CHALMERS

Annotating Software Requirement Specifications with Mobile Speech Recognition

Master of Science Thesis in the Master Degree Programme Software Engineering

OLA PETERSSON

VIKTOR MELLGREN

Department of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2013
Master's Thesis 2013:1

Annotating Software Requirement Specifications with Mobile Speech Recognition

OLA PETERSSON,
VIKTOR MELLGREN,

Examiner: SVEN-ARNE ANDREASSON
Supervisor: ROBERT FELDT

Chalmers University of Technology
University of Gothenburg
Department of Software Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Cover: The Android robot speaking to a document

Department of Software Engineering
Gothenburg, Sweden June 2013

Abstract

Requirements are crucial in software engineering and there is ample support for how to elicit and document them; however, relatively little support exist for requirements maintenance. We argue that lightweight methods for annotating requirements are needed and present a system based on speech recognition to enable it. This paper describes the system design and a set of experiments and user tests to validate its use. For more realistic evaluation our system has been adapted to the commercial requirements management tool SystemWeaver. Results show that the accuracy of free text speech input is not high enough to enable free-form addition and edits to requirements. However, requirements identification and annotation is practical by extending the system with string distance calculations to the set of requirements being matched. Lookup times on industrial requirements data was deemed practical in user tests, in particular for annotations on the go and during requirement reviews.

Keywords: requirements engineering, speech recognition, lightweight, annotations, quality assurance, documentation, maintenance, mobile software, Android

Acknowledgements

We would like to thank Systemite AB for their cooperation, help and engagement in this thesis. Moreover, we would like to thank our supervisor Robert Feldt. Without you the thesis would not have been what it is today. We would also like to thank whoever coined the expression ”how much wood could a woodchuck chuck if a woodchuck could chuck wood”. It has been a tremendous help during our many hours of testing and evaluation of the speech recognizer. Finally, we thank our friends and family for their love and support.

- Ola Petersson & Viktor Mellgren, Göteborg 13/05/15

Contents

1

Introduction

You can trace many errors in software products and systems to the management of requirements (Tseng & Jiao 1998, Wiegers 2009). The fact that requirements in many ways is people oriented, not technical oriented, makes room for errors of a more complex nature. These errors might be that requirements changes during the lifespan of a project (Bjarnason et al. 2011, Saiedian & Dale 2000), and are not properly updated. A key challenge in requirements engineering is that customers typically express their needs in natural language, while the requirements engineer is tasked to elicit and translate this informal information into a more formal requirement specification. This makes room for interpretational errors and ambiguities (Al-Rawas & Easterbrook 1996, Ross & Schoman Jr 1977), where information gets lost in translation (Bjarnason et al. 2011). Even though companies and developers acknowledges problems associated to requirement management, many do not put enough effort and time into the task of requirement maintenance (Forward & Lethbridge 2002). In 2012 Wnuk et al. (2012) presented that 84.3% of their recipients considered obsolete requirement specifications to be either serious or somewhat serious, 76.8% of those recipients reported that they did not have any tool, method or process for handling obsolete requirement specifications. With the agile methodologies that have become more popular in the last years, more responsibility is put on the developers (and customers) to be responsive to changes (Highsmith & Cockburn 2001, Ibrahim 2012).

Since its introduction in 1952 (Davis et al. 1952), the development of speech recognition has reached a point where it has become good enough to be used with a satisfying result (Ballinger et al. 2010, 2011, Schalkwyk et al. 2010). We believe that using speech recognition can be one way to lower the time needed and perceived barriers to query and update requirements during software maintenance and during the evolution of requirements understanding in a development project. This could act as a lightweight access mechanism for potentially large requirements databases and allow more active ways of working with requirements. There is a risk that much information is currently lost, and

even forgotten, if it is not updated or recorded immediately. There is also a risk that navigating large requirements databases takes longer time and requires more effort than more lightweight interactions would allow.

Quality assurance of requirements are often talked about in the context of verification and validation (Chemuturi 2010, Denger & Olsson 2005). But there has been relatively little work on the quality of Software Requirement Specification (SRS) and in particular for individual requirements. The focus is still on reviews, audits and walkthroughs and primarily on the SRS as a whole. Verification of requirements is primarily described as actions taken periodically or at special points in time during a project such as gates or partial deliveries. It does not address continuous or ongoing quality assurance of requirements. Denger & Olsson (2005) stresses the importance of starting with QA as early as possible , in the requirement elicitation phase. They also talk about techniques to minimize the chance of introducing defects in requirements documents and the need for tool support and how that could facilitate "other" quality aspects, not currently being addressed.

IEEE Recommended Practice for Software Requirements Specifications (1998) defines quality characteristics and how to create quality SRSs, it does not describe processes on how to work with requirements in a continuous fashion.

This thesis will describe our design and evaluation of an annotation system for requirements databases that is based on speech recognition input from a smartphone. The thesis aims to evaluate if a speech recognizer is a possible tool for working towards large scale SRSs, and how such a tool is perceived by end users. Furthermore, we will evaluate if an annotation system can enable a way of continuously working with quality assurance of a requirement specifications.

In order for our evaluation of the system to be more realistic this research was conducted in co-operation with the Swedish company Systemite AB. Systemite develops a tool that support the management and development of product lines of software systems, called SystemWeaver. An essential part of this tool is its requirements management and requirements engineering support. Through the collaboration with Systemite we could evaluate our annotation system on four industrial requirements databases from a large Swedish company developing software-intensive, embedded systems.

1.1 Purpose

The aim of the thesis is to present a way of incorporating mobile speech recognition (using a smart device) to existing large scale requirement specifications to make it easier for companies to actively work with the maintenance of requirement documentation. With today's technologies in smart devices, speech recognizers have become more accurate and available. We believe that we therefore can enable the people associated to a requirement document to be able to, through mobile smart devices, maintain the documentation anytime, anywhere. Furthermore, the purpose of the product that is developed during this research is to make it easier for developers to work concurrently with the development of software and the maintenance of that software's requirements, hopefully resulting in up

to date requirements and fewer errors associated with the requirement documentation. The first part of the product is the speech recognition annotation application, which annotates requirement documents. The second part of the product is to create reports from the annotations added from the speech recognition application and present the data collected for analysis.

We will also share our experiences with the developed system and its associated technologies as well as giving our thoughts on annotating software artifacts and using speech recognition. In the thesis we will also provide ideas for future work in this area.

We will investigate if this product will make it easier to work continuously with requirements. Furthermore we will also look into which interesting data could be used from the annotated documents.

1.2 Goals

The goal is to develop a mobile smart device application which uses speech-recognition to annotate requirement specifications in a software project. Furthermore the goal is to extract data from the annotations to visualize information about the requirements. The data will be analyzed and we will investigate which valuable aggregations and metrics that could be extracted from the data to allow an owner of the requirements to easily check the status of the specification, thus hopefully be able to recognize were improvements and clarifications needs to be done to higher the quality of the requirement specification.

Finally, we will evaluate if the developed product is feasible within large scale requirement projects.

1.3 Scope and limitations

The scope is to evaluate the concept of annotating requirement documentations with the help of mobile speech recognition. We will not investigate speech recognizers and the speech recognition will be considered a black box, something that can transform spoken words into literal strings. For this research, Android's provided speech recognizer will be used. The reason is that we will not build a speech-to-text engine to fit our purpose, we rather want to explore a state of practice speech recognizer and how it can be integrated into our solution.

To have a realistic evaluation, we will look at real requirements from real software projects. We will however delimit us to software projects within the automotive industry, since it is the integration between our developed product and requirement databases that lies in focus, we consider that requirement databases from one industry will be general enough.

Even though the technical solutions will aim to be general, our product will be integrated into Systemite AB's SystemWeaver (see Section 2.5) and other requirement managing software will not be taken into consideration.

2

Background

The following subsections will address the background of speech recognition technology and requirement management. It will also indulge the reader in the background of Systemite AB, the company where this research and development of the system SpeechWeaver was conducted, as well as their application product line management software, SystemWeaver. Finally, it will give the reader an brief description of the Android Operating System and its history since its release in 2008.

2.1 Speech Recognition

Speech recognition as a technology has been available for a long time and can be traced back to an article by Davis et al. (1952) where they could recognize spoken telephone quality digits at normal speech rates by a single individual, having an accuracy varying between 97 and 99 percent. In the beginning of the history of software speech recognizers the computing power and algorithms were not good enough to give satisfying results. The speech recognizers have developed over the years to become more efficient and accurate. Recently, algorithms have started to run in parallel and sequentially to give better results. Furthermore the corpora have developed over the years to become more and more extensive (Baker et al. 2009).

Since the introduction of smartphones, it is now an easy task to capture the users' speech in the phone and offload the speech recognition to cloud data centers. This makes speech recognition available anywhere the user might be when connected to the Internet. Speech recognition is also available offline with minor accuracy losses in the recognition capabilities (Barra 2012).

October 4, 2011 Apple included the intelligent personal assistant application "Siri" with the release of IOS5 and not long thereafter Google launched their equivalent on Android called "Google Now" (Richmond et al. 2011, Shankland 2012). These tools have

made speech recognition available to the general public.

2.2 Annotations

Annotations are commonly used in software development to provide additional information in the source code and code development artifacts, e.g. constraint annotations (pre-/post conditions) in code or annotations to UML diagrams. However, for other types of documentation, e.g. requirements and specification documents, annotation of individual entries is uncommon. An example of such an annotation, i.e. an informal annotation, could be "the test results need to go here", which could serve as a reminder for future work or other additional information to clarify the entity. However, most annotations consists of meta data that help simplify search and filter functionality, i.e. formal annotations. Formal annotations are primarily intended to be read by machines, e.g. for Semantic-web applications (Uren et al. 2006). An example could be the annotation "Paris", which can be related to the abstract concept "City" or the country "France" given an ontology that helps reduce ambiguity of which "Paris" the annotation refers to. A considerable body of work has been devoted to formal annotation whilst lightweight informal annotations has been left an unexplored area.

2.3 Requirement management

Forward and Lethbridge conducted a survey in 2002 about Software Documentation (Forward & Lethbridge 2002). In the survey, they present that 54% of the 32 participants thought that word processors was a useful documentation technology. In the same year, Juristo et al. (2002) wrote an article that presented that many organizations considered word processors to have an advantage in its simplicity for applications with few requirements. However, for those with more than 1000 requirements (approximately) , the disadvantages, such as scalability and no baseline, outweighed the advantages. Forward and Lethbridge also asked about how often the documentation was updated when changes occurred in the system, and they recipients answered that much of the design documents such as the SRS was rarely updated. The result from the question "How often is documentation updated when changes occur in a software System?" can be seen in Figure 2.1.

Document Type	Mode	% of Mode	In Words
Requirements	2	52 %	Rarely
Specifications	2	46 %	Rarely
Detailed Design	2	42 %	Rarely
Low Level Design	2	50 %	Rarely
Architectural	2	40 %	Rarely
Testing / Quality Documents	5	41 %	Within days

Figure 2.1: Results presented by Forward and Lethbridge about maintenance of documentation. The question asked was "How often is documentation updated when changes occur in a software System?" (Forward & Lethbridge 2002)

From the survey Forward and Lethbridge presents two suggestions:

> "Document content can be relevant even if it is not up to date. (However, keeping it up to date is still a good objective). As such, technologies should strive for easy to use, lightweight and disposable solutions for documentation." (Forward & Lethbridge 2002)

and

> "Documentation is an important tool for communication and should always serve a purpose. Technologies should enable quick and efficient means to communicate ideas as opposed to providing strict validation and verification rules for building facts." (Forward & Lethbridge 2002).

Further support for the need of lightweight requirement practices is given by Zhang et al. (2010) who discusses the importance of lightweight requirement processes in agile development. Kauppinen et al. (2004) claims that organizations can gain requirements engineering (RE) benefits by defining simple RE processes and focusing on a small set of RE practices, and by supporting the systematic usage of these practices.

