

Explaining Source Code via Natural Language Generation

Vladimir Yesipov

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Bachelor of Science
of the
University of Aberdeen.



Department of Computing Science

2019

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed: Vladimir Yesipov

Date: 2019

Abstract

Research has shown that most effort of today's software development is maintenance and evolution. Developers often use integrated development environments, debuggers, and tools for code search, testing, and program understanding to reduce the tedious tasks. One way to make software development more efficient is to support program comprehension. However, source code is not always the most suitable way to explain computations. Although inline documentation is added to source code to aid program comprehension, such documentation could not be guaranteed to truly reflect what the source code actually does. There have been many approaches proposed and developed for this purpose many of which make graphical representations of programs. Among these approaches, descriptive text generation for source code along with the source code snippets is gaining more and more attention and has become a popular research area. In this project, this problem is researched and a system generating a descriptive text along with the code snippets proposed, the challenges and future opportunities are discussed.

Creating textual documentation directly from source code is a hard problem because the meaning of the source code, which is essentially dynamic computations, creation and management of structured data, is not a direct property of the source code. Instead, it is generated from source code by a compiler or interpreter. Moreover, the meaning attributed by a compiler or interpreter is computational in nature and needs to be mapped into a semantic representation expressible in natural language.

The ability to automatically analyse source code, explore the code elements (e.g. methods, variables and statements) and build natural language description of this code fragments has been a longstanding goal in computer science. In addition, finding a way to adequately summarise, describe and visualise the information from such analysis is important if it is to be useful.

This needs motivate the project work. This project's main aim is to investigate computational techniques to translate important elements extracted from Java source code into the natural language descriptions and study the extent to which the generated document helps users understand the code better.

Acknowledgements

I would first like to thank my project supervisors Wamberto Vasconcelos and Yaji Sripada, for their support, expert feedback, patience and enthusiasm over the duration of this project. Their help has been invaluable.

I would like to thank my family for understanding and giving me the time for my course study, for giving me support and strength to get through it all, and for making it all worthwhile.

Contents

1	Introduction	9
1.1	Overview	9
1.2	Objectives	10
1.2.1	Source Code Parsing	10
1.2.2	Data Model Creation	10
1.2.3	Graphical Representation	11
1.2.4	Natural Language Generation (NLG)	11
1.2.5	Testing and Evaluation	11
1.3	Document Structure	12
2	Background and related work	13
2.1	Source Code Parser	13
2.1.1	JFlex	13
2.1.2	Eclipse JDT	14
2.1.3	JavaParser	15
2.1.4	Yaml	16
2.1.5	IntelliJ IDEA	16
2.1.6	The Reflection API	17
2.1.7	Parser research summary	18
2.2	Natural Language Generators Research	19
2.3	Related Work	21
3	Requirements and Architecture	23
3.1	Functional requirements	23
3.2	Non-functional requirements	24
3.3	Architecture	25
4	Design	26
4.1	Source code parsing	27
4.1.1	Source code files finder design	27
4.1.2	JavaParser design	27
4.2	Data model design	28
4.3	Natural Language Generator design	29
4.4	Flowchart Generator design	30

5	Implementation	31
5.1	Development Tools	31
5.1.1	Integrated Development Environment (IDE)	31
5.1.2	Version Control	32
5.1.3	Backups	32
5.2	Source Code Parsing Implementation	32
5.2.1	Java Code Finder	32
5.2.2	JavaParser Implementation	33
5.3	Data Model Creation	34
5.4	Flowchart Generator Implementation	35
5.5	Natural Language Generator Implementation (NLG)	35
6	Testing and Evaluation	40
6.1	Testing	40
6.1.1	Testing during the implementation cycles	40
6.1.2	The final system testing	43
6.2	Evaluation	45
7	Discussion, Conclusion and Future Work	48
7.1	Discussion and Future Work	48
7.2	Conclusion	49
A	User Manual	53
A.1	Pre-Requisites	53
A.2	Running the Project System	53
B	Maintenance Manual	56
B.1	Requirements	56
B.2	Dependencies	56
B.3	Installation Instructions	56
B.4	Installation and running in Eclipse IDE	57
B.5	Compiling from Eclipse IDE	57
B.6	Compilation Instruction	57
B.7	Files	58

List of Tables

2.1	Java Model (Vogella, 2018)	15
2.2	Code translation systems comparison table 1	18
2.3	Code translation systems comparison table 2	19
6.1	Test Plan table	41
6.2	Test Log table	42
6.3	Metrics evaluation with BLEU	46

List of Figures

3.1	An architecture diagram.	25
4.1	The system process stages design scheme.	26
4.2	The structure of the document generated for each class.	29
4.3	The flowchart blocks design.	30
A.1	The 'code2text.bat' in the explorer window.	53
A.2	Launching the system from command prompt.	54

Chapter 1

Introduction

Software plays a central role in modern society, touching billions of lives on a daily basis. Writing and maintaining the software in the form of the source code is a core activity of software developers, who aim to provide reliable and functional software. However, writing and maintaining source code is a costly and error-prone activity. Software developers need to constantly look at the documentation and online resources, and they need to make sense of large existing projects. Both of these can be challenging, costly and slow down the development process.

This need motivates the project work. This project's main aim is to investigate computational techniques to translate important elements extracted from Java source code into the natural language descriptions and study the extent to which the generated document helps users understand the code better.

The system developed in this project supports the software development process by making it easier and cheaper. This software will be useful for people such as project leaders or architects who are trying to make sense of the project code, which could be too laborious to read as a plain code. Those who study Java language coding could use the system for learning, practising and enhancing knowledge and skills from particular examples.

1.1 Overview

Converting source code into natural language gives a number of opportunities and perspectives. This may include:

- understanding of the code, processes and operations within the code;
- code reuse, or reuse of the critical parts of the source code in a new project;
- describing the functionality of methods;
- debugging of the source code, finding the parts of the code affecting the incorrect behaviour and output;
- translating the given source code into a different language;
- increase the code reuse from existing code-base;
- studying the code development techniques or language syntax;

There can be much more benefits from the code translation into the natural language in case the natural language generated from the source code would give us the full understanding of the variables, methods, classes used within the source code project. This means the natural language output should provide a detailed static analysis of all components of the code and give the representation of the code easy to approach and understand for the user.

Another important aspect of the code analysis is the graphical code representation. Graphic representation is a better way of communicating the logic of a system to all concerned or involved. The maintenance of an operating program becomes easy with the help of the flowchart. It helps the programmer to put efforts more efficiently. The flowchart is designed for this system to reflect the sequence of methods called within the source code. This gives us the possibility to find the methods which are the most critical for the project and most frequently addressed within this project. In case we are doing source code debugging, it gives the possibility to find the method that delivers an error and to fix it. The natural language description given for each method makes it possible even for the person who is not involved in the source code development process to understand the code.

1.2 Objectives

This section describes the main objectives to be achieved in the project.

1.2.1 Source Code Parsing

The source code parsing should deliver to the project most important parts of the source code in the form that would be easy to analyse and perform all needed operations to generate the output in the form of the natural language and the flowchart diagram. More important that parser would give all descriptive data about the source code that would be possible to store or parse to the data model creation stage (see figure 4.1).

Another system objective is to provide a syntactic analysis of the Java source code component parts in order to facilitate the natural language generation and generation of the flowchart diagram.

This project is researching existing libraries and software currently available to find the most reliable and convenient way of source code parsing. The parser needed for the project must be selected based on being independent of the input Java code syntax or format. This means the system should take as an input Java code developed with the use of any development environment, and input code can contain errors. It is desirable that parser would not need code preparation. The parser outcome information should give out all the necessary information for the system. The information received from the parser should be easy to transfer or transformed into the data model convenient to be processed by the system.

1.2.2 Data Model Creation

Data model creation is a process used to define and analyse data requirements needed to support natural language generator with the scope of corresponding source code Java project.

The information from the parser has to be transformed into the data format or conceptual data model that can be used to create a final data model for the system. The final data model is used by the natural language and flowchart generators. The conceptual model gives the ability to extract and structuring the data.

This stage of the project is considering and researching on the best way of storing data delivered by the parser. The data stored should be full, relevant, easy to recognise and reuse. The data model created in the project should give the possibilities of easy access, use and modification.

1.2.3 Graphical Representation

A flowchart is a type of diagram that represents a work-flow or process (SEVOCAB, 2019). A flowchart is a graphical representation of the sequence of the steps where steps are represented by boxes of various kind and of the sequence of the steps shown by arrows connecting the boxes. Flowcharts are used in analysing, designing, documenting or managing a process or program in various fields.

In this project, the flowchart designed to represent a method calls sequence within the Java source code project.

1.2.4 Natural Language Generation (NLG)

Natural Language Generation is a computational task to convert non-linguistic input (such as data extracted from source code) into natural language descriptions. In this project, short text descriptions of important elements extracted from source code are auto-generated by using a corpus containing pairs of inputs and outputs. The system approach relies on naturally occurring verbal descriptions produced by a human developer. Thus, system input data is more “naturally occurring” in the sense that it has not been designed specifically for an NLG system.

For this project, natural language is generated step by step based on the fragments of natural language text generated from the source code and depending on the source code content. The natural language generated text are simple sentences describing method.

1.2.5 Testing and Evaluation

The testing and evaluation chapter provides with information about the testing strategy during the system development. This chapter describes the testing process and gives the final results and explanations of the testing process. Testing allows to deliver a system that meets expectations, prevents unexpected results, and improves the long term maintenance and re-usability of the system. Depending upon the system requirements, the appropriate testing methodologies are applied.

The evaluation section 6.2 is evaluating the project outcome. The future work and the project improvements are considered in this section as well. This section is accessing the initial requirements and evaluate how the requirements implemented in the project and what features were implemented for each requirement. Different methods of natural language generator evaluation are described in this section. Machine generated text is evaluated with metrics evaluation which compares generated texts to the texts written by people from the same input data, using BLEU automatic metric. If machine generated sentences would be compared to the sentences written by the developer it would give 100 per cent identity score. To avoid this, we need to compare the sentence generated by the machine and the sentence made by a third-party human. To do this I compare the machine-generated sentences with sentences written by one of my supervisor’s result of this evaluation supplied with the report.

1.3 Document Structure

This report structured as follows:

Chapter 2 - Background and related work

Chapter 2 provides an overview on different source code parsing software and libraries, compares these parsers related to the project system and concludes on the parser selection. This chapter also includes the natural language generator related works overview regarding to the project system.

Chapter 3 Requirements and Architecture

Chapter 3 outlines the projects functional and non-functional requirements, followed by an introduction to the chosen architecture.

Chapter 4 Design

Chapter 4 introduces design of the software system regarding to the architecture.

Chapter 5 Implementation

Chapter 5 describes in details of how individual components of the project have been implemented.

Chapter 6 Testing and Evaluation

Chapter 6 explains the testing strategy with tests provided for the system, followed by the system critical evaluation with future work suggestions.

Chapter 7 Discussion, Conclusion and Future Work

Chapter 7 contains a discussion and finishing with the conclusions on the project as a whole and its parts with providing future work suggestions regarding the implementation of this project.

Appendix

Appendices containing the Maintenance Manual and the User Manual for the system are provided at the end of this document.

