

Streamlining a recruitment process

SW1, Group 2

Supervisor: Henning Olesen

Number of Characters: 145.645

Number of Pages: 59



University Of Aalborg
Denmark
December 16, 2021

Preface

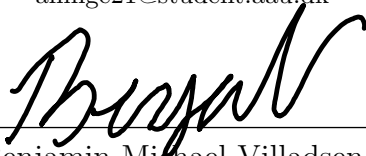
Aalborg University, December 16, 2021



Andreas Hinge
ahinge21@student.aau.dk



Bence Szabo
bszabo21@student.aau.dk



Benjamin Michael Villadsen
bmvi21@student.aau.dk



Goran Kirovski
gkirov21@student.aau.dk



Gustav Lundhus
glundh21@student.aau.dk



Karina Botes
kbotes21@student.aau.dk



Louise Foldøy Steffens
lfst21@student.aau.dk



Abstract

In today's age, the act of job-hunting is more effortless than ever, largely due to the countless technological advancements made, since the general public's introduction to the world-wide-web. A simple internet search, and immediately, tens of thousands of potential work-opportunities are shown on the computer screen. However, the simplicity of our time's job-application process proposes numerous problems in itself. Logically, such a trivial process is exploited by many, creating an ever-increasing competition on the market of human-resources. Recruiting specialists deal with enormous volumes of applicants for a wide variety of positions, with no time to thoroughly assess every candidate. This leads to frustration, stress, mistakes, and ultimately, an inadequate and superficial hiring-process. Fortunately, this issue can be addressed by utilising a computer-program. This project seeks to explore a possible software-solution, by parsing a large number of text files belonging to potential future employees, and only collecting the most valuable data from each text in a separate file, ordered from most to least qualified. Thus, providing a transparent, universally formatted, and compressed brief and ranking of each candidate. To create a working prototype, which has a real world relevance, esteemed feedback from experts within the employment industry, and numeral data collected from a sizable group of job-seeking individuals, are both considered. This helps raise the questions of the efficiency of such solutions and the ideal implementation methods, so that the broadest range of individuals stand to benefit by allowing good candidates to be pinpointed promptly simultaneously with the employer having a trust in the software's capabilities. The design of the proposed solution will be engineered through multiple phases, with implementation and testing will be done concurrently. The entire process will be showcased in depth, with an eventual conclusion on the capabilities of the solution.

Table Of Contents

1	Introduction and motivation	5
2	Methodology	6
2.1	Problem Based Learning	6
2.2	Imperative Programming	7
2.3	Supplementary Insights	10
2.4	Utilisation and justification of the chosen methods	12
3	State-of-the-art	14
3.1	Introduction to the main topic	14
3.2	Manual screening of job applications	14
3.3	Applicant tracking systems - ATS	14
3.4	Software Examples	15
3.4.1	Competitors on the market	16
3.5	Limitations by automation of Short-Listing	17
4	Analysis	18
4.1	Stakeholder Analysis	18
4.2	Salience Model Diagram	19
4.3	Stakeholders	19
4.3.1	Within The Project	19
4.3.2	Within The University	20
4.3.3	Outside The University	20
4.4	Interviews	20
4.4.1	Questions and answers	21
4.4.2	Sub Conclusion of the Interviews	25
4.5	Quantitative Survey	27
4.5.1	Sub Conclusion of the Quantitative survey	30
4.6	Interviews in comparison with the Quantitative survey	30
4.7	Persona	31
4.8	Requirement specifications	32
4.8.1	Must Have	33
4.8.2	Should Have	33
4.8.3	Could Have	35
4.8.4	Won't Have	36
5	The product: A program that solves a problem	38
5.1	Design	38
5.1.1	Diagrams	39
5.2	Phases during product creation	42
5.2.1	Achievability phase	42
5.2.2	Essentials phase	43
5.2.3	Post-interview phase	44
5.3	How does the software work?	44
5.4	Implementation	47
5.4.1	Pre-processing units	47
5.4.2	Organising data	47
5.4.3	Main function	48
5.4.4	makeFileKeywords	49

5.4.5	makeFileWithLineBreaks	50
5.4.6	contactFinder	51
5.4.7	readFileKeywords	53
5.4.8	contactRating	54
5.4.9	outputInFile	55
5.4.10	validation	58
6	Discussion of the product	59
6.1	The final product compared to requirements	59
6.1.1	MacOS and Windows builds	59
6.1.2	File processing	59
6.1.3	Tests	60
6.2	Future work on the product	60
6.2.1	Other considerations	61
7	Conclusion	63
8	Figures	64
9	References	65

1 Introduction and motivation

This semester project is based on the subject "A program that solves a problem". This means that a product is to be developed concurrently with a report, which solves a problem written in the C-programming language. The deadline for the project is the 17th of December 2021, giving a total project duration of 2 months.

In this case, it was decided to focus on the topic of job application management and professional recruitment. Large companies must process an abundant pile of job applications for each and every position, and this practice can require a massive amount of time and resources, which could be better spent elsewhere. The time consumption aspect of the complex recruitment process is a highly relevant issue in our modern capitalistic society, in which multinational companies constantly compete to get the most suited employees, and thus gain considerable advantages over their competitors. In the recent years, by virtue of digital solutions, it has become commonplace to advertise and apply for a job using the internet. Popular job posts can result in innumerable relevant applications and candidates, which the hiring company would have to manually look through and choose from. Unfortunately, in a considerable amount of cases, many of the applications may not be well suited for the job, since it has become incredibly easy for job seekers to simply apply for a lot of jobs in the hope of getting accepted to any one of their options. Even in these cases however, the application still has to be screened, taking valuable time and resources from the company involved.

Based on the group's research and personal interviews, the scope of the problem was subsequently further reduced to the handling of cover-letters during the recruitment process, rather than looking into creating a CV-sorting solution. With the help of professional insight, it was clear, that rather than taking the responsibility for a central part of a recruitment process, it is generally much more appreciated to automatise a branch of the work, which is widely overlooked due to a lack of time. With focus on the topic of job application management as a part of a recruitment process, the problem statement is defined as the following:

Using a software solution, how can the hiring process be streamlined for companies with many applicants, thereby reducing their resource consumption on identifying optimal candidates for jobs with specific requirements? Furthermore, how can the same software solution provide the recruitment specialists a more thorough picture of the applicant, summarising the most crucial information from a written document?

Throughout the project, in-depth research, a considerable number of personal interviews, data analysis, and coding is performed with the intention of being able to come forward with a satisfactory answer and a functional prototype for the software solution for the initial problem statement. In order to conduct research, which depicts the problem field as authentically as possible, a considerable amount of professional first-hand information and numeric data is gathered on today's recruitment solutions and job application management, in addition to the knowledge gained from fundamental literature. Finally, the empirical data collected is used to design a computer program, that can provide a stable prototype of a software solution for a real-life problem, encapsulated by the formerly cited problem statement.

2 Methodology

The purpose of this section is to introduce the specific selection of methods, that was used throughout the project in order to create a solution for pertaining to the problem statement. The selection of these methods have therefore greatly contributed to the ability to streamline the recruitment process for companies with many applicants, and allow for future/hypothetical clients to implement a recruitment system that is significantly less resource intensive than their current one. These methods were both relevant in terms of problem identification, delineation, and eventual solution. More specifically, this section will primarily look at general planning methods in problem-based learning, preferred approaches in imperative programming, and overall introduction to the main methods, that helped answer the problem statement.

2.1 Problem Based Learning

The structure of the report is largely based on the concept of Harboe's mixed methods [1]. The work process was structured according to a mixed method. It begins with an exploratory qualitative knowledge search along with various qualitative studies, such as interviews with experts and specialists. This is followed by quantitative research, i.e. questionnaires, which can be analysed with graphs. In essence, qualitative research takes the role of pilot-research throughout the preliminary investigation, and the research builds upon the fundamentals laid by it.

The first qualitative method used was systematic information search. Here, thesaurus and Boolean operators were utilised for more controlled results. Unfortunately, due to the niche nature of the problem area, the group could not use smaller databases, such as the AAU Library. Instead, the group's focus was placed on the information available on Google Scholar, which became a central database for the sources of the task. Following Aarhus University's recommendation [20], all members of the group were especially careful when selecting sources located on Google Scholar, always double-checking the authors' credibility, and the publishing date of the paper/book in question.

Another key aspect of information gathering throughout the project was formal interviews with recruitment professionals. In these interviews, the conversations were kept open, so that the individuals had the opportunity to freely express what they thought was important about the issue. The following discussions, however, always ran along a fixed wire-frame, so that there were specific questions to be asked each time, meaning that the diverse responses could be compared and trends could be extracted from the dialogues. The results of these surveys were crucial for the program's requirements and design.

The first quantitative survey conducted was a questionnaire targeted at job-seekers. As mentioned, the concept of this questionnaire is strongly linked to the interviews mentioned earlier. In this case, it was chosen to ask job-seeking youth about their perceptions of a good CV/application, and thereby used the collected data to perform a comparative analysis with the expectations of recruitment professionals. In addition, a segment of the responses were used to confirm some preconceptions and hypothesis, that had been formed during earlier work. An example would be

the response times for job searches being too long, and that people ordinarily accept the first (or one of the first) job opportunities they are presented with.

The project was planned using the back-casting method [23], and visualised with a Gantt chart [23], which had been continuously updated until the final deadline was met. Due to the relatively short nature of the project, it was expected that the software solution would resemble more of a prototype, than a fully completed and polished product.

2.2 Imperative Programming

To program and implement a product, that can address the problems presented by the formal research and the surveys/interviews, an approach called the "Software Development Life Cycle" [22] was used. From now on, it will be referred to by its acronym, SDLC. Briefly, SDLC provides an opportunity to "provide a systematic framework to design, develop and deliver software applications, from beginning to end" [22]. This method divides the process up into the following seven phases [22]:

1. Formation
2. Requirement/Planning
3. Design
4. Development/Construct
5. Testing
6. Release
7. Maintenance

The initial formation phase of product development is where the idea is originally conceived, either from an existing software solution or by creating one's own software [22]. The requirement/planning phase mainly focuses on gathering information and requirements through, for instance surveys/interviews or directly from the potential user, to have the resources to formulate a design plan. Additionally, a Gantt-timeline is initialised during this phase, to create an overview over the development process of the program. The requirements and specifications for the product are formulated here, in accordance with the results of the qualitative research. The design phase is where all the knowledge from the analysis of the problem, and the requirement specifications are put together to get an overview of how the software should run [22]. One can use different diagrams and flowcharts to visualise the design of the software.

The development phase is essentially about building prototypes, performing code reviews, and making improvements, executing bug fixes and implementing optimisations to the eventual software solution [22]. During this phase, the requirement specifications take a central role, as the significance and complexity of the implementation defines the process of the prototype-development. The program could be running perfectly on certain applications, but not every one of them by some unknown reason. This is why a testing phase exists. It is to make sure the program is working as intended at all times by catching bugs, defects, errors etc. so a quality solution and product is produced in the end [22]. The release phase is a tad bit

unconventional in this case, since a full commercial product is not ultimately being developed. Therefore, the release phase is now interpreted as the deadline for turning in the report. By this time, the program should have gone through a thorough testing process, and should be able to run smoothly without any critical bugs. The maintenance phase is ignored in this case, because it is unlikely that this software solution will be worked on further after submission. If continued work were to be done, then it is in this phase that one would make sure the program is up to date and meets the adequate standards.

The phases are visualised in figure 1. It is important to see, that throughout the process from phase 3 to phase 6, documentation is present. Documentation is to help individuals understand the functionalities of the program, and therefore easily modify it in all coding phases. The training aspect of the figure is more referred to software solutions with some or only AI/machine-learning applications. The program has no need for training in this development cycle, and therefore is irrelevant in this case.

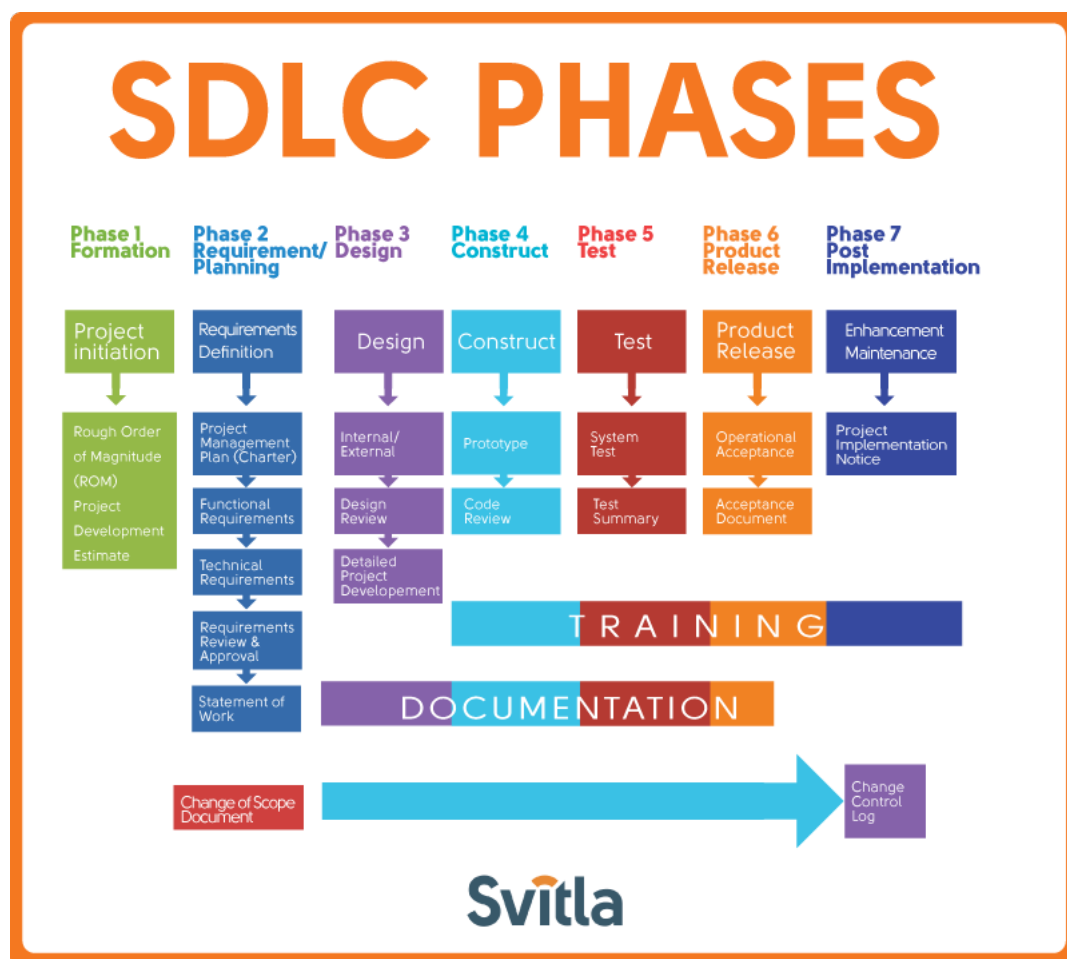


Figure 1: The phases of SDLC [22].

The approach of SDLC [22] contains plenty of peculiar methods of getting through each of the phases mentioned above in an effective way to enable the user to define a satisfactory solution to their problem. In this project, implication of the Iterative Incremental Model [25] seen in figure 2 was used. Resulting in the requirements being divided into different priorities; Must Have, Should Have, Could Have, and

Won't Have [17]. With regard to the priority and complexity of the particular requirements, the assignment has been divided into individual phases, and taken one requirement at a time and designed, coded, and implemented a prototype in the version-controlled master-file, before starting over with the next requirement on the list. These steps are to be repeated for each requirement until the a completed product phase is reached. The strengths of using this method is that one can prioritise requirements (using the MoSCoW method [17]), and achieve faster product development (which is crucial due to the limited time). This also allows for the ability to dynamically change the priority and specifications of the requirements in the meantime. Thus, also creating a functional program without critical bugs and defects - of which one is aware of. To employ this model with as few weaknesses as possible, efficiency in the planning of the iterations was vital, and helped with a clear design plan to include every must-have requirement throughout the iterations of the product.

This model fitted the program best, because some of the requirements were known beforehand, and could easily be changed or prioritised. New results were acquired continuously from the surveys/interviews in our development/construction phase to update the requirements throughout. The model was also a quick way to obtain a functional running piece of software in a short amount of time, which was one of the guiding principles for this project [25].

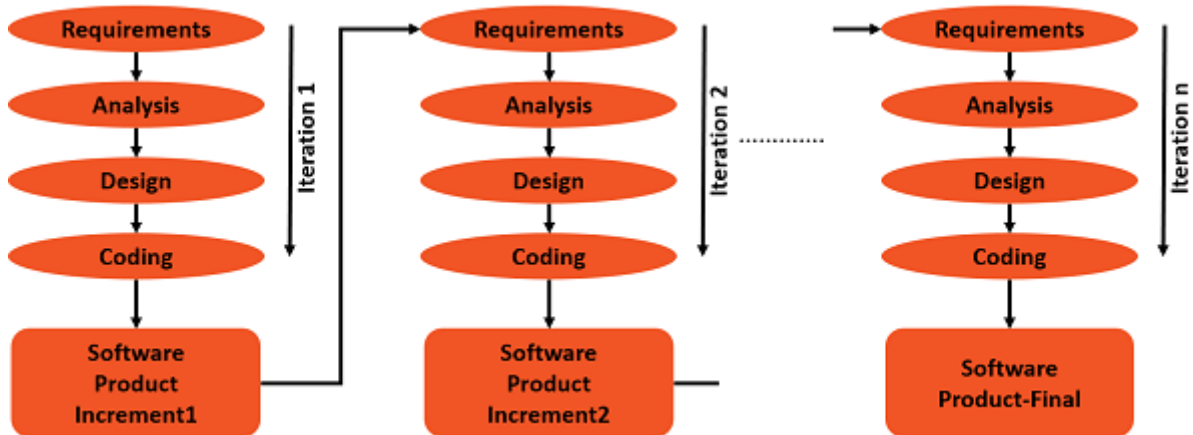


Figure 2: The Iterative Incremental Model [25].

While constructing the program itself, the programming style of top-down programming was utilised, also known as step-wise refinement. In the book, "Problem Solving and Program Design in C" written by Jeri R. Hanly and Elliot B. Koffman [7], this particular software development technique is described as the following; "A problem-solving method in which you first break a problem up into its major sub-problems and then solve the sub-problems to derive the solution to the original problem" [7]. The technique was adopted by identifying all underlying problems of the main solution, and often even dividing these sub-problems into smaller bits, then solving them individually [7]. The particular sub-problems could for example be importing text data from a text file, or creating a simple algorithm, that can find predetermined words in any given text file. Once a solution was found, a specific function was created for it. By constructing the solution this way, the individual members of the group got the chance to work in parallel groups on different seg-

mented sub-problems. This method was used until all of the requirements were met, again using the Iterative Incremental Model [25]. This resulted in a relatively clean master file and main function, because the final solution simply consisted of individual function calls. Furthermore, it ensured the readability of the program, assuring the possibility of future improvements and patches. Top-down programming design can be visualised by using a structure chart [7], where it starts with the original problem at the chart's level 0. At level 1, it defines the biggest sub-problems without going into details. Level 2 is where the sub-problems get more detailed. A structure chart can have as many levels with detailed sub-problems as needed to solve the original problem. By using the chart in this project, the group gained a satisfactory overview of the sub-problems at every level and one could easily add modified or new sub-problems after each iteration of the Iterative Incremental Model [25]. An example of a Structure Chart is in figure 3, where it is based on the function `contactFinder` from the code.