Annotations are perceived to raise requirement quality and the use of annotations should therefore be integrated into the company's requirements engineering process

2.3.1 Quality of requirements

Guidelines for how to work with SRSs and how to work with software life-cycle processes has been developed by IEEE (*IEEE/EIA Standard Industry Implementation of International Standard ISO/IEC 12207: 1995 (ISO/IEC 12207) Standard for Information Technology Software Life Cycle Processes* n.d., *IEEE Recommended Practice for Software Requirements Specifications* 1998). *IEEE Recommended Practice for Software Requirements Specifications* (1998) describes what characteristics that make a good SRS,

in this way describing how to make high quality SRSs. *IEEE Standard Glossary of Software Engineering Terminology* (1990) defines quality as:

1. *The degree to which a system, component, or process meets specified requirements.*
2. *The degree to which a system, component, or process meets customer or user needs or expectations.*

The first description defines quality of systems to which degree they meet the specified requirements, this means that if the specification is not up to date, and not clear enough, high quality can not be achieved, since it does not longer meet the specified requirements. This is however covered in the second description. If you look at the quality of an SRS, it should conform to the same two descriptions, but the requirements should conform to the software instead, i.e. the quality of an SRS should be to the degree it meets the *actual, intended* functionality of the system as well as being precise and unambiguous. This gives that in order of a project to be of high quality, both these should be met to a high degree.

2.4 Systemite AB

Systemite AB was founded in 2000 based of the recognition of problems with traditional (and often document based) processes in requirement management, configuration management and product data management. The three founders Anderson, Strömberg and Söderberg, all with experience from the automotive industry, developed a platform, *SystemWeaver*, to gather all the design processes within an engineering organization into the same knowledge database.

2.5 SystemWeaver

SystemWeaver is a product line management software application developed by Systemite AB. It is an enterprise-wide data repository that assembles design information from all components of a software system, such as internal structure, interfaces, variants, versions, requirements and component status, in models (*Systemite Products and services* n.d.). The models can be traversed and edited in a graphical interface, SystemWeaver Explorer. An abstract overview of SystemWeaver can be seen in Figure 2.2.

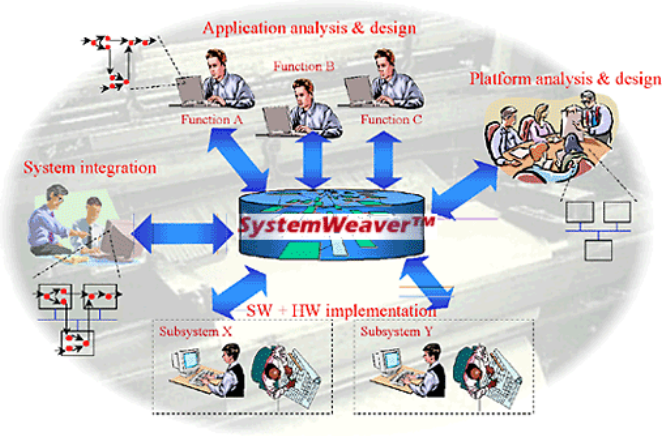

Figure 2.2: An overview of *SystemWeaver*

2.5.1 Parts & Items

The foundations of all of *SystemWeaver's* models are *items* and *parts*. Items are the entities in the system, e.g. components, whilst the parts are the relationships between these entities. An item or a part must be an object within a model, each with an unique identifier and a type identifier. The type identifier is called a SystemWeaver Identification (SID) that specifies the type of a model object. For example, every object that has the SID 'RQQ' is a 'Generic requirement' whilst and object with the SID 'HWC' is a 'Hardware Component'. An example is a query based on 'Generic Requirement' items, with the subitems 'Functional Requirement' and 'Quality Requirement', that returns all of the items. This enables queries on specific levels at the same time as it makes it possible to aggregate data connected to items that are derived from the same type of items.

Parts, in turn, are as mentioned relationship between items. Parts always points in one direction and can define relationships such as 'Item A contains Item B' or 'Item A is derived from Item B'. The fact that the parts only points in one direction means that you would have to create two parts to make a two way relation.The relationships an item can have to another item is within a model is defined by the meta model that was developed by Systemite AB's application engineers for the project.

2.5.2 Meta-Models and Models

Systemite AB are very flexible with their deployment of *SystemWeaver* to their customers. This flexibility is made possible since all correlations and the behavior of items and parts are defined by a structured meta model that supports architectural and solution reuse as well as tailored solutions for specific customers. Therefore, each project carried out in SystemWeaver is still unique but also very adaptable. Furthermore, because of the meta model solution, items and parts can look and behave in several different ways in different models dependent on what constrictions and rules the meta model has

defined.

2.5.3 Issues

SystemWeaver also supports task definition and tracking. Tasks in SystemWeaver are called 'issues', and are, in a simple analogy, similar to notes that can be attached to items. Issues are used for case management of the items that they are attached to and can be set to different states, which are customizable in the meta-model (e.g. started, closed, assigned to). Issues can either point to items or other issues through an issue relation. This relation can only be applied if the origin of the relation is an issue. Furthermore, similar to both items and parts, issues themselves all have SIDs. A graphical representation of how the artifacts in SystemWeaver fits together can be seen in Figure 2.3

SystemWeaver was used as a requirement database and backend in this project to facilitate the annotation scheme of SpeechWeaver.

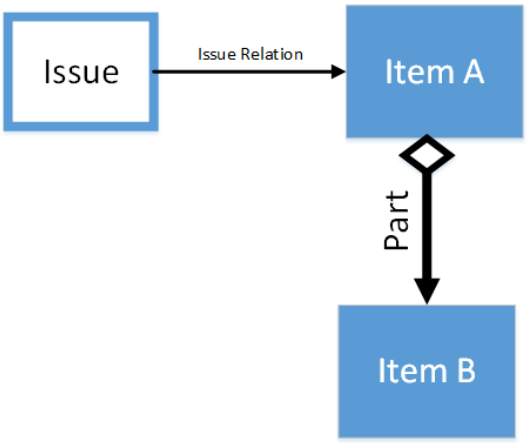

Figure 2.3: A basic graphical representation of the correlations between the artifacts in SystemWeaver

2.5.4 SystemWeaver Report Generation

Built into *SystemWeaver* is an XML-based script language which allows for a user to generate reports based on information that can be gathered from the items and parts. The script language uses the parts to traverse through the item structure, and uses the items to collect information, e.g. present the title-field, creation date and description-field from all items of type "RQQ" (*SystemWeaver help manual* n.d.).

2.6 Android Operating System

The following section will give a background to the Android platform. We will briefly go through Android's history and development since its release give some background on the introduction of speech recognition in the platform.

2.6.1 History

Android is a Linux-based Operating System mainly focused on touch-devices such as tablets and smartphones. Android is Open Source and released under the Apache License 2.0 (*Android Open Source Project* n.d.).

Android was originally a product of Android Inc. In 2005, Google acquired Android Inc. and began to develop the platform (Elgin 2005). Google reached out to several different companies, working with hardware, handset and networks, to form the Open Handset Alliance which was announced in 2007 (*Industry Leaders Announce Open Platform for Mobile Devices* 2007). The alliance works for advancing open standards for smart phones, tablets and other mobile smart devices. In 2008 the first phone running Android 1.0 was released (Helal et al. 2012, Morrill 2008).

Android developed and refined the Operating System and released several updates and versions over the upcoming years. In November 2012, version 4.2 (code name Jelly Bean) was released. By then, Android had become the most popular mobile platform with over 600.000 applications and games available for the platform, and over 400.000.000 activated Android devices (*Discover Android* n.d.).

2.6.2 Speech Recognition in Android

Along with the release of version 2.1 (code name Eclair), Android included a speech-input feature to their keyboard. The featured allowed for users to dictate messages with their speech instead of typing them with their fingers. The distribution of Android devices capable of running speech recognition is about 99.9% as can be seen in Table 2.1 (Gruenstein 2010).

Table 2.1: Android Platform version distribution (Data collected during a 14-day period ending on April 2, 2013)(*Android Developers* 2013)

Version	Codename	API	Distribution
1.6	Donut	4	0.1%
2.1	Eclair	7	1.7%
2.2	Froyo	8	4.0%
2.3 - 2.3.2	Gingerbread	9	0.1%
2.3.3 - 2.3.7	Gingerbread	10	39.7%
3.2	Honeycomb	13	0.2%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	29.3%
4.1.x	Jelly Bean	16	23.0%
4.2.x	Jelly Bean	17	2.0%

3

Related Work

In the following chapter we present previous work done in the area, both in research and in product development.The chapter is divided into the two topics Speech Recognition an Natural Language Processing and Requirement Management.

3.1 Speech Recognition

In recent years, personal assistants using speech recognition have been released, such as Siri (Richmond et al. 2011) and Google Now (Shankland 2012). These applications interprets input from a users speech, and dependent on the interpreted input, performs an action and presents the outcome of the action. These actions could be entering a string in a search engine and present the result, or setting an alarm at a spoken time. A list of available voice actions in Android can be seen in Figure 3.1.

Say	Followed by	Examples
"Open"	App name	"Open Gmail"
"Create a calendar event"	"Event description" & "day/date" & "time"	"Create a calendar event: Dinner in San Francisco, Saturday at 7:00PM"
"Map of"	Address, name, business name, type of business, or other location	"Map of Golden Gate Park, San Francisco."
"Directions to" or "Navigate to"	Address, name, business name, type of business, or other destination	"Directions to 1299 Colusa Avenue, Berkeley, California" or "Navigate to Union Square, San Francisco."
"Post to Google+"	What you want posted to Google+	"Post to Google+ I'm going out of town."
"What's this song?"		When you hear a song, ask "What's this song?"
"Scan a barcode"	Scan a barcode or QR code to learn more about a product.	"Scan a barcode" and hold barcode in front of the device's camera.
"Go to"	Search string or URL	"Go to Google.com"
"Send email"	"To" & contact name, "Subject" & subject text, "Message" & message text (speak punctuation)	"Send email to Hugh Briss, subject, new shoes, message, I can't wait to show you my new shoes, period."
"Note to self"	Message text	"Note to self: remember the milk"
"Set alarm"	"Time" or "for" & time, such as "10:45 a.m." or "20 minutes from now," "Label" & name of alarm	"Set alarm for 7:45 p.m., label, switch the laundry"
"Listen to"	Play music in the Google Play Music app by speaking the name of a song, artist, or album.	"Listen to: Smells Like Teen Spirit"

Figure 3.1: Examples of different voice actions in Android (*Voice action commands* 2013)

Moreover, Nuance has developed a Software called Dragon Naturally Speaking which allows for a user to use their voice as the input channel to a computer. Dragon Naturally Speaking allows for the user to create and edit documents and emails, as well as launching applications or controlling a computer mouse, all through spoken commands (Nuance 2013).

3.2 Natural Language Processing

The following section will present former research and techniques used in Natural Language Processing. The section presents methods for processing natural language text to produce unambiguous data, and discusses string edit distance functions as a method to find similarities or dissimilarities between strings.

3.2.1 Information Extraction and Information Retrieval

In natural language processing, Information Extraction (IE), the process of taking unseen text as input and producing fixed-format, unambiguous data as output, has been researched and implemented in different ways over the years. In 2000, Cunningham et al. (2000) presented five different types of IE:

- **Named Entity recognition (NE):** Which finds entities in texts, such as places, names, organizations etc.
- **Co-reference resolution (CO):** Which identifies the *identity* relationships between the found entities, i.e. if two different words refers to the same person (Ms. Smith and she).
- **Template Element construction (TE):** Who adds descriptive information to the found entities using the the relationships found in CO.
- **Template Relation construction (TR):** Which extracts the relationships between the Template Elements, i.e. the relationship between entities (Ms.Smith studies at Chalmers).
- **Scenario Template production: (ST)** Who take the results found in TE and TR and matches them into event scenarios

Another process in NLP is Information Retrieval (IR) which is the process of finding relevant text and present them to users (typical use case would be an Internet search engine).

