

---

# PRAKTIKUMSBERICHT

---

Herr  
**Hannes Steiner**

## **Entwicklung einer Dokumenten-Scanner-App**

**Digitalisierung der Verwaltung an der Hochschule  
Mittweida**

2020

Fakultät **Angewandte Computer- und  
Biowissenschaften**

---

# **PRAKTIKUMSBERICHT**

---

## **Entwicklung einer Dokumenten-Scanner-App**

**Digitalisierung der Verwaltung an der Hochschule  
Mittweida**

Autor:  
**Hannes Steiner**

Studiengang:  
Softwareentwicklung

Seminargruppe:  
IF17wS-B

Matrikelnummer:  
46540

Erstprüfer:  
Prof. Dr. Mark Ritter

Zweitprüfer:  
N.N.

Mittweida, März 2020

---

## **Bibliografische Angaben**

Steiner, Hannes: Entwicklung einer Dokumenten-Scanner-App, Digitalisierung der Verwaltung an der Hochschule Mittweida, 40 Seiten, 13 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Praktikumsbericht, 2020

Dieses Werk ist urheberrechtlich geschützt.

## **Referat**

Im Rahmen eines zwölfwöchigem Forschungspraktikums an der Hochschule Mittweida arbeiteten der Student Tobias Kallauke und der Verfasser des Berichts gemeinsam an einem Projekt mit dem Hintergrund der Digitalisierung der Verwaltung von Lehr- und Forschungseinrichtung. Dabei entwickelten sie ein prototypisches Software-System, womit Hochschulmitarbeiter kontrollierte Klausuren einscannen und digitalisieren können, sodass das Übertragen der Noten in das Notensystem nicht mehr per Hand erledigt werden muss.

# I. Inhaltsverzeichnis

Inhaltsverzeichnis . . . . .	I
Abbildungsverzeichnis . . . . .	II
Abkürzungsverzeichnis . . . . .	III
1 Einleitung . . . . .	1
2 Hochschule Mittweida . . . . .	2
3 Problemstellung . . . . .	3
3.1 Digitalisieren der Klausur-Daten . . . . .	3
3.2 Klausuren-Vorlage . . . . .	4
3.3 Klausuren-Vorlage verbessern . . . . .	4
3.4 Weitere Anmerkungen . . . . .	4
4 Anforderungen . . . . .	5
5 Konzept der Dokumenten-Scanner-App . . . . .	7
5.1 Scan-Vorlage erstellen und speichern . . . . .	7
5.2 Scan-Vorlage verwenden . . . . .	9
5.3 Erweiterung des Konzepts . . . . .	10
6 Entwicklung des ersten Prototyps . . . . .	12
6.1 Anforderungsplanung . . . . .	12
6.2 Planung und Vorbereitung bzw. Analyse und Definition . . . . .	12
6.3 Grundlagen . . . . .	13
6.4 Entwurf und Design . . . . .	15
6.5 Implementierung . . . . .	17
6.5.1 Registrieren und Anmelden . . . . .	18
6.5.2 Scan-Vorlagen erstellen und speichern . . . . .	19
6.5.3 Scan-Vorlage verwenden . . . . .	23
7 Fazit . . . . .	28
7.1 Stand des Prototyps . . . . .	28
7.2 Probleme und Grenzen der App . . . . .	28
7.2.1 Probleme beim Erkennen von Dokumenten . . . . .	28

---

7.2.2	Probleme der Klausur-Vorlage beim Scannen . . . . .	29
7.2.3	Weitere Probleme der App . . . . .	30
7.3	Verbesserung der Klausuren-Vorlage . . . . .	30
8	Ausblick . . . . .	32
A	Workflow . . . . .	35
B	Tätigkeitsbericht . . . . .	36
	Literaturverzeichnis . . . . .	39

## II. Abbildungsverzeichnis

3.1 Umsetzung der Klausuren-Vorlage der Fakultät CB. . . . .	3
5.1 Schematisches Schaubild zum Verständnis des Schablonen-Konzepts . . . . .	7
5.2 Scan-Vorlage erstellen und auf dem Server speichern . . . . .	8
5.3 Schematischer Ablauf beim Verwenden einer Scan-Vorlage . . . . .	9
5.4 Scan-Vorlage verwenden . . . . .	10
6.1 Architektur Schemata . . . . .	16
6.2 Willkommen- Registrier- und Anmelde-View . . . . .	18
6.3 Die ersten Views zur Erstellung einer Scan-Vorlage . . . . .	19
6.4 Views zur Erstellung von Regionen . . . . .	20
6.5 Seiten-Vorschau-View . . . . .	21
6.6 Views zur Erstellung von Kontrollmechanismen . . . . .	22
6.7 Listen- und Detail-Ansicht von Scan-Vorlagen . . . . .	23
6.8 Vorlagen-Ansicht vor und nach der Texterkennung . . . . .	24

### III. Abkürzungsverzeichnis

API .....	application programming interface, deutsch: Programmierschnittstelle, Seite 17
CRUD .....	Das Akronym CRUD steht für Create/Erstellen, Read/Lesen, Update/Aktualisieren und Delete/Löschen und umfasst die vier grundlegenden Operationen persistenter Speicher, Seite 6
Fakultät CB .....	Fakultät für angewandte Computer- und Biowissenschaften, Seite 2
HSMW .....	Hochschule Mittweida - university of applied science, Seite 2
ID .....	Steht für einen einzigartigen Identifikator, Seite 22
IDE .....	integrated development environment, deutsch: Integrierte Entwicklungsumgebung, Seite 17
MVVM .....	Model View ViewModel, Seite 13
OCR .....	optical character recognition, deutsch: optische Zeichenerkennung und im deutschen Synonym für Texterkennung, Seite 9

# 1 Einleitung

Digitalisierung wird gängig als Integration von digitaler Technologie in den Alltag verstanden und soll helfen Zeit einzusparen. Mit diesem Gedanken initiierten die Mitarbeiter Holger Langner und Falk Schmidsberger der Hochschule Mittweida das Projekt *Memo Space*. Im Zuge dessen sollen kleinere Forschungsergebnisse entstehen, die richtungsweisend für die Digitalisierung der Verwaltung von Lehr- und Forschungseinrichtung sind.

Im Rahmen eines zwölfwöchigem Forschungspraktikums an der Hochschule Mittweida arbeiteten der Student Tobias Kallauke und der Verfasser des Berichts gemeinsam an einem Forschungsprojekt von Memo Space. Dabei entwickelten sie ein prototypisches Software-System, mit dem die Arbeit von vielen Hochschulmitarbeitern erleichtert und auch Zeit eingespart werden soll.

## 2 Hochschule Mittweida

Die Hochschule Mittweida - university of applied science (HSMW) wurde vor über 150 Jahren gegründet. Heute lehrt und forscht sie mit ca. 6000 Studenten in fünf Fakultäten und vier Forschungsschwerpunkten [5]. Eines der Schwerpunkte ist die Angewandte Informatik, in dem Memo Space angesiedelt ist.

Nach eigener Einschätzung, schreibt jeder Student der HSMW ca. 5 Prüfungen pro Semester. Das bedeutet, dass im Jahr um die 60.000 Klausuren kontrolliert werden müssen. Dazu kommt, dass die Zensuren, sowie die Eckdaten der Studenten, per Hand in ein digitales Format gebracht werden. Grund dafür ist die Übertragung der Noten in das Notensystem der Hochschule. Da die Mitarbeiter der Fakultät *Angewandte Computer- und Biowissenschaften* (Fakultät CB) Holger Langner und Falk Schmidsberger selbst Klausuren kontrollieren und die Problematik genau kennen, entstand hier eine der ersten Ideen für Memo Space.

## 3 Problemstellung

An der Kontrolle von Klausuren arbeiten zum Ende eines Semesters Hochschulmitarbeiter über Tage. Diese Aufgabe muss stets mit hoher Konzentration erledigt werden und lässt sich in den meisten Fällen nur schwer durch Maschinen ersetzen. Unter keinen Umständen dürfen bei der Bewertung Fehler unterlaufen, was jedoch bei der kognitiven Belastung der Prüfer immer wieder passiert. Auch bei der anschließenden Eingabe der Noten in eine digitale Tabelle ist hohe Achtsamkeit notwendig. In diese muss die Matrikelnummer, der Vor- und Nachname sowie die Note des Studenten eingetragen werden. Hier kommt es vor allem bei der Matrikelnummer und der Zensur auf die Richtigkeit jedes Zeichens darauf an.

### 3.1 Digitalisieren der Klausur-Daten

Für genau diesen Vorgang des Digitalisierens wird eine Lösung gesucht. Die Prüfer sollen effektiv und möglichst zeitsparend diese Aufgabe verrichten, ohne dabei ihre Aufmerksamkeitsspanne zu überlasten. Des Weiteren müssen die Ergebnisse der Prüfungen, sowie die Eckdaten der Studenten in ein geeignetes digitales Format gebracht werden, um es der Notenfreigabe weiterzuleiten. Darüber hinaus empfiehlt es sich, digitale Kopien der Klausuren abzuspeichern, um sie nicht nur analog zu archivieren.

**(a) Deckblatt der Klausur**

**(b) Erste Seite der Klausur mit drei Aufgaben**

Abbildung 3.1: Umsetzung der Klausuren-Vorlage der Fakultät CB.

*Die Bilder sind beim Erstellen einer Scan-Vorlage mit der App entstanden.*

## 3.2 Klausuren-Vorlage

Eine Klausuren-Vorlage bzw. ein Gestaltungsleitfaden für Klausuren der Fakultät *Ange-wandte Computer- und Biowissenschaften* bietet außerdem die Möglichkeit der Kontrolle des Prüfers. Durch das vorgegebene Layout der Klausur ist es möglich, Fehler des Prüfers zu erkennen. Die Klausur in Abbildung 3.1 ist nach dieser Klausuren-Vorlage angefertigt worden. Auf dem Deckblatt (3.1a) der Prüfung ist eine Tabelle, mit drei Zeilen und für jede Aufgabe eine Spalte. In der ersten Zeile befinden sich die Nummern der Aufgaben, in der zweiten die zu erreichenden Punkte der Aufgabe. Und in der dritten Zeile trägt der Prüfer die erbrachten Punkte des Studenten ein. Unter der Tabelle befindet sich ein Feld für die erreichte Gesamtpunktzahl sowie ein weiteres für die aus den Punkten resultierende Note. Es soll nun überprüft werden, ob die Summe der erreichten Punkte mit der Gesamtpunktzahl übereinstimmt und weiter soll auch die Note errechnet und mit dem Ergebnis der Prüfers verglichen werden. Eine weiteres Merkmal der Klausuren-Vorlage ist ein Feld, für die vom Studenten erreichten Punkte über jeder Aufgabenstellung auf den hinteren Seiten. Zu sehen sind solche Felder in Abbildung 3.1b am rechten Seitenrand. Die dortige Angabe sollte mit der, in der Tabelle auf dem Deckblatt (3.1a) übereinstimmen und bietet somit noch eine weitere Möglichkeit der Kontrolle an.

## 3.3 Klausuren-Vorlage verbessern

Nachdem ein erster Prototyp zum Digitalisieren der Klausur-Daten entwickelt wurde, sollen außerdem Prüfungs-Vorlagen entstehen, die für die Digitalisierung optimiert sind. Dabei sollen Probleme, die beim Einstellen der aktuellen Vorlage erkannt wurden, behoben werden.

## 3.4 Weitere Anmerkungen

Ferner soll bei der Problemlösung von der Anschaffung neuer Technologien und Geräte abgesehen werden. Grund dafür ist neben den Anschaffungskosten, die Idee, dass das Ergebnis des Forschungsprojekts in weiteren Lehr- und Forschungseinrichtungen Anwendung findet. Außerdem hat so gut wie jeder Mitarbeiter an einer Lehr- und Forschungseinrichtung ein eigenes oder Zugang zu einem Smartphone oder Tablet, welche durch die eingebaute Technik in der Lage sind, diese Aufgabe zu übernehmen.

## 4 Anforderungen

In diesem Kapitel wird erläutert, was die Anforderungen an das Software-System sind.

Es soll eine App für das Betriebssystem iOS entstehen, mit der Klausuren digitalisiert werden können. Genauer müssen die, für die Notenfreigabe relevanten Daten der Klausur, in ein tabellarisches Format gebracht werden. Dazu soll die iOS-Anwendung Bilder von ausgefüllten Prüfungen aufnehmen können und den Inhalt mithilfe von Texterkennung digitalisieren.

Trotzdem nicht jede Klausur ein identisches Layout hat, soll die App in der Lage sein, alle relevanten Informationen, wie z. B. Name, Matrikelnummer und Note, richtig zu identifizieren und anschließend zu digitalisieren. Dafür soll der Benutzer digitale Vorlagen, sogenannte Scan-Vorlagen erstellen, durch die die App, die Informationen leichter findet und schneller digitalisiert, als wenn eine komplette Seite digitalisiert und analysiert werden müsste.

Zusätzlich soll es dem Benutzer möglich sein, Kontrollmechanismen fest zu legen. Wie im Abschnitt 3.2 beschrieben, soll überprüft werden, ob die Punkte auf dem Deckblatt mit den Punkten auf den hinteren Seiten übereinstimmen und ob die Gesamtpunktzahl korrekt summiert wurde sowie die daraus resultierende Note richtig ist. Ein Kontrollmechanismus soll so gestaltet sein, dass die eben genannten Überprüfungsmöglichkeiten vom Benutzer selbst umgesetzt werden können. Beim Digitalisieren einer Klausur sollen so dem Benutzer Fehlermeldungen angezeigt werden, wenn bei der Überprüfung durch einen Kontrollmechanismus etwas falsch ist. Beispielsweise hat der Prüfer auf dem Deckblatt fünf Punkte eingetragen, bei der Aufgabenstellung auf den hinteren Seiten jedoch eine sechs. Die App soll dann eine Fehlermeldung ausgeben, dass die Punktzahlen bei der Aufgabe nicht übereinstimmen und nochmals kontrolliert werden müssen.