Figure 3: Structure Chart for function `contactFinder`.

With regard to current industry standards, camel-case has been used for variable name assignment [2]. In short, camel-case is the practice of writing phrases without spaces or punctuation, indicating the separation of words with a single capitalised letter, and the first word starting with either case. Common examples include "iPhone" and "eBay". An example of camel-case used for the project is `deletePunctuation`.

2.3 Supplementary Insights

In this subsection, the relevant methods beyond the scope of the first semester AAU courses will be detailed.

In order to make the preparation of the program development phase as manageable as possible, a variety of diagrams and charts were used. The first diagram adapted was a simple flow diagram [10]. Flow diagrams represent the workload and process of complex tasks. It is a step by step overview of the program in various boxes, which are connected to each other by arrows to make these complex tasks more clear and

understandable. This gives an approximate idea about how the program should be running. It also visualises the kinds of input and output parameters to be worked with and implemented in each function [10]. This diagram can be viewed in figure 13.

Flowcharts also represent the process of a problem and how to plan a process to solve a problem [10]. When working with a flow diagram, focus was placed on how to solve the problem by using functions, and how to implement them in a certain order. The flowchart presents how ideas were arrived at, and which functions should be created and added to the flow diagram and final solution. In other words, the flowchart helped to evaluate and select the best ideas for a possible solution. This method was applied to every requirement and idea to make the process faster and more effective [10]. One of the flowcharts, that were used for this software, is shown in figure 15, and is based on the keywordFinder function.

The next step in the planning was to prepare a UML Sequence Diagram [15], where the passage of time can be seen on the y-axis, and the functions (normally classes) are distributed along the x-axis. The sequence diagram gives a better overview of the functions in the program, and is a useful tool for planning how the code should be structured. On the sequence diagram it is clearly shown which functions are called and in what temporal order, during the entire run-time of the program. The UML diagram for the program, that the group made, is at figure 14. Only a selected number of functions were chosen to be visualised through the sequence diagram, based on their importance in visualising the inner workings of the finished prototype of the program.

The last tool used in the preparation of program development was pseudo-code [12]. Pseudo-code is a way to set up a problem by writing in a syntax-free language, and although it is not a requirement, pseudo-code more often than not has some constructs to make it more readable. Using pseudo-code is a technique to write distinct steps in a complex algorithm or problem that is easy to understand for everyone, not just programmers. Pseudo-code was used to achieve better readability for specific code-snippets, and a better understanding of how an idea is implemented in code [12]. It also served as a middle-round between the flow diagram and writing actual C-code. This eased the process of converting the functions from the flow diagram in figure 13 to code by having this intermediary between the code and flow diagram.

In the analysis part of the report in figure 17, figure 16 and figure (real code) is there shown how to go from Flowchart to pseudo-code to real code in a text editor.

Schema.org [18] was utilised when researching the properties of a person, because it provided suitable ideas and frameworks for extracting information about an applicant. Other than that, the website also specifies the expected data-type of these variables, which also gave the group a rough idea of how these variables should be extracted and manipulated in the program [18].

Lastly, the MoSCoW Method [17] was used to organise the product's requirement specifications. MoSCoW stands for Must Have, Should Have, Could Have and Won't Have, and it is a prioritisation technique for managing requirements. Before starting a MoSCoW-analysis, it is essential that the product team is aligned on what objectives are included, and should have at least a superficial understanding of all

starting elements of the requirements. In this project, all potential requirements are formulated in accordance with the preliminary research [17].

The "Must Have" section includes "non-negotiable product needs that are mandatory for the team" [17]. The initiatives and features included here must be asked the question; "Will the product work without it?". If the the product team's collective answer is indecisive or the answers are controversial, either further discussion is needed, or the requirement should be downgraded to the "Should Have" section [17].

The "Should Have" section contains the "important initiatives that are not vital, but add a significant value" [17]. The program will still function without the "Should Have" initiatives. They are significant for the program but not vital, and they are often performance enhancing routines [17].

Next comes the "Could Have" section. It is the "nice to have initiatives that will have a small impact if left out" [17]. "Could Have" initiatives are nice-to-have functions, but unlike "Should Have", they do not represent huge step-back if they are left out of the program. If the "Must Have" and "Should Have" sections end up with being larger than expected, then "Could Have" is the first section to be deprioritised [17].

The last section in the method is "Won't Have". In this section contains the "initiatives that are not a priority for this specific time frame" [17]. Placing initiatives and features in this sections allows the group to manage the expectations for this project. Given the time frame, the groups knew what were priorities and what initiatives is maybe for the future or not at all [17].

The MoSCoW-method can be applied for more than just one limit such as time [17]. The method can for instance also be used for budgetary constrains. The "Must Have" and "Should Have" is prioritised and later can the "Could Have" and "Won't Have" be programmed if the budget allows.

Another factor is the skill sets that the programmers have. This way can the initiatives be prioritised based on what is possible or not. The group based some of the initiatives in the "Won't Have" section in the analysis by this approach. As well as the most used approach, that is deadline based, which can be seen in the "Could Have" section [17].

There is few disadvantages for using the MoSCoW method [17]. Firstly, initiatives could be placed in a wrong category by incoherent evaluation, or the group could have biases considering some initiatives, which also could lead to false placement in the sections. Other than that, in a real work-environment, the software-development team is expected to consult with other branches to ensure that their prioritisation is acceptable [17].

2.4 Utilisation and justification of the chosen methods

This subsection is a discussion of the chosen methods and their use. Harboe's mixed-methods [1] were chosen, because of the need for continuous information search, and thesaurus and Boolean operators for collecting the correct data/information. Interviews with recruitment specialists, made the specification, design, and organisation of program-development a lot simpler. By having these interviews, time aimed towards re-designing the product and testing was minimised, compared to simply making assumptions within the group, and only using online sources.

The second method in Harboe's mix methods [1] is quantitative research. It was chosen to create a survey to gather information about job-seekers. Amongst other things, it benefited the group by providing the data for a comparative analysis of job-seekers perceptions of a good application and the expectations of the recruitment specialists. Lastly, the method of back-casting [23] was also a crucial element in project-organisation. This method gave an overview over the project planning and timeline over the expected deadlines. The method's success shows in how most of the deadlines were met, and the desired solution was completed in time.

For imperative programming methods; SDLC [22], Iterative Increment Model [25] and camel-case [8] were used. It is difficult to make a program to solve a given problem without a structure or a process. That is why SDLC [22] is chosen to be applied. This method benefited us with a common insight into the amount of work to be put into each phase with respect to the deadline, and what each phase should contain. The Iterative Increment Model [25] was chosen, because it made most sense with the kind of program, being created. New information and requirements are continually gathered throughout the project. That is why a prototype is logical for each new requirement via the model. Camel-case [8] was chosen for two simple reasons; it is an industry standard, and contributes to the readability of the code.

The third and last subsection in methodology is supplementary insights. Flow-chart and flow-diagrams [10] were chosen to be implemented, because their ability to simplify the steps of the process of ideas (flowchart) and implementation of functions (flow-diagram). By simplifying specific routines, more complex ideas and functions were quicker to be understood by everyone in the group and therefore resulted in a more streamlined work-environment. Pseudo-code [12] had the same effect on grasping functions as the flow-diagram. It was used to explain convoluted functions more directly, and made it more manageable to apply the functions in real code. The inclusion of UML-Sequence diagrams [15] was chosen to play the role of a planning tool for the program. It was beneficial for us, because it gave us a much less convoluted overview of the run-time of the program, depicting only the relevant function-calls and their return values. Schema.org [18] was utilised because it gave the group an insight of which properties, and what type of data was going to be worked with. Furthermore, the MoSCoW-method [17] was chosen, because the functionalities of the program had to be planned in a systematic manner, and sorted in accordance with their importance. This was especially paramount due to the short nature of this project.

3 State-of-the-art

3.1 Introduction to the main topic

The following section consists of a presentation of the problem area in the domain of the topic concerning the process of recruitment in relatively large companies. This is a problem field, in which a considerable amount of research and problem solving was done, and a number of companies have come forward with a plethora of solutions.

Whenever a company is in need of new applicants for a job, it needs to overcome the process of hiring - a process usually requiring several hours. According to Andrew Fennel, formal recruitment specialist and founder of Standout CV [3], the average time to review one applicant approximately takes up ten seconds, excluding the time taken to open and close each application document which - if accounted for - results in about a minute [4]. The amount of applications a company receives from various applicants, naturally varies, for instance, Julie Juvera head of HR at Texas Roadhouse Inc., claims to receive 400 job applications within 24 hours of posting a job offer. In addition, Starbucks had 6.7 million job applications during the year 2012, regarding 65 thousand jobs [6] [29]. On the basis of the 400 applicants Texas Roadhouse received, the product of the time used per applicant as well as the amount of resumes received, results in roughly 6.5 hours. This is almost equivalent to a whole day's work, spent on going through resumes.

3.2 Manual screening of job applications

When looking through job resumes it has come to light that many employers are not reading the full resume, but simply skim it with the "Ctrl + F" search command [19]. The command is used for allocating keywords relevant for the job such as: certain skills related to the profession, social media accounts, ability of articulation, previous workplaces, etc. Information and other aspects that are not considered particularly crucial and therefore not being read and noted very often, according to Ulrich Schild, a Senior HR BP at Chartered Accountants Australia and New Zealand, are mostly fancy formatting and cover letters. Limitations by going through job applications manually, are the fact that the task will become very monotonous as the amount of documents increases, resulting in some resumes being overlooked or not evaluated with the same dedication as others.

Several companies have published their take on improving the mentioned situation by automating the screening via software, and processing the received resumes and/or C.V.s creating a shortlist for employers to read [30]. A shortlist is a list consisting of the best candidates found during the screening of resumes. This spares them from the tedious process of manually going through hundreds of documents, making their minds more suitable and clear for choosing the correct applicants for the job afterwards. Furthermore, the mentioned automation might create a greater sense of satisfaction for the applicants themselves, as it could result in reduced time before receiving a reply.

3.3 Applicant tracking systems - ATS

More advanced programs take on machine learning when approving suitable job candidates, thus using NLP - Natural Language Processing technology, which involves

the machine being able to understand human language in context and automatically conduct summaries of applications and split the topics and respective skills of the resume into different sections [9]. These advanced software systems differentiate skills into two different categories; soft skills and hard skills. Soft skills refers to broad spectrum skills, not necessarily being specifically required in the job description. Examples of soft skills would be being disciplinary, good at teamwork, empathetic, etc. Hard skills, on the other hand, means the specific skills matching the ones written to be required in the given job posting. Thus, examples of hard skills could be the ability of programming in a certain language, being good at solving mathematical problems or writing good articles.

The term for software used by recruiters, doing this exact thing - screening and rating various applicants' resumes to determine whether or not they are fit for a specific job offering - is broadly known as ATS, "Applicant Tracking System" [30]. Usually, an ATS is not solely dependant on parsing the resumes of the candidates, but also on other determining factors, namely questionnaires chosen by the person responsible of conducting the recruitment process. The mentioned questionnaires may be a skill test, profiling of personality, background checks, or other types of information. All of them having a rating in common, which is then returned and used for filtering out the least optimal candidates.

3.4 Software Examples

Vervoe is one of many examples on software that is used for automating the pre-hiring assessment [27]. The software consists of three steps. Firstly, the employer must choose an assessment method from *Vervoe*'s library - or they can create their own. A library is a section of questions that the applicants need to answer to be rated. Secondly, the person hiring needs to invite the potential job candidates via a link. Lastly, the applicants are screened for the job, and the employer, will receive a ranking of the people who have applied. *Vervoe*, like many of the software used to perform the pre-assessment, does this through applicants answering a questionnaire/survey. *Vervoe* is currently used by Walmart, Australia Post, Omnicorp Group and G4S.

Other similar programs are **Zoho Recruit** [24] and **Fresh team** [13]. Zoho Recruit creates a list based on how much a given resume matches the job description. Fresh teams' ATS automatically removes applicants whose resumes has a match rate below 20 percent, but the other approved applicants are, as with Zoho Recruit, sent to the employer accommodated by their rating. Beyond the ATS parts of the software, both companies provide their own user interface, further automating the recruitment process. The interface functions as a type of social media where employers can browse through and contact various applicants.

The biggest ATS software system in the world according to David Mehsulam, founder of JobTestPrep, is **Taleo** [11]. The system requires the applicant to upload their resume, written according to a specific *Taleo* template, as well relevant social media accounts, such as LinkedIn, to the cloud. Thereafter, the CV is, via machine learning, scanned for relevant skills and matching keywords. Thus, the website automatically fills out a form, that the applicant must review and potentially edit, before submitting it to a job listing. Next, the applicant must complete the men-

tioned questionnaire tailored for the specific job, which accounts for the applicant's rating also. For a more thorough description of the potential competitors, read the following section.

3.4.1 Competitors on the market

A more thorough research of the competitors' software solutions has been done in order to look into the current climate of market for software that may be comparable to the product, which concludes this project. Here is a general overview of a few different candidates that provide a service similar to the one currently being developed for the project. Naturally, features that seem common across the board are prioritised for the purposes of comparison.

1. Company: Fresh Team [\[13\]](#)

Description:

This Company offers software which can do the following: manage job postings sourcing candidates, and resume screening.

Features:

Services offered: Recruitment automation How Freshteam can help: The Freshteam Autopilot cuts down as much as 75 percent of everyday recruiter routines and releases time and resources to focus on core missions like proactively building an organisational culture or strategising talent acquisition.

- Reject candidates if their test scores are below 20 percent.
- Send out communication to all stakeholders when a candidate reaches the interview stage.
- Send an online test when the candidate reaches the technical interview stage.
- Advance candidates who have shown interest in working for night shifts.
- Archive all unselected candidates from a specific vendor to the talent pool.
- Send highly personalised emails to all referred candidates.
- Reject candidates who do not have a work-visa in your country.
- Advance candidates who are located within the same city.

Pricing

Free: up to 50 people per business

Growth: 1,2 € / employee and 71€ platform fee / month

Pro: 2,4 € / employee and 119€ platform fee / month

Enterprise: 4,8€ / employee and + 203€ platform fee / month

2. Company: Zoho Recruit [\[24\]](#)

Description:

This Company offers software which can do the following: manage job postings, source candidates, screen resumes.

Features:

Services they offer: Recruitment automation

- Match and associate the right candidates exponentially quicker. Get a list of candidates that are graded and ranked based on how much they match with the job requirements.
- Set must-have and nice-to-have skills for a job opening even before publishing. Zoho recruit's AI engine calculates the applicant's skill score and identify candidates within seconds.

Pricing

Zoho Recruit combines the Applicant Tracking and Candidate Relationship Management

Standard 25€ / month

Professional 50 € / month

Enterprise 75€ / month

3. Company: Oracle Taleo [11]**Description:**

This Company offers a software package **Oracle recruiting** which can do the following: Candidate attraction, candidate engagement, hiring, onboarding, analytics and reporting.

Features:

One of the services they offer: Oracle recruiting

- Match and recommend candidates fast by using AI.
- Improve productivity by leveraging tools to automate job postings, candidate communication, screening, interviews, and offers.
- Learn more about the experience, skills, and aspirations of employees with access to talent data covering all stages of employment.

Pricing

According to Oracle's own global price list, for an average company, the price can be estimated to be around 5000 € / month [14].

3.5 Limitations by automation of Short-Listing

The limitations by companies and their increasing use of different ATS-software are primarily, that some qualified candidates might become sorted out of the initial assessment mistakenly, because of the fact that the software used by recruiters only takes certain specific keywords into consideration [29]. However, many different articles will arise by a Google search, describing how to write job resumes the most optimal way, more specifically teaching how to stand out whenever a company uses keyword-scanning programs for screening their big list of applications, which is used by most big companies, according to Indeed Editorial Team [21]. Online programs, such as *jobscan.co* and *cvscan.uk*, will compare job resumes to job offerings and return match percentage and recommendations for other keywords, that could make the applicants more likely at securing a place on the short-list.

4 Analysis

The main purpose of this section is to work towards defining the exact way in which the group will attempt to answer the problem statement. This includes the interpretation and analysis of empirical data, more in-depth, and further delimited research. At last, the definition of a list of requirement specifications for the product itself are defined, based upon the formerly mentioned empirical data and forthcoming analysis. The bottom line is, this section concludes in a definition of a prioritised set of values and expectations proven by factual data.

The group's preeminent goal is to, via a software solution, help ease the process of analysing and compiling cover-letters in a simple and efficient manner for employers. During the analysis, it is necessary to keep in mind what the group is working towards, as all of the conclusions must build towards an eventual answer to the problem statement. The typical employer processes a multitude of applications for each job-listing, and as such either end up skimming, or generally not reading certain aspects of a candidate's provided information meaning that there is a spot for a reliable product in this market.

Getting an insight into competitor products, and current real world solutions is vital and gives an idea of both what is possible, and generally seen as the golden standard/ideal while also allowing a certain niche to be carved out if found to be an area untouched. Real world free software that filters out in terms of keywords and matches has been evaluated, and compared to the current program in development. Paid programs have not been explored, but it is estimated that the product quality would be higher if it had been chosen to do so.

The group has conducted multiple interviews with numerous experienced employees (typically) situated in the field of human resources at various Danish firms. These interviews have served to create a context for implementing a product that could both be useful, and sought after in the real world. These interviews have established a connection to those that are the target audience for the product, and to drive development towards practical and realistic ideals. A typical interview would establish who the person was, the employee hiring protocols, the weaknesses and strengths of the process, and the desires of a program that helps with the initial sorting of employees.

Questionnaires were sent out in order to get the other side of the coin. This means that it was with a focus on other students and the hiring process that they went through. This could for example deal with the amount of time they waited for an application, the information they would typically sent off to an employer and so on.

