VC Wiki Reader

7th Semester Project Report

Christian Friis Lyngbo Andersen
Claus Nygaard Madsen
Mathias Johns Toustrup
Patrick Grønhøj
Sean Skov Them
Steen Friis Jensen

sw704e15

Aalborg University
Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
9220 Aalborg East
http://www.cs.aau.dk/

Title:
VC Wiki Reader

Theme:
Internet Technology

Project:
7th Semester

Project Period:
Fall Semester, 2015

Project Group:
sw704e15

Participant(s):
Christian Friis Lyngbo Andersen
Claus Nygaard Madsen
Mathias Johns Toustrup
Patrick Grønhøj
Sean Skov Them
Steen Friis Jensen

Supervisor(s):
Peter Dolog

Number of Copies: 8
Number of Pages: 54
Appendices: 10 pages, 1 CD
Date of Completion: January 21, 2016

Abstract:
Web accessibility deals with the removal of certain barriers which may prevent disabled people fully interacting with websites and web applications. When websites are designed, developed and maintained with this in mind, the barriers hindering these people's usage of the Internet are removed.
VC Wiki Reader is a web enabled desktop application, which though dialogue, can search, navigate, and present Wikipedia articles for people with visual impairment. The application builds on commonalities and lacking features, discovered in an analysis of other accessibility tools, and provides its interaction through spoken words. The user interacts with the application, by utilizing the Microsoft speech recognition technology, and the application presents the articles through their Text-to-Speech technology. For the information available through the application, a snapshot of the Wikipedia database was accessed utilizing the Solr search engine platform. VC Wiki Reader works as a technology demonstration of web accessibility software based on verbal navigation for information acquisition.

Signature(s):

Christian Friis Lyngbo Andersen
✉ can11@student.aau.dk

Claus Nygaard Madsen
✉ cnma12@student.aau.dk

Mathias Johns Toustrup
✉ moust12@student.aau.dk

Patrick Grønhøj
✉ pgronh12@student.aau.dk

Sean Skov Them
✉ sthem12@student.aau.dk

Steen Friis Jensen
✉ sfje12@student.aau.dk

Preface

In order for the reader to benefit from reading this report, it is advantageous to have prior knowledge of methods and terminology used by students who have attended 7^{th} semester of the Master of Science in Engineering (Software) at Aalborg University.

The sources used throughout the report are in either English or Danish.

The authors will be referring to themselves in first person plural.

In code segments the reader will encounter ↙ and ↳, which symbolises the continuation of a line.

Contents

1 Introduction

Access to the Internet has become a highly valued commodity and integral part of daily life. Through the Internet, you can manage and interact with both personal and public elements of daily life, such as, doing bank transfers, reading and writing emails, searching for information, buying products, etc. In the western world, access to the Internet has almost become mandatory, as a majority of the important personal information flows exclusively through it, for instance in Denmark, where all mail from the government is exclusively available through the Internet [Retsinformation, 2015].
While the majority of the population with internet access have no impairments that limits their interaction with the Internet, some groups of disabled people have disadvantages accessing the Internet. To improve the utilization of the Internet for these groups, developers have, in recent years, taken steps towards enabling impaired users to perceive, navigate, and interact with websites more efficiently. One group, estimated to consist of 285 million people worldwide, that is directly affected by their disability, are the visually impaired [WHO, 2015]. For these people to be able to utilize websites to their fullest capabilities, it is necessary to remove the accessibility barriers, such as websites containing visual media without textual description or transcription, limiting people with visual impairments from getting the full user experiences of websites. A number of technologies have been developed, that assist people with visual impairments accessing websites.

1.1 How to Navigate the Internet as a Visually Impaired Person

A person with normal vision can read a webpage and quickly sort relevant material at a glance. For example, sorting out advertisements and interpret images. A visually impaired person is challenged in navigating a website, since most websites are designed to be represented visually. Though visually impaired people may not be able to see the website, it does not mean that they can not perceive it. A visually impaired person relies on additional tools besides a browser to navigate the Internet, for example screen readers. Screen readers provide an alternative way of perception, utilizing hearing. A screen reader is, in essence, a program, that reads the content of the screen, an application, or a website aloud. Screen readers are only able to read clear text, which means, text represented in images are omitted and not read aloud. One problem is that not all websites are structured in a manner that is suitable for screen readers or is constructed with screen readers in mind. Another problem is that the information, which is desired by the user, may be located at the end of the page, forcing the reader to go through a potentially large chunk of less relevant information line by line. Usually the screen readers are able to skip to headers on a page to help alleviate some of the navigational issues [**?**].

1.2 Motivation

A study of blind users frustrations on the Internet with current screen reader technologies shows that the three most frequent causes of frustration are inaccessible Flash/PDF, conflict between screen reader and applications, and page layout causing confusing screen reader feedback [The University Library, 2015]. Other minor frustrations include pop-ups, misleading links, missing descriptive text for images and screen reader errors. The participants in the study reported an average of 30.4% time lost to such frustrations. These issues needs to be solved to reduce the amount of time wasted on the Internet for visually impaired users. This can be reduced by making the Internet more accessible and creating better software for visually impaired users to navigate and get information from the Internet.

Instead of making a general application that can access all web pages, but possibly would have the problems

stated above, an application tailored to a single domain may reduce frustration of the users.

The most frequently used websites, generally fall within four categories: general search engines, social media, online marketplaces, and knowledge bases [Alexa, 2015]. This indicates that searching, online social interaction, shopping, and knowledge acquisition is commonplace among internet users. We make the assertion, that these common trends holds for the visually impaired as well, as the needs of these individuals are assumed to be aligned with the needs of people with normal vision.

Social medias and online marketplaces both use pictures in context, that can not otherwise be conveyed through text, while pictures on knowledge bases often includes a description. The categories were narrowed down to search engines and knowledge bases, where Wikipedia was chosen for its uniform article design and amount of articles available.

1.3 Available Accessibility Tools

The different tools available to visually impaired people are screen readers, braille terminals, screen magnification software, and speech recognition software.

Screen reader software reads out the content displayed on the monitor with a synthesized voice [Wikipedia, 2015j]. Usually they have their own hotkey layout, with buttons for navigating around the different application.

Braille terminals are equipped with small pins, that can raise and lower in different combinations, to display the information on the monitor to the user. Refreshable braille displays cost from $3.500 and up, depending on the amount of characters they can show at a time and the quality [AFB, 2015].

Screen magnification software are used by people with low vision, and are able to magnify the content on the screen. Screen magnifiers feature colour inversion, smoothing of text, and cursor customisation [Wikipedia, 2015g].

Speech recognition software is a technology still in heavy development. It takes speech from the user, and translate it into text. These systems are often embedded into other technologies, for example search engines [Wikipedia, 2015i].

The time wasted by visually impaired people due to problems with current accessibility tool technologies, may be reduced by creating new software for accessing information on the Internet. In this documentation we will describe a software application, VC Wiki Reader, that more specifically, is designed to hopefully reduce the amount of problems when looking up information on Wikipedia.

2 Problem Analysis

The motivation from the earlier chapter will be used to further specify what requirements have to be satisfied to aid visually impaired people with gathering information through the Internet. First, the problem domain will be specified to narrow down on the points of interest. After reaching a better understanding on the problem of web accessibility, a problem formulation have been detailed.

As mentioned in Chapter 1, visually impaired people have problems with current accessibility technologies when trying to access the Internet. These problems can be split into three problem areas. As illustrated in Figure 2.1, the problems faced by the users, while searching for information, can be split into entering input, receiving output, and accessing Wikipedia. The input symbolises the user's interaction with web accessibility tools to enter in commands for searching and navigating Wikipedia. Wikipedia symbolises the online server system, where articles are stored. Output, similar to input, symbolises the presentation of articles.

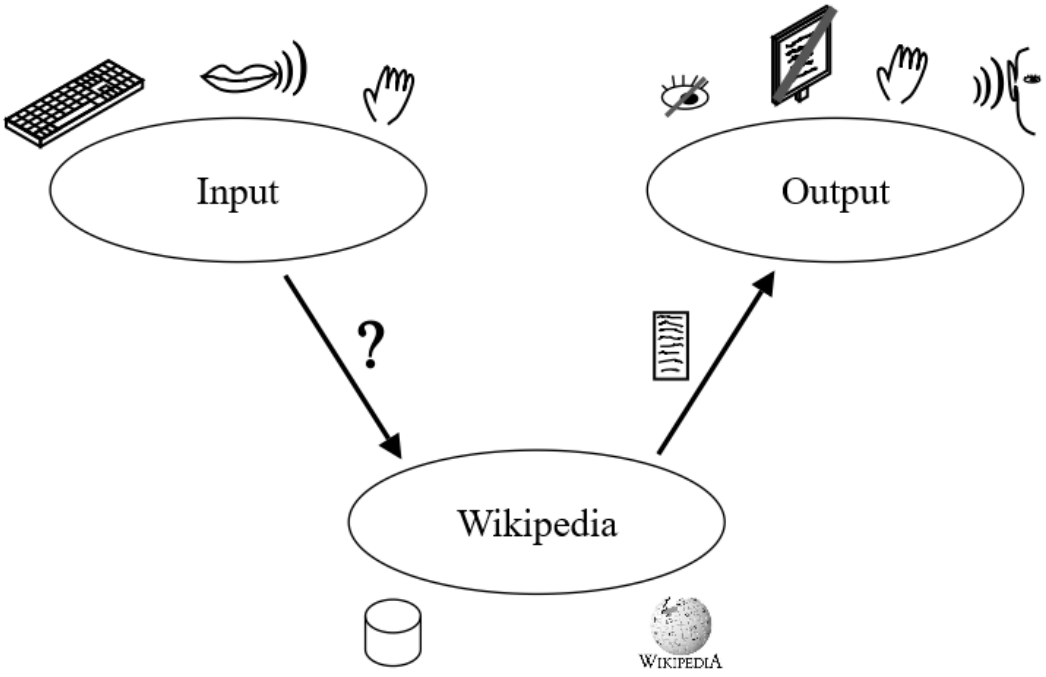

Figure 2.1: Graph of the problem domain

To further explore the requirements for the systems used for input and output, the principle of web accessibility will be elaborated upon.

2.1 Web Accessibility

Visually impaired people use Text-to-Speech (TTS) software to navigate the Internet, using mainly keyboard and audio, as they may not be able to fully perceive their monitor. As described in Section 1.2 there are problems with current accessibility technologies, and these problems could be reduced if VC Wiki Reader was designed with web accessibility in mind. Using some key web accessibility principles [WebAIM, 2015b], our software must be:

- **Perceivable:** The users must at any moment be able to tell where they are in VC Wiki Reader and the article. This is important as they are not able to use their monitor. If the users gets lost during use of the application, it can be difficult to recover, and they might be forced to restart the program.

  There must be a way for the user to get access to commands, and learn about the navigation options available as they use the application.

- **Operable:** The users must be able to navigate the different articles and jump to headings in the active article. The system must be operational similar to current accessibility software technologies. Braille terminals will not be supported because of the high expense in acquiring such a device.

- **Understandable:** Visually impaired users, have no perception of how text is presented on the screen. VC Wiki Reader should read Wikipedia articles and present the relevant information by reading it aloud. Links should be distinguished from the text, and presented in a manner that gives the user clarity.

- **Robust:** Having a robust system is already essential when working with untrained users, and is even more important for visually impaired people, as they are not able to see if the system crashes or behaves unintentionally.

These key principles were taken into consideration when designing and implementing VC Wiki Reader, to ensure accessibility for visually impaired users. With these principles of web accessibility outlined, a problem formulation was specified.

2.2 Problem Formulation

Looking up information on the internet can be second nature for many people; however, for visually impaired people, it is a difficult task to accomplish with current technologies. The problems found in Section 1.2, can be summarized to this question.

How can we make a solution to the problems faced by visually impaired people when they look up information on Wikipedia?

- How do we make searching for information on Wikipedia more accessible for visual impaired people?
- How do we receive input from the user?
- How should the results be presented?
- What information should be presented?
- How should the users navigate through articles on Wikipedia?

To create a tailored solution to the problem stated above, and to ensure quality, some restrictions and requirements were specified.

2.2.1 Restrictions and Requirements

A number of restrictions and requirements were created to guide the analysis and design towards higher quality. These specifications were created in a pursuit of solving issues with current technologies, and increase web accessibility to Wikipedia. These points describe requirements for both the server and client parts of the application. Following are the requirements specified for VC Wiki Reader:

- It requires access to a database of Wikipedia articles.
- It is a proof of concept, as such, the server is not required to be hosted for the public, and is not required to handle heavy traffic.
- It should be able to present Wikipedia articles in an understandable manner.
- It should be able to run on a modern Windows platforms.
- The user should be able to search for and navigate through the articles.

As this is intended to be a proof of concept, no elaborate user tests will be performed.

Implementing these functionalities should provide visually impaired users with an alternative method for looking up information on Wikipedia.

3 Analysis

With an outlined problem formulation, and restrictions and requirements, an analysis on accessibility software, how to handle Wikipedia articles, and how to connect a client and server is detailed. This analysis will mount to a choice of technologies to use for this project.

On Figure 3.1, also seen in Chapter 2, is a graph of the problem areas that affect our solution. There are five points that were required to be analysed further, before a design could be made for the system.

- **Point 1**: symbolises the input channel from the user to VC Wiki Reader. This channel requires further analysis of the current technologies, to decide on an appropriate method for entering search queries, and on navigation of articles.
- **Point 2**: symbolises the output channel from VC Wiki Reader to the user. Like input, it requires further analysis of the current accessibility technologies, to make the correct decision of how and in what format Wikipedia articles should be presented in.
- **Point 3**: symbolises Wikipedia, where the articles are stored. This should be analysed to find an appropriate way of storing, accessing, and retrieving the articles.
- **Point 4 & 5**: symbolises the connection between client and server. How the requests are send to the server, and how the responses are returned to the client.

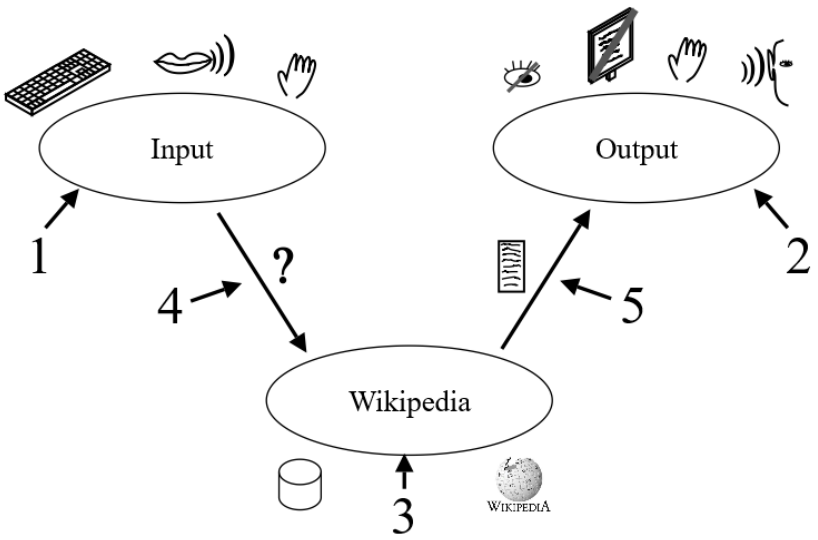

Figure 3.1: Graph of the problem domain

3.1 Accessibility Software

The market for accessibility software have been dominated by JAWS for years, but within the last few years, three other accessibility softwares have started gaining popularity, namely ZoomText, Window-Eyes, and NVDA. In a survey conducted by WEBAIM in July 2015 the distribution of accessibility software usage was as seen in Table 3.1 [WebAIM, 2015a].

Application	Share
Jaws	30.2%
ZoomText	22.2%
Window-Eyes	20.7%
NVDA	14.7%
Other	12.3%

Table 3.1: Accessibility software distribution

It is important to note that this distribution does not take into account whether the user is blind or has low vision. When these conditions are factored in, 53,5% of those with low vision uses ZoomText, while 38,9% of those that are blind uses JAWS. This indicates that ZoomText is preferred by those with low vision, while JAWS is preferred by those who are blind. The four dominant accessibility software will be further outlined below.

3.1.1 Jaws

JAWS was one of the first screen readers on the market, and to this day it is still being supported and updated. Important features include output through TTS, and refreshable Braille interface, with a web-mode for use with Internet Explorer, and navigation using key commands. In addition JAWS allows the user to configure a number of settings, giving a high degree of customization [Wikipedia, 2015d].

3.1.2 ZoomText

Was at first made to make it easier to read text on the screen by magnifying the content on the screen, but has later also released versions that included a screen reader. Important features include, dual monitor support, magnification of the screen up to 36 times, finding hyperlinks and controls on websites, turning documents/web pages into a mp3 sound file to listen to at a later time, reading a text in the background while navigating other websites, reading aloud the keys the user presses, and saving unique settings for individual applications [Wikipedia, 2015m; ZoomText, 2015].

3.1.3 Window-Eyes

Window-Eyes supports Microsoft's controls and can be used with most applications without any extra configuration changes required. It supports output in the form of TTS in multiple languages or as Braille [Window Eyes, 2015].

3.1.4 NVDA

NonVisual Desktop Access is a prominent free screen reader that offers the basic functionalities of a screen reader. It supports output as both TTS and Braille, and gives support for several browsers [Wikipedia, 2015f].

3.1.5 General Features

All the accessibility software support a TTS output, while some of them also support Braille output. Additionally they all support some form of input in the form of key commands to navigate; however, none of them support speech recognition. Using Speech-to-Text (STT) could possibly enhance the user experience.

