1 Einleitung

Eine angenehme Erfahrung für den Nutzer einer Software entsteht unter anderem dann, wenn ihm die richtigen Information zur richtigen Zeit präsentiert werden. In Formularen spielen Einfach- und Mehrfachauswahl Felder – im Englischen unter dem Begriff multiple choice Zusammengefasst – eine Rolle.

Die richtigen Informationen zur richtigen Zeit zu präsentieren könnte in diesem Kontext bedeuten, nur solche Auswahloptionen anzubeten, welche mit den bisherigen gewählten Optionen Sinn ergeben. Für die Datenerfassung von Maßnahmen auf landwirtschaftlich genutzten Flächen stellt dies eine Herausforderung dar, denn die Auswahlfelder und Optionen sind zahlreich und ihre Bedingungen komplex. Es lassen sich folgende Probleme ableiten.

1.1 Problemstellung

Das primäre Problem und damit Musskriterium der Formular-Anwendung ist, dass sich die Auswahlfelder untereinander beeinflussen. Wird eine Option in einem Auswahlfeld selektiert, so werden die möglichen Auswahlfelder von potenziell jedem weiteren Auswahlfeld dadurch manipuliert. Es muss eine Möglichkeit gefunden werden, die Abhängigkeiten in einer einfachen Art und Weise für jede Auswahloption zu hinterlegen und bei Bedarf abzurufen.

Das sekundäre Problem, welches sich vom primären Problem ableiten lässt, ist die Laufzeitgeschwindigkeit. Wenn die Auswahl in einem Auswahlfeld die Auswahlmöglichkeiten in potenziell allen anderen Auswahlfeldern manipuliert, so könnte dies zu einer hohen Last beim erneuten Zeichnen der Oberfläche zur Folge haben. Wann immer der Nutzer eine Selektion tätigt, müsste das gesamte Formular neu gezeichnet werden, um sicherzustellen, dass invalide Auswahloptionen gekennzeichnet werden. Bei einem Formular mit wenigen Auswahlfeldern wäre das kein Problem, doch die nötigen Auswahlfelder für das Eintragen von Maßnahmen des Europäischen Landwirtschaftsfonds für die Entwicklung des ländlichen Raums (ELER) sind zahlreich. Ein automatisierter Integrationstest, welcher im Formular Daten einer beispielhaften Maßnahme einträgt, zählt zum Zeitpunkt der Erstellung dieser Arbeit bereits 58 aufgerufene Auswahlfelder und 107 darin selektierte

Auswahloptionen. Das bedeutet, dass bei jedem dieser 107 Selektionen die 58 Auswahlfelder und all ihre Kinder neu gezeichnet werden müssten. Es entstehen also Wartezeiten nach jedem Auswählen einer Option. Das Formular soll in Zukunft zudem noch erweitert und auch für die Eingabe ganz anderer Datensätze mit potenziell noch mehr Auswahlfeldern eingesetzt werden können. Die Dateneingabe wäre mit den Wartezeiten trotzdem möglich. Daher ist es ein Wunschkriterium, dass ein Mechanismus gefunden wird, der nur die Elemente neu zeichnet, die sich wirklich ändern.

Ein weiteres Wunschkriterium ist, dass der Benutzer beim Anwählen einer deaktivierten Auswahloption eine Mitteilung darüber erhält, welche der zuvor ausgewählten Optionen zu der Inkompatibilität mit dem gewünschten Optionen führt.

Ziel dieser Masterarbeit ist es eine geeignete Technologie für die Umsetzung auszuwählen und die Umsetzbarkeit der oben genannten Kriterien zu evaluieren.

1.2 Gliederung

Kapitel **??** evaluiert die Kandidaten der Frontendtechnologien, die für eine nähere Betrachtung infrage kommen. Dazu werden die Umfrageergebnisse der Stack Overflow -Umfragen sowie das relative Suchinteresse dieser Technologien auf Google Trends analysiert. Da die Technologien React Native und Flutter die am verbreitetsten Technologien hervorgingen, werden sie daraufhin einem detaillierteren Vergleich unterzogen.

Da als Frontendtechnologie für die Entwicklung der Formularanwendung Flutter gewählt wurde, beschäftigt sich Kapitel **??** mit den Grundlagen des Frameworks und der zugrunde liegenden Programmiersprache Dart.

Die Kapitel 2 bis **??** dokumentieren die nötigen Entwicklungsschritte, um die einzelnen aufeinander aufbauenden Funktionalitäten hinzuzufügen. Die während der Arbeit im Thünen-Institut entstandene Anwendung wurde zu diesem Zweck auf die für die Problemstellung bedeutsamsten Funktionalitäten reduziert. Die Anzahl der Auswahlfelder beschränkt sich darüber hinaus auf ein Mindestmaß, welches die Bedingungen der Auswahloptionen untereinander erkennbar macht.

Kapitel 2 stellt die grundlegende Struktur der Anwendung her. Kapitel 3 fügt Hilfsmethoden hinzu, welche das Hinzufügen weiterer Formularfelder in den folgenden Schritten vereinfachen wird.

In Kapitel 4 erhält die Anwendung die grundlegende Funktion, Felder zu validieren. Kapitel **??** erweitert die Validierung schließlich um die Bedingungen der Auswahloptionen. Als Konsequenz werden alle Formularfelder neu gezeichnet, sollte der Benutzer eine beliebige

Auswahloption selektieren. Durch die Validierung geschieht es nach dem Neuzeichnen, dass invalide Auswahlfelder rot markiert werden. Die erforderlichen Änderungen, um nur die Auswahlfelder zu aktualisieren, die ihre Validität oder ihren eigenen Inhalt ändern, wird in Kapitel **??** hinzugefügt.

Kapitel **??** ergänzt die Möglichkeit, Mehrfachauswahlfelder zu verwenden. Kapitel **??** sorgt dafür, dass auch benutzerdefinierte Bedingungen für die Auswahlfelder hinterlegt werden können.

Kapitel 5 setzt sich mit den Erkenntnissen auseinander, die während der Entwicklung der Anwendung gesammelt wurden. Kapitel 6 bewertet die Erkenntnisse, ergänzt sie um einen Ausblick und vergleicht die Ergebnisse der Entwicklung mit den Anforderungen.

2 Schritt 1 - Formular in Grundstruktur erstellen

2.1 Widget SelectionCard

Das Listing 2.1 zeigt die Struktur des Widgets `SelectionCard`. Die Klasse hat einen generischen Typparameter (Z. 15). `<ChoiceType extends Choice>` bedeutet, dass die SelectionCard nur für Typen verwendet werden kann, die von `Choice` erben. Das ist eine wichtige Voraussetzung, da auf den übergebenen Werten Operationen ausgeführt werden sollen, die nur `Choice` unterstützt. Alle Parameter, die dem Konstrukt übergeben werden, leiten ebenso von diesem Typparameter ab. Einzige Ausnahme dabei ist der `titel` `16`.

Listing 2.1: Die Klasse SelectionCard

Mit dem Stream `selectionViewModel` verwaltet die `SelectionCard` ihren eigenen Zustand. Der Stream ist mit dem generischen Typen `BuiltSet<ChoiceType>` konfiguriert. Das macht es unmöglich, den aktuell hinterlegten Wert anzupassen, ohne das Gesamtobjekt auszutauschen. Der Tausch des Objekts wiederum bewirkt, dass ein Ereignis über den Stream ausgelöst wird. Über dieses Ereignis zeichnet die SelectionCard Teile seiner Oberfläche neu. Allerdings erhält der Konstruktor kein Argument des Typs `BehaviorSubject`, sondern stattdessen vom `Iterable<ChoiceType>` (Z. 24). Damit wird der Benutzer nicht darauf eingeschränkt, einen Stream zu übergeben. Er kann auch eine gewöhnliche Liste oder Menge setzen. Die Umwandlung der ankommenden Kollektion erfolgt in der Initialisierungsliste `29-30`. Nur so ist es möglich, die Instanzvariable mit `final` als unveränderbar zu kennzeichnen. Initialisierungen solcher Variablen müssen im statischen Kontext der Objekterstellung geschehen. Der Konstruktor-Körper gehört dagegen nicht mehr zum statischen Teil. Im Konstruktor-Körper können Operationen der Instanz verwendet werden, denn das Objekt existiert bereits. Der Versuch eine mit `final` gekennzeichnete Instanzvariable im Konstruktor-Körper zu setzen, führt zu einem Compilerfehler in Dart. Der Konstruktor `seeded` der Klasse `BehaviorSubject` wird mit einem `BuiltSet` gefüllt (Z. 29). Dieses wiederum wird mit dem benannten Konstruktor `from` von `BuiltSet` mit der Kollektion aufgerufen (Z. 30). Er wandelt die Liste in eine unveränderbare Menge um. Die Liste aller Auswahloptionen `allChoices` (Z. 18) gewährleistet über den generischen Typparameter, dass nicht versehentlich Auswahloptionen übergeben werden, die nicht zum

Typ der `SelectionCard` passen. Die Rückruffunktionen (Z. 19, 20), die bei Selektion und Deselektion von Optionen ausgelöst werden, bieten einen besonderen Vorteil dadurch, dass sie mit dem generischen Typen konfiguriert sind. Die Signaturen der Rückruf-Funktionen (Z. 7-8, 10-11) geben nämlich vor, dass der erste Parameter vom Typen `ChoiceType` sein muss. Wenn nun der Benutzer der SelectionCard einen Typ wie etwa `LetzterStatus` für den Typparameter übergibt, so erhält er auch eine Rückruffunktion, dessen erster Parameter vom Typ `LetzterStatus` ist. Ohne eine Typumwandlung - englisch type casting - von (Z. Choice) in `LetzterStatus`, können keine Operationen auf das Objekt angewendet werden, die nur die Klasse `LetzterStatus` unterstützt.

