights were transferred back.

The second step was explaining how to execute the software program. This will briefly be explained here as well. An executable file is created when the project is built. There are several arguments that can be provided to the program as well. These arguments are explained in 4.2.1.

As for the input file, it needs to have at least a column with the uniform titles, labelled "TITEL". Additionally, it can have an optional column with the composer, labelled with "COMPONIST". Finally, the input file can contain any number of optional columns for characteristics, as long as the column starts with "TREFWOORD".

9.2. Tool Elaboration

In Section 6.3.4, several tools that were used during the process are described. Since Muziekweb has shown interest in using more tools for coding in the future, the three static analysis tools are elaborated further in this section.

StyleCop, as the name might imply, is the tool that was used to get a consistent style among the developers. StyleCop comes with a big list of rules, which can each be turned on or off. By default, almost all rules are turned on. However, if one desires to not get warnings on specific issues, this can be easily changed. StyleCop also has a plugin for Visual Studio, generating warnings during the build of the solution. These are then displayed in Visual Studio as well.

Secondly, we got FxCop. FxCop analyses the compiled code as well. There are several issues that FxCop can detect, such as naming, security, and performance. In contrast to StyleCop, which mainly focuses on making the code more readable, FxCop focuses more on issues that can happen during run-time. As with StyleCop, FxCop also comes with a big list of rules to check for, in which each rule can be turned on or off separately. Therefore, if specific rules seem inappropriate in a developer's eyes, it could always be turned off.

Finally, we got ReSharper. ReSharper is a plugin for Visual Studio, created by JetBrains. It does not only check the code statically, it also helps with programming itself. It can help with applying solution-wide refactorings, generate code and gives suggestions for fixing issues automatically. It does not check for as many rules as the previous two tools, the extra features do help out a lot while coding.

10

Conclusion

This project started with a challenge posed by Muziekweb; they needed a system that is able of translating all of their current and future Dutch uniform titles to the languages English, French, and German. Clearly, this challenge has been accepted by us. Nine weeks of hard work followed in which a strategic plan was created, a research was conducted and then incrementally an architecture was built, features were added, the system was tested, feedback was taken into account, and the system was evaluated.

The final product solves the problem by splitting it up into subproblems, solving the subproblems, and eventually merging these solutions. Firstly, it reads an input TSV file containing uniform titles and optionally composers and keywords. Then it splits up each uniform title in its fields and these are delegated to relevant translators. Fields requiring standard translations are translated via lookup tables, fields requiring simple translation rules for classical music are translated via such rules, and fields that cannot directly be translated, as they require expertise in the field of classical music, are translated via external databases when they are not already available in the corresponding cache file. The external databases that we use are WikiData and Cantorion, via SPARQL queries and web-scraping respectively. Before actually using these translations, the translations are assessed. WikiData is used for retrieving translations and Cantorion is only used to verify them and increase their certainty rate. Taking the translations into account of both external databases, a decision algorithm decides whether to use WikiData's translation or the original Dutch version based on a certainty value. The results of translations that require external databases are added to the corresponding cache file, including their certainty value. The translations of all parts of the uniform title are eventually merged and written to a TSV file.

Based on the process evaluation in Chapter 6, the system has been developed using a set of good practice software engineering methodologies and involves all kinds of tools in order to facilitate or otherwise enhance the development process. Mostly, we also evaluate these methodologies and tools as good practice components for the process and appreciate the way in which they helped improve it.

Based on the final product evaluation in Chapter 7, the architecture meets all design goals initially stated in Chapter 4, has been tested very well and meets most of the initial requirements introduced in Chapter 2 (at least all major requirements). The requirements that have not literally been met as they were originally stated, have mostly been otherwise met in the form of a similar solution, but a better applicable one. For example, no learning algorithm was required, as the `Cacher` that has been implemented covers the goal of the learning algorithm.

Based on the previous evaluations, we can proudly state that we have found a solution for the problem that was initially posed by Muziekweb and that the project can be seen as a success. It has been developed very well from a software engineering perspective and it meets the initial requirements posed by Muziekweb, as well as the requirements that were posed later during the weekly meetings. Therefore, the final product really is a 'final' product.

(This page has been intentionally left blank.)

11

Recommendations

This chapter closes the report by offering several recommendations related to future use (see Section 11.1) and development of the final product (see Section 11.2).

11.1. Future Use

Initially, the system will be used to translate all uniform titles that Muziekweb currently has in the database (over 250.000). There is not much to recommend here other than just run the system with the input. However, when that is done, we do have several recommendations.

11.1.1. Enjoy the Benefits of the Cacher

First of all, perhaps trivially, preferably do not throw the cache files away once the system has translated the 250.000 uniform titles. It contains data that does not need to be looked up anymore by the external databases, which could save a significant amount of time in the future.

Also, we recommend to let the experts of the classical music department manually verify the nicknames and non-generic names that could not have been translated perfectly, according to them. By doing so, future translations will be improved and the certainty of the translations is increased. The frequency of such manual verifications is up to Muziekweb; it could be weekly, but also on a monthly basis. It would especially be beneficial to manually verify the cache files once the first 250.000 uniform titles have been translated, before incrementally translating new uniform titles.

11.1.2. Use the Lookup Tables

The system includes several lookup tables containing standard translations for several components, as described in Section 4.2.2. These lookup tables should not be seen as 'static' parts of the system. If one would want to add an item to the lookup tables, we can only encourage this. That is what the lookup tables are for; ease of extension.

11.1.3. Integrate the System

It would be useful to extend Muziekweb's database in such a way to include the translations of the uniform titles to the database and link them to the original ones. We could imagine that accessing the translations via the database is easier for any employee of Muziekweb than via a TSV file.

Furthermore, for ease, it is advisable to implement the system into the current Muziekweb cataloguing environment. One (or more) button can be added to the current user interface where the data of uniform titles is added to the database, which calls the translation system. Or the call to the translation system can even be added to a current button and just be automated. Such a button could, for instance, write the uniform title or a set of new uniform titles to a TSV file, feed it to the translation system, read its output, and store it in the database.

11.1.4. Every Translation Can Count

Perhaps superfluous to mention, but every translation can count in improving the system in at least the speed of translating new uniform titles. Of course, this will only be the case if the uniform title contains a nickname or non-generic name. Even if only one uniform title requires to be translated, eventually just use the translation system. However, we regard this as superfluous, as based on the evaluation results in Chapter 7, we could not imagine that one would ever again want to translate a uniform title again manually.

11.2. Future Development

Based on the evaluation of the final product from Chapter 7, we can safely state that the product really is a 'final' product; it meets at least all major requirements. However, small improvements can always be made. Besides, a product can always be extended. Therefore, we have several recommendations related to improvements and extensions of the final product.

11.2.1. Improvements

An improvement could be related to providing more feedback via the console about the translation status. A counter can be added to show, for instance, the amount of uniform titles that still require to be translated. As this was no hard requirement and was posed close to the end of the project, we have not added such feedback to the console, but we could imagine that it can be useful.

11.2.2. Extensions

From the infinite set of features that one could add to the system, we will recommend how to develop the three most logical ones.

We could imagine that in the future an extra language can be added in order to translate the uniform titles to. In this case, we recommend the future developers to start with adding the language to the language enumeration from the `Utilities` subsystem 4.2.5 and add default translations to the lookup tables for this new language. The external databases can easily be used to translate the uniform titles to the new language as well by making some minor changes, as both external databases support all languages, in principle.