Zur Digitalisierung von Buchstaben soll Zeichen bzw. Texterkennung verwendet werden, die auf dem Gerät statt findet. Allerdings darf die App auch Texterkennung auf externen Servern unterstützen. Hierfür müssen dann die entsprechenden Server-Schnittstellen zur Kommunikation implementiert oder für einen eigenen Server entwickelt werden.

Die digitalisierten Daten, die für die Texterkennung entstandenen Klausuren-Bilder und die erstellten digitalen Scan-Vorlagen sollen außerhalb der App auf einem Server gespeichert werden. Somit ist die Möglichkeit gegeben, die Vorlagen wieder zu verwenden und anderen Benutzern der App zur Verfügung zu stellen. Aus den digitalisierten Daten sollen wiederum Tabellen entstehen, die an die Notenfreigabe weiter geben werden können.

Damit Synchronität der Daten auf den Geräten und dem Server gewährleistet werden kann, benötigt die App hierfür ebenfalls Schnittstellen. Diese sollten den standardmäßigen *CRUD*-Operationen entsprechen.

Da die gesamte Kommunikation über das Internet geschieht, muss das Softwaresystem den üblichen Datenschutz- und Datensicherheit-Richtlinien entsprechen, bzw. sie in der technischen Implementierung umsetzen. Für genauere Details zu den Sicherheitsmechanismen und den *CRUD*-Operationen zum Server siehe im Praktikumsbericht von Tobias Kallauke.

## 5 Konzept der Dokumenten-Scanner-App

Diesem Kapitel beschreibt, wie das Softwaresystem die Anforderungen umsetzt. Zur Erinnerung, die Dokumenten-Scanner App soll wichtige Daten auf dem Deckblatt (3.1a) einer Klausur, wie Vor- und Nachname des Studenten, seine Matrikelnummer sowie die Note erkennen, digitalisieren und in ein für die Notenfreigabe geeignetes Format bringen. Die digitalisierten Daten sollen anschließend an einen Server gesendet werden, auf dem sie und die beim Einstellen entstandenen Bilder gespeichert werden. Verwendet eine einzuscannende Klausur die Klausuren-Vorlage (3.1) der Fakultät CB oder ähnliche Vorlagen, die eine Punkteübersicht haben, dann ist es außerdem möglich, die Punkte sowie die Note auf Richtigkeit zu überprüfen.

### 5.1 Scan-Vorlage erstellen und speichern

Im folgenden Abschnitt wird zuerst verkürzt und anschließend detailliert geschildert, wie eine Scan-Vorlage zu erstellen ist.

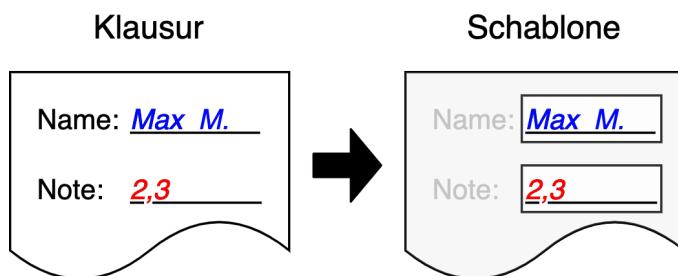


Abbildung 5.1: Schematisches Schaubild zum Verständnis des Schablonen-Konzepts

Um wirklich nur relevante Daten zu digitalisieren, benötigt es die sogenannte Scan-Vorlage, die in der App angelegt werden kann. Eine Scan-Vorlagen funktioniert ähnliche, wie eine Schablone. Diese wird beispielsweise auf das Deckblatt gelegt, sodass nur noch die wichtigen Informationen, die digitalisiert werden sollen, zu sehen sind. Alles andere, was nicht von Interesse ist, wird von der Schablone überdeckt. In der Abbildung 5.1 ist dieses Konzept schemenhaft zu sehen. Aus dem analogen Beispiel abgeleitet, benötigt es zwei grobe Schritte, um eine digitale Schablone anzufertigen. Zuerst muss ein Foto von der Seite aufgenommen werden und anschließend müssen die Regionen eingezeichnet werden, die digitalisiert werden sollen. Eine Scan-Vorlage ist jedoch nicht nur eine einzige digitale Schablone sondern einer Sammlung aller zu einem Dokument. Denn jede Seite benötigt eine eigene individuelle.

In der folgenden Aufzählung wird nun detailliert beschrieben, wie die Abfolge beim Erstellen einer Scan-Vorlage ist. Der Vorgang ist in Abbildung 5.2 schematisch dargestellt.

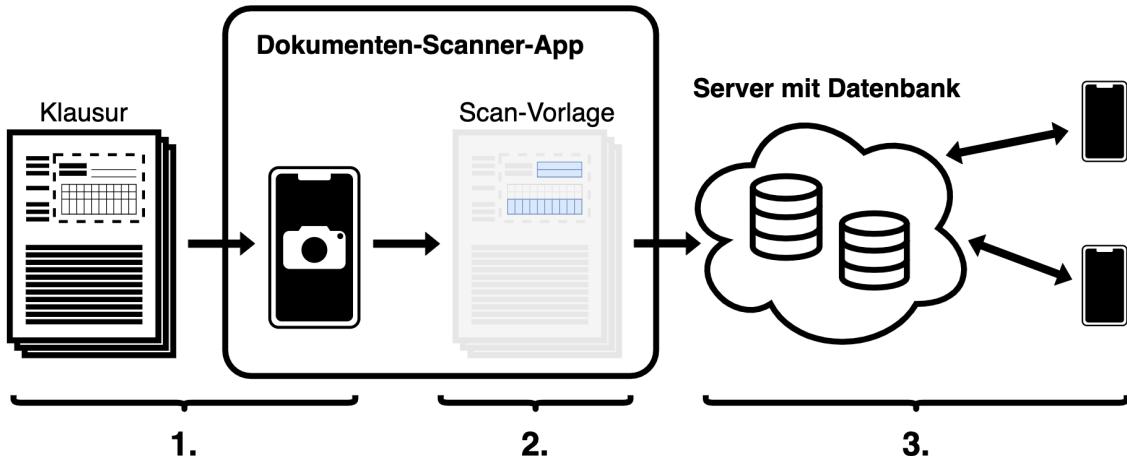


Abbildung 5.2: Scan-Vorlage erstellen und auf dem Server speichern

1. Zu Beginn wird jede Seite der Klausur fotografiert. Dabei wird in jedem Bild das Dokument erkannt, vom Hintergrund getrennt bzw. ausgeschnitten und perfekt ausgerichtet. Zudem wird der Kontrast erhöht, sodass die Schrift leichter lesbar wird. Die Abbildung 3.1 resultierte aus dem Prozess (siehe auch Abbildung 6.3b).
2. Anschließend markiert der Benutzer diejenigen Regionen auf jeder Seite, die zu digitalisieren und/oder zu kontrollieren sind. In dem Schema 5.2 sind die Regionen blau hinterlegt (siehe auch Abbildung 6.5b). Zusätzlich muss jeder Region eine Bezeichnung und einer der folgenden Datentypen zugeordnet werden: Unbekannt, Vorname, Nachname, Matrikelnummer, Seminargruppe, Punkte und Note (siehe auch Abbildung 6.5a und 6.5b). Die Angabe des Datentyps ist wichtig, da dadurch eindeutig wird, ob es sich um die Note oder den Vornamen des Studenten handelt. Diese Eindeutigkeit wird nicht nur für die automatische Erstellung der Tabelle benötigt, sondern auch für die Optimierung der Texterkennung.
3. Als letztes wird die Vorlage abgespeichert, welche dann automatisch an einen Server gesendet wird, so dass andere diese Vorlage ebenfalls benutzen können.

Zusätzlich können in der Phase der Erstellung einer Scan-Vorlage die in Kapitel 4 beschriebenen Kontrollmechanismen festgelegt werden. Dies geschieht nach Punkt 2, also nachdem alle Regionen angelegt wurden. Diese Kontrollmechanismen beruhen auf zwei simplen Konzepten.

Das erste Konzept ist der Vergleich. Zur Erinnerung, es sollen die eingetragenen Punkte auf dem Deckblatt mit den Punkten neben der Aufgabenstellung auf den hinteren Seiten auf Gleichheit überprüft werden. Das zweite Konzept baut auf dem ersten auf. Statt einfach zwei Regionen bzw. Komponenten zu vergleichen, wird eine oder werden beide zuvor berechnet. Beispielsweise soll die Summe der Teilaufgaben mit Gesamtpunktzahl übereinstimmen. Das bedeutet, es wird erst die Summe berechnet und anschließend mit der vom Prüfer eingetragenen Punktzahl verglichen. Oder aber die Gesamtpunktzahl wird in die Note umgerechnet und mit der vom Prüfer eingetragenen Note verglichen.

Beim festlegen der Kontrollmechanismen wird wie folgt vorgegangen:

1. Der Benutzer wählt ein Kontrollmechanismus aus. Zur Auswahl stehen, entweder *zwei Regionen, die Gesamtpunktzahl* oder *die Note* vergleichen.
2. Aus den zuvor angelegten Regionen müssen nun die passenden Regionen ausgewählt werden. Abhängig von dem ausgewählten Kontrollmechanismus geben die beiden zu vergleichende Komponente dem Benutzer Auskunft darüber, welche und wie viele Regionen auszuwählen sind.
3. Der Kontrollmechanismus kann gespeichert werden, sobald in jeder zu vergleichenden Komponente die Mindestzahl an auszuwählenden Regionen erreicht wurde. Diese werden in der Scan-Vorlage hinterlegt und mit an den Server gesendet, sodass alle Benutzer bei den Vorlagen die selben Kontrollmechanismen besitzen.

## 5.2 Scan-Vorlage verwenden

Im folgenden Abschnitt wird zuerst verkürzt und anschließend detailliert geschildert, wie eine erstellte Scan-Vorlage zu verwenden ist und wie diese funktioniert.

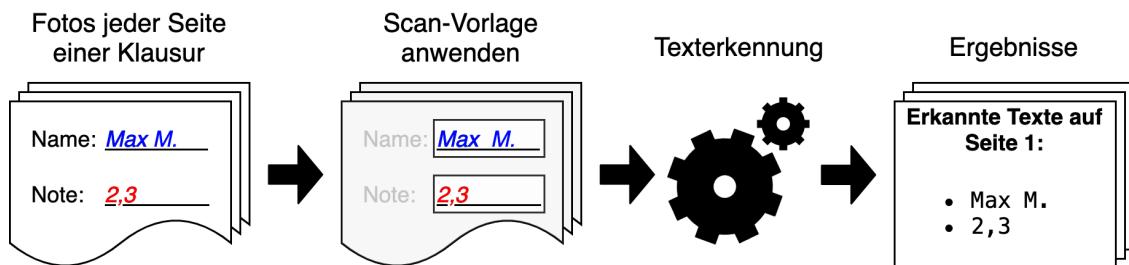


Abbildung 5.3: Schematischer Ablauf beim Verwenden einer Scan-Vorlage

Nach dem Erstellen einer Scan-Vorlage kann diese auf ausgefüllten Klausuren angewandt werden. In Abbildung 5.3 wird der Ablauf schematisch dargestellt. Zuerst werden die ausgefüllten Prüfungs-Seiten einer Klausur fotografiert. Anschließend stellen die digitalen Schablonen die relevanten Informationen jeder Seite für den nächsten Schritt bereit. Optical character recognition (OCR), auf deutsch optische Zeichenerkennung, wird dazu benutzt, die Daten aus den Schablonen zu digitalisieren.

In der folgenden Aufzählung wird detailliert beschrieben, wie die Abfolge beim Verwenden einer Scan-Vorlage ist. Der Vorgang ist in Abbildung 5.4 schematisch dargestellt.

1. Die Seiten der Klausur müssen zuerst in derselben Reihenfolge, wie in der Scan-Vorlage fotografiert werden. Anschließend werden die Regionen der Scan-Vorlage auf das jeweilige Bild angewendet. Bei dem Prozess entstehen Bildausschnitte, in denen die relevanten Informationen enthalten sind.

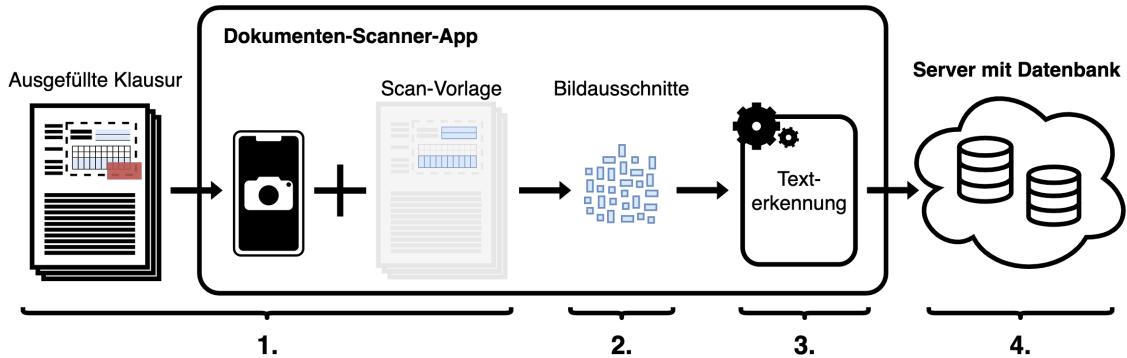


Abbildung 5.4: Scan-Vorlage verwenden

2. Die entstandenen Bildausschnitte werden Seitenweise in die Texterkennung überführt.
3. Die App digitalisiert mithilfe von OCR den Inhalt der Ausschnitte. Der Datentyp der zugehörigen Region aus der Scan-Vorlage nimmt nun Einfluss auf das Ergebnis. Durch ihn wird während der Worterkennungsphase eine Liste an vordefinierten möglichen Ergebnissen eingespeist. Diese Liste hat Vorrang vor dem sogenannten Standard-Wörterbuch und verbessert dadurch die Texterkennung. Ein Beispiel für solch eine Liste könnten, alle möglichen Noten sein. Auch vorstellbar ist, dass alle Seminargruppen oder Namen von Personen, die an der Klausur teilgenommen haben, dort verwendet werden.
4. Die Ergebnisse werden im Anschluss an den Server gesendet. Zuvor kann der Benutzer jedoch die Ergebnisse anpassen bzw. korrigieren, falls es bei der Texterkennung zu Fehlern kam. Auch die Kontrollmechanismen finden vor dem Upload zum Server statt und geben dem Nutzer durch Fehlermeldungen Hinweise.