Das erste Element, welches von der `build`-Methode zurückgeben wird, ist ein `StreamBuilder` (Listing 2.2, Z. 47). Er horcht auf das `selectionViewModel` (Z. 48). Sobald also eine Selektion getätigt wurde, aktualisiert sich auch die dazugehörige Karte. Das Aussehen einer Karte wird durch das Widget `Card` erreicht (Z. 51). Dadurch erhält es abgerundete Ecken und einen Schlagschatten, der es vom Hintergrund abgrenzt. Ein `ListTile` Widget erlaubt es dann, den übergebenen `titel` als Überschrift zu setzen (Z. 54) und die aktuell ausgewählten Selektionen als Untertitel anzuzeigen (Z. 56). Zu diesem Zweck wandelt die Methode `map` alle Elemente von `selectedChoices` in `String`-Objekte um, indem es von dem `Choice`-Objekt lediglich den Beschreibungstext `description` verwendet. Anschließend sammelt der Befehl `join` die resultierende `String`-Objekte ein, formt sie in einen gemeinsamen `String` zusammen und trennt sie darin jeweils mit einem `", "` voneinander.

Listing 2.2: Die Build Methode der SelectionCard

Das `ListTile` erhält ein `FocusNode`-Objekt (Z. 53), damit der Benutzer beim Zurücknavigieren von der Unterseite im Formular wieder in der gleichen vertikalen Position der Karte landet, die er zuvor ausgewählt hat. Der Benutzer würde ansonsten in Formular wieder an der obersten Position herauskommen. Der `FocusNode` wird einmal zu Anfang der `build`-Methode erstellt (Z. 35). Damit ist er außerhalb der Methode `builder` des `StreamBuilder`-Widgets und bleibt somit beim Neuzeichnen der Karte erhalten.

Klickt der Benutzer die Karte an, navigiert er schließlich zur Unterseite, wo er die Auswahloptionen präsentiert bekommt. Die verschachtelte Funktion `navigateToSelectionScreen` kommt dafür zum Einsatz (Z. 37-45). Da das Wechseln zur Unterseite bevorsteht, fordert der `focusNode` den Fokus für das angeklickte `ListTile` an (Z. 38). Schließlich navigiert der Benutzer mit `Navigator.push` zur Unterseite. Es handelt sich um den Auswahlbildschirm, auf dem der Benutzer die gewünschte Option anwählen kann. Die Besonderheit dieses Mal: die Route ist nicht als Widget deklariert und wird nicht über einen Namen aufgerufen, so wie es bei dem Übersichtsbildschirm und der Eingabemaske war. Stattdessen baut eine Funktion bei jedem Aufruf die Seite neu. Das dynamische Bauen der Seite hat einen besonderen Vorteil, der am Listing 2.3 erklärt wird.

2.1.1 Bildschirm für die Auswahl der Optionen

Die Funktion `createMultipleChoiceSelectionScreen` (Listing 2.3) gibt einen `Scaffold` zurück, der die gesamte Seite enthält (Z. 65). Das erste Kind des `Scaffold` ist wiederum ein `StreamBuilder` (Z. 69). Hier wird der Vorteil der dynamischen Erzeugung der Seite offensichtlich: die Unterseite kann das gleiche ViewModel wiederverwenden, welches auch von der `SelectionCard` genutzt wird. Auch alle weiteren Instanzvariablen der `SelectionCard` können wiederverwendet werden. Würde es sich stattdessen um eine weitere Route handeln, so müssten alle diese Informationen über den Navigator zur neuen Unterseite übergeben werden. Sollte der Nutzer die Auswahl beenden, so müsste auch ein Mechanismus für das Zurückgeben der selektierten Daten implementiert werden. Dadurch, dass die `SelectionCard` und der Auswahlbildschirm sich das gleiche ViewModel teilen, kann sogar ein weiterer Vorteil in Zukunft genutzt werden: in einem zweispaltigen Layout könnte auf der linken Seite die Eingabemaske und auf der rechten Seite der Bildschirm der Auswahloptionen eingeblendet werden. Sobald sich Auswahloptionen im rechten Auswahlbildschirm verändern, so würden sich die Änderungen auf der linken Seite für den Benutzer direkt widerspiegeln.

Innerhalb des `StreamBuilder` werden die Auswahloptionen gebaut. Dazu speichert die lokale Variable `selectedChoices` die aktuellen Selektionen des Streams zunächst zwischen (Z. 72). Die Optionen werden in einem ListView präsentiert (Z. 73). Er ermöglicht es, Listen-Elemente in einem vertikalen Scrollbereich darzustellen. Die Funktion `map` konvertiert alle Objekte in der Liste aller möglichen Optionen `choices` in Elemente des Typs `CheckboxListTile` (Z. 74-98). In der Standard-Variante sind die Checkboxen rechtsbündig. Der Parameter `controlAffinity` kann genutzt werden, um dieses Verhalten zu überschreiben (Z. 80).

Das `CheckboxListTile` erhält einen Titel, der aus dem Beschreibungstext `description` des `Choice`-Objekts gebildet wird (Z. 81). Ob eine Option aktuell bereits ausgewählt ist, kann mit dem Parameter `value` übertragen werden (Z. 82). Sollte sich die Selektion ändern, erfolgt die Mitteilung über die Rückruffunktion `onChanged` (Z. 83-94). Der erste Parameter der anonymen Funktion gibt dabei die ausgewählte Selektion an. Eine Fallunterscheidung überprüft zunächst, ob der Parameter `selected` nicht `null` ist, denn sein Parametertyp `bool?` lässt Null-Werte zu. Durch die Typ-Beförderung ist `selected` innerhalb des Körpers der Fallunterscheidung dann vom Typ `bool` (Z. 84-94).

Darin wird zunächst der Zustand des ViewModels der `SelectionCard` aktualisiert. Die `replace`-Methode des *„Builder“*-Objekts kann die gesamte Kollektion im `BuiltSet` austauschen, ungeachtet dessen, dass es sich beim Argument selbst nicht um ein `BuiltSet` handelt. Die `replace`-Methode wandelt das Argument dafür automatisch um. Durch Zuweisung des neuen Wertes erhält das ViewModel der `SelectionCard` ein neues Ereignis. Damit wird die `SelectionCard` und der dazugehörige Auswahlbildschirm aktuali-

siert. Während der Erstellung dieser Arbeit wurde versucht, die SelectionCard als ein `StatefulWidget` zu erstellen. Mittels `setState` sollte dafür gesorgt werden, dass sowohl SelectionCard als auch der Auswahlbildschirm aktualisiert werden. Doch bei diesem Vorgehen zeichnet sich nur die `SelectionCard` neu. Der Auswahlbildschirm bleibt unverändert, denn er wird zwar von der `SelectionCard` gebaut, doch ist er nicht tatsächlich Kind der `SelectionCard`. In Wahrheit ist der Auswahlbildschirm ein Kind von `MaterialApp` - genau wie `MassnahmenMasterScreen` und `MassnahmenDetailScreen`.

Neben dem ViewModel der `SelectionCard` muss jedoch auch das ViewModel der Eingabemaske aktualisiert werden. Mit den Rückruffunktionen `onSelect` (Z. 90) und `onDeselect` (Z. 92) hat die aufrufende Ansicht die Möglichkeit, auf Selektionen zu reagieren.

Schließlich ist noch der `FloatingActionButton` Teil der Unterseite (Z. 99-103). Mit einem Klick darauf gelangt der Benutzer zurück zur Eingabemaske (Z. 100).

Listing 2.3: Die Funktion createMultipleChoiceSelectionScreen

2.2 Integrations-Test zum Test der Oberfläche

Ein automatisierter Integrationstest soll verifizieren, dass die Oberfläche wie vorgesehen funktioniert. Der Integrationstest simuliert einen Benutzer, der die Applikation verwendet, um eine Maßnahme einzutragen. Bei Abschluss des Tests soll überprüft werden, ob die eingegebenen Daten mit den Inhalten der JSON-Datei übereinstimmen.