Also, if for some reason the cataloguing standard changes and requires an extra field, we recommend the future developers to take advantage of the hierarchies that have been added. Depending on the field, a new lookup engine, simple translator, or external database translator class can be added for the translation. As the system is very much open for extension, this should relatively be easy. Trivially, the `Splitter` class will require an extra method, which should be an easy change as well. Besides, the new class should implement the `ITranslate` interface, for ease and consistency.

Finally, extra external databases can be added in order to build a more complex certainty assessment algorithm for even providing more certainty about translations found by current external databases or receiving new ones. Again we recommend to use the extensibility of the system related to the certainty assessment; e.g. the new external database adapter should implement the `ITranslate` interface, for ease and consistency.

A

Original Project Description

Note that the following project description is in Dutch.

Dit bachelorproject heeft als doel de database en website van Muziekweb verder te verrijken en ontsluiten. Het project past binnen het streven van Muziekweb om verdere stappen te zetten op het gebied van internationalisering en de koppeling met andere databases. Hier hebben we al een begin mee gemaakt. Zo werkt Muziekweb veel samen met de Nederlandse bibliotheekwereld, en wordt er data geleverd aan Vlaamse bibliotheken. Biografieën van artiesten en componisten zijn deels afkomstig van Wikipedia, en nieuw toegevoegde muziek uit de collectie wordt inmiddels automatisch gekoppeld aan Spotify en Youtube. De eerste stappen zijn dus gezet, maar we willen dit verder gaan uitbreiden. We lopen daarbij tegen diverse problemen waarvoor we een oplossing zoeken.

We willen bijvoorbeeld klassieke titels (metadata) vertalen naar het Engels, Frans en Duits, zodat deze ook internationaal gebruikt kunnen worden. Er is al een handmatige testvertaling van 1000 uniforme titels gemaakt, maar de klassieke collectie is te groot om dit geheel op deze manier te doen. We zouden dus graag de rest van de collectie geautomatiseerd kunnen vertalen. Een ander probleem waarmee we geconfronteerd worden is dat klassieke muziek specifieke regels kent voor metadateren. Dat maakt het lastig oudere cd's geautomatiseerd te koppelen aan streamingdiensten, die veelal andere (eigen) regels hanteren. Misschien kan er een tool ontwikkeld worden waarmee dit makkelijker kan, zodat we ook dit deel van de collectie nog toegankelijker kunnen maken. Het koppelen van relevante titels uit onze collectie met boektitels uit de Bibliotheek.nl-database, en deze vervolgens presenteren op onze eigen website, is een ander streven dat we graag willen realiseren.

(This page has been intentionally left blank.)

B

Project Plan

This document was written in week 1 and informs about the approach that we defined within this project.

B.1. Deadlines

The schedule (see B.1) below provides the self-binding deadlines as well as the global deadlines of the Bachelorproject.

Table B.1: Project schedule containing all deadlines.

Week	Day	Date	Deadlines	Global deadline	Done
4.1	Wed	26-4	Sprint 0		yes
	Thu	27-4	King's Birthday		
	Fri	28-4	Project Plan ('Plan van Aanpak', this document)	yes	yes
4.2	Wed	3-5	Sprint 1		yes
	Thu	4-5	Research Report	yes	yes
	Fri	5-5	Liberation Day		
4.3	Wed	10-5	Sprint 2		yes
4.4	Wed	17-5	Sprint 3		yes
4.5	Th/Fr	25-5	Ascension Day + the day after		
4.6	Wed	31-5	Submit software to SIG (round 1)	yes	yes
	Wed	31-5	Mid-project Meeting (with Cynthia)	yes	yes
	Wed	31-5	Sprint 4 (note: 1½ week sprint!)		yes
4.7	Mon	5-6	Whit Monday		
	Wed	7-6	Sprint 5		yes
4.8	Wed	14-6	Sprint 6		yes
4.9	Fri	23-6	Sprint 7 (note: 1½ week sprint!)		yes
	Sun	25-6	Submit software to SIG (round 2)	yes	yes
	Sun	25-6	Final report sent to committee	yes	yes
4.10	Fri	30-6	Sprint 8 (Prepare presentation)		
4.11	Tue	4-7	BEP presentation	yes	

B.2. User Requirements

The requirements regarding functionality and service are grouped under the Functional Requirements. Within these functional requirements, four categories can be identified using the MoSCoW model[1]for prioritising requirements: Must haves, Should haves, Could haves and Won't haves.

B.2.1. Must haves

- The program must be able to translate uniform titles for classical music pieces.
- The translated form of these uniform titles must maintain the structure of the title.
- The program's input and output must be in the form of a .TSV file.
  - The input file must have three columns: Link (identification number), Uniform title, Readable title.
  - The output file must maintain these columns and have additional columns for the translated uniform titles.
- The program must translate Dutch parts of the uniform titles to the following target languages:
  - English
  - French
  - German
- The program must translate the titles while satisfying the PICA constraints.
- "Nicknames" (i.e. names in quotes) that are not in Dutch, must be left in their original language.
  - This means that the program must recognise which language a piece of text is written in. This is done by consulting external music databases.

B.2.2. Should haves

- The .TSV file should be tab-separated to avoid extra string escaping (since commas, semicolons and quotation marks are also used in the structure of the uniform titles).
- The program should be able to distinguish Dutch parts from Latin parts of titles.
- Latin parts should remain in Latin in any (modern) language.
- The program should have an interactive mode in which ambiguous titles can be translated directly by the user.

B.2.3. Could haves

- The program could be able to read the input via the standard input and write the output to the standard output.
- The system could use a learning algorithm, which can improve previously given translations.

B.2.4. Won't haves

- The program won't be able to translate titles of popular music pieces.
- The program won't be able to translate titles to languages other than the specified target languages.

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

B.3. Non-functional Requirements

Besides the provided functionality and services, design constraints need to be included in the requirements specification as well. These requirements do not indicate what the system should do, but instead indicate the constraints that apply to the system or the development process of the system.

- The program shall be able to run on Windows (7 or higher) and Linux.
- The program shall be implemented in C#.
- For every iteration, the Scrum methodology will be applied.
- The implementation of the program shall have at least 83.5% of meaningful line test coverage (where meaningful means that the tests actually test the functionalities of the program and for example do not just execute the methods involved).
- The results of the translation will also be verified manually, since this cannot be automated.
- The implementation of the program shall have no warnings from static analysis tools. Any warnings that are suppressed, should be well-documented with a proper reason at the place where the warning is suppressed.
- All methods must be documented using XML documentation comments.
- The development team will use a number of tools:
  - Microsoft Visual Studio or JetBrains Rider
  - Git(Hub)
  - Travis CI
  - Waffle
  - Slack
  - Static analysis tool: StyleCop
  - Code coverage tools: Coveralls, more tools TBD
- Tabs in IDEs are done with 4 spaces (‘ ’ or ASCII 0x20).
- Line endings are in Unix format, with a single line-feed character (‘\n’ or ASCII 0x0A).

B.4. Roadmap

Table B.2 represents the roadmap of the project.

Table B.2: Roadmap

High-level Feature	**Sprint**
Research report and project set-up (see Sprint Plan 1)	1
Handle input/output	2
Read and write TSV files	
Parse structure of uniform title	
Simple translation	3
Translate parts of the uniform title that are not “Nicknames” to the target languages	
Le Grande Translate	3,4
Translate “Nicknames” to the target languages using external databases	
Interactive Mode	5
“Nicknames” that are ambiguous can be translated on the fly by the user	
Learning	6
Based on previously made interactive translation decisions, the program could adapt its future decision-making.	
Write report	7
Prepare presentation	8

(This page has been intentionally left blank.)

C

Group Cooperation Contract

This document was written in week 1 and defines the initial rules that we defined for the process of the project.