3.2 Speech Recognition & Text-to-Speech

To find which tools to use for TTS and STT, we analysed and compared different solutions. For these comparisons, the focus was aimed mostly at free TTS and STT solutions. While paid solutions may offer more than the free solutions, they will not be covered as much as that would require additional funding. It should be noted that the TTS quality comparisons, are done subjectively and is not necessarily a universal preference.

3.2.1 iSpeech

iSpeech provides TTS in multiple languages, for example English and Mandarin. These languages may have multiple dialects and gendered voices to choose from, which is convenient for personal preferences. iSpeech is a subscription based service, where the provider limits the functionality of the service based on the subscription. In the case of iSpeech, there are three personal packages available, two pay-per-month and one free package. The free option allows for a limited number of voices and limits how much audio is returned from the service, the same goes for the two other packages which allow for longer audio samples. It provides its functionalities through a number of APIs, including .NET [ISpecch, 2015b]. In addition to the TTS capabilities of this service, it also provides an STT service for dictating text [ISpecch, 2015a].

3.2.2 AT&T Natural Voices Text-to-Speech

The AT&T solution provides eight English and two Spanish voices. Like the prior solution, it only provides paid solutions, including a paid trial. The quality of the AT&T TTS is a decent, but synthetic voice [AT&T, 2015].

3.2.3 Microsoft Speech API

The Microsoft Speech API (SAPI) is a free widely available API for the Windows platforms, it provides support for a few language and voices, with additional available for download [Microsoft, 2015]. Like with the AT&T the voices are decent, but also synthetic, however it also differs from platform to platform. The speech recognition is well integrated in the Windows platforms and is accessible through well documented C# libraries. The libraries uses the integrated speech recognition in the platform, which provides functionalities to improve the recognition through training of the speech recognition engine. This means that an untrained engine may not perform as well as a trained, which leads to a difference in usability of the STT. As for the TTS part, it uses the integrated TTS engine in Windows, which comes with voices in a multitude of languages, more than covering the needs for the proof of concept we were developing.

3.2.4 NeoSpeech

NeoSpeech, like iSpeech, provide multiple languages in several dialects and genders. It is also available in a number of subscription packages, with the free being limited to 100 words per request, and audio output is limited to the 16bit PCM wave format. The free package does, in contrast to iSpeech, have build-in advertisement for their product. It provides its functionalities through a number of C-like APIs, including .NET [NeoSpeech, 2015]. .

3.2.5 Acapela VasS

Acapela Voice as a Service is a cloud web based service that provides TTS. It offers a number of languages spoken in the western hemisphere with no limitations to the input and output; however, it does not provide a free solution, only paid and an evaluation solution [Acapela VasS, 2015].

3.2.6 Ivona

Ivona is a Java based service which provides multiple western languages in a number of voices. It, like Acapela VasS, it only offers a paid solution. Ivona provides a free solution, which can be used during the application development [Ivona, 2015].

3.2.7 Summarization

iSpeech provides an STT and high quality multilingual TTS solution at a relatively low cost, while most other solutions are comparatively much more expensive. AT&T provides only two languages subjectively sounding very synthetic. SAPI provides a free STT and multilingual TTS solution, which, as the AT&T subjectively sounds very synthetic. NeoSpeech, provides a good quality TTS, however the limitations imposed by the free subscription could easily become an annoyance to the end user. Acapela VasS and Ivona are two expensive solutions, which however provide fewer voices and languages and a lower quality TTS than most of the other solutions.

Solution	**Free**	**STT**	**TTS Quality**
iSpeech	Yes. Each returned sound file is max 1 minute long.	Yes	High
AT&T	No	Yes	Very synthetic
SAPI	Yes. Is included with a windows operating system.	Yes	Varying
NeoSpeech	Yes. 100 word limit per request.	No	Good
Acapella VasS	No	No	Poor
Ivona	Yes. For development only.	No	Poor

Table 3.2: Comparison between the different solutions for TTS and STT

3.2.8 Choice

We have decided to work with the SAPI. This was decided based on its availability and its ease of access in C#. It provides both STT and TTS, which also uses the same syntax. This eases the development by reducing the amount of time spent on learning the syntax for the solutions. Knowing that the recognition engine is likely untrained we would have to account for that in some way during development.

3.3 Accessing and Storing Wikipedia Articles

VC Wiki Reader should be able to search through all of the articles and only show the relevant results, and do this within a fair amount of time. In addition, because of the strict deadline of this project, the solution to this search problem is preferred to have low development time. The results of the search should be precise, but does not require to be able to guess the correct results in case of bad search keys.

3.3.1 Crawler

Solving the search problem by designing and implementing a crawler would allow us to optimize the search engine to our needs, at the expense of additional development time. Basic crawler designs could be used and refitted to cut down on this extended development time albeit ensuring politeness could be a greater concern. While the politeness might be simple to ensure in the long run, as crawling just once a day could be sufficient, it would limit testing during development. In addition we would have to ensure the correctness of the query-matching ourselves, which could extend the time spent testing. The benefits of creating a new crawler is the increased opportunity for expanding this project in the future. Adding future domains to the list of searchable domains or even creating a general information searcher for visually impaired people would be easier if the web pages were crawled and indexed by a crawler designed for this task.

Pros	**Cons**
Optimizable for our needs	Extended development time
Opportunity for expansion to other domains	Politeness limits testing

Table 3.3: Pros and cons for crawler

3.3.2 DBpedia

DBpedia is a relational database with data from Wikipedia. The data is structured in a RDF structure allowing for efficient querying. As the structuring of the database is crowd-sourced there are inconsistencies in how exactly the data is organised and in the naming of the elements. So while the queries might be efficient, it also requires an extensive knowledge of the internal structure to ensure the correct results. As we can not expect our users to be experts in the art of querying, we should ensure this correctness ourselves through coding. As the naming of the elements of the database is not guaranteed to be consistent, this could hamper the scalability of this code. Currently the servers, that DBpedia uses, are slowed down due to the amount of traffic their site have. For our implementation we could instead use a snapshot of their database hosted on our own server to provide faster querying [DBpedia, 2015].

Pros	**Cons**
Efficient Querying	Inconsistencies in the structure
Have a server providing the data	Based on outdated wikipedia dump

Table 3.4: Pros and cons for DBpedia

3.3.3 Solr

Solr is an open source, enterprise search platform, where documents are indexed via JSON, CSV, XML or binary. The querying happens via HTTP and the results are returned as either a JSON, CSV, XML or binary file. This is great for representing the search results, since a number of programming languages are capable of handling these formats. Amongst the important features are indexing of rich text content (PDF, Word documents) and near real time indexing [Apache Lucene, 2015].

Pros	**Cons**
Efficient querying	Storing the server takes a lot of storage space
Near real time indexing	Based on outdated Wikipedia dump
Compatible with many programming languages	

Table 3.5: Pros and cons for Solr

3.3.4 Choice

The criterias for the storage and indexing solution, are quick querying, short development time, and precise results. In Table 3.6, is a comparison on five factors for the different solutions. The five factors are how configurable the choice is, the development time, how expandable is it, how quick the querying is, and the amount of resources consumed. The scale goes from *Low* → *Medium* → *High*.

Criteria	**Crawler**	**DBpedia**	**Solr**
Configurable	High	Low	Medium
Development time	High	Medium	Low
Expandable	High	Low	Medium
Querying	Low	High	High
Resources	High	Low	High

Table 3.6: Comparison between the storage and indexing solutions

When choosing what to use, the querying part is important, as well as development time. A crawler would have too much development time. DBPedia and Solr are both good in querying and the development time is not that much for either. Resources are not an issue as a local server is available to use. As Solr is more configurable and expandable, than DBpedia, we decided on Solr as the best option.

3.4 Wiki Markup

Solr was chosen to store and index a Wikipedia dump. The Wikipedia dump contains a number of articles, which are written using the wiki markup language, therefore an analysis of the wiki markup language is needed. As wiki markup is not necessarily a format that is easily presentable, it may be necessary to parse some parts of it, to get a presentable format. This will be used later for when we design our parser.

3.4.1 Different Components

The wiki markup language have several different ways of describing content and through multiple different methods. Most of this section have been written based on information from Wikipedias own help page about the wiki markup [Wikipedia, 2015c]. Further in this section we describe some of the most important types of wiki markup code, and if we think it should be parsed.

Text formating

Like HTML, wiki markup describe some of the formating by encapsulating it with some special characters. This include heading, bold, italic, etc.

Headings are encapsulated in "=" and the number of consecutive "=", is the power of the heading.

- =Heading= → < $h1$ > heading < /$h1$ >
- ==Heading== → < $h2$ > heading < /$h2$ >

Bold and italic uses the single quote sign, where italic used two consecutive single quotes and bold uses three. For bolditalic it is five single quotes.

- "italic" → *Italic*
- "'bold"' → **Bold**
- """'BoldItalic"""' → ***BoldItalic***

Lists

For lists, wiki markup uses prefixes. The types of lists wiki markup uses are ordered-lists, unordered-lists, and definitions. To describe these, wiki markup uses some special characters as a prefix. These special characters are "#", "*", and ":" which are used for ordered, unordered, and definition lists respectively. The amount of special characters in the prefix, describe how much the list should be indented, and the last special character how the list should be shown. A list stops when a new type of code is started on a new line, or when there is a line of only whitespace.

Tables

In wiki markup, tables are described using a combination of encapsulation and prefix. The table itself is one big encapsulation, and all the rows and cells, are described using a prefix notation [Wikipedia, 2015b]. A table start with "{|" and ends with "|}", each on a separate line. The "{|" can also have some style code on the same line.

On the second line of the code, there is often a caption for the table. This caption is indicated by "|+" at the start of the line.

Rows in the table is indicated by "|–" on its own line. The "|–" can have some style code after it on the same line.

The separation of cells can be written in three different ways. With a "|" and content on a line, and then a newline and the same for the next cell. With a "||" between the cells on the same line, or with a newline, before the "||". Headers for tables are indicated with a "!" at the start of the line, and can include some code, to indicate if it is a row or column header, and a "|" to separate the code from the text.

An example of a table is provided in Listing 3.1.

```
1  {| class="wikitable"
2  |+ Table Caption
3  |-
4  ! scope="col" | Column 1
5  ! scope="col" | Column 2
6  |-
7  | Cell 1 || Cell 2
8  |-
9  | Cell3
10 | Cell4
11 |}
```

Listing 3.1: Wiki markup example

This code would produce a table similar to Table 3.7.

Column 1	**Column 2**
Cell 1	Cell 2
Cell 3	Cell 4

Table 3.7: Example of a wikitable

Special characters
Not all letters in the wiki page is written directly as seen in the browser. Some of the more uncommon are instead written as special characters. Special characters are in wiki markup written like the character encoding in html, with a prefix '&', a name/code in the middle, and a ';' as a postfix. For example © = ©. In addition to this newer versions of the wiki markup language also supports UTF-8 characters.

Math
Wiki markup uses two ways to write math. $< math > a2 + b2 = c2 < /math >$ and {{math| $a2 + b2 = c2$ }}. Because math is very complex, we have decided not to include the math in the parser for our proof of concept, and will instead use these encapsulations to identify when there is some math to skip.

Auto Generated content
Some of the content on a wiki page, is not generated directly from some specific code. It is instead auto generated after all the code has been looked through. Some examples of this is the table of content and the list of references. We do not plan to make the auto generated content for our proof of concept.

Links
Wiki markup has three different types of links, a urls, internal links, and external links. A url is just a link to a website written in plain text. Internal links are links to other Wikipedia articles, and is written by encapsulating a pagename in double square brackets, like this "[[pagename]]". External links are links to other websites, and are just a url encapsulated in a single square bracket like this "[http://website.com]".

Variables
For data that might change over time, wiki markup have variables. These variables include, what day and week it is, and how many wiki pages exist.

Variables are in the wiki markup language written with a double namespace indicated by {{VariableName}}, as for example {{CURRENTYEAR}} which returns 2015, for the time this article was published.

3.5 Client Server Communication

Communication over the Internet can be established in different ways. The primary concern is to analyse and decide which technique to use. The techniques analysed in this section are an ordinary web service, Window Communication Foundation (WCF) and Solr's own API as a mean to contact a server from a client in order to execute a search query.

3.5.1 Web Service

The web service offers a single point of contact. The request is sent by a device with a web service application installed and a functioning internet connection. The web application service, then pass on the search string to Solr via the local network. The Solr server returns the results as a XML document and the web application server simply passes it on.

The web application service has a single public webmethod available, which is a Query, that takes exactly one search string. In order, for the client, see Figure 3.2, to know the availability of webmethods on the web application service, it needs to know the interface of the web service on web application server. This interface contains the information about the available methods, the parameters, and return types. Furthermore the client requires an endpoint, this acts as a contact point to the web application server. This is configured in the application configuration file, which is expressed in XML. However, when this is setup, the client can call the Query method as if it was local. The communication goes through the HTTP protocol [Microsoft, 2015c].

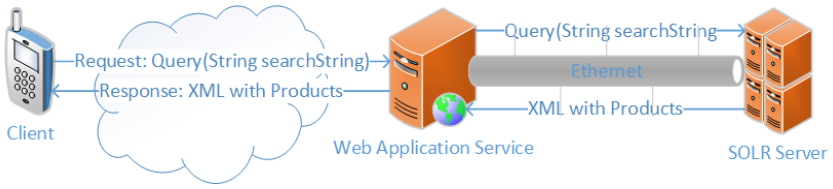

Figure 3.2: Overview of a web service using Solr

3.5.2 Windows Communication Foundation

WCF is the successor of the web service. It offers the same functionalities as an ordinary web service, but includes additional features. As the list of features is exhaustive, only the most important features will be mentioned [Microsoft, 2015b; Codeproject, 2015]. The features are as follows:

- The old XMLSerializer has been replaced by the new DataContractSerializer, which has better performance, is more configurable, and can serialize objects that the XMLSerializer cannot. For example, classes that implements the IDictionary interface such as Hashtables are not serializable with the XMLSerializer, while the DataContractSerializer can serialize them.
- WCF can also use TCP and send streams of binary data from one endpoint to another endpoint asynchronously.
- The security from web services has also been improved. It is now possible to use X.509 certificates that ensures authentication.

3.5.3 SolrNet API

At the heart, Solr is a web application. It is build on open protocols, meaning any client application is able to connect to a Solr service. Solr uses a RESTful service, which is more lightweight than using a SOAP service. Solr communicates via HTTP, and client applications can send HTTP requests and parse the responses. Queries are executed by generating an URL with the query parameters. Solr then process this request and return the results [Apache Lucene, 2015].

3.5.4 REST vs SOAP

REST and SOAP are two methods used by web services to exchange information. REST, seen on Figure 3.3, requires that both the client and the server have a shared interface. This interface contains all the available methods that the server offers. REST provides a lightweight alternative to SOAP. Instead of using XML to make a request, REST relies on a simple URL in most cases.

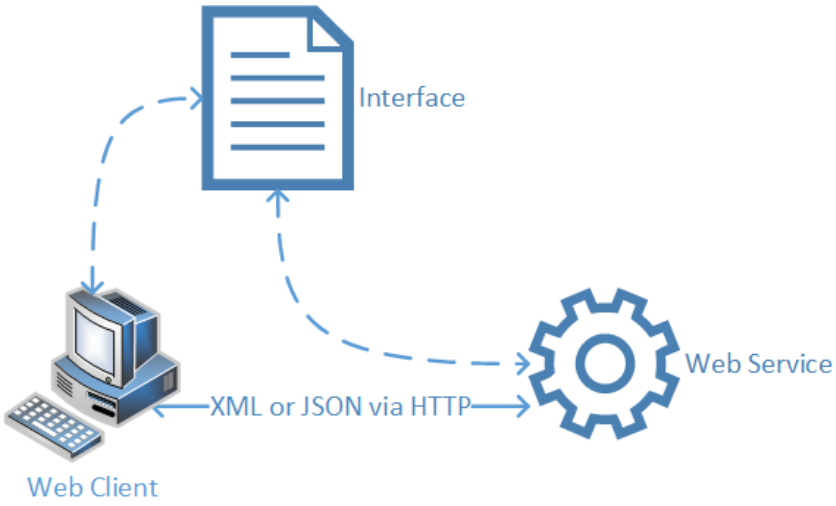

Figure 3.3: REST service

SOAP, seen on Figure 3.4, does not use a common interface between the client and the server, but relies on the server to understand the request from the client and return an appropriate response. The message of the request could be given in the URL, for example, http://localhost:31933?query=title:AlfredHitchcock.

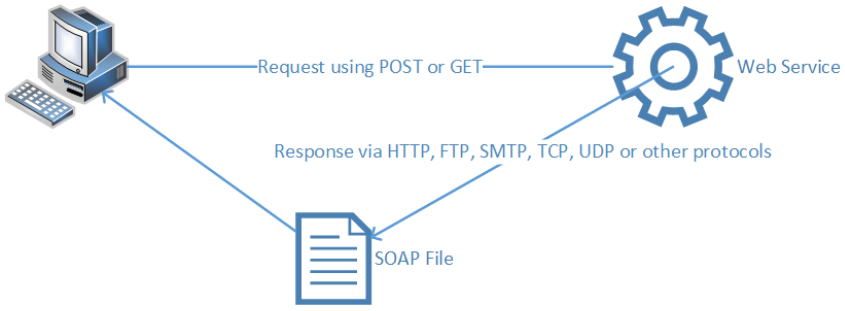

Figure 3.4: SOAP service

Architectural Constraints:	REST	SOAP
Scalability	High	Medium
Security	Medium	High
Reliability	Medium	High
Interoperability	High	Low
Performance	High	Low
Transactions	Medium	High
Multi Device	High	Low
Multi Transport	Low	High
Time to market	High	Medium

Table 3.8: REST vs SOAP comparison

Table 3.8 shows a comparison of the constraints on REST and SOAP. These constraints were compared to the requirements posed by VC Wiki Reader to make a decision of Scalability, reliability and performance, which are all deemed to be important. Most of the requirements to the data transfer method are on high in REST. Because we wished to use the SAPI with its C# interface, we found it obvious to also use a similar C# API for communicating with Solr, hence using the SolrNet API, which uses the REST protocol. In addition to this, the performance and scalability of REST are higher than SOAP's, which are some of the criterias that are considered preferable for VC Wiki Reader, resulting in the SolrNet API with its REST-based communication being chosen for further development [infosysblogs.com, 2015].

