

ENTWICKLUNG EINER FORMULARANWENDUNG MIT
KOMPATIBILITÄTSVALIDIERUNG DER EINFACH- UND
MEHRFACHAUSWAHL-EINGABEFELDER

Vorgelegt von:

Alexander Johr

Erstprüfer: Prof. Jürgen Singer Ph.D.

Zweitprüfer: Prof. Daniel Ackermann

Datum: 31.08.2021

THEMA UND AUFGABENSTELLUNG DER MASTERARBEIT
MA AI 29/2021

FÜR HERRN ALEXANDER JOHR

ENTWICKLUNG EINER FORMULARANWENDUNG MIT
KOMPATIBILITÄTSVALIDIERUNG DER EINFACH- UND
MEHRFACHAUSWAHL-EINGABEFELDER

Das Thünen-Institut für Ländliche Räume wertet Daten zu Maßnahmen auf landwirtschaftlich genutzten Flächen aus. Dafür müssen entsprechende Maßnahmen bundesweit mit Zeitbezug auswertbar sein und mit Attributen versehen werden. Um die Eingabe für die Wissenschaftler des Instituts zu beschleunigen und um fehlerhafte Eingaben zu minimieren, soll eine spezielle Formularanwendung entwickelt werden. Neben herkömmlichen Freitextfeldern beinhaltet das gewünschte Formular zum Großteil Eingabefelder für Einfach- und Mehrfachauswahl. Je nach Feld kann die Anzahl der Auswahloptionen mitunter zahlreich sein. Dem Nutzer sollen daher nur solche Auswahloptionen angeboten werden, die zusammen mit der zuvor getroffenen Auswahl sinnvoll sind.

Im Wesentlichen ergibt sich die Kompatibilität der Auswahloptionen aus der Bedingung, dass für dasselbe oder ein anderes Eingabefeld eine Auswahlmöglichkeit gewählt bzw. nicht gewählt wurde. Diese Bedingungen müssen durch Konjunktion und Disjunktion verknüpft werden können. In Sonderfällen muss ein Formularfeld jedoch auch die Konfiguration einer vom Standard abweichenden Bedingung ermöglichen. Wird dennoch versucht, eine deaktivierte Option zu selektieren, wäre eine Anzeige der inkompatiblen sowie der stattdessen notwendigen Auswahl ideal.

Die primäre Zielplattform der Anwendung ist das Desktop-Betriebssystem Microsoft Windows 10. Idealerweise ist die Formularanwendung auch auf weiteren Desktop-Plattformen sowie mobilen Endgeräten wie Android- und iOS-Smartphones und -Tablets lauffähig. Die Serialisierung der eingegebenen Daten genügt dem Institut zunächst als Ablage einer lokalen Datei im JSON-Format.

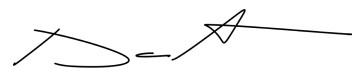
Die Masterarbeit umfasst folgende Teilaufgaben:

- Analyse der Anforderungen an die Formularanwendung
- Evaluation der angemessenen Technologie für die Implementierung
- Entwurf und Umsetzung der Übersichts- und Eingabeoberfläche
- Konzeption und Implementierung der Validierung der Eingabefelder
- Entwicklung von automatisierten Testfällen zur Qualitätskontrolle
- Bewertung der Implementierung und Vergleich mit den Wunschkriterien

Digital unterschrieben von
Juergen K. Singer
o= Hochschule Harz,
Hochschule fuer
angewandte
Wissenschaften, l=
Wernigerode
Datum: 2021.03.23 12:30:
26 MEZ



Prof. Jürgen Singer Ph.D.
1. Prüfer



Prof. Daniel Ackermann
2. Prüfer

Inhaltsverzeichnis

Abbildungsverzeichnis	9
Tabellenverzeichnis	11
Listingverzeichnis	13
I. Einleitung und Gliederung	19
1. Einleitung	21
1.1. Problemstellung	21
1.2. Gliederung	22
II. Vorbereitung	25
2. Technologie Auswahl	27
2.1. Trendanalyse	27
2.1.1. <i>Stack Overflow</i> -Umfrage	27
2.1.2. Google Trends	28
2.1.3. Frameworks mit geringer Relevanz	28
2.1.4. Frameworks mit sinkender Relevanz	31
2.1.5. Frameworks mit steigender Relevanz	31
2.2. Vergleich von <i>React Native</i> und <i>Flutter</i>	34
2.2.1. Vergleich zweier minimaler Beispiele für Formulare und Validierung .	34
2.2.2. Automatisiertes Testen	38
2.3. Fazit und Begründung der Auswahl	41
3. Grundlagen	43
3.1. Flutter	43
3.1.1. <i>Stateful Widgets</i>	44
3.1.2. <i>Stateless Widgets</i>	45
3.1.3. <i>Inherited Widgets</i>	45
3.2. <i>Dart</i> Grundlagen	45
3.2.1. <i>AOT</i> und <i>JIT</i>	46
3.2.2. <i>Set</i> - und <i>Map</i> -Literale	47
3.2.3. Typen ohne Null-Zulässigkeit	49

3.2.4. Typen mit Null-Zulässigkeit	49
3.2.5. Asynchrone Programmierung	52
4. Konzeption	57
4.1. Der Übersichtsbildschirm	57
4.2. Die Eingabemaske	58
4.3. Der Selektionsbildschirm	59
III. Implementierung	61
5. Schritt 1 – Grundstruktur der Formularanwendung	63
5.1. Auswahloptionen hinzufügen	64
5.2. Serialisierung einer Maßnahme	68
5.2.1. Unittest der Serialisierung einer Maßnahme	72
5.2.2. Unittest der Deserialisierung einer Maßnahme	74
5.3. Serialisierung der Maßnahmenliste	76
5.3.1. Unittest der Serialisierung der Maßnahmenliste	76
5.4. Der Haupteinstiegspunkt	78
5.4.1. Das <i>Model-View-ViewModel</i> -Entwurfsmuster	78
5.4.2. <i>Service Locator</i> und <i>Dependency Inection</i>	79
5.4.3. Der Service für den applikationsübergreifenden Zustand	80
5.5. Speichern der Maßnahmen in eine <i>JSON</i> -Datei	82
5.6. Abhängigkeit zum Verwalten der Maßnahmen	84
5.7. Übersichtsbildschirm der Maßnahmen	86
5.7.1. Auflistung der Maßnahmen im Übersichtsbildschirm	87
5.8. Das <i>MassnahmenTable-Widget</i>	90
5.9. Das <i>ViewModel</i>	94
5.10. Eingabeformular	96
5.10.1. Ausgabe der Formularfelder	98
5.10.2. Eingabefeld für den Maßnahmentitel	99
5.10.3. Speicher-Routine	100
5.11. Das <i>SelectionCard-Widget</i>	100
5.11.1. Bildschirm für die Auswahl der Optionen	103
5.12. Integrationstest zum Test der Oberfläche	106
5.12.1. Generierung des <i>Mocks</i>	106
5.12.2. Hilfsfunktionen für den Integrationstest	108
5.12.3. Test des Übersichtsbildschirms	111
5.12.4. Test der Eingabemaske	112
5.12.5. Test der Speicherung	112
6. Schritt 2 - Refaktorisierung zum Hinzufügen weiterer Eingabefelder	115
6.1. Integrationstest erweitern	116
6.2. Hinzufügen der Auswahloptionen	118

6.3.	Aktualisierung des <i>Models</i>	118
6.4.	Aktualisierung der Übersichtstabelle	119
6.5.	Aktualisierung des <i>ViewModels</i>	120
6.5.1.	Aktualisierung der <i>Setter</i> -Methode	120
6.5.2.	Aktualisierung der <i>Getter</i> -Methode	121
6.6.	Aktualisierung der Eingabemaske	121
7.	Schritt 3 - Implementierung der grundlegenden Validierungsfunktion	125
7.1.	Einfügen des <i>Form-Widgets</i>	125
7.2.	Validierung des Maßnahmentitels	126
7.3.	Validierung der Selektionskarte	128
7.4.	Speichern der Eingaben im Entwurfsmodus	130
8.	Schritt 4 - Kompatibilitätsvalidierung	135
8.1.	Hinzufügen der Bedingungen zu den Auswahloptionen	136
8.2.	Hinzufügen der Momentaufnahme aller ausgewählten Optionen im gesamten Formular	138
8.3.	Reagieren der Selektionskarte auf die ausgewählten Optionen	140
8.4.	Reagieren des Auswahlbildschirms auf die ausgewählten Optionen	142
8.4.1.	Hinzufügen der Momentaufnahme zur Validierung	144
9.	Schritt 5 - Laufzeitoptimierung	145
10.	Schritt 6 - Hinzufügen von Mehrfachauswahlfeldern	149
10.1.	Integrationstest erweitern	150
10.2.	Hinzufügen der Menge der Nebenziele	151
10.3.	Aktualisierung des <i>Models</i>	151
10.4.	Aktualisierung der Übersichtstabelle	152
10.5.	Aktualisierung des <i>ViewModels</i>	152
10.5.1.	Aktualisierung der <i>Setter</i> -Methode	153
10.5.2.	Aktualisierung der <i>Getter</i> -Methode	154
10.6.	Aktualisierung der Selektionskarte	154
10.7.	Aktualisierung des Selektionsbildschirms	155
10.8.	Aktualisierung der Eingabemaske	156
10.8.1.	Auslagerung der Validierungsfunktion	156
10.8.2.	Konfiguration der Selektionskarte für die Mehrfachauswahl	157
11.	Schritt 7 - Benutzerdefinierte Validierungsfunktionen	159
11.1.	Aktualisierung der Selektionskarte	160
11.1.1.	Strategie-Entwurfsmuster	162
11.2.	Aktualisierung der Eingabemaske	162

IV. Fazit	167
12. Diskussion	169
12.1. Reevaluation des Zustandsmanagements	169
12.2. Anzeige von fehlerhaften Teilkomponenten der Bedingungen von deaktivierten Auswahloptionen	173
13. Schlussfolgerung-und-Ausblick	175
Literatur	177
V. Anhang	181
A. Minimalistische <i>Flutter</i>-Formularanwendung	183
B. Minimalistische <i>React Native</i>-Formularanwendung	185
C. Schritt 1 Anhang	191
D. Schritt 2 Anhang	193
E. Schritt 3 Anhang	197
F. Schritt 4 Anhang	199
G. Schritt 6 Anhang	203
H. Schritt 7 Anhang	205
Eidesstattliche Erklärung	211

Abbildungsverzeichnis

2.1. Stimmen der Stack Overflow Umfrage von 2013 bis 2020	29
2.2. Suchinteresse der Frameworks mit geringer Relevanz	29
2.3. Stimmen für <i>Cordova</i> und <i>PhoneGap</i>	30
2.4. Stimmen für <i>Xamarin</i> und <i>Cordova</i>	31
2.5. Suchinteresse der Frameworks mit sinkender und steigender Relevanz	31
2.6. Stimmen gewünschter Frameworks: <i>React Native</i> , <i>Flutter</i> , <i>Xamarin</i> und <i>Cordova</i>	32
2.7. Stimmen verwendeter Frameworks: <i>React Native</i> , <i>Flutter</i> , <i>Xamarin</i> und <i>Cor-</i> <i>dova</i>	33
2.8. Gegenüberstellung der minimalistischen Flutter- und <i>React Native</i> -Formular- applikationen	36
4.1. Konzeption des Übersichtsbildschirms	57
4.2. Konzeption der Eingabemaske	58
4.3. Konzeption des Selektionsbildschirms	59
5.1. Der Übersichtsbildschirm in Schritt 1	63
5.2. Die Eingabemaske in Schritt 1	63
5.3. Der Selektionsbildschirm in Schritt 1	64
5.4. UML-Diagramm der Beziehungen zwischen den Bildschirmen und dem <i>App-</i> <i>State</i>	81
5.5. UML-Diagramm des <i>inversion of control pattern</i> für <i>MassnahmenMasterS-</i> <i>creen</i> und <i>MassnahmenTable</i>	93
6.1. Die Eingabemaske in Schritt 2	115
6.2. Der Übersichtsbildschirm in Schritt 2	116
7.1. Die Eingabemaske in Schritt 3	125
8.1. Der Selektionsbildschirm in Schritt 4	135
8.2. Die Eingabemaske in Schritt 2 mit einem selektierten invaliden Wert	135
8.3. Der Selektionsbildschirm in Schritt 4 mit einem selektierten invaliden Wert .	136
9.1. Das <i>Card-Widget</i> wird sechsmal neu gezeichnet	145
9.2. Das <i>Card-Widget</i> wird einmal neu gezeichnet	148
9.3. Das <i>Card-Widget</i> wird zweimal neu gezeichnet	148

10.1. Die Eingabemaske in Schritt 6	149
10.2. Im Selektionsbildschirm für die <i>Nebenziele</i> können mehrere Optionen gewählt werden	149
11.1. Im Selektionsbildschirm für das <i>Hauptziel</i> ist <i>Biodiversität</i> ausgewählt . . .	159
11.2. Im Selektionsbildschirm für die <i>Nebenziele</i> kann <i>Biodiversität</i> nicht ausgewählt werden	159
11.3. Im Selektionsbildschirm für die <i>Nebenziele</i> wird die selektierte invalide Option <i>Biodiversität</i> rot gekennzeichnet	160
11.4. <i>UML</i> -Diagramm des <i>Strategie</i> -Entwurfsmusters	162
11.5. <i>UML</i> -Diagramm des <i>Strategie</i> -Entwurfsmusters für den <i>ChoiceMatcher</i> . . .	163

Tabellenverzeichnis

2.1. Anzahl der Quelltextzeilen der minimalistischen <i>Flutter</i> und <i>React Native</i> - Formular-Applikationen.	37
--	----

Listingverzeichnis

3.1. <i>Java Swing</i> -Beispiel: Ein Button, welcher die Anzahl der Klicks anzeigt . . .	44
3.2. <i>Flutter</i> -Beispiel: Ein Button, welcher die Anzahl der Klicks anzeigt	44
3.3. Ein <i>Set</i>	47
3.4. Das <i>collection for</i> in einer Menge	48
3.5. Das <i>collection for</i> in einer Hashtabelle	48
3.6. <i>Collection if</i> in einer Liste	48
3.7. Fehlerhafter Zugriff auf eine Variable mit Null-Zulässigkeit	49
3.8. Zugriff auf eine Variable mit Null-Zulässigkeit durch <i>type promotion</i>	50
3.9. Fehlerhafter Zugriff auf eine Instanzvariable mit Null-Zulässigkeit	50
3.10. Überschreiben des Instanzattributs mit einer <i>Getter</i> -Methode	51
3.11. Erzwungener Zugriff auf eine Instanzvariable mit Null-Zulässigkeit	51
3.12. Zuweisung der Instanzvariablen zu einer lokalen Variablen	51
3.13. Der asynchrone Aufruf <i>readAsString</i>	52
3.14. Aufruf von <i>then</i> auf dem <i>Future</i> -Objekt	52
3.15. Aufruf der asynchronen Methode <i>readAsString</i> mit dem <i>await</i> -Schlüsselwort	54
3.16. Abhören eines <i>Streams</i>	54
5.1. Die Klasse <i>LetzterStatus</i>	65
5.2. Die Klasse <i>Choice</i>	65
5.3. Die Menge <i>letzterStatusChoices</i>	66
5.4. Die Klasse <i>Choices</i>	66
5.5. Abgeändertes <i>Live Template</i> für die Erstellung von <i>built_value</i> Boilerplate- Code in <i>Android Studio</i>	69
5.6. Der Wertetyp <i>Massnahme</i>	69
5.7. Der Wertetyp <i>Identifikatoren</i>	71
5.8. Der Wertetyp <i>LetzteBearbeitung</i>	71
5.9. Der Serialisierer für die Wertetypen <i>Massnahme</i> und <i>Storage</i>	72
5.10. Unittest der Serialisierung einer Maßnahme	73
5.11. Unittest der Deserialisierung einer Maßnahme	74
5.12. Instanzvariable <i>letzteBearbeitung</i> gibt einen <i>LetzteBearbeitungBuilder</i> zurück	75
5.13. Der Wertetyp <i>Storage</i>	76
5.14. Unittest der Serialisierung der Maßnahmenliste	77
5.15. Instanzvariable <i>massnahmen</i> gibt einen <i>SetBuilder</i> zurück	77
5.16. Der Haupteinstiegspunkt <i>MassnahmenFormApp</i>	79
5.17. Der Service <i>AppState</i> für den applikationsübergreifenden Zustand	81

5.18. Die Klasse <i>MassnahmenJsonFile</i>	83
5.19. Die Klasse <i>MassnahmenModel</i>	85
5.20. Die Struktur der Klasse <i>MassnahmenMasterScreen</i>	87
5.21. Die Ausgabe der Maßnahmen	89
5.22. Die Klasse <i>MassnahmenTable</i>	90
5.23. Die Hilfsmethode <i>_buildColumnHeader</i>	91
5.24. Die Hilfsmethode <i>_buildSelectableCell</i>	91
5.25. Die Typdefinition <i>OnSelectCallback</i>	92
5.26. Die Ausgabe der Maßnahmen	93
5.27. Die Klasse <i>MassnahmenFormViewModel</i>	95
5.28. Die Struktur des Bildschirms <i>MassnahmenDetailScreen</i>	97
5.29. Die Ausgabe der Formularfelder	99
5.30. Die Funktion <i>createMassnahmenTitelTextFormField</i> in Schritt 1	99
5.31. Die Funktion <i>saveRecordAndGoBackToOverviewScreen</i>	100
5.32. Die Klasse <i>SelectionCard</i>	101
5.33. Die <i>build</i> -Methode der Klasse <i>SelectionCard</i> in Schritt 1	103
5.34. Die Methode <i>createMultipleChoiceSelectionScreen</i>	105
5.35. Initialisierung des Integrationstests	107
5.36. Initialisierung des Integrationstests	107
5.37. Initialisierung des <i>Widgets</i> für den Integrationstest	108
5.38. Die Hilfsmethode <i>tabSelectionCard</i>	108
5.39. Die Hilfsmethode <i>tabConfirmButton</i>	110
5.40. Die Hilfsmethode <i>tabOption</i>	110
5.41. Die Hilfsmethode <i>fillTextFormField</i>	111
5.42. Der Button zum Kreieren einer Maßnahme wird ausgelöst	111
5.43. Der letzte Status wird ausgewählt	112
5.44. Der Maßnahmentitel wird eingegeben	112
5.45. Validierung des Testergebnisses	113
6.1. Der Integrationstest klickt 5 weitere Karten	116
6.2. Der Integrationstest überprüft im <i>JSON</i> -Dokument den Schlüssel <i>massnahmenCharakteristika</i>	117
6.3. Die Klassenvariable <i>aukm_ohne_vns</i> vom Typ <i>FoerderklasseChoice</i>	118
6.4. <i>massnahmenCharakteristika</i> wird dem Wertetyp <i>Massnahme</i> hinzugefügt	118
6.5. Der Wertetyp <i>Massnahmencharakteristika</i>	118
6.6. Die <i>Maßnahmencharakteristika</i> werden dem Tabellenkopf hinzugefügt	119
6.7. Die <i>Maßnahmencharakteristika</i> werden dem Tabellenkörper hinzugefügt	119
6.8. Die <i>Maßnahmencharakteristika</i> werden dem <i>ViewModel</i> hinzugefügt	120
6.9. Konvertierung des <i>Models</i> in das <i>ViewModel</i> für die <i>Maßnahmencharakteristika</i>	120
6.10. Konvertierung des <i>ViewModels</i> in das <i>Model</i> für die <i>Maßnahmencharakteristika</i>	121
6.11. Die Methode <i>buildSelectionCard</i>	122

6.12. Der Aufruf von <i>buildSelectionCard</i> für die Menge <i>letzterStatusChoices</i> . . .	122
6.13. Die <i>Maßnahmencharakteristika</i> Selektionskarten werden ergänzt	123
6.14. Die Hilfsfunktionen <i>buildSectionHeadline</i> und <i>buildSubSectionHeadline</i> . . .	123
7.1. Einfügen des <i>Form-Widgets</i>	125
7.2. Der <i>formKey</i> wird erstellt	126
7.3. Die Funktion <i>createMassnahmenTitelTextFormField</i> mit Validierung	127
7.4. Die Methode <i>buildSelectionCard</i> mit Validierung	128
7.5. <i>errorText</i> wird der <i>SelectionCard</i> hinzugefügt	129
7.6. <i>errorText</i> wird ausgegeben	129
7.7. Der <i>FloatingActionButton</i> zum Speichern der Maßnahmen im Entwurfsmodus	130
7.8. Die Fehlerbehandlung im <i>WillPopScope</i>	131
7.9. Die Funktion <i>saveDraftAndGoBackToOverviewScreen</i>	131
7.10. Die Funktion <i>inputsAreValidOrNotMarkedFinal</i>	132
7.11. Die Funktion <i>showValidationError</i>	133
8.1. Der Klassenvariablen <i>al</i> des Typs <i>ZielflaecheChoice</i> wird eine Bedingung hinzugefügt	136
8.2. Der Klassenvariablen <i>wald</i> des Typs <i>ZielflaecheChoice</i> wird eine Bedingung hinzugefügt	136
8.3. Der Klasse <i>Choices</i> wird die Instanzvariable <i>condition</i> hinzugefügt	137
8.4. Das <i>BehaviorSubject</i> <i>priorChoices</i>	139
8.5. Dem Konstruktor der <i>SelectionCard</i> wird das <i>BehaviorSubject</i> <i>priorChoices</i> übergeben	140
8.6. Die Klasse <i>SelectionCard</i> erhält die Instanzvariable <i>priorChoices</i>	141
8.7. Die <i>SelectionCard</i> reagiert auf Änderungen des Streams <i>priorChoices</i>	141
8.8. Der Selektionsbildschirm in Schritt 4	143
8.9. Die <i>validator</i> Funktion von <i>FormField</i> in Schritt 4	144
9.1. Der Klassenvariablen <i>ha</i> des Typs <i>ZielflaecheChoice</i> wird eine Bedingung hinzugefügt	146
9.2. Der <i>Stream</i> <i>validityChanged</i> in Schritt 5	147
10.1. Der Integrationstest klickt die Karte für die <i>Nebenziele</i> und selektiert darin 2 Optionen	150
10.2. Der Integrationstest überprüft im <i>JSON</i> -Dokument den Schlüssel <i>nebenziele</i>	150
10.3. Die Menge <i>nebenzielsetzungLandChoices</i>	151
10.4. Die <i>Nebenziele</i> werden dem Wertetyp <i>MassnahmenCharakteristika</i> hinzugefügt	151
10.5. Die <i>Nebenziele</i> werden dem Tabellenkopf hinzugefügt	152
10.6. Die <i>Nebenziele</i> werden dem Tabellenkörper hinzugefügt	152
10.7. Die <i>Nebenziele</i> werden dem <i>ViewModel</i> hinzugefügt	152
10.8. Konvertierung des <i>Models</i> in das <i>ViewModel</i> für die <i>Nebenziele</i>	153
10.9. Konvertierung des <i>ViewModels</i> in das <i>Model</i> für die <i>Nebenziele</i>	154
10.10. Die Klasse <i>SelectionCard</i> erhält die Instanzvariable <i>multiSelection</i>	154
10.11. Dem <i>CheckboxListTile</i> wird die Mehrfachselektion hinzugefügt	155

10.12	Der Aufruf von <i>buildMultiSelectionCard</i> für die Menge <i>nebenzielsetzungLandChoices</i>	156
10.13	Die Methode <i>buildSelectionCard</i> mit dem Aufruf der ausgelagerten Funktion <i>validateChoices</i>	156
10.14	Die Methode <i>buildMultiSelectionCard</i>	157
11.1.	Der <i>choiceMatcher</i> wird der Klasse <i>SelectionCard</i> hinzugefügt	161
11.2.	Die Methode <i>buildSelectionCard</i> mit dem Parameter <i>choiceMatcher</i>	163
11.3.	Der benutzerdefinierte <i>choiceMatcher</i> für die Menge <i>nebenzielsetzungLandChoices</i>	165
11.4.	Die <i>Getter</i> -Methoden <i>hasRealValue</i> und <i>hasNoRealValue</i>	165
11.5.	Der <i>Stream</i> <i>validityChanged</i> in Schritt 7	165
12.1.	Verwendung der Klasse <i>ChangeNotifier</i>	169
12.2.	Die <i>Widgets Provider</i> , <i>ChangeNotifierProvider</i> und <i>MultiProvider</i>	170
12.3.	Das <i>Widget Consumer</i>	170
12.4.	Die Klasse <i>CartBloc</i>	171
12.5.	Die vereinfachte Klasse <i>CartBloc</i>	172
A.1.	Validierungs-Funktionen der minimalistischen <i>Flutter</i> -Formularanwendung .	183
A.2.	Haupteinstiegspunkt der minimalistischen <i>Flutter</i> -Formularanwendung . . .	184
B.1.	Haupteinstiegspunkt der minimalistischen <i>React Native</i> -Formularanwendung	185
B.2.	Logo der minimalistischen <i>React Native</i> -Formularanwendung	186
B.3.	Validierungs-Funktionen der minimalistischen <i>React Native</i> -Formularanwendung	186
B.4.	Schnittstellen <i>Props</i> der minimalistischen <i>React Native</i> -Formularanwendung	187
B.5.	<i>Form</i> -Komponente der minimalistischen <i>React Native</i> -Formularanwendung .	188
B.6.	<i>Input</i> -Komponente der minimalistischen <i>React Native</i> -Formularanwendung .	189
C.1.	Unittest der Deserialisierung der Maßnahmenliste	191
D.1.	Die Menge <i>foerderklasseChoices</i>	193
D.2.	Die Menge <i>kategorieChoices</i>	194
D.3.	Die Mengen <i>zielflaecheChoices</i> und <i>zieleinheitChoices</i>	195
D.4.	Die Menge <i>hauptzielsetzungLandChoices</i>	196
E.1.	Die Methode <i>saveRecord</i>	197
F.1.	Die Klasse <i>KategorieChoice</i> in Schritt 4	199
F.2.	Die Klasse <i>ZielflaecheChoice</i> in Schritt 4	200
F.3.	Die Klasse <i>ZieleinheitChoice</i> in Schritt 4	201
G.1.	Die Funktion <i>validateChoices</i>	203
H.1.	Der <i>choiceMatcher</i> wird in der Methode <i>buildSelectionCard</i> hinzugefügt . .	205
H.2.	Der <i>choiceMatcher</i> wird in der Methode <i>buildMultiSelectionCard</i> hinzugefügt	206

H.3. Der <i>choiceMatcher</i> wird in der <i>build</i> -Methode der Klasse <i>SelectionCard</i> hinzugefügt	207
H.4. Der <i>choiceMatcher</i> wird in der Methode <i>createMultipleChoiceSelectionScreen</i> hinzugefügt	208
H.5. Der <i>choiceMatcher</i> wird in der Funktion <i>validateChoices</i> hinzugefügt	209

Teil I

EINLEITUNG UND GLIEDERUNG

1. Einleitung

Eine angenehme Erfahrung für den Nutzer einer Software entsteht unter anderem dann, wenn ihm die richtigen Informationen zur richtigen Zeit präsentiert werden.

In Formularen spielen Einfach- und Mehrfachauswahlfelder – 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 anzubieten, 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, welche 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 den 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* als 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 Konzeption der grafischen Benutzeroberfläche wird in Kapitel 4 durchgeführt.

Die Kapitel 5 bis 11 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.

In Kapitel 5 wird die grundlegende Struktur der Anwendung hergestellt. Kapitel 6 fügt Hilfsmethoden hinzu, welche das Hinzufügen weiterer Formularfelder in den folgenden Schritten vereinfachen wird.

In Kapitel 7 erhält die Anwendung die grundlegende Funktion, Felder zu validieren. Kapitel 8 erweitert die Validierung schließlich um die Bedingungen der Auswahloptionen. Als Konsequenz werden alle Formularfelder neu gezeichnet, sollte der Benutzer eine beliebige Auswahloption selektieren. Durch die Validierung geschieht es nach dem Neuzeichnen, dass invalide Auswahlfelder rot markiert werden. Die erforderlichen Änderungen, um nur die Auswahlfelder zu aktualisieren, die ihre Validität oder ihren eigenen Inhalt ändern, werden in Kapitel 9 hinzugefügt.

Kapitel 10 ergänzt die Möglichkeit, Mehrfachauswahlfelder zu verwenden. Kapitel 11 sorgt dafür, dass auch benutzerdefinierte Bedingungen für die Auswahlfelder hinterlegt werden können.

Die während der Entwicklung der Anwendung gesammelten Erkenntnisse werden in Kapitel 12 dargelegt. Kapitel 13 bewertet die Erkenntnisse, ergänzt sie um einen Ausblick und vergleicht die Ergebnisse der Entwicklung mit den Anforderungen.

Teil II

VORBEREITUNG

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 infrage kommenden Technologien identifiziert. Anschließend wird der Trend der Popularität dieser Technologien miteinander verglichen. Die daraus resultierenden Kandidaten sollen dann detaillierter untersucht werden. Im 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 Suchinteresse von Google Trends.

2.1.1. *Stack Overflow*-Umfrage

Die Internetplattform *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 Domainnamens 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 auch die

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

Angabe der aktuell genutzten und in Zukunft gewünschten Frontend-Technologien. *Stack Overflow* erstellt aus diesen gesammelten Daten Auswertungen und Übersichten. Die zugrunde liegenden 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.

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 Suchinteresse 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. 2.1). Auch das Suchinteresse auf Google ist für diese Frameworks äußerst gering. In Abbildung 2.2 wird das relative Suchinteresse von *NativeScript*, *Sencha*, *Appcelerator*, *Adobe PhoneGap* und *Apache Cordova* miteinander verglichen.