3.2.2 String Edit Distance Functions

Speech recognition, as stated, has over the recent years become more efficient and accurate, but there are still limitations. One way to mitigate these limitations, which was deployed in this research, is to use natural language processing and string distance functions, e.g. the Levenshtein or Hamming distance algorithms (Hamming 1950, Levenshtein 1966). These algorithms calculates similarities or dissimilarities between string by measuring how many operations (add, modify, delete) that are needed to transform one string to another.

3.3 Requirement management and NLP

Joshi & Deshpande (2012) showed that natural language can be integrated into the creation of software artifacts. A tool which extracts UML-diagrams from natural language was developed with a precision and recall rate of 85%. The method used was to apply already existing natural language parsers and then apply algorithms to filter and correct the words, remove suffixes and affixes, as well as identifying classes, attributes and relationships.

Gervasi & Nuseibeh (2002) conducted a case study with the aim to validate natural language requirements. The validation was made possible by implementing the process of validation into two steps, *set-up* and a *production phase*.

In the *set-up*, the rules of the domain was defined. Rules of style, structure and language of the requirement document, as well as which properties that were desirable to check, and finally defined models who the selected properties could be checked against. In the second phase, the *production phase*, the document with the natural language was pre-processed and then parsed with regards to the defined properties. This parsing process provided a validation of the syntactic language in the document (Gervasi & Nuseibeh 2002).

4

Methods

The following chapter will present the research questions that were the core of the thesis as well as the methods and experiments used during this research. The chapter will describe how the experiments were performed and their motivation and purpose.

4.1 Design Research

The main objectives of the thesis was to answer the research questions seen below. To be able to answer these, a design research was carried out by the development of SpeechWeaver (see Chapter 5).

RQ1: How do you connect the annotation-data to a requirement?
RQ2: How do you adapt annotations for working with mobile speech recognition?
RQ3: How does the user experience the interaction with mobile speech recognition software in the context of Software Requirement Specifications.
RQ4: How can you aggregate the annotations in a presentable and usable way?
RQ5: Is speech recognition annotations scalable on large Software Requirement Specifications

The design research was preceded by interviews to elicit requirements on SpeechWeaver, as well as gaining knowledge about SystemWeaver, the subjects background and how they worked with requirement management and SystemWeaver. The design research was complemented with four evaluation steps as shown in Figure 4.1. Steps 1 and 2 focused on evaluating to what degree output from speech recognition is useful for identification (step 1) and free-text annotation (step 2) of requirements. Scaling to

realistic requirements database sizes is critical for industrial applicability and since the response time of the system was considered a key characteristic by the company and the identification of requirements dominated the round-trip time, evaluation step 3 measured this time on the server for four different requirement sets. Finally, step 4 evaluated the SpeechWeaver through a user test and a follow-up interview. Below we describe the evaluation steps in detail.

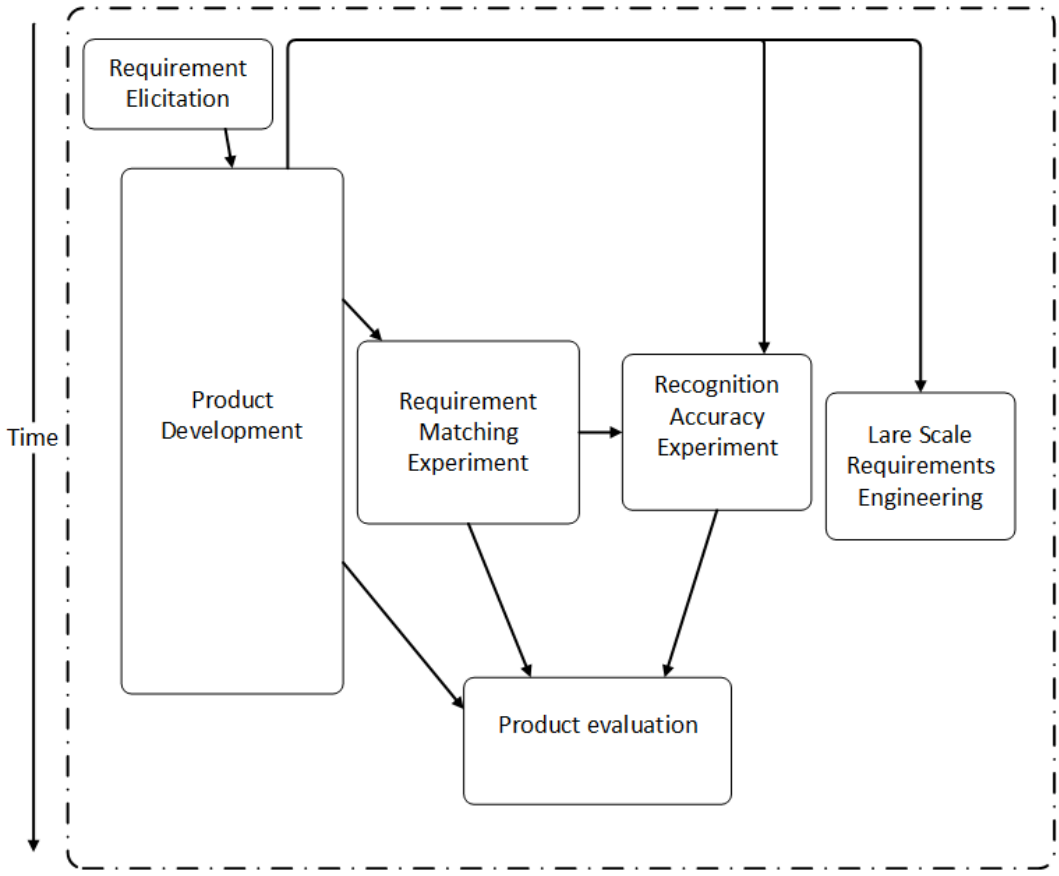

Figure 4.1: An overview of the design research.

4.1.1 Development

The product was developed with an agile process. The main objectives of the product was:

- **The main action for giving the system input should be through speech:** To be able to investigate and evaluate how well a speech recognizer behaves in a technical domain.
- **To create a connection between the mobile smart device application**

and a requirement database: To be able to utilize the mobility offered by a smartphone to annotate SRSs.

- **Annotations on requirements:** To investigate how a requirement could be annotated from a technical perspective (e.g. as meta-data or as an attribute etc)
- **Extract the annotation data from the requirements:** To visualize and analyze the data that the annotations provided.

The sprint lengths for the project was one week. The smart device application and SpeechServer as described in chapter 5 were developed in parallel. The solutions aimed to be as general as possible to prove the concept of annotating requirement entities.

4.1.2 Requirement Elicitation

The requirements on the system originated from the the authors as product owners, as well as from interviews and discussions with requirement engineers, application engineers and software developers at Systemite AB. After each sprint new problems were identified, requirements were elicited and prioritized, which allowed for an adaptive and responsive process.

4.1.2.1 Interview

In the very beginning of the thesis an interview was conducted to give an understanding about the background information of SystemWeaver and requirement management today, but also to elicit requirements on the system that was developed. The interview subjects were application engineers and software developers at Systemite with a span of three to ten years of experience in requirement management. The elicitation interview focused on four main objectives.

- **Personal information:** To get an understanding of the subjects educational background and experiences, as well as their role in the company and their usage of SystemWeaver.
- **SystemWeaver:** Since the smart device would be integrated with SystemWeaver, the purpose was to get a better understanding of SystemWeaver's semantics, structure, flaws and advantages. It was also necessary to find out how companies and people use SystemWeaver.
- **Requirement management:** To get insight into how people who work with requirement and SRSs every day experiences how the change management and maintenance of the documents functions.
- **Speech recognition:** A brief questioning about if the subject had any earlier experience with speech recognition software, and if so, what advantages and limitations they felt the software had.

The questions that the interviews were based on can be found in appendix A

4.2 Requirement Matching Experiment

Even though speech recognition algorithms have improved in recent years the error rates can still be high. This is especially true for the speech recognition available in mobile devices which is typically speaker-independent (which has higher error rates) and has a limited compute power. Our initial results also confirmed that there was often a large discrepancies between the intended requirements supplied as speech input and the output from the speech recognizer. To make requirements identification possible we extended the system with the Levenshtein string distance as described in sectino 5.4.1.1. To evaluate how well this solution performed an experiment was conducted.

The experiment had 10 subjects, all who were students at Chalmers University of Technology, and all with English as their second language. Each student was given 10 randomly selected requirement titles from an SRS with 4544 requirements and were asked to speak the titles into the speech recognizer. The most likely output from the speech recognizer was then saved and used to evaluate lookup of requirements in that SRS through the Levenshtein edit-distance algorithm. For each input we noted the input length (string length of output from speech recognizer), the target requirement length, distances from input to all requirement titles in the SRS, as well as the rank of the target requirement in the ranking created by the distance function. An example of the data saved for each input can be found in Table 4.1

The algorithm was extended to also check for perfect matches (i.e. the interpreted input was exactly the same as the requirement title), and if it was found, the position of the returned result from the speech recognizer were saved. This extension was made to utilize that the speech recognizer returned five different interpretations of the spoken input. If a perfect match was found in any of the five results, the input was never run through the distance function. If a perfect match was not found, only the most likely result from the speech recognizer was run through the distance function.

Requirement matching experiment design

Purpose: To evaluate how a string edit distance function can help to find a sought requirement in a specific SRS.

Motivation: Because there are inaccuracies in speech recognizers, we want to investigate and evaluate how string edit distance function can help mitigate the accuracy problems by returning similar strings to the given input.

Subject Sample: 10 randomly selected students from the Software Engineering programme at Chalmers University of Technology (both undergraduate students and graduate students). The subject have English as their second language.

Experiment Setup: The requirement titles (which is a short description of the requirement, the human readable ID) are taken from an SRS from automotive industry , and the context within that SRS which has the most requirements in it (a context is defined as a delimited part of an SRS such as different abstraction levels (e.g. Analysis Level, Design Level, Implementation Level)). 10 requirements for each subject are randomly taken from the the selected context.

Execution: The subject is asked to say each of the 10 selected requirements. The requirement titles are shown to the subjects on a screen. Each input of a title is followed by a pause and the subject will have as much time as they need to read and prepare for the next input.

For each requirement title, the input is recorded by an Android device and the results from the speech recognition engine (see section 5.2.2) are recorded in a file. Each subject's data will be recorded in separate files.

Evaluation: From the files, the best result for each requirement title is run through the requirement matching algorithm to find what ranking it places the correct requirement. The ranking is based on the requirement title's string edit distance in comparison with all other requirement titles' string edit distances in the same context (using Levenshtein edit distance algorithm, an example is illustrated in Table 4.1). The evaluation will also look at *perfect matches*, which is defined as 1 of the 5 results from the speech engine being exactly the same as the sought requirement title (it will also look at how often these occur and how often a perfect match is found, but it is not the correct requirement).

Table 4.1: Table showing how Levenshtein distance and ranking is calculated for a given input in a closed domain of requirement titles.

Input	Requirement titles	Distance	Rank
And its allies buffet	Initialize buffer	9	1
	Erroneous APLM buffer	15	3
	State monitoring buffer	14	2
	Transaction: Buffer on - Buffer off	26	5
	Buffer out of range	18	4

4.3 Recognition Accuracy Experiment

Experiment 2 was conducted to evaluate how the speech recognizer behaved with free text input. Even if this is a typical use of speech recognition we are targeting a specific, technical domain and not everyday speech/text. Thus, the performance of speech recognition can be expected to be worse. Since free-form speech recognition would be needed in allowing arbitrary annotations to requirements this is a key concern for our application.

