Computer Vision for Exam Grading

Final Report

Robin Bijl
Timotei Jugariu
Richard van de Kuilen
Hidde Leistra
Ruben Young On

Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

Computer Vision for Exam Grading

by

Robin Bijl
Timotei Jugariu
Richard van de Kuilen
Hidde Leistra
Ruben Young On

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be presented on July 2, 2019 at 2:00 PM.

Project duration:	April 23, 2019 – July 2, 2019	
Project committee:	Coach	S. Hugtenburg, MSc
	Client	Dr. A.R. Akhmerov
	Coordinator	Ir. O.W. Visser
	Coordinator	Dr. H. Wang

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

PREFACE

This document is the final report of the bachelor project of Computer Science and Engineering at the Delft University of Technology. The report documents the development process of our additions to Zesje and the research conducted to address the problems.

We would like to thank Stefan for his coaching during our project, and for providing us with test data to check the functionality of Zesje.

We would also like to thank Anton, Hugo and Joseph for their frequent and valuable feedback on our development process.

Robin Bijl, Timotei Jugariu, Richard van de Kuilen, Hidde Leistra, Ruben Young On

Delft, June 2019

SUMMARY

Grading exams is a time-consuming activity for teachers. Zesje is an open-source tool created to aid teachers in exam grading and streamline the grading process. Zesje currently uses computer vision techniques to realign images, and automatically find student numbers. However, teachers can currently only use Zesje to grade questions manually. Moreover the computer vision capabilities of Zesje can be improved. To make it easier to grade exams, it should be possible for teachers to have multiple choice questions graded automatically.

This project describes various improvements for Zesje, most notably using computer vision for the automatic grading of multiple choice questions, improving the accuracy of aligning scanned submissions, and automatically detecting blank solutions.

The team had to make several choices regarding implementations and choice of technology. Design goals were also created to serve as a guideline for the project. At the end of the project, with the features implemented by the team, Zesje can automatically grade multiple choice questions, identify blank solutions and has the corresponding front-end changes that allow the user to create multiple choice checkboxes on the exam PDF. These features have been tested extensively.

The use of Zesje also poses some ethical challenges. Using automated grading may result in the event that some submissions may never be seen by a grader.

By using benchmarks to compare the performance of processing scans in Zesje, the team found out that the grading time has greatly been reduced.

KEYWORDS
Computer vision, auto-grading, digital grading, open source

Contents

INTRODUCTION

Every student is familiar with preparing for the upcoming exam period. During this period, they have to spend as much time as possible studying, and need to make sure they have enrolled for their exams in time. Exams are usually the main factor that will determine if a student passes a course. Not passing an exam means taking a resit exam. Since resits are generally planned during the next semester, students will have to study for resit exams while also taking other courses. Should students not pass a resit exam, they will have to follow the course again in the following year. For students that are not in the first year, this means they may face a delay in their study program, and as a result have to pay for additional studying costs. For first-year students, the consequences of failing an exam are even more severe, as their results will decide if they get their positive BSA.[1] Otherwise, their registration for the study program will be terminated.[2] Altogether, this means that exam periods are usually a very stressful time for students.

However, the exam period is also not a relaxing time for teachers and Teaching Assistants (TAs). Before the exam period starts, teachers have to create an exam with possibly different versions. Since an exam has to test if students possess the knowledge required to pass the course, all questions should test the course material in a balanced way. Exam questions should not contain any mistakes, since any mistake may disrupt this balance. After the exam is over for students, hundreds of student submissions will need to be graded. This makes grading exams a very time-consuming activity for teachers.

During the academic year of 2018-2019, student enrolments at the TU Delft increased by almost 50%[1], with more than 1000 enrolments for the bachelor course Computer Science and Engineering [2]. This increase of students naturally means more students have to take an exam, and that more space is needed to provide space for students to take their exams. This means that grading exams is a very significant part of the time and money spent grading for teachers and TAs.

One solution to make grade exams easier is to make students take exams digitally. Some exams can be taken online or with software such as MapleTA,[3] which means they can be graded electronically. However, paper-based exams are still commonly used today, since not all exams can be properly 'digitized'. For example, if students need to write complex mathematical formulas, this can be more effectively done on paper.

Compared to electronic exams, a clear disadvantage of paper-based exams is that student submissions have to be distributed among graders. Teachers will have to distribute submissions among TAs, or need to host 'marking parties' where all TAs meet with the teacher to grade exams together. Distributing exams among TAs does not scale very well, and paper can easily get lost or damaged in this process. Marking parties may be more efficient than distributing the work among TAs [3]. However, marking parties have some disadvantages. Since all TAs need to meet at one specific time, it may be hard to find times at which every TA is free. This does not scale very well if there is a large amount of TAs. Another disadvantage of paper-based exams is the way in which feedback is provided to students after the exam is finished. With electronic exams, feedback can be sent to students via e-mail. For paper-based exams, teachers will need to determine a special day in which students can visit them to see their exams, which also costs time.

Ultimately, this means that some exams may take over 300 man hours to grade [4]. Not only does this take time for teachers that could be spent for other things, it is also quite an expensive process for universities, since teachers and teaching assistants are paid per hour. Therefore, any process that (partially) automates exam grading will aid teachers and make grading exams easier and less stressful. For example, teachers would no longer need to do manual bookkeeping of student scores per question in large Excel files. Automated exam grading could also mean that the feedback on specific questions can be sent to students via e-mail.

[1] Binding recommendation. The BSA means students have to achieve a certain amount of credits (ECTS) during their first year of studies.
[2] https://www.tudelft.nl/studenten/faculteiten/lr-studentenportal/onderwijs/bachelor/binding-recommendation-bsa/
[3] http://www.icto.tudelft.nl/en/tools/maple-ta/

Zesje[4] is a tool that assists teachers in the grading process. Its goal is to provide a systematic way for teachers to create and grade exams and to reduce the amount of time it takes to grade exams [5]. Zesje is a free open-source tool, which means that it is free to use and anyone can contribute to it. Zesje was created with a minimal effort approach, as its name suggests.[5] Although it originally worked for its goal, this 'minimal effort' approach meant that the quality of Zesje was not very high. However, it has become more widely used and has undergone significant improvements since it was first created [6].

Although Zesje has changed significantly since its creation, there is still room for improvement. Since Zesje was created by programmers, it is currently most useful to teachers who have technical knowledge. It also needs graders to manually grade student answers. Zesje also makes use of computer vision techniques to automate a few tasks related to the grading process, such as identifying information about a student submission by using a QR code, aligning a scanned image using corner markers and automatically reading out student numbers. However, more steps of the grading process can be automated. Teachers may want to test the knowledge of their students with multiple choice questions, and the process of grading multiple choice questions can be partially automated. Although there already is software to grade multiple choice questions, it has several limitations.[6] Another way to aid teachers and TAs to grade exams is by automatically identifying which solutions are not filled in by students. Therefore, the main goal of this project was to improve on the current computer vision techniques by providing a way to automatically grade multiple choice questions, and by automatically identifying solutions that were not filled in by students.

In Chapter 1, the problem that Zesje was created to address is outlined, and the current main components of Zesje are summarized. Chapter 2 contains an analysis of the areas in which Zesje falls short, the main parts of the requirements devised in the project, the design goals, and the focus of this project. Chapter 3 describes the changes in the typical workflow that were introduced in this project, and the design goals and technological choices are summarized. In Chapter 4, the development process is described. Chapter 5 contains an explanation on the implemented features. Chapter 6 contains an evaluation of development process, the features implemented and discusses the quality of our work. In Chapter 7, the ethical considerations of a product such as Zesje are discussed, as well as a discussion on which possible features to implement in the future. In the final chapter, the report is concluded. The appendices include the original project description as provided on BEPSys, the info sheet, the feedback received from the Software Improvement Group[7] (SIG) and a list of the original requirements prioritized according to the MoScoW model.

[4] The code for Zesje can be found at https://gitlab.kwant-project.org/zesje/zesje

[5] The name 'Zesje' is derived from the Dutch *zesjescultuur*, which refers to a student culture in which students are satisfied with the lowest sufficient grade of a 6 (zes).

[6] Multiple choice question grading is already automated at the TU Delft with proprietary software such as EvaSys (https://en.evasys.de/main/home.html) and ConTest (http://www.icto.tudelft.nl/tools/contest/). However, it can only be used with single-answer multiple choice questions, and requires separate answer sheets.

[7] https://www.softwareimprovementgroup.com/

1

CURRENT SYSTEM

This chapter provides a description of the current Zesje together with a background on why Zesje was created. Furthermore the chapter summarizes the typical workflow and identifies the main components of the application. In this chapter the word 'current' should be read as 'Zesje before the changes that are described in this document'.

1.1. HISTORY

Zesje was created by Anton Akhmerov and Joseph Weston [6, p. 7] as a tool to make exam grading more systematic and less time-consuming [5]. At the time, Anton was lecturing an undergraduate course at the TU Delft. More than 200 students participated in his course, and students had to take a paper-based exam every week [7]. The prospect of grading all of these exams was something Anton and Joseph did not look forward to. Therefore, Anton and Joseph looked into ways to streamline this process.

There were already commercial systems available to automatically grade exams, however these proved to be quite costly [7] and raised privacy concerns, since confidential exam data would now be in possession of a third party [5]. Since third party systems may be optimized for other uses, they decided that a solution that works well enough would be satisfactory. They started Zesje, a minimal prototype of a tool that works well enough for the job at hand. The aim for simplicity and a minimum working product means that code complexity can be kept at a low.

Since their exams were already created in LaTeX[1], Zesje was written to be used in combination with LaTeX. The LaTex dependency was later removed as part of another bachelor thesis [6]. QR codes were attached to every page with an identifier for the system to match when scanning in the students' submissions.

The front page of the exam also came (and still comes) with a box for the students to fill in their student number. Joseph and Anton attempted to use OpenCV as a first solution to automatically read student numbers. This was without success, since the accuracy depended a lot on the quality of the scan. In the original implementation student numbers were checked manually. This was accepted, as the reasoning was that it's similar to just another problem to grade.

Another addition to Zesje that proved to be helpful for students was the possibility for graders to add personalized feedback. Each student can be assigned a set of feedback options from a defined rubric for each problem. Each feedback option can be assigned a number of points, and a more detailed remark. By summing the points the student obtained, a grade can be calculated. Once grading is finished, students can be notified by sending an e-mail from inside the system. This also made it possible to send all feedback to the student in a predefined template.

The data in this system is saved in an SQLite database, staying in tune with the minimal product philosophy. SQLite is a lightweight database system that can be approached through a SQL-like interface and is

[1] https://www.latex-project.org/

saved on disk as a local file. It is run as part of the process that requires it. The front-end is a React app that gets its input data from the API served by a Flask application written in Python.

Although Zesje was originally developed with little effort, it has undergone numerous improvements since its creation. Cleintuar, Van der Krieken, and Mahabier have contributed to Zesje as part of their 2018 bachelor thesis [6]. Their contributions include:

- Remove the dependency on LaTeX so that any PDF in A4 format is a valid entry.
- Automatically reading out student numbers.
- Inserting corner markers to correctly position a scanned PDF.
- Generation and insertion of barcodes to automatically identify exam instances.
- Viewing and editing locations of answer boxes.
- Preview an exam with barcodes, corner markers, and student widgets.
- Generating multiple copies of exams for printing

1.2. CURRENT WORKFLOW

This section describes the workflow of Zesje before the changes in this document (see Chapter 5) were implemented. They are presented to give the reader an understanding of the system and to provide context to the new features described later in this document. Note that this explanation does not serve as a manual. For the right workflow, please refer to the public sandbox instance.[2]

When a teacher starts Zesje, they are shown the home screen. On the home screen, a tutorial is displayed on how to use Zesje. The teacher can do multiple things from the home screen:

- Add an exam
- Add students or graders
- Send students an e-mail about their result
- View statistics of a graded exam
- Export the database

1.2.1. ADDING AN EXAM

Exams can be added by uploading any A4-shaped PDF and giving it a name. Zesje allows the teacher to switch between exams at any time. Once uploaded, Zesje will bring the teacher to the next panel, where they can edit the exam.

1.2.2. EDITING THE EXAM

Once a teacher has uploaded the exam, they will be shown the exam editor. This shows the pages of the exam that were uploaded, and the user can navigate back and forth between the pages. Here, the teacher must drag the student number box to the desired location on the first page. It is also possible to move the barcode location on each page for later identification when scanning in submissions. This view also adds corner markers to each page, which cannot be moved. Once done, the user can finalize the exam. This locks in place the locations of these elements.

DESIGNATING PROBLEMS

In this view and the next, the user may also define the problems of the exam for the system to recognize. To do this, the user draws a rectangular box around the relevant area. This has to include the answer space for the question, as this will later be shown to graders. Furthermore, titles can be given to these problems for the graders to know which problem they are looking at.

FINALIZING THE EXAM

The teacher can click on 'Finalize' to finalize an exam. Finalizing the means that the teacher can no longer

- move the student number widget,
- modify the position of the barcode,

After finalizing, problems can still be created, deleted and moved around on the page.

[2] https://sandbox.grading.quantumtinkerer.tudelft.nl/

DOWNLOADING EXAM COPIES

Once the teacher has finalized an exam, the teacher can specify an amount of copies of the exam to generate. These copies will have a unique barcode on each page, containing the identifier. The user must themselves print these copies and ensure they are given to the students taking the exam.

1.2.3. HANDLING AND GRADING SUBMISSIONS

Handling submissions is the main goal of Zesje. It consists of multiple parts.

ADDING STUDENTS

To automatically match the student number from the exam to an actual student, the user needs to add the students to the database. This can be done by exporting student data from Brightspace as a .csv file, and then uploaded into Zesje. This has to be done before uploading the submissions from students.

UPLOADING SUBMISSIONS

After the exam is taken, the teacher will have to scan student all submissions. Once all submissions have been scanned, the teacher can go to the 'Submission' tab to upload the student submissions as PDFs. Uploading the submissions can take quite a while to get processed.

ADDING GRADERS

Zesje can have multiple graders, and the names of graders are shown to the students when sending the results. Simply going to the 'add grader' option allows the user to add a new grader. The active grader can be selected from a drop down menu.

GRADING

The answers students wrote in the designated problem boxes are presented to graders. To give feedback, a feedback option has to be selected that consists of a name, description, and the amount of points obtained for getting that feedback. Multiple feedback options can be checked for the submission. In this way, it is easy to create a problem that contains partial grades for partial solutions. For example, getting two steps right out of three steps in a question could still warrant half the points in this way. Feedback options can be added in the grading view too.

1.2.4. EXAM OVERVIEW

The exam overview provides statistics on how each student performed. In the overview, the user can see how well students performed on each question, but showing the cumulative distribution of the scores. Zesje also computes two statistics for each question.

- Cronbach's α (alpha)[3]
- Rir Score

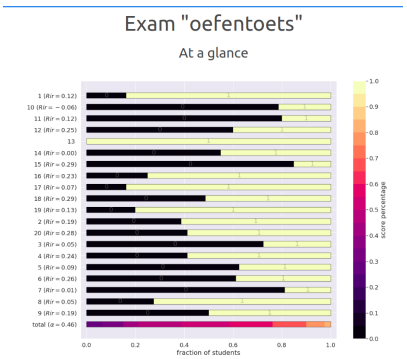
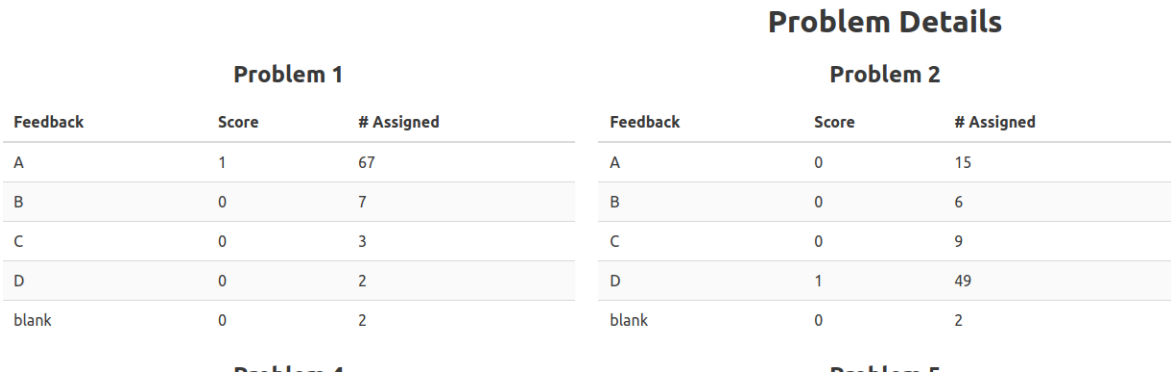


Figure 1.1: The overview graph showing Rir, alpha and point distribution.