Flutter erlaubt über einen eigenen Testtreiber solche Integrationstest durchzuführen. Dabei wird die Applikation zur Ausführung gebracht, und jeder Schritt so visualisiert, wie es bei der Ausführung der realen Applikation der Fall wäre. Der Entwickler hat damit die Möglichkeit, die Eingaben und Interaktionen zu beobachten und gegebenfalls zu bemerken, warum ein Testfall nicht korrekt ausgeführt wird.

Das Ergebnis des Integrationstests soll allerdings nicht mit der tatsächlich geschriebenen JSON-Datei überprüft werden. Der Test soll nicht tatsächlich Daten auf der Festplatte speichern. Das würde die Gefahr bergen, das vergangene Eingaben manipuliert werden. Stattdessen soll der Test in einer Umgebung stattfinden, die keine Auswirkung auf die Haupt-Applikation oder zukünftige Tests haben soll. Zu diesem Zweck können sogenannte Mocks genutzt werden. Das Paket *„mockito“* erlaubt über Annotationen solche Mocks für die gewünschten Klassen über Quellcode-Generierung zu erstellen.

Integrationstests werden im Ordner integration_test angelegt. Während des Zeitpunkt der Erstellung dieser Arbeit war es in der Standardkonfiguration der Quellcode-Generierung

und dem Paket *„mockito“* nicht möglich, Mocks auch im integration_test Ordner zu generieren. Lediglich innerhalb des test Ordners, der für die Unit-Tests vorgesehen ist, hat die Annotation generate mocks funktioniert. Zu diesem Fehlverhalten existiert ein entsprechendes Issue im GitHub Repository des Mockito packages. Ref Um das Generieren von Mocks auch für Integrationstest verfügbar zu machen, hat der Autor dieser Arbeit einen entsprechenden Lösungsansatz recherchieren und im Issue beschrieben. Ref

Damit der integration_test Ordner für die Quellcode-Generierung der Mocks integriert wird, muss ein entsprechender Eintrag in der Build-Konfiguration vorgenommen werden. Damit das Paket *„source_ gen“* die entsprechenden Dateien analysiert, müssen sie in der Rubrik `sources` angegeben werden (Listing 2.5, Z. 3-8). Wird der Ordner integration_test darin eingefügt (Z. 8), bezieht *„source_ gen“* den Ordner in der Quellcode-Generierung mit ein. Zusätzlich dazu muss die Rubrik `generate_for` von dem `mockBuilder` des *„mockito“*-Pakets (Z. 11-13) um die gleiche Angabe des Ordners ergänzt werden (Z. 13).

Listing 2.4: Initialisierung des Integrations Tests

Anschließend kann mit der Annotation and generate mocks (Listing 2.5, Z. 20) ein Mock für `MassnahmenJsonFile` angefordert werden. In der Kommandozeile ist `flutter pub run build_runner` einzugeben, damit der entsprechende Quellcode generiert wird. Mit dem Mock kann der Integrationstest ausgeführt werden, ohne dass befürchtet werden muss, dass die JSON-Datei tatsächlich beschrieben wird. Stattdessen kann darauf gehorcht werden, wenn Operationen auf dem Objekt ausgeführt werden.

Listing 2.5: Initialisierung des Integrations Tests

Die Funktion `testWidgets` startet den Test und erhält als ersten Parameter das `tester`-Objekt (Z. 22). Darüber ist die Interaktion mit der Oberfläche während des Tests möglich. In den Zeilen 22 bis 25 wird der Testtreiber initialisiert. Ref. Anschließend wird ein Objekt der generierten Klasse `MockMassnahmenJsonFile` erstellt. Wenn das Model nun während der Applikation versucht, aus der JSON-Datei zu lesen, soll der Mock eine leere Liste von Maßnahmen zurückgeben (Z. 28). Dazu wird die entsprechende Methode `when` verwendet. Als erster Parameter wird die Methode `readMassnahmen` des Mocks übergeben. Im darauffolgenden Aufruf `thenAnswer` kann angegeben werden, welche Rückgabe die Methode liefern soll.

Über den `tester` kann mit Hilfe der Methode `pumpWidget` ein beliebiges Widget in der Test-Ausführung konstruiert werden. In diesem Fall ist es die gesamte Applikation, die getestet werden soll. Dementsprechend ist hier erneut der komplette Haupteinstiegspunkt angegeben (Listing 2.6). Doch der Konstruktor von (Z. MassnahmenModel) erhält dieses Mal nicht das `MassnahmenJsonFile`, sondern den entsprechenden Mock (Z. 31).

Weil während des Integrationstest immer wieder die gleichen Operationen wie das Selek-

Listing 2.6: Initialisierung des Widgets für den Integrations Tests

tieren einer Selektions-Karte, das Auswählen einer Option, das Anklicken des Buttons zum Akzeptieren der Auswahl und das Füllen eines Eingabefeldes auftauchen, wurden entsprechende Hilfsfunktionen erstellt.

Der Funktion `tabSelectionCard` (Listing 2.8) benötigt lediglich die Liste der Auswahloptionen `choices`, die ihr hinterlegt ist.

Listing 2.7: Die Hilfsmethode tabSelectionCard

Um Objekte während des Testens in Oberfläche zu finden, stellt die Klasse `Finder` nützliche Funktionalitäten zur Verfügung. `Finder`-Objekte können über Fabrikmethoden des Objekts `find` abgerufen werden.

Fabrikmethoden Bei der Fabrikmethode handelt es sich um ein klassenbasiertes Erzeugungsmuster. Anstatt ein Objekt einer Klasse direkt über einen Konstruktor zu erstellen, erlaubt ein Erzeuger das Objekt zu konstruieren. Dabei entscheidet der Erzeuger darüber, welche Implementierung der Klasse zurückgegeben wird. Der aufrufende Kontext muss die konkrete Klasse dazu nicht kennen.[1] Er arbeitet lediglich mit der Schnittstelle. In diesem Fall ist `find` dieser Erzeuger. Über die Fabrikmethode `text` wird ein `_TextFinder` konstruiert, jedoch über die Schnittstelle `Finder` zurückgegeben. Eine weitere Fabrikmethode ist `ancestor`. Sie gibt einen `_AncestorFinder` zurück, welcher ebenso hinter der Schnittstelle `Finder` versteckt wird. `Ref`. Die Fabrikmethoden werden hier deshalb verwendet, weil sie die Lesbarkeit verbessern. Anstatt `Finder titel = new _TextFinder("Maßnahmentitel")` ist `Finder titel = find.text("Maßnahmentitel")` deutlich leichter zu erfassen.

Um die Selektions-Karten zu finden, wird lediglich der Titel- Text benötigt. Angenommen der Test ruft `tabSelectionCard` mit dem Argument `letzterStatusChoices` auf, so entspricht choices.name dem String `"Status"`. Der Ausdruck `find.text("Status")` lokalisiert den Titel innerhalb der Selektions-Karte (Z. 50).

Die Funktion `expect` erwartet als ersten Parameter einen `Finder` und als zweiten einen sogenannten *„Matcher“* (Z. 51). Der Aufruf von `expect` mit dem entsprechenden `Finder`-Objekt und dem Matcher `findsWidgets` verifiziert, dass mindestens ein entsprechendes Text Element gefunden wurde.

Wurde das Text-Element gefunden, so muss noch den Vater gesucht werden, der vom Typ `Card` ist (Z. 53). Das kann mit `find.ancestor` erfolgen. Über den Parameter `of`

[1] Vgl. 2, S. 107–116.

erhält er den `Finder` des Kind-Elements und der Parameter `matching` erhält als Argument die Voraussetzung, die vom Vater-Objekt erfüllt werden soll, als weiteren `Finder`. `find.byType(Card)` sucht also alle Elemente vom Typ `Card`. `find.ancestor` sucht anschließend alle Entsprechungen, in der eine `Card` ein Vater des `Finder textLabel` ist. Wiederum überprüft die Funktion `expect`, dass die Karte gefunden wurde. Doch dieses Mal muss es genau ein Widget sein, welches mit dem *„Matcher“* `findsOneWidget` verifiziert werden kann (Z. 54). Sollte mehr als nur eine Karte gefunden werden, so wäre nicht klar, welche geklickt werden soll.

Um eine Karte tatsächlich anzuwählen muss sie im sichtbaren Bereich sein. Die Methode *„ensureVisible“* scrollt den Bildschirm zur entsprechenden Position, damit die Karte sichtbar ist (Z. 56). Schließlich sorgt `tab` mit dem `Finder card` dafür, dass die Karte ausgewählt wird. `pumpAndSettle` (Z. 58) ist eine obligatorische Methode, die nach jeder Aktion durchgeführt werden muss. Sie sorgt dafür, dass der Test so lange pausiert, bis alle Aktionen in der Oberfläche und damit auch alle angestoßenen Animationen vorüber sind. Zusätzlich kann eine Dauer angegeben werden, die darüber hinaus gewartet werden soll.