From the same SRS as mentioned above (see section 4.2), 100 sentences were randomly selected . The same 10 students as mentioned above were then asked to speak 10 of the sentences into the speech recognizer.

The best interpreted result from the speech recognizer was then saved in a text file and were later given a value between 0-3, were the numeric defined the level of severeness of the error, as can be seen in Table 4.2. The design of the experiment can be seen below.

Recognition Accuracy Experiment Design

Purpose: To find out how accurate speech to text as input is for technical descriptions of requirements.

Motivation: To investigate the speech recognizers accuracy on sentences and evaluate if it is a viable option to have as free text input towards technical domains.

Subject Sample: 10 randomly selected students from the Software Engineering programme at Chalmers University of Technology (both undergraduate students and graduate students). The subject have English as their second language. (Same subjects as in the Requirement matching experiment)

Experiment Setup: 100 sentences are taken from the description from the same requirements as in Requirement Matching Experiment 4.2

Execution: The subject each says ten of the sentences. The input is recorded by the Android speech recognizer, and the interpreted inputs are saved in a file.

Evaluation: The evaluation will look at the *severity* of the input error on a scale of 0-3 as described in Table 4.2. The result is presented in a table to see how much errors are introduced by the free text speech input.

4.4 Large Scale Requirements Engineering

Software projects out in the industry can be very large in terms of complexity and size. To evaluate how scalable our solution was, a test suite was ran on medium-scaled and

Table 4.2: Table showing interpretations of severity.

Severity	Meaning
0	No error
1	Minor error, still comprehensible
2	Error, hard/impossible to comprehend
3	Severe Error, meaning of text has changed, but has a viable meaning in the context

large-scaled SRSs (Regnell et al. 2008) taken from the automotive industry.

The purpose was to test how the product behaved on different sized software projects, and variables were measured when the product was in action. The variables that were interesting to look at were:

SRS size: The number of requirements within the SRS

Execution time: How much time (in milliseconds) it took from the moment the server received the input until it have fetched the results, and is ready to send to the client. This to evaluate how the system scales in execution time.

For each of the SRSs, a requirement was identified. A user then gave the name of the identified requirement as input to the system through speech. Each SRS was queried 30 times and the different values for the above mentioned variables where then saved for analysis.

4.5 Product Evaluation

The end user evaluation of SpeechWeaver contained two parts, monitored user tests followed by interviews. The subjects for the user test was five requirement- and application-engineers at Systemite AB.

The subjects was given a short introduction about the features and the design of the application and was then given five randomly selected requirements within a chosen context and was asked to create an issue to each of the requirement with the application. Out of these five, two of the issues required an attached file (see Section 5.2.1). After the requirements were annotated the user sat down and explored the result of the actions by looking at the updates that had been added in SystemWeaver (see Section 2.5). Finally the user was interviewed about the experience of using the application. The main focus of the interview was the experience of using speech as input, but also overall impressions of the usability and ease of use of the product as a whole. The questions the interview was based on can be seen in Appendix C.

5

Technical solution

This chapter will discuss the technical solution and implementation choices made. The chapter will give an overview of the system architecture, and continue on to present the individual parts. The chapter will also cover some solutions in detail and present example data from the system.

5.1 System Architecture

The overall architecture exists of clients and a server (called SpeechServer). The client is an application implemented in Java for the Android platform. The client is the GUI that interacts between the user-input and the SpeechServer. Between the client and the SpeechServer a connection is established by TCP/IP and sends strings captured through the speech recognizer on the smart device. The strings that are sent between SpeechServer and the client follows an internal protocol which is described in Section 5.3.

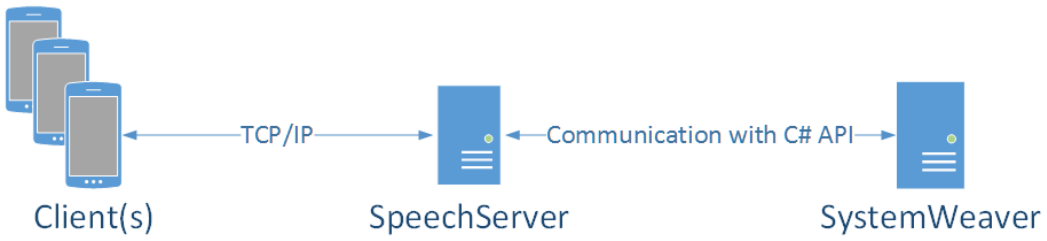

Figure 5.1: Architectural overview.

SpeechServer is the connection between the client and the requirements in SystemWeaver. SpeechServer interacts with SystemWeaver through a C#-API provided by Systemite. This allows for SpeechServer to take input from the client and work with the requirement entities in SystemWeaver.

5.2 Client

All user interaction is performed with the client application in SpeechWeaver system, which is developed for Android 4.1 and above to facilitate offline voice typing (Davies 2012). However, because of the string based communication, the system does not require an Android based client, only that the client can send and receive TCP/IP messages. Hence, it is possible to develop clients for different platforms, e.g. desktop applications or web applications, in order to take advantage of their respective advantages in different use cases. However, in this project the focus has been on smartphones in order to take advantage of the mobility that they provide. Therefore all further references in this article to "the client" will refer to the developed Android client.

5.2.1 Usage

The client follows the protocol as described in Section 5.3. Within the application, information that is sent from the server to the client is spoken by the application by Android's text-to-speech engine. Before the user can interact with the requirement database, the user types in and sends user credentials (username and password) to the SpeechServer for validation in SystemWeaver. Once the client is authorized, the user is presented with available projects and the user taps to select the project that user wants to search in. Once a project has been selected, there are three possible actions that can be taken, either:

1. Annotating a requirement - By tapping a button and provide the server with a requirement ID and then an annotation (through their speech)

2. Annotating a requirement with extra data - By selecting a file from the smart device (e.g. a picture or a recorded sound file) and after that proceeding with the actions mentioned in 1. After those actions are done the application also uploads the selected file to the SpeechServer.

3. Generate a report - By tapping a button which triggers the SpeechServer to generate a report (see section 5.6).

A picture illustrating the client's flow can be seen below in Figure 5.2

Figure 5.2: The sequence flow of the client

The use of auditory interaction with SpeechWeaver removes all need for the user to focus on the application. When the server expects input from a client, an identifier and an explanatory string is sent to the user which is then spoken by the application through Android's text-to-speech-engine. However, to start any of the above mentioned actions, the user needs to tap a button on the screen. Action 1, *Annotate a requirement* can however also be initiated through the button on a bluetooth headset. Thus facilitating concurrent usage of the system during, for instance, requirements review.

5.2.2 Speeech Recognition

The Android application captures the users voice using the built in API for speech recognition (*RecognizerIntent* 2013). When launching the speech recognizer, a pop up dialog appears and a sound is generated to inform the user that the speech recognizer is ready to capture input. The speech recognizer will then record the users input until it does not receive any new input for one second, i.e. until the user goes silent after stating the desired phrase. This event will cause the pop up dialog to close and a sound is generated to to inform the user that the speech recognizer has caught the input and is processing it. When the speech recognizer has processed the input, it returns up to five results. These results are based on how confident the recognizer is that the returned result was correct, where the first of the five results is what the recognizer interpreted as

the most likely input from the user. An example of the results from a speech recognizer can be seen below in Table 5.1

Table 5.1: The five results returned from Android's speech recognizer for the input "Speech recognizer and confidence scores.

Rank	Result
1	Speech recogniser and confidence scores
2	Speech recognizer and confidence scores
3	Speech recogniser and confidential scores
4	Speech recogniser and confidence course
5	Speech recognizer and confidential scores

All five alternative user input strings are then sent to the SpeecServer for post-processing.

Android's standard speech recognizer supports different languages. However, through out this project, English US was selected used as the primary input language, meaning that the domain of possible interpretations for the speech recognizer were words from the american english dictionary. In addition to this dictionary, the speech recognizer also provides the commands "period", "comma", "exclamation mark/point" and "question mark" which are translated into their corresponding characters (*Type text by speaking* 2013).

5.3 Communication Protocol

The communication between SpeechServer and the clients is handled by a custom protocol built on tags that represent different states in the communication. The tags are sent over a TCP/IP-socket and are interpreted at the receiving end. The tags attached to the messages are dependent on which state the system currently is in, i.e. when the client trigger the annotation by sending a requirement title, the messages is tagged with ID (Identification). When the server has found a requirement title (the ID), it returns a message with the tag WFT (Waiting For Tag) and the next output containing the annotation from the client to the server is tagged with ANOT (Annotation).

During the setup-phase, after a connection has been established between the client and the server and the user has logged into the system, the server acts as the active part to form a context for the whole system to work within. Once the setup-phase is over, SpeechServer takes a passive role where different operations are triggered based on the what is received from the client. This implementation allows the communication states to be decided from the client and also means that they don't need to be followed if the client decides to abort the current communication state. Once the setup-phase is over, operations (sending an ID, sending an annotation, create a report etc) are always

triggered by the client. The different operations can at any point be aborted and re-selected by the client.

5.4 SpeechServer

The server is written in C# and has one thread for each client and can handle multiple clients simultaneously. Since SystemWeaver is a real time collaboration tool used in the industry that handles large requirements databases, the API and SystemWeaver can handle many simultaneous users with ease. Clients connect to SpeechServer by sending input to a predefined socket that the server listens to. User authentication, in turn, is handled by checking with SystemWeaver that the provided username and password are both valid.

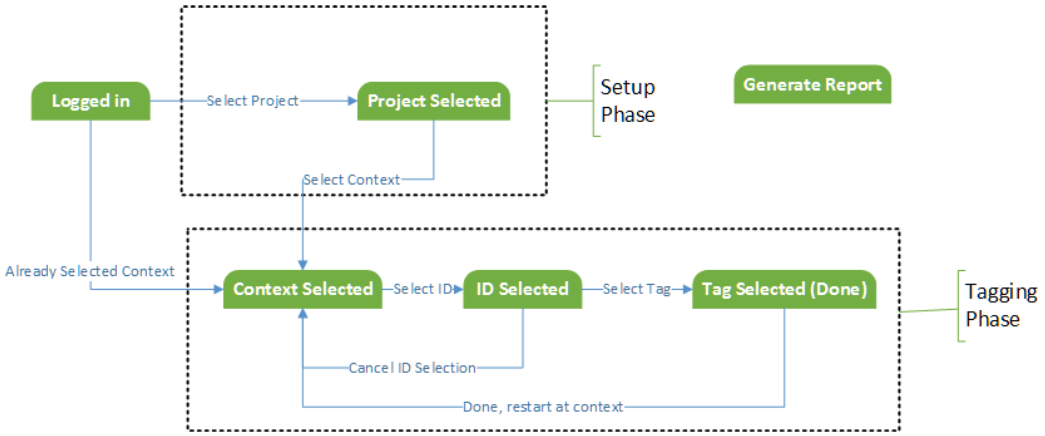

Figure 5.3: A simplified state machine showing the states of the SpeechServer handling one client.

As soon as a client connects to the server, the client is given a broker-object by the C# API to be able to interact with SystemWeaver. The broker provides the client with all necessary operations to interact with SystemWeaver, such as authentication, fetching items/parts and writing to SystemWeaver through a connection that is kept alive as long as the client stays connected.

The flow in the server is described by a simplified state machine shown in Figure 5.3. If the client has not provided a project to work in, the server fetches all the projects with the broker, and sends all available projects to the client and asks which project to work in. The contexts from a selected project, which is a delimited part of a project or an abstraction level (e.g. Analysis Level, Design Level, Implementation Level) are treated in the same way by the user, but is traversed to from the project node. The server sends all context within that project and the user selects one. This is the *Setup Phase* which is accessible from all other states if the user should choose to change.