4 Design

In this chapter, the design of the system will be shown and explained. This chapter contains an explanation of the representation of articles in VC Wiki Reader, the voice dialog command system, the architecture of the system and its components

4.1 Presentation of Articles

The visual impaired requires a different representation of articles than people with normal vision. Typography and text structure is not conveyed well into speech. For the semantics of Wikipedia articles to be correctly received by the user, the different presentation techniques should be handled accordingly.

4.1.1 Typography

The applied typographical techniques impact the way an article is written. The application should handle different techniques accordingly, to convey the information correctly. The techniques handled by VC Wiki Reader was limited to the most common ones found in Wikipedia, that still have an effect on how an article is read. These are emphasis, symbols, arrangements, and structures. Typefaces, text size, line length, style, colour, etc. will therefore not be included. The following is a list of the techniques interesting for VC Wiki Reader, and the solutions that will be used:

- **Bold** & *Italic*: is read with exaggeration of the text. It attracts the attention of the reader and should therefore also be read with extra weight in comparison to the rest of the article. To simulate bold and italic, the text will be read aloud with a slightly lower pace and a deeper pitch.
- ALL CAPS: is used to emphasise single words or sentences. Another usage is for acronyms and entity names. Companies such as MSI can be written in all caps. ALL CAPS will not be simulated in the application as there are no easy solution for differentiating between acronyms, entity names, and emphasising usage.
- Special characters: are used by Wikipedia. Some of these symbols have written names that are commonly known such as π (Pi) while other symbols such as $\triangleright$, triangle right arrow or bullet right arrow, can be found, but are much harder to convey through speech. There are too many symbols to handle in the proof of concept. A suitable solution should be able to read and convey the information of the a few of the common symbols found in the articles. The application will read the symbols that are allowed in the TTS solution.
- Arrangement and structure: An article can be written in a single section with all text, or with the text split up into smaller parts with different sizes and fonts. In some articles, highlighted text can be found with quotes or other significance. These sections have to be read in the correct order and in the case of self quoting, not read at all. Fortunately for VC Wiki Reader, all articles are written under the same format, creating an easy order of text pieces to follow.
- Headings (title, subsections): To differentiate headings from the remaining text, a short pause will be placed before and after the heading/subsection. This will highlight the beginning of a new section.

4.1.2 Links

In Wikipedia articles, links are highlighted with blue text, and underlined when moused over. This is the HTML standard that most sites follow. When links are read by the most common accessibility softwares, it is often that they will say "link" followed by either the link name or address. In Wikipedia, links can be arbitrarily long and each sentence can contain arbitrarily many links. It is not uncommon to find sentences with four or more consecutive links when listing items. Our approach was to simply not announce the links, however with a command, the user could have all links from the current paragraph listed.

4.2 Voice Commands

As our system will be voice based, it was important to have all functionalities accessible through spoken commands. The commands should allow the user to navigate in a similar fashion to using a screen-reader, while adding a commands for searching the articles on Wikipedia. The users should be able to navigate VC Wiki Reader using these commands:

- "Pause reading" Pauses the TTS. The system will continue to listen for other commands.
- "Resume reading" Starts the TTS if it is paused.
- "Back to start" Jumps to the start of the current article.
- "Next section/paragraph" Jumps to the next section or paragraph in the current article.
- "Previous section/paragraph" Jumps to the previous section or paragraph in the current article.
- "Repeat section/paragraph" Restarts the TTS on the current section or paragraph.
- "List links" List all links in the current paragraph.
- "List content" Gives a list of sections and headers in the article.
- "Search for" General search command that starts the search process. The search term spoken after "search for" is used for the query.

Using this list of commands it should be possible to navigate an article, to quite accurately get to any given part of the text, as long as the article is well structured with sections and headers. Pausing and resuming the TTS is for the convenience of the user, as they could be interrupted or distracted for short periods of time, or while they used the search command to find the article, they are looking for.

4.3 System Overview

To give a better overview of how the overall system should be designed, we made an overall system architecture, see Figure 4.1. The architecture followed a Model View Control (MVC) design, where the models was designated for data and the controllers would do all the heavy work. The architecture consisted of two parts, a client and a server. The client was a desktop application that was installed on the user's computer, and the server was a remote server, the client connected to when requesting data. As we split the system up into a client and server part, we had to choose which version of the SAPI to use, either the client or server version. The advantage of using the client side API was that we would reduce potential server workload, reduce response time, and limit scalability issues by moving most of the computation to the client side.

On the graph, the arrows specify calls to another component of the system, with the exception of the arrows in the model, which describe the relations between components. For example, a section could contain a number of paragraphs, bookmarks, and sections recursively in its structure. The arrow that lead out of the model, describe the components of the system, that stored data for a longer period of time, as opposed to a function that builds, returns, and then deletes the internal version of the data.

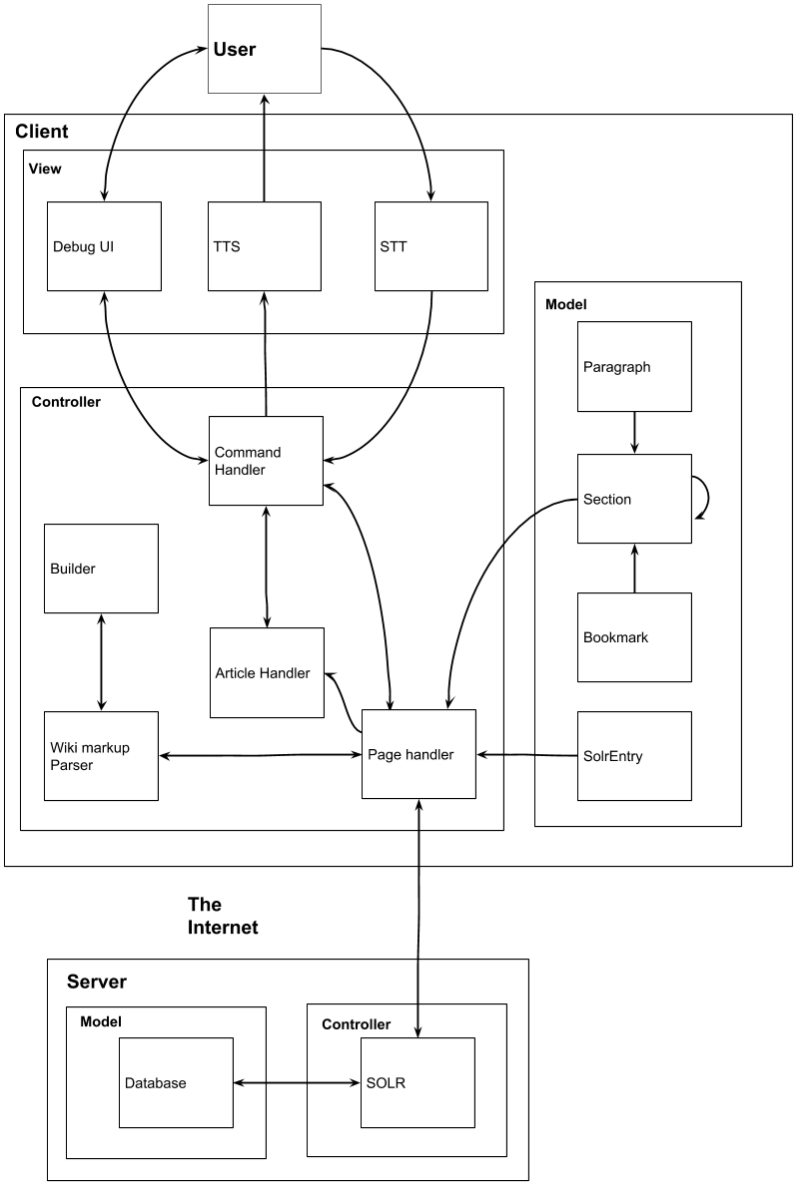

Figure 4.1: Graph of the overall system architecture

In addition to making an overall system architecture, we made a state transition diagram for the client side of the system, see Figure 4.2. This diagram shows how the system changed state, based on user input. From the initial branch of the diagram, the TTS is setup to start running as soon as an article was to be read. The second branch represents the same transitions with the difference that the TTS is paused and needs to be resumed before the reading can be continued.

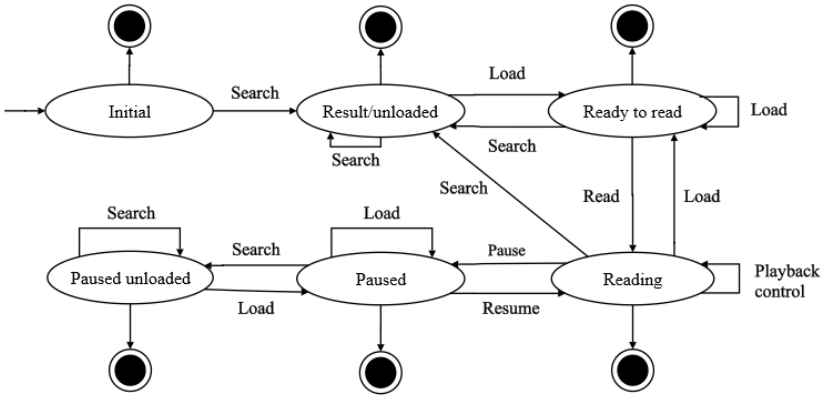

Figure 4.2: State transaction diagram of the client

4.4 Client

We decided that the client should be constructed such that the views are decoupled from the controllers, which was the reason, we designed the communication between these two layers as events and only presents an overview of the lower two layers of the MVC model. The input views communicate with the application through the command handler, which then relays the commands to the other controllers.

From the UML diagram on Figure 4.3, we see that a generalization of a IReadable object is inherited by sections and paragraphs. The section is constructed by the parser, which itself is instantiated by the page handler. The parser internally utilizes builder objects to construct the paragraphs contained in the section objects. From the page handler, a section can be provided to the article handler for the user to interact with though a command from a decoupled view, which again are being relayed through the command handler.

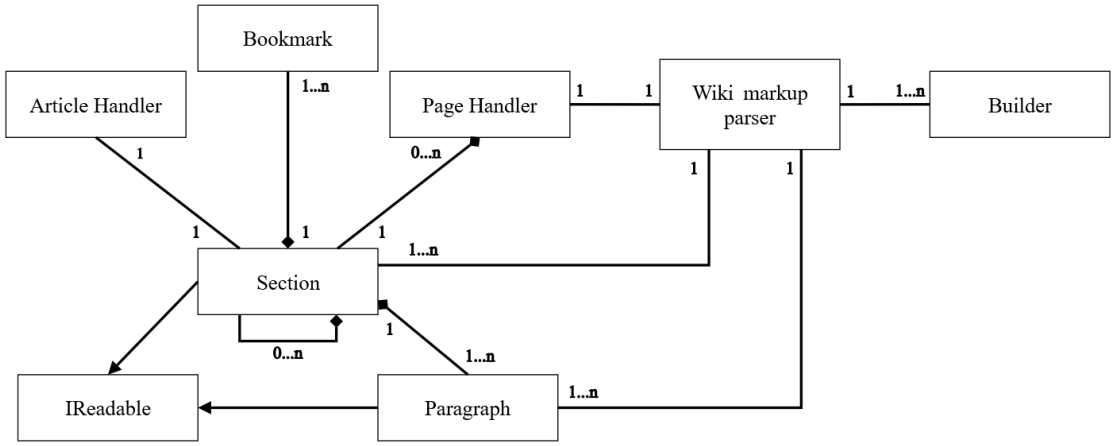

Figure 4.3: Graph showing the relation between the different components in the client.

4.4.1 Models For Representing an Article

To simplify the construction of an article, we reduced the elements of an article down to three simple concepts: a paragraph, a bookmark, and a section. The relation between these data types, is as seen on Figure 4.4.

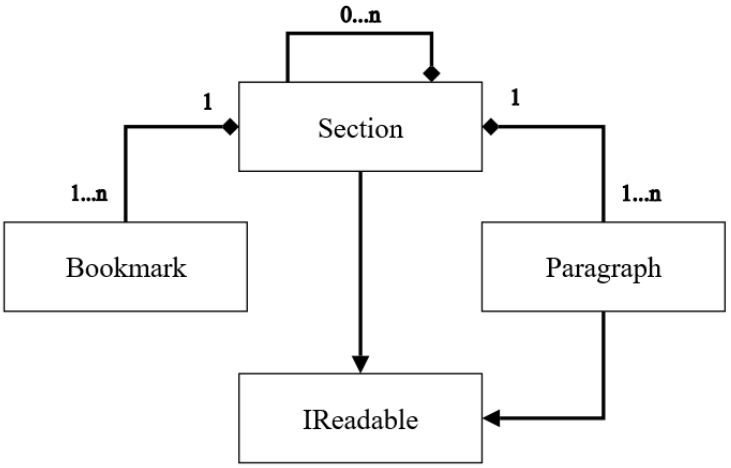

Figure 4.4: Graph showing the relation between the data types used to construct articles.

Paragraph
A paragraph represents the most basic data container needed for a single paragraph of text. In this case it provides the data in its native format for the TTS view to read aloud, as well as all links contained within the paragraph.

Bookmark
A bookmark represents a named location in an article, commonly we assumed that named locations would refer to sections, but it could also be applied to other elements such as tables.

Section
A section corresponds to the structure of a piece of written text with a caption. For any given section we wanted to be able to push any readable object onto it without having to enforce strict rules on which order such object would be stored and read. Therefore the section model contains a list of abstract objects, which can be read and are owned by the section, this includes both subsections and paragraphs. When an article object is constructed, it is a hierarchical structure of nested paragraphs and subsections, to provide an easy way of accessing specific sections or paragraphs, as well as providing easy random access, the list of readable objects are flattened into a single list of paragraphs in the root section of the article, and a series of bookmarks are constructed to index into the list of paragraphs. Once the article has been constructed, it can provide its data to the article handler through the page handler.

4.4.2 Debug UI: View

We designed a small graphical interface so it would be simpler to debug different aspects of the system without being reliant on spoken input at all times, see Figure 4.5 for the design. The interface included buttons corresponding to every command, the idea being that the buttons would each emulate a spoken input. The search button would take the text written in the search field and send it to the system as a search command. The article selector is a drop down menu allowing us to select the article we wanted from the list of articles returned by the search.

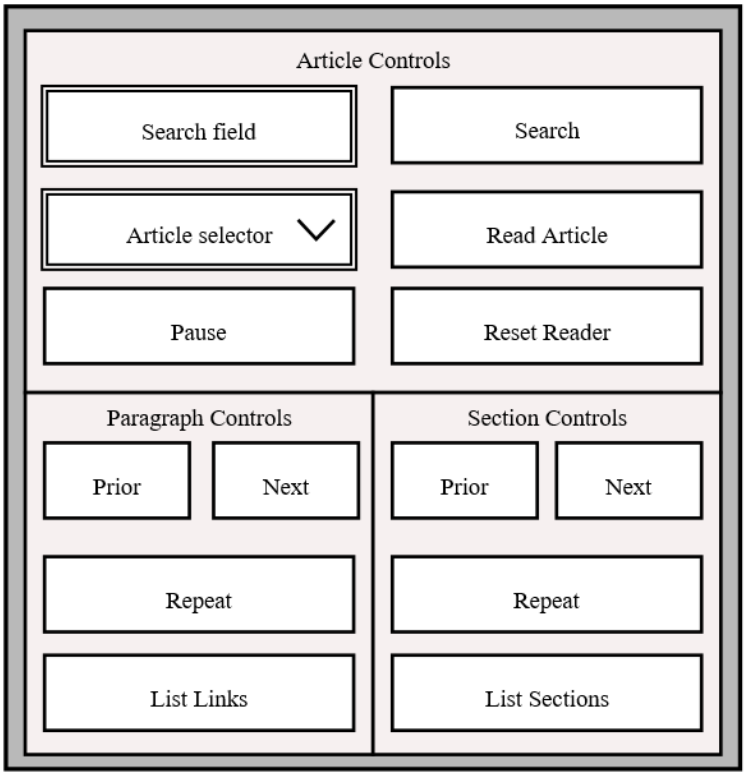

Figure 4.5: Design of the Debug UI

4.4.3 TTS: View

The purpose of the TTS was to read aloud the text in the paragraphs that had been sent to it from the article handler. Each time the TTS had read a paragraph to the end, it informed the article handler, so that it would know where the TTS was in the paragraphs. As a last functionality, the TTS should be able to pause and resume on commands sent through the command handler. All of this is illustrated in Figure 4.6.

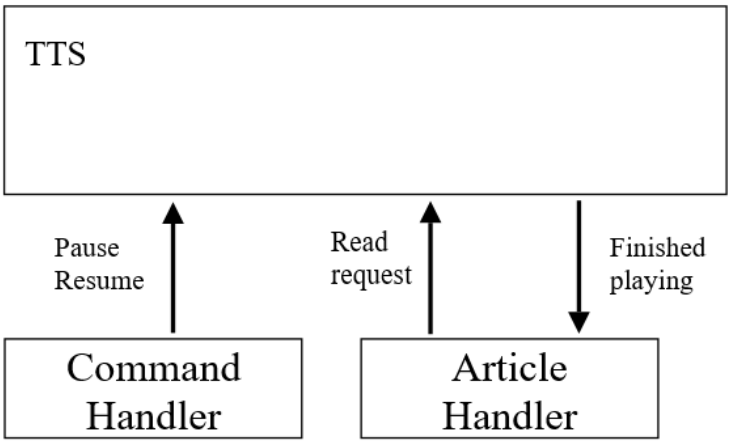

Figure 4.6: Architecture for the TTS

4.4.4 STT: View

The main input source from users was through the use of the SAPI, which had to be capable of processing the spoken commands given, as well as any words used in a search command. The functionality of the STT system would be to check whether the spoken input is a command or search.