### 5.3 Erweiterung des Konzepts

Weiter ist eine zusätzliche Möglichkeit der Korrektur sinnvoll. Angenommen, jemand verschreibt sich bei seiner Seminargruppe und vergisst ein Zeichen. Die Texterkennung erkennt zwar jedes Zeichen richtig, jedoch ist das Ergebnis keine korrekte Seminargruppe. Deshalb sollte im Nachhinein an die Texterkennung das Ergebnis mit einer richtigen Seminargruppe, die die größte Übereinstimmung mit dem erkannten Wort hat, ersetzt werden. Bei diesem Vorgehen ist zu beachten, dass nur die Seminargruppen verwendet werden, die tatsächlich die Klausur geschrieben haben. Ähnliches gilt wieder für die Namen oder Matrikelnummern der Studenten.

Allerdings entstehen auch durch das OCR Fehler. Aus diesem Grund ist die sogenannte confidence neben jedem Ergebnis sichtbar. Diese sagt aus, wie sicher sich die Texterkennung bei dem Ergebnis ist. Jedoch kann es auch hier falsche Positiv-Ergebnisse geben, weshalb auch das kein perfekter Indikator für die Richtigkeit ist. Deshalb gibt es, wie im Kapitel 5 Punkt 4 schon beschrieben, die Möglichkeit, die Ergebnisse noch

einmal zu überprüfen und zu korrigieren, bevor sie an den Server zum Abspeichern gesendet werden. Auch die Kontrollmechanismen reagieren auf die Änderungen der Ergebnisse, sodass Fehlermeldungen behoben werden können.

# 6 Entwicklung des ersten Prototyps

Dieses Kapitel beschreibt die Entwicklung der iOS-App in einer ähnlichen Reihenfolge, wie das bekannte Wasserfall-Model über die Verwaltung der Entwicklung großer Softwaresysteme nach Winston W. Royce [7].

## 6.1 Anforderungsplanung

Vor Beginn des Praktikums wurde diskutiert und kalkuliert, welches Thema aus dem Projekt Memo Space für ein zwölfwöchiges Praktikum geeignet ist. Dabei entstand eine Art Durchführbarkeits- / Machbarkeitsstudie, welche zur Entscheidung führte, ein Dokumenten-Scanner-Softwaresystem umzusetzen. Aufgrund der Kenntnisse und Erfahrung, soll ein Backend mit entsprechenden Schnittstellen und eine Android-App von Tobias Kallauke umgesetzt werden, während vom Verfasser eine iOS-App gefordert ist. Die iOS-Anwendung soll den Anforderungen, die im gleichnamigen Kapitel 4 zu finden sind, erfüllen. Für weitere Details über den Server und die Android-App siehe im Praktikumsbericht von Tobias Kallauke.

## 6.2 Planung und Vorbereitung bzw. Analyse und Definition

Die Aufgaben bzw. Ziele dieser Phase sind:

1. die Auseinandersetzung mit der Problemstellung und den Anforderungen,
2. das Betreiben von einer Problemanalyse,
3. die Entwicklung von Ideen und eines genauen Konzepts der iOS-App sowie
4. das Entwickeln eines ersten minimalen Prototyps, als Machbarkeitsnachweis.

Bei der Entstehung des Konzepts, welches im Kapitel 5 zu finden ist, spielen die Dokumentationen der Frameworks *Vision*<sup>1</sup>, *VisionKit*<sup>2</sup> und *PhotoKit*<sup>3</sup> von Apple eine entscheidende Rolle. Aus ihnen wird klar, welche Funktionen der App noch zu entwickeln und welche in Frameworks schon vorhanden sind. Z. B. ist das Erkennen und das gerade Ziehen von Dokumenten in Echtzeit, sowie die Texterkennung auf Bildern in *Vision* und *VisionKit* enthalten. Bei der Entwicklung des ersten Prototyps half eine Beispiel-App von Apple [1], die das Erkennen von Objekten in Standbildern mithilfe der genannten Frameworks umsetzt.

<sup>1</sup> Dokumentation von Vision - <https://developer.apple.com/documentation/vision>

<sup>2</sup> Dokumentation von VisionKit - <https://developer.apple.com/documentation/visionkit>

<sup>3</sup> Dokumentation von PhotoKit - <https://developer.apple.com/documentation/photokit>

Durch die Auseinandersetzung mit den Bibliotheken konnte festgestellt werden, dass die zu entwickelnde App nur Geräte mit iOS 13.0 oder neuer unterstützen werden kann. Grund dafür sind die Apple Frameworks *SwiftUI* und *VisionKit*. *SwiftUI* bietet die Möglichkeit, Benutzeroberflächen für alle Apple-Plattformen in der Programmiersprache *Swift* zu erstellen. Die deklarative *Swift*-Syntax ist einfach zu lesen und schnell zu schreiben, so dass es möglich ist, die App in wenigen Wochen für iPhone und iPad zu entwickeln. Als Alternative gibt es noch *UIKit* oder auch *AppKit*, die unter alle iOS Versionen funktionieren. Diese Frameworks sind allerdings nicht deklarativ sodass, Views sowohl im Code als auch in Interface-Dateien getrennt voneinander erstellt und konfiguriert werden müssen [8]. Dadurch dauert die Entwicklung einer App, im Gegensatz zu *SwiftUI* deutlich länger, wie man auch hier in dem Video<sup>4</sup> von Paul Hudson einem in der Swift-Community sehr bekannten Programmierer und Autor sieht. Sehr ähnliche Erfahrung hat der Verfasser vor dem Praktikum in seiner Freizeit mit den beiden Frameworks gemacht. *VisionKit* dagegen, ist das Framework zum Scannen der Dokumente. Auch hierfür gibt es eine Alternative. Das Framework ist von den Entwicklern von *WeTransfer* und funktioniert auch auf älteren iOS Versionen. Allerdings unterstützt *WeScan*<sup>5</sup> noch kein stapelweises Scannen. Das bedeutet, man kann immer nur ein Foto machen, welches erst abgespeichert werden muss, bevor man das nächste machen kann. Das ist für den Benutzer nicht bequem und spart wahrscheinlich auch keine Zeit. Zu Informationen zu anderen verwendeten Frameworks siehe im Kapitel 6 Abschnitt 6.5.

Abschließend zu dieser Phase ist zu erwähnen, dass das Scrum-Konzept<sup>6</sup>, welches sich für agile Softwareentwicklung anbietet, zur Projektplanung und zum Projektmanagement verwendet wurde. Als Versionsverwaltung der Software wurde Git<sup>7</sup> in Kombination mit GitHub Issues<sup>8</sup> und GitHub Project Boards als Planungstools benutzt. So konnte der Verfasser selbstständig jedem so genannten Sprint Aufgaben zuordnen und den Fortschritt nachvollziehen.

## 6.3 Grundlagen

In den folgenden zwei Absätzen werden wichtige Konzepte und Grundwissen vermittelt, die im folgenden Abschnitt 6.4 Entwurf und Design nochmal aufgegriffen werden.

**Model-View-ViewModel** Das sogenannte Entwurfsmuster Model View ViewModel (MVVM) entstand bei Microsoft im Jahr 2005 mit der *Windows Presentation Foundation* (WPF) und *Silverlight-Technologien*. Ein Entwurfsmuster ist eine bewährte Lösungsvorlage für

<sup>4</sup> SwiftUI vs UIKit – Comparison of building the same app in each framework - <https://www.youtube.com/watch?v=qk2y-TiLDZo>

<sup>5</sup> WeScan GitHub Repository - <https://github.com/WeTransfer/WeScan>

<sup>6</sup> Der Scrum Guide™ - <https://www.scrumguides.org/scrum-guide.html>

<sup>7</sup> Git Internetseite - <https://git-scm.com/>

<sup>8</sup> Mastering Issues - <https://guides.github.com/features/issues/>

wiederkehrende Entwurfsprobleme. MVVM verwendet das Konzept eines Schichtmodells und ist eine abstrakte Darstellung einer Benutzeroberfläche, in Form einer Datenstruktur. Diese enthält Daten, die auf der Benutzeroberfläche angezeigt werden sollen und Anweisungen, die auf der Benutzeroberfläche aufgerufen werden können. Dieses sogenannte ViewModel, weiß nichts von Views, wie es sonst bei anderen Entwurfsmustern üblich ist, um Daten auf der Benutzeroberfläche anzuzeigen. Stattdessen verwendet eine MVVM-View eine Bindungsfunktion (data binging) zur bidirektionalen Zuordnung von Daten aus dem ViewModel zu den jeweiligen Eigenschaften auf der View. Z. B. Einträge in einer Dropdown-Menü. Aber auch das binden von Daten aus dem Model zu Benutzereingaben durch Maus, Tastatur oder Touch-Screens ist möglich. Beispielsweise kann ein Mausklick eine Anweisung in dem ViewModel auslösen. Diese verändert einen Wert im Model, wodurch die View durch data binding aktualisiert wird. [6] [4] [3]

**Redux.js** Redux.js ist eine Bibliothek<sup>9</sup> für die Programmiersprache JavaScript. Diese stellt einen sogenannten vorhersehbaren Zustandscontainer<sup>10</sup> für JavaScript Anwendungen bereit. In diesem Container wird der Zustand der gesamten Anwendung in einem Objektbaum innerhalb eines einzigen Speichers gespeichert. Diesen Baum kann man sich vorstellen, wie eine Matroschka die weitere Puppen in sich hat. Der wichtige Unterschied zwischen einer Matroschka und einem Baum ist jedoch, dass in eine Puppe genau nur eine andere gesteckt werden kann. Bei einem Objektbaum hingegen können mehrere Objekte nebeneinander in ein Objekt "gesteckt" werden und so weiter. Im Bezug auf einen Zustandscontainer enthält dieser dann States (Objekte) auf deutsch Zustände und Daten, die unteranderem auf der Benutzeroberfläche angezeigt werden sollen. Diese Aufteilung in die States ist dazu gedacht, einer View oder mehrere zusammenhängende Views nur die wirklich benötigten Daten bereit zustellen. Diese Struktur erleichtert das Testen oder Untersuchen der Anwendung und ermöglicht es, durch hinzufügen eines neuen States den aktuellen Entwicklungsstand der Anwendung beizubehalten und dadurch den Entwicklungsprozess zu beschleunigen.

Ein weiteres wichtiges Prinzip<sup>11</sup> von Redux ist, dass die Daten in den States Schreibgeschützt sind. Die einzige Möglichkeit den Zustand zu ändern, besteht darin, eine Aktion auszulösen, die beschreibt, was passiert. Dadurch wird sichergestellt, dass weder die Views noch Netzwerk-Rückrufe jemals direkt an den Zustand ändern können. Stattdessen bringen sie nur die Absicht zum Ausdruck, den Zustand zu verändern und lösen eine Aktion aus. Da alle Änderungen zentralisiert sind und eine nach der anderen in einer strikten Reihenfolge erfolgen, gibt es weniger Programmfehler-Quellen.

<sup>9</sup> Redux.js Github Repository - <https://github.com/reduxjs/redux>

<sup>10</sup> Redux Core Concepts - <https://redux.js.org/introduction/core-concepts>

<sup>11</sup> Redux Three Principles - <https://redux.js.org/introduction/three-principles>

## 6.4 Entwurf und Design

Eine typische Aufgabe dieser Entwurfsphase ist die Entscheidung über die Datenhaltung, die Verteilung im Netz und die Benutzeroberfläche des Software-Systems<sup>12</sup>. Jedoch standen diese Grundsatzentscheidungen schon zu Beginn des Praktikums, durch die Kenntnisse von Tobias Kallauke und dem Verfasser fest. Der Server verwendet zur Datenspeicherung PostgreSQL<sup>13</sup>, eine relationale Datenbank, sowie das Dateiverzeichnis für Bilder, während die App keine Daten persistent speichert. Durch die Aufteilung von App und Server handelt es sich um eine sogenannte Client/Server-Architektur. Für die Benutzeroberfläche der iOS-App wird, wie schon erwähnt, das deklarative SwiftUI verwendet. Weitere Informationen zum Backend und der Android-App befinden sich im Praktikumsbericht von Tobias Kallauke.

Jedoch mussten noch Arbeit in den Workflow und in die Architektur der App gesteckt werden. Zur Definition einzelner Teil-Workflows, die zusammenhängende Aufgaben umfassen, wurden eine Art Zustandsautomaten bzw. Flussdiagramm beschrieben. Ein Beispiel, was noch einmal geändert in der App verwendet wurde befindet sich im Anhang A. Die Modelle halfen dabei Views zu entwickeln und deren Design festzulegen. Das Aussehen der App sollten sich jedoch im Laufe der Zeit noch ändern, denn erst einmal stand die Umsetzung des Konzepts im Vordergrund.

Aus den Designs und den Teil-Workflows heraus entstand ein grober Plan, wie die Daten in der App gehandhabt werden sollten. Da *SwiftUI* das Entwurfsmuster MVVM umsetzt, benötigt es eine Struktur für das ViewModel und für den Datenfluss der asynchronen Server-Rückrufe. Jedoch stellte sich die Entwicklung dieser Strukturen, auch schon während des allerersten Prototyps, als problematisch heraus. Denn auch bei steigender Komplexität soll das ViewModel noch übersichtlich bleiben, um eine schnelle Weiterentwicklung zu gewährleisten. Aus diesen Gründen begann die Suche nach einer besseren Lösung, bei der die JavaScript-Bibliothek *Redux.js* zur Verwaltung von Zustandsinformationen in Webanwendung entdeckt wurde.

