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 Formularanwendung 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 2 evaluiert die Kandidaten der Frontend-Technologien, 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 Frontend-Technologie für die Entwicklung der Formularanwendung *Flutter* gewählt wurde, beschäftigt sich Kapitel 3 mit den Grundlagen des Frameworks und der zugrunde liegenden Programmiersprache *Dart*.

Die Kapitel 4 bis 10 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 4 stellt die grundlegende Struktur der Anwendung her. Kapitel 5 fügt Hilfsmethoden hinzu, welche das Hinzufügen weiterer Formularfelder in den folgenden Schritten vereinfachen wird.

In Kapitel 6 erhält die Anwendung die grundlegende Funktion, Felder zu validieren. Kapitel 7 erweitert die Validierung schließlich um die Bedingungen der Auswahloptionen. Als Konsequenz werden alle Formularfelder neu gezeichnet, sollte der Benutzer eine beliebi-

ge 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 8 hinzugefügt.

Kapitel 9 ergänzt die Möglichkeit, Mehrfachauswahlfelder zu verwenden. Kapitel 10 sorgt dafür, dass auch benutzerdefinierte Bedingungen für die Auswahlfelder hinterlegt werden können.

Kapitel 11 setzt sich mit den Erkenntnissen auseinander, die während der Entwicklung der Anwendung gesammelt wurden. Kapitel 12 bewertet die Erkenntnisse, ergänzt sie um einen Ausblick und vergleicht die Ergebnisse der Entwicklung mit den Anforderungen.

2 Technologie Auswahl

Die folgenden drei Kapitel behandeln die Auswahl der Frontend-Technologie für die Umsetzung der Formularanwendung. Dazu werden im ersten Schritt die dafür in Frage kommenden Technologien identifiziert. Anschließend wird der Trend der Popularität dieser Technologien miteinander verglichen. Die daraus resultierenden Kandidaten sollen dann detaillierter untersucht werden. In Hinblick auf die Anforderungen an die Formularanwendung soll dabei die angemessenste Frontend-Technologie ausgewählt werden.

2.1 Trendanalyse

Zwei Quellen wurden für die Analyse der Technologie-Trends ausgewählt: die Ergebnisse der jährlichen Stack Overflow-Umfragen und das Such-Interesse von Google Trends.

2.1.1 Stack Overflow Umfrage

Die Internet-Plattform Stack Overflow richtet sich an Softwareentwickler und bietet ihren Nutzern die Möglichkeiten, Fragen zu stellen, Antworten einzustellen und Antworten anderer Nutzer auf- und abzuwerten.

Besonders für Fehlermeldungen, die häufig während der Softwareentwicklung auftreten, findet man auf dieser Plattform rasch die Erklärung und den Lösungsvorschlag gleich mit. So lässt sich auch die Herkunft des Domain-Namens herleiten:

> We named it Stack Overflow, after a common type of bug that causes software to crash – plus, the domain name stackoverflow.com happened to be available.
> — Joel Spolsky, Mitgründer von Stack Overflow [1]

Aufgrund des Erfolgsrezepts von Stack Overflow ist die Plattform kaum einem Softwareentwickler unbekannt. Dementsprechend nehmen auch jährlich tausende Entwickler an den von Stack Overflow herausgegebenen Umfragen teil. Seit 2013 beinhalten die Umfragen

[1] Spolsky, *How Hard Could It Be?: The Unproven Path*

auch die Angabe der aktuell genutzten und in Zukunft gewünschten Frontend-Technologien. *Stack Overflow* erstellt aus diesen gesammelten Daten Auswertungen und Übersichten und die zugrundeliegenden Daten werden ebenfalls veröffentlicht. [2]

Um den Trend der Beliebtheit der Frontend-Technologien aufzuzeigen, wurde ein Jupyter Notebook erstellt. Es transformiert die Daten in ein einheitliches Format, da die Umfrageergebnisse von Jahr zu Jahr in einer unterschiedlichen Struktur abgelegt wurden. Anschließend erstellt es Diagramme, die im Folgenden analysiert werden. Das Jupyter Notebook ist im Anhang zu finden.

2.1.2 Google Trends

Suchanfragen, die über die Suchmaschine Google abgesetzt werden, lassen sich über den Dienst Google Trends als Trenddiagramm visualisieren. Die Ergebnisse werden normalisiert, um das relative Such-Interesse abzubilden und die Ergebnisse auf einer Skala von 0 bis 100 darstellen zu können. [3]

> Google Trends ist keine wissenschaftliche Umfrage und sollte nicht mit Umfragedaten verwechselt werden. Es spiegelt lediglich das Suchinteresse an bestimmten Themen wider. [4]

Genau aus diesem Grund wird Google Trends im Folgenden lediglich zum Abgleich der Ergebnisse der Stack Overflow Umfrage eingesetzt.

2.1.3 Frameworks mit geringer Relevanz

NativeScript, Sencha (bzw. Sencha Touch) und Appcelerator spielen in den Umfrageergebnissen eine untergeordnete Rolle. Dies ist in den aufsummierten Stimmen von 2013 bis 2020 für alle in der Umfrage auftauchenden Frontend-Technologien zu sehen (Abb. **??**).

Abbildung 2.1: Summe der Stimmen der Stack Overflow Umfrage von 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: `Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb`, Daten-Quelle: `FEHLT!`

Auch das Suchinteresse auf Google ist für diese Frameworks äußerst gering. In Abbildung 2.2 werden NativeScript, Sencha, Appcelerator und auch Adobe PhoneGap mit Apache Cordova für das relative Suchinteresse verglichen.

[2] Stack Exchange, Inc., *Stack Overflow Insights - Developer Hiring, Marketing, and User Research*
[3] Vgl. Google LLC, *Häufig gestellte Fragen zu Google Trends-Daten - Google Trends-Hilfe*
[4] Google LLC, *Häufig gestellte Fragen zu Google Trends-Daten - Google Trends-Hilfe*

Abbildung 2.2: Suchinteresse der Frameworks mit geringer Relevanz, Quelle: Eigene Abbildung, Notebook: Charts/GoogleTrends/GoogleTrends.ipynb, Daten-Quelle: Google Trends[5]

Verwandte Technologien zu Apache Cordova

Das Ionic Framework taucht in den Ergebnissen der Stack Overflow Umfragen nicht auf. Ein Grund dafür könnte sein, dass es auf Apache Cordova aufbaut[6], welches bereits in den Ergebnissen vorkommt. Adobe PhoneGap taucht zwar in den Ergebnissen von 2013 mit 1043 Stimmen auf (Siehe Abbildung 2.3), verliert jedoch in den Folgejahren mit weniger als 10 Stimmen abrubt an Relevanz. Das stimmt nicht mit dem Suchinteresse auf Google überein, da Adobe PhoneGap dort erst ab 2014 anfängt, langsam an Relevanz zu verlieren, wie in Abbildung 2.2 zu sehen ist. 2013 existierte PhoneGap noch als extra Mehrfachauswahlfeld in den Daten, während es ab 2014 nur noch in dem Feld für die sonstigen Freitext Angaben auftaucht [7]. Auch Adobe PhoneGap baut auf Apache Cordova auf[8]. Für diese Auswertung spielen diese verwandten Technologien eine untergeordnete Rolle, da sie auch in den Google Trends weit hinter Apache Cordova zurückbleiben (Abb. 2.2).

Abbildung 2.3: Stimmen für Cordova und PhoneGap 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb, Daten-Quelle: FEHLT!

Am Beispiel von Adobe PhoneGap wird deutlich, wie wichtig es ist, auf eine Technologie zu setzen, die weit verbreitet ist. Im schlimmsten Fall wird die Technologie sogar vom Betreiber aufgrund zu geringer Nutzung komplett eingestellt, wie es bei PhoneGap bereits geschehen ist. Adobe gab am 11. August 2020 bekannt, dass die Entwicklung an PhoneGap eingestellt wird und empfiehlt die Migration hin zu Apache Cordova.[9]

2.1.4 Frameworks mit sinkender Relevanz

Die Technologien Xamarin und Cordova zeigen bereits einen abfallenden Trend, wie in Abbildung 2.4 ersichtlich ist. Im Fall von Xamarin gibt es immerhin mehr Entwickler, die sich wünschen, mit dem Framework zu arbeiten, als Entwickler, die tatsächlich mit Xamarin arbeiten. Cordova scheint in diesem Hinblick dagegen eher unbeliebt: Es gibt mehr Entwickler, die mit Cordova arbeiten, als tatsächlich damit arbeiten wollen.

Abbildung 2.4: Stimmen für Xamarin und Cordova 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb, Daten-Quelle: FEHLT!

In Abbildung 2.5 ist noch einmal zu sehen, dass Google Trends die Erkenntnisse aus der

[6] Lynch, *The Last Word on Cordova and PhoneGap*
[7] Vgl. Stack Exchange, Inc., *Stack Overflow Insights - Developer Hiring, Marketing, and User Research*
[8] Vgl. Adobe Inc., *FAQ | PhoneGap Docs*
[9] Vgl. Adobe Inc., *Update for Customers Using PhoneGap and PhoneGap Build*

Stack Overflow Umfrage reflektiert; und es wird auch sichtbar, welche beiden Technologien möglicherweise der Grund für den Rückgang von Xamarin und Cordova sind.

Abbildung 2.5: Suchinteresse sinkende und steigende Relevanz, Quelle: Eigene Abbildung, Notebook: `Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb`, Daten-Quelle: FEHLT!

2.1.5 Frameworks mit steigender Relevanz

Besser ist es, auf Technologien zu setzen, die noch einen steigenden Trend der Verbreitung und Beliebtheit zeigen. In Abbildung 2.6 wird sichtbar, dass es sich dabei um *Flutter* und – immerhin im Hinblick auf die Verbreitung – auch um *React Native* handelt. Ungünstigerweise wird *React Native* in der Stack Overflow Umfrage erst seit 2018 als tatsächliches Framework abgefragt. Vorher erschien lediglich das Framework React, welches sich nicht für den Vergleich der Cross-Plattform-Frameworks eignet, da es sich um ein reines Web-Framework handelt. Doch auch die Ergebnisse von Google Trends zeigen einen ähnlichen Verlauf für die Jahre 2019 und 2020 (Abb. 2.5).

Abbildung 2.6: Stimmen für *React Native* und *Flutter* von 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: `Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb`, Daten-Quelle: FEHLT!

Im Vergleich des Jahres 2019 mit 2020 wird sichtbar, dass die Zahl der Entwickler, die sich wünschen, mit *React Native* zu arbeiten, gesunken ist. Dennoch ist die Anzahl der Entwickler, die mit *React Native* arbeiten möchten noch weit höher, als die der Entwickler, die tatsächlich mit *React Native* arbeiten.

Es ist möglich, dass der abfallende Trend daran liegt, dass die Zahl der Entwickler, die mit *Flutter* arbeiten möchten im selben Jahr gestiegen ist. React Native hat im Vergleich zu *Flutter* jedoch noch immer mehr aktive Entwickler und die Tendenz ist steigend. Doch die Anzahl der aktiven *Flutter*-Entwickler zeigt einen noch stärker steigenden Trend. So könnte es sein, dass die Zahl der *Flutter*-Entwickler die der *React Native*-Entwickler in einem der nächsten Jahre überholt. Im Such-Interesse hat sich diese Entwicklung bereits vollzogen (Abb. 2.5).

Nichtsdestotrotz scheinen beide Technologien als Kandidaten für einen detaillierteren Vergleich für dieses Projekt in Frage zu kommen. Im nächsten Kapitel soll evaluiert werden, welches Framework für die Entwicklung der Formularanwendung angemessener ist.

2.2 Vergleich von *React Native* und *Flutter*

2.2.1 Vergleich zweier minimaler Beispiele für Formulare und Validierung

`verweise auf Listings Anhang, erstelle Tabelle mit Zusammenfassung`

Es soll eine Formularanwendung mit komplexer Validierung im Rahmen dieser These erstellt werden. Es ist durchaus sinnvoll, die beiden Technologien anhand von Beispielanwendungen, welche Formulare und die Validierung dieser beinhalten, zu vergleichen. Deshalb soll nachfolgend jeweils eine solche Beispielanwendung der jeweiligen Technologie gefunden werden. Die Anwendungen werden sich stark voneinander unterscheiden, weshalb sie im nächsten Schritt vereinfacht und aneinander angeglichen werden. Anschließend wird ersichtlich werden, nach welchen Kriterien sich die Technologien im Hinblick auf die Entwicklung der Formularanwendung vergleichen lassen.