`tabConfirmButton` funktioniert ähnlich (Listing 2.8). Das Finden des Buttons ist jedoch einfacher, da es nur einen Button zum Akzeptieren auf jeder Oberfläche gibt. Der Button enthält keinen Text, lässt sich aber auch über seinen Tooltip lokalisieren (Z. 62). Die Hilfsfunktion klickt den Button (Z. 63) und wartet dann erneut auf Vollendung aller angestoßenen Animationen (Z. 64).

Listing 2.8: Die Hilfsmethode tabConfirmButton

Ist der Integrationstest aktuell in dem Auswahlbildschirm, so sorgt `tabOption` dafür, dass Auswahloptionen gewählt wird (Listing 2.9). Dazu wird die gewünschte Option dem Parameter `choice` übergeben. Um die Checkbox der Option zu finden, muss jedoch zunächst der Text der Auswahloption gefunden werden (Z. 68). Erst wenn verifiziert wurde, dass auch nur genau ein Label mit diesem Text existiert, läuft der Test weiter (Z. 69).

Listing 2.9: Die Hilfsmethode tabOption

Ein Klick auf das Text-Label reicht bereits aus, denn damit wird das Vater-Element - das `CheckboxListTile` - ebenfalls getroffen. Der `tester` holt es in den sichtbaren Bereich `71`, klickt es `72` und wartet auf Abschluss aller Animationen (Z. 73). Sollte der optionale Parameter `tabConfirm` auf `true` gesetzt sein (Z. 75), so wird der Auswahlbildschirm anschließend direkt wieder geschlossen, nachdem die Option ausgewählt wurde (Z. 76).

Schließlich kann mit der Hilfsfunktionen `fillTextFormField` ein Formularfeld über dessen Titel gefunden und der entsprechende übergebende Text eingetragen werden (Listing 2.10). Sie findet das `TextFormField`, indem es zunächst nach dem Titel mit `find.text(title)` und anschließend dessen Vater-Element vom Typ `TextFormField` sucht (Z. 83). Sollte

sowohl der Hinweistext als auch der Titel den gleichen Text enthalten, so kann es sein, dass zwei solche Elemente gefunden werden. In Wahrheit ist es aber zwei Mal dasselbe `TextFormField`. Mit `.first` wird lediglich das erste Element geliefert (Z. 85). Nachdem feststeht, dass das Element existiert (Z. 85) und es in den sichtbaren Bereich gescrollt wurde (Z. 87), gibt der Integrationstest den gewünschten Text in das Eingabefeld ein (Z. 88). Anschließend wird erneut auf Abschluss aller Animationen gewartet (Z. 89).

Listing 2.10: Die Hilfsmethode fillTextFormField

Während der Integrationstest startet, öffnet sich als Erstes der Übersichts-Bildschirm. Zunächst wird gewartet, dass alle Widgets korrekt initialisiert wurden (Listing 2.11, Z. 92). Es folgt der Klick auf den Button zum Erstellen einen neuen Maßnahme (Z. 95). Dazu wird der Button über den entsprechenden `Key` gefunden (Z. 94). Vor allem jetzt ist das Abwarten mittels `pumpAndSettle` (Z. 96) unablässig, denn es wird auf einen anderen Bildschirm navigiert. Angenommen der Test wartet nicht ab, so würden die Aktionen noch immer auf den Elementen des alten Bildschirms Anwendung finden.

Listing 2.11: Der Button zum Kreieren einer Maßnahme wird ausgelöst

Der Integrationstest öffnet nun den Auswahl-Bildschirm, in dem die Selektions-Karte zum Setzen des letzten Statuses angewählt wird (Listing 2.12, Z. 98). Anschließend fällt die Wahl auf die Option für *„abgeschlossen“* (Z. 98). Dabei sorgt `tabConfirm: true` für die sofortige Rückkehr zum Eingabeformular nach der Auswahl.

Listing 2.12: Der letzte Status wird ausgewählt

Nachfolgend soll der Test das Eingabefeld für den Maßnahmen-Titel überprüfen (Listing 2.13). Es erfolgt die Erstellung eines beispielhaften Titels anhand des aktuellen Datums und der aktuellen Uhrzeit (Z. 101, 102). Der erstellte Text dient als Eingabe für das Eingabefeld (Z. 104).

Die nötigen Eingaben sind erfolgt. Daher kann der Test nun den Klick auf den Button zum Speichern simulieren (Listing **??**, Z. 106-108). Dadurch würde in der Anwendung nun das Speichern der Maßnahmen in der JSON-Datei erfolgen. Doch da stattdessen ein Mock verwendet wurde, passiert dies nicht. Das Model ruft aber dennoch die entsprechenden Methoden - wie zum Beispiel `saveMassnahmen` - auf. Die Methoden haben nur nicht die ursprüngliche Funktion. Stattdessen protokollieren sie sowohl die Aufrufe, als auch die übergebenen Argumente. Durch die Methode `verify` (Z. 111) kann überprüft werden, ob die entsprechende Methode `saveMassnahmen` ausgeführt wurde. Der *„Matcher“* `captureAny` ermöglicht die Überprüfung auf irgendeine Übergabe und stellt die übergebenen Argumente über den Rückgabewert bereit.

Die Rückgabe ist vom Typ `VerificationResult` und enthält eine Getter-Methode mit

Listing 2.13: Der Maßnahmentitel wird eingegeben

Listing 2.14: Validierung des Testergebnisses

dem Namen `captured`. Dabei handelt es sich um eine Liste aller Argumente, die in den vergangenen Aufrufen übergeben wurden. Mit `last` lässt sich auf das Argument des letzten Aufrufes zurückgreifen.

Nun soll sich zeigen, ob das übergebene Argument mit dem erwarteten Wert übereinstimmt. Weil das Ergebnis eine Liste mit lediglich einer Maßnahme ist,soll auch ausschließlich diese Maßnahme verglichen werden. Der Schlüssel `'massnahmen'` greift auf die Liste zurück und der Schlüssel `0` auf die erste und einzige Maßnahme. Die lokalen Variable `actualMassnahme` speichert sie zwischen (Z. 113).

Es ist unklar, welche zufällige guid bei der Erstellung der Maßnahme generiert wurde. Auch der Zeitstempel hinter dem Schlüssel `"letzteBearbeitung"` ist unbekannt. Eine mögliche Lösung wären weitere Mocks, welche die Erstellung der guid und des Datums überwachen und - anstelle einer zufälligen - immer die gleiche Zeichenkette zurückgibt. Es ist jedoch auch möglich, die Vergleiche der guid und des Zeitstempels auszuschließen. Dazu reicht es die entsprechenden Schlüssel-Werte-Paare über die Schlüssel `"guid"` und `"letztesBearbeitungsDatum"` aus der Ergebnis-Hashtabelle zu entfernen (Z. 114-115).

Die lokale Variable `expectedJson` speichert das erwartete Ergebnis der eingegebenen Maßnahme (Z. 117-120). Die Methode `expect` und der *„Matcher“* `equals` überprüfen beide Objekte auf Gleichheit (Z. 122).

Der Befehl `flutter test integration_test/app_test.dart` startet den Test. Die App öffnet sich und der Ausführung des Tests kann zugesehen werden Punkt am Endeerfolgt in dem Terminal die Ausgabe des Ergebnisses: `All tests passed!`

3 Schritt 2

Abbildung 3.1: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel

Abbildung 3.2: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel

In diesem Schritt sollen weitere Selektions-Karten für die Einzelauswahlfelder hinzugefügt werden. Es handelt sich um die Einzelauswahlfelder für Förderklasse, Kategorie, Zielfläche, Zieleinheit und Zielsetzung.

Darüber hinaus soll das Erstellen der Selektions-Karten in einer Methode abstrahiert werden. Das ermöglicht die Konfiguration der Selektions-Karten in der aufrufenden Eingabemaske, ohne dafür die Klasse `SelectionCard` ändern zu müssen.

3.0.1 Integrationstest erweitern

Noch vor der Implementierung der Änderungen soll zunächst der Integrationstest um die zusätzlichen Selektionen erweitert werden (Listing 3.1). Nach den letzten Eingaben und bevor der Button zum Speichern ausgelöst wird, erfolgt die Selektion der fünf Optionen (Z. 106-119).

Listing 3.1: Der Integrationstest klickt 5 weitere Karten

Nach der Auswahl und der anschließenden Serialisierung sollen die entsprechenden Werte auch in der Json-Datei auftauchen. Die Json-Datei erhält ein neues Schlüssel-Werte-Paar mit dem Schlüssel `'massnahmenCharakteristika'` und einem Objekt für die fünf neuen Werte (Listing 3.2, Z. 135-141).