C.1. Role Allocation

Every week on Wednesday, we will create a sprint plan for the upcoming sprint and a sprint retrospective for the sprint that has passed. In the sprint plans, we will allocate tasks to people. Each task has an associated:

- estimated effort (using the Waffle scale: 1,2,3,5,8,13,20,40,100, trying not to use weights > 8).
- priority (indicated by A-E where A is the highest priority).

In the sprint retrospectives, each team member will indicate per task:

- whether they have finished it
- the actual amount of hours they spent on it
- any extra notes if needed.

There are tasks that do not fit inside a sprint plan and need to be done throughout the entire sprint or in preparation for meetings or during meetings. Types of roles:

- Sprint Planner*	Rob	Pick and assign tasks for upcoming sprint
- Sprint Reviewer	Rob	Responsible for creating the sprint retrospective
- Making meeting notes	Rob	During the meetings, write down what is discussed + resolutions
- Chair of meetings	Mitchell	Guide the conversation during meetings
- Making agenda	Mitchell	Decide which points will be discussed during the meeting
- Static Analyzer	Mitchell	Make sure SAT violations are minimised for each PR
- Testing Overviewer	Mitchell	Document code coverage
- PR Master	Maarten	Make sure that all PRs get reviewed in a proper manner
- Software Architect	Maarten	Document architecture decisions + UML

*) Officially, Casper Karreman is our Scrum Master, but he will not decide which tasks we will do. Therefore, the Sprint Planner can be seen as Scrum Master within our team.

C.2. Decision Making

If there is a decision who not everybody agrees upon, we will first try to reach a consensus, and if that does not work, we will vote. As our group consists of three people, we will always have a majority if we have a dilemma.

C.3. Presence and Availability

Group members are pro tanto expected to be on time: for both the work-days and the meetings. If there is a valid reason for a member to be absent or not on time, then there will not be any (significant) consequences. However, if this is not the case, then the members will have to bring cake. Tasks of an absent member will be reassigned by the other group members, and the absent member will have to compensate for his absence. We will use Slack ('#onderweg' channel) to communicate whenever any of us is late or have to discuss reasons to be absent.

C.4. Meetings and Schedules

Every Wednesday morning, we will have a meeting with the stakeholders of Muziekweb (Casper Karreman and Ingmar Vroomen). During this meeting, we will present our progress and ask for feedback. After this meeting, we will review the previous sprint and set up the next one. We will send these documents also to our TU supervisor (Cynthia Liem) so she can monitor our progress. When needed, we will arrange live or digital meetings with her approximately once every two weeks.

On other days, we will all work from home and communicate through Slack about our progress. On Monday night is the "feature freeze", meaning that no new tasks can be started on Tuesday to make time for final bugfixes and reviewing each other's work. On Tuesday night is the deadline for finishing all tasks of the sprint, so that we have a fully working demo during the meeting on Wednesday.

C.5. Task Completion and Responsibility

This section elaborates on when we, as a team, define a task to be 'completed' or 'done'. Each member is expected to complete his own tasks. The types of the tasks are listed alphabetically.

Each task with a product that needs to be peer reviewed has the following requirements:

- The product has been reviewed by two peers
  - In the case that one team member is not able to complete a review in a timely manner, the PR master may decide that one review is enough
  - Any comments on the product should have been resolved by the person responsible for the task

C.5.1. Bug Fixes

An activity that involves bug fixes is considered as 'done' if

- the bug has been fixed, according to the person that is responsible for the activity
- the absence of the bug involved is ensured by additional tests
- the Pull Request of the bug fix has been peer reviewed before merging

C.5.2. Document Construction

An activity that involves constructing documents is considered as 'done' if

- it at least meets the criteria (in terms of an available template or guidelines) that have been provided for it
- the document has been peer reviewed

C.5.3. Implementing Features

An activity that involves implementation of features is considered as 'done' if

- for this feature, tests have been written based on the requirement before implementing it

- the feature has been implemented
- the feature has been sufficiently tested (at least 75% of meaningful[1] line test coverage)
- the feature has been documented
- if the feature included a change in architecture, the architecture document should have been updated
- the code of the feature has been analysed with static analysis tools
- the Pull Request of the feature has been peer reviewed before merging

C.5.4. Research and Subsequent Decision Making

An activity that involves research is considered as 'done', if

- the results of the research have been fully documented
  - using arguments, conclusions and provision of relevant information and/or citations
- there is one unambiguous decision made by the group in the conclusion of this research
- the research includes at least 2 other methods to solve the problem
  - In case that there is only one alternative, the research is considered as done if this alternative has been documented, including a comparison of the features between the two solutions

C.5.5. Sprints

Sprints, in general, are considered as 'done' if

- the tickets of all features that have been selected before the feature freeze have been closed
  - Tickets can be dropped before the feature freeze if either an unexpected problem arises or there is not enough time to finish them
  - Features that have not been completed have been documented in the Sprint Retrospective
- the retrospective document of the sprint has been completed

[1] Meaningful means that the tests actually test the functionalities of the extension and for example do not just execute the methods involved

(This page has been intentionally left blank.)

D

UML Diagrams

In this appendix, the full UML diagrams can be found. The UML diagram for the `Utilities` subsystem is shown on this page in Figure D.1. The subsystems `Initializer` and `Translate` are shown in Figure D.2. The subsystems `Decide` and `Database` are shown in Figure D.3.

In the UML diagrams, abstract classes or methods have their name in *italics*.

Symbols are used to denote access levels on fields and methods. The following symbols are used: `+` for `public`, `~` for `internal`, `v` for `protected`, and `–` for `private`.

Not all types are fully written out to save space. The following list of abbreviations is used:

- `str` - stands for `string`.
- `List` - can be of type `IEnumerable`, `IReadOnlyList` or `IList`.
- `Dict` - stands for `Dictionary`.
- `Lang` - stands for `Language`, an enumeration in the `Utilities` subsystem.

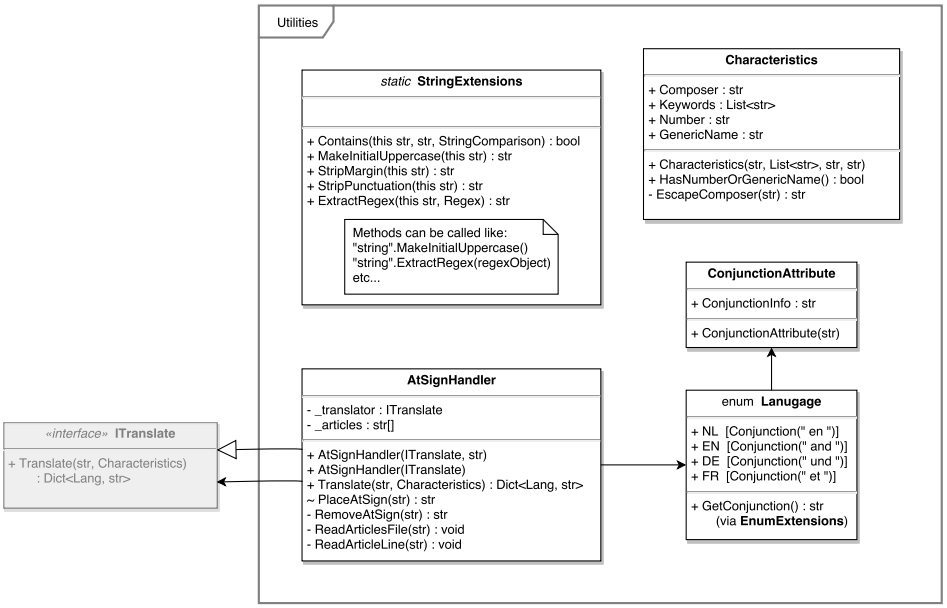

Figure D.1: The full UML diagram, shown for the `Utilities` subsystem.