React Native

React native stellt nur eine vergleichsweise geringe Anzahl von eigenen Komponenten zur Verfügung und zu diesen gehören keine, welche die Validierung von Formularen ermöglichen. Doch die im react.js Raum sehr bekannten Bibliotheken Formic, Redux Forms und React Hook Form sind alle drei kompatibel mit *React Native*.[10][11][12]

Für die Formularanwendung ist die Validierung komplexer Bedingungen nötig. Die Formular-Validierungs-Bibliotheken bieten in der Regel Funktionen an, welche überprüfen, ob ein Feld gefüllt ist oder der Inhalt einem speziellen Muster entspricht – wie etwa einem regulären Ausdruck. Doch solche mitgelieferten Validierungs-Funktionen reichen nicht aus, um die Komplexität der Bedingungen abzubilden. Stattdessen müssen benutzerdefinierte Funktionen zum Einsatz kommen.

Keiner der drei oben genannten Validierungs-Bibliotheken ist in dieser Hinsicht limitiert. Sie alle bieten die Möglichkeit, eine JavaScript Funktion für die Validierung zu übergeben. Diese Funktion gibt einen Wahrheitswert zurück – wahr, wenn das Feld oder die Felder valide sind, falsch, falls nicht. In *React Hook Form* ist es die Funktion *register*, die ein Parameter-Objekt namens *RegisterOptions* erhält. Der Eigenschaft *validate* dieses Objekts kann eine JavaScript-Funktion für die Validierung übergeben werden.[13] In Redux Form ist es die Initialisierungs-Funktion reduxForm, die ein Konfigurations-Objekt mit dem Namen config erhält, in welchem die Eigenschaft ebenfalls validate heißt.[14] Auch in Formic ist der

[10] Vgl. *React Native | Formik Docs*.
[11] Vgl. *Does redux-form work with React Native?*
[12] Vgl. *React Native | React Hook Form - Get Started*.
[13] Vgl. *register | React Hook Form - API*
[14] Vgl. *reduxForm | Redux Form - API*

Bezeichner validate, und ist als Attribut in der Formic Komponente zu finden.[15]

Es ist also absehbar, dass die Formularanwendung in *React Native* entwickelt werden kann. Die nötigen Funktionen werden von den Bibliotheken bereitgestellt. Einziger Nachteil hierbei ist, dass es sich um Drittanbieter Bibliotheken handelt, welche im Verlauf der Zeit an Beliebtheit gewinnen und verlieren können. Möglicherweise geht die Beliebtheit einer der Bibliotheken mit der Zeit zurück, weshalb es weniger Kontributionen wie etwa neue Funktionalitäten oder Fehlerbehebungen, sowie Fragen und Antworten und Anleitungen zu diesen Bibliotheken geben wird, da die Entwickler sich für andere Bibliotheken entscheiden. Die Wahl der Bibliothek kann also schwerwiegende Folgen wie Mangel an Dokumentation oder Limitationen im Vergleich zu anderen Bibliotheken mit sich bringen. Eine Migration von der einen Bibliothek zu einer anderen könnte in Zukunft notwendig werden, wenn diese Limitationen während der Entwicklung auffallen. Aus dem Grund ist es in der Regel von Vorteil, wenn solche Funktionalitäten bereits im Kern der Frontend-Technologie integriert sind. Der Fall, dass die Kernkomponenten an Relevanz verlieren und empfohlen wird, auf externe Bibliotheken zuzugreifen, ist zwar nicht ausgeschlossen, geschieht aber im Wesentlichen seltener.

Flutter

Die *Flutter*-Dokumentation stellt in ihrer *cookbook* Sektion ein Beispiel einer minimalistischen Formularanwendung mit Validierung bereit.[16] Das Rezept ist Teil einer Serie von insgesamt fünf Anleitungen, welche Formulare in *Flutter* behandeln.[17]

`Auf Listing im Anhang verweisen`

2.2.2 Automatisiertes Testen

Automatisierte Tests in *React Native*

Die *React Native*-Dokumentation führt genau eine Seite mit einem Überblick über die unterschiedlichen Testarten. Dabei wird das Konzept von Unit Tests, Mocking, Integrations Tests, Komponenten Tests und Snapshot Tests kurz erläutert, jedoch ohne ein Beispiel zu geben oder zu verlinken. Vier Quellcodeschnipsel sind auf der Seite zu finden: Ein Schnipsel zeigt den minimalen Aufbau eines Tests; zwei weitere Schnipsel veranschaulichen beispielhaft, wie Nutzerinteraktionen getestet werden können. Letzteres zeigt die textuelle Repräsentation der Ausgabe einer Komponente, die für einen Snapshottest verwendet wird.

[15] Vgl. *<Formik /> | Formik Docs API*
[16] Vgl. Google LLC, *Build a form with validation*
[17] Vgl. Google LLC, *Forms | Flutter Docs Cookbook*

Weiterhin wird auf die Jest API Dokumentation verwiesen, sowie auf ein Beispiel für einen Snapshot Test in der Jest Dokumentation.[I]

Um die notwendigen Anleitungen für das Erstellen der jeweiligen Tests ausfindig zu machen, ist es notwendig, die Dokumentation von *React Native* zu verlassen.

Die Dokumentation von Jest enthält mehr Details zum Einsatz der Testbibliothek, welche für mehrere Frontend-Frameworks kompatibel ist, die auf JavaScript basieren[II]. Somit muss zum Erstellen der Unit-Tests immerhin nur dieses Framework studiert werden.

Zum Entwickeln von Tests für *React Native*-Komponenten wird unter anderem auf die Bibliothek *React Native Testing Library* verwiesen. Anders als der Name vermuten lässt, handelt es sich nicht um eine von *React Native* bereitgestellte Bibliothek. Im Unterschied zur React Testing Library, von der sie inspiriert ist, läuft sie ebenso wie *React Native* selbst nicht in einer Browser-Umgebung.[18] Herausgegeben wird die *React Native Testing Library* vom Drittanbieter Callstack – einem Partner im *React Native*-Ökosystem.[19]

Sie verwendet im Hintergrund den React Test Renderer[III], welcher wiederum vom React Team angeboten wird und auch zum Testen von react.js Anwendungen geeignet ist. Der React Test Renderer wird ebenfalls empfohlen, um Komponententests zu kreieren, die keine *React Native* spezifischen Funktionalitäten nutzen.

Um Integrationstests zu entwickeln – welche die Applikation auf einem physischen Gerät oder auf einem Emulator testen – wird auf zwei weitere Drittanbieter-Bibliotheken verlinkt: Appium[IV] und Detox[V]. Es wird darauf hingewiesen, dass Detox speziell für die Entwicklung von *React Native*-Integrationstests entwickelt wurde. Appium wird lediglich als ein weiteres bekanntes Werkzeug erwähnt.

Es lässt sich damit zusammenfassen, dass der Aufwand der Einarbeitung für automatisiertes Testen in *React Native* vergleichsweise hoch ist. Die Dokumentation ist auf die Seiten der jeweiligen Anbieter verteilt. Der Entwickler muss sich den Überblick selbst verschaffen und zusätzlich die für das Framework *React Native* relevanten Inhalte identifizieren. Notwendig ist auch das Erlernen von mehreren APIs um alle Testarten abzudecken. Für einen Anfänger kommt erschwerend hinzu, dass eine Entscheidung für die eine oder andere Bibliothek notwendig wird. Um diese Entscheidung treffen zu können, ist eine Auseinandersetzung mit den Vor- und Nachteilen der Technologien im Vorfeld vom Entwickler zu leisten.

[18] Vgl. Borenkraout, *Native Testing Library Introduction | Testing Library Docs*
[19] Vgl. Facebook Inc., *The React Native Ecosystem*

[I] https://jestjs.io/docs/snapshot-testing
[II] https://jestjs.io/docs/getting-started
[III] https://reactjs.org/docs/test-renderer.html
[IV] http://appium.io/
[V] https://github.com/wix/detox/

Automatisierte Tests in *Flutter*

Die *Flutter*-Dokumentation erklärt sehr umfangreich auf 11 Unterseiten die unterschiedlichen Testarten mit Quellcodebeispielen und verlinkt für jede Testart eine bis mehrere detaillierte Schritt-für-Schritt-Anleitungen, wie ein solcher Test erstellt wird.

Eine Seite erklärt den Unterschied zwischen Unit-Tests, Widget-Tests und Integrationstests[VI]. Eine weitere Seite erklärt Integrationstests detaillierter[VII].

Ein sogenanntes Codelab führt durch die Erstellung einer minimalistischen App und der anschließenden Implementierung von zwei Unit-, fünf Widget- und zwei Integrationstests für diese App[VIII].

Im sogenannten Kochbuch tauchen folgende Rezepte auf:

- 2 Rezepte für Unit Tests
  - eine grundlegende Anleitung zum Erstellen von Unit-Tests [IX]
  - Eine weitere Anleitung zum Nutzen von Mocks in Unit Test mithilfe der Bibliothek mockito [X]
- 3 Rezepte für Widget Tests
  - Eine grundlegende Anleitung zum Erstellen von Widget Tests [XI]
  - Ein Rezept mit detaillierteren Beispielen zum Finden von Widgets zur Laufzeit eines Widget Tests [XII]
  - Ein Rezept zum Testen vom Nutzerverhalten wie dem Tab, dem Drag und dem Eingeben von Text [XIII]
- 3 Rezepte für Integrationstests
  - Eine grundlegende Anleitung zum Erstellen eines Integrationstests [XIV]

[VI] https://flutter.dev/docs/testing
[VII] https://flutter.dev/docs/testing/integration-tests
[VIII] https://codelabs.developers.google.com/codelabs/flutter-app-testing
[IX] https://flutter.dev/docs/cookbook/testing/unit/introduction
[X] https://flutter.dev/docs/cookbook/testing/unit/mocking
[XI] https://flutter.dev/docs/cookbook/testing/widget/introduction
[XII] https://flutter.dev/docs/cookbook/testing/widget/finders
[XIII] https://flutter.dev/docs/cookbook/testing/widget/tap-drag
[XIV] https://flutter.dev/docs/cookbook/testing/integration/introduction

- eine Anleitung zum Simulieren des Scrollens in der Anwendung während der Laufzeit eines Integrationstests [XV]

- eine Anleitung zum Performance Profiling [XVI]

2.3 Fazit und Begründung der Auswahl

`Zusammenfassung als Kapitel mit Tabelle und Wahl` `Tabelle mit auflisting der Pros und`

Zusammengefasst: Der Aufwand der Einarbeitung in das Testen in *Flutter* ist gering. Alle Werkzeuge werden vom *Dart*- und *Flutter*-Team bereitgestellt. Die Dokumentation ist umfangreich, folgt jedoch einem roten Faden. Eine Übersichtsseite fasst die Kerninformationen zusammen und verweist auf die jeweiligen Seiten für detailliertere Informationen und Übungen.

[XV] https://flutter.dev/docs/cookbook/testing/integration/scrolling
[XVI] https://flutter.dev/docs/cookbook/testing/integration/profiling

3 Grundlagen

Für die Formular Anwendung wurde die Programmiersprache *Dart* und das Frontend-Framework *Flutter* gewählt. Kapitel 2 erläutert die Entscheidungsgrundlage dafür.

Nachfolgend soll auf die Grundlagen der beiden Technologien eingegangen werden.

3.1 Flutter

Flutter ist ein Framework von Google zur Entwicklung von Oberflächen. Es unterstützt eine breite Anzahl an Zielsystemen. Dazu gehören:

- Desktop:[1]
  - Windows:
    - Win32,
    - Universal Windows Platform,
  - macOS,
  - Linux,
- Mobile Endgeräte[2]:
  - Android,
  - iOS,
- und das Web[3].

Flutter ist inspiriert durch das Web-Framework *React* und deren Oberflächenelemente, die *Components* genannt werden[4]. Die visuellen Oberflächenelemente in *Flutter* werden dagegen *Widgets* genannt. *react Components* verfügen über einen Zustand – *State* genannt – der bei Veränderung das Neuzeichnen der visuellen Repräsentation erwirkt. *Flutter* unterscheidet allerdings zwischen zwei Arten von Widgets: denen, die einen Zustand pflegen – den *Stateful Widgets* – und solchen, die keinen Zustand haben – den *Stateless Widgets*.

[1] Vgl. Google LLC, *Desktop support for Flutter*.
[2] Vgl. Google LLC, *Flutter - Beautiful native apps in record time*.
[3] Vgl. Google LLC, *Web support for Flutter*.
[4] Vgl. Google LLC, *Flutter - Introduction to widgets*.

Stateful Widgets pflegen einen Zustand, der mittels der Methode `setstate` gesetzt werden kann. Beim Aufrufen der Methode wird das gesamte Widget neu gezeichnet. Der Zustand selbst ist dabei im visuellen Baum als Vater der visuellen Elemente des Widgets verankert und bleibt erhalten, während die dazugehörigen Oberflächenelemente ausgetauscht werden.