Chapter 2

Background and related work

This chapter covers background information and highlights related work to the project objectives.

2.1 Source Code Parser

The system is building a translator in the form of the interpreter which is the form of the compiler. The complexity of building a compiler described in a so-called ‘Dragon book’ (Aho et al., 1986), as its cover depicts a knight and a dragon in battle, a metaphor for conquering complexity. The project system needs a parser to make access to a source code easier and finding of the necessary source code methods and other elements quick. A parser is a software component that takes input data (source code) and builds a data structure, often some kind of parse tree, abstract syntax tree or other hierarchical structure, giving a structural representation of the input. The parsing is followed by other steps to generate the output. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. This section considers the parsers and the systems containing the parsers with the intention to implement one of these parsing methods into the project system. The comparison of the requirements of the project system parser and considered software is provided at the end of this section.

2.1.1 JFlex

JFlex¹ is a lexical analyser generator for Java written in Java. It is also a rewrite of the tool JLex (Berk, 1996) which was developed by Elliot Berk at Princeton University.

A lexical analyser generator takes as input a specification with a set of regular expressions and corresponding actions. It generates a program (a lexer) that reads input, matches the input against the regular expressions in the spec file, and runs the corresponding action if a regular expression matched. Lexers usually are the first front-end step in compilers, matching keywords, comments, operators, etc, and generating an input token stream for parsers. They can also be used for many other purposes (Team JFlex, 2018).

This tool is unable to take initial source code project as it takes the “specification” as an input. This means that the input source code must be modified to be taken as an input. Also, for this tool to work, we need to arrange a number of JFlex specific regular expressions for the specification file to find necessary elements or information in the source code. If match found this would initiate launching another code that must be made to prepare the information from JFlex for natural language data model creation. This tool would do the job we need to do but seems too

¹<https://jflex.de/>

complicated and needs a good knowledge of JFlex specification files creation and specification files syntax. The project software would be easier to develop without using JFlex, as a stand-alone application without the need to create specification files and communications between source code and JFlex and between JFlex and the project system.

The JFlex usability for the system overview. The JFlex is a tool described as “lexical analyzer generator (also known as scanner generator) for Java, written in Java.” (Team JFlex, 2018). But if we take a closer look it does not take the Java source code as an input. It takes as an input a specification with a set of regular expressions and corresponding actions. This means that the specification structure has to be created from the source code with the use of data sets and a particular corresponding actions which will give the output in abstract form according to the actions set in a JFlex specification. This output file also needs to be modified to provide the system with necessary information about source code. This already sounds complicated and time-consuming as the knowledge of specification file creation for the JFlex is comparable with learning a new programming language with the procedures, syntax, and structure. This why this tool is rejected from implementation for the project.

2.1.2 Eclipse JDT

The Eclipse Java Development Tool (JDT) ² project provides APIs to access and manipulate Java source code. JDT provides access to the Java code with generation of the Java Model and creation of the Abstract Syntax Tree.

Java Model

Each Java project is internally represented via a model. This model is a light-weight and fault tolerant representation of the Java project. It does not contain as many information as the AST but is fast to create. For example the Outline view is using the Java model for its representation to be fast.

The Java model is defined in the org.eclipse.jdt.core plug-in. The Java model is represented as a tree structure which can be described via the following table.

²<http://www.vogella.com/tutorials/EclipseJDT/article.html>

Project Element	Java Model element	Description
Java project	IJavaProject	The Java project which contains all other objects.
src folder / bin folder / or external library	IPackageFragmentRoot	Hold source or binary files, can be a folder or a library (zip / jar file)
Each package	IPackageFragment	Each package is below the IPackageFragment-Root, sub-packages are not leaves of the package, they are listed directly under IPackageFragment-Root
Java Source File	ICompilationUnit	The Source file is always below the package node
Types / Fields / Methods	IType / IField / IMethod	Types, fields and methods

Table 2.1: Java Model (Vogella, 2018)

Abstract Syntax Tree (AST)

The AST is a detailed tree representation of the Java source code. The AST defines an API to modify, create, read and delete source code.

The main package for the AST is the `org.eclipse.jdt.core.dom` package and is located in the `org.eclipse.jdt.core` plug-in.

Each Java source element is represented as a subclass of the `ASTNode` class. Each specific AST node provides specific information about the object it represents.

To find an AST Node you could check of levels in the AST. A better solution is to use the visitor pattern via the `ASTVisitor` class (Vogella, 2018).

The Eclipse JDT usability for the system overview. The Eclipse JDT provides with a set of the tools necessary for the project as it can give necessary information about the source code for the system. The Eclipse JDT is a tool to develop an Eclipse IDE plug-in and another advantage of the system based on this tool that it can take any imported into the Eclipse IDE Java project source code as an input. But at the same time, it is a disadvantage as the system would be tied to the Eclipse environment which is against the Section 3 NFR3 requirement. This is a very good tool where the parsing is very similar to the `JavaParser` tool evaluated below but rejected because it is tied to the Eclipse IDE.

2.1.3 JavaParser

`JavaParser` is both the name of the set of tools and of the main component: “the parser” (`JavaParser Contributors Team`, 2019). The parser is extra lightweight library (no dependencies at all!) and can give you quickly and efficiently an Abstract Syntax Tree (AST described above Section 2.1.2) for your Java code. Sometimes an AST is not enough because you want:

- Understand to which element a certain name refer: a field? A variable?

- You need to know the type of a complex expression
- You want to get a list of all the interfaces implemented (also indirectly) by a certain class

All these and many more is possible with use of JavaParser library. **The JavaParser usability for the system overview.** The JavaParser provides with the set of the tools to parse, analyse, transform and generate the source code. The necessary tool for this project system is the parsing tool, but the analyse feature is also used in the project as well as the code transformation and the code generation can be used for future project development. The JavaParser is easy to approach and learn as it gives a library to implementation within the Java code. The JavaParser is easy to learn for the person who is already familiar with Java language. The JavaParser works fast as it is a Java library and Java is a well known as for its optimisation and Java Virtual Machine. This is why the JavaParser is selected for the project system as a source code parser to be implemented.

2.1.4 Yaml

YAML³ is well-known format within Ruby community, quite widely used for a long time now. But the project system is developed in Java which means YAML would mostly deal with property files and XML⁴ files. The project to generate a description text would need some Java source code configuration and properties to be extracted from these property and XML files.

Though JSON⁵ is becoming a popular format for web applications, using JSON files to describe the configuration is a bit cumbersome and is not as expressive as YAML. Let's see what YAML can do to make the project development easier. The YAML can give the system information about the code most of the interactive character. Such as version and release date, database connection parameters, list of supports protocols, list of users with their passwords (DZone, 2012).

The YAML usability for the system overview. The YAML is the system that provides tools to work with property and XML files in case we need some configuration for Java apps. It is possible to use this tool to generate the natural language project description, but it is very interesting to access a source code itself to implement such features like fault finding, debugging, refactoring or to generate the natural language describing the source code methods or other elements. This is why the YAML is rejected. The project is attempting to locate important code elements and to create a description of the whole project based on the description of the methods used in the project. The YAML implementation for our system would give us the information about the source code project and its properties, but what we would do to get the information about the source code methods. The JSON and XML project properties files are not able to provide the source code methods or variables information.

2.1.5 IntelliJ IDEA

The IntelliJ IDEA⁶ tool provides developers with static code analysis.

After IntelliJ IDEA has indexed your source code, it offers a quick response and intelligent experience by giving relevant suggestions in every context: instant code completion, code analysis, and re-factoring tools (JetBrains, 2019).

³<https://yaml.org/>

⁴XML - Extensible Markup Language file format used to create common information formats.

⁵JSON - JavaScript Object Notation is an open-standard file format that uses human-readable text to transmit data objects.

⁶<https://www.jetbrains.com/idea/>

Mission-critical tools such as integrated version control systems and a wide variety of supported languages and frameworks are all to hand – no plugin hassle included (JetBrains, 2019).

While IntelliJ IDEA is an IDE for Java, it also understands and provides intelligent coding assistance for a large variety of other languages such as SQL, JPQL, HTML, JavaScript, etc., even if the language expression is injected into a String literal in your Java code (JetBrains, 2019).

The IntelliJ IDEA usability for the system overview. The IntelliJ IDEA is an IDE system which provides some text generation features from the source code, but this IDE system is too complex and cannot be integrated into the project system neither it cannot integrate the project system as there is no access to the source code of the IntelliJ IDEA. Most of the features the project system interested in available only in the paid version. The system rejected because of the reasons above.

2.1.6 The Reflection API

The Reflection API⁷⁸ is commonly used by programs which require the ability to examine or modify the runtime behaviour of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.

The drawback of Reflection is that Reflection should not be used indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it. The following concerns should be kept in mind when accessing code via reflection.

Performance Overhead: Because reflection involves types that are dynamically resolved, certain Java virtual machine optimisation cannot be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts and should be avoided in sections of code which are called frequently in performance-sensitive applications.

Security Restrictions: Reflection requires a runtime permission which may not be present when running under a security manager. This is an important consideration for code which has to run in a restricted security context, such as in an Applet.

Exposure of Internals: Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behaviour with upgrades of the platform (Oracle, 2017).

The Reflection API usability for the system overview. The Reflection API is a set of programming tools provided by Oracle to examine or modify the runtime behaviour of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. The Reflection provides a dynamic analysis of an application runtime behaviour which makes the Reflection very complex and too complicated for the project system. The project system is concentrated on the static code analysis and natural language generation this why this tool is rejected from being implemented for this project.

⁷<https://docs.oracle.com/javase/tutorial/reflect/index.html>

⁸API - Application Programming Interface is a software intermediary that allows two applications to communicate.

2.1.7 Parser research summary

The background parser research shows what the translation system needs in common. The comparison of the functionalities of the project system and considered software technologies is provided below. The main disadvantages of the software technologies in front of the project system are taken as a comparison parameters. It is hard to compare these technologies because they are either the complete systems or either apart of the system or just library and API.

The JavaParser provides with the set of the tools to parse, analyse, transform and generate the source code. The necessary tool for this project system is the parsing tool, but the analyse feature is also used in the project as well as the code transformation and the code generation can be used for future project development. The JavaParser is easy to approach and learn as it gives a library to implementation within the Java code. The JavaParser is easy to learn for the person who is already familiar with Java language. The JavaParser works fast as it is a Java library and Java is a well known as for its optimisation and Java Virtual Machine. This is why the JavaParser is selected for the project system as a source code parser to be implemented.

These two tables bellow table 2.2 and table 2.3 show the aims of the project system with compare to the systems described in the background research.

















	Project System	JFlex	JavaParser	Eclipse JDT
No code preparation needed				 *
Delivers natural language description				
Delivers diagram or any other graphical representation	 **			 ***
No additional software needed				

Table 2.2: Code translation systems comparison table 1

















	Project System	YAML	IntelliJ IDEA	Reflection
No code preparation needed				
Delivers natural language description				
Delivers diagram or any other graphical representation				
No additional software needed		 *****		

Table 2.3: Code translation systems comparison table 2

* - here the Eclipse JDT is a tool for developing plug-ins which work under the Eclipse which means no code preparation but the project must be imported into the Eclipse workspace for plug-in to work.