²Vgl. Stack Exchange, Inc., *Stack Overflow Insights / Stack Overflow Annual Developer Survey*.

³Vgl. Google LLC, *Google Trends-Hilfe / Häufig gestellte Fragen zu Google Trends-Daten*.

⁴Google LLC, *Google Trends-Hilfe / Häufig gestellte Fragen zu Google Trends-Daten*.

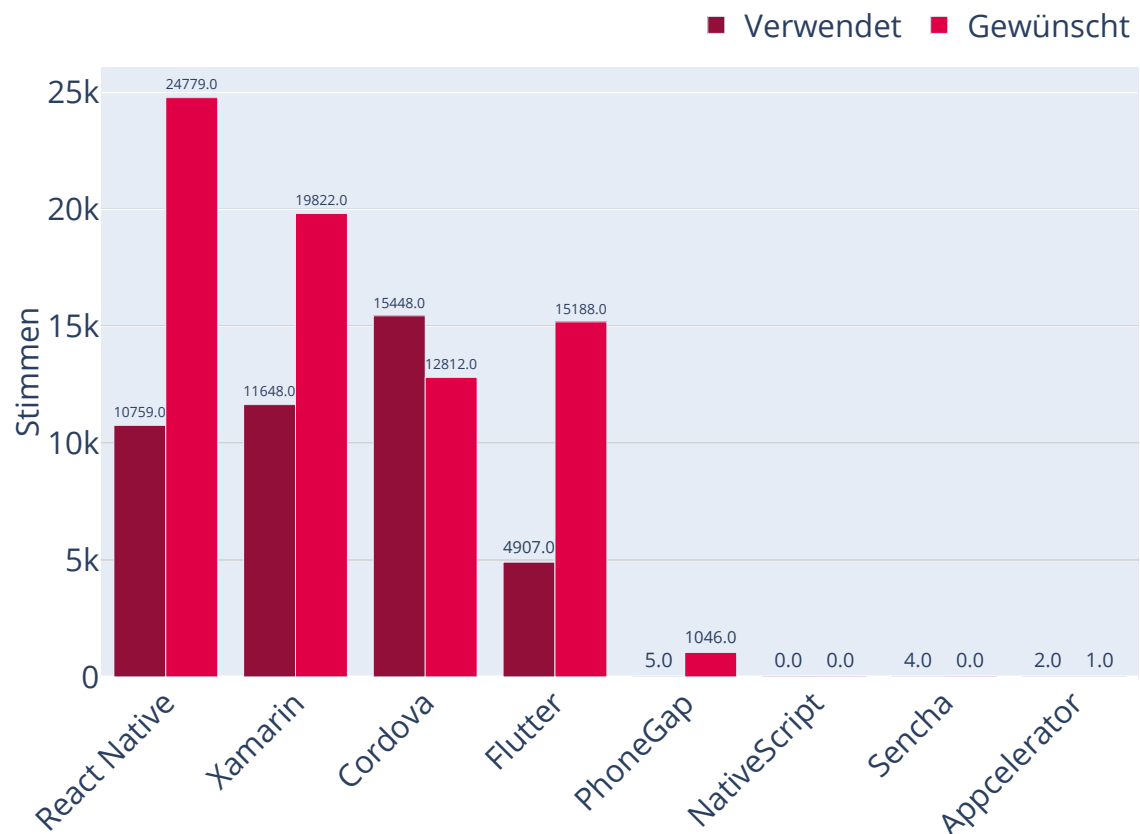


Abbildung 2.1.: Summe der Stimmen der *Stack Overflow*-Umfrage von 2013 bis 2020, Quelle: Eigene Abbildung, Notebook: [Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle: <https://insights.stackoverflow.com/survey>

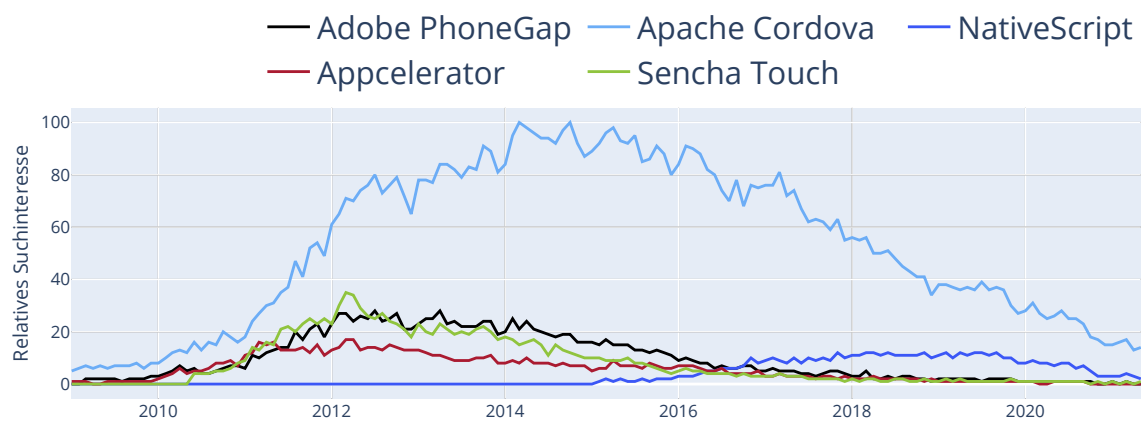


Abbildung 2.2.: Suchinteresse der Frameworks mit geringer Relevanz, Quelle: Eigene Abbildung, Notebook: [Charts/GoogleTrends/GoogleTrends.ipynb](#), Daten-Quelle: Google Trends

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* aufbaut⁵, welches bereits in den Ergebnissen vorkommt. *Adobe PhoneGap* taucht zwar in den Ergebnissen von 2013 mit 1043 Stimmen auf (Abb. 2.3), verliert jedoch in den Folgejahren mit weniger als 10 Stimmen abrupt an Relevanz.

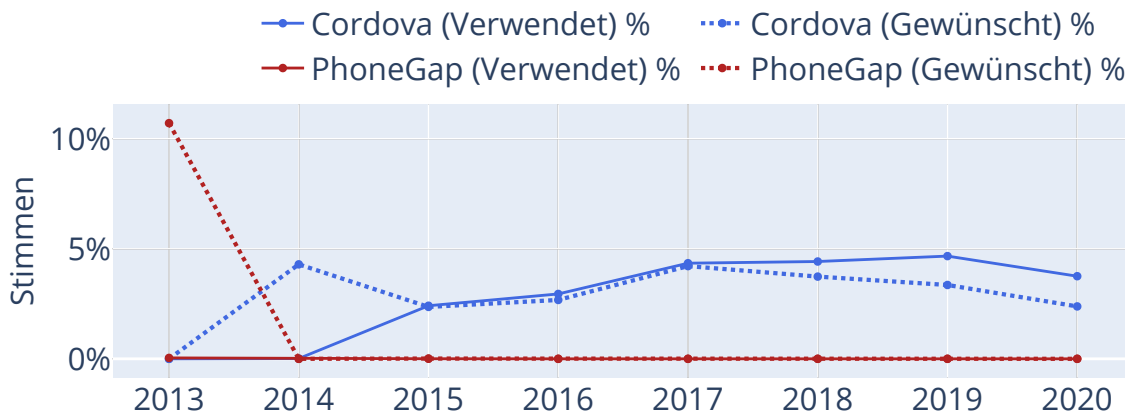


Abbildung 2.3.: Stimmen für *Cordova* und *PhoneGap*, Quelle: Eigene Abbildung, Notebook:[Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle:<https://insights.stackoverflow.com/survey>

Das stimmt nicht mit dem Suchinteresse auf Google überein, da *Adobe PhoneGap* dort erst ab 2014 anfängt, langsam an Relevanz zu verlieren (Abb. 2.2). 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.⁶ Auch *Adobe PhoneGap* baut auf *Apache Cordova* auf.⁷ Für diese Auswertung spielen diese verwandten Technologien eine untergeordnete Rolle, da sie auch in den Google Trends weit hinter *Apache Cordova* zurückbleiben.

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*.⁸

⁵Vgl. Lynch, *The Last Word on Cordova and PhoneGap*.

⁶Vgl. Stack Exchange, Inc., *Stack Overflow Insights / Stack Overflow Annual Developer Survey*.

⁷Vgl. Adobe Inc., *PhoneGap Docs / FAQ*.

⁸Vgl. Adobe Inc., *Update for Customers Using PhoneGap and PhoneGap Build*.

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.

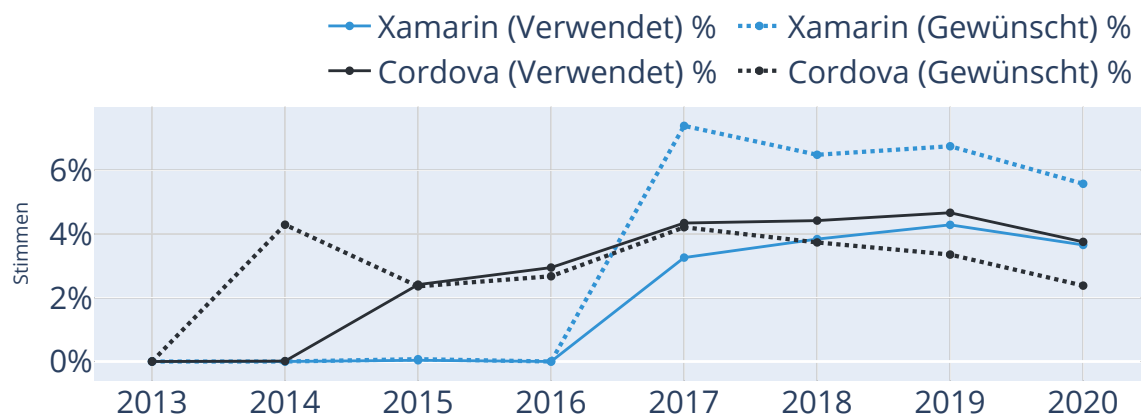


Abbildung 2.4.: Stimmen für *Xamarin* und *Cordova*, Quelle: Eigene Abbildung, Notebook:[Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle:<https://insights.stackoverflow.com/survey>

In Abbildung 2.5 ist noch einmal zu sehen, dass Google Trends die Erkenntnisse aus der *Stack Overflow*-Umfrage reflektiert; und es wird auch sichtbar, welche beiden Technologien möglicherweise der Grund für den Rückgang von *Xamarin* und *Cordova* sind.

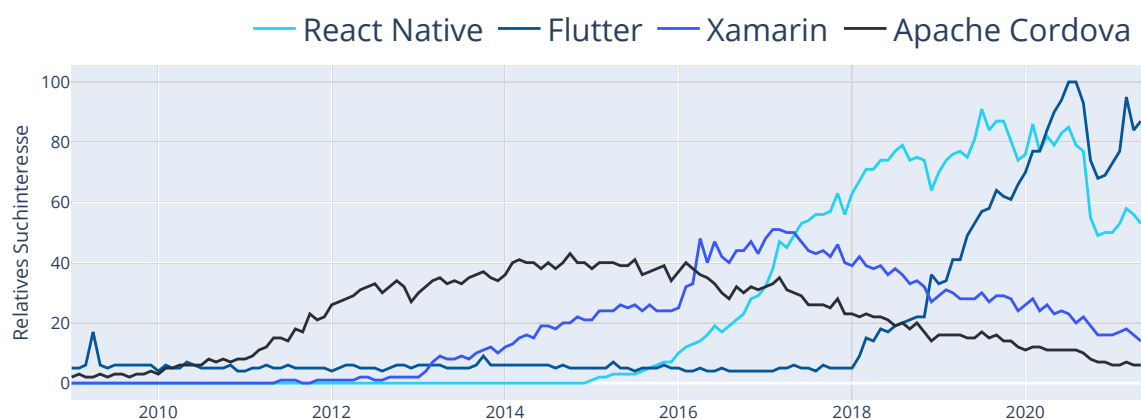


Abbildung 2.5.: Suchinteresse der Frameworks mit sinkender und steigender Relevanz, Quelle: Eigene Abbildung, Notebook:[Charts/GoogleTrends/GoogleTrends.ipynb](#), Daten-Quelle:Google Trends

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 Webframework handelt. Doch auch die Ergebnisse von Google Trends zeigen einen ähnlichen Verlauf für die Jahre 2019 und 2020 (Abb. 2.5).

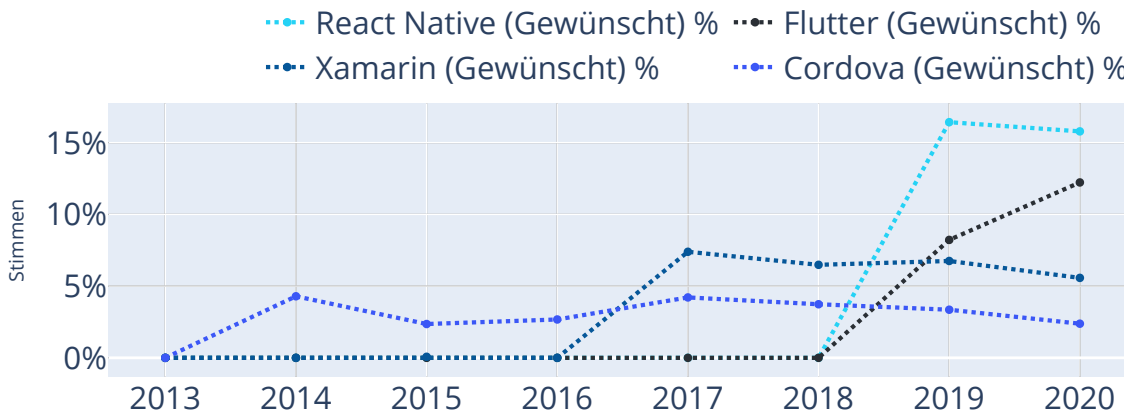


Abbildung 2.6.: Stimmen gewünschter Frameworks: *React Native*, *Flutter*, *Xamarin* und *Cordova*, Quelle: Eigene Abbildung, Notebook: [Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle: <https://insights.stackoverflow.com/survey>

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 Suchinteresse hat sich diese Entwicklung bereits vollzogen (Abb. 2.5). Auch in der Anzahl der Entwickler, welche *Flutter* einsetzen, ist ein steiler Aufwärtstrend zu beobachten (Abb. 2.7). Damit überholt *Flutter* die etablierten Frameworks *Xamarin* und *Cordova*. Gleichzeitig muss betrachtet werden, dass die Anzahl der Nutzer, die *React Native* als verwendete Technologie angegeben haben, noch immer höher ist und – wenn auch stagnierend – weiter steigt.

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

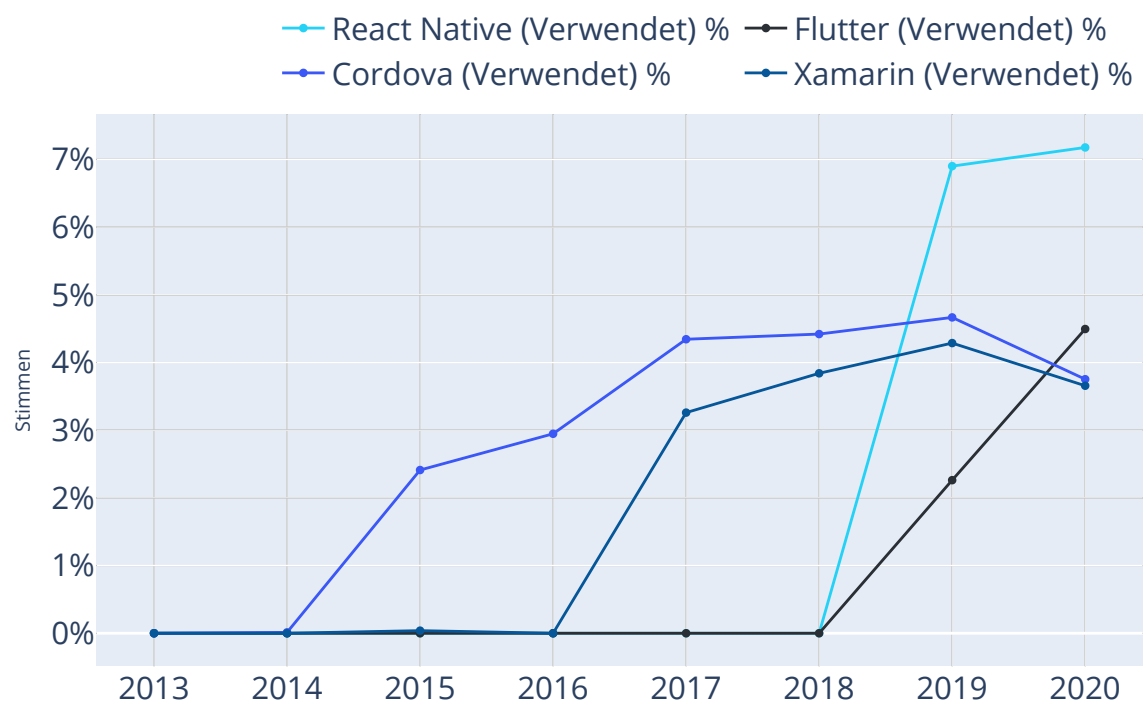


Abbildung 2.7.: Stimmen verwendeter Frameworks: *React Native*, *Flutter*, *Xamarin* und *Cordova*, Quelle: Eigene Abbildung, Notebook: [Charts/StackOverflowUmfrage/StackOverflowUmfrage.ipynb](#), Daten-Quelle: <https://insights.stackoverflow.com/survey>

2.2. Vergleich von *React Native* und *Flutter*

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, nach welchen Kriterien sich die Technologien im Hinblick auf die Entwicklung der Formularanwendung vergleichen lassen.

2.2.1. Vergleich zweier minimaler Beispiele für Formulare und Validierung

Formulare in *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 *Formik*, *Redux Forms* und *React Hook Form* sind alle drei kompatibel mit *React Native*.^{9,10,11}

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.

Keine 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 Parameterobjekt namens *RegisterOptions* erhält. Der Eigenschaft *validate* dieses Objektes kann eine *JavaScript*-Funktion für die Validierung übergeben werden.¹² 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.¹³ Auch in *Formik* ist der Bezeichner *validate*, und ist als Attribut in der *Formik*-Komponente zu finden.¹⁴

⁹Vgl. *Formik Docs* / *React Native*.

¹⁰Vgl. *Does redux-form work with React Native?*

¹¹Vgl. *React Hook Form - Get Started*.

¹²Vgl. *React Hook Form - API* / *register*.

¹³Vgl. *Redux Form - API* / *reduxForm*.

¹⁴Vgl. *Formik Docs API* / *<Formik />*.

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, 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.

Für den Vergleich wurde eine Schritt-für-Schritt-Anleitung zum Erstellen eines Formulars mit *React Hook Form* ausgewählt.¹

Formulare in *Flutter*

Die *Flutter*-Dokumentation stellt in ihrer *cookbook*-Sektion ein Beispiel einer minimalistischen Formularanwendung mit Validierung bereit.¹⁵ Das Rezept ist Teil einer Serie von insgesamt fünf Anleitungen, welche Formulare in *Flutter* behandeln.¹⁶

Die Rezepte wurden genutzt, um eine Formularanwendung zu implementieren, welche dem Ergebnis der *React Hook Form*-Applikation in Funktionalität und Layout ähnlich ist.

¹⁵Vgl. Google LLC, *Build a form with validation*.

¹⁶Vgl. Google LLC, *Flutter Docs Cookbook / Forms*.

¹<https://dev.to/elaziziyousseuf/forms-in-react-native-the-right-way-4d46>

Ergebnisse des Vergleiches

Abbildung 2.8 zeigt die *Flutter*-Anwendung (links) und die *React Native*-Anwendung (rechts).

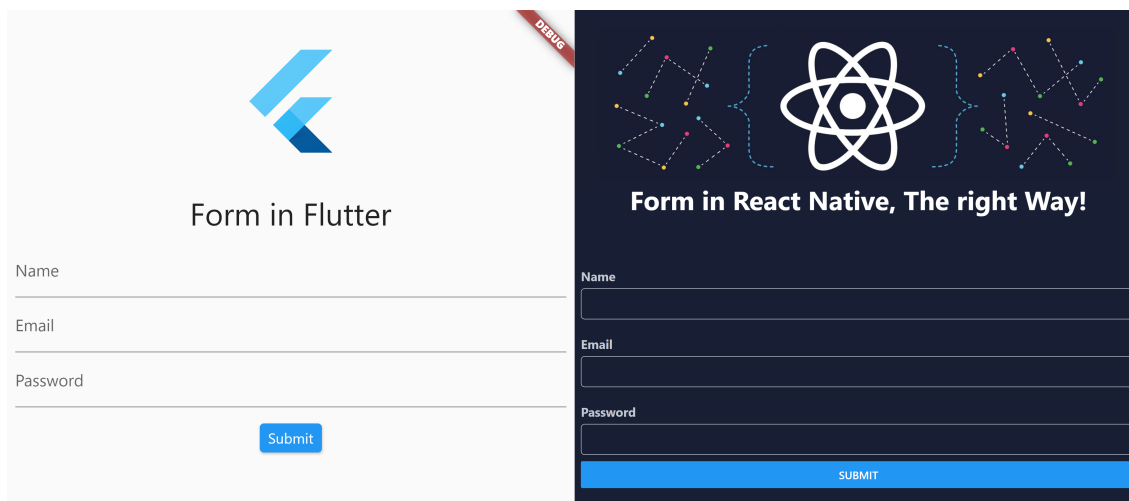


Abbildung 2.8.: Gegenüberstellung der minimalistischen Flutter- und *React Native*-Formularapplikationen, Quelle: Eigene Abbildung

Die Listings A.1 und A.2 des *Flutter*-Formulars sind in Anhang A auf den Seiten 183 und 184 zu finden. Anhang B beinhaltet die Listings B.1, B.2, B.3, B.4, B.5 und B.6 der *React Hook Form*-Anwendung auf den Seiten 185 bis 189.

Folgende Unterschiede waren bei der Untersuchung beider Applikationen auffällig:

- Die Anzahl der benötigten Quelltextzeilen unterscheidet sich stark.
- *Flutter-Widgets* bringen ohne zusätzliche Konfiguration bereits ein ansprechendes Aussehen mit. *React Native* Anwendungen müssen dafür mit zusätzlichen Stylesheets ausgestattet werden.

Tabelle 2.1 auf Seite 37 stellt die Anzahl der Quelltextzeilen der unterschiedlichen Anwendungen auf Dateiebene gegenüber.

Kommentare, leere Zeilen und *import*-Anweisungen wurden bei der Untersuchung ignoriert. Es wurde kein Versuch unternommen, den Quellcode der *React Native*-Applikation zu refaktorisieren, sodass auch dieser die minimal nötige Anzahl von Zeilen aufweist. Die allgemeine Anzahl der Quelltext-Zeilen ist für die Evaluierung der Technologie deshalb zweitrangig.

Vor dem Hintergrund, dass die Wissenschaftler und Wissenschaftlerinnen des Thünen-Instituts keine Anforderungen an das Aussehen der Oberfläche stellen, soll sich für die Technologie entschieden werden, welche den Aufwand für das Gestalten der Oberfläche gering hält. In diesem Fall ist das *Flutter*, da es in der Standardkonfiguration bereits ein ansprechendes Design mitbringt. Die *React Native*-Applikation benötigte dagegen mit 55

Flutter	main.dart		60
	validation.dart		15
	Summe		75
React Native	Programmcode	App.tsx	15
		Hero.tsx	9
		Input.tsx	17
		Form.tsx	52
		validation.tsx	12
		Summe	112
	Stylesheets	App.tsx	14
		Hero.tsx	17
		Input.tsx	24
		Summe	55
	Summe		167

Tabelle 2.1.: Anzahl der Quelltextzeilen der minimalistischen *Flutter* und *React Native*-Formular-Applikationen.

Zeilen in den *Stylesheets* bereits 73 % der Zeilenanzahl, die insgesamt für die Entwicklung des Formulars in *Flutter* nötig waren.

2.2.2. Automatisiertes Testen

Um die Codequalität zu gewährleisten, sollen bei der Entwicklung Unit- und Integrations-tests zum Einsatz kommen. Im Folgenden sollen die Frameworks *Flutter* und *React Native* im Hinblick auf die Dokumentation zu diesen Themen untersucht werden.

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 Unittests, *Mocking*, Integrationstests, Komponententests 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 *Snapshot*-Test verwendet wird. Weiterhin wird auf die *Jest*-API-Dokumentation verwiesen sowie auf ein Beispiel für einen *Snapshot*-Test in der *Jest*-Dokumentation.^{II}

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^{III}. Somit muss zum Erstellen der Unittests 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 – so wie *React Native* auch – nicht in einer Browser-Umgebung.¹⁷ Herausgegeben wird die *React Native Testing Library* vom Drittanbieter *Callstack* – einem Partner im *React Native*-Ökosystem.¹⁸

¹⁷Vgl. Borenkraout, *Testing Library Docs* / *Native Testing Library Introduction*.

¹⁸Vgl. Facebook Inc., *The React Native Ecosystem*.

^{II}<https://jestjs.io/docs/snapshot-testing>

^{III}<https://jestjs.io/docs/getting-started>

Sie verwendet im Hintergrund den *React Test Renderer*^{IV}, 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*^V und *Detox*^{VI}. 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.

Automatisierte Tests in *Flutter*

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

Eine Seite erklärt den Unterschied zwischen Unittests, *Widget*-Tests und Integrationstests^{VII}. Eine weitere Seite erklärt Integrationstests detaillierter^{VIII}.

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^{IX}.

^{IV}<https://reactjs.org/docs/test-renderer.html>

^V<http://appium.io/>

^{VI}<https://github.com/wix/detox/>

^{VII}<https://flutter.dev/docs/testing>

^{VIII}<https://flutter.dev/docs/testing/integration-tests>

^{IX}<https://codelabs.developers.google.com/codelabs/flutter-app-testing>

Im sogenannten Kochbuch tauchen folgende Rezepte auf:

- 2 Rezepte für Unittests
 - eine grundlegende Anleitung zum Erstellen von Unittests ^X
 - Eine weitere Anleitung zum Nutzen von *Mocks* in Unittest mithilfe der Bibliothek *mockito* ^{XI}
- 3 Rezepte für *Widget*-Tests
 - Eine grundlegende Anleitung zum Erstellen von *Widget*-Tests ^{XII}
 - Ein Rezept mit detaillierteren Beispielen zum Finden von *Widgets* zur Laufzeit eines *Widget*-Tests ^{XIII}
 - Ein Rezept zum Testen vom Nutzerverhalten wie dem Tab, dem Drag und dem Eingeben von Text ^{XIV}
- 3 Rezepte für Integrationstests
 - Eine grundlegende Anleitung zum Erstellen eines Integrationstests ^{XV}
 - eine Anleitung zum Simulieren des Scrollens in der Anwendung während der Laufzeit eines Integrationstests ^{XVI}
 - eine Anleitung zum Performance-Profiling ^{XVII}

^X<https://flutter.dev/docs/cookbook/testing/unit/introduction>

^{XI}<https://flutter.dev/docs/cookbook/testing/unit/mocking>

^{XII}<https://flutter.dev/docs/cookbook/testing/widget/introduction>

^{XIII}<https://flutter.dev/docs/cookbook/testing/widget/finders>

^{XIV}<https://flutter.dev/docs/cookbook/testing/widget/tap-drag>

^{XV}<https://flutter.dev/docs/cookbook/testing/integration/introduction>

^{XVI}<https://flutter.dev/docs/cookbook/testing/integration/scrolling>

^{XVII}<https://flutter.dev/docs/cookbook/testing/integration/profiling>

2.3. Fazit und Begründung der Auswahl

Die beiden Frameworks *Flutter* und *React Native* wurden in diesem Kapitel auf die Möglichkeit zur Erstellung von Formularanwendungen geprüft. Beide Frameworks stellen die nötigen Komponenten dafür bereit. In *Flutter* ist der Vorteil zudem, dass die Komponenten in der Standardbibliothek enthalten sind. In *React Native* müssen Bibliotheken wie etwa *Formic*, *Redux Forms* oder *React Hook Form* dafür eingebunden werden.

Für einen Vergleich wurde eine minimalistische *Flutter*-Formularanwendung implementiert, welche einer Beispielanwendung für die Bibliothek *React Hook Form* ähnlich ist. Die *Flutter*-Anwendung hatte weniger Quellcode, entscheidender ist aber: Es war ein äußerst geringer Aufwand nötig, um eine ähnlich benutzerfreundliche Oberfläche zu erstellen. Auch in diesem Punkt ist *Flutter* *React Native* vorzuziehen.

Schließlich wurden beide Frameworks hinsichtlich der Erstellung von automatisierten Tests untersucht. Die Anleitungen für das Testen in *React Native* sind auf mehreren Webportalen verteilt und eine Evaluierung der Bibliotheken und Testtreiber durch den Entwickler ist nötig. Der Aufwand der Einarbeitung in das Testen in *Flutter* ist dagegen 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.

Als Ergebnis dieser drei Vergleiche wird dementsprechend *Flutter* als Technologie zur Umsetzung der Formularanwendung ausgewählt.

3. Grundlagen

Für die Formularanwendung 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 Endgeräte²:
 - *Android*,
 - *iOS*,
- und das Web³.

Flutter ist inspiriert durch das Webframework *React* und deren Oberflächenelemente, die *Components* genannt werden⁴. Die visuellen Oberflächenelemente in *Flutter* werden dagegen *Widgets* genannt. Bei *Flutter* und *React* handelt es sich um deklarative Frontend-Frameworks. Dem stehen die imperativen Frameworks gegenüber.