*Redux* hilft durch dessen Konzept, komplexe Views mit vielen Daten schnell und einfach zu implementieren. Auch werden so Serverantworten und zwischengespeicherte Daten sowie lokal erstellte Daten, die noch nicht auf dem Server gespeichert wurden, strukturiert und zentral abgelegt. Das erleichtert nicht nur das Wiederverwenden von Daten, sondern spart auch wiederholte Server-Aufrufe ein. Es erschien nun möglich, trotz der begrenzten Zeit im Praktikum, möglichst viel mit wenig Fehlern umzusetzen. Denn *Redux* ermöglicht durch die modulare Aufteilung des State Containers eine schnelle Weiterentwicklung der App, auch wenn die Komplexität der Anwendung steigt. Und die Vorteile von *Redux* hinsichtlich des Testens und Untersuchens der App sollten helfen,

<sup>12</sup> Vorlesung 9, Folie 27 aus Softwaretechnik Grundlagen von Prof. Dr.-Ing. Wilfried Schubert (2019) - [https://www.staff.hs-mittweida.de/~wschub/intranet/ss19/Fach\\_SWT/Fach\\_SWT\\_Zeitplan.htm](https://www.staff.hs-mittweida.de/~wschub/intranet/ss19/Fach_SWT/Fach_SWT_Zeitplan.htm)

<sup>13</sup> PostgreSQL Internetseite - <https://www.postgresql.org/>

Fehler möglichst schnell ausfindig zu machen, trotz asynchroner Programmabschnitte.

Daher lag es nah die wesentlichen Konzepte von *Redux* umzusetzen und einen *Redux* ähnlichen State Container, als ViewModel zu implementieren. Aus der Definition von Teil-Workflows sollte der Container oder auch Store genannt mindestens 5 States haben:

- für Authentifizierung sowie Registrierung,
- für das Anlegen von Vorlagen, um Zwischenergebnisse zu speichern,
- für das Ausführen von Server-Aufrufen zum Speichern und Abrufen von Vorlagen,
- für die Steuerung des Workflows bzw. den stellvertretenden Views sowie
- für sonstige Daten, die sehr häufig verwendet werden.

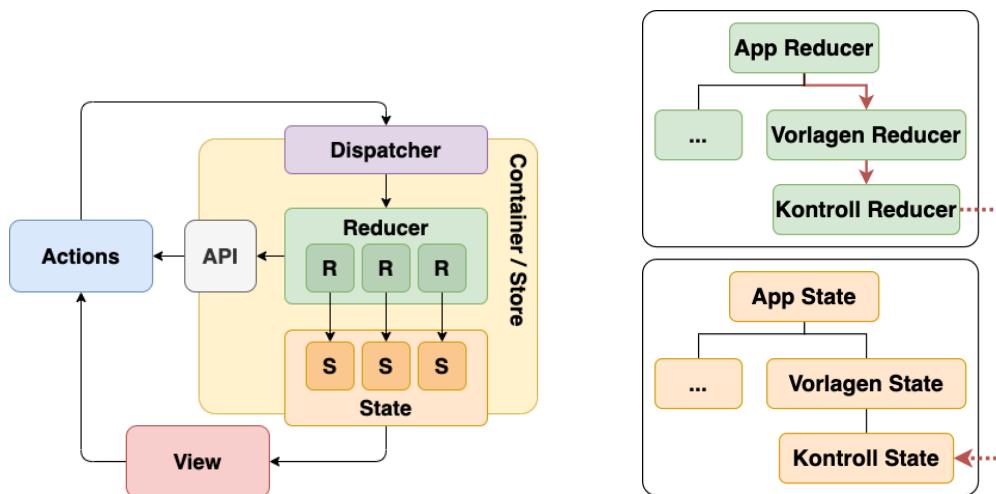


Abbildung 6.1: Architektur Schemata

Das Schema 6.1a zeigt, die in der App umgesetzte *Redux* ähnliche Struktur. Über eine View (rot) können Aktionen (blau) aufgerufen werden. Diese gelangt zuerst in einen sogenannten Dispatcher (lila), welcher als Verteiler dient. Anhand der Art einer Aktion, wird der entsprechende Reducer (grün) vom Dispatcher beauftragt, die Aktion auszuführen. Ein Reducer ist für genau eine Art von Aktionen zuständig. Allerdings ist, wie in der Abbildung 6.1b zu sehen ist, eine Schachtelung von Reducern (grün) möglich. Dies bedeutet auch, dass Aktionen-Arten ineinander geschachtelt werden können. Das hat den Hintergrund, dass die States wie ein Objektbaum aufgebaut sind. So hat jeder State seinen eigenen Reducer, was für die oben erwähnte Modularität sorgt. Möchte man beispielsweise eine Änderung im Kontroll-Mechanismen State (siehe 6.1b) vornehmen, muss die eigentliche Aktion in einer Kontroll-Mechanismen Reducer Aktion gekapselt werden und diese wiederum in einer Vorlagen Reducer Aktion. Zum ausführen der Aktion, werden dann die entsprechenden Reducer die Kapselung von außen nach innen

auflösen. Nachdem die Aktion ausgeführt und eine State-Änderung herbeigeführt wurde, aktualisiert sich durch das data binding von MVVM die View. Genauer werden alle Views, die eine Bindung zu dem Datum haben, über die Änderung benachrichtigt und daraufhin aktualisieren diese sich.

Ein weiterer wichtiger Punkt sind die Server-Aufrufe. Für diese gibt es einen eigenen Reducer und eine besondere Schnittstelle, welche in der Abbildung 6.1a grau markiert und mit API beschriftet ist. Diese hat die Besonderheit selbst Aktionen an den Container zu senden. Beispielsweise löst ein Knopfdruck in einer View die Aktion aus, alle Vorlagen vom Server zu laden. Diese Aktion gelangt über den vorgesehenen Reducer zu der API-Schnittstelle. Dort wird ein entsprechender Server-Aufruf gestartet und ohne andere Prozesse anzuhalten auf den Server-Rückruf gewartet. Sobald die Antwort des Servers angekommen ist, wird diese in einer Aktion zurück zum Container gesendet. Dort wird sich wieder ein entsprechender Reducer um eventuelle Fehlerbehandlung oder das Ablegen der Vorlagen kümmern.

## 6.5 Implementierung

Zur Realisierung der entworfenen Systemkomponenten wurde ausschließlich, die von Apple entwickelte IDE Xcode<sup>14</sup>, verwendet. Diese stellt Geräte-Simulatoren zur Verfügung, auf denen die App getestet werden kann. Die Simulatoren bieten unter anderem auch die Möglichkeit an, die Anwendung schnell auf verschiedenen iOS bzw. iPadOS Versionen zu testen. Auch ist das Testen der App auf echten Geräten durch Xcode möglich, was bei der Entwicklung unumgänglich war. Grund dafür ist die Benutzung der Kamera, die bei den Simulatoren zu gewollten Abstürzen führt, da diese keinen Zugriff auf eine Kamerasystem besitzen.

Die App wurde ausschließlich in der Programmiersprache Swift geschrieben. Zusätzlich wurden die Frameworks SwiftUI, Vision und VisionKit von Apple, sowie das Framework Kingfisher<sup>15</sup>, zum Downloaden und Cachen von Bildern verwendet. Das Vision-Framework führt Erkennung von Gesichts- und Gesichtsmarkierungen, Texterkennung, Barcode-Erkennung, Bildregistrierung und allgemeine Merkmalsverfolgung durch. Jedoch wurde in der App nur die integrierte Texterkennung benutzt. Es ist aber anzunehmen, dass Algorithmen aus VisionKit von Vision zur Dokumenten-Erkennung verwendet. Für mehr Informationen über die Frameworks SwiftUI und VisionKit siehe im Kapitel 6 Abschnitt 6.2.

Zum Ändern und Testen des Backends wurde außerdem Visual Studio Community bzw. Visual Studio Code<sup>16</sup> mit dem REST Client Plugin<sup>17</sup>, zum Testen der Server-

---

<sup>14</sup> Xcode Internetseite - <https://developer.apple.com/xcode/>

<sup>15</sup> Kingfisher GitHub Repository - <https://github.com/onevcat/Kingfisher>

<sup>16</sup> Visual Studio Internetseite - <https://visualstudio.microsoft.com/de/>

<sup>17</sup> REST Client - <https://marketplace.visualstudio.com/items?itemName=humao>.

Schnittstellen verwendet. Das Backend konnte dann als lokaler Server auf dem Entwicklungscomputer gestartet und über die IP-Adresse 0.0.0.0 sogar im lokalen Netzwerk aufgerufen werden. Für die Verwaltung der PostgreSQL-Datenbank genügte das Tool PgAdmin<sup>18</sup>. Damit ist es möglich einzelne Einträge oder auch ganze Tabellen der Datenbank zu bearbeiten.

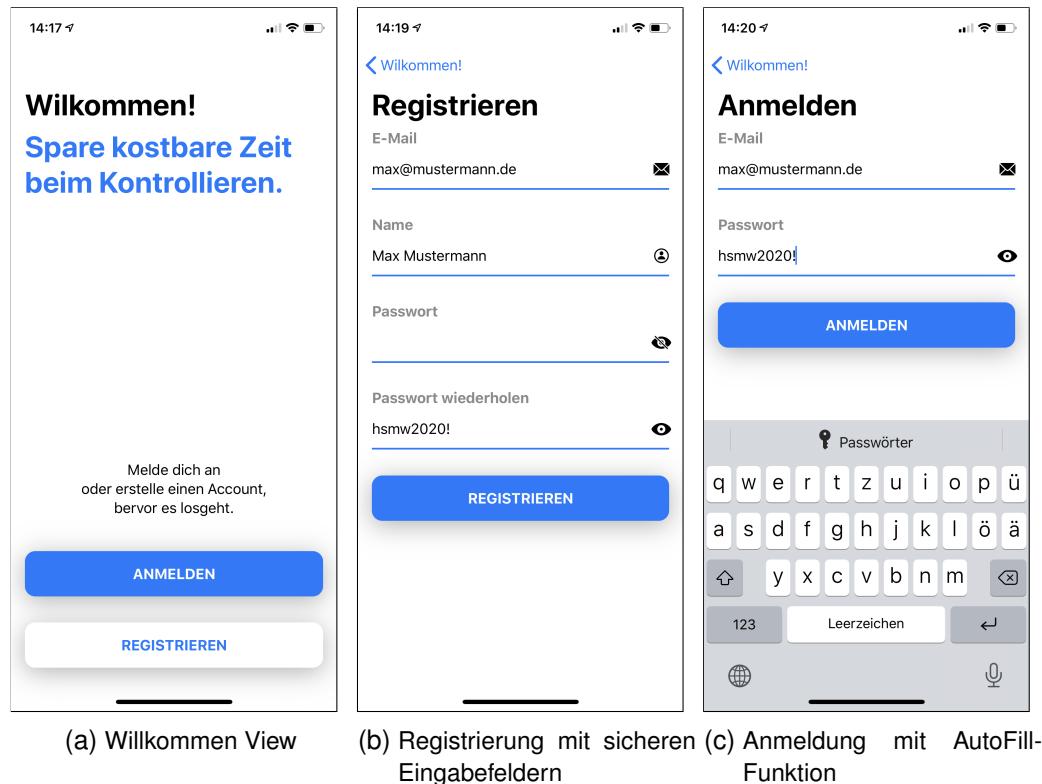


Abbildung 6.2: Willkommen- Registrier- und Anmelde-View

### 6.5.1 Registrieren und Anmelden

Die technische Implementierung der Registrierung und Anmeldung beinhaltet typische Charakteristiken von Anmelde- und Registrier-Formularen. Die E-Mail-Textfelder besitzen, wie auch das Passwort-Feld der Anmeldung eine AutoFill-Funktion. Dadurch kann das iOS-Gerät Registrier- und Anmelde-Daten vorschlagen und automatisch in die entsprechenden Felder einsetzen. Dies sieht man im dritten Bild 6.2c anhand des "Passwörter"-Knopfes über der Tastatur. Weiter sind alle Passwort-Felder gesichert. Das bedeutet, dass der Inhalt zum Schutz der Privatsphäre standardmäßig ausgeblendet und mit Punkten ersetzt wird. Durch das drücken des Auges rechts des Textfeldes kann der Inhalt jedoch eingesehen werden. Im mittlerem Bild 6.2b sind wegen der Sicherheitsstandards von iOS, bei einer Aufnahme des Bildschirms nicht einmal die Ersatz-Punkte des ersten Passwort-Textfelds zu sehen. Zusätzlich validieren alle Text-Felder

---

rest-client

<sup>18</sup> PgAdmin Internetseite - <https://www.pgadmin.org/>

ihren Inhalt mithilfe von regulären Ausdrücken. Beispielsweise muss ein Passwort aus 8 Zeichen bestehen und zwei von den drei folgenden Eigenschaften erfüllen:

1. Es ist mindestens ein Sonderzeichen enthalten.
2. Es ist mindestens ein Großbuchstabe enthalten.
3. Es ist mindestens eine Zahl enthalten.

Die Validierung ist jedoch nicht standardmäßig, sondern wurde selbst entwickelt.

### 6.5.2 Scan-Vorlagen erstellen und speichern

Bei der Implementierung des ersten Arbeitsschritts der Scan-Vorlagen wurde das schriftliche Flussdiagramm, welches im Anhang A zu finden ist, leicht verändert benutzt.