4.1 Stakeholder Analysis

It is considered worthwhile to analyse the respective stakeholder in order to create an overview of both their needs, and possibilities for cooperation. The reason that both the internal and external stakeholders are considered is simply because it gives a broad overview of both the people involved in the project, their needs, and how it both reflects and mirrors that of the targeted users of the final product; Namely the HR department of firms to aid in recruiting capabilities. The Salience model

will be utilised to aid in this stakeholder analysis, and can be broken up into three elements: [16]

- Power - Ability to influence the project
- Legitimacy - Justified involvement in the project
- Urgency - Expectation of communication from the project team

4.2 Salience Model Diagram

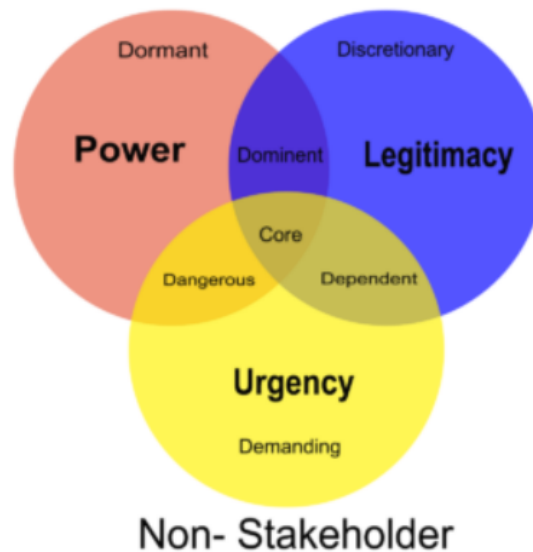


Figure 4: Salience Model Diagram [16].

4.3 Stakeholders

4.3.1 Within The Project

Software Group 2 The groups purpose is to create an intuitive, functional and well designed application sorting software that can be utilised by the non-university based actors. The group is responsible for coordinating and facilitating both the product itself, and the information gathering from the other interested parties. This is in order to take heed of both their requirements, and preferences. This means that group 2 is at the core of the actors. Strength within the group is the amount of people that can be utilised in order to cover a lot of ground both in terms of programming and information gathering at once, whereas a weakness can be seen to be the lack of experience as this is a first semester project and the solution could be perceived as overly ambitious. The group has both power, legitimacy, and internal urgency and is therefore classified as "core".

Henning Olesen Functioning as guidance counselor for the group, and has a vested interest in achieving a great product. Olesen also helps with contacting to the various parties, and helps with any questions and concerns throughout the project process at various meetings. Olesen is remarkably experienced, and has good real world understanding that helps push the process of creating the product in the right

direction. A weakness may be that the expectation can be set high, and perhaps the ambition level reaches beyond that of a first semester project (such as the interviews and more). Henning Olesen would also be classified as "core" given that he holds high influence over the course of the project duration, rightful involvement and expects timely responses from the team.

4.3.2 Within The University

Other Student and Teachers: Throughout the project there will be milestones where other students and teachers will analyse the project and come with constructive feedback and discussion-points for the future. A benefit to including other students is that they genuinely want to help, and enjoy the process of going through another groups project. This also means that they tend to be honest about the pros/cons of the project. This helps with future iterational work. A negative can be that perhaps there is not much valuable feedback to fetch or that it is simply miscellaneous talking-points that were already considered. Here, the teachers would be considered "dominant" stakeholders, as they both have legitimacy, and power over the process of P1 and the outcome of the project. However, there is no urgency throughout the process to keep them updated/satisfied. Other students are discretionary which means justified involvement, but no power or urgency as they primarily function by giving feedback to the project process occasionally.

4.3.3 Outside The University

Interviewees: The interviewees have taken part in personal interviews in order to gather information about the specifications of the program, and the processes that they themselves engage in. This means that since they have personally agreed to participate in these interviews that they both carry an interest in furthering the project, and have useful information to give for future iterations. A downside can possibly be fluff information, or spending a lot of time on information gathering that may lead nowhere productive, especially if they cannot visualise or consider factors that they would want in a program, or if they simply just would not use it. Classified as Dormant. Power to influence the project, but no urgency or actual involvement. More of a passive power.

HR Departments Within Companies: This actor represents the actual intended user of the product. It is important that the product eases the initial part of the recruiting process and addresses this in an intuitive and efficient manner as otherwise doing it manually would be considered better. Strength would be the feedback, and general idea of the product in everyday use whereas a negative might be the indication of the program might depend on the actual user as typically the IT literacy of the average person is not extremely high, especially if dealing with an older person that tends to do things manually. Classified as Dependent. Legitimacy, urgency but no power over the project and could potentially change product if their expectations/service is not up to their standards.

4.4 Interviews

This chapter acts as a walk-through of the qualitative interview, in terms of the questions, people, and conclusions drawn from the interviews.

Throughout the project it was assessed to be profoundly beneficial to gather information from experts in order to create a product better suited to solve the problems people are facing in the real world. With the perspective and information gained, better requirements can be formed and shape the product to be ultimately more useful at solving the problem statement. Unfortunately, a crucial mistake that was made during the arrangement of the interview sessions. The medium of the transcription was not specified in the emails that were sent to the interviewees, and all of the meetings were arranged on a short notice. Recruitment is a sensitive topic with an endless list of many problematic legal and moral aspects, and the specialists interviewed did not allow voice or video recordings without a preliminary consultation with their superiors. Thus, the only way of documentation the group was allowed to employ was to take notes in real-time as the interview progressed. The quotes seen in the following sections are a direct transcription of the meetings, however due to a lack of physical evidence, it must be disclosed that these are notes taken by a member of the group, and were not confirmed by an additional recording. The following people agreed to participate in our interview:

- Beth Rønhoff from AlfaNordic
- Søren Bay Mark from JyskeBank
- Nikita Faber Nielsen from DSB
- Bjørk Jeppesen from Børnehuset Kregme

4.4.1 Questions and answers

Prior to conducting the interviews, it was crucial for the group to discuss how it will be conducted, and what information is to be extracted from the responses. Starting with a brief introduction, the interviewees (members of the group) have to set the scene, and disclose what the interview will be used for, and who they are. Other than making the group's intentions public, it also sets a friendly tone, and highlights that although a set of questions were sent out, it is not a manuscript that must be followed throughout the conversation. Now, that the interviewees have presented themselves, it is the expert's turn to tell a bit about themselves. The information gathered from here is used to verify the relevancy of the person's attendance, and also raises the trustworthiness of the subsequent statements. Next, the interviewee asks about the expert's current recruitment process. With these questions, the interviewee must attempt to recognise some habits, preferences, and get the expert to reflect on the advantages and disadvantages of their system-at-work. During later work, information here will be central in designing a solution, as it should fit in this environment. Additionally, numeral data considering time-management and efficiency can eventually also be extracted here, providing more information for further optimization of the product. At last, the interviewee inquires about the professional's subjective view on what is important when evaluating an application / CV, and what the most important aspects of an application / CV are. The information extracted here is important in deciding what data deserves most of the group's focus, when prioritizing the program's analytical functions / data extraction from text files.

Before the interviews questions were held, the individuals also received the following list of question beforehand, to be able to reflect over them before the conversation.

Here is the list of questions with the associated answers:

1. Introduction

- **1.1 What is your name?**

Rønhoff: Beth Rønhoff

Mark: Søren Bay Mark

Nielsen: Nikita Faber Nielsen

Jeppesen: Bjørk Jeppesen

- **1.2 What is your background, can you tell us a bit about yourself?**

Rønhoff: I have a bachelor's degree in psychology and philosophy, and a master's in international business with a focus on stress within a workplace.

Mark: I have worked in human resources for about 20 years and as leading manager of HR for 7-8 years.

Nielsen: I have a master's degree in human resources from RUK, and have worked with recruitment since 2013 starting with a commercial bureau.

Jeppesen: I began my higher education in the field of pedagogy and finished with a master's degree in psychology

- **1.3 What is your current position as an employee? What does your job entail?**

Rønhoff: I currently work in the HR-field, more specifically stress management, and general prevention of stress. I am also partially responsible for recruiting and finding potential candidates for job openings, and forwarding relevant screenings of job candidates.

Mark: I am one of the partners and leaders of the HR-department of Jyske Bank with around 400 successful recruitment operations each year.

Nielsen: I am currently employed as a recruitment consultant for DSB with a broad portfolio of recruitment-fields.

Jeppesen: I am the pedagogic overseer, and head of the kindergarten, "Børnehuset Kregme", which currently has 30 employees. Other than that, I am also responsible for employee-management and hiring among other things.

2. Recruitment process

- **2.1 On average, how many applications do you receive per job offer?**

Rønhoff: I get approximately 20-30 applications per job-posting.

Mark: I receive 40-100 applications per ordinary job-postings, but commonly less than 10 per special jobs (special jobs meaning jobs with a requirement for specific higher education).

Nielsen: DSB has approximately 7 applicants per listing at the moment, with the predominant category being academic jobs.

Jeppesen: I tend to get 10-30 depending on the job specified within the job posting.

- **2.2 What are the typical steps for you when reading and sorting CVs/job applications? (If not mentioned, ask if they use software and if not, if they would do so)**

Rønhoff: I look at CV first, focusing on buzzwords, and the formatting of the CV.

Mark: I look at CV first, looking for key skills, and additionally at the application after I have found a good match in the CV using ATS software.

Nielsen: I focus on checking experience, length and nature of active years at previous companies, and how often job switches occur.

Jeppesen: I look at CVs first, checking if their home to work road is realistic and if experience is relevant. Afterwards I focus on the application, mainly skim-reading.

- **2.3 What advantages/disadvantages do you experience with your current procedure of application procession?**

Rønhoff: AlfaNordic does not have much focus on the application but only on the CV. There is simply not enough time to read and analyse the application which is frustrating, since it could give much more insight into the personality and competencies of each candidate.

Mark: I do not have time to look at the applications, and generally hire for jobs without requiring one. This is an advantage of our recruitment process having a fill-out form for questions, and contact information.

Nielsen: Time I spend on checking the pointless candidates is frustrating/a time waster. It would take too many resources to fully focus on every application in depth.

Jeppesen: There is not enough focus on the application due to time deadlines, it takes too much time to read through the whole application.

- **2.4 On average, how much time do you spend on each CV/job application?**

Rønhoff: I spend 10 minutes as a maximum on every CV, and very shortly skim the application.

Mark: I spend around 2 min per CV and then sort in 3 piles: rejects, maybes, perfect, and thereafter I would look at the relevant applications.

Nielsen: I usually spend a few minutes on skim-reading the CVs, take a glance at the attached application if possible.

Jeppesen: I skim-read the CVs in the first sorting of candidates. Then more closely read CVs in the second sorting and skim-read the applications attached for any stand out information.

3 Content of application

- **3.1 What information are you primarily looking for in an application?**

Rønhoff: AlfaNordic does not really look at applications. But in the CV,

there is a focus on buzzwords given from the company to look after, formalities of the CV, correct grammar, relevant work experience, and the applicant's ability to sell themselves as a candidate.

Mark: I look for a personality that fits the firm. A good personality makes up for a lack of experience to me. Here, CV is weighted more, because I select only some applications to read based on that.

Nielsen: The application is generally not considered a ton, again, DSB mainly focuses on the experiences and previous work experience of the person listed via the CV.

Jeppesen: I look for use of relevant technical terms (buzzwords given in the job posting), how good the applicants' written language is, address (work to home road is realistic), the seriousness of the candidate.

- **3.2 What would reject an application (or give it a low score), and what are the first red flags?**

Rønhoff: If the applicant "defines" what they think is important for the employer to know and do not give all the relevant information, only give the previous job titles but do not describe what their job actually was, bad grammar, bad setup of CV.

Mark: I do not have any typical red flags, but the perfect CV should maximum fill 2 pages, with a short intro with a bit of a personality and then relevant experience.

Nielsen: For me, red flags would mostly be if a person hops between jobs consistently, and often. Job shoppers are not ideal. Otherwise typical bad grammar, lack of experience, or just generally lackluster effort.

Jeppesen: I think a red flag to me is missing information, bad grammar, bad setup of CV and application, if the application is too long or too short. If their work experience and education on their CV do not add up.

- **3.3 Do you personally feel that the initial sorting can be frustrating at times or that it could be more efficient?**

Rønhoff: I think it is very frustrating that they do not get time to analyze the application, and if a program was made that would optimise that issue, it would definitely be helpful.

Mark: I think it is frustrating that there is a number of people who only apply for the job, because they have to get social benefits. My team and I sort them out on the first read of the applicant, and place them in the unqualified pile. It would be effective to disqualify those applicants from the start to save some time.

Nielsen: To me it could definitely be more efficient, especially the initial bulk of candidates where a lot of the reading is time wasted as many need to get sorted away from the more "serious" pile.

Jeppesen: It is frustrating that there is not enough time to fully analyse application for buzzwords and the context of those buzzwords.

- **3.4 Do you think that a software solution could help your current**

recruitment process, more specifically application screening?

3.5 (If yes) What would the ideal CV sorting program look like for you?

Rønhoff: AlfaNordic and Niras already uses a "recruiting-processing" software, but it could be made so much better if these requirements were included: Input your own keywords and not only use the already given ones, additionally comparing the keywords with the CV and the application. Searching for synonyms for the given buzzword would also be ideal.

Mark: Jyske Bank is already using a software solution, where they use to handle the CVs not read or screen them. I would want a solution that reads the CVs and compares them to each other. I also wish to have a dashboard where the information from the CVs: years of experience, skills, keywords and personality test is displayed.

Nielsen: Most interesting for DSB would be a percentage match, which in turn would allow a sorting of the "top" x candidates, in order to remove the floor of candidates that do not meet a certain threshold in order to save a ton of time, and allow more in depth analysis of the serious candidates. DSB prefers qualification focus over keywords as a whole.

Jeppesen: In Halsnæs municipality every company is required to use a given software to keep track of job posting and those who seek jobs, but that does not include an application or CV-screening service. A service like that should contain these requirements: Input own keywords and search applications and show the sentence where keywords are used. Count how many times the keywords are used and if they even are used. Compare the use of the keywords across all the given CVs.

- **3.6 Who is responsible for deciding when to introduce new screening protocols and potential software solutions?**

Rønhoff: We have an IT-department here (at AlfaNordic), who manage these kinds of things, so I do not really know about the process that goes into implementing new software. I personally do not propose new programs either, but my higher-ups are always on the lookout for smart digital solutions.

Mark: I have a long history of requesting new software from our IT-department here at Jyske Bank. Unfortunately, it is a very demotivating process, since getting a green-light for local-implementation can take up to 2-3 years, and in most cases, the propositions fall through.

Nielsen: The people from the IT-department are mostly responsible for software-related work, but my higher-ups are often the ones who want to introduce us to new solutions.

Jeppesen: I am sorry, but I do not think I can give a good answer to that question. I welcome new software, but I personally try to focus to get the most out of my own field of work.

4.4.2 Sub Conclusion of the Interviews

The knowledge gathered and conversations had throughout the interviews has provided a clearer picture of what is needed and wanted in a job application screening software. In particular, the specific requirements' priority the interviewees have

presented gives a good insight into what work needs to be completed up until the deadline to deliver a satisfactory program.

It can be seen across the interviews that a job application does not receive all that much exposure compared to a CV. One of the key reasons for this seems to be the time-consuming nature of analysing applications, and their lack of standardised structure. The points raised can for instance be seen in Beth Rønhoff's, Bjørk Jeppesen's and Søren Bay Mark's interview answers to question 2.3. Mark has answered to question 2.3, that it's mainly time spent checking the pointless candidates and its frustrating/a waste of time. Based on the interviewees' answers to this question, it can be confirmed that a product specification requirement could be something along the line of "input keywords" requirement should be implemented in our program. A requirement like this would help the user more quickly find some more specific information in the application, so the application does not "go to waste" but does not take so much time up to read and analyse either.

To question 3.4 and 3.5, Beth Rønhoff from AlfaNordic says the "perfect software solution" would both compare the users keywords input across the CV and the application. Søren Bay Mark from Jyske Bank wishes a software that compares multiple CV's with regard to specific keywords. Nikita Nielsen from DSB wishes a software would sort the candidates based on the top percentages of matches with respect to unique buzzwords. And Bjørk Jeppesen from Børnehuset Kregme wishes a software would compare but the CVs' use of keywords across multiple CVs, but also count how many times the a unique keyword is used.

4.5 Quantitative Survey

This research is an extension to the interviews, yielding a possibility for a comparative analysis of the specialists' expectations and the job-seekers' expectations as for what a CV/motivational letter should contain. Other than that, a couple of these questions contribute to the project by confirming some prior hypotheses and expectations. During its 3 week run-time, the quantitative survey has had 25 anonymous participants, all of them being university students between the ages 19 and 27. Naturally, the group's conclusions from these results should be taken with a grain of salt, as with such a small and non-diverse sample-size, the answers' overall ability of representation is severely weakened.

1. Have you written a CV/job application before?

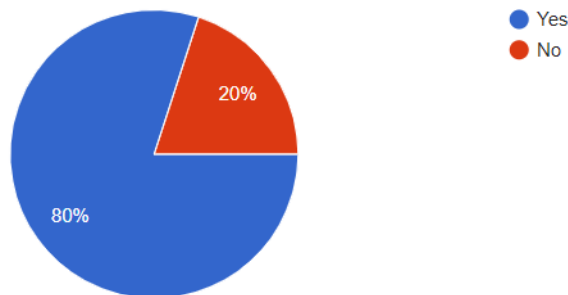


Figure 5: Response to Question 1 in the survey.

The first question ensures that the participants' answers are relevant. Figure 5 above, shows that 80% of the participants in our survey has written a CV / job application before. This should be kept in mind looking through the rest of the survey answers.

2. Have you applied for several jobs at the same time?

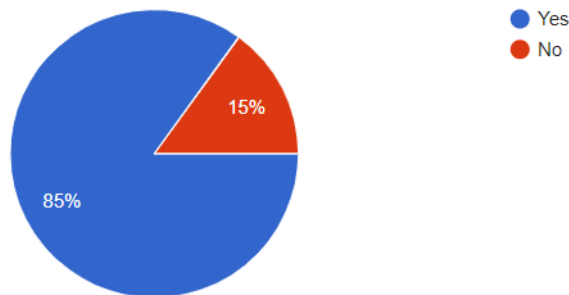


Figure 6: Response to Question 2 in the survey.

85% has responded with; Yes, they have applied for several jobs at once, as can be seen on Figure 6 above. This question helps find evidence that the faster an answer is delivered, the higher the chance that the job position is accepted.

2.1 If yes, did you take the job you were offered first?

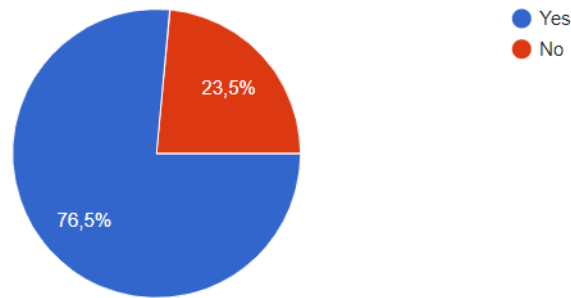


Figure 7: Response to Question 2.1 in the survey.

As seen in Figure 7, 76.5% has answered; Yes, they took the job that was first offered. This shows alongside question 2 that the quicker the hiring companies respond, the higher the chance that the job candidate will accept. So the response time can be a determining factor if multiple companies wants to hire the candidate.