¹Vgl. Google LLC, *Flutter / Desktop support for Flutter*.

²Vgl. Google LLC, *Flutter / Beautiful native apps in record time*.

³Vgl. Google LLC, *Flutter / Web support for Flutter*.

⁴Vgl. Google LLC, *Flutter / Introduction to widgets*.

In imperativen Frameworks werden Befehle genutzt, um Änderungen direkt an der Oberfläche vorzunehmen, wie am *Java Swing*-Beispiel in Listing 3.1 zu sehen ist. Es wird ein Button erstellt, welcher die Anzahl der auf ihn angewendeten Klicks anzeigt. Bei jedem Klick wird die *Setter*-Methode `setText` genutzt, um den Text direkt zu manipulieren.

```
1 final JButton button = new JButton("Anzahl der Klicks auf diesen Button: 0");
2
3 button.addActionListener(new ActionListener() {
4     @Override
5     public void actionPerformed(ActionEvent e) {
6         timesClicked = timesClicked + 1;
7         button.setText("Anzahl der Klicks auf diesen Button: " + timesClicked);
8     }
9 });
```

Listing 3.1.: *Java Swing*-Beispiel: Ein Button, welcher die Anzahl der Klicks anzeigt, Quelle: Eigenes Listing

In deklarativen Frontend-Frameworks wie *Flutter* existieren solche *Setter*-Methoden nicht. Stattdessen wird die Oberfläche einmal deklariert und darauf vorbereitet, auf Änderungen eines Zustands zu reagieren. Das *Flutter*-Beispiel in Listing 3.2 zeigt dieses Verhalten. Beim Klick auf den Button modifiziert die Methode `setState` den Zustand und löst damit die Aktualisierung der Oberfläche aus.

```
1 return ElevatedButton(
2     onPressed: () {
3         setState(() {
4             timesClicked = timesClicked + 1;
5         });
6     },
7     child: Text("Anzahl der Klicks auf diesen Button: $timesClicked")
8 )
```

Listing 3.2.: *Flutter*-Beispiel: Ein Button, welcher die Anzahl der Klicks anzeigt, Quelle: Eigenes Listing

In *react Components* wird dieser Zustand *State* genannt. *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*.

3.1.1. *Stateful 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.

3.1.2. *Stateless Widgets*

Stateless Widgets haben dagegen keinen solchen Mechanismus. Wie alle *Widgets* werden sie neu gezeichnet, wenn es durch das Framework angeordnet wurde. Das ist der Fall,

- wenn das *Widget* zum ersten Mal in der Oberfläche auftaucht,
- das Väterelement und damit alle Kindelemente neu gezeichnet werden
- oder wenn das *Widget* von einem *InheritedWidget* abhängig ist und dieses aktualisiert wird.⁵

*StatefulWidget*s sind nur eine von vielen Möglichkeiten, den Zustand des Programms zu verwalten. Die Formularanwendung verwendet ausschließlich *StatelessWidgets*, da die Verwaltung des Zustands über das sogenannte *Model-View-ViewModel*-Entwurfsmuster umgesetzt wird. Mehr dazu im Kapitel 5.4.1 *Das Model-View-ViewModel-Entwurfsmuster* auf Seite 78.

3.1.3. *Inherited Widgets*

Neben den zwei Basisklassen *StatelessWidget* und *StatefulWidget* sei noch das *InheritedWidget* zu nennen. Es wird in der Regel nicht für eine visuelle Repräsentation genutzt. Stattdessen zeichnet es das ihm übergebene Kindelement unverändert. Die Hauptaufgabe eines *InheritedWidget* ist es dagegen, dem *Widget*-Baum Informationen bzw. Services verfügbar zu machen.⁶ Mehr dazu im Kapitel 5.4.3.

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

⁵Vgl. Google LLC, *Flutter* / *StatelessWidget class*.

⁶Vgl. Google LLC, *Flutter* / *InheritedWidget class*.

aktualisiert, die tatsächlich verändert wurden.⁷ 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 Zielarchitekturen *ARM32*, *ARM64* und *x86_64* wird *Dart* in Maschinencode kompiliert⁸.

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 Codegenerierung.

⁷Vgl. Google LLC, *Flutter* / *Hot reload*.

⁸Vgl. Google LLC, *Dart: The platforms*.

Codegenerierung

Das Package *source_gen* erlaubt das Auslesen der Metainformationen und ermöglicht das Generieren von Quellcode, der von diesen Eigenschaften abgeleitet werden kann.⁹ So verwendet beispielsweise das Package *built_value* die Codegenerierung.¹⁰ 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.

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 3.3 erstellt werden.

```
1 var menge = Set();
2 menge.add("Text");
```

Listing 3.3.: Ein *Set*, Quelle: Eigenes Listing

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: "Wert 1", 2: "Wert 2"}`). 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

⁹Vgl. Google LLC, *source_gen*.

¹⁰Vgl. Google LLC, *built_value_generator*.

genutzt oder davon abgeleitet werden. Listing 3.4 geht beispielsweise durch die Liste der Temperaturangaben 97.7,105.8, die in Fahrenheit gelistet sind.

```
1 var gradCelsiusTemperaturen = {  
2     for (var f in [97.7, 101.3, 105.8])  
3         (f - 32) * 5 / 9  
4 };
```

Listing 3.4.: Das *collection for* in einer Menge, Quelle: Eigenes Listing

Für jeden Schleifendurchlauf wird die Schleifenvariable `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 3.5 wird durch die gleiche Liste von Temperaturen in Fahrenheit iteriert.

```
1 var celsiusUndFahrenheit = {  
2     for (var f in [97.7, 101.3, 105.8])  
3         (f - 32) * 5 / 9 : f  
4 };
```

Listing 3.5.: Das *collection for* in einer Hashtabelle, Quelle: Eigenes Listing

Für jede Schleifenvariable `f` wird für das resultierende Schlüssel-Werte-Paar das Ergebnis in Grad Celsius als Schlüssel und die ursprüngliche Temperatur in Fahrenheit 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 3.6 iteriert durch eine Anzahl von Temperaturen in Grad Celsius.

```
1 var fieberTemperaturen = [  
2     for (var c in [36.5, 38.5, 41])  
3         if (c >= 38.5) c  
4 ];
```

Listing 3.6.: Das *collection if* in einer Liste, Quelle: Eigenes Listing

Nur in dem Fall, dass die Temperatur der Schleifenvariable `c` größer oder gleich `38,5` ist, wird die Temperatur der Liste hinzugefügt. 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 Typen mit und ohne Null-Zulässigkeit unterschieden. In *Java* ist es nur bei atomaren Datentypen wie *int* und *float* verpflichtend, 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.¹¹ 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 erhöht 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 Typ 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 3.7 wird versucht, auf die Variable `fahrenheitTemperature` den Operator `-` anzuwenden, um sie mit `32` zu subtrahieren.

```
1 void printTemperatureInCelsius(int? fahrenheitTemperature) {
2   print((fahrenheitTemperature - 32) * 5 / 9);
3 }
```

Listing 3.7.: Fehlerhafter Zugriff auf eine Variable mit Null-Zulässigkeit, Quelle: Eigenes Listing

¹¹Vgl. Thomsen, *Announcing Dart 2.12*.

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 – 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.¹² Der Code in Listing 3.8 lässt sich damit wieder kompilieren.

```
1 void printTemperatureInCelsius(int? temperature) {  
2   if (temperature != null) {  
3     print((temperature - 32) * 5 / 9);  
4   }  
5 }
```

Listing 3.8.: Zugriff auf eine Variable mit Null-Zulässigkeit durch *type promotion*, Quelle: Eigenes Listing

Eine Besonderheit stellen dabei allerdings Instanzvariablen dar. In *Dart* wird syntaktisch nicht zwischen dem Aufruf einer *Getter*-Methode oder einer Instanzvariablen unterschieden. In Listing 3.9 könnte sich hinter den Aufrufen von `temperature` in den Zeilen 6 und 7 die Instanzvariable verbergen, die in Zeile 2 deklariert ist.

```
1 class Patient {  
2   num? temperature;  
3   Patient({this.temperature});  
4  
5   void printTemperatureInCelsius() {  
6     if (temperature != null) {  
7       print((temperature - 32) * 5 / 9);  
8     }  
9   }  
10 }
```

Listing 3.9.: Fehlerhafter Zugriff auf eine Instanzvariable mit Null-Zulässigkeit, Quelle: Eigenes Listing

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 3.10.

Beim ersten Aufruf von `temperature` wird die Zählvariable `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

¹²Vgl. Nystrom, *Dart / Understanding null safety / Type promotion on null checks*.

```

1 class UnusualPatient extends Patient {
2   int counter = 0;
3
4   num? get temperature {
5     counter++;
6     if (counter.isOdd) {
7       return 97.7;
8     } else {
9       return null;
10    }
11  }
12 }

```

Listing 3.10.: Überschreiben des Instanzattributs mit einer *Getter*-Methode, Quelle: Eigenes Listing

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 mit einem nachgestellten `!` erzwingen, dass die Variable als nicht *null* angesehen wird (Listing 3.11, Z. 3).

```

1 void printTemperatureInCelsius() {
2   if (temperature != null) {
3     print((temperature! - 32) * 5 / 9);
4   }
5 }

```

Listing 3.11.: Erzwungener Zugriff auf eine Instanzvariable mit Null-Zulässigkeit, Quelle: Eigenes Listing

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 3.12, Z. 2).

```

1 void printTemperatureInCelsius() {
2   num? temperature = this.temperature;
3   if (temperature != null) {
4     print((temperature - 32) * 5 / 9);
5   }
6 }

```

Listing 3.12.: Zuweisung der Instanzvariablen zu einer lokalen Variablen, Quelle: Eigenes Listing

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 bereitsteht. Er gleicht damit dem sogenannten *Promise* – deutsch Versprechen – in *JavaScript*¹³.

Das Listing 3.13 zeigt mit dem Lesen einer Datei ein Beispiel für den Aufruf einer asynchronen Operation.

```
1 var fileContent = file.readAsString();
```

Listing 3.13.: Der asynchrone Aufruf *readAsString*, Quelle: Eigenes Listing

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 – deutsch Rückruffunktion. Listing 3.14 zeigt, wie auf den Dateiinhalt zugegriffen werden kann.

```
1 fileContent.then((text) {  
2   print("Der Dateiinhalt ist: $text");  
3 });
```

Listing 3.14.: Aufruf von *then* auf dem *Future*-Objekt, Quelle: Eigenes Listing

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.

¹³Vgl. MDN contributors, *MDN / Promise - JavaScript*.

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.

3. Grundlagen

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^{14,15}.

Listing 3.15 zeigt, dass das Anwenden des Schlüsselwortes `await` vor der asynchronen 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.

```
1 printFileContent() async {  
2   var fileContent = await file.readAsString();  
3   print("Der Dateiinhalt ist: $fileContent");  
4 }
```

Listing 3.15.: Aufruf der asynchronen Methode *readAsString* mit dem *await*-Schlüsselwort, Quelle: Eigenes Listing

Streams

Streams liefern nicht nur einen Wert – wie im Fall eines *Future*-Objektes – sondern eine Serie von Werten, die in der Zukunft geliefert werden. Listing 3.16 zeigt, wie auf einen solchen *Stream* gehorcht werden kann.

```
1 var countStream = Stream<num>.periodic(const Duration(seconds: 1), (count) {  
2   return count;  
3 });  
4  
5 countStream.listen((count) {  
6   print("Gezählte Sekunden: $count");  
7 });
```

Listing 3.16.: Abhören eines *Streams*, Quelle: Eigenes Listing

Der `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 `count` 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*.

¹⁴Vgl. MDN contributors, *MDN / async function - JavaScript*.

¹⁵Vgl. 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. Konzeption

In diesem Kapitel soll die grafische Benutzeroberfläche konzipiert werden. Dazu gehören der Übersichtsbildschirm, die Eingabemaske und der Selektionsbildschirm.

4.1. Der Übersichtsbildschirm

Bei Programmstart wird der Benutzer mit dem Übersichtsbildschirm begrüßt (Abb. 4.1) . Er listet die bisher eingegebenen Maßnahmen auf. Sie werden in zwei Rubriken gruppiert. Maßnahmen, in welchen bereits alle Eingabefelder gefüllt und valide sind, werden in der Gruppe *Abgeschlossen* eingeblendet. Maßnahmen, welche sich noch im Bearbeitungsmodus befinden, da ihnen Inhalte in den Eingabefeldern fehlen oder die Eingaben nicht valide sind, erscheinen in der Rubrik *in Bearbeitung*.

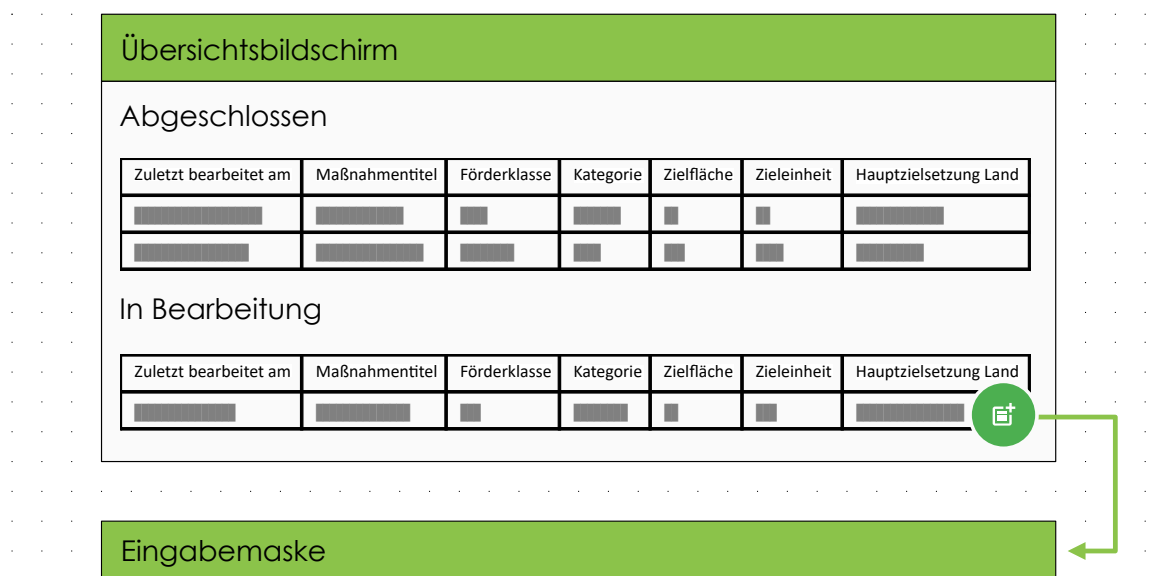


Abbildung 4.1.: Konzeption des Übersichtsbildschirms, Quelle: Eigene Abbildung

Klickt der Benutzer auf den Aktionsbutton unten rechts, so gelangt er auf den zweiten Bildschirm: die Eingabemaske.

4.2. Die Eingabemaske

Die Eingabemaske (Abb. 4.2) listet die Eingabefelder für die Eigenschaften der Maßnahme. Bedeutsam sind hierbei vor allem die Einfach- und Mehrfachauswahlfelder. Sie werden als Karten dargestellt. Unterhalb des Titels der Eigenschaft, dessen Wert mit der Selektionskarte ausgewählt werden soll, erscheint auch die Anzeige des bisher ausgewählten Wertes – für Einfachauswahlfelder – bzw. der aktuell ausgewählten Werte – für Mehrfachauswahlfelder. Selektionskarten, welche invalide Werte enthalten, werden rot eingefärbt. Die Selektionskarten können über Überschriften in Gruppen zusammengefasst werden, um die Übersichtlichkeit zu erhöhen. Auch Zwischenüberschriften sollen möglich sein. Auch Zwischenüberschriften sollen möglich sein.

Die zwei Aktionsbuttons unten rechts ermöglichen das Speichern der Maßnahme. Der untere der beiden wird dazu verwendet, die Maßnahme vor dem Speichern zu validieren. Nur wenn die Validierung erfolgreich ist, wird die Maßnahme auch gespeichert und der Benutzer gelangt zurück zum Übersichtsbildschirm. Anderenfalls erhält er eine Fehlermeldung. Mit dem Button darüber wird die Maßnahme dagegen direkt im Entwurfsmodus abgespeichert, ohne eine Validierung durchzuführen. Auch nach Anklicken dieses Buttons gelangt der Nutzer zurück zum Übersichtsbildschirm. Klickt der Benutzer auf den Zurück-Button oben links im Bildschirm, so wird versucht, die Eingabemaske wieder zu verlassen, sofern dies möglich ist. Ist die Maßnahme im Entwurfsmodus, so gelangt der Benutzer mit diesem Button direkt zurück zum Übersichtsbildschirm, ist die Maßnahme dagegen im Modus *Abgeschlossen*, so wird zunächst eine Validierung ausgeführt.



Abbildung 4.2.: Konzeption der Eingabemaske, Quelle: Eigene Abbildung

Mit einem Klick auf die Selektionskarten wird der Benutzer auf den Selektionsbildschirm weitergeleitet, um Auswahloptionen für das angeklickte Auswahlfeld auszuwählen.

4.3. Der Selektionsbildschirm

Auf dem Selektionsbildschirm (Abb. 4.3) werden alle möglichen Auswahloptionen aufgelistet. Mit einem Klick darauf kann der Benutzer die Optionen aktivieren und deaktivieren.

Auswahloptionen, welche mit den in anderen Auswahlfeldern ausgewählten Werten nicht kompatibel sind, erscheinen am Ende der Liste. Sie sind mit einem Kreuz-Symbol gekennzeichnet. Optionen, welche zuvor angewählt waren und durch eine neue Selektion nun invalide geworden sind, erscheinen mit einem roten Hintergrund.

Klickt der Benutzer auf den Aktionsbutton unten rechts im Bildschirm oder auf den Zurück-Button oben links, so gelangt er zurück zur Eingabemaske.

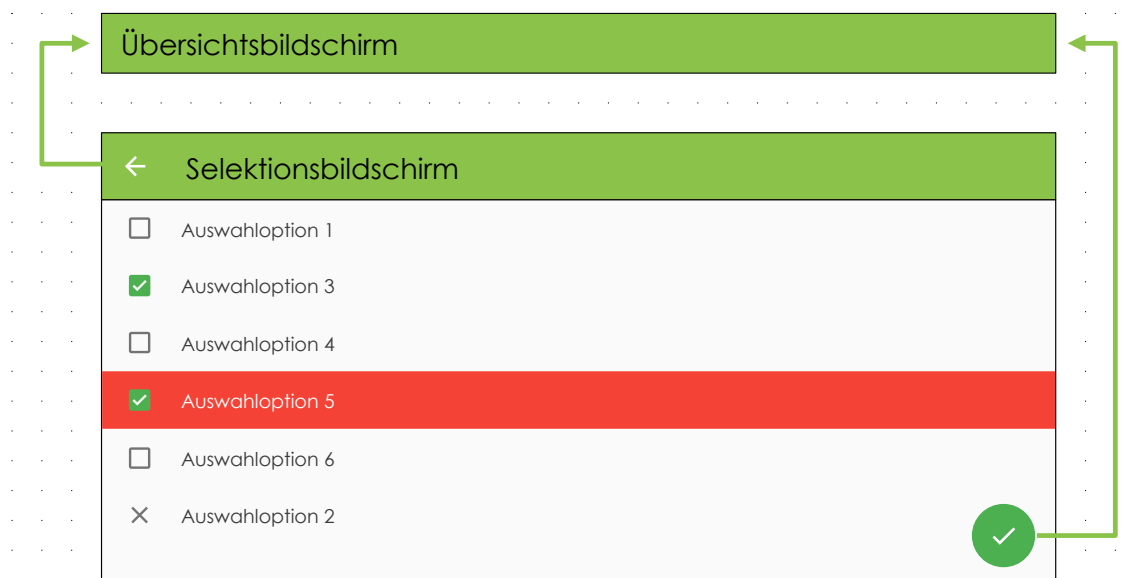


Abbildung 4.3.: Konzeption des Selektionsbildschirms, Quelle: Eigene Abbildung

Teil III

IMPLEMENTIERUNG

5. Schritt 1 – Grundstruktur der Formularanwendung

Im ersten Schritt soll die Formularanwendung in ihrer Grundstruktur entwickelt werden. Das beinhaltet alle drei Oberflächen, welche in den darauf folgenden sechs Schritten erweitert werden. Das Formular erhält noch keine Validierung. Somit sind alle Eingaben oder nicht kompatible Selektionen erlaubt. Die erste Ansicht, welche der Benutzer sieht, soll die Übersicht der bereits eingetragenen Maßnahmen sein (Abb. 5.1).

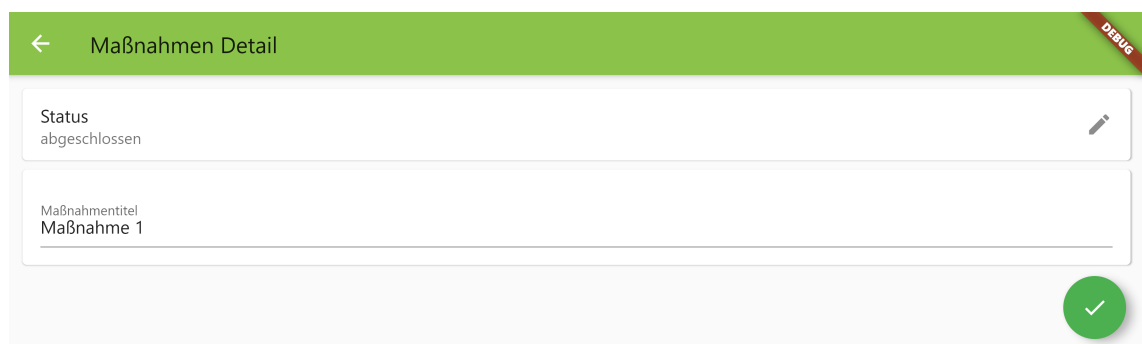


Zuletzt bearbeitet am	Maßnahmentitel
2021-8-4 13:20	Maßnahme 1

Zuletzt bearbeitet am	Maßnahmentitel
2021-8-4 13:17	Maßnahme 2

Abbildung 5.1.: Der Übersichtsbildschirm in Schritt 1, Quelle: Eigene Abbildung

Die Auflistung der Maßnahmen erfolgt in den Kategorien *In Bearbeitung* und *Abgeschlossen*. Innerhalb dieser Rubriken werden die Maßnahmen in einer Tabelle angezeigt. Mit einem Klick auf den Button unten rechts im Bild wird der Benutzer auf die die Eingabemaske weitergeleitet (Abb. 5.2).



Status
abgeschlossen

Maßnahmentitel
Maßnahme 1

Abbildung 5.2.: Die Eingabemaske in Schritt 1, Quelle: Eigene Abbildung

Sie ermöglicht die Eingabe des Maßnahmentitels über ein simples Eingabefeld. Außerdem ist die Selektionskarte für den Status zu sehen. Mit einem Klick auf diese Karte öffnet sich der Selektionsbildschirm. Er ermöglicht die Auswahl der Auswahloptionen, in diesem Fall die Optionen *in Bearbeitung* und *abgeschlossen* (Abb. 5.3).



Abbildung 5.3.: Der Selektionsbildschirm in Schritt 1, Quelle: Eigene Abbildung

5.1. Auswahloptionen hinzufügen

Dart verfügt – anders als beispielsweise *Java*¹ – nicht über Aufzählungstypen mit zusätzlichen Eigenschaften. Das Schlüsselwort `enum` in *Dart* erlaubt lediglich die Auflistung konstanter Symbole². Für die Auswahloptionen ist es jedoch notwendig, dass es zwei Eigenschaften gibt:

- die Abkürzung, die in der resultierenden Datei gespeichert werden soll
- und der Beschreibungstext, welcher in der Oberfläche angezeigt wird.

Das hat den Hintergrund, dass die Abkürzungen weniger Speicherplatz einnehmen und die Beschreibung sich in Zukunft auch ändern darf. Würde anstatt der Abkürzung die Beschreibung als Schlüssel verwendet werden, so würde eine Datei, die mit einer älteren Version des Formulars erstellt wurde, nicht mehr von neueren Versionen der Applikation eingelesen werden können. Der alte Beschreibungstext würde nicht mehr mit dem Text übereinstimmen, der als Schlüssel in der Anwendung verwendet wird.

Die beiden Zustände *in Bearbeitung* und *abgeschlossen* werden daher in Listing 5.1 als statische Klassenvariablen deklariert (Z. 6-7). Die beiden Konstruktoraufrufe übergeben dabei als erstes Argument die Abkürzung und als zweites Argument die Beschreibung. Der Konstruktor selbst (Z. 9-10) deklariert sie als sogenannte *positionale Parameter*.

Positionale Parameter Im Vergleich zu den benannten Parametern ist bei den positionalen Parametern nur ihre Reihenfolge in der Parameterliste ausschlaggebend. Das Argument für die `abbreviation` steht dabei also immer an erster Stelle und das Argument

¹Vgl. Gosling u. a., *The Java® Language Specification Java SE 16 Edition*, S. 321.

²Vgl. Google LLC, *Dart Programming Language Specification 5th edition*, S. 74f.


```

5 class LetzterStatus extends Choice {
6   static final bearb = LetzterStatus("bearb", "in Bearbeitung");
7   static final fertig = LetzterStatus("fertig", "abgeschlossen");
8
9   LetzterStatus(String abbreviation, String description)
10    : super(abbreviation, description);
11 }

```

Listing 5.1.: Die Klasse *LetzterStatus*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/choices/choices.dart](#)

für `description` immer an der zweiten (Z. 6-7). Positionale Parameter sind verpflichtend. Werden sie ausgelassen, so gibt es einen Compilerfehler.³

Die Klasse `LetzterStatus` erbt von der Basisklasse `Choice` (Z. 5). Der Konstruktor der Klasse (Z. 9) übergibt beide Parameter als Argumente an den Konstruktor der Klasse `Choice`. Genau wie in *Java* wird mithilfe des Schlüsselwortes `super` (Z. 10) der Konstruktor der Basisklasse aufgerufen. Doch anders als in *Java* erfolgt der Aufruf des `super` Konstruktors nicht in der ersten Zeile des Konstruktor-Körpers⁴. Weil das Aufrufen des Konstruktors der Basisklasse zum statischen Teil der Objekt-Instanziierung gehört, muss der Aufruf von `super` in der Initialisierungsliste erfolgen. Die Initialisierungsliste wird mit dem `:` nach der Parameterliste eingeleitet (Z. 10)⁵.

Die Basisklasse `Choice` (Listing 5.2) deklariert lediglich die beiden Felder `description` und `abbreviation` jeweils als `String` (Z. 4-5). Beide sind mit `final` gekennzeichnet, was sie zu unveränderlichen Instanzvariablen macht. Nach der Initialisierung können sie keine anderen Werte annehmen.⁶ Die Initialisierung der beiden Variablen muss im statischen Kontext der Instanziierung erfolgen. Mit der abgekürzten Schreibweise `this.abbreviation` und `this.description` im Konstruktor (Z. 7) werden die Parameter den Feldern zugewiesen.

```

3 class Choice {
4   final String description;
5   final String abbreviation;
6
7   const Choice(this.abbreviation, this.description);

```

Listing 5.2.: Die Klasse *Choice*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/choices/base/choice.dart](#)

Die Angabe des Parametertyps mittels `(String abbreviation, String description)` ist daher nicht nötig, denn der Typ des Parameters kann bereits durch Angabe des Typs in der Instanzvariablen-Deklaration (Z. 4-5) abgeleitet werden. Außerdem entfällt die Zuweisung in der Form `this.abbreviation = abbreviation` und `this.description = description`.⁷

³Vgl. Google LLC, *Dart Programming Language Specification 5th edition*, S. 74f.

⁴Vgl. Gosling u. a., *The Java® Language Specification Java SE 16 Edition*, S. 310.

⁵Vgl. Google LLC, *Dart Programming Language Specification 5th edition*, S. 42.

⁶Vgl. Google LLC, *Dart Programming Language Specification 5th edition*, S. S16.

⁷Vgl. Google LLC, *Dart Programming Language Specification 5th edition*, S. 40f.

Die Variable `letzterStatusChoices` (Listing 5.3, Z. 13) fasst die beiden statischen Klassenvariablen als eine Kollektion zusammen. Da es sich um eine solche Kollektion handelt, in der jedes Element nur ein einziges Mal vorkommen darf, ist hier von einer Menge zu sprechen. Auffällig hierbei ist, dass das Schlüsselwort *new* fehlt. In *Dart* ist das Schlüsselwort für die Konstruktion von Instanzen optional. Die Klasse, die zur Konstruktion dieser Menge verwendet wird, ist die selbst erstellte Klasse `Choices`. Über das Typargument `LetzterStatus` wird erreicht, dass ausschließlich Variablen dieses Typs in der Menge eingefügt werden dürfen. Wird stattdessen eine Variable eingefügt, die weder vom selben Typ ist, noch von einem Typ, der von `LetzterStatus` erbt, so gibt es einen Compilerfehler. Dies dient einzig und allein dem Zweck, dem Fehler vorzubeugen, dass aus Versehen falsche Optionen in der Menge eingetragen werden. Über den Parameter `name` ist es möglich, dieser Menge die Beschriftung `"Status"` hinzuzufügen. Es handelt sich hier um einen benannten Parameter.

```
13 final letzterStatusChoices = Choices<LetzterStatus>(  
14     {LetzterStatus.bearb, LetzterStatus.fertig},  
15     name: "Status");
```

Listing 5.3.: Die Menge `letzterStatusChoices`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/choices/choices.dart](#)

Listing 5.4 zeigt die Klasse `Choices`. Sie erbt von `UnmodifiableSetView` und erlaubt damit die Erstellung einer eigenen Menge – auch *Set* genannt. Methoden, die man von einem *Set* erwartet, lassen sich somit direkt auf Instanzen der Klasse `Choices` aufrufen. Darunter ist unter anderem die *contains*-Methode, welche erlaubt, das Vorhandensein eines Objektes im *Set* zu überprüfen.

```
10 class Choices<T extends Choice> extends UnmodifiableSetView<T> {  
11     final String name;  
12     final Map<String, T> choiceByAbbreviation;  
13  
14     T? fromAbbreviation(String? abbreviation) => choiceByAbbreviation[abbreviation];  
15  
16     Choices(Set<T> choices, {required this.name})  
17         : choiceByAbbreviation = {  
18             for (var choice in choices) choice.abbreviation: choice,  
19         },  
20         super(choices);  
21 }
```

Listing 5.4.: Die Klasse `Choices`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/choices/base/choice.dart](#)

Die Instanzvariable `name` (Z. 11) wird im Konstruktor (Z. 16) zugewiesen. Auffällig hierbei ist, dass der Parameter in geschweiften Klammern geschrieben steht und das Schlüsselwort `required` vorangestellt ist. Das macht den Parameter zu einem verpflichtenden benannten Parameter.