Der Integrationstest ist damit aktualisiert. Die Implementierung ist jedoch noch gar nicht erfolgt. Die Selektions-Karten können nicht geklickt werden, da sie in der Oberfläche noch nicht auftauchen. Die neuen Schlüssel-Werte-Paare können nicht in der Hash-Tabelle auftauchen, da sie dem entsprechenden Werte-Typ noch nicht hinzugefügt wurden. Der Integrationstest kann also unmöglich erfolgreich sein. Der Quellcode kann noch nicht einmal kompilieren, da die entsprechenden Symbole – wie zum Beispiel `FoerderklasseChoice` –

Listing 3.2: Der Integrationstest klickt 5 weitere Karten

fehlen. Das hier angewendete Vorgehensmodell wird Test-Driven Development – deutsch Testgetriebene Entwicklung – genannt.

> *„Development is driven by tests. You test first, then code. Until all the tests run, you aren't done. When all the tests run, and you can't think of any more tests that would break, you are done adding functionality.“*
>
> — Kent Beck[1]

Es folgt das Hinzufügen der fehlenden Symbole, damit der Quellcode wieder kompiliert werden kann. Anschließend erfolgt die Weiterentwicklung des Models, ViewModels und Views damit der Integrationstest erneut erfolgreich abschließt.

3.0.2 Hinzufügen der Auswahloptionen

Der Integrationstest selektiert unter anderem die Förderklasse mit der Abkürzung `aukm_ohne_vns`. Sie wird den Auswahloptionen hinzugefügt, wie in Listing 3.3 zu sehen ist. Die Liste aller hinzugefügten Auswahloptionen in diesem Schritt ist in Anhang **??** auf den Seiten **??** bis **??** zu finden.

Listing 3.3: Die Klasse FoerderklasseChoice

3.0.3 Aktualisierung des Models

Damit der Hash-Tabelle der Schlüssel `'massnahmenCharakteristika'` hinzugefügt wird, muss der entsprechende Eintrag im Werte-Typ `Massnahme` hinzugefügt werden. Die Getter-Methode `massnahmenCharakteristika`, die das Paket *„built_value“* dazu veranlaßt, den Quellcode für die Eigenschaft zu generieren, wird unterhalb der Getter-Methode `identifikatoren` hinzugefügt (Listing 3.4, Z. 15).

Listing 3.4: massnahmenCharakteristika wird Massnahme hinzugefügt

Bei dem Datentyp handelt es sich um einen weiteren Werte-Typ: `MassnahmenCharakteristika`, welcher in Listing 3.5 zu sehen ist. Die darin enthaltenen Getter-Methoden sind dagegen lediglich gewöhnliche Zeichenketten, da sie die Abkürzungen der ausgewählten Optionen abspeichern. Da sie auch im Entwurfsmodus auch nicht gefüllt sein können, wird ihnen mit dem Suffix `?` erlaubt, auch Null-Werte anzunehmen (Z. 70-74).

[1] 1, S. 9.

Listing 3.5: Der Werte-Typ Massnahmencharakteristika

Der Werte-Typ wurde hinzugefügt. Der Befehl `flutter pub run build_runner build` generiert den Quellcode für die Serialisierung und die Builder-Methoden.

3.0.4 Aktualisierung der Übersichtstabelle

Der Übersichtsbildschirm bzw. die Übersichtstabelle können auf das Model ohne den Umweg über das ViewModel zugreifen. Der Tabellenkopf listet die Überschriften der hinzugefügten Werte auf (Listing 3.6, Z. 23-27).

Listing 3.6: Maßnahmencharakteristika werden dem Tabellenkopf hinzugefügt

Für jede Zeile der Tabelle werden weitere selektierbare Zellen generiert (Listing 3.7, Z. 33-42). Im Unterschied zur Zelle des Maßnahmen-Titels können die Getter-Methoden der Maßnahmen-Charakteristika jedoch Null-Werte enthalten. Doch das `Text`-Widget akzeptiert keine Null-Werte als Argument. Deshalb wird der Operator `??` verwendet. Dabei handelt es sich um die *„If-null Expression“*. Sie überprüft den Ausdruck links vom Operator `??`. Ist er `null`, so wird der Wert rechts vom Operator verwendet. Ist der dagegen nicht `null`, so wird der Wert links vom Operator `??` genutzt.[2] Ist der Wert also nicht gefüllt, so wird in allen Fällen der leere String `""` als Argument übergeben.

Listing 3.7: Maßnahmencharakteristika werden dem Tabellenkörper hinzugefügt

3.0.5 Aktualisierung des ViewModels

Damit die Eingabefelder die neuen Werte eintragen können, muss das ViewModel oder die beobachtbaren Subjects bereitstellen (Listing 3.8, Z. 12-17). `Subjects und Observer in Schritt 1 e`

Die Konvertierung des Models in das ViewModel erfolgt wie gewohnt über das Heraussuchen des korrekten Objektes aus der Menge der Auswahloptionen über die Abkürzung (Listing 3.9, Z. 29-36).

Wenn in jeder Zeile der Ausdruck `model.massnahmenCharakteristika` stehen würde, wäre die Leserlichkeit stark eingeschränkt. Das wurde für weitere Zeilenumbrüche sorgen. Deshalb speichert die lokale Variable `mc` den Ausdruck zwischen und kann in den folgenden Zeilen verwendet werden (Z. 27). Damit die variable `mc` jedoch nur Gültigkeit für die

[2] Vgl. 5, S. 165.

Listing 3.8: Maßnahmencharakteristika werden dem ViewModel hinzugefügt

Listing 3.9: Konvertierung des Models in das ViewModel

folgenden Zeilen hat, begrenzen die öffnenden und schließenden geschweiften Klammern den Sichtbarkeitsbereich (Z. 26,37).

Bei der Konvertierung des Models in das ViewModel wurde bereits beim letzten Schritt die Methode `update` verwendet, um das Objekt des geschachtelten Wertetyps `Identifikatoren` anzupassen (Listing 3.10, Z. 44). So ist es auch für den geschachtelten Wertetyp `MassnahmenCharakteristika` der Fall. Der Unterschied: Es handelt sich um Auswahloptionen, weshalb nur die Abkürzungen abgespeichert werden (Z. 46-50), so wie es auch schon bei `letzterStatus` geschah (Z. 42).

Listing 3.10: Konvertierung des ViewModels in das Model

3.0.6 Aktualisierung der Eingabemaske

Nach der Anpassung des ViewModels kann schließlich die Eingabemaske erweitert werden.

Im letzten Schritt nahm die Selektionskarte für den letzten Status 11 Zeilen ein `R`. Das wäre für jede weitere Karte nun auch der Fall. Damit die Übersichtlichkeit darunter nicht leidet, soll nun zunächst eine Methode erstellt werden, welche die Erstellung der Selektionskarten abstrahiert und damit den Aufruf auf 3 Zeilen reduziert. Dies erlaubt auch die Konfiguration der Selektionskarten außerhalb der Klasse `SelektionCard`. In den folgenden Schritten soll diese Konfigurationsmöglichkeit genutzt werden, um weitere Funktionalitäten hinzuzufügen, ohne die Klasse selbst zu manipulieren. Die Methode `buildSelectionCard` bekommt dazu nur die Argumente für die Liste aller Auswahloptionen `allChoices` (Listing 3.13, Z. 49) und das Subject `selectionViewModel` (Z. 50) übergeben. Nun übernimmt die Methode die Übergabe der Argumente an den Konstruktor der `SelectionCard`. Dazu verwendet die `SelectionCard` wie zuvor den Namen der Menge der Auswahloptionen als Titel (Z. 52). Außerdem wird dieselbe Menge unverändert an die SelektionCard weitergegeben (Z. 53).

Der Grund, warum die Klasse `SelectionCard` den Titel aus der Menge der Auswahloption nicht selbständig extrahiert ist, dass die Klasse auf diese Weise auch für mehrere Anwendungsgebiete genutzt werden kann. Es muss nicht immer der Fall sein, dass der Titel auf diese Art und Weise ausgelesen werden kann. Somit erlaubt die Methode `buildSelectionCard` nun den Aufruf trotzdem zu vereinfachen und die Anwendbarkeit

Listing 3.11: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

der Klasse `SelectionCard` durch dessen direkte Veränderung nicht einzuschränken.

Das betrifft auch das ViewModel. Durch die Methode `buildSelectionCard` muss lediglich das `BehaviorSubject` übergeben werden. Die Methode kümmert sich bei Initialisierung der Selektionskarte um das Auslesen des aktuellen Wertes (Z. 54-56) und die Aktualisierung dessen über die Methoden `onSelect` (Z. 57) `onDeselect` (Z. 58). Damit ist die Erstellung der Selektionskarte für den letzten Status mit 3 Zeilen (Listing 3.12) nun deutlich kürzer als die ursprüngliche Variante mit 11 Zeilen (siehe Seite **??**).

Listing 3.12: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

Unterhalb des Eingabefeldes für den Maßnahmen-Titel können nun die weiteren Selektionskarten ergänzt werden, die jeweils ebenfalls bloß 3 Zeilen einnehmen und damit eine hohe Übersichtlichkeit gewährleisten (Listing 3.13, Z. 82-98).