3. Which of the following is the most relevant general information to include in a job application? Please tick more than one option.

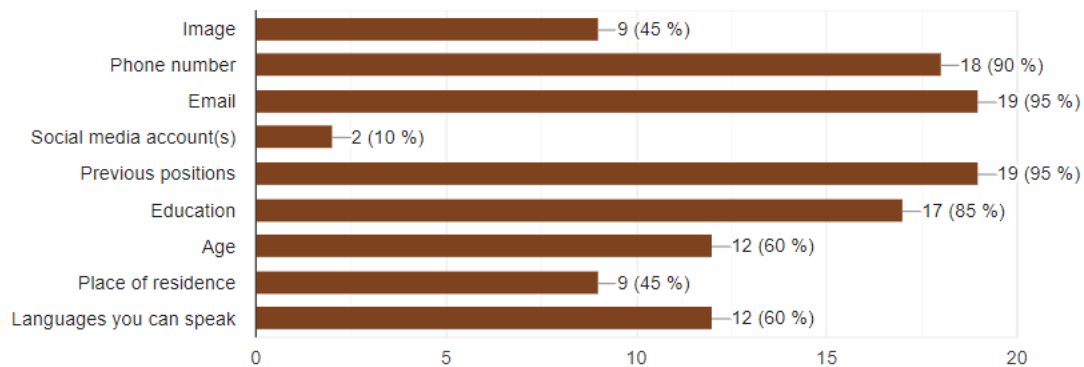


Figure 8: Response to Question 3 in the survey.

This question helps collect data for a subsequent comparison of what employers think are key elements of a CV and what job-seekers find important. Here, the focus lays on general information, such as contact info, location, and work experience. On the Figure 8 above, can the answers be seen. At the top, marked as the most relevant general information, is both "Email" and "Previous positions" at 95 %. Following marked as the second most relevant general information is "Phone number" at 90% and afterwards "Education" at 85% these are according to our survey participants' the most relevant information's to include in a job application.

4. Which of the following is the most relevant personal information to include in a job application? Please tick more than one option.

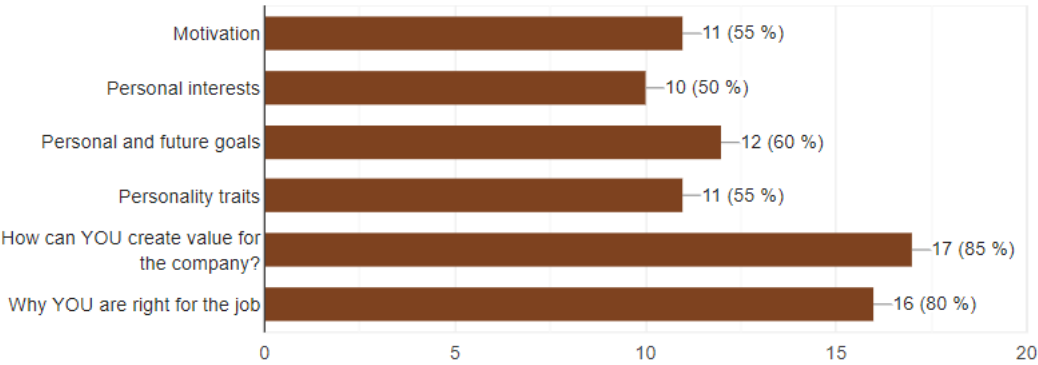


Figure 9: Response to Question 4 in the survey.

This question helps collect data for a subsequent comparison of what employers think are key elements of a CV and what job-seekers find important. Here personal information is in the spotlight, meaning data such as contact info, location, and work experience. On the Figure 9 above, can the answers be seen. At the top, marked as the most relevant personal information is "How can YOU create value for the company" at 85%. As the second most important is "Why YOU are right for the job" at 80%. It is worth noticing that the lowest rated option is at 50%, so according to the survey participants' all of the options are seemingly important since all of the options are rated mostly equal. The options at 85% and 80% will just be the focus from this question.

5. On average, approximately how long did it take from sending your application to receiving a response?

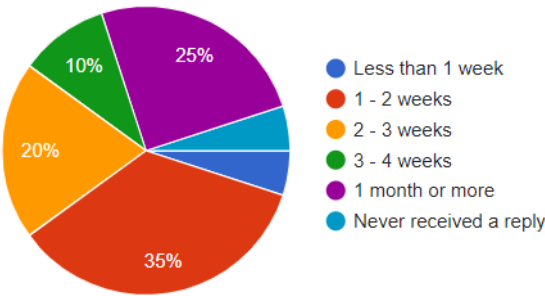


Figure 10: Response to Question 5 in the survey.

This question helps provide an educated estimate on average wait time for job seekers. 35 % which in this case is the majority of the survey participants experience it approximately takes "1-2 weeks" from sending application to receiving response. The second most average response time experienced is it takes "1 month or more" with 25%. A smaller percentage of the survey participants experience they never receive a reply.

4.5.1 Sub Conclusion of the Quantitative survey

From the Quantitative survey, information and insight from the job seekers perspective has been gathered.

From question 5. **On average, approximately how long did it take from sending your application to receiving a response?** It can be concluded that the response time for job applications can vary a lot, 35% experience it only takes "1-2 weeks" while 20% experience it can take "1 month or more". This can be due to the hiring company having to read a bunch of CVs, sorting them etc. This process could be optimised and take less time.

From the questions 2. **Have you applied for several jobs at the same time? And 2.1 If yes, did you take the job you were offered first?** it can be concluded that the reason the majority of the survey participants apply for several jobs at once, and afterwards saying yes to the first one offered, could be due to the long waiting and response time from the hiring companies perspective. When so many of the participants experience "1 month or more" for response, it makes sense to apply for several jobs in hope to get hired faster.

4.6 Interviews in comparison with the Quantitative survey

This section compares the results from the [Interviews-section](#) and the [Quantitative Survey-section](#). These comparisons will contribute to some prior hypotheses and product specification requirements.

The interviews has provided a perspective from the hiring companies and the Quantitative survey has provided a perspective from the job seekers. The two perspectives can now be compared.

In perspective of relevant information to include in a job application, this can be gathered from the Quantitative survey question 3: The survey participants have marked they believe the most relevant information to include is "Email" at 95% marks, "Previous positions" at 95% marks, "Phone number" at 90% marks and "Education" at 85% marks. In the Interview section; Answers for question 3 What information are you primarily looking for in an application. The Interviewees has stated they among other things primarily look for "work experience" - Beth Rønhoff from AlfaNordic, Søren Bay Mark from Jyske Bank mainly focuses on the "experiences and general work trends". A red flag in question 3.2 given by Bjørk Jeppesen from Børnehuset Kregme states: "If their work experience and education doesn't add up". These statements from the interviewees in comparison with the Quantitative survey's answers adds up, in general there is a some what agreement from both perspective that previous positions/ work experience and education are some of the most relevant information to include in a job application.

In the Quantitative survey question it could be relevant taking a closer look the high marks on "Phone number" and "Email". The interviewees didn't speak of looking after a phone number or email, simply because it is a obvious MUST and required when applying for a job. Based on these factors, it should be heavily considered that this projects program could extract these information, making them easily accessible and quicker to find.

From the Quantitative survey there can be seen a varied experience on the response time, 20% experience "1 month or longer". And despite that out interviewees state

in question 3.4 that there isn't enough time for them to read the job application. Both of these issues benefits our prior hypothesis about a software solution could benefit these situations.

4.7 Persona

The target audience of the software solution is rather narrow, and therefore it is sub-optimal to allocate further resources into conducting a large-scale study of this group of people. Instead, a persona is created to provide a generalised picture of the ideal client. A persona is a fictional character with a relevant career background and personality based upon the research conducted throughout the project. The purpose of constructing this persona is to create a character, which serves as a guide for choosing the optimal participants for product testing and future interviews. Additionally, it is considered worthwhile to keep the constructed persona in mind while defining the requirement specifications. The group's persona, Peter, was predominantly inspired by the literature reading during research spliced together with the personality, career, and habits of the consulted recruitment professionals. In a way, this persona could be described as a general characterisation of all professionals, who are confronted daily with the problems the product is attempting to address.

Meet Peter Johansen. Peter is a 38 year old male from Silkeborg, Denmark. Peter studied psychology, and received his cand.mag at Copenhagen's University. He boasts a large professional portfolio, having worked as a HR-consultant at Sparekassen Sjælland and Nordea, a project-coordinator at MAERSK, and currently works as an HR-manager at WOLT. Peter takes his career seriously, and works overtime several days of the week. Due to the lack of time for his personal life, he does not have any children, but spends most of his weekends with his wife, whom he had met at a company-organised Christmas party. Although his work can be frustrating at times, Peter enjoys it, and is still involved in majority of fields of the HR-department, with a focus on the recruitment process. He trusts his instincts, and believes that he has a unique skill in spotting the needle in the haystack when looking through CVs. He does not have a particularly positive opinion of ATS systems, and generally does not trust software, which make important decisions instead of him, as being in the backseat makes him feel powerless. At times, however, he must skip some less crucial steps in during his work, such as reading cover letters, or doing a complex sorting process of the CVs he has to go through. At times like these, at the end of the day, he often feels like he could have done a better and more thorough job, which make him unsatisfied with his workday.



Figure 11: The constructed persona, Peter Johansen. This picture was created by an AI [28], thus does not represent a real individual, nor is a subject to copyright.

4.8 Requirement specifications

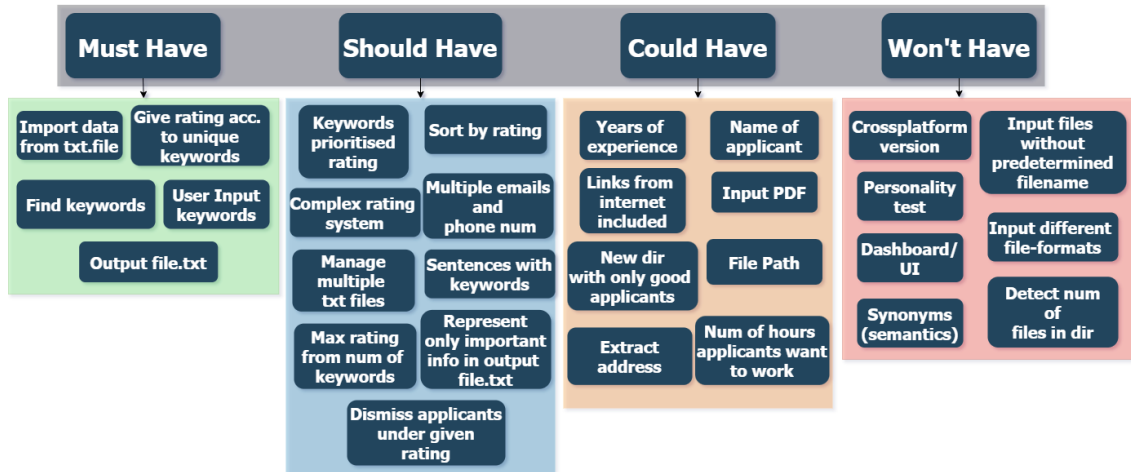


Figure 12: The MoSCoW method.

For each requirement presented, it is important to be aware of how it was decided to be significant enough to be depicted on the model, and what it contributes to the project. The MoSCoW model [17] a tool that provides an insight into the requirements, desires, and general priorities of a customer. It would be dim-witted to fully develop a product only to discover that the market it was made for simply does not need the functionalities and priorities that were falsely seen as pivotal parts of a solution. The MoSCoW model is an acronym that details following grouping of requirements:

- M - Must have (Required to meet the business needs)
- S - Should have (Worthwhile to have, but not required for success)
- C - Could have (Could be added if the other functionalities are present. More in line with shorter term quality of life changes)
- W - Won't have (Future additions that would be nice, but not a priority at this time)

Essentially, it boils down to the vital, important, possibly worthwhile, and future additions for a system/product to reach market success. The interviews have helped with understanding the needs of the real world compared to simply the assumptions made by the group initially, and the variety of interviewees from different branches of work also allow for a more objective view into the hiring process and priorities to ensure that the product is more generalised for the populous, rather than focusing on e.g. only one employer and specialising it for them.

The requirement specifications contains both functional and non-functional requirements [5]. The functional requirements pertain to what the program actually does, whereas the non-functional address how the software system should fulfill the functional requirements [5]. Most of the functional requirements were defined by the group, while the non-functional requirements are considered quality attributes of the program defined by feedback from the interviews and surveys.

4.8.1 Must Have

The "Must Have" category contains the fundamental requirements for the program. First, there is importing data from text file. It is known from the group's personal experience, and can be contributed to common sense, that the majority type of data handled when applying to a job is human-readable text-based data. This assumption is also confirmed in the research phase and in the interviews. Therefore it is imperative for the program to work with text-files. As a starting point, the exact type of text-file is not defined, but in a professional environment, PDF-files would be in the foreground of the group's research.

Finding keywords is also placed in the "Must Have" category, because the feedback from the group's interviews showed a general tendency for a need to find certain keywords in a given text. Rønhoff from AlfaNordic stated, that the company could make the most use out of a software, in which user-defined keywords could be compared with both a large number of CVs and applications. As for this project, the main focus has been set to exclusively handling applications, nonetheless, it is still a valid requirement to include. Also, Jeppesen from Kegme Børnehus mentioned, that they would greatly appreciate a software solution, in which the user-defined keywords were searched for in a given text-file, and then relevant subsections of the text were extracted and put into a transparent overview. It is also mentioned in the start of the [Analysis section](#), that finding keywords is a way of tackling the problem at hand.

User-defined keywords is important to be included in the most fundamental requirements, because of the overall important role it played during the interviews. It makes the program dynamic for the individual user, thus provides a much broader range of applicability. It is an alternative from having a static list of keywords, that the user does not necessarily wish to - or need to - search for, when looking for an optimal candidate. Furthermore, it was likewise mentioned by Jeppesen and Rønhoff, that the user should have the possibility to input their own keywords, and that they were on the lookout for a software solution, which functioned with only a few, more specific keywords, instead of working with an abundant list of seemingly relevant, but in reality a large, yet superficial set of words.

Again, providing a weighted rating to an unique keyword, instead of the amount of keywords included, was a highly sought out feature in all of the interviews. Especially, Rønhoff from AlfaNordic expressed her frustration with applicants spamming their cover-letters with keywords they could find in the job listing. According to her, in many ATS-systems, it results in a misleading, and for many good applicants, disadvantageous results.

The final requirement in the "Must Have" category is output in a text-file. That is, simply because it makes sense to store the results of a successful run-time of the program. When the results are saved in the form of a text file, they not lost in the terminal if it closes, thereby the user can always go back and inspect the results, even after closing the program. It makes the results from the program a lot more user friendly, when output in text file instead of the terminal.

4.8.2 Should Have

"Should Have" contains the category of features, that would take a basic standard program in the direction of a functioning application. The first feature is sort by

rating. It is straightforward, that the program should have a sorting system to filter out the worst candidates. In relation to both the problem statement and the mere idea of reducing time-consumption, it makes a lot of sense. Additionally, during the interview with Nielsen from DSB, it was clearly stated that the company was missing a sorting system where there were top "x" candidates and the rest was filtered out.

In extension to being able to sort candidates from best to worst, the program should also dismiss applicant under a given rating. The given rating is planned to be user-defined, and therefore it is a dynamic feature that allows the user to define at what rating the candidates would be filtered out. This gives the user more control over the process of hiring, which was mentioned in the interviews, referring to the example from sort by rating.

The idea of assigning specific weights to respective keywords was also mentioned in the interviews. The interviewees also had a general consensus about the fact that a prioritised keyword system would greatly benefit them in their search for the right candidate. The group's program solution is planned to prioritise keywords in sequential order. This decision was made to be able to construct a universal system, which gives realistic results when including all other factors in rating a text-file. On the other hand, this decision also comes with some disadvantages. More specifically, the user does not have the freedom to freely weight their keywords, even though in reality, the prioritisation of keywords is a much more nuanced process, than putting them in sequential order.

Multiple emails and phone numbers are in the "Should Have" section, because of two things. Firstly, it is frivolous to apply for a job without writing an email or phone number in the application. Therefore it is important to extract the contact info and filter those out, who did not bother including their contact-information on the application submitted. Secondly, the problem statement is about reducing the use of time and resources. By extracting the email and phone number into the new shortlist file, then it is easier for the job provider to contact the candidates.

In relation to having multiple emails and phone numbers and dismissing applicant under given rating, a fairly complex rating system was implemented. This feature gives applicants with missing email or phone number minus 10 points, because it would be widely considered unprofessional. Other than that, a point is also given per unique keyword, at last, the total point of each text-file is calculated, and by the user-defined minimum-point-boarder, it is finally determined if a text file gets a place on short-list or not. Like the other rating-related features, this was requested by most of the interviewees.

The next element in this section is the definition of a maximum possible rating. This was feature added to "Should Have", because it gives the user an good idea of how far away the actual candidate is from the best possible rating. It's also important to include, because the point system is fairly unknown for the user. They should not have to use time to add the ratings together themselves, because it goes against the guiding principle of the problem statement, which is saving valuable time.

When a recruitment specialist must find optimal candidates, one of the main actors in creating time-waste, is the sheer number of applicants. This is why managing multiple text files is a "Should Have" function (nearly Must-have). It is not nec-

essary for the program to work, but it is clearly a very sought out feature and is coherent in terms of the problem statement, where it is implied that the software solution should manage a great number of applicants.

In an interview conducted, Jeppesen from Børnehuset Kregme stated the following: ”..The ideal program should be able to summarise the sentences where keywords are used...”, when talking about the perfect software solution for streamlining their own hiring process. Another consideration for this feature is negations. An applicant could possibly collect a lot of points from keywords, but what if the keywords are mentioned by negation. By extracting the whole sentences, the user can quickly see if the use of the keyword is valid or not. Lastly, the feature correlates with the problem statement by extracting only sentences with keywords into the shortlist file. This results in a reduction of the reading-time of an application.

The last feature in the should-have category is being able to represent only important info in output text file. Referring back to outputting a text file, this feature is important to the program, because the program needs to save the results and relevant information from the applications into a file. That is, so the user can open the file at any time without having to run the program again. Therefore also saving time for the user, hence a sub-solution to the problem statement.

4.8.3 Could Have

”Could Have” category encapsulates functionalities, which lean towards the non-essential aspects of the program, as well as quality of life additions that improve the user’s experience by increasing the utility of the program.

The first feature in this category is the extraction of information concerning previous workplaces and years of experience. This would allow an employers to quickly glance over the relevant years of work that a potential candidate may have, which would be especially useful if a particular amount of experience is at minimum required, or at least expected before being applicable for a given job. It was mentioned by Mark from Jyske Bank, that ”the right experience is better than many years of experience”.