Verpflichtende benannte Parameter Gewöhnliche benannte Parameter sind optional. Wird ihnen das Schlüsselwort `required` vorangestellt, so müssen sie gesetzt werden, da

sonst ein Compilerfehler ausgelöst wird. Der Ausdruck `{required this.name}` ist im Konstruktor nötig, denn es handelt sich bei `name` um eine Variable vom Datentyp `String`, der den Wert `null` nicht annehmen kann. Würde der Parameter aber optional sein, so wäre es nicht möglich, das Programm zu kompilieren, denn bei Aufrufen des Konstruktors wäre es möglich, das Argument für den Parameter auszulassen. Doch in diesem Fall gäbe es keinen Initialwert für `name` und somit müsste der Instanzvariablen der Wert `null` zugewiesen werden. Der Datentyp `String` erlaubt jedoch keine Null-Werte. Dürfte `name` den Wert `null` annehmen, so würde es sich um den Datentyp `String` mit Null-Zulässigkeit – also mit der Notation `String?` – handeln. In der statischen Analyse muss daher sichergestellt werden, dass solche Instanzvariablen mit absoluter Sicherheit initialisiert werden. Zu diesem Zweck kann der benannte Parameter durch das `required`-Schlüsselwort als verpflichtend gekennzeichnet werden. Somit kann er nicht ausgelassen werden. Damit ist garantiert, dass die Instanzvariable `name` einen Wert erhält, der nicht `null` ist.

Neben `name` wird mit `choiceByAbbreviation` eine weitere Instanzvariable deklariert (Z. 12). Es handelt sich um den Datentyp `Map` – eine Kollektion, welche Daten mittels Schlüssel-Werte-Paaren ablegen kann. Als Schlüssel wird die Abkürzung mit dem Datentyp `String` verwendet. Als Wert ist der generische Typparameter `T` angegeben. Er ist in Zeile 10 deklariert und muss mindestens von der Klasse `Choice` erben. In `choiceByAbbreviation` werden also die Auswahloptionen über ihre Abkürzungen abgelegt und können über dieselbe wieder referenziert werden. Da es sich auch hier um eine unveränderliche Instanzvariable handelt, muss sie schon in der Initialisierungsliste initialisiert werden (Z. 17-19). Dabei wird zunächst mit der öffnenden geschweiften Klammer (Z. 17) ein sogenanntes Literal einer *Map* begonnen, welches mit einer schließenden geschweiften Klammer (Z. 19) endet. Mehr zu *Map*-Literalen in dem Grundlagenkapitel 3.2.2 *Set- und Map-Literale* auf Seite 47.

Auffällig ist jedoch, dass in Zeile 18 dem *Set*-Literal keine einfache Auflistung von Werten übergeben wird. Stattdessen wird mit dem sogenannten *collection for* eine Wiederholung verwendet.

In Zeile 18 wird durch die Menge aller Auswahloptionen `choices` iteriert und dabei in jedem Schleifendurchlauf die Auswahloption in der Variablen `choice` gespeichert. Während des Schleifendurchlaufs wird dann ein Schlüssel-Werte-Paar gebildet, wobei `choice.abbreviation` der Schlüssel ist und das Objekt `choice` der Wert.

Die `Map` `choiceByAbbreviation` erlaubt es, nach der Initialisierung mithilfe der Methode `fromAbbreviation` (Z. 14) über die Abkürzung das dazugehörige `Choice`-Objekt abzurufen. Beispielsweise gibt der Befehl `letzterStatusChoices.fromAbbreviation("fertig")` das Objekt `LetzterStatus("fertig", "abgeschlossen")` zurück. Auffällig dabei ist, dass der Parameter `abbreviation` mit dem Typ `String?` und der generische Rückgabetyt mit `T?` gekennzeichnet sind. Das Suffix `?` macht beide zu Typen mit Null-Zulässigkeit.

Die Methode `fromAbbreviation` soll für die Deserialisierung genutzt werden. Sollten im Formular Auswahlfelder leer gelassen worden sein, so haben entsprechende Variablen den Wert `null`. Wenn nun das Formular abgespeichert wird, so tauchen auch in der abgespeicherten *JSON*-Datei keine Werte für das Feld auf. Aus der *JSON*-Datei werden ausschließlich die Abkürzungen der Auswahloptionen gelesen. Die Methode `fromAbbreviation` wandelt sie wieder in die entsprechenden Objekte des Datentyps `Choice` um. Sollte jedoch kein Wert hinterlegt sein, so wird `letzterStatusChoices.fromAbbreviation(null)` aufgerufen. Dadurch wird klar, dass der Parameter Null-Werte zulassen muss. Es impliziert auch, dass potenziell `null` zurückgeben werden kann, da für den Schlüssel `null` kein Wert in der `Map` hinterlegt sein kann. Deshalb erlaubt auch der Rückgabety `T?` Null-Werte.

5.2. Serialisierung einer Maßnahme

Damit die Daten angezeigt und verändert werden können, müssen sie zunächst serialisierbar sein, sodass sie auf einen Datenträger geschrieben und von dort auch wieder gelesen werden können. Die zwei bekanntesten Bibliotheken zum Serialisieren in *Dart* heißen `json_serializable` und `built_value`.⁸ Beide haben gemeinsam, dass sie Quellcode generieren, welcher die Umwandlung der Objekte in *JSON* übernimmt.

Das Paket `built_value` bietet im Gegensatz zu `json-serializable` jedoch die Möglichkeit, unveränderbare Wertetypen – sogenannte *immutable value types* – zu erstellen. Da diese unveränderbaren Werte noch bei der Erstellung des sogenannten *ViewModels* – mehr dazu im Kapitel 5.9 *Das ViewModel* auf Seite 94 – hilfreich werden, fiel die Entscheidung auf diese Bibliothek.

Ein Wertetyp für `built_value` erfordert einige Zeilen Boilerplate-Code, um den generierten Quellcode mit der selbst geschriebenen Klasse zu verknüpfen. Entwicklungsumgebungen wie *Visual Studio Code* und *Android Studio* erlauben, solchen Boilerplate-Code generieren zu lassen und dabei nur die erforderlichen Platzhalter auszutauschen. In *Visual Studio Code* werden diese Templates *Snippets* genannt, in *Android Studio* heißen sie *Live Templates*. Listing 5.5 zeigt, wie das *Live Template* für das Generieren eines Wertetyps für `built_value` aussieht. Templates für `built_value` wie dieses und weitere müssen nicht vom Nutzer eingegeben werden, sondern existieren bereits als Plug-in für die beiden Entwicklungsumgebungen^{XVIII,XIX}.

`$ClassName$` wird dabei jeweils durch den gewünschten Klassennamen ersetzt. *Android Studio* erlaubt, dass beim Einfügen des *Live Templates* der Klassenname nur einmalig

⁸Vgl. Google LLC, *Flutter / JSON and serialization*.

^{XVIII}<https://plugins.jetbrains.com/plugin/13786-built-value-snippets>

^{XIX}<https://marketplace.visualstudio.com/items?itemName=GiancarloCode.built-value-snippets>

```

6 part '$file_name$.g.dart';
7
8 abstract class $ClassName$ implements Built<$ClassName$, $ClassName$Builder> {
9   $todo$
10
11   $ClassName$. _();
12   factory $ClassName$([void Function($ClassName$Builder) updates]) = _$$$ClassName$;
13
14   static Serializer<$ClassName$> get serializer => _$$$className$serializer;
15 }

```

Listing 5.5.: Abgeändertes *Live Template* für die Erstellung von *built_value* Boilerplate-Code in *Android Studio*, Quelle: Vgl. *Jetbrains Marketplace Built Value Snippets Plugin*

eingetragen werden muss. Anschließend wird mithilfe des *Live Templates* der Boilerplate-Code generiert.

In Listing 5.6 ist der Wertetyp `Massnahme` zu sehen. Die Zeilen 6 bis 8 sowie 18 bis 23 wurden dabei automatisch erstellt. Die Zeilen 9 bis 16 wurden hinzugefügt. Zunächst soll die Maßnahme über den `guid` eindeutig identifiziert werden können (Z. 9). Ein GUID – Kurzform von *Globally Unique Identifier* – ist eine Folge von 128 Bits, die zur Identifikation genutzt werden kann.⁹ Eine solche GUID hat eine textuelle Repräsentation wie beispielsweise die folgende: `'f81d4fae-7dec-11d0-a765-00a0c91e6bf6'`

```

6 part 'massnahme.g.dart';
7
8 abstract class Massnahme implements Built<Massnahme, MassnahmeBuilder> {
9   String get guid;
10
11   LetzteBearbeitung get letzteBearbeitung;
12
13   Identifikatoren get identifikatoren;
14
15   static void _initializeBuilder(MassnahmeBuilder b) =>
16     b..guid = const Uuid().v4();
17
18   Massnahme._();
19
20   factory Massnahme([void Function(MassnahmeBuilder) updates]) = _$Massnahme;
21
22   static Serializer<Massnahme> get serializer => _$massnahmeSerializer;
23 }

```

Listing 5.6.: Der Wertetyp *Massnahme*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/massnahme.dart](#)

Die Attribute `letzteBearbeitung` (Z. 11) und `identifikatoren` (Z. 13) sind im Gegensatz zu dem `String`-Attribut `guid` zusammengesetzte Datentypen, die im Folgenden weiter beleuchtet werden.

Auffällig ist, dass es sich hier um eine abstrakte Klasse handelt und die drei Attribute jeweils *Getter*-Methoden ohne Implementierung sind. Eine solche *Getter*-Methode speichert keinen Wert, sondern gibt lediglich den Wert eines Feldes zurück. Die dazugehörigen

⁹Vgl. Leach, Salz und Mealling, *A Universally Unique Identifier (UUID) URN Namespace*, S. 1.

Felder, *Setter*-Methoden, die konkrete Klasse und der restliche generierte Code sind in der gleichnamigen Datei mit der Endung *.g.dart* (Z. 6) zu finden. Diese Datei wird von *built_value* gefüllt.

Die Klassen-Methode `_initializeBuilder` kann in jedem Wertetyp hinterlegt werden, um Standardwerte für Felder festzulegen.¹⁰ Die Methode wird intern von *built_value* aufgerufen. Bei dem Feld `guid` handelt es sich um einen `String`, der keine Null-Werte zulässt. Könnte das Feld auch Null-Werte annehmen, so wäre die Notation in *Dart* dafür stattdessen `String? get guid;`. *built_value* erwartet also immer einen Wert für dieses Feld. Sollte die Datei gelesen werden, welche die Maßnahmen enthält, so enthält jede Maßnahme bei der Deserialisierung den abgespeicherten Wert für die `guid` und somit wird das Feld gefüllt. Doch sollte eine leere Maßnahme über einen Konstruktor erstellt werden, so wäre das Feld `guid` leer und *built_value* würde einen Fehler auslösen. Aus diesem Grund wird in der Methode `_initializeBuilder` für das Feld `guid` ein Standardwert festgelegt: nämlich eine zufällig generierte ID, die dem Standard *Uuid* der Version 4 entspricht (Z. 16). Zu diesem Zweck wird das *Builder*-Objekt verwendet. Die Klasse `MassnahmeBuilder` gehört dabei zu dem von *built_value* generierten Quellcode. Der Parametername wird hier – wie so häufig im *builder pattern* – mit einem `b` für *Builder* abgekürzt. Die Syntax `=>` leitet einen sogenannten *arrow function body* ein. Dabei handelt es sich schlicht um einen Funktionskörper, der aus genau einer Anweisung besteht. Deshalb muss er nicht von geschweiften Klammern umgeben werden.¹¹ Auf dem *Builder*-Objekt können dann die Eigenschaften so gesetzt werden, als wären sie die Eigenschaften von dem Objekt `Massnahme`. In Wahrheit werden sie aber nur auf dem *Builder*-Objekt angewendet. Ebenfalls auffällig ist die Syntax `b.guid`. Statt eines `.` zum Zugriff auf Attribute des Objektes wird hier der sogenannte Kaskadierungs-Operator `..` benutzt.

Der Kaskadierungs-Operator Durch Eingabe von zwei aufeinanderfolgenden Punkten `..` können mehrere Operationen an einem Objekt ausgeführt werden, ohne das Objekt zuvor einer Variablen zuzuweisen oder die Operationen über deren Namen wiederholt aufzurufen.¹² Beispiel: Die zwei Aufrufe `objekt.tueEtwas();` und `objekt.tueEtwasAnderes();` sind äquivalent mit dem Einzeiler `objekt..tueEtwas()..tueEtwasAnderes();`.

Da der Kaskadierungs-Operator jedoch dazu verwendet wird, mehrere Operationen auf einem Objekt auszuführen, hat er in Zeile 16 keine Funktion. Doch bei Änderung eines Objektes über das *builder pattern* werden für gewöhnlich mehrere Operationen am gleichen *Builder*-Objekt ausgeführt, weshalb der Einheitlichkeit wegen der Kaskadierungs-Operator immer im Zusammenhang mit dem *Builder*-Objekt verwendet werden soll.

¹⁰Vgl. Google LLC, *built_value Changelog*.

¹¹Vgl. Google LLC, *Dart Programming Language Specification 5th edition*, S. 18f., 234.

¹²Vgl. Google LLC, *Dart Programming Language Specification 5th edition*, S. 149f.

Die Attribute `letzteBearbeitung` und `identifikatoren` (Z. 11, 13) erhalten dagegen ganz automatisch Standardwerte in Form von Instanzen der dazugehörigen Klassen. Diese wiederum konfigurieren ihre eigenen Felder und deren initiale Werte.

Der Wertetyp `Identifikatoren` enthält das Attribut `massnahmenTitel` (Listing 5.7, Z. 27), welches im Eingabeformular durch das Texteingabefeld gefüllt wird.

```

25 abstract class Identifikatoren
26     implements Built<Identifikatoren, IdentifikatorenBuilder> {
27     String get massnahmenTitel;
28
29     static void _initializeBuilder(IdentifikatorenBuilder b) =>
30         b..massnahmenTitel = "";

```

Listing 5.7.: Der Wertetyp *Identifikatoren*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/massnahme.dart](#)

Zusätzlich enthält der Wertetyp `LetzteBearbeitung` die Attribute `letztesBearbeitungsDatum` (Listing 5.8, Z. 43) und `letzterStatus` (Z. 50). Im Eingabeformular wird der Selektionsbildschirm den Inhalt des Feldes `letzterStatus` bestimmen. Der initiale Wert wird auf einen konstanten Wert gesetzt, der dem Zustand `'in Bearbeitung'` entspricht (Z. 54).

```

41 abstract class LetzteBearbeitung
42     implements Built<LetzteBearbeitung, LetzteBearbeitungBuilder> {
43     DateTime get letztesBearbeitungsDatum;
44
45     String get formattedDate {
46         final date = letztesBearbeitungsDatum;
47         return "${date.year}-${date.month}-${date.day} ${date.hour}:${date.minute}";
48     }
49
50     String get letzterStatus;
51
52     static void _initializeBuilder(LetzteBearbeitungBuilder b) => b
53         ..letztesBearbeitungsDatum = DateTime.now().toUtc()
54         ..letzterStatus = LetzterStatus.bearb.abbreviation;

```

Listing 5.8.: Der Wertetyp *LetzteBearbeitung*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/massnahme.dart](#)

Das Attribut `letztesBearbeitungsDatum` ist dagegen nicht im Formular änderbar, sondern wird einmalig auf den aktuellen Zeitstempel gesetzt (Z. 53). Zugehörig zu diesem Attribut gibt es noch eine abgeleitete Eigenschaft namens `formattedDate` (Z. 45-48). Es ist eine Hilfsmethode, die das letzte Bearbeitungsdatum in ein für Menschen lesbares Datumsformat umwandelt. In dem Übersichtsbildschirm (Abb. 5.1, S. 63) ist das Datumsformat sichtbar. Die Spalte *Zuletzt bearbeitet am* enthält die Datumsangaben *2021-8-4 13:20* und *2021-8-4 13:17*. Da diese *Getter*-Methode eine Implementierung besitzt, wird für sie von *built_value* kein Quellcode für die Serialisierung generiert.

Bevor die Wertetypen serialisiert werden können, muss *built_value* jedoch noch mitgeteilt werden, für welche Wertetypen Serialisierungs-Funktionen generiert werden sollen. Dazu werden über die Annotation `@SerializersFor` die gewünschten Klassen aufgelistet (Listing 5.9, Z. 10). Die Zeilen 11 und 12 sind dabei immer gleich, es sei denn, es ist ein anderer Serialisierungs-Algorithmus gewünscht. In diesem Fall wird das `StandardJsonPlugin` verwendet.

```
10 @SerializersFor([Massnahme, Storage])
11 final Serializers serializers =
12     (_$serializers.toBuilder()..addPlugin(StandardJsonPlugin())).build();
```

Listing 5.9.: Der Serialisierer für die Wertetypen *Massnahme* und *Storage*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/serializers.dart](#)

Wird nun der Befehl `flutter pub run build_runner build` ausgeführt, so wird der Quellcode generiert und die Wertetypen können für die Serialisierung genutzt werden.

5.2.1. Unittest der Serialisierung einer Maßnahme

Das Ergebnis der Serialisierung wird im dazugehörigen Unittest ersichtlich (Listing 5.10). In Zeile 7 wird ein Objekt der Klasse `Massnahme` instanziiert. Anders als bei gewöhnlichen Datentypen lassen sich bei diesem unveränderlichen Datentyp keine Attribute nach der Erstellung anpassen. Die einzige Möglichkeit besteht darin, ein neues Objekt mit dem gewünschten Attributwert zu erstellen und die restlichen Werte des alten Objektes zu übernehmen. Dies ist mithilfe des sogenannten *Builder*-Entwurfsmuster möglich, welches in *built_value* Anwendung findet.

Erbauer-Entwurfsmuster Das *Erbauer*-Entwurfsmuster – englisch *builder pattern* – ist ein Erzeugungsmuster, welches die Konstruktion komplexer Objekte von ihrer Repräsentation trennt. Es gehört zu der Serie von Entwurfsmustern der *Gang of Four*.¹³ Im Fall von *built_value* trennt es die unveränderlichen Objekte von ihrer Konstruktion. Über den *Builder* lassen sich Änderungen an diesen unveränderlichen Objekten vornehmen, wodurch eine Kopie dieses unveränderlichen Objektes mit der gewünschten Änderung zurückgegeben wird.

In den Zeilen 8 bis 9 wird so ein neues Objekt von der Klasse `Massnahme` mithilfe der Methode `rebuild` erzeugt und anschließend der Referenz `massnahme` zugewiesen, wodurch sie ihren alten Wert verliert. Über die generierte Methode `serializers.serializeWith` kann das Objekt in *JSON* übersetzt werden (Z. 11). Der erste Parameter `Massnahme.serializer` gibt dabei an, wie diese Serialisierung erfolgen soll. Auch das `serializer`-Objekt wurde von *built_value* generiert. Der zweite Parameter ist die tatsächliche `massnahme`, die in *JSON* umgewandelt werden soll. Die Zeilen 13 bis 21 erstellen das *JSON*-Dokument, mit dem das

¹³Vgl. Gamma u. a., *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, S. 119.

serialisierte Ergebnis am Ende verglichen werden soll. Dabei werden die gleichen Eigenschaften eingetragen. So etwa die `guid` (Z. 14), welche bei der Initialisierung der Maßnahme automatisch und zufällig erstellt wurde. Außerdem das letzte Bearbeitungsdatum, welches den Zeitstempel erhält, zu dem die Maßnahme generiert wurde (Z. 16-17). Da `built_value` bei der Serialisierung die Datumswerte in Mikrosekunden umwandelt, muss für das erwartete *JSON*-Dokument das Gleiche passieren. Der Wert des Schlüssels `'letzterStatus'` (Z. 18) wird hierbei auf den Standardwert `'bearb'` gesetzt und der Wert des Schlüssels `'massnahmenTitel'` (Z. 20) auf den gleichen Wert, der in Zeile 9 übergeben wurde. Schließlich vergleicht die Methode `expect` das tatsächlich serialisierte *JSON*-Dokument mit dem, welches zuvor zum Vergleich aufgebaut wurde (Z. 23). Der zweite Parameter ist ein sogenannter *Matcher* und die Variante mit dem Namen `equals` überprüft auf absolute Gleichheit.

```

6 test('Massnahme serialises without error', () {
7   var massnahme = Massnahme();
8   massnahme = massnahme
9     .rebuild((b) => b.identifikatoren.massnahmenTitel = "Massnahme 1");
10
11   var actualJson = serializers.serializeWith(Massnahme.serializer, massnahme);
12
13   var expectedJson = {
14     'guid': massnahme.guid,
15     'letzteBearbeitung': {
16       'letztesBearbeitungsDatum': massnahme
17         .letzteBearbeitung.letztesBearbeitungsDatum.microsecondsSinceEpoch,
18       'letzterStatus': 'bearb'
19     },
20     'identifikatoren': {'massnahmenTitel': 'Massnahme 1'}
21   };
22
23   expect(actualJson, equals(expectedJson));

```

Listing 5.10.: Unittest der Serialisierung einer Maßnahme, Quelle: Eigenes Listing, Datei: [Quellcode/Sc
hritt-1/conditional_form/test/data_model/massnahme_test.dart](#)

5.2.2. Unittest der Deserialisierung einer Maßnahme

Analog zur Serialisierung testet der Unittest in Listing 5.11 auch die Deserialisierung. Das *JSON*-Dokument ist dabei sehr ähnlich und unterscheidet sich lediglich in zwei Details. Der `'guid'` wird auf einen festen Wert festgelegt (Z. 38). Im Initialisierungsprozess der Maßnahme wird er dagegen zufällig generiert. Außerdem wird auch das `letztesBearbeitungsDatum` festgesetzt, nämlich auf die Mikrosekunde `0` (Z. 40).

```

36 test('Massnahme deserialises without error', () {
37   var json = {
38     'guid': "test massnahme id",
39     'letzteBearbeitung': {
40       'letztesBearbeitungsDatum': 0,
41       'letzterStatus': 'bearb'
42     },
43     'identifikatoren': {'massnahmenTitel': 'Massnahme 1'}
44   };
45
46   var expectedMassnahme = Massnahme((b) => b
47     ..guid = "test massnahme id"
48     ..identifikatoren.massnahmenTitel = "Massnahme 1"
49     ..letzteBearbeitung.update((b) {
50       bletztesBearbeitungsDatum =
51         DateTime.fromMicrosecondsSinceEpoch(0, isUtc: true);
52     }));
53   var actualMassnahme =
54     serializers.deserializeWith(Massnahme.serializer, json);
55
56   expect(actualMassnahme, equals(expectedMassnahme));

```

Listing 5.11.: Unittest der Deserialisierung einer Maßnahme, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/test/data_model/massnahme_test.dart](#)

Zum Vergleich wird in den Zeilen 46 bis 52 eine Maßnahme über das *Builder*-Entwurfsmuster generiert und die gleichen festen Werte werden für die Eigenschaften übergeben. Dabei ist darauf zu achten, dass die Instanzvariable `letzteBearbeitung` keinen Wert über den Zuweisungs-Operator `=` erhält, sondern stattdessen die Methode `update` darauf aufgerufen wird (Z. 49).

Da es sich bei der Instanzvariablen `letzteBearbeitung` genauso um ein Objekt eines Wertetyps handelt, ist sie ebenso unveränderlich. Deshalb kann sie nur über einen *Builder* manipuliert werden. Ein Blick in den generierten Quellcode offenbart, dass es sich bei dem Attribut `letzteBearbeitung` in Zeile 49 nicht um die *Getter*-Methode des Wertetypen `Massnahme`, sondern in Wahrheit um einen *Builder* des Typs `LetzteBearbeitungBuilder` handelt (Listing 5.12, Z. 224-225).

Die Mikrosekunden für das Datum müssen zunächst in ein Objekt von `DateTime` umgewandelt werden. Dafür wird der benannte Konstruktor `fromMillisecondsSinceEpoch` von `DateTime` (Z. 51) aufgerufen.

```

216 class MassnahmeBuilder implements Builder<Massnahme, MassnahmeBuilder> {
217   _$Massnahme? _$v;
218
219   String? _guid;
220   String? get guid => _$this._guid;
221   set guid(String? guid) => _$this._guid = guid;
222
223   LetzteBearbeitungBuilder? _letzteBearbeitung;
224   LetzteBearbeitungBuilder get letzteBearbeitung =>
225     _$this._letzteBearbeitung ??= new LetzteBearbeitungBuilder();
226   set letzteBearbeitung(LetzteBearbeitungBuilder? letzteBearbeitung) =>
227     _$this._letzteBearbeitung = letzteBearbeitung;

```

Listing 5.12.: Instanzvariable *letzteBearbeitung* gibt einen *LetzteBearbeitungBuilder* zurück, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/massnahme.g.dart](#)

Benannte Konstruktoren In Programmiersprachen wie beispielsweise *Java* können Methoden überladen werden, indem ihre Methodensignatur geändert wird. Beim Aufruf der Methode kann über die Anzahl und die Typen der übergebenen Argumente die gewünschte Methode gewählt werden. Das Gleiche gilt für Konstruktoren. Wird ein weiterer Konstruktor für eine Klasse in *Java* benötigt, so besteht einzig und allein die Möglichkeit darin, den Konstruktor zu überladen. Sowohl überladene Methoden als auch überladene Konstruktoren existieren in *Dart* nicht. Wird also in *Dart* ein alternativer Konstruktor gewünscht, so muss er einen Namen bekommen. Beim Aufruf des Konstruktors wird dieser Name dann mit einem `.` nach dem Klassennamen angegeben, um den gewünschten Konstruktor zu benennen.¹⁴

Ganz ähnlich wie bei der Serialisierung wird nun mit `serializers.deserializeWith` unter Angabe des Objektes, welches die Deserialisierung übernehmen soll – nämlich wiederum `Massnahme.serializer` – das *JSON*-Dokument in ein Objekt des Wertetyps `Massnahme` deserialisiert (Z. 53-54). Schließlich wird in Zeile 56 das Ergebnis der Deserialisierung mit dem gewünschten Ergebnis verglichen.

Durch Eingabe des Befehls `flutter test test/data_model/massnahme_test.dart` in der Kommandozeile startet die Ausführung aller Tests in der Testdatei. Wenn alle Tests erfolgreich ausgeführt wurden und beide Ergebnisse mit den verglichenen Werten übereinstimmen, erfolgt die Ausgabe: `All tests passed!`.

¹⁴Vgl. Google LLC, *Dart - Language tour - Named constructors*.

5.3. Serialisierung der Maßnahmenliste

Damit alle Maßnahmen – statt nur einer einzigen – in einer Datei zusammengefasst werden können, müssen die Maßnahmen zunächst zu einer Menge zusammengefasst werden, die ebenfalls serialisierbar ist. Der Wertetyp `Storage` ist dafür vorgesehen (Listing 5.13). Er deklariert allein das `BuiltSet` `massnahmen` (Z. 10). Ein `BuiltSet` ist die Abwandlung eines gewöhnlichen `Sets`, jedoch unter anderem mit der Möglichkeit, es mit einem `Builder` zu erstellen und das `Set` zu serialisieren. Die Übergabe des Typarguments `<Massnahme>` gewährleistet, dass keine anderen Objekte eingefügt werden können, die weder eine Instanz der Klasse `Massnahme` sind, oder einer Klasse, die von `Massnahmen` erbt.

```

9 abstract class Storage implements Built<Storage, StorageBuilder> {
10   BuiltSet<Massnahme> get massnahmen;

```

Listing 5.13.: Der Wertetyp `Storage`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/storage.dart](#)

Der Befehl `flutter pub run build_runner build` stößt erneut die Quellcodegenerierung für den Wertetyp `Storage` an.

5.3.1. Unittest der Serialisierung der Maßnahmenliste

Nun soll noch überprüft werden, ob die Menge von Maßnahmen mit genau einer eingetragenen Maßnahme korrekt serialisiert. Auch das wird von einem Unittest überprüft (Listing 5.14). In Zeile 8 wird das leere Objekt `storage` erstellt. In Zeile 9 wird es dann wiederverwendet, um aufbauend auf der Kopie Änderungen mithilfe der `rebuild`-Methode durchzuführen.