After the setup phase, the *Annotation Phase* begins, this is where the system adds the annotations. As the client provides an ID-tag together with the string input for the title

of the requirement, the SpeechServer tries to locate it using methods described further in Section 5.4.1. When the name is found on the server it stores this temporarily, and asks the client for what annotation to annotate the requirement with. When the annotation is chosen by the user it gets processed into the one of the predefined annotations in the server and annotates the requirement with the specified annotation by adding an issue to the requirement via the C#-API of SystemWeaver.

5.4.1 Finding a Requirement Using Natural Language Processing

When the server processes the string input it does it in several steps and is dynamic depending on its accuracy of the results. Using name input as the example in the following scenario we will describe the process. The client provides several strings for interpreted results in the order of likelihood (weighted on the confidence scores as described in 5.2.2). First the server looks for a perfect string match for the five inputs. If no perfect matches were found, the Levenshtein distance algorithm as described in 5.4.1.1 is used to find the most similar string, from now on called closest matches, on the most likely interpreted result from the speech recognizer. Then the server returns the closest matches sorted by their Levenshtein distance from the input string so the user can choose from these.

There are a few reasons why not to handle the entire input string in one message (as described in the Section 3.2). The first reason is that there are problems breaking the input string down in its parts since it's complicated to find which part of the string is the id, and what parts are annotations. If the system would have listened for special syntax (example highlighted in bold) such as:

> ***Requirement:*** *"System should not create a file."* ***Annotation: "Ambiguous"***

there is a possibility that the requirement could contain these keywords or other syntax words which would result in a faulty delimitation of the string. The second reason is that a single string solution requires more from the user since he/she needs to remember and formulate the syntax, keywords, requirement ID and the annotation before speaking (Wilding 2001). Having separated the input into different stages lessens the burden for the system to compensate for these issues, as well as lessen the burden for the user since the user now only needs to answer the questions stated by the application. The question-response communication implementation also makes failure mitigation easier if an operation fails since the user can quicker just abort or retry the operation.

5.4.1.1 String Edit Distance Function - Levenshtein distance

A way of improving the speech recognizer is to add an algorithm which gives us an indication of what input the user *wanted*, as opposed to what the speech recognizer interpreted. When searching for entities within a finite interval, as in the example with

The distance between two strings x,y given by $\text{lev}_{x,y}(|x|,|y|)$ where

$$\text{lev}_{x,y}(i,j) = \begin{cases} \max(i,j) & , \min(i,j) = 0 \\ \min \begin{cases} \text{lev}_{x,y}(i-1,j) + 1 \\ \text{lev}_{x,y}(i,j-1) + 1 \\ \text{lev}_{x,y}(i-1,j-1) + [x_i \neq y_j] \end{cases} & , \text{ else} \end{cases}$$

Figure 5.4: The edit-distance (Levenshtein distance) between two strings, x,y, where all operations has a cost of one.

requirement names in a requirement database, we improve this by measuring a string edit distance (see Section 3.2.2) between the interpreted input from the speech recognizer and the values in the target domain. This allows us to narrow down the number of possible results and thereby identify the correct result with higher certainty.

In Levenshteins string distance function, whether you add, delete or modify a character within a string, it is calculated as a cost of one. An string edit distance derived from an algorithm where this property is fulfilled is often called a Levenshtein distance (Blockeel et al. 2004).

In our system we use Levenshtein distances when a given input x does not have a a requirement name that perfectly matches x. We then calculate the distance between x and all requirements in the given database and returns a set of requirements with the lowest Levenshtein distances, and the user can then chose which requirement the user originally sought.

5.5 Annotations

In order to create high quality software requirements specifications (SRS), IEEE have developed a recommended practice for SRSs (*IEEE Recommended Practice for Software Requirements Specifications* 1998). The guideline also shows how the practice connects to software lifecycle processes (*IEEE/EIA Standard Industry Implementation of International Standard ISO/IEC 12207: 1995 (ISO/IEC 12207) Standard for Information Technology Software Life Cycle Processes* n.d.). The decision was to use chapter *4.3 - Characteristics of a good SRS* from IEEE Recommended Practice for Software Requirements Specifications as the basis for the annotations.

The definition of a high quality SRS used in the context of SpeechWeaver is that all requirements and requirement groups should conform to the characteristics described in Table 5.2. This is somewhat broader than the practice since it only discusses the whole SRS and the particular requirements, not intermediate collections of requirements and individual requirements.

The annotations provided are the complementary antonyms of these characteristics, with the exception of *Unprioritized* since it captures the meaning better, is a single word and is more understandable in natural language. These annotations are just a suggestion

Table 5.2: IEEE Quality Characteristics and the corresponding SpeechWeaver Annotations.

IEEE Quality Characteristics	SpeechWeaver Annotations
Unambiguous	Ambiguous
Correct	Incorrect
Complete	Incomplete
Consistent	Inconsistent
Ranked for importance and/or stability	Unprioritized
Verifiable	Unverifiable
Modifiable	Unmodifiable
Traceable	Untraceable

that captures many of the problems a requirement can have. This is customizable per customer and organization, to suit problems that a particular organization encounters. This is also customizable in run time and can be changed and perfected continuously during a project to give more narrow definitions of the annotations. It is also possible to have synonyms to the annotation tags, so that a user can say *"Not clear."* or *"Not well defined"* instead of *"Ambiguous"*. This reduces the burden to remember exact syntax, but all these synonyms maps to the same tag within the system, that is *Ambiguous*. This is needed to be able to interpret the tags in the same way, as well as to be able to sort and filter within SystemWeaver and extract the annotations for aggregations in the report generation.

5.6 Report Generation

The annotations made by the users are stored in SystemWeaver as individual entities, this allows several users to annotate the same requirement with the same annotation several times. These annotations are stored in an *issue* (see section 2.5.3)connected to the requirement, and the connection can be traversed in any direction. The report generation uses these annotations while iterating over the report structure. The output of the report generation are the statistics for the requirements and how many have been annotated with the different annotations. An example can be seen in Table 5.3 and Table 5.4. Note that the first table shows how many requirements are tagged with the different annotations. This means that if a requirement is annotated several times with the same tag by different users, i.e. several people think that a requirement is *ambiguous*, then the annotation is only counted once for that requirement, to see the status over all requirements. This also means that one requirement can be represented in all categories. In the other table we see the total number of annotations, i.e all annotations are counted.

The generated report also gives annotations grouped in a headline so the reader can

Table 5.3: Example statistics from report generation output, annotated requirements.

SpeechWeaver Annotations	Tagged RQs	%
Ambiguous	5	1%
Incorrect	10	2%
Incomplete	50	10%
Inconsistent	1	0.2%
Unprioritized	10	2%
Unverifiable	100	20%
Unmodifiable	20	4%
Untraceable	80	16%
Total: Annotated requirements	**276**	
Total: Requirements	**500**	

Table 5.4: Example statistics from report generation output, statistics on annotations.

SpeechWeaver Annotations	Annotation Count	%
Ambiguous	10	3.3%
Incorrect	20	6.7%
Incomplete	60	20%
Inconsistent	5	1.6%
Unprioritized	15	5%
Unverifiable	110	36.7%
Unmodifiable	30	10%
Untraceable	50	16.7%
Total: Annotations	**300**	

see all requirements annotated with a specific annotation. Furthermore there is headlines for each requirement so the reader can see what problems a particular requirement has. The output shows the description for each requirement so the review process is simple.

The annotations can also be shown for specific time periods or iterations, depending on the project setup. This means that statistics can be shown over time and the data can be plotted to see the history and development of the annotations.

6

Results

During the project experiments were conducted to evaluate state of practice speech recognition and the efficiency and scalability as well as the usefulness of the developed system. The following sections will present the results of these experiments.

6.1 Speech Recognition

This section will present the findings from the experiments on speech recognition, see Section 4.2 and Section 4.3. The data from the experiments can be found in Appendix B and Table 6.1.

6.1.1 Recognition Accuracy

The results from the Recognition Accuracy Experiment (see 4.3) in Table 6.1 shows that free text speech input is not good enough to provide accurate descriptions as of now. As many as 64% are not comprehensible, and 10% can introduce errors if interpreted as written. The number of samples is 98 rather than the stated 100 in the Method chapter (see Chapter 4), this is because the experiment equipment failed to record two of the inputs from two different subjects.

Table 6.1: Number of descriptions in each severity ranking.

Severity	Frequency (n=98)	Explanation
0	8	No error
1	17	Minor error, still comprehensible
2	63	Error, hard/impossible to comprehend
3	10	Severe Error, meaning of text has changed, but has a viable meaning in the context

6.1.2 Requirement Matching

The requirement matching experiment showed that a string edit distance algorithm such as Levenshtein's edit distance can give satisfiable results for matching natural language to name conventions used in an SRS. When the speech recognizers interpreted input of the subjects spoken title of requirements was run through the Levenshtein edit distance algorithm and used towards an SRS of 4544 requirements, 93% ended up in the rank of one to five. Figure 6.1 shows a graph for the percentage of requirements that was ranked among the N top requirements. From the graph we can see that for 73% of the requirements the target requirement was top ranked and in 93% of the cases the target rank was among the top five among the 4544 requirements in this SRS. The mean rank was 6.96 but this is dominated by a few requirements where the speech recognizer gave output very far from the target requirement; the median rank was 1.

The speech recognizer delivered 15 perfect matches, and of these 12 where at the first position of the returned results from the speech recognizer. In two of the cases the perfect match was at the speech recognizer's second result, and in one of the cases the perfect match was at the fifth result. In all of the cases, the input to the Levenshtein edit distance algorithm (i.e. the first result returned from the speech recognizer) had the Levenshtein rank 1. The ranges of the inputs that resulted in perfect matches were between 15 to 43 characters, this is most likely because longer strings are more vulnerable to error since any word can be misinterpreted, shorter requirement titles seemed to contain more abbreviations and symbols such as "->" or parentheses/brackets, presumably a consequence of that the recognizer's expected input is words of natural language. None of the given input returned a false perfect match.

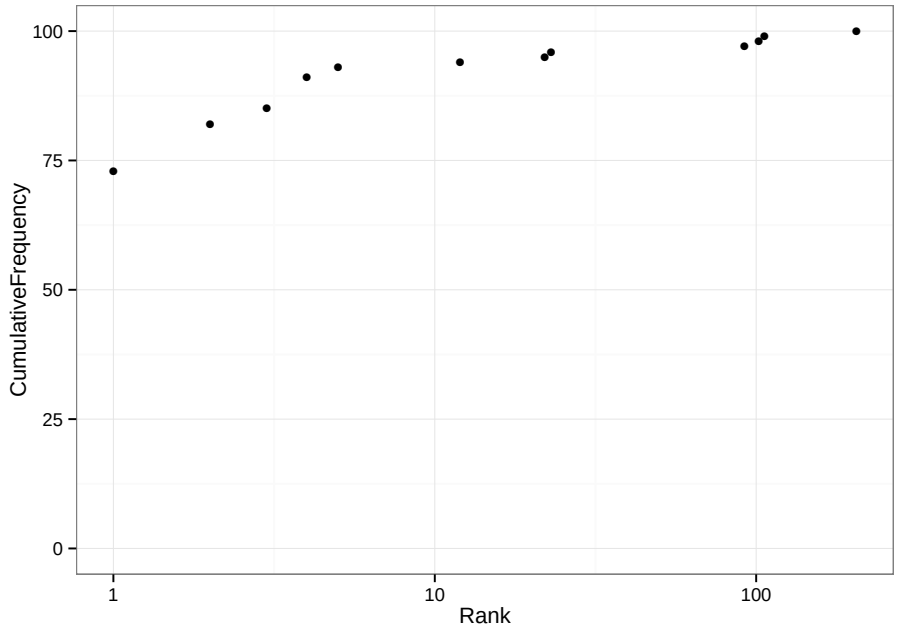

Figure 6.1: Graph showing the cumulative frequency and the edit distance rank

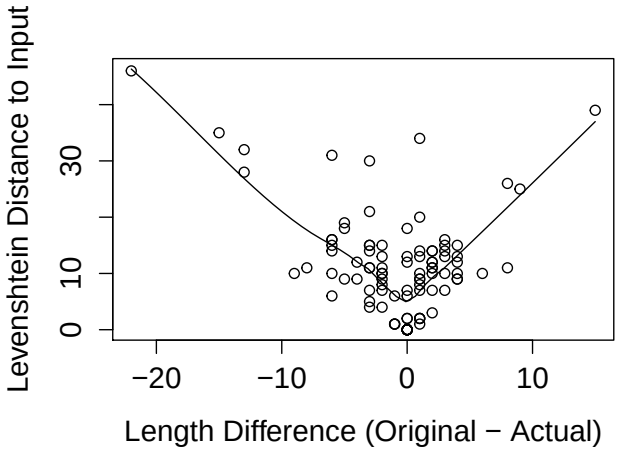

Figure 6.2: The Levenshtein distance and its correlation with the difference in input length, the line shows a locally fitted regression.