Stateless Widgets haben dagegen keinen solchen Mechanismus. Wie alle Widgets werden sie neu gezeichnet, wenn es durch das Framework angeordnet wurde. Das kann unter anderem der Fall sein, wenn das Widget zum ersten Mal in der Oberfläche auftaucht, oder das Vaterelement und damit alle Kinderelemente neu gezeichnet werden. `oder es von Inherited Widget abhängt?`

StatefulWidgets sind nur eine von vielen Möglichkeiten den Zustand des Programms zu verwalten. Die Formularanwendung verwendet ausschließlich `StatelessWidget`s, da die Verwaltung des Zustands über das sogenannte BloC Pattern umgesetzt wird. Mehr dazu im Kapitel `Kapitel einfügen`. `BLoC Pattern erklären? Oder einfach bei BehaviourSubject verweisen, dass`

`Inherited Widgets?`

3.2 *Dart* Grundlagen

Flutter-Anwendungen werden in der Programmiersprache *Dart* geschrieben. Nachfolgend soll auf eine Reihe von Besonderheiten von *Dart* im Vergleich zu anderen objektorientierten Programmiersprachen eingegangen werden.

Dart ist eine Hochsprache, die hauptsächlich für die Entwicklung von Oberflächen entwickelt wurde, sich jedoch ebenso dazu eignet, Programme für das Back-End zu entwickeln.

Ein Hauptaspekt bei dem Design der Sprache ist die Produktivität des Entwicklers. Mechanismen wie das *hot reload* verkürzen die Entwicklungszyklen erheblich. Das *hot reload* ermöglicht es, während eine Anwendung im Debugmodus ausgeführt wird, Änderungen an deren Quellcode vorzunehmen. Daraufhin werden nur die Teile der laufenden Applikation aktualisiert, die tatsächlich verändert wurden. Währenddessen bleibt die Anwendung in der gleichen Ansicht, anstatt zum Hauptbildschirm zurückgesetzt zu werden, von der aus der Entwickler erneut zur gewünschten Ansicht zurücknavigieren müsste.

3.2.1 AOT und JIT

Nicht nur für die reibungslose Entwicklung sondern auch für das Laufzeitverhalten der finalen Applikation wurde die Sprache optimiert. Für die Ziel-Architekturen ARM32, ARM64

und x86_64 wird *Dart* in Maschinencode kompiliert[5].

Dementsprechend kommt während der Entwicklung eine virtuelle Maschine – die *Dart VM* – über Just-in-time-Kompilierung (JIT) zum Einsatz. Für die Kompilierung in Maschinencode wird dagegen Ahead-of-time-Kompilierung (AOT) eingesetzt.

tree shaking

Für die Minimierung der Dateigröße des resultierenden Kompilats wird das sogenannte *tree shaking* eingesetzt. Das Hauptprogramm importiert über das Schlüsselwort `import` Funktionalitäten aus weiteren *Dart*-Dateien oder sogar ganzen Bibliotheken. Diese Dateien importieren wieder Weitere. Dadurch wird ein Baum aufgespannt. Das *tree shaking* identifiziert, welche Funktionalitäten tatsächlich vom Programm verwendet werden und welche nicht. Dies bringt aber eine wichtige Einschränkung mit sich. Die Metaprogrammierung (der Zugriff auf sprachinterne Eigenschaften, wie etwa Klassen und ihre Attribute) ist damit stark eingeschränkt.

Metaprogrammierung

Bei der Kompilierung werden die Original-Bezeichner durch Symbole ersetzt, welche minimalen Speicherbedarf haben. Aber nicht nur das, denn durch das *tree shaking* werden auch etwaige Eigenschaften und Funktionalitäten entfernt, die nicht verwendet werden. Die sogenannte *Reflexion* oder *Introspektion* versucht auf solche Metainformationen während der Laufzeit zuzugreifen. Da die Eigenschaften aber nicht mehr verfügbar sind, ist *Reflexion* nicht anwendbar. *Dart* greift daher auf eine andere Variante der Metaprogrammierung zurück: die Quellcode-Generierung.

Quellcode-Generierung

Das Package *source_gen* erlaubt das Auslesen der Metainformationen und ermöglicht das Generieren von Quellcode, der von diesen Eigenschaften abgeleitet werden kann. So verwendet beispielsweise das Package *built_value* die Quellcode-Generierung. Zunächst werden Eigenschaften wie Klassennamen und Instanzvariablen mit ihren Bezeichnern und Datentypen gelesen. Die Eigenschaften können dann genutzt werden, um unveränderliche Wertetypen und dazugehörige sogenannte *Builder*-Objekte des *Erbauer*-Entwurfsmusters, sowie Funktionen zum Serialisieren und Deserialisieren von Objekten zu generieren. `Referenzen`

[5] Vgl. Google LLC, *Dart: The platforms*.

3.2.2 Set und Map Literale

Dart erlaubt es Listen (`List`), Mengen (`Set`) und Hashtabellen (`Map`) als sogenannte Literale zu deklarieren. Ein Literal ist die textuelle Repräsentation eines Wertes eines speziellen Datentyps. Beispielsweise ist `"Text"` ein String-Literal für eine Zeichenkette mit den Elementen *T*, *e*, *x*, *t*. So ist auch `{"Text"}` ein Literal für eine Menge (`Set`). Eine Menge mit den gleichen Werten könnte genauso auch wie in Listing **??** erstellt werden.

Es entfällt also die Instanziierung einer Liste, einer Menge oder einer Hashtabelle über den Klassennamen und der darauffolgenden Zuweisung der einzelnen Werte. Stattdessen startet das `Set` und `Map` Literal mit einer öffnenden geschweiften Klammer und endet mit einer schließenden geschweiften Klammer. Innerhalb der Klammern werden die Werte im Fall eines Sets mit `,` getrennt nacheinander aufgeführt (`{1,2}`). Im Fall einer Map werden der Schlüssel und der Wert durch einen `:` voneinander getrennt und die Schlüssel-Wertepaare wiederum durch `,` getrennt nacheinander aufgelistet (`{1: "erster Wert",2:"zweiter Wert"}`). Eine Liste wiederum wird mit eckigen Klammern geöffnet und geschlossen. Die Werte werden erneut mit `,` getrennt voneinander angegeben (`[1,2]`).

Collection for

Dart erlaubt es Schleifen innerhalb von Listen-, Mengen- und Hashtabellen-Literalen zu verwenden. Dabei darf die Schleife jedoch keinen Schleifen-Körper besitzen. Lediglich der Schleifen-Kopf wird dazu im Literal geschrieben. Darauf folgt der Wert, der bei jedem Schleifendurchlauf hinzugefügt werden soll. Dabei kann der Wert von der Schleifenvariable genutzt oder davon abgeleitet werden. Listing **??** geht beispielsweise durch die Liste der Temperatur-Angaben 97.7,105.8, die in Fahrenheit gelistet sind.

Für jeden Schleifendurchlauf wird die Schleifen-Variable `f` mit der entsprechenden Formel in Grad Celsius umgewandelt. Das Ergebnis ist somit äquivalent mit dem `Set`-Literal `{36.5, 38.5, 41}`.

Gleiches gilt für Hashtabellen. Hierbei wird ein Schlüssel-Werte-Paar übergeben. Links von einem `:` ist der Schlüssel und rechts davon der Wert. In Listing **??** wird durch die gleiche Liste von Temperaturen in Fahrenheit iteriert.

Für jede Schleifenvariable `f` wird für das resultierende Schlüssel-Wörter-Paar das Ergebnis in Grad Celsius als Schlüssel und das Ergebnis als Wert eingetragen. Das Ergebnis von `celsiusUndFahrenheit` ist dementsprechend eine `Map` mit dem Wert: `{36.5: 97.7, 38.5: 101.3, 41: 105.8}`

Collection-if

Neben dem Collection-for ist auch die Nutzung von Fallunterscheidungen in Kollektionen erlaubt. Vor dem Wert, der in die Kollektion aufgenommen werden soll oder nicht, kann das Schlüsselwort `if` mit einer darauffolgenden Bedingung in Klammern gesetzt werden. Listing **??** iteriert durch eine Anzahl von Temperaturen in Grad Celsius.

Nur in dem Fall, dass die Temperatur der Schleifen-Variable `c` größer oder gleich 38,5 ist, wird die Temperatur der Liste zugefügt. Das Ergebnis der Liste `fieberTemperaturen` ergibt also `[38.5, 41]`.

3.2.3 Typen ohne Null-Zulässigkeit

Im Vergleich zu vielen anderen Programmiersprachen – wie beispielsweise Java – wird in *Dart* zwischen gewöhnlichen Typen und nullable Typen unterschieden. In Java ist es nur bei atomaren Datentypen wie `int` und `float` vorgeschrieben einen Wert anzugeben. `null` ist bei diesen primitiven Datentypen nicht als Wert erlaubt. Doch nicht atomare Datentypen erlauben immer die Angabe von `null` als Wert. `null` drückt dabei immer das Nicht-Vorhandensein von Daten aus. Ab *Dart* 2.12 kann allen Datentypen standardmäßig kein Null-Wert zugewiesen werden. Das hat den Vorteil, dass der Compiler sich darauf verlassen kann, dass eine Variable niemals den Wert `null` haben kann. Das ist besonders dann nützlich, wenn auf einem Objekt eine Methode aufgerufen wird. Ist das Objekt in Wahrheit `null`, so gibt es erst zur Laufzeit einen Fehler, da die Methode auf der Referenz `null` nicht aufgerufen werden kann. Damit ein Laufzeitfehler geworfen werden kann, muss vor jedem Aufruf einer Methode auf einer Referenz überprüft werden, ob die Referenzen nicht `null` sind. Würde diese Überprüfung nicht stattfinden, so könnte kein Laufzeitfehler geworfen werden und das Programm würde ohne Fehlermeldung abstürzen. Handelt es sich allerdings um eine Referenz, die niemals den Wert `null` annehmen kann, so kann der Compiler die Überprüfung auf Null-Werte für diese Referenzen überspringen. Damit erhört sich zusätzlich die Ausführungsgeschwindigkeit, da die Überprüfung Zeit in Anspruch nimmt. Vor allem aber ist es vorteilhaft für den Entwickler, da der Compiler Fehlermeldungen und Warnungen mitteilen kann, wenn Operationen auf Variablen mit potenziellen Null-Werten verwendet werden. Die Abwesenheit von Daten ist jedoch bei der Entwicklung sehr wichtig. Nicht alle Variablen können immer einen Wert haben. Aus diesem Grund gibt es in *Dart* auch die Typen, die Null-Werte zulassen. Allerdings gelten besondere Regeln für diese Typen.

3.2.4 Typen mit Null-Zulässigkeit

Wird in *Dart* hinter einem Typen ein `?` angegeben, so kann die Variable nicht nur Werte annehmen, die dieser Datentyp zulässt, sondern zusätzlich auch noch den Wert `null`. Methoden auf Objekten mit Null-Zulässigkeit aufzurufen ist nicht ohne Weiteres möglich.

Im Listing **??** wird versucht, auf die Variable `fahrenheitTemperature` den Operator `-` anzuwenden, um sie mit `32` zu subtrahieren.

Der Compiler liefert jedoch einen Fehler, da der Wert der Variablen `null` sein kann, wie die Notation `int?` anzeigt. Solange nicht feststeht, dass die Variable zur Laufzeit tatsächlich nicht `null` ist, kann das Programm nicht kompiliert werden.

Zu diesem Zweck macht *Dart* von der sogenannten *type promotion* – deutsch Typ Beförderung oder Typeinschränkung – Gebrauch. Mithilfe einer Fallunterscheidung kann vor dem Anwenden der Operation nachgesehen werden, ob der Wert der Variablen nicht `null` ist. Innerhalb des Körpers der Fallunterscheidung wird der Typ der Variablen automatisch in einen Typ ohne Null-Zulässigkeit befördert.[6] Der Code in Listing **??** lässt sich damit wieder kompilieren.

Eine Besonderheit stellen dabei allerdings Instanzvariablen dar. In *Dart* wird syntaktisch nicht zwischen dem Aufruf einer Getter-Methode oder einer Instanzvariablen unterschieden. In Listing **??** könnte sich hinter den Aufrufen von `temperature` in den Zeilen 6 und 7 die Instanzvariable verbergen, die in Zeile 2 deklariert ist.

Genauso könnte es aber auch sein, dass eine Klasse von `Patient` erbt und das Feld `temperature` mit einer gleichnamigen Getter-Methode überschreibt. Auch wenn es sehr unwahrscheinlich ist, könnte es trotzdem vorkommen, dass der Aufruf von `temperature` in Zeile 6 einen Wert zurückgibt, der nicht `null` ist und der darauffolgende Aufruf in Zeile 7 `null` liefert. So provoziert es die Klasse `UnusualPatient` im Listing **??**.