Listing 3.13: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

Auffällig hierbei sind Überschriften (Z. 80, 82) und ine Zwischenüberschrift (Z. 89) über den Selektionskarten. Sie sorgen für sichtbare Gruppierungen in der Oberfläche.

Die Hilfsmethode `buildSectionHeadline` und `buildSubSectionHeadline` bauen die Überschriften (Listing 3.14, Z. 131-134) bzw. Zwischenüberschriften (Z. 136-139) mit unterschiedlichen Abständen zur Außenkante (Z. 132, 137) und unterschiedlicher Schriftgröße (Z. 133, 138). Der benannte Konstruktor `fromLTRB` der Klasse `EdgeInsets` erlaubt die Abstände zur außenkante im Uhrzeigersinn für jede Seite festzulegen. Die Abkürzung `LTRB` steht dabei für für left, top, right, bottom – deutsch links, oben, rechts, unten.

Damit ist die Implementierung für Schritt 2 beendet.

Der Integrationstest kann nun verifizieren, dass die Eingaben erfolgen und in der Json-Datei auftauchen werden.

Listing 3.14: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

4 Schritt 3

Abbildung 4.1: XXX Die Eingabemaske zeigt im Schritt 1 eine Karte zum Selektieren des Status und ein Eingabefeld für den Titel

In diesem Schritt soll die grundlegende Validierungsfunktion hinzugefügt werden. Maßnahmen, die als abgeschlossen markiert sind, dürfen keine leeren Eingabefelder enthalten und der Maßnahmentitel darf nicht doppelt belegt sein. `Flutter` stellt das Widget `Form` für die Validierung von Eingabefeldern bereit.

4.1 Einfügen des Form-Widgets

Das Widget `Form` ist ein Container, welcher die Validierung für alle Kinderelemente des Typs `FormField` ausführt. Damit es alle Eingabefelder im Formular umgibt, wird es oberhalb des `Stack` eingefügt (Listing 4.1, Z. 161). Das `Form`-Widget muss über einen `key` registriert werden (Z. 162), damit auf die Validierungsfunktionen zurückgegriffen werden kann.

Listing 4.1: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

Die Erstellung des `formKey` findet zu Beginn der `build`-Methode des Eingabeformulars statt (Listing 4.2, Z. 20). Der `GlobalKey` identifiziert ein Element, welches durch ein Widget gebaut wurde, über die gesamte Applikation hinweg. Es erlaubt darüber hinaus auf das `State`-Objekt zuzugreifen, welches mit dem `StatefulWidget` verknüpft ist. Ohne Angabe eines Typparameters kann nur Zugriff auf Funktionen des Typs `State` gewährt werden. Doch die gewünschte Methode `validate` ist nur Teil des Typs `FormState`. Damit das Element, welches über den `GlobalKey` registriert wurde, auch den `FormState` liefert, kann der entsprechende Typparameter `<FormState>` bei der Erstellung des GlobalKey übergeben werden.

Listing 4.2: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

4.2 Validierung des Maßnahmentitels

Das Eingabefeld für den Maßnahmen-Titel ist ein `TextFormField` (Listing 4.3, Z. 88). Es erbt vom Typ `FormField` und wird daher mit dem Vaterelement `Form` verknüpft. Es beinhaltet bereits einen Parameter für die Validierungsfunktion namens `validator` (Z. 93). Die übergebene Funktion erhält im ersten Parameter den für das Textfeld eingetragenen Wert. Die Funktion soll `null` zurückgeben, wenn keine Fehler in der Validierung geschehen sind. In jedem anderen Fall soll der Text zurückgegeben werden, der als Fehlermeldung angezeigt werden soll.

Listing 4.3: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

Sollte der Parameter `null` sein oder aber ein leerer String (Z. 94), so wird die entsprechende Fehlermeldung `'Bitte Text eingeben'` angezeigt (Z. 96). Damit der Benutzer direkt zu dem fehlerhaften Eingabefeld geführt wird, kann ein Objekt der Klasse `FocusNode` verwendet werden. Er wird vor der Konstruktion der Karte erstellt (Z. 84) und dem Parameter `focusNode` des `TextFormField` übergeben (Z. 89). Sollte ein Fehler bei der Validierung gefunden werden, kann mit der Methode `requestFocus` angeordnet werden, den Cursor in das betreffende Feld zu setzen (Z. 95). Das sorgt auch dafür, dass das Eingabefeld in den sichtbaren Bereich gerückt wird.

Sollte das Textfeld nicht leer sein, so soll noch überprüft werden, ob der Maßnahmen-Titel bereits vergeben ist. Über das Model kann die Liste der Maßnahmen angefordert werden (Z. 99). Die Funktion `any` akzeptiert als Argument eine Funktion, die für alle Elemente der Liste ausgeführt wird (Z. 99-102). Wenn die Rückgabe der Funktion auch nur in einem Fall `true` ist, so evaluiert auch `any` mit `true`. Andernfalls ist die Rückgabe `false`. Die anonyme Funktion schließt zunächst den Vergleich mit derselben Maßnahme aus, welche sich gerade in Bearbeitung befindet. Der Vergleich der guid ist dafür ausreichend. Sollte es eine andere Maßnahme geben, welche den gleichen Titel hat (Z. 101-102), so wird Die lokale Variable `massnahmeTitleDoesAlreadyExists` auf `true` gesetzt. Der Benutzer bekommt die entsprechende Fehlermeldung `'Dieser Maßnahmentitel ist bereits vergeben'` zu lesen (Z. 106). Wenn keine der beiden Fallunterscheidungen das `return`-Statement (Z. 96, 106) auslöst, so erfolgt schließlich die Rückgabe von `null`. In dem Kontext der `validator`-Funktion bedeutet die Rückgabe von `null` (Z. 108), dass die Validierung erfolgreich war.

Das `Form`-Widget validiert lediglich Kindelemente vom Typ `FormField`. Dementsprechend wird das Widget `SelectionCard` nicht in die Validierung miteinbezogen. Es erbt nicht von `FormField`. Es wäre möglich, eine weitere Klasse zu erstellen, die von `FormField` erbt und alle Parameter für die Erstellung einer Selektions-Karte wiederverwendet. Doch das würde bedeuten, dass für alle folgenden Schritte jeder weitere Parameter in beiden Konstruktoren der Klassen gepflegt werden müsste. Um der Arbeit leichter folgen zu können, wurde sich für einen anderen, simpleren Weg entschieden: Die Selektionskarte kann ebenso

von einem `FormField` umgeben werden (Listing 4.4, Z. 121-148), welches die Selektionskarte in der `builder`-Funktion erstellt und an den Parametern nichts ändert, außer einen weiteren hinzuzufügen: der Text für die Fehlermeldung (Z. 147). Der erste Parameterder `builder`-Funktion ist das `State`-Objekt das `FormField`. Es enthält die Getter-Methode `errorText`, die bei gegebenenfalls fehlgeschlagener Validierung die zurückgegebene Fehlermeldung enthält.

Listing 4.4: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

Die anonyme Funktion, die als Argument dem Parameter `validator` übergeben wird (Z. 122-132), erstellt eine temporäre Menge, die den Wert des `selectionViewModel` enthält, wenn dieser nicht `null` ist, andernfalls ist sie eine leere Menge (Z. 123-125). Die `validator`-Funktion gibt eine Fehlermeldung zurück, sollte die Menge leer sein (Z. 127-129). Ist die Menge dagegen gefüllt, so gibt sie `null` zurück, um mitzuteilen, dass die Validierung erfolgreich war (Z. 131).

Der `errorText` wird im Konstruktor der Klasse `SelectionCard` übergeben (Listing 4.5, Z. 29). Da er `null` sein darf, ist er mit dem Suffix `?` als Typ mit Null-Zulässigkeit gekennzeichnet (Z. 21).

Listing 4.5: errorText wird der SelectionCard hinzugefügt

Durch Einfügen einer `Column` zwischen der `Card` (Listing 4.6, Z. 53) und dem ListTile (Z. 57) kann die visuelle Repräsentation der Selektionskarte in der Höhe erweitert werden. Sollte der `errorText` gesetzt sein (Z. 65), so erscheint unter dem Titel und dem Untertitel eine entsprechende Fehlermeldung (Z. 66-71).