The correlation between Levenshtein distance to the sought requirement title and the difference in length between input and what the speech recognition interpreted can be seen in Figure 6.2. This figure shows that the Levenshtein distance gives better results for strings with similar length. Since the input and interpreted string length often is

quite close, this gives good results. This also gives a hint that it's better for the user to try to pronounce the whole title, even hard to pronounce symbols and abbreviations to make the matching better.

A table with the data gathered from the experiment is shown in Appendix B

6.2 Large Scale Requirements Engineering

Approximate times for the different look-ups are described in Table 6.2 and 6.3. The times shown are from the start of the look-up on the SpeechServer, until the SpeechServer has returned the items and is ready to send to the client. This shows that even for large scale requirements engineering these kinds of operations are feasible. The times in Table 6.2 shows that the number of projects (and contexts within a project) are so low that the times are irrelevant. The time to look up requirements takes a bit longer as can be seen in Table 6.3. SpeechServer and SystemWeaver is running on a laptop with an dual core Intel i5-350 processor with hyper-threading and 4GB RAM.

The times to look up items is sometimes unpredictable since SystemWeaver as well as SpeechServer does caching internally, this depends on what has been searched before and other factors, such as other users and how much can be cached. These results however are close to what can be expected. How the user perceives the interaction is discussed in Section 6.3.2.

Table 6.2: Approximate times to look up projects and contexts, mean of 10 runs.

Type	Quantity	Time in ms
Projects	89	2
Contexts	3	1

Table 6.3: Times to look up item requirements, mean of 31 runs.

Requirements	Time in ms
4544	1450
959	39
210	209
2548	797

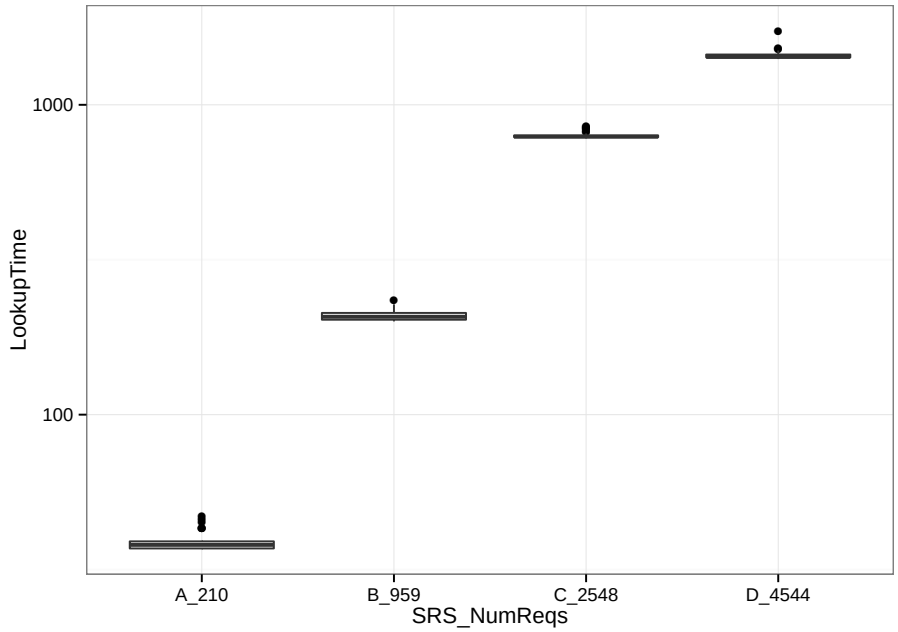

Figure 6.3: The boxplots show the distribution of the 31 runs for the different sizes.

6.3 Product Evaluation

The product was evaluated through a user-test and a following interview. What lied in focus during the interview was the annotations that were created in themselves, but also the user's experience of the application. The results from this interview will be divided into two parts, the annotations and the usefulness of those, and the overall experience of the application.

6.3.1 Annotations on Softer Artefacts

All of the interviewed subjects used TODO-lists in the format of analog or digital notes to their ongoing projects. However, what they found to be a huge profit with the annotations on the requirements was that there were a direct connection between the TODO-entity (the annotation) and the actual requirement.

A factor that was considered as very important for all of the interviewees were the content of the annotations. They found it very important that the naming of the annotations fitted the way they worked with maintenance of projects, and that there needed to be a clear definition and distinction between each and one of the annotations. The interviewees expressed that the annotations without any more descriptive information (a file or a description field) were better for peer-reviewing purposes (i.e. letting users go through the SRS and go through the annotations in a group format), where as an annotation that contained descriptive information were better for actual modifications

and improvements on the requirement.

6.3.2 User Experience

The key notes that the subjects expressed as very positive was the accuracy and the response time of the application.

Many subjects expressed the usefulness of the mobility in the application. To be able to query an SRS with several thousand of requirements anywhere, from the car or the bar, was considered a huge advantage. The solution was also considered very lightweight, and the flow combined with the low amount of time to create an issue were expressed as an easy solution to directly ventilate their thoughts towards the SRS, instead of a tidy operation when they would otherwise have to make several operations to navigate and interact with SystemWeaver.

Some felt uncomfortable using speech as the main input for the application. Moreover, requirement titles that contained special characters (e.g. ”#”, ”->”, ”=”) or abbreviations were found hard translate to spoken words. Another problem that was pointed out was the sound that was produced during the operation, and several of the subjects stated that it might not be so good to use it in shared environments.

7

Discussion

The product developed during this thesis introduces a new way of interacting with, and maintaining SRSs. Therefore, the concept is unexplored, and several findings were made during the development and evaluation of the product. This section will discuss these findings and possible outcomes of the concept. In the end of this section the authors will give their thoughts about the future work that could be applied to this thesis.

7.1 Implementation Discussion

During the course of the thesis, several design decision were made. In this chapter we will discuss the affect that those decisions had on the result and the solution.

7.1.1 Annotations - where?

In the solutions presented in this thesis, the annotations are connected as meta-data to the requirement that the annotation is associated with. There are several advantages with this solution. When having the annotations as meta-data they are isolated from the requirements, something that enables the requirement to be intact. Another solution that was considered and discarded was to have the annotations as attributes on the requirement. However, this would not allow for isolation. Having the annotations as meta-data also allows an easy way for several annotations to the same requirement. When the annotations are isolated, they are light weight and disposable.

7.1.2 Generelizability - SystemWeaver

The platform that the solution was built on was an already existing application product line managing software - SystemWeaver. However, the solutions that are presented aims to be as general as possible. If you break it down, what is needed for this solution to

work is a requirement database whom which you can query for individual and sets of requirements, and a way of creating and connecting issues to the queried requirements. In the solution presented, there are no special demands within the issue, nor the requirement, that is needed for the solution to work. However, SystemWeaver has proven to be very efficient. An equal solution for another system would of course give different results in terms of execution time dependent on the implementation of that system.

7.1.3 Levenshtein Edit Distance Algorithm

A core part of the improvement and adaption of spoken input in our solution is the string edit distance algorithm. The choice of using Levenshtein string edit distance algorithm affected the post-processing of the spoken input. Since the algorithm has the same weight on all operations (add, delete, modify) it makes the length of the strings a bigger influence. Although, as shown in the results (see Section 6.1.2), it gave satisfying results. However, it also disables the user to search for substrings, even if those are correctly interpreted by the speech recognizer.

Searching for requirements using a matching algorithm works well when the user knows exactly what requirement to search for, but is a problem if they are searching for a requirement containing a particular substring. It could be the case that the user is looking for a requirement that has *"acceleration"* in it, but the user does not remember the whole title. This could be solved with another search option looking for requirement titles containing exact substrings. However, using free text search requires more effort in terms of tweaking your search term to give few enough results and can be harder and more complex if you need to search for *"gearbox AND clutch"*. This extra effort in terms of speed and complexity might be worth it if the use case for the product would develop towards a more look-up oriented approach. The use of string metric algorighm could be chosen differently, but the focus was not to evaluate string metric algorithms, the results from the experiment shows that using Levenshtein gives a good accuarcy increase, but other algorithms might be more accurate for requirement titles. Some other algorithms could be Jaro-Winkler, Soundex, N-grams or Jaccard. These different algorithms could give different benefits for longer or shorter names, some might handle misspellings better and some might suit special characters better.

7.1.4 Android's speech recognizer

The speech recognizer was considered a black box which provided the ability to interpret spoken words. In this thesis we chose to work with Android's provided speech-to-text engine. The solution is built upon handling strings, something that makes the solution general for any speech-to-text engine. However, the "perfect-match" algorithm that was implemented using the five interpreted results based on the confidence scores from the recognizer was Android specific. If you were to use a speech recognizer which only caught one input you would have lesser likely hood of finding perfect matches. However, as shown in the results, 12 out of 15 perfect matches was at the first position and this would likely not decrease the experience dramatically.

7.1.5 Large Scale Requirements Engineering

In the solution presented there is mainly one operation that has the biggest impact on the time efficiency in the system, the look up of the entities (the requirements). The results presented in section Large Scale Requirement Engineering (6.2) shows that the system scales very well. Even when querying large scale SRSs the user is provided feedback within seconds. The look-up is done very efficiently due to that the requirement entities are cached in both the SpeechServer and the SystemWeaver server.

Direct improvements could be done to the edit distance calculation. As can be seen in section 6.1.2, 93% of the sought requirements ended up in a top five ranking. From this information, you could make cuts in the calculations to have a dynamic roof for when you stop to calculate the distance, e.g. if the given input has a higher distance than the fifth element in the list, then it does not need to complete the run of the algorithm and can return infinity.

7.2 Speech Recognition

This section discusses state-of-practice speech recognition. What lies in focus is the accuracy of, and found problems that are associated with, speech recognition.

7.2.1 Accuracy

The requirement matching experiment (6.1.2) shows that a speech recognizer combined with a string edit distance algorithm can be used very efficiently in a closed domain such as an SRS with thousands of requirements. However, it was also found that the speech recognizer was not sufficient enough for providing descriptive text in terms of technical sentences (see 6.1.1). The experiemnt also showed that 10% of the interpreted inputs could give another meaning to the SRS. This would be contradictive, and would lower rather than increase the quality of the SRS.

7.2.1.1 Problems Associated to Speech recognition

Other problems was that the speech recognition had problems finding requirements that was misspelled, since the speech recognizer only provide words spelled correctly (thankfully). If there is only one requirement misspelled this is no problem for example: *"re*k*uirement"* only has a distance of 1 if the user sought for *"requirement"* and would end up highest in the ranking. The problem arises when there are many requirements spelled correctly, but a few misspelled. e.g. *"requirement 1"*, *"requirement 2"*, *"requirement 3"* and *"re*ck*uirement 4"*, then if the user sought for *"requirement 4"*, then 1,2,3 would end up before 4 anyway. Moreover, the recognizer did not handle special characters and abbreviations in a satisfiable way. Due to the speech recognizer being built upon a spoken language (in this case English US) there were some homophone problems, e.g. the letter "C" was sometimes interpreted as "see". Another case was when two words are pronounced in the same way, such as "see" and "sea".