** - only attempted to create a graphical representation as a flowchart diagram.

*** - again the Eclipse JDT is a tool for developing plug-ins which work under the Eclipse which means graphical representation can be delivered by another Eclipse plug-in.

**** - the YAML is a language that works with the property files, this means there is a possibility that a certain additional library may be needed to do the certain objective.

To conclude the background research we can say that there is no stand alone application was found that would return a Java source code natural language description and at the same time project code graphical representation. And we can see that the project system will make a software developing process easier and cheaper because the structured elements of the code with description are easier to approach and modify.

Based on the background research JavaParser selected as the Java source code parsing tool. The JavaParser selected because it gives all necessary tools to find the Java project files, create the nodes to the whole code or parts of this code these needed.

2.2 Natural Language Generators Research

NLG generation is an interesting challenge as it has a significant impact on a user's of the modern software system. More and more systems using auto-generated text from the non-linguistic input. A full survey of NLG would be very time consuming as there are a huge number of ongoing projects with the use of data-driven methods as well as new applications and tools for data processing. This project considers only data pairing with the text and natural language text

generation from the pairs created. These may help the investigation on the possibility of natural language from source code and evaluate the extent at which the generated description helps to understand this code. The aim of the background research is to build up the picture and aims for the project.

This text below describes the natural language generation complexity and suggests typical stages of natural-language generation. Consider Philip M. Parker, who offered more than 100,000 books for sale via Amazon.com, including for example his “The 2007-2012 Outlook for Tufted Washable Scatter Rugs”, “Bathmats”, and “Sets That Measure 6-Feet by 9-Feet or Smaller in India”. Obviously, Parker did not write these 100,000 books by hand. Rather, he used a computer program that collects publicly available information, possibly packaged in human-written texts, and compiles these into a book Gatt and Krahmer (2017). Parker’s example is an example of text-to-text generation method. This project aims to generate text from Java source code which is referred to data-to-text generation method. Many applications have been developed over the last years using this method:

- soccer reports;
- virtual ‘newspapers’ from sensor data and news reports on current affairs;
- text addressing environmental concerns, such as wildlife tracking, personalised environmental information and enhancing engagement of citizen scientists via generated feedback;
- weather and financial reports;
- summaries of patient information in clinical contexts;
- interactive information about cultural artefacts, for example in a museum context;
- text intended to persuade or motivate behaviour modification;

A sophisticated NLG system needs to include stages of planning and merging of information to enable the generation of text that looks natural and does not become repetitive. The typical stages of natural-language generation, as proposed by Dale and Reiter, (Dale and Reiter, 2000) are:

Content determination: Deciding what information to mention in the text. For instance, in the pollen example above, deciding whether to explicitly mention that pollen level is 7 in the south-east.

Document structuring: Overall organisation of the information to convey. For example, deciding to describe the areas with high pollen levels first, instead of the areas with low pollen levels.

Aggregation: Merging of similar sentences to improve readability and naturalness. For instance, merging the two following sentences (Wikipedia, 2019):

- Grass pollen levels for Friday have increased from the moderate to high levels of yesterday and
- Grass pollen levels will be around 6 to 7 across most parts of the country into the following single sentence:

- Grass pollen levels for Friday have increased from the moderate to high levels of yesterday with values of around 6 to 7 across most parts of the country.

Lexical choice: Putting words to the concepts. For example, deciding whether medium or moderate should be used when describing a pollen level of 4.

Referring expression generation: Creating referring expressions that identify objects and regions. For example, deciding to use in the Northern Isles and far northeast of mainland Scotland to refer to a certain region in Scotland. This task also includes making decisions about pronouns and other types of anaphora.

Realization: Creating the actual text, which should be correct according to the rules of syntax, morphology, and orthography. For example, using will be for the future tense of to be.

An alternative approach to NLG is to use "end-to-end" machine learning to build a system, without having separate stages as above. In other words, we build an NLG system by training a machine learning algorithm (often an LSTM) on a large data set of input data and corresponding (human-written) output texts. The end-to-end approach has perhaps been most successful in image captioning, that is automatically generating a textual caption for an image.(Wikipedia, 2019)

2.3 Related Work

Text generation from data concerns the problem of generating well-formed, natural language description texts from non-linguistic, source code formal meaning representations (Gatt and Krahmer, 2017). In our case, the input to a project system is a source code representation. In order to learn a natural language generation (NLG) system from data, a parallel corpus containing pairs of source code inputs and text outputs must be constructed. In many studies on text generation from data, these parallel resources are relatively small, compared to work on sportscasting (Chen and Mooney, 2008), weather reporting (Belz, 2007), or biology facts (Banik et al., 2012). This project system is generating these pairs with the logic described in Implementation chapter 5 section 5.5.

A recent trend is the use of crowd-sourcing to obtain parallel NLG data (Wen et al., 2016); (Novikova et al., 2016); (Gardent et al., 2017). Crowd-workers are presented with some meaning representation (e.g., triples from a knowledge base) and asked to verbalise these representations in natural language. This method allows for fast annotation, and thus solves the data scarcity problem, it also raises some new issues. For instance, sentences or utterances are produced by crowd-workers without much context, which puts to question the naturalness of the resulting text. Novikova et al.(Novikova et al., 2016) compare collecting data from logic-based meaning representations, and pictorial meaning representations, and find that the former approach leads to less natural and less informative descriptions. This might be related to the problem that the natural language sentence is a very close verbalisation of the "logic" input, i.e., many terms in the meaning representation can be simply taken up in the sentence (Richardson et al., 2018).

The system approach relies on naturally occurring verbal descriptions produced by the developer. Thus, system input data is more "naturally occurring" in the sense that it has not been designed specifically for an NLG system (as compared to (Wen et al., 2016) who randomly generate input representations) yet it still corresponds to a formal language. There is relatively little lexical correspondence expected between source code representations and verbal descriptions. This is an interesting challenge for the project system.

NLG work related to the current project has been reported in (Lavoie et al., 1997). This work described software models represented in graphical diagram form. The paper presents the ModelExplainer, or ModEx for short. This tool represents the data in the form of the fluent English text. ModEx output integrates tables with text generated automatically and text entered by the user. This is another interesting example of the related work where automatically generated text includes paragraphs describing the relations between classes, and paragraphs describing examples.(Lavoie et al., 1997)

Chapter 3

Requirements and Architecture

The overall architecture, functional and non-functional requirements of the project software are detailed in this chapter. The proposed architecture is presented and discussed in Section 3.3.

3.1 Functional requirements

- **FR1 Read/Parse Java code requirement.** The software must be capable to take files from a Java source code project folder as an input. It is necessary to get at least one .java source code file and parse the elements of this code to the system for the processing. This means that the program must be capable to take these files and parse their content for the processing.
- **FR2 Provide representation for the parts of the parsed code.** The software must be able to locate required elements of the input source code project and represent the information in a structured form. The project system must be able to process the source code and structure it in the blocks according to the intuitive understanding and logic.
- **FR3 Output in natural language requirement.** The software must produce the text output which describes in natural language the input source code class structure, methods used within the class, method calls and variables declaration statements.
- **FR4 The graphical representation output requirement(optional).** The software optionally must produce the output file with the source code graphical representation. This is an optional requirement to generate a diagram or any other graphical representation for the source code to improve the source code understanding provided by the system to the user.
- **FR5 Statistic information requirement.** The system must provide basic statistic information on the source code project. The system must return quantitative information about the project.

3.2 Non-functional requirements

- **NFR1 Cross-platform.** This non-functional requirement is expanding the project system user base for the different operating systems users. The software should work under any operating system (Microsoft Windows, Mac OS, Linux, Solaris and Android platform) where the Java virtual machine (JVM)¹ is present. Java programming language chosen for the system development is a platform independent programming language, but requires the JDK² software installed on the system which automatically includes JVM installation on the system.
- **NFR2 All Technologies Open Source or Free.** It is important that technologies, libraries and any external modules must be open source or otherwise available for free. These means that the project must not require any extra monetary value being spent on software modules or technologies needed to fulfil the project requirements.
- **NFR3 Generic and Robust.** The model should be able to work with the various Java source code projects; it should not be tied to a particular Java source code project. This time a lot of different developing tools and editors available for Java developers and this project system must be able to process a Java source code developed in a different developing environment or editor. The software must provide a timely response. This means that the execution of the program should be reasonably timed depending on the source code project size. The system should be able to cope with errors during the process to be robust.

¹JVM - a Java virtual machine is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

²JDK - The Java Development Kit is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

3.3 Architecture

The architecture of the system contains five main parts (Figure 3.1):

- Source code finder. This part is responsible for finding all project files at the specified location. In our case, it is a project directory or folder.
- The parser. The JavaParser was selected as the parser. JavaParser creates the nodes to the source Java code. Through these nodes, the system is able to extract the source code as a whole with creating a compilation unit or its parts with the use of visitor pointing with operators which parts should be extracted and parsed with node.
- The objects Database creation. This stage is for transforming the information from the node objects into the data format that gives the system a possibility to perform the next steps manipulations with this data.
- Natural language text generation and saving this text into the file. On this stage, all variants of the code lines should be predicted and described in natural language. Natural English language sentences must be generated along with the code snippets they describe.
- Flowchart graphics generation. On this stage, we need to generate the graphics shapes such as rectangles and diamonds with corresponding text inside these shapes and place them into the sequence. Each of these graphics shapes represents a block for the diagram corresponding to the method from the source code or to the 'if' statement.

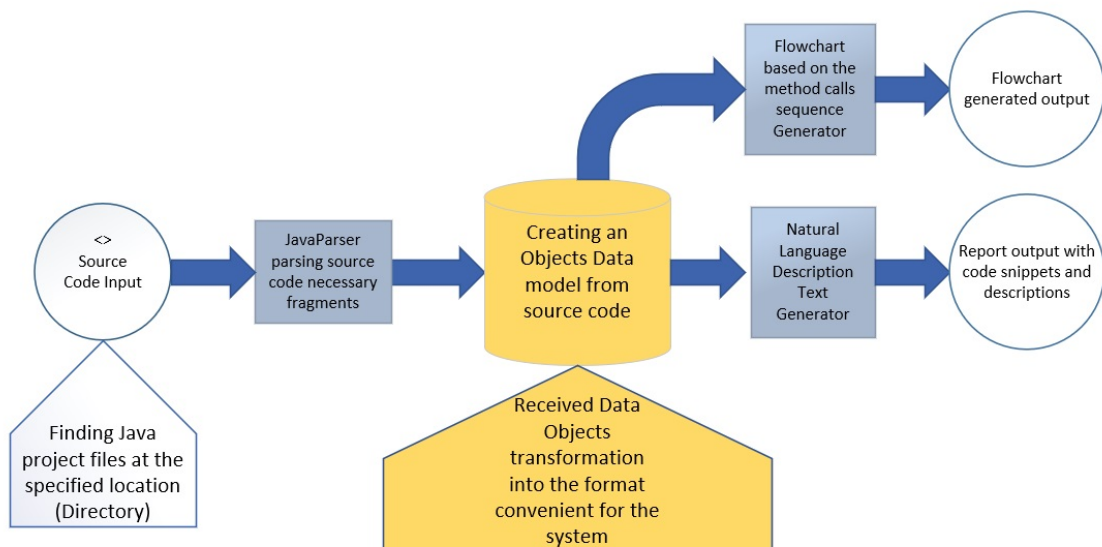


Figure 3.1: An architecture diagram.