Beim ersten Aufruf von `temperature` wird die Zähl-Variable `counter` von `0` auf `1` erhöht. Die Abfrage, ob es sich bei dem Wert von `counter` um eine ungerade Zahl handelt, ist erfolgreich (Z. 6), weshalb mit `97,7` ein valider Wert zurückgegeben wird. Beim zweiten Aufruf erhöht sich `counter` allerdings auf `2`. Die gleiche Abfrage schlägt dieses Mal fehl. Deshalb liefert die Getter-Methode nun `null` (Z. 9). Ein solches Szenario ist schon sehr unwahrscheinlich, doch die Typ-Überprüfung des Compilers arbeitet mit Beweisen. Im Fall von Instanzvariablen kann nicht bewiesen werden, dass zur Laufzeit ein solcher Fall ausgeschlossen werden kann.

Sollte sich der Entwickler sicher sein, dass die Variable nicht `null` sein kann, so kann er

[6] Vgl. Nystrom, *Dart - Understanding null safety - Type promotion on null checks*.

mit einem nachgestellten `!` erzwingen, dass die Variable als nicht `null` angesehen wird (Listing **??**, Z. 3).

Sollte es dann dennoch passieren, dass die Variable `null` ist, so wird eine Fehlermeldung beim Aufruf der Variablen geworfen.

Eine noch sicherere Variante ist es, die Instanzvariable zuvor in eine lokale Variable zu speichern (Listing **??**, Z. 2).

Die lokale Variable hat keine Möglichkeit zwischen den zwei Aufrufen einen unterschiedlichen Wert anzunehmen. Somit kann auch das Suffix `!` weggelassen werden (Z. 4).

3.2.5 Asynchrone Programmierung

Wird auf eine externe Ressource zugegriffen – wie zum Beispiel das Abrufen einer Information von einem Webserver, oder das Lesen einer Datei im lokalen Dateisystem – so handelt es sich um asynchrone Operationen.

Im Sprachkern stellt *Dart* Schlüsselwörter und Datentypen für die asynchrone Programmierung bereit. Das sind unter anderem die Datentypen `Future` und `Stream` sowie die Schlüsselwörter `async` und `await`.

Future

Ein `Future`-Objekt repräsentiert einen potenziellen einmaligen Wert, der erst in der Zukunft bereit steht. Er gleicht damit dem sogenannten `Promise` – deutsch Versprechen – in JavaScript[7].

Das Listing **??** zeigt mit dem Lesen einer Datei ein Beispiel für den Aufruf einer asynchronen Operation.

Anders als erwartet, befindet sich in der Variablen `fileContent` in Wahrheit kein Text mit dem Inhalt der Datei. Stattdessen hat die Variable den Datentyp `Future<String>` und ist lediglich ein sogenannter *Handle* – deutsch Referenzwert – für das potenzielle und zukünftige Ergebnis der Operation.

Mit der Übergabe einer Funktion, die bei Vollendung der Operation aufgerufen wird, kann der Wert ausgewertet werden. Man nennt diese Operation auch *Callback-Funktion*

[7] Vgl. MDN contributors, *Promise - JavaScript | MDN*.

– deutsch Rückruffunktion. Listing **??** zeigt, wie auf den Dateiinhalt zugegriffen werden kann.

Über die Methode `then` wird eine Funktion übergeben, die genau einen Parameter hat. In diesem Parameter wird der Text der gelesenen Datei bei Vollendung der Operation übergeben.

Der Einsatz von *Callback-Funktionen* kann den Quellcode stark verkomplizieren. Man spricht von der sogenannten *callback hell* – deutsch Rückruffunktionen-Hölle –, wenn solche *Callback-Funktionen* über etliche Level hinweg ineinander verschachtelt sind.

Um genau das zu verhindern, existieren in *Dart* die Schlüsselwörter `async` und `await`. Genauso heißen sie auch in anderen Sprachen wie etwa C# ab Version 4.5 und JavaScript ab Version ES2017[8][9].

Listing **??** zeigt, dass das Anwenden des Schlüsselwortes `await` vor der Operation `file.readAsString` dafür sorgt, dass der zukünftige Wert direkt in `fileContent` gespeichert wird.

Ganz ohne *Callback-Funktion* kann der Dateiinhalt in der darauffolgenden Zeile ausgegeben werden.

Doch jede Funktion, die auf andere Funktionsaufrufe wartet, muss selbst als asynchron gekennzeichnet werden. Dazu dient das `async` Schlüsselwort vor Beginn des Methoden-Körpers.

Streams

Streams liefern nicht nur einen Wert – wie im Fall eines `Future` – sondern eine Serie von Werten, die in der Zukunft geliefert werden. Listing **??** zeigt wie auf einen solchen *Stream* gehorcht werden kann.

`countStream` liefert jede Sekunde einen neuen Wert, nämlich die aktuelle Sekunde – von 0 beginnend. Mit `countStream.listen` kann eine Funktion übergeben werden, die immer dann ausgeführt wird, wenn dem `countStream` ein neuer Wert hinzugefügt wurde. Der erste Parameter ist dabei der hinzugefügte Wert.

Es wird zwischen zwei Arten von *Streams* unterschieden. Solche, die genau einen Empfänger haben – *single subscription streams* – und solche, die beliebig viele Empfänger haben können – *broadcast streams*.

[8] Vgl. MDN contributors, *async function - JavaScript | MDN*.
[9] Vgl. Bray, *Async in 4.5: Worth the Await*.

Für die Formularanwendung sind ausschließlich *broadcast streams* zu berücksichtigen. Die *Streams* sollen verwendet werden, um Änderungen in der Eingabemaske zu behandeln. Die Oberflächenelemente horchen auf diese Änderungen. Teile der Oberfläche und damit die Oberflächenelemente, welche auf die *Streams* horchen, werden immer wieder neu gezeichnet. Dabei werden die Elemente entfernt und durch neu konstruierte ersetzt. So melden sich immer wieder Zuhörer vom *Stream* ab und neue Elemente melden sich an. Aufgrund dessen kommen nur *broadcast streams* infrage.

4 Schritt 1 – Formular in Grundstruktur erstellen

4.1 Widget SelectionCard

Das Listing 4.1 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 4.1: 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 4.2, 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 4.2: Die *build*-Methode der Klasse *SelectionCard* in Schritt 1

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 4.3 erklärt wird.

4.1.1 Bildschirm für die Auswahl der Optionen

Die Methode `createMultipleChoiceSelectionScreen` (Listing 4.3) 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 4.3: Die Methode *createMultipleChoiceSelectionScreen*

4.2 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 4.5, 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 4.4: Initialisierung des Integrations Tests

Anschließend kann mit der Annotation and generate mocks (Listing 4.5, 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 4.5: 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 4.6). 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 4.6: 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 4.8) benötigt lediglich die Liste der Auswahloptionen `choices`, die ihr hinterlegt ist.

Listing 4.7: 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. Gamma u. a., *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, S. 107–116.

erhält er den `Finder` des Kindelements 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 4.8). 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 4.8: Die Hilfsmethode tabConfirmButton

Ist der Integrationstest aktuell in dem Auswahlbildschirm, so sorgt `tabOption` dafür, dass Auswahloptionen gewählt wird (Listing 4.9). 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 4.9: Die Hilfsmethode tabOption

Ein Klick auf das Text-Label reicht bereits aus, denn damit wird das Vaterelement – 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 4.10). Sie findet das `TextFormField`, indem es zunächst nach dem Titel mit `find.text(title)` und anschließend dessen Vaterelement vom Typ `TextFormField` sucht (Z. 83). Sollte so-

wohl 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 4.10: Die Hilfsmethode fillTextFormField

Während der Integrationstest startet, öffnet sich als Erstes der Übersichts-Bildschirm. Zunächst wird gewartet, dass alle Widgets korrekt initialisiert wurden (Listing 4.11, 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 4.11: 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 4.12, 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 4.12: Der letzte Status wird ausgewählt

Nachfolgend soll der Test das Eingabefeld für den Maßnahmen-Titel überprüfen (Listing 4.13). 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 4.13: Der Maßnahmentitel wird eingegeben

Listing 4.14: 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!`

5 Schritt 2

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. In der Eingabemaske sollen sie in einer Rubrik mit der Überschrift Maßnahmencharakteristika auftauchen (Abb. 5.1).

Abbildung 5.1: Die Eingabemaske in Schritt 2

In den Tabellen im Übersichtsbildschirm erscheinen die Werte rechts vom Maßnahmentitel (Abb. 5.2).

Abbildung 5.2: Der Übersichtsbildschirm in Schritt 2

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.

5.0.1 Integrationstest erweitern

Noch vor der Implementierung der Änderungen soll zunächst der Integrationstest um die zusätzlichen Selektionen erweitert werden (Listing 5.1). Nach den letzten Eingaben und bevor der Button zum Speichern ausgelöst wird, erfolgt die Selektion der fünf Optionen (Z. 106-119).

Listing 5.1: 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 5.2, 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

Listing 5.2: Der Integrationstest überprüft im *JSON*-Dokument den Schlussel *massnahmenCharakteristika*

nicht auftauchen. Die neuen Schlüssel-Werte-Paare können nicht in der Hash-Tabelle auftauchen, da sie dem entsprechenden Wertetyp 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` – 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 des *Views* damit der Integrationstest erneut erfolgreich abschließt.

5.0.2 Hinzufügen der Auswahloptionen

Der Integrationstest selektiert unter anderem die Förderklasse mit der Abkürzung `aukm_ohne_vns`. Sie wird den Auswahloptionen hinzugefügt, wie in Listing 5.3 zu sehen ist. Die Liste aller hinzugefügten Auswahloptionen in diesem Schritt ist in Anhang **??** auf den Seiten **??** bis **??** zu finden.

Listing 5.3: Die Klassenvariable *aukm_ohne_vns* vom Typ *FoerderklasseChoice*

5.0.3 Aktualisierung des *Models*

Damit der Hash-Tabelle der Schlüssel `'massnahmenCharakteristika'` hinzugefügt wird, muss der entsprechende Eintrag im Wertetyp `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 5.4, Z. 15).

Listing 5.4: *massnahmenCharakteristika* wird dem Wertetyp *Massnahme* hinzugefügt

[1] Beck, *Test-driven development: by example*, S. 9.

Bei dem Datentyp handelt es sich um einen weiteren Wertetyp: `MassnahmenCharakteristika`, welcher in Listing 5.5 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).

Listing 5.5: Der Wertetyp *Massnahmencharakteristika*

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

5.0.4 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 5.6, Z. 23-27).

Listing 5.6: Die *Maßnahmencharakteristika* werden dem Tabellenkopf hinzugefügt

Für jede Zeile der Tabelle werden weitere selektierbare Zellen generiert (Listing 5.7, 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 5.7: Die *Maßnahmencharakteristika* werden dem Tabellenkörper hinzugefügt

5.0.5 Aktualisierung des *ViewModels*

Damit die Eingabefelder die neuen Werte eintragen können, muss das *ViewModel* oder die beobachtbaren Subjects bereitstellen (Listing 5.8, 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 5.9, Z. 29-36).

[2] Vgl. Google LLC, *Dart Programming Language Specification 5th edition*, S. 165.

Listing 5.8: Die *Maßnahmencharakteristika* werden dem *ViewModel* hinzugefügt

Listing 5.9: Konvertierung des *Models* in das *ViewModel* für die *Maßnahmencharakteristika*

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 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 5.10, 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 5.10: Konvertierung des *ViewModels* in das *Model* für die *Maßnahmencharakteristika*

5.0.6 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 5.13, 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 Auswahl-

Listing 5.11: Die Methode *buildSelectionCard*

option 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 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 5.12) nun deutlich kürzer als die ursprüngliche Variante mit 11 Zeilen (siehe Seite **??**).

Listing 5.12: Der Aufruf von *buildSelectionCard* für die Menge *letzterStatusChoices*

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 5.13, Z. 82-98).

Listing 5.13: 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 Hilfsfunktionen `buildSectionHeadline` und `buildSubSectionHeadline` bauen die Überschriften (Listing 5.14, 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 5.14: Die Hilfsfunktionen *buildSectionHeadline* und *buildSubSectionHeadline*

6 Schritt 3

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. Auf Validierungsfehler wird in der Eingabemaske mit Benachrichtigungen in rot gefärbter Schrift hingewiesen (Abb. 6.1).

Abbildung 6.1: Die Eingabemaske in Schritt 3

6.1 Einfügen des *Form-Widgets*

`Flutter` stellt das Widget `Form` für die Validierung von Eingabefeldern bereit. 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 6.1, Z. 161). Das `Form`-Widget muss über einen `key` registriert werden (Z. 162), damit auf die Validierungsfunktionen zurückgegriffen werden kann.

Listing 6.1: Einfügen des *Form-Widgets*

Die Erstellung des `formKey` findet zu Beginn der `build`-Methode des Eingabeformulars statt (Listing 6.2, 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 6.2: Der *formKey* wird erstellt

6.2 Validierung des Maßnahmentitels

Das Eingabefeld für den Maßnahmen-Titel ist ein `TextFormField` (Listing 6.3, 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 6.3: Die Funktion *createMassnahmenTitelTextFormField* mit Validierung

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.