7.2.2 Mobility

Since today's smart devices have become powerful enough to have speech recognizers within them, something that was found during the product evaluation was that the mobility of the devices gave the advantage that you could maintain documentation from anywhere. Another thing that arose during the product evaluation was that it was a benefit to be able to connect annotations to the actual requirement immediately instead of having an icebox of updates to the system, separate from the system.

Since the speech recognition did not prove to be efficient for long descriptive sentences, other features of the smart devices were implemented. A smart device can take pictures, record movies or audio (such as interviews or short memos) and such information could compensate for the lack of efficiency from the speech recognizer. Direct solutions that were seen during this thesis were to attach notes/blueprints from white boards or from paper, as well as stating why a requirement needed to be updated with the origin from a recorded interview on the device.

7.3 Quality of SRSs and Requirements

This sections will discuss impact on the quality of SRSs and requirements. It will present problems that was found during the thesis that has impact on the quality or continuous work with quality. The sections will also present metrics and possible solutions for how annotations can be used and what impact they can have on the quality aspect of an SRS.

7.3.1 Duplicates in SRSs

Problems encountered was that sometimes names of requirements were duplicated, although they were distinguishable from each other within SystemWeaver thanks to the structure of the items. This is however a problem with speech searching where there are no visual clues on the structure, they could be presented with their parent node appended with the title, but this would require extra work both in terms of development as well as computing time since the linking is only done in one way, one would have to traverse from the top and find the parent node. Examples of this could be *"Gear 1 - Clutch should not * " "Gear 2 - Clutch should not * "* and so on. Another way of solving this would be to force the requirement titles to be unique in the first place. This however needs much more maintenance effort, since these rules might be hard to follow, and might hinder the creation of requirements. This could be built into the RQ-database tool, but better searching and displaying is probably the better approach.

7.3.2 Metrics

The annotations can be aggregated and used in different metrics calculated continuously as well as periodically. These metrics could be used for checkpoints that could be either time driven or event driven, to do inspections of the requirements. These metrics

can be presented and calculated as described in *IEEE Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering Measurement Process* (2009). These indicators can be used to trigger events, such as requirement specification reviews. An example could be if more than 10% of the requirements are tagged as ambiguous, a review of the requirements should take place. These metrics could also be used in stage gates, e.g. before continuing to the next step all of specified annotation types must be resolved.

7.3.3 How can Annotations Increase the Quality of SRSs

If the product owner or requirement engineer receive feedback in the form of annotations, continuously in an agile fashion, that person can act upon these annotations immediately. If quality issues in a requirement are corrected at once, before others read and misinterpret the requirement, some quality issues in the product might be avoided. The annotations can be done continuously over a project, but also at certain review sessions, where the solution can be used as a review tool. This is also one of the most mentioned use cases during the interviews (see section 6.3).

In a larger scale, this information can be helpful to do root cause analysis (RCA) and find out why these issues are introduced (lack of training, communication problems etc). This can be done using any RCA method, but an easy first step could be the 5-whys technique described by Bulsuk (2009).

Another interesting analysis could be to see what requirements (or type of requirements) that have the most annotations (or a specific category of annotations) to see what requirements that have much problems, it could be that they are vague, or highly complex. If a requirement has many annotations and there seems to be much problem understanding it, actions can be taken to mitigate these problems by having formal or informal discussions about the requirement.

7.3.4 What can Annotations be Used for?

If annotations are provided continuously during the lifetime of a project, the annotations can provide insight on when different quality issues occur. From this, trends could be seen for different annotations. This data can be useful to do analysis on the different annotations in the SRS. If one for example know that ambiguity is noticed when implementation has started, maybe effort can be done beforehand to reduce ambiguity. If it's known when different quality issues occur in projects it can be analyzed for project maturity status.

The annotation data could possibly be used in a similar manner to how defect inflow can be predicted as described by Staron & Meding (2008).

7.4 Future work

During the course of this thesis, several areas appeared that could be investigated in the future. In this section we present topic that we consider to be interesting to evaluate or

implement.

7.4.1 Investigate String Distance Algorithms for Speech Input Towards Technical Domains

As described in Section 7.1.3 work could be done to see what algorithms could be used to give more accurate rankings or faster results. Investigations could be done to see if algorithms in parallel give better results, or dynamic depending och content and length. There might be ways to customize algorithms for the particular domain as well.

7.4.2 Enabling SRSs in Smartphones

One byproduct of the thesis was the possibility of working with SRSs in smart phones with limited screen size. The application could be extended or other applications to include functionality operating with SRSs, such as finding information more rapidly. Users could ask the application for descriptions of specific requirements or finding any attributes of a requirement such as the owner/author of a requirement. This way, developers and testers can get fast reminders of requirements to always be up to date on current status.

7.4.3 Create Custom Word Lists in Organizations, or Adapted Corpora

Since problems was encountered related to language issues and speech recognition, we suggest that more work should to be done extracting organization specific language. Since there are differences in languages depending on what is common in different domains, the predictions can be made more accurate if all common domain specific words (and acronyms) are available in word lists as well as how they occur in natural language sentences. By having tailored corpora for different organizations the sentence and word predictions can be made better.

7.4.4 Investigate SRS Quality

It is described in section 7.3.4 how annotations can be used, but work needs to be done to investigate what real data from a project would look like, and what further conclusions can be drawn from the annotations. It needs to be done over a longer period of time to see if any trends can be seen and if annotation statistics are generalizable. The same goes for annotation predictions. The proposed future work for these annotations are only a fraction of what might be possible once tried in a real project and it is likely that the annotations can be altered and used for other purposes.

7.4.5 (Speech) Annotations for other Software Engineering Artefacts

This paper focused on annotating requirements in an SRS, the same system could also work for less formal requirements, such as user stories in agile development. We also

believe that more artifacts can benefit from lightweight, mobile and easy accessible tools. Other custom made tools for different tasks handling other (softer) software artifacts using similar technologies could be developed. These tools could be used by testers to report test statuses and notifications, it could be used in verification/validation to assign statuses and accept/decline parts of a system and so on.

8

Conclusion

This thesis presents the design, implementation and evaluation of a lightweight system for annotating requirements specifications. By connecting the default speech recognition available in a present-day (Android-based) mobile device to a commercial software development management tool, we can allow access and annotation of large-scale requirements databases from anywhere where you can reach the Internet.

Speech recognition has improved considerably in recent years and its accuracy has increased. However, one of our experiments show that the accuracy of speech recognition available in mobile devices today is not yet adequate to allow free text input, such as comments or larger edits, in the often specialized-vocabulary, technical domains of industrial requirements specifications (SRSs). Even when used for shorter inputs, such as the lookup of a requirement prior to annotating it, accuracy is often poor. By extending our system with a simple string distance measurement we could improve accuracy considerably for the more limited lookup and annotation tasks. With the off-the-shelf speech recognizer and input from 10 non-native English speaking test subjects (with English as their second language) the system ranked the targeted requirement among the top five requirements returned 93% of the time for 100 requirements randomly sampled from an SRS containing 4544 requirements.

Moreover, a scalability experiment showed that with a mobile device, you can query and work with large scale SRSs with feasible round-trip times. Results from the database lookups were presented to the users within seconds even for SRSs with several thousand requirements. Together with speech recognizers within mobile smart devices this enables users to reach and query large scale requirement databases anytime, anywhere.

In user testing with five requirement engineers and managers at our industrial collaborator the system was considered surprisingly accurate and responsive. The users considered the system a suitable addition to their workflow, in particular as a substitute for their present day lists of requirements to change and updates to be made. They also

noted that the system was useful in requirements reviews. For more extensive maintenance work it would need to allow longer input of more free form text. Some users also was concerned about the noise they would generate while using the application; they said they would be reluctant to use it in an open office layout.

Annotations on 'softer' software engineering artifacts, such as requirements, is an unexplored area. We have presented a way to use these annotations as a lightweight maintenance- and peer-review tool for marking requirements during development and maintenance of large-scale industrial requirements specifications. Future research should investigate the long-term effects of such annotations on the quality and efficiency of requirements evolution and maintenance. Our system can also be used to collect metrics on requirements churn, quality and change frequencies. To improve speech recognition accuracy future research could build up specific speech recognition locales and dictionaries specific to the technical terms commonly seen in the technical areas related to software engineering and development.

Bibliography

Al-Rawas, A. & Easterbrook, S. (1996), 'Communication problems in requirements engineering: a field study', *COGNITIVE SCIENCE RESEARCH PAPER-UNIVERSITY OF SUSSEX CSRP* .

Android Developers (2013), `http://developer.android.com/about/dashboards/index.html`. Visited April 2013.

Android Open Source Project (n.d.), `http://source.android.com/source/licenses.html`. Visited February 2013.

Baker, J., Deng, L., Glass, J., Khudanpur, S., Lee, C.-H., Morgan, N. & O'Shaughnessy, D. (2009), 'Developments and directions in speech recognition and understanding, part 1 [dsp education]', *Signal Processing Magazine, IEEE* **26**(3), 75 –80.

Ballinger, B., Schalkwyk, J., Cohen, M. & Allauzen, C. (2010), 'Language model selection for speech-to-text conversion'. US Patent App. 12/976,920.

Ballinger, B., Schalkwyk, J., Cohen, M., Allauzen, C. & Riley, M. (2011), 'Speech to text conversion'. US Patent App. 13/249,181.

Barra, H. (2012), 'Google i/o 2012 - keynote day 1', `http://www.youtube.com/watch?v=VuC0i4xTyrI`. Visited January 2013.

Bjarnason, E., Wnuk, K. & Regnell, B. (2011), Requirements are slipping through the gaps—a case study on causes & effects of communication gaps in large-scale software development, *in* 'Requirements Engineering Conference (RE), 2011 19th IEEE International', IEEE, pp. 37–46.

Blockeel, H., Demoen, B., Duval, E., Lavrac, N. & JACOBS, N. (2004), 'Relational sequence learning and user modelling'.

Bulsuk, K. G. (2009), 'An introduction to 5-why', `http://www.bulsuk.com/2009/03/5-why-finding-root-causes.html`. Visited May 2013.

Chemuturi, M. (2010), *Mastering Software Quality Assurance : Best Practices, Tools and Technique for Software Developers*, J. Ross Publishing Inc., Ft. Lauderdale, FL, USA.

Cunningham, H. et al. (2000), *Software architecture for language engineering*, University of Sheffield.

Davies, C. (2012), 'Android 4.1 jelly bean adds offline voice typing', `http://www.slashgear.com/android-4-1-jelly-bean-adds-offline-voice-typing-27235871/`. Visited April 2013.

Davis, K. H., Biddulph, R. & Balashek, S. (1952), 'Automatic recognition of spoken digits', *The Journal of the Acoustical Society of America* **24**(6), 637–642.

Denger, C. & Olsson, T. (2005), Quality assurance in requirements engineering, *in* A. Aurum & C. Wohlin, eds, 'Engineering and Managing Software Requirements'.

Discover Android (n.d.), `http://www.android.com/about/`. Visited April 2013.

Elgin, B. (2005), 'Google buys android for its mobile arsenal', `http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal`. Visited January 2013.

Forward, A. & Lethbridge, T. (2002), The relevance of software documentation, tools and technologies: a survey, *in* 'Proceedings of the 2002 ACM symposium on Document engineering', ACM, pp. 26–33.