Figure D.2: The full UML diagram, shown for the `Initializer` and `Translate` subsystems.

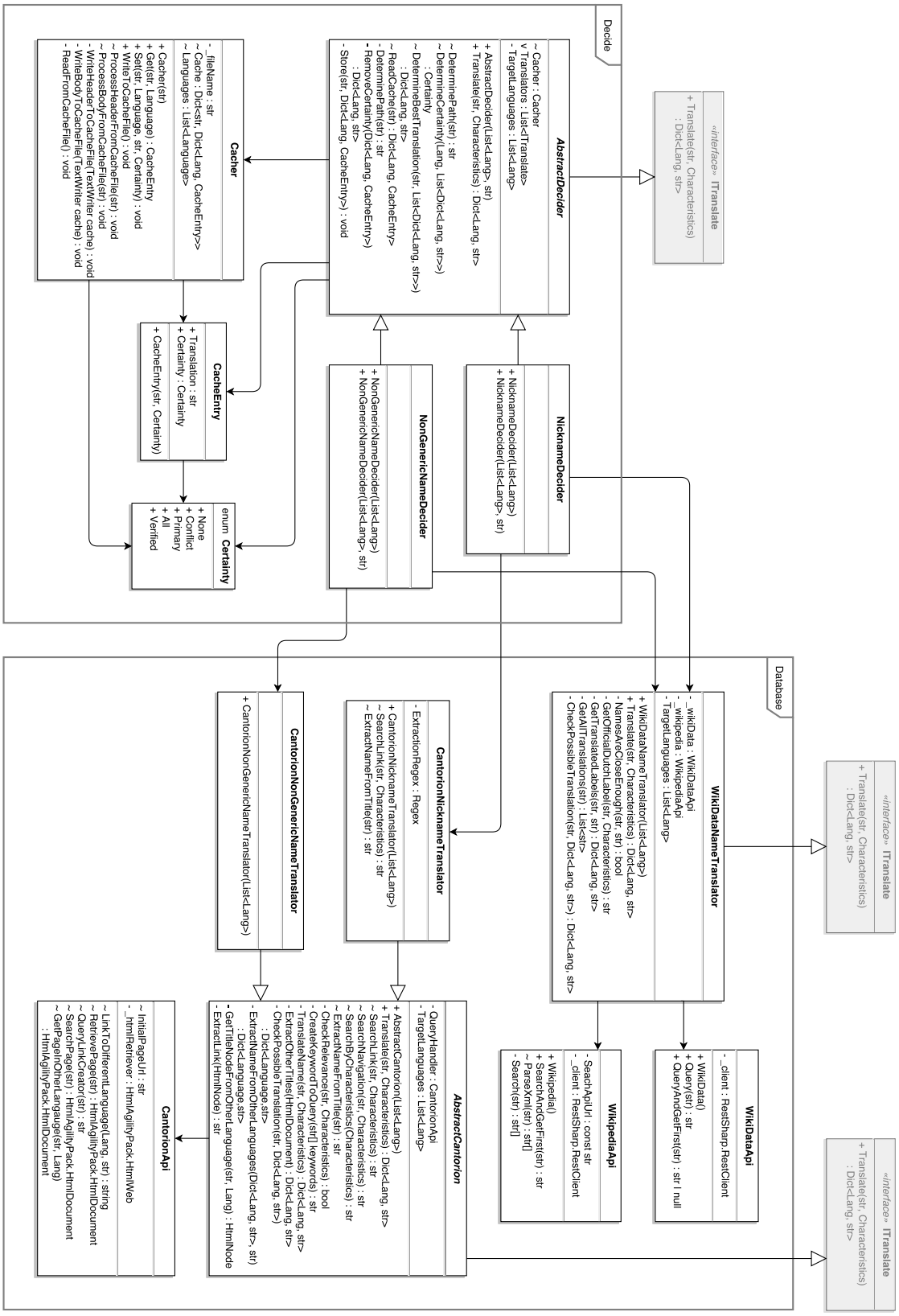

Figure D.3: The full UML diagram, shown for the `Decide` and `Database` subsystems.

(This page has been intentionally left blank.)

E

Test Report

This appendix elaborates on the latest status of the source code in terms of testing. Parts of the code that are excluded from coverage, have their reasons documented here. Also, parts of the code that cannot be covered are discussed. All classes that are not included in the discussion, are fully covered. In addition to this, we would like to suggest the reader to analyse the coverage report as well. In order to generate a coverage report on your local machine, perform the necessary steps described in the file 'README.md'. Consider this document as a supplement to the coverage report. The Codecov coverage changes can be found here: `https://codecov.io/gh/mpsijm/CDRStage2017-Translate`. At the end of the report, an overview has been added (see Figure E.1).

The test coverage of the source code is 99.90% at the end of sprint 7. This amount is sufficient, since we want to have at least 83.5% meaningful line coverage. We have documented which classes we can not (fully) test in a list below.

E.1. List of Not Fully Covered Methods

- Database: `AbstractCantorion`.
  - 99.06%
  - `ExtractNameFromOtherLanguages`
    - The method contains a null-check branch that cannot be covered, since it requires a search result Cantorion page on which there is no title (meta-data) node available. This check has been added to make sure that this case can be handled and prevent a nullpointer. Since we have not yet encountered such a page, we cannot come up with a test case to trigger this branch.

E.2. List of Classes Containing Coverage Exclusions

- Initializer: `Program`
  - The methods ShowHelp and ShowOptionError cannot be tested, since all they do is write messages to the standard error stream.
- Initializer: `FileHandler`
  - The protected method Dispose(bool) cannot be fully tested, some lines are only partially hit. One of these lines would only be hit if this method were called by the garbage collector. The other two lines can only be hit if some of the fields of FileHandler are both null before disposal, which can never happen.

For an overview of the code coverage, please see the next page. Note that this overview is directly extracted from Codecov.io.

E.3. Overview

The following overview (see Figure E.1) is extracted from the coverage report at the end of sprint 7. The percentages represent the line coverage per item.

Files					Coverage
Translate/Database/AbstractCantorion.cs	107	106	1	0	99.06%
Translate/Database/CantorionApi.cs	19	19	0	0	100.00%
Translate/Database/CantorionNicknameTranslator.cs	15	15	0	0	100.00%
Translate/Database/CantorionNonGenericNameTranslator.cs	3	3	0	0	100.00%
Translate/Database/WikiDataApi.cs	18	18	0	0	100.00%
Translate/Database/WikiDataNameTranslator.cs	62	62	0	0	100.00%
Translate/Database/WikipediaApi.cs	22	22	0	0	100.00%
Translate/Decide/AbstractDecider.cs	70	70	0	0	100.00%
Translate/Decide/CacheEntry.cs	5	5	0	0	100.00%
Translate/Decide/Cacher.cs	95	95	0	0	100.00%
Translate/Decide/NicknameDecider.cs	8	8	0	0	100.00%
Translate/Decide/NonGenericNameDecider.cs	8	8	0	0	100.00%
Translate/Initializer/EntryPoint.cs	3	3	0	0	100.00%
Translate/Initializer/FileHandler.cs	73	73	0	0	100.00%
Translate/Initializer/Program.cs	58	58	0	0	100.00%
Translate/Translate/AbstractLookupEngine.cs	104	104	0	0	100.00%
Translate/Translate/AbstractTranslator.cs	23	23	0	0	100.00%
Translate/Translate/GenericNameEngine.cs	20	20	0	0	100.00%
Translate/Translate/InstrumentEngine.cs	16	16	0	0	100.00%
Translate/Translate/KeyTranslator.cs	53	53	0	0	100.00%
Translate/Translate/NumberEngine.cs	14	14	0	0	100.00%
Translate/Translate/PartInfoEngine.cs	22	22	0	0	100.00%
Translate/Translate/Splitter.cs	80	80	0	0	100.00%
Translate/Utilities/AtSignHandler.cs	54	54	0	0	100.00%
Translate/Utilities/Characteristics.cs	16	16	0	0	100.00%
Translate/Utilities/ConjunctionAttribute.cs	5	5	0	0	100.00%
Translate/Utilities/EnumExtensions.cs	13	13	0	0	100.00%
Translate/Utilities/StringExtensions.cs	46	46	0	0	100.00%
Project Totals (28 files)	1,032	1,031	1	0	99.90%

Figure E.1: Code coverage overview of the last sprint.