The project system architecture is made the way where the elements or parts of the system are easy to access and improve as well as the whole system.

Chapter 4

Design

This chapter focuses on designing a structure for main elements of the system architecture Section 3.3 including justification for the design structure selected.

The design of the system is based on the architecture described in the Chapter 3 Section 3.3 and contains the five main elements (see Figure 3.1 and Figure 4.1). This chapter gives a detailed design description of these elements. The design technique is based on the JavaParser choice for the Java source code parsing. The system structure is built around the information JavaParser is able to give us.

The project system is designed from the parts shown on the figure below (Figure 4.1) to process the source code, to generate the natural language and at the same time to generate the flowchart output. The final stage “Critical elements identification” is generated automatically when the system is going through the “Natural language generation” and “Flowchart generation”. The “Critical elements identification” is the section where the system finds the code lines which system could not recognise. This part of the design made to support the robustness of the system and at the same time to collect the information about the code elements which are not paired with natural language phrase or in case of the flowchart with the flowchart block.

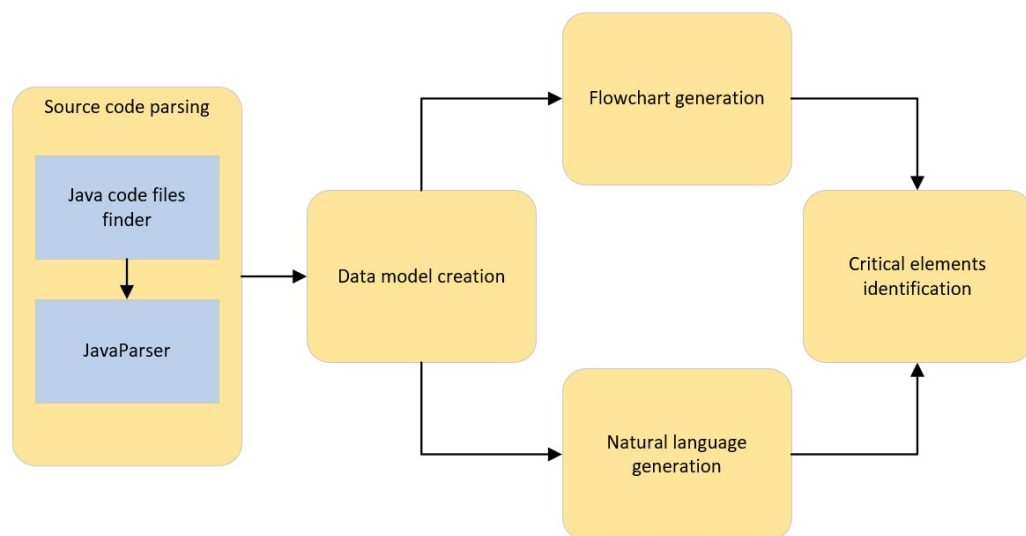


Figure 4.1: The system process stages design scheme.

4.1 Source code parsing

The source code parsing design consists of two parts:

- The Java code files finder. The Java code files finder part is responsible for finding java code files and classes in a project directory.
- The JavaParser. This part is designed to deliver all the information from the code needed to create a Data Model with the data necessary for the system.

4.1.1 Source code files finder design

When using JavaParser there are some operations that the system has to do every time, for example finding and taking Java files as input. It is necessary for the project system to operate on a whole project, so the system explores all Java files at the given directory. This has been done within the software product with the class named “DirExplorer”. This class is responsible for exploring, filtering and handling the Java files from the target directory.

At this stage, the system also extracts a class name from the Java class file name. This is done because the JavaParser returns class names and interface names with the same command described below in implementation Chapter 5.

4.1.2 JavaParser design

The JavaParser is a set of libraries that support:

- Code analysis
- Code refactoring
- Code generation

Code analysis is the main feature used for the project, but for the product growth and improvement the other two features could also be implemented. Refactoring can be used to modernise the code legacy, to update dependencies or to change some usage patterns. Code generation can be used to generate a piece of code inside of the existing code or for the code replacement.

The Abstract Syntax Tree (AST) has to be built for each Java file. And then this AST is used to navigate through the code. There are two main strategies to do so:

1. To operate on specific types of AST nodes the best strategy is to use a visitor
2. To process all sort of different nodes and iterate through them the recursive iterator strategy should be selected

The project system uses both the visitor strategy and the recursive iterator strategy. For example, to extract class names the visitor strategy is the best, because of the specific kind of node “ClassOrInterfaceDeclaration” should be used. But when the project system needs to iterate over the nodes, the node recursive iterator should be used. For example, to get statement lines and their numbers there are several classes extending the Statement base class. The system could use a visitor but with the visitor, the same code in several visit methods needs to be rewritten, one for each subclass of Statement. In addition, the system needs only to get the top level statements, not the

statements inside it. For example, a 'for' statement could contain several other statements. With the system custom, "NodeIterator" class system can easily achieve this.

The choice of the nodes has been made depending on what information is needed from the code to understand the project:

1. The class name node. To analyse the project classes connections and recognise the methods it is necessary to extract the class names. For example, to represent the structure of the project source code as folders and subfolders the class names folders would be the top subfolders inside of the project folder.
2. The node that returns method declaration lines. This would be the next stage in folders representation of the source code. The information about each method declared within the project is returned by this node.
3. The node that returns all method calls. These nodes can be used to create the sequence of the methods called within the project. Also, this information may help the system to find the methods which are called more often and show the information about each method on its usefulness according to a quantitative evaluation of the method being called within the project.
4. The node that returns variables declaration lines. This node can be used to describe the bottom level in the folder system - variables. This would show what variables declared within the different classes and methods.
5. The node to return statement lines. This node is needed to obtain the information about the sequence flow and all the information that cannot be obtained from the previous nodes and is necessary to build the natural language description.

These five nodes are designed to parse into the project system abstract information sets for further processing with natural language generator or any other way that can be used within the system.

4.2 Data model design

To design the data model for the system, the question is what data the natural language generator needs to create the code interpretation. This chapter describes what data is collected and how it is structured in the system.

One of the system objectives is to build a translator in the form of the interpreter from Java machine code into the natural language. This is because interpreters translate machine code line by line where compilers take the whole file and translate it into another language. In the case of an interpreter, the code fragments can be iterated line by line in order for the code to be executed. By iterating code line by line, a pair generation of a short text description and the code can be achieved. In case of fault finding it might give a corresponding message, sending the information to the critical element identification part and proceeding to the next line.

The data set needed for the data model is generated from the method declaration node parsed by the JavaParser. This node object is formatted and stored into the array of code string lines. At this stage, the database of these pairs can be created to make the system that could train itself with the processing of the new project.

The project system text is generated for each created node. The array of objects is created for the flowchart diagram. The object in this array represents the method declared within the project and consists of:

- The class name - to recognise the class of each method belongs to.
- The method name - to recognise the method within the class.
- The method text. This is an array of the method code string lines.
- The identification number. This is a unique number for each method from the project.

The natural language sentences can be added to this array to create a consistent database that can be trained.

4.3 Natural Language Generator design

The natural language generator is designed to generate the natural language text for each node opened. The Figure 4.2 represents the structure of the generated report for each class of the source code project. The system returns Java class file location and the sentence that introduces the class name for each class node. Then the description of the information belongs to the class should be written in the following order:

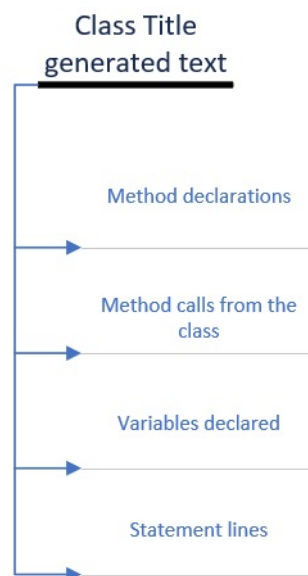


Figure 4.2: The structure of the document generated for each class.

1. Each method declared within the class in the form:

- The method declaration <id number> and the <Class Name>. Where <id number> is a method number within the class, <Class Name> is a name of the class the method belongs to.
- Method declaration lines. This line shows the numbers of the start and the finish lines for the method.

- The method code by itself.
 - The natural language text describing the method.
2. Method calls. Methods called within the class with the line numbers they called from.
 3. Variable declaration. The variables declared within the class with the line numbers.
 4. Statement lines. The statement lines within the class with the start and the finish line numbers.

4.4 Flowchart Generator design

Optionally for the project the flowchart should be generated for the user of the system to be able to see the information about each element or block from the flowchart with the click or as a pop-up. To do this the flowchart should be designed from the blocks corresponding to each method call or “if” statement (Figure 4.3).

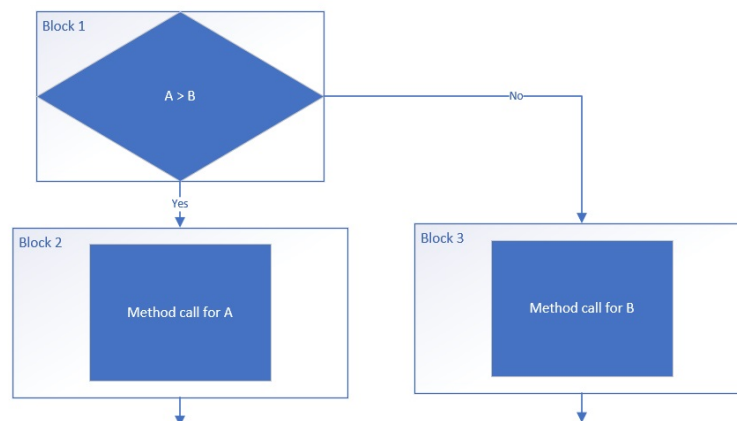


Figure 4.3: The flowchart blocks design.

The block structure gives a possibility to link flowchart blocks to the method information and method natural language description. The rectangle blocks on a diagram represent certain methods from the source code. The diamond blocks represent the if statements from the source code.

Chapter 5

Implementation

This chapter describes the implementation process of the designed elements, tools used in the developing process, the difficulties met during the implementation and solution found or suggested. The software developed for the project is the part of the research on the way to represent the Java code in natural language and to give the user understanding of the project source code and its parts.

The agile development approach is used for this project. The agile approach means that every functionality improvement or implementation made in a separate development cycle has been tested within this cycle. After the functionality is implemented to the system the whole system is tested as well.

Initially, the system was generating output, but later the text file report has been implemented because the amount of information generated is huge (2 thousand text lines just for the project from 5 classes with 200-500 code lines each). The amount of text generated is large, but it is very convenient to read because the text is formatted and all main sections of the generated document are separated and easy to find.

5.1 Development Tools

5.1.1 Integrated Development Environment (IDE)

The integrated development environment is very useful for the development of the project that uses many different technologies and combines them. The advantages of Eclipse IDE are:

1. Navigation is made easier within the project and other projects which could be connected to the project development. Other Java projects are used as an input for this project which makes navigation through the input projects important.
2. Using IDE would cost less time and effort.
3. Error debugging is easy, you can easily navigate to the error line.
4. All files can be viewed and managed at the same time and on the same screen.
5. Refactoring gives a possibility to find and replace everything from variable, function or class name to functionality.
6. Syntax checking and auto-completion assist the developer. There is no need to remember everything.