#### Scan-Vorlage erstellen

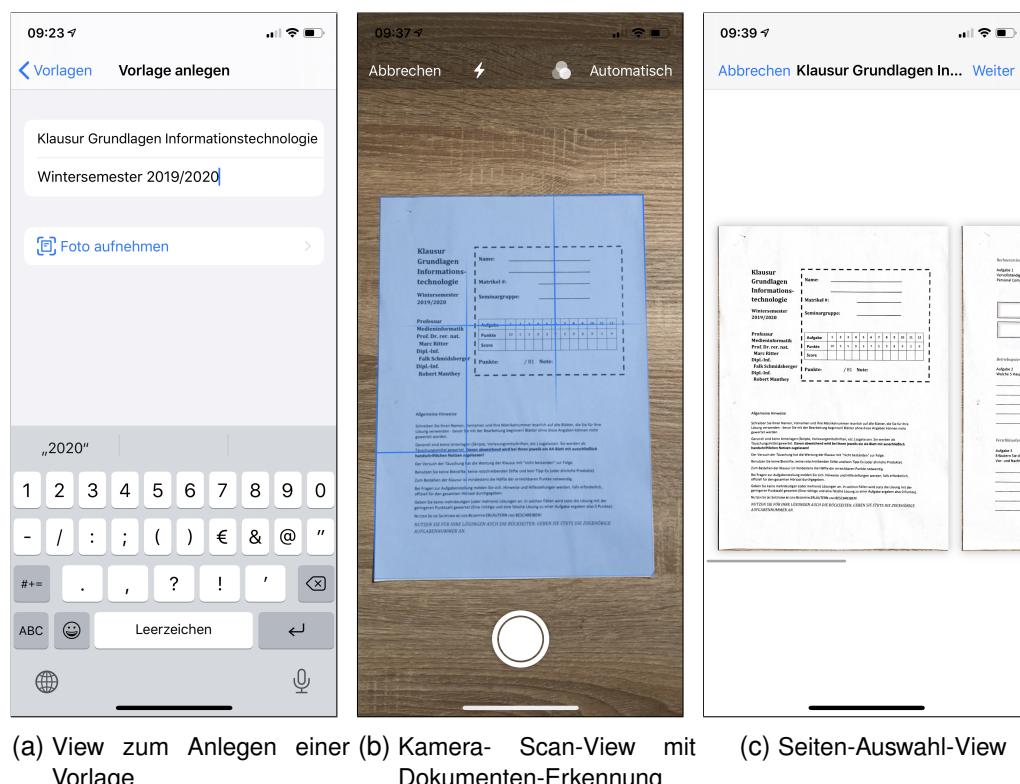
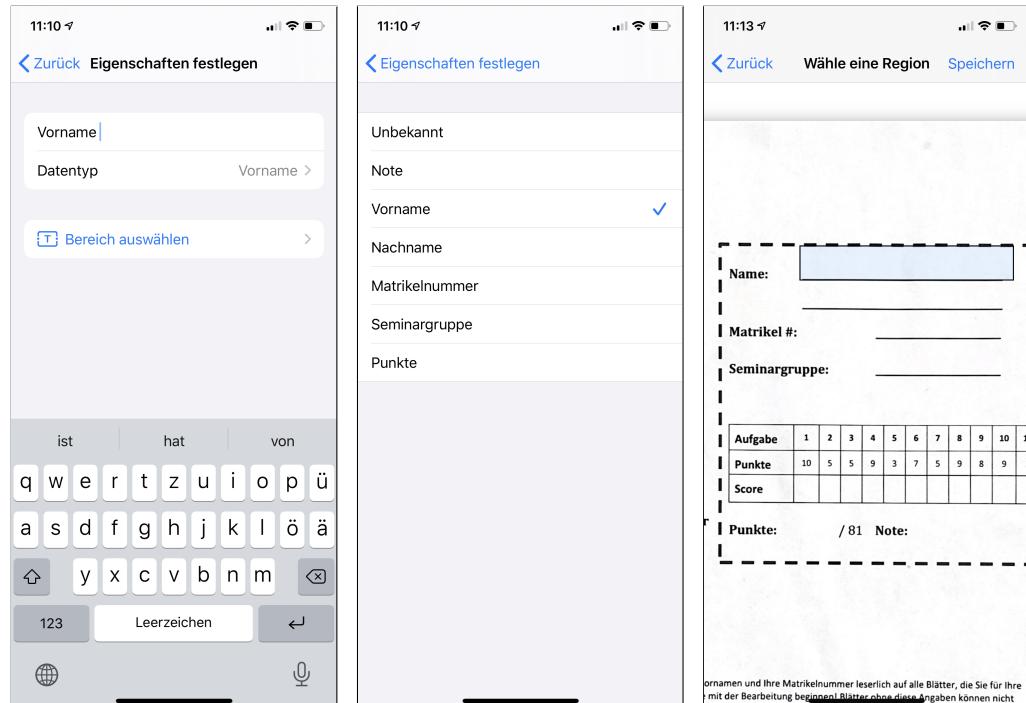


Abbildung 6.3: Die ersten Views zur Erstellung einer Scan-Vorlage

Im ersten Schritt sind ein Name und weitere Informationen, zu der Vorlage anzugeben (siehe 6.3a), um im Anschluss die Fotos aufzunehmen. Bei der Kamera-View (siehe 6.3b) handelt es sich um die Scan-View des Frameworks VisionKit. Mithilfe von Kantenerkennung und anderen Algorithmen, die Tobias Kallauke in seinem Bericht beschreibt, kann das Dokument sobald es erkannt ist, automatisch fotografiert werden.

In Echtzeit wird das erkannte Dokument aus dem Bild ausgeschnitten und gerade gezogen. Ein manuelles Auslösen des Fotos und Anpassen der Dokumenten-Kanten im Bild, ist ebenfalls möglich. Des Weiteren werden alle erstellten Bilder zu einer Gruppe gesammelt. Diese können vor dem Abspeichern angeschaut und nochmal bearbeitet werden. Die Abbildung 3.1 entstand durch diesen Prozess.

## Regionen erstellen



(a) View zum Festlegen von den Regionen Eigenschaften      (b) Auswahl des Datentyps      (c) Markierung der Region auf dem Dokument

Abbildung 6.4: Views zur Erstellung von Regionen

Im nächsten Schritt sind die Regionen auf den Dokumenten-Seiten zu markieren, deren Inhalt beim Einscannen digitalisiert werden soll. Zu erst wird die gewünschte Seite aus einer Übersicht (siehe 6.3c) ausgewählt. Anschließend ist eine Vorschau der Seite mit alle eingetragenen Regionen zu sehen (siehe 6.5a bzw. 6.5b). Über einen Button können weitere Regionen hinzugefügt werden. Dazu legt man einen Namen (siehe 6.4a) und einen Datentyp (siehe 6.4b) fest. Die Datentypen sind für die Texterkennung und Erstellung der Tabelle für die Notenfreigabe wichtig. Dazu später noch mehr. Wenn die Eigenschaften der Region festgelegt sind, muss diese noch auf dem Bild markiert werden. Dazu zieht man mit einem Finger ein Rechteck in einer beliebigen Größe ein (siehe 6.4c). Die markierte Region kann anschließend noch bewegt oder neu gemacht werden. Um kleinere Regionen präzise zu markieren kann mit einer Zwei-Finger-Geste auch an die Seite heran bzw. auch heraus gezoomt werden. Dieses Vorgehen muss dann für alle nötigen Regionen auf den jeweiligen Seiten wiederholt werden.

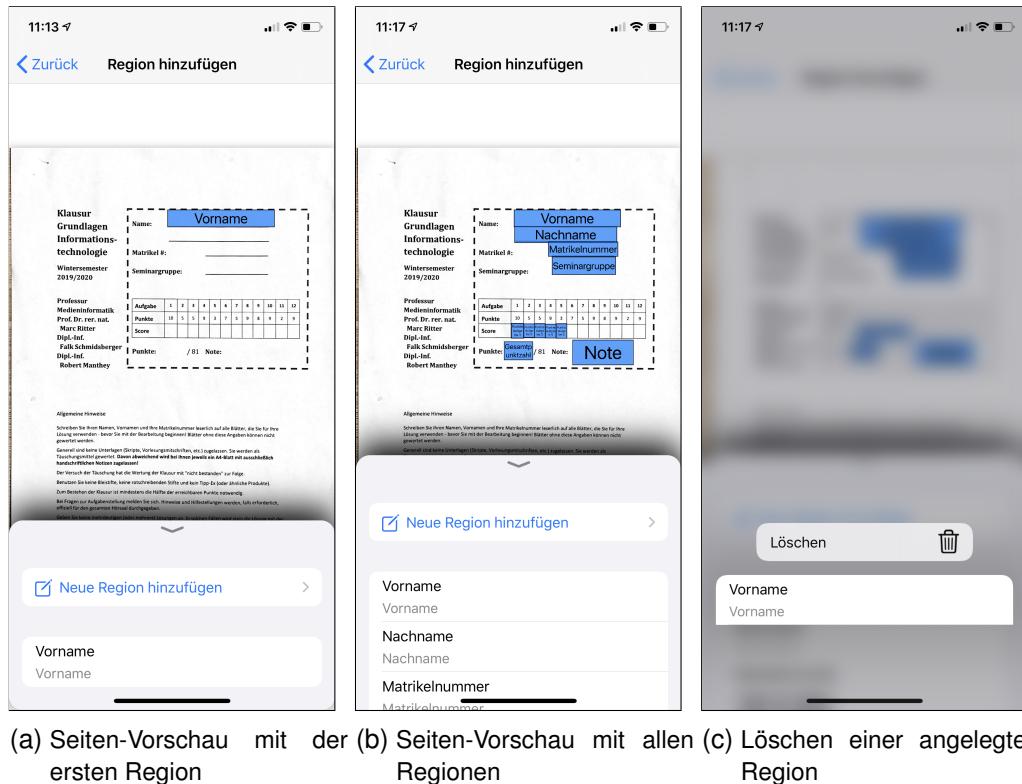


Abbildung 6.5: Seiten-Vorschau-View

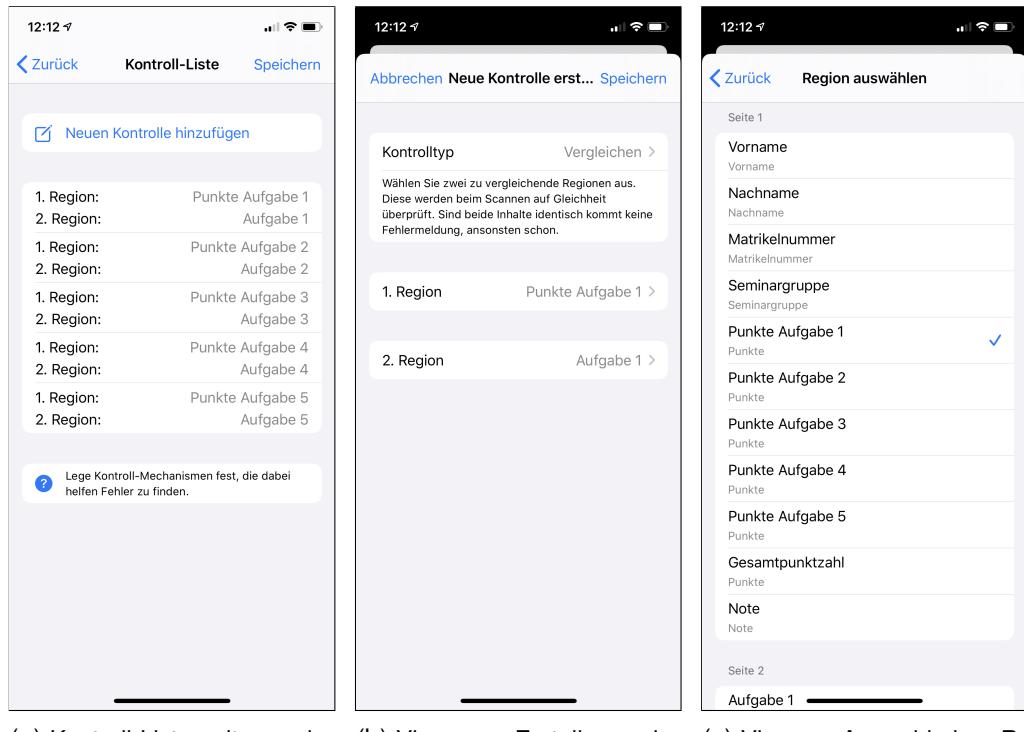
## Kontrollmechanismus erstellen

Zum Schluss können noch die sogenannte Kontrollmechanismen erstellt werden. Diese Funktion ist allerdings noch nicht sehr weit fortgeschritten und beinhaltet aktuell nur den Kontrollmechanismen-Typ zum Vergleichen von Regionen, wie im Kapitel 5 beschrieben ist. Beim Erstellen einer Kontrolle dieses Typs wählt man zwei Regionen aus (siehe Abbildung 6.6b), deren Inhalt nach der Texterkennung auf Gleichheit überprüft wird. Zur Auswahl stehen alle vorher angelegten Regionen, wie in Abbildung 6.6c zu sehen ist. In Abbildung 6.6a ist eine Übersicht aller angelegten Kontrollen zu der Scan-Vorlage zu sehen sowie ein Button zum Speichern der Vorlage.

## Scan-Vorlage speichern

Wenn eine Vorlage gespeichert werden soll, passiert das in mehreren Schritten. Dazu wird die entsprechenden Server-Schnittstellen aufgerufen und auf die Server-Rückrufe gewartet. Für genauere Details zur API des Servers siehe im Praktikumsbericht von Tobias Kallauke.

Im folgenden Abschnitt, ist beschrieben, in welcher Reihenfolge das Hochladen einer Vorlage zum Server geschieht. Dabei werden mögliche Fehler von Seiten des Server, der Internetverbindung und des Clients ignoriert. Jedoch ist im aktuellen Stand der App



(a) Kontroll-Liste mit angelegten Kontrollmechanismen (b) View zur Erstellung einer Kontrolle (c) View zur Auswahl einer Region

Abbildung 6.6: Views zur Erstellung von Kontrollmechanismen

das Abfangen des Fehler schon integriert, die Fehlerbehandlung aber noch nicht.

1. Der Name und die Beschreibung der Vorlage sowie die Liste an Kontrollmechanismen wird gesendet. In der Antwort des Servers befindet sich die Vorlagen-ID , die in den nächsten Schritten benötigt wird.
2. Die Bilder der Seiten werden gesendet. Als Antwort zu jedem Bild wird der Pfad zurück geschickt, wo das Bild vom Server gespeichert wurde. Dieser Pfad wird im nächsten Schritt benötigt.
3. Die Seiten mit einer Seiten-Nummer und dem Pfad zu dem Bild, wird mit der Vorlagen-ID gesendet. Als Antwort wird eine Seiten-ID zurückgegeben, die im nächsten Schritt verwendet wird.
4. Die Regionen der Seiten werden gesendet. Eine Region hat X- und Y-Koordinaten. Diese repräsentieren den Abstand vom Bildursprung, der sich in jedem Bild oben links befindet. Außerdem besitzt eine Region noch eine Höhe und Breite sowie einen Namen und einen Datentyp (6.4a und 6.4c). Die Koordinaten, sowie die Höhe und Breite sind alle in Pixel angegeben. Zusammen mit der Seiten-ID wird das Attribut an den Server gesendet.