As can be seen in Figure 4.7, the STT should receive an audio input from the user, and send it to the recognizer to verify, whether the input matches the grammar of recognized words. The recognizer should then send the recognized text to the command handler for further use.

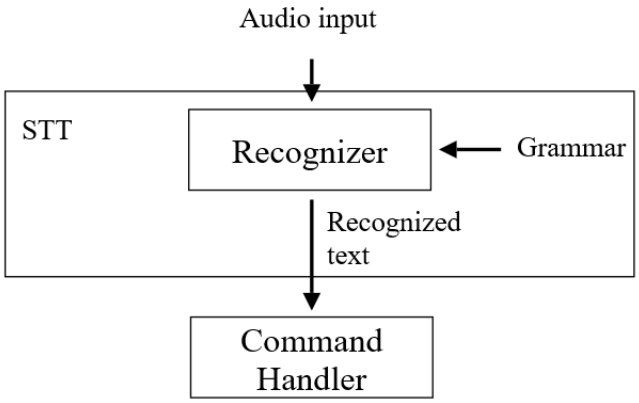

Figure 4.7: Graph of the components in the STT.

4.4.5 Command Handler: Controller

This controller acts as event relay for all commands and depending on which ones are issued, it will dispatch them to the interested listeners. In our case we have three categories of commands: article reading, searching, and playback. While commands share the same event, not all listeners may be interested in the event, i.e. the TTS will not care about the searching. Therefore the relay was introduced to provide a single point for the user inputs to bind to, that distributed them to interested listeners, see Figure 4.8.

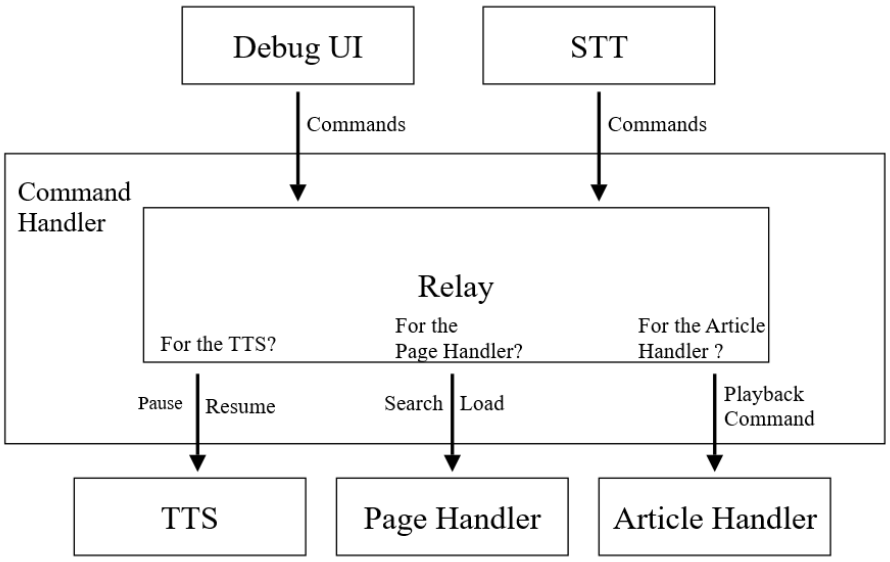

Figure 4.8: Architecture for the Command Handler

4.4.6 Page Handler: Controller

The page handler is responsible for the querying of articles from Solr, the construction of article objects returned from said query, and providing the storage and access to the articles that may be requested by the user. Internally to the controller, it will use the wiki markup parser to transform the wiki markup code to a section, see Figure 4.9.

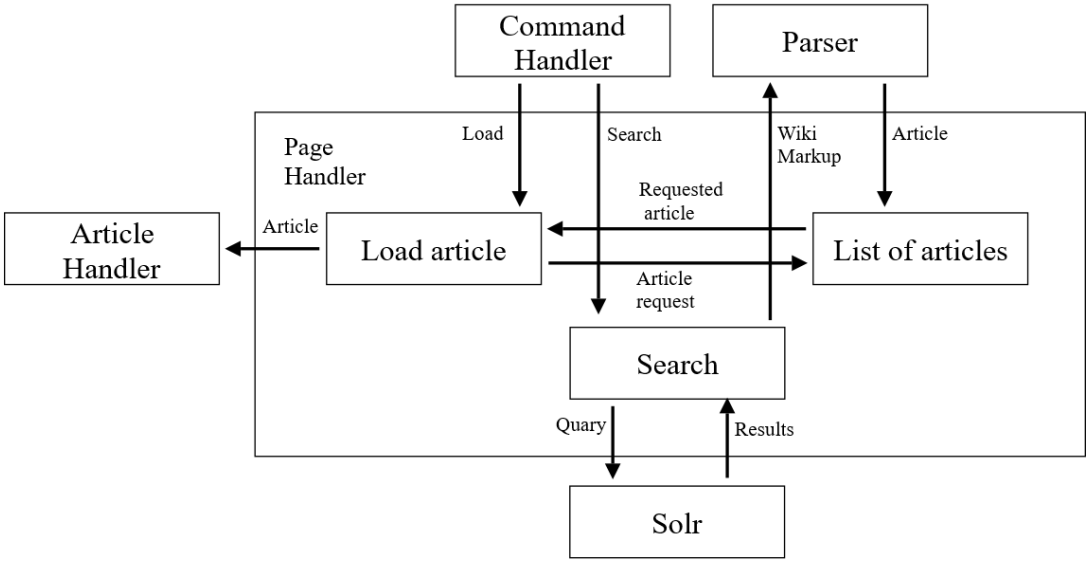

Figure 4.9: Architecture for the Page Handler

4.4.7 Article Handler: Controller

The intent for the article handler was to facilitate communication between the user input, article data, and outputs. The article handler is provided a section from the page handler, which it then, upon user request, reads aloud, see Figure 4.10. The reading of the article is done by sending the related data contained within the article to the TTS. It also listens for events from the TTS to decide if the articles internal state should be modified. Additionally it also receives commands from the command handler, and based upon the commands issued, it will either update the state of the article and return the results of the update to the reader, or provide other data contained within the article, such as the table of contents.

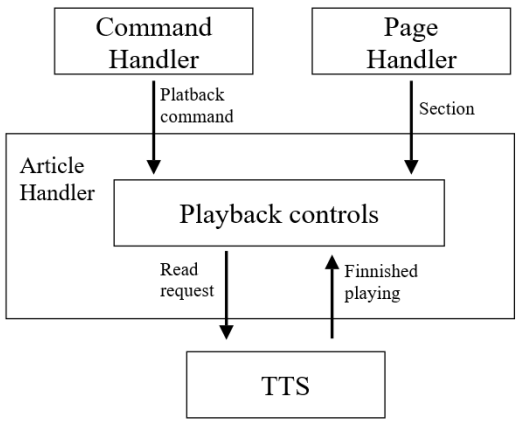

Figure 4.10: Architecture for the article handler

4.4.8 Wiki Markup Parser: Controller

ANTLR

ANTLR can, when given a grammar, generate a parser, that builds an Abstract Syntax Tree (AST), a visitor, and listener for that AST [ANTLR, 2015]. ANTLR was used in the project for two reasons. Firstly, ANTLR can be implemented in C#, matching the rest of the client, and secondly, it is not a strict parser, as it allows ambiguity; increasing readability, writability, and easing the conversion of the wiki markup EBNF grammar to one usable by the parser.

An interesting aspect about ANTLR is that it is an LL(*) compiler, meaning it is a left recursive compiler with arbitrary many lookaheads [Terence Parr, 2015]. This makes ANTLR a powerful compiler generator, as it can create parsers for a variety of different language and grammar types. The different language and grammar types include:

- Context-sensitive language
- Grammar with arbitrary action execution
- All grammars any LL(K) parsers can use

Structure of the parser

The wiki markup parser consist of several different components, some of which have been generated by ANTLR. In Figure 4.11 is a graph of the different components and their interactions.

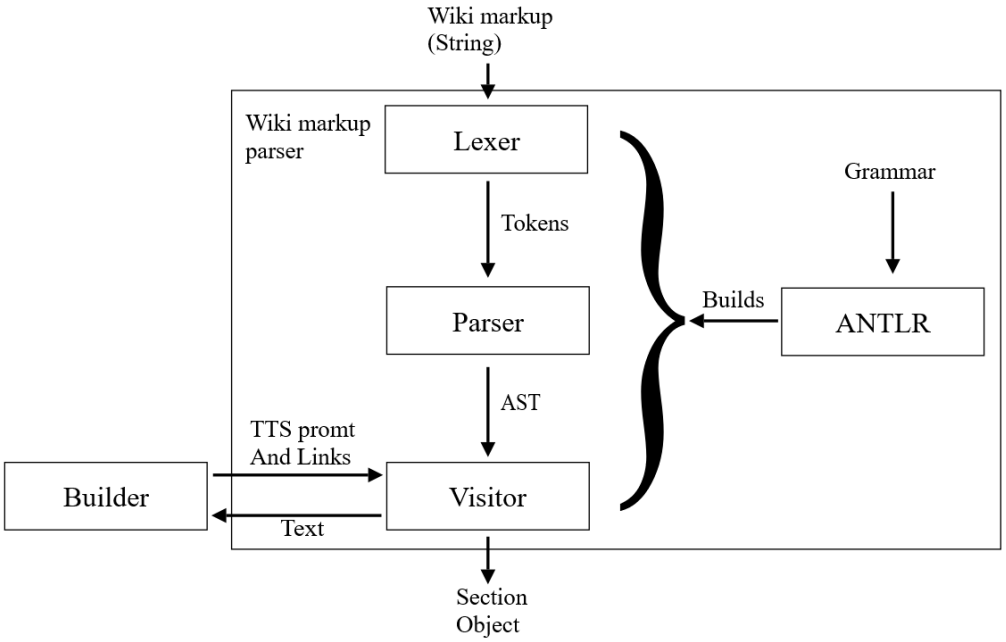

Figure 4.11: Graph of the component in the wiki markup parser, and their connections.

As seen in the graph, ANTLR receives a grammar, and builds a parser consisting of a lexer, a parser, and a visitor. The visitor build by ANTLR is only a base visitor, which has no functionality implemented other than visiting. This visitor was used to create a visitor that constructs a section when given an AST.

The parser consists of a chain of processing elements, where the lexer receives a string containing wiki markup code and generates a tokenstream.The parser takes this tokenstream as input, and generates an AST for the visitor. The visitor then visits the nodes in the AST and construct a section, that is in a format that can be used by the article handler.

Grammar
As mentioned, the grammar is used for creating the main components of the parser, by sending them through ANTLR. The grammar was based on a wiki markup grammar in EBNF provided by mediawiki [**?**]. It was however discovered that the the mediawiki grammar was incomplete. It was modified, based on the analysis of the wiki markup specifications, to better support the articles in the dump. The complete wiki markup grammar in EBNF for VC Wiki Reader can be found in Appendix A.1.

As stated in the analysis of wiki markup Section 3.4, not all components should be parsed for the user. The grammar is build from 15 different parts, eight of which should be parsed for the user and seven that should just be identified and then ignored by the visitor, which would otherwise result in lexical errors. The types of code that is visited are:

- **Headings:** Standard heading for an wiki article.
- **Text:** Regular plain text like the one in this sentence.
- **Formatted text: Bold**, *italic*, and ***BoldItalic*** text.
- **Links:** Internal and external links, URLs, and redirects.
- **Table:** Standard wikitable.
- **List:** ordered, unordered, and definition lists.
- **ISBN Numbers:** A number used for book identification.
- **Next Paragraph:** Double newline, indicating a break in the text.

The types of code that are identified, but not visited are:

- **Comments:** Comments in the code
- **Html:** Standard HTML code.
- **Includes:** Math, variables, citation, everything inside “{{“...“}}” and ”{{{“...“}}}”
- **Signatures:** Signatures about editors
- **Horizontal lines:** A horizontal line
- **Behavior switches:** Where to place table of content
- **Header links:** Similar to a comment.

Visitor
Before the visitor was able to construct a section from the AST that it received, we had to add functionalities to the visitor that was constructed by ANTLR. The task of the visitor was to visit the nodes in the AST, constructing a section from the AST. This was done by first making a section with the title of the wiki article, and populate it with nested sections and paragraphs.

To further explain how this structure should work given an input, an example for each will be demonstrated, using the wiki markup code in Listing 4.1.

```
1  == Heading2.1 ==
2  This is an example text, that show how text should look in section.
3  '''This text show the same for bold.'''
4  Lastly some text after formated text.
5
6  === Heading3.1 ===
7  Just some text to show
8
9  How next paragraph works
10 ISBN 256-452-25x
11
12 == Heading2.2 ==
13 {|
14 |+ Example table
15 |-
16 | The firs cell in the first row
17 |-
18 | First cell in second row
19 | Second cell in Second row
20 |}
21
22 == Heading2.3 ==
23 * example
24 ** of a
25 *** unordered
26 ** list
27
28 ==== Heading4.1 ====
29 [http://google.com]
30 [[Caption:page1|labeltext]]externallabeltext
31 [[Caption:page2|labeltext]]
32
33 http://www.youtube.com/
```

Listing 4.1: Example of some wiki markup code.

Headings
In wiki markup, headings have six levels. This had to be reflected in the structure of the section the visitor constructed by using a stack to keep track of the most recent section that was added. Each time a heading was visited in the visitor, the stack was reduced by adding all the headings of a higher level than the current heading visited, to the element below it in the stack. This was also reflected upon in the way text is added, as the text was added to the heading at the top of the stack. If the parser made a run with the code in Listing 4.1 as input, excluding all other types of code then the headings on line 1, 6, 12, 22, and 28, the parser should produce the data structure shown in Figure 4.12.

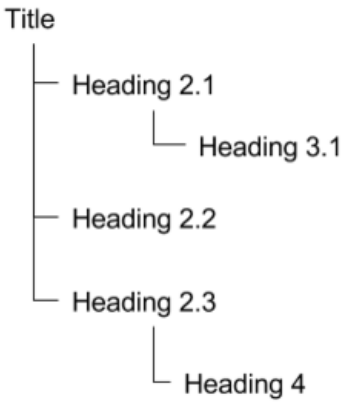

Figure 4.12: Graph of how headings should be connected according to level.

Text, formatted text, ISBN, and next paragraph
Because text, formatted text, ISBN numbers, and next paragraph, have many similarities, in how they were visited, they are all described here. Text and ISBN number should be handled in the same way, as they would be added to a paragraph directly. Formatted text is a bit different as it should be added with a modifier, that describe how the TTS should pronounce the text. Lastly, the next paragraph rule, which should be used to switch the paragraph, that is added text to, if for example the code from line 1 to line 10 in Listing 4.1, was parsed, it should construct a tree like the one in Figure 4.13.

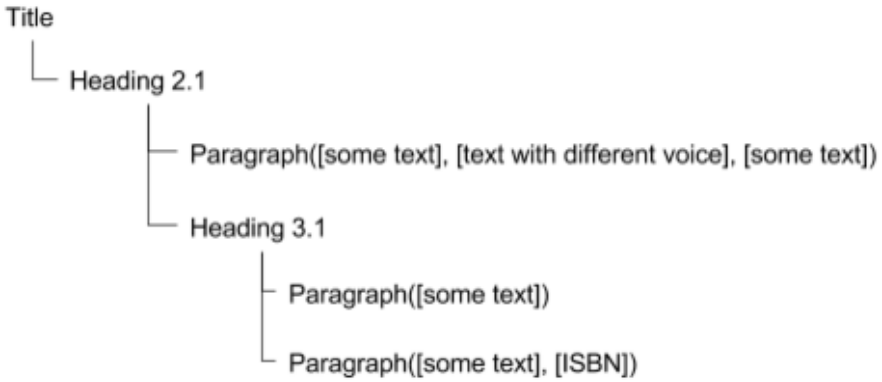

Figure 4.13: Graph of how text are related to the headings.

Tables
There are no data structure for making tables. To simulate tables, sections were used, where the title of the section was the table heading or default “Table”, if no table heading was found, and a paragraph for each row containing the text for all the cells in that row. Taking line 12-20 on Listing 4.1 as an example of an input, the structure would look like in Figure 4.14.

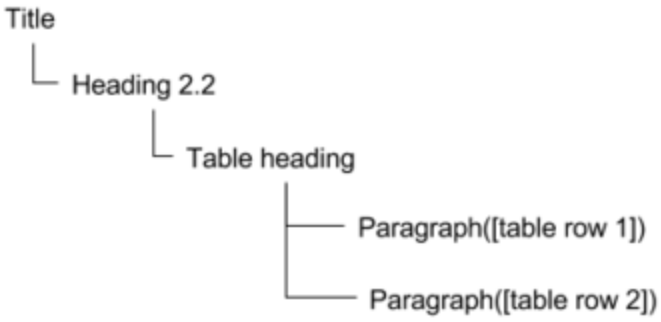

Figure 4.14: Graph of how the data structure looks for a table.

Lists
Just as tables did not have a dedicated data structure, neither did lists. To have some way to be able to jump between the different elements in the list, each element would be added as its own paragraph. Parsing line 22-26 on Listing 4.1 should produce the structure in Figure 4.15.