6.3 Validierung der Selektionskarten

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 6.4, 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 6.4: Die Methode *buildSelectionCard* mit Validierung

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 6.5, Z. 29). Da er `null` sein darf, ist er mit dem Suffix `?` als Typ mit Null-Zulässigkeit gekennzeichnet (Z. 21).

Listing 6.5: *errorText* wird der *SelectionCard* hinzugefügt

Durch Einfügen einer `Column` zwischen der `Card` (Listing 6.6, 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 6.6: *errorText* wird ausgegeben

6.4 Speichern der Eingaben im Entwurfsmodus

Oberhalb des vorhandenen `FloatingActionButton` wird nun ein weiterer eingefügt, der zum Speichern des Entwurfs mit der Funktion `saveDraftAndGoBackToOverviewScreen` genutzt werden soll (Listing 6.7, 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 6.7: Der *FloatingActionButton* zum Speichern der Maßnahmen im Entwurfsmodus

Auch der `WillPopScope` erhält die gleiche Fehlerbehandlung (Listing 6.8). 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 6.8: 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 6.9: Die Funktion *saveDraftAndGoBackToOverviewScreen*

Die Hilfsfunktion `inputsAreValidOrNotMarkedFinal` überprüft zunächst, ob der letzte Status ein anderer ist als *abgeschlossen* (Listing 6.10, 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).

Listing 6.10: Die Funktion *inputsAreValidOrNotMarkedFinal*

Sollte es zu einem Fehler kommen, so zeigt die Hilfsfunktion `showValidationError` dem Benutzer die entsprechende Fehlermeldung an (Listing 6.11). Sie bietet ihm darüber hinaus 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 6.11: Die Funktion *showValidationError*

7 Schritt 4

Die im letzten Schritt implementierte Validierung überprüft lediglich auf leere Eingabefelder. Im Folgenden soll die Überprüfung der Kompatibilität der Auswahloptionen untereinander in die Validierung miteinbezogen werden. Deaktivierte Auswahloptionen sind nicht anwählbar und werden im Selektionsbildschirm mit einem vorangestellten Kreuz gekennzeichnet (Abb. 7.1).

Abbildung 7.1: Der Selektionsbildschirm in Schritt 4

Wenn eine Auswahloption selektiert ist und durch eine weitere Selektion in einem anderen Feld anschließend invalide geworden ist, wird diese rot gekennzeichnet (Abb. 7.2).

Abbildung 7.2: Die Eingabemaske in Schritt 2 mit einem selektierten invaliden Wert

In der Eingabemaske wird dann das gesamte Eingabefeld rot eingefärbt (Abb. 7.3).

Abbildung 7.3: Der Selektionsbildschirm in Schritt 4 mit einem selektierten invaliden Wert

7.1 Hinzufügen der Bedingungen zu den Auswahloptionen

Es gibt einfache Bedingungen wie beispielsweise die der Zielfläche *AL*. Dessen Auswahl kann nur dann erfolgen, wenn nicht die Kategorie *Anbau Zwischenfrucht/Untersaat* ausgewählt ist (Listing 7.1).

Doch es tauchen auch komplexe Bedingungen auf, wie etwa die Abhängigkeit der Zielfläche *Wald/Forst* (Listing 7.2). Um sie auszuwählen, muss die Förderklasse eine von 3 Werten beinhalten: *Erschwernisausgleich* (Z. 97), *Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz* (Z. 98) oder *Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten , aber OHNE Vertragsnaturschutz* (Z. 99).

Gleichzeitig darf für die *Kategorie* weder *Anbau Zwischenfrucht/Untersaat* (Z. 100) noch *Förderung bestimmter Rassen / Sorten / Kulturen* (Z. 101) gewählt sein.

Äußerst wichtig ist hier die Auswahl der richtigen logischen Operatoren. Innerhalb des gleichen Typs – wie etwa der *Förderklasse* – muss das logische Oder `||` verwendet werden

Listing 7.1: Der Klassenvariable *al* des Typs *ZielflaecheChoice* wird eine Bedingung hinzugefügt

(Z. 97,98,100). Das logische Und würde hier keinen Sinn ergeben, da es unmöglich ist, in einem Einfachauswahlfeld gleichzeitig zwei Optionen ausgewählt zu haben. Um Bedingungen unterschiedlichen Typs miteinander zu verknüpfen, ist dagegen das logische und `&&` zu benutzen (Z. 99), denn die Bedingungen der *Förderklasse* und der *Kategorie* müssen gleichzeitig erfüllt sein. Hier ist wiederum das Nutzen des logischen Oders nicht angemessen, denn es wäre nicht ausreichend, wenn nur die Bedingungen eines der beiden Typen erfüllt wäre. Wäre also beispielsweise für die *Förderklasse* die Option *Erschwernisausgleich* gewählt, so wäre es völlig unerheblich, welche Auswahl für die *Kategorie* selektiert wurde. Die Bedingung wäre trotzdem erfüllt, auch wenn für die *Kategorie* die nicht erlaubte Option *Anbau Zwischenfrucht/Untersaat* gewählt ist.

Listing 7.2: Der Klassenvariable *wald* des Typs *ZielflaecheChoice* wird eine Bedingung hinzugefügt

Für die Liste aller hinzugefügten Bedingungen siehe Anhang **??** auf den Seiten **??** bis **??**.

Bei der Bedingung handelt sich um eine Funktion, die einen Wahrheitswert `bool` zurückgibt und als Parameter die Menge aller bisher ausgewählten Auswahloptionen `Set<Choice>` übergeben bekommt. Die Signatur dieser Funktion wird als Typdefinition mit dem Namen `Condition` deklariert (Listing 7.3, Z. 3). Über diese Typdefinition kann sie als Instanzvariable in der Klasse `Choice` deklariert werden (Z. 8). Der Konstrukteur erhält einen weiteren Parameter für die Bedingung (Z. 12).

Er ist optional, da es Auswahloption gibt, die keine Bedingung haben. Deshalb wird mit der Notation `Condition?` erreicht, dass die Bedingung auch ausgelassen werden kann und in diesem Fall `null` ist. Sollte das der Fall sein, so soll eine Standardfunktion verwendet werden. Diese Standardfunktion ist `_conditionIsAlwaysMet` (Z. 15). Unerheblich davon, welche Auswahloptionen in Vergangenheit gewählt wurden, gibt diese Funktion immer `true` zurück. Denn eine Auswahloption, die keine Bedingung hat, ist immer auswählbar. Sollte die übergebene Bedingungen ausgelassen worden und damit `null` sein, so wählt die *If-null Expression* den Ausdruck rechts von dem `??` und damit die Standardfunktion `_conditionIsAlwaysMet` aus, welche der Instanzvariablen `condition` zugewiesen wird (Z. 13). Ansonsten speichert der Konstruktor die übergebene Funktion. Aus diesem Grund ist es nicht möglich, dass die condition in der Instanzvariablen nur sein kann, weshalb sie ohne den Suffix? Als variable ohne zu null Zulässigkeit deklariert werden kann. Da der Ausdruck rechts von dem `??` nicht `null` sein kann, so kann auch der gesamte Ausdruck der vorliegenden *If-null Expression* nicht `null` sein. Damit ist es möglich, die Instanzvariable `condition` ohne den Suffix `?` als Variable ohne Null-Zulässigkeit zu deklarieren. Die Instanzmethode `conditionMatches` ruft die übergebene Funktion für die Bedingung über die Methode `call` auf (Z. 10). Das erlaubt den Ausdruck durch vereinfacht darzustellen. Der Ausdruck `wald.condition(priorChoices)` kann dadurch durch die explizitere

Schreibweise `wald.conditionMatches(priorChoices)` ersetzt werden.

Listing 7.3: Die Klasse *Choices* wird die Instanzvariable *condition* hinzugefügt

7.2 Hinzufügen der Momentaufnahme aller ausgewählten Optionen im gesamten Formular

Die Menge der bisherigen Ausfülloptionen setzt sich aus den aktuellen Inhalten der Auswahlfelder zusammen. Sie ist also die Momentaufnahme aller Werte, die jeweils über die Getter-Methode `value` von allen `BehaviorSubject`-Objekten im *ViewModel* abgerufen werden kann. Doch genau diese Momentaufnahme muss immer dann neu erstellt werden, wenn sich auch nur ein Auswahlfeld ändert. Genau darum kümmert sich das `BehaviorSubject priorChoices` im *ViewModel* (Listing 7.4).

Listing 7.4: Das *BehaviorSubject priorChoices*

Es wird mit dem Typparameter `Set<Choice>` deklariert (Z. 20) und mit einer Momentaufnahme initialisiert: einer leeren Menge) (Z. 21). Im Konstruktor des *ViewModels* wird dann auf Änderung aller `BehaviorSubject`-Objekte im *ViewModel* gehorcht. Dies wird durch die Funktion `combineLatest` des Pakets `rx.dart` ermöglicht `24`. Sie erlaubt die Übergabe einer Kollektion von *Streams*. In diesem Fall alle `BehaviorSubject`-Objekte des *ViewModels* (Z. 25-29). Wenn auch nur einer dieser *Streams* ein neues Ereignis sendet, so emittiert auch der kombinierte *Stream* ein neues Ereignis. Dem zweiten Parameter der Funktion `combinelatest` kann als Argument eine Funktion übergeben werden, die das zu emittierende Ereignis konstruiert (Z. 30-37). Der erste Parameter dieser Funktion enthält alle letzten Ereignisse der übergebenen *Streams*. Doch der vorliegende Aufruf hat keine Verwendung für den Parameter. Statt eines Variablennamens wird hier ein Unterstrich `_` verwendet (Z. 30).

In Sprachen wie etwa *JavaScript* und *Python* ist dies gängige Praxis für die Benennung von Parametern, die nicht genutzt werden. In *Kotlin* und *Dart* wurde diese Praxis zur Konvention gemacht[1][2] . Die anonyme Funktion gibt eine Menge zurück, in welcher alle Werte der `BehaviorSubject`-Objekte integriert werden (Z. 31-37). Das *Collection if* Statement schließt dabei jeweils den Wert `null` aus (Z. 32-36). Somit taucht niemals der Wert `null` in der Menge auf und damit kann die Menge mit dem Typparameter `Choice` ohne Null-Zulässigkeit deklariert werden. Sollte ein Auswahlfeld nicht gewählt und damit der Wert des `BehaviorSubject` `null` sein, so taucht diese Option einfach nicht in der Menge auf. Sind alle Auswahlfelder nicht belegt und damit `null`, so ist die Menge leer. Doch der

[1] Vgl. Google LLC, *Dart - Effective Dart - Style - PREFER using _, __, etc. for unused callback parameters.*

[2] Vgl. JetBrains s.r.o., *Kotlin - High-order functions and lambdas - Underscore for unused variables.*

kombinierte *Stream* `choicesStream` liefert immer nur die neuen Ereignisse und speichert nicht den zuletzt übermittelten Wert. Deshalb wird das `BehaviorSubject` `priorChoices` verwendet. Die Methode `listen` horcht auf Änderungen des `choicesStream`-Objekts und fügt das übertragene Ereignis immer `priorChoices` hinzu. Damit existiert immer ein Wert für die Momentaufnahme der aktuell ausgewählten Auswahloptionen. Sie ist ursprünglich die leere Menge `{}` und nachfolgend immer das zuletzt übermittelte Ereignis des `choicesStream`.

7.3 Reagieren der Selektionskarte auf die ausgewählten Optionen

Dadurch , dass `priorChoices` nun im *ViewModel* verfügbar ist , kann es im Eingabeformular bei der Konstruktion der `SelectionCard` als Argument übergeben werden (Listing 7.5, Z. 143).

Listing 7.5: Dem Konstruktor der *SelectionCard* wird das *BehaviorSubject priorChoices*

Die Klasse `SelectionCard` deklariert die `priorChoices` als Instanzvariable (Listing 7.6, Z. 19) und initialisiert sie direkt bei der Übergabe im Konstruktor, ohne sie zu modifizieren (Z. 28).

Listing 7.6: Die Klasse *SelectionCard* erhält die Instanzvariable *priorChoices*

Dadurch, dass das `BehaviorSubject` ein *Stream* ist, kann die Selektionskarte auf Änderungen reagieren, die sich an `priorChoices` vollziehen, obwohl diese Änderungen außerhalb der Klasse geschehen. Würde stattdessen eine Liste der bisherigen Auswahloption übergeben werden, so wäre diese eine Kopie. Diese Kopie hätte den Zustand einer Momentaufnahme aller bisherigen Auswahloptionen zum Zeitpunkt der Konstruktion des `SelectionCard`-Elementes. Alle Änderungen, die nach diesem Zeitpunkt an den Auswahloptionen geschehen sind, würden sich nicht darin widerspiegeln. Eine Selektionskarte würde daher auch keinen Fehler anzeigen, wenn ihre ausgewählten Optionen durch Änderungen von außen invalide werden würden. Der Grund dafür ist, dass sie noch eine alte Kopie der bisherigen Auswahloptionen verwendet.