### 6.5.3 Scan-Vorlage verwenden

Um eine Klausur zu digitalisieren, muss die vorher erstellte Scan-Vorlage ausgewählt werden (6.7a). Danach bietet die Benutzeroberfläche über einen Button die Möglichkeit an, eine Klausur ein zu scannen (6.7b). Bevor die Bilder aufgenommen werden können, erscheint ein Dialogfenster (6.8a), in dem der Benutzer sich entscheiden muss, welche OCR-Engine benutzt werden soll. Aktuell gibt es zwei Möglichkeiten.

- Die OCR-Engine des Vision Frameworks von Apple, welche direkt auf dem Gerät und ohne eine Internetverbindung die Texterkennung durchführt,
- oder die OCR-Engine Tesseract, die über eine Schnittstelle des Servers aufzurufen ist.

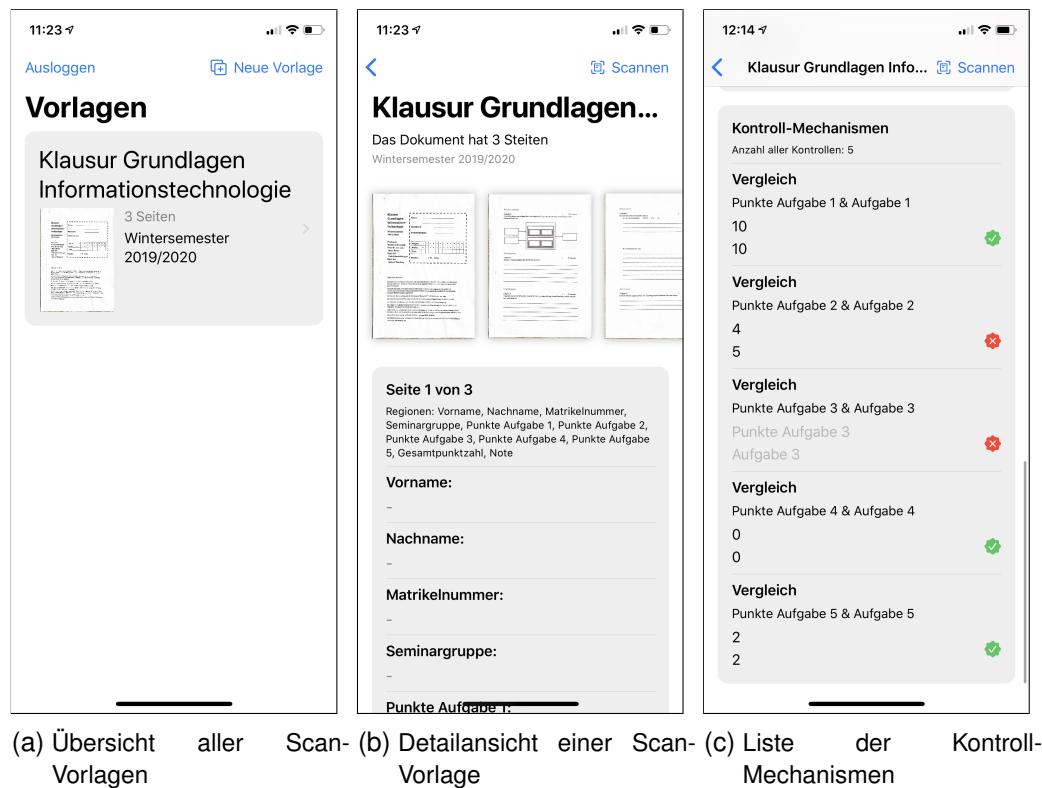


Abbildung 6.7: Listen- und Detail-Ansicht von Scan-Vorlagen

Nach der Auswahl öffnet sich die Kamera und die Bilder können aufgenommen werden. Wichtig dabei ist, dass die Seiten beim Fotografiert die selbe Reihenfolge, wie in der Scan-Vorlage haben. Auch muss die Anzahl der eingescannten Seiten mit denen in der Vorlage übereinstimmen. Bei der Implementierung der Scan-View wurde auf die Selbe zurückgegriffen, wie sie auch beim Vorlagen Erstellen zu finden ist (siehe 6.3b). Grundsätzlich konnten viele Komponenten der gesamten Benutzeroberfläche dank SwiftUI immer wieder verwendet werden. Durch die Scan-View wird die Seiten aus dem Bild ausgeschnitten, geglättet und der Kontrast des Bildes wird ebenfalls leicht angepasst, damit Buchstaben leichter zu erkennen sind und Schatten bzw. Falten verschwinden.

Nach dem Einscannen der Klausur-Seiten beginnt der Prozess der Texterkennung. Durch Aktivitätsanzeigen wird dem Benutzer mitgeteilt, dass die Anwendung gerade beschäftigt ist. Die Abläufe bei der online OCR-Engine sind jedoch leicht anders, als die, des Vision Frameworks.

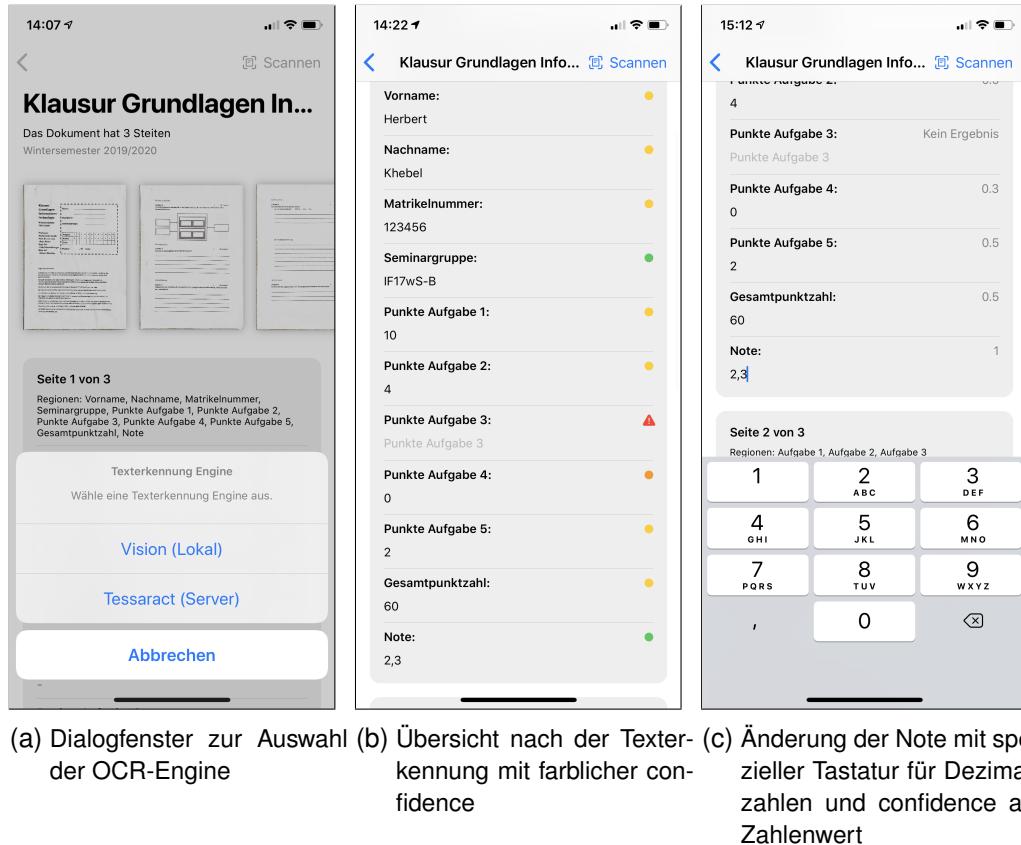


Abbildung 6.8: Vorlagen-Ansicht vor und nach der Texterkennung

## Ablauf mit Vision

Bei der Verwendung von Vision gibt es folgende grobe Schritte:

1. Die markierten Regionen in der Vorlage auf die neuen Bilder anwenden
2. Mit den Regionen jedes Bilds Bildausschnitten erstellen
3. Die Ausschnitte in die OCR-Engine geben
4. Texterkennungs-Ergebnisse darstellen

**Zu 1.: Die markierten Regionen in der Vorlage auf die neuen Bilder anwenden**  
 Damit ist gemeint, dass nun das erste Bild des neuen Scans mit den Regionen der ersten Seite der Vorlage verarbeitet wird. Zur Erinnerung aus Kapitel 5: Das Ziel ist es, die markierten Regionen in der Vorlage auf die zu digitalisierende Seite anzuwenden und

minimale Bildausschnitte zu erstellen, in denen der zu digitalisierende Text steht (siehe Abbildung 5.4). Diese Ausschnitte werden dann von der Texterkennung verarbeitet. Diesen Vorgang wird für jede Seite wiederholt. Jedoch ergibt sich hierbei ein Problem.

Wie man in den Abbildungen 3.1a und 3.1b sieht, haben beide, durch die App entstandenen Bilder, unterschiedliche Maße. Die Abbildung 3.1a ist größer und etwas breiter als die Abbildung 3.1b. Das liegt daran, dass sich der Winkel und der Abstand der Kamera zum Dokument bei den Aufnahme geändert hat, da das Gerät, während des Fotografierens in den Händen gehalten wurde. Das bedeutet also, wenn die Bilder in der Vorlage nicht genau so groß sind, wie die Bilder des neuen Scans, können im schlechtesten Fall wichtige Informationen durch die absoluten Positionen der Regionen abgeschnitten werden. Würde man immer den selben Abstand und Winkel garantieren, wie das beispielsweise in einem richtigen Kopiergerät oder Scanner der Fall ist, könnte man nun die Berechnung der absoluten Positionen in relative Positionen der Regionen weglassen. Zusätzlich sind die relative Höhe und Breite der Regionen ebenfalls zu berechnen.

**Zu 2.: Mit den Regionen jedes Bilds Bildausschnitten erstellen** Nachdem die neue Position und Maße aller Regionen einer Seite bestimmt sind, entstehen daraus Bildausschnitte für die Texterkennung. Zur Erinnerung, die Regionen dienen als eine Art Schablone, so dass alles, was sich außerhalb dieser befindet, nicht verwendet wird (siehe Abbildung 5.1). Auch trotz der Umrechnung in relative Positionen kann es bei falscher Erstellung der Vorlagen oder bei falschem Einscannen der Klausur passieren, dass trotzdem Teile der Regionen wegfallen. Für mehr Details über falsches Erstellen oder Einscannen siehe im Unterabschnitt 7.2.1.

**Zu 3.: Die Ausschnitte in die OCR-Engine geben** Zu jedem Bildausschnitt wird eine sogenannte Texterkennungs-Anfrage konfiguriert. Auch hierfür wird immer noch die zugehörige Region des Ausschnitts benötigt. In der Konfiguration werden an Hand des Region-Datentyps (6.4b) unterschiedliche Einstellungen getroffen. Beispielsweise bekommt die Anfrage einer Region, vom Typ Note, eine Liste an möglichen Noten. Die Liste hat, wie schon im Kapitel 5 erwähnt, Vorrang vor dem Standard-Wörterbuch, in der Wörterkennungsphase und sorgt für bessere Ergebnisse<sup>19</sup>. Auch könnte hier die Sprache des zu erkennenden Texts angegeben oder die Texterkennungs-Stufe eingestellt werden. Diese bestimmt, welche Techniken bei der Texterkennung verwendet werden. Entweder man gibt der Geschwindigkeit Vorrang vor der Genauigkeit oder aber man setzt auf eine längere, rechenintensivere Erkennung. In der Anwendung ist die OCR-Genauigkeit entscheidend und auf die Einstellung der Sprache wurde verzichtet, da es sich in keinem Fall um normalen Text handelt, sondern um Eigennamen, Abkürzungen und Ziffern.

<sup>19</sup> Vision *costumWords* Dokumentation <https://developer.apple.com/documentation/vision/vnrecognizetextrequest/3152640-customwords>

Standardmäßig werden bei einer Texterkennungs-Anfrage zunächst alle Glyphen oder Zeichen im Eingabebild lokalisiert und dann jede Zeichenfolge analysiert<sup>20</sup>. Anschließend wird eine Korrektur an den erkannter Wörter auf der Grundlage eines Wörterbuchs und anderer Wahrscheinlichkeits-Heuristiken für Zeichenpaare durchgeführt [2]. Der genaue Ablauf der Texterkennung oder die verwendeten Algorithmen von Vision sind allerdings nicht bekannt.

Anschließend an die Texterkennung erfolgt in einigen Fällen noch eine selbst entwickelte Korrektur, wie sie in Kapitel 5 Abschnitt 5.3 beschreiben ist. Auch hier spielt der Regionen-Datentyp (6.4b) wieder eine Rolle. Beispielsweise wird der Text einer vermeidlichen Seminargruppe mit allen in der App hinterlegten Seminargruppen verglichen. Der Seminargruppen-Name mit der größten Übereinstimmung wird dann angezeigt. Die confidence wird ebenfalls angepasst. Zur Erinnerung, diese sagt aus, wie sicher sich die Texterkennung ist, dass das Ergebnis stimmt. Ähnliche Korrekturen gibt es für die Note und Punkte.

Danach werden die erkannten Texte bzw. Worte und deren confidences an den State-Container der App gesendet. Für die Ergebnisse der Texterkennung ist ein eigener State vorgesehen, in dem auch die Server-Ergebnisse von Tesseract gesichert werden. Somit ist das Anzeigen der Ergebnisse bei beiden OCR-Engines gleich.