F

Software Improvement Group Evaluation

This Appendix contains the two evaluations received by the SIG on the midterm submission (Section F.1) and the final submission (Section F.2). Note that these reviews are in Dutch.

After we received the evaluation on the midterm submission, we tried to take their feedback into account. How this was done, is mentioned in Section F.1.1.

F.1. Evaluation of Midterm Submission

De code van het systeem scoort 4,5 ster op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de 'AbstractLookupEngine.PreprocessItemNames'-methode, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld 'Decides whether translations are needed in only a singular form, plural form or the only available form' zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

F.1.1. How This Feedback Was Resolved

After receiving the feedback that our software project contained lengthy methods, we started hunting the code base for methods that were either long by itself (we decided to check methods longer than twenty lines) or contained behaviour-defining comments inside the method.

The following methods were split up or refactored:

- `AbstractCantorion.ExtractNameFromOtherLanguages` - refactored, split up into three methods and added protected method `ExtractNameFromTitle` that is overridden in the `CantorionNicknameTranslator`
- `AbstractCantorion.SearchNavigation` - refactored
- `AbstractLookupEngine.PreprocessLookupTable` - split up into two methods
- `AtSignHandler.ReadArticlesFile` - split up into two methods
- `Cacher.WriteToCacheFile` and `Cacher.ReadFromCacheFile` - both methods are split up into three methods

- `FileHandler.ParseFile` - split up into five methods
- `KeyTranslator.ToEnglish` and `KeyTranslator.ToFrench` - extracted field `FlatNoteRegex`
- `Program.ParseArgs` - refactored, ending up with an extra method and an extra field
- `Splitter.Split` - refactored and split up into two methods

F.2. Evaluation of Final Submission

In de tweede upload zien we dat zowel de omvang van het systeem als de score voor onderhoudbaarheid is gestegen. De stijging is veroorzaakt door een verbetering op het gebied van Unit Size, dat in de feedback op de eerste upload nog als verbeterpunt werd genoemd.

Ook is het goed om te zien dat jullie naast nieuwe productiecode ook aandacht hebben besteed aan het schrijven van nieuwe testcode.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject.

G

Project Evaluations

In order to provide a more personal overview of how the individual team members and the client have experienced the project, this appendix contains evaluations of both the individual team members and the client.

G.1. Evaluations by the Individual Team Members

G.1.1. Mitchell Dingjan

Aside from my role as a team member of the development team, I also covered the following four roles: Chair of meetings, agenda maker, static analyser, and testing overviewer. In advance of every meeting (both meetings with the client and the coach) I created an agenda. After a quick peer review, I shared the agenda, in advance, on Slack (the communication tool that we used to, among other things, communicate with the client and coach). I also chaired these meetings, which was new for me, as I had not really chaired any meetings before over such a period of time on a regular basis. More related to the code, for every pull request at least I made sure that the changes did no cause any violations (and hints) based on our static analysis tools FxCop, StyleCop, and a bit later during the project also Rider. Furthermore, I created the initial test report (see Appendix E), weekly updated it, and kept an eye on whether the changes by pull requests were well tested.

I think that I have taken my responsibility in each of these roles. Every agenda has been created, peer reviewed, and shared (well) on time. I also think that these agendas were well formatted and covered at least most things that we wanted to discuss and present. During the meetings, I have tried to let everyone talk about their progress and reserve time for feedback moments and question rounds. Thus, I think that my goal of making the meetings a place for sharing progress and ideas has been accomplished. Also, never a pull request has been merged that contained a violation. If there has been a violation in the development branch, then this was caused because someone too soon pushed something to the development branch. However, this happened rarely and if it happened, the violation was resolved shortly after. Furthermore, every test report, I think, has been well formatted and is clear. I experienced these reports useful as a baseline for keeping an overview of the test status. There also rarely was the occasion in which a pull request was merged and the code coverage was decreased; if this was the case, the code coverage decrease was less than three percent.

In my opinion, I have taken my role as a team member seriously. I communicated well with the others about the status of my features and provided feedback on the questions and issues that they posed. I have completed most of the issues that I worked on (well) in time and those that were not required extra time because they turned out to be a bit more complex than we initially thought. Besides, for every pull request, I have also met the requirements of the Definition of Done that is stated in the Group Cooperation Contract.

Two features I have mainly worked on are the Lookup Engines and the nickname and non-generic name translator based on Cantorion. I enjoyed working on the Lookup Engines, as their introductions directly showed progress in the translations. Not only did they directly

give a purpose, but they were also quite challenging to develop, as the fields that use the lookup engines often differ in the way they are formatted. The number of instruments are added to each instrument via '[X]', where X represents a natural number greater than 2), for instance, which is not applicable to the others. Therefore, the challenge was to put most functionality in the abstract Lookup Engine class in an elegant way, which we, in my opinion, definitely succeeded in. As I posed the idea to use Cantorion as a verifying second external database, I decided to do the implementation. I found it interesting to design search and lookup algorithms for the translating the nicknames and non-generic names. After a peer review on the design, I implemented these algorithms. I am happy with the way things worked out for this external database; using Cantorion achieved the goal of having a verifying extra database to include in the certainty assessment.

When reflecting on the cooperation of the development team, I think we worked really hard, communicated well, and enjoyed working together. If someone had a question or was temporarily stuck on an issue, then the others would provide feedback and help this team member out. Our cooperation with the client went smoothly as well; the employees at Muziekweb were very open and kind. If we had a question, we could always send a mail or, if we were at Muziekweb, walk up to them. Our cooperation with the coach went smooth as well and was pretty simple; we shared our progress and ideas on a regular basis and received useful feedback. Also, whenever we had a question we could always contact her.

The final product definitely meets my expectations. It meets the requirements and turned out to be something which, I think, we can be proud of, as it is very well designed from a software engineering perspective and meets the design goals. Also, I like the fact that the system is well tested on different levels.

As a conclusion, I think that the project has been a success. One of the most important things that I have learned of this project is to be able of working in a development team with an actual real-world client. Also, I learned that in a well-cooperating team of three developers you can achieve quite a lot in nine weeks.

G.1.2. Rob Kapel

My role during this project, besides developing, can be split up in three smaller roles: Sprint Planner, Sprint Reviewer and making meeting notes. As a Sprint Planner, I was responsible for picking and assigning tasks for the upcoming sprint. If we somehow were not able to come to a consensus, I was the person who had the final say in it. Luckily, discussions on deciding what to do during the next sprint, ran pretty smoothly. Therefore, this ”power” did not have to be used. This made this role quite easy, only responsible for making the sprint available on the Git repository.

As the Sprint Reviewer, I was responsible for creating the sprint retrospective. Afterwards, I had to make sure this document was also uploaded on the Git repository. However, since we usually worked on the retrospective together, it was easily discussed who would upload it once we finished it. Therefore, the retrospectives were consistently uploaded online, making this role successfully executed.