7. Simplifies downloading of the required packages and libraries.
8. Organises imports.
9. It is free and open source software.
10. Automatic backups.
11. Eclipse can be used to develop applications in other languages such as C++, HTML, PHP, Ruby etc.
12. Extensions and plugins can be used.

This list is not exhaustive, but these are the main highlights useful for this project. As this project is being developed in Java it is logical to use an IDE that the most Java developers use because of its benefits.

5.1.2 Version Control

The use of version control is important to any software project. Version control allows you to maintain the history of changes that have been made to the source files. This is useful in detecting the point at which bugs are introduced into the system when files are changed. It also allows maintaining a stable version of the project that is known to work, while continuing development on the latest iteration. For this project Eclipse IDE is used for version control.¹

5.1.3 Backups

Backups are often forgotten in the modern age of cloud computing. There is no replacement to a good local or remote copy of the entire project. During the project development regular local backups were made every week, along with using services such as Dropbox and Google Drive to maintain safe online copies of the project.^{2,3}

5.2 Source Code Parsing Implementation

The source code parsing is designed from two parts: Java code finder and JavaParser (Section 4.1). This section describes the implementation of each part.

5.2.1 Java Code Finder

The Java code finder is implemented in the class “DirExplorer”. This class takes directory location as an input parameter and finds and filters the file names with the use of two interfaces “FileHandler” and “Filter”. Then it explores the sub-directories with the “explore” method, searching and filtering files again. The JavaParser method calls are wrapped inside the finder. This gives the project system the possibility to work with each class sequentially. This means that for each java project source code class found, the system finds the smaller elements such as methods and variables and then proceeds to the next class file.

¹www.eclipse.org

²<https://drive.google.com>

³www.dropbox.com

5.2.2 JavaParser Implementation

The JavaParser creates the Abstract Syntax Tree (AST) and then uses this AST to navigate through the code. It finds and extracts certain fragments that project system code commands it to find. In order to access this fragments project system code uses so-called “visitors” strategy described at “Design” chapter 4 Section 4.1.

The following five nodes are implemented:

1. The class names node. This node with the command keyword “ClassOrInterfaceDeclaration” calls from AST to return all the class and interface names. Because of this species of JavaParser, the project system needs the real class name duplicate to recognise which method belongs to which class. To make this possible the duplicate name is extracted from the Java source code file name. It is necessary for the Java syntax that the file name with extension .java must be the same as the class name.
2. The node that returns method declaration lines. This node is the most important node for the system as it returns all the function list for the class and gives a possibility for the system to create most of the Data model object elements. This node visits AST with the command keyword “MethodDeclaration” and gives access to the source code method declaration code lines. Using this node the system gets the method start line number, finish line number and method declared code lines for each method declared within the class.

The counters of methods declared within the class and methods declared within the project are implemented in this part. These counters are implemented with the intention to give statistical information to the user and to identify the methods with unique numbers.

3. The node that returns all method calls. This node visitor is implemented with the parameter keyword “MethodCallExpr”. This node returns the method call line and numbers of the starting and finishing lines for this method call. Another specific of JavaParser that it returns method call lines which are inside of the method call. For example:

```
btn_help.addActionListener(new java.awt.event.ActionListener() {  
    public void actionPerformed(java.awt.event.ActionEvent evt) {  
        btn_helpActionPerformed(evt);  
    }  
})
```

This piece of Java code for user interface element calls the method “addActionListener” which creates a new event object for “ActionListener” with the method “actionPerformed” as a parameter. These calls of the method inside of another method call are very often used within the graphical interface usage and generation. It is complicated to describe this method call because normally methods are called from one line and all other method calls are not returned by JavaParser from inside of the method call. These exceptional cases are difficult to describe. Iterating and recognising these method calls need more research on finding ways to process these exceptions. This is why such cases should be identified at the Data model creation stage as critical and be processed accordingly.

The counters of method calls within the class and method calls within the project are implemented in this part. These counters are implemented with the intention to give statistical information to the system user and to identify the method calls with unique numbers.

4. The node that returns variable declaration lines. This node visitor is implemented with the parameter keyword “VariableDeclarationExpr”. With this node, the system receives variable declaration lines and they line numbers from the source code.

The counters of variables declared within the class and variables declared within the project are implemented to this part. These counters are implemented with the intention to give statistical information to the system user and to identify the methods with unique numbers.

5. The node to return statement lines. In this case, the system needs to find all sort of statements. In this case, there are several classes extending the Statement base class so a visitor can be implemented but the same code in several visit methods would need to be written, one for each subclass of Statement. In addition, the system needs only to get the top level statements, not the statements inside it. For example, a ‘for’ statement could contain several other statements. This logic is implemented with the use of “NodeIterator” class.

5.3 Data Model Creation

This part implemented according to the design described in Chapter 4 Section 4.2. The model implemented corresponds to the system needs. This model is not enough for developing a large natural language generation system. The database development should be considered to make a system that could generate a large amount of descriptive text. But as the time of the project is limited the system is simplified to describe the code with logic ‘if the element is present’ in particular lines of the code. This is the reason why the model created corresponds more to the flowchart generator than a natural language generator.

The data model is implemented in the array of objects “methodDeclObjList”. The class “DataModelObj” is implemented to construct the objects. The constructor of the class is made to take the declared method text along with the parameters of this method: unique identification number within the project which is corresponding to the method counter number within the project, method name which is extracted from the first string of method declaration and the class name which the method belongs to.

The data model created gives a possibility to iterate through the method lines along with the possibility to fully identify the method within the project. To make the identification easier the numbers of lines corresponding to the method declaration can be added, but the usefulness of line numbers is doubtful at this stage because the system would need these numbers only in the case of information being used from different nodes. The difficulty is that the system needs to extract this number from the text line returned by JavaParser. JavaParser for the command to return start and finish line numbers returns the string: “Optional[(line 495,col 5)]-Optional[(line 504,col 5)]”. This JavaParser specific feature makes the line numbers extraction a bit complicated and because of the doubtful usefulness at this stage was not implemented.

5.4 Flowchart Generator Implementation

The flowchart generator is an optional task for the project and was attempted to be implemented last. Implementation of this generator started with the method called “generateFlowchart”. The logic for this generator is to start from the method called “main” and to find the first method call. If this method call is inside of the decision-making operator draw a decision-making block, draw a method call block, proceed to the method called declaration, iterate the lines and find the next method call, repeat the drawing procedure until the end of the method reached, repeat the drawing procedure until the end of the “main” method reached.

The Data Model for this part is implemented and described above. The graphical part of the diagram generator is designed consisting of the blocks of different method calls and ‘if’ statements. The ‘if’ statements can be extracted from the statement lines or from the method declaration code lines. The first possible solution is to find and extract the ‘if’ statement from the statement lines. The system would need the statement line number to identify the ‘if’ statement location relative to the method call line location. The line number can be extracted from the objects returned by JavaParser with `getStart()` and `getEnd()` methods. The system would need to transform the object into the string and cut the integer number of the line out of this string. The second solution is to extract the ‘if’ statement lines and the method call lines from the method declaration lines. The system would need to iterate through the method code lines and find ‘if’ statements or method call lines. In this case, the system would need to identify the lines of ‘if’ statements and method call lines with numbers or otherwise generate each graphic block inside of the iteration process.

The size of the block should be evaluated before the system would be able to draw the shape diamond or rectangle corresponding to the code line. The size can be evaluated by counting the characters for each string placed inside of the shape.

The difficulties described above and possible unidentified difficulties brought the decision to save time for more important implementation parts.

5.5 Natural Language Generator Implementation (NLG)

The natural language generator implementation along with the parser implementation is the most important parts of the project. The natural language generator used in this project system to generate a description text has got the following stages:

1. **Content determination:** Deciding what information to mention in the text.
2. **Document structuring:** Overall organisation of the information to convey.
3. **Aggregation:** Merging of similar sentences to improve readability and naturalness.

The structure of the document generated is organised in the way that information is given to the system user is consistent. Information about the class goes first. Methods, variables and statements follow after. Then at the end, there are some statistical sentences generated about the whole project. Natural language text is generated for each piece of the code delivered to the system by JavaParser.

First of all, the system generates a title for the class found in the project directory. This sentence consists of two parts:

1. Introduction phrase: “The project Java class file found at location:”
2. The path to the class file found, for example:

D:\PDFtextExtractor2.2.2.1\src\my\PDFtextExtractor\PDFfiles.java

The next part of the text generated is the class name introduction. Because of the specific of JavaParser the command “ClassOrInterfaceDeclaration” is the command to return class name. To avoid the confusion between the class name and the interface name the system compares the name of the class extracted from the file name and the name delivered by JavaParser. Here is an example of the text generated by the system on its source code from the class “DirExplorer” to show the interface and class text representation:

```
=====
The Interface 'FileHandler' declared within the class 'DirExplorer'
=====
The Interface 'Filter' declared within the class 'DirExplorer'
=====
The file content is the Java class code. The Class Name is *** DirExplorer ***
=====
```

The names of the interfaces are used within the class parsed by JavaParser before the actual class name. This is another specific of the JavaParser that can be avoided by creating a database and constructing the text from this database.

The method declaration part is the part where the biggest amount of the text is generated and the biggest generator logic is implemented. To create a good structure and aggregate sentences system generates a title for the whole section and heading for each method:

```
=====
Methods, declared within the class, are:
-----
--<Method declaration #1 within the class DataModelObj>--
--- Method declaration lines:
Optional[(line 17,col 2)]-Optional[(line 19,col 2)]
```

To generate further text the system must find a first line of the method declaration text. The text generated from this line describes the method in the form of six sentences, for example:

```
public int getId() {
    return id;
}
```

The method is declared with the name 'getId'.

The method returns an integer number variable.

The method access type is 'public', accessible everywhere.

The method type is not 'static'.

The method takes no parameters.

The method throws no exception.

The logic generating the first sentence extracts the method name and inserts this name into the sentence.

The second sentence generated is based on the information about what type of variable method returns:

- “void” - no parameters returned from the method
- “int” - method returns integer value
- “int[]” - method returns an array of integer type values
- “byte” - method returns a value of a byte type
- “byte[]” method returns an array of byte type values
- “short” - method returns a value of a short type
- “short[]” method returns an array of short type values
- “long” - method returns a value of a long type
- “long[]” method returns an array of long type values
- “float” - method returns a value of a float type
- “float[]” method returns an array of float type values
- “double” - method returns a value of a double type
- “double[]” method returns an array of double type values
- “boolean” - method returns a value of a boolean type
- “boolean[]” method returns an array of boolean type values
- “char” - method returns a value of a character type
- “char[]” method returns an array of character type values
- “string” - method returns a value of a string type
- “string[]” method returns an array of string type values
- “File” - method returns a value of a File type
- “File[]” method returns an array of File type values

The logic contains 21 different variants on the function return variable type. This type of logic does not provide the information to the user if the object was changed inside of this method. This can be realised only by iterating through the method code lines and finding the line pattern which corresponds to the object parameter change operation.