Gervasi, V. & Nuseibeh, B. (2002), 'Lightweight validation of natural language requirements', *Software: Practice and Experience* **32**(2), 113–133.

Gruenstein, A. (2010), 'Android developers blog', `http://android-developers.blogspot.se/2010/03/speech-input-api-for-android.html`. Visited March 2013.

Hamming, R. W. (1950), 'Error detecting and error correcting codes', *Bell System technical journal* **29**(2), 147–160.

Helal, S., Bose, R. & Li, W. (2012), 'Mobile platforms and development environments', *Synthesis Lectures on Mobile and Pervasive Computing* **7**(1), 1–120.

Highsmith, J. & Cockburn, A. (2001), 'Agile software development: The business of innovation', *Computer* **34**(9), 120–127.

Ibrahim, N. (2012), 'An overview of agile software development methodology and its relevance to software engineering', *Jurnal Sistem Informasi* **2**(1).

IEEE/EIA Standard Industry Implementation of International Standard ISO/IEC 12207: 1995 (ISO/IEC 12207) Standard for Information Technology Software Life Cycle Processes (n.d.), *IEEE/EIA 12207.0-1996* pp. i–75.

IEEE Recommended Practice for Software Requirements Specifications (1998), *IEEE Std 830-1998* pp. 1–40.

IEEE Standard Adoption of ISO/IEC 15939:2007 Systems and Software Engineering Measurement Process (2009), *IEEE Std 15939-2008* pp. C1–40.

IEEE Standard Glossary of Software Engineering Terminology (1990), *IEEE Std 610.12-1990* pp. 1–84.

Industry Leaders Announce Open Platform for Mobile Devices (2007), `http://www.openhandsetalliance.com/press_110507.html`. Visited January 2013.

Joshi, S. & Deshpande, D. (2012), 'Textual requirement analysis for uml diagram extraction'.

Juristo, N., Moreno, A. M. & Silva, A. (2002), 'Is the european industry moving toward solving requirements engineering problems?', *Software, IEEE* **19**(6), 70–77.

Kauppinen, M., Vartiainen, M., Kontio, J., Kujala, S. & Sulonen, R. (2004), 'Implementing requirements engineering processes throughout organizations: success factors and challenges', *Information and Software Technology* **46**(14), 937–953.

Levenshtein, V. (1966), Binary coors capable or 'correcting deletions, insertions, and reversals, *in* 'Soviet Physics-Doklady', Vol. 10.

Morrill, D. (2008), 'Android developers blog', `http://android-developers.blogspot.se/2008/09/announcing-android-10-sdk-release-1.html`. Visited February 2013.

Nuance (2013), 'Dragon speech recognition software', `http://www.nuance.com/dragon/index.htm`. Visited April 2013.

RecognizerIntent (2013). Visited April 2013.
URL: *http://developer.android.com/reference/android/speech/RecognizerIntent.html*

Regnell, B., Svensson, R. B. & Wnuk, K. (2008), Can we beat the complexity of very large-scale requirements engineering?, *in* 'Requirements Engineering: Foundation for Software Quality', Springer, pp. 123–128.

Richmond, S., Barnett, E. & Williams, C. (2011), 'Apple iphone 4s event: as it happened'.

Ross, D. & Schoman Jr, K. (1977), 'Structured analysis for requirements definition', *Software Engineering, IEEE Transactions on* (1), 6–15.

Saiedian, H. & Dale, R. (2000), 'Requirements engineering: making the connection between the software developer and customer', *Information and Software Technology* **42**(6), 419–428.

Schalkwyk, J., Beeferman, D., Beaufays, F., Byrne, B., Chelba, C., Cohen, M., Kamvar, M. & Strope, B. (2010), 'Google search by voice: A case study', *Visions of Speech: Exploring New Voice Apps in Mobile Environments, Call Centers and Clinics* **2**, 2–1.

Shankland, S. (2012), 'Android 4.1 gets faster; better notifications; google now'.

Staron, M. & Meding, W. (2008), 'Predicting weekly defect inflow in large software projects based on project planning and test status', *Information and Software Technology* **50**(7–8), 782 – 796.

Systemite Products and services (n.d.), `http://systemite.se/content/products-services/`. Visited February 2013.

SystemWeaver help manual (n.d.).

Tseng, M. & Jiao, J. (1998), 'Computer-aided requirement management for product definition: a methodology and implementation', *Concurrent Engineering* **6**(2), 145–160.

Type text by speaking (2013). Visited May 2013.
URL: *http://support.google.com/android/bin/answer.py?hl=en&answer=1650147*

Uren, V., Cimiano, P., Iria, J., Handschuh, S., Vargas-Vera, M., Motta, E. & Ciravegna, F. (2006), 'Semantic annotation for knowledge management: Requirements and a survey of the state of the art', *Web Semantics: science, services and agents on the World Wide Web* **4**(1), 14–28.

Voice action commands (2013), `http://support.google.com/android/bin/answer.py?hl=en&answer=1715292`. Visited April 2013.

Wiegers, K. E. (2009), *Software requirements*, Microsoft press.

Wilding, J. (2001), 'Over the top: Are there exceptions to the basic capacity limit?', *Behavioral and Brain Sciences* **24**(01), 152–153.

Wnuk, K., Gorschek, T. & Zahda, S. (2012), 'Obsolete software requirements', *Information and Software Technology* .

Zhang, Z., Arvela, M., Berki, E., Muhonen, M., Nummenmaa, J. & Poranen, T. (2010), 'Towards lightweight requirements documentation', *Journal of Software Engineering and Applications* **3**(9), 882–889.

A

Elicitation/Background - Interview Questions

Introduction
A presentation of the background, content and purpose of the interview.

1. **Personal information**
   (a) Present who you are and your background?
   (b) What is your role (in the company)?
   (c) What is your role as a SystemWeaver user?
2. **SystemWeaver usage**
   (a) For how long have you been using SystemWeaver?
   (b) How do you use SystemWeaver?
   (c) What do you experience as the advantages with the product?
   (d) What do you experience as obstacles using the product?
   (e) Have you tried other requirement management software?
   If so, what do you feel is the main difference between SystemWeaver and those?
   If not, why?
3. **Requirement management**
   (a) What information do you provide to the documentation?
   (b) What information do you use in the documentation?
   (c) Do you have a structured way of working with requirements?

(d) How often do you feel a need of change in the documentation (i.e. in which situations do you observe that you need to do changes/additions/comments to requirement documentation, design-documents, etc)?

(e) What are the causes for these changes? (i.e. the requirement is wrong, it is poor stated, it is ambiguous or simply wrong etc.)

(f) When does these changes/additions/comments happen? (i.e. are you going through the documentation after you have been doing your task or are you doing them in parallel?)

(g) When would you like these changes to occur?

(h) What information do you think a requirement should contain?

(i) How are other people affected based on your contribution to the requirements and documentation?

(j) How do you assess the quality of the requirements?

4. **Voice Recognition Software**

(a) Have you tried any speech control applications

(b) What do think about it/them? / Why are you not using it?

5. **Open question**

(a) Do you have any questions?

(b) Is there something you would like to add?

B

Requirement Matching Experiment Results

Since we are not allowed to publish the requirement titles, they are removed together with the speech recognizers interpreted title. The false perfect match column is also removed (for readability), since there were no false perfect matches.

Input Length: Length of the string that went into the matching algorithm.

Original String Length: Length of the requirement title that was sought, and spoken by the subject.

Length Difference: The difference in length between the original and the actual input ($Original - Actual$)

Algorithm Rank: The ranking that the algorithm returned.

Levenshtein Distance to Input: The Levenshtein distance between the sought item and the input.

Levenshtein Distance to Input: Which of the inputs was a perfect match (as described in section 4.2), where 0 indicating no perfect match.

Subject: The subject, replaced by a letter for anonymity.

Table B.1: Experiment results

Input Length	Original String Length	Length Difference	Algorithm Rank	Levenshtein Distance to Input	Perfect Match	Subject
51	45	-6	1	14	0	A
25	19	-6	1	6	0	A
75	73	-2	1	10	0	A
56	53	-3	1	7	0	A
31	31	0	1	0	1	A
39	36	-3	1	11	0	A
31	32	1	1	14	0	A
65	59	-6	1	16	0	A
24	24	0	1	0	1	A
37	37	0	1	10	0	A
21	18	-3	22	15	0	B
42	44	2	1	11	0	B
28	29	1	1	2	0	B
13	11	-2	1	4	0	B
39	40	1	1	2	0	B
51	45	-6	23	31	0	B
27	24	-3	1	5	5	B
54	54	0	1	13	0	B
29	30	1	1	1	3	B
31	34	3	1	16	0	B
34	37	3	1	7	0	C
78	73	-5	1	19	0	C
33	31	-2	1	15	0	C
56	53	-3	1	4	0	C
32	36	4	1	13	0	C
34	32	-2	1	10	0	C
72	59	-13	1	28	0	C
32	24	-8	1	11	0	C
21	18	-3	2	11	0	C

42	44	2	1	11	0	C
39	37	-2	1	13	0	D
29	29	0	1	0	1	D
45	39	-6	1	15	0	D
24	24	0	1	2	0	D
39	36	-3	1	15	0	D
33	33	0	1	0	1	D
24	24	0	1	0	1	D
42	33	-9	2	10	0	D
43	43	0	1	0	1	D
42	38	-4	1	12	0	D
35	37	2	1	14	0	E
34	35	1	205	20	0	E
16	15	-1	1	1	3	E
28	25	-3	2	14	0	E
19	19	0	1	0	1	E
36	37	1	4	9	0	E
27	30	3	4	13	0	E
50	50	0	2	18	0	E
34	35	1	1	34	0	E
19	18	-1	1	1	0	E
30	29	-1	1	6	0	F
38	32	-6	1	16	0	F
30	32	2	2	14	0	F
25	19	-6	1	10	0	F
37	35	-2	1	9	0	F
67	64	-3	1	30	0	F
27	25	-2	3	11	0	F
51	55	4	1	9	0	F
37	38	1	1	2	0	F
45	54	9	4	25	0	F
53	68	15	102	39	0	G
26	30	4	4	10	0	G

38	46	8	106	26	0	G
31	35	4	2	15	0	G
50	51	1	1	9	0	G
36	36	0	5	12	0	G
33	36	3	5	15	0	G
29	33	4	1	9	0	G
19	19	0	1	2	0	G
42	39	-3	2	21	0	G
20	20	0	1	0	1	H
31	29	-2	1	7	0	H
23	25	2	1	3	0	H
19	19	0	1	0	1	H
15	15	0	1	0	1	H
36	38	2	1	12	0	H
36	31	-5	1	18	0	H
10	18	8	92	11	0	H
48	35	-13	12	32	0	H
68	46	-22	3	46	0	H
29	30	1	2	10	0	I
56	58	2	1	10	0	I
35	30	-5	4	9	0	I
19	18	-1	1	1	0	I
30	30	0	1	7	0	I
36	34	-2	1	8	0	I
27	31	4	2	12	0	I
34	37	3	1	10	0	I
32	34	2	3	12	0	I
41	44	3	1	14	0	I
25	25	0	1	0	1	J
25	25	0	1	0	1	J
13	15	2	4	7	0	J
33	39	6	1	10	0	J
32	33	1	1	8	0	J

36	37	1	1	7	0	J
38	39	1	1	13	0	J
27	23	-4	1	9	0	J
33	33	0	1	6	0	J
70	55	-15	1	35	0	J

C

Evaluation - Interview Questions

A short introduction about the features of the application and how to work with it.

Interview

1. If you look at the issue created. What are your thoughts of this way of working with the maintenance of an SRS?
2. In what situations can you see a problem with the solution?
3. Would you prefer to create an issue to a requirement through desktop (keyboard and mouse, SystemWeaver) or through the application. Elaborate?
4. What was bad about the experience?
5. What was good about the experience?
6. What is your general impression of the product?