[3]Cronbach's α measures how well a specific question measures the overall performance of students.



Problem Details

Problem 1

Feedback	Score	# Assigned
A	1	67
B	0	7
C	0	3
D	0	2
blank	0	2

Problem 2

Feedback	Score	# Assigned
A	0	15
B	0	6
C	0	9
D	1	49
blank	0	2

Figure 1.2: An overview of the feedback distribution.

1.2.5. SENDING E-MAILS

Once all submissions have been graded, teachers can send an email to students with their personalized feedback. To send emails to every student, Zesje provides a template for sending each email. Here, the teacher can define various things, such as specify the student name, and show all the feedback provided by graders.

1.3. CURRENT DESIGN

In the following subsections, the current state of the system will be described by looking at the different parts of the application: back-end, API, database and front-end.

1.3.1. BACK-END

The back-end consists of two parts, the API run through a Flask app, and an SQLite database.

API

The back-end of Zesje is written entirely in Python as a Flask app. This provides the connection between the database and the API. The API for the back-end is a RESTful API that users can interact with, and data is sent back and forth in JSON format. This makes it easy to integrate it with a web front-end, which frequently uses JSON to communicate.

DATABASE

As a database engine, Zesje currently uses SQLite. SQLite allows saving the database in an on-disk file or in-memory. Zesje currently stores the database on disk.

To make changes to the database and define its structure, Zesje uses an ORM in Python. Originally, PonyORM was used. However, PonyORM proved not to be the ideal candidate for two reasons. First, PonyORM's semantics were not easy to understand. Secondly, PonyORM does not yet have a tool to automatically generate database migrations.[4] During development, this meant that migration files would have to be created manually for every database change, which would take unnecessary work. Up until this point, the 'solution' to migrations was to not migrate the database at all. The main reason for this was that each course is used in a fresh Zesje instance, and that courses typically do not run for a very long time.[4]

At the time this project started, the Zesje team switched to SQLALchemy. SQLAlchemy is used more frequently, and has a tool to automatically create migration scripts, Alembic. However, using SQLAlchemy introduced its own problems.[5] First, since SQLAlchemy required a connection to the Flask web app, other libraries had to be installed such as Flask-SQLAlchemy and Flask-Migrate to execute the migration scripts. Flask-SQLAlchemy uses multiple sessions to connect to the database. These sessions are however not thread-safe. Another problem the team encountered was that libraries cannot be used by multiple sessions at once, and would create an error otherwise. It also means that multiple PDFs cannot be processed by Zesje with Flask-SQLAlchemy. A solution to this was to use the library Celery for parallel PDF processing. Celery is a Python library that provides a distributed task queue. Celery was introduced with the switch to SQLAlchemy.

[4] https://gitlab.kwant-project.org/zesje/zesje/issues/238
[5] https://gitlab.kwant-project.org/zesje/zesje/merge_requests/123

1.3.2. FRONT-END

From our experience user interfaces are prone to a lot of changes. Developing a user interface that has a stylish look, is responsive and easily extensible is not an easy task. Since Zesje was developed as a minimal working prototype, an intricate, modern user interface was not a priority. Hence, the choice of frameworks was important to make the development of the user interface efficient and effortless to a certain degree.

Bulma is a open source CSS framework which is used to build responsive, elegant user interfaces with minimal effort. It defines CSS classes that can be assigned to HTML elements in order to give them a certain style or layout.

React.js is a JavaScript library which enables the web developer to create components that are similar to HTML elements. These components are defined as JavaScript classes that extend *React.Component* and implement a render method which describes how that component should be displayed.

1.3.3. OVERVIEW

Table 1.1 provides an overview of the technologies, frameworks and packages that are used by the current Zesje implementation.

Component	**Purpose**	**URL**
Flask	Web framework in Python	http://flask.pocoo.org/
OpenCV	Library for computer vision	https://opencv.org/
NumPy	Python library for scientific computing	https://www.numpy.org/
SQLAlchemy	SQL toolkit and ORM	https://www.sqlalchemy.org/
Alembic	Database migration tool	https://alembic.sqlalchemy.org/
Flask-SQLAlchemy	Flask extension for SQLAlchemy support	https://flask-sqlalchemy.palletsprojects.com/
Flask-Migrate	Flask extension for migration support	https://flask-migrate.readthedocs.io/
Celery	Aynchronous task queue	http://www.celeryproject.org/
Bulma	CSS framework	https://bulma.io/
React	JavaScript library for UI	https://reactjs.org/

Table 1.1: A summary of components used in the current Zesje

2

Problem Analysis

In the previous chapter we discussed features and state of Zesje at the start of the project. In this chapter, we turn our attention to the shortcomings of Zesje. Furthermore, the focus of the project is outlined to see which of these problems can be addressed within the time span of the project by creating and prioritizing a list of requirements, and by determining general design guidelines that should be followed.

2.1. Shortcomings of Zesje

In this section, the shortcomings of Zesje that were identified by the team are listed.

2.1.1. Scan Processing

Zesje currently uses computer vision in scan processing for three main features. First off, the scanned student submissions are aligned by using corner markers. Second, the student number is automatically read from a submission using a student number widget. Third, information about an exam page is added using a QR code, so that the exams do not have to be scanned in a specific order.

However, computer vision can be improved in these areas, and can be extended for other uses such as grading multiple choice questions. There are issues with the accuracy of realigning the scans based on the detected corner markers. Also, no computer vision techniques are used to assist in grading, which means Zesje requires graders to manually grade exams. One example of computer vision that can be used in *open* questions is automatically identifying of blank solutions.

Multiple choice questions

There is also no support for automatically grading multiple choice questions. Zesje currently requires teachers to manually grade questions. Providing the possibility for teachers to automatically grade multiple choice questions would reduce the workload of grading. Each multiple choice question can contain a set of automatically generated checkboxes. For each student submission, computer vision can be used to detect if these checkboxes are filled in. The information about whether This means a multiple choice question can be graded automatically.

Scan Realignment Accuracy

To show only the relevant parts of a question to a grader, the scanned images need to be aligned correctly. Zesje currently uses corner markers on each submission to realign it after it has been scanned. Computer vision is used to identify the location of the corner markers for each submission. After the corner markers have been identified, the submission is realigned. Realigning a scan is not perfect, and an offset may still exist compared to the original exam in the database.

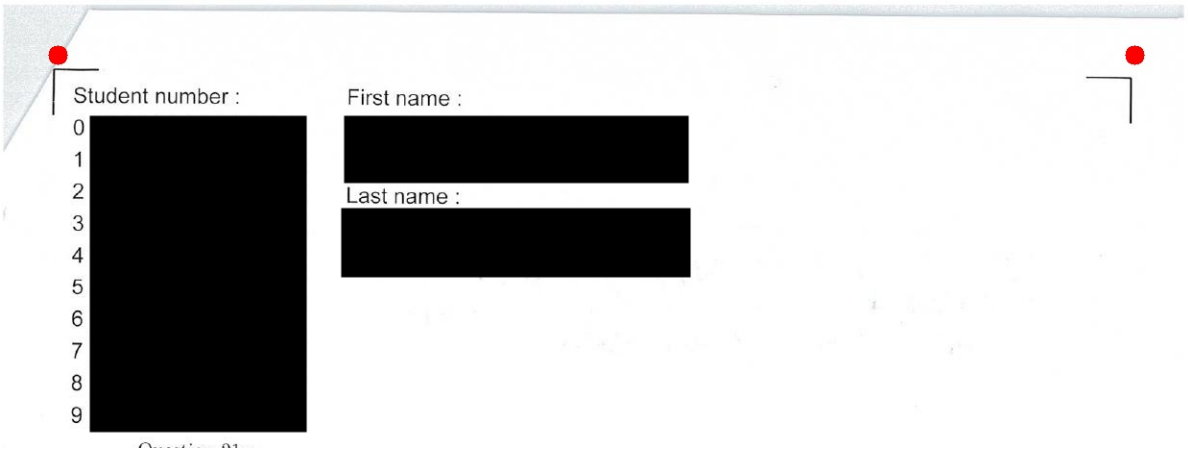


Figure 2.1: An example where the image is not aligned. The edges of the corner markers should be moved to where the red dots are.

Zesje has functionality for this, but it is not extremely accurate. The realignment is only accurate to about 10-20 pixels. It is still sufficient to show text based answers when a generous margin is applied over the designated problem area (subsection 5.1.3).

For multiple choice questions, answer boxes will only take up around a dozen pixels side to side. Therefore, this accuracy is not enough to handle the answering boxes for multiple choice questions. The goal for this step is to obtain realignment accuracy within the range of only a few pixels compared to the original exam.

2.1.2. DATABASE

Since Zesje uses SQLite, using migrations means that it is impossible to create a migration that edits a table column name.[1] This means that a migration which changes a table column has to rename the entire old table, create a new table, and then copy the data from the old table into the new table. This is quite an inefficient approach, since all the data has to be moved around between tables. However, using a different database than SQLite would mean that the database cannot be simply managed in memory. For example, using a system such as MySQL means that the user has to setup a local server to host the database on, which makes it harder to use Zesje. Our decision was to not use a system such as MySQL, since it was not a desired feature for the client.

2.1.3. USABILITY ON SINGLE MACHINES

Although Zesje can be used as a web-based tool, it is currently not optimized for this purpose. If a teacher wishes to use Zesje, they have to request a separate running instance of Zesje. This makes Zesje harder to use since each teacher needs to request access, and a specific domain has to be created for each instance of Zesje. Ideally, Zesje should be a web application that requires only one running instance. However, Zesje currently does not contain any type of user access control to check if a teacher has access to a specific course. Access control would be crucial to ensure that Zesje as a web application can be used by multiple users at the same time.

A downside of making Zesje a fully functional web-based tool would be privacy considerations. If all data that Zesje uses is stored in a web application, this would mean that exam data is now stored somewhere on the web. However, privacy problems are already present in Zesje, which will be elaborated on in Chapter 7.

2.1.4. PDF UPLOADING

At the start of the project, uploading and processing PDFs was not optimal. It was not possible to upload multiple PDFs at the same time safely. Each process of uploading a PDF would require a connection to the database. These connections would time out since these processes were all making use of the database at the same time. There was no queue system either, so exams where the scans consisted of multiple PDFs had to be uploaded one at a time. This was impractical and could take a lot of time. On top off it not being desirable

[1] https://www.techonthenet.com/sqlite/tables/alter_table.php

to upload multiple PDFs, if it was done, it only made use of one thread and therefore did not make full use of the system's resources.

2.1.5. NON-FUNCTIONAL SHORTCOMINGS

Zesje also has some non-functional shortcomings. First off, Zesje has been used primarily by people with some know-how of computers. Zesje has not been adapted to be easy to use for teachers that do not possess the technological know-how. This due to the fact that if Zesje is used outside of the expected workflow it can run into an error or crash, such as trying to upload to upload a CSV file of student numbers that does not have the same formatting as files exported from Brightspace. If an error is displayed, it can be hard for a user to understand if they don't know the system well in a technical aspect.

On top of this, at the start of the project the API was not tested. The main reason the API was not tested was that testing the API means that the database has to be tested as well. Without any unit tests it meant that any changes to the API had to be tested manually, which has a negative impact on the maintainability of Zesje.

Lastly, at the start of the project, there was a series of tests, but no way to measure test coverage. Without any coverage and visualization tools, it is difficult to find untested code, and therefore negatively impacts the maintainability of Zesje.

2.2. REQUIREMENTS ANALYSIS

At the start of the research phase, the team did a comprehensive analysis of all the ways the team can improve Zesje. Besides the requirements related to computer vision, the team considered the need for access control in Zesje and disputed improvements to the user experience, code quality and overall performance.

2.2.1. COMPUTER VISION

This category contains features that improve the image processing of scanned student submissions in Zesje. Many of the features were provided directly by the client and represent core requirements for the final product.

Currently, Zesje uses computer vision to automatically read student numbers and QR codes that contain information about a page, such as copy numbers. However, Zesje cannot be used to automatically grade multiple choice questions. Automatically grading multiple choice questions would drastically reduce the time spent grading and is, therefore, a must-have for an application such as Zesje. Another missing feature is the identification of blank solutions, which automatically detects submissions that are not filled in and displays a warning to the user indicating that an answer is most likely blank. Additionally, the client proposed to look into Optical Character Recognition (OCR) for detecting handwritten numbers. With this feature, Zesje could for example pre-grade submissions to questions that are a single number. At last, the positioning of the scans based on the markers that Zesje adds to the exam PDFs is another feature that needs improvement. Many of these features require in turn the implementation of an additional feature. For instance, to make automatic multiple choice grading possible, Zesje needs to allow teachers to create feedback options before uploading the submissions. Zesje cannot pre-grade the multiple choice questions if it does not know what the right answer is and what feedback to select.

2.2.2. ACCESS CONTROL

Development of Zesje for different users is currently only possible if each user gains access to a separate instance of Zesje. This means that someone has to maintain each instance of Zesje to grant access to teachers that want to use an instance for their courses. This would not scale very well if a large amount of users were to use Zesje, since it would require somebody to maintain all instances. It would also be nearly impossible to update all instances of Zesje if, say, a change is introduced.

The current implementation of Zesje also does not have any type of authorization. It can only be implemented on the entire website and even then, it does not allow for selective access. This means that it is currently impossible to grade two different exams on the same instance of Zesje without giving the graders access to both exams. Should the capabilities of Zesje be extended in the future, this approach also does not scale very well.

A solution would be to implement some type of access control specifically focused around graders and exams. The first step is to create a log-in system and add authorization to the APIs. The second step is to split up the access to exams. Any user can upload an exam. Once the exam has been created, only the uploader should be able to give other users access to that specific exam. This way a user can only access their own exams or exams that he or she is grading. The exception to this rule are administrators who should have access to all the data.

2.2.3. USER EXPERIENCE

The workflow of Zesje can be improved, specifically considering the installation of the application and its usability. The set up could be simplified in two ways: either by creating an easy-to-install local application or by building a full web application that could be deployed by a system administrator. The second area of improvement is usability. There are some minor tweaks that could improve the user experience significantly. The upload screen could be extended with logs or other error messages if an upload is not processed for any reason (e.g. duplicate submission or PDF with no image). Modifying the feedback for an existing question is currently not intuitive. Another possibility to improve usability would be to add the possibility to split up feedback into two new options where all previous submissions (containing this feedback) would have to be regraded. In general it would be a benefit if it were possible to filter the submissions by specific feedback.

2.2.4. PERFORMANCE

Image processing can take quite some time if exams have many participants. It would therefore be very useful to increase the performance of the image processing pipeline by either speeding up the processing or adding multi-core support. Benchmarks need to be created to measure these improvements in speed and accuracy.

2.2.5. CODE IMPROVEMENTS

As Zesje was built as a minimal working prototype, not much attention has been devoted to test coverage and maintainability. Although a 'minimal effort' approach may seem attractive because it does not require a lot of time, it means that Zesje will be hard to maintain from the start. If other developers contribute to the Zesje on a temporary basis [6], this means that they either have to rewrite existing code to make it easier to maintain, or ignore the existing code. The first approach requires time not spent on developing new features, and the second approach would make Zesje progressively harder to maintain, as it cannot be expected that every developer produces code that is perfectly maintainable. The team's proposal was to improve the existing code in terms of fixing some known issues, increasing the test coverage and providing more documentation. After discussions with the client and coach, the team considered that these proposed ideas are not of high priority as they do not add any new features to the application. For the code we add to Zesje, we do however aim to employ best programming practices and to test the code we contribute to the project as much as possible.

