Zesje

Web-based paper exam grading system

Nick Cleintuar
Justin van der Krieken
Jamy Mahabier

TU Delft

Zesje

Web-based paper exam grading system

by

Nick Cleintuar
Justin van der Krieken
Jamy Mahabier

to obtain the degree of Bachelor of Science

at the Delft University of Technology,

to be defended publicly on Tuesday July 3, 2018 at 11:00.

Project duration:	April 23, 2018 – July 3, 2018	
Project committee:	A.R. Akhmerov	TU Delft, client
	M.F. Aniche	TU Delft, coach
	H. Wang	TU Delft, coordinator

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

TU Delft

Preface

This document is the final report for our bachelor end project of the Bachelor of Science in Computer Science at the Delft University of Technology. The project consisted of a research phase of two weeks, followed by an implementation phase of eight weeks. The report includes various details about the development on the already existing system called Zesje, an online exam grading tool for paper-based exams. It details the various steps and decisions taken to add the functionality for teachers to create compatible exams using Zesje.

We express our gratitude to a number of people for making this project possible and supporting us throughout: the Zesje team, consisting of Anton Akhmerov (our client), Thomas Roos and Joseph Weston (listed in alphabetical order); and our coach, Maurício Aniche.

Nick Cleintuar
Justin van der Krieken
Jamy Mahabier

Delft, June 2018

Summary

Grading can be a very time-consuming activity for teachers. For this reason, numerous tools exist to aid teachers in grading. One of these tools is Zesje, a web application that allows electronic grading of paper-based exams.

A major drawback of Zesje was that teachers were required to create their exams with LaTeX. Having to use LaTeX meant two things: not being able to use other software to create exams and poor performance when compiling the LaTeX source.

The goal of the project was to deprecate the LaTeX template from the software stack of Zesje, so that instructors would not be forced to use LaTeX and the Zesje template, and so Zesje would be easier to maintain.

To accomplish this goal, a set of requirements and design goals has been produced in collaboration with the client. Research has been conducted to investigate the requirements and what would be needed to complete them. After proposing the initial design to the client, development started. The team adhered to a development methodology to streamline the development process. This meant, among other things, that weekly feedback sessions with the client had been held. This gave the client the opportunity to steer the direction of development to satisfy their wishes and amend the planning where needed.

The team faced and has overcome several challenges regarding library compatibility and computer vision. Grading also introduces a number of ethical considerations around the privacy of students and bias in grading. A number of recommendations for future work are given in the areas of maintainability, performance and ethics.

At the end of the project, the required functionality to use arbitrary PDFs was successfully delivered. This functionality was implemented within the existing Zesje application. An end-to-end test showed that when using a high-volume scanner, the system including all newly implemented functionality works. This means that the main goal has been achieved. The client is satisfied with the resulting product, successfully completing the project.

Contents

1 Introduction

Exam period in education is a regularly occuring period, in which both students and course staff have an increased workload. Students have to study for days, sometimes even weeks, for an exam to make sure that they get a passing grade. Instructors and teaching assistants, on the other hand, have to go through hundreds of exams to grade them. This makes this period busy for both parties. However, it goes without saying that exams are necessary. This is because exams are one of the many ways to test on and to give feedback about the student's understanding of the course material [1].

Not only does grading a large amount of paper-based exams take time, it is also inflexible in that every exam made by a student only has a single copy, which introduces logistical overhead. Many approaches to grading have been introduced to scale better with the amount of students. Examples include using forms that can be automatically processed ('Scantron') for multiple-choice questions or giving an exam digitally (such as programming exams for Computer Science students). The reality, however, is that many classes still make use of paper-based exams for examination.

Zesje is a web application that attempts to aid in grading paper-based exams. Zesje aims to only change the way an exam is graded, so that the examination format does not have to change. This is logistically beneficial, as it is not necessary to ensure the availability of one computer per student during examinations. This has several benefits over classical paper-based exam grading such as:

- a decreased overhead in logistics if an exam is graded by multiple graders
- a way to change grading rubrics after the fact without having to go back through all previously graded exams
- more consistent grading as a grader only sees a single answer and not the whole exam
- easier to give in-depth feedback because if students give similar answers feedback can be reused automatically
- as all feedback is stored digitally, statistical analysis is possible with little overhead

Most of advantages are not impossible to obtain without Zesje, but Zesje does make it much easier. However, one major problem has been a thorn in the eye of both the Zesje developers and teachers who would like to use Zesje for their exams: the necessity of the Zesje LaTeX template. A more exact definition and analysis of this problem can be found in chapter 2. The research done to solve this problem, which outlines the choices made between algorithms and libraries, is written in chapter 3. In chapter 4 can be found what methods were used for the development of the solution. Chapter 5 contains the design of our solution, followed by the actual implementation of the solution in chapter 6. How the quality of the implementation was checked, is detailed in chapter 7. The discussion on

various challenges, ethical considerations and possible future work is described in chapter 8. Finally, a conclusion on the final product is given in chapter 9.

2

Problem definition and analysis

This chapter gives more insight into the problem surrounding Zesje. It does this by first giving some more background information about Zesje. Next, a more concrete definition of the problem is given. Lastly, this chapter contains an analysis of the problem.

2.1. Background

Zesje is an online exam grading tool originally created by Anton Ahkmerov and Joseph Weston. At that time, Anton was lecturing a course of more than 200 students, which had a paper-based mini-exam every week. The grading of these exams, however, was something that the team behind the course was not looking forward to. To improve the process of grading, Zesje was created so that these paper-based exams can be graded via a web application.

The first iteration of Zesje was 'hacked' in a couple of days, and therefore the code was not of the best quality. However, the project was a success: grading scanned exams felt easier and more insightful that grading exams by paper. It was more advantageous for both the course team and the students. Teachers and teaching assistants saved time as grading went faster, while students also had the ability to receive more personalized feedback. A second iteration of Zesje was created to improve on the flaws in the previous iteration, for example the quality of the code and the user interface.

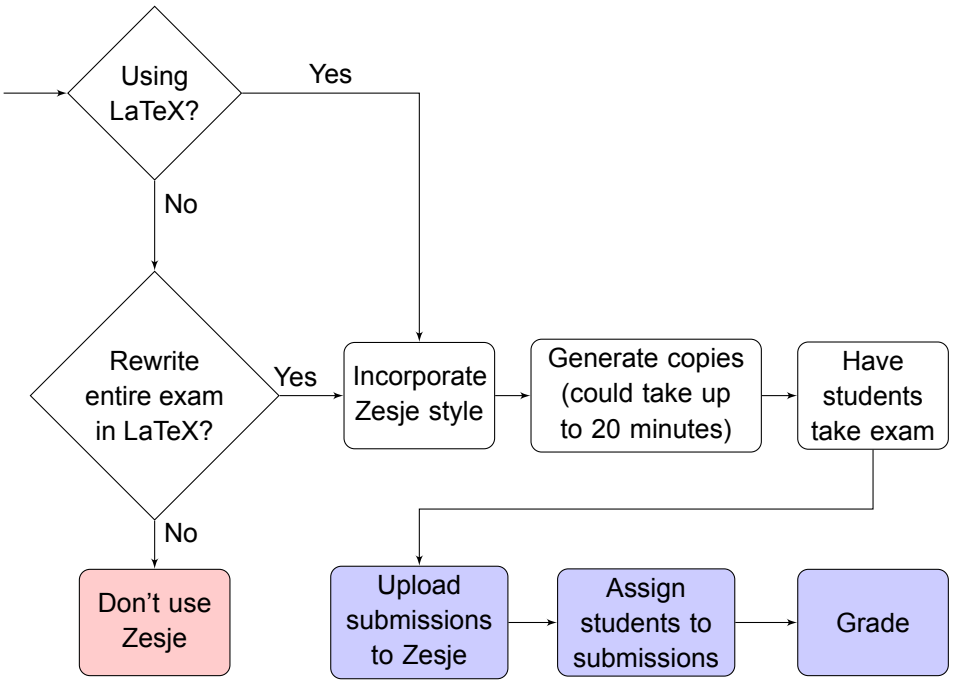
2.2. Problem definition

Nevertheless, the one thing that has not changed in the second iteration was how compatible exams were created. In both iterations, a LaTeX template was used to create exams that are compatible with Zesje. However, working with this template has some major downsides. First of all, the LaTeX template for Zesje is responsible for tasks that LaTeX was never designed for, such as generating QR barcodes, keeping track of copy numbers and generating metadata and output this to a (YAML) file. Futhermore, requiring exams to be made in LaTeX excludes every instructor that does not use LaTeX already. The client wishes to make transitioning to Zesje as easy as possible, and this means deprecating the LaTeX template for a system that is input-agnostic. As every paper-based exam has to be printed eventually, PDF is the perfect intermediate file format.

The most logical way to replace the LaTeX package is to be able to extend Zesje such that any PDF can be made into a compatible exam for use with Zesje. The major problem, then, is how this should be done.

2.3. Problem analysis

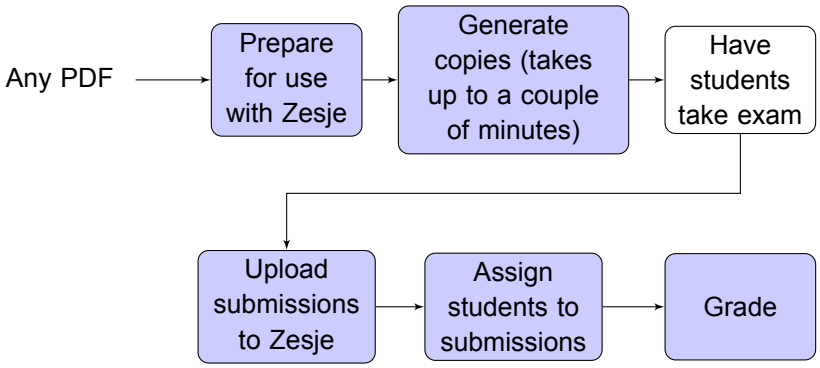
To analyze the problem a better understanding of how Zesje is used is needed. The typical workflow for Zesje used to be as follows:



In the diagram, the blue blocks are steps that are done in the Zesje application. Note the hard requirement on having the source exam being written in LaTeX. If an instructor cannot or will not write their exam in LaTeX, they can not make use of Zesje.

There is also a performance consideration. The 'generate copies' step takes up a relatively large amount of time. This is caused by the LaTeX compiler having to do things (generating QR barcodes, generating a big YAML file, keeping track of coordinates) it was never designed to do.

The proposed alternative by the client is as follows:



The proposal enables an instructor to use whatever software they want to use for making they exam, assuming that eventually they export to PDF. This is reasonable as PDF is very common to use as intermediate format to send to printers. Also, preparing an exam to use with Zesje can then be done from within the application itself. This provides a much better user experience and the application can guide an instructor through the process. To accomplish this, a way to modify existing PDFs in Zesje

is needed and must be provided in such a way that it is easy for a user to make use of this. The front end must have a clear interface that guides a user through the process such that learning how to use Zesje can be done in a small amount of time. Another important aspect is the reliability of processing submissions. Most faults can be manually resolved after the exam was made; however, one important exception is the case in which a page of a submission is not able to be identified (by the barcode). In that case, there is no way of knowing which student it belongs to, as it is possible that the submissions are scanned out-of-order.