In the same manner, the address of a user could be extracted. This is important for employers, as a potential employee that lives closer to the workplace is seen as preferable option in comparison to someone that lives much further away, or even abroad in certain circumstances. Jeppesen from Børnehuset Kregme stated, that one of the first things she looks at when reading applicants is whether the commute is realistic, which in most cases narrows down applicants fairly quickly. The reason it is not in the ”Should Have” category is merely that an applicant could mention that they are willing to move for the job, or the location may not be a crucial factor at all. Nonetheless, they could get filtered out before the employer has the opportunity to locate that information and thereby miss out on a good candidate.

The extraction of the name of an applicant could make the summation document more thoroughgoing, but was considered unnecessary as the name will be stated clearly on the original application, and because it could potentially lead to discriminatory outcomes based on ethnicity. *Rørnhoff from AlfaNordic stated that the applicants are not allowed to disclose their names in their online application-template, or even include it in any attached papers, because it could be discriminating or establish a unwarranted bias.*

The number of hours an applicant is willing to work could also be displayed, but this may not be a common element for potential candidates to include in their application thus may be insignificant. Hours, and pay-negotiation is a topic that typically is brought up during the interview processes rather than during the initial screening stages. Another aspect of time-management, which could be looked at is if an applicant is looking to be employed full-time or part-time; this problem could however be solved with the already included keyword system.

Links from the internet was ultimately chosen to be categorised as "Could Have", given the time/programming constraints of the project. The feature would link the relevant information stated on the application like e.g. GitHub repositories, or LinkedIn, creating an even more user friendly approach to the final summation document.

Another neat feature would be to be able to utilise, and add the input file as a PDF instead of purely text-files. This would mean direct conversion from the PDFs that are being handed over to the employers, and an instant way to process it to retrieve the keywords/relevant information as a whole. Programming this feature was attempted, but the idea was discarded given the time constraints and issues creating such a function within C. That is considering there were a lot of other requirements that had to be met before the deadline.

A directory that contains the "good" applicants could also be useful. Especially, as it would make the management and processing of multiple applicants much simpler when only the relevant ones in the end remain in the final directory. This would also allow the employer to know exactly which text files are to be read more thoroughly and as all the junk would be filtered out by default to ease the process. The reason it is not in the "Should Have" section is that the library including the directory functions in C-language was only available for Linux platforms. Therefore, to include it in the program, the group needed to create the function, which was determined to be too time consuming.

The inclusion of links/paths to be able to open certain text-files was also considered. It is a way for the user to click on the applicants on the shortlist file, and gaining instant access to their submissions. This routine could reduce time for the user as they do not need to scour through the entire directory to find the text-files correlating to a given candidate. However, it was not prioritised.

4.8.4 Won't Have

The "Won't Have" category facilitates the features, and ideas that were not considered worthwhile for the product at this time, or the foreseeable future. While objectively potentially useful features the time constraint, priority placing, or development strain just simply makes it unworthy to pursue.

The first feature to discuss is a cross-platform version. Ensuring compatibility across multiple platforms would simply just not be worth the time, or effort especially as the benefit to having such a version would yield minimal benefits. Having the program function on one system that the masses use is enough for now.

A built-in personality test was considered, but as personality tests as a whole are individual in how a person prefers to structure and analyse them it make it incredibly difficult to make an objective personality test that companies would be happy to utilise. Many would not even use the function in the first place, and thus makes

it pointless and much to demanding to consider.

Given the development time and resource constraint, it was decided to not include a proper dashboard/user-interface. This would of course make the user experience more polished, clean and appealing but for now a fundamental working prototype was the main goal, thus it was discarded for now as it does not contribute enough to answering the problem statement. Mark from Jyske Bank mentioned that a dashboard with relevant information from the screening of applications could be beneficial for them, and it would generally flatten the learning curve of the program.

Identifying synonyms were an incredibly interesting topic discussed throughout the process, as not everyone utilises the same words, and thus how would one determine if a keyword was in fact used but just in a different way. In this case, while it makes sense to consider synonyms and account for all of the different formats, words and stylistic choices a person may use to treat everyone fairly, there was simply not enough time to discover the possibilities of this problem-field during this project. Currently, the program utilises pre-determined file names, and this will continue to be the case as it was not seen as a priority that it could handle all of the random names-formats that people may give their files. One way to handle this without any implementation would be to beforehand specify a preferred format of naming convention, or an automatic renaming with regard to the order received. Another way is the create a function for reading a whole directory since the existing directory only works for Linux.

The possibility of importing different file formats was also vetoed as currently text files, and potentially PDFs in the future are the main focus. Still, working with different file formats other than .txt would without a doubt be beneficial, because applicants often submits in mainly PDF or Word formats.

Dynamic detection of the number of files in directory was essentially implemented via the dirent.h library, but this could only be used by Linux platforms and was thus scrapped.

5 The product: A program that solves a problem

On the background of the product requirements, the main goal now becomes the creation of the product-prototype. Throughout this project, the product and report have been developed concurrently, thus the development process have also been updated consistently in correlation with the ongoing research.

5.1 Design

This section describes the design and development of the job application analysis tool, including the implementation of functions partitioned in three main phases; the achievability phase, the essentials phase, and the post-interview phase. These phases are described in more detail below. Additionally, a handful of code-snippets will be analysed, with a primary focus on what problem they solve, how it is solved, and why the proposed solution is appropriate. It is also essential to disclose, that the routines and inner workings of the final program were defined as a result of the research, analysis, and requirement specifications. All functions were developed in smaller groups of 2-3 people, and was later implemented in a master file collectively, using GitHub. The main advantages of this method are efficiency, universal readability, and dynamic range of usability.

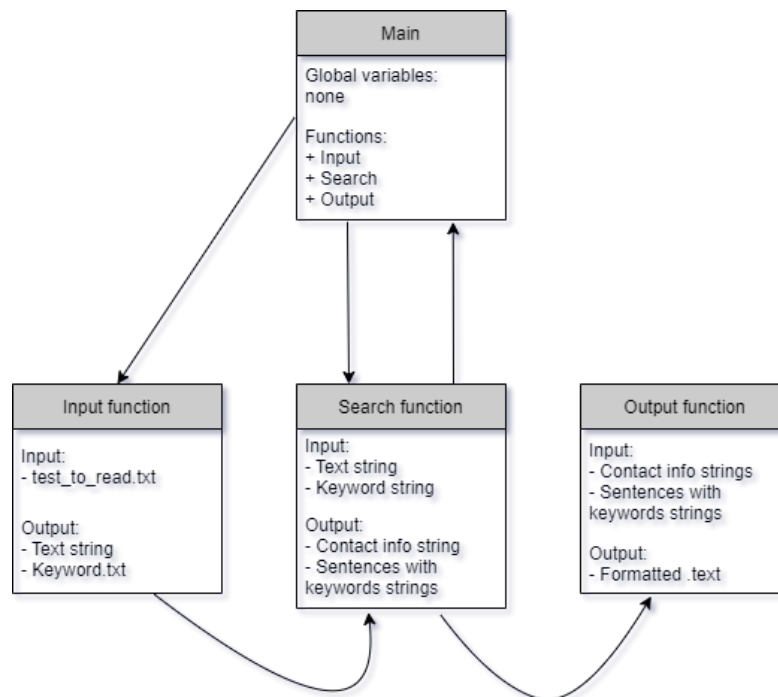


Figure 13: Flow-diagram for the main segments of the program.

As it can be seen in figure 13 the software solution has been divided into four main parts which are executed sequentially. These main parts are subsequently also divided into smaller, less challenging sub-problems that are solved in a segmented manner with individual functionalities. Early ideas took form in the form of collective brainstorming, and diagrams for preliminary program structures. In terms of programming, testing was being done concurrently alongside development. Meaning that every time a function or new protocol is implemented that it is immediately tested, and troubleshoot if the need arose.

The main-function of the program is to be kept as brief as possible. The main goal for this section of the code is to exclusively include function calls and as few global variables as possible - preferably none. At the same time, the function calls should establish a satisfactory insight into the overarching operations of the program.

The input function imports data from text files, and inserts them into appropriate storage spaces. At this stage, the most tangible solutions discussed were strings and string arrays. Later, the imported data was put into data structures.

The search function encompasses the analysis of a given text, using the previously imported data objects. Specifically, the text is scanned for keywords, contact information, and numerous other elements, which are crucial for an appropriate representation of the contents.

Finally, **the output function** summarises the data collected about the text in question, and structures it in a considerably more readable format, than the raw data. Additionally, when working with multiple files, the output is sorted in accordance with the rating of the texts.

5.1.1 Diagrams

On figure 14, the UML Sequence Diagram depicts the inner workings of the final iteration of the software solution. The main purpose of the diagram is to give an overview of the sequence of function calls, and the placement of their implementation in the finished product. For this diagram, it is chosen to only include the most central functions for better visibility and a more coherent overview. As mentioned in the [Methodology section](#), the y-axis represents the passing of time, and the most vital functions are depicted alongside the x-axis. It is also important to mention that this diagram does differ from what could be considered a conventional UML Sequence Diagram. Normally, classes would be used as headers, and their own functions would be shown under them, but here, a single function is a header, and a call is depicted with an arrow to a blank rectangle.

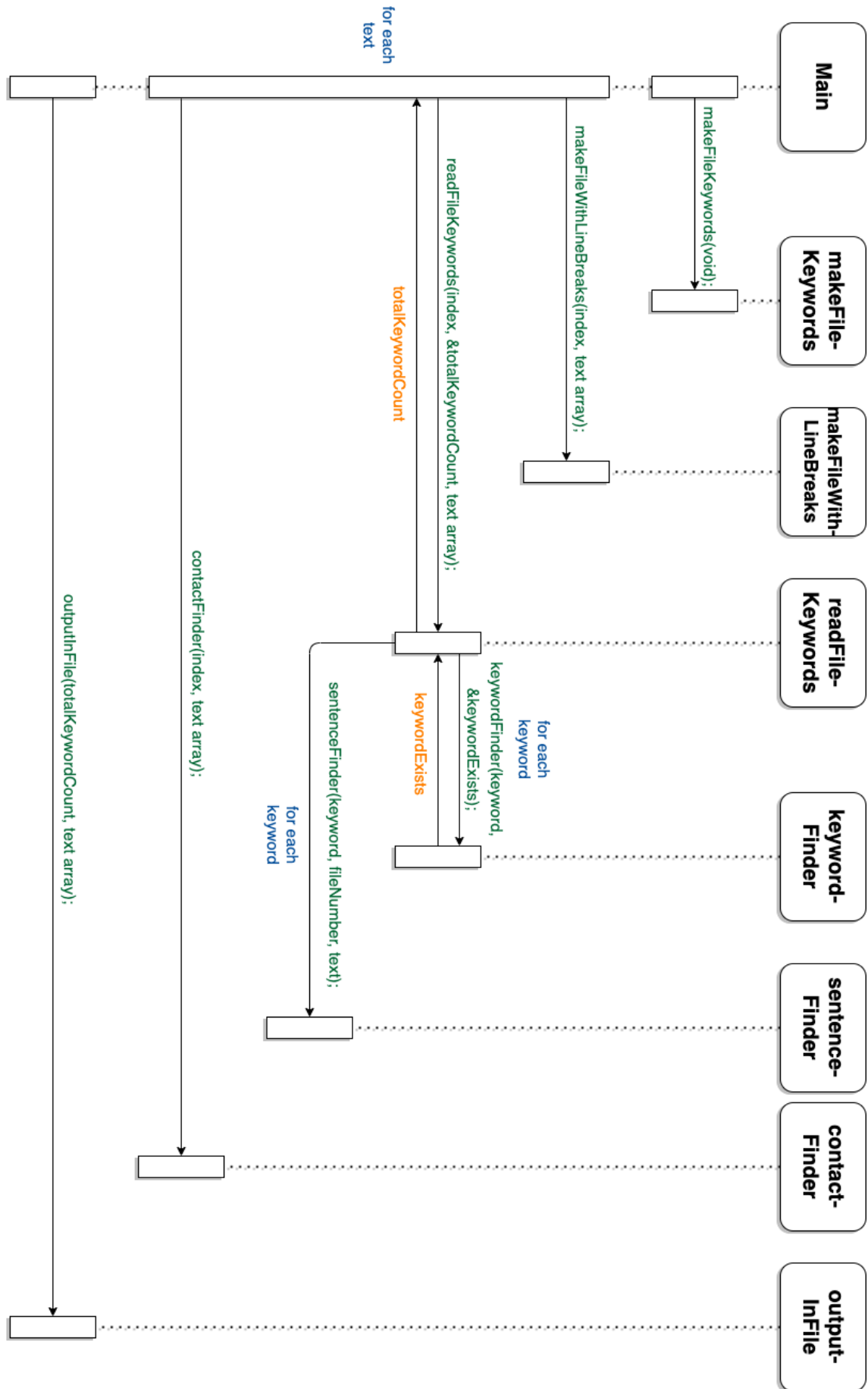


Figure 14: UML Sequence Diagram to the finished product.

For the construction of increasingly complicated functions, flowcharts such as the one on figure 15 were constructed. These diagrams are helpful in understanding the logic behind a functions inner workings when it may be too difficult to understand outright.

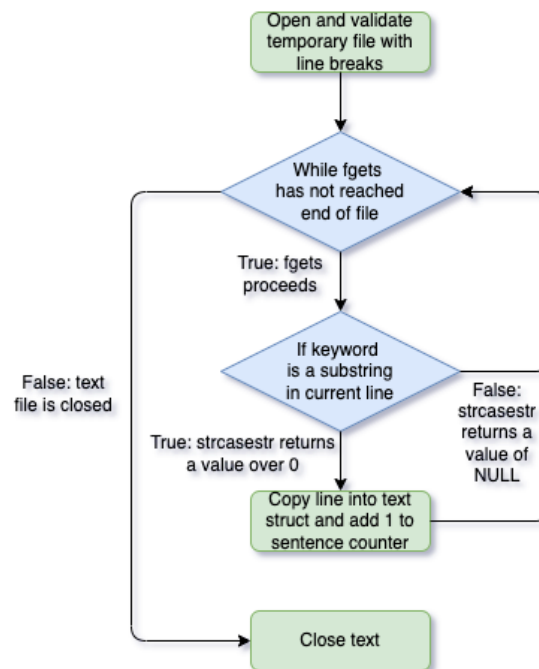


Figure 15: Flowchart showing the keywordFinder function.

When working with complex functions, pseudo code was often used to obtain an understanding of how a problem should be solved. Throughout the project, pseudo code was mostly written on a blackboard, but one of them has been digitalised on figure 16 to provide a more aesthetic representation of the method in-use. The functional code snippet for the same routine can also be seen in the "Implementation of code"-subsection.

```

readFileKeywords{
  keywords = keywords.txt -> open text file and validate
  number of keywords = 1 -> start with 1 because there is no space before the first word

  -----This part is to count the number of keywords to be able to make an array for them-----
  while ((char = getc(keywords)) != EOF) -> Read through keyword file
    if char == *space*
      number of keywords += 1

  reset pointer used to scan keywords

  -----This part is to import keywords from txt file into string array-----
  initialize keyword string array

  for (i < number of keywords)
    fscanf(put keywords from txt file in keyword string array)

  -----This part is to locate keywords in the text, and sentences with the respective keywords-----
  for (i < number of keywords)
    keywordFinder(keyword[i], &keywordExists) -> This function determines if the given keywords exists in the text

    if(keywordExists == true)
      update rating
      update number of keywords in text

    sentenceFinder(keyword[i]) -> This function extracts all sentences that include keyword as a substring
}

```

Figure 16: Pseudo code for the readFileKeywords function.

A simple flowchart diagram for the systematic implementation of ideas was also constructed, and can be seen on the figure 17 below. Each time a new function was to be developed and implemented, this process was followed. The relevance of this model comes into play, as in the beginning of software-development, the group was lacking professional opinions on what functions to include. By running all ideas through this workflow, the group could (up to a certain degree) give an educated guess to evaluate a routine’s relevance in the light of the problem statement and previous research.

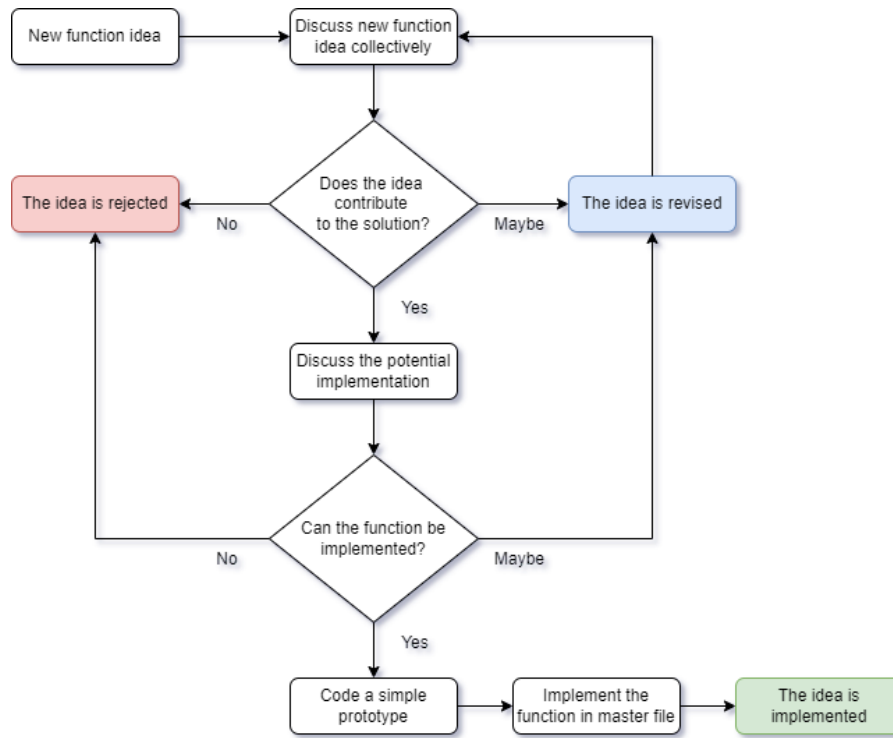


Figure 17: Route-diagram to the systematic implementation of ideas.

5.2 Phases during product creation

The following pages contain a detailed description on the different development phases of the product. The phase descriptions are to a large extent based upon the specification requirements from the MoSCoW model (figure 12) discussed previously, and each phase essentially focuses on tackling their share of specification requirements, and/or building upon and enhancing previously solved specification requirements.

5.2.1 Achievability phase

The achievability phase represents the most bare-bones prototype of the product. During this phase, the absolute fundamentals of the program are implemented, meaning all of the functions included are a necessity for a working product. Because of the short time-frame of the project, the development began before the majority of the research was completed. Therefore, it is logical that the first prototype only contains essential functionalities, meanwhile future research and expansion is being

planned. Additionally, this phase also greatly contributed to the common understanding of the goals and level of ambition, which should be set for the product itself. Due to the group's lack of coding experience, it was crucial to set realistic goals.