2.3. PROJECT FOCUS

Fixing all previously listed issues of Zesje would take too much time to fit in the time span of the project. Therefore, a focus had to be defined to determine which issues should be addressed in this project. For the features that were not implemented, refer to Chapter 7.

The team met with the client and the coach and refined the list of proposals to a concrete list of specific requirements and prioritized these according to the MoScoW[2] model. The full list of requirements prioritized according to the MoSCoW model can be found in Appendix A. Out of these requirements it was decided to implement at least the 'must have' and 'should have' requirements. These requirements are summarized in the following list:

- **Must have**
  - Identify blank solutions
  - Add the possibility to add checkboxes to the front-end.
  - Add the option to recognize multiple choice answers.
  - Add a pre- or auto-grading in the UI.
  - Perform benchmarking on image processing.

[2] a concrete list of Must have, Should haves, Could haves, and Won't haves

  - Make sure the user can enter feedback options before the exam is taken.
- **Should have**
  - Connect feedback to multiple choice options.
  - Improve positioning accuracy of scans based on the corner markers.
  - Implement QR codes on odd pages.
  - Add a queue system for uploading scanned PDFs
  - Parallelization of uploading and processing PDFs.
  - Implement data visualization in the back-end to show various statistics.
  - Add possibility to view another problem of the same student.
  - Automatically add question title for new problems.
  - Use data visualization in the front-end.

2.4. DESIGN GOALS

In this section, the design goals of the project will be described in further detail. The design goals should serve as guidelines throughout the project and will be evaluated at the end.

2.4.1. MAINTAINABILITY

Since Zesje is an open-source tool, it is almost certain that other developers will contribute to it during and after our project. Therefore, one goal is to ensure the changes made to Zesje in this project should be easy to understand and maintainable for current and future contributors. Changes should also not create conflicts with changes that other developers want to implement. For each feature, a merge request is created to the main repository of Zesje [3]. Although it is not straightforward to define a quantifiable measure for maintainability, the feedback from SIG[4] provided some indication on maintainability.

2.4.2. PERFORMANCE

Teachers and TAs may need to grade hundreds of exams during the exam period. Grading exams is already a time-consuming process, as exams have to be scanned to PDFs and analyzed in Zesje. Therefore, the code added to the project should not negatively affect performance. This would make it even harder for teachers to currently grade exams. One way to measure performance is that 80% of all exams should be processed within one night.

2.4.3. USABILITY

As grading exams is a time-consuming process for teachers, Zesje should be a usable tool. Any changes that would reduce usability may therefore negatively affect the grading process. Our goal is to ensure grading exams does not take any extra time.

2.4.4. RELIABILITY

Although Zesje can be used as a web-based tool, it is not optimized for this purpose. If a teacher wants to run Zesje locally, it still needs to be reliable enough to ensure there are no crashes during the grading process. As the process of grading may take quite some time, if Zesje crashes at any point, the grading process may be disrupted. Any errors that may occur during the use of Zesje should be caught properly and at no time should make the tool crash.

2.4.5. TEST COVERAGE

Test coverage can provide an overview of all code that is tested, and may be a good indicator of maintainability. For the sake of reliability, maintenance and code quality, the code that is written by the team should be tested. The aim is to reach more than 80 percent code coverage. This does not include coverage over code that is not written by the team.

2.4.6. BENCHMARKING

Benchmarking can provide information about the performance of a part of the system. Since our project includes changes to performance-intensive tasks, our goal was to benchmark several of the changes we made.

[3] For a more detailed analysis of merges that were performed, see subsection 6.3.4.
[4] For more details, see subsection 6.2.5.

This includes the speed when loading in PDFs, but also aspects such as the accuracy of the image processing features implemented by the team. With benchmarks, the performance of these tasks in the old version of Zesje can be compared to our newer implementation. Our goal was to ensure our changes do not make these tasks slower. For a more detailed analysis on which benchmarks were performed, see Appendix E.

3

System Design

In this chapter, the choices that were made regarding the implementation are documented. From this point, WOMM is used to refer to the new implementation and Zesje to the product in general.

3.1. Changes to Workflow

The changes to the workflow are described in this section. From now on Zesje will be used to describe an instance of the original system. WOMM refers to Zesje including the changes made by this project.

3.1.1. Adding an exam

When the teacher selects a problem, WOMM now tries to infer the problem title automatically. If Zesje cannot guess the title of the problem, the problem name will be set 'New Problem'.

Adding feedback options

Once a teacher has designated a problem, feedback options can be added to it in the exam editor. Each feedback option can be designated by a name, a more extended description and given a score. Once a feedback option has been created, it can also be edited and deleted. Feedback options are to be used when grading the exam.

Adding Multiple Choice Options

For each problem, the teacher can flip a switch to indicate that the selected problem is a multiple choice problem. Once the teacher has flipped, the switch, they have the possibility to add checkboxes to the problem. The teacher is able to define the amount of checkboxes, and the labels above the checkboxes.

Label type	Example	Amount of options
Alphabetical	A, B, C, ...	1 - 9
Numerical	1, 2, 3, ...	1 - 9
True/False	True, False	2
No label		1 - 9

Table 3.1: Types of labels that can be added to checkboxes.

For each label, a feedback option will be added to the problem panel. These feedback options are later used to automatically grade multiple choice questions in student submissions.

Grading Policy

For each problem, a grading policy can now be set. This grading policy determines if the solution to this problem should be manually graded in each student submission. The user can set three options:

- Show everything.
- Show everything that is not blank.
- Show only multiple choice question that has multiple answers.

WOMM will then automatically grade and approve any solutions that are within the grading policy.

3.1.2. GRADING EXAMS

Grading is almost the same as in Zesje. There are three main differences. Some solutions will already be graded before a grader starts looking a the solution. These are: all the solutions that are set by the grading policy. Other solutions might not be graded yet but will have some feedback selected. These are the solutions that should be manually approved by a grader. The grader can then either modify the feedback or press 'A' to approve what has already been set. The final change can only be found with multiple choice questions. WOMM will now high lite the check boxes that it has found.

3.2. CHOICE OF TECHNOLOGY

In this section, all choices that were made throughout the project regarding the use of specific technologies to solve the problems of Zesje are explained.

3.2.1. COMPUTER VISION

There are various computer vision and imaging libraries for Python. These will be mentioned below.

NUMPY

Numpy is a Python library that has a heavy focus on making mathematical operations easier. It supports matrices in which images can be represented as well as some basic image manipulation tools. As it is mostly made for numerical methods rather than images, we do not think it is a good option that will suit all of the goals we need it for.

OPENCV

OpenCV is a library written in C++ and designed to perform image processing tasks. It is accessible for our Flask application because it has Python bindings. In our research we find that OpenCV is most used among computer vision libraries. Furthermore, it has many features for dealing with 2D images, which is very closely approximated with flat scanned exam sheets. It should be noted that OpenCV is already present in the current packages list of Zesje. Currently it is used to pre-process scans of taken exams by finding the corner markings and interpreting the student number. This means that using OpenCV does not add any new dependencies for this item.

PILLOW

Pillow is a fork of the Python Image Library (PIL). Pillow supports basic image processing functionality, including point operations, filtering with a set of built-in convolution kernels, and colour space conversions [8]. The library also supports image re-sizing, rotation and arbitrary affine transforms. One advantage of Pillow is that it works on Windows, Linux and MacOS.

CONCLUSION

To conclude, OpenCV was chosen for implementing computer vision. It has many features that the other libraries do not have, as well as features shared. It should be noted that all three of these libraries are already used by Zesje and are compatible with each other, if some aspects can be done better in another library, there is the option to do that selected part in another library.

3.2.2. PROBLEM TITLE EXTRACTION

One feature was selecting the problem title automatically in an exam. Our initial idea was to select a specific region of a page, and then perform OCR on this region to find the text belonging to a problem. But there are also other techniques of getting this information.

OCR

Optical character recognition (OCR) is a technique used to read characters from images. OCR may be useful in Zesje for reading text on the exam and for name recognition. Currently, after an exam is scanned, teachers have to re-enter the title of a question before providing feedback. OCR can be used here on the printed exam text to read out the question title automatically. OCR can be as accurate as 95% on printed text [9]. However, scanned PDFs may produce images with a noisy or shaded background, which may negatively affect accuracy.

PYTESSERACT

Tesseract is an OCR engine originally developed as proprietary software by HP. Later, Tesseract was released as an open-source project.[1]

Since then, Tesseract has gained significant popularity and is well-documented. It makes use of machine learning and provides the option to train it in order to reach higher accuracy for specific fonts. Tesseract uses various techniques to recognize words, such as: chopping up joined characters, associating broken characters, and linguistic analysis. Tesseract uses additional checks to determine if a scanned word is correct. For example, the nearest word or the nearest word in a language-specific dictionary is used.

Since the back-end of Zesje is written in Python, a library to use Tesseract in Python has to be used. One library for OCR in Python is PyTesseract.[2] PyTesseract is based on Google's Tesseract OCR Engine. PyTesseract is compatible with other image libraries such as OpenCV and PIL/Pillow. This is useful if any preprocessing needs to be performed on an image

OCROPUS

OCRopus, also known as OCRopy[3] is an open-source set of tools for retrieving text from images. It was developed by the German Research Centre in Artificial Intelligence with funding from Google. OCRopus was designed to adapt the use of OCR to be more compatible with large scale digital library applications, unlike commercial OCR libraries which are optimized for desktop use such as scanning letters and memos. [10]. This makes it an unsuitable candidate to use in Zesje, since only small amounts of text need to checked to find the title of a problem.

TENSORFLOW

Another way to implement OCR is by using neural network libraries. One of the most used libraries is TensorFlow. There are multiple models designed for OCR and well-known models are Attention-OCR networks[4] such as the one used to remove numbers from street view in France[11].

A benefit of using one of these models is that it can be highly tailored to a specific use, but that may require training the network and acquiring enough data to train the network. Unfortunately, neural networks are known to require a lot of processing power compared to traditional algorithms and therefore might not be well-suited for Zesje. Another issue is that there can be a substantial speedup depending on if the system running it has a Nvidia graphics card. This raises issues with both our 'usability' and 'maintainability' design goals. If GPU-based libraries would be used, Zesje would require a full installation of those libraries, which are usually quite large. A full local installation of Nvidia CUDA 10.1 requires at least 2.5 GB [12]. Furthermore, all developers may now need additional knowledge of GPUs and GPU programming to be able to contribute to Zesje, which negatively impacts maintainability. Another problem is that Zesje is now GPU-dependent, which means that any user without the right GPU cannot use Zesje. Overall, we think that Tensorflow does not suit Zesje compared to specially tailored OCR solutions like Tesseract.

PDF TEXT EXTRACTION

Another approach to find the problem title is to directly extract the text from a PDF. There are various libraries to extract text from pdfs, such as Textract, Tika and PyPDF. This would mean, however, that selecting the problem title no longer falls under the computer vision aspect. However, a lot of these libraries are unsuitable for the task, since selecting a problem title requires text extraction in a specific region of the PDF. It would still technically be possible to extract the text from a specific location using these libraries, but this would mean the elements of a PDF file have to be parsed manually, which would be a lot of unnecessary work

Most libraries mentioned above could only extract the text of an entire document or page, which is redundant and would mean information about the location of text is lost. There were a few libraries that would return text in a specific location, such as PyMuPDF and Pdfminer. Eventually, Pdfminer was used, since it did not require any external dependencies.

[1] https://github.com/tesseract-ocr/
[2] https://pypi.org/project/pytesseract/
[3] https://github.com/tmbdev/ocropy
[4] https://github.com/tensorflow/models/tree/master/research/attention_ocr

CONCLUSION

After looking at the different libraries, our choice was to use pdfminer, since it does not require any external libraries.

3.2.3. VISUALIZATION TOOLS

There were a few should-have requirements of the product that were related to displaying statistics. These requirements/ideas were proposed by the development team and were approved and described as important features to have by the product owner. One of the requirements was to display time heuristics including average time spent grading per question, total time needed to grade an exam, time spent grading per TA and so on. Another requirement was more related to statistics regarding students. Displaying how many students passed or grade distribution can be valuable data for the end users. Yet another informative statistic is displaying exam grading progress in total and per question.

As it was at the beginning of the research phase, Zesje had a color plot on the Overview page, showing the score distribution for each problem. The issue with that plot was that it was difficult to extract information from it and it was rendered server side which caused a bit of a delay when loading the page.

For all applications described above, the development team could have used: pie charts, histograms, progress bars and so on. A choice had to be made regarding the usage of a helpful library that would aid the users in displaying the aforementioned statistics. The team did not spend much time deciding implementation details of the statics plots and therefore the first criterion for choosing a data visualization library was adaptability. The second criterion was the variety of widgets and diagrams provided by the library. The third criterion was the popularity of the tool. The last criterion the team used is the availability of the documentation, as it can provide a good overview of all the function a tool can provide.

Some of the libraries that the team considered are: Plot.ly, DevExtreme Datagrid, Chart.js, Chartist.js and D3.js. The advantages and disadvantages of each choice are described below.

MATPLOTLIB

Matplotlib is a very well-known graphing library in Python, it forms the basis for many other graphing libraries. As mentioned above, Zesje used Matplotlib to render the color plot server side. However, it was not the most suitable library for Zesje since it created a static picture (a PNG) that was served to the client displayed. For a teacher, it may be more useful to see interactive charts. An advantage of interactive charts is that a teacher can gain different insights into an exam by looking at different parts of these charts.

PLOT.LY

Plot.ly is a visualization tool that can make interactive charts centered around JSON. Plot.ly has libraries in: Python, Matlab, R and JavaScript.

Plot.ly's Dash framework is a very handy visualization tool for Python. Dash provides a way to interact with HTML callbacks and can be used in conjunction with Flask. Next to compatibility with Flask, it can also work with an SQL connection to retrieve data, a feature that could be useful when Zesje switches to SQL. Overall, Dash provides a large variety of widgets that can be used in Zesje.

The Plot.ly JavaScript library is by far the most popular of the libraries on GitHub. It provides a way to render graphs client side which can be useful for alleviating tasks on the server side. There is also the React Plotly.js library which allows one to embed charts in react.

DEVEXTREME

DevExtreme[5] is a suite of components UI components including charts, tables, forms and other widgets. The diagrams are compiled on the client side and are available for JQuery (among others) and some also for React based applications.

[5] https://js.devexpress.com/

CHART.JS

Chart.js[6] is an open-source library that provides a large variety of charts. Chart.js is arguably the easiest library to install. It is even available via CDNs (Content Delivery Network) and can be included on a webpage by simply including a script tag. The library is well documented and very popular and thus, there is quite some support for it.

OTHER ALTERNATIVES

There are plenty of alternatives to choose from. Each library provides similar diagrams and ways to display informative statistics. However some are better documented and maintained than others. This report limits itself to the libraries described above as they provided sufficient features and documentation for the development to work with.

CONCLUSION

The team chose to use Plotly.js, due to its robustness, ease of use, and large variety of available diagrams. The diagrams are generated client-side which reduces the load on the server and makes a clear separation between the generation of the data (server-side) and how it is displayed (client-side). Plot.ly is arguably better documented, maintained and supported than the other alternatives.

3.3. DESIGN CHOICES

The design choices made are documented and explained here. The advantages and disadvantages of certain approaches are touched upon as well, and for each item there is a conclusion.

3.3.1. PROBLEM TITLE EXTRACTION

To implement the requirement of automatically adding the title to a problem, There has to be a way to get this title. Previously, two possible methods were chosen, and we will now go deeper into detail about the differences and select the best implementation.

ADVANTAGES AND DISADVANTAGES

Using OCR compared to direct PDF text extraction has a few advantages and disadvantages.

Disadvantages of OCR The main disadvantages of using OCR is the fact that OCR requires images as input to work, and that it is not always very accurate [9]. OCR is also generally slower than direct text extraction. Finding the title of a problem means that a very specific section of the image has to be searched.

PyTesseract does not provide any support to find text in a specific region of an image. Therefore, the image of a page in which the problem is located has to be cropped, which is a rather expensive operation since it means part of the image has to be copied.

Another disadvantage of OCR is that it requires additional dependencies. OCR depends on various image processing techniques, such as image segmentation and pre-processing [13]. Additional libraries may negatively impact the usability of Zesje, especially if they are platform-dependent.