As a third, I had the role of making notes during the meetings. Everything useful that was said during these meetings, was documented, highlighting action points that needed to be done. Since I needed to explain features during meetings as well now and then, Maarten took over this role during these parts. At the end of the meetings, I was also responsible for adding these action points to our todo list. Since these tasks were highlighted during the meeting as well, this was easily done. Therefore, this role went relatively smoothly. Only part I had trouble with, when a lot of important things were said in a row, I had trouble keeping up with making notes. Over the weeks though, I improved on this aspect, being able to write notes more quickly, coming back to them later when needed. Additionally, it helped that Maarten also assisted in writing these notes, ensuring nothing was missed.

In the end, I feel like I have fulfilled my role in this project. I noticed though that I am not as skilled of a programmer as my fellow developers. Some problems I got stuck on for a while, my teammates easily suggested a solution for them. This made me focus a lot on an issue, with sometimes neglecting other tasks that had to be done. However, even though I might

have not been able to implement solutions as smoothly as others, I feel like my contributions to the project have been successful for the final product with some extra effort put in. I made sure all the contributions adhered to the Definition of Done, so it was ensured no extra effort had to be put in to finish it up.

Two of the main components I worked on were the `Splitter` and the `Decide` subsystem. I enjoyed working on the `Splitter` a lot. It involved utilizing a lot of regular expressions, which I recently learned about. I was excited solving a problem with this recently acquired knowledge. One part of this which did cause some troubles, was that it involved a lot of corner cases. Since the type of symbols that can appear in a title were more diverse than initially thought, it took multiple sprints to find them all. Even though this took a while to get it right, it gave a lot of satisfaction covering these.

Besides that, I also worked on the `Decide` subsystem. I enjoyed finding an optimal solution for combining the two databases, figuring out which database contained more correct data, and how it could be combined optimally. Even though Cantorion did not prove as useful to fully rely on, I still enjoy the solution of it having it change how sure the translation is.

Communication went pretty well overall in my opinion. Everyone tried to keep the rest of the development team what they are working on. This was useful to see how everyone was doing. I've noticed that I have not done this all the time. When being stuck on an issue, I tended to not completely inform the others, not admitting that my part was not done yet. I know that this is a flaw of mine, which I do have improved on during the course of the project. In the end, when I found an issue I had trouble with, I made sure I notified it, so it could be solved as quickly as possible.

Besides communication among the team, there was also communication with the people at Muziekweb. I think communication with them went very well. Whenever an issue or a suggestion was found by them, they would let us know in either the weekly meeting, or through Slack. This kept us up to date on the needs of Muziekweb during the project. Also, any questions we had for them, were quickly answered, either in the meetings or on Slack if we were not in the office.

Finally, we also had communication with our coach at the TU Delft. However, since we did not have many problems relating to the setup of the project, we have only had a meeting with the coach every two weeks, either online through Skype or in person. Whenever we had questions, they were quickly answered through the Slack channel. Therefore, it did not raise any problems.

I am satisfied with the final product we have delivered. I feel like it will solve the problem that Muziekweb provided us with. During the meetings, I got the feeling they also seemed very enthusiastic whenever we showed new features of the program. I hope this is also a good indication for the final product as well. Even though the final product does not have all the requirements we set up at the start of the product, I am still satisfied with the product, delivering the best we could have done in this amount of time. The quality of the code we delivered has been maintained properly, so that it will be easy for Muziekweb to implement it into their systems, but also extend it wherever needed. I will look forward to see the results of our program on the website as well, once it has been integrated.

In the end, I feel this project has been a great way to finish of the Bachelor programme. Not only did it give experience working with a real-world client, it also helped me improve on some personal aspects as well. I think we worked quite well together as a team, and I hope we can work together again in the future.

G.1.3. Maarten Sijm

My role during this project consisted of two parts: Pull Request (PR) Master and Software Architect.

As the PR Master, I kept track of all PRs that were opened and made sure that they were peer reviewed as soon as possible, to avoid the piling up of PRs at the end of the sprint.

Another reason to have PRs merged as soon as possible, is that some features might have dependencies on one or more PRs that were still open. In my opinion, most PRs were reviews and merged in a timely manner.

As the Software Architect, I kept an eye on the architecture of the entire project. In order to do that, I built the UML diagrams and maintained the architecture design document. While documenting the cohesion between the separate components, there were moments that I thought: "But wait a second, this is not logic at all," after which I would update these components to make them fit better into the big picture. In the final sprint, I visited all components in our software product and refactored them wherever possible, leaving the final product (in my eyes) with a well-designed software architecture.

During the project, I worked on two main features: the `Initializer` subsystem and the name translator based on WikiData.

I really enjoyed working on the `Initializer`. Because I am a huge fan of the command-line, I already had some experience with initializing a CLA. It took little effort to find out how this was done in C# and I am happy with the result.

The `WikiDataNameTranslator` was my second feature. I find it interesting how WikiData (and DBpedia) take the data that is already available on Wikipedia and put it in a machine-readable format. During the research phase, it took some time to find out how to structure a query in such a way that the translations for any Dutch page can be retrieved. In the end, the way we query Wikipedia and WikiData is sufficient and it works to translate the names of classical music pieces.

In my eyes, the cooperation with the rest of the team went smoothly. On Slack, I tried to give updates for whatever I was doing, whenever I was stuck or when I needed feedback from my peers. The Definition of Done that we set up in the first week of the project helped me to finish my tasks in a structured manner, and I always tried to keep myself to it. The sprint plans were a good way of communicating what we would do every week, and during the week it was nice to see feature cards moving more and more to the right on our Waffle board, indicating progress for the sprint.

The cooperation with the employees at Muziekweb went very smoothly as well. We could always come with questions whenever something was not fully clear to us. I am glad that we put some of them in the Slack group as well, so we could also ask questions when we were working from home.

The communication with our coach was sporadic, because we did not have many problems during our project. Once every two weeks, we had a meeting with her, sometimes via Skype, and this was sufficient. Any questions during the project about deliverables could be asked via Slack as well.

I am content about how we delivered the final product. It contained all the features that Muziekweb wanted, it was tested well, and the design of the software was clear. Not all of the initial requirements were met in the final product, but I am fine with that, since this was agreed upon with the client. Some requirements were changed into others, that fit more into how Muziekweb would use the final product, and some were discarded because some things worked out differently than planned. The weekly meetings helped to indicate what we have been working on and to ask for feedback from our client. During these meetings, we agreed on which features we would build during the next sprint and which requirements could be discarded.

My overall opinion about this project is that it has been completed successfully. It was a good experience to work with a real-life client as an integrated part of their team. It really makes me eager to work as a programmer for a small company like Muziekweb in the future.

G.2. Evaluations by the Client

G.2.1. Casper Karreman

Job title: Developer
Company: Muziekweb / Centrale Discotheek Rotterdam

With 13 years of experience as developer at Muziekweb I have been assigned to the BEP translate project as technical supervisor. With other colleagues we have decided the boundaries and initial system requirements for the project.

Formally I have been scrum master for the development team, but only to be able to decide which features should be built in the system. I have not interfered in the planning, the workload decisions or the system design. However, I have given some feedback on the implementation of the code.

My Opinion on the Process

Communication

Mitchell, Maarten and Rob have communicated well in the process of analysing the problem. They have contacted the right persons for information when needed and were able to acquire the information needed to decompose the problem.

In the process of designing the solution they were able to decide together and inform each other well on personal thoughts. Also they have a positive attitude when giving or receiving feedback.