3

Research

The first phase of the project was the research phase, which took place during the first two weeks of the project. To solve the problem, various algorithms and libraries are needed. In this chapter we research and discuss such algorithm and libraries.

3.1. Related work

The grading of students work can be a time consuming process. Especially for courses with a large number of students grading does not scale well. An instructor is able to employ more graders but this tends to hurt feedback consistency.

Tools that aid in grading are therefore becoming more popular. One approach is automatic grading. For assignments consisting only of multiple choice questions an instructor can use scantrons to automate its grading. One major setback is the lack of constructive feedback for students with multiple choice questions [2].

Instead of fully automating, some solutions try to mainly assist with and accelerate the conventional workflow of grading. One example is given by Bloomfield and Groves [3]. Their method does involve a dependency on specialized widgets for the exam pages, but there is no automatic grading, just a way to do it digitally. This enables instructors to grade large quantities without having to manage and share physical exams, and can grade the same student (but different questions) simultaneously. Gradescope is another (commercial) example of such grading software [1]. Their approach involves even less dependencies on the exam. An instructor can choose to identify the student when they hand in their exam, or batch-scan all exams and identify as usual via reading their name from the exam. Gradescope helps with this by making it easy to look up a student and link them to an exam in their software. To aid in giving feedback to similar answers, Gradescope uses neural networks and computer vision for grouping of similar answers.

3.2. Choice of barcode type

Clearly, the data about the exam, exam instance and page need to be encoded somehow. In the current implementation, QR codes are used to encode this information. We investigated whether QR codes are indeed the best choice or whether there is a better format.

One of the options that was considered is using a 1D barcode instead of a 2D barcode. The advantage of 1D barcodes is that their height can be fixed, even when the length of the name of the exam differs,

and this height does not need to be as large as it needs to be for 2D barcodes. As we would like to insert barcodes at the top of the page, this means that the top margin of exam pages does not need to be as large. However, 1D barcodes also have the following major disadvantages:

- their width scales linearly with the amount of data, so if the name of an exam is very long, the barcode may not fit on the page anymore. This could be fixed by assigning each exam a number, storing this number and encoding it instead in the barcode. However, this would be a lot of extra effort.
- they do not feature error correction, while 2D barcodes do [4]. Error correction is important, because scans are not perfect, and Zesje must be able to decode barcodes correctly in the face of this imperfectness. This problem can be alleviated by using our own error correction code. However, this also requires extra code and complexity, while 2D barcodes have error correction 'for free'.

Another option is to use 2D barcodes, which exist in a number of different formats. One of the requirements of the format is that there are encoding/decoding libraries of sufficient quality available for the format in Python 3, as this is the language the backend is written in, and the code for barcode generation and recognition will live there.

Based on our research, the formats QR and Data Matrix are the only 2D barcode formats that have libraries of sufficient quality available for both encoding and decoding. Other 2D barcode formats also exist (e.g. PDF417 or Aztec), but there do not appear to exist any 2D barcode encoding/decoding libraries of sufficient quality for these formats for Python 3.

In the research phase, a decision was made to use QR instead of Data Matrix due to the following factors:

- the ability to decode QR codes is already present in the code. Initially (see problem below), we will be able to simply keep using this code instead of needing to write code to support Data Matrix codes.
- QR codes are more popular. Therefore, it is more likely that the libraries for QR codes will continue to be well-maintained in the future.

However, it later became clear that the current library used for QR reading requires some ugly hacks to work correctly. Also, the library currently used for reading QR codes has been unmaintained since 2013 (see 3.7.1). Therefore, during the implementation phase of the project, a switch was made to use Data Matrix instead.

Generating and reading barcodes (and the libraries to use for this task) are discussed in sections 3.7.1 and 3.8.2.

3.3. Student identification grid

Students are identified by their student number. To link an exam to a student, a method to communicate between a human (the student) and a computer is needed, preferably an easy one but more importantly a reliable one. This can be achieved by making it easy to encode a student number while a computer can reliably decode this number. A method that is similar to multiple-choice questions can be used, where a student can mark black a set of boxes.

Sonate, TU Delft's system of predefined multiple-choice exam forms [5], uses such a method. Their form contains a grid of boxes in which students can enter their student number by ticking the corre-

sponding boxes. A student is able to write their number out above the grid, this minimizes (human) mistakes with ticking the corresponding boxes. For the decoding, a computer is easily able to parse the boxes just like any other multiple choice answer sheet.

This approach fits the requirements for Zesje and is already well-known under students. Therefore, for Zesje, a widget will be designed that is similar to the Sonate student number grid (figure 3.1).

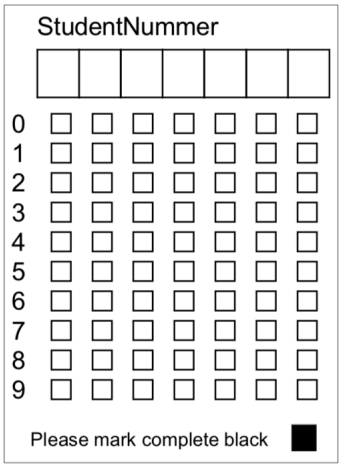


Figure 3.1: Sonate student number grid

3.4. Pages for extra space

One common occurrence is that a student requires more space to write their answer than is provided. This is not a problem with exams that do not provide answer boxes but separate paper to write answers on. For exams that do provide answer boxes but are not graded with such digital grading workflow, usually a page with 'extra space' is provided on which a student can write their name and are expected to write down the question for which the space is used. For Zesje this problem can be solved by a similar technique. Every exam should also come with a generated 'extra space' page. On this page a student identification widget is placed together with a similar widget but for identifying the corresponding question. In the front end both the regular answer box and the extra space box should be displayed together.

3.5. UI

3.5.1. Viewing PDF

When preparing an exam for use with Zesje a teacher is presented with an user interface in which their PDF is rendered. It is important that the teacher is shown an accurate presentation of the PDF because the data about where widgets are placed depend on what the teacher sees. Mozilla has built PDF.js [6] which is included as default PDF viewer for the Firefox browser [7]. It is an open source library to render PDFs, completely written in javascript and thus a plausible option for rendering PDFs for viewing by a user. The examples on PDF.js' homepage show that the library is also very flexible and easy to use. Because Firefox is a very popular browser, PDF.js receives much support and is a reliable option.

3.5.2. Upload of exam components

An interface is presented to the user to view the various insertions they make in the PDF. A set of needed components can be concluded from the requirement 'Ability to view and edit locations of answer boxes'. These are a framework to wrap the aforementioned PDF.js for use with React components and draggable windows to draw over the PDF to designate answer boxes. For these components there exist various frameworks to use. For these components, react-pdf-js [8] and react-draggable [9] can be used. React-pdf-js provides a wrapper to use PDF.js as a React component. React-draggable gives draggable windows as React components to use on top of the PDF.

3.6. Blank space detection

The ability for teachers to create Zesje-compatible exams without using a required LaTeX template gives them more flexibility in the design of the exams. However, it is not safe to assume that every exam has the required space for the insertion of the various widgets mentioned in this research report. Therefore, it is of importance to implement a module in the system which checks whether there is enough blank space (at a fixed location) in the exam to place these widgets. In this section, the various steps of this problem are explained and choices of frameworks and algorithms are given. Section 3.6.1 explains why it is of importance to use a different format than PDF. After this, section 3.6.2 explains the choice of framework for image processing and how the problem can be solved using image processing.

3.6.1. PDF conversion

Blank space, in the context of this problem, is a visual concept: it is not only necessary to know whether there exists enough blank space, but also to know where it is. One of the issues with PDF however is that without fully rendering the PDF, it is cumbersome to determine where texts, images and other objects end up visually. As the detection of these elements is the inverse of blank space detection, it is also annoying to detect blank space using this file format. This makes an image of each page in the PDF is a better input for the problem. Thus, the first step in the algorithm is converting a PDF into its set of images.

We chose to use the Wand library [10], which is a well-maintained binding for the ImageMagick MagickWand API. Wand can easily convert from PDF to image. It requires external installation of ImageMagick and GhostScript, but these programs are well-known, ubiquitous and available for all well-known operating systems.

3.6.2. Image processing

Image processing should then be used for thresholding and image enhancements. Thresholding should be done to segmentate text, images and other objects from the background [11]. Image enhancements such as blurring and filtering operations can be used to sharpen edges and remove potential noise [12] from the thresholding step. With respect to the design goals, the best choice of framework for this is OpenCV [13]. First of all, OpenCV is already used in Zesje en therefore does not require the addition of an extra framework. Furthermore, OpenCV is the de facto standard computer vision library for Python. Thus using OpenCV keeps the maintainability as high as possible.

3.7. Generating final PDF with Python

Once an exam is uploaded by a teacher, we will have to generate output PDFs containing QR codes, corner markers and a student identification widget. Corner markers and student identification widgets can be handled by the library we will use for PDF generation (see section 3.7.2), because they do not change, so they can be hardcoded (in either the PDF generation code itself or in a separate template file which is inserted on generation). However, QR codes need to be generated for every page. Section 3.7.1 will discuss the generation of QR codes, and section 3.7.2 will discuss the generation of the PDF files themselves.

3.7.1. Barcode generation

In the resarch phase, it was determined that QR codes were to be used. The most popular library for QR code generation in Python 3 is qrcode [14]. It has over 1000 stars on GitLab and is maintained, and it supports all options in the QR code specification. Another option is pyqrcode [15], but it has not been maintained since 2016. Therefore, it was decided that the qrcode library was to be used.

When the decision was made to switch to Data Matrix, this subject was revisited. The only Data Matrix generation library for Python 3 was pyStrich [16], and it appeared to work well, so the decision was made to use it.

During the project, the author of the pylibdmtx [17] library updated the library to also support Data Matrix generation. At that time, code using pyStrich was already written for the project. Therefore, this option was not explored.

3.7.2. PDF generation

The source exam PDF needs to be modified to contain the relevant QR codes, corner markers and the student identification widget. Unfortunately, there are no Python 3 libraries available which can modify pre-existing PDFs. However, since we only need to add elements to each page, one way to achieve our goal is by generating a new PDF with the necessary additions (the 'watermark'), and then merging the watermark and the original PDF. These two steps will be detailed below.

Watermark generation

The de facto standard library for PDF generation in Python 3 is ReportLab [**\gls {reportlab}**]. Another option is pyfpdf [18], but because of the overwhelming popularity of ReportLab, we will use ReportLab.

Merging of PDF pages

There are two libraries that can be used to merge PDFs, namely PyPDF2 and pdfrw. PyPDF2 is the current library used in the project. However, benchmarks performed by us show that pdfrw performs faster on large amounts of copies, consistently performing over five times faster than PyPDF2. In the interest of maintainability, we prefer only to use one of these libraries in the project. PyPDF2 is currently only used in one function in the project, and it is not a lot of effort to rewrite this function to use pdfrw instead. Therefore, pdfrw will be used.

Another PDF generation approach that could have been used was to use pdfrw to generate ReportLab objects, and then use ReportLab to directly generate the final PDFs. However, merging is more 'low-level' than converting the PDF pages to ReportLab objects and then writing them back to PDF. This minimizes the chance of unwanted visual changes in the PDF, so we have decided to use the merging approach.

3.8. Processing of scanned pages

3.8.1. Deskewing documents

The prevention of skewed documents during the scanning of exams is close to inevitable [19]. Skewing in documents can cause a drop in reliability and efficiency during further steps of the processing of

scanned pages [20]. This creates a need for an algorithm or approach which corrects a possible skew in the document.

After some research, a list of general steps is formalized. The first step in the approach is to apply some preprocessing. This is because a scanned document is not always scanned cleanly: the paper or scanner can be affected by dusts or spots [12]. The next step is to actually detect the amount of skewing in the document. The current implementation in Zesje does this by solely looking at the rotation and position of the QR code. However, as noted by the client, the main issue with this approach is that a small error in the positioning of an interest point in the QR code can result in a large error in the detection of skewing. This is due to the small size of the inserted QR codes. The proposed solution for this is to make use of the corner markers since they're more spread apart. An approach is to identify these marker, get the corner interest point for each of these markers using for example the Harris detector [21]. The angle can then be calculated using the coordinates of the corner points. This is, given the design goals, the best solution, since of its rather fast performance and simplicity. No extra library is needed since this can all be done in OpenCV.

For future implementations, research has also be done into skew detection algorithms which do not require a dependency on the corner markers. One of them is the Hough transform which maps lines in the spatial domain to lines in the Hough parametric space. The main advantages of the algorithm is its robustness and simplicity. However, the time and space complexity is a concerning factor [22, 23]. Another simple, but fast, solution is the use of the smearing or run length smoothing algorithm. It uses morphological operations to create block segments from which the skew angle can be calculated [24]. The main problem with this approach though, is that it is not as robust as other algorithms: other algorithms can handle large skewing better. If the angle can not get higher than 15°, the algorithm performs comparable to other approaches [25]. The last group of researched solutions are the horizontal projection profile algorithms. This stores a count of the number of black pixels for each row as a location in the profile [26]. The algorithm is very similar to the run length smoothing algorithm with respect to the fact that only low skewing can be handled correctly without adjustments to the algorithm. Although it is not mentioned directly, the time complexity seems higher than that of run length smoothing [27].

3.8.2. Reading barcodes

The current QR code reader in use in the code is zbar-py [28]. It works well in the current system, but it is sensitive to the orientation of QR codes [29]. However, code has already been written in the project to mitigate this problem, so this is not a big issue.

One problem with zbar-py is that it does not install on Windows [30]. Therefore, if we implement the requirement 'Ability to run Zesje on Windows and macOS' (which is a 'could have' requirement under the MoSCoW model), it will be necessary to switch to a different library. Several libraries which work with Python 3 were investigated:

- pyzbar [31]: requires an external component to be installed on macOS and Linux, is very well-tested
- qrtools [32]: requires an external component to be installed
- zbarlight [33]: requires an external component to be installed, is complicated to install on Windows [34]
- qreader [35]: can only read QR codes generated on a computer [35], and cannot correctly decode some QR codes [36]

Based on this research, pyzbar, qrtools and (to a lesser extent) zbarlight are all viable options. In our

opinion, pyzbar is the best option, since it is very well-tested. Therefore, if Windows support is to be added, the decision was made during the research phase to use pyzbar.

During the implementation phase of the project, it became clear that zbar [37], the C library for which pyzbar, zbar-py and qrtools are bindings, requires some ugly hacks to work and has been unmaintained since 2013 [38]. This invalidated the advantages of QR codes (see 3.2). Therefore, it was decided to use Data Matrix codes instead of QR codes.

When this decision was made, the subject of barcode reading was revisited. The only well-documented Data Matrix reading library for Python 3 is pylibdmtx [17]. It appears to work well and is well-maintained. It is a binding for the libdmtx C library [39], which is also well-maintained. One downside of pylibdmtx is that it requires an external component to be installed on macOS and Linux (namely libdmtx). However, this downside was also present for pyzbar, the library which was settled upon for QR code generation. Therefore, it was decided to use pylibdmtx.

3.8.3. Page orientation

Pages will be de-skewed with a certain degree of freedom because the corner markers are not unique. Namely pages can be oriented upside up or upside down. Since the location of the QR code is a controlled variable it is possible to identify the orientation of pages after de-skewing with use of the QR code by checking both possibilities. It is very likely that only one of the two orientations will result in a readout by our QR code decoder.

3.8.4. Reading boxes

On the exam several widgets are used that can be automatically processed. This speeds up the grading process because a grader does not have to do it manually, and only has to verify correctness. These so-called machine readable forms come in many shapes and sizes but most commonly as boxes or bubbles a student can fill. Techniques to determine whether an answer box is filled out or not are trivial because the locations of where the boxes are located are known beforehand. The pixel values of the locations of the boxes can then be used to determine whether the student has filled them out or not. Some considerations to take into account are students that don't fill out any of the boxes for a given form, students might use different methods to mark the box (fill, tick, cross), and some scanners are configured to scan binary black and white images so no gray scale information is available.

Student identification widget

Identification of the student is an important function of Zesje. This allows for batch-scanning all exam pages without risk of not knowing which student a page belongs to. As long as the student can be linked to an unique exam the QR code can be used to identify all other pages. To accomplish this a student is required to fill-out his name and student number just like they are used to do on regular paper exams, except that the student number is also filled out in the identification grid. With this an exam can be reliably linked to a student.

Multiple choice questions

With the readout of multiple choice answers it is also possible to (partially) automate the grading process of such questions. If feedback is given for a particular answer, Zesje should be able to suggest or automatically link the same feedback to equivalent answers. One important consideration here is that a grader must verify every given answer and feedback. To be able to do this a grader must be able to

distinguish between automatically graded answers and manually graded answers. This can for example be achieved by having Zesje keep track of automatically graded answers and have the grader verify them manually.

3.9. Decentralized workflow

One workflow for Zesje is one in which users run their own instance and are able to combine their grading results with others. This requires a way to combine grading results without ending up with duplicate data or having conflicts, making Zesje distributed. One approach would be to implement a way to export and import (relevant parts of) the database of a Zesje instance. This enables users to share their databases with others via supported methods. For this to be successful the model of the grading result data and Zesje's interpretation must be clear on how to handle cases such as (but not limited to) different instances lacking the original PDFs and its data, having different versions of exams with duplicate names, existing results for the same exam and student combination and so on. If the model and interpretation of the model clearly describes what should or should not happen in such cases, import/export features can be built and Zesje can be used in a distributive workflow. If the software detects a conflict either the user should be prompted or Zesje should complain. Finally changes should only be applied when all of the important data is checked, either by software or user, as to not leave the database is an inconsistent state between pre- and post-importing the new data set.

3.9.1. Compatibility with Windows and macOS

If the distributed workflow is implemented, then the client would like for Zesje to also run on Windows and macOS (it is currently only hosted and known to work by the client on Linux). Based on our own tests, the Zesje project does run on macOS and does not run on Windows.

Although the project does run on macOS, further testing will need to be done to examine whether the entire system works on macOS. As for Windows, the zbar-py library which is currently used for QR code decoding does not install on Windows (an error is thrown). A good alternative for zbar-py is pyzbar, which is discussed in section 3.8.2. Further testing should also be done to determine whether there are other problems on Windows. Because this is a 'could have' requirement, this testing had not been done yet at the time of writing of the research report.

3.10. Fuzzy number matching

During the implementation phase, it was noted that the detection of student numbers is not always correct. Not only that, it is also not always true that students enter their number correctly: mistakes can always happen. Therefore the client noted that it may be of importance to do some research about the fuzzy matching of student numbers.

It is important to look at the supported edit operations when making the choice of matching algorithm for this problem. We distinguished between these operations when choosing an algorithm:

- Insertion (123467 -> 1234**5**67)
- Deletion (1234**4**567 -> 1234567)
- Substitution (1**9**34567 -> 1**2**34567)

Covering these operations is considered important for future possibilities. Also, another important fact

when choosing the library/algorithm, is the fact that the difference between each digit is also of importance. The chance that a ‘4‘ should be substituted with a ‘5‘ should be higher than wth a ‘9‘ for example. This is due to the distances between the checkboxes for marking each digit. Therefore the library should support this by being able to adjust the cost of the edit operation per case.

After some research, the best library we found was weighted-levenshtein [40]. First of all, the levenshtein edit distance in general can work with all aforementioned edit operations [41]. Second of all, the library, as the name implies, can put weights on each edit operation. This allows us to cover the latter point of importance mentioned. Finally, there were no other libraries we deemed as usable in this case: either they allowed not all edit operations or did not allow the adjustment of the costs.

4

Development process

This chapter details the development process. First, the software development methodology used by the team is explained. Finally, a list of development tools used by the team is given.

4.1. Methodology

As agreed upon during the project planning (see section C.5), ideas from SCRUM were used as the basis of our development methodology.

At the beginning of every week, tasks were evaluated and the sprint was planned, and at the end of every week, a meeting with the client (Anton Akhmerov) and, dependent upon their availability, the other members of the Zesje team (Thomas Roos and Joseph Weston), was held.

During the first weeks of the implementation phase, it became clear that the team worked well without a daily standup, but by simply meeting on at least 3 days every week, so it was decided to scrap the daily standup.

4.1.1. Code review

Again as agreed upon during the project planning (see section C.6.2), before merging any code into the main project branch, that code had to be reviewed and approved by at least one other Zesje developer. Any changes suggested by other developers during their review of the code would have to be considered and either implemented or refuted.

4.2. Tools

To aid the development process, various tools were used:

- **GitLab.** GitLab is a Git repository hosting service, that also provides features such as an issue tracker, merge requests (also known as pull requests) and continuous integration support. Development of Zesje already took place on the GitLab server hosted by the client, and we simply used that infrastructure for our development. Code review was faciliated by using the merge request functionality of GitLab (also known as pull request functionality in other tools).
- **Trello.** Trello is a project management tool. It can be thought of as a virtual whiteboard on which you can post cards. We used Trello as SCRUM board, where we tracked which tasks were in the

project and sprint backlogs, in progress, waiting for a code review and done.

- **Mattermost.** Mattermost is an instant messaging platform designed for teams. We used Mattermost to chat between team members, but also with the client and the rest of the Zesje development team (the Mattermost server hosted by the client was already in use for communication in the existing Zesje team). This kept discussions central and was useful to get quick feedback from the client and the other Zesje developers.

5 Design

Zesje was an already existing application at the start of the project. In this chapter, we describe the existing system architecture and the changes made by the team.

5.1. Existing system design

Zesje is a webapp comprised of a back-end and a front-end. These two components communicate with each other via a RESTful HTTP API.

The back-end is comprised of an API layer (written using the Flask web server and the Flask-RESTful extension) (files `zesje/api.py` and those contained in `zesje/resources/`) and a 'helper' layer, which contains the business logic in a number of files (contained in `zesje/helpers/`). Data is stored in a SQLite database, and the Pony ORM is used as the access layer for this database.

The front-end is a React single-page application (files contained in `client/`, in which every page of the application is a React component. Finally, as mentioned in chapter 2, compatible exams are not generated by Zesje itself, but using a separately available LaTeX template, which also creates a YAML file that is used as input for Zesje. State is managed somewhat intelligently by the front-end: `index.jsx`, the entry point of the front-end, gets exam data from the back-end server only once and then stores it in its `state` object for all components of the front-end to use.

5.2. Changes in system design

The main change made by the project team, as mentioned in section 2.2, is to deprecate the LaTeX template, and implement its functionality in Zesje itself. The overall architecture of Zesje remains unchanged, but in collaboration with the client, the former 'helper' files were moved to the `zesje/` folder, and the API layer files were moved to the `zesje/api/` folder.

5.2.1. Database

Several changes had to be made to the database (see figure 5.1). The changes mainly consist of the addition of a `Widget` table and the related relations. Furthermore some deletions of unused or deprecated rows and a rename of PDF to Scans.

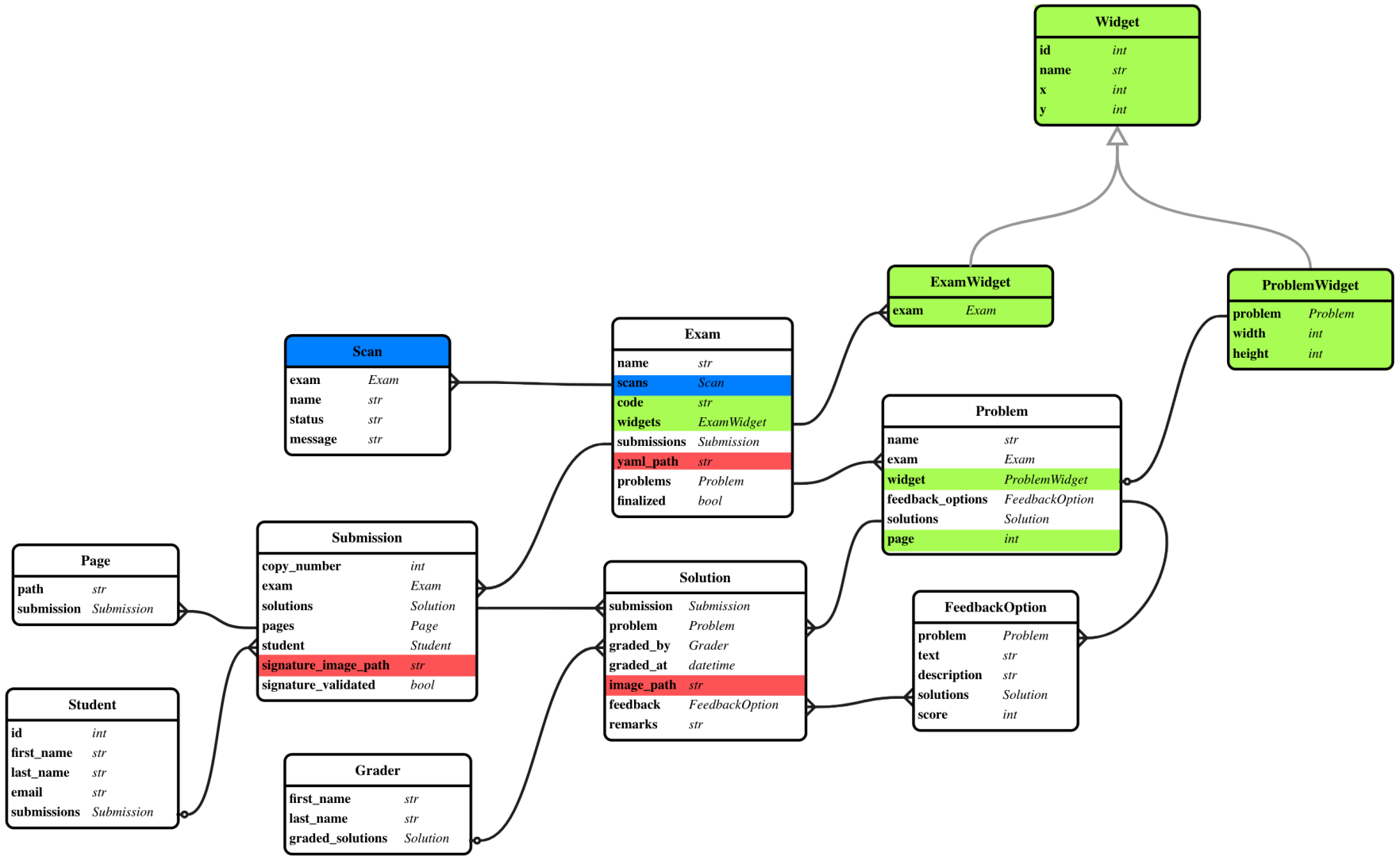


Figure 5.1: Database changes. Red=deletion, green=addition, blue=rename

6

Implementation

In this chapter, we discuss the implementation of the functionality added during the project. We do this by walking through the workflow followed by a user of the system in order to explore the implementation of the various features.

6.1. Exam PDF uploading

The first step for the user is to upload their exam as a PDF to Zesje. This happens in the `AddExam.jsx` file. To do this, they select a file using a `Dropzone` (from the react-dropzone library). The user is then shown a preview, which is generated by the react-pdf library (which is a wrapper around the PDF.js pdf rendering library). See figure 6.1 for an example. The user enters a name for the exam and clicks the upload button, upon which a POST call is made to the `/api/exams` endpoint with the PDF and the name. This call is handled by the `Exams.post` method in `zesje/api/exams.py`, which stores the exam and the student ID widget and barcode widget in the database.

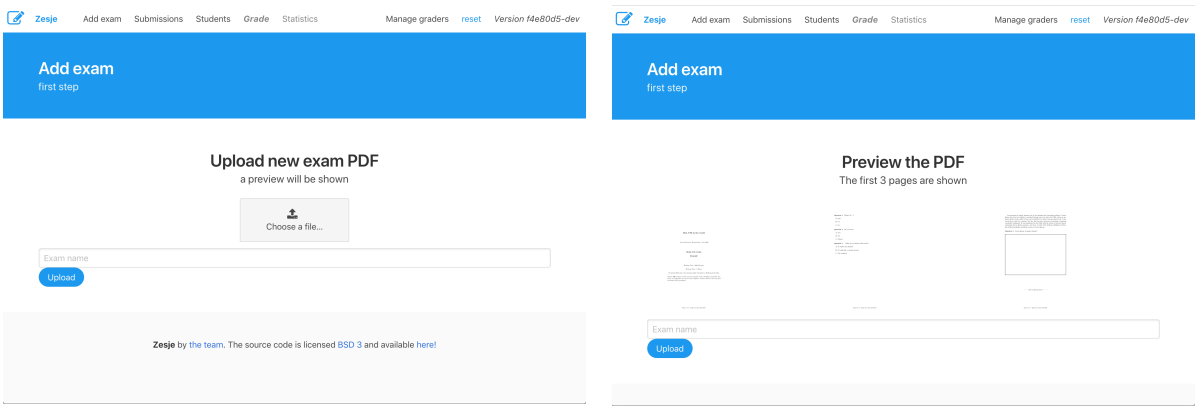


Figure 6.1: Uploading an exam

6.2. Exam preparation

Preparing (i.e. editing, previewing and generating) an uploaded exam happens in the `Exam.jsx` file. First, this component checks whether the exam is finalized (a concept which will be introduced later). Depending on whether the exam is finalized a different page is shown. For non-finalized exams the exam editing page is shown, and for finalized exams the generate exams page is shown.

6.2.1. Exam editing

This page allows for the addition and modification of widgets. A widget in Zesje is anything that is added to an exam by Zesje and relates to location on a page. They fall in two main categories: exam widgets and problem widgets. Exam widgets are bound to an exam and consist of a barcode for each page and the student identification grid on the first page. Problem widgets are widgets that represent the problems.

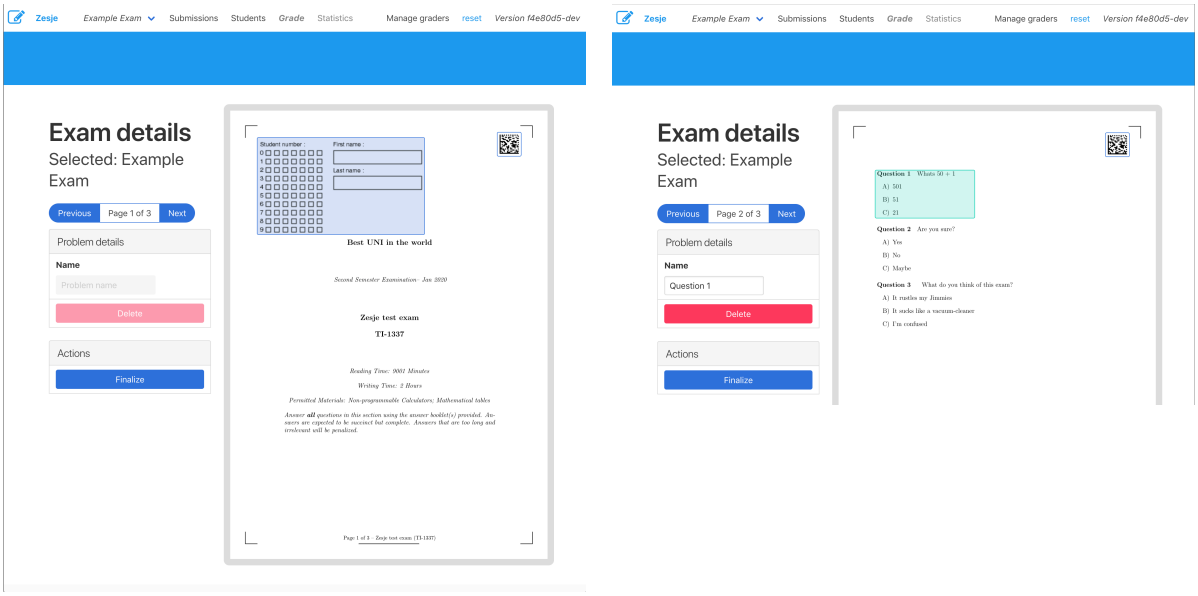


Figure 6.2: Editing widgets for an exam

In figure 6.2 an example is shown. The left screenshot shows the first page of an exam with the student identification grid and the barcode visible as widgets and the right screenshot shows a problem widget highlighted.

For each problem a widget must be added. The widget contains the information needed to present a grader with just the answers.

To use an exam with Zesje there must at least be two exam widgets: the barcode (on every page) and the student identification grid. Both widgets are automatically added, the user can only position those.

A finalize button is available. The user can press this to enter the previewing stage explained below.

6.2.2. Exam previewing

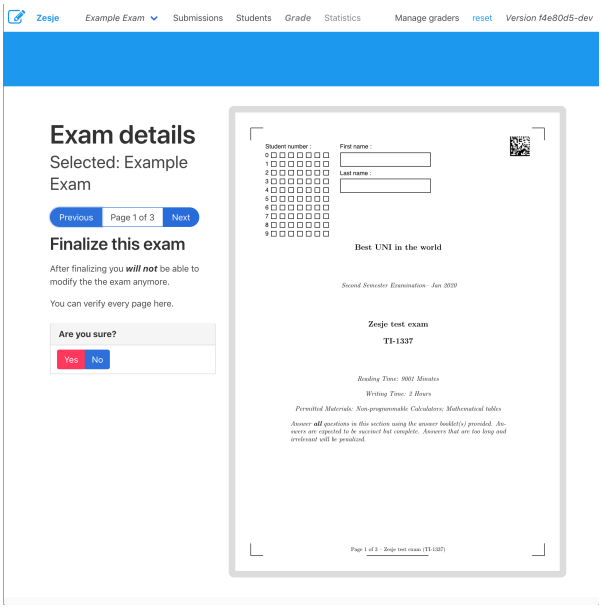


Figure 6.3: Previewing an exam

As the user can not un-finalize an exam after it has been finalized, a separate preview step is introduced. This step is implemented by executing a GET to `/api/exams/<exam_id>/preview`. The backend will then render a preview of the current exam with an example barcode and the client will present this to the user. Also the user is presented with a piece of text that explains the what the finalization of the exam means. An example can be seen in figure 6.3.

This step accomplished two things: the user is confronted to check the correctness of the exam; and the user is made aware that the finalize action is irreversible.

6.2.3. Exam generation

If the user is happy with the exam and has finalized it, the exam page wil present the generate and download page. An example is shown in figure 6.4. On this page the user can request a range of copy numbers to be generated by Zesje. Furthermore the type of file can be chosen: ZIP or PDF. This will serve a ZIP archive with every copy as separate file or a PDF with all copies concatenated respectively. The former is useful if the printer supports stapling, as this is on a per-file basis. Otherwise the latter is useful as only a single file must be send to the printer. As there are no more changes possible to the exam, as it is finalized, the result for a given copy number will always be the same. Because of this, only the first time a copy number is requested it will have to be generated. This saves processing time if an instructor later wants to request a different range or changes their mind about the download format.

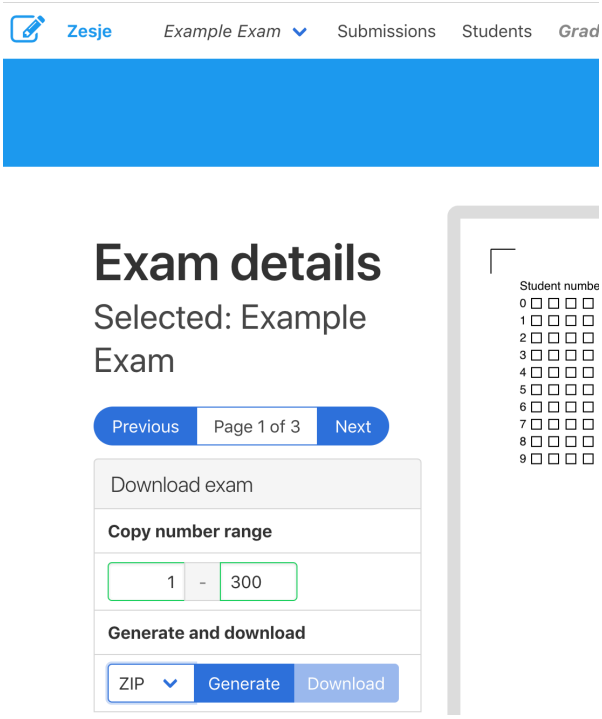


Figure 6.4: Generating and downloading prepared exams

6.2.4. Uploading submissions

After the exam has been made by students, all submissions must be scanned. Here, a nice feature of Zesje becomes clear: the order of the pages doesn't matter, as long as the barcode can be decoded. In figure 6.5, it is shown that a user can easily drag and drop multiple scanned PDFs into Zesje. It does not matter if the pages are out of order or exist in separate PDFs. Each page will be processed and the following preprocessing steps will be taken for every page:

- orient the page in portrait mode
- identify corner markers
- deskew page based on corner markers
- decode barcode to:
  - figure out the actual orientation of the page
  - extract exam token, copy number and page number
- if the page contains the student identification grid, readout the student number
- if a student number was found, link that student to the submission

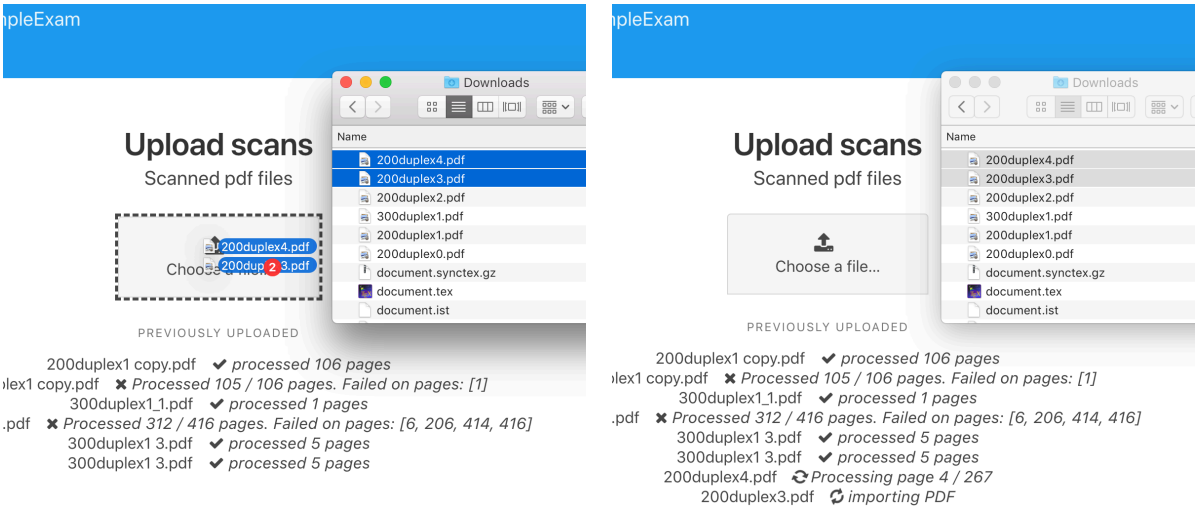


Figure 6.5: Uploading scanned submissions

6.2.5. Linking students to submissions

Before graders can start grading, they have to link each submission to a student. For each submission, Zesje automatically tries to detect and suggest which student it is from. This is done by using image processing on the student number grid.

The automatic detection is done by first detecting the checkboxes of the identification grid. After running a flood fill algorithm to fill enclosed spaces, a detector is used to detect objects which look like squares. The end result is a list of interest points which are represented in figure 6.6 as red circles. After this, a general region for each interest point is checked for the amount of filling as seen in figure 6.7. For each column in the grid, the most filled in square is mapped to their row number. This gives in the end the complete student number.

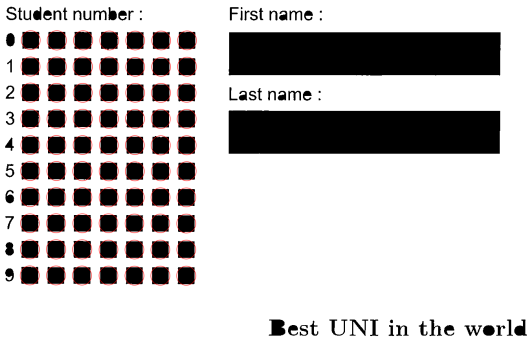


Figure 6.6: Detection of checkboxes in student number recognition

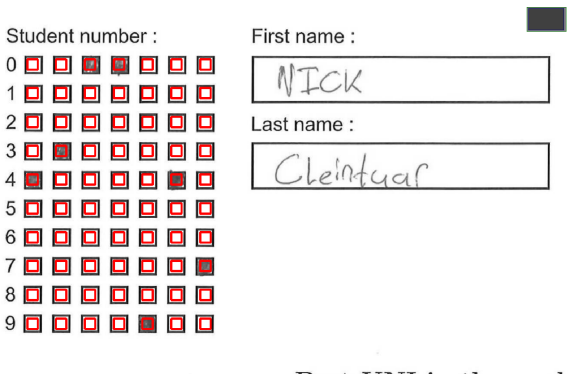


Figure 6.7: Areas checked for filling in student number reconigtion

In the end, the result of this automatic detection is always a suggestion. The teacher should always confirm that the linking of submissions with students is done correct, as can be seen in figure 6.8. In the center of the interface is the cutout of the identification grid for the user to read. In this case the system has succesfully detected the correct student number and now the user only has to click on the

student name. The addition of automatic detection of student numbers therefore gives the user the ability to transition faster into the grading part. If the wrong student number is detected, the user is able to manually search by entering (a part of) the students name or student number.

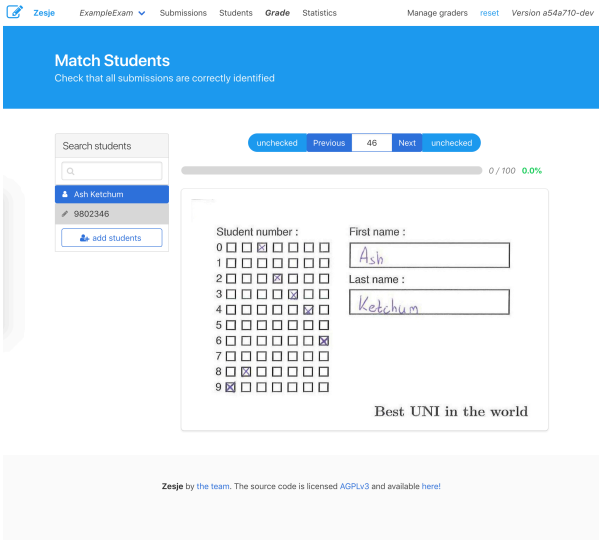


Figure 6.8: User interface where submissions can be linked to students

6.2.6. Other steps

The remaining steps in the workflow are:

- adding graders
- adding students
- grading answers
- viewing exam statistics

Code for these steps has not been altered by the project team, so we do not discuss them here.

7

Quality assurance

In this chapter, we elaborate on the steps taken by us to assure a high enough quality for our work. We also present our findings of manually testing the system. Finally, we discuss quality evaluations performed by a third party (SIG).

7.1. Unit tests

The original developers did not make use of a testing framework. To test the functionality of the new code in the backend, pytests is added. For the new additions by the project team to the logic layer, unit tests were written (in the `tests/` directory).

7.2. Static analysis

The original developers did not use any static analysis tools while developing Zesje. This meant that over time the codestyle was not consistent in aspects that tools can easily detect. Consistency can grealy improve readibilty and thus maintainabilty so the team has added a linter for the back-end code. Flake8, a popular Python linter was chosen.

7.3. Code review

As mentioned in section 4.1.1, to maintain a high quality of the code, we did not merge any code into the main project branch (`no-latex`) until the code had been reviewed by at least one other Zesje developer. Any suggested changes would have to be considered before merging was possible.

7.4. Manual end-to-end test

At the end of the project, the entire Zesje system was tested by generating and printing 100 copies of a test exam, filling these out, scanning them and inputting these scans into Zesje.

7.4.1. Methodology

The scanning was done with 3 separate configurations:

- using a Ricoh Alficio MP C4501, a high-volume scanner which is one of the standard scanner models at TU Delft, at both 200 and 300 dpi
- using a Brother ADS-1100W, a low-volume portable scanner, at 200 dpi. For this scanner, the paper guides were intentionally left with too much slack, so that pages would be fed into the scanner at slight random angles.

For both models of scanners, the ADF (automatic document feeder) was used. We think it is unlikely that teachers will scan exams page-for-page using the flatbed glass, so the flatbed glass of the Ricoh Alficio MP C4501 was not used (the Brother ADS-1100W is an ADF-only scanner).

7.4.2. Results and findings

The results of this end-to-end test were:

- using the scans obtained from the Ricoh Alficio MP C4501, after deleting blank pages, Zesje worked well and there were no problems.
- using the scans obtained from the Brother ADS-1100W, due to scanning artifacts, Zesje was not able to process all the pages. This scanner generated two types of artifacts that are significant for the processing of the scans:
  - solid black lines at the edges of pages, when pages were slightly rotated, which were mistaken by Zesje for the bottom bar on pages (see figure 7.1)
  - artifacts which caused parts of modules of Data Matrix codes to be missing (see figure 7.2), which meant that it was difficult for pylibdmtx to decode these codes

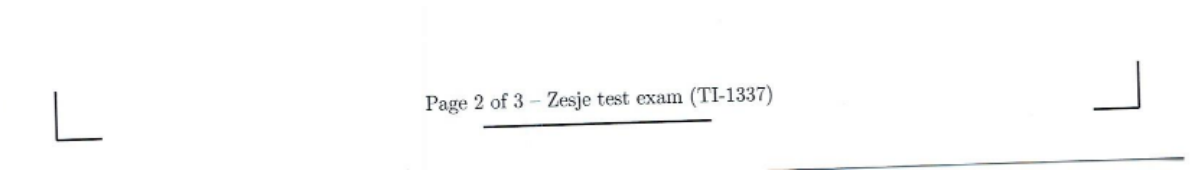


Figure 7.1: Bottom part of scanned page with artifact (bottom right) that is similar to a bar

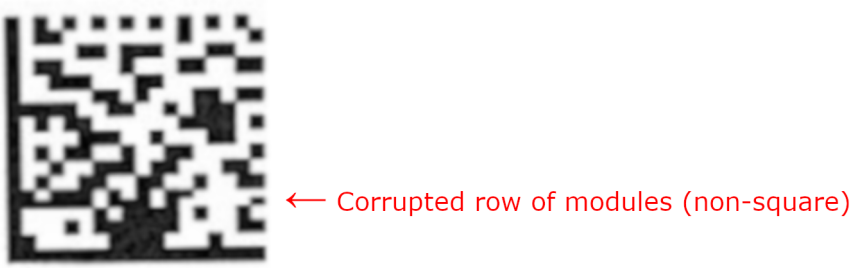


Figure 7.2: Scanned Data Matrix code with artifact, 4x enlarged

Based on the findings using the Brother ADS-1100W, changes were successfully made to the scan processing code to increase the rate of success in the face of such artifacts[1] (see sections 8.2.1 and

[1] at the time of writing, these changes have not yet been integrated into the main codebase

8.2.2). After these changes, Zesje was still not able to process all pages scanned using the Brother ADS-1100W. We think this is not a big problem, for two reasons:

- we expect that the majority of Zesje users will use a high-volume scanner, similar to the Ricoh Alficio MP C4501, located at their institution, because this is much faster and requires much less work than using a low-volume scanner such as the Brother ADS-1100W
- even if users opt to use a low-volume scanner similar to the Brother ADS-1100W, if they observe their scanners' operating instructions and do not leave excess slack when positioning the paper guides, we expect the scanner to generate less artifacts than was the case in our test, although we have not tested this ourselves

7.5. SIG code evaluation

As part of the project, our code was evaluated at two points in time by the Software Improvement Group (SIG).

7.5.1. First evaluation

SIG feedback

The feedback received from SIG is reproduced here:[2]

> The code of the system scores 3.7 stars [out of 5] on our maintainability model, which means that the code is, on average, maintainable, compared to market standards. Points of improvement identified by us are Unit Size and Unit Complexity.
>
> As discussed via e-mail, when suggesting points of improvement, we will only consider cases actually written by you [the project team].
>
> For Unit Size, we looked at the percentage of code that is longer than average. Splitting such methods into smaller pieces ensures that every part is easier to understand and test and therefore easier to maintain. In the longer methods in this system, such as, separate parts of functionality can be identified which could be re-factored to separate methods.
>
> For example, in the file exams.py, we see a number of examples of long methods. There are various causes for this. ExamGeneratedPdfs.get() is long because this method contains too much responsibility. Handling a ZIP and PDF are separate units of functionality, and it would be good to move these to new sub-methods which can then be called from get(). This also makes it easier to unit test those parts separately.
>
> In the same file, Exams._get_single() is also long, but due to a different cause. In that method, a data structure is transformed to a different format. While it is possible to do that in this way, you have probably noticed that this can easily lead to code that is difficult to read and modify. In most cases, it would be preferable to use a library for this. If that takes too much time, another option is to construct the result object in separate parts, instead of doing this entirely inline in the return statement.
>
> For Unit Complexity, we looked at the percentage of code that is more complex than average. Splitting such methods into smaller pieces ensures that every part is easier to understand

[2] freely translated from the original Dutch

and test and therefore easier to maintain. By moving each of the functionalities to a separate method with a descriptive name, each of the parts can be tested separately, and the overall flow of the method will be easier to understand.

In your project, renderWidgets() in PDFEditor.jsx is a good example. In this method, logic and the displaying of a component are mixed. In React, it is your own responsibility to maintain a good separation between those two concerns; when compared to other frameworks like Angular, the separation is much less strict. That gives you much more flexibility, but also makes it more difficult to differentiate the various parts of code in the long term. Calculating the width/height of the component is now part of the rendering itself, it is better to clearly separate these.

Finally, we note that we did not find any (unit) test code in the code upload. It is strongly advisable to define automated tests for at least the most important parts of functionality, in order to ensure that possible changes do not lead to unwanted behaviour. [note from the team: there were automated unit tests included in the code upload, we are not sure whether SIG accidentally overlooked these tests]

In general, there are some possible improvements. We hope that it will be possible to realize these in the rest of the implementation phase.

Changes made in codebase

For code that was newly written or refactored after the feedback was received, more care was taken not to construct methods or functions that are too long or too complex. We give two examples here:

- when reworking `exams.py` to support ranges of copies (as opposed to a single range starting at copy number 0), care was taken not to construct a very large `ExamGeneratedPdfs.post()` method, but to factor some of its code into `_generate_exam_pdfs()`, and generating previews has been factored out into the new `ExamPreview` class in the same file.
- when reworking `Exam.jsx` with a new layout as requested by the client, constituent parts of the previously large `render()` method have been factored into smaller components and methods, such as `Pager` and `SidePanel`.

Regretfully, where we should have planned to also fix these problems in existing (non-changed) code, this has not been done.

7.5.2. Second evaluation

At the time of this writing, the team has not yet received the results of the second evaluation.

8

Discussion and future work

This chapter contains discussions about the project itself and about ethics and privacy. Furthermore, a number of suggestions for future work is laid out.

8.1. Changing requirements

Because development took place in an agile way and weekly meetings with the client were held to discuss progress, the requirements slightly shifted over the course of the project.

The priority of the must-have requirement of detection of whitespace in uploaded exams was lowered once the user interface employed a more “What You See Is What You Get“ experience by also rendering corner markers, bottom bar, an identification grid and example barcode. This means that the user can now easily see if there are any visual discrepancies and does not have to rely on a warning message telling them that there is not enough space.

A new requirement that was introduced during the course of the project and that can be classified as a should-have is fuzzy number matching for student numbers. Research has been done into this subject, as discussed in section 3.10. However, the implementation of this requirement has not been finished.

8.2. Challenges

In this section, we discuss some of the challenges encountered by us during the project.

8.2.1. Bar orientation

Initially, it was decided to detect whether a scanned page is oriented correctly (i.e. whether it is right side up or upside down) by using a bar at the bottom of the page. However, when we tested scanning an exam with a portable ADF scanner, it turned out that when pages are slightly rotated, the scanner may produce artifacts that are very similar to the aforementioned bar (see section 7.4 and figure 7.1).

Because of this, the scan processing code (`scans.py`) had to be changed at the end of the project so that it doesn't use the bar for detection[1]. This was successfully done and the location of the Data Matrix code is now used to determine orientation.

8.2.2. Data Matrix code artifacts

During testing, it was discovered that in some cases, scanners can produce artifacts in the Data Matrix codes, which make them more difficult to be decoded (see section 7.4 and figure 7.2). To combat this, the reading code was altered so that if a Data Matrix code is not detected by pylibdmtx, the code is rotated at various angles[1]. This increases the rate of success in reading such corrupted Data Matrix codes.

We speculate that the severity of such artifacts may be reduced even further if the Data Matrix codes are made larger than they currently are. However, this has not been tested. Alternatively, if QR codes prove to be more resilient to such artifacts, it is possible to switch back to QR; the code has been architected in such a way that such a change can easily be made by only changing the `generate_datamatrix()` method in `pdf_generation.py`.

8.2.3. Behavior of pylibdmtx on macOS

During development, an issue with the chosen decoding library pylibdmtx was discovered [42]. On Linux and Windows the results are as expected, but on macOS the library seemed to give the raw data represented by the Data Matrix code. This only happened when some characters were used in the encoded data, which was always the case for our chosen format. The cause of this bug was unable to be determined, as decoding the images with the command line utilities of libdmtx work, and pylibdmtx directly passes the results from libdmtx. Parsing of the raw data is trivial for the format of the codes used by Zesje, so if macOS is detected as host OS, an extra decoding step is implemented.

8.2.4. Unsupported PDF image compression format

During the end-to-end testing discussed in section 7.4, the library we used to parse the pages of a PDF file, PyPDF2, gave an error we had not seen before. The issue was that the PDF file included a very uncommon compression format for the scan (JBIG2), which PyPDF2 does not support. Luckily, in the end-to-end test that was performed (see 7.4), this format only seems to be used on empty (completely white) pages as this is a seemingly very efficient format for such a page. We can safely ignore this as empty pages are not of interest. Empty pages only occur when the exam has an odd number of pages and is printed and scanned in duplex.

8.3. Ethical and privacy considerations

One could wonder whether Zesje protects the privacy of students. Zesje can be self-hosted by any institution or instructor wishing to use it. This means that personally identifiable information (PII) of students, i.e. name, student number and exam answers, does not leave the system where Zesje is hosted, which means that this PII is not shared with others.

Zesje also presents opportunities for anonymous grading. There is evidence that knowledge of the identity of students while grading introduces grading bias [43, 44, 45, 46, 47], also at the university level [46, 43], and this bias may even be introduced when trying to be objective [43]. Zesje could be extended so that it is possible to hide the names of students while grading, which would reduce this bias. Nevertheless, when dealing with handwritten exams, it may never be possible to completely eliminate bias, as bias may also be introduced by factors such as handwriting quality [48]. This latter fact cannot be changed by the usage of Zesje, but is inherent to the grading of handwritten exams.

[1] at the time of writing, these changes have not yet been integrated into the main codebase

8.4. Future work

There remain many avenues for future work to improve Zesje even further. We discuss the most promising ones here.

8.4.1. Further implementing project requirements

One obvious avenue for future work is the implementation of the 'should have', 'could have' and 'won't have' requirements, as these were not implemented during this project. They are listed in section C.3.

8.4.2. Anonymous grading

As discussed in section 8.3, anonymous grading provides a promising opportunity to reduce bias in grading. Therefore, it would be a good idea to add the possibility to hide names of students during grading in Zesje.

8.4.3. Performance improvements

The current implementation of the processing of scanned pages is not optimized for performance. The old implementation used to be able to process about 5-7 pages per second (on the machines used by the project team). The new image processing steps in the new implementation cut this down to about 1-2 pages per second, and for higher quality scans, the speed appears to slow down exponentially. This processing is done naively, and the whole page is passed around in the program between processing stages. Most stages only need a piece of the page, so passing along the whole page is not efficient. Furthermore, images are redundantly converted between Pillow images and NumPy arrays many times, which is also very inefficient. An analysis (profiling) and subsequent refactoring of `scans.py` would likely yield improvements to the processing time.

8.4.4. Removing unnecessary dependencies

New PDF handling code written in the project is written using the pdfrw library (see section 3.7.2). Currently, the PyPDF2 library is still in use to extract images from scanned PDFs in the `extract_images()` method of `scans.py`, but that could also be done by pdfrw, and that code could be easily rewritten to use pdfrw instead. Also, the pyStrich library is in use to generate Data Matrix codes in the `generate_datamatrix()` method of `pdf_generation.py`, but that could also be done by pylibdmtx (see section 3.7.1).

8.4.5. Continuous integration

Because unit tests were added by us, it would be wise to run these tests automatically using a continuous integration service. That way, tests could not be accidentally broken. GitLab CI is already set up on the client's GitLab server, so it should be easy to configure this for Zesje.

9 Conclusion

The main goal of the project was to deprecate the Zesje LaTeX template. This was successfully accomplished. Users can now upload an exam and prepare it for printing wholly within Zesje itself. This means that users are no longer required to use LaTeX or the Zesje LaTeX template, freeing them to create their exam in any way of their choosing. All revised must-have requirements are completed, and an end-to-end test showed that the system works correctly, which means the project is a success; the client is satisfied with the end result, and will start using the end product as of the next deployment of Zesje.

nr	met	description
1	yes	Ability to upload an exam (as PDF) via the frontend of Zesje
2	partial	Detection of whitespace in uploaded exams for inserting corner markers, student identification grids and
3	yes	Insertion of barcodes to uniquely identify exam instances and pages.
4	yes	Insertion and usage of corner markers to orient a scan
5	yes	Insertion of student identification widgets
6	yes	Reading out of student numbers
7	yes	Ability to view and edit locations of answer boxes
8	yes	Ability to preview an exam including the mentioned insertions
9	yes	Ability to generate multiple copies of exams for printing

A

Info sheet

Project title: Automation of exam grading
Client organization: Delft University of Technology
Final presentation date: 3 July 2018
Description: Zesje is a tool used by a number of courses at the Delft University of Technology for grading paper-based exams online. One of the main issues with Zesje currently is the mandatory use of a LaTeX style class to create exams usable by Zesje. The client is thus searching for a solution where Zesje can make non-compatible exams compatible with their system, so that teachers can create their exams in their own preferred way. The main challenge of this project was to fulfill the requirements in a maintainable way. Another challenge was the research and comparison of existing libraries useful for the system. Agile development has been used for the project. Sprints and product/sprint backlogs were used to keep track of the progress during the whole project. Weekly meetings with the client were organized to review whether the implementation satisfies the requirements of the client. The final product is an expansion of Zesje where the user is able to upload their own exams to make them compatible with Zesje. The team made some recommendations on the aspects of performance, methodology and functionality. The additions made by the team will be incorporated into the next production deployment of Zesje.

Team members:
Name: Nick Cleintuar
Interests: multimedia information technology, artifical intelligence
Contributions: computer vision, unit testing

Name: Justin van der Krieken
Interests: embedded systems, programming languages
Contributions: front-end development, API development

Name: Jamy Mahabier
Interests: network engineering, software development, UI/UX design, software architecture
Contributions: PDF generation development, unit testing, back-end cleanup

All team members contributed to the final report and presentation.

Client: Anton Akhmerov - Kavli Institute of Nanoscience, Delft University of Technology
Coach: Maurício Aniche - Software Engineering Research Group[1], Delft University of Technology

Contacts:

Nick Cleintuar	nickdepinda@gmail.com
Justin van der Krieken	bep@vdkrieken.com
Jamy Mahabier	jamysanjay@gmail.com

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

[1] Faculty of Electrical Engineering, Mathematics and Computer Science

B

Project description

The original project description is reproduced here:

> You are going to join the team developing the open source software Zesje (`https://gitlab.kwant-project.org/zesje/zesje/`) 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.
>
> Within this bachelor project you will extend the open source exam grading software. You will implement a module:
>
> - that processed user-generated pdfs of exam forms
> - Identifies coordinates of all boxes for answers and multiple choice questions
> - Prepares multiple copies of these files and adds auto-generated QR codes and markup for automated processing
>
> Further in the pipeline you will use computer vision libraries to implement identification of student number and student answers from the scanned exam. You will also extend the webapp to implement the user interface for generating exam PDFs.
>
> 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.
>
> This project is together with Anton Akhmerov (`hi@antonakhmerov.org`), associate professor at Delft University of Technology, where he's one of the leaders of the Quantum Tinkerer.

C

Project plan

C.1. Goal

The goal of the project is to improve the *Zesje* exam grading tool.

The main subgoal of the project is to remove the dependency in Zesje on LaTeX. This will be done by building a new exam generation component, which will not be LaTeX-based. If this subgoal is completed, the client considers the project successfully completed.

However, if the aforementioned subgoal has been completed and more time is left in the project, other subgoals include:

- further automating the exam generation and grading processes
- expanding Zesje to support a decentralized workflow in which Zesje is run on the local machines of graders
- expanding Zesje to also be able to generate and grade homework assignments

C.2. Design goals

Design goals assist the team in making choices during the research and development phase. Concluding from a meeting with the client, the members of the team consider the following design goals as most important. Each design goal is named and briefly explained.

C.2.1. Maintainability

The client has expressed that it's of importance that the source code should stay open source. Also, as new ideas are created, it should be easy for future developers to extend on the end product. Therefore, it's of importance that the software is maintainable.

C.2.2. Performance

One of the reasons the client is seeking for us to fulfill the requirements of the project is the inadequate performance of the current LaTeX-based solution. Therefore, our solution should be faster than the current solution.

C.2.3. Usability

For teachers, the system should be easy to use and to understand. Teachers should not need a very complicated guide on how to use Zesje. It should also feel more satisfying to use Zesje than to grade exams by paper. Thus, usability is something that also should be valued.

C.3. Requirements

The requirements of the project are categorized using the MoSCoW method. They are formulated at the start of the project during the research period and are as follows:

C.3.1. Must Have

- Ability to upload an exam (as PDF) via the frontend of Zesje
- Detection of whitespace in uploaded exams for inserting corner markers, student identification grids and barcodes (e.g. QR codes)
- Insertion of barcodes to uniquely identify exam instances and pages.
- Insertion and usage of corner markers to orient a scan
- Insertion of student identification widgets
- Reading out of student numbers
- Ability to view and edit locations of answer boxes
- Ability to preview an exam including the mentioned insertions
- Ability to generate multiple copies of exams for printing

C.3.2. Should Have

- Ability for students to get extra answer space without having to submit duplicate exams
- Ability to generate homework assignments and e-mail these to students
- Reading out and automatically linking feedback to multiple choice answers

C.3.3. Could Have

- Automatic detection of locations of multiple choice questions and open answer boxes
- Possibility for teachers to send only answers back to students, while keeping questions secret
- Ability to run Zesje on Windows and macOS
- Possibility to import/export data from the database to share with other Zesje instances

C.3.4. Won't Have

- Ability to authenticate using SAML

C.4. Technologies

On a high level, Zesje consists of two major components: the frontend and the backend. The frontend and backend are implemented in React and Python 3 respectively. SQLite is used by the backend to store data about the exams. Git is used as source control system and hosted by the client. For this project, we will keep using these technologies. If there is a need for a new technology, the team will consider integration with the existing implementation.

C.5. Methodology

The team will use ideas from SCRUM (an Agile methodology for software development).

The team will itemize every requirement into units of work into tasks. These tasks will be held in a product backlog (i.e. a list of tasks that need to be done, originating from the requirements).

The implementation phase is split into time periods called sprints. The length of a sprint for this project will be one week. Sprints will start every Monday and will end every Friday.

At the start of every sprint, the team will hold a planning meeting, in which the team members will move a number of tasks from the product backlog to the sprint backlog. The team will work on the tasks in the sprint backlog during the sprint. At the end of every sprint, the team will hold a retrospective meeting, in which the team looks back on the sprint and identifies possible improvements for future sprints. Furthermore, at the end of each sprint, the team will meet with the client to show the current progress and receive feedback.

This methodology is chosen because the team is familiar with it and fits the length and pacing of the project. It also gives the client the opportunity to give frequent feedback to steer the team in the right direction.

C.6. Quality assurance

In this section, the methods used to assure the quality of our system will be named. Section C.6.1 details how the team will accomplish this through testing, while section C.6.2 details how this will be done using code reviews.

C.6.1. Testing

The team will make an effort to write automated tests for most of their code. The existing Zesje project contains no tests, and for this reason, it is unknown to the team how much of the new code will be able to be written in a testable way while also remaining interoperable with the current code. Therefore, the team can currently not provide a concrete quantification for how much of their code will be accompanied by automated tests.

The team will also conceive a number of use cases. The team will investigate whether it is a lot of

effort to write automated use case tests. If not, the team will write automated tests for these use cases. Otherwise, manual use case tests will be performed.

C.6.2. Code reviews

The team will use the merge request workflow (also known as the pull request workflow). Before a merge request will be able to be merged, at least one of the members of the Zesje development team (i.e. the project team and the other Zesje developers) will have to review and approve the changes in the merge request.

C.7. Planning

The project will consist of two phases: the research phase and the development phase. The research phase starts at the beginning of week 17 (4.1) and ends at the end of week 18 (4.2). The development phase starts right after in week 19 (4.3), and ends with the final presentation in week 27 (4.11).

The research phase consists of the exploration of various possible approaches for the given problem. Approaches are compared to each other and the most optimal approach is chosen. The end product of the research phase is the research report, summarizing the research done in this period.

The development phase consists of the implementation of the optimal solution researched in the research phase, using the methodology as specified in C.5. Examples of deliverables which are submitted during this phase are the final report and final presentation.

Throughout both phases, we will also consider relevant literature in the field to support decision-making processes.

The important dates and deadlines can be found below:

- 2018, May 4 - End of research phase and deadline research report
- 2018, May 7 - Start of development phase
- 2018, June 1 - First code submission to SIG
- 2018, June 22 - Second code submission to SIG
- 2018, June 25 - Final report submission
- 2018, July 3 - Final presentation

C.8. Team members

The team members are expected to work on the project for eight hours per working day (official university holidays excluded). The location (at the teaching lab or at home) will be discussed by the team members. If a team member will not be able to work for eight hours per day, the other team members should be notified of this. The absent team member should then compensate for the missed hours in their own free time.

Glossary

CI continuous integration; the practice of automatically running tests etc. upon pushing commits to the repository 35

duplex *(in printing)* double-sided 34

Git a free, open source, fast and efficient version control system 19

GitLab a Git repository hosting service with features such as an issue tracker, merge request management and continuous integration 14, 19, 35

LaTeX a markup language and document typesetting software especially suited for scientific documents 2, 7, 8, 13, 21, 36

linter refers to tools that analyze source code to detect faults such as but not limited to: syntax errors, stylistic errors and programming errors 29

Mattermost an open source messaging application for teams 20

ORM object-relational mapping; a library helping to access and update data in a relational database in an object-oriented way 21

pdfrw a Python library for modifying and reading existing PDF files 14, 35

Pony an ORM for Python 21

PyPDF2 a Python library for modifying and reading existing PDF files 14, 34, 35

pytest a Python testing framework 29

QR code (barcode) a type of 2 dimensional barcode 10, 11, 13–17, 34, 38

qrcode (Python library) a Python library for QR code generation 14

React a JavaScript library for building user interfaces 12, 21, 32

ReportLab a Python library for creating PDF files 14

RESTful adhering to the principles of REST 21

SCRUM an agile framework for managing work with an emphasis on software development 19

SQLite a light-weight, SQL-based relational database contained in a single file 21

Trello a web-based project management board 19

widget *(in Zesje)* any 'box' or overlay that can be added on an exam. Problems, student identification grid and the barcodes are all widgets 24

YAML a human-readable data serialization language 7

References

[1] Arjun Singh et al. “Gradescope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work”. In: *Proceedings of the Fourth ACM Conference on Learning @ Scale, L@S 2017, Cambridge, MA, USA, April 20-21, 2017*. 2017, pp. 81–88.

[2] Nina V. Stankous. “Constructive Response Vs. Multiple-Choice Tests In Math: American Experience And Discussion”. In: *European Scientific Journal, ESJ* 12.10 (2016).

[3] Aaron Bloomfield and James F. Groves. “A Tablet-based Paper Exam Grading System”. In: *SIGCSE Bull.* 40.3 (June 2008), pp. 83–87.

[4] Adrian Cobb. [Accessed on 01/05/2018]. URL: https://www.dynamsoft.com/blog/barcode-reader/the-comprehensive-guide-to-1d-and-2d-barcodes/.

[5] *Sonate*. [Accessed on 21/06/2018]. URL: www.icto.tudelft.nl/en/tools/sonate/.

[6] *PDF.js*. [Accessed on 02/05/2018]. URL: https://mozilla.github.io/pdf.js.

[7] *GitHub: mozilla/pdf.js*. [Accessed on 02/05/2018]. URL: https://github.com/mozilla/pdf.js/blob/master/README.md#firefox.

[8] *GitHub: mikecousins/react-pdf-js*. [Accessed on 06/05/2018]. URL: https://github.com/mikecousins/react-pdf-js.

[9] *GitHub: mzabriskie/react-draggable*. [Accessed on 01/05/2018]. URL: https://github.com/mzabriskie/react-draggable.

[10] *Wand*. [Accessed on 07/05/2018]. URL: http://docs.wand-py.org/en/0.4.4/.

[11] N. Senthilkumaran and S. Vaithegi. “Image Segmentation By Using Thresholding Techniques For Medical Images”. In: *Computer Science & Engineering: An International Journal* 6 (Feb. 2016), pp. 1–13.

[12] Sachin Kumar S et al. “Text/Image Region Separation for Document Layout Detection of Old Document Images Using Non-linear Diffusion and Level Set”. In: *Procedia Computer Science* 93 (Dec. 2016), pp. 469–477.

[13] *OpenCV*. [Accessed on 30/04/2018]. URL: https://opencv.org/.

[14] *GitHub: lincolnloop/python-qrcode*. [Accessed on 02/05/2018]. URL: https://github.com/lincolnloop/python-qrcode.

[15] *GitHub: mnooner256/pyqrcode*. [Accessed on 02/05/2018]. URL: https://github.com/mnooner256/pyqrcode.

[16] *GitHub: mmulqueen/pyStrich*. [Accessed on 13/06/2018]. URL: https://github.com/mmulqueen/pyStrich.

[17] *GitHub: NaturalHistoryMuseum/pylibdmtx*. [Accessed on 13/06/2018]. URL: https://github.com/NaturalHistoryMuseum/pylibdmtx.

[18] *GitHub: reingart/pyfpdf*. [Accessed on 03/05/2018]. URL: https://github.com/reingart/pyfpdf.

[19] Sepideh Barekat Rezaei, Hossein Sarrafzadeh, and Jamshid Shanbehzadeh. “Skew detection of scanned document images”. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists 2013* (2013).

[20] M. Basavanna and S.S. Gornale. “Skew Detection and Skew Correction in scanned Document Image using Principal Component Analysis”. In: *International Journal of Scientific & Engineering Research* 6.1 (2015).

[21] Konstantinos G. Derpanis. “The Harris Corner Detector”. 2004.

[22] Bhavesh Kumar, Gautam Kumar, and Ashish Kumar. “An approach for Skew Detection using Hough Transform”. In: *International Journal of Computer Applications* 136 (Feb. 2016), pp. 20–23.

[23] Chandan Singh, Nitin Bhatia, and Amandeep Kaur. “Hough transform based fast skew detection and accurate skew correction methods”. In: *Pattern Recognition* 41.12 (2008), pp. 3528–3546.

[24] Ibrahim S.I. Abuhaiba. “Skew correction of textural documents”. In: *Journal of King Saud University - Computer and Information Sciences* 15 (2003), pp. 73–93.

[25] A.K. Das and B. Chanda. “A fast algorithm for skew detection of document images using morphology”. In: *International Journal on Document Analysis and Recognition* 4.2 (Dec. 2001), pp. 109–114.

[26] Jonathan J. Hull. “Document image skew detection: Survey and annotated bibliography”. In: World Scientific, 1998, pp. 40–64.

[27] Bishakha Jain and Mrinaljit Borah. “A Comparison Paper on Skew Detection of Scanned Document Images Based on Horizontal and Vertical Projection Profile Analysis”. In: *International Journal of Scientific and Research Publications*. 2014.

[28] *GitHub: zplab/zbar-py*. [Accessed on 30/04/2018]. URL: `https://github.com/zplab/zbar-py`.

[29] *Zesje: `zesje/helpers/pdf_helper.py`, line 224*. Commit: `7b28c58d` [Accessed on 01/05/2018]. URL: `https://gitlab.kwant-project.org/zesje/zesje/blob/7b28c58d/zesje/helpers/pdf_helper.py#L224`.

[30] *GitHub: zplab/zbar-py - Window build errors*. [Accessed on 30/04/2018]. URL: `https://github.com/zplab/zbar-py/issues/8`.

[31] *GitHub: NaturalHistoryMuseum/pyzbar*. [Accessed on 30/04/2018]. URL: `https://github.com/NaturalHistoryMuseum/pyzbar/`.

[32] *GitHub: primetang/qrtools*. [Accessed on 26/04/2018]. URL: `https://github.com/primetang/qrtools`.

[33] *GitHub: Polyconseil/zbarlight*. [Accessed on 26/04/2018]. URL: `https://github.com/Polyconseil/zbarlight`.

[34] Zephor5 (GitHub). *how_to_install_zbarlight_on_windows.md*. [Accessed on 30/04/2018]. URL: `https://gist.github.com/Zephor5/aea563808d80f488310869b69661f330/286554fc73cfd678ec61dc14aa38e303e1c1770d`.

[35] *GitHub: ewino/qreader*. Commit: `5c94f17`. URL: `https://github.com/ewino/qreader/tree/5c94f1707f4fb79e6e3365d44cd648aafb4bc68b`.

[36] *GitHub: ewino/qreader - Bug*. [Accessed on 26/04/2018]. URL: `https://github.com/ewino/qreader/issues/6`.

[37] *ZBar bar code reader*. [Accessed on 13/06/2018]. URL: `http://zbar.sourceforge.net/`.

[38] *ZBar bar code reader*. [Accessed on 13/06/2018]. URL: `https://sourceforge.net/projects/zbar/`.

[39] *GitHub: dmtx/libdmtx*. [Accessed on 13/06/2018]. URL: `https://github.com/dmtx/libdmtx`.

[40] *GitHub: infoscout/weighted-levenshtein*. [Accessed on 02/06/2018]. URL: `https://github.com/infoscout/weighted-levenshtein`.

[41] Karen Kukich. “Techniques for Automatically Correcting Words in Text”. In: *ACM Computing Surveys* 24.4 (Dec. 1992), pp. 377–439.

[42] *GitHub: NaturalHistoryMuseum/pylibdmtx - Decoding on MacOS gives raw data*. [Accessed on 21/06/2018]. URL: `https://github.com/NaturalHistoryMuseum/pylibdmtx/issues/24`.

[43] John Malouff. “Bias in Grading”. In: *College Teaching* 56.3 (2008), pp. 191–192.

[44] John M Malouff and Einar B Thorsteinsson. “Bias in grading: A meta-analysis of experimental research findings”. In: *Australian Journal of Education* 60.3 (2016), pp. 245–256.

[45] Robert Person. “Blind Truth: An Examination of Grading Bias”. 2013.

[46] John M. Malouff, Ashley J. Emmerton, and Nicola S. Schutte. “The risk of a halo bias as a reason to keep students anonymous during grading”. In: *Teaching of Psychology* 40.3 (2013), pp. 233–237.

[47] Meike Bonefeld and Oliver Dickhäuser. “(Biased) Grading of Students’ Performance: Students’ Names, Performance Level, and Implicit Attitudes”. In: *Frontiers in Psychology* 9 (2018), p. 481.

[48] Kathryn J. Morris. “Does Paper Presentation Affect Grading: Examining the Possible Educational Repercussions of the Quality of Student Penmanship”. Honors Theses. 30. 2014.