Advantages of OCR The advantage of OCR compared to direct PDF text extraction is that OCR is more likely to find any text.

If a PDF is composed of a series of images, there would be no text to extract. In such a scenario, however, OCR *would* be able to extract text since it requires an image as input. However, selecting a problem title is not a feature that has to be perfect, since it is relatively easy for a teacher to manually enter the title of a problem.

CONCLUSION

Eventually, PDF text extraction was used to determine the title of a problem since it did not require that images had to be extracted from the PDF or cropped. Extracting the problem title takes around 40 milliseconds, but this depends slightly on the size of the exam.

[6] https://www.chartjs.org/

3.3.2. ADDING MULTIPLE CHOICE QUESTIONS

In most exams, teachers want to test the knowledge of their students by using multiple choice questions. Since multiple choice questions only need a student to pick one or more options, it should be fairly easy to grade them automatically. Below, the approach to design such multiple choice questions for automatic grading is outlined.

Multiple choice questions typically have a layout such as in Figure 3.1.

2. Which of these famous physicists published a paper on Brownian Motion?
   A. Stephen Hawking
   B. Albert Einstein
   C. Emmy Noether
   D. I don't know

Figure 3.1: Typical example of a multiple choice question in an exam.

Each option is designated with a label, such as an alphabetical character like A, B, or C, or with a number such as 1, 2, or 3.

One idea was to have students simply draw a circle around the options they wanted to pick. In Zesje, a student submission could then be compared to the original exam to see where a student has used a pen. Places where the student used a pen would be the places in which a student may have placed a circle. However, such an approach would be difficult to implement, since it required large changes to the user interface where a teacher would have to mark the locations in which they expect a student to draw a circle. Furthermore, this check would depend strongly on the orientation in which a student submission is scanned. If a scan is not properly aligned with the original exam, this approach would result in a lot of false positives as it would simply detect text instead.

Another idea was to create checkboxes for each option. This means each checkbox is coupled to an option in a multiple choice question. Unlike the previous approach, checkboxes all have the same form. This makes them easier to identify using computer vision, even if a scan is not correctly aligned. It is also easier to check if they are filled once the location of a checkbox is determined. When students take the exam, they should then be able to fill these checkboxes. Once submissions have been scanned, Zesje should check which boxes are filled, and grade the multiple choice question automatically.

There were a few considerations to be made regarding the placement of such checkboxes. For example, checkboxes could be placed next to each option like in Figure 3.2. Our idea was to assume as little as possible about the layout of a multiple choice question in the exam. Allowing only one format would negatively impact the usability of Zesje if teachers wish to use multiple formats. However, this means teachers have to manually place all checkboxes in the right location.

2. Which of these famous physicists published a paper on Brownian Motion?
   ☐ A. Stephen Hawking
   ☐ B. Albert Einstein
   ☐ C. Emmy Noether
   ☐ D. I don't know

Figure 3.2: Example of how checkboxes can be added to a multiple choice question.

As a consequence, one decision that had to be made was whether the user should move all checkboxes around individually, or whether they should be part of a larger block. In the case of moving each checkbox individually, a teacher could simply move each checkbox to the desired position. In the case of a block of checkboxes, a teacher should then be able to only move this block of checkboxes around. However, this limits the possibilities of the teacher, since they can no longer place each checkbox next to the right option. One problem

with this approach is that with a large amount of multiple choice questions, it would take a lot of time for teachers to move all checkboxes to the right position. Therefore, our decision was that checkboxes should only be moved as part of a larger block.

FEEDBACK OPTIONS

To provide the possibility to grade multiple choice questions automatically, each multiple choice option that the user entered in the interface had to be coupled to a certain answer. Zesje currently uses feedback options to grade student submissions. However, feedback options can currently only be created after the exam has been uploaded. Therefore, one of the features implemented in this project was to add the possibility to create feedback options before students take the exam. For more details on the implementation, see subsection 5.1.1. As a solution, a feedback option was created for each multiple choice option. Such a feedback option would have the same name as the label of the multiple choice option so the user knows they are related. This also means the user can directly assign points to the correct option. When Zesje checks if each box in a student submission is filled, it automatically selects the coupled feedback option.

A major downside of this approach is that if a student decides to fill in each option, they would get all the points. Additionally, a teacher may want to provide a more complex grading scheme than simply counting only points for the right answer. However, there are many possibilities to implement a grading scheme. Some teachers may want to deduct points for a wrong answer instead of simply giving 0 points. Therefore, it is currently not possible for teacher to provide a grading scheme for automatically grading multiple choice questions.

MANUAL GRADING

As a result, teachers may still want to manually check if students have filled in the right boxes, and create a manual grading scheme for each question. To fix this, a couple of options are provided when creating an exam that allow the teacher to specify which multiple choice questions have to be checked manually.

CONCLUSION

Checkboxes would be used to grade multiple choice questions. Each question can be moved around in a block by a teacher, since this requires the least amount of work for teachers. However, an extension to this approach where a teacher can move checkboxes around individually could be implemented in the future. Each multiple choice question is be coupled to a feedback option that is automatically selected by Zesje. It is currently not possible to provide a grading scheme for multiple choice questions, but teachers still have the possibility to check these questions manually. For more details on the technical implementation of multiple choice questions, see subsection 5.1.4.

3.3.3. IDENTIFYING MULTIPLE CHOICE QUESTIONS

Once the multiple choice questions have been added to the exam and filled in by the students, these questions need to be automatically graded. This saves time and is big part of our project.

Once a general area has been given where a checkbox can be found, it is time to find the exact location of the checkbox. We explored two methods of finding these boxes. One was similar to the one used in finding checked boxes of student numbers, the other was based on finding white in an image.

The method used in finding student boxes is based on using a blob detector function. The image is first preprocessed such that the background is white and the boxes are black and then the blob detector is applied. This method works well if there are other objects around which are significantly different in size than a checkbox. But it has trouble finding boxes that have been marked in such a way that there are also lines outside the box. An example is the case where a line extends partially from a filled box. This could potentially be taken advantage of by having students mark mistakes with very big crosses, making the implementation not detect any checkbox.

Another downside to this implementation is that the size of the area it has to detect a box in has a significant effect on it. As discussed before, checkboxes were designed to be placed on many different locations. This means that to avoid having any other objects that might also resemble boxes, The size of the search area would be preferably as small as possible. This implementation fails if the box is too close to the border of the

image.

The other implementation works by finding white in the image. Here, the background is made black and then the white of the inside of a checkbox is found. This implementation is simpler and works very well with unusual ways of filling in the box, such as circling it or applying a big check mark. It also is more tolerant to smaller search areas, but doesn't do as well as the other implementation when it comes to having other objects in the search area.

We decided to use the implementation based on finding white since it worked better with smaller search spaces and was more lenient to boxes that had been filled outside the drawing area.

3.3.4. IDENTIFYING UNANSWERED QUESTIONS

The implementations methods researched for identifying and pre-grading blank solutions can broadly be split up in two categories: counting average colour and counting elements. The first category boils down to a more simple implementation of calculating the average color of the image of a solutions. while generating the exam a single reference score is calculated for each problem. The same score is then calculated for every solution and identified as blank if they are similar. This method turned out to be too in precise for two reasons. The resolution of the image had too much impact on the score and the amount of noise in the scan could quite easily be larger than a short answer. The second method uses OpenCV to find elements in the image, The theory being that an answered question should have more words and therefore more elements. This however turns out not to be the case. In a perfect scenario this works quite well. The real world turned out to be a bit more difficult. One issue is again noise where short answers result in a smaller change then scan artifacts do. Another however is the answer field itself. in some cases, for example, this field contains lines and if every word touches the line no new elements can be discovered. A segmented version of the first implementation turned out to be precise enough which will be explained in more detail in section 5.1.7

4

METHODOLOGY

In the following sections of this chapter, the team's chosen software development process is described and the adopted debugging methods are presented.

4.1. SOFTWARE DEVELOPMENT PROCESS

The choice of an adequate software development process that is followed attentively during the development process is a necessary step towards a well functioning development team and a final product that is complete and of high quality. The team chose for the Agile methodology as it flexible in terms of planning and development while ensuring high quality code with up to date documentation.

In the research phase, the team came with a list of proposals and after discussions with the client and the coach the list was refined to the list of requirements specified in A. The team also wrote a product plan and a road map. In these documents the team reserved time for eventual delays.

4.1.1. SPRINTS

The team used weekly sprints. This decision was influenced by a few factors, the first being the short duration of the development period which was just six weeks. The second factor was the team's intention to meet as often as possible with the client. The team strived to meet the client every week in order to receive as much feedback as possible. A weekly sprint was ideal because the team had a set of completed items before every meeting with the client. Having received the client's review, the team discussed the progress made during the week and proposed improvement points for the following weeks. Afterwards, a set of items were selected from the product backlog and placed in the sprint backlog.

4.1.2. DIVISION OF TASKS

One of the challenges that come with developing in a team is division of tasks. After the sprint backlog for the next week was put together, the team had to assign items to the different members such that the workload was split evenly. There are a few factors that come into play and it is not always straightforward how to make a perfect division. One must consider someone's preference or knowledge to work on a certain part of the application or with a certain sets of technologies. If, for instance, a team member is highly skilled in React, it would be more efficient and productive to assign the front-end related work to that person. However, one learns more by working on parts of the application that he/she is not completely familiar with. On top of that, it is highly beneficial to understand the entire application in order to discover possible bugs, better implementations or to foresee compatibility issues between all the different parts.

4.1.3. COMMUNICATION

It goes without saying that communication is a key element for a well functioning team. The team communicated with client and the coach during the weekly meetings but also online via the chat service Mattermost. The team's intention was to keep the client informed and involved in most decisions they made. As the client was not always available, efficient communication was needed as for some decisions it was not worth spending too much time debating.

Within the team, every member needed to be on the same page regarding planning, designs, decisions about implementation and so on. It was important that the team communicated to prevent compatibility issues when merging multiple features into the application. Any team is more efficient when the members help each other and in order to do so one must communicate their progress and the issues that are holding him/her back to the rest of the team.

4.1.4. USE OF GITLAB

Version control is an elementary tool for managing changes in the documents of a software project. Version control helps the software engineers to create, delete or merge different versions of the same documents, revert documents to previous versions, compare different versions of the same documents and so on. GitLab is a popular open-source online repository manager which on top of the features related to version control, provides issue trackers, Continuous Integration (CI) pipeline, statistics and the list goes on. Git together with GitLab are already used by the developers of the current Zesje and there was and still is little to no reason for the WOMM team to change it.

The team agreed that every feature should be implemented on a separate branch and once it is finished a merge request should be opened. Moreover, a merge request should only merged after a proper review by at least one other team member. In case a reviewer asks for changes, these requests have to be resolved before the branch can be merged.

Continuous Integration is used to ensure that what is being uploaded to GitLab builds successfully and passes all the tests. On top of these checks, the CI pipeline also checks for code style issues such that the entire codebase complies with the same coding standards. The developers of the current Zesje adhere to PEP8[1] for the Python code, and to StandardJS[2] for the JavaScript code. From the start of the project, the WOMM team adhered the same standards to maintain a consistent code style.

The issue tracker feedback is an useful tool to organize the product backlog, the sprint backlog and to keep track of progress made on items. Labels and columns can and have been used to distinguish between items that are in progress, being tested or already finished.

4.1.5. CODE REVIEW

To have the opinion of another developer on the implementation is of utmost importance. A reviewer can suggest better implementations or he/she can spot bugs or compatibility issues with other features. Code review is also an excellent way to check the readability of the code. It is also easier to resolve eventual issues that arose from code reviews, if the newly written code is maintainable.

4.1.6. BENCHMARKS

To understand the performance of the work done, benchmarks can be used to quantify the changes made. They are important because they give us hard facts to compare against.

Benchmarks for accuracy can be made by our own metrics, whereas benchmarking for runtime is left to libraries specifically made for this purpose such as cProfile.

The results of these benchmarks can be found in subsection 6.2.2. As no heavy calculations are performed in the front-end there is no work worth benchmarking.

4.2. DEBUGGING

One of the most important skills that a software engineer must posses is the knowledge on how to debug quickly and efficiently. Debugging is essential in finding the cause of a problem, but it can also useful in visualizing the execution of the code and with that, ensuring that it executes according to the design. Therefore, it is of utmost important a software engineer is able to make efficient use of the available debugging tools. Considering the client-server architecture of Zesje, different methods and tools were used to debug the back-end

[1] https://www.python.org/dev/peps/pep-0008/
[2] https://standardjs.com/

and the front-end, as explained in the following subsections.

4.2.1. JUPYTER NOTEBOOKS

When it comes to testing the back-end of Zesje, one might want to only test aspects of the system without needing the entire flask instance or database. Having to start up a flask instance can take considerable time and a lot of time can be saved by having the option to quickly rerun the code with changes made. Flask has a debug mode where changes to the code are applied immediately, but there were cases where the change did not come through unless restarting. Jupyter notebooks allows one to write Python code in snippets and run them individually. It also allows one to show images embedded between the snippets.

With Jupyter notebooks, it is possible to import functions directly from Zesje. Combined with the ability to display images inline allows one quickly check multiple intermediate images and determine at which step it is going wrong exactly. This is important since with normal debugging, the images are large matrices and a human can not make anything out of them. It is possible to also display images in the code using OpenCv, but a bit more cumbersome depending on if the images are saved and have to be manually opened or opening multiple windows with images.

Using notebooks also helps with prototyping parameters related to image processing. One can easily load an image and change variables and check what effects result in this change. This turned out to be useful when debugging the pre-grading functions when it was run on 300 exams. Since grading went well in most cases, it was a very select few cases that needed to evaluated. Pre-grading was also part of the processing pipeline, and so doesn't have the option to pre-grade specific cases or twice for that matter. Therefore a notebook that loads a single exam and pre-grades it gives much insight on what went wrong with that specific case.

4.2.2. UNIT TEST DEBUGGING

Similar to with Jupyter notebooks, it is possible to run unit tests without having to start the entire application. Some IDEs (Integrated Development Environment) such as Visual Code [3] have easy interfaces which allows one to not only run individual tests, but also debug these tests. Being able to debug individual tests is very useful. First off, tests are made to be repeatable with the same input parameters regardless of state of configuration of the database or whole system. This makes narrowing down causes of bugs easier. Secondly, if running a specific function or set of functions require something setup, like a database for example, the tests testing these functions already have a setup and a takedown of the database which is always the same, so one can easily make a change to the code of the function, run the test and see what happens. Lastly, it is also possible (at least in Visual Code) to run tests in debug mode. This allows one to put breakpoints in the function code itself and follow the exact steps variable values, Something which is harder in Jupyter Notebooks.

4.2.3. POSTMAN

Postman [4] is a tool used to make API calls. Since API calls were not testable at the start of the project, Postman was frequently used to manually test responses from API calls. Manually testing the API would be done by starting a development instance of Zesje, and then making API requests via Postman to `http://localhost:8881/`.

4.2.4. DEVELOPER TOOLS

Most browsers nowadays come with a set of tools that enables any web developer to inspect much of the behaviour and code of a web page. This set of tools is often called "Developer tools" and provides the means to inspect the layout of the HTML elements, the CSS styling, the requests that are sent and the received responses and so on. Because Zesje's front-end is built in React, the team used the "React Developer tool" to inspect the states and properties of the React components. Modern browsers also offer the feature to debug scripts written in JavaScript or JSX.

[3] `https://code.visualstudio.com/`
[4] `https://www.getpostman.com/`

4.2.5. SIG

One of the methods used to improve code quality is the use of SIG. they will give feedback on the maintainability of our code. Normally this would be done by uploading the source code to them and getting feedback. This does not work for WOMM because most of the code already exists. It might be interesting to get an overall score but that is not within the scope of this project to work on. Therefore 2 versions will be uploaded. The source code of Zesje and our changes. SIG will focus its feedback on the code that has changed between these two versions. How we process this feed back will be discussed in section 6.2.5

5

IMPLEMENTATION

In this chapter, the implementation of the features that were implemented in WOMM are discussed. This chapter further touches on the database changes at the end, as this isn't an implemented feature on its own, rather than part of other changes.