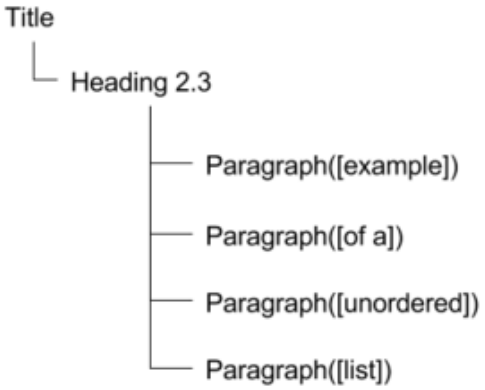

Figure 4.15: Graph of how the data structure looks for lists.

Links
The last type of code is the links. These consists of four different types, internal and external links, urls, and redirects. All links should be added as a paragraph, but which parts of internal and external links that should be inserted, should vary depending on whether they have a label or not. If they do not have a label, the link itself should be added to the paragraph, and if there is a label, the label should be added. The urls should just be added to the paragraph, while the redirects should add the "#REDIRECT" and what the internal link after it would have added. This would for line 28-33 in Listing 4.1 give the structure shown in Figure 4.16.

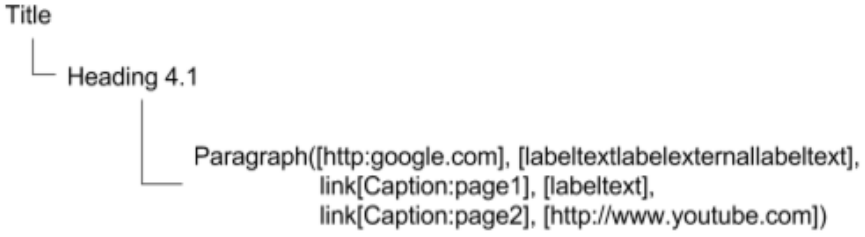

Figure 4.16: Graph of the data structure for links.

4.4.9 Builder: Controller

This wrapper object encapsulates the objects used to construct readable data, along with some information about the contained data. It is used internally by the parser to transform and store the text from the wiki markup in the format specified by the parser. Once the parser has completed a paragraph from the wiki markup, it will extract the data from the builder to the paragraph.

4.5 Server

In Section 3.3, it was decided to use Solr as the server option. Solr used a Wikipedia dump, which would then be used to create an indexed version of the dump.

4.5.1 Solr: Controller

To be able to use Solr, an understanding of what data, should be retrieved, indexed, and become searchable, would be required.

Wikipedia dump

Solr was used to index a Wikipedia dump from 1st September 2015. This dump contained a large number of articles. Each article was enclosed inside a page field. This field was the baseline for indexing the dump, because the individual articles were the goal. Inside the dump, there existed a number of fields. These fields are shown in Listing 4.2.

```
1  <page>
2      <title>Example Article</title>
3      <ns>0</ns>
4      <id>1</id>
5      <revision>
6          <id>1337</id>
7          <parentid>123456</parentid>
8          <timestamp>2015-08-26T14:00:54Z</timestamp>
9          <constributor>
10             <username>Constantine</username>
11             <id>987</id>
12         </constributor>
13         <minor />
14         <comment>Add better examples of an example article.</comment>
15         <model>wikitext</model>
16         <format>text/x-wiki</format>
17         <text>This is an example of a wikipedia article. This shows the ↙
               ↳ different fields.</text>
18         <sha1>4ro7vvppa5kmm0o1egfjztzcwd0vabw</sha1>
19     </revision>
20 </page>
```

Listing 4.2: Example article

The parser in Section 4.4.8, required some wiki markup code to be able to parse each article. This was extracted from the text field, then saved and used for easy retrieval as part of the search results.

Solr

To get Solr to work with the Wikipedia dump, the solrconfig.xml, schema.xml, and data-config.xml files had to be configured to do the desired operations and work with the fields, of the Wikipedia dump. The main goal was to be able to search on the title and text for each article, and retrieve the wiki markup code for the articles.

5 Implementation

This chapter will cover the implementation of VC Wiki Reader. We detail how parts of the system function and describe some of the issues we faced during the implementation.

5.1 Representing an Article

5.1.1 Paragraph

The Paragraph class, as mentioned, represents any textual segments in an article. However it is also the only class that contains textual information, therefore this class contains all the relevant data for the TTS reader, this would mainly be an SAPI PromptBuilder object. However to provide additional metadata about the segment, such as internal links to other articles contained within the segment. The prompt is created in the parser though a Builder object which encapsulates the PromptBuilder providing a more convenient interface for our purposes, as well as storing the metadata before the prompt and metadata is transferred into the paragraph object.

5.1.2 Section

The Section class was implemented to represent an article as a hierarchical tree of paragraphs and nested sections. Each section contained a list of IReadable objects, an empty interface both the Paragraph and Section classes inherit from. This was to allow a more dynamic way to order paragraphs and sections within a containing section. Once the tree was constructed by the wiki markup parser, the tree was flattened into an array of paragraphs and an array of bookmarks. A Bookmark was a simple key-value pair of section names and an offset into the paragraph list where the section begins. The tree itself was flattened from the root down through a depth first approach. To allow for the potentially random order of paragraphs and nested sections, we utilized the runtime type information to know which type of derived object, we were dealing with in any given case. If it is a paragraph, simply append it to the list, else add a bookmark with the name of the section and the index as the current size of the paragraph array, then recursively construct the list for the nested section, see Listing 5.1. We get a decent random access time of $\mathcal{O}(1)$ for a paragraph by using this approach, and an $\mathcal{O}(n)$ for random access at the section granularity. It also gives an $\mathcal{O}(1)$ for getting the next, prior, or current paragraph or section, which was the access pattern, we assumed to be the most common. This is an improvement over the worst case running time for the tree. In addition to constructing these lists, it provided functionalities to alter article playback through skipping or repeating paragraphs or sections, or reading relevant metadata, such as relevant links or the table of contents. Lastly, we also provided a function to increment the internal index into the paragraph, indicating which paragraph would be read next.

```
1  public void Construct()
2  {
3      bookmarks.Add(new Bookmark() { Name = Name, Index = index });
4      Construct(ref paragraphs, ref bookmarks);
5      bookmarks.Add(new Bookmark() { Name = string.Empty, Index = ↙
           ↳ paragraphs.Count });
6  }
7
8  private void Construct(ref List<Paragraph> p, ref List<Bookmark> b)
9  {
```

```
10      foreach (var r in readables)
11      {
12          if (r is Section)
13          {
14              var s = (Section)r;
15
16              b.Add(new Bookmark() { Index = p.Count, Name = s.Name });
17              s.Construct(ref p, ref b);
18          }
19          else
20              p.Add(r as Paragraph);
21      }
22  }
```

Listing 5.1: Paragraph construction

5.2 Client Controllers

5.2.1 The Article Handler

The ArticleHandler was the controller for the root section of a parsed Wikipedia article. The purpose of this controller was to provide the speech prompts from the section to the TTS and receive notifications from the TTS, when the prompt had been read. Upon a completed read, the sections paragraph index should be incremented, and if the prompt was completed or canceled, the controller would issue a call to update the state of the section, see Listing 5.2. This was because only some prompts should be updated in the internal state of the section, i.e. when a user wants to have the table of contents read, the paragraph index should remain unaltered, in contrast to when the user wishes to continue reading from the same point.

```
1  public void OnReadRequestCompleted(object sender, ↙
       ↳ ReadRequestCompletedEventArgs args)
2  {
3      if(args.Increment && !args.Cancelled)
4          Article.IncrementIndex();
5  }
```

Listing 5.2: Conditional state updating

5.2.2 The Page Handler

The PageHandler was, despite its humble size, one of the most central pieces in the application. It sets up the connection to Solr and searches are performed though this controller, see Listing 5.27 for the Solr setup and example querying. To setup Solr we needed a clientside class, SolrEntry, for the representation of the data stored on the server, see Listing 5.3 for the general structure of such a class. If the search returned any results, the wiki markup code was extracted and passed to the parser, which transformed the code into a flattened section object the handler, then stores until it was requested by a user, at which point it would provide the article to the ArticleHandler. As some searches may provide multiple and large articles, we added very basic parallelization to parse the articles, see Listing 5.4, this was done as it was expected that these scenarios would greatly reduce response time of the application.

```
1  class SolrEntry
2  {
3      [SolrUniqueKey("id")] public string id { get; set; }
4
5      [SolrField("field")] public T field { get; set; }
6  }
```

Listing 5.3: Sample client representation of a Solr entry

```
1 Parallel.ForEach(results, (r) =>
2 {
3     // code ommitted
4 });
```

Listing 5.4: Parallelized parsing

5.2.3 The Command Handler

The CommandHandler was a simple controller that simply relayed voice commands to any listeners interested. As all commands recognized by the STT would be simple strings, we converted them to a corresponding enumeration value within the controller and depending on the type of command, issue commands to the respective listeners, i.e. the page handler would not be interested in the playback commands, the article handler would be interested in, and to avoid cluttering the project with numerous event classes, we chose to take this approach instead.

5.3 Debug UI

In relation to the command handler, we decided to implement a debugging UI, as the STT proved to be rather finicky to get to respond. To speed up testing and development we cemented the idea of having a debug UI to emulate voice controls and get at least some information about the articles available to the end user. The emulation part of the UI generally takes the form of **private void** cmd_click(**object** ⬋ ↳ s, EventArgs args){ target.OnCmd(parameters); } this allows for quick testing and faster prototyping of new controls. It also provides a more traditional search box, where the user could type search terms and a drop down menu with all the results returned by the page handler. Any article from this list could be loaded into the article handler and played back using the rest of the emulated voice commands. The debug UI can be seen on Figure 5.1.

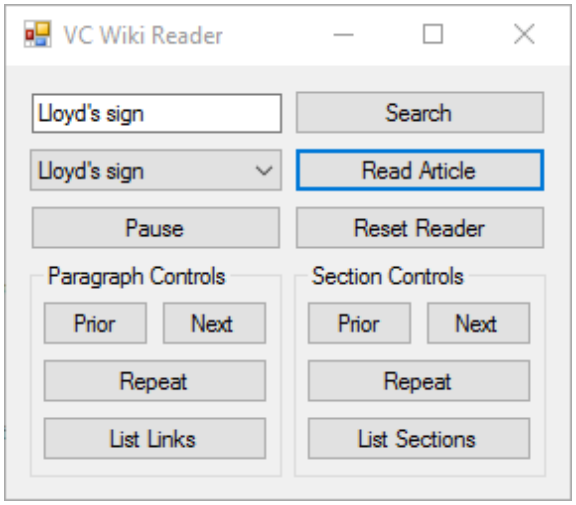

Figure 5.1: The Debug UI

5.4 Events

The design we used between the controllers and views was chosen to be event based to keep them decoupled. Because of this, in addition to some default events used by the technologies we applied, we also defined three additional events, the CommandRecognized, ReadRequested, and ReadRequestCompleted. The first being a simple event that is raised by the STT and subscribed to by the command handler. It contained the command enumeration and an optional search term for the search functionality provided by the page handler. The second containing a list of prompts for the TTS to read, a flag to indicate if the prompts should modify the internal state of the article upon completion, and a flag to indicate if the prompt(s) should be read immediately or queued for reading after the already issued reads. The last event was raised

by the TTS to the article handler to inform it about potential state changes to be made within the article. This event contained two flags, one to indicate if the state should be updated and one to indicate if the prompt was completed or canceled, as the TTS raises the SpeechCompleted event even for canceled events.

5.5 Speech Recognition

The SAPI poses some limitations, as it only works on Windows Vista and later, and only if the system language supports speech recognition. The recommended system language is English, as the voice commands are in English. The speech recognition has not been tested on languages other than English.

To improve the accuracy of the STT, we kept the number of words in the STT grammar small. While this approach ensures a high consistency, when the commands are used, it also allows the STT to recognize anything that was not meant to be a command and attempt to match it to a command anyway. For instance "I am learning about Newton" could be interpreted as "search for Isaac Newton", due to the word Newton appearing in both. This could be partially solved by adding checks on how accurately the STT has recognized a certain phrase. In Microsoft's Speech Recognition this is available through "Confidence", which is a measure for how certain the STT is that it recognized a certain phrase. From our limited experience with the STT, this "Confidence" is largely inaccurate, which is why we have not implemented this. Another option was to use a dictation grammar, but this makes the STT highly inaccurate. It is possible to increase the accuracy but as we show in Section 6.2 this does not work well.

5.6 Wiki Markup Parser

The implementation of the parser consisted of four parts: Adding ANTLR to the project, implementing the grammar, making a visitor for the AST, and how to call the different parts of the system to parse wiki markup code.

5.6.1 Adding ANTLR to a Project

Before ANTLR could be used in the project, it had to be added to the project through a package manager. The version that we installed was Antlr4 version 4.3.0, as this was the most recent stable version. With this done, the project was ready to use ANTLR, and all that had to be done, was to make a grammar that ANTLR could use.

5.6.2 Implementation of the Grammar

For the first part in implementing the grammar, a file with some special properties had to be added to the project. After that the grammar could be used in the project.

Preparing the File for Use

For the first step in making a grammar for the parser, a file was created with the ".g4" extension and UTF-8 encoding, as the file was going to contain special characters. The build options had to be set to "Antlr4", Custom tool as "MSBuild:Compile", and the namespace had to be changed to what we wanted it to be. In the system, that was selected to be "WikiParser.Grammar".

For the second part the project had to be unloaded. After that the project had to be edited, by right clicking on it and selecting "edit ProjectName.csproj". In the window that opens, the code for the file was found, which can be seen in Listing 5.5. This code was changed to the code that can be seen in Listing 5.6. These changed made it so that ANTLR would, in addition to creating a lexer and a parser, also create a base listener and a base visitor.

```
1 <Antlr4 Include="Grammar\WikiMarkup.g4">
2     <Generator>MSBuild:Compile</Generator>
3     <CustomToolNamespace>WikiParser.Grammar</CustomToolNamespace>
4 </Antlr4>
```

Listing 5.5: Code before change

```
1 <Antlr4 Include="Grammar\WikiMarkup.g4">
2     <Generator>MSBuild:Compile</Generator>
3     <CustomToolNamespace>WikiParser.Grammar</CustomToolNamespace>
4     <Listener>True</Listener>
5     <Visitor>True</Visitor>
6 </Antlr4>
```

Listing 5.6: Code after change

After we had done this, we reloaded the project, and the project was then ready to use the grammar file.

Writing the Grammar

The grammar that we wrote was build based on the EBNF grammar in Appendix A.1, where some changes were made to make it easier to access the data in the AST that gets generated by the parser.

Syntax for a Grammar in ANTLR

The grammar for ANTLR had to be written in a different way compared to how a grammar is written in EBNF, but for the most part this difference was minor, see Table 5.1.

Description	EBNF notation	ANTLR notation
Definition	=	:
Terminal string	"...."	'.....'
Concatenation	Term, Term	Term Term
Grouping	(.......)	(.......)
Or	|	|
0 or more times	{.....}	*
1 or more times	{.....}	+
0 or 1 time	[......]	?
termination	;	;

Table 5.1: Notation differences in EBNF and ANTLR.

In addition to having some changes in how notation were written, the grammar also had to be split into lexer rules and parser rules. This was done in the grammar by having the name of the lexer rules written in full caps, and using all lowercase letters for the parser rules. The lexer rules are used by ANTLR to generate the lexer, and the parser rules were used to generate the parser. The entirety of grammar used in the project can be found in Appendix A.2.

Lexer Rules

In the grammar, the lexer rules are a step below the parser rules. Because of that the lexer rules could not contain any parser elements. The lexer rules in the grammar were implemented as two different types of rules, single character rules and character collection rules. The first type was used to specify for example digits, letters, or special characters, while the second type was used to specify, what input the lexer was allowed to combine into one token. The second type was also for the most part, constructed from other lexer rules.

An example of the single character rules can be seen on Listing 5.7, that shows how we implemented Digits, ASCII letters, and Unicode characters, in the grammar. The "[0-9]" on line 1, indicates that a digit is a number between 0 and 9. On line 2 for ASCII the rule states that an ASCII letter is the letters 'a' to 'z' in both upper and lower case. For unicode it was specified that all unicode characters, except the ones used in other rules, are accepted, this includes Digits and ASCII. The reason for this, was that the lexer at times would find it difficult to give a token type for an input, if the token types overlapped, and because the escape characters, were not supposed to be identified as a unicode character token type.

```
1 DIGIT          : [0-9] ;
2 ASCII          : [a-zA-Z];
3 UNICODECHAR    : [\u0000-\u001f] [\u0080-\ufffe];
```

Listing 5.7: Lexer rules for Digits, ASCII letters, and Unicode characters.

An example of a lexer rule that uses other lexer rules can be found in Listing 5.8. This lexer rule was used to identify ISBN numbers. For the first part, the rule had the text "ISBN", and after that it has one digit followed by a number of digits and "-". Lastly the ISBN number could have an upper or lower case "x", or nothing.

```
1 ISBN           : 'ISBN ' (DIGIT) (DIGIT | '-')*
2                    ( 'X' | 'x' | SPACE | LINEBREAK);
```

Listing 5.8: Lexer rule for ISBN numbers.

Parser Rules

The parser rules worked very much like the lexer rules, but as the parser rules are above the lexer rules in the grammar, the parser rules could have lexer elements in them. The parser rules were also very important for the visitor, as it was the parser rules that defined which parts of the transformed input code could be accessed when visiting a node in the AST. Because of this, we had to make some changes in how we wrote the grammar for the tablebody in the grammar in relation to the EBNF version. To describe this more clearly, the code for the table body can be seen in Listing 5.9 and Listing 5.10. As it can be seen, the ANTLR grammar has three rules, while EBNF only has one.

```
1 tablebody      : tablebodycell+ tablebody2+;
2 tablebody2     : (tablerow tablebodycell+);
3 tablebodycell  : (tableheadercell | tablecell);
```

Listing 5.9: Table body in ANTLR.