All routines implemented during this phase are specifications from the "Must Have" section. Specifically, "Import data from text-file", "Find keywords", "User Input keywords" and "Output text-file". To meet the specification requirements in the current phase, the following functions were developed: *makeFileWithLineBreaks*, *readFileKeywords*, *makeFileKeywords* and *keywordFinder*. The *makeFileWithLineBreak* function creates a temporary duplicate of a text-file, the only difference being that the duplicate does not include any punctuation, and a line-break separates each sentence. Generally, when scanning string-based data in C-language, line-breaks indicate the end of a set of information with a so-called s zero-character. This is important when using standard C-library functions such as "fgets" or "fscanf". The temporary file with line-breaks is then used in *keywordFinder*, and is overwritten every time a new text-file is to be analysed - this also reduces clutter. The *makeFileKeyword* function creates or overwrites an already existing keywords.txt file, storing all of the user-defined keywords. This file is then used in *readFileKeywords*. Thereafter, *readFileKeywords* reads the keywords located in keywords.txt, and calls *keywordFinder* for each word within the file. When *keywordFinder* is ran, the file created through *makeFileWithLineBreak* is scanned and checked for the existence of each keyword as a sub-string of each line.

5.2.2 Essentials phase

The essentials phase represents a further improved version of the final product, now containing a set of functions which are not vital for the program itself, but are central for a satisfactory solution to our problem statement. This phase takes ongoing research and data from the survey into account. Some of the most crucial problems tackled during this phase are extracting multiple email addresses and phone numbers, managing multiple text-files, and allocation of sentences with keywords. *contactFinder* and *sentenceFinder* were developed as suitable functions for these tasks, and main-function was modified to give the user the possibility to work with several files at once. For now, it was straightforwardly implemented as a for-loop, running each function for every text. The majority of the other functions were also slightly altered and optimised to be able to work with multiple files. The purpose of *contactFinder*, is to scan cover letters for the applicants' phone numbers and email addresses, and save them in their respective char arrays (they are later altered to be saved in structs, and a maximum number of 4 email addresses and 4 phone numbers is defined). *readFileKeywords* and *keywordFinder* have been upgraded to now also count and print out the the amount of unique keywords found. This optimisation also sparked a fascinating discussion of the implementation of a universal rating system, which is also a central element of the next and final phase. The function *sentenceFinder* opens the previously described temporary file created by *makeFileWithLineBreaks*, and prints out every line that contains a keyword (this is later optimised to save these lines in data structs rather than printing them). At last, a *validation* function was created to check if the file-pointer created when importing a file is a void-pointer. If it is, an error message is displayed, since it means that the file does not exist in the directory. The function was developed for better code

readability and also to reduce the program size.

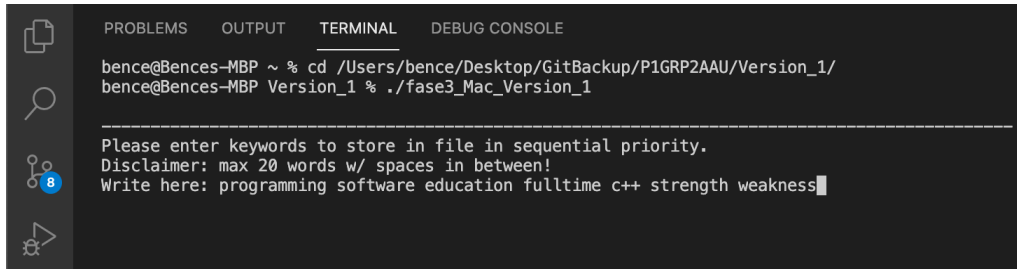
5.2.3 Post-interview phase

The post-interview phase is an expansion that largely utilises data from the gathered from professional interviews. This phase indubitably represents the biggest leap in the quality of the software solution, as it is here, that the solution really got tailored to the needs of the user. This phase focuses on implementing the last "Must Have" requirement specification "giving rating according to unique keywords". The late implementation of this central routine was chosen to be allocated in the final phase, since trustworthy data was crucial to define a rating system to best suit the needs of real-life recruitment specialists. The implementation of all the remaining requirement specifications from the "Should Have" section is also done here, since their concrete functionalities are also highly dependent on the feedback the group received. To meet the specification requirements the following functions were developed: "contactRating", "qsortComparison" and "outputInFile", as well as many functions from the previous phase were further developed and optimised to support the current phases requirement specification. A struct-array named *texts* was developed to contain all the information of the cover letters, which makes the information of the all applications more manageable and easier to access in the different functions. *sentenceFinder* is now improved to also save the sentences in char arrays in *texts*, and tackle them individually, using a dynamic char-array. In the *keywordFinder* function, the amount of matches of a single keyword is not needed anymore, thus it has been discarded. The *contactFinder* function now stores the found contact information of the cover letter in a string arrays defined as struct-elements in *texts*. *contactRating* now checks what form of contact information each cover letter has included, and depending on the outcome the cover letters could either lose some points or be completely disqualified. *qsortComparison* is a comparative function used in combination with "qsort"; it takes two struct-pointers as function parameter, and the ratings of the two text-structs are then compared in the function. "Qsort" is the algorithm used to sort every element of the struct-array containing information for all cover letters before printing the results in the shortlist-file. This very file is created by the function *outputInFile*, and contains a list containing a brief of all qualified cover letters starting with the one with the highest score.

5.3 How does the software work?

In a relatively larger company's recruitment process, the first step for a person hiring is to go through all applications and cover-letters and to look for pre-determined keywords appropriate for the given job, and save the ones that are most qualified. The factors determining the level qualification have been selected by analysing results from the groups' research. The process of going through applicants' documents is what the software is going to do for the job recruiter, specifically focusing on cover-letters, as a lot of software for screening CVs has already been done, and professionals are generally more appreciative for software which gives them extra information, while leaving their usual workflow in-tact. According to our interviews with recruitment professionals, cover-letters are an incredible, but unregulated and muddy pool of data, which is not usually explored to a point of satisfaction. With the group's software solution, working with cover-letters can get much less tiresome, as it presents a clear and systematic overview of all applicants.

In general terms, the software helps job recruiters sort through (a potentially large number of) cover-letters in a quick and easy manner. When the user has received the documents of all potential candidates, all the user has to do is place every text-file in the program folder, start the program, and follow two very simple instructions communicated via prompt messages in the terminal. The first one asks the user to manually input a list of keywords to be searched for in each document in an order of sequential priority.

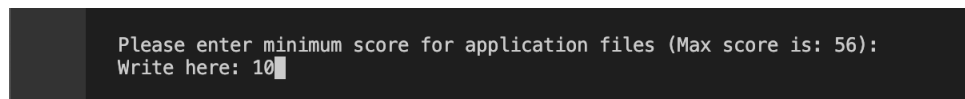
A screenshot of a terminal window with a dark background. The window has tabs at the top labeled 'PROBLEMS', 'OUTPUT', 'TERMINAL', and 'DEBUG CONSOLE'. The 'TERMINAL' tab is active. The prompt shows the user is at 'bence@Bences-MBP' in the directory '~ %'. The command executed is 'cd /Users/bence/Desktop/GitBackup/P1GRP2AAU/Version_1/'. The output shows the program's prompt: 'Please enter keywords to store in file in sequential priority. Disclaimer: max 20 words w/ spaces in between! Write here: programming software education fulltime c++ strength weakness'.

```
bence@Bences-MBP ~ % cd /Users/bence/Desktop/GitBackup/P1GRP2AAU/Version_1/
bence@Bences-MBP Version_1 % ./fase3_Mac_Version_1

-----
Please enter keywords to store in file in sequential priority.
Disclaimer: max 20 words w/ spaces in between!
Write here: programming software education fulltime c++ strength weakness
```

Figure 18: The user-defined keywords are received via the terminal in a sequential priority.

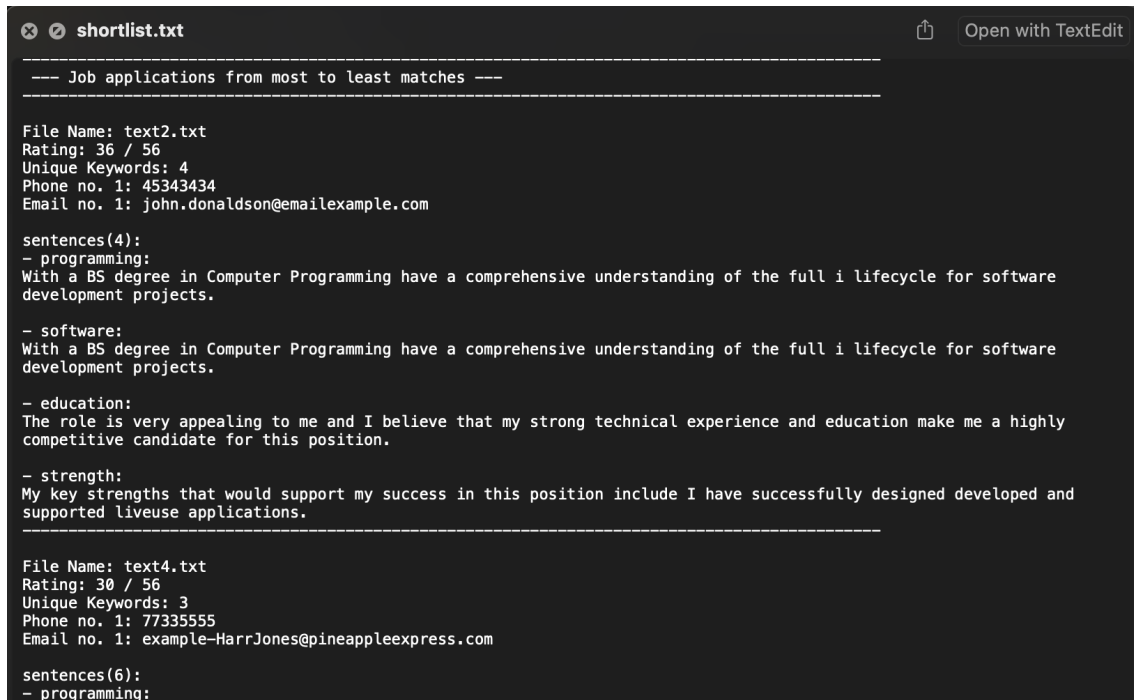
Various files will now be rated: The most prioritised keywords, if matched in a text, will result in ten points, the next nine, and so forth until a two point threshold is reached. In addition, every unique keyword will also give one point. If no email is present, 10 points will be deducted. This also applies to phone numbers, but if neither phone or email is detected, 100 points will deducted, technically disqualifying the candidate. Thereafter, the maximum possible score is shown, and the user is asked again, to input the minimum score, below which a cover-letter should be dismissed (meaning it will not appear in the final shortlist file).

A screenshot of a terminal window with a dark background. The prompt shows the program asking for a minimum score: 'Please enter minimum score for application files (Max score is: 56): Write here: 10'.

```
Please enter minimum score for application files (Max score is: 56):
Write here: 10
```

Figure 19: The user-defined keywords are received via the terminal in a sequential priority.

That simply, a text file called "shortlist" is created, containing a brief for all cover-letters that possess a rating above the user-defined minimum. The mentioned brief consists of a total rating (that naturally also decides the sequence in which the briefs are printed), a number of unique keywords used, and all of the relevant sentences extracted, which contain a keyword (case and pre/suffixation are both ignored). Furthermore, the files' names, and all email addresses and phone numbers are extracted for each cover-letter included.



```
shortlist.txt
Open with TextEdit

--- Job applications from most to least matches ---

File Name: text2.txt
Rating: 36 / 56
Unique Keywords: 4
Phone no. 1: 45343434
Email no. 1: john.donaldson@emailexample.com

sentences(4):
- programming:
With a BS degree in Computer Programming have a comprehensive understanding of the full i lifecycle for software
development projects.

- software:
With a BS degree in Computer Programming have a comprehensive understanding of the full i lifecycle for software
development projects.

- education:
The role is very appealing to me and I believe that my strong technical experience and education make me a highly
competitive candidate for this position.

- strength:
My key strengths that would support my success in this position include I have successfully designed developed and
supported liveuse applications.

File Name: text4.txt
Rating: 30 / 56
Unique Keywords: 3
Phone no. 1: 77335555
Email no. 1: example-HarrJones@pineappleexpress.com

sentences(6):
- programming:
```

Figure 20: The top-candidate from the text-file generated by the software.

After the rating, all documents scoring above or equal to the minimum score are put into a document, shortlist.txt, accommodated by the content of the documents relevant to the employers, namely: contact information, rating, file name and matched keywords and the sentences they are used in. To elaborate on the workings of the program, an example is provided:

A job posting, looking for software engineers with capabilities in C# has been created, and 100 people have submitted their CVs and cover letters. The recruiter uses some software to screen the CVs, and wants to use the software described in this report for processing the cover letters. In this case, it is especially important to the recruiter that applicants excel in teamwork. Relevant keywords to the offered position is selected by the job recruiter to be: “Communication”, “Teamwork”, “C#”, “Community”, and “Software”. The highest achievable score would in this case be 45, and the recruiter only wishes to see applications scoring 30 or higher. A list of files true to the conditions specified above will be created for the recruiter, providing them with a condensed and highly representative list of the candidates’ degree of optimality.

It is worth mentioning that, the more keywords recruiters provide, the more precise and fulfilling the results will be. Furthermore, simulatory exercises are done consistently throughout the project. This is in order to continue testing, troubleshooting and to promote further progress. Throughout the iteration process, various test files are continually run with a mix of consciously chosen keywords. In addition various edge cases are investigated to evaluate and further test the product for issues that need correcting. The process of coding software and associated functions will be described more thoroughly in the next chapter.

5.4 Implementation

This section provides insight to the development of the final program by showing the implementation of the concepts, ideas and specifications previously discussed. The thought-process behind the development of the different functions will also be elaborated on. The procedure of developing the program was split into multiple phases; the achievability phase, the essentials phase, and the post-interview phase (a thorough description of each phase can be found in the [Design section](#)). In each phase, a collection of requirement specifications were implemented in the form of individual routines. Each routine has been developed whilst being aware about its complexity, structure, efficiency (big-O), as well as overarching compatibility. This section takes a closer and more detailed look at a few smaller functions in their entirety, as well as snippets from some of the larger functions. These snippets are all exclusively taken from the final prototype of the program (version 1.0).

5.4.1 Pre-processing units

The program makes use of the following C libraries by using `#include` C directive:

- `stdlib`
- `stdio`
- `string`

The libraries employed in the finished program are exclusively C-language standard library functions. The inclusion of the mentioned libraries gives access to various functions enabling usage of input and output of data, utility functions like `"qsort"` and `"malloc"`, and functions used for modifying strings, such as `"strcasestr"` and `"strcpy"`.

The program also makes use of symbolical constants by using `#define` C pre-processor directive. The use of symbolic constants enables tweaking certain aspects of the code by changing multiple variables' predefined values by simply changing a single number at the top of the source code. Additionally, working with symbolic constants also makes the source code more readable. Some examples of symbolic constants are: a constant for the maximum length of an extracted sentence, a constant for the number of emails or phone numbers that should be stored for every text file, and a constant for the maximum length of a word.

5.4.2 Organising data

To store data in an organised and manageable manner, data structures (structs) were chosen to be implemented in the final prototype of the product. At the starting point of the development process, none of the developers had knowledge about structs, as it was a subject taught in one of the final lectures in the imperative programming course. For this reason, early versions of the product were storing and passing data through numerous arrays. Data such as keywords, sentences, and contact information, were consequently stored in different arrays. In comparison with the code with structs implemented, the earlier iterations could quickly become disorienting to work through, and confusing to make changes in. Due to the lack of an "object-oriented programming"-like system, resolving errors were also much more challenging. Subsequently to the lectures about structs, it was almost immediately

decided that their inclusion was a main priority for future iterations. In the final version of the software, a struct is used to collectively store all data relating to each text file individually. Namely, rating, number of keywords, number of sentences with keywords, file-name, email(s), phone number(s), and the identified keywords and their respective sentences in the file. For every single text, a new instance of this struct is put into a struct array. The array is defined in the main function of the software alongside other functions. Some of these functions take the struct arrays as input parameters, and use/manipulate members of the structs in ways described in the subsections below. Typedef is used for easier access. A snippet of the struct used in the function is presented below:

```
typedef struct
{
    int rating;
    int keywordCounter;
    int sentenceCounter;
    char filename[WORD_LENGTH];
    char keywords[STRING_NUM][WORD_LENGTH];
    char sentences[STRING_NUM][SENTENCE_LENGTH];
    char email[PHONE_OR_EMAIL_LENGTH][SENTENCE_LENGTH];
    long phone[PHONE_OR_EMAIL_LENGTH];
} text_file;
```

Figure 21: Data structure used in product.

5.4.3 Main function

As previously mentioned, the main function predominantly consists of function calls. This allows for better readability of the source code. Generally, it is expected from a main function, that it provides a satisfactory overall picture of the program's functionality without too much clutter. The main function starts by dynamically allocating memory for an struct-array called *texts* via *malloc*. This struct-array contains variables to store the information waiting to be extracted from cover-letters. This can be seen on figure 22, in which case the number of *texts* is defined by a symbolic constant named *TEXT_NUM*. If the memory cannot be allocated, an error message is printed, and the program is closed, otherwise the program continues to run and calls the *makeFileKeywords* function. The *makeFileKeywords* function creates a file containing user-defined keywords entered through the terminal. Afterwards, the functions *makeFileWithLineBreak*, *contactFinder*, *readFileKeywords* are run via a for-loop, until the condition is equal to the constant *TEXT_NUM*. Firstly, *makeFileWithLineBreak* removes punctuation and creates a line-break every time it spots a punctuation symbol, and secondly it exports the edited file as a temporary file. As formerly stated, this temporary file is overwritten with every new call. Consequently, *contactFinder* extracts contact information from the cover letters, and *readFileKeywords* puts keywords from a keywords.txt file into a string array, and calls other functions. The functions *contactRating* and *outputInFile* are only called once. *contactRating* checks each cover letter for contact information, depending on the severity of the missing contact information a score penalty is applied accordingly. *outputInFile* outputs the extracted text-based data into *shortlist.txt* as a ranked list with information over all the cover letters, that have scored more than the user-defined minimum score. Lastly, via *free* the dynamically allocated memory

for the array of structure *texts* is deallocated to make space for new data in the array. Lastly, a message is printed in the terminal if all operations have successfully been executed.

```
text_file *texts = malloc((TEXT_NUM) * sizeof(text_file));
if (texts == NULL)
{
    printf("Memory could not be allocated.\n");
    exit(EXIT_FAILURE);
}
```

Figure 22: malloc - Dynamical data allocation.