Bei der Instanzvariablen `massnahmen` der Klasse `Storage` handelt es sich um ein `BuiltSet`. Der Aufruf von `b.massnahmen` gibt allerdings nicht dieses `BuiltSet` zurück. Wäre es so, so könnte die Operation `add` nicht darauf angewendet werden. Ein `BuiltSet` stellt keine Methoden zur Manipulation des `Sets` zur Verfügung. In Wahrheit gibt der Ausdruck `b.massnahmen` einen `SetBuilder` zurück. Das kann im generierten Quellcode nachgesehen werden (Listing 5.15, Z. 95-96).

Der `SetBuilder` wiederum erlaubt es, Änderungen am `Set` vorzunehmen und stellt dafür die – für ein `Set` übliche – Methode `add` bereit. Im Aufruf von `add` wird dann ein Objekt des Wertetyps `Massnahme` konstruiert (Listing 5.14, Z. 10). Dazu wird dieses Mal die anonyme Funktion zum Konstruieren der Maßnahme gleich dem Konstruktor übergeben.

Diesmal konstruiert die Methode `serializers.serializeWith` mit dem generierten Serialisierer `Storage.serializer` ein weiteres `JSON`-Objekt (Z. 12). Genau wie zuvor wird ein

```

7  test('Storage with one Massnahme serialises without error', () {
8    var storage = Storage();
9    storage = storage.rebuild((b) => b.massnahmen.add(
10      Massnahme((b) => b.identifikatoren.massnahmenTitel = "Massnahme 1")));
11
12    var actualJson = serializers.serializeWith(Storage.serializer, storage);
13
14    var expectedJson = {
15      "massnahmen": [
16        {
17          "guid": storage.massnahmen.first.guid,
18          "letzteBearbeitung": {
19            "letztesBearbeitungsDatum": storage
20              .massnahmen
21              .first
22              .letzteBearbeitung
23              .letztesBearbeitungsDatum
24              .microsecondsSinceEpoch,
25            "letzterStatus": "bearb"
26          },
27          "identifikatoren": {"massnahmenTitel": "Massnahme 1"}
28        }
29      ]
30    };
31    expect(actualJson, equals(expectedJson));

```

Listing 5.14.: Unittest der Serialisierung der Maßnahmenliste, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/test/data_model/storage_test.dart](#)

```

91 class StorageBuilder implements Builder<Storage, StorageBuilder> {
92   _$Storage? _$v;
93
94   SetBuilder<Massnahme>? _massnahmen;
95   SetBuilder<Massnahme> get massnahmen =>
96     _$this._massnahmen ??= new SetBuilder<Massnahme>();

```

Listing 5.15.: Instanzvariable *massnahmen* gibt einen *SetBuilder* zurück, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_model/storage.g.dart](#)

JSON-Dokument vorbereitet (Z. 14-30), welches der *Matcher* `equals` mit dem serialisierten Dokument des soeben konstruierten Objektes `storage` vergleicht (Z. 31). Das *JSON*-Dokument unterscheidet sich nur darin, dass es einen Knoten namens `'massnahmen'` enthält, der als Wert eine Liste hat. Die Liste hat nur ein Element. Weil dieses Mal das Objekt des Typs `Massnahme` nicht direkt zugreifbar ist, muss es zunächst über die Liste der Maßnahmen aus dem `storage`-Objekt abgerufen werden. Das ist mit dem Befehl `first` möglich, der das erste Objekt – und in diesem Fall einzige Objekt – der Kollektion zurückgibt (Z. 17, 21). Darüber kann erneut der `guid` und das `letztesBearbeitungsDatum` abgerufen werden.

Ein weiterer Unittest überprüft, ob auch die Deserialisierung eines `storage`-Objektes erfolgreich ist. Er ist in Listing C.1 im Anhang C auf Seite 191 zu finden. Auch dieser Test ist der Deserialisierung des Objektes des Typs `Massnahme` sehr ähnlich. Er unterscheidet sich nur darin, dass das `Massnahme`-Objekt in der Liste `massnahmen` des `storage`-Objektes enthalten ist.

5.4. Der Haupteinstiegspunkt

Das Listing 5.16 zeigt den Haupteinstiegspunkt des Programms. Darin ist erkennbar, dass sich die Applikation in drei Rubriken einteilen lässt:

- das *Model* (Z. 24)
- der *View* (Z. 35-38)
- das *ViewModel*. (Z. 25)

5.4.1. Das *Model-View-ViewModel*-Entwurfsmuster

Das *Model-View-ViewModel*-Entwurfsmuster – kurz *MVVM* – wurde zunächst von John Gossman für die *Windows Presentation Foundation* beschrieben. Das *Model* beschreibt die Datenzugriffs-Komponente, welche die Daten in relationalen Datenbanken oder hierarchischen Datenstrukturen wie *XML* oder *JSON* ablegt. Der *View* beschreibt die Oberflächenelemente wie Texteingabefelder und Buttons. Diese beiden Komponenten sind auch aus dem *Model-View-Controller*-Entwurfsmuster bekannt. Das *Model-View-ViewModel*-Entwurfsmuster ist eine Weiterentwicklung davon und integriert das sogenannte *ViewModel*. Es ist dafür zuständig, als Schnittstelle zwischen *View* und *Model* zu fungieren. Die Daten des *Models* lassen sich in der Regel nicht direkt mit Oberflächenelementen verknüpfen. Denn es kann notwendig sein, dass die Oberfläche weitere temporäre Daten benötigt, die aber nicht mit den Daten des *Models* gespeichert werden sollen. Das *ViewModel* übernimmt diese Arbeit, indem es die Daten des *Models* abrufen und sie in veränderter Form den Oberflächenelementen zur Verfügung stellt. Andersherum formt es die Eingaben in der Nutzeroberfläche so um, dass sie im strikten Datenmodell des *Models* Platz finden.¹⁵

`MassnahmenModel` (Z. 24) verwaltet die eingegebenen Daten der Maßnahmen und nutzt die Abhängigkeit `MassnahmenJsonFile`, um die Daten auf einem Datenträger als eine *JSON*-Datei zu speichern. Somit gehören diese beiden Klassen dem *Model* an.

`MassnahmenFormViewModel` (Z. 25) greift die Daten des *Models* ab und formt diese um, so dass sie von dem *View* `MassnahmenDetailScreen` (Z. 38) verändert werden können. Sollen die Daten gespeichert werden, so stellt `MassnahmenFormViewModel` ebenfalls Methoden zur Verfügung, um die Daten wieder in das Format des *Models* einpflegen zu können.

`MassnahmenMasterScreen` (Z. 36) stellt eine Ausnahme dar, denn dieser *View* präsentiert die Daten aus dem *Model* ohne eine Schnittstelle über ein *ViewModel*. Das ist möglich, weil die Daten nicht manipuliert, sondern nur angezeigt werden müssen.

¹⁵Vgl. Gossman, *Introduction to Model/View/ViewModel pattern for building WPF apps*.

Damit sowohl *ViewModel* als auch *Model* von jedem *View* heraus abrufbar sind, werden sie in eine Art Service eingefügt (Z. 23-25). Das *Widget AppState* ist dieser Service. Es erhält das *Model* (Z. 24) und das *ViewModel* (Z. 25) im Konstruktor. Die Abhängigkeit zum Schreiben des *Models* in eine *JSON*-Datei *MassnahmenJsonFile* bekommt das *Model* ebenfalls im Konstruktor übergeben (Z. 24). *AppState* ist das erste Element, welches im *Widget*-Baum auftaucht. Die gesamte restliche Applikation ist als Kindelement hinterlegt (Z. 26). Damit können alle *Widgets* auf den Service zugreifen.

```

18 class MassnahmenFormApp extends StatelessWidget {
19   const MassnahmenFormApp({Key? key}) : super(key: key);
20
21   @override
22   Widget build(BuildContext context) {
23     return AppState(
24       model: MassnahmenModel(MassnahmenJsonFile()),
25       viewModel: MassnahmenFormViewModel(),
26       child: MaterialApp(
27         title: 'Maßnahmen',
28         theme: ThemeData(
29           primarySwatch: Colors.lightGreen,
30           accentColor: Colors.green,
31           primaryIconTheme: const IconThemeData(color: Colors.white),
32         ),
33         initialRoute: MassnahmenMasterScreen.routeName,
34         routes: {
35           MassnahmenMasterScreen.routeName: (context) =>
36             const MassnahmenMasterScreen(),
37           MassnahmenDetailScreen.routeName: (context) =>
38             const MassnahmenDetailScreen()
39         },
40       ));
41   }
42 }

```

Listing 5.16.: Der Haupteinstiegspunkt *MassnahmenFormApp*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/main.dart](#)

5.4.2. Service Locator und Dependency Injection

Das *service locator pattern* folgt dem Umsetzungsparadigma *inversion of control* – deutsch Umkehrung der Steuerung. Frameworks folgen diesem Muster, indem sie als erweiterbare Skelett-Applikationen fungieren. Anstatt, dass die Applikation den Programmfluss steuert und dabei selbst Funktionen aufruft, wird die Programmflusssteuerung an das Framework abgegeben und mithilfe von Ereignissen ermöglicht, dass das Framework Funktionen des Nutzers aufruft.¹⁶ Im *service locator pattern* werden Komponenten darüber hinaus zentral registriert und über dieses Register anderen Komponenten zur Interaktion zur Verfügung gestellt.¹⁷ Damit ist es möglich, die Komponenten nicht direkt miteinander verknüpfen zu müssen. Vor allem für automatisierte Tests ist dies von Vorteil, da solche Abhängigkeiten ausgetauscht werden können, um ganz spezielle Teilfunktionalitäten eines Programms zu testen. Mehr dazu im Kapitel 5.12 *Integrationstest zum Test der Oberfläche* auf Seite 106.

¹⁶Vgl. Johnson und Foote, *Designing reusable classes*.

¹⁷Vgl. Fowler, *Inversion of Control Containers and the Dependency Injection pattern*.

Anders als der Name vermuten lässt, steuert `MaterialApp` (Z. 26) nicht nur das Aussehen der Applikation im *Material Design*-Look, sondern das *Widget* stellt auch Grundfunktionalitäten einer App wie etwa den *Navigator* bereit. Damit hat die Applikation die Möglichkeit – ähnlich wie bei einer Website – auf Unterseiten zu navigieren. Hat der Benutzer die Arbeit in der Unterseite vollendet, so kann der *Navigator* gebeten werden, zur vorherigen Ansicht zurückzukehren. Mit dem Parameter `routes` (Z. 34-39) erfolgt die Angabe der Unterseiten, die besucht werden können. Über `initialRoute` (Z. 33) kann die Startseite angegeben werden.

5.4.3. Der Service für den applikationsübergreifenden Zustand

Um Daten für alle Kindelemente zugreifbar zu machen, werden die sogenannten *InheritedWidgets* genutzt. Der Service `AppState` (Listing 5.17) ist ein solches *InheritedWidget*. Im Konstruktor erhält es zunächst den Parameter des Typs `key` (Z. 7). Es ist gängige Praxis, in *Flutter*, jedem *Widget* im Konstruktor zu ermöglichen, einen solchen Schlüssel zu übergeben. Es ist jedoch optional. Ein solcher Schlüssel kann genutzt werden, um das *Widget* eindeutig zu identifizieren und es unter anderem über den Schlüssel wiederzufinden. In den Zeilen 8 und 9 werden das *Model* und das *ViewModel* dem Objekt im Konstruktor übergeben. In den Zeilen 13 und 14 sind sie deklariert. Der letzte Parameter im Konstruktor ist `child` (Z. 10). Ihm wird der *Widget*-Baum übergeben, der Zugriff auf das *InheritedWidget* haben soll.

Der Aufruf des Basiskonstruktors mit den Argumenten `key` und `child` ist in Zeile 11 zu sehen. Die Basisklasse von *InheritedWidget* ist *ProxyWidget* und erhält exakt dieselben Argumente. Das *ProxyWidget* verwendet das Kindelement, um es im *Widget*-Baum unterhalb von sich selbst zu zeichnen. Eine eigene Methode zum Zeichnen muss also nicht für das *InheritedWidget* implementiert werden. Die einzige Methode, welche implementiert werden muss, ist `updateShouldNotify` (Z. 23). Immer dann, wenn das *InheritedWidget* selbst aktualisiert wird, kann es alle *Widgets*, die davon abhängig sind, benachrichtigen. In dem Fall werden diese *Widgets* ebenfalls neu gezeichnet. Für die Formularapplikation ist das allerdings nicht gewünscht. Die Aktualisierung der Oberfläche soll in den nachfolgenden Schritten selbst kontrolliert werden. Deshalb erfolgt die Rückgabe `false`, da in Zukunft nicht gewünscht ist, den Applikationszustand komplett auszutauschen. Um die Aktualisierung der Oberfläche kümmern sich sowohl *Model* als auch *ViewModel*.

Damit ein *Widget* eine Abhängigkeit von dem `AppState` anmelden kann, verwendet es in seiner eigenen *build*-Methode den Ausdruck `dependOnInheritedWidgetOfExactType<AppState>()`. Der Aufruf der Methode erfolgt auf dem Objekt vom Typ *BuildContext*. Weil dieser Kontext bei jedem Zeichnen allen Kindern übergeben wird, kann jedes Kind darüber die Väterelemente wiederfinden.

Damit der Aufruf leichter lesbar und kürzer ist, empfiehlt das *Flutter*-Team, eine eigene Klassenmethode zu erstellen, welche die Methode für den Benutzer aufruft (Z. 16-17). Auch eine Fehlermeldung kann bei dieser Auslagerung geworfen werden, sollte im Kontext kein Objekt des gewünschten Typs vorhanden sein (Z. 18). Das *Widget*, welches den *AppState* benötigt, kann dann über die vereinfachte Schreibweise `AppState.of(context)` darauf zugreifen.

```

5 class AppState extends InheritedWidget {
6   const AppState({
7     Key? key,
8     required this.model,
9     required this.viewModel,
10    required Widget child
11  }) : super(key: key, child: child);
12
13   final MassnahmenFormViewModel viewModel;
14   final MassnahmenModel model;
15
16   static AppState of(BuildContext context) {
17     final AppState? result = context.dependOnInheritedWidgetOfExactType<AppState>();
18     assert(result != null, "Kein AppState im 'context' gefunden");
19     return result!;
20   }
21
22   @override
23   bool updateShouldNotify(covariant AppState oldWidget) => false;
24 }

```

Listing 5.17.: Der Service *AppState* für den applikationsübergreifenden Zustand, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/app_state.dart](#)

Abbildung 5.4 zeigt die Beziehung zwischen den Bildschirmen und dem *AppState* auf. Sowohl *MassnahmenMasterScreen* und *MassnahmenDetailScreen* müssen auf *MassnahmenModel* und *MassnahmenFormViewModel* zugreifen können. Zu diesem Zweck erstellt *MassnahmenFormApp* den *AppState*. Er enthält sowohl *ViewModel* als auch *Model*. Über ihn können beide Bildschirme auf *Model* und *ViewModel* zugreifen.

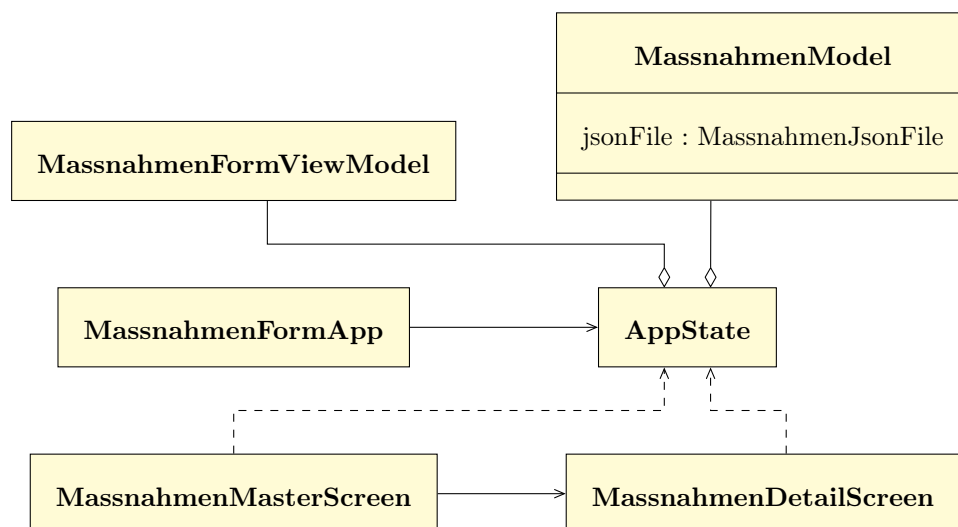


Abbildung 5.4.: UML-Diagramm der Beziehungen zwischen den Bildschirmen und dem *AppState*, Quelle: Eigene Abbildung

5.5. Speichern der Maßnahmen in eine *JSON*-Datei

Das *Model* wird durch die Klasse `MassnahmenJsonFile` in eine *JSON*-Datei gespeichert (Listing 5.18). Der Dateipfad wird dabei durch die Methode `_localMassnahmenJsonFile` (Z. 8-11) abgerufen. Die Hilfsmethode `getApplicationSupportDirectory` (Z. 9) gibt aus dem Nutzerverzeichnis des aktuellen Nutzers den zur Applikation zugeordneten Dateiordner zurück. Auf Windows-Betriebssystemen wäre das beispielsweise `C:\Users\AktuellerNutzer\AppData\Roaming\com.example\conditional_form`.

Dadurch, dass dem Methoden-Bezeichner `_localMassnahmenJsonFile` ein Unterstrich vorangestellt ist, ist die Methode privat und kann nur innerhalb der Klasse aufgerufen werden. *Dart* hat damit eine Konvention zum Standard werden lassen. In Programmiersprachen wie beispielsweise C++ wurde der Unterstrich zusätzlich den Bezeichnern von Instanzattributen vorangestellt, die mit dem *private* Schlüsselwort gekennzeichnet sind, damit sie überall im Quellcode als private Attribute identifizierbar sind, ohne dazu die Klassendefinition ansehen zu müssen. In *Dart* gibt es dagegen das *private* Schlüsselwort nicht. Stattdessen wird der Unterstrich vor dem Bezeichner verwendet, um ein Instanzattribut privat zu deklarieren.

Die *Getter*-Methode `_localMassnahmenJsonFile` hat den Rückgabetypp `Future<File>` und ist zudem mit dem Schlüsselwort `async` gekennzeichnet (Z. 8). Asynchron muss die Methode deshalb sein, weil sie auf den Aufruf `getApplicationSupportDirectory` warten muss, der ebenfalls asynchron abläuft (Z. 9).

Der Funktion `saveMassnahmen` (Z. 13-16) wird ein *JSON*-Objekt in Form einer Hashtabelle übergeben. Sie ruft die Hilfs-*Getter*-Methode `_localMassnahmenJsonFile` (Z. 14) auf und schreibt den Dateiinhalt in die Datei des abgefragten Pfades (Z. 15). Zuvor wird dazu das *JSON*-Objekt in eine textuelle Repräsentation überführt. Dazu dient die Funktion `jsonEncode`.

Das Äquivalent dazu stellt die Methode `readMassnahmen` (Z. 18-30) dar. Auch sie ruft den Dateipfad ab (Z. 19), überprüft allerdings im nächsten Schritt, ob die Datei bereits existiert (Z. 21). Sollte das der Fall sein, so wird die Datei eingelesen (Z. 23). Die textuelle Repräsentation aus der Datei wird mittels der Methode `jsonDecode` in ein *JSON*-Objekt in Form einer Hashtabelle gespeichert (Z. 24) und schließlich zurückgegeben (Z. 26). Sollte die Datei nicht existieren, führt das zu einer Ausnahme (Z. 28), welche von der aufrufenden Funktion behandelt werden kann.

```

7  class MassnahmenJsonFile {
8    Future<File> get _localMassnahmenJsonFile async {
9      var directory = await getApplicationSupportDirectory();
10     return File("${directory.path}/Maßnahmen.json");
11   }
12
13   Future<void> saveMassnahmen(Map<String, dynamic> massnahmenAsJson) async {
14     var file = await _localMassnahmenJsonFile;
15     await file.writeAsString(jsonEncode(massnahmenAsJson));
16   }
17
18   Future<Map<String, dynamic>> readMassnahmen() async {
19     var file = await _localMassnahmenJsonFile;
20
21     var fileExists = await file.exists();
22     if (fileExists) {
23       final fileContent = await file.readAsString();
24       final jsonObject = jsonDecode(fileContent) as Map<String, dynamic>;
25
26       return jsonObject;
27     } else {
28       throw MassnahmenFileDoesNotExistException("$file was not found");
29     }
30   }
31 }

```

Listing 5.18.: Die Klasse *MassnahmenJsonFile*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/persistence/massnahmen_json_file.dart](#)

5.6. Abhängigkeit zum Verwalten der Maßnahmen

Die Art und Weise, wie die Maßnahmen abgerufen werden, sollte nach Möglichkeit abstrahiert werden. Das erlaubt, den Mechanismus in Zukunft auszutauschen, ohne dabei den Rest der Applikation verändern zu müssen. So wäre es beispielsweise denkbar, statt zu einer *JSON*-Datei eine direkte Verbindung zu einer relationalen Datenbank herzustellen. Auch das Austauschen der Abhängigkeit mit einem Platzhalter, der lediglich die Aufrufe der Methoden zählt, ist damit möglich. Ein solches Platzhalterobjekt wird *Mock* genannt und für automatisiertes Testen eingesetzt (siehe Kapitel 5.12 *Integrationstest zum Test der Oberfläche* auf Seite 106). Ebenso abstrahiert werden soll der Umgang mit Ausnahmen. Sollte die Datei nicht verfügbar sein, so muss die Oberfläche davon nicht zwingend betroffen sein. Stattdessen kann der Service sich entscheiden, eine leere Liste von Maßnahmen zurückzugeben. Sobald die Liste manipuliert wird, kann eine neue Datei angelegt werden und sie mit den eingegebenen Daten beschrieben werden. Die Klasse `MassnahmenModel` (Listing 5.19) tut genau das.

Sie bekommt `MassnahmenJsonFile` im Konstruktor übergeben (Z. 11). Daraufhin ruft der Konstruktor gleich die `init`-Methode auf (Z. 12), welche in den Zeilen 15 bis 22 deklariert ist. Darin wird der *Stream* `storage` (Z. 19) initialisiert. Es handelt sich um eine Erweiterung eines *broadcast streams* mit dem Namen `BehaviorSubject` (Z. 9). Es entstammt dem Paket *RxDart*, welches die *Streams* in *Dart* um eine Reihe von weiteren Funktionalitäten erweitert. Der Begriff *Behaviour* stammt aus der funktionalen reaktiven Programmierung: “Behaviors are time-varying, reactive values”¹⁸ Das *Subject* – deutsch *Subjekt* – gehört wiederum neben dem *Beobachter* zu den Akteuren des *Beobachter*-Entwurfsmusters und ist dafür zuständig, die *Beobachter* über Änderungen zu benachrichtigen.¹⁹ Ein `BehaviorSubject` hat im Gegensatz zu gewöhnlichen *Streams* die Besonderheit, dass es den Wert des letzten Ereignisses zwischenspeichert. Die *broadcast streams* haben für gewöhnlich den Nachteil, dass neue Zuhörer des *Streams* nur die neuen Ereignisse erhalten. Alle in der Vergangenheit erfolgten Ereignisse sind nicht mehr verfügbar. Vor allem dann, wenn in der Oberfläche der letzte Wert eines *Streams* verwendet werden soll, um Elemente zu zeichnen, ist das von einem besonderen Nachteil. Denn wenn der *Stream* zuvor initialisiert wurde, so gibt es keine Daten zu dem Zeitpunkt, wenn die Oberfläche gezeichnet wird. Sollte die Oberfläche gezeichnet werden, bevor der *Stream* initialisiert wurde, so existieren ebenfalls keine Daten. Hier kommt das `BehaviorSubject` ins Spiel. Sobald die Oberfläche gezeichnet wird und der *Stream* bereits initialisiert ist, kann dennoch auf den zuletzt übertragenen Wert zurückgegriffen werden. Anschließend überträgt der *Stream* die folgenden Aktualisierungen für die Oberfläche mit jedem neuen Ereignis, so wie es für *Streams* üblich ist.

Der *Stream* kann nicht bereits in der Initialisierungsliste des Konstruktors mit den Daten aus der *JSON*-Datei gefüllt werden. Das liegt daran, dass die *JSON*-Daten dazu zunächst

¹⁸Elliott und Hudak, *Functional Reactive Animation*, S. 1.

¹⁹Vgl. Gamma u. a., *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, S. 288.

gelesen werden müssen, was nur durch eine Reihe von asynchronen Operationen möglich ist. In einer Initialisierungsliste können allerdings keine asynchronen Operationen ausgeführt werden. Deshalb wird `init` erst im Konstruktor-Körper aufgerufen (Z. 12). Damit der *Stream* anfangs nicht leer ist, füllt ihn der benannte Konstruktor `seeded` mit einem leeren Objekt des Typs `Storage` (Z. 9). Sobald die Datei gelesen (Z. 17) und deserialisiert wurde (Z. 20), erhält der *Stream* über die *Setter*-Methode `value` ein neues Ereignis mit dem gelesenen Wert (Z. 19). Die Initialisierung ist von einem `try`-Block umgeben. Sollte die Initialisierung fehlschlagen, weil die *JSON*-Datei nicht existiert, wird die entsprechende Fehlerbehandlung ausgeführt (Z. 21). Diese ist leer, da sich im *Stream* bereits ein leeres `Storage`-Objekt befindet. Mit diesem leeren Objekt kann die Oberfläche weiterarbeiten. In Zukunft könnte es sinnvoll sein, innerhalb der Fehlerbehandlung eine Meldung an den Benutzer zu geben, um darüber zu informieren, dass eine neue Datei angelegt wurde.

Mit `putMassnahmeIfAbsent` (Z. 24-33) steht eine Methode bereit, um gleichzeitig sowohl die Oberfläche als auch die *JSON*-Datei zu aktualisieren. Sollte die eingetragene Maßnahme schon existieren, wird sie zunächst gelöscht (Z. 26). In jedem Fall wird die neue Maßnahme der Menge `massnahmen` hinzugefügt (Z. 27). Durch Austauschen des gesamten Objektes mit der Zuweisung zu `storage.value` (Z. 25) erhält der *Stream* erneut ein neues Ereignis, womit er die Oberfläche benachrichtigen kann, sich neu zu zeichnen. Außerdem wird die Serialisierung des `Storage`-Objektes angestoßen (Z. 29-30). Die neue Liste von Maßnahmen wird im darauffolgenden Schritt zurück in die *JSON*-Datei gespeichert (Z. 32).

```

7 class MassnahmenModel {
8   final MassnahmenJsonFile jsonFile;
9   final storage = BehaviorSubject<Storage>.seeded(Storage());
10
11   MassnahmenModel(this.jsonFile) {
12     init();
13   }
14
15   init() async {
16     try {
17       final massnahmenAsJson = await jsonFile.readMassnahmen();
18
19       storage.value =
20         serializers.deserializeWith(Storage.serializer, massnahmenAsJson)!;
21     } on MassnahmenFileDoesNotExistException {}
22   }
23
24   putMassnahmeIfAbsent(Massnahme massnahme) async {
25     storage.value = storage.value.rebuild((b) => b.massnahmen
26       ..removeWhere((m) => m.guid == massnahme.guid)
27       ..add(massnahme));
28
29     var serializedMassnahmen =
30       serializers.serializeWith(Storage.serializer, storage.value);
31
32     await jsonFile.saveMassnahmen(serializedMassnahmen as Map<String, dynamic>);
33   }
34 }

```

Listing 5.19.: Die Klasse `MassnahmenModel`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/data_access/massnahmen_model.dart](#)

5.7. Übersichtsbildschirm der Maßnahmen

Der erste Bildschirm – für die Übersicht der Maßnahmen – kann auf das im letzten Schritt erstellte *Model* zugreifen. In Listing 5.20 ist die Struktur des Übersichtsbildschirms zu sehen. Über die Route `/massnahmen_master` ist der Bildschirm erreichbar (Z. 14). Die `build`-Methode zeichnet die Oberfläche (Z. 19-99).

Mittels `AppState.of(context)` ist nun der Zugriff auf sowohl *Model* als auch *ViewModel* möglich. Zur einfacheren Verwendung sind sie als lokale Variablen zwischengespeichert (Z. 20-21).

Das *Widget Scaffold* – deutsch Gerüst – stellt ein grundlegendes Layout mit einer Überschrift und einem Bereich für den Inhalt bereit (Z. 23). Das `Scaffold` kann auch Mitteilungen an den Benutzer am unteren Bildschirmrand einblenden.

Die Überschrift wird in der sogenannten `AppBar` hinterlegt (Z. 24). Sie unterstützt weitere Funktionalitäten. Sollte es sich bei der aktuell besuchten Route um eine Unterseite handeln, taucht links von der Titel-Überschrift ein Button zum Zurücknavigieren auf. Weiterhin können rechts von der Titelleiste Aktionsbuttons hinzugefügt werden. Das ist für die Formularanwendung allerdings nicht nötig.

Zusätzlich kann dem `Scaffold` ein Button für die primäre Aktion auf diesem Bildschirm hinzugefügt werden: der sogenannte `FloatingActionButton` (Z. 88-97). Bei Aktivierung dieses Buttons navigiert die Applikation zur Eingabemaske, um eine neue Maßnahme anzulegen (Z. 96).

Das Eingabeformular sollte den Benutzer auffordern, tatsächlich leere Eingabefelder zu füllen. Deshalb muss die Aktivierung des Buttons auch das *ViewModel* neu initialisieren. Dies geschieht durch Zuweisung einer leeren Maßnahme zur *Setter*-Methode `vm.model` (Z. 95). Ohne die Neuinitialisierung würde die Eingabemaske immer die zuletzt eingetragene Maßnahme enthalten, was zu einer großen Verwirrung beim Benutzer führen würde.