For the first part, we changed "{ table header cell — table cell }" into "tablebodycell+". This was done because this makes it possible to access the cells in the right order, while if two possible types had been used, it would only have been possible to visit all of one type and then all of the other type, mixing up the order they should have been visited. This problem does no longer exist, when the rule in line 3 of the ANTLR grammar is visited, because at this point there is only one element. The second part on line 1 of the ANTLR grammar, the same apply for "tablebody2", where it in the EBNF was "{ table row, (table header cell — table cell) }".

```
1 table body     = { table header cell | table cell },
2                  { table row, ( table header cell | table cell ) };
```

Listing 5.10: Table body in EBNF.

Another part of the grammar that had to be changed from the original EBNF, was the lists. In the EBNF grammar, the rules for the list was all left-recursive. This is not something ANTLR supports, and because of that it had to be changed. The change in the grammar, was mostly from which direction the list was build. In the EBNF version, the lists were build from the last to the first element, while the changes in the ANTLR grammar, made it so that it started with the first element in the list. The grammar for part of the lists can be seen on Listing 5.11 and Listing 5.12.

```
1 list                      : unorderedlist | orderedlist | difinitionlist;
2 unorderedlist             : '*' rank text+ continueunorderedlist* LINEBREAK;
3 continueunorderedlist     : LINEBREAK '*' rank text+;
```

Listing 5.11: Part of grammar for lists in ANTLR.

As can be seen in the the EBNF version of the grammar on line 1, the list first visits the element “continue unordered list”, instead of visiting “unordered list”. This was flipped in the grammar for ANTLR. In addition to that, a lexer rule called “rank” was also made for the special characters “:”, “*”, and “#”. This was to be able to identify how deep in the list the element was, as the number of special characters in “rank” would be that depth. The same changes as shown for unordered list, were made for ordered lists and defines.

```
1 list                    = continue unordered list | continue ordered list
2                         | continue definition list;
3 unordered list          = "*", text;
4 continue unordered list = (unordered list | continue unordered list
5                           |":"|"*"|"#"),
6                            linebreak, unordered list;
```

Listing 5.12: Part of grammar for lists in EBNF.

5.6.3 Implementation of the Visitor

After the grammar had been implemented, the project was build. This was done because the ANTLR package had to build a lexer, parser, and base visitor for the system. This base visitor was then used for the implementation of the visitor, that constructed an article from an AST.

This was done by making a class that inherit the properties of the base visitor constructed by ANTLR. An example of this can be seen on Listing 5.13. on the first line, the class is named WikiMarkupVisitor, and is set to inherit from Grammar.WikiMarkupBaseVisitor<**string**>, where the base visitor is generalized as a string type. On line 3 is an example of how to make a function in the visitor. The name "VisitArticle" specifies that this is a visitor function for the article parser rule in the grammar. The "Article" part of the name can be changed to any other name of a parser rule, where the first letter is in uppercase. Next on the line the input of the function is specified, which in this case is of the type "Grammar.WikiMarkupParser.ArticleContext". Just like in the name, the "Article" part of the type name, can be changed to the name of a parser rule with an uppercase starting letter. It should be noted that the name of the function and the name of the type should use the same parser rule name.

```
1 class WikiMarkupVisitor : Grammar.WikiMarkupBaseVisitor<string>
2 {
3     public override string ↙
          ↳ VisitArticle(Grammar.WikiMarkupParser.ArticleContext context)
4     {.....}
5 .....
6 }
```

Listing 5.13: Example of how to make a visitor class, from the base visitor.

The task of the visitor was to create a section object with the content of the article in it. For this five internal data structures and variables where needed. These five data structures can be seen in Listing 5.14.

```
1 private Stack<Section> Headerlevel = new Stack<Section>();
2 private Section TableSection = new Section();
3 public Section Article { get; set; }
4
5 Builder TableHeading = new Builder();
6 Builder CurrentParagraph = new Builder();
```

Listing 5.14: Code for instantiating internal variables.

The first variable on line 1, was "Headerlevel" which was a stack used to keep track of the headings, and the text belonging to these headings. The stack contained a section object and an integer that indicates the depth for a heading in the stack. The next "TableSection" on line 2 was a dedicated section for generating a table. The third was an "Article" with was used for storing the final data when the visitor had visited the whole tree. "TableHeading" on line 5, was also used for tables, But only to store the title of the table. Lastly on line 6 is the "CurrentParagraph" which was used to store the paragraph that was getting build at the time.

To show how some of these object were used in the visitor some examples are described below. The examples include Headings, Text, and Nextparagraph.

Headings

The heading was the main building stone, as headings determined, when a section should be added to the main article or another section. Like stated in Section 4.4.8, the headings had to keep their hierarchy in the returned article. For this the stack "Headerlevel" was used. On Listing 5.15 is an example for how the function for visiting heading4 was implemented in the code.

```
1  public override string ↙
       ↳ VisitHeading4(WikiParser.Grammar.WikiMarkupParser.Heading6Context ↙
       ↳ context)
2  {
3        VisitHeadingBody(4, context.headertext().GetText());
4        return context.headertext().GetText() + "\r\n";
5  }
```

Listing 5.15: Code for the visitor to heading 4.

As seen on line 3, the heading calls a function called "VisitHeadingBody" with the parameters "4" and "context.headertext().GetText()". The first parameter is the depth of the heading visited, in this case 4, and the other is the text that is in the currently visited heading. The function called can be seen on Listing 5.16. The reason this function was made, was because all six versions of heading made the same operation with only minor differences.

```
1  private void VisitHeadingBody(int depth, string caption)
2  {
3        CurrentParagraph.EndSentence().EndParagraph();
4        Headerlevel.Peek().Add(CurrentParagraph);
5
6        ColapseStackToLevel(depth);
7        AddToStack(caption);
8
9        CurrentParagraph = new Builder();
10       CurrentParagraph.BeginParagraph(VoiceStyles.Normal).BeginSentence();
11 }
```

Listing 5.16: Code for the VisitHeadingBody function.

The "VisitHeadingBody" function starts by ending the "CurrenParagraph" on line 3, and on line 4, the paragraph is added to the heading on the top of the "Headerlevel" stack. Right after, on line 6, the function "ColapseStackLevel" is called with the depth of the current heading. This function checks if the top element of the "Headerlevel" stack is on a higher or the same level, as the current heading, and if it is, the top element is popped from the stack and added to the new top element of the stack. This process is repeated, till the top element on the stack is at a lower level than the current heading. After that the "AddToStack" function is called on line 7, where the current heading is added to the top of the "Headerlevel" stack. Lastly on line 9 and 10, the current paragraph is reset, and set to have the voice option as "normal".

Text

The visitor for the text, was the visitor that was expected to be used most often, as wiki articles for the most part contains plain text. The visitor was implemented with the code in Listing 5.17. The visitor for text did not make many operations. The only thing the visitor did, was to add the text in the token to the "CurrentParagraph" object, which is shown on line 3. The implementation for the different types of formatted text, is almost the same as this one with the exception, that they for the "CurrentParagraph.Add()" have an additional parameter, that changes the indicated voice for the text.

```
1 public override string ↙
      ↳ VisitTEXT(WikiParser.Grammar.WikiMarkupParser.TEXTContext context)
2 {
3       CurrentParagraph.Add(context.text().GetText());
4       return context.text().GetText();
5 }
```

Listing 5.17: Code for the Text visitor.

Nextparagraph

As a last example of a visitor function we have the Nextparagraph function, that is shown on Listing 5.18. In this visitor, the function "EndAndRenewParagraph" was called on line 3. This function first ended the "CurrentParagraph", then added it to the heading on the top of the "Headerlevel" stack, and lastly it reset the "CurrenctParagraph" object so that it was empty.

```
1 public override string ↙
      ↳ VisitNextparagraph(WikiParser.Grammar.WikiMarkupParser.NextparagraphContext ↙
      ↳ context)
2 {
3       EndAndRenewParagraph();
4       return context.GetText();
5 }
```

Listing 5.18: Code for the Nextparagraph visitor.

The rest of the visitor functions in the visitor works with the same objects and functions, thus explaining them would be repetitive.

5.6.4 Combining the Components

As we had constructed all the different part of the parser, the last step was to connect them. This whole procedure was constructed with 12 lines of code, which can be seen on Listing 5.19. The first step on line 1 was to create an inputstream for the lexer with the string of wiki markup code. Next on line 2, the lexer is instantiated with the inputstream as input. On line 3 this lexer is used to create a tokenstream, which then is used on line 4 to instantiate the parser. The part of the AST to be visited, is the on line 5 assigned to the variable "tree". On line 6 the visitor is instantiated, and on line 7 the title variable for the visitor is set to the title of the article that is getting parsed. This was done because the title of an argument is not a part of the wiki markup code. In the last step on line 8, the AST is visited, and the article is then generated as the tree is visited. As the visitor did not itself return a section object, three lines of code had to be included after this. The first line of code on line 10 gives the article a name, line 11 construct the article so that it can be used by the article handler, and lastly on line 12, the article is pulled out of the visitor.

```
1 var input = new AntlrInputStream(SolrResult.WikiMarkUp);
2 var lexer = new WikiParser.Grammar.WikiMarkupLexer(input);
3 var tokens = new CommonTokenStream(lexer);
4 var parser = new WikiParser.Grammar.WikiMarkupParser(tokens);
5 var tree = parser.article();
6 var visitor = new WikiMarkupVisitor();
```

```
7  visitor.Title = SolrResult.TitleText;
8  visitor.Visit(tree);
9
10 visitor.Article.Name = r.TitleText;
11 visitor.Article.Construct();
12 articles.Add(visitor.Article);
```

Listing 5.19: Code for combining the different parts of the parser

5.7 Solr

To get started with Solr, a few requirements had to be met. The Java Runtime Environment version 1.7 or higher was needed. Also the latest, stable version of Solr was preferable. If it was not specified already, it was recommended to specify Java in the environment variables if on Windows, which we did after encountering a few problems [Apache Lucene, 2015].

5.7.1 Setup

At this stage, all requirements had been fulfilled and the folder could be setup correctly. Solr uses, what is called "cores", that can each be created and used as a search engine. To be able to create a new core, a folder had to be predefined, with designated subfolders and files. In the Solr version used, basic template files were provided. On Figure 5.2 the folders and files can be seen. A few of these files were not mandatory, but for our purpose, they were required.

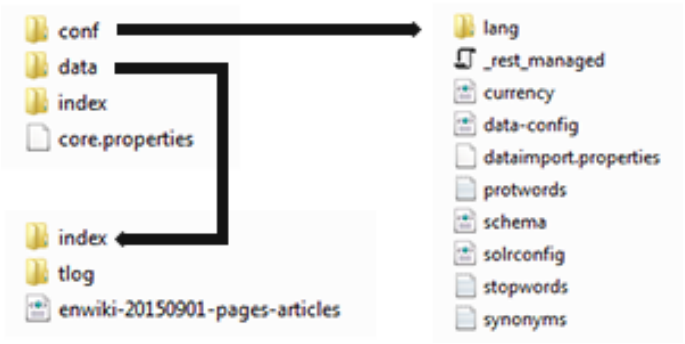

Figure 5.2: The folder structure for the Solr implementation.

solrconfig.xml, schema.xml, and data-config.xml were the three most important files for indexing the Wikipedia dump. Before being able to index, some lines had to be added to solrconfig.xml, schema.xml, and data-config.xml. On Listing 5.20, the lines added in the solrconfig.xml are listed. These lines helped avoid Java heap space errors while indexing, and then there were the DataImportHandler, which was used for the import of the Wikipedia dump.

```
1  <!--These three lines are to help avoiding Java Heap Space Errors-->
2  <ramBufferSizeMB>1024</ramBufferSizeMB>
3  <maxBufferedDocs>1000</maxBufferedDocs>
4  <maxIndexingThreads>8</maxIndexingThreads>
5
6  <!--These lines are for the dataimporthandler-->
7  <requestHandler name="/dataimport" ↙
       ↳ class="org.apache.solr.handler.dataimport.DataImportHandler">
8      <lst name="defaults">
9          <str name="config">data-config.xml</str>
10     </lst>
11 </requestHandler>
```

Listing 5.20: solrconfig.xml

On Listing 5.21, the lines added in schema.xml can be seen. These lines tells Solr which fields to work with, what type the field was, if it should be indexed (Searchable), and if it should store the data.

```
1  <field name="id" type="string" indexed="true" stored="true" ↙
       ↳ required="true" multiValued="false" />
2  <field name="title"      type="string"  indexed="true" stored="false"/>
3  <field name="revision"   type="int"     indexed="true" stored="true"/>
4  <field name="user"       type="string"  indexed="true" stored="true"/>
5  <field name="userId"     type="int"     indexed="true" stored="true"/>
6  <field name="text"       type="text_en" indexed="true" stored="false"/>
7  <field name="timestamp"  type="date"    indexed="true" stored="true"/>
8  <field name="titleText"  type="text_en" indexed="true" stored="true"/>
9  <field name="wikimediaMarkup" type="text_general" indexed="false" ↙
       ↳ stored="true"/>
10
11 <uniqueKey>id</uniqueKey>
12 <copyField source="title" dest="titleText"/>
```

Listing 5.21: schema.xml

For this implementation, it was important to be able to search the title and text. The goal was to get the wiki markup code, and then use it in the parser. The fields specified in the schema.xml, also had to be represented in the data-config file, unless it was like "titleText" and was being copied.

In Listing 5.22, the complete data-config.xml file can be seen. This file was not a default file, and had to be created to be used.

```
1  <dataConfig>
2      <dataSource type="FileDataSource" encoding="UTF-8" />
3      <document>
4          <entity name="page" processor="XPathEntityProcessor" ↙
               ↳ stream="true" forEach="/mediawiki/page/" ↙
               ↳ url="D:\SOLR\server\solr\WikiGiant\data\enwiki-20150901-pages-articles.xml"
               ↳ transformer="RegexTransformer,DateFormatTransformer">
5              <field column="id"          xpath="/mediawiki/page/id" />
6              <field column="title"       xpath="/mediawiki/page/title" />
7              <field column="revision"    xpath="/mediawiki/page/revision/id" />
8              <field column="user"        ↙
                   ↳ xpath="/mediawiki/page/revision/contributor/username" />
9              <field column="userId"      ↙
                   ↳ xpath="/mediawiki/page/revision/contributor/id" />
10             <field column="text"        ↙
                   ↳ xpath="/mediawiki/page/revision/text" />
11             <field column="timestamp"   ↙
                   ↳ xpath="/mediawiki/page/revision/timestamp" ↙
                   ↳ dateTimeFormat="yyyy-MM-dd'T'hh:mm:ss'Z'" />
12             <field column="wikimediaMarkup" ↙
                   ↳ xpath="/mediawiki/page/revision/text"/>
13             <field column="$skipDoc"    regex="^#REDIRECT .*" ↙
                   ↳ replaceWith="true" sourceColName="text"/>
14         </entity>
15     </document>
16 </dataConfig>
```

Listing 5.22: data-config.xml

In the data-config.xml, it was specified how Solr should handle the data. As the Wikipedia dump had what was called "pages", we specified /mediawiki/page/ to get all data within the "page" tags. Inside the page tags, existed a number of other tags. We specified tags such as "id", "title", "text", and "wikimediaMarkup". They got specified by their location. As noted both the "text" and "wikimediaMarkup" tag have the same location [Apache Lucene, 2015].

5.7.2 Indexing

When the indexing had been done, the fields which we required had to be specified. This was done in the schema.xml and the data-config.xml, as seen on Listing 5.21 and Listing 5.22. We encountered a number of issues while indexing. The major issue was the Java heap space issue, where we would run out of memory. Since the way Solr was indexing, it required a decent amount of memory to store pages from the Wikipedia dump. After reading some documentation on the issue, the conclusion was that, it was very difficult to fully avoid this issue and it required fine tuning to use the correct amount of memory. On Listing 5.20, the first three lines were added to help set boundaries while indexing. This was to prevent the Java heap space issue, while also allocating enough memory to the JVM. The time required for indexing would be different depending on how we choose to do the indexing. Our indexing took between three and four hours.

After the indexing, we were only able to search on the titles of the articles. This issue limited the query options. As seen in Listing 5.22, it can be seen that the field column "text" and the field column "wikimediaMarkup" had "/mediawiki/page/revision/text" as their path. These were conflicting with each other, and only the "wikimediaMarkup" was stored, while the "text" was not indexed. This issue was left unresolved, and it affected the quality of the search, while it did not prevent it. A query has to be specified on the title instead [Apache Lucene, 2015].

5.7.3 Querying

In Solr, there is a number of ways to make specific queries. A query is broken up into terms and operators. There are two types of terms: Single terms and phrases. A single term is a single word, like "hello" or "world", whereas a phrase is a group of words, which are surrounded by double quotes, like "Hello world". A number of terms can be combined by using boolean operators to create a more complex query.

To be able to specify a query, it is possible to do queries on specified fields. This can be done by typing in the field name followed by a colon ":" and the term wished to be specified. The default field which was being used to search in, was the text field. For example "title:Donald Duck" is the same as "title:Donald AND text:Duck". It is important to remember double quotation marks to indicate it as a phrase, if Donald Duck was what was wished returned from the query.

It is also possible to do wildcard searches within single terms, but not within phrase queries. To perform a single character wildcard search, the "?" symbol is used. To perform a multiple character wildcard search, the "*" symbol is used. In Listing 5.23, three examples can be seen.

```
1 te?t
2 test*
3 te*t
```

Listing 5.23: Examples of wildcard searches.

Solr also supports fuzzy search, which is based on the Levenshtein Distance, or Edit Distance algorithms. To do a fuzzy search, it is required to use the tilde, "~" symbol at the end of a single word term. In Listing 5.24, two examples can be seen.

```
1 roam~
2 roam~0.8
```

Listing 5.24: Examples of fuzzy searches.