The logic generating the third sentence represents the method access type where only 4 variants are considered: private, public, protected, default.

The fourth sentence indicates if the method is static or not.

The fifth sentence reveals if the method takes any parameters. If the method takes parameters it extracts and shows these parameters.

The sixth sentence is generated based on the information about the exceptions (if there are any) that the method has.

These output sentences eventually were improved with the use of the natural language generation technique aggregation. The sentences were aggregated to make the text closer to the natural language. Now there are only four sentences which are delivering the same information:

```
public int getId() {
    return id;
}
```

The method 'getId' from class 'DataModelObj' is 'public',
accessible from everywhere.

It takes no parameters as an input and returns an integer number variable.

This method type is not 'static'.

The method throws no exception.

This improvement is an illustration of natural language aggregation applied to the structured text built on the determined content.

The method calls text consists of the heading, method call lines number and method call lines from the source code, for example:

Method calls:

```
-----
--<Method call #1 within the classDirExplorer>--
Method call lines:
Optional[(line 23,col 9)]-Optional[(line 23,col 28)]
explore(0, "", root)

--<Method call #2 within the classDirExplorer>--
Method call lines:
Optional[(line 27,col 13)]-Optional[(line 27,col 30)]
file.isDirectory()
```

Variable declaration text example:

```
Variable declarations:
-----
--<Variable declaration #1 within the classDirExplorer>--
Variable declaration line: Optional[(line 28,col 18)]File child
```

In this example, the variable is declared inside of the 'for' statement, but for this part, it is not important and this information is skipped from the output. In the code declaration:

```
for (File child : file.listFiles()){
```

The block text consists of the heading statement line numbers and the statement itself from the source code.

Here is the example of the generated statement lines text:

```
Statement lines: Optional[(line 17,col 64)] - Optional[(line 20,col 5)]
-----
0 - {
1 -     this.filter = filter;
2 -     this.fileHandler = fileHandler;
3 - }
```

The text is very similar to the previous two blocks: method calls and variable declaration. The block text consists of the heading statement line numbers and the statement itself from the source code.

The final part of the generated text is statistical information about the whole project. Example:

```
--<The total amount of classes in the project: 5>--
--<The total amount of methods declared within the project: 20>--
--<The total amount of method calls within the project: 178>--
--<The total amount of variables declared within the project: 39>--
```

This is information from the nodes opened on how many classes in the project, how many methods are declared, how many times different methods are called and how many variables are declared within the project source code. There is no statistical information from the “statements” part.

Chapter 6

Testing and Evaluation

This chapter describes the testing techniques used to test the project system. It provides the testing results for the system on different size projects. Also, this chapter is evaluating the system by looking over the initial project requirements and the implemented features.

6.1 Testing

The agile development approach is used for this project. This means that every agile cycle for the functionality being implemented or improved the test is provided. After the functionality was implemented to the system the whole system was tested as well.

This section provides the information on the testing during the implementation process and the final system testing on the projects of different size from different resources.

For testing during the implementation cycles a white box top-down integration testing used. Which means the stubs developed inside the main application and then the integration of the components in a whole application prototype being tested.

The final system testing is a white box testing on the final state of the system developed. As a result of this testing approach, unnoticed design errors may be detected and described.

6.1.1 Testing during the implementation cycles

This section gives information on two main implementation stages: the parser implementation and the natural language generator implementation. The testing information is represented by the two tables Table 6.1 (Test Plan table) and Table 6.2 (Test Log table).

The first Table 6.1 represents the planned functionality that must be tested, the variable or set of data must be pointed for each test and the description of expected result should be given or explained otherwise by giving the document reference.

The second Table 6.2 (Test Log table) represents a test log information where the first column corresponds to the program run number, the second column is the number or numbers of tests from the first table 6.1, the third column is the attempt taken to test the planned test if the fix applied the attempt must be bigger than 2, the observed result column is for the information on the test result, the comments and actions column is for the information on importance of the bug found or the information on how the bug should be fixed or was fixed.

Test	Purpose of Test	Test Data or Page Ref	Expected Results or Page Ref
1	Test DirExplorer class delivers the correct files from the project directory	The line to print out the full path is added into the code to provide the test System.out.println(file);	All file locations of Java class file names from the project are delivered correctly.
2	JavaParser class or interface visitor test	className variable	The correct class name returned from this variable
3	JavaParser method declaration visitor test	methodDecl node data	The node data represent a code lines of method declared within the class
4	JavaParser method calls visitor test	methodCall node data	The node data represent a code lines of method call within the project class
5	JavaParser variable declaration visitor test	variableDecl	The node data represent a code lines of variable declaration within the project class
6	JavaParser statements line block test and NodeIterator class test	The “node” variable data test	The data delivered by the ‘node’ variable is corresponding to the source code data statements lines
7	Test if the output text file generated	nlgtext.txt file	File created at the correct location (D:/NLGtext/ folder)
8	Output text file content test	nlgtext.txt file	The content of the output file corresponds to the expected result, no residual text left from previous generation, text content is consistent and represented in the correct order
9	The data objects array for storing Data Model objects test	“methodDeclObjList” list of objects	The array list returns the objects in the expected order with expected parameters created by the constructor

Table 6.1: Test Plan table

Run	Tests Covered	Attempt	Observed results	Comments and actions
1	1	1	Expected result achieved	-
2	2, 3, 4, 5, 6	1	Satisfactory result achieved	For line numbers printed string consisting the integer line numbers. To extract these numbers as integers the string manipulation needs to be performed. No action on the result needs to be performed this time
3	7, 8	1	The new content attached to the old content of the text file instead of being replaced	Should be fixed by clearing file content before placing a new content
4	8	2	Expected result achieved	Fixed by adding 2 lines in the code <code>"writer.print("");"</code> and <code>"writer.close();"</code> after the line 45 <code>"PrintWriter writer = new PrintWriter("d:/...txt");"</code> in the class "TFCmain"
5	9	1	The array contents the equal objects all objects the same as the last object created. The array length is as expected	Needs to be fixed to proceed to the next implementation stage
6	9	2	Expected result achieved	Fixed by overriding "equals" and "hashCode" methods in the "Data-ModelObj" class

Table 6.2: Test Log table

6.1.2 The final system testing

The final test of the system has been made with six different Java projects. Two of these projects are small under a thousand lines of the code, two of the average size and two are relatively big with more than a hundred classes. The project did not return any errors while the system was processing these projects source code. The execution time was not measured, but approximate time for the system to process the biggest projects is up to five seconds. The text generated as it was expected in full. There are no unexpected elements found in the text generated for each Java project. The statistical information returned by the system for each project from smallest to largest is:

- Number Guessing game ¹ 139 lines of text generated:

```
--<The total amount of classes in the project: 1>--  
--<The total amount of methods declared within the project: 2>--  
--<The total amount of method calls within the project: 6>--  
--<The total amount of variables declared within the project: 4>--
```

- Hot Swap ² 244 lines of text generated:

```
--<The total amount of classes in the project: 3>--  
--<The total amount of methods declared within the project: 6>--  
--<The total amount of method calls within the project: 13>--  
--<The total amount of variables declared within the project: 2>--
```

- PDF to Text the project developed by me myself two years ago 1894 lines of text generated:

```
--<The total amount of classes in the project: 3>--  
--<The total amount of methods declared within the project: 22>--  
--<The total amount of method calls within the project: 246>--  
--<The total amount of variables declared within the project: 42>--
```

- The project system statistics on itself code 2396 lines of text generated:

```
--<The total amount of classes in the project: 5>--  
--<The total amount of methods declared within the project: 20>--  
--<The total amount of method calls within the project: 178>--  
--<The total amount of variables declared within the project: 39>--
```

- Airlines Reservation system ³ 25395 lines of text generated:

¹<https://www.codewithc.com/number-guessing-game-java/>

²<https://github.com/skeeto/sample-java-project>

³<https://www.codewithc.com/airlines-reservation-system-java-project/>

```
--<The total amount of classes in the project: 97>--  
--<The total amount of methods declared within the project: 107>--  
--<The total amount of method calls within the project: 4201>--  
--<The total amount of variables declared within the project: 162>--
```

- Simple NLG ⁴ 67985 lines of text generated:

```
--<The total amount of classes in the project: 133>--  
--<The total amount of methods declared within the project: 986>--  
--<The total amount of method calls within the project: 6959>--  
--<The total amount of variables declared within the project: 1657>--
```

The test of the system with these projects did not show any hidden errors. The output text file generated and saved. No unexpected errors found in the text generated. The final test did not uncover any bugs within the code. The generated text cannot be compared with the description text for these projects because the project system has the intention to describe what the source code small elements such as methods do and the description for the project describes what the whole project does and what functionality it delivers.

⁴<https://github.com/simplenlg/simplenlg>

6.2 Evaluation

This section is evaluating the project outcome and its success. The future work and the project improvements are considered in this section as well. This section is accessing the initial requirements and evaluate how the requirements implemented in the project and what features were implemented for each requirement.

FR1 Read/Parse Java code requirement. The software must be capable to take files from a Java source code project folder as an input. It is necessary to get at least one .java source code file and parse the elements of this code to the system for the processing. This means that the program must be capable to take these files and parse their content for the processing.

This is the biggest part of the project research on different methods of parsing the java code described in Chapter 2 of this document. The result of this research and the tool selection process are described below.

The different tools were researched. Some of these tools were tested to implement the parsing for the project. These tools using a different approach to retrieve the information from the source code. These tools are JFlex, Eclipse JDT, JavaParser, YAML, IntelliJ IDEA, The Reflection API.

The JavaParser provides with the set of the tools to parse, analyse, transform and generate the source code. The necessary tool for this project system is the parsing tool, but the analyse feature is also used in the project as well as the code transformation and the code generation can be used for future project development. The JavaParser is easy to approach and learn as it gives a library to implementation within the Java code. The JavaParser is easy to learn for the person who is already familiar with Java language. The JavaParser works fast as it is a Java library and Java is a well known as for its optimisation and Java Virtual Machine. This is why the JavaParser is selected for the project system as a source code parser to be implemented.

Conclusion for FR1 evaluation and description of features implemented for this functionality. The JavaParser selected for the project system. It is able to deliver all necessary information for the system and provide the analysis that can be used in the system. There are no big problems with the parser implementation as it is simple and very straight forward. The features are added to improve the parser implementation:

- The feature implemented in the “DirExplorer” class. This is the feature to find and load the Java source code file from the project directory. This feature also gives a possibility to process all project class files sequentially.
- The feature implemented in the “NodeIterator” class. This feature gives a possibility to iterate over the different Nodes opened by the JavaParser to the AST in case the system needs to receive and filter the information from these nodes. The iterator implemented in a separate class which gives the possibility for this feature to be reused in case the new set of information from the source code will be needed.

This functionality and the features implemented make the improvement of the parsing part of the system easier in the future.

FR2 Provide representation for the parts of the parsed code. This functionality achieved by creating, structuring and aggregating of the parsed information and the information generated by the system. This is achieved by the system code structure and some Java methods to build the

sting arrays or strings from node objects. The JavaParser provides the information from the code with the use of nodes. The nodes delivering objects according to the request made from a system. Then the system has to extract needed object parameter and format this parameter into the array of strings, string or other formats the system needs to operate the information. This is implemented for each object parsed into the system to create the Data Model for the system is able to operate.