Listing 4.6: errorText wird ausgegeben

Oberhalb des vorhandenen `FloatingActionButton` wird nun ein weiterer eingefügt, der zum Speichern des Entwurfs mit der Funktion `saveDraftAndGoBackToOverviewScreen` genutzt werden soll (Listing 4.7, Z. 207-216). Der ursprüngliche `FloatingActionButton` speichert nun ausschließlich dann, wenn die Maßnahme als *„in Bearbeitung“* markiert ist oder alle Eingabefelder valide sind. Dazu nutzt er die Hilfsfunktion `inputsAreValidOrNotMarkedFinal` (Z. 222). Ist das der Fall, so folgt die Speicherung der Maßnahme mithilfe der bereits implementierten Funktion `saveRecord` (Z. 223). Diese funktioniert wie in den letzten Schritten, nur dass sie keinen Rückgabewert mehr hat (siehe Listing **??** in Anhang **??** auf Seite **??**). Anschließend wird der Navigator erneut aufgefordert, zum Übersichtsbildschirm zurückzukehren (Z. 224). Sollte es allerdings zur Ausführung des `else`-Blocks führen (Z. 225-227), da die Maßnahme doch als *„abgeschlossen“* markiert und nicht alle Eingabefelder valide waren, so erhält der Benutzer eine Fehlermeldung. Die neu implementierte Hilfsfunktion `showValidationError` wird dafür verwendet (Z. 226).

Listing 4.7: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

Auch der `WillPopScope` erhält die gleiche Fehlerbehandlung (Listing 4.8). Hier wird ebenfalls überprüft, ob die Maßnahme als *„abgeschlossen“* markiert wurde und ob alle Eingabefelder valide sind (Z. 153). Falls ja, wird die Maßnahme direkt gespeichert Und ein Objekt des asynchronen Types `Future` zurückgegeben, welches direkt zu `true` evaluiert (Z. 155). Das führt dazu, dass dem Zurücknavigieren zum Übersichtsbildschirm zugestimmt wird. Sollte allerdings der `else`-Block ausgeführt werden, so erscheint erneut die entsprechende Fehlermeldung (Z. 157) und dieses Mal evaluiert das `Future`-Objekt zu `false`, um die Navigation zu unterbinden `158`.

Listing 4.8: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

Die Funktion `saveDraftAndGoBackToOverviewScreen` funktioniert ähnlich wie die nun ausgetauschte Funktion `saveRecord`. Sie zeigt dem Benutzer an, dass die Maßnahme im Entwurfsmodus gespeichert wird (Z. 23-26), speichert sie im Model ab (Z. 31), und navigiert zur letzten Route zurück (Z. 32), welcher der Übersichtsbildschirm ist. Einer der beiden Unterschiede ist, dass die Maßnahme zuvor umgebaut wird. Unerheblich dessen, welchen letzten Status sie aktuell besitzt, erhält sie den letzten Status `"in Bearbeitung"` (Z. 28-29). Der zweite der beiden Unterschiede ist, dass die Funktion nun keinen Rückgabewert hat, während `saveRecord` einen Wert vom Typ `Future<bool>` zurückgeben musste. Der Grund dafür ist, dass die Funktion nur noch über den Aktionsbutton zum Speichern der Maßnahme im Entwurfsmodus ausgelöst wird. Der FloatingActionButton setzt keinen Rückgabewert der ausgelösten Funktion voraus.

Listing 4.9: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

Die Hilfsfunktion `inputsAreValidOrNotMarkedFinal` überprüft zunächst, ob der letzte Status ein anderer ist als *„abgeschlossen“* (Listing 4.10, Z. 71). Da in diesem Fall keine weiteren Überprüfungen notwendig sind, gibt die Funktion direkt `true` zurück (Z. 73). Andernfalls validiert das Formular die Eingabefelder (Z. 76). Dazu muss das Element vom Typ `Form` in den Vaterelementen gefunden werden. Genauer gesagt wird dessen `State`-Objekt benötigt. Der Zugriff auf das Element ist einfach, da es über einen `GlobalKey` registriert wurde. Über `formKey.currentState` kann das `State`-Objekt des Elements abgerufen werden (Z. 76). Die Funktion `validate()` führt dann alle Funktionen aus, die jeweils als Argument dem Parameter `validator` aller Kindelemente des Typs `FormField` übergeben wurden. Sollten alle `validator`-Funktionen `null` zurückgegeben haben – was bedeutet, dass keine Fehler bei der Validierung geschehen sind – so erfolgt die Rückgabe von `true` (Z. 77). Anderenfalls bleibt nur die Rückgabe von `false` übrig (Z. 80).

Sollte es zu einem Fehler kommen, so zeigt die Hilfsfunktion `showValidationError` dem Benutzer die entsprechende Fehlermeldung an (Listing 4.11). Sie bietet ihm darüber hin-

Listing 4.10: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

aus an, über einen Button die Maßnahme direkt als Entwurf zu speichern. Das ist möglich, da die `SnackBar` (Z. 45) nicht nur die Anzeige von gewöhnlichem Text erlaubt, sondern von jedem beliebigen Widget. Zunächst kommt dazu das Widget `Row` zum Einsatz (Z. 46). Ähnlich wie das Widget `Column` erlaubt es Kinderelemente in einer Reihe aufzulisten. Im Gegensatz zur `Column` allerdings nun horizontal statt vertikal. Als letztes Element der `Row` wird der `ElevatedButton` verwendet. Genauso wie bereits der `FloatingActionButton` zum Speichern der Maßnahme im Entwurfsmodus verwendet nun auch dieser `ElevatedButton` die Funktion `saveDraftAndGoBackToOverviewScreen` (Z. 52).

Listing 4.11: Die Maßnahmencharakteristika Selektionskarten werden ergänzt

5 Diskussion

5.1 Reevaluation des Zustandsmanagements

Während der Implementierung wurde eine passende Vorgehensweise gesucht, um den Zustand der Applikation zu verwalten und damit die Aktualisierung der Oberfläche auszulösen. Für simple Applikationen empfiehlt Google den integrierten Mechanismus der *„StatefulWidgets“* und deren Methode *„setState“* zu verwenden[1]. Doch durch die hohe Anzahl der Oberflächenelemente in der finalen Applikation ist diese Vorgehensweise nicht empfehlenswert. Sie setzt das Aktualisieren gesamter Widgets bei Anpassung des Zustandes voraus, was für die Laufzeitgeschwindigkeit die intensivste Belastung darstellt. Stattdessen wurde versucht, einem Mechanismus zu verwenden, der es erlaubt, nur Teile der Oberfläche neuzuzeichen, die wirklich eine Aktualisierung benötigen.

Zu diesem Zweck empfiehlt Google das Nutzen des Pakets *„provider“* der Flutter Community[2]. Dieser Ansatz wurde in der Implementierung ursprünglich verwendet. Das Paket hat den Nachteil, dass für jeden Zustand, der die Aktualisierung eines Teils der Oberfläche bewirken soll, eine neue Klasse erstellt werden muss, die von `ChangeNotifier` erbt. Eine Möglichkeit ist, dass jede dieser Klassen den nötigen Boilerplate-Quellcode enthält, welcher die Oberfläche über die Methode `notifyListeners` benachrichtigt. Eine andere Möglichkeit ist es, für den gleichen Datentyp den benötigten BoilerplateCode in einer eigenen Basisklasse auszulagern und dann von dieser Klasse zu erben wie in Listing zu sehen. `ChoiceChangeNotifier` verwaltet den internen privaten Zustand `_choices` (Z. 3) über die öffentlichen Schnittstellen zum Lesen (Z. 4) und Schreiben (Z. 6-9). Bei Aktualisierung des Wertes erhalten alle Listener eine Benachrichtigung (Z. 8). `LetzterStatusViewModel` erbt dieses Verhalten, doch hat die Klasse darüber hinaus keine Implementierung.

Anschließend muss jeder `ChangeNotifier` als ein `ChangeNotifierProvider` registriert werden (Listing **??**, Z. 7). Der `MultiProvider` kann genutzt werden, um mehrere Provider in einer Liste zu übergeben. Dort werden auch andere Services wie etwa `MassnahmenFormViewModel` (Z. 3) und `MassnahmenModel` (Z. 6) hinterlegt.

Dann ist der `ChangeNotifier` in dem Widget, welches den Parameter `child` übergeben

[1] Vgl. 3.
[2] Vgl. 6.

wird und darüber hinaus allen Kindern-Elementen dieses Widgets verfügbar. Über einen `Consumer` kann in der Oberfläche auf Änderungen des `ChangeNotifier` reagiert werden (Listing **??**).

Doch diese Vorgehensweise bietet im Vergleich zu den von Flutter mitgelieferten *„Widgets“* keine Vorteile. Das Äquivalent zum `Consumer` ist das mitgelieferten Widget `StreamBuilder`, welcher mit jeder Art von *„Stream“* verwendet werden kann.

Damit unterstützt er ein breiteres Spektrum von Einsatzmöglichkeiten. Beispielsweise kann ein transformierter *„Stream“* übergeben werden, wie im Kapitel **??** gezeigt.

Die einzige fehlende Komponente dafür ist ein *„Stream“*, der den zuletzt übermittelten Wert speichert und den neuen StreamBuilder Elementen übermittelt. Deshalb wurde sich für das Package *„rx.dart“* entschieden, welches genau dieses Verhalten mit dem *„BehaviorSubject“* abdeckt. Durch dessen Verwendung kann sowohl auf das Registrieren des `ChangeNotifierProvider` verzichtet werden und es muss keine weitere Klasse für die einzelnen beobachtbaren Objekte erstellt werden.