Meetings

When it comes to the meetings I can say that they were always well prepared. They kept us up to date on the progress of the project, the work they've done and the steps to take in the next sprint.

At the end of the meetings there was a review on the created documents or a demonstration of the implemented features. This worked really well for me as developer but also for other non-technical staff.

Working Method

The working method the team set up has been structured and explanatory. From the steps of analysing the problem to solving it and implementing the theoretical solution.

Specifically on implementation:

The team has set up their own working environment to be able to have an automated test driven environment. This setup took some time before it was up and running as an automated system but in the end they managed to get all things working.

The Final Product

The team has delivered a stable working final product. The delivered product has an adaptable and extendable design so it can be used for the initial bulk translations and we can integrate the code in our own software products.

The code is refactored to be not too abstract or complex. This makes it manageable for us in the future.

Would you like more cooperation between TU Delft and Muziekweb?

Based on the positive experience on this project I would like to see more cooperation with TU Delft. There is satisfaction on the delivered quality but also the fresh ideas and the demonstration of the working process were pleasant.

G.2.2. Ingmar Vroomen

Function: Project manager
Company: Muziekweb / Centrale Discotheek Rotterdam

Role in the project: In this project my function was that of client. I wrote the initial project description for the project. Since I have no technical or programming experience, I could only keep an eye of the project as a whole.

On the Process

Maarten, Mitchell and Rob quickly recognised our questions and needs, and although the topic (classical music) was perhaps new for them or a bit outside their area of interest, they made it their own. They worked in a very structured way. They made a detailed planning for every week, kept track of their activities, results and findings and presented their progress in weekly meetings in a clear way. They handled our feedback very well, and also gave useful feedback for our database when they discovered mistakes. Also important: they were nice colleagues to have around for 10-11 weeks.

On the Final Product

The end product looks promising. It meets our requirements, and the way it's build will allow us to implement the design in our own system and to adapt when needed. It will help Muziekweb to improve our database and website. Like their other schedules and reports, Maarten's, Mitchell's and Rob's final report is very structured.

Will Muziekweb work again with TU Delft?

This was the first time Muziekweb participated with the TU Delft in a BEP, and it proved to be a success. Not only do we now have a program that can actually contribute to our system, but I think Maarten, Mitchell and Rob also gave us new ideas and insights, for example on how to set up a project or on linking to external databases. Hopefully, Muziekweb and TU Delft can collaborate in more 'BEPs' in the future.

Appendix H Info Sheet

Title of the project:
Muziekweb: database- en websiteverrijking van de grootste muziekcollectie van Nederland
(Database and website enrichment of the biggest music collection of the Netherlands)
Name of the client organisation: Muziekweb / Centrale Discotheek Rotterdam
Date of the final presentation: 4 July 2017

Description

Muziekweb maintains a catalogue containing all music titles ever published in the Netherlands. The challenge was to translate the classical music titles from Dutch to English, French, and German. These titles have a strict format that should be adhered to during translation. Also, their common names can be very different from one language to another, so literal translation is not possible.

During the research phase, the students learned about the cataloguing standard used by Muziekweb. The plan was to translate the titles part by part. Some parts were trivial to translate, others required making a connection to external databases.

The process was done using the Scrum methodology. During weekly meetings, the client was informed about the progress and was able to give feedback on the current state of the product. One initial requirement (an interactive mode to request feedback from an expert during translation) was entirely discarded during one of the meetings, because it turned out not to fit in how the final product would be used. It was replaced by similar feature, that allows for feedback after the translation is finished.

The final product was a command-line application that accepts an input file containing titles and writes the translations of these titles to an output file. Using 1000 titles that were already manually translated by Muziekweb, we could track our progress in terms of the amount of correctly translated titles by the program. Also, unit tests were made to test the individual components of the software product.

As an outlook to the future, the client will be using this final product to translate their catalogue of classical music titles. The client was advised to keep using this software program even for one title at a time, making use of the built-in feedback system, since the program can improve its results from the feedback that it has had.

Members of the project team

All team members contributed to the final report and the final presentation.

Mitchell Dingjan
Interests: Multimedia analysis, programming, and applied mathematics
Contributions: Testing, translations via lookup tables, one database connection

Rob Kapel
Interests: Algorithmics, multimedia analysis, and graphical design
Contributions: Splitting up titles, decision algorithm, connecting program components

Maarten Sijm
Interests: Programming, automation, and organising structures
Contributions: Software architecture, command-line interface, one database connection

Client	Ingmar Vroomen	ingmar@muziekweb.nl	Muziekweb	(contact person)
Client	Casper Karreman	ckarreman@muziekweb.nl	Muziekweb	
Coach	Cynthia Liem	c.c.s.liem@tudelft.nl	Intelligent Systems	Multimedia Computing Group

The final report for this project can be found at: `http://repository.tudelft.nl`

(This page has been intentionally left blank.)

Glossary

Bash Unix shell and command language 28

decorator pattern design pattern, which allows behaviour to be added to another object with the same base type, without changing the underlying implementation of this other object 23, 25

`Dictionary` data structure in C# (comparable to `HashMap` in Java) allowing the creation of a key-value structure, comparable to a real-life dictionary 20–23, 59

generic name composition type of a classical music piece 4–6, 22, 26

generic title uniform title that starts with a generic name 4, 5, 25, 40

non-generic name nickname of a classical music piece for which it is commonly known 4, 5, 20, 23–25, 38, 40, 41, 47, 48

non-generic title uniform title that does not contain a composition of the orchestra field ('/2') 4, 5, 25, 40

uniform title catalogued classical music piece 1, 3–7, 19–21, 23, 26, 27, 29–31, 34, 40, 43, 45, 47, 48

(This page has been intentionally left blank.)

Acronyms

API Application Programming Interface. 24, 25, 41

CDR Centrale Discotheek Rotterdam. 1

CI Continuous Integration. 35

CLA Command-Line Application. 21, 70

HTML HyperText Markup Language. 25

IDE Integrated Development Environment. 7, 34

OCLC Online Computer Library Center. 4

PR Pull Request. 69, 70

SIG Software Improvement Group. 2, 38, 65

SPARQL SPARQL Protocol and RDF Query Language. 25, 45

TDD Test-Driven Development. 33, 34

TSV Tab-Separated Values. 6, 7, 22, 40, 45, 47

UML Unified Modelling Language. 2, 19–25, 59–61, 70

(This page has been intentionally left blank.)

Bibliography

[1] AllMusic. An der schönen, blauen Donau | Details, 2017. URL http://www.allmusic.com/composition/mc0002395606. [Online; accessed 2 May 2017].

[2] Laura Auria and Rouslan A Moro. Support vector machines (SVM) as a technique for solvency analysis. *Information Systems Frontiers*, pages 1–12, 2008. doi: 10.1007/s10796-016-9689-z.

[3] British Library. Changing the record - A concise guide to the differences between the UKMARC and MARC 21 bibliographic formats, 2002. URL https://www.bl.uk/bibliographic/pdfs/marcchange.pdf. [Online; accessed 2 May 2017].

[4] Peter F. Brown, John Cocke, Stephen A. Della Pietra, Vincent J. Della Pietra, Fredrick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. A Statistical Approach to Machine Translation. *Comput. Linguist.*, 16(2):79–85, June 1990. ISSN 0891-2017. URL http://dl.acm.org/citation.cfm?id=92858.92860.

[5] Buildkite. Buildkite, 2017. URL https://buildkite.com/#learn-more. [Online; accessed 23 June 2017].

[6] S. Chand. Empirical survey of machine translation tools. In *2016 Second International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, pages 181–185, Sept 2016. doi: 10.1109/ICRCICN.2016.7813653.