Conclusion for FR2 evaluation and description of features implemented for this functionality. The data transferred with the system is a simple and easy process which approves the usefulness of the parser implemented for the system. The feature might be considered to implement is the representation of the data into the database where the system would be able to train the data and improve the output of the system.

FR3 Output in natural language requirement. This functionality is implemented into the system by creating a logic to associate the data parsed from the code with the natural language sentence fragment. This pairing of information gives a possibility to generate simple sentences in natural language describing the code elements received from the source code.

The evaluation for this functionality implementation is provided with the use of metrics evaluation. The metrics evaluation compares generated texts to the texts written by people from the same input data, using an automatic metric such as BLEU. This evaluation method does not provide a semantic comparison. The task-based (extinct) evaluation method is based on providing the text to the coder, and assess how well it helps him to write the code. The human rating evaluation is based on providing the text to the people and asking them to rate the quality and usefulness of the text. The task-based method and the human rating method need professionals who are not involved in the project development process. But as time is limited and the professional coders are not easy for me to find the metrics evaluation method was selected. To do this the online Interactive BLEU score evaluator is used ⁵. To do this evaluation need to compare the sentence generated by the machine and the sentence made by a human. The table 6.3 shows the result of such comparison. The bold text shows the difference between sentences.

Sentence 1	BLEU	Length ratio	Text
Human	100.00	1.00	The method 'getPDFnames' from class 'PDF-files' is publicly accessible everywhere .
Machine	53.44	1.18	The method 'getPDFnames' from class 'PDF-files' is ' public ', accessible from everywhere .
Sentence 2	BLEU	Length ratio	Text
Human	100.00	1.00	It takes ' selPDF ' , an array of File objects as an input parameter and returns an array of string objects .
Machine	28.49	1.14	It takes parameter (s) (File [] selPDF) as an input and returns an array of string value variables .

Table 6.3: Metrics evaluation with BLEU

⁵<https://www.letsmt.eu/Bleu.aspx>

From the table 6.3 we can see that this evaluation method is not very good as for example machine generates ‘public’ and the human wrote publicly which objectively can have the same meaning and the professional coder reader might not spot the difference. The other methods of evaluation may be considered to get a different evaluation result.

Conclusion for FR3 evaluation and description of features implemented for output in natural language requirement. The descriptive text generated evaluated with the metrics evaluation method automated system. The BLEU score for two sentences compared is 37.14 out of 100. But this evaluation is not really represented how understandable the generated text, because it compares the words amount and similarity from sentences. Also, this evaluation method compares punctuation symbols such as commas, quotation marks and brackets this is why the score is so low for the second sentence (Table 6.3).

FR4 The graphical representation output requirement (optional). This is an optional requirement for this project. This requirement is only approached in the project by building a data model and design for the graphical representation of the code, but graphic elements were not created because of the lack of time on the project. For the Data Model created an object array list where the objects generated with the “DataModelObj” class constructor. Design for this part is described in a Chapter 4 Section 4.4.

FR5 Statistic information requirement. The statistical information collected and presented at the end of the generated file. This information gives the quantitative analysis of the source code and gives an idea about the size of the project.

NFR1 Cross-platform. The multi or cross-platform is achieved by the use of Java programming language. The software expected to work under any operating system (Microsoft Windows, Mac OS, Linux, Solaris and Android platform) where the Java virtual machine is present.

NFR2 All Technologies Open Source or Free. The use of paid products avoided within the process of the system building. **NFR3 The project flexibility on input Java source code.** The project system is built as an independent stand-alone system. The JavaParser and the Java source code finder feature implemented are able to find all kind of Java source code files inside of the given project directory location.

NFR4 Generic and Robust. The software was tested on relatively large Java projects with execution time not exceeding 10 seconds. This testing shows the robustness of the system. The system designed to be generic by made out of the blocks, classes and functions responsible to perform the action or operation, for example, the classes “DirExplorer” and “NodeIterator” are good examples of the system being generic and modular.

Chapter 7

Discussion, Conclusion and Future Work

This chapter contains a discussion and finishing with the conclusions on the project as a whole and its parts with providing future work suggestions regarding the implementation of this project.

7.1 Discussion and Future Work

The project overall successful achieving most of the goals and requirements. The deep research on source code parsers provided. The system is able to provide a report on a Java source code along with the source code snippets.

There are many different ways to use the project system: this report can help developers to maintain an existing Java project, learn about methods and variables from the project, it can be used for studying on how to code, it can be used to remind the project content to the developer who developed it.

Of course, the graphical representation would be an advantage for the system. As graphics are always easier to understand for human perception. Unfortunately, because of the time limit, I was not able to finish this optional task for this project. But the data structure created for this task and only a graphical blocks and blocks sequence logic have to be created to finish the task.

The natural language challenge is basically successful, but there is a lot of ways to improve the natural language generator which is not easy or even impossible to do within the project time scale. For example, for the natural language, a machine learning system could be implemented and trained. Another option is to create a database on the Java syntax, pair this syntax with the natural language description phrases and generate the sentences from these phrases.

The project task is very complex and involves a lot of different tasks to do research on, most of these researches are provided within this document, but a lot left behind a scene. For example, instead of the plain text generated there might be an html document or pdf LATEX structure created by the system, which was considered within the project development process.

The natural language descriptions generated can help the students to learn Java code. The structure of the report generated can help the developer to find a particular method or to refresh memory on the code developed years ago. Text generated may get improved to produce the natural language descriptions for some of the simple functions such as 'getters' and 'setters'. There is a way to improve the generated text by building more sophisticated interpreter to analyse the sequence of the operations in each method and compare this sequence to the knowledge database with this sequence natural language description.

7.2 Conclusion

Looking over the initial project requirements, the implemented features in the project system successfully fulfil all of these requirements except for the optional graphical representation. The project system is evaluated and tested with the six different projects of a different size downloaded from the different resources to show that the system generic on input and robust. These tests did not return any bugs or errors for the project system. The output report file is generated and saved.

There is some of the features can be implemented on the top of the system provided. It is can be natural language improvements, different graphical implementation improvements, the parser functionality and the structure of the object parsed improvements, additional database implementation, different output file format or set of the files might be generated. All this can be done with a reasonable time scale and resources. This system provides a basis for future improvements with all researches and components it delivers.

The project has fulfilled its main objective on the investigation of computational techniques to translate source code important elements extracted into the natural language descriptions and study the extent to which the generated document helps users understand the code better. The system developed in this project supports the software development process by making it easier and cheaper. This software will be useful for the people who need to make sense of a large Java projects source code that would be too laborious to read as the plain code even for the expert. For example project leader or architect who is trying to make sense of the project code and its elements. This system also can be used to learn Java language coding from particular examples.

The source code parser (JavaParser) found during the investigation and implemented found useful to parse the source code elements and information about the source code. The source code parsing delivers to the project most important parts of the source code in the form that is easy to analyse and perform all needed operations to generate the output in the form of the natural language and the flowchart diagram. More important that parser gives all descriptive data about the source code these are possible to store and parse to the data model creation stage. The research to find the most reliable and convenient way of source code parsing is provided. Different methods, libraries and software compared in section 2.1. In the project parsing is the process of analysing source code and the most important parts extraction. The JavaParser selected because it gives all necessary tools to work with the Java project files, create the nodes to the whole code or parts of this code these needed.

To create a data model for the system the project progressing the data from source code parser to the actual data model to be used by the natural language and flowchart generators. The data model created gives a possibility to iterate through the method lines along with the possibility to fully identify the method within the project. To make the identification easier the numbers of lines corresponding to the method declaration can be added in the future project development.

The flowchart generator design and data model created. No graphics generator implemented for the flowchart, because of the complexity of the task for the time scale given. The flowchart is an optional task for the project.

Simple natural language generator logic implemented within the project. In order to create a content for natural language generator a parallel corpus containing pairs of source code inputs and

text outputs is constructed. The structure of the generated document considered in ‘Implementation’ chapter 5 section 5.5 gives class by class description of the method declarations, method calls, variable declarations and statement lines. This structure suits the objective for the source code describing and finding the most critical elements for the user. The descriptive text aggregated in the sentences connected semantically for each declared method.

The evaluation and test information of the project provided. The testing and evaluation chapter provides with information about the testing strategy during the system development. The chapter 6 section 6.1 describes the testing process and gives the final results and explanations of the testing process. This section is accessing the initial requirements and evaluate how the requirements implemented in the project and what features were implemented for each requirement. Different methods of natural language generator evaluation are described in this section. Machine generated text is evaluated with metrics evaluation which compares generated texts to the texts written by people from the same input data, using BLEU automatic metric. Disadvantages of this evaluation method are described in section 6.2 and the task-based method and the human rating method are suggested as a good replacement for the metrics evaluation method. The task-based method and the human rating method need professionals who are not involved in the project development process. But as time is limited and the professional coders are not easy for me to find the metrics evaluation method was selected.

Bibliography

- Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc.
- Banik, E., Gardent, C., Scott, D., Dinesh, N., and Lina, F. (2012). Kbggen - text generation for knowledge bases as a new shared task. *The seventh International Natural Language Generation Conference. Starved Rock, Illinois, USA*.
- Belz, A. (2007). Automatic generation of weather forecast texts using comprehensive probabilistic generation-space models. *Cambridge University Press*.
- Berk, E. (1996). Jlex: A lexical analyzer generator for java. <https://www.cs.princeton.edu/~appel/modern/java/JLex/>. Accessed: 04 Feb, 2019.
- Chen, D. L. and Mooney, R. J. (2008). Learning to sportscast: A test of grounded language acquisition. *In Proceedings of the 25th International Conference on Machine Learning (ICML), Helsinki, Finland*.
- Dale, R. and Reiter, E. (2000). *Building natural language generation systems*. Cambridge, U.K.: Cambridge University Press.
- DZone (2012). Using yaml for java application configuration. <https://dzone.com/articles/using-yaml-java-application>. Accessed: 04 Feb, 2019.
- Gardent, C., Shimorina, A., Narayan, S., and Perez-Beltrachini, L. (2017). Creating training corpora for nlg micro-planning. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Vancouver, Canada*.
- Gatt, A. and Krahmer, E. (2017). Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. *arXiv:1703.09902*.
- JavaParser Contributors Team (2019). Javaparser. <https://javaparser.org/>. Accessed: 04 Feb, 2019.
- JetBrains (2019). Capable and ergonomic ide for jvm. <https://www.jetbrains.com/idea/>. Accessed: 04 Feb, 2019.
- Lavoie, B., Rambo, O., and Reiter, E. (1997). Customizable descriptions of object-oriented models. *Proceedings of the Fifth Conference on Applied Natural Language Processing, Washington, DC*.
- Novikova, J., Lemon, O., and Rieser, V. (2016). Crowd-sourcing nlg data: Pictures elicit better data. *Proceedings of The 9th International Natural Language Generation conference, Edinburgh, UK*.
- Oracle (2017). Trail: The reflection api. <https://docs.oracle.com/javase/tutorial/reflect/index.html>. Accessed: 04 Feb, 2019.
- Richardson, K., Zarrieb, S., , and Jonas (2018). The code2text challenge: Text generation in

- source code libraries. *Institute for Natural Language Processing, University of Stuttgart*,.
- SEVOCAB (2019). Sevocab: Software systems engineering vocabulary. https://pascal.computer.org/sev_display/search.action;jsessionId=23887a13ed103e4e09df3fd206f7. Accessed: 9 Apr, 2019.
- Team JFlex (2018). Jflex user's manual. <http://jflex.de/manual.html>. Accessed: 04 Feb, 2019.
- Vogella (2018). Eclipse jdt - abstract syntax tree (ast) and the java model. <http://www.vogella.com/tutorials/EclipseJDT/article.html>. Accessed: 04 Feb, 2019.
- Wen, T.-H., Gašić, M., Mrkšić, N., Rojas-Barahona, L. M., Su, P.-H., Vandyke, D., and Young, S. (2016). Multi-domain neural network language generation for spoken dialogue systems. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 120–129, San Diego, California. Association for Computational Linguistics.
- Wikipedia (2019). Natural language generation. https://en.wikipedia.org/wiki/Natural-language_generation. Accessed: 04 Feb, 2019.