Eine andere Möglichkeit wäre, eine Setter-Methode zu implementieren, die den Wert der bisherigen Auswahloptionen neu setzt. Doch das Programm verwaltet keine Referenzen auf alle gebauten Selektionskarten. Somit kann auch nicht über eine Referenz eine Setter-Methode aufgerufen werden, denn eine solche Referenz existiert nicht. Die übliche Vorgehensweise wäre in *Flutter*, das gesamte Widget neu zu zeichnen. Bei Einsatz eines *Stateful-Widgets* und Zustandsänderungen über die `setState`-Methode würde dies das Neuzeichnen des gesamten Formulars bedeuten.

Performanter ist es dagegen, wenn nur die Inhalte der Selektionskarten ausgetauscht werden. Anstatt ausschließlich auf die Änderungen der eigenen Auswahloptionen zu reagieren, horcht der `StreamBuilder` nun auf den *Stream* `priorChoices` (Listing 7.7, Z. 52) und damit auf die Änderungen aller Auswahlfelder. Vor der Konstruktion der Karte wird nun überprüft, ob einer der ausgewählten Auswahloptionen in `selectedChoices` eine invalide Auswahl enthält (Z. 55-56). Das kann über die Funktion any herausgefunden werden, indem für jede ausgewählte Option die Methode `conditionMatches` mit der Menge aller ausgewählten Optionen im gesamten Formular aufgerufen wird (Z. 56). Die rote Farbe der Selektionskarte wurde bereits bei der Validierung im letzten Schritt verwendet, wenn der dem Konstruktor ein `errorText` übergeben wurde. Nun wird diese Bedingung erweitert. Sollte es auch nur eine falsche Selektion geben oder aber der `errorText` gesetzt sein, so ist die Karte rot. Anderenfalls wird dem Parameter `tileColor` `null` übergeben (Z. 70). `null` bedeutet, dass keine Farbe übergeben und damit die Standardfarbe verwendet wird.

Listing 7.7: Die *SelectionCard* reagiert auf Änderungen des Streams *priorChoices*

7.4 Reagieren des Auswahlbildschirms auf die ausgewählten Optionen

Der Auswahlbildschirm wird im Folgenden um zwei weitere Funktionalitäten erweitert (Listing **??**).

Sollten durch neue Selektionen im Formular bereits selektierte Optionen im Auswahlbildschirm nun invalide sein, so werden diese rot gefärbt. Weiterhin erscheinen invalide Optionen, die nicht ausgewählt sind, am Ende der Liste ohne Checkbox zum Auswählen. Außerdem erhält die Option ein Kreuz-Icon als Indikator dafür, dass sie nicht angewählt werden kann.

Zu diesem Zweck konstruiert der `StreamBuilder` vor der Rückgabe des `ListView` zwei Mengen. Die Menge `selectedAndSelectableChoices` (Z. 95) beinhaltet alle Auswahloptionen, die entweder selektiert oder selektierbar sind. Dies beinhaltet auch Optionen, die invalide und trotzdem selektiert sind. Die zweite Menge `unselectableChoices` (Z. 96) dagegen beinhaltet alle Optionen, die invalide sind, und nicht selektiert sind. Eine Schleife iteriert über alle verfügbaren Optionen, welche der Auswahl Bildschirm anzeigt (Z. 90-105). Sollte die Option in den selektierten Optionen enthalten (Z. 99), oder aber mit den Selektionen aller anderen Auswahlfelder kompatibel sein, so wird sie der Menge `selectedAndSelectableChoices` hinzugefügt (Z. 101). In jedem anderen Fall wird die Option Teil der Menge `unselectableChoices` (Z. 103).

Für die Konstruktion der `CheckboxListTile`-Elemente wurde zuvor die Menge aller Aus-

wahloptionen verwendet. Nun wird stattdessen nur die Menge der selektierbaren und selektierten Auswahloptionen genutzt (Z. 108). Neben dem Vergleich, ob die Option selektiert ist (Z. 109), erfolgt nur noch ein weiterer Vergleich, ob die Option inkompatibel mit den ausgewählten Optionen aller anderen Auswahlfelder ist (Z. 111). Das Ergebnis des Vergleiches wird in der lokalen Variable `selectedButDoesNotMatch` gespeichert.

Sollte diese Variable `true` sein, so erscheint das `CheckboxListTile`-Element mit einem rot eingefärbten im Hintergrund. Der Benutzer hat über die Checkbox dann die Möglichkeit, diese Auswahl zu deselektieren. Da das hinterlegte *ViewModel* durch diese Deselektion direkt aktualisiert wird (Z. 122-123), so baut der `StreamBuilder` auch den `ListView` neu. Die deselektierte Option wird dann Teil von der Menge `unselectableChoices` (Z. 103) sein. So erscheint sie dann – ganz genau wie alle anderen unselektierbaren Auswahloptionen – ohne roten Hintergrund aber auch ohne anklickbare Checkbox am Ende der Liste (Z. 134-142).

Solche unselektierbaren Optionen werden schlicht als `ListTile`-Element statt als `CheckCoxListTile` gezeichnet (Z. 135-139). Damit fehlt ihnen die Checkbox zum Selektieren. Über den Parameter `leading` kann jedoch anstelle der Checkbox ein beliebiges Widget – in diesem Fall ein Icon – eingefügt werden. `Icons.close` zeichnet ein Kreuz-Symbol, um zu signalisieren, dass diese Option nicht anwählbar ist.

7.4.1 Hinzufügen der Momentaufnahme zur Validierung

Alle bisher eingefügten Vergleiche hatten lediglich den Zweck, die invaliden Optionen einzufärben und von der Selektion durch den Benutzer auszuschließen. Doch noch sind sie nicht Teil der Validierung des Formulars. Sollte der Benutzer die aktuell eingetragene Maßnahmen im abgeschlossenen Status abspeichern wollen, so kann dies auch mit invaliden Optionen erfolgen. Um das zu verhindern, wird noch ein Vergleich zu der anonymen Funktion hinzugefügt, welche als Argument dem Parameter `validator` des `FormField` übergeben wird (Listing 7.8).

Listing 7.8: Die *validator* Funktion von *FormField* in Schritt 4

Sollte auch nur eine der selektierten Optionen `choices` die ihr hinterlegte Bedingungen nicht erfüllen (Z. 132), so speichert die lokale Variable `atLeastOneValueInvalid` den Wert `true` ab (Z. 131).

In dem Fall gibt die Funktion die entsprechende Fehlermeldung an den Benutzer zurück (Z. 135). Somit ist es nun auch nicht mehr möglich, eine Maßnahme abzuspeichern, wenn sie invalide Auswahloptionen enthält. Erst wenn alle Auswahlfelder gefüllt sind und die gefüllten Optionen alle die jeweils hinterlegten Bedingungen erfüllen, so werden die `validator`-

Funktionen `null` statt einer Fehlermeldung zurückgegeben (Z. 138). Nur dann kann eine Maßnahme mit dem Status *abgeschlossen* gespeichert werden.

8 Schritt 5

Im letzten Schritt wurde das primäre Problem der Formularanwendung gelöst: Auswahloptionen sollen nur dann anwählbar sein, wenn sie die ihr hinterlegte Bedingung erfüllen. Darüber hinaus können nur Maßnahmen gespeichert werden, deren Auswahloptionen untereinander kompatibel sind.

Durch das Lösen dieses Problems ist ein neues Problem entstanden: Alle Selektionskarten müssen bei einer Selektion neu gezeichnet werden. Bei einer geringen Anzahl von Auswahlfeldern sollte das noch keine gravierenden Auswirkungen auf das Laufzeitverhalten der Applikation haben. Doch je zahlreicher die Auswahlfelder werden, desto länger dauert die Aktualisierung der Oberfläche.

Das Problem kann folgendermaßen entschärft werden: Noch bevor das Widget `SelectionCard` den `StreamBuilder` in der `build`-Methode zurückgibt, wird ein neuer *Stream* namens `validityChanged` erstellt (Listing 8.1, Z. 51-54).

Es handelt sich um eine sogenannte Transformation des *Streams* `priorChoices`, welcher die Momentaufnahme aller ausgewählten Optionen im gesamten Formular übermittelt. Immer dann, wenn der *Stream* `priorChoices` ein neues Ereignis sendet, geschieht für die Abwandlung dieses *Streams* folgendes: Die Methode `map` wandelt jedes Ereignis in ein neues Objekt um (Z. 52). Die aktuelle Momentaufnahme der Auswahloptionen im Formular wird dazu im Parameter `choices` gespeichert. Bei der Umwandlung des Ereignisses werden die ausgewählten Optionen der aktuellen Selektionskarte über `selectionViewModel.value` abgerufen. Sollte es sich beispielsweise bei der aktuellen Selektionskarte um das Auswahlfeld der *Kategorie* handeln, so könnte der ausgewählte Wert *Düngemanagement* sein. Für den Wert oder die Werte wird nun überprüft, ob sie mit der neuen Momentaufnahme der Selektionen im Formular kompatibel sind. Wurde also beispielsweise bei der neuen Selektion in der *Förderklasse* nun *Ökolandbau* ausgewählt, so würde die Option *Düngemanagement* nun invalide werden, da sie nur mit der Förderklasse *Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz* bzw. *Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten, aber OHNE Vertragsnaturschutz* kompatibel ist. Die Methode `map` wandelt also das neue Ereignis der Momentaufnahme aller Selektionen im Formular in einen einzigen Wahrheitswert um. Ist der Wahrheitswert `true`, bedeutet dies, dass alle ausgewählten Optionen in der aktuellen Selektionskate valide sind. Ist er dagegen `false`, so ist wenigstens eine der Auswahloption mit den restlichen Auswahloptionen der anderen

Auswahlfelder im Formularen nicht kompatibel.

Der resultierende *Stream* wird weiter transformiert: Durch die Funktion `distinct` (Z. 54) werden nur Ereignisse gesendet, sofern sie sich von dem letzten Ereignis unterscheiden. Ein Beispiel: Für die *Kategorie* ist *Düngemanagement* ausgewählt. Für die *Förderklasse* ist *Erschwernisausgleich* im letzten Ereignis ausgewählt worden. *Düngemanagement* ist mit *Erschwernisausgleich* nicht kompatibel, weshalb das letzte Ereignis des durch `map` transformierten *Streams* `false` war. Nun wird für die *Förderklasse* eine weitere Selektion vorgenommen: *Ökolandbau* wird ausgewählt. Auch diese Option ist mit *Düngemanagement* nicht kompatibel. Der durch `map` transformierten *Stream* wird also erneut ein Ereignis mit dem Wert `false` senden. Doch bereits das letzte Ereignis war `false`. Die Methode `distinct` verhindert, dass dieses redundante Ereignis weitergeleitet wird. Nun erfolgt noch eine weitere Selektion: Für die *Förderklasse* wird *Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz* selektiert. Nun ist die *Kategorie Düngemanagement* mit der neuen Selektion kompatibel. Der aus der Methode `map` resultierende *Stream* liefert dieses Mal den Wert `true`. Das letzte Ereignis hatte den Wert `false`. Die Werte der beiden letzten Ereignisse unterscheiden sich also, was dazu führt, dass die Methode `distinct` das veränderte Ereignis nicht filtert sondern weiterleitet.

Der *Stream* `validityChanged` sendet also immer genau dann Ereignisse, wenn sich etwas an der Validität der Auswahloptionen der aktuellen Selektionskarte ändert. Doch dieser *Stream* kann nicht für den `StreamBuilder` benutzt werden. Denn wenn sich die Auswahl in der aktuellen Selektionskarte ändert und die Validität dadurch unverändert bleibt, so erfolgt kein neues Zeichnen der Selektionskarte. Deshalb ist eine Kombination der *Streams* `validityChanged` und `selectionViewModel` erforderlich. Das `BehaviorSubject` `needsRepaint` soll als diese Kombination fungieren (Z. 56). Es wird mit dem Wert (Z. true) initialisiert. Es ist unerheblich, welcher Wert in dem *Stream* aktuell gespeichert ist. Lediglich dass ein neues Ereignis hinzugefügt wird, um die Aktualisierung der Oberfläche auszulösen, ist wesentlich. Mit der Methode `listen` wird nun sowohl auf den *Stream* `validityChanged` (Z. 57) als auch auf `selectionViewModel` (Z. 58) gehorcht. Jedes empfangene Ereignis wird dabei dem `BehaviorSubject` `needsRepaint` hinzugefügt.

Dadurch, dass `needsRepaint` für den `StreamBuilder` verwendet wird (Z. 61), zeichnet sich die Selektionskarte immer dann neu, wenn sich die beinhaltenden Auswahloptionen oder aber dessen Validität ändert.