5.1. IMPLEMENTED FEATURES

The implemented features in this work are described in more detail.

5.1.1. ADDING FEEDBACK ON EXAM START

In the existing Zesje implementation, feedback options to grade a submission with could only be added during grading itself. In this way, a user would have to take the exam first and scan in some submissions before being able to define these options. This is not necessarily hard to work with, but can create some nuisances. However, this workflow also prevents a user from defining the correct and incorrect answers in a multiple choice exam such that multiple choice questions are automatically graded.

To achieve the goal of automatically grading multiple choice questions, the user must be able to tell the system what options are correct, incorrect, and the score attached to them. The same view that allows adding and editing feedback in the grading workflow is now also placed in the exam creation screen. A difference is that in the grading page clicking an option would toggle that feedback option for the submission shown, whereas in the exam editor it opens a panel to edit the feedback option with. This panel can always be shown in both cases by clicking the marker on the right. See figure

5.1.2. QR CODES ON ODD PAGES

Another feature implemented is to automatically make the amount of pages in an exam even. This is necessary to ensure all pages can be recognized by WOMM. When an odd numbered exam is printed on double sided paper 1 blank page is created on the final page. WOMM cannot process this since that page does not contain any corner markers or qr codes. Adding the extra page to odd numbered documents ensures that WOMM will generate all these markers on the empty side as well. If an exam is printed on two sides of an A4 paper, the last page would be visible to students. As a solution, if the amount of pages in an exam is uneven, an extra blank page is added at the end of the exam.

5.1.3. DESIGNATING PROBLEMS

When the teacher selects a problem, WOMM tries to guess the problem title automatically. This is done by searching for the first line between two consecutive problem areas. If a problem is at the top of a page, WOMM searches for text between the top of the page and selected problem.

Selecting a problem will extract the text from the PDF, and then filter out the text that is not between the currently selected problem, and the problem directly above it. Next, the first line in the text that is left out is returned by splitting the text that is found on newline characters (`\n`). Finally, trailing and leading whitespace characters (such as spaces and tab characters) are removed from the title. If no text is found, the title of

a problem will be set to 'New Problem'.

The problem title is only guessed when the teacher creates a new problem, and not when the problem is moved around. This is because a teacher might want to manually specify a problem title before moving it around.

When a teacher has selected a problem, a panel appears on the left side (see Figure 5.1).

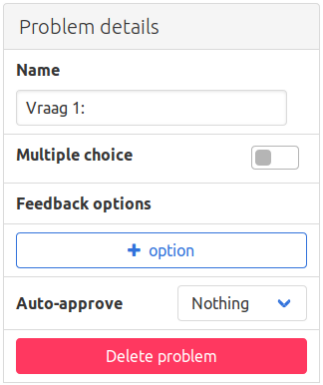


Figure 5.1: Panel that appears once the teacher has selected a problem.

5.1.4. ADDING MULTIPLE CHOICE OPTIONS

When selecting a problem, a teacher can flip a switch to add multiple choice options to a problem. After the teacher has flipped the switch, the following options appear: The # field, in which the teacher can also specify the amount of multiple choice options that should be generated, up to a maximum of 9. The 'Labels' field, where the teacher can specify if the checkboxes should be designated with numbers or alphabetical characters, or a special 'True/False' case if there are only 2 options where the checkboxes will be labeled with 'T/F'.

Next, a rectangle with the generated checkboxes appears inside the problem area. The teacher can move the rectangle with the generated checkboxes around within the problem area.

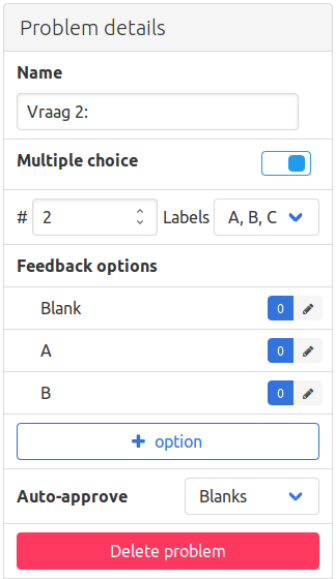


Figure 5.2: Problem widget if the teacher decides to flip the 'Multiple choice' switch.

This generates a set of checkboxes in the problem area that the teacher has selected.

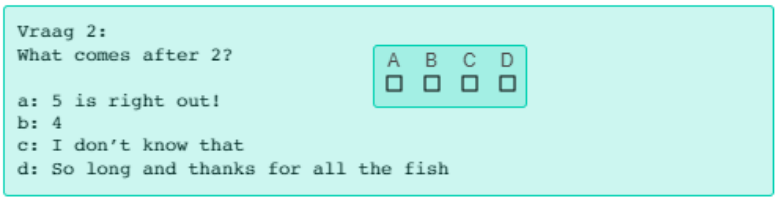


Figure 5.3: Problem area with generated checkboxes.

Adding a multiple choice option means that the labels had to be saved to the database. For each label, a feedback option is created with the same name as the label. This is done so that teachers can manually enter which multiple choice option a student has filled in.

This meant that for every multiple choice option, a corresponding feedback option had to be created.

These feedback options cannot be deleted. If the amount of labels is changed, or the teacher sets the problem to no longer be a multiple choice problem, then the related feedback options will also be removed.

Each multiple choice option had to be coupled to a feedback option that contains the label of each multiple choice question. To mark a question, a feedback option forms a relation with a submission. marking a multiple choice question makes use of the same logic as a normal question.

5.1.5. UPLOADING A SCANNED EXAM

After an exam is uploaded, the page in each submission is processed. First, the scanned submission is realigned by comparing the corner markers of the original exam to the scanned submission. Second, the information about each page is extracted by reading the barcode on it. The information read out by the barcode is then used to perform additional checks, such as checking if the submission belongs to the original exam. Next, the exam is "pre-graded".

5.1.6. GRADING MULTIPLE CHOICE QUESTIONS

When grading multiple choice questions, WOMM first detects if a checkbox is filled. This starts off by first finding the location of a checkbox for each ungraded student submission. The coordinates and which page the checkbox is stored on are stored in the database on creation. But since the checkboxes are on scans of papers, there is a degree of inaccuracy even after realigning the image. Figure 5.4 shows a case where the image is not well aligned and a crop has been made with generous amount of padding so that the checkbox is still on the crop. In an ideal case, the checkbox would be directly in the center and the padding could be smaller so that the crop does not include labels.

Afterwards, the image is converted to grayscale and a threshold is applied to remove artifacts and create hard outlines of black and white. The white background outside the checkbox is filled with black using a flood fill function, causing the crop to only contain white inside the checkbox. This can be seen in figure 5.5. This masks any characters that may have been cropped along.

Now another crop can be made to using the remaining white as a reference of the location of the checkbox. The results can be seen in Figure 5.6. Now, it could be that still some white is left over after filling the white background due to certain characters containing closed areas of white. In that case the crop is cut and another crop is made based on the white left. After cropping to the checkbox, the average color value can be taken +and if this is above a certain threshold, it indicates that the box has been filled in. If no white pixels are found or the crop is of a size not similar to the reference material, it means the box has been filled in substantially and an average is not even needed.

Figure 5.4: A crop of an image with a checkbox and label.

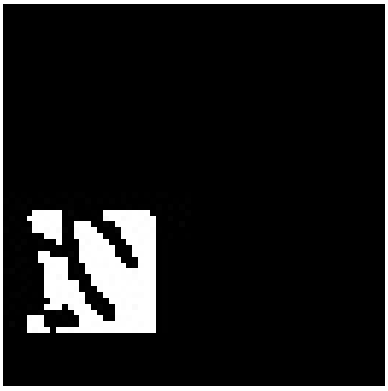


Figure 5.5: The same image as on the left, only converted to black and white and flood filled.

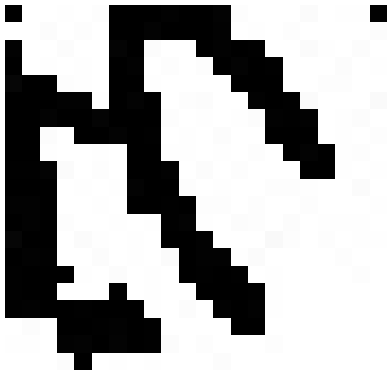


Figure 5.6: The resulting crop directly on the checkbox.

After detecting that the box is filled, the feedback that is associated to that box is added to the solution of the question. Now grader might wish to review all or some of the solutions that have been detected by WOMM. Depending on what grading policy is set for that question, the solution still has to be approved by a grader or automatically graded by the grader called "Zesje". There are three grading policies, namely:

- Let a grader manually approve all solutions.
- Automatically approve all solutions where nothing has been filled in by.
- Automatically approve all solutions where nothing or one option has been detected.

Manually approving a detected solution can be done similar to grading an open question, but with a two changes. First off, the checkboxes that are linked to selected feedback are highlighted. Figure 5.7 shows a set of multiple choice options and option D has been detected as filled and the feedback has been added to the solution. This highlight is applied in the back-end when the image is requested by an API call. The benefit of highlighting a selected option is that a grader does not need to look somewhere else to see what feedback as been set. The second feature plays into this fact and that is a shortcut to approve the selected feedback. Like this, a grader can quickly approve many of multiple choice questions in a short period of time.

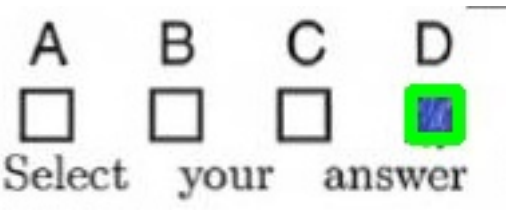


Figure 5.7: Multiple choice option that has been filled in and highlighted.

5.1.7. IDENTIFYING UNANSWERED QUESTIONS

A question that has not been answered should automatically be graded as blank. If a solution is detected as blank, a feedback option named ‘blank’ will now automatically be added to the solution. There are two types of questions that can be considered blank: multiple choice and open questions. A multiple choice question will be identified as blank if no option has been selected. Open questions are checked separately. First a reference image with equal resolution is loaded from file. If the file does not exist it is generated from an empty exam. This image is then compared to the image extracted from the scan. Both images are split up in rows with a width of 50 points. The average black value is calculated for each band. If any of the bands in the scanned image is darker then the reference image, the question is considered not blank. When all the bands have been checked and none are considered filled, the solution is set to blank. The grader Zesje is than set for the solution if that problem has been set to auto-grade blank solutions. This allows the grader to skip these when grading.

5.1.8. PARALLELIZATION

To see how parallelization works, a stress test was done on Zesje by uploading a large amount of submissions. As a result, it was discovered that uploading one PDF with a lot of submissions takes over 16 GB of memory.

Since Zesje has one function to process a PDF file and convert it to an image, a first idea was to look if this function was the culprit. To find out why this happens, a memory profiler in Python was used.
The result from the profiler can be found below in Appendix F. From the result (lines 59 through 62) it is clear that converting a PDF file to a `WandImage` object takes a lot of memory. After this debugging was done, and the result was shown to the client, another Zesje developer then started working on making sure the PDF was split per page[1] before being processed. After this had been implemented, memory usage stabilized to around 300 MB while processing multiple PDFs.

5.2. DATABASE

Adding a multiple choice option meant the locations of the checkboxes had to be saved to the database. Since the database already uses a table to save the location of widgets, the table that includes information of the multiple choice questions could extend the **Widgets** table. This can be seen in Figure 5.8 under the table **mc_option**. Another change that has been made can be seen in the problem table. This entry is used to indicate the grading policy for each problem. Overall, a lot of the database has been reused to implement the features.

[1] https://gitlab.kwant-project.org/zesje/zesje/issues/322

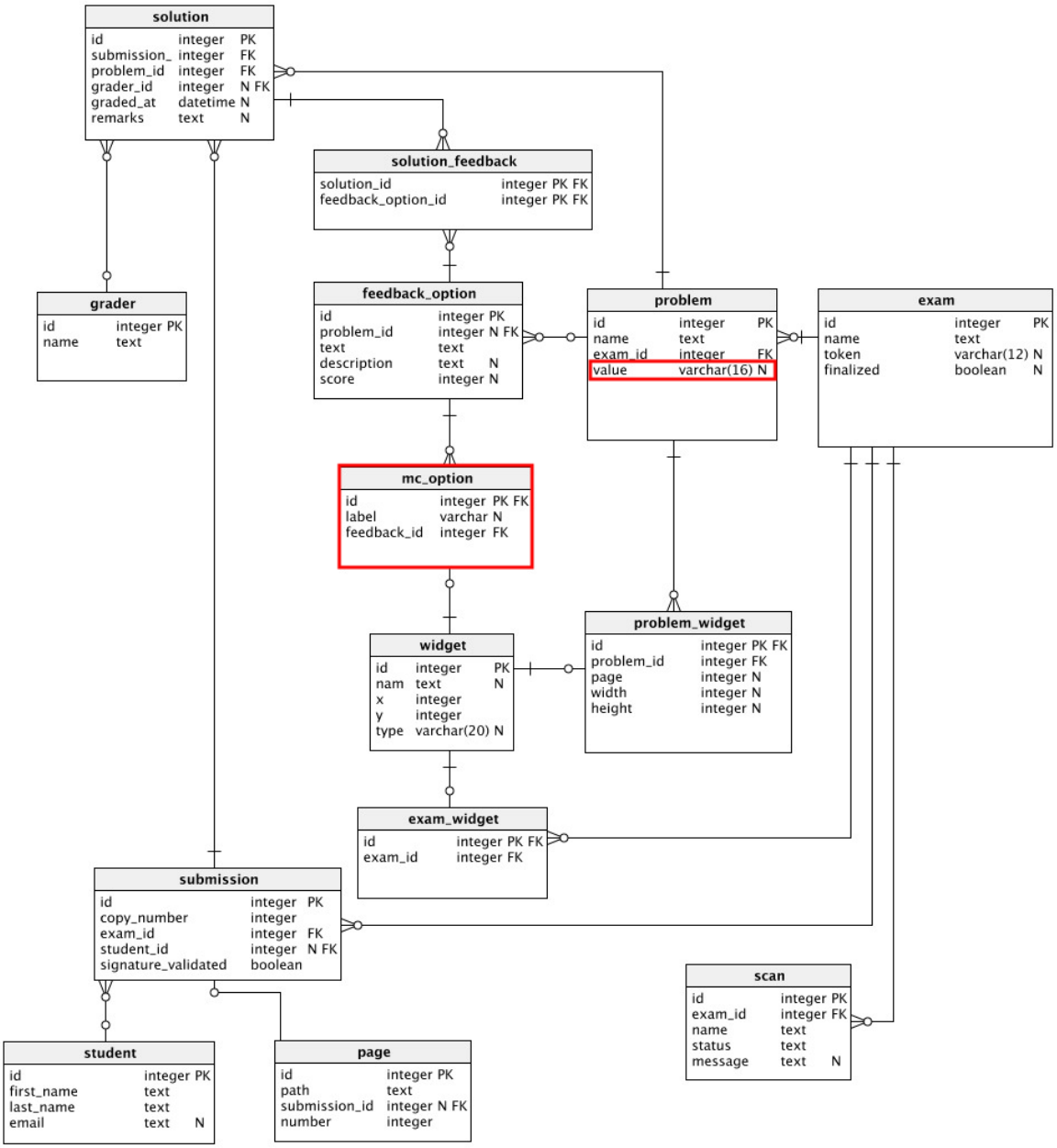


Figure 5.8: The database schema with multiple choice functionality. Additions are enclosed in red boxes.

6

EVALUATION

In Chapter 2 we discussed what our focus was and what we wanted to accomplish. In this chapter, we set out to discuss if we have met our goals, how well we met our goals and reflect back on the process.

6.1. PRODUCT EVALUATION

As mentioned in section 2.3 a series of requirements have been set up have been setup according to the MoSCoW method. In This section, we reflect on if we have achieved these requirements. We do this by providing a table where the first column describes the requirement, the second indicates if it was achieved and final column gives a brief description of the result. For detailed numbers of how well the system performed, see the next section (section 6.2).