Auch der `MultiProvider` erscheint auf den ersten Blick als sehr nützlich. Doch das Anbieten der Services durch ein eigens implementiertes `InheritedWidget` erlaubt einen Zugriff, der kürzer und expliziter ist. Durch die Umstellung konnte der Zugriff auf das ViewModel mithilfe des Ausdrucks `Provider.of<MassnahmenFormViewModel>(context, listen: false)` durch `AppState.of(context).viewModel` ersetzt werden.

Eine ganz ähnliche, wenn auch deutlich kompliziertere Variante dieser Vorgehensweise, wurde auf der Google I/O 2018 von Filip Hracek und Matt Sulliivan vorgestellt. Doch anstatt lediglich das BehaviorSubject für das ViewModel zu verwenden, sorgte die Präsentation durch den zusätzlichen – jedoch überflüssigen – Einsatz zwei weiterer Stream-Klassen für schweres Verständnis (Listing **??**)[3].

Obwohl das `BehaviorSubject` die Funktionsweise des `ViewModels` bereits löst, wurde ein Objekt des Typs `Sink` verwendet, um Ereignisse von dem View an das `ViewModel` senden zu können (Z. 4). `StreamController` verwendet. Ein Sind implementiert jedoch ausschließlich Methoden zum Hinzufügen von Ereignissen Punkt um den Stream zu lesen, wird ein dazugehöriger `StreamController` erstellt (Z. 6). Er hat im Gegensatz zum `Sink` auch lesenden Zugriff auf die Ereignisse. Sobald ein Ereignis eintrifft, so wird es dem `Model _cart` hinzugefügt (Z. 17). Es existiert außerdem ein weiterer `Stream itemCount` (Z. 8) welcher lediglich die transitive Eigenschaft der Anzahl der hinzugefügten Elemente darstellt `18`. Er nutzt das BehaviorSubject `10`, verwendet allerdings keine der bedeutsamen Methoden. Es könnte genauso gut durch einen weiteren `StreamController` ersetzt werden.

[3] 4, TC: 27:37.

Der gesamte Quellcode kann stark vereinfacht werden (Listing **??**).

Durch Einsatz der für das `BehaviorSubject` einzigartigen Getter-Methode `value` kann dem *„Stream“* ein neues Objekt hinzugefügt werden, wodurch er gleichzeitig ein neues Ereignis sendet (Z. 4). Die Zuweisung hat zwar ansonsten keinen Zweck, da das Objekt vor und nach der Zuweisung das gleiche ist, denn es handelt sich um einen Referenztyp und nicht um einen Werttyp. Die Erstellung weiterer `StreamController` zum Senden der transitiven Eigenschaft `itemCount` ist nicht nötig. Sendet das `BehaviorSubject` `_cart` ein neues Event (Z. 4), so wird auch die Methode `map` ausgelöst und ein transformiertes Eigenschaft gesendet (Z. 6).

Durch eine Anleitung mit diesem Ergebnis könnten gegebenenfalls weitere Entwickler das *„BloC-Pattern“* dem Paket *„provider“* vorziehen.

5.2 Anzeige von fehlerhaften Teilkomponenten der Bedingungen von deaktivierten Auswahloptionen

Einen Wunschkriterium für die Formularapplikation war es, bei der Auswahl von deaktivierten Optionen einen Hinweise zu erhalten, warum diese deaktiviert ist.

In Kapitel **??** ist die Umsetzung der Deaktivierung von Optionen beschrieben. Eine Funktion zur Überprüfung der Bedingung einer Optionen wird der Option bei dessen Erstellung im Konstruktor übergeben. Sie wird bei Überprüfung der Kompatibilität der Auswahloption mit den restlichen im Formular ausgewählten Optionen ausgeführt. Die Konjunktion, Disjunktion und Negation wird mit den Operatoren für das logische Und und das logische Oder sowie das logische Nicht umgesetzt. Doch auf diese Art und Weise ist es nicht möglich, herauszufinden, welche der einzelnen Abfragen zu einem Fehler führte. Auf den Inhalt der Funktion kann zur Laufzeit nicht zugegriffen werden. Die Einzelkomponenten der Bedingung sind damit also nicht bekannt. Es ist daher nur möglich, auf die Komponenten der Bedingung zuzugreifen, wenn die gesamte Bedingung als eine Datenstruktur abgelegt ist. Diese Datenstruktur muss die Konjunktion, Disjunktion und Negation unterstützen.

Die Konzeption und Implementierung einer solchen Datenstruktur und des dazugehörige Algorithmus zur Identifizierung der inkompatiblen Komponenten bedarf einer intensiven wissenschaftlichen Recherche und Ausarbeitung. Als Wunschkriterien steht diese Funktion somit nicht im Kosten-Nutzen-Verhältnis, weshalb sich gegen die Ausarbeitung in dieser wissenschaftlichen Arbeit entschieden wurde.

6 Schlussfolgerung-und-Ausblick

In dieser Arbeit wurde gezeigt, dass das Hauptproblem der Formular Anwendung mit Hilfe von Funktionsobjekten und logischen Operatoren gelöst werden konnte.

Auch die Aktualisierung der sich tatsächlich ändernden Elemente in der Oberfläche wurde umgesetzt. In jedem Fall war die deklarative und reaktive Programmierung der Oberfläche eine Erleichterung und Voraussetzung dafür. Die Implementierung hätte auch mit *„React Native“* stattfinden können, da es ebenso einen deklaratives Oberflächen Framework ist. Die Stream Transformationen aus der Kern-Bibliothek von Dart und aus *„RxDart“* haben ihre Äquivalente in der Bibliothek *„RxJS“*. `https://www.learnrxjs.io/learn-rxjs/operators/filterir`

Die Wahl von Flutter für die Entwicklung war trotzdessen aus den folgenden Gründen eine gute Entscheidung:

Die gesichteten Anleitungen für die Einarbeitung in das automatisierte Testen ebneten eine vollumfängliche und zielgerichtete Einarbeitung. Keine weiteren Quellen von Drittanbietern mussten genutzt werden, um die im Rahmen dieser Masterarbeit entstandenen *„Unit-“* und *„Integrationstests“* zu entwickeln. Lediglich die initialen Probleme bei der Generierung von Mocks im Ordner für die Integrationstest stoppten die Entwicklung für einen Moment.

Hätte die Umsetzung in the React Native stattgefunden, so hätte die Einarbeitung in die Entwicklung von *„Unit-“* und *„Integrationstest“* eventuell einen höheren Aufwand bedeutet, da die Dokumentation auf den unterschiedlichen Web-Portal in der Drittanbieter verstreut ist.

Auch die Rezepte im Flutter-Kochbuch boten die benötigten Funktionalitäten wie die Formularvalidierung, die Navigation über Routen

Allerdings fällt die Wahl für das angemessene Zustandsmanagement für einen Anfänger in der deklarativen Programmierung nicht leicht. Die Empfehlung von Google das Paket *„Provider“* zu nutzen führte zu Schwierigkeiten, wie in Sektion 5.1 beschrieben. Das ursprünglich von Google beworbener *„Bloc-pattern“*, welches bei der *„Flutter“*-Community weniger beliebt ist, war am Ende die angemessene Technologie. Es fehlte aber die Dokumentation darüber, wie es richtig eingesetzt wird. Die Erkenntnisse, die im Rahmen dieser Masterarbeit bezüglich der reibungslosen Implementierung des Zustandsmanagements mit

„*RxDart*“ gesammelt wurden, sollen in Zukunft mit der „*Flutter*“-Community geteilt werden.

Das Wunsch Kriterium, den Benutzer auch die fehlerhafte Auswahl anzuzeigen, die verhindert, eine spezielle Option zu wählen, konnte nicht umgesetzt werden. Vor dem Hintergrund der für diese Arbeit festgelegten Ziele und der Komplexität des Problems wurde sich gegen die Konzeption und Implementierung entschieden. An den bisherigen Erkenntnissen soll jedoch weiter gearbeitet werden. Nutzerumfragen sollen darüber hinaus zeigen, in welcher Art und Weise eine solche Fehlermeldung präsentiert werden könnte.

Eidesstattliche Erklärung

Ich erkläre, dass ich die vorliegende Masterarbeit *„Entwicklung einer Formularanwendung mit Kompatibilitätsvalidierung der Einfach- und Mehrfachauswahl-Eingabefelder“* selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und dass ich alle Stellen, die ich wörtlich oder sinngemäß aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe. Die Arbeit hat bisher in gleicher oder ähnlicher Form oder auszugsweise noch keiner Prüfungsbehörde vorgelegen.

Ich versichere, dass die eingereichte schriftliche Fassung der auf dem beigefügten Medium gespeicherten Fassung entspricht.

Wernigerode, den 01.09.2021

Alexander Johr