[7] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *CoRR*, abs/1409.1259, 2014. URL http://arxiv.org/abs/1409.1259.

[8] Codecov. Codecov, 2017. URL https://codecov.io/. [Online; accessed 20 June 2017].

[9] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20 (3):273–297, 1995.

[10] Coveralls. Coveralls, 2017. URL https://coveralls.io/. [Online; accessed 20 June 2017].

[11] DBpedia. The Blue Danube, 2017. URL http://dbpedia.org/page/The_Blue_Danube. [Online; accessed 2 May 2017].

[12] Bala Deshpande. 4 key advantages of using decision trees for predictive analytics, 2011. URL http://www.simafore.com/blog/bid/62333/4-key-advantages-of-using-decision-trees-for-predictive-analytics. [Online; accessed 5 May 2017].

[13] Discogs. Explore Classical on Discogs, 2017. URL https://www.discogs.com/search/?genre_exact=Classical. [Online; accessed 2 May 2017].

[14] FxCop. Fxcop, 2017. URL https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx. [Online; accessed 20 June 2017].

[15] Massimo Gentili-Tedeschi. Music Presentation Format: Toward a Cataloging Babel? *Cataloging & Classification Quarterly*, 53(3-4):399–413, 2015.

[16] Hoorn. Instrumenten vertaald in het Nederlands, Italiaans, Duits, Engels en Frans, 2008. URL http://www.hoorn.be/vertalingen.htm. [Online; accessed 2 May 2017].

[17] William John Hutchins and Harold L Somers. *An introduction to machine translation*, volume 362. Academic Press London, 1992. URL http://www.hutchinsweb.me.uk/IntroMT-TOC.htm.

[18] Joint Steering Committee for Development of RDA. RDA — Resource Description and Access - A Prospectus, 2009. URL http://www.rda-jsc.org/archivedsite/rdaprospectus.html. [Online; accessed 5 May 2017].

[19] Library of Congress. MARC-8 Encoding Environment, 2007. URL http://www.loc.gov/marc/specifications/speccharmarc8.html. [Online; accessed 2 May 2017].

[20] Library of Congress. About LOC, 2012. URL http://www.loc.gov/catdir/cpso/news_rda_implementation_date.html. [Online; accessed 2 May 2017].

[21] Library of Congress. Overview of the BIBFRAME 2.0 Model, 2016. URL https://www.loc.gov/bibframe/docs/bibframe2-model.html. [Online; accessed 2 May 2017].

[22] Eric Miller, Uche Ogbuji, Victoria Mueller, and Kathy MacDougall. Bibliographic framework as a web of data: Linked data model and supporting services. In *Washington, DC: Library of Congress*, 2012.

[23] Casey Mullin. Best Practices for Music Cataloging: Using RDA and MARC21, 2014.

[24] MusicBrainz. Work “An der schönen blauen Donau, op. 314” – Aliases, 2017. URL https://musicbrainz.org/work/6763c20d-f2df-3347-8a0e-7abad42a5a56/aliases. [Online; accessed 2 May 2017].

[25] MuziekWeb. Muziekweb in de bibliotheek, 2017. URL https://www.muziekweb.nl/Muziekweb/Luister. [Online; accessed 5 May 2017].

[26] Network Development and MARC Standards Office. Marc2Bibframe, 2017. URL https://github.com/lcnetdev/marc2bibframe/issues. [Online; accessed 2 May 2017].

[27] NuGet. NuGet, 2017. URL https://www.nuget.org/. [Online; accessed 23 June 2017].

[28] OCLC. Gecodeerde informatie over de instrumentale bezetting, 2006. URL http://support.oclc.org/ggc/richtlijnen/php/showPresentation.php?id=12&ln=nl&sec=k-1103#instrumenten_en_groepen. [Online; accessed 2 May 2017].

[29] OCLC. Non-MARC Information, 2017. URL http://experimental.worldcat.org/marcusage/887.html. [Online; accessed 2 May 2017].

[30] OCLC. What is WorldCat?, 2017. URL https://www.worldcat.org/whatis. [Online; accessed 2 May 2017].

[31] OpenCover. Opencover, 2017. URL https://github.com/opencover/opencover/. [Online; accessed 20 June 2017].

[32] Sunil Ray. Essentials of Machine Learning Algorithms (with Python and R Codes), 2015. URL https://www.analyticsvidhya.com/blog/2015/08/common-machine-learning-algorithms/. [Online; accessed 5 May 2017].

[33] RDA Steering Committee. About RDA, 2017. URL http://rda-rsc.org/content/about-rda. [Online; accessed 2 May 2017].

[34] RDA Steering Committee. About RDA Toolkit, 2017. URL http://www.rdatoolkit.org/node/44. [Online; accessed 2 May 2017].

[35] RDA Steering Committee. Rda value vocabularies, 2017. URL http://www.rdaregistry.info/termList/. [Online; accessed 2 May 2017].

[36] RDA Steering Committee. Rda in Translation, 2017. URL http://www.rdatoolkit.org/translation. [Online; accessed 2 May 2017].

[37] RDA Steering Committee. Rda Toolkit Release - april 11, 2017, 2017. URL http://www.rdatoolkit.org/april2017release. [Online; accessed 2 May 2017].

[38] ReSharper. Resharper, 2017. URL https://www.jetbrains.com/resharper/. [Online; accessed 23 June 2017].

[39] Christopher Roach. Building Decision Trees in Python, 2006. URL http://www.onlamp.com/pub/a/python/2006/02/09/ai_decision_trees.html. [Online; accessed 5 May 2017].

[40] Neil Rubens, Dain Kaplan, and Masashi Sugiyama. Active learning in recommender systems. In *Recommender systems handbook*, pages 735–767. Springer, 2011.

[41] SAS Institute. Machine Learning: What it is and why it matters, 2017. URL https://www.sas.com/en_us/insights/analytics/machine-learning.html. [Online; accessed 5 May 2017].

[42] Amanda Spink, Jung-ran Park, and Lynne C Howarth. *New directions in information organization*. Emerald Group Publishing, 2013.

[43] Stackoverflow. XUnit or NUnit, 2017. URL https://stackoverflow.com/questions/3273315/xunit-or-nunit-what-advantages-and-disadvantages-of-each-other. [Online; accessed 23 June 2017].

[44] StyleCop. Stylecop, 2017. URL https://github.com/StyleCop/StyleCop. [Online; accessed 20 June 2017].

[45] Waffle. Waffle, 2017. URL https://waffle.io/. [Online; accessed 20 June 2017].

[46] Wikipedia. List of Italian musical terms used in English — Wikipedia, The Free Encyclopedia, 2014. URL https://en.wikipedia.org/wiki/List_of_Italian_musical_terms_used_in_English. [Online; accessed 2 May 2017].

[47] Wikipedia. The Blue Danube — Wikipedia, The Free Encyclopedia, 2017. URL https://en.wikipedia.org/wiki/The_Blue_Danube. [Online; accessed 1 May 2017].

[48] Wikipedia. Toonsoort — Wikipedia, De Vrije Encyclopedie, 2017. URL https://nl.wikipedia.org/wiki/Toonsoort. [Online; accessed 4 May 2017].

[49] Yale University Library. The Names of Instruments and Voices in English, French, German, Italian, Russian, and Spanish, 2014. URL http://www.library.yale.edu/cataloging/music/instname.htm. [Online; accessed 2 May 2017].

[50] Yale University Library. The names of keys in English, French, German, Italian, and Spanish, 2015. URL https://www.library.yale.edu/cataloging/music/keylang.htm. [Online; accessed 4 May 2017].