Der `FloatingActionButton` erhält den Schlüssel `createNewMassnahmeButtonKey` (Z. 89). Er ist als `GlobalKey` deklariert (Z. 11). Er findet beim Integrationstest Anwendung, um den Button zu finden (Siehe 5.12.3 *Test des Übersichtsbildschirms* auf Seite 111).

Der Inhaltsbereich des `Scaffold` beinhaltet das *Widget StreamBuilder* (Z. 27). Er kann auf *Streams* horchen, welche die Ereignisse des Typs `Storage` übermitteln. Er horcht auf Änderungen im *Model*, um genau zu sein auf Änderungen des *Streams* `model.storage` (Z. 28). Sobald der `StreamBuilder` ein Ereignis erhält, so führt er die Methode aus, die als Argument des Parameters `builder` hinterlegt ist. Alle *Widgets* außerhalb davon, wie etwa

das `Scaffold`, erhalten dabei keine Aufforderung zum Neuzeichnen, sobald eine Maßnahme hinzugefügt wird. Das wirkt sich positiv auf die Laufzeitgeschwindigkeit aus.

```

11 final createNewMassnahmeButtonKey = GlobalKey();
12
13 class MassnahmenMasterScreen extends StatelessWidget {
14   static const routeName = '/massnahmen_master';
15
16   const MassnahmenMasterScreen({Key? key}) : super(key: key);
17
18   @override
19   Widget build(BuildContext context) {
20     final model = AppState.of(context).model;
21     final vm = AppState.of(context).viewModel;
22
23     return Scaffold(
24       appBar: AppBar(
25         title: const Text('Maßnahmen Master'),
26       ),
27       body: StreamBuilder<Storage>(
28         stream: model.storage,
29         builder: (context, _) {
30           return SingleChildScrollView(
31             /* ... */
32           );
33         },
34       ),
35       floatingActionButton: FloatingActionButton(
36         key: createNewMassnahmeButtonKey,
37         child: const Icon(
38           Icons.post_add_outlined,
39           color: Colors.white,
40         ),
41         onPressed: () {
42           vm.model = Massnahme();
43           Navigator.of(context).pushNamed(MassnahmenDetailScreen.routeName);
44         },
45       ),
46     );
47   }
48 }

```

Listing 5.20.: Die Struktur der Klasse *MassnahmenMasterScreen*, Quelle: Eigenes Listing, Datei: [Quelcode/Schritt-1/conditional_form/lib/screens/massnahmen_master.dart](#)

5.7.1. Auflistung der Maßnahmen im Übersichtsbildschirm

Der Inhalt der `builder`-Methode ist in Listing 5.21 auf Seite 89 dargestellt. Das erste *Widget* ist ein `SingleChildScrollView` (Z. 30).

Das Argument *scrollDirection* ist nicht gefüllt, weshalb die Standardoption – die vertikale Scrollrichtung – gewählt wird. Sollte die Liste der Maßnahmen die Höhe des Fensters überschreiten, so kann der Benutzer vertikal über die Liste scrollen.

Das Kind des Scrollbereichs ist ein `Column-Widget` (Z. 31). Es zeichnet *Widgets*, die als Argument des Parameters `children` gesetzt sind, von oben nach unten (Z. 33). Der Parameter `crossAxisAlignment` gibt an, wie die Kindelemente ausgerichtet sein sollen (Z. 32).

`crossAxis` bedeutet dabei die entgegengesetzte Achse zur Anzeigerichtung. Da die `Column` vertikal zeichnet, ist mit `crossAxis` die horizontale Achse gemeint. `CrossAxisAlignment.start` beschreibt, dass Elemente entlang der horizontalen Achse an deren Startpunkt auszurichten sind. Dadurch sind alle Elemente der Liste linksbündig.

Zuerst kommt die Auflistung der Maßnahmen, welche als abgeschlossen markiert sind. Die Überschrift `"Abgeschlossen"` (Z. 37) soll einen Abstand von jeweils 16 Pixel in alle Richtungen haben. Das ermöglicht das `Padding-Widget` (Z. 34-40) mit dem Argument `EdgeInsets.all(16.0)`. Nach der Überschrift erscheint als zweites Element in der `Column` ein weiterer `SingleChildScrollView` (Z. 41-58), allerdings dieses Mal mit horizontaler Scroll-Richtung (Z. 42). Sollten die Informationen der Maßnahmen die Breite des Fensters überschreiten, kann der Nutzer von links nach rechts scrollen.

Die Informationen der Maßnahmen werden in einer Tabelle angezeigt. Dies übernimmt das selbst geschriebene `Widget MassnahmenTable` (Z. 45). Als erstes Argument erfolgt die Übergabe der anzuzeigenden Maßnahmen aus dem `Model`. `storage.value.massnahmen` gibt den aktuellen Wert des `Streams` des `storage`-Objektes zurück und greift auf die Liste der Maßnahmen zu (Z. 46). Mit der Methode `where` (Z. 47) kann ein Filter auf die Liste angewendet werden. Die übergebene anonyme Funktion (Z. 47-49) überprüft, ob der letzte Status auf `fertig` gesetzt ist. Dazu reicht der Vergleich der Abkürzung. Nur wenn diese Bedingung erfüllt ist, bleibt die Maßnahme in der gefilterten Kollektion zurück. Ein solcher Filter gibt ein sogenanntes *lazy Iterable* zurück.²⁰ Erst beim Zugriff auf das Ergebnis findet der Filter Anwendung. Da es keinen Zwischenspeicher für die gefilterten Elemente gibt, filtert jeder Zugriff die Elemente neu. Der Aufruf `toSet` bewirkt allerdings das Speichern der Ergebnisse in einer Menge (Z. 50). Das `Widget MassnahmenTable` erhält das Resultat, um es in einer Tabelle anzuzeigen.

Ein weiterer Parameter ist `onSelect` (Z. 50). Als Argument kann eine Funktion mit genau einem Parameter gesetzt werden. Sollte der Benutzer in der Tabelle eine Maßnahme auswählen, so löst er damit die Funktion aus. Der erste Parameter enthält dann die ausgewählte Maßnahme. Daraufhin soll sich wieder die Eingabemaske öffnen (Z. 55-56). Dann beinhalten die Eingabefelder jedoch die Werte der ausgewählten Maßnahme. Um das zu erreichen, reicht eine Zuweisung der Maßnahme an das `ViewModel` (Z. 51). Allerdings soll die Maßnahme zuvor ein neues letztes Bearbeitungsdatum mit dem aktuellen Zeitstempel erhalten (Z. 51-53).

Unterhalb der Rubrik der finalen Maßnahmen listet die Übersicht die Maßnahmen, welche sich noch im Entwurf befinden (Z. 59-83). Daher ist das dritte Element der `Column` wiederum eine Überschrift: `"In Bearbeitung"` (Z. 62), gefolgt von einem weiteren horizontalen Scrollbereich (Z. 66-83) mit einer Tabelle von Maßnahmen (Z. 70-82). Der einzige Unterschied ist hier die Bedingung der Filterfunktion. Dieses Mal filtert die Kollektion auf Maßnahmen in Bearbeitung (Z. 73-74).

²⁰Vgl. Google LLC, *Flutter* / *where method*.


```

30 return SingleChildScrollView(
31   child: Column(
32     crossAxisAlignment: CrossAxisAlignment.start,
33     children: [
34       const Padding(
35         padding: EdgeInsets.all(16.0),
36         child: Text(
37           "Abgeschlossen",
38           style: TextStyle(fontSize: 20),
39         ),
40     ),
41     SingleChildScrollView(
42       scrollDirection: Axis.horizontal,
43       child: Padding(
44         padding: const EdgeInsets.all(16.0),
45         child: MassnahmenTable(
46           model.storage.value.massnahmen
47             .where((m) =>
48               m.letzteBearbeitung.letzterStatus ==
49               LetzterStatus.fertig.abbreviation)
50             .toSet(), onSelect: (selectedMassnahme) {
51               vm.model = selectedMassnahme.rebuild((m) => m
52                 ..letzteBearbeitung.letztesBearbeitungsDatum =
53                 DateTime.now().toUtc());
54             },
55           Navigator.of(context)
56             .pushNamed(MassnahmenDetailScreen.routeName);
57         )),
58     ),
59     const Padding(
60       padding: EdgeInsets.all(16.0),
61       child: Text(
62         "In Bearbeitung",
63         style: TextStyle(fontSize: 20),
64       ),
65     ),
66     SingleChildScrollView(
67       scrollDirection: Axis.horizontal,
68       child: Padding(
69         padding: const EdgeInsets.all(16.0),
70         child: MassnahmenTable(
71           model.storage.value.massnahmen
72             .where((m) =>
73               m.letzteBearbeitung.letzterStatus ==
74               LetzterStatus.bearb.abbreviation)
75             .toSet(), onSelect: (selectedMassnahme) {
76               vm.model = selectedMassnahme.rebuild((m) => m
77                 ..letzteBearbeitung.letztesBearbeitungsDatum =
78                 DateTime.now().toUtc());
79             },
80           Navigator.of(context)
81             .pushNamed(MassnahmenDetailScreen.routeName);
82         )),
83     ),
84   ],
85 ),
86 );

```

Listing 5.21.: Die Ausgabe der Maßnahmen, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_master.dart](#)

5.8. Das *MassnahmenTable*-Widget

Die `MassnahmenTable` ist ein `StatelessWidget` (Listing 5.22, Z. 6). Zur Anzeige eignet sich das `Widget Table` (Z. 15-31). Im Verlauf der Erstellung der Arbeit wurde versucht, das `Widget DataTable` zu verwenden. Doch im Gegensatz zur `DataTable` erlaubt es das `Widget Table`, unterschiedlich hohe Zeilen zu zeichnen. Die Höhe der Zeile wird dazu in Abhängigkeit von dem benötigten Platz des Inhalts der Zellen berechnet. Die Breite und Ausrichtung der Spalten kann konfiguriert werden. Die Eigenschaft `IntrinsicColumnWidth` sorgt dafür, dass die Spalten immer genau so groß sind, wie der Inhalt es benötigt (Z. 17). Zeilenumbrüche für die Texte in den Spalten sind somit nicht notwendig. `TableCellVerticalAlignment.middle` lässt die Tabelle die Inhalte zentriert darstellen (Z. 18).

```

4 typedef OnSelectCallback = void Function(Massnahme selectedMassnahme);
5
6 class MassnahmenTable extends StatelessWidget {
7   final Set<Massnahme> _massnahmenToDisplay;
8   final OnSelectCallback? onSelect;
9
10  const MassnahmenTable(this._massnahmenToDisplay, {this.onSelect, Key? key})
11    : super(key: key);
12
13  @override
14  Widget build(BuildContext context) {
15    return Table(
16      border: TableBorder.all(width: 3),
17      defaultColumnWidth: const IntrinsicColumnWidth(),
18      defaultVerticalAlignment: TableCellVerticalAlignment.middle,
19      children: [
20        TableRow(children: [
21          _buildColumnHeader(const Text("Zuletzt bearbeitet am")),
22          _buildColumnHeader(const Text("Maßnahmentitel"))
23        ]),
24        ..._massnahmenToDisplay.map((m) {
25          return TableRow(children: [
26            _buildSelectableCell(m, Text(m.letzteBearbeitung.formattedDate)),
27            _buildSelectableCell(m, Text(m.identifikatoren.massnahmenTitel)),
28          ]);
29        }).toList(),
30      ],
31    );
32  }

```

Listing 5.22.: Die Klasse `MassnahmenTable`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/massnahmen_table.dart](#)

Der Parameter `children` erhält als Argument eine Liste von `TableRow`-Elementen (Z. 19-30). Die erste Tabellenzeile beinhaltet die Spaltenbezeichnungen (Z. 20-23). Jede `TableRow` hat wiederum den Parameter `children`. Das Argument bezieht sich hier auf die Zellen in der Zeile. Dabei ist wichtig, dass jede `TableRow` die gleiche Anzahl von Zellen hat. Weicht nur eine Zeile davon ab, zeichnet sich die gesamte Tabelle nicht und eine Ausnahme wird ausgelöst.

Nach den Spaltenbezeichnungen folgen die Zeilen für die Daten der Maßnahmen (Z. 24-29). Die Methode `map` (Z. 24) ermöglicht es dazu, durch die Liste der Maßnahmen zu iterieren und für jede Maßnahme ein Element eines völlig anderen Typs – in diesem Fall `TableRow` – zurückzugeben. Bei den vorangestellten Punkten `...` in Zeile 24 handelt es sich um den *spread operator*. Die Filtermethode `map` und die darauffolgende Methode `toList` liefern eine Liste von `TableRow`-Elementen. Die umgebende Liste der Zeilen `children` (Z. 19-30) erwartet jedoch Elemente des Typs `TableRow` und keine Elemente des Typs `List`. Der *spread operator* ermöglicht, alle Elemente der inneren Liste in die äußere Liste einzufügen.²¹

Für die Spaltenbezeichnungen wurde eine Hilfsmethode kreiert: `_buildColumnHeader` (Listing 5.23). Sie zeichnet die Spalten mit einem Abstand von `8` Pixeln in alle Richtungen.

```
34 Widget _buildColumnHeader(Widget child) => Padding(
35   padding: const EdgeInsets.all(8.0),
36   child: child,
37 );
```

Listing 5.23.: Die Hilfsmethode `_buildColumnHeader`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/massnahmen_table.dart](#)

Eine weitere Hilfsmethode `_buildSelectableCell` erstellt Zellen, die anklickbar sind (Listing 5.24). Das Widget `TableRowInkWell` (Z. 41-51) kann in Tabellen verwendet werden, um einen anklickbaren Bereich zu erstellen. Beim Anklicken breitet sich ausgehend von der Position des Klicks ein Tintenklecks aus. Dabei überschreitet der Tintenklecks nicht den Bereich, der von der umgebenden Zeile begrenzt ist. Beim Auslösen des Ereignisses `onTap` erfolgt die Ausführung der Rückruffunktion `onSelect` (Z. 44) mit der ausgewählten Maßnahme. Doch zuvor muss überprüft werden, ob die Rückruffunktion auch initialisiert wurde (Z. 43). Wie hier zu sehen ist, reicht es nicht aus, abzufragen, ob `onSelect` gesetzt ist. Es erfolgt keine Typ-Beförderung zu einem Typ ohne Null-Zulässigkeit, denn es handelt sich um eine Instanzvariable. Deshalb muss das Suffix `!` gesetzt sein (Z. 44) (Siehe Grundlagenkapitel 3.2.4 *Typen mit Null-Zulässigkeit* auf Seite 49).

```
39 Widget _buildSelectableCell(Massnahme m, Widget child,
40   {double padding = 8.0}) =>
41   TableRowInkWell(
42     onTap: () {
43       if (onSelect != null) {
44         onSelect!(m);
45       }
46     },
47     child: Padding(
48       padding: EdgeInsets.all(padding),
49       child: child,
50     ),
51   );
```

Listing 5.24.: Die Hilfsmethode `_buildSelectableCell`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/massnahmen_table.dart](#)

²¹Vgl. Google LLC, *Dart / Language tour / spread operator*.

Bei `onSelect` handelt es sich um eine Rückruffunktion. An diesem Beispiel kann das *inversion of control pattern* visualisiert werden. Abbildung 5.5 zeigt, wie die Akteure zusammenarbeiten. Der *MassnahmenMasterScreen* verwendet die *MassnahmenTable*. Die Tabelle enthält ein Objekt namens *onSelect*. Dabei handelt es sich um ein Funktionsobjekt. Anstatt eine neue Klasse mit einer beinhaltenden Funktion zu deklarieren, kann das Gleiche über eine Abkürzung erreicht werden: dem Schlüsselwort `typedef` (Listing 5.25).

```
4 typedef OnSelectCallback = void Function(Massnahme selectedMassnahme);
```

Listing 5.25.: Die Typdefinition *OnSelectCallback*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/massnahmen_table.dart](#)

Hier erlaubt es, eine Funktionssignatur als einen Typ zu deklarieren. Der *MassnahmenMasterScreen* wiederum instanziiert ein anonymes Funktionsobjekt, welches der Schnittstelle – und damit der Funktionssignatur – entspricht.

Weil es der Signatur der Typdefinition von `OnSelectCallback` entspricht, kann es der Tabelle als Argument für den Parameter `onSelect` übergeben werden (Listing 5.26, Z. 75-82).

Das *inversion of control pattern* ist auch unter dem Namen *hollywood pattern* bekannt, da es ähnlich wie die typische Antwort auf eine Bewerbung für einen Hollywood Film – don’t call us, we’ll call you – funktioniert.²²

Genauso arbeiten der Übersichtsbildschirm und die Tabelle zusammen. Der Übersichtsbildschirm verwendet die Tabelle, welche nicht wissen muss, wofür sie eingesetzt wird. Sobald die Tabelle eine Selektion des Benutzers bemerkt, kommuniziert sie wieder mit dem Übersichtsbildschirm, worauf dieser über den *Service Locator* auf das *ViewModel* zugreift, um die selektierte Maßnahme zu übergeben.

²²Vgl. Fowler, *InversionOfControl*.

```

70 child: MassnahmenTable(
71     model.storage.value.massnahmen
72     .where((m) =>
73         m.letzteBearbeitung.letzterStatus ==
74         LetzterStatus.bearb.abbreviation)
75     .toSet(), onSelect: (selectedMassnahme) {
76 vm.model = selectedMassnahme.rebuild((m) => m
77     ..letzteBearbeitung.letztesBearbeitungsDatum =
78         DateTime.now().toUtc());
79
80     Navigator.of(context)
81         .pushNamed(MassnahmenDetailScreen.routeName);
82 }),

```

Listing 5.26.: Die Ausgabe der Maßnahmen, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_master.dart](#)

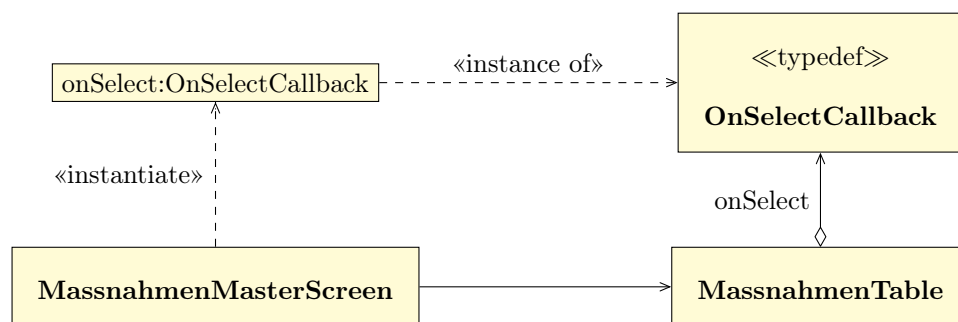


Abbildung 5.5.: UML-Diagramm des *inversion of control pattern* für `MassnahmenMasterScreen` und `MassnahmenTable`, Quelle: Eigene Abbildung

5.9. Das *ViewModel*

Listing 5.27 zeigt das *ViewModel*. Im ersten Schritt enthält es nur drei *Streams* vom Typ `BehaviorSubject`. Eines für den letzten Status (Z. 6), eines für den *guid* (Z. 8) und eines für den Titel der Maßnahme (Z. 10). Anhand dessen wird offensichtlich, warum ein *ViewModel* nötig ist. Die Daten, die in der Oberfläche angezeigt werden, entstammen *Streams*, die neue Werte annehmen können. Wann immer sich ein Wert ändert, löst der *Stream* ein neues Ereignis aus. Auf dieses Ereignis kann der *View* reagieren. Das *Model* bietet die Eigenschaften der Maßnahmen dagegen nicht als *Streams* an.

Da sich *Model* und *ViewModel* in ihrer Struktur unterscheiden, gibt es zwei Methoden, welche die Konvertierung in beide Richtungen vornehmen. Die *Setter*-Methode `model` (Z. 12-18) erhält ein Objekt des Wertetyps `Massnahme`. Die einzelnen Eigenschaften werden dann in das Format des *ViewModels* umgewandelt: in *Streams*. Dafür wird der *Setter*-Methode `value` von jedem `BehaviorSubject` der entsprechende Wert aus dem *Model* zugewiesen. Besonders ist auch, wie die Auswahloptionen sich im *Model* und *ViewModel* unterscheiden. Im *ViewModel* sind es abgeleitete Objekte der Basisklasse *Choice*, wie beispielsweise `LetzterStatus` (Z. 6). Im Gegensatz dazu speichert das *Model* die Auswahloptionen lediglich über die Abkürzung als *String* ab. Mithilfe der Methode `fromAbbreviation` kann anhand der Abkürzung das entsprechende Objekt wiedergefunden werden (Z. 16).

Die *Getter*-Methode `model` (Z. 20-26) dagegen konvertiert in das exakte Gegenteil. Die aktuellen Werte von jedem `BehaviorSubject` werden über die *Getter*-Methode `value` ausgelesen und anschließend der entsprechenden Eigenschaft des Objektes vom Wertetyp `Massnahme` zugewiesen. Die Auswahloption, die für den letzten Status hinterlegt wurde, wird dabei wiederum nur als Abkürzung eingetragen. Dementsprechend ist bloß die Eigenschaft `abbreviation` abzufragen (Z. 22).

Allerdings kann bei Auswahlfeldern auch keine Option gewählt sein. Die *Getter*-Methode `value` kann daher also auch *null* zurückgeben. Der Compiler gibt einen Fehler aus, wenn versucht wird, auf `value` eine Operation auszuführen, sollte es sich um einen Typ mit Null-Zulässigkeit handeln. So ist es bei dem Aufruf von `abbreviation` der Fall (Z. 22). Der Fehler kann nur damit behoben werden, indem das Präfix `?` der Operation vorangestellt wird. In diesem Fall wird die *Getter*-Methode `abbreviation` aufgerufen, sollte `value` nicht *null* sein. Ist `value` dagegen *null*, so wird die Operation nicht ausgeführt und der gesamte Ausdruck gibt direkt *null* zurück.

```

5 class MassnahmenFormViewModel {
6   final letzterStatus = BehaviorSubject<LetzterStatus?>.seeded(null);
7
8   final guid = BehaviorSubject<String?>.seeded(null);
9
10  final massnahmenTitel = BehaviorSubject<String>.seeded("");
11
12  set model(Massnahme model) {
13    guid.value = model.guid;
14
15    letzterStatus.value = letzterStatusChoices
16      .fromAbbreviation(model.letzteBearbeitung.letzterStatus);
17    massnahmenTitel.value = model.identifikatoren.massnahmenTitel;
18  }
19
20  Massnahme get model => Massnahme((b) => b
21    ..guid = guid.value
22    ..letzteBearbeitung.letzterStatus = letzterStatus.value?.abbreviation
23    ..letzteBearbeitung.letztesBearbeitungsDatum = DateTime.now().toUtc()
24    ..identifikatoren
25      .update((b) => b..massnahmenTitel = massnahmenTitel.value));
26 }

```

Listing 5.27.: Die Klasse *MassnahmenFormViewModel*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

5.10. Eingabeformular

Das soeben erstellte *ViewModel* kann nun für die Eingabemaske verwendet werden. Listing 5.28 zeigt die grundlegende Struktur der Klasse `MassnahmenDetailScreen`. Wiederum werden das *ViewModel* und das *Model* über das *InheritedWidget* `AppState` abgerufen und in die jeweiligen lokalen Variablen `vm` und `model` gespeichert (Z. 16, 17).

Nachfolgend werden zwei Funktionen innerhalb der `build`-Methode deklariert: `saveRecord` (Z. 19-28) und `createMassnahmenTitelTextFormField` (Z. 30-44). Solche sogenannten *nested functions* – deutsch verschachtelte Funktionen – sind in *Dart* erlaubt, was zu einer weiteren Besonderheit führt. Der Sichtbarkeitsbereich von Variablen ist in *Dart* lexikalisch. Die Bindung der Variablen ist also durch den umgebenden Quelltext bestimmt. Die lokalen Variablen `model` und `vm` sind also im gesamten Bereich sichtbar, der durch die öffnenden und schließenden geschweiften Klammern der Methode `build` aufgespannt wird (Z. 15-103). Damit sind sie auch innerhalb der beiden verschachtelten Funktionen verfügbar. Innerhalb der Funktionen kann auf `model` und `vm` zugegriffen werden, ohne sie über einen Parameter übergeben zu müssen.

Das erste *Widget* im Inhaltsbereich des `Scaffold`-Elements ist ein `WillPopScope` (Z. 50). Es erlaubt das Verlassen einer Route an eine Abhängigkeit zu knüpfen. Bei dem Eingabeformular handelt es sich um eine Unterseite. Dadurch erscheint in der `AppBar` (Z. 47-48) links von der Überschrift ein Button, der es ermöglicht, zur letzten Ansicht zurück zu navigieren. Dabei stellt sich jedoch die Frage, was mit der bis zu diesem Zeitpunkt eingetragenen Maßnahme passieren soll. Für die Formularanwendung soll in diesem Fall die Maßnahme im aktuellen Zustand abgespeichert werden. Dazu wird dem Parameter `onWillPop` als Argument die Funktion `saveRecord` zugewiesen.

Anders als im Übersichtsbildschirm erhält das `Scaffold` kein Argument für den Parameter `floatingActionButton`. Der Hintergrund dafür ist, dass auf diesem Bildschirm in den nächsten Schritten nicht nur ein, sondern zwei solcher Buttons zur Verfügung stehen sollen. Daher muss der Button manuell angelegt werden. Das ist nur mithilfe eines `Stack`-Widgets möglich (Z. 52). Ein `Stack` erlaubt es, mehrere Ebenen in der Tiefe anzulegen. Das unterste Element soll die Auflistung der Eingabefelder sein. Der `SingleChildScrollView` (Z. 54-79) bietet einen vertikalen Scrollbereich an, in dem die Eingabefelder in einer `Column` (Z. 58-76) untereinander aufgelistet sind. Die Ebene, die über den Eingabefeldern eingeblendet wird, soll die beiden Aktionsbuttons zeichnen. Das *Widget* `Align` erlaubt es, in dieser Ebene festzulegen, wo die Elemente angeordnet sein sollen (Z. 80-99). Wie für den `FloatingActionButton` üblich wurde die untere rechte Bildschirmecke gewählt (Z. 81). Die Buttons sollen in Zukunft übereinander angeordnet sein, weshalb ein `Column`-Widget zum Einsatz kommt (Z. 84-97). Zum ersten Mal taucht der Parameter `mainAxisSize` auf (Z. 85). Mit dem Argument `MainAxisSize.min` nimmt die `Column` in der Höhe nur so viel Platz ein, wie durch die Kindelemente notwendig ist.


```

7  const saveMassnahmeTooltip = "Validiere und speichere Massnahme";
8
9  class MassnahmenDetailScreen extends StatelessWidget {
10   static const routeName = '/massnahmen-detail';
11
12   const MassnahmenDetailScreen({Key? key}) : super(key: key);
13
14   @override
15   Widget build(BuildContext context) {
16     final vm = AppState.of(context).viewModel;
17     final model = AppState.of(context).model;
18
19     Future<bool> saveRecord() {
20       /* ... */
21     }
22
23     Widget createMassnahmenTitelTextFormField() {
24       /* ... */
25     }
26
27     return Scaffold(
28       appBar: AppBar(
29         title: const Text('Maßnahmen Detail'),
30       ),
31       body: WillPopScope(
32         onWillPop: () => saveRecord(),
33         child: Stack(
34           children: [
35             SingleChildScrollView(
36               child: Center(
37                 child: Padding(
38                   padding: const EdgeInsets.all(8.0),
39                   child: Column(
40                     /* ... */
41                   ),
42                 ),
43             ),
44             Align(
45               alignment: Alignment.bottomRight,
46               child: Padding(
47                 padding: const EdgeInsets.all(16.0),
48                 child: Column(
49                   mainAxisAlignment: MainAxisAlignment.min,
50                   children: [
51                     FloatingActionButton(
52                       tooltip: saveMassnahmeTooltip,
53                       heroTag: 'save_floating_action_button',
54                       child: const Icon(Icons.check, color: Colors.white),
55                       onPressed: () {
56                         saveRecord();
57                         Navigator.of(context).pop();
58                       },
59                     ),
60                   ],
61                 ),
62             ),
63           ],
64         ),
65       ),
66     );
67   }
68 }

```

Listing 5.28.: Die Struktur des Bildschirms *MassnahmenDetailScreen*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Als bisher einziges Element in der `Column` taucht nun der `FloatingActionButton` auf (Z. 87-95), der die aktuell eingetragenen Daten abspeichern (Z. 92) und zur Übersicht zurückkehren soll (Z. 93). Wenn der Nutzer den Mauszeiger über diesen Button bewegt, wird ein Tooltip angezeigt: "Validiere und speichere Massnahme" (Z. 88). Der Tooltip ist als Konstante angelegt (Z. 7). Das hat vor allem den Grund, dass er auch für den folgenden Integrationstest genutzt wird. Elemente können darin über einen beinhaltenden Text oder Tooltip gefunden werden.

5.10.1. Ausgabe der Formularfelder

Listing 5.29 zeigt die Ausgabe der Formularfelder in einer `Column` (Z. 58). Das Auswahlfeld für den letzten Status verwendet ein selbst geschriebenes `Widget` namens `SelectionCard` (Z. 61-72). Da die Menge der Auswahloptionen auch den Namen der Liste enthält, kann er als Titel der Selektionskarte verwendet werden (Z. 62). In diesem Fall ist das der Text *Status*. Die Auswahloptionen, welche der Selektionsbildschirm anzeigen soll, sind dem Parameter `allChoices` hinterlegt (Z. 63).