5.4.4 makeFileKeywords

makeFileKeywords is the first function called during the run-time of the program. This function is in charge of receiving user-inputted keywords, playing a central part in the algorithm's sorting process. The function opens a file - *keywords.txt* - in write-mode, creating a file pointer. Next, the validation function is run to confirm that the file exists, and the user input is read via *fgets*, which looks at the user-input and puts it into a char array (string), *data*, defined at the top of the function. The size of the array declared to be the symbolic constant named *MAX*. *MAX* simply represents a large number that is able to tackle edge-cases. The content of the array is then put into *keywords.txt* via the function *fputs*. A snippet of the full *makeFileKeywords* function can be seen here:

```
// Creates keyword file containing user input
void makeFileKeywords()
{
    char data[MAX]; // Stores terminal input

    FILE *keywords = fopen("keywords.txt", "w");
    validation(keywords);

    printf("\n-----\n");
    printf("Please enter keywords to store in file in sequential priority.\n");
    printf("Disclaimer: max 20 words w/ spaces in between!\nWrite here: ");
    fgets(data, MAX, stdin); // Puts keywords from terminal to array
    fputs(data, keywords); // Puts keywords from array to file

    // Close file to save file data
    fclose(keywords);
    printf("\nKeyword file created and saved successfully.\n\n");
}
```

Figure 23: makeFileKeywords.

The following functions in main are all executed within a for-loop, making them process all of the cover-letter text-files: *makeFileWithLineBreaks*, *contactFinder*, and *readFileKeywords*. The loop is run until the integer, *i*, used for determining which file number is processed, is equal to the defined symbol, *TEXT_NUM*, which is the number of applications imported.

5.4.5 makeFileWithLineBreaks

In general terms, *makeFileWithLineBreaks* creates a temporary file, removing unnecessary symbols, and replaces periods with line breaks. This temporary file is used in later functions, as it enables the possibility of identifying and storing individual sentences. More specifically, the function takes the earlier mentioned file number and a file with a not yet specified name as parameters. The process of defining the name of the files is also done in this function. First, char array *fileNameFormat* is initialised. The formats of the file names is *Text *integer placeholder*.txt*. The char array named *fileName* is also defined, though only by its size, which is the size of *fileNameFormat*. The mentioned char arrays are then used as parameters in *snprintf*, putting the string *fileNameFormat* into *fileName*, where the integer for the placeholder in the string is the file number parameter of *makeFileWithLineBreak*. The combination of these functions results in file names such as "text1.txt", text2.txt", etc. This string is then via *strcpy*, copied into the earlier mentioned struct-array in the *filename* member at the respective index. A snippet of the while-loop replacing periods with newlines, and removing unnecessary symbols are displayed below:

```
// Get strings from file
while (fgets(input, MAX, text) != NULL)
{
    // INCLUDE THE FOLLOWING LINE ON WINDOWS
    // puts(input);

    for (int i = 0; input[i] != '\0'; i++)
    {
        // Flag is a boolean used to mark if punctuation has been spotted
        int flag = 0;

        // Change char subsequent to period to line break in input string
        if (input[i] == '.')
        {
            input[i + 1] = '\n';
        }

        // Compare to all the different punctations
        for (int j = 0; j < 13; j++)
        {
            // Check all punctuation
            if (input[i] == punctuation[j])
            {
                // Flag is true
                flag = 1;
                break;
            }
        }

        // If flag is not true, put input array value into output list
        if (!flag)
        {
            output[outputChar++] = input[i];
        }
    }
}
```

Figure 24: makeFileWithLineBreaks.

A file with the recently created name will now be opened, and the validation function will be called. Also, two char arrays named *Input* and *Output*, sized by the *MAX* symbol are defined. If no errors occur in the validation function, a while-loop running until *fgets* until returning *NULL* will be run, thus iterating over every line in the

text-file and copying the sentences into the string, *input*. Within the while-loop, a for-loop is called, comparing every char in the given sentence to every char in a previously defined char array, containing punctuations wished to be removed from the text, enabling the workings of functions called later on. Whenever a char is equal to a punctuation, a boolean named *flag*, assigned the value true/one, and thereby breaks out of the for-loop. An if-statement will add the chars from *Input* to *Output* if the boolean is false (0 in C-language). Another if-statement before the for-loop will replace the iterated char with a new line, if equal to a period. The text file is then closed and the *Output* is copied into the temporary textfile via *strcpy*. Lastly a success message is printed.

5.4.6 contactFinder

As the name of the function suggests, *contactFinder*, identifies phone numbers and emails in the text files and copies those into the struct-array at the index of *i*, representing the file number. Emails are copied into the email member of the structs, which are string arrays, and phone numbers are copied into the long int arrays named *phone*. The members storing contact information are two-dimensional arrays, as it allows storing multiple emails and phone numbers, though no more than four of each as a cover letter is unlikely to contain more. The length of the arrays are defined in the struct with the symbolic integer constant named *PHONE_OR_EMAIL_LENGTH*. The function use file number and text file the same way *makeFileWithLineBreak* does, and also creates the name from file number and file name format accordingly. After this, the text-file is opened in read-mode. Two while loops are present: one for identifying phone numbers, and one for identifying emails. These could definitely be combined to a shorter function, but are not due to a lack of time and low-priority in the development process. Both of the while loops use "fscanf" to copy strings from the file into the char array defined above it, named *full_text*. One string at a time, they are examined by *if* statements and functions from the *string.h* library.

An if-statement will be true if the length of examined string are 8 or 10 characters. If true, the *phoneNumber* variable of type integer is set equal to the return value of *strtol*, a function that confirms whether or not a string is numeric. *strtol* examines s string one char at a time, until a non-digit char or a limit, provided as input parameter, has been reached - in this case, the limit is 10. Another if-statement - if true- copies the phone numbers into the array of *phoneNumbers* in the struct-array at the index of file number and at the index of *phoneCounter* in the *phoneNumber* array. *phoneCounter* is an integer functioning as a counter and is incremented by one every time a phone number is identified. The while-loop identifying phone numbers is shown here:

```

while (fscanf(text, "%s", fullText) != EOF)
{
    // -----THIS FOR PHONE-----
    // Check if scanned word is 8 characters long
    if (strlen(fullText) == 8 || strlen(fullText) == 10)
    {
        // Check if scanned word contains digits, if not 0 is returned,
        // sometimes it returns some of the digits, causes problems...
        long phoneNumber = strtol(fullText, &point, 10);

        // check validity of phone number
        if (phoneNumber >= 10000000)
        {
            // Add phone number to array
            texts[i].phone[phoneCounter] = phoneNumber;
            phoneCounter++;
        }
    }
}

```

Figure 25: contactFinder - identifying phone.

The truth value of the if-statement handling mail identification is dependant on the return value of the function *strrchr*. The function examines if a string contains the char "@". If the statement returns true, the examined string is copied into the a string array named *email* at the index of *mailCounter* - an int declared above the loop, which is increased by one whenever a mail is identified. A for-loop outside of the while-loop is examining every mail found and replacing periods at the end of the emails with empty spaces (if there are any). Within this for-loop, the identified mails are copied into the char-array of emails in the struct at the index of the momentary file number, via *strcpy*. The while-loop identifying emails is shown in the snippet underneath:

```

while (fscanf(text, "%s", fullText) != EOF)
{
    if (strrchr(fullText, '@'))
    {
        strcpy(mail[mailCounter], fullText);
        mailCounter++;
    }
}

for (int j = 0; j < mailCounter; j++)
{
    int mailLength = strlen(mail[j]);
    if (mail[j][mailLength - 1] == '.')
    {
        mail[j][mailLength - 1] = '\0';
    }
    strcpy(texts[i].email[j], mail[j]);
}

```

Figure 26: contactFinder - indentifying mail.

5.4.7 readFileKeywords

The *readFileKeywords* function is one of the larger functions included in the product. It plays a critical role in the core purpose of the product, as it is handling the process of finding keywords and the sentences they are used in, as well as rating the text files. The function does so, by the help of the functions called within the function itself, namely *keywordFinder* and *sentenceFinder*. Both functions read the temporary text file created earlier by *makeFileWithLineBreaks*, in which no unnecessary symbols are present, and all sentences are separated by line-breaks, which is notably useful for *sentenceFinder*. Before calling mentioned functions, the file *keywords.txt* is opened in read-mode, and a while-loop counts the number of keywords in the text file by counting the amount of spaces, since the keywords in the file are separated by those. The amount of keywords is then stored in the int pointer *totalKeywordCount*, a parameter in the function. Two other parameters, namely "fileNumber" and "texts" are also present. Next, a for-loop inserts the keywords in *keywords.txt* in the string array named *keyword*, iterated by a for-loop, in which *keywordFinder* and *sentenceFinder* are run.

keywordFinder looks for a string in a text-file and returns the value, one, if the string was found. *keywordFinder*, has two parameters: a string representing the keyword to scan for, and an integer-pointer used as a boolean for determining if the keyword is found in the text file. In the function, a while-loop scans each string and puts it into the *fullText* char array. In the for-loop, an if-statement increments the integer *keywordNum* by one, each time the iterated string is determined to be the same as the keyword examined. This is achieved by the use of *strcasestr*, a function comparing two strings without case sensitivity, returning one if the two compared strings are the same. *strcasestr* is a function only available on MacOS, and therefore, a case-sensitive alternative, *strstr* is used in the Windows build. The while-loop determining if a keyword exists in the text can be seen here:

```
// Checks number of keywords
while (fscanf(text, "%s", fullText) != EOF)
{
    // IT MUST BE strstr ON WINDOWS BECAUSE OF MISSING SYSTEM PACKAGE
    // FOR MAC: strcasestr ignores case, thus gives more accurate results
    // Checks if keyword exists
    if (strcasestr(fullText, keyword) > 0)
    {
        keywordNum++;
    }
}

if (keywordNum > 0)
{
    *keywordExists = 1;
}
```

Figure 27: keywordFinder.

When *keywordFinder* has been called, a for-loop calculates the rating of the struct in the struct-array at the index of *fileNumber*. The first keyword in the array gives ten points, next eight, and so forth, until the the minimum of two has been reached. For every unique keyword matched, an additional point is given. The points are stored in the respective structs' rating members.

sentenceFinder is run after the rating process, parsing through each sentence in the text, looking for the existence of a keyword, and copies both sentence and keyword into two separate string arrays in the struct if true. In this function, a while-loop is running until the function *fgets* no longer returns a string. *fgets* examines the file *text_temp.txt* line by line, which is why periods were earlier replaced with line-breaks, as each line in *text_temp.txt* corresponds to a sentence from punctuation to punctuation in the original text file. In the while-loop, an if-statement scans the line for the keyword taken as input parameter, and is true if the keyword is found in the sentence. As in *keywordFinder*, this is done by the function *strcasestr*. *strcasestr* returns the string found in the line and returns nothing if nothing was found, and this is why the if-statement is true if the return value of *strcasestr* is greater than 0. When the if-statement is true, *strcpy* is used to copy the line into the struct member *sentences* (a string array), at the index of *fileNum* in the struct-array, and at the index of the iterated struct's *sentenceCounter* member. The *sentenceCounter* of the struct is increased by one, for each iteration of the while-loop. The keyword is stored in the *keyword* member of the struct (also a string array) at the index of *fileNum* in the struct array, and at the index of the same struct's *sentenceCounter* member subtracted by one. Snippet of the while-loop identifying and adding lines and keywords to struct:

```
// Checking for substring
while (fgets(line, sizeof(line), text))
{
    // IT MUST BE strstr ON WINDOWS BECAUSE OF MISSING SYSTEM PACKAGE
    // FOR MAC: strcasestr ignores case, thus gives more accurate results
    if (strcasestr(line, keyword) > 0) // warning: ordered comparison between pointer and zero
    {
        strcpy(texts[fileNum].sentences[texts[fileNum].sentenceCounter++], line);
        strcpy(texts[fileNum].keywords[texts[fileNum].sentenceCounter - 1], keyword);
    }
}
```

Figure 28: *sentenceFinder* - while-loop.

Looking at the software's two functions *sentenceFinder* and *keywordFinder*, one might realise, that the *keywordFinder* could be considered obsolete, if the variable from *keywordExists* was moved into *sentenceFinder*, and the for-loop taking care of the rating process, was moved under the function call of *sentenceFinder*. This is because the two mentioned functions are very similar in their workings, and both parse the texts in search of keywords. This will be discussed in future sections.

5.4.8 contactRating

In main, when the functions, *makeFileWithLineBreaks*, *contactFinder* and *read-FileKeywords*, have been called and ran through the formerly described for-loop, the function *contactRating* is called (outside of the loop). *contactRating* is a relatively small function, and is part of the rating system of the software. *contactRating* iterates each text file and deducts ten points if no phone number is found in the respective text. Ten points will also be deducted if no email is found, and if no email or phone number is present in a text, one-hundred points will be deducted, technically disqualifying the candidate. The function takes the struct array of text-files as a parameter, and examines each text within a for-loop. The for-loop runs until the integer *j* - initially having a value of zero - is bigger than the symbolic constant

TEXT_NUM. Within the loop, there are three if-statements. The first statement is true, if the length of the mail is smaller than five. The mail examined is the first element in the string array named *email*, at the index of *j* in the struct-array. A length smaller than five is adequate, as no mail would be shorter than five characters in length. An email consists of at least one character, followed by "@" and lastly a website domain of at least three characters, e.g: *a@b.cd*. If the if-statement is true, ten points will be deducted from the member named *rating* in the struct array at index *j*. The next if-statement is true if the phone number of the same struct as before at index zero is equal to zero. This is because the phone number will be "0", if no phone number is found in *contactFinder*. If true, ten points will be deducted the same way as before. The last if-statement, is simply the two if-statements, spoken about above, combined in one argument with the logical expressions of two "and" signs. If this statement is true, one-hundred points will be subtracted from the rating member the same way as the other instances. One might wonder how the index of zero in the arrays of phone numbers and emails, when determining if no mail or phone number exists, might be sufficient. This is simply because any contact information - if found - always will be put in the first slot of the mentioned arrays, and it is therefore not necessary to examine the other slots. Snippet of for-loop determining rating from identified phone numbers and emails can be observed below:

```
for (int j = 0; j <= TEXT_NUM; j++)
{
    if (strlen(texts[j].email[0]) < 5) // If there is no mail, then mail is "None", so strlen returns 4
    {
        texts[j].rating -= 10;
    }

    if (texts[j].phone[0] == 0) // If there is no phone number, then phone is "0"
    {
        texts[j].rating -= 10;
    }

    if ((strlen(texts[j].email[0]) < 5) && (texts[j].phone[0] == 0))
    {
        texts[j].rating -= 100; // Text is disqualified if there is not contact information
    }
}
```

Figure 29: contactRating.

5.4.9 outputInFile

The last function called, concluding the development-phase of this project, is the *outputInFile* function. This is also one of the longer functions in the product, and what it does is to great extent already explained in the name itself; It outputs the data found in the text files and stored in the structs, in a file named *shortlist.txt*. This is what users of the product are going to find useful. The function takes the integer, *totalKeywordCount* and the struct array as parameters. The first step in the function is to open a file named *shortlist.txt* in write mode. This is done to create a completely empty file every time the product is run, and is necessary if the software is used multiple times. When the file has been opened in write mode and in this fashion created, the file is closed again. This is done to remove the previous data, which could be left over from previous operations. Next, the file is opened in append mode. The same algorithm to calculate the ratings of the text files in terms of keywords, described in the *contactFinder* section, is run for all

keywords entered by the user at the start of the program. This is done to calculate the highest achievable score if all keywords are matched to a file, and both email and phone number are identified. A snippet of the for-loop calculating *maxPoints* can be seen in the following figure:

```
for (int i = 0; i < totalKeywordCount; i++)
{
    maxPoints += 1;
    if (i < 8)
    {
        maxPoints += 10 - i;
    }
    else
    {
        maxPoints += 2;
    }
}
printf("\nPlease enter minimum score for application files (Max score is: %d):\nWrite here: ", maxPoints);
scanf("%d", &minPoints);
printf("\n");
```

Figure 30: outputInFile - calculating maximum score.

The user is now asked to enter a minimum score for the application files and is, for the purpose of giving context, presented with the highest achievable score. Only text files with a score equal or above the entered minimum score will be eligible to be output in the *shortlist.txt* file. The sorting algorithm *qsort* is now used in cooperation with the custom comparison function named *qsortComparison*, to sort the struct-array from highest to lowest rating. The comparison function straightforwardly returns one, if the rating of text one is bigger than text two, and returns negative one if the opposite is the case. If none of mentioned instances are the case, zero is returned.

The following part of the function is consisting of various methods of modifying strings via *strcpy* and *strcat*, as well as some outputting in the char array, *textToPrint*, done by *snprint*. A for-loop iterating each struct-array is used to output the contents of each text file, one at a time. To iterate through the structs, the for-loop uses the integer *i*, defined to have the value of *TEXT_NUM* at the initialisation of the loop. For every iteration *i* is decremented by one. In the loop is an if-statement, parenting all of the string modifying and outputting. The statement is true if the *rating* member of the struct-array at the index of *i* is larger or equal to *minPoints* - the integer storing the user-defined minimum value of the minimum scoring threshold of the applications. This ensures, that the contents of text files with a score below the minimum, is not outputted into the shortlist file. At the starting lines of the for-loop, within the if-statement, the char-array, *textToPrint* with the length of the symbolical constant *PRINT_LENGTH* is defined. *snprintf* is used to output *filename*, *rating*, *maxPoints* and *keywordCounter* into the char array in a readable fashion. Below, a snippet shows the for-loop iterating each element in the struct-array containing the contents of the text files and copying file name, rating and amount of keywords into *textToPrint*. The for-loop described below the snippet, copying phone numbers the string to print is also shown:

```

for (int i = TEXT_NUM; i > 0; i--)
{
    //textfiles's rating must at least be equal to minPoints
    if (texts[i].rating >= minPoints)
    {
        // Create an empty string
        char textToPrint[PRINT_LENGTH];
        snprintf(textToPrint, PRINT_LENGTH, "\nFile Name: %s\nRating: %d / %d\nUnique Keywords: %d\n",
            texts[i].filename, texts[i].rating, maxPoints, texts[i].keywordCounter);

        // Add phone numbers to text to print
        char phoneTemp[50];
        for (int j = 0; j < 4; j++)
        {
            strcpy(phoneTemp, "");
            if (texts[i].phone[j] > 0)
            {
                snprintf(phoneTemp, 50, "Phone no. %d: %ld\n", j + 1, texts[i].phone[j]);
                strcat(textToPrint, phoneTemp);
            }
        }
    }
}

```

Figure 31: outputInFile - For-loop for iterating over files, and copying phone numbers into textToPrint.

Next, a for-loop is used to place each phone number located accompanied with its index into a temporary char array named *phoneTemp*. The phone numbers are only be copied into *phoneTemp* if they are bigger than zero. *snprint* converts the long int, *phone*, and the integer *j* used for indexing, into chars and copies them into *phoneTemp*. This is crucial, as *strcat* on the next line copies *phoneTemp* into *textToPrint*, and only is capable of handling strings. Another for-loop executes the same procedure for the emails, though using *emailTemp* as the temporary file to copy into *textToPrint*. The char arrays, as mentioned, copy the strings into *textToPrint* via *strcat*, which inserts the text at the end of a string. A following for-loop, also via *strcat*, copies the keywords, with their sentence underneath. All of the loops use the earlier mentioned *i* as index in the struct array, and *j* as index in the array of *phone numbers*, *emails*, *keywords* and *sentences*. The last function call in the for-loop is *fputs*, which copies the *textToPrint* char array into *shortlist.txt*. *textToPrint* only contains the contents of one file at a time, which is why the for-loop is ran *TEXT_NUM* times. Lastly, the file is closed and a success-message is printed. Image showing the process of copying sentences and keywords into *textToPrint*, as well as outputting *textToPrint* into *shortlist.txt* is shown below:

```

for (int j = 0; j < texts[i].sentenceCounter; j++)
{
    strcat(textToPrint, "- ");
    strcat(textToPrint, texts[i].keywords[j]);
    strcat(textToPrint, ":\n");

    texts[i].sentences[j][strlen(texts[i].sentences[j]) - 1] = '.';
    texts[i].sentences[j][strlen(texts[i].sentences[j])] = '\0';

    strcat(textToPrint, texts[i].sentences[j]);
    strcat(textToPrint, "\n\n");
}
textToPrint[strlen(textToPrint) - 1] = '\0';
strcat(textToPrint, "-----\n");
fputs(textToPrint, shortlist);

```

Figure 32: outputInFile - copying keywords and sentences to textToPrint, and outputting textToPrint in shortlist.txt.

5.4.10 validation

A short function, which is used throughout the entirety of the product whenever a file is opened, is the *validation* function. It takes a file pointer as input parameter, and examines if the file is equal to *NULL*. If the condition statement is evaluated to be true, an error message is printed and the *exit* function is called. As mentioned, considerations and choices made afterwards and during the product development will be discussed in the following sections.

```
// Checks if a file exists
void validation(FILE *file)
{
    if (file == NULL)
    {
        // File not located hence exit
        printf("Unable to locate file.\n");
        exit(EXIT_FAILURE);
    }
}
```

Figure 33: Validation - checks if a file exists.

6 Discussion of the product

This section encapsulates the discussion of the final product in relation to the product requirement specifications, the limitations the group has faced, and the process of testing. Additionally, the future of the program solution will also be elaborated on.

6.1 The final product compared to requirements

The final product resulted in successfully fulfilling all requirements in MoSCoW's "Must Have" and "Should Have" categories, but unfortunately very few of the features shown in "Could Have" and "Wont Have" sections were implemented at last. Considerations were made by the group about which features were most important as formerly displayed in the [Requirement specifications section](#). As explained in the [Design section](#), the "Must-have" routines were developed at first, as they were the most crucial parts for making the software usable. "Should Have" features were made next and most of these were accomplished. The "Could haves" were barely achieved, but still contained some features that would be necessary, should the program be published for use.

6.1.1 MacOS and Windows builds

Throughout the development process, the group had experienced a rather large number of situations, in which hardware-limitations lead to less-than-optimal solutions, or entirely cutting an intended routine. At last, the group decided to split the program up into two builds, one for MacOS, and one for Windows - the only major difference being the inclusion of the "strcasestr" function. In short, the "strcasestr"-function, only works on MacOS due to a missing system-package on Windows, though the parent function "strstr", is available on both MacOS and Windows operating systems. "Strcasestr" is a function used when scanning for substrings. In the group's product solution, it is used for scanning keywords without case-sensitivity and determining in which sentences various keywords exist. Sadly, the inclusion of "strstr" in the Windows-build largely kneecaps one of the main selling points of the product, which is being able to identify words despite extra characters, suffixes, prefixes, etc. Discussions were held about whether or not a workaround should be made making it work on both MacOS and Windows, but ultimately it was decided that the software would only be fully optimised for MacOS, as a solution would require too much time to implement. Naturally, this would not be the decision made by the group if the software was supposed to be published for common use, and was mainly based on the time span remaining for completion of the project, as various other features of the program was.

6.1.2 File processing

The finished product of this project will, as mentioned in previous sections, only be able to take .txt files as input files and the files must have a specific name format, namely "text" followed by the index of the file in context to other files. An example could be: "text1.txt" or "text2.txt". The number of text files to be imported has not been made dynamic either, meaning that the number of files, must be defined in the source-code of the software beforehand. Mentioned features are both seen in the product's MoSCoW model, in the "Wont Have"-section. The reason why, is

touched upon in the [Requirement specifications section](#) of the report. It was decided not to implement these features, because it was difficult and time consuming, and the abilities of the developers were limited, as well as the time given. For future work, these features would be implemented, as it would be crucial for employers to work with different file formats. It is debatable whether or not the group, had they been able to have a second attempt at the project, should have prioritised the features higher, because they, as mentioned, would make or break a job recruiters' desire to implement the software in their hiring processes.

6.1.3 Tests

Due to limited amount of time, no satisfactory user-focused tests of the software was performed. The group discussed if it was important to include testing in the future. Since the program is to a large extent based on the customers' feedback and wishes, the group concluded that for the future, it is crucial to include user-testing. These tests should have been done in collaboration with people who could act representative for our demography, see our persona for instance. During these tests, they would have been presented with the software and asked to perform specific actions, for example search for the keywords "programming" and "Python" in all given texts, or find out which of the given motivational letters is the best with their own set of sequentially prioritised keywords. While the individuals attempted to solve their assignments, their interaction, comments, and non-verbal reactions would be monitored, and be collectively analysed for future development of the program. This routine would have been done within the additional guidelines of the "think-audibly test" [26], which is a method commonly used when testing websites, user-interfaces, and a wide range of other digital solutions.

6.2 Future work on the product

If the project was to be worked on in the future, it could be relevant to conduct a marketing analysis. Briefly, during a marketing analysis, a wide range of tools are used to analyse how much potential a given company's upcoming product is expected to have on the respective market. In this case, the marketing analysis could for instance include the model: Porters Five Forces framework. This model has its main focus on visualising the five most decisive elements of a products competitive environment. They are categorised as "buyer bargain power", "threat of substitution", "supplier bargain power", "threat of new entrants", and "competitive rivalry".

From Porters Five Forces Framework, in context with this product, it would be most relevant to make a deeper analysis based on the "buyer bargain power" and predominantly "competitive rivalry" and "threat of substitution products". These forces are important to analyse and understand, and has been chosen due to the following rationality:

The product and marketing has to be shaped towards what the markets buyers want, therefore the buyers has the most power of the market. The competitive rivalry is the second most controlling force on this market, if the competitors service and pricing is better, the buyers are more likely to choose them. The threat of substitution is the third most controlling force, because there are other solutions to our problem than a software. These other solutions can take buyers away from the market, and therefore has to be taking in consideration as well. With this understanding of the market, one can take that in consideration with the development and marketing of

the product, so it will be receive as best as possible. That is if the program were to be deployed to the market.

6.2.1 Other considerations

This subsection discusses what other considerations, the groups had for this product in the future, that were not mentioned in "The final product compared to requirements" or the "MoSCoW" sections.

Firstly, during the very first iterations of the program, there were numerous (seemingly random) integer parameters as function-parameters instead of symbolic constants. The program was edited in the last phase to have symbolic constants instead of these variables. For future work, it would be beneficial to start with making symbolic constants to save the editing time, and it would not effect the program as heavily as it did in this project.

In the same matter as using symbolic constants, the group learned about the "malloc" function in the later part of the imperative programming course. Since the program is so thoroughly based on string-arrays, it would definitely make the process of creating the program a lot faster to use "malloc" from the start. It was also an aspect, that made the editing time slower than it possibly could have been.

Another consideration the group had was allowing more than twenty keywords at a time, and conjugation of the keywords. The program, it's final iteration, can contain up to twenty keywords with each of them having a maximum length of fifty characters. The reason it is set to those numbers are, as mentioned in the [Design section](#), that it is unlikely for the user to input more than twenty words that each are longer than fifty characters. Though, if it were to happen, then the program would run into an error. For future development, rewriting/optimising some of the code to tweak the limit on keywords user-defined could be advantageous for escaping that kind of error. In continuation, being able to write a keyword in one conjugation and only searching for that tense is also limiting. If the program was to be worked further on, then making a function that would automatically conjugate the given keywords in every tense and search for them, could be ideal. On the grounds of that, being more accommodating to different conjugations could catch good candidates that used keywords, just in a different tense than the user inputs tense. Some could, however argue that when recruitment professionals work with keywords, they only tend to use about 5-10, so in the majority of cases, the user should not run into the problem of using an illegal amount of keywords.

Other considerations after looking at the finished product and evaluating the quality of the code, a couple of ways to improve the software by simplifying and merging the code were discovered. The code is working, but with some improvements it could run even faster, and the complexity of the functions could be reduced, rendering it more efficient. As mentioned in the [Implementation section](#) for instance, while-loops could have been merged in the *contactFinder* function. The *keywordFinder* and *sentenceFinder* functions would not both be necessary either if the some of the mentioned variables and loops had been moved around - this is described more thoroughly in the [Implementation section](#). The root cause of why the mentioned discoveries have not been acted upon is, once again, the factor of time. These aspects would definitely be worked on if future work on the project would come into play. Nonetheless, the software works as intended, a the small final adjustments and patches on the software were not the highest priority.

Lastly, it was also considered to continue working with another programming language if the program was to be further developed. Although it was a requirement for the software solution of the project be written in C, Python would make a lot more sense for the group to work with in the future. That is, because Python handles strings vastly differently than C does. In C-language, one needs to allocate memory in the sting arrays to have enough space for the stings/characters. That isn't needed in Python, and therefore it is also not needed to free the data inside of the arrays. It will save time in writing to program, which could be used on implementing more and better quality features and initiatives. Python also has several cross-platform libraries, which would solve a great deal of the current OS-related limitations.

Further reflection on the group's project organisation, programming experience, and cooperation can be read in the attached "Process analysis for P1 - Group 2, SW1"-document.

7 Conclusion

Overall, the group has created a software solution to the proposed problem statement, which was designed, and specified to fulfil a real world problem that recruitment specialists face daily. To ensure quality and relevancy - expert-opinions were incorporated, extensive online research was conducted, and countless theoretical, and implementation ideas were thoroughly discussed and tested. This process successfully utilised all aspects of the group's current coding-competencies, and project-organising capabilities. Amongst numerous programming-related models and methods, flowcharts/diagrams, and the construction of a UML Sequence diagram was proven to be most fitting with the group's current implementation framework. The approach of top-down programming in combination with an iterative workflow strengthened the group's general ability to fragment the problem into smaller bites, and solve the individual sub-problems in smaller groups. This led to a smooth cycle of defining and implementing new ideas.

The product acts as a commendable contribution to a field, in which human resources are still abundantly consumed by a long list of monotonous activities, that at the end of the day could, and eventually will be automatised by software. It is, however, essential to understand that the employed software, at the end of the day, still enriches a human experience, rather than taking the human element out of a job. Human resources is a field that works intimately with other people so rather than taking over a section of work, the group has conceived a product, that provides new possibilities, and expands the degree of proficiency for a recruitment specialist. In the meanwhile, the problem statement was answered, as the product successfully streamlines a line of operations in the field of recruitment, provides valuable data in an easily digestible and compact format, and makes it easier for the employer to identify the most optimal candidates for jobs with specific requirements. It is, however essential to express, that the product presented is merely an early prototype, which would in reality only represent a snippet of the full-scale solution. Nevertheless, even as an early prototype, the program already possesses a variety of routines and ideas, which none of the competing software solutions do, due to the developer-team working so close with the people who experience the problem themselves. Thus, the problem statement is answered, and the first semester-project of the group is concluded.

8 Figures

List of Figures

1	The phases of SDLC [22].	8
2	The Iterative Incremental Model [25].	9
3	Structure Chart for function contactFinder.	10
4	Salience Model Diagram [16].	19
5	Response to Question 1 in the survey.	27
6	Response to Question 2 in the survey.	27
7	Response to Question 2.1 in the survey.	28
8	Response to Question 3 in the survey.	28
9	Response to Question 4 in the survey.	29
10	Response to Question 5 in the survey.	29
11	The constructed persona, Peter Johansen. This picture was created by an AI [28], thus does not represent a real individual, nor is a subject to copyright.	31
12	The MoSCoW method.	32
13	Flow-diagram for the main segments of the program.	38
14	UML Sequence Diagram to the finished product.	40
15	Flowchart showing the keywordFinder function.	41
16	Pseudo code for the readFileKeywords function.	41
17	Route-diagram to the systematic implementation of ideas.	42
18	The user-defined keywords are received via the terminal in a sequen- tial priority.	45
19	The user-defined keywords are received via the terminal in a sequen- tial priority.	45
20	The top-candidate from the text-file generated by the software.	46
21	Data structure used in product.	48
22	malloc - Dynamical data allocation.	49
23	makeFileKeywords.	49
24	makeFileWithLineBreaks.	50
25	contactFinder - identifying phone.	52
26	contactFinder - indentifying mail.	52
27	keywordFinder.	53
28	sentenceFinder - while-loop.	54
29	contactRating.	55
30	outputInFile - calculating maximum score.	56
31	outputInFile - For-loop for iterating over files, and copying phone numbers into textToPrint.	57
32	outputInFile - copying keywords and sentences to textToPrint, and outputting textToPrint in shortlist.txt.	57
33	Validation - checks if a file exists.	58

9 References

References

- [1] Linda Ahrenkiel. *Mixed methods*. 2020. URL: <https://laeremiddel.dk/viden-og-vaerktoej/videnskabsteori/metoder/mixed-methods/>. (accessed: 11.11.2021).
- [2] *Camel Case*. 2021. URL: https://en.wikipedia.org/wiki/Camel_case. (accessed: 12.11.2021).
- [3] Andrew Fennel. *Andrew Fennel*. 2021. URL: <https://uk.linkedin.com/in/andrew-fennell-67520332>. (accessed: 12.11.2021).
- [4] Andrew Fennell. *How long do recruiters spend looking at your CV?* 2021. URL: <https://standout-cv.com/how-long-recruiters-spend-looking-at-cv>. (accessed: 12.11.2021).
- [5] GeeksforGeeks. *Functional vs Non Functional Requirements*. 2021. URL: <https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/>. (accessed: 15.12.2021).
- [6] Jannes Innes. *The CV Book 2nd edn: Your definitive guide to writing the perfect CV*. Pearson, UK, 2012, p. 304. ISBN: 9780273776611.
- [7] Elliot B. Koffman Jeri R. Hanly. *Problem Solving and Program Design in C*. Pearson, 2016, p. 144. ISBN: 978-1-292-09881-4.
- [8] Elliot B. Koffman Jeri R. Hanly. *Problem Solving and Program Design in C*. Pearson, 2016, p. 169. ISBN: 978-1-292-09881-4.
- [9] Swatee B. Kulkarni. *Intelligent Software Tools for Recruiting*. 2021. URL: <https://scholarworks.lib.csusb.edu/cgi/viewcontent.cgi?article=1398%5C&context=jitim>. (accessed: 12.11.2021).
- [10] lucidchart.com/. *What is a Flowchart*. 2021. URL: <https://www.lucidchart.com/pages/what-is-a-flowchart-tutorial>. (accessed: 12.11.2021).
- [11] David Meshulam. *Beat Oracle's Taleo Applicant Tracking System (ATS)*. 2021. URL: <https://www.jobtestprep.com/taleo-applicant-tracking-system>. (accessed: 17.11.2021).
- [12] Sara A. Metwalli. *Pseudo-code 101: An Introduction to Writing Good Pseudo-code*. 2021. URL: <https://towardsdatascience.com/pseudocode-101-an-introduction-to-writing-good-pseudocode-1331cb855be7>. (accessed: 17.11.2021).
- [13] *Modern Applicant Tracking Software*. 2021. URL: <https://www.freshworks.com/hrms/features/applicant-tracking/>. (accessed: 17.11.2021).
- [14] Oracle. *Oracle Taleo Cloud Service Global Price List*. 2018. URL: <https://www.oracle.com/us/corporate/pricing/taleo-price-list-2949065.pdf>. (accessed: 11.12.2021).
- [15] Visual Paradigm. *What is Unified Modeling Language (UML)?* 2021. URL: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/>. (accessed: 15.12.2021).
- [16] pmStudyCircle. *Salience Model to Analyze Project Stakeholders*. 2015. URL: <https://pmstudycircle.com/salience-model-to-analyze-project-stakeholders/>. (accessed: 05.12.2021).

- [17] ProductPlan. *What is MoSCoW Prioritisation*. 2021. URL: <https://www.productplan.com/glossary/moscow-prioritization/>. (accessed: 08.12.2021).
- [18] Schema. *Person*. 2021. URL: <https://schema.org/Person>. (accessed: 12.11.2021).
- [19] Ulrich Schild. *How HR & Recruiters Actually Read Your Resume*. 2021. URL: <https://www.linkedin.com/pulse/how-i-actually-read-your-resume-ulrich-schild/>. (accessed: 12.11.2021).
- [20] *Systematic literature search*. 2021. URL: <https://library.au.dk/forskere/systematisklitteratursoegning>. (accessed: 11.11.2021).
- [21] Indeed Editorial Team. *How To Find Keywords in Job Descriptions and Use Them in Your Resume*. 2021. URL: <https://www.indeed.com/career-advice/resumes-cover-letters/finding-keywords-in-job-descriptions>. (accessed: 12.11.2021).
- [22] Svita Team. *SDLC Methodologies*. 2019. URL: <https://svitla.com/blog/sdlc-methodologies>. (accessed: 11.11.2021).
- [23] Jette Egelund Holgaard Thomas Ryberg Nikolaj Stegeager Diana Stentoft Anja Overgaard Thomassen. *Problembaseret læring og projektarbejde ved de videregående uddannelser*. Samfundslitteratur, 2014, pp. 62–63. ISBN: 978-87-593-1878-2.
- [24] *Transform hiring with our Corporate HR Software*. 2021. URL: <https://www.zoho.com/recruit/corporate-hr-software.html>. (accessed: 17.11.2021).
- [25] Tutorialspoint. *SDLC - Iterative Incremental Model*. 2015. URL: https://www.tutorialspoint.com/adaptive_software_development/sdlc_iterative_incremental_model.htm. (accessed: 11.11.2021).
- [26] UserDesign.dk. *Tænke-højt test*. 2021. URL: <https://www.userdesign.dk/usability-test/taenke-hojt-test/>. (accessed: 09.12.2021).
- [27] Vervoe. *Hire great people, every single time*. 2021. URL: <https://vervoe.com/?ssrid=ssr>. (accessed: 12.11.2021).
- [28] Phillip Wang. *this person does not exist*. 2019. URL: <https://thispersondoesnotexist.com/>. (accessed: 16.12.2021).
- [29] Lauren Weber. *Your Resume vs. Oblivion*. 2012. URL: <https://www.wsj.com/articles/SB10001424052970204624204577178941034941330>. (accessed: 17.11.2021).
- [30] *What is ATS Software?* 2021. URL: <https://matchr.com/ats-software/what-is/>. (accessed: 17.11.2021).