As can be seen from Table 6.1, most of the requirements have been met. Our biggest focus was on creating a system that could grade multiple choice questions and we have been able to do this with success. Improving positioning of the scans has been a great contribution to the performance of grading since it means there are less distortions of the image which also makes existing features better.

There were a couple of features we did not implement. These were mainly due to when the team started on the project, Zesje made use of PonyORM. This ORM left a lot to be desired and we planned on improving the existing system. A week or two into the development phase, Zesje made a switch to SQLAlchemy and Celery which added features to fix problems the team wanted to address. Later on in the development process another look was taken into Parallelization and it was discovered that uploading a PDF with a lot of submissions could take over 16 GB. The source of this bug was identified and passed on to the Zesje team and it has been fixed now.

As for the data visualization features, these have been partially implemented. A switch was made to a new plotting library and the existing statistics have been reproduced in the new library. A couple of new prototype statistics have also been made, but not all. Overall, the ground work has been laid and these features can be easily implemented later.

Overall, We consider the development process a success. We have met all the requirements that must be included and most of the should have requirements. Some of the should haves that were not implemented by us did end up being implemented by the other team, so they are still in Zesje. And as for the partial should haves, the ground work has been laid and with a bit more time can also be implemented.

Requirement description	Achieved	Comment
Must have		
Identify blank solutions.	yes	The system can automatically determine which solutions are blank. And enable graders to skip them. See subsection 5.1.7 for more details.
Add the option to add multiple choice problems to the exam.	yes	When uploading an exam, one has a clean UI with many features. The checkboxes are printed on each exam sheet.
Add the option to recognize multiple choice answers.	yes	From the tests of the system, as can be seen in subsection 6.2.4 in detail, there is a very high accuracy.
Connect feedback to multiple choice options.	yes	detected options are automatically linked to feedback.
Add a pre- or auto-grading in the UI.	yes	A grader has the possibility to quickly check and approve detected answers, and can even be set to automatically approve a subset of answers.
Perform benchmarking on image processing.	yes	Accuracy and runtime benchmarks have been done on realigning images, as well as the accuracy of identifying checkboxes. See section 6.2 for more details.
Should have		
improve positioning of the scans based on the markers.	yes	as can be seen in subsection 6.2.2, there is very little difference between a realigned image and the original image.
Implement QR codes on odd pages.	yes	pages are added to exams to make them even.
Add a queue system for uploading scanned PDFs	not by us	This was already implemented when the zesje team switched to Celery and SQLAlchemy.
Parallelization of uploading PDFS.	not by us	Parallelization was already implemented by Zesje using Celery, but work was done on improving memory usage.
Implement data visualization in the back-end to show various statistics	partially	the functionality to receive data for visualization is there, but limited in the number of features
Use data visualization in the front-end.	partially	A switch has been made towards Plot.ly, but so far there are only a couple of visualizations
Add possibility to view another problem of the same student.	no	This was not considered to be important enough by the client later on in the project and focus was put on other aspects
Add question title automatically.	yes	When selecting a problem, the title is automatically found and added. This is done by extracting it out of the original pdf

Table 6.1: The requirements set out at the start of the project allow with whether it has been achieved or not.

6.2. QUALITY OF WORK

This section presents the actions undertaken and measurements made that describe the quality of our work. This consists of static analysis, such as unit tests, benchmarks, and a live test on a public high school.

6.2.1. UNIT TESTS AND CODE COVERAGE

By writing and executing unit tests, one can verify that code does exactly what is expected of it. At the same time unit tests provide a way to see if any code changes do not break existing functionality. Zesje was created by other developers, and although tests were present, not all code was easy to test. For example, API calls were not tested yet, so testing any changes to the API was not possible at the start of the project. After discussing with our coach, we set out to have a degree of coverage of 80%.

Before the start of the project, unit tests were already present, but there was no way to measure test cov-

erage. We decided not to write tests for the entire Zesje project, only for what we wrote ourselves, since it would have taken too much time. Therefore measures were taken to bring the code coverage into perspective using Pytest-cov[1]. Pytest-cov can used to show coverage in a terminal, but it can also export details to an xml file. This file can be used to highlight specific lines of code that are not covered by tests. On Visual Studio, this has been done using Coverage Gutters[2] plug-in which displays colored bars next to each line of code to indicate if it is covered by tests. This helps get coverage up for every line of code.

COVERAGE COMPARISON

In this section, we present A comparison of code coverage has been made as can be seen in Table 6.2. This table covers the Python unit tests for both the API and back-end of Zesje. For a clear overview, only changes between the Zesje master and the WOMM develop branch are shown.

The type of coverage here is statement coverage and the total number of statements can be seen in the second and fifth columns. A statement equates to code that can be executed and the number of statements is a good indicator of the size of the file itself. The misses indicate how many statements are not executed during testing. The coverage is therefore the percentage of statements that are executed during unit testing. At the time of writing (20 June, 2019), there are a total of 31 python files in the develop and 28 in the master. The Zesje branch also has a total test coverage of 50% while the develop branch has 54%.

Another difference between Zesje and WOMM is the number of tests. The Zesje master branch contains 56 different tests. Meanwhile, the WOMM branch contains a hundred tests. New code that was added was not tested only for coverage, but also for many different cases. For example, take the tests related to detecting checkboxes. Running a test where the box in perfectly centered will give you a decent coverage. But adding more cases where the box is to the side or top will cover much of the same statements as a perfect case, but there tests could fail if the implementation does not work well.

A thing to note is that on the Zesje master branch, the API was not tested. Files associated with the API are the files in the first column with `.../api/...` in the name. On our branch (develop), we were able to implement API tests and for some of more complex files, such as the `api/mult_choice.py` file, we were able to achieve 97% code coverage!

Since the table only displays summations of statements and misses, it is hard to quantify the coverage of only the code changed, but the increase in the number of statements is a good indicator. Overall the number of statements increased by 16% while the number of misses only increased by 7%. Taking the total differences in statements and misses between the two branches, we can calculate the percentage of new code covered by tests. We conclude that the total coverage of additions to statements is 78% as can be seen below. This is just slightly below what we set out to achieve.

$$Coverage\_additions = \frac{(1914-1653)-(888-830)}{1914-1653} = 78\%$$

One of the reasons that the total coverage falls below the initial target is that the changes made during the project depended on existing code. The existing code was occasionally hard to test, making the changes hard to test by extension. The API was hard to test initially, and finding out how to create good tests of database and API took several weeks.

[1] documementation can be found at: https://pytest-cov.readthedocs.io
[2] for more details: https://marketplace.visualstudio.com/items?itemName=ryanluker.vscode-coverage-gutters

Name	Works on my machine Statements	Misses	Coverage	Zesje Statements	Misses	Coverage
zesje/__init__.py	18	4	78%	19	6	68%
zesje/api/exams.py	181	125	31%	172	132	23%
zesje/api/feedback.py	53	28	47%	50	34	32%
zesje/api/images.py	39	32	18%	27	21	22%
zesje/api/mult_choice.py	71	2	97%			
zesje/api/problems.py	57	33	42%	49	33	33%
zesje/api/solutions.py	86	69	20%	68	55	19%
zesje/api/widgets.py	24	14	42%	22	17	23%
zesje/pdf_generation.py	124	21	83%	91	10	89%
zesje/pdf_reader.py	50	36	28%			
zesje/pregrader.py	45	5	89%			
zesje/scans.py	342	84	75%	345	87	75%
Total	**1914**	**888**	**54%**	**1653**	**830**	**50%**

Table 6.2: A table overview of the code coverage differences between the Zesje master and the Works on my machine develop branch.

6.2.2. BENCHMARKS

Benchmarking is a great way to quantify the performance of an implementation and determine the quality there of. In this section we present the benchmarks for our work.

IMPROVEMENT OF IMAGE REALIGNMENT

To get the most out of the benchmark for image realignment, the data presented is a measure from both the old system and the new system with the same data set. This data set is obtained from the first exam of the 2018/2019 version of the CSE course Algorithms & Data Structures (CSE1305). Any results published in this document relating to this data set does not use any personal information, and can not be traced back to a student submission even with access to the same data set. There are about 800 student submissions in this dataset. Around 300 of these submissions are processed at a time in the benchmark script. The reason for this is a physical limit, namely not enough memory on the machine doing the benchmark (without using swap).

The old implementation used a mostly in-house implementation that relied on finding the rotation angle of the scan relative to the ideal exam manually between corner markers. Rotation and shifting of the image happened in two separate steps. The newer method leaves most of the work to OpenCV and accomplishes both objectives of rotating and shifting the image to the desired position in a single step.

The benchmarks for both implementations are presented in two dimensions. The first dimension presented is the accuracy of the implementations, and the second is the runtime between the systems.

ACCURACY

The accuracy of the realignment process is defined as the difference between the location of the corner markers as placed on the original PDF and the ones found after realigning the scanned submission. The lower this difference, the more successful the applied method is in realigning the image. The python script used to generate these benchmarks can be found in Appendix E.

Finding the corner markers is done using the `find_corner_marker_keypoints` method that existed before this work. Through a unit test written as part of this project the accuracy of this method is accurate within two pixels on either axis X/Y from the point.[3]

As an example, a few results are displayed for each method, along with the mean error over the *full* processed dataset and the standard deviation, which gives an indication over how large the spread from the mean is. The presented examples are matrices with four rows of x, y pairs, where x, y is the distance between the found corner marker after realignment and the ideal ones, on the respective axis. The results displayed in this subsection are calculated over 300 submissions.

[3] https://gitlab.kwant-project.org/zesje/zesje/merge_requests/164

We first present five example matrices using the Zesje implementation, to give an indication of the error in this version.

$$\begin{bmatrix} 0 & 0 \\ -1 & 0 \\ -7 & 8 \\ -5 & 8 \end{bmatrix} \quad \begin{bmatrix} -1 & 1 \\ -1 & 0 \\ -6 & 9 \\ -4 & 8 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ -3 & -1 \\ -7 & 7 \\ -5 & 7 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 4 & 0 \\ 0 & 0 \\ 4 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ -2 & 0 \\ -7 & 7 \\ -4 & 7 \end{bmatrix}$$

As can be seen, there is usually one marker that is aligned perfectly. However, the others can be all over the place. This accuracy is good enough for grading open questions, as a grader just needs to see the problem field well enough to be able to process the submission. The mean error using this method is **2.956** pixels and the standard deviation is **3.146**.

The new method generates matrices that look more like the following.

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ -1 & 0 \end{bmatrix}$$

Using this method the alignment comes much closer to the perfect case. The mean error is down to **0.092** pixels, and the standard deviation is **0.288**. This accuracy is enough to get a very close alignment between the scan and the original exam. With it, automatically grading multiple choice questions becomes more reliable as now the locations of checkboxes stored during the creation of the exam match the locations of the boxes on the scan. this is a definite improvement over the old system.

RUNTIME

Profiling the runtime of the alignment is done using cProfile, a deterministic profile that is part of the standard python libraries. In the result below, `ncalls` is the number of calls to each function. This is equal to the amount of cases processed. `tottime` is the total time spent in the function itself, excluding sub-functions called by it. `percall` is the amount of time per such call. `cumtime` is more interesting, it is the cumulative time spent during the function and all of its calls to subfunctions. Indeed, this is the time it takes to enter and exit the function, and the best measure of time spent. It should be noted that the time reported by cProfile is in real world seconds, and not CPU time. Due to this, we ran this benchmark ten times. The full results per run can be seen in Appendix E.

The results are shown in Table 6.3. `Zesje` is the old implementation, whereas `WOMM` is the new implementation[4]. It can be seen that the full time spent by the new method is less than that of the old, existing method. This is a nice improvement, the new method executes about **10.5%** faster in these tests. Even though alignment doesn't take much time in the overall workflow of uploading scans, it is good to verify that the new method does not cause severe increases in runtime.

Average time	Method
`Zesje`	7.378
`WOMM`	6.601

Table 6.3: Average time of 10 runs with each run processing 300 images. Full results can be found in Appendix E

6.2.3. USABILITY

No full usability test has been done to test this. The focus during the design however, has always been on reducing the time it takes to grade an exam. This can only be achieved when the system is easy to use. Examples of this would be the way a feedback option is highlighted in the exam creation tool when hovering over

[4] n.b. this naming is not entirely accurate, the pre-existing implementation also relied in part on CV2

its linked multiple choice checkbox. Because the overall the grading time has immensely been reduced, one can argue that the application has become more user friendly. There are also some risks as the pre-grading takes over some of the work from the grader. This should be done with care such that it does not become a black box where the user does not know what is happening.

6.2.4. LIVE TESTING

The pre-grading capabilities of WOMM were tested by using it during a civics [5] course test at a high school. This test consisted of a total of 20 multiple choice questions with 4 options each. The test was used in 3 classes and all were graded using WOMM, resulting in a data set of 80 students.

TESTING

A separate instance of the WOMM that was set up for this test. Students were given a 7 digit code and asked to only sign their work with their initials in order to ensure the privacy of the students. This list was connected to the correct students by a teacher at the school. The list of numbers with initials was uploaded to WOMM. A simple answer sheet was created with just 20 sets of check boxes. All the preparations within WOMM were done by the project team. The grading itself was done by a teacher with assistance from the team.

It took some time to upload the results for the first class. This is probably because the only assistance given was done by phone and the person uploading the test had not used it before. This was reduced to 8 minutes (time from scanner to export) for the final class.

RESULTS

To check the accuracy of the multiple choice grading the results are split in 3 categories.

1. Correctly identified, which counts all the questions correctly graded by WOMM and are equivalent to the amount of true positives combined with the amount of true negatives (TP + TN).
2. False negative, this is any question where pre-grading failed to identify a filled in check box.
3. False positive, these where all the instances where pre-grading identified a box as filled in when in was not. The initial results are as followed.

Category	**Count**
Total amount of questions	1620
Total correctly identified (TP + TN)	1600
False negative (FN)	11
False positive (FP)	9

The accuracy (ACC) of the results can be defined as.

$$\text{ACC} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

This means that the accuracy of the pre-grading amounts to 98.8%.

There are a few methods to increase the accuracy. All false negatives were a result of students using pencil and lightly coloring in the boxes. A simple fix would be to ask students to use a pen. This is particularly important because false negatives are hard to check for as a human grader since they will not be shown automatically. It can also be argued that there are only 4 real false positives since the other 5 where the result of a student drawing on the corner markers. These parameters should be added to the instructions. This would increase the accuracy to 99.7%. but more importantly completely erases false negatives.

To check the effect of pre-grading on the workflow the results were split in four other categories.

1. automatically identified, these are all the problem that were not shown to a human grader.
2. True positives, These were questions where WOMM correctly identified a double input.
3. False positives, These were solutions where only one answer was selected but WOMM identified multiple.
4. Input error these are solutions that were falsely identfied because the input was invalid.

Category	Count
Total amount of questions	1620
Automatically identified	1547
double input (TP)	64
double input (FP)	4
input error	5

Another metric to investigate is the number of questions that had to be approved by a human. In this case that is 4.5%, Which means that grading has been sped up by a factor of 22. There are two ways of reducing this percentage even further. One would be to reduce the number of false positives either by adding new rules for the user as discussed above, or increasing the precision of the pre-grading code. The second would be to convince the students to only write down their final answer e.g. by making an exam and separate answer sheet. Given the distribution this last option would probably have the most impact. It should be noted that false negatives are marked as automatically identified in this scenario since the system cannot flag them.

6.2.5. SIG

During the project, it was required to submit the code to the Software Improvement Group (SIG): once during the 6th week of the project, and once at the end of the 9th week. The feedback from the first submission should be integrated in the project.

Since Zesje was created by other developers, submitting the entire repository for review[6] would mean SIG may provide feedback on code that has not been written by the project team. Although this feedback may be useful for other developers of Zesje, it would not be a meaningful measure of the quality of the code in this project specifically. As a solution, an e-mail was sent asking if SIG could compare two versions of the repository of Zesje. One version would contain the repository as it was right before the start of the project, and the other would include the repository with the changes from the project team. SIG then replied saying that this was indeed possible, and compared the two versions to see how the code has changed since the start of the project, and provided feedback based on the changes. The full feedback received from SIG (in Dutch) is included in Appendix D.