Listing 8.1: Der *Stream validityChanged* in Schritt 5

Dieses Verhalten kann auch bei Ausführung der Applikation im Debugmodus in Android Studio beobachtet werden. Der *Flutter Performance*-Tab gibt eine Übersicht über die Anzahl der im letzten Frame neu gezeichneten Widgets (Abb. 8.1). Angenommen für die *Förderklasse* ist *Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz* und für die *Kategorie* ist *Düngemanagement* ausgewählt. Wenn nun für die `Förderklasse` die Option

Abbildung 8.1: Das *Card-Widget* wird einmal neu gezeichnet

Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten, aber OHNE Vertragsnaturschutz selektiert wird, so ist im *Flutter Performance*-Tab zu beobachten, dass das Widget Card nur einmal neu gezeichnet wurde.

Das ergibt Sinn, denn es hat sich nichts an der Validität eines anderen Auswahlfeld geändert. Lediglich die Selektionskarte für die Förderklasse muss neu gezeichnet werden, da sich seine Selektion angepasst hat. Wird nun aber die *Förderklasse Ökolandbau* ausgewählt, so ist zu beobachten, dass das `Card` Widget zweimal gebaut wurde: Einmal für die Selektionskarte der *Förderklasse*, da sich dessen *ViewModel* änderte; Ein weiteres Mal für die Selektionskarte der *Kategorie*, da die Auswahl *Düngemanagement* nicht länger valide ist und die Karte deshalb mit einem roten Hintergrund eingefärbt werden muss (Abb. 8.2).

Abbildung 8.2: Das *Card-Widget* wird zweimal mal neu gezeichnet

Ohne die Änderungen in diesem Schritt zeigt der *Flutter Performance*-Tab, dass sich bei jeder Auswahl einer Option sechs `Card`-Elemente aktualisieren (Abb. 8.3). Das ist der Fall, weil es in Summe sechs Auswahlfelder gibt.

Abbildung 8.3: Das *Card-Widget* wird siebenmal neu gezeichnet

9 Schritt 6

In diesem Schritt soll das Formular um Mehrfachauswahlfelder erweitert werden. Im Speziellen handelt es sich um das Auswahlfeld *Nebenziele* (Abb. 9.1).

Abbildung 9.1: Die Eingabemaske in Schritt 6

Es beinhaltet die gleichen Auswahloptionen wie das Auswahlfeld *Hauptzielsetzung* (Abb. 9.2).

Abbildung 9.2: Im Selektionsbildschirm für die *Nebenziele* können mehrere Optionen gewählt werden

9.1 Integrationstest erweitern

Zunächst wird der Integrationstest um die Auswahl der Nebenziele erweitert (Listing 9.1).

Listing 9.1: Der Integrationstest klickt die Karte für die *Nebenziele* und selektiert darin 2 Optionen

Zu diesem Zweck löst der Test nach der Auswahl der Hauptzielsetzung (Z. 118-119) nun einen Klick auf die Selektionskarte für die Nebenzielsetzung aus (Z. 121). Dadurch öffnet sich der Auswahlbildschirm, in welchem die Option *Bodenschutz* (Z. 122) und anschließend die Option *Klima* (Z. 123) gewählt wird. Mit Auswahl der letzten Option und durch die damit verbundene Übergabe des Arguments `true` für den optionalen Parameter `tabConfirm` wird der Auswahlbildschirm umgehend wieder geschlossen. Anschließend erfolgt erneut das Speichern der Maßnahme (Z. 125-126).

Anders als bei den bisherigen Schlüssel-Werte-Paaren innerhalb des Objektes `'massnahmenCharakteris` kann der Wert der Nebenziele nicht als einzelner String gespeichert werden (Listing 9.2).

Bei dem Inhalt der Mehrfachauswahlfelder handelt es sich schließlich um eine Auflistung mehrerer Werte. Sie wird im erwarteten JSON-Dokument als Array-Literal codiert (Z. 140-143).

Listing 9.2: Der Integrationstest überprüft im *JSON*-Dokument den Schlussel *nebenziele*

9.2 Hinzufügen der Menge der Nebenziele

Für die Menge der Nebenziele müssen keine weiteren Auswahloptionen hinzugefügt werden. Es werden die gleichen Optionen verwendet, die auch bei der Menge mit dem Namen *Hauptzielsetzung Land* zum Einsatz kommen (Listing 9.3, Z. 123-124).

Listing 9.3: Die Menge *nebenzielsetzungLandChoices*

9.3 Aktualisierung des *Models*

Um die Liste der Nebenziele im Wertetyp `MassnahmenCharakteristika` einzufügen, kann der Datentyp `BuiltSet` verwendet werden (Listing 9.4, Z. 77). Die Getter-Methode

Listing 9.4: Die Nebenziele werden dem Wertetyp *massnahmenCharakteristika* hinzugefügt

`nebenziele` bedarf keiner Null-Zulässigkeit, da das Nicht-Vorhandensein von Werten darüber erreicht werden kann, dass die Menge leer ist.

9.4 Aktualisierung der Übersichtstabelle

Für das Einfügen der Überschrift in der Übersichtstabelle gibt es keine Unterschiede zum bisherigen Vorgehen. Die Überschrift wird nach der Spaltenüberschrift für die *Hauptzielsetzung* eingefügt (Listing 9.5, Z. 28).

Die Anzeige der Werte in den TabellenZellen ist dagegen unterschiedlich (Listing 9.6). Dieses Mal handelt es sich um die Aufzählung von mehreren Werten, weshalb ein `Column`-Widget die einzelnen Einträge untereinander auflistet (Z. 46-49). Jedes Element des `BuiltSet` `nebenziele` (Z. 47) wird über die Methode `map` jeweils in ein Element des Widgets `Text` konvertiert (Z. 48).

Listing 9.5: Die *Nebenziele* werden dem Tabellenkopf hinzugefügt

Listing 9.6: Die Nebenziele werden dem Tabellenkörper hinzugefügt

9.5 Aktualisierung des *ViewModels*

Die Nebenziele werden – erneut Mit dem Datentyp BuiltSet – im *ViewModel* hinzugefügt (Listing 9.7).

Listing 9.7: Die Nebenziele werden dem *ViewModel* hinzugefügt

Der benannte Konstruktor `seeded` initialisiert die Instanzvariable mit einer leeren Menge (Z. 20). Dafür wird der parameterlose Konstruktor von `BuiltSet` aufgerufen (Z. 21). Dadurch unterscheidet sich das `BehaviorSubject` von den anderen im *ViewModel* und muss dementsprechend bei der Konvertierung zwischen *Model* in *ViewModel* gesondert behandelt werden.

Bei Konvertierung von *Model* in *ViewModel* sind für alle Auswahloptionen – genau wie in den Schritten zuvor – jeweils nur die Abkürzungen verfügbar. Die Liste der gespeicherten Abkürzungen der Nebenziele muss dementsprechend zuerst in eine Menge von Auswahloptionen konvertiert werden, bevor sie dem `BuiltSet` übergeben werden kann (Listing 9.8). Die Methode `map` löst das Problem, indem sie die ihr als Argument übergebene Funktion für jede Abkürzung in der Menge *Nebenziele* aufruft (Z. 65). Die übergebene anonyme Funktion konvertiert die Abkürzung in die zugehörige Auswahloption. Die resultierende Menge kann dem Konstruktor von `BuiltSet` übergeben werden (Z. 64-65).

Ähnlich verhält es sich bei der Umwandlung des *ViewModels* in das *Model* (Listing 9.9).

In diesem Fall muss die Menge der Auswahloptionen der *Nebenziele* in die entsprechenden Abkürzungen umgewandelt werden, bevor sie im *Model* gespeichert wird. Die Methode `map` er hält zu diesem Zweck erneut eine anonyme Funktion, welche die Abkürzung der Auswahloptionen abfragt (Z. 81). Die resultierende Menge wird als Parameter dem Konstruktor `SetBuilder` übergeben (Z. 80-81). Der `SetBuilder` wiederum kümmert sich um das Bauen des `BuiltSet`, sobald ein Objekt des Typs Massnahme gebaut wird.

Listing 9.8: Konvertierung des *Models* in das *ViewModel* für die *Nebenziele*

Listing 9.9: Konvertierung des *ViewModels* in das *Model* für die *Nebenziele*

9.6 Aktualisierung der Eingabemaske

Unterhalb des Auswahlfeldes für das Hauptziel wird die Selektionkarte für die Nebenziele eingefügt (Listing 9.10).

Listing 9.10: Der Aufruf von *buildMultiSelectionCard* für die Menge *nebenzielsetzungLandChoices*

Allerdings handelt es sich dieses Mal um ein Mehrfachauswahlfeld, weshalb eine neue Methode namens `buildMultiSelectionCard` aufgerufen wird (Z. 215-217).

Da nun zwei Methoden zum Erstellen von Elementen des Widgets `SelectionCard` existieren, ist es sinnvoll, den Quellcode zu refaktorisieren, um redundanten Code zu vermeiden.

Innerhalb der bereits vorhandenen Methode `buildSelectionCard` wird die Routine, welche für die Validierung das Formulares genutzt wird, in eine neue Methode namens `validateChoices` (Listing 9.11, Z. 123-128) ausgelagert.

Sie bekommt die Attribute für den Namen der Menge (Z. 124), die zu validierenden Optionen (Z. 125-127) und schließlich die bisher ausgewählten Optionen aller Auswahlfelder (Z. 128) übergeben. Die ausgelagerte Funktion ist in Anhang **??** in Listing **??** auf Seite **??** zu finden.

Für die Erstellung der Mehrfachauswahlfelder ist die Methode `buildMultiSelectionCard` zuständig (Listing 9.12).

Das übergebene `selectionViewModel` unterstützt mit dem Typometer `BuiltSet` die Auswahl von mehreren Auswahloption (Z. 146). Bei `selectionViewModel` handelt es sich bereits um eine Menge. Für die Validierung `150` sowie für die Übergabe des initialen Wertes an den Konstruktor der `SelectionCard` (Z. 157) ist eine Umwandlung in eine Menge daher nicht mehr nötig. Dem Konstruktor `SelectionCard` wird weiterhin über den Parameter `multiSelection` mitgeteilt, dass mehr als eine Auswahl zum gewählt werden darf (Z. 154). Die Methoden `onSelect` und `onDeselect` ersetzen nun nicht mehr den aktuell gespeicherten Wert über eine einfache Zuweisung. Sie nutzen stattdessen die Methode `rebuild` des `BuiltSet` um ein Element mit Hilfe von `add` hinzuzufügen (Z. 160) bzw. mit `remove` Elemente zu entfernen (Z. 163). Der Methodenaufruf `rebuild` sorgt jedoch nicht für das Hinzufügen oder Löschen am Original-Objekt, sondern erstellt eine Kopie der

Listing 9.11: Die Methode *buildSelectionCard* mit dem Aufruf der ausgelagerten Funktion *validateChoices*

Listing 9.12: Die Methode *buildMultiSelectionCard*

Liste mit der gewünschten Änderungen. Deshalb erfolgt eine Zuweisung der Kopie zum Wert des `BehaviorSubject`-Objekts, was wiederum das Auslösen eines neuen Ereignisses bewirkt (Z. 158,161).

9.7 Aktualisierung der Selektionskarte

Diese Selektionskarte wird um die Instanzvariable `multiSelection` erweitert (Listing 9.13, Z. 17), dessen Wert im Konstruktor übergeben wird (Z. 27) aber auch ausgelassen werden kann, da der Standardwert `false` angegeben ist.

Listing 9.13: Die Klasse *SelectionCard* erhält die Instanzvariable *multiSelection*

Die Rückruffunktion `onChanged` des CheckboxListTile unterscheidet schließlich zwischen Mehrfach- und Einzel-Selektion (Listing 9.14). Sollte `multiSelection` mit `true` gesetzt sein (Z. 133), so erstellt die Methode `rebuild` von `BuiltSet` eine Kopie des aktuellen *ViewModels* der Selektionen. In der anonymen Funktion, welche für die Manipulationen an der Kopie genutzt wird, wird in einer Fallunterscheidung überprüft, ob das angewählte Element bereits selektiert ist (Z. 136). Sollte das der Fall sein, so wird diese bereits selektierte Option, die nun erneut angewählt wurde, mit der Methode `remove` des Builder-Objekts aus dem `BuiltSet` entfernt (Z. 137). Anderenfalls war die Option nicht selektiert, weshalb sie mit der Methode `add` hinzugefügt wird.

Listing 9.14: Dem *CheckboxListTile* wird die Mehrfachselektion hinzugefügt

10 Schritt 7

Nachdem im letzten Schritt nun die Mehrfachauswahl für die *Nebenziele* hinzugefügt wurde, soll in diesem Schritt die Möglichkeit geschaffen werden, benutzerdefinierte Abhängigkeiten für Auswahloptionen anzugeben. Denn die *Nebenziele* haben mehrere besondere Voraussetzungen:

Sollte das *Hauptziel* nicht gesetzt sein oder die Option *keine Angabe/Vorgabe* oder *bitte um Unterstützung* enthalten, so ist es nicht sinnvoll, dass ein tatsächliches *Nebenziele* gewählt wird. In diesem Fall kommen wiederum nur die Werte *keine Angabe/Vorgabe* oder *bitte um Unterstützung* infragen.