Die Selektionskarte soll ihren eigenen Zustand pflegen. Sie erhält dazu lediglich den initialen Wert, der aktuell im `ViewModel` gespeichert ist. Bei allen Änderungen, die innerhalb der Selektionskarte erfolgen, sollen die gleichen Änderungen auch im `ViewModel` nachgepflegt werden. Sollte also der Wert des letzten Status im `ViewModel` verfügbar sein (Z. 65), so wird er als Startwert dem Parameter `initialValue` (Z. 64-67) übergeben. Dabei ist zu beachten, dass das Argument eine Menge ist. Sie wird mit den öffnenden und schließenden geschweiften Klammern erstellt. Das *collection if* wird hier verwendet, um genau ein Element diesem *Set*-Literal hinzuzufügen, sollte es nicht `null` sein (Z. 65). Ist das Element allerdings `null`, so bleibt das *Set*-Literal einfach leer.

Wenn der Benutzer eine Auswahloption selektiert, so wird die dementsprechende anonyme Funktion aufgerufen. Sie ist für den Parameter `onSelect` hinterlegt (Z. 68-69). Das Gleiche gilt für Auswahloptionen, welche deselektiert werden (Z. 70-71). Das Auswahlfeld erlaubt nur einen Wert. Deshalb reicht es aus, den Wert bei Selektion zu ersetzen und ihn bei Deselektion zu leeren, also ihn auf `null` zu setzen (Z. 71).

```

58 child: Column(
59   crossAxisAlignment: CrossAxisAlignment.start,
60   children: [
61     SelectionCard<LetzterStatus>(
62       title: letzterStatusChoices.name,
63       allChoices: letzterStatusChoices,
64       initialValue: {
65         if (vm.letzterStatus.value != null)
66           vm.letzterStatus.value!
67       },
68       onSelect: (selectedChoice) =>
69         vm.letzterStatus.value = selectedChoice,
70       onDeselect: (selectedChoice) =>
71         vm.letzterStatus.value = null,
72     ),
73     createMassnahmenTitelTextFormField(),
74     const SizedBox(height: 64)
75   ],
76 ),

```

Listing 5.29.: Die Ausgabe der Formularfelder, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

5.10.2. Eingabefeld für den Maßnahmentitel

Unterhalb der ersten Selektionskarte soll das Eingabefeld für den Maßnahmentitel erscheinen (Z. 73). Listing 5.30 zeigt die Implementierung der verschachtelten Funktion zum Zeichnen dieses Eingabefeldes. Es handelt sich um das *Widget* `TextFormField` (Z. 34-41).

```

30 Widget createMassnahmenTitelTextFormField() {
31   return Card(
32     child: Padding(
33       padding: const EdgeInsets.all(16.0),
34       child: TextFormField(
35         initialValue: vm.massnahmenTitel.value,
36         decoration: const InputDecoration(
37           hintText: 'Maßnahmentitel', labelText: 'Maßnahmentitel'),
38         onChanged: (value) {
39           vm.massnahmenTitel.value = value;
40         },
41       ),
42     ),
43   );
44 }

```

Listing 5.30.: Die Funktion `createMassnahmenTitelTextFormField` in Schritt 1, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Hier wird klar, wovon die Selektionskarte inspiriert ist. Denn auch das `TextFormField` erhält einen initialen Wert über den Parameter `initialValue` (Z. 35). Sobald sich der Wert des Formularfelds ändert, kann der neue Wert im *ViewModel* über die anonyme Funktion aktualisiert werden, welche dem Parameter `onChanged` übergeben wurde (Z. 38-40).

5.10.3. Speicher-Routine

Die Funktion, die dem Parameter `onWillPop` des `WillPopScope` übergeben wurde, ist in Listing 5.31 zu sehen. Die Voraussetzung für diese Funktion ist, dass ihr Rückgabebetyp ein `Future<bool>` ist. Das erlaubt der Methode, asynchron zu sein. Der `Future`, der von der Funktion zurückgegeben werden soll, muss in der Zukunft den Wert `true` zurückgeben, wenn dem *Navigator* erlaubt werden soll, zurück zu navigieren. Da die Implementierung der Methode allerdings nicht asynchron ist, soll der Wahrheitswert direkt zurückgegeben werden. Mit dem benannten Konstruktor `value` der Klasse `Future` ist es möglich, genau das zu tun (Z. 27). Der Wahrheitswert ist damit in einem `Future`-Objekt gekapselt und steht ohne Verzögerung zur Verfügung. Aktuell soll die Maßnahme lediglich abgespeichert werden (Z. 25), da noch keine Validierung erfolgt.

Der Benutzer erhält noch eine Mitteilung, dass die Maßnahme erstellt wurde. Das aktuelle `Scaffold`-Objekt kann über `ScaffoldMessenger.of` adressiert werden (Z. 20). Sollte bereits eine Mitteilung vorliegen, wird diese wieder versteckt, um Platz für die neue zu schaffen (Z. 21). Anschließend wird eine sogenannte `SnackBar` mit dem entsprechenden Text angezeigt (Z. 22-23).

```

19 Future<bool> saveRecord() {
20   ScaffoldMessenger.of(context)
21     ..hideCurrentSnackBar()
22     ..showSnackBar(
23       const SnackBar(content: Text('Massnahme wird gespeichert ...')));
24
25   model.putMassnahmeIfAbsent(vm.model);
26
27   return Future.value(true);
28 }

```

Listing 5.31.: Die Funktion `saveRecordAndGoBackToOverviewScreen`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

5.11. Das *SelectionCard*-Widget

Das Listing 5.32 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 Instanzvariablen werden über diesen Typparameter generalisiert (Z. 17-20), mit Ausnahme der Instanzvariablen `title`, denn sie erhält kein Typargument (Z. 16).

```

7  typedef OnSelect<ChoiceType extends Choice> = void Function(
8      ChoiceType selectedChoice);
9
10 typedef OnDeselect<ChoiceType extends Choice> = void Function(
11     ChoiceType selectedChoice);
12
13 const confirmButtonTooltip = 'Auswahl übernehmen';
14
15 class SelectionCard<ChoiceType extends Choice> extends StatelessWidget {
16     final String title;
17     final BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel;
18     final Choices<ChoiceType> allChoices;
19     final OnSelect<ChoiceType> onSelect;
20     final OnDeselect<ChoiceType> onDeselect;
21
22     SelectionCard(
23         {required this.title,
24         required Iterable<ChoiceType> initialValue,
25         required this.allChoices,
26         required this.onSelect,
27         required this.onDeselect,
28         Key? key})
29         : selectionViewModel = BehaviorSubject<BuiltSet<ChoiceType>>.seeded(
30             BuiltSet.from(initialValue)),
31           super(key: key);

```

Listing 5.32.: Die Klasse SelectionCard, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/selection_card.dart](#)

Mit dem *Stream* `selectionViewModel` verwaltet die `SelectionCard` ihren eigenen Zustand. Der *Stream* ist mit dem generischen Typ `BuiltSet<ChoiceType>` konfiguriert (Z. 17). Das macht es unmöglich, den aktuell hinterlegten Wert anzupassen, ohne das Gesamtobjekt auszutauschen. Der Tausch des Objektes 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 (Z. 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 Typ konfiguriert sind. Die Signaturen der Rückruffunktionen (Z. 7-8, 10-11) geben nämlich

vor, dass der erste Parameter vom Typ `ChoiceType` sein muss. Wenn nun der Benutzer der `SelectionCard` einen Typ wie etwa `LetzterStatus` für den Typparameter übergibt, so erhält er auch die Rückruffunktionen `onSelect` und `onDeselect`, deren erster Parameter vom Typ `LetzterStatus` ist. Ohne eine Typumwandlung – englisch *type casting* – von `Choice` in `LetzterStatus` können so Operationen auf dem Objekt angewendet werden, die nur die Klasse `LetzterStatus` unterstützt.

Das erste *Widget*, welches von der `build`-Methode zurückgegeben wird, ist ein `StreamBuilder` (Listing 5.33, 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` erzielt (Z. 51). Dadurch erhält es abgerundete Ecken und einen Schlagschatten, der es vom Hintergrund abgrenzt. Ein `ListTile-Widget` erlaubt es dann, den übergebenen `title` 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 `c` lediglich den Beschreibungstext `description` verwendet. Anschließend sammelt der Befehl `join` die resultierenden `String`-Objekte ein, formt sie in einem gemeinsamen `String` zusammen und trennt sie darin jeweils mit einem `","` voneinander.

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 im Formular wieder an der obersten Position ankommen. 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). Sie wird dem Parameter `onTap` des `ListTile-Widgets` übergeben (Z. 58). 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 (Z. 40). Es handelt sich um den Selektionsbildschirm, auf dem der Benutzer die gewünschte Option anwählen kann. Die Besonderheit dieses Mal ist: 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 5.34 erklärt wird.

```

34 Widget build(BuildContext context) {
35   final focusNode = FocusNode();
36
37   navigateToSelectionScreen() async {
38     focusNode.requestFocus();
39
40     Navigator.push(
41       context,
42       MaterialPageRoute(
43         builder: (context) =>
44           createMultipleChoiceSelectionScreen(context)));
45   }
46
47   return StreamBuilder(
48     stream: selectionViewModel,
49     builder: (context, snapshot) {
50       final selectedChoices = selectionViewModel.value;
51       return Card(
52         child: ListTile(
53           focusNode: focusNode,
54           title: Text(title),
55           subtitle:
56             Text(selectedChoices.map((c) => c.description).join(", ")),
57           trailing: const Icon(Icons.edit),
58           onTap: navigateToSelectionScreen,
59         ),
60       );
61     });
62 }

```

Listing 5.33.: Die *build*-Methode der Klasse *SelectionCard* in Schritt 1, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/lib/widgets/selection_card.dart](#)

5.11.1. Bildschirm für die Auswahl der Optionen

Die Methode `createMultipleChoiceSelectionScreen` (Listing 5.34, S. 105) 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 Selektionsbildschirm 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 Selektionsbildschirm 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 `allChoices` 74 in Elemente des Typs `CheckboxListTile` (Z. 77).

In der Standard-Variante sind die Checkboxes 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*-Objektes `c` gebildet wird (Z. 81). Ob eine Option aktuell bereits ausgewählt ist, kann mit dem Parameter `value` gekennzeichnet 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 (Z. 84), denn sein Parametertyp `bool?` lässt Null-Werte zu (Z. 83). Durch die Typ-Beförderung ist `selected` innerhalb des Körpers der Fallunterscheidung dann vom Typ `bool` ohne Null-Zulässigkeit (Z. 85-93).

Darin wird zunächst der Zustand des *ViewModels* der *SelectionCard* aktualisiert (Z. 85-88). Die `replace`-Methode des *Builder*-Objektes kann die gesamte Kollektion im *BuiltSet* austauschen (Z. 87), ungeachtet dessen, dass es sich beim Argument selbst nicht um ein *BuiltSet* handelt. Die `replace`-Methode wandelt das Argument dafür automatisch um. Wenn die Auswahloption ausgewählt war, wird sie beim Anklicken abgewählt, indem eine leere Liste `[]` als neuer Wert gesetzt wird. Wurde die Auswahloption aber angewählt, so wird eine Liste mit der Option als einziges Element übergeben: `[c]`. Durch Zuweisung des neuen Wertes erhält das *ViewModel* der *SelectionCard* ein neues Ereignis. Damit werden die *SelectionCard* und der dazugehörige Selektionsbildschirm aktualisiert. 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 Selektionsbildschirm aktualisiert werden. Doch bei diesem Vorgehen zeichnet sich nur die *SelectionCard* neu. Der Selektionsbildschirm bleibt unverändert, denn er wird zwar von der *SelectionCard* gebaut, doch ist er nicht tatsächlich Kind der *SelectionCard*. In Wahrheit ist der Selektionsbildschirm 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).


```

64 Widget createMultipleChoiceSelectionScreen(BuildContext context) {
65   return Scaffold(
66     appBar: AppBar(
67       title: Text(title),
68     ),
69     body: StreamBuilder(
70       stream: selectionViewModel,
71       builder: (context, snapshot) {
72         final selectedChoices = selectionViewModel.value;
73         return ListView(children: [
74           ...allChoices.map((ChoiceType c) {
75             bool isSelected = selectedChoices.contains(c);
76
77             return CheckboxListTile(
78               key: Key(
79                 "valid choice ${allChoices.name} - ${c.abbreviation}"),
80               controlAffinity: ListTileControlAffinity.leading,
81               title: Text(c.description),
82               value: isSelected,
83               onChanged: (bool? selected) {
84                 if (selected != null) {
85                   selectionViewModel.value =
86                     selectionViewModel.value.rebuild((b) {
87                       b.replace(isSelected ? [] : [c]);
88                     });
89                 if (selected) {
90                   onSelect(c);
91                 } else {
92                   onDeselect(c);
93                 }
94               }
95             });
96           }).toList(),
97         ]);
98     },
99     floatingActionButton: FloatingActionButton(
100       onPressed: () => Navigator.of(context).pop(),
101       tooltip: confirmButtonTooltip,
102       child: const Icon(Icons.check),
103     ),
104   );
105 }
106 }

```

Listing 5.34.: Die Methode `createMultipleChoiceSelectionScreen`, Quelle: Eigenes Listing, Datei: [Quellocode/Schritt-1/conditional_form/lib/widgets/selection_card.dart](#)

5.12. Integrationstest 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 Integrationstests 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 gegebenenfalls 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 keine Daten auf der Festplatte speichern, denn das würde die Gefahr bergen, dass vergangene Eingaben manipuliert werden. Stattdessen soll der Test in einer Umgebung stattfinden, die keine Auswirkungen auf die Hauptapplikation 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 Codegenerierung zu erstellen.

5.12.1. Generierung des *Mocks*

Integrationstests werden im Ordner *integration_test* angelegt. Während des Zeitpunkts der Erstellung dieser Arbeit war es in der Standardkonfiguration der Codegenerierung und dem Paket *mockito* nicht möglich, *Mocks* auch im *integration_test*-Ordner zu generieren. Lediglich innerhalb des *test*-Ordners, der für die Unittests vorgesehen ist, hat die Annotation *@GenerateMocks* funktioniert. Zu diesem Fehlverhalten existiert ein entsprechendes *Issue* im *GitHub* Repository des *mockito* Pakets.²³ Um das Generieren von *Mocks* auch für Integrationstests verfügbar zu machen, hat der Autor dieser Arbeit einen entsprechenden Lösungsansatz recherchiert und im *Issue* beschrieben.²⁴

Damit der *integration_test* Ordner für die Codegenerierung 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 5.35, Z. 3-8). Wird der Ordner *integration_test* darin eingefügt (Z. 8), bezieht *source_gen* den Ordner in die Codegenerierung mit ein. Zusätzlich

²³Vgl. GitHub-Nutzer nt4f04uNd, *GitHub* / *dart-lang* / *mockito* / *Mocks are not generated not in test folder*.

²⁴Vgl. Jahr, *GitHub* / *dart-lang* / *mockito* / *Antwort auf “Mocks are not generated not in test folder”*.

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).

```

1 targets:
2   $default:
3     sources:
4       - $package$
5       - lib/$lib$
6       - lib/**/*.dart
7       - test/**/*.dart
8       - integration_test/**/*.dart
9     builders:
10      mockito|mockBuilder:
11        generate_for:
12          - test/**/*.dart
13          - integration_test/**/*.dart

```

Listing 5.35.: Initialisierung des Integrationstests, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/build.yaml](#)

Anschließend kann mit der Annotation `@GenerateMocks` (Listing 5.36, Z. 20) ein *Mock* für `MassnahmenJsonFile` angefordert werden. In der Kommandozeile ist – genau wie bei der Generierung der Wertetypen – der Befehl `flutter pub run build_runner build` 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.

```

18 const durationAfterEachStep = Duration(milliseconds: 1);
19
20 @GenerateMocks([MassnahmenJsonFile])
21 void main() {
22   testWidgets('Can fill the form and save the correct json', (tester) async {
23     final binding = IntegrationTestWidgetsFlutterBinding.ensureInitialized()
24       as IntegrationTestWidgetsFlutterBinding;
25     binding.framePolicy = LiveTestWidgetsFlutterBindingFramePolicy.fullyLive;
26
27     final massnahmenJsonFileMock = MockMassnahmenJsonFile();
28     when(massnahmenJsonFileMock.readMassnahmen()).thenAnswer((_) async => {});

```

Listing 5.36.: Initialisierung des Integrationstests, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

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 23 bis 25 wird der Testtreiber initialisiert. Anschließend wird ein Objekt der generierten Klasse `MockMassnahmenJsonFile` erstellt (Z. 27). 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 mithilfe 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 5.37). Doch der Konstruktor von `MassnahmenModel` erhält dieses Mal nicht das `MassnahmenJsonFile`, sondern den entsprechenden *Mock* (Z. 31).

```

30 await tester.pumpWidget(AppState(
31   model: MassnahmenModel(massnahmenJsonFileMock),
32   viewModel: MassnahmenFormViewModel(),
33   child: MaterialApp(
34     title: 'Maßnahmen',
35     theme: ThemeData(
36       primarySwatch: Colors.lightGreen,
37       accentColor: Colors.green,
38       primaryIconTheme: const IconThemeData(color: Colors.white),
39     ),
40     initialRoute: MassnahmenMasterScreen.routeName,
41     routes: {
42       MassnahmenMasterScreen.routeName: (context) =>
43         const MassnahmenMasterScreen(),
44       MassnahmenDetailScreen.routeName: (context) =>
45         const MassnahmenDetailScreen()
46     },
47   ));

```

Listing 5.37.: Initialisierung des *Widgets* für den Integrationstest, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Weil während des Integrationstests immer wieder die gleichen Operationen wie das Selektieren einer Selektionskarte, das Auswählen einer Option, das Anklicken des Buttons zum Akzeptieren der Auswahl und das Füllen eines Eingabefeldes auftauchen, wurden entsprechende Hilfsfunktionen erstellt.

5.12.2. Hilfsfunktionen für den Integrationstest

Die Funktion `tabSelectionCard` (Listing 5.38) benötigt lediglich die Liste der Auswahloptionen `choices`, die ihr hinterlegt ist.

```

49 Future<void> tabSelectionCard(Choices choices) async {
50   final Finder textLabel = find.text(choices.name);
51   expect(textLabel, findsWidgets);
52
53   final card = find.ancestor(of: textLabel, matching: find.byType(Card));
54   expect(card, findsOneWidget);
55
56   await tester.ensureVisible(card);
57   await tester.tap(card);
58   await tester.pumpAndSettle(durationAfterEachStep);
59 }

```

Listing 5.38.: Die Hilfsmethode `tabSelectionCard`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Um Objekte während des Testens in der Oberfläche zu finden, stellt die Klasse `Finder` nützliche Funktionalitäten zur Verfügung. `Finder`-Objekte können über Fabrikmethoden des Objektes `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.²⁵ Er arbeitet lediglich mit der Schnittstelle. In diesem Fall ist `find` dieser Erzeuger. Über die Fabrikmethode `text` wird ein `_TextFinder` konstruiert. Eine weitere Fabrikmethode ist `ancestor`. Sie gibt einen `_AncestorFinder` zurück. Die Fabrikmethoden werden hier deshalb verwendet, weil sie die Lesbarkeit verbessern, denn `Finder titel = find.text("Maßnahmentitel")` ist deutlich leichter zu erfassen als `Finder titel = new _TextFinder("Maßnahmentitel")`.

Um die Selektionskarten 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(choices.name)` lokalisiert den Titel innerhalb der Selektionskarte (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 der Vater gesucht werden, der vom Typ `Card` ist (Z. 53). Das kann mit `find.ancestor` erfolgen. Über den Parameter `of` erhält er den `Finder` des Kindelements und der Parameter `matching` erhält als Argument die Voraussetzung, die vom Vaterobjekt erfüllt werden soll, als weiteren `Finder`. `find.byType(Card)` sucht also alle Elemente vom Typ `Card`. `find.ancestor` sucht anschließend alle Entsprechungen, in denen 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 angeklickt 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

²⁵Vgl. Gamma u. a., *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, S. 107–116.

der Oberfläche und damit auch alle angestoßenen Animationen vorüber sind. Zusätzlich kann eine Dauer angegeben werden, die darüber hinaus gewartet werden soll.

Die Hilfsmethode `tabConfirmButton` funktioniert ähnlich (Listing 5.39). 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 auf den Button (Z. 63) und wartet dann erneut auf Vollendung aller angestoßenen Animationen (Z. 64).

```

61 Future<void> tabConfirmButton() async {
62   var confirmChoiceButton = find.byTooltip(confirmButtonTooltip);
63   await tester.tap(confirmChoiceButton);
64   await tester.pumpAndSettle(durationAfterEachStep);
65 }

```

Listing 5.39.: Die Hilfsmethode `tabConfirmButton`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Ist der Integrationstest aktuell in dem Selektionsbildschirm, so sorgt `tabOption` dafür, dass eine Auswahloption gewählt wird (Listing 5.40). 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).

```

67 Future<void> tabOption(Choice choice, {bool tabConfirm = false}) async {
68   final choiceLabel = find.text(choice.description);
69   expect(choiceLabel, findsOneWidget);
70
71   await tester.ensureVisible(choiceLabel);
72   await tester.tap(choiceLabel);
73   await tester.pumpAndSettle(durationAfterEachStep);
74
75   if (tabConfirm) {
76     await tabConfirmButton();
77   }
78 }

```

Listing 5.40.: Die Hilfsmethode `tabOption`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

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 (Z. 71), klickt es an (Z. 72) und wartet auf Abschluss aller Animationen (Z. 73). Sollte der optionale Parameter `tabConfirm` auf `true` gesetzt sein (Z. 75), so wird der Selektionsbildschirm anschließend direkt wieder geschlossen, nachdem die Option ausgewählt wurde (Z. 76).

Schließlich kann mit der Hilfsfunktion `fillTextFormField` ein Formularfeld über dessen Titel gefunden und der übergebene Text eingetragen werden (Listing 5.41). Die Hilfsfunktion findet das `TextFormField`, indem sie zunächst nach dem Titel mit `find.text(title)` und anschließend nach dessen Vaterelement vom Typ `TextFormField` sucht (Z. 83). Sollte sowohl

der Hinweistext als auch der Titel den gleichen Text enthalten, so kann es sein, dass zwei solche Elemente gefunden werden. In Wahrheit ist es aber zweimal dasselbe `TextFormField`. Mit `.first` wird lediglich das erste Element geliefert (Z. 84). 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).

```

80 Future<void> fillTextFormField(
81     {required String title, required String text}) async {
82     final textFormField = find
83         .ancestor(of: find.text(title), matching: find.byType(TextFormField))
84         .first;
85     expect(textFormField, findsOneWidget);
86
87     await tester.ensureVisible(textFormField);
88     await tester.enterText(textFormField, text);
89     await tester.pumpAndSettle(durationAfterEachStep);
90 }

```

Listing 5.41.: Die Hilfsmethode *fillTextFormField*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

5.12.3. Test des Übersichtsbildschirms

Während der Integrationstest startet, öffnet sich als Erstes der Übersichtsbildschirm. Zunächst wird gewartet, dass alle *Widgets* korrekt initialisiert wurden (Listing 5.42, Z. 92). Es folgt der Klick auf den Button zum Erstellen einer 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.

```

92 await tester.pumpAndSettle(durationAfterEachStep);
93
94 var createNewMassnahmeButton = find.byKey(createNewMassnahmeButtonKey);
95 await tester.tap(createNewMassnahmeButton);
96 await tester.pumpAndSettle(durationAfterEachStep);

```

Listing 5.42.: Der Button zum Kreieren einer Maßnahme wird ausgelöst, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

5.12.4. Test der Eingabemaske

Der Integrationstest öffnet nun die Eingabemaske, in der die Selektionskarte zum Setzen des letzten Status angewählt wird (Listing 5.43, Z. 98). Anschließend fällt die Wahl auf die Option für *abgeschlossen* (Z. 99). Dabei sorgt `tabConfirm: true` für die sofortige Rückkehr zum Eingabeformular nach der Auswahl.

```
98 await tabSelectionCard(letzterStatusChoices);
99 await tabOption(LetzterStatus.fertig, tabConfirm: true);
```

Listing 5.43.: Der letzte Status wird ausgewählt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Nachfolgend soll der Test das Eingabefeld für den Maßnahmentitel überprüfen (Listing 5.44). Es erfolgt die Erstellung eines beispielhaften Titels anhand des aktuellen Datums und der aktuellen Uhrzeit (Z. 101-103). Der erstellte Text dient als Eingabe für das Eingabefeld (Z. 104).

```
101 final now = DateTime.now();
102 var massnahmeTitle =
103     "Test Maßnahmen ${now.year}-${now.month}-${now.day} ${now.hour}:${now.minute}";
104 await fillTextFormField(title: "Maßnahmentitel", text: massnahmeTitle);
```

Listing 5.44.: Der Maßnahmentitel wird eingegeben, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

5.12.5. Test der Speicherung

Die nötigen Eingaben sind erfolgt. Daher kann der Test nun den Klick auf den Button zum Speichern simulieren (Listing 5.45, Z. 106-108). Dadurch würde in der Anwendung nun das Speichern der Maßnahmen in der *JSON*-Datei erfolgen. 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 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


```

106 var saveMassnahmeButton = find.byTooltip(saveMassnahmeTooltip);
107 await tester.tap(saveMassnahmeButton);
108 await tester.pumpAndSettle(durationAfterEachStep);
109
110 var capturedJson =
111     verify(massnahmenJsonFileMock.saveMassnahmen(captureAny)).captured.last;
112
113 var actualMassnahme = capturedJson['massnahmen'][0] as Map;
114 actualMassnahme.remove("guid");
115 actualMassnahme["letzteBearbeitung"].remove("letztesBearbeitungsDatum");
116
117 var expectedJson = {
118     'letzteBearbeitung': {'letzterStatus': 'fertig'},
119     'identifikatoren': {'massnahmenTitel': massnahmeTitle},
120 };
121
122 expect(actualMassnahme, equals(expectedJson));

```

Listing 5.45.: Validierung des Testergebnisses, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart](#)

Maßnahme verglichen werden. Der Schlüssel `'massnahmen'` greift auf die Liste zurück und der Schlüssel `0` auf die erste und einzige Maßnahme. Die lokale Variable `actualMassnahme` speichert sie zwischen (Z. 113).

Es ist unklar, welcher 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 des *guid* und des Datums überwachen. Anstatt eines zufälligen *guid* könnte beispielsweise immer die gleiche Zeichenkette zurückgegeben werden. Es ist jedoch auch möglich, die Vergleiche des *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. Am Ende erfolgt in dem Terminal die Ausgabe des Ergebnisses: `All tests passed!`

6. Schritt 2 - Refaktorisierung zum Hinzufügen weiterer Eingabefelder

In diesem Schritt sollen weitere Selektionskarten für die Einzelauswahlfelder hinzugefügt werden. Darüber hinaus soll das Erstellen der Selektionskarten in einer Methode abstrahiert werden. Das ermöglicht die Konfiguration der Selektionskarten in der aufrufenden Eingabemaske, ohne dafür die Klasse `SelectionCard` ändern zu müssen.

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. 6.1).

The screenshot shows a mobile application interface for 'Maßnahmen Detail'. At the top is a green header bar with a back arrow and the title 'Maßnahmen Detail'. A red 'DEBUG' label is in the top right corner. Below the header, the section 'Maßnahmencharakteristika' is displayed. It contains five selection cards, each with a title, a description, and an edit icon (pencil):

- Förderklasse**: Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten, aber OHNE Vertragsnaturschutz
- Kategorie**: Extensivierung
- Zielfläche**: AL
- Zieleinheit**: ha
- Hauptzielsetzung Land**: Biodiversität

The 'Hauptzielsetzung Land' card has a green circular button with a white checkmark in its bottom right corner, indicating it is the selected option.

Abbildung 6.1.: Die Eingabemaske in Schritt 2, Quelle: Eigene Abbildung

In den Tabellen im Übersichtsbildschirm erscheinen die Werte rechts vom Maßnahmentitel (Abb. 6.2).

Maßnahmen Master						
Abgeschlossen						
Zuletzt bearbeitet am	Maßnahmentitel	Förderklasse	Kategorie	Zielfläche	Zieleinheit	Hauptzielsetzung Land
2021-8-4 13:15	Maßnahme 1	aukm_ohne_vns	extens	al	ha	biodiv
In Bearbeitung						
Zuletzt bearbeitet am	Maßnahmentitel	Förderklasse	Kategorie	Zielfläche	Zieleinheit	Hauptzielsetzung Land
2021-8-4 13:17	Maßnahme 2					

Abbildung 6.2.: Der Übersichtsbildschirm in Schritt 2, Quelle: Eigene Abbildung

6.1. Integrationstest erweitern

Noch vor der Implementierung der Änderungen soll zunächst der Integrationstest um die zusätzlichen Selektionen erweitert werden (Listing 6.1). Nach den letzten Eingaben und bevor der Button zum Speichern ausgelöst wird, erfolgt die Selektion der fünf Optionen (Z. 106-119).

```

106 await tabSelectionCard(foerderklasseChoices);
107 await tabOption(FoerderklasseChoice.aukm_ohne_vns, tabConfirm: true);
108
109 await tabSelectionCard(kategorieChoices);
110 await tabOption(KategorieChoice.extens, tabConfirm: true);
111
112 await tabSelectionCard(zielflaecheChoices);
113 await tabOption(ZielflaecheChoice.al, tabConfirm: true);
114
115 await tabSelectionCard(zieleinheitChoices);
116 await tabOption(ZieleinheitChoice.ha, tabConfirm: true);
117
118 await tabSelectionCard(hauptzielsetzungLandChoices);
119 await tabOption(ZielsetzungLandChoice.biodiv, tabConfirm: true);
120
121 var saveMassnahmeButton = find.byTooltip(saveMassnahmeTooltip);
122 await tester.tap(saveMassnahmeButton);
123 await tester.pumpAndSettle(durationAfterEachStep);

```

Listing 6.1.: Der Integrationstest klickt 5 weitere Karten, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/integration_test/app_test.dart](#)

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 6.2, Z. 135-141).

```

132 var expectedJson = {
133   'letzteBearbeitung': {'letzterStatus': 'fertig'},
134   'identifikatoren': {'massnahmenTitel': massnahmeTitle},
135   'massnahmenCharakteristika': {
136     'foerderklasse': 'aukm_ohne_vns',
137     'kategorie': 'extens',
138     'zielflaeche': 'al',
139     'zieleinheit': 'ha',
140     'hauptzielsetzungLand': 'biodiv'
141   },
142 };

```

Listing 6.2.: Der Integrationstest überprüft im *JSON*-Dokument den Schlüssel *massnahmenCharakteristika*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/integration_test/app_test.dart](#)

Der Integrationstest ist damit aktualisiert. Die Implementierung ist jedoch noch gar nicht erfolgt. Die Selektionskarten können nicht angeklickt werden, da sie in der Oberfläche noch nicht auftauchen. Die neuen Schlüssel-Werte-Paare können nicht in der Hashtabelle 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 Beck¹

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.