SIG provided a measure of maintainability of the submitted code, and stated that it was averagely maintainable compared to other projects they have evaluate. Additionally, SIG provided feedback on two aspects: *Unit Size* and *Unit Complexity*. SIG also noted that no test code was found in the submission. Although the amount test code in Zesje originally was limited, the remark which stated that no test code had been found did not seem correct.

UNIT SIZE

Unit Size measures the length of pieces of code, in this case methods. SIG notes that code with a high Unit Size may have various reasons, but that it is usually a result of methods containing too much functionality. SIG notes that methods usually start off with a relatively small amount of code, but will expand with time. Additionally, SIG notes that providing comments to separate pieces of code is another indication that a method contains multiple responsibilities.

To reduce the Unit Size of a method, SIG says that it should be split up into multiple smaller methods, and that each smaller method would have a clear and functional scope, and that documentation is easier via method names.

Changes For the functions that had a large Unit Size, the function `pregrader.py:box_is_filled()` contained a large amount of comments. Nearly each line contained a comment to make the code more readable, but some lines of code were simple enough that a comment would be redundant. Therefore, any unnecessary comments have been removed to simplify the function.

[5] In Dutch: *maatschapijleer*
[6] Since Zesje is open-source, submitting the entire repository should not be a problem.

The feedback regarding `Exams._get_single()` was not addressed, since this function was already long before the start of the project, and therefore not sufficiently relevant.

UNIT COMPLEXITY

SIG measures Unit Complexity by checking code that has an above-average complexity. SIG notes that this complexity is usually the result of a piece of code having too many responsibilities or that the implementation of its underlying logic is unnecessarily complex.

To reduce the Unit Complexity of a method, SIG advises to split each method into smaller parts so that each part is easier to understand, test and maintain. By splitting separate functionality in a method with a unique name, each part can be tested separately, and makes the overall flow of the method easier to understand. SIG adds that for small and complex methods, Unit Complexity can be reduced by splitting the problem that is solved in the method into smaller sub-problems, and by putting each sub-problem in a separate method. The original method can then call these new smaller methods and combine their results.

Changes The feedback regarding `images.py:fix_corner_markers()` was fixed by completely changing the entire way corner markers were detected. After some feedback from the client, the team decided that this function was unnecessarily complicated and that the problem could be solved in a much simpler way.

The feedback provided on `pregrader.py:add_feedback_to_solution()` was fixed by simplifying the responsibilities that this function had and by splitting it up in smaller functions.

6.3. PROCESS EVALUATION

In this section presents an evaluation of the software development process chosen by the team and described in section 4.1. The team approached as much as possible the methodology that was agreed upon, however some aspects could have been improved.

6.3.1. SPRINTS

The WOMM team followed weekly sprints as decided at the beginning of the project. Although the team considered eventual delays in the product planning, some items were not completed by the end of the corresponding sprint and had to be shifted to the next.

6.3.2. DIVISION OF TASKS

In terms of the division of tasks, there were no serious issues as the team members invested roughly the same amount of effort and time. Although some team members are now more familiar with a certain part of the application, all members have contributed to both the front-end and to back-end.

6.3.3. COMMUNICATION

There were no major issues in the way the team communicated internally or with the coach or client. Most decisions were discussed thoroughly between the members and at times the client or the coach were asked for their opinion as well. Every member was aware of the overall progress and the tasks the others were working on. The coach and client were periodically updated on the progress of the team and the eventual problems that were encountered.

6.3.4. USE OF GITLAB

The WOMM team made efficient use of the features offered by Git and GitLab. To make good use of the issue trackers but also to differentiate the team's efforts from the changes brought by the current Zesje developers, the team decided to create a separate fork. On the WOMM fork, the team made the master branch mirror the Zesje branch at all times. A branch was created called 'develop' which was the branch where all the separate features were merged into to test for compatibility. Every new branch was created based on the latest commit on the master and once it was finished a merge request was opened directly to Zesje's master. The idea behind this approach was to have the benefits of a separate fork but at the same time prevent the two forks from deviating too much.

Unfortunately, many features were related to each other in such a way that it was difficult to create a merge request for every feature. For instance, the front-end needs to allow the creation of multiple choice checkboxes in order to make it possible to test the code responsible for detecting if a checkbox is filled. In order to have that feature in the front-end, the API and the database must be changed such that they store and serve the locations of multiple choice widgets. Because the client preferred only complete features in the master, the team did not attempt to merge intermediate states of the features. For instance, there was no point in having the front-end allow the creation of checkboxes on the PDF if the automatic pre-grading feature did not work. Some features were independent and were indeed merged separately, but the automatic pre-grading of multiple choices feature resulted in a large merge request.

6.3.5. CODE REVIEW

As agreed at the start of the project, every feature was developed on a separate branch and was reviewed and approved by other team members before it was merged. Additionally, most merge requests were reviewed by the client and the other developers of Zesje. This means that all code contributed throughout the project was thoroughly reviewed and met the demands of the client.

The client's reviews were often related to relatively small issues that were fixed immediately. At times however, the client did requested changes that costed a lot of time, the team ultimately ran into delays trying to resolve them. The feedback of the client also helped the team prioritize what items where important for the following sprint.

7

Discussion and Recommendations

This chapter provides a discussion on the result of the project, and a consideration of the ethical implications of our changes to a product such as Zesje. We touch on the security of the system, and make our recommendations for future endeavours with this system.

7.1. Discussion

In this section, we present a high level discussion on the consequences of this work and Zesje in general.

7.1.1. Ethical Considerations

Zesje is a system that will process a lot of user data. Teachers need to upload their exams, graders need to store their feedback inside the system, and students will have their exam submissions scanned and kept in the database. Therefore, everyone involved in the exam process is in some way involved with it. Presented below are a few ideas worth taking into account for an ethical consideration of this product.

Privacy

Automating the grading process may be easier for teachers, but it also introduces several privacy problems. In an automated tool like Zesje, it is extremely easy to use a screen capture tool to obtain confidential student data.

This is a clear disadvantage of automated grading when compared to grading exams on paper. With paper exams, illicitly obtaining valuable student data for a TA can only be done by taking a screenshot with a phone or stealing an exam. Taking a screenshot with a phone is unlikely to go unnoticed by a teacher, and stealing an exam is easier to find out for a teacher than whether a TA took a screen capture of an exam.

One recommendation here is to ensure graders will have to sign an NDA, which means they can be held accountable if they decide to steal confidential exam data. Another recommendation is to ensure grading is anonymous, meaning that TAs cannot directly get to know about which student they are grading.

Accountability of results

Although the goal of Zesje is to reduce the time it takes to grade exams, automation of the grading process means that some parts of grading can be performed without any human supervision. In the case of an exam with a large amount of students, it could be entirely possible that some solutions that are automatically graded by Zesje. For example, if a large amount of solutions are identified as blank, they may never be seen by any grader. The same can apply to multiple choice questions, depending on the grading policy used.

This introduces a larger problem, if nobody will look at an answer, a system for Zesje might be responsible for the grades that students receive. If any answers are incorrectly marked by Zesje, students may receive incorrect grades.

One solution is that Zesje provides the possibility to send students emails with their personalized feedback.

This would however mean that sending personalized feedback has to be required to ensure a small amount of mistakes. So any 'mistakes' made by Zesje can be caught by students. However, this would place the burden on students to ensure that the results they get from their exams are correct. This raises the question of how much a student is responsible for ensuring that their exams are graded correctly, compared to teachers. Students may also not be inclined to report possible mistakes if it is to their advantage.

However, with paper based exams, this problem of accountability is still present. With paper-based exams, teachers need to schedule a day after all exams are graded for students to view their graded exams. An advantage that Zesje provides is that teachers can easily send e-mails to all students with their personalized feedback. This makes it easier for students to view their feedback in the first place, and might provide a better incentive to check the feedback they have received. If students do not show up to check the results exams, they would also not know if their teacher has made a mistake.

We recommend that teachers should clearly make it known to their students that their exams are graded automatically, and that they will need to check if the feedback they received is correct. Another measure to combat this problem is to provide a metric of confidence in how likely computer vision parts are correct. This makes it possible for teachers to see whether they should manually check an answer that has been graded automatically.

7.1.2. CYBER SECURITY

In the online world, attacks by malicious actors are frequent. Some instances of Zesje are hosted on public websites for ease of access. There are several issues with this, that we want to briefly touch upon despite them not being part of our work per se. Both these issues rely on an instance being openly accessible.

DDOS ATTACKS

Opening up Zesje to the internet by hosting it on the servers of TU Delft allows any TA authorized to grade to grade, regardless of where he or she is. But this also brings risks with it.
One potential risk of this is that the system is vulnerable to DDOS attacks. DDOS attacks would overload the hosting servers and prevent anyone to interact with Zesje, preventing TAs and graders from grading exams. DDOS attacks already often happen against the learning tools of highschools. Therefore, it is recommended that there are measures taken together with the server hosts to prevent mitigate such attacks. A benefit of Zesje is that it can also be run on a local laptop. So if a public instance is unavailable, grading can be continued locally. But if Zesje happens to be down and users want to continue grading, they would not want to start all over again, and likely would not have a full copy of the database. Therefore it is recommended that there is a system that keeps backups of the files. This provides redundancy for cases where data may be lost or corrupted.

7.1.3. API MISUSE

There is no authentication on the back-end API. This leaves it very vulnerable to outside attacks, that can mess with the state of the system. This is not an easy problem to fix either, as Zesje is designed to be a minimum working prototype. Making Zesje resilient to these kinds of attack would require adding an authentication layer for both the back-end, possibly through tokens, and the front-end, possibly through Single-Sign-On.

7.2. RECOMMENDATIONS

A lot of functionality is implemented in this thesis. However, there are always things that had to be left out. These are discussed in this section as recommendations for future work. Also, some ideas presented here are out of scope for this thesis, yet could be found interesting enough to warrant an investigation for the future.

7.2.1. PROJECT VISION AND DESIRED AUDIENCE

Zesje is a content rich tool that offers a lot of flexibility by allowing any type of A4 PDF to be turned into an exam. Currently, using Zesje often requires computer knowledge outside the realm of most teaching staff. This can be a roadblock to entry. As an alternative, users can request an instance of Zesje specifically for them through contacting the Zesje team. However, this still requires human intervention and more importantly, does not scale.

In our eyes, Zesje could be transformed into a tool that has access control such that multiple teachers could create their own exams without being able to see each other's. This instance can be hosted on a central server for the entire technical university to enjoy. In this way, there is a central instance locked under the university's Single Sign On system that provides access control.

However, this would require a drastic change in the way Zesje works. Applying a layer of access control stretches beyond putting the system behind SSO, there would preferably be some role management such that some users can have more privileges than others. For example, a teacher might want to make an exam and grade it (partially) later, whereas some course staff should only be able to grade exams they are assigned to. Furthermore, not all exams and submissions should be visible by everyone, but only by those related to the course.

Aside from scaling and both constant and instant availability, another advantage is that this way an archive can be built up over multiple iterations of the course across the years. Coupled with the statistics tooling this could provide insight in the development of the course over the years.

We recommend that this should be done by another project group.

7.2.2. DATABASE TECHNOLOGY

Although SQLite does its job, it has a number of limitations that had a negative impact on the development process. The main limitation was that database tables could not simply be altered. This meant that changes to the database tables means that the tables had to be copied and recreated, which is not a very efficient process for a relatively small change. We propose that another group could look into moving towards MySQL or another relational database.

7.2.3. DEBUGGING TOOLS

Debugging is an important process in making good quality code and as mentioned in section 4.2 of Chapter 4, we made use of a number of ways to debug Zesje. Debugging a full Webserver and database is not as simple as normal applications and we would like to therefore share the methods we have found to do this. Zesje does have a development mode where the system is started up and any changes made to the code will automatically be reloaded, but it does not allow one to pause the system with break points. Break points are a crucial part of debugging as it allows one to see exactly what is happening in the code in intermediate steps instead of only seeing the end result. We propose that the documentation of Zesje is expanded with a more in depth look at how to debug using break points. This will not only help developers who are new to Zesje, but existing developers might also find these methods useful.

7.2.4. MULTIPLE CHOICE EXTENSIONS

Although Zesje has been extended with multiple choice question functionality, this functionality could still be improved.

GRADING SCHEME

Currently, it is not possible for teachers to manually create a grading scheme. If a question is identified as blank, a student gets 0 points. Otherwise a student gets the points for all the options that have been filled in. A grading scheme has not been implemented throughout this project since creating a grading scheme can be quite complex. Some teachers want to give 0 points for an answer, while other may want to deduct points. One recommendation for future work is to create the possibility for teachers to make such a grading scheme.

SUPPORT FOR OTHER FORMATS

The implementation provided in this work is sufficient and unambiguous, however the format of multiple choice questions used differs between teachers. Also, it may be the case that a teacher has already created their own answer boxes on an exam. We cannot ask from these teachers to rewrite their exams. Zesje should work with any PDF of the right format (A4 paper). An improvement over our method is to let the teacher select their own answer boxes on their exam.

GRID PLACEMENT OF CHECKBOX

To create a 'slicker' look of the location of the answer boxes, especially when placed close to each other, the movement of the location of these answer boxes could be placed on a grid or a similar solution. The problem being solved is that currently these boxes can be placed as freely as possible. However, this introduces misalignment between boxes on the same page; although not impossible, it will be hard to get the multiple choice answer boxes for two questions to line up directly above or next to each other. Therefore, limiting the placement of these boxes could aid in creating a better and more professional look for the overall exam.

8

Conclusion

The aim of this project is to enhance the grading tool Zesje with new features to automatically grade certain types of questions and submissions. A two-week research phase was done to evaluate realistic design goals, and decide what on what features to implement, and what to consider less important. What followed is an six-week implementation phase to bring these goals to life.

In this stage prototypes were made and tests were set up, including a mid term exam during the halfway stage and a high school exam nearing the final week. These efforts ensure that the new features work as well as is required.

What has been delivered is a dedicated method to define multiple choice questions on an exam, which provides the user with flexibility over the policy to grade these with. Due to an improved realignment method we can provide the accuracy required to detect which answers are filled in can be provided, capable of saving teachers countless of hours of effort. Question titles can be automatically filled in, and adding feedback options is easier now than it was before.

Zesje still remains prototype software. It solves the problems at hand, but steps can still be made to improve the user experience when setting up Zesje. Efforts to put Zesje on a pubic website with access control can turn the tool into a great product of the TU Delft. However, this has not been the focus of this project, and we make no claim to this.

We can definitely say that Zesje is now a richer product than before, the main goals of the project are achieved, and users can benefit from these improvements straightaway.

BIBLIOGRAPHY

[1] Connie van Uffelen. International run on computer science and engineering, Jan 2018. URL https://www.delta.tudelft.nl/article/international-run-computer-science-and-engineering.

[2] Saskia Bonger. Computer sciences remains popular, Feb 2019. URL https://www.delta.tudelft.nl/article/computer-sciences-remains-popular.

[3] Brian Harrington, Marzieh Ahmadzadeh, Nick Cheng, Eric Heqi Wang, and Vladimir Efimov. Ta marking parties: Worth the price of pizza? In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 232–240. ACM, 2018.

[4] Stefan Hugtenburg. On the creation of exams and the nightmares of grading them. *Machazine*, 23(2): 16–17, Feb 2019.

[5] Connie van Uffelen. Grading more systematic and less time-consuming, Jan 2018. URL https://www.delta.tudelft.nl/article/grading-more-systematic-and-less-time-consuming.

[6] Nick Cleintuar, Justin van der Krieken, and Jamy Mahabier. Zesje: Web-based paper exam grading system, 2018.

[7] Joseph Weston. Low effort exam grading, Sep 2017. URL https://quantumtinkerer.tudelft.nl/blog/zesje/.

[8] Fredrik Lundh. Pillow documentation. URL https://pillow.readthedocs.io/en/stable/about.html. Accessed on 30 April 2019.

[9] S Vijayarani and Ms A Sakila. PERFORMANCE COMPARISON OF OCR TOOLS. *International Journal of UbiComp*, 6(3):19–30, 2015. doi: 10.5121/iju.2015.6303.

[10] Thomas M Breuel. The ocropus open source ocr system. In *Document Recognition and Retrieval XV*, volume 6815, page 68150F. International Society for Optics and Photonics, 2008.