It is also possible to do a proximity search, to determine if for instance two words in a phrase are close enough to each other, by using the tilde, "~" symbol at the end of a phrase, together with a number, it will check for this. In Listing 5.25, an example can be seen.

```
1 "Newton Apple"~10
```

Listing 5.25: Examples of proximity searches.

Solr supports range searches. A range search lets the user perform a search between two terms. In Listing 5.26, two examples can be seen. The first example shows a range search between two dates. Since this search is inclusive, due to the square brackets, it will include the dates in the result. On the second example, it uses curly brackets, which means it will not include the two terms in the result, as it is exclusive.

```
1 timestamp:[20140101 TO 20141231]
2 title:{Albert TO Isaac}
```

Listing 5.26: Example of range searches.

It is possible to boost a term, making that specific term more relevant. By using a caret symbol "$\hat{\{\}}$" with a boost factor (a number) at the end of a term.

There are boolean operators which lets one construct more complex queries. The boolean operators are AND, "+", OR, NOT and "-". They have to be in all caps. These operators, together with the option of grouping and field grouping, can be used to construct complex queries [Apache Lucene, 2015].

As mentioned in Section 5.7.2, we have an unresolved issue with the indexing, and is limited to only querying on the title. This limitation leave us with some options. It is required to explicitly specify the field to be the title, as in "title:" and then the term appended to form the query. The two most relevant options are to use fuzzy and proximity searches to get the most results, to avoid getting zero results.

We used a simple search query on the title, because the parser was not able to parse all articles without errors.

5.7.4 Scalability

Solr is built to be scalable, and used by multiple users. Our implementation of Solr was limited by hardware, specifically it is a single server machine, and not a distributed system.

5.7.5 Fine Tuning

It is always important to preserve resources. It is possible to configure Solr to use a specific amount of memory, but this is not equal to not running out of memory. To limit the resources used, while limiting the Java heap space issues, it requires some testing and adjustments at run time.

We had not made any specific adjustments, although we had made some changes to ensure, as much as possible, to not encounter Java heap space issues. We allocated a large amount of memory and made some limitations in the solrconfig.xml. This was not a resource efficient solution, and could be greatly improved.

5.7.6 Implementation

We used SolrNet to connect to our Solr server. In Listing 5.27, an example implementation of how to connect to the Solr server is done. A QueryOptions argument is not required, but it makes a request more specific and helps avoid current issues with memory. First the initialization should be done, then a service locator had to be defined, with the class Product. The class Product defined which fields should be returned from Solr. When this was defined, it was possible to send a query to the Solr server. The results were saved in a variable named *products*, which then would contain a number of Product objects. This can be seen on Listing 5.27.

```
1  // Initialize the connection.
2  Startup.Init<Product>("http://172.25.23.121:8984/solr/WikiGiant");
3
4  // Define the ServiceLocator.
5  var solr = ServiceLocator.Current.GetInstance<ISolrOperations<Product>>();
6
7  // Create QueryOptions.
8  QueryOptions QO = new QueryOptions();
9
10 // Limit the number of results.
11 QO.Rows = 20;
12
13 // Sending a query with QueryOptions and receive results.
14 var products = solr.Query(new SolrQuery("title:Anar~"), QO);
```

Listing 5.27: Example of a C# implementation of Solr

6 Test

In this section the tests that were made on the system are presented.

6.1 Test of Parser

We made two types of tests for the parser. The first to see, whether the parser could parse Wikipedia articles, and if not, then find out why that was the case. The second test was to see how fast the parser was. For this we just made our own test data, so we knew, how much was parsed.

6.1.1 Test of Parsing Ability

To test whether the parser could parse Wikipedia articles, we decided to perform tests on four different articles, two short and two long articles.

The articles we performed the test on were:

- Westwood Elementary [Wikipedia, 2015k]
- Lloyd's Sign [Wikipedia, 2015e]
- Sivaji (Soundtrack) [Wikipedia, 2015h]
- Feitsui Dam [Wikipedia, 2015a]

For the two short articles the parsing produced the expected results and made a section with the right content, which could be used by the article handler. For the two longer articles, the parser crashed. By inspecting what caused the crash for the two articles, we found that in both cases, the articles contained nested code, for instance having links in lists. This was not taken into consideration, when we made the grammar for the parser. Both the help pages on Wikipedia and the EBNF page for wiki markup on MediaWiki, did not contain any rules for dealing with nested code [Wikipedia, 2015c; **?**]. For a future version, the parser should be improved upon to support nested code.

6.1.2 Test of Parsing Performance

For testing performance when parsing Wikipedia articles of varying lengths, we made some test code, that had all the elements found in the grammar. The length of the code snippet was about two pages long and can be found in Appendix A.3. To simulate longer articles, the code was duplicated. The results of these tests are shown in Table 6.1. As can be seen in the table, the performance when parsing is close to linear between the performance and the length of the article. An estimation of fairness in response time is any duration less than ten seconds, which is exceeded when parsing above 50 pages of wiki markup code. As the average length of an English Wikipedia article is 590 words [Wikipedia, 2015l], which is less than two pages, this is not considered to be a major concern for the time being.

Pages of code	Parsing time (ms)
2	551
4	963
8	1714
16	3317
32	6408
64	13738
128	25456

Table 6.1: Relation between number of pages and parsing time.

6.2 Test of Microsoft Speech Recognition

The commands were tested were tested to ensure they functioned as intended. Additionally, under development, Microsoft Speech Recognition was loosely tested and trained in an attempt to increase its accuracy. After training no improvements in the recognition were noticed, despite Microsoft's claims that the training will improve the recognition [Microsoft, 2015a]. These tests and results were not documented and no statistical analysis was performed. One observation, however, was that the STT consistently recognized the same audio input as the same text, when using a pre-recorded input.

7 Reflection

7.1 Input and Output

While we attempted to help visually impaired people by offering a new system for interaction with the Internet through spoken input and auditory output, the SAPI used for the input and output, was lackluster in its fulfillment of our needs. The TTS performed adequately for a free solution, it was the best of the free TTS options; however, in spite of this the synthesized voice could become an annoyance to the user due to its unnatural pronunciation. The STT was not a good solution to the problem of receiving input, as the recognition engine had a tendency to recognize noise as actual input. This could lead to very confusing use of VC Wiki Reader, as several of the commands alter what is read aloud. We did not test other STT solutions, but a paid solution might have offered functionalities. Additionally the SAPI restricted us to develop for the Windows platform. While we do not know which platforms visually impaired people prefer, this could prove to extend the time spend on potential ports to other platforms.

7.2 Parser

To create our parser we opted to use the ANTLR package. We found it to be a good decision, as it provided us with great help when creating the grammar, as many of the grammar rules could be written in the exact same way, as the EBNF version of the grammar. By using the parser generator, we greatly reduced the development time by allowing us to focus more on other parts of the project. It was, however a little problematic, as we could not find a complete specification of the wiki markup language, which later gave us some problems when parsing articles.

7.3 Solr

We chose to use Solr to index and search for Wikipedia articles. We found it to be a good decision as Solr is a powerful searching platform to use. We experienced a number of issues, which were tedious, although other solutions would possibly have posed as many, if not more, issues.

Solr uses a RESTful method of communication. However, an additional communication layer could have been implemented between Solr and the client, in order to take advantage of other transfer protocols, such as TCP, UDP, or SOAP. The already built-in method in the SolrNet API proved not only to be sufficient, but the most optimal method for VC Wiki Reader.

8 Conclusion

VC Wiki Reader is a rough proof of concept, that it is generally not usable by the target audience, for the reasons explained below.

We were able to make information on Wikipedia partially accessible through the use of sound, by parsing the articles upon request. The system is operable through the use of voice commands, though these commands are finicky, and may not always behave as expected. Currently the parser is not able to parse nested code. While it is able to parse the article, some elements, such as includes, are not handled. In regards to fulfilling the concepts presented in Section 2.1, there are some issues with three of the four points. VC Wiki Reader is not very perceivable, as there is no functionality for receiving help, or to learn the different commands, and if the user wants to know where in the article they are, they would have to make an educated guess based on the different section titles, that can be presented to them, or they would have to restart the entire article.
Operability is fulfilled to some extend. However, the users are not able to follow links found in the article. Likewise, if the STT does not recognize what the user is saying, a blind person would not be able to use VC Wiki Reader. VC Wiki Reader can not be operated through the use of hotkeys.
The robustness of VC Wiki Reader is inadequate. VC Wiki Reader is prone to crashing, from the STT recognizing the wrong sentence, causing the program to attempt to execute code that might not be executable at that point in time.
VC Wiki Reader is however understandable, to the degree at which we have implemented the parser.

We used the SAPI to handle both TTS and STT, but it leaves much to be desired. The TTS, while understandable, sounds synthetic, and the API lacks customization of the voice used and control over the playback options. The STT is usable but it becomes highly inaccurate as the grammar it recognizes grows in size. While SAPI functions well enough for the proof of concept, both its components should be handled by specialized systems if the system should be generally useful.

VC Wiki Reader is fully scalable the way it is currently implemented, and adding additional users would not affect the system notably. The parser, takes up to 10 seconds when articles reach 50 pages of markup code, which might slightly annoy some users, though the average article is only 590 words long, which would take less than a second.

9 Future Works

In this chapter we describe the incomplete functionalities and ideas for additional features, that could be implemented in the system, to improve its overall quality. These features would be implemented if development was to be continued.

The application was developed with future works in mind, leaving the possibility of further development open. The minimum features have been implemented in the system. However, some have not been implemented, due to time constraints, resulting in an incomplete system. With more development time, these features could be refined and the proof of concept could evolve into a fully operational solution.

9.1 Additional Domains

As described in Section 1.2, the domain was limited to Wikipedia. With further development additional domains could be supported. VC Wiki Reader could be expanded to additional domains, by building additional databases — or expanded to a data warehouse — and a parser for each domain. Some of the domains, which could be expanded into, includes general information, health information, and technical documentation.

9.2 Improved Parser

As described in Section 3.4, we chose to not implement certain parts of wiki markup, including but not limited to, variables, math, tables and citations. These structures need to be implemented for the complete system to increase the quality of information given to the user. Some of these parts would however require some larger reworkings of the parser.

9.3 Internal Links

Some Wikipedia articles link to other Wikipedia articles which have not yet been written. These links appear in a red text color, but our application has no way of detecting these. To not confuse the users, these links should not be read aloud. VC Wiki Reader has nothing to gain from showing the users an empty Wikipedia article, as the intended use of the application does not include letting the users write new articles. There are no ways of detecting whether an article exists from the wiki markup code. Instead, the application has to open the links and check the destinations for content. There can be a performance cost associated with this and the value of implementing this feature has to be evaluated on the basis of the time spent checking links, versus the frustration of opening an empty link.

As described in Section 5.1.1, links were implemented as metadata, but we did not include any methods for navigating through these links. There will not be any representation for external links, as these lead to pages outside the domain.

Implementing these methods are important for increasing usability of VC Wiki Reader. Currently, the users have to make a new search, using the link text that was presented to them.

9.4 Database

As described in Section 5.6 and Section 6.1.2, parsing an article is handled on the client side, and can take several seconds depending on the length and complexity of the article. This wait time could be reduced by preprocessing the articles, moving the workload from the client side to server side. To do this, we would set up a database of all articles parsed. Preprocessing all the articles would increase the storage needed and parsing all the articles would take a long time.

9.5 Solr

The issues encountered with Solr, as described in Section 5.7, would have to be addressed. It is important to be able to search through the Wikipedia article's text, and not just be able to search for the titles. This would most likely return a larger amount of results to the user. In addition to being able to search more widely, it would be wise to implement a more advanced indexing and ranking method to make the search results more relevant. This could be accomplished by using vector space ranking, with Solr's tf-idf implementation, and adding cosine similarity. This should help rank the articles by score with respect to the query. A minor issue is that the cosine score computation could be a bottleneck. It is possible to alleviate this computational burden, but this could possibly affect the top results slightly. This is not necessarily a bad thing, as cosine similarity is just an approximation of user happiness.

After the indexing issue has been resolved, it is fruitful to use more sophisticated queries. It is difficult to know exactly what the user wishes to find when searching, and how their query should be outlined. A basic search should look through article titles and text. Multiple options, in the available commands, could make it possible to do a proximity search to determine if two words are within a specified distance of each other. It could also be a range search and receive results in the specified range. It could be relevant to add a boost to a term, for example it could be relevant to boost the title, compared to the text.

If VC Wiki Reader should evolve from a proof of concept, Solr should be configured into a distributed system, to be able to handle the potential workload. This would require fine tuning of the resources spend while sending query requests and making sure the response time is short.

9.6 Speech Recognition

As described in Section 5.5 and Section 6.2, the Microsoft Speech Recognition is not of adequate quality for all purposes. For future works, the speech recognition should be improved. There are better speech recognition software on the market today, however, the ones which are not limited in their usage are expensive solutions. Either additional free and high quality options become available, or VC Wiki Reader has to become a commercial product, to pay for the premium speech recognition software. Another option would be to develop a speech recognition solution specifically for VC Wiki Reader.

9.7 User Interface

To make VC Wiki Reader more user-friendly towards the general public, and not just the visually impaired, a cleaner version of the current GUI could be made. While a graphical interface is of little to no use to the visually impaired, people with normal sight could get an easier level of entry to the system by having the visual cues that they may be used to. Adding shortcuts to such an interface would also allow the visually impaired to use a keyboard to interact with the system rather than being reliant on voice controls.

Adding a "help" command, would give the user assistance in using VC Wiki Reader. Additional commands could include sound controls, and changing the voice and speed for the TTS.

Another idea is to add more audio cues for events for user feedback from the system, so a visually impaired user would be able to more easily be made aware of changes in the system, and for instance, will not be waiting for an article to read the next section, after the article has been read to the end.

Bibliography

Acapela VasS (2015). Acapela VasS TTS API. http://acapela-vaas.com/. [Online; accessed 18-November-2015].

AFB (2015). Refreshable Braille Displays. https://www.afb.org/ProdBrowseCatResults.asp?CatID=43. [Online; accessed 16-December-2015].

Alexa (2015). The top 500 sites on the web. http://www.alexa.com/topsites. [Online; accessed 18-November-2015].

ANTLR (2015). ANTLR. http://www.antlr.org/. [Online; accessed 10-December-2015].

Apache Lucene (2015). Solr. http://lucene.apache.org. [Online; accessed 15-December-2015].

AT&T (2015). Wizzardsoftware. http://www.wizzardsoftware.com/text-to-speech-sdk.php. [Online; accessed 18-November-2015].

Codeproject (2015). WCF differences. http://www.codeproject.com/Articles/139787/What-s-the-Difference-between-WCF-and-Web-Services. [Online; accessed 16-December-2015].

DBpedia (2015). About. http://wiki.dbpedia.org/about. [Online; accessed 20-December-2015].

infosysblogs.com (2015). REST vs SOAP. http://www.infosysblogs.com/microsoft/2009/08/how_i_explained_rest_to_a_soap.html. [Online; accessed 16-December-2015].

ISpecch (2015a). ISpeech IFrame. http://www.ispeech.org/developer/iframe?src=/dotnetsdkdoc/. [Online; accessed 18-November-2015].

ISpecch (2015b). ISpeech #Personal. http://www.ispeech.org/#/personal. [Online; accessed 18-November-2015].

Ivona (2015). Ivona TTS API. http://b2b.support.ivona.com/?l=en_US. [Online; accessed 18-November-2015].

Media Wiki (2015). Markup spec/EBNF. https://www.mediawiki.org/wiki/Markup_spec/EBNF. [Online; accessed 10-December-2015].

Microsoft (2015). Microsoft Speech API. https://msdn.microsoft.com/en-us/library/ee125102%28v=vs.85%29.aspx. [Online; accessed 18-November-2015].

Microsoft (2015a). Set up Speech Recognition. http://windows.microsoft.com/en-us/windows/set-speech-recognition#1TC=windows-7. [Online; accessed 20-December-2015].

Microsoft (2015b). WCF. https://msdn.microsoft.com/en-us/library/ms731074%28v=vs.110%29.aspx. [Online; accessed 16-December-2015].

Microsoft (2015c). Web Service. https://msdn.microsoft.com/en-us/library/ms972326.aspx. [Online; accessed 16-December-2015].

NeoSpeech (2015). NeoSpeech TTS API. http://neospeech.com/pricing. [Online; accessed 18-November-2015].

Retsinformation (2015). Lov om Offentlig Digital Post. https://www.retsinformation.dk/Forms/R0710.aspx?id=142234. [Online; accessed 18-November-2015].

Terence Parr, K. S. F. (2015). LL(*): The Foundation of the ANTLR Parser Generator. http://www.antlr.org/papers/LL-star-PLDI11.pdf. [Online; accessed 15-December-2015].

The University Library (2015). What Frustrates Screen Reader Users on the Web: A Study of 100 Blind Users. http://www.tandfonline.com/doi/full/10.1080/10447310709336964. [Online; accessed 18-November-2015].

WebAIM (2015a). Screen Reader User Survey #6 Results. http://webaim.org/projects/screenreadersurvey6/. [Online; accessed 18-November-2015].

WebAIM (2015b). Visual Disabilities. http://webaim.org/articles/visual/blind. [Online; accessed 10-December-2015].

WHO (2015). Visual impairment and blindness. http://www.who.int/mediacentre/factsheets/fs282/en/. [Online; accessed 18-November-2015].

Wikipedia (2015a). Feitsui Dam. https://en.wikipedia.org/wiki/Feitsui_Dam. [Online; accessed 15-December-2015].

Wikipedia (2015b). Help:Table. https://en.wikipedia.org/wiki/Help:Table. [Online; accessed 10-December-2015].

Wikipedia (2015c). Help:Wiki markup. https://en.wikipedia.org/wiki/Help:Wiki_markup. [Online; accessed 10-December-2015].