Sollte dagegen ein *Hauptziel* gesetzt sein, so darf das *Nebenziele* nicht die gleiche Option enthalten. Ist also beispielsweise für das *Hauptziel Biodiversität* ausgewählt (Abb. 10.1), so darf die Option im Selektionsbildschirm für die *Nebenziele* nicht zur Verfügung stehen (Abb. 10.2).

Abbildung 10.1: Im Selektionsbildschirm für das *Hauptziel* ist *Biodiversität* ausgewählt

Abbildung 10.2: Im Selektionsbildschirm für die *Nebenziele* kann *Biodiversität* nicht ausgewählt werden

Das bedeutet auch, dass wenn für die *Nebenziele* bereits *Biodiversität* ausgewählt war und anschließend für das *Hauptziel* ebenfalls *Biodiversität* gewählt wird, so muss die invalide Auswahl im Selektionsbildschirm der *Nebenziele* rot gekennzeichnet werden (Abb. 10.3).

Abbildung 10.3: Im Selektionsbildschirm für die *Nebenziele* wird die selektierte invalide Option *Biodiversität* rot gekennzeichnet

Diese Bedingungen lassen sich nicht mit Funktion `condition` der Basisklasse `Choice` lösen.

Denn das Argument `priorChoices`, welches der Funktion `condition` übergeben wird, enthält zwar alle Auswahloptionen, die im gesamten Formular gewählt worden, gibt aber keine Auskunft darüber, von welchem Auswahlfeld sie stammen. Sollte also die Auswahloptionen *Biodiversität* in der Menge der `priorChoices` auftauchen, so ist unklar, ob sie im Auswahlfeld für das *Hauptziel* oder dem der *Nebenziele* gewählt wurde.

Wenn der Selektionskarte aber eine benutzerdefinierte Funktion übergeben werden könnte, welche im aufrufenden Kontext auch Zugriff auf das *ViewModel* hat, so könnte direkt auf die Auswahlfelder zugegriffen werden.

Zu diesem Zweck wird der Klasse `SelectionCard` die Instanzvariable `choiceMatcher` hinzugefügt (Listing 10.1, Z. 27). Ein Parameter des gleichen Namens wird den Hilfsmethoden `buildSelectionCard` und `buildMultiSelectionCard` welche ihn unverändert an den Konstruktor der Klasse `SelectionCard` weitergeleitet. Die entsprechenden Listing sind in Anhang **??** auf den Seiten **??** und **??** zu finden.

Listing 10.1: Der *choiceMatcher* wird der Klasse *SelectionCard* hinzugefügt

Der initialisierende Wert kann im Konstruktor gesetzt (Z. 41), aber auch ausgelassen werden, da er nicht mit dem `required`-Schlüsselwort gekennzeichnet und damit nicht verpflichtend ist. Doch aus diesem Grund kann der Parameter den Wert `null` annehmen, weshalb er mit dem Suffix `?` gekennzeichnet werden muss. In der Initialisierungsliste erfolgt die Initialisierung der Instanzvariable `choiceMatcher` (Z. 46). Sollte der im Konstruktor übergebene Parameter nicht `null` sein, so wird er der Instanzvariable zugewiesen. Ist der aber `null`, so sorgt die *If-null Expression* dafür, dass der Standardwert rechts von dem `??` zugewiesen wird: die Funktion `defaultChoiceMatcherStrategy 46`. Diese Funktion kapselt die Überprüfung der Abhängigkeiten – welche die Auswahloption und untereinander haben – so wie sie in den letzten Schritten durchgeführt wurde (Z. 16-18). Ihr wird die zu überprüfende Auswahloption `choice`, sowie die Menge `priorChoices` – die mit allen bisher ausgewählten Auswahloptionen im Formular gefüllt ist – übergeben (Z. 16). Die Auswahloption `choice` ruft – wie zuvor auch – die Methode `conditionMatches` auf und übergibt ihr das Objekt `priorChoices` (Z. 17). Diese Implementierung soll immer dann verwendet werden, wenn kein benutzerdefinierter `choiceMatcher` übergeben wurde. An dem Namen `defaultChoiceMatcherStrategy` wird offensichtlich, um welches Entwurfsmuster es sich hierbei handelt: das *Strategie*-Entwurfsmuster.

Strategie-Entwurfsmuster Das *Strategie*-Entwurfsmuster ist ein Verhaltensmuster der Gang of Four. Es erlaubt Algorithmen zu kapseln und auszutauschen[1]. Abbildung **??** zeigt das UML-Diagramm des *Strategie*-Entwurfsmusters.

Die Typdefinition `ChoiceMatcher` (Z. 13) kann nach dem *Strategie*-Entwurfsmuster als die Schnittstelle namens *Strategie* interpretiert werden. Sie definiert, welche Voraussetzung an die Schnittstelle gegeben ist. In diesem Fall ist die Voraussetzung, dass es sich um eine Funktion mit dem Rückgabewert `bool` handelt, der als erstes Argument eine Auswahloption – der Parameterbezeichner lautet `choice` – und als zweites Argument eine Menge von Auswahloptionen – der Parameterbezeichner ist `priorChoices` – übergeben wird. Sollte der Parameter `choiceMatcher` gesetzt sein, so tauscht er die standardmäßig

[1] Vgl. Gamma u. a., *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, S. 373.

genutztes Strategie `defaultChoiceMatcherStrategy` durch die benutzerdefinierte Strategie aus (Z. 46). Beide werden nach dem *Strategie*-Entwurfsmuster als *konkrete Strategien* bezeichnet. Im Entwurfsmuster gibt es noch den Akteur *Kontext*, wobei es sich um die aufrufende Klasse handelt, welche die Strategien verwendet. In diesem Fall ist das die *Klasse* `SelectionCard`. Abbildung **??** zeigt das UML-Diagramm der konkreten Implementierung des *Strategie*-Entwurfsmusters für die *Strategie* `ChoiceMatcher`. Da sich bei der konkreten Strategie für das Auswahlfeld der *Nebenziele* um eine anonyme Funktion handelt, wurde sie zum besseren Verständnis im UML-Diagramm *nebenzieleChoiceMatcherStrategy* genannt.

Im Diagramm ist ebenfalls der *View* `MassnahmenDetailScreen` enthalten, denn er verwendet die konkrete Strategie `defaultChoiceMatcherStrategy` für die Validierung (Listing 10.2).

Listing 10.2: Die Methode *buildSelectionCard* in Schritt 7

Sollte nämlich ein Argument für den Parameter `choiceMatcher` übergeben werden (Z. 122), so wird es auch für die Validierung verwendet (Z. 130). Ist das Argument aber nicht gesetzt und damit `null`, so sorgt die *If-null Expression* dafür, dass die `defaultChoiceMatcherStrategy` für die Validierung verwendet wird.

Außerdem erstellt `MassnahmenDetailScreen` die konkrete Strategie *nebenzieleChoiceMatcherStrategy*, wie in Listing 10.3 zu sehen ist.

Listing 10.3: Dem Aufruf von *buildMultiSelectionCard* für die Menge *nebenzielsetzungLandChoices* wird ein benutzerdefinierter *choiceMatcher* übergeben

Der Aufruf `buildMultiSelectionCard` wird um die Übergabe einer anonymen Funktion für den Parameter `choiceMatcher` erweitert (Z. 224-239). In der ersten Fallunterscheidung wird überprüft, ob die gewählte Option ein tatsächliches *Nebenziele* ist (Z. 225). Dies kann über die Getter-Methode `hasRealValue` abgefragt werden. Ist dies nicht der Fall, so handelt es sich um die Auswahloptionen *keine Angabe/Vorgabe* bzw. *bitte um Unterstützung*, weshalb `true` zurückgegeben werden kann (Z. 237), da diese Auswahloptionen immer erlaubt sind. Sollte sich dagegen um ein tatsächliches *Nebenziele* handeln, so überprüft die nächste Fallunterscheidung, ob das *Hauptziel* entweder nicht gesetzt ist oder mit einem nicht tatsächlichen *Hauptziel* belegt ist (Z. 226-228)., Dazu wird die Getter-Methode `hasNoRealValue` benutzt, welche als Gegenteil zu `hasRealValue` fungiert, und dementsprechend `true` zurückgibt wenn die Auswahloption entweder *keine Angabe/Vorgabe* oder *bitte um Unterstützung* ist (Z. 226-228). Sollte das *Hauptziel* keinen tatsächlichen Wert einer Zielsetzung enthalten, dann ist die Wahl eines oder mehrerer *Nebenziele* nicht sinnvoll. Waren beide zuvorigen Bedingungen nicht wahr, so steht bereits fest, dass sowohl das *Hauptziel*, als auch den *Nebenziele* gesetzt sind und weder die Option *keine Angabe/Vorgabe* oder *bitte um Unterstützung* enthalten. Nun soll eine letzte Fallunterscheidung überprü-

fen, ob das *Nebenziele* bereits im *Hauptziel* gesetzt ist (Z. 230-321). Das ist nicht erlaubt, weshalb `false` zurückgegeben werden soll (Z. 232). Anderenfalls sind alle Bedingungen erfüllt und `true` kann zurückgegeben werden.

An diesem Beispiel wird auch offensichtlich, welchen Nutzen die Generalisierung der Klasse `SelectionCard` über den Typ Parameter `ChoiceType` hat . Über eine Reihe von Methoden- und Konstruktoraufrufen gelangt das Typargument `ZielsetzungLandChoice` in die Klasse `SelectionCard` und somit auch zu der Instanzvariable `choiceMatcher`. Zuerst wird es der Methode `buildMultiSelectionCard` übergeben (Z. 221). Mit dem Konstruktoraufruf `SelectionCard<ChoiceType>` wird es an die Selektionskarte weitergereicht (Siehe Listing **??** in Zeile 157 in Anhang **??** auf Seite **??**. Schließlich erhält die Instanzvariable das Typargument über die Deklaration `ChoiceMatcher<ChoiceType> choiceMatcher` (Listing 10.1, Z. 31). Der Typparameter wird durch das Typargument ersetzt. Somit hat `choiceMatcher` dann den Typ `ChoiceMatcher<ZielsetzungLandChoice>`. Damit handelt es sich also auch bei dem ersten Parameter `choice` der anonymen Funktion – die dem Parameter `choiceMatcher` übergeben wird (Listing 10.3, Z. 224) – um den Typ `ZielsetzungLandChoice`. Aus diesem Grund können die Methoden `hasRealValue` (Z. 225) und `hasNoRealValue` (Z. 228) auf dem Objekt `choice` aufrufen werden, obwohl sie nur Teil der Klasse `ZielsetzungLandChoice` aber nicht der Basisklasse `Choice` sind. Ohne Parametrisierung über den Typ müsste das Objekt `choice` in einen anderen Typen umgewandelt werden. Doch nach dieser Typumwandlung könnte ein Laufzeitfehler geschehen, sollte es sich bei dem Objekt tatsächlich nicht um den gewünschten Typ handeln. Durch die Generalisierung der Klassen und die Angabe des Typparameters ist das Vorhandensein des richtigen Typs garantiert und keine Typumwandlung nötig.

Die beiden neuen Methoden sind in Listing **??** zu sehen.

`hasRealValue` vergleicht, ob der aktuelle Wert weder *keine Angabe/Vorgabe* noch *bitte um Unterstützung* ist (Z. 201). `hasNoRealValue` ruft dagegen intern `hasRealValue` auf und negiert Wert (Z. 203).

Überall dort, wo zuvor der Ausdruck `choice.conditionMatches(priorChoices)` verwendet wurde, muss nun der Aufruf des `choiceMatcher` erfolgen. So zum Beispiel der *Stream*, welcher die Validität der Auswahlfelder prüft (Listing 10.4).

Listing 10.4: Der *Stream validityChanged* in Schritt 7

Alle Vorkommnisse, die durch den neuen Ausdruck ersetzt werden, sind im Anhang **??** auf den Seiten **??** bis **??** zu finden.

11 Diskussion

11.1 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 Boilerplate-Code 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 dem Parameter `child` übergeben

[1] Vgl. Google LLC, *Adding interactivity to your Flutter app*.
[2] Vgl. Google LLC, *Provider - A recommended approach*.

wird, und darüber hinaus allen Kinderelementen 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] Google LLC, *Build reactive mobile apps with Flutter (Google I/O '18)*, 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.

11.2 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.

12 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 Frontend-Framework ist. Die *Stream*-Transformationen aus der *Dart*-Standardbibliothek und aus *RxDart* haben ihre Äquivalente in der Bibliothek *RxJS*. https://www.learnrxjs.io/learn-rxjs/operators/filtering

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 Webportal 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 11.1 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