[11] Julian Ibarz. Updating Google Maps with Deep Learning and Street View, May 2017. URL https://ai.googleblog.com/2017/05/updating-google-maps-with-deep-learning.html.

[12] Cuda installation on windows. URL https://developer.nvidia.com/cuda-downloads?target_os=Windows&target_arch=x86_64&target_version=10&target_type=exelocal.

[13] Pranob K Charles, V Harish, M Swathi, and CH Deepthi. A review on the various techniques used for optical character recognition. *International Journal of Engineering Research and Applications*, 2(1):659–662, 2012.

A

MoSCoW Requirements

This chapter contains the requirements devised at the start of the project prioritized according to the MoSCoW model.

Must haves

- Add the possibility in Zesje to identify blank solutions.
- Automatically pre-grade multiple choice questions. Show the grader the answer that Zesje identified, and let the user perform check to see if Zesje has correctly identified the option that the grader filled in.
- Provide systematic image processing pipeline performance benchmarking, optimize it within reason.
- When selecting a question in a new exam, a grader shall be able to add feedback options before the exam is graded.

Should haves

- The grader shall be able to view multiple test heuristics. For example: average time spent per question, an estimate on total time needed to grade an exam, time spent grading by a TA; or the total time spent on exam.
- Precise positioning of the scans based on the markers.
- Zesje should be able to use computer vision techniques to provide an estimate of the amount of words in written answers.
- When delimiting the problems of a new exam, have the grader select the problem title. Zesje should read the contents and automatically assign the name of the problem to the corresponding box.
- Show how many people have passed the test or have a certain amount of points.
- Add a feature for a grader to view another problem of the same student while grading in the same exam.
- If an exam has odd number of pages, insert an extra one with a QR code.
- Show how much of the total exam has been graded, both per question and in total.
- When a teacher uploads multiple PDFs, the PDFs should be placed in a queue.
- Processing scanned PDFs in Zesje should be done in parallel.

Could haves

- Apply a heuristic to the exam templates to automatically mark the answer boxes.
- Use OCR techniques to read handwritten numbers for answers that only require the student to write down a number.
- Word count for answers that are written outside of the bounding box of a question, by using manual selection.
- Recovery system or redundancy for QR codes in case of damaged, removed or modified QR codes. In case a submission is rejected, there should be a way to recover the submissions that were rejected and assign exams manually to students.
- Add OCR to read the student name in the specified box, and add a check to see if student number and name match.

- If any error occurs during grading, graders should be shown an error that provides more information on what exactly went wrong.
- Collision avoidance if multiple graders are grading the same exam. Zesje should show a message that another grader is grading the question that is being viewed.
- Allow order of questions to be changed.

Won't haves

- Adding a role management system to Zesje.
- Add an option in the user interface to select/navigate by feedback assigned.
- Add an option in the user interface to split existing feedback.
- Calculate the average of grades given by multiple graders. If those grades are too far apart, the user should be notified.
- Use Zesje to grade other types of activities such as presentations, homeworks, and reports.
- Write documentation for the technically illiterate such as tooltips.
- Use another database such as MySQL instead of SQLite.
- Add more tests, and check for e.g. mocking, for pre-existing features.
- Use different ORM.
- Improve on current continuous integration (CI) functionality.
- Add logging to existing features.

B

Project Description BEPSys

Computer vision for exam grading

Provided by Zesje for the BEP Computer Science & Engineering 2018/2019 Q4 course.

Project Description

You are going to join the team developing the open source software Zesje [1] for online exam grading. Zesje is developed within TU Delft and currently used for grading by several courses within TNW and EEMCS faculties. It's even possible that you already received feedback about your exam provided by Zesje.

In this project you are going to develop automated tools for processing scanned exams, namely:

- Precise positioning of the scans based on the markers
- Identify blank solutions
- Apply a heuristic to the exam templates to automatically mark the answer boxes
- Automatically pre-grade multiple choice questions
- Explore OCR of handwritten numbers
- Within this investigation you will also work with large datasets of scanned solutions and develop statistical testing for the robustness of your algorithms.

In this project you will learn full stack development, learn basic computer vision libraries in Python, deal with React frontend framework, contribute to open source software, and help improve teaching in your university. Because Zesje is actively used, you will also have an opportunity to directly interact with the users and test the new functionality "in the field".

Zesje

The team that develops the open source software Zesje for online exam grading. Zesje is developed within TU Delft and currently used for grading by several courses within TNW and EEMCS faculties.

[1] https://gitlab.kwant-project.org/zesje/zesje/

C

INFO SHEET

Title of project: Computer Vision for Exam Grading
Name of the client organization: TU Delft
Date of final presentation: July 2nd, 2019
Description:
Zesje was created by Anton Akhmerov, our client, as an open source web application meant to simplify and streamline the grading process. The main challenge was to improve Zesje such that it can align images precisely, identify blank solutions and automatically pre-grade multiple choice questions. A lot of research has been conducted in comparing frameworks and technologies that can aid the team in achieving the aforementioned goals. The team decided to work with the Agile methodology with weekly sprints and weekly meetings with the coach and client. Although the team took eventual delays into consideration when planning the project, some tasks were shifted to other weeks for mostly external reasons. The features that the team implemented in Zesje work well and are tested thoroughly using real test data from multiple exams. Although the product will most probably be used by teachers that are knowledgeable in the field of Computer Science, it might be somewhat difficult for non-technical users to use the application as it is difficult to set up. One of the team's recommendations is to consider the end user as someone that does necessarily have a technical background.
Members of the project team:
Name: Robin Bijl
Interests: Machine Learning, Python, Computer Vision
Contributions: Checkbox placing and detecting, image realignment, Unit tests
Name: Timotei Jugariu
Interests: Embedded Systems, Operating Systems, Web development, Algorithm Design
Contributions: Front-end, API, API Tests
Name: Hidde Leistra
Interests: Low level, security, electronics, Linux
Contributions: Feedback options in exam editor, image realignment, grading policy, benchmarks
Name: Richard van de Kuilen
Interests: back-end development, Linux
Contributions: pre-grade workflow, blank solution identification, odd number of pages
Name: Ruben Young On
Interests: Theoretical computer science, ethics in computer science
Contributions: API and database, image realignment, front-end
All team members contributed to the project planning, research report, final report and final presentation.
Client & Coach
Name and affiliation of the client: Anton Akhmerov, TU Delft
Name and affiliation of the coach: Stefan Hugtenburg, Distributed Systems Group, TU Delft
Contacts

Anton Akhmerov	Client	hi@antonakhmerov.org
Stefan Hugtenburg	Team coach	S.Hugtenburg@tudelft.nl

The final report for this project can be found at: http://repository.tudelft.nl

D

SIG FEEDBACK

This appendix includes the feedback received from the SIG after it was submitted in weeks 6 and 9.

WEEK 6:

Zoals besproken hebben we voor de feedback alleen gekeken naar de code die tussen de eerste en tweede is aangeraakt. De code van het systeem scoort 3.7 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Size en Unit Complexity vanwege de lagere deelscores als mogelijke verbeterpunten. Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen. Voorbeelden in jullie project:

```
Exams._get_single(exam_id)
pregrader.py:box_is_filled
 (box  page_img  corner _ keypoints,  marker_margin threshold cut_padding box_size )
```

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Dit betekent overigens niet noodzakelijkerwijs dat de functionaliteit zelf complex is: vaak ontstaat dit soort complexiteit per ongeluk omdat de methode te veel verantwoordelijkheden bevat, of doordat de implementatie van de logica onnodig complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is, en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een beschrijvende naam kan elk van de onderdelen apart getest worden, en wordt de overall flow van de methode makkelijker te begrijpen. Bij grote en complexe methodes kan dit gedaan worden door het probleem dat in de methode wordtd opgelost in deelproblemen te splitsen, en elk deelprobleem in een eigen methode onder te brengen. De oorspronkelijke methode kan vervolgens deze nieuwe methodes aanroepen, en de uitkomsten combineren tot het uiteindelijke resultaat. Voorbeelden in jullie project:

images.py:fix_corner_markers(corner_keypoints,shape)
pregrader.py:add_feedback_to_solution(sub,exam,page,page_img,corner_keypoints)

Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen. Op lange termijn maakt de aanwezigheid van unit tests je code ook flexibeler, omdat aanpassingen kunnen worden doorgevoerd zonder de stabiliteit in gevaar te brengen. Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

WEEK 9

No feedback received as of yet.

E

Realignment Benchmark Data

Runtime Benchmarks

`process_via_cv2` is the runtime from this work, `process_via_traditional` is the pre-existing implementation. This data is filtered to only show the runtime of those functions. This output was generated through: `for n in 1..10; do python -m cProfile benchmark_positioning.py | grep process_via » bmout.txt; done`

```
   ncalls  tottime  percall  cumtime  percall (function)
      300    0.007    0.000    6.957    0.023 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.003    0.000    6.276    0.021 benchmark_positioning.py:27(
          process_via_cv2)
      300    0.007    0.000    7.758    0.026 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.705    0.022 benchmark_positioning.py:27(
          process_via_cv2)
      300    0.007    0.000    7.603    0.025 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.578    0.022 benchmark_positioning.py:27(
          process_via_cv2)
      300    0.007    0.000    7.359    0.025 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.663    0.022 benchmark_positioning.py:27(
          process_via_cv2)
      300    0.007    0.000    7.275    0.024 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.486    0.022 benchmark_positioning.py:27(
          process_via_cv2)
      300    0.008    0.000    7.396    0.025 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.655    0.022 benchmark_positioning.py:27(
          process_via_cv2)
      300    0.008    0.000    7.281    0.024 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.615    0.022 benchmark_positioning.py:27(
          process_via_cv2)
      300    0.007    0.000    7.275    0.024 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.605    0.022 benchmark_positioning.py:27(
          process_via_cv2)
```

```
      300    0.007    0.000    7.301    0.024 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.641    0.022 benchmark_positioning.py:27(
          process_via_cv2)
      300    0.007    0.000    7.575    0.025 benchmark_positioning.py:16(
          process_via_traditional)
      300    0.002    0.000    6.781    0.023 benchmark_positioning.py:27(
          process_via_cv2)
```

Benchmark Code (Python)

```
'''
    Run this file via:
        python -m cProfile benchmark_positioning.py | grep process_via
'''
import sys
import os
import sys
import numpy as np
import PIL
import glob
import zesje.zesje.scans as scans
import zesje.zesje.images as images
import realign as realign


def process_via_traditional(image):
    corner_keypoints = scans.find_corner_marker_keypoints(image)
    try:
        scans.check_corner_keypoints(image, corner_keypoints)
    except RuntimeError as e:
        return f"Messed␣up:␣{e}"
    else:
        (image_array, new_keypoints) = realign.rotate_image(image, corner_keypoints)
        return realign.shift_image(image_array, new_keypoints)


def process_via_cv2(image):
    return realign.realign_image(image)


def get_keypoints(image):
    keypoints = scans.find_corner_marker_keypoints(image)
    return keypoints


# Get current directory
path = os.path.dirname(f"{os.getcwd()}/{sys.argv[0]}")
test_path = f"{path}/testdata/"
test_path = "/home/hidde/zesje_beun/testdata/"
print(f"Finding␣files␣in␣{test_path}...")
iterator = glob.glob(test_path + "*/*.jpg", recursive=True)
print(f"Found␣{len(list(iterator))}␣files...")

files = []
count = 0
```

```
size = 50
skipped = 0
for image_path in iterator:
    pil_image = PIL.Image.open(image_path)
    pil_image = pil_image.convert('RGB')
    image = np.array(pil_image)
    corner_markers = scans.find_corner_marker_keypoints(image)
    try:
        if len(corner_markers) != 4:
            raise RuntimeError
        scans.check_corner_keypoints(image, corner_markers)
    except RuntimeError:
        skipped = skipped + 1
        print(f"Skipping {image_path}")
        continue

    files.append({'image': image, 'corners': corner_markers})
    count = count + 1
    print(f"\rLoading {count} of {size}... ", end="", flush=True)
    if count == size:
        print(f"done!")
        print(f"Skipped {skipped} entries due to not having a testable amount of
            corner markers (not 3 or 4)")
        break

count = 0
avg = 0
cv2_diff = []
traditional_diff = []
for im in files:
    image = im['image']
    perfect_corner_markers = scans.original_corner_markers('A4', 200)
    image_cv2 = process_via_cv2(image)
    image_traditional = process_via_traditional(image)


    try:
        traditional_corner_markers = get_keypoints(image_traditional)
    except UnboundLocalError:
        im2 = PIL.Image.fromarray(image_traditional)
        im2.save("out.jpg")
        break
    cv2_corner_markers = get_keypoints(image_cv2)
    if len(traditional_corner_markers) > 4 or len(traditional_corner_markers) < 3:
        continue
    diff = np.subtract(perfect_corner_markers, traditional_corner_markers)
    traditional_diff.append(diff)

    diff = np.subtract(perfect_corner_markers, cv2_corner_markers)
    cv2_diff.append(diff)

    count = count + 1

    print(f"\rProcessing {count} of {len(files)}... ", end="", flush=True)
print("")
print(f"Final amount of cases considered: {count}")
```

```
print(cv2_diff)
print(traditional_diff)
print("Done benchmarking.")
print(f"CV2 mean: {np.mean(np.abs(cv2_diff))}")
print(f"CV2 std: {np.std(np.abs(cv2_diff))}")
print(f"Traditional mean: {np.mean(np.abs(traditional_diff))}")
print(f"Traditional std: {np.std(np.abs(traditional_diff))}")
```

F

MEMORY PROFILER RESULT

This appendix includes the result of using a memory profiler on PDF processing from subsection 5.1.8.

Filename: process_pdf_test.py

```
Line #    Mem usage    Increment  Line Contents
================================================
    22     37.7 MiB     37.7 MiB   @profile(stream=fp)
    23                             def extract_images(filename):
    31     37.7 MiB      0.0 MiB       with open(filename, "rb") as file:
    32     37.7 MiB      0.0 MiB           use_wand = False
    33     37.7 MiB      0.0 MiB           pypdf_reader = None
    34     37.7 MiB      0.0 MiB           wand_image = None
    35     37.7 MiB      0.0 MiB           total = 0
    36
    37     37.7 MiB      0.0 MiB           try:
    38     37.7 MiB      0.0 MiB               pypdf_reader = PyPDF2.PdfFileReader(file)
    39     37.8 MiB      0.1 MiB               total = pypdf_reader.getNumPages()
    40                                     except Exception:
    41                                         # Fallback to Wand if opening the PDF with
    PyPDF2 failed
    42                                         use_wand = True
    43
    44     37.8 MiB      0.0 MiB           if use_wand:
    45                                         # If PyPDF2 failed we need Wand to count the
    number of pages
    46                                         wand_image = WandImage(filename=filename,
    resolution=300)
    47                                         total = len(wand_image.sequence)
    48
    49     37.8 MiB      0.0 MiB           for pagenr in range(total):
    50     37.8 MiB      0.0 MiB               if not use_wand:
    51     37.8 MiB      0.0 MiB                   try:
    52                                                 # Try to use PyPDF2, but catch any
    error it raises
    53     37.8 MiB      0.0 MiB                     img = extract_image_pypdf(pagenr,
    pypdf_reader)
    54
    55     37.8 MiB      0.0 MiB                   except Exception:
    56                                                 # Fallback to Wand if extracting with
    PyPDF2 failed
    57     37.8 MiB      0.0 MiB                     use_wand = True
    58
    59     37.8 MiB      0.0 MiB               if use_wand:
    60     37.8 MiB      0.0 MiB                   if wand_image is None:
```

```
61   1367.4 MiB   1329.6 MiB                    wand_image = WandImage(filename=
     filename, resolution=300)
62   1377.0 MiB      9.6 MiB                img = extract_image_wand(pagenr, wand_image
     )
63
64   1377.0 MiB      0.0 MiB            if img.mode == 'L':
65   1410.0 MiB     33.0 MiB                img = img.convert('RGB')
66
67   1410.0 MiB      0.0 MiB            return img, pagenr+1
68
69                                  if wand_image is not None:
70                                      wand_image.close()
```