Appendix A

User Manual

A.1 Pre-Requisites

This program has been designed to run on any platform where the Java virtual machine is installed. To install JDK 8 Java development kit version 8 from Oracle which includes JVM please visit the Oracle website.¹

A.2 Running the Project System

Windows platform. The software system can be launched on the Windows platform from the ‘code2text.bat’ file or from the command line launching the ‘code2text.jar’ file.

Launching from ‘.bat’ file.

1. To execute the system processing itself source code double click the ‘code2text.bat’ in the explorer window with the project folder open (Figure A.1).

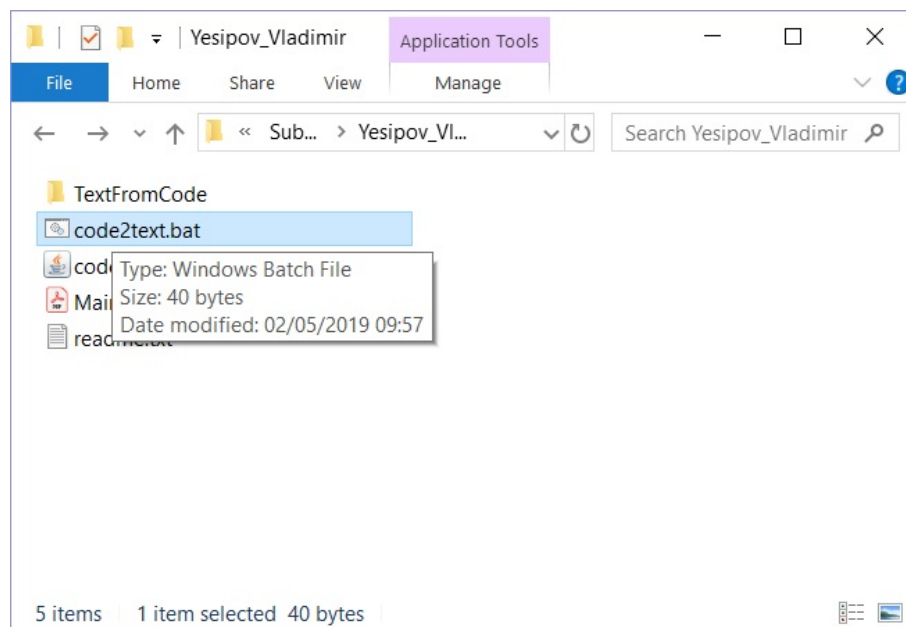


Figure A.1: The ‘code2text.bat’ in the explorer window.

2. To execute the system on a different project source code folder you need to change the argument defined in a ‘.bat’ file on the path to the desired project folder.

¹<https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

```
java -jar code2text.jar .\TextFromCode
```

where the `<.\TextFromCode>` should be changed
with the path to the desirable source code project folder.
Example:
`<C:\Some Folder\Project Folder>`

The generated text report file 'nlgtxt.txt' is generated in the main project folder.

Launching from command prompt or 'PowerShell'

1. Open command prompt or 'PowerShell' window in a project folder (Figure A.2).

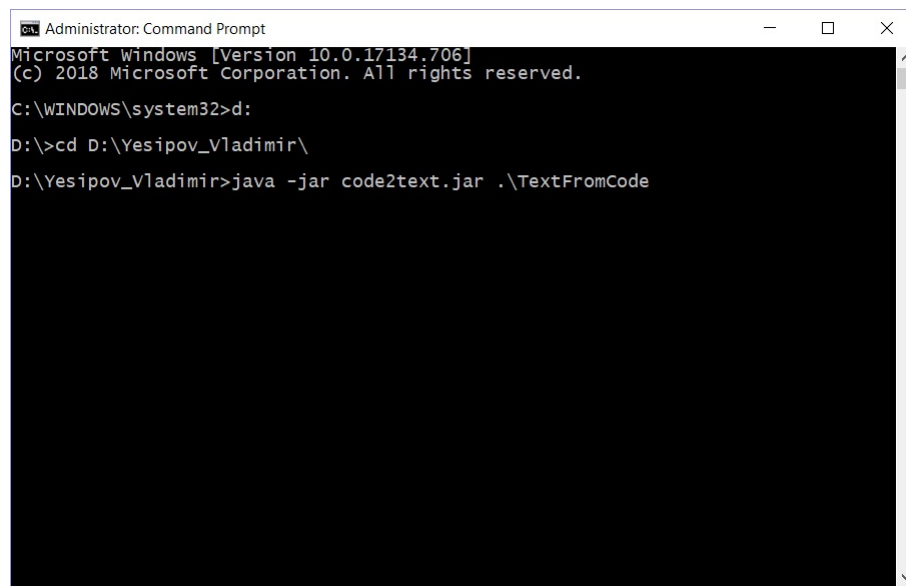


Figure A.2: Launching the system from command prompt.

2. Launch the system on the project of your choice with the command:

```
java -jar code2text.jar C:\Some Folder\Project Folder
```

where the `<C:\Some Folder\Project Folder>` should be replaced
with the path to the desirable source code project folder.

The generated text report file 'nlgtxt.txt' is generated in the main project folder.

Linux platform.

1. Open terminal window in a project folder.
2. Launch the system on the project of your choice with the command:

```
java -jar code2text.jar "C:\Some Folder\Project Folder"
```

where the `<C:\Some Folder\Project Folder>` should be replaced
with the path to the desirable source code project folder.

The generated text report file 'nlgtxt.txt' is generated in the main project folder.

Mac Os platform. Should be able to run with the “Jar launcher” if the JDK for Mac installed. The project source code folder should be passed as an argument. Not tested as I have no access to the hardware with Mac Os.

The generated text report file 'nlgtxt.txt' is generated in the main project folder.

Appendix B

Maintenance Manual

B.1 Requirements

The main requirement for the system to work is JVM¹. The system is tested on the hardware with i7-6700HQ CPU @ 2.60 GHz with 8 GB RAM. Minimum recommended RAM is 512MB.

B.2 Dependencies

The following dependencies are not included in the project and will need to be installed separately on the system:

- Java 8 from Oracle²
- Eclipse IDE³

The following dependencies referenced in the project:

- Eclipse jdt core⁴
- Java Parser core⁵
- Guava⁶

B.3 Installation Instructions

For installation on a Windows machine assuming that the external dependencies have been installed. Navigate to the download directory `cd /foo/bar` Extract the files from the archive file

Vladimir_Yesipov.zip

Run the .jar file using instructions above in section A.2. For installation on a Linux machine assuming that the external dependencies have been installed. Navigate to the download directory `cd /foo/bar` Extract the files

Vladimir_Yesipov.zip

Run the .jar file using instructions above in section A.2. The generated text report file 'nlgtxt.txt' is generated in the main project folder.

¹<https://www.oracle.com/technetwork/java/javase/downloads/>

²<https://www.oracle.com/technetwork/java/javase/downloads/>

³<https://www.eclipse.org/downloads/>

⁴<http://download.eclipse.org/eclipse/updates/3.8/compositeArtifacts.jar>

⁵<https://github.com/javaparser/javaparser>

⁶<https://github.com/google/guava/wiki/Release23>

B.4 Installation and running in Eclipse IDE

To run the application inside Eclipse:

Import the project into Eclipse
File->Import->"Existing Project into Workspace"
->Next
Browse to the TextFromCode directory, click ok.
Tick the Search for nested projects
Click finish.
Then Run.

The generated text report file 'nlgtxt.txt' is generated in the main project folder.

B.5 Compiling from Eclipse IDE

To create a new runnable JAR file in the workbench:

1. From the menu bar's File menu, select Export.
2. Expand the Java node and select Runnable JAR file. Click Next.
3. In the Opens the Runnable JAR export wizard Runnable JAR File Specification page, se
4. In the Export destination field, either type or click Browse to select a location for
5. Select Include libraries in a jar file.

Run jar file from the command line or from bat file (Explained above).

The generated text report file 'nlgtxt.txt' is generated in the main project folder.

B.6 Compilation Instruction

Open Eclipse.

Project->Untick build automatically.

Make sure that Eclipse knows there is Java 8 installed

Window -> Preferences -> Java -> Installed JREs

Click search

Select lib contained in /usr click OK

Eclipse will then search for Java Runtime Environments.

Tick java-8-oracle

Import the project into Eclipse:

File->Import->"Existing Projects into Workspace"

Browse to the TextFromCode directory, click ok.

In the options tick search for nested projects.

Click Finish.

Add User Libraries to Eclipse:

Windows->Preferences

Java->Build Path->User Libraries->Import

Browse to the file called libs.userlibraries in the TextFromCode folder, hit ok.

Tick all the libraries inside.

Hit OK.

This should allow you to compile the Project from Eclipse

To run the application inside Eclipse:

Import the project into Eclipse

File->Import->"Existing Project into Workspace"

->Next

Browse to the TextFromCode directory, click ok.

Tick the Search for nested projects

Click finish.

Then Run.

The generated text report file 'nlgtxt.txt' is generated in the main project folder.

B.7 Files

The list of the files from software zip file:

- **"code2text.jar"** - This Java jar file is located at the main folder. This file is a runnable jar file for the project system. This file to run requires a source code directory to be pointed as an argument.
- **"code2text.bat"** - This file is located at the main folder. This file is made to run the project system from an explorer window.
- **"readme.txt"** - This file is located at the main folder. This file summarises the content of the software zip file.
- **"TextFromCode directory"** - The directory contains the Java project source code files.
- **"/src/com/textfromcode/DataModelObj.java"** - The Java class file to generate data objects from the information parsed by JavaParser.
- **"/src/com/textfromcode/DirExplorer.java"** - The Java class file to find all of the java source code files from pointed directories and sub-directories.
- **"/src/com/textfromcode/NodeIterator.java"** - The Java class file to iterate over the different 'Nodes' created by the JavaParser to parse the source code.
- **"/src/com/textfromcode/package-info.java"** - The Java file contains the information about the package automatically generated by Eclipse IDE.
- **"/src/com/textfromcode/TFCmain.java"** - The Java main file contains the parser implementation, natural language generator and flowchart generator code. Most of the flowchart generator code is commented out because the implementation of this part is not finished.