Wikipedia (2015d). JAWS (screen reader). https://en.wikipedia.org/wiki/JAWS_(screen_reader). [Online; accessed 18-November-2015].

Wikipedia (2015e). Lloyd's sign. https://en.wikipedia.org/wiki/Lloyd%27s_sign. [Online; accessed 15-December-2015].

Wikipedia (2015f). NonVisual Desktop Access. https://en.wikipedia.org/wiki/NonVisual_Desktop_Access. [Online; accessed 18-November-2015].

Wikipedia (2015g). Screen magnifier. https://en.wikipedia.org/wiki/Screen_magnifier. [Online; accessed 16-December-2015].

Wikipedia (2015h). Sivaji (soundtrack). https://en.wikipedia.org/wiki/Sivaji_(soundtrack). [Online; accessed 15-December-2015].

Wikipedia (2015i). Speech recognition. https://en.wikipedia.org/wiki/Speech_recognition#In-car_systems. [Online; accessed 19-December-2015].

Wikipedia (2015j). Web accessibility. https://en.wikipedia.org/wiki/Web_accessibility. [Online; accessed 16-December-2015].

Wikipedia (2015k). Westwood Elementary. https://en.wikipedia.org/wiki/Westwood_Elementary. [Online; accessed 15-December-2015].

Wikipedia (2015l). Wikipedia:Size comparisons. https://en.wikipedia.org/wiki/Wikipedia:Size_comparisons. [Online; accessed 15-December-2015].

Wikipedia (2015m). ZoomText. https://en.wikipedia.org/wiki/ZoomText. [Online; accessed 18-November-2015].

Window Eyes (2015). GW Micro - Window-Eyes. http://www.gwmicro.com/Window-Eyes/. [Online; accessed 18-November-2015].

ZoomText (2015). ZoomText Website. http://www.zoomtext.com/products/zoomtext-magnifierreader/. [Online; accessed 18-November-2015].

A Appendix

A.1 Wiki Markup EBNF

This section contains the grammar for our grammar for the parser, written in standard EBNF form.

```
1  article          = expr+;
2
3  expr             = comment
4                   | commentary
5                   | html
6                   | heading
7                   | formating
8                   | horizontalrule
9                   | signature
10                  | links
11                  | include
12                  | behaviorswitch
13                  | headerlink
14                  | ISBN
15                  | list
16                  | table
17                  | text
18                  | nextparagraph
19                  | whitespace
20                  | linebreak;
21
22 comment          = "<!--", text, "-->";
23
24 html             = "<", html type, ">", text, "</", html type, ">"
25                  | "<", html type, ">";
26 htmltype         = {text | "\"" | "="};
27
28 heading          = heading6 | heading5 | headin4 | heading3 | heading2 | ↙
       ↳ heading1;
29 heading6         = "=======", text, "=======";
30 heading5         = "======", text, "======";
31 heading4         = "=====", text, "=====";
32 heading3         = "====", text, "====";
33 heading2         = "===", text, "===";
34 heading1         = "=", text, "=";
35
36 formating        = bolditalic | bold | italic;
37 bolditalic       = "'''''", text, "'''''"; // 5 * '
38 bold             = "'''", text, "'''";     // 3 * '
39 italic           = "''", text, "''";       // 2 * '
40
```

```
41 horizontalrule  = "----", ("-")*;
42
43 signature               = usersignature | usersignaturewithdate | currentdate;
44 usersignature           = "~~~";
45 usersignaturewithdate   = "~~~~";
46 currentdate             = "~~~~~";
47
48 link           = internal link | external link | redirect | url;
49 internal link  = "[[", full pagename, ["|", label], "]]", label extension;
50 external link  = "[", url, [whitespace, label], "]", label extension;
51 redirect               = "#REDIRECT", internal link;
52 url            = {ASCII letter}, "://", {URL char};
53 full pagename  = [ namespace, ":" | ":" ] pagename;
54 namespace      = Unicode char, { Unicode char };
55 pagename       = Unicode char, { Unicode char };
56
57 include        = template | tplarg;
58 template       = "{{", title, { "|", part }, "}}";
59 tplarg         = "{{{", title, { "|", part }, "}}}";
60 part           = [ name, "=" ], value ;
61 title          = balanced text ;
62 name           = balanced text ;
63 value          = balanced text ;
64 balanced text  = text without consecutive equal braces,
65                  { include, text without consecutive equal braces };
66
67 behaviorswitch          = place TOC | force TOC | disable TOC | disable section edit
68 place TOC               = {whitespace|linebreak}, "__TOC__", {whitespace|linebreak}
69 force TOC               = {whitespace|linebreak}, "__FORCETOC__", {whitespace|linebreak};
70 disable TOC             = {whitespace|linebreak}, "__NOTOC__", {whitespace|linebreak};
71 disable section edit    = {whitespace|linebreak}, "__NOEDITSECTION__", {whitespace|linebreak};
72
73 list                    = unordered list | ordered list | definition list;
74 unordered list          = "*", text;
75 continue unordered list = (unordered list|continue unordered list|":"|"*"|"#"),
76                           linebreak, unordered list;
77 ordered list            = "#", text;
78 continue ordered list   = (ordered list|continue ordered list|":"|"*"|"#"),
79                           linebreak, ordered list;
80 definition list         = [text], ":", text;
81 continue definition list = (definition list|continue definition list|":"|"*"|"#"),
82                            linebreak, definition list;
83
84 table            = table start, [table header], [table row], table body, table end;
85 table start      = "{|", {style|whitespace}, linebreak;
86 table end        = "|}";
87 table header     = "|+", text, linebreak;
```

```
88  table body         = ( table header cell | table cell ),
89                       { table row, ( table header cell | table cell ) };
90  table header cell  = (linebreak, "!", ({style|whitespace}- "|"), text)
91                     | (table cell, ("!!" | "||"), ({style|whitespace}- ↙
                           ↳ "|"), text);
92  table cell         = (linebreak, "|", ({style|whitespace}- "|"), text)
93                     | (table cell, "||", ({Style|WhiteSpace}- "|"), text);
94  table row          = linebreak, "|-", {"-"}, {style|whitespace}, linebreak;
95
96  text               = {Unicode char}
97  next paragraph     = linebreak linebreak;
98  whitespace         = (" " | "\t");
99  linebreak          = ("\n" | "\r\n");
100
101 digit              = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | ↙
        ↳ "9" ;
102
103 ASCII letter       = ("a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" ↙
        ↳ | "j" | "k" | "l"
104                      | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" ↙
                           ↳ | "v" | "w" | "x"
105                      | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" ↙
                           ↳ | "H" | "I" | "J"
106                      | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" ↙
                           ↳ | "T" | "U" | "V"
107                      | "W" | "X" | "Y" | "Z");
108
109 Unicode char        = (*valid unicode characters*)
110
111 URL char           = (ASCII letter | digit | "-" | "_" | "." | "~" | "!" | ↙
        ↳ "*" | "'" | "("
112                    | ")" | ";" | ":" | "@" | "&" | "=" | "+" | "$" | "," ↙
                           ↳ | "/" | "?" | "%"
113                    | "#" | "[" | "]");
```

A.2 Implementation Grammar

This section contains the grammar that was used with ANTLR to generate a lexer, parser, and base visitor.

```
1  grammar WikiMarkup;
2
3  parser::membersprotected const int EOF = Eof;lexer::members
4  {
5  protected const int EOF = Eof;
6  protected const int HIDDEN = Hidden;
7  }
8
9  /*
10 * Parser Rules
11 */
12
13 article          : expr+;
14
15 expr             : comment              # COMMENT
16                  | html                 # HTML
17                  | heading              # HEADING
18                  | formating            # FORMATING
19                  | horizontalrule       # HORIZAONTALRULE
20                  | signature            # SIGNATURE
21                  | links                # LINKS
22                  | headerlink           # HeaderLink
23                  | include              # INCLUDE
24                  | behaviorswitch       # BEHAVIORSWTCH
25                  | ISBN                 # Isbn
26                  | list                 # LIST
27                  | table                # TABLE
28                  | text                 # TEXT
29                  | nextparagraph        # NEXTPARAGRAPH
30                  | ws                   # WS
31                  ;
32
33 /// Comments
34 comment          : COMMENTSTART commenttext COMMENTEND;
35 commenttext      : (text | LINEBREAK)*;
36
37 html             : '<' (text | '"' | '=' | '/')+ '>' htmltext+ '<'
38                    (text | '"' | '=')+ ('/')? '>' | '<' (text | '"' | ↙
                       ↳ '=')+ '>';
39
40 htmltext         : (text | ws | '|' | '=' | '_' | '<!--' | '-->' | ':'
41                  | QBOLD | QITALIC | '|-' | '[' | ']');
42
43 /// Headings
44 heading          : heading6 | heading5 | heading4
45                  | heading3 | heading2 | heading1;
46 heading6         : H6 headertext H6 space* linebreak;
47 heading5         : H5 headertext H5 space* linebreak;
48 heading4         : H4 headertext H4 space* linebreak;
49 heading3         : H3 headertext H3 space* linebreak;
50 heading2         : H2 headertext H2 space* linebreak;
51 heading1         : H1 headertext H1 space* linebreak;
52 headertext       : (text | LINEBREAK)+;
```

```
53
54  /// Text formatting
55  formating       : bolditalic | bold | italic;
56  bolditalic      : QBOLD QITALIC formatingtext QITALIC QBOLD;
57  bold            : QBOLD formatingtext QBOLD;
58  italic          : QITALIC formatingtext QITALIC;
59  QBOLD           : QUOTE QUOTE QUOTE;
60  QITALIC         : QUOTE QUOTE;
61  formatingtext   : text+;
62  //formatingtext : (WORD | ws | UNICODECHAR | QUOTE)+;
63
64  /// Horizontal line
65  horizontalrule  : '-' '-' '-' '-' ( '-' )* LINEBREAK;
66
67  /// Signatures
68  signature           : usersignature | usersignaturedate | currentdate;
69  usersignature       : '~' '~' '~' ;
70  usersignaturedate   : '~' '~' '~' '~' ;
71  currentdate         : '~' '~' '~' '~' '~' ;
72
73
74  /// Links
75  links           : internallink | externallink | redirect | url;
76  internallink    : STARTLINK STARTLINK fullpagename (VERTICALBAR linklabel)?
77                    ENDLINK ENDLINK (labelextension)?;
78  externallink    : STARTLINK url (externlabel)? ENDLINK (labelextension)?;
79  redirect        : '#REDIRECT ' internallink;
80
81  url             : urlprefix urlseperator link;
82  urlprefix       : WORD;
83
84  fullpagename    : (namespace ':' | ':' )* pagename;
85  namespace       : ((WORD | SPACE)+ | UNICODECHAR | '"' | QUOTE | ','
86                  | '.' | '(' | ')' | '-' | QUOTE | '#')+;
87  pagename        : (text | '#')+;
88
89  externlabel     : SPACE label;
90  linklabel       : text+;
91  labelextension  : WORD;
92  label           : text+;
93
94  /// Header link
95  headerlink      : STARTHEADER headerlinktext ENDHEADER;
96  headerlinktext  : (text | ws )+;
97
98
99  /// Parser rules for Include
100 include         : template | tplarg;
101 template        : ( '{{' (include | includetext+)+ '}}');
102 tplarg          : ( '{{{' (include | includetext+)+ '}}}');
103 includetext     : (text | ws | '|' | '=' | '_' | '<!--' | '-->' | ':'
104                 | '<' | '>' | QBOLD | QITALIC | '|-' | '[' | ']');
105
106
107 /// Parser rules for Table of content
108 behaviorswitch  : placetoc | forcetoc | disabletoc | disableedit;
```

```
109 placetoc          : '__TOC__';
110 forcetoc          : '__FORCETOC__';
111 disabletoc        : '__NOTOC__';
112 disableedit       : '__NOEDITSECTION__';
113
114
115 /// Parser rules for lists
116 list              : unorderedlist | orderedlist | difinitionlist;
117
118 unorderedlist           : '*' rank text+ continueunorderedlist* LINEBREAK;
119 continueunorderedlist   : LINEBREAK '*' rank text+;
120 orderedlist             : '#' rank text+ continueorderedlist* LINEBREAK;
121 continueorderedlist     : LINEBREAK '#' rank text+;
122 difinitionlist          : definetext? ':' text+ continuedifinitionlist*
123                           LINEBREAK;
124 continuedifinitionlist  : LINEBREAK definetext? ':' rank text+;
125 rank                    : (':' | '*' | '#')*;
126 definetext              : text+;
127
128
129 /// Tables
130 table           : tablestart (tableheader)? (tablerow)? tablebody tableend;
131 tablestart      : '{|' (style)? linebreak;
132 tableend        : linebreak? '|}';
133 style           : text+;
134 tablebody       : tablebodycell+ tablebody2+;
135 tablebody2      : (tablerow tablebodycell+);
136 tablebodycell   : (tableheadercell | tablecell);
137 tablecell       : (linebreak)? '|' (style+ '|')? text+
138                 | tablecell '|' '|' (style+ '|')? text+;
139 tableheader     : '|+' text+ linebreak;
140 tableheadercell : linebreak '!' (style+ '|')? text+
141                 | tableheadercell ('!' '!') ( style+ '|')? text+;
142 tablerow        : (linebreak)? '|-' ('-')* (style)? linebreak;
143
144
145 /// Text and whitespace
146 text            : ((WORD | SPACE)+ | UNICODECHAR | '"' | QUOTE
147                 | ',' | '.' | '(' | ')' | '-' | '/' | '-' | ':'
148                 | '?' | '&' | '#' | '$' | ';');
149 nextparagraph   : LINEBREAK LINEBREAK;
150 ws              : (LINEBREAK | SPACE)+;
151 space           : SPACE;
152 linebreak       : LINEBREAK;
153
154
155 /*
156 * Lexer Rules
157 */
158 COMMENTSTART    : '<!--';
159 COMMENTEND      : '-->';
160
161 /// Headers
162 H6              : '======' ;
163 H5              : '=====' ;
164 H4              : '====' ;
```

```
165 H3              : '===' ;
166 H2              : '==' ;
167 H1              : '=' ;
168
169 /// Text formating
170 QUOTE           : '\'';
171
172 /// Handles URLs like "http://google.com"
173 urlseperator    : ':' '/' '/';
174 link            : (urlchar | WORD)+;
175
176 urlchar         : '-' | '_' | '.' | QUOTE
177                 | '~' | '!' | '*' | '('
178                 | ')' | ';' | ':' | '@' | '&'
179                 | '=' | '+' | '$' | ',' | '/'
180                 | '?' | '%' | '#' | '[' | ']' ;
181
182 /// Character types
183 WORD            : (ASCII | DIGIT)+ ;
184 DIGIT           : [0-9] ;
185 ASCII           : [a-zA-Z];
186 UNICODECHAR     : [\u0000-\u001f] [\u0080-\ufffe];
187
188 /// Links
189 STARTLINK       : '[';
190 ENDLINK         : ']';
191 VERTICALBAR     : '|';
192
193 /// Encapsulation of header links
194 STARTHEADER     : '/*';
195 ENDHEADER       : '*/';
196
197 /// ISBN numbers
198 ISBN            : 'ISBN ' (DIGIT) (DIGIT | '-')*
199                   ( 'X' | 'x' | SPACE | LINEBREAK);
200
201 /// Whitespace
202 SPACE           : (' ' | '\t' );
203 LINEBREAK       : ( '\r' | '\n' | '\r\n' );
204
205 WS              :   (SPACE | LINEBREAK);
```

A.3 Parser Performance Test Code

This section contain the code that was used to test the speed of the parser.

```
Test message
====== this is the tittle ======
blabla http://google.com blabla
[[Category:Page|Label]]

[http://youtube.com inerlabel]extent
#REDIRECT [[Categoty:Page]]

ISBN 65-35-87x
/* jhsd sdfkjh s kfsdj sdkjf hsdkfj hsdfk hsf kjsdhf */

{|
|+ The tables caption
|-
| cell code goes here
|-
| next row cell code goes here
| next cell code goes here
|}

---------
{{ Include }}

<nowiki> bla bla bla </nowiki>

* Firt entry'
** Second Entry
*****Third Entry
* Blablabla

# Text
#* Order unorder
** Unorder

<!-- Comment -->
<comment> Comment 2 </comment>

=== Heading 3 ===
Large text Large text Large text Large text, Large text Large text Large ↙
    ↳ text Large text Large text Large text Large text Large text Large ↙
    ↳ text Large text Large text Large text, Large text Large text Large ↙
    ↳ text Large text Large text Large text Large text, Large text Large ↙
    ↳ text Large text Large text Large text Large, text Large text Large ↙
    ↳ text Large text.
Large text Large text Large text Large text, Large text Large text Large ↙
    ↳ text Large text Large text Large text Large text Large text Large ↙
    ↳ text Large text Large text Large text, Large text Large text Large ↙
    ↳ text Large text Large text Large text Large text, Large text Large ↙
    ↳ text Large text Large text Large text Large, text Large text Large ↙
    ↳ text Large text.
Large text Large text Large text Large text, Large text Large text Large ↙
    ↳ text Large text Large text Large text Large text Large text Large ↙
    ↳ text Large text Large text Large text, Large text Large text Large ↙
```

```
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
42  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
43  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
44  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
45  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
46  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
47  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
48  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
49  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text Large ↙
    ↪ text Large text Large text Large text Large, text Large text Large ↙
    ↪ text Large text.
50  Large text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text Large text Large ↙
    ↪ text Large text Large text Large text, Large text Large text Large ↙
    ↪ text Large text Large text Large text Large text, Large text.
```

A.4 CD

Include in physical version of the report, is a CD with the following content:

- Two executable versions of the program, one with speech recognition active, one with speech recognition disabled.
- The source code as a solution.
- A digital version of the report.
- A pdf with the source code.