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

6.2. Hinzufügen der Auswahloptionen

Der Integrationstest selektiert unter anderem die Förderklasse mit der Abkürzung *“aukm_ohne_vns”*. Sie wird den Auswahloptionen hinzugefügt, wie in Listing 6.3 zu sehen ist. Die Liste aller hinzugefügten Auswahloptionen in diesem Schritt ist im Anhang D auf den Seiten 193 bis 196 zu finden.

```
11 static final aukm_ohne_vns = FoerderklasseChoice("aukm_ohne_vns",
12     "Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten, aber OHNE
    ↳ Vertragsnaturschutz");
```

Listing 6.3.: Die Klassenvariable *aukm_ohne_vns* vom Typ *FoerderklasseChoice*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/choices/choices.dart](#)

6.3. Aktualisierung des *Models*

Damit der *JSON*-Datei 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 6.4, Z. 15).

```
13 Identifikatoren get identifikatoren;
14
15 MassnahmenCharakteristika get massnahmenCharakteristika;
```

Listing 6.4.: *massnahmenCharakteristika* wird dem Wertetyp *Massnahme* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/data_model/massnahme.dart](#)

Bei dem Datentyp handelt es sich um einen weiteren Wertetyp: *MassnahmenCharakteristika*, welcher in Listing 6.5 zu sehen ist. Die darin enthaltenen *Getter*-Methoden sind dagegen lediglich gewöhnliche Zeichenketten, da sie die Abkürzungen der ausgewählten Optionen abspeichern. Da sie im Entwurfsmodus nicht gefüllt sein müssen, wird ihnen mit dem Suffix *?* erlaubt, Null-Werte anzunehmen (Z. 70-74).

```
67 abstract class MassnahmenCharakteristika
68     implements
69         Built<MassnahmenCharakteristika, MassnahmenCharakteristikaBuilder> {
70   String? get foerderklasse;
71   String? get kategorie;
72   String? get zielflaeche;
73   String? get zieleinheit;
74   String? get hauptzielsetzungLand;
```

Listing 6.5.: Der Wertetyp *Massnahmencharakteristika*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/data_model/massnahme.dart](#)

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

6.4. 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 6.6, Z. 23-27).

```

22 _buildColumnHeader(const Text("Maßnahmentitel")),
23 _buildColumnHeader(const Text("Förderklasse")),
24 _buildColumnHeader(const Text("Kategorie")),
25 _buildColumnHeader(const Text("Zielfläche")),
26 _buildColumnHeader(const Text("Zieleinheit")),
27 _buildColumnHeader(const Text("Hauptzielsetzung Land")),

```

Listing 6.6.: Die *Maßnahmencharakteristika* werden dem Tabellenkopf hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/widgets/massnahmen_table.dart](#)

Für jede Zeile der Tabelle werden weitere selektierbare Zellen generiert (Listing 6.7, Z. 33-42). Im Unterschied zur Zelle des Maßnahmentitels können die *Getter*-Methoden des Wertetyps `massnahmenCharakteristika` 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 der leere *String* `""` als Argument übergeben.

```

32 _buildSelectableCell(m, Text(m.identifikatoren.massnahmenTitel)),
33 _buildSelectableCell(
34   m, Text(m.massnahmenCharakteristika.foerderklasse ?? "")),
35 _buildSelectableCell(
36   m, Text(m.massnahmenCharakteristika.kategorie ?? "")),
37 _buildSelectableCell(
38   m, Text(m.massnahmenCharakteristika.zielflaeche ?? "")),
39 _buildSelectableCell(
40   m, Text(m.massnahmenCharakteristika.zieleinheit ?? "")),
41 _buildSelectableCell(m,
42   Text(m.massnahmenCharakteristika.hauptzielsetzungLand ?? "")),

```

Listing 6.7.: Die *Maßnahmencharakteristika* werden dem Tabellenkörper hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/widgets/massnahmen_table.dart](#)

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

6.5. Aktualisierung des *ViewModels*

Damit die Eingabefelder die neuen Werte eintragen können, muss das *ViewModel* die beobachtbaren *Subjekte* bereitstellen (Listing 6.8, Z. 12-17).

```

5 class MassnahmenFormViewModel {
6   final letzterStatus = BehaviorSubject<LetzterStatus?>.seeded(null);
7
8   final guid = BehaviorSubject<String?>.seeded(null);
9
10  final massnahmenTitel = BehaviorSubject<String>.seeded("");
11
12  final foerderklasse = BehaviorSubject<FoerderklasseChoice?>.seeded(null);
13  final kategorie = BehaviorSubject<KategorieChoice?>.seeded(null);
14  final zielflaeche = BehaviorSubject<ZielflaecheChoice?>.seeded(null);
15  final zieleinheit = BehaviorSubject<ZieleinheitChoice?>.seeded(null);
16  final hauptzielsetzungLand =
17    BehaviorSubject<ZielsetzungLandChoice?>.seeded(null);

```

Listing 6.8.: Die *Maßnahmencharakteristika* werden dem *ViewModel* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

6.5.1. Aktualisierung der *Setter*-Methode

Die Konvertierung des *Models* in das *ViewModel* erfolgt wie gewohnt über das Heraussuchen des korrekten Objektes aus der Menge der Auswahloptionen über dessen Abkürzung (Listing 6.9, Z. 29-36).

```

19 set model(Massnahme model) {
20   guid.value = model.guid;
21
22   letzterStatus.value = letzterStatusChoices
23     .fromAbbreviation(model.letzteBearbeitung.letzterStatus);
24   massnahmenTitel.value = model.identifikatoren.massnahmenTitel;
25
26   {
27     final mc = model.massnahmenCharakteristika;
28
29     foerderklasse.value =
30       foerderklasseChoices.fromAbbreviation(mc.foerderklasse);
31     kategorie.value = kategorieChoices.fromAbbreviation(mc.kategorie);
32
33     zielflaeche.value = zielflaecheChoices.fromAbbreviation(mc.zielflaeche);
34     zieleinheit.value = zieleinheitChoices.fromAbbreviation(mc.zieleinheit);
35     hauptzielsetzungLand.value =
36       hauptzielsetzungLandChoices.fromAbbreviation(mc.hauptzielsetzungLand);
37   }
38 }

```

Listing 6.9.: Konvertierung des *Models* in das *ViewModel* für die *Maßnahmencharakteristika*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

Wenn in jeder Zeile der Ausdruck `model.massnahmenCharakteristika` stehen würde, wäre die Leserlichkeit stark eingeschränkt. Das würde 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).

6.5.2. Aktualisierung der *Getter*-Methode

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 6.10, Z. 44). So ist es auch für den geschachtelten Wertetyp *MassnahmenCharakteristika* der Fall (Z. 45). 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).

```

40 Massnahme get model => Massnahme((b) => b
41   ..guid = guid.value
42   ..letzteBearbeitung.letzterStatus = letzterStatus.value?.abbreviation
43   ..letzteBearbeitung.letztesBearbeitungsDatum = DateTime.now().toUtc()
44   ..identifikatoren.update((b) => b..massnahmenTitel = massnahmenTitel.value)
45   ..massnahmenCharakteristika.update((b) => b
46     ..foerderklasse = foerderklasse.value?.abbreviation
47     ..kategorie = kategorie.value?.abbreviation
48     ..zielflaeche = zielflaeche.value?.abbreviation
49     ..zieleinheit = zieleinheit.value?.abbreviation
50     ..hauptzielsetzungLand = hauptzielsetzungLand.value?.abbreviation));

```

Listing 6.10.: Konvertierung des *ViewModels* in das *Model* für die *Maßnahmencharakteristika*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

6.6. Aktualisierung der Eingabemaske

Nach der Anpassung des *ViewModels* kann schließlich die Eingabemaske erweitert werden. Im letzten Schritt nahm die Selektionskarte für den letzten Status 12 Zeilen ein. 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 6.11, Z. 49) und das *Subjekt* `selectionViewModel` (Z. 50) übergeben. Nun übernimmt die Methode die Übergabe der Argumente an den Konstruktor der `SelectionCard` (Z. 51). 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).

```

48 Widget buildSelectionCard<ChoiceType extends Choice>(
49     {required Choices<ChoiceType> allChoices,
50     required BehaviorSubject<ChoiceType?> selectionViewModel}) {
51     return SelectionCard<ChoiceType>(
52         title: allChoices.name,
53         allChoices: allChoices,
54         initialValue: {
55             if (selectionViewModel.value != null) selectionViewModel.value!
56         },
57         onSelect: (selectedChoice) => selectionViewModel.value = selectedChoice,
58         onDeselect: (selectedChoice) => selectionViewModel.value = null,
59     );
60 }

```

Listing 6.11.: Die Methode `buildSelectionCard`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Der Grund, warum die Klasse `SelectionCard` den Titel aus der Menge der Auswahloptionen nicht selbstständig 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 deren 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) und `onDeselect` (Z. 58). Damit ist die Erstellung der Selektionskarte für den letzten Status mit 3 Zeilen (Listing 6.12) nun deutlich kürzer als die ursprüngliche Variante mit 12 Zeilen (siehe Seite 99).

```

77 buildSelectionCard(
78     allChoices: letzterStatusChoices,
79     selectionViewModel: vm.letzterStatus),

```

Listing 6.12.: Der Aufruf von `buildSelectionCard` für die Menge `letzterStatusChoices`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Unterhalb des Eingabefeldes für den Maßnahmentitel können nun die weiteren Selektionskarten ergänzt werden, die jeweils ebenfalls bloß 3 Zeilen einnehmen und damit eine hohe Übersichtlichkeit gewährleisten (Listing 6.13, Z. 82-98).

```

80 buildSectionHeadline("Identifikatoren"),
81 createMassnahmenTitelTextFormField(),
82 buildSectionHeadline("Maßnahmencharakteristika"),
83 buildSelectionCard(
84     allChoices: foerderklasseChoices,
85     selectionViewModel: vm.foerderklasse),
86 buildSelectionCard(
87     allChoices: kategorieChoices,
88     selectionViewModel: vm.kategorie),
89 buildSubSectionHeadline("Zielsetzung"),
90 buildSelectionCard(
91     allChoices: zielflaecheChoices,
92     selectionViewModel: vm.zielflaeche),
93 buildSelectionCard(
94     allChoices: zieleinheitChoices,
95     selectionViewModel: vm.zieleinheit),
96 buildSelectionCard(
97     allChoices: hauptzielsetzungLandChoices,
98     selectionViewModel: vm.hauptzielsetzungLand),

```

Listing 6.13.: Die *Maßnahmencharakteristika* Selektionskarten werden ergänzt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

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

Die Hilfsfunktionen `buildSectionHeadline` und `buildSubSectionHeadline` bauen die Überschriften (Listing 6.14, Z. 131-134) bzw. Zwischenüberschriften (Z. 136-139) mit unterschiedlichen Abständen zur Außenkante (Z. 132, 137) und unterschiedlichen Schriftgrößen (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 *left, top, right, bottom* – deutsch links, oben, rechts, unten.

```

131 Widget buildSectionHeadline(String text) => Padding(
132     padding: const EdgeInsets.fromLTRB(0, 24, 0, 8),
133     child: Text(text, style: const TextStyle(fontSize: 22)),
134 );
135
136 Widget buildSubSectionHeadline(String text) => Padding(
137     padding: const EdgeInsets.fromLTRB(4, 12, 0, 4),
138     child: Text(text, style: const TextStyle(fontSize: 14)),
139 );

```

Listing 6.14.: Die Hilfsfunktionen *buildSectionHeadline* und *buildSubSectionHeadline*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Damit ist die Implementierung für Schritt 2 beendet. Der Integrationstest kann nun verifizieren, dass die Eingaben erfolgen und in der *JSON*-Datei auftauchen würden.

7. Schritt 3 - Implementierung der grundlegenden Validierungsfunktion

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. 7.1).

Abbildung 7.1.: Die Eingabemaske in Schritt 3, Quelle: Eigene Abbildung

7.1. 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 Kindelemente des Typs `FormField` ausführt. Damit es alle Eingabefelder im Formular umgibt, wird es oberhalb des `Stack` eingefügt (Listing 7.1, Z. 161). Das `Form-Widget` muss über einen `key` registriert werden (Z. 162), damit auf die Validierungsfunktionen zurückgegriffen werden kann.

```
161 child: Form(  
162   key: formKey,  
163   child: Stack(  

```

Listing 7.1.: Einfügen des *Form-Widgets*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Die Erstellung des `formKey` findet zu Beginn der `build`-Methode des Eingabefelds statt (Listing 7.2, 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 das entsprechende Typargument `<FormState>` bei der Erstellung des `GlobalKey` übergeben werden.

```
17 Widget build(BuildContext context) {
18   final vm = AppState.of(context).viewModel;
19   final model = AppState.of(context).model;
20   final formKey = GlobalKey<FormState>();
```

Listing 7.2.: Der `formKey` wird erstellt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

7.2. Validierung des Maßnahmentitels

Das Eingabefeld für den Maßnahmentitel ist ein `TextFormField` (Listing 7.3, Z. 88). Es erbt vom Typ `FormField` und wird daher mit dem Väterelement `Form` verknüpft. Es beinhaltet bereits einen Parameter für die Validierungsfunktion namens `validator` (Z. 93-109). 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.

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 Maßnahmentitel bereits vergeben ist. Über das `Model` kann die Liste der Maßnahmen angefordert werden (Z. 99). Die Filterfunktion `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 des `guid` ist dafür ausreichend (Z. 100).

```

83 Widget createMassnahmenTitelTextFormField() {
84   final focusNode = FocusNode();
85   return Card(
86     child: Padding(
87       padding: const EdgeInsets.all(16.0),
88       child: TextFormField(
89         focusNode: focusNode,
90         initialValue: vm.massnahmenTitel.value,
91         decoration: const InputDecoration(
92           hintText: 'Maßnahmentitel', labelText: 'Maßnahmentitel'),
93         validator: (title) {
94           if (title == null || title.isEmpty) {
95             focusNode.requestFocus();
96             return 'Bitte Text eingeben';
97           }
98           var massnahmeTitleDoesAlreadyExists =
99             model.storage.value.massnahmen.any((m) =>
100               m.guid != vm.guid.value &&
101               m.identifikatoren.massnahmenTitel ==
102                 vm.massnahmenTitel.value);
103
104           if (massnahmeTitleDoesAlreadyExists) {
105             focusNode.requestFocus();
106             return 'Dieser Maßnahmentitel ist bereits vergeben';
107           }
108           return null;
109         },
110         onChanged: (value) {
111           vm.massnahmenTitel.value = value;
112         },
113       ),
114     ),
115   );
116 }

```

Listing 7.3.: Die Funktion *createMassnahmenTitelTextFormField* mit Validierung, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

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` (Z. 108). In dem Kontext der `validator`-Funktion bedeutet die Rückgabe von `null`, dass die Validierung erfolgreich war.

7.3. 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 Selektionskarte 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 7.4, Z. 121), welches die Selektionskarte in der `builder`-Funktion erstellt und an den Parametern nichts ändert, außer einen weiteren hinzuzufügen: den Text für die Fehlermeldung (Z. 143). Der erste Parameter der `builder`-Funktion ist das *FormFieldState*-Objekt von `FormField`. Es enthält die *Getter*-Methode `errorText`, die bei gegebenenfalls fehlgeschlagener Validierung die zurückgegebene Fehlermeldung enthält.

```

118 Widget buildSelectionCard<ChoiceType extends Choice>(
119     {required Choices<ChoiceType> allChoices,
120     required BehaviorSubject<ChoiceType?> selectionViewModel}) {
121     return FormField(
122         validator: (_) {
123             Iterable<Choice> choices = {
124                 if (selectionViewModel.value != null) selectionViewModel.value!
125             };
126
127             if (choices.isEmpty) {
128                 return "Feld ${allChoices.name} enthält keinen Wert!";
129             }
130
131             return null;
132         },
133         builder: (field) => SelectionCard<ChoiceType>(
134             title: allChoices.name,
135             allChoices: allChoices,
136             initialValue: {
137                 if (selectionViewModel.value != null)
138                     selectionViewModel.value!
139             },
140             onSelect: (selectedChoice) =>
141                 selectionViewModel.value = selectedChoice,
142             onDeselect: (selectedChoice) => selectionViewModel.value = null,
143             errorText: field.errorText,
144         ));
145 }

```

Listing 7.4.: Die Methode `buildSelectionCard` mit Validierung, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

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

```

19 final OnSelect<ChoiceType> onSelect;
20 final OnDeselect<ChoiceType> onDeselect;
21 final String? errorText;
22
23 SelectionCard(
24   {required this.title,
25   required Iterable<ChoiceType> initialValue,
26   required this.allChoices,
27   required this.onSelect,
28   required this.onDeselect,
29   this.errorText,
30   Key? key})

```

Listing 7.5.: `errorText` wird der `SelectionCard` hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/widgets/selection_card.dart](#)

Durch Einfügen einer `Column` zwischen der `Card` (Listing 7.6, 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).

```

53 return Card(
54   child: Column(
55     crossAxisAlignment: CrossAxisAlignment.start,
56     children: [
57       ListTile(
58         focusNode: focusNode,
59         title: Text(title),
60         subtitle: Text(
61           selectedChoices.map((c) => c.description).join(", "),
62         trailing: const Icon(Icons.edit),
63         onTap: navigateToSelectionScreen,
64       ),
65       if (errorText != null)
66         Padding(
67           padding: const EdgeInsets.all(8.0),
68           child: Text(errorText!,
69             style:
70               const TextStyle(fontSize: 12.0, color: Colors.red)),
71         )
72     ],
73   ),
74 );

```

Listing 7.6.: `errorText` wird ausgegeben, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/widgets/selection_card.dart](#)

7.4. 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 7.7, Z. 207-213). Der ursprüngliche `FloatingActionButton` speichert fortan 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 E.1 in Anhang E auf Seite 197). Anschließend wird der `Navigator` erneut aufgefordert, zum Übersichtsbildschirm zurückzukehren (Z. 224). Sollte es allerdings zur Ausführung des `else`-Blocks kommen (Z. 225-227), da die Maßnahme doch als *abgeschlossen* markiert wurde und nicht alle Eingabefelder valide waren, so erhält der Benutzer eine Fehlermeldung. Die neu implementierte Hilfsfunktion `showValidationError` wird dafür verwendet (Z. 226).

```

206 children: [
207   FloatingActionButton(
208     mini: true,
209     heroTag: 'save_draft_floating_action_button',
210     child: const Icon(Icons.paste, color: Colors.white),
211     backgroundColor: Colors.orange,
212     onPressed: saveDraftAndGoBackToOverviewScreen,
213   ),
214   const SizedBox(
215     height: 10,
216   ),
217   FloatingActionButton(
218     tooltip: saveMassnahmeTooltip,
219     heroTag: 'save_floating_action_button',
220     child: const Icon(Icons.check, color: Colors.white),
221     onPressed: () {
222       if (inputsAreValidOrNotMarkedFinal()) {
223         saveRecord();
224         Navigator.of(context).pop();
225       } else {
226         showValidationError();
227       }
228     },
229   ),
230 ],

```

Listing 7.7.: Der `FloatingActionButton` zum Speichern der Maßnahmen im Entwurfsmodus, Quelle: Eigenes Listing, Datei: `Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart`

Auch der `WillPopScope` erhält die gleiche Fehlerbehandlung (Listing 7.8). 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 Typs `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 Fehler-

meldung (Z. 157) und dieses Mal evaluiert das `Future`-Objekt zu `false`, um die Navigation zu unterbinden (Z. 158).

```

151 body: WillPopScope(
152   onWillPop: () {
153     if (inputsAreValidOrNotMarkedFinal()) {
154       saveRecord();
155       return Future.value(true);
156     } else {
157       showValidationError();
158       return Future.value(false);
159     }
160   },

```

Listing 7.8.: Die Fehlerbehandlung im *WillPopScope*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

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 anschließend im *Model* ab (Z. 31) und navigiert zur letzten Route zurück (Z. 32), welche 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.

```

22 void saveDraftAndGoBackToOverviewScreen() {
23   ScaffoldMessenger.of(context)
24     ..hideCurrentSnackBar()
25     ..showSnackBar(
26       const SnackBar(content: Text('Entwurf wird gespeichert ...')));
27
28   var draft = vm.model.rebuild((b) =>
29     b.letzteBearbeitung.letzterStatus = LetzterStatus.bearb.abbreviation);
30
31   model.putMassnahmeIfAbsent(draft);
32   Navigator.of(context).pop();
33 }

```

Listing 7.9.: Die Funktion *saveDraftAndGoBackToOverviewScreen*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Die Hilfsfunktion `inputsAreValidOrNotMarkedFinal` überprüft zunächst, ob der letzte Status ein anderer ist als *abgeschlossen* (Listing 7.10, Z. 72). 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 *FormFieldState*-Objekt benötigt. Der Zugriff darauf ist einfach, da es über einen *GlobalKey* registriert wurde. Über `formKey.currentState` kann das *FormFieldState*-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 war mindestens ein Formularfeld invalide und damit bleibt nur die Rückgabe von `false` übrig (Z. 80).

```

71 bool inputsAreValidOrNotMarkedFinal() {
72   if (vm.letzterStatus.value != LetzterStatus.fertig) {
73     return true;
74   }
75
76   if (formKey.currentState!.validate()) {
77     return true;
78   }
79
80   return false;
81 }

```

Listing 7.10.: Die Funktion *inputsAreValidOrNotMarkedFinal*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Sollte es zu einem Fehler kommen, so zeigt die Hilfsfunktion `showValidationError` dem Benutzer die entsprechende Fehlermeldung an (Listing 7.11). 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 auch von jedem beliebigen *Widget*. Zunächst kommt dazu das *Widget Row* zum Einsatz (Z. 46). Ähnlich wie das *Widget Column* erlaubt es Kindelemente in einer Reihe aufzulisten. Im Gegensatz zur *Column* allerdings nun horizontal statt vertikal. Als letztes Element der *Row* wird der `ElevatedButton` verwendet (Z. 51). Genauso wie bereits der *FloatingActionButton* zum Speichern der Maßnahme im Entwurfsmodus verwendet nun auch dieser `ElevatedButton` die Funktion `saveDraftAndGoBackToOverviewScreen` (Z. 52).

```

44 void showValidationError() {
45   ScaffoldMessenger.of(context).showSnackBar(SnackBar(
46     content: Row(
47       children: [
48         Text(
49           'Fehler im Formular trotz Status "${LetzterStatus.fertig.description}"',
50           const SizedBox(width: 4),
51         ElevatedButton(
52           onPressed: saveDraftAndGoBackToOverviewScreen,
53           child: Padding(
54             padding: const EdgeInsets.fromLTRB(4, 4, 8, 4),
55             child: Row(
56               children: const [
57                 Icon(Icons.paste, color: Colors.white),
58                 SizedBox(width: 4),
59                 Text(
60                   "Entwurf speichern?",
61                   style: TextStyle(fontSize: 18.0, color: Colors.white),
62                 ),
63               ],
64             ),
65           ),
66         ],
67       )),
68   ));
69 }

```

Listing 7.11.: Die Funktion `showValidationError`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

8. Schritt 4 - Kompatibilitätsvalidierung

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. 8.1).



Abbildung 8.1.: Der Selektionsbildschirm in Schritt 4, Quelle: Eigene Abbildung

Wenn eine Auswahloption selektiert ist und durch eine weitere Selektion in einem anderen Feld anschließend invalide geworden ist, wird diese rot gekennzeichnet (Abb. 8.2).

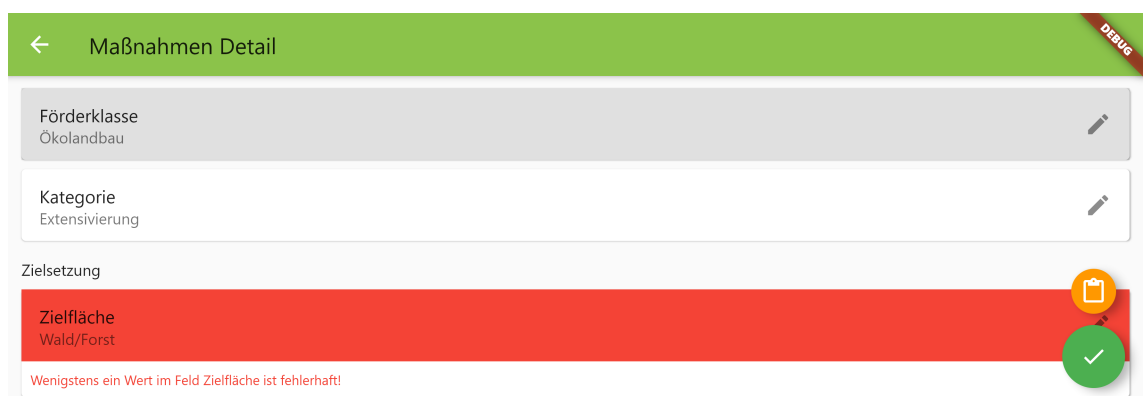


Abbildung 8.2.: Die Eingabemaske in Schritt 2 mit einem selektierten invaliden Wert, Quelle: Eigene Abbildung

In der Eingabemaske wird dann das gesamte Eingabefeld rot eingefärbt (Abb. 8.3).

Abbildung 8.3.: Der Selektionsbildschirm in Schritt 4 mit einem selektierten invaliden Wert, Quelle: Eigene Abbildung

8.1. Hinzufügen der Bedingungen zu den Auswahloptionen

Es gibt einfache Bedingungen wie beispielsweise die der Zielfläche *AL*, deren Auswahl nur dann erfolgen kann, wenn die *Kategorie Anbau Zwischenfrucht/Untersaat* nicht ausgewählt ist (Listing 8.1).

```
79 static final al = ZielflaecheChoice("al", "AL",
80   condition: (choices) => !choices.contains(KategorieChoice.zf_us));
```

Listing 8.1.: Der Klassenvariablen *al* des Typs *ZielflaecheChoice* wird eine Bedingung hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/choices.dart](#)

Doch es tauchen auch komplexe Bedingungen auf, wie etwa die Abhängigkeit der Zielfläche *Wald/Forst* (Listing 8.2). Um sie auszuwählen, muss die *Förderklasse* einen der drei folgenden Werte 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.

```
95 static final wald = ZielflaecheChoice("wald", "Wald/Forst",
96   condition: (choices) =>
97     (choices.contains(FoerderklasseChoice.ea) ||
98       choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
99       choices.contains(FoerderklasseChoice.aukm_ohne_vns)) &&
100     (!choices.contains(KategorieChoice.zf_us) ||
101       !choices.contains(KategorieChoice.bes_kult_rass)));
```

Listing 8.2.: Der Klassenvariablen *wald* des Typs *ZielflaecheChoice* wird eine Bedingung hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/choices.dart](#)

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

(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 Bedingung eines der beiden Typen erfüllt wäre. Sollte also beispielsweise für die *Förderklasse* die Option *Erschwernisausgleich* gewählt sein, so wäre es völlig unerheblich, welche Auswahl für die *Kategorie* selektiert ist. Die Bedingung wäre trotzdem erfüllt, auch wenn für die *Kategorie* die nicht erlaubte Option *Anbau Zwischenfrucht/Untersaat* gewählt ist. Für die Liste aller hinzugefügten Bedingungen siehe Anhang F auf den Seiten 199 bis 201.

Bei der Bedingung handelt es sich um eine Funktion, die einen Wahrheitswert `bool` zurückgibt und als Parameter die Menge aller bisher ausgewählten Auswahloptionen vom Typ `Set<Choice>` übergeben bekommt. Die Signatur dieser Funktion wird als Typdefinition mit dem Namen `Condition` deklariert (Listing 8.3, Z. 3). Über diese Typdefinition kann sie als Instanzvariable in der Klasse `Choice` deklariert werden (Z. 8). Der Konstruktor erhält einen weiteren Parameter für die Bedingung (Z. 12). Er ist optional, da es Auswahloptionen 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.

```

3  typedef Condition = bool Function(Set<Choice> choices);
4
5  class Choice {
6      final String description;
7      final String abbreviation;
8      final Condition condition;
9
10     bool conditionMatches(Set<Choice> choices) => condition.call(choices);
11
12     const Choice(this.abbreviation, this.description, {Condition? condition})
13         : condition = condition ?? _conditionