Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Maurice Hostettler & Lukas Boner

Digitalization of Chess Score Cards

Semester Thesis

Centre for Artificial Intelligence
Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

Supervision

Mark Cieliebak
Pius von Däniken

December 2022

Abstract

Chessreader is a web application for the digitization of handwritten chess score cards. There have been several works at ZHAW that have developed and implemented this application over the last two years. The goal of our Semester thesis was to to decouple the front and backend of this application, design a clean UI and improve the user experience. We used Angular as a framework for our changes.

The application in its current for uses the OCR services from Microsoft, Azure and Google to analyze the uploaded image to extract the handwritten text from it. The recognised moves are then displayed to the user, so corrections can be made if there are any mistakes.

Zürcher Hochschule für Angewandte Wissenschaften

School of Engineering

DECLARATION OF ORIGINALITY

Semester Thesis at the School of Engineering

DECLARATION OF ORIGINALITY
Semester Thesis at the School of Engineering

By submitting this Semester thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Semester thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date:	**Name Student:**
Winterthur, 23.12.22	Lukas Boner
New York, 23.12.22	Maurice Hostettler

Contents

Chapter 1

Introduction

1.1 Description of the project

The earliest form of Chess can be traced back to India in the sixth century and today, more than a millennium later it is still one of the most popular games around the globe. Chess is being played on smartphones, tablets, computers and of course on chessboards.

As we are currently in a transition from a paper-based society to a digital one, so is Chess. It is played digitally more and more and big games are streamed on video platforms for the whole world to see. But many tournaments playing on actual chessboards still keep track of the individual moves of a game on a piece of paper. These scorecards then need to be digitized manually in order to catalogue and preserve them.

Chessreader is meant to solve that problem. The goal is to be able to take a picture of a score sheet and through Optical Character Recognition, being presented with the game in digital form, thus making manual transcribing obsolete. This is no easy task. While chess notation is more or less standardized, the scoresheets themselves are not. This means that Chessreader needs to be able to handle vastly different layouts while still being quick and reliable.

The goal of our thesis was to improve the usability and maintainability of the Chessreader project code. This means that we're starting to decouple the frontend and backend, switching to a industry standart as our framework and documenting our code rigorously while obeying the rules of clean code. While the goal was to be able to use the new frontend with the old application standing in as a backend replacement, we also kept in mind that eventually a proper REST-API is the goal, so the interface needs to be flexible enough to allow switching to a different API without major rewrites. Our starting point were multiple previous works that were all written at the ZHAW Centre for Artificial Intelligence since 2020.

While the original scope of the project intended that some additional functionality would be added to the program, we decided to instead focus our efforts on rewriting the front end. This was done to achieve a code base that was more manageable and maintainable so that future programmers would have an easier time developing the project further. This means to deal with a lot of code that has grown more or less organically and a project that multiple different groups of developers had attached their work onto. We'll summarize some of those previous works that have been written to give an idea of the history of the project and some insight into the code base.

1.2 Previous works on the topic

This thesis is based on several other bachelor and semester theses written at ZHAW. Other articles and projects have also attempted to implement an application in a similar function. We will try to give an overview of these previous works and projects for context, however a more detailed account can be found in those papers directly. By no means does this overview completely cover the progress made in each work.

1.2.1 ZHAW projects

Dreher, Horvath Bachelor thesis

The original project description and implementation came from the Dreher, Horvath thesis, which used ABBYY as a OCR tool to recognize the moves. The backend was written in Python with a Flask web server[1].

Abduli Semester thesis

The Abduli thesis improves the Preprocessing of the images and repairs broken characters, leading to an overall improvement in detection accuracy[2].

Caglayan, Nielsen Bachelor thesis

Caglayan and Nielsen introduce different OCR tools, namely Google Vision, Azure Cognitive Services and Amazon Rekognition. They also analyse the time savings of the tool compared to traditional methods of digitizing the scoresheets[3].

Ambrosini, Hellinger Bachelor thesis

Finally Ambrosini and Helliger introduce the current version of the Frontend, including dark mode. They also extend the functionality to be able to take pictures with a webcam and directly use them. They also suggest reworking the Front end with a JavaScript based framework, decoupling the back end from the front end and cleaning up the code in the backend[4].

1.2.2 Other projects

There have been a few attempts at solving the issue of digitizing chess moves. The earliest project we could find dates back to a medium article from 2017 where Marek Śmigielski outlines his attempt at automating the digitization of chess scorecards[5]. Śmigielski described a process of preparing the scoresheet for OCR, however he abandoned the project in 2018 and instead refered interested Medium readers to Rithwik Sudharsan's Reine project [6]. The Reine project, uses a very similar approach to us, however they concluded that it was too difficult to try and analyze any score sheet, due to the vast variety of available layouts. In order to solve this they created a very specific score sheet, where every single letter is separated into its own box, which makes the Character Recognition a lot easier. Their approach used OpenCV and TensorFlow to perform the Character Recognition. For this to work flawlessly, the tournament organizers would have to agree to use these specific score sheets, or at least a similar score sheet.

Another Project is the CheSScan app for IOS and Android[7]. This is a mobile application and since its a paid service, their source code is not freely available. However testing their application, we saw that the accuracy of their recognized moves is not good enough to be all that useful. After every scan, checking every recognized move, would take the same amount of time, or even longer than entering each move into a program by hand.

1.3 Functionality

Chessreader's main function is to process a picture of a Chess score sheet and using OCR recognizing the moves played in a game. This can be broken down into several smaller features that our web applications needs to fulfill this task. They can be broken down as follows:

1. Upload a new Score sheet
2. Select the last move of the game
3. Process the Score sheet
4. Review the processed Result
5. Download the PGN File

In order to help with processing of the score sheet, previous teams decided to also have the user select the last move of the game so that no empty boxes would be sent to the OCR tools. This is why there is an additional selection step added.

Since we decided that we were not going to change the back end functionality of the project all that much, we kept the basic structure of the workflow the same as can be seen in figure 1.1).

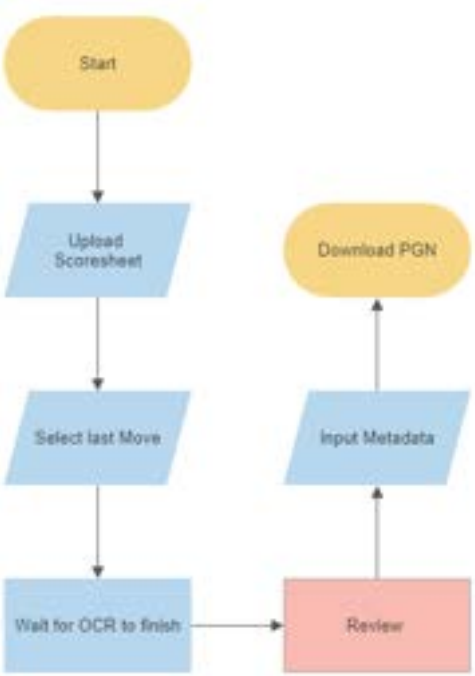

Figure 1.1: Flowchart of the process a typical user goes through. The review step is by far the most time consuming one.

(a) Screen representing the Upload Scoresheet Step

(b) Screen representing the Select Step

Figure 1.2: Screenshots taken at the start of our thesis, depicting the UI for Upload and Select Step

In figure 1.2 and figure1.3 We see the screens the user works with while using the application. In figure 1.2a they select the image they want to upload to the tool, by clicking on the green button. Then in figure 1.2b they select the last box that contains a move that was played in the game they want to digitize and in figure 1.3 they review all the moves that ChessReader detected and goes through them. For that they use the controls on the left.

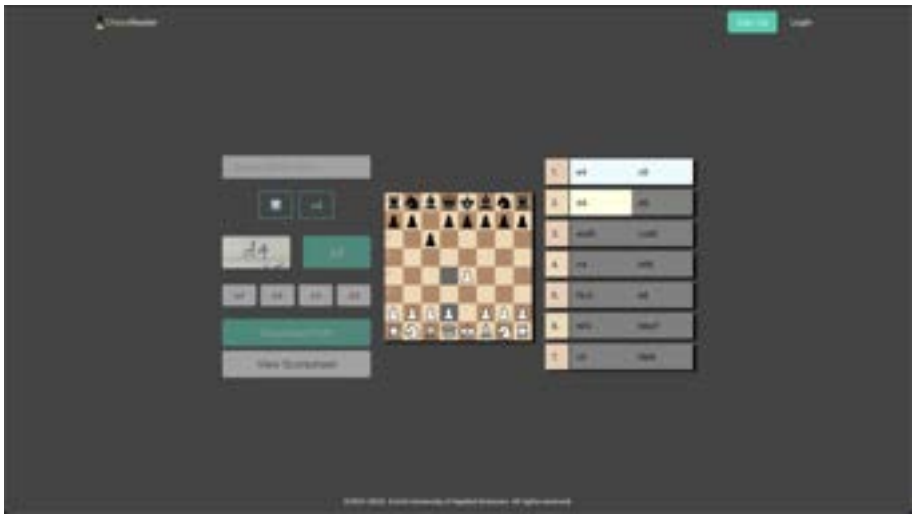

Figure 1.3: Screenshot of the UI depicting the review step of the process, taken before our changes

Chapter 2

Background

In this chapter we would like to give some deeper insight into the technical details of our project. These include the optical character recognition itself, as well as a look at what the input and output formats of our application are.

2.1 Optical Character Recognition

Optical Character Recognition is an overarching term containing several methods concerned with extracting text from images. For our application we are specifically interested in extracting handwriting from a picture or potentially a scan of a document. As can be seen in figure 2.1, even for a human it can be hard to do that. One of the best modern ways this gets done is by using different Machine Learning techniques. According to Shamim one of the best ways is the Multilayer Perceptron approach[8]. Since OCR tools rely on a tremendous amount of training data, we use commercial tools to achieve the best possible results.

In order to read the moves of a chess sheet, we first need to extract all and categorize where on the image the moves are written down. Luckily most chess score sheets contain a grid with clearly visible lines seperating the moves.

Preprocessing

Before the picture can be sent to an OCR tool, Chessreader corrects for skew and shadow present in the image. This helps with recognizing the boxes as well as the writing later on.

Box extraction

In a first step Chessreader extracts the position of these boxes by evaluating the output from one of the OCR tools. Then the output is saved to a dictionary that contains each move number and the box position on the page for said move.

Move extraction

In a second step, every individual box gets cut out of the image and sent to the OCR tools individually. Then the output gets adjusted, because we can exclude all results that don't represent valid chess notation.

Figure 2.1: Example of a filled out score sheet, from the famous game between Boris Spassky and Bobby Fischer at the 1970 chess Olympiad[5]

OCR tools used

In the original thesis Abbyy was used for both the extraction of both the boxes on the page as well as the reading of the moves from those boxes[1]. In later works Google Vision AI, Azure Cognitive Services and Amazon Rekogintion was introduced.

2.2 Chess score keeping

While on the worlds biggest stages there is a whole team of staff tracking every players move and replaying them for a live audience on electronic boards, in most tournaments chess players write down their moves on score sheets. This is done for several reasons, most importantly it is done so that a referee would be able to trace a game back in case of an interruption, but it is also done for official record keeping of said games. After these games score sheets are turned in to tournament organizers which either scan them for safekeeping or in some cases manually replay all the moves in a game in order to create an digital record of that game as well, so that chess databases, fans, coaches and anyone else interested can replay and analyze the games in question. This can be a rather tedious task if it is done for big tournaments, or it will not be done at all at smaller tournaments where official record keeping is less of an issue. In those cases however the players themselves or these players coaches and teachers would find it useful to be able to replay and analyze these games as well, in order to learn from mistakes or find weaknesses in the opponents play.

While there are many different kinds of sheets they all are structured in a similar way (see 2.2). Every Move has a white and a black turn, and the players note down their own and their opponents

(a) Score sheet by the popular website Chess.com [9] (b) Official US Chess Federation score sheets[10]

Figure 2.2: Chess score sheets come in a variety of different shapes and sizes

turn. There are different chess notations that can be used, but by far the most common one is the Algebraic chess notation outlined in the official FIDE (Fédération Internationale des Échecs) Handbook[11]. An excerpt from the handbook explaining how the english version works can be found in Appendix B.1.

2.3 Portable Game Notation

The final product of our program is going to be a .pgn file, better known as Portable Game Notation. It was designed to make exchanging games digitally simpler[12]. It works by listing tag pairs in brackets for all kinds of Metadata around the game, like the location and the players, followed by a "movetext" describing the game.

```
1   [Event "F/S Return Match"]
2   [Site "Belgrade, Serbia JUG"]
3   [Date "1992.11.04"]
4   [Round "29"]
5   [White "Fischer, Robert J."]
6   [Black "Spassky, Boris V."]
7   [Result "1/2-1/2"]
8
9   1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 {This opening is called the Ruy Lopez.}
10  4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11  11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5
12  Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
13  23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
14  hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5
15  35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
16  Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

Figure 2.3: An Example of a .pgn file containing a different game between Spassky and Fischer played in 1992

Using this filetype allows users to export their detected games into all different kinds of chess databases and analysis tools as all of them will support this standard.

Chapter 3

Architecture and Design

3.1 User Interface Design

Since our team didn't include anyone that was well versed in UI Design we chose to rely on some easy to use Wire-framing tools to help us. In this section we quickly describe our thought process behind the decisions we made for each stage of the UI design process.

3.1.1 Interacting with the application

For the first 3 steps in the workflow, namely upload, select and process we kept a lot of the design elements from the previous works. We did however add a sidebar to our project that would make it easier to navigate through the process as well as track where in the workflow the user currently is.

3.1.2 Review Page

By far the most time and changes went into updating the Review Page. This is because most of the time spent when using the program is checking if the OCR made any mistakes and identified moves wrongly. So naturally the largest time saving would come from making this process quicker. While part of that certainly is to improve the OCR itself, we felt the application would also profit from a more intuitive and easier to use Review process.

In the previous version of Chessreader (see figure 3.1) all of the control over the program was on the left hand side of the screen. Leaving the center and the right hand side to display the information in form of a digital chess score sheet and an animated chessboard that would move the pieces as the program progresses through the moves.

In our first iteration we decided to change this since we felt it was important that the most central information to make a decision about a moves correctness should be located at the center of the screen. We also made the decision to offer two different methods to review the detected moves. One for players and one for organizers or data entry workers.

Player Mode

In figure 3.2 we can see our idea of the player version of the review page. In this mode we assume that the player who played the game themselves is using this application. Therefore we assume that they are familiar with the game played, as most chess players that are well versed at the game, will recall what moves they played in a given game[13]. However we still felt it would be useful for newer player to have access to the score sheet as it was originally uploaded so we display it as an optional window on the left side. There is however an option to close it if the user finds it too

Figure 3.1: Original Version of Chessreader UI

Figure 3.2: This is our first version of the UI, created with MockFlow

distracting. The idea here is that board displays the current position as well as the detected next move. It will mark the next move on the board and either moving the piece to that position or hitting the next button in the game controls in the bottom right will confirm that the move has been correctly detected. Making a different move on the board will correct the detected move, to the new one.

We kept the games digital score sheet on the left side of the screen as it is common that way on many of chess's online platforms, as seen in figure 3.3 and thus most players will be used to that layout.

Data Entry Mode

The second use case is that of tournament organizers or the staff that help run such an event. Here we assume that a user of our application doesn't have much knowledge of the games played or even chess itself and such we need a different approach for reviewing the OCR result.

(a) Example of the UI from the Website Chess.com (b) Example of the UI from the website Lichess.org

Figure 3.3: We want to design our player mode so that looks familiar to some of the most popular chess programs already used by players.

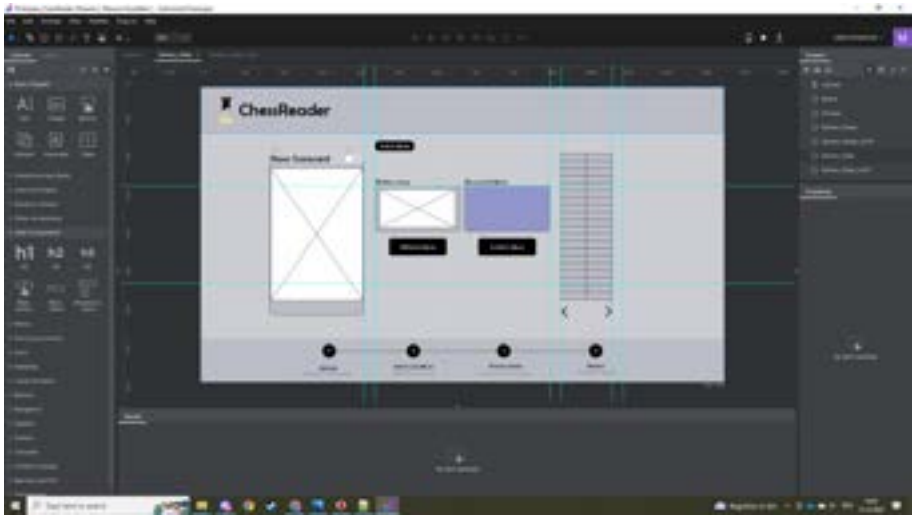

Figure 3.4: This is our second version of the UI, created with Justinmind

As seen in figure 3.4 the layout here is based on the same principle that the most important information is centered on the screen. However in this mode, the most important information is what the player wrote down and what the OCR detected. In order to display that, we put 2 equally sized boxes in the middle of the screen with the left hand side containing an enlarged cutout of the score sheet where this move was recorded and the right hand side contains a big display of what the OCR detected. Underneath there are 2 buttons to either confirm the result the OCR detected or enter a different move in case of a mistake. We kept the original score sheet as an optional display on the left, because there are errors with box detection and then the user will have to refer to the original sheet to see if the move was written somewhere outside the box intended for it.

3.2 Frontend Framework

Reworking the frontend was one of the key aspects of this project. So we essentially had the choice to build off what was already there from previous years, or to start from scratch. We decided to do the latter. The main reason for this was that the older version's frontend was not properly separated from the backend. Instead certain API calls would return HTML code that then was displayed and made interactive using native JavaScript. While there is nothing inherently wrong with this approach, modern design principles tell us to instead decouple the front and the backend completely[14]. Not only will this bring some improvement in maintainability of the code, it will also allow us to define a clear API which will help us when we implement more security features.

Therefore we decided to rewrite the frontend completely. This meant we needed to settle on a modern frontend Framework to get this project done in time. The three most popular frameworks for this are Angular, React and Vue.js.

3.2.1 Why Angular

Angular is developed by a team at Google, meaning there are a lot of resources both for learning how to use the framework as well as ready to use components that we could use as building blocks. There is also a good amount of prior knowledge of the framework already in our team.

Angular uses Typescript instead of JavaScript, which would also be an advantage for us since coding in Typescript seemed a lot more comfortable than coding in JavaScript. However all JavaScript code is also valid Typescript code, so there wouldn't be any compatibility issues when we would eventually integrate it to the already running parts of Chessreader.

Another advantage offered by Angular are Services. Since we needed to integrate the backend without having a clear API definition or a clear decoupling from the frontend, we needed to have some middle layer between the frontend application and the already written backend. This is where services come in. A service in Angular is a class that handles the logic of a frontend. Angular components themselves are not supposed to do any processing of data, they should only do basic input validation and then pass it on to a service. So using a service we would be able to neatly wrap our backend and create the illusion of decoupling. This would save time later when the backend gets rewritten and actual decoupling is implemented.

We've also looked at the other frameworks which we'll talk about briefly here:

Like Angular, ReactJS is one of the biggest currently popular frameworks. It mainly distinguishes itself by it's backward compatibility and the built in option for server-side rendering. It aims to be "just a library" which results in a more minimalist framework which offers greater flexibility to the developer.

The other framework we considered initially is Vue.js. One of it's biggest advantages is its tiny size. When stored as a zipped file it's only 18KB big and is therefore quickly loaded by a browser when being accessed. This and other performance advantages result in a very slight improvement over other frameworks in loading and updating time of a web page. It's also said Vue.js has a very smooth learning curve, with only basic knowledge in HTML, CSS and JavaScript being required.

In the end we decided to use Angular. All three frameworks use reusable components, but only Angluar separates HTML from code by default, which improves readability. It's also the more fully-fleged solution, which does result in a steeper learning curve, but also means that we do not need to rely on many third party libraries. In our opinion these libraries would contribute greatly to the complexity of ReactJS and Vue.js, offsetting their initial advantage. What also speaks for Angular is the use of TypeScript instead of JavaScript. The typing used by TypeScript helps preventing errors while coding and allows for better structured and organized code.

3.2.2 Structure of our project

One of the challenges when taking over this project was to understand how previous teams had built their program. Part of that was understanding how the project was set up in detail. So in this thesis we want to specifically show how we planned our project and how its structured in addition to the documentation of the code itself to provide an easy starting point for later developers to add to the project.

As we can see in figure 3.5 we built the application by separating the different pages of our application from our core functionality. The pages then contain both all of our individual pages with our custom components specifically built for Chessreader as well as a shared folder for all the Angular standard components we used.

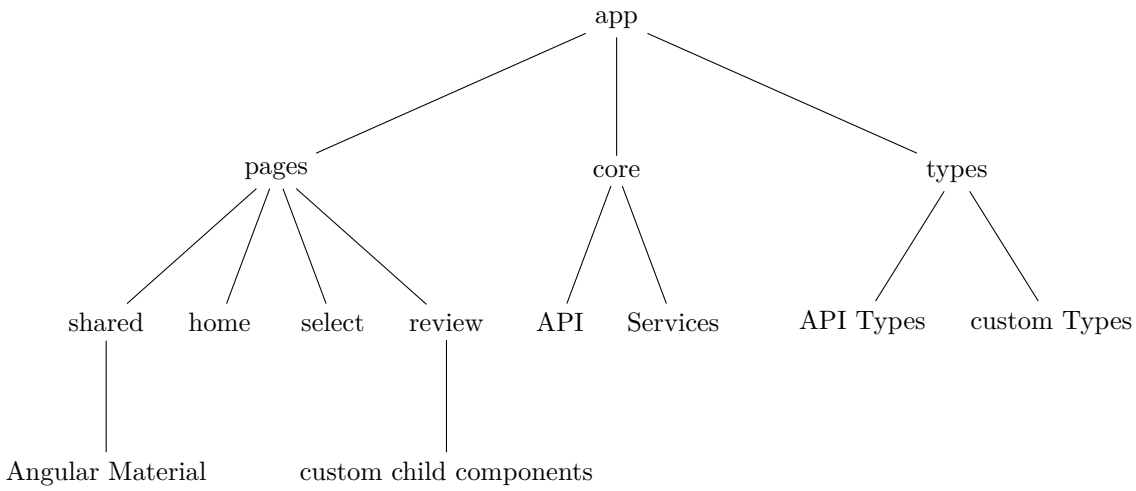

Figure 3.5: Graphical Representation of our project structure

Every component is built out of 4 different files:

- HTML file
- SCSS file
- Spec.ts file
- Typescript file

Roughly speaking the HTML file and the SCSS file are responsible for the look of the component while the Typescript file contains the functionality of each component and the Spec file is used to test the application using the Karma testing framework. This is all Angular standard and more information as to how exactly this works can be found in Angulars documentation[15].

3.2.3 Custom Angular Components

Since we needed some additional functionality that we were not going to get from the Angular Material Library, we created some of our own components. We we will quickly describe the purpose and functionality of each of them in this sub-chapter.

Clickable Scoreboard

This components function is to represent the final data after the OCR is completed, it displays a chess scoreboard where every turn that was read by the OCR is represented. A user is able to see all the prior moves made, the current move being reviewed as well as future moves the OCR has detected. A user can quickly switch between different positions in the game by clicking on the move they want to see.

In figure 3.6 we define a new class ChessMove containing a turn number and the moves of both players in their turn. Then we define an Array for all the moves represented in the game. This will fill our table accordingly with all the moves registered in the array.

In the HTML file seen in figure 3.7 we then add a click event for both players move in any turn.

```
export class ClickableScoreBoardComponent implements OnInit {
  @Input() TurnsDatasource: ChessTurn[] = []
  displayedColumns = ["turnNumber","whiteMove","blackMove"]
  .
  .
  .
  getMostLikely(move: ChessMove): string {
    let score: MoveScore = {}
    score[move.move.az] = 0
    score[move.move.gl] = 0
    score[move.move.gl2] = 0
    score[move.move.rk] = 0

    score[move.move.az] = score[move.move.az] + 1
    score[move.move.gl] = score[move.move.gl] + 1
    score[move.move.gl2] = score[move.move.gl2] + 1
    score[move.move.rk] = score[move.move.rk] + 1

    if(score[move.move.az] > 2) {
      return move.move.az
    } else if(score[move.move.rk] > 2) {
      return move.move.rk
    } else if(score[move.move.gl] > 2) {
      return move.move.gl
    } else {
      return move.move.gl2
    }
  }
}
```

Figure 3.6: Code Snippet from the Clickable Scoreboard Typescript file

```
<table mat-table class="chess-table mat-elevation-z8"
[dataSource]="TurnsDatasource">
    <ng-container matColumnDef="turnNumber">
        <th mat-header-cell *matHeaderCellDef>#</th>
        <td mat-cell *matCellDef="let chessMove">
            {{chessMove.turnNumber}}
        </td>
    </ng-container>
    <ng-container matColumnDef="whiteMove">
        <th mat-header-cell *matHeaderCellDef>White</th>
        <td mat-cell *matCellDef="let chessMove">
            {{getMostLikely(chessMove.whiteMove)}}
        </td>
    </ng-container>
    <ng-container matColumnDef="blackMove">
        <th mat-header-cell *matHeaderCellDef>Black</th>
        <td mat-cell *matCellDef="let chessMove">
            {{getMostLikely(chessMove.blackMove)}}
        </td>
    </ng-container>
    <tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
    <tr mat-row *matRowDef="let chessMove; columns:displayedColumns" (click
        )="handleClick(chessMove)"></tr>
</table>
```

Figure 3.7: Code Snippet from the Clickable Scoreboard HTML File

3.2.4 Services

Services are the parts of an Angular application that are responsible for the handling and moving of data. In contrast to components they have no visual elements and do not directly affect what the user sees.

ApiService

The ApiService is essentially our interface to the backend. It implements methods providing access to the different endpoints and acts as a translation layer between the Angular application and the old Chessreader application which is now only used as an API.

The structure of the service is for the most part very simple. A function making a POST-request would look something like this:

```
1         recognizeMoves(match_id: number, selected_boxes: RecognizeMovesRequest):
              Observable<OcrResponse> {
2             return this.http.post(this.baseUrl + `chess-match/${match_id}/moves/
                  recognize`, selected_boxes, {observe: 'response', responseType: '
                  text'})
3               .pipe(
4                 map( response => {
5                   let ocrResponse: OcrResponse = JSON.parse((this.ocrRegex.exec(
                        response.body as string)[1]))
6                   return ocrResponse
7                 })
8               )
9           }
```

Figure 3.8: Example of a function making a post-request and then returning the response body

Since we're relying on an application as an API that was not built for this purpose, things get a little bit more complicated. As some of the endpoints return HTML-code instead of JSON objects, we need to extract the relevant data out of the responses using regular expressions (often referred to as "regex"). A regex is an expression describing the structure of a string and can be used to select and isolate specific parts of this string. This is obviously a workaround, but allows us to use already existing parts of the old project.

This process can be seen in figure 3.8. The post request is made with an object containing the selected boxes as payload and the data is then extracted from the returned HTML code using the regex. The returned string is a so called "stringified JSON object" (just an object encoded as a string) which is turned back into an object using JSON.parse(...)

The big advantage of using such a service is that if the backend changes, only this service needs to be changed, instead of having to change every component that is somehow relying on interacting with the API.

MoveService

The move service is responsible for storing and providing the received dataset containing the recognized moves. Since it is not possible to transfer data between components directly and since we are currently limited by the endpoints that are provided by the old application we have to request the data via a service.

The select component is where the user selects the last box still containing a move transfers the list of all boxes to the service and then redirects the user to the next page. During this transition, the move service requests the API to analyze the boxes and stores the result in it's variables. Other

components and services (like the review component) can subscribe to changes and updates in this data.

```
1         private current$ = new BehaviorSubject<OcrResponse | null>(null)
2         public current = this.current$.pipe(
3             filter((it: OcrResponse | null): it is OcrResponse => it != null)
4         )
```

Figure 3.9: The variables which are responsible for storage and distribution of the data

The BehaviourSubject we see in figure 3.9 is a special type of an observable. In general, observables are values that are not yet locked down. One can subscribe to such an observable and receive the value once the function that has issued the observable has finished. BehaviourSubjects basically work the same, but provide the last set of values they had immediately once a new subscriber is added to them. This means in our case that even if the response of the API came in quicker than the redirect to the new page was made, the data will still be available for the review component to request.

3.3 Deployment

Like in previous iterations of the project the Angular application is deployed using docker. Docker has many advantages over just loading the files on a server and serving it this way to users. One of the biggest advantages is the easy deployment of the application on a new server. This means that if a part of the application proves to be a bottleneck, it could be relatively easily be scaled up by installing more instances. This would of course require some load balancing logic as well, but already having a containerized application makes this significantly easier.

The Chessreader project consists of multiple Docker containers. Currently there are four in use:

- The Angular application of Chessreader
- The database instance
- A nginx-webserver instance
- The old Chessreader application

Since we are currently still using the old Chessreader application as the backend, we need to keep this instance running. It, the database and a nginx-webserver instance are running on the development server in order to not interfere with the new Angular application. The deployment of these already existing containers was fairly straight forward, with only some minor fixes being required.

In order to be able to create the containers of the old project on the server, the old image needed to be reuploaded to dockerhub. For this a Chessreader-Account was created, so that future builds can be uploaded there as well and future issues with the docker-compose configuration are prevented.

The new Angular application is also running in a container on the production server. It is bundled with an nginx-webserver instance and accesses the old application when making backend calls. The docker-compose file sets everything up by itself so only a single command needs to be executed

Chapter 4

Discussion

In this chapter we would like to quickly highlight some of the changes that we would like to see in the future. These include both features we designed but didn't get to implement due to time constraints aswell as features that were left out to reduce the scope or that weren't explored in detail yet.

4.1 Backend

One of the problems we noticed during the first couple of weeks on the project is that the codebase is an issue. There have been many people writing and working on differnt parts of this project. This led to a very chaotic codebase with a lot of parts working not very smoothly together and an overall lack of overarching design principles. In order to fix that we need to properly separate the backend from the frontend and then restructure, comment and document the different parts of the backend. For this project we wrote a functional version of a modern web application frontend, but in order to make it work we had to write another layer on top, which means we somewhat added to the problem. We did however lay the groundwork, so that a rewrite of the backend is now possible without making major changes to the frontend.

4.2 Further Changes to the UI/UX

There are some features that we originally planned to put into this beta version of our application. However due to time constraints we didn't end up implementing them, however the designs for these features were already made.

Better Design

The current design is an improvement, but there is still work left to do. Adding animations would help making the experience feel more premium and increase the perceived quality of the application drastically. Designing these elements takes a lot of time and resources, which we were unable to invest without compromising other more essential features. But the way we built our components allows for changes or additions to be made without requiring major rewrites

Player Mode

The player mode is one of the features that we feel like really should be implemented in the very near future. We think it would greatly improve the quality of life for users that played these matches themselves and want to use it to transfer them to another analysis tool. We ultimately decided to cut the feature for this release as the potential time to implement it properly would have been

quite big. The back end of the application was not set up to support this kind of user interaction. That means this feature will be much easier to implement once the back end of the application has been reworked to support this feature properly.

Autoplay Feature

Throughout the design process we talked a lot about having a feature that would let the user play through the moves automatically, without having to press a button to validate the moves. This is a feature that was implemented in the last version of ChessReader. We believe its a good feature to have since it helps speed up the process, however we have not decided how exactly we would make sure that every move with no input is validated as a correct move, and not just a move the user skipped on accident because the autoplay was too fast. Due to time constraints we ended up not implementing this feature yet since we haven't found a solution to this issue.

4.3 Additional Features

In this section we would like to look at some of the features that we could see being useful in the future. This includes both suggestions made from testers of the original Chessreader as well as our own ideas.

German Chess Notation

Since this project started with the help of the Swiss Chess Federation, one of the immediate requests was for the program to support languages other than English as well. So far we only try to identify the english chess notation on a score sheet. However in practice a lot of swiss players use the german version of the algebraic chess notation as well, so it would be a big help if our tool would also recognize and translate the german notation into the correct moves.

Direct import into Databases and Analysis Tools

Since one of the big use cases for our tool is to offer players the ability to offer an Analysis of their over the board games, a feature that would be very useful is the direct export to different chess analysis tools. Currently this can already be done by downloading the PGN and then uploading it to a tool of their choice. However a big improvement to the usability of the tool would be the ability to directly switch to some of the most popular analysis tools with the click of a button, instead of manually managing their games.

Mobile App

With the frontend now implemented in a widely used framework, the step to develop this web application into a mobile application wouldn't be as daunting. A good API of the backend would take care of most of the computational requirements while a framework like Flutter could be used to run our frontend on a mobile device.

Chapter 5

Conclusion

In this project our goal was to move the Chessreader application a step closer to a fully fledged release. One of the suggested changes by the last bachelor thesis was a decoupled frontend with a JavaScript based framework. We agreed with that approach and decided to use Angular as said framework since its fairly easy to pick up and has a easy to understand structure. We redesigned the UI the user was going to interact with and rewrote the core functionality in the new framework. Then we encapsulated the backend code into an Interface that we can call cleanly from our frontend.

While we believe a good step towards a more complete application has been made, there is still a lot to do. During our meetings with our supervisor we often ran into the problem of having too many building sites to properly cover all of them.

Appendix A

Appendix I

A.1 Backend API Documentation

Here we include the API Documentation we wrote for the backend, we always included an Input to the Call, a short description of what the call does and then what the call returns.

Documentation Settings

PA-Backend

API used by the Backend for Chessreader.

Base URL: https://Chessreader.org

POST Upload new picture

https://chessreader.org/

Input:

In body

a picture in the png, jpeg, jpg format

`file`

Process:

Stores the picture in the db and generates a match_id

Output:

An ID identifying the Match uniquely

```
match_id: Number
```

HEADERS

X-RapidAPI-Key

b580adde8fmsh61d9bba14863bdfp13230djsn571d9cbb37b6

BODY formdata

file

Example Request

Upload new picture

```
curl --location --request POST 'https://chessreader.org/' \
--form 'file=@"/Users/lukas/Desktop/B4721DD1-4678-46A8-BF25-722391B7A9FF.jpg"'
```

GET Align picture straight

https://chessreader.org/chess-match/40/align

Input:

`match_id: Number`

Process:

Aligns the picture correctly so that lines are facing vertical and removes shadows on the picture.

Output:

a html file with rendered confirmboxes

Example Request — Align picture straight

```
curl --location --request GET 'https://chessreader.org/chess-match/40/align'
```

GET Detect box outlines on aligned picture

https://chessreader.org/chess-match/40/boxes/scan

Input:

`match_id: Number`

Process:

Detects where on the page the boxes are and divides the picture into multiple smaller chunks for each move.

Output:

☰

`json_dict`

Example Request — Detect box outlines on aligned picture

```
curl --location --request GET 'https://chessreader.org/chess-match/40/boxes/scan'
```

POST Recognize moves

https://chessreader.org/chess-match/40/moves/recognize

Input:

In route

`match_id: Number`

In request.data

A json string containing a dictionary with a custom box type containing the x,y on the page and the width and height of the box.

`json_dict`

Process:

cuts the aligned picture into pieces according to the boxes passed in the json_dict. Then sends those pictures to OCR and stores everything in the database.

Output:

HTML page (editorPage.html) for editing the result

BODY raw

```
{
    "selected_boxes": [
        {
            "h": 107,
            "w": 408,
            "x": 531,
            "y": 1125
        },
```

Example Request

Recognize moves

```
curl --location --request POST 'https://chessreader.org/chess-match/40/moves/recognize' \
--data-raw '{
    "selected_boxes": [
        {
            "h": 107,
            "w": 408,
            "x": 531,
            "y": 1125
        }
```

View More

GET Get box image

https://chessreader.org/chess-match/40/moves/1/box_image

Input:

`match_id: Number`

`box_nr: Number`

Process:

Redirects you to the editors page for this specific box and match

Output:

HTML page (editorPage.html) for this specific move

Example Request

Get box image

```
curl --location --request GET 'https://chessreader.org/chess-match/40/moves/1/box_image'
```

POST Update moves

https://chessreader.org/chess-match/40/moves/update

Input:

In route

`match_id: Number`

In request.json

`move: Move`

Process:

Updates the move in the database

Output:

JSON response if successful or not

Example Request Update moves

```
curl --location --request POST 'https://chessreader.org/chess-match/40/moves/update'
```

GET Get ChessMatch png

https://chessreader.org/chess-match/40/pgn

Input:

`match_id: Number`

Process:

Generates the PGN for the specified game.

Output:

A Json String containing a message and the PGN string.

`{'message': f'match {match_id} found.', 'pgn': pgn}`

Example Request Get ChessMatch png

```
curl --location --request GET 'https://chessreader.org/chess-match/40/pgn'
```

GET Get Match metadata

https://chessreader.org/chess-match/40/metadata

Input:

`match_id: Number`

Process:

Gets the Metadata for this match from the DB

Output:

A Json String containing a message and the metadata String.

`{'message': 'Metadata found', 'data': data}`

Example Request Get Match metadata

```
curl --location --request GET 'https://chessreader.org/chess-match/40/metadata'
```

POST Create or update Match metadata

https://chessreader.org/chess-match/40/metadata

Input:

In route

`match_id: Number`

In request.json

Process:

creates or updates the metadata for this game

Output:

status code

Example Request — Create or update Match metadata

```
curl --location --request POST 'https://chessreader.org/chess-match/40/metadata'
```

DEL Delete a match

https://chessreader.org/chess-match/40

Input:

`match_id: Number`

Process:

Deletes the match from the DB

Output:

Status Code

Example Request — Delete a match

Appendix B

Appendix II

B.1 English Algebraic chess notation

Excerpt from the FIDE Handbook[11]

> *Description of the Algebraic System*
>
> *C.1 In this description, 'piece' means a piece other than a pawn.*
>
> *C.2 Each piece is indicated by an abbreviation. In the English language it is the first letter, a capital letter, of its name. Example: K=king, Q=queen, R=rook, B=bishop, N=knight. (N is used for a knight, in order to avoid ambiguity.)*
>
> *C.3 For the abbreviation of the name of the pieces, each player is free to use the name which is commonly used in his country. Examples: F = fou (French for bishop), L = loper (Dutch for bishop). In printed periodicals, the use of figurines is recommended.*
>
> *C.4 Pawns are not indicated by their first letter, but are recognised by the absence of such a letter. Examples: the moves are written e5, d4, a5, not pe5, Pd4, pa5.*
>
> *C.5 The eight files (from left to right for White and from right to left for Black) are indicated by the small letters, a, b, c, d, e, f, g and h, respectively.*
>
> *C.6 The eight ranks (from bottom to top for White and from top to bottom for Black) are numbered 1, 2, 3, 4, 5, 6, 7, 8, respectively. Consequently, in the initial position the white pieces and pawns are placed on the first and second ranks; the black pieces and pawns on the eighth and seventh ranks.*
>
> *C.7 As a consequence of the previous rules, each of the sixty-four squares is invariably indicated by a unique combination of a letter and a number.*
>
> *C.8*
>
> *Each move of a piece is indicated by the abbreviation of the name of the piece in question and the square of arrival. There is no need for a hyphen between name and square. Examples: Be5, Nf3, Rd1. In the case of pawns, only the square of arrival is indicated. Examples: e5, d4, a5. A longer form containing the square of departure is acceptable. Examples: Bb2e5, Ng1f3, Ra1d1, e7e5, d2d4, a6a5.*
>
> *C.9*
>
> *When a piece makes a capture, an x may be inserted between:*
>
> *C.9.1*

the abbreviation of the name of the piece in question and

C.9.2

the square of arrival. Examples: Bxe5, Nxf3, Rxd1, see also C.10.

C.9.3

When a pawn makes a capture, the file of departure must be indicated, then an x may be inserted, then the square of arrival. Examples: dxe5, gxf3, axb5. In the case of an 'en passant' capture, 'e.p.' may be appended to the notation. Example: exd6 e.p.

C.10

If two identical pieces can move to the same square, the piece that is moved is indicated as follows:

C.10.1

If both pieces are on the same rank by:

C.10.1.1

the abbreviation of the name of the piece,

C.10.1.2

the file of departure, and

C.10.1.3

the square of arrival.

C.10.2

If both pieces are on the same file by:

C.10.2.1

the abbreviation of the name of the piece,

C.10.2.2

the rank of the square of departure, and

C.10.2.3

the square of arrival.

C.10.3

If the pieces are on different ranks and files, method 1 is preferred. Examples:

C.10.3.1

There are two knights, on the squares g1 and e1, and one of them moves to the square f3: either Ngf3 or Nef3, as the case may be.

C.10.3.2

There are two knights, on the squares g5 and g1, and one of them moves to the square f3: either N5f3 or N1f3, as the case may be.

C.10.3.3

There are two knights, on the squares h2 and d4, and one of them moves to the square f3: either Nhf3 or Ndf3, as the case may be.

C.10.3.4

If a capture takes place on the square f3, the notation of the previous examples is still applicable, but an x may be inserted: 1) either Ngxf3 or Nexf3, 2) either N5xf3 or N1xf3, 3) either Nhxf3 or Ndxf3, as the case may be.

C.11 In the case of the promotion of a pawn, the actual pawn move is indicated, followed immediately by the abbreviation of the new piece. Examples: d8Q, exf8N, b1B, g1R.

C.12 The offer of a draw shall be marked as (=).

C.13 Abbreviations

0-0 = castling with rook h1 or rook h8 (kingside castling) 0-0-0 = castling with rook a1 or rook a8 (queenside castling) x = captures + = check ++ or = checkmate e.p. = captures 'en passant' The last four are optional.

Sample game: 1.e4 e5 2. Nf3 Nf6 3. d4 exd4 4. e5 Ne4 5. Qxd4 d5 6. exd6 e.p. Nxd6 7. Bg5 Nc6 8. Qe3+ Be7 9. Nbd2 0-0 10. 0-0-0 Re8 11. Kb1 (=) Or: 1. e4 e5 2. Nf3 Nf6 3. d4 ed4 4. e5 Ne4 5. Qd4 d5 6. ed6 Nd6 7. Bg5 Nc6 8. Qe3 Be7 9 Nbd2 0-0 10. 0-0-0 Re8 11. Kb1 (=) Or: 1. e2e4 e7e5 2.Ng1f3 Ng8f6 3. d2d4 e5xd4 4. e4e5 Nf6e4 5. Qd1xd4 d7d5 6. e5xd6 e.p. Ne4xd6 7. Bc1g5 Nb8c6 8. Qd4e3+ Bf8e7 9. Nb1d2 0-0 10. 0-0-0 Rf8e8 11. Kb1 (=)

We include the current README file of our Github Page for completeness. An updated version can be found under *https://github.zhaw.ch/chessreader/ChessReader$_A$ngular/blob/main/README.md*.

Appendix C

ChessReader Angular README

This is the client side application for the ChessReader project. It is written in Typescript using the Angular framework and the material library.

C.1 Project structure

The project ist set up like a typical Angular project (more general information about Angular below).

There are a few special things to be noted though: - Folder structure - The folder called "pages" contains components representing entire views/pages of the application - The folder "shared" contains two folders - In "components" you find all small components that are used in one or more pages - "mat-all" does not need to concern you except when you import a new material component. The module in it imports all material components to be used in the application - The folder "core" contains all services/pipelines/other files that don't directly correspond to a visual element of the application - In the folder "types" you can find all type declarations

C.2 Help and Introduction to Angular

If you are not familiar with Angular you can find help and an introduction using the links below.

Many of the components used in the project are imported from Angular Material[1].

C.2.1 Tutorial Videos

Angular & Typescript Tutorial for Beginners[2]

C.2.2 Development server

Run `ng serve` for a dev server. Navigate to `http://localhost:4200/`. The application will automatically reload if you change any of the source files.

C.2.3 Code scaffolding

Run `ng generate component component-name` to generate a new component. You can also use `ng generate directive|pipe|service|class|guard|interface|enum|module`.

[1] `https://material.angular.io/`
[2] `https://www.youtube.com/watch?v=k5E2AVpwsko`

C.2.4 Build

Run `ng build` to build the project. The build artifacts will be stored in the `dist/` directory.

C.2.5 Running unit tests

Run `ng test` to execute the unit tests via Karma[3].

C.2.6 Further help

To get more help on the Angular CLI use `ng help` or go check out the Angular CLI Overview and Command Reference[4] page.

[3] `https://karma-runner.github.io`
[4] `https://angular.io/cli`

Bibliography

[1] B. Horváth and C. Dreher, "Document Digitization for Chess Scorecards," *unpublished*, Jun. 2020.

[2] A. Abduli, "Document Digitization for Chess Scorecards," *unpublished*, Dec. 2020.

[3] V. Caglayan and B. Nielsen, "Digitalisierung von Schach-Formularen mittels Character Recognition Ensembling und Zug-Korrektur," *unpublished*, Jun. 2021.

[4] N. Ambrosini and G. Hellinger, "Digitalization of Chess Scorecards," *unpublished*, Jun. 2022.

[5] M. Śmigielski, "From chess score sheet to ICR with OpenCV and image recognition." Mar. 2019. [Online]. Available: https://medium.com/@mareksmigielski/from-chess-score-sheet-to-icr-with-opencv-and-image-recognition-f7bed2cc3de4

[6] R. Sudharsan, "Scannable Chess Scoresheets with a Convolutional Neural Network and OpenCV," Oct. 2021. [Online]. Available: https://heartbeat.comet.ml/scannable-chess-scoresheets-with-a-convolutional-neural-network-and-opencv-3e10dc1c91ba

[7] "CheSScan." [Online]. Available: https://chesscan.com/#home

[8] S. M. Shamim, "Handwritten digit recognition using machine learning algorithms," *Global Journal of Computer Science and Technology*, vol. 18, no. D1, p. 17–23, Jan. 2018. [Online]. Available: https://computerresearch.org/index.php/computer/article/view/1685

[9] "Chess Score Sheet (+ Free PDF Template) - Chess.com." [Online]. Available: https://www.chess.com/terms/chess-score-sheet

[10] "Amazon.com : US Chess Federation Carbonless Chess Score Sheets - Pack of 100 Sheets : Office Products." [Online]. Available: https://www.amazon.com/House-Staunton-Carbonless-Chess-Sheets/dp/B0182OK9LK

[11] FIDE, "FIDE Handbook." [Online]. Available: https://handbook.fide.com/

[12] S. Edwards, "Standard: Portable Game Notation Specification and Implementation Guide." [Online]. Available: https://ia902908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt

[13] D. M. Lane and Y.-H. A. Chang, "Chess knowledge predicts chess memory even after controlling for chess experience: Evidence for the role of high-level processes," *Memory & Cognition*, vol. 46, no. 3, pp. 337–348, Apr. 2018. [Online]. Available: https://doi.org/10.3758/s13421-017-0768-2

[14] M. Noback, *Advanced Web Application Architecture*. leanpub, Sep. 2021.

[15] "Angular - Introduction to the Angular docs." [Online]. Available: https://angular.io/docs