**Zu 4.: Texterkennungs-Ergebnisse darstellen** Die Ergebnissen der Texterkennung und die jeweilige confidence werden auf der Benutzeroberfläche, nach Seiten sortiert, angezeigt (6.8b). Alle OCR-Ergebnisse können zudem noch angepasst werden. Abhängig vom Datentyp der Region passt sich die Tastatur beim Ändern der Texte an (6.8c). Beispielsweise sind die Tastaturen beim Anpassen von Noten und Punkten auf Zahlen und Dezimalpunkte spezialisiert. Zudem kann die confidence farblich (siehe Abbildung 6.8b) aber auch als Zahlenwert (siehe Abbildung 6.8c) angezeigt werden. Wurde kein Text erkannt, ist dementsprechend ein Warnsymbol an der Stelle der confidence zu sehen.

## Ablauf mit Tesseract

Etwas anders ist der Ablauf, wenn die Texterkennung Tesseract auf dem Server verwendet wird:

1. Bilder an den Server senden
2. Auftrag zur Texterkennung an den Server senden
3. Serverseitige Texterkennung

---

<sup>20</sup> Quelle: [VNRecognizeTextRequest](https://developer.apple.com/documentation/vision/vnrecognizetextrequest) Dokumentation <https://developer.apple.com/documentation/vision/vnrecognizetextrequest>

#### 4. Texterkennungs-Ergebnisse darstellen

**Zu 1.: Bilder an den Server senden** Nachdem das Dokument fotografiert wurden, werden die Aufnahmen an den Server gesendet. Dieser gibt, wie beim Speichern der Scan-Vorlage, die Pfade der Bilder als Antwort zurück.

**Zu 2.: Auftrag zur Texterkennung an den Server senden** Anschließend wird für jeden Pfad eine Auftrag zur Texterkennung an die OCR-Schnittstelle des Servers gestellt. In der Anfrage wird der Pfad des Bildes und die passende Seiten-ID der ausgewählten Vorlage gesendet. Ab hier übernimmt der Server die Arbeit der Texterkennung.

**Zu 3.: Serverseitige Texterkennung** Mithilfe der Seiten-ID, die in der Datenbank des Servers hinterlegt ist, werden dann die Regionen der Seite auf das Bild, welches sich an dem gesendet Pfad befindet, angewendet. Für mehr Details siehe im Praktikumsbericht von Tobias Kallauke.

**Zu 4.: Texterkennungs-Ergebnisse darstellen** Nach der Texterkennung sendet der Server die Ergebnisse der Seiten als Antwort zurück. Diese werden dann im selben State des State-Containers gesichert, wie es auch bei Vision der Fall ist. Die Ergebnisse bestehen aus der Region-ID, zur Zuordnung, dem erkannten Text, sowie der confidence des OCR, zu dem jeweiligem Wort. Auch ähnlich, wie es bei Vision der Fall ist. Die Darstellung der Ergebnisse und das Verhalten der Benutzeroberfläche ist ebenfalls identisch.

## 7 Fazit

Dieses Kapitel reflektiert den aktuellen Stand der App und zieht einen Vergleich mit den Anforderungen aus Kapitel 4.

### 7.1 Stand des Prototyps

Während der Entwicklung konnten viele Anforderungen umgesetzt werden. Jedoch gibt es aktuell keine Möglichkeit die Ergebnisse der Texterkennung an den Server zu senden und somit ist es auch nicht möglich die Daten in ein tabellarisches Format für die Notenfreigabe zu überführen. Dieser essentieller Bestandteil des Software-Systems benötigt mehr Zeit als angenommen und konnte deshalb nicht umgesetzt werden. Außerdem wurde nur einer der Kontrollmechanismen umgesetzt. Grund dafür ist, dass dafür zu wenig Zeit in der Planung investiert wurde. Grundsätzlich könnten die fehlenden Mechanismen recht schnell implementiert werden, allerdings ist der aktuelle Ansatz der dahinter liegenden Datenstruktur nicht zum Speichern in der Datenbank geeignet. Sobald Änderungen an der Datenstruktur vorgenommen werden könnten, kann es zu Fehlern kommen. Es existiert jedoch schon eine Idee zur Umsetzung einer besseren Datenstruktur. Dafür müssten allerdings neben der App auch der Server und die Datenbank angepasst werden. Des Weiteren wurden Fehlermeldungen und/oder -behandlungen bezüglich des Servers nur sporadisch implementiert. Aber auch hierfür sind die Grundlagen schon gelegt, da Fehler stets aufgefangen und zentral abgelegt werden.

### 7.2 Probleme und Grenzen der App

In diesem Abschnitt werden Probleme aufgeführt, die beim Verwenden der App beobachtet werden können. Auch wird hier über mögliche Lösungen zu einigen Problemen diskutiert.

#### 7.2.1 Probleme beim Erkennen von Dokumenten

Bei der Erkennung von Dokumenten in einem Bild wird in der Bildverarbeitung auf Algorithmen der Segmentierung zurück gegriffen. Im Praktikumsbericht von Tobias Kallauke wird ein gängiger Ablauf zum Erkennen von Dokumenten dargestellt. Die genaueren Hintergrund und Schwächen der Algorithmen, welche die grundlegenden Probleme und Grenzen der App begründen, würden den Rahmen des Praktikumsberichts sprengen, weshalb darauf nicht weiter eingegangen wird. Allerdings muss klar sein, dass für die Erkennung des Dokuments es wichtig ist, dass die Ecken sowie die Seiten sich gut vom

Hintergrund abheben.

Das bedeutet im Umkehrschluss, dass wenn die Seiten und Ecken sich nicht gut abheben, kann das Dokument nicht oder nur schlecht erkannt werden. Allerdings gibt es für diese Probleme zwei effiziente Lösungen.

- Die erste Lösung besteht darin, einen Blitz zu verwenden. Meistens schaltet sich bei zu wenig Licht die zusätzliche Belichtung automatisch ein. Grundsätzlich empfiehlt es sich aber, immer eine Blitz zu verwenden, da dadurch das Dokument gleichmäßig belichtet wird. Außerdem werden Schatten vom Benutzer oder durch kleinere Falten beseitigt, welche durch eine Deckenbeleuchtung entstehen könnten. Bei einer zu starken Grundbelichtung verschlechtert ein Blitz jedoch nur das Ergebnis, da dadurch Text und Kanten unkenntlich gemacht werden könnten. Nach eigener Erfahrung ist es am geeignetsten, bei Tageslicht und mit Blitz die Dokumente aufzunehmen.
- Auch wenn bei guter Belichtung, die Wahrscheinlichkeit für falsche Kantenerkennung reduziert worden ist, kommt es bei schnellen Bewegungen der Kamera oder zu spitzem Winkel zum Dokument sowie ungeeignetem Hintergrund trotzdem dazu, dass ein falscher Bildausschnitt als Dokument gewählt wird. Deshalb ist es möglich das Bild noch einmal aufzunehmen, oder aber das Dokument im Bild selbst zu markieren. Dafür stellt VisionKit eine besondere View zur Verfügung, die es einem ermöglicht, die Ecken des Dokuments per Hand auszuwählen. Das hilft auch gegen die Problematik, wenn die Ecken des Papiers umgekippt oder abgerissen sind, da man selbst die Position der Ecke approximiert angeben kann.

### 7.2.2 Probleme der Klausur-Vorlage beim Scannen

Dieser Abschnitt bezieht sich ausschließlich auf entstandene Probleme mit der Klausur, welche in Abbildung 3.1 zu sehen ist. Diese stand während des Praktikums als Beispiel-Klausur zur Verfügung und ist stellvertretend für alle Klausuren-Vorlagen der Fakultät CB. Im Abschnitt 7.3, werden mögliche Änderungen und Lösungen zu den hier aufgeführten Problemen diskutiert.

Wie in der Abbildung 3.1a zu sehen ist, besitzt die Klausur ein Deckblatt mit Feldern, die der Student auszufüllen hat. Darunter Name, Artikelnummer und Seminargruppe. Diese Informationen sind essentiell bei der Digitalisierung und doch wird nicht klar zwischen einem Feld für den Vornamen und einem Feld für den Nachnamen differenziert. In den Abbildungen 6.4 und 6.5 wird diese Problematik nochmal dargestellt. Bei den Datentypen der Regionen (6.4b) wird zwischen Vornamen und Nachnamen unterschieden, auf der Klausur (6.4c) jedoch nicht. In der abgebildeten Vorlage (6.5b) wurde der Vorname auf den ersten und der Nachname auf dem zweiten Strich des Feldes Namen gesetzt. Auch verleitet ein einzelner Strich dazu, über die Länge hinaus zu schreiben.

Wenn die Regionen der Scan-Vorlage dann zu klein gewählt ist, können Informationen fehlen. Bei dem Feld für die Note wird dies noch extremer. Da nicht mal ein Markierung für die Note vorhanden ist, müsste man einen sehr viel größeren Bereich markieren (6.5b). Allerdings wird hinter der Note auch die Unterschrift des Prüfers gesetzt, welche bei der Digitalisierung nicht mit auftauchen sollte. Im Gegensatz dazu sind die Felder in der Tabelle für die Punkte zu jeder Aufgabe zu klein. Die geringe Größe verleitet dazu, dass Feld komplett aus zu nutzen und die Ziffer möglichst groß rein zu schreiben. Dazu kommt dass solch kleinen Regionen kaum Spielraum für Toleranz lassen und somit entstehen hier immer wieder Fehler. Entweder die Ziffer ist abgeschnitten, wenn man die Region zu klein wählt oder aber die Ränder des Feldes werden als eine Eins oder als der Buchstabe I erkannt, wenn man die Region zu groß wählt.

### 7.2.3 Weitere Probleme der App

Obwohl SwiftUI die Entwicklung der App erst möglich gemacht hat, bringt das Framework von Juni 2019 einige Schwierigkeiten mit sich. Seit der Veröffentlichung gibt es nur eine sehr kleine Dokumentation und der Schnittstellen-Umfang, kann auch noch lange nicht mit dem des Frameworks von UIKit mithalten, welches seit Jahren in der iOS-Entwicklung als Standard benutzt wird. Grund für den geringen Umfang und schlechter Dokumentation ist vermutlich, die ständige Weiterentwicklung Seitens Apple. Das führt jedoch zu einigen provisorischen Lösungen und macht das Auffinden von Fehlern besonders schwierig. Deshalb kann es bei der Benutzung der App zu unerwarteten Abstürzen oder Aufhängern kommen. Des Weiteren sind die verwendeten Texterkennung-Engines nur für gedruckte Schrift ausgelegt. Handschriftliche Zeichen werden nur dann gut erkannt, wenn sie in Druckschrift geschrieben sind. Bei Ziffern ist es besonders auffällig, da z. B. eine gedruckte 4 oder 7 ganz andere Charakteristiken haben, als wenn sie geschrieben wurde.

## 7.3 Verbesserung der Klausuren-Vorlage

Dieser Abschnitt beschäftigt sich mit Verbesserungen und Änderungen an Hand der Klausur in Abbildung 3.1. Sie steht jedoch stellvertretend für die Klausuren-Vorlage der Fakultät CB. Alle folgenden Vorschläge haben den Hintergrund die App bei der Digitalisierung zu unterstützen und basieren auf der gesammelten Erfahrung mit der Anwendung, während der Entwicklung sowie den angegeben Problemen aus Abschnitt 7.2.

Der erste Vorschlag bezieht sich auf die Problematik, dass die Ecken und Kanten bei der Erkennung des Dokuments entscheidend sind. Sie sind die wichtigsten Referenz-Punkte bei der Objekt-Erkennung, können aber unter bestimmten Bedienungen, wie im Unterabschnitt 7.2.1 erklärt, nicht immer erkannt werden. Aus diesem Grund empfiehlt es sich eigene Referenzpunkt auf dem Dokument anzubringen. Beispielsweise durch

einen Rahmen, wie es schon auf der Klausur (3.1a) gemacht wurde, oder durch QR-Code ähnliche Muster. Der Vorteil bei eigenen Referenzpunkt ist, dass sie so gestaltet werden können, dass sie sich immer vom Blatt abheben. So mit ist die Dokumenten-Erkennung nur noch vom Licht abhängig. Ein weiterer Vorteil ist, dass dadurch die Kamera noch näher an den Text heran kann und die Auflösung noch besser wird. Der größte Nachteil daran ist allerdings, dass die Erkennung nicht mehr allein von Frameworks übernommen werden kann. Es müsste ein neuronales Netz trainiert werden, welche diese Referenzpunkte in Bildern erkennt.

Der zweite Vorschlag bezieht sich auf den Unterabschnitt 7.2.2. Einige dort angesprochenen Probleme bezogen sich auf die Ungewissheit, wie groß eine Region abzustecken ist. Die einfachste Lösung ist es, einen genauen Rahmen für alle Felder fest zu legen. Auch müssen alle Felder ausreichend beschriftet sein, so dass jedem klar ist, wo was hineinzutragen ist. Und um das Problem zu vermeiden, dass der Rahmen als ein Buchstabe oder eine Zahl erkannt wird, kann dieser nur schwach eingezogen werden. Mithilfe von Bildverarbeitung-Algorithmen könnten diese feinen Linien dann auch ohne Verlust der wichtigen Daten heraus gerechnet werden.

Abschließend ist zu erwähnen, dass laut der Problemstellung aus Kapitel 3 eine Klausuren-Vorlage entwickelt werden sollte. Auf Grund mangelnder Zeit war das jedoch nicht möglich.

## 8 Ausblick

Der aktuelle Stand der App bietet eine gute Grundlage für mögliche Weiterentwicklungen. Jedoch fehlen einige wichtige Elemente zur Inbetriebnahme des Software-Systems, wie schon im Kapitel 7 verdeutlicht. In den folgenden Abschnitten werden einige Ideen erläutert, wie die App und das gesamte Software-System weiter entwickelt und verbessert werden kann.

**Verbesserung der Benutzerfreundlichkeit und des Workflows** Gerade weil die App noch ein Prototyp ist, sollten bei einer Weiterentwicklung zu allererst Dinge implementiert werden, die die Benutzung verbessern. Beispielsweise werden Login-Daten und schon heruntergeladene Scan-Vorlagen nicht gespeichert. Auch muss zu jeder Region ein Name vergeben werden. Allerdings gibt auch der Datentyp (6.4b) der Region in den meisten Fällen darüber Auskunft, wie die Region heißen sollte (siehe dazu Abbildung 6.4a). Handelt es sich um den Vor- bzw. Nachnamen, die Matrikel-Nummer, Seminar-Gruppe oder die Note, taucht diese Region auch nur genau einmal in einer Scan-Vorlage auf. Ausschließlich der Datentyp Punkte sollte mehrmals vergeben werden können. Aus diesen Aspekten sollte der Workflow beim erstellen von Regionen angepasst werden.

**Weitere Dokument- und Datentypen** Auf dem Abschnitt *Verbesserung der Benutzerfreundlichkeit und des Workflows* aufbauend, ist folgende Idee entstanden. Durch die generischen Vorlagen ist es möglich neben Klausuren auch andere Dokumente nach diesem Schema zu digitalisieren. Beispielsweise Krankenscheinen, Urlaubsanträgen oder Arbeitsverträgen. Dafür könnten spezielle Workflows zum Erstellen einer Vorlage implementiert werden und weitere Regionen-Datentypen hinzukommen, wie z. B. Datum, IBAN-Nummern, E-Mail-Adressen und Telefonnummern.

**Server-Schnittstellen** Zum Zeitpunkt der Abgabe sind in der iOS-App nicht alle verfügbaren Server-Schnittstellen implementiert. Es fehlen noch das Ändern und Löschen ganzer Vorlagen und deren Einzelteile, wie Seiten und Regionen. Dafür müssten Views entwickelt werden, die das Ändern und Löschen umsetzen. Durch SwiftUI ist es aber möglich Views und View-Komponenten wiederzuverwenden, so dass nicht von vorne begonnen werden muss.

**Erweiterung der Texterkennung** Wie im Abschnitt 7.2 erwähnt unterstützten Vision und Tesseract das Erkennen von Handschrift nicht. Jedoch wäre es möglich eigne

Neuronale-Netze zu trainieren und implementieren, die diese Aufgabe übernehmen. Beispielsweise existiert ein Datensatz an handgeschriebene Ziffern<sup>21</sup>, mit dem solch ein einfaches Netz für das Erkennen von Ziffern, trainiert werden könnte. Außerdem bietet die Server Architektur die Möglichkeit, weitere oder mehr OCR-Engines zu implementieren. Für mehr Details siehe im Praktikumsbericht von Tobias Kallauke.

**Klausuren-Einsicht** Die archivierten Klausuren-Bilder könnten über ein Web-Portal dazu genutzt werden, eine online Einsicht der Prüfungen zu gestalten. Jedoch bringt solch eine Plattform auch ein Problem mit sich. Durch sie ist die Verbreitung von Klausuren noch einfacher als zuvor, wodurch die Professoren dazu angehalten werden, immer wieder neue Aufgaben auszudenken. Aber auch dafür könnte es in Zukunft eine Lösung geben.

**Weitere Plattformen** Neben der iOS-App sollte Tobias Kallauke die selbe für Android Geräte entwickeln. Jedoch existierten zum Zeitpunkt der Entwicklung keine Frameworks, die das Erkennen von Dokumenten in der Kamera übernimmt. Deshalb entwickelte Tobias ein Prototyp, der sich mit diesem Problem beschäftigt. Allerdings könnte das Software-System auf noch mehr Plattformen Anwendung finden. Neben einem Programm für Linux, MacOS und Windows ist auch einen Web-Plattform erdenklich. Alle könnten den selben Umfang besitzen nur das Erstellen der Scans erfolgt beispielsweise über Scan- oder Kopiergeräte.

**Intelligente Referenzpunkte** In Abschnitt 7.3 wurde erwähnt, dass eigene Referenzpunkte auf den Klausuren Vorteile mit sich bringen. Möglich wäre auch, dass die Referenzpunkte ähnlich wie QR-Codes aufgebaut sind. Diese enthalten die nötigen Informationen zu den Regionen auf der Seite und ermöglichen das Digitalisieren, ohne die Auswahl der richtigen Vorlage. Daran angelehnt könnten ein LATEX-Paket oder Word-PlugIn entwickelt werden, wodurch die QR-Codes automatisch für jede Seite und deren Regionen generiert werden.

**Klausuren-Vorlage entwickeln** Auf Abschnitt 7.3 und den *intelligenten Referenzpunkten* aufbauen könnten Klausuren-Vorlagen umgesetzte und getestet werden. Auch die App müsste angepasst werden, so dass die Referenzpunkte erkannt werden, wie im Abschnitt 7.3 beschrieben.

**Benutzer Akzeptanz Studie** Aktuell ist unklar, ob die App tatsächlich Zeit einspart. Aus diesem Grund bietet es sich an eine umfangreiche Benutzer Akzeptanz Studie an-

---

<sup>21</sup> MNIST-Datensatz - <http://yann.lecun.com/exdb/mnist/>

zusehen legen. Außerdem könnten so weitere Probleme entdeckt und Lösungen entwickelt werden, bevor die App benutzt werden kann.

## Anhang A: Workflow

### Vorlage erstellen

1. "Neue Vorlage erstellen"
2. Foto machen
3. Frage: Ist Foto gut?
  - a) Ja: gehe zu 4.
  - b) Nein: gehe zu 2.
4. Neues Attribut hinzufügen
5. Bereich auf Bild auswählen
6. Frage: Ist Bereich gut?
  - a) Ja: gehe zu 7.
  - b) Nein: gehe zu 5.
7. Name für Attribut festlegen
8. Datentyp für Attribut festlegen (Name, Matrikelnummer, Note, ...)
9. Frage: Sind alle Attribute vorhanden?
  - a) Ja: gehe zu 10.
  - b) Nein: gehe zu 4.
10. Fertig
11. Vorlage an Server senden

## Anhang B: Tätigkeitsbericht

**24.02. - 01.03.** Zunächst habe ich mich mit der Problemstellung auseinander gesetzt, Ideen gesammelt, Problemanalyse betrieben und einen kleinen Prototypen entwickelt. Dazu erstellte ich eine minimale Projektplanung, arbeitete mich in die Frameworks *Vision* und *VisionKit* ein und setzte eine Versionsverwaltung auf. Zusätzlich suchte ich nach einer passenden App-Architektur, die für das deklarative GUI-Framework SwiftUI sowie für asynchrone Aufgaben, wie z. B. API-Aufrufe geeignet ist. Dabei stieß ich auf *Cleancode Architecture* und *Redux*.

**02.03. - 08.03.** In dieser Woche habe ich die Texterkennung auf den berechneten Regionen eines neuen Fotos implementiert, den Workflow sowie viele andere Kleinigkeiten in der App verbessert und alle Fehler der letzten Woche behoben, sodass ich neue Dinge implementieren konnte. Zudem probierte ich CI sowie Lint für das Projekt aus. Da CI für eine iOS-App mit *Github Actions* schwer aufzusetzen war und ab April kostenpflichtig wurde, verwarf ich meine Pläne. Des Weiteren pflegte ich das Projekt Management durch *Issues* und *Project Boards* in GitHub. Anschließend programmierte ich den App-Workflow so um, dass nun mehr als eine Seite aufgenommen und analysiert werden konnte.

Abgesehen von neuen Quellcode begann ich mit dem Schreiben des Praktikumsberichts und arbeitete mich dazu in  $\text{\LaTeX}$  und die Bachelorarbeit-Vorlage für  $\text{\LaTeX}$  der Hochschule Mittweida ein.

**09.03. - 15.03.** Zu Beginn der dritten Woche schaute ich mir Möglichkeiten für serverseitiges OCR an. Dabei sammelte ich Informationen zu dem Framework Vapor und Swift unter Linux. Da die Frameworks *Vision* und *CoreML* von *Apple* unter Linux nicht funktionierten, stellte sich IronOCR als beste Option herausstellte. Mithilfe der in der App verwendeten Datentypen entwickelte ich ein Datenbankmodell und erstellte dazu noch eine JSON-Struktur, die später für die APIs verwendet werden könnte. Außerdem gab es ein Meeting, in dem Tobias Kallauke und ich unseren aktuellen Stand präsentierten, um weitere Schritte und Aufgaben zu planen. Bis zum Ende der Woche arbeitete ich fortlaufend an meinem Beleg und schrieb den Datenfluss in der App um. Nun ähnelte er sehr dem Redux-Model.

**16.03. - 22.03.** Anfangs schrieb ich meinen Praktikumsbericht weiter, bearbeitete alte Issues und fügte neue dem Project Board hinzu. Außerdem gepflegte ich die Dokumentation, um anschließend Kontrollmechanismen hinzuzufügen. Dabei entstanden neue Views. Der Redux-Store musste dadurch angepasst werden. Es kam eine Erweiterung für die Texterkennung hinzu, so dass man durch die Auswahl eines Datentyps, das Resultat der Erkennung verbessern konnte. Des Weiteren habe ich bis zum Ende der Woche den Kontroll-Typ *Vergleich* vollständig implementiert und die App auf Fehler und

Abstürze kontrolliert sowie den Beleg um einige Kapitel erweitert.

**23.03. - 29.03.** Ich begann den Workflow und die Navigation in der App zu verbessern und vereinfachen. Dabei beseitigte ich einigen Quellcode des Prototyps, erweiterte die Dokumentation und behob einige Fehler. Anschließend überarbeitete ich einige Views, sodass sie übersichtlicher und einfacher zu benutzen sind. Nach dem iOS 13.4 Update Mitte der Woche funktionierte ein Teil der App nicht, da sich das Verhalten von Views geändert hatte. Ich behob die Fehler, testete ausgiebig die App und fügte iPad Unterstützung hinzu. Des Weiteren entstand eine neue verbesserte permanente Scan-Vorlage und ich schrieb einen großen Teil des Berichts.

**30.03. - 05.04.** Das Backend für die App war soweit, dass ich es aufsetzen und die API-Schnittstellen implementieren konnte. Dazu erstellte ich Views für Registrieren und Anmelden, die mithilfe von regulären Ausdrücken, die Eingaben überprüfen. Außerdem entwickelte ich für die anfallenden asynchronen Aufgaben einen Schicht im App-Store. Dabei las ich mich in das Framework Combine ein und überlegte mir einen geeigneten Aufbau. Da der Ansatz von Combine sehr neu für mich war, dauerte es zwei Tage, bis ein erster API-Service mit Fehler-Handling funktionierte. Zum Ende der Woche waren alle der Create-Schnittstellen implementiert, getestet und dokumentiert. Nebenbei erstellte ein paar Issues für das Backend und sprach mich mit Tobias über OCR auf dem Server ab.

**06.04. - 12.04.** Diese Woche startete mit dem Umschreiben der Kontroll-Mechanismen und deren Analyse. Anschließend integrierte ich die Neuerungen vom Backend und erstellte die API für den Upload von Bildern. Danach fügte ich die APIs zusammen, um Vorlagen vollständig auf dem Server zu speichern und ab zu rufen. Dazu schrieb ich eine eigene JSON-Decoder-Funktion, um die App internen Datentypen zu unterstützen. Zusätzlich wurde die App etwas benutzerfreundlicher und ein Problem mit dem Start der iPad Version wurde behoben. Nach einem Meeting folgten noch weitere Absprachen mit Tobias und ich arbeitete weiter an dem Bericht.

**14.04. - 19.04.** Ein kritisches Problem mit den Sessions der vorherigen Woche konnte in dieser endlich gelöst werden. Dazu konnte ich die Anwendung etwas optimieren und einen sehr großen Teil des Beleges fertig stellen. Dabei half auch die Beantwortung vieler Fragen während eines Meetings mit dem Betreuer. Allerdings entstanden Probleme mit der Datenbank, die die Funktionalität der App einschränkten.

**20.04. - 26.04.** Ziel dieser Woche war es, so viel wie möglich des Belegs zu schreiben. Dafür fertigte ich einige Schemata und Bilder an. Zusätzlich konnte ich einige Fehler der App analysieren und beheben, sodass die Performance verbessert und Server-Aufrufe eingespart wurden. Außerdem kam die Online-Texterkennung über den Server hinzu sowie die zweite Korrektur in der Texterkennung mit Vision. Jedoch konnte die Online-Texterkennung auf Grund von Server-Problemen nicht getestet werden. Auch in

dieser Woche konnten einige Fragen zum Aufbau des Berichts in einem Meeting geklärt werden.

**27.04. - 03.05.** Im Mittelpunkt der Woche stand der Bericht, an dem ich täglich arbeitete. Ich erstellte einige Bilder für ihn und den Vortrag. Auch gab es diese Woche wieder ein Meeting.

**04.05. - 10.05.**

**11.05. - 15.05.**

## Literaturverzeichnis

- [1] APPLE. Detecting Objects in Still Images | Apple Developer Documentation (2019). Verfügbar unter: [https://developer.apple.com/documentation/vision/detecting\\_objects\\_in\\_still\\_images](https://developer.apple.com/documentation/vision/detecting_objects_in_still_images).
- [2] APPLE. Text Recognition in Vision Framework - WWDC 2019 - Videos (2019). Verfügbar unter: <https://developer.apple.com/videos/play/wwdc2019/234/>.
- [3] BRAGGE, M. *Model-View-Controller architectural pattern and its evolution in graphical user interface frameworks*. Bachelorarbeit (2013). Verfügbar unter: <https://lutpub.lut.fi/handle/10024/92156>.
- [4] FREEMAN, A. *Pro ASP.NET Core MVC 2*. Expert's Voice in .NET. Apress, 2 (30. Juni 2010) Edition (2017). ISBN 978-1-4842-3150-0. Verfügbar unter: <http://www.books24x7.com/marc.asp?bookid=135443>.
- [5] MITTWEIDA, H. Hochschule Mittweida: Portrait. Verfügbar unter: <https://www.hs-mittweida.de/hochschule/portrait.html>.
- [6] PAPA, J. Fundamental MVVM - (2011). Verfügbar unter: <https://visualstudiomagazine.com/articles/2011/08/15/fundamental-mvvm.aspx>.
- [7] ROYCE, D. W. W. MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS. (1970), 11. Verfügbar unter: <http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf>.
- [8] SILLMANN, T. Einstieg in SwiftUI (2019). Verfügbar unter: <https://www.heise.de/developer/artikel/Einstieg-in-SwiftUI-4594018.html>.

## **Erklärung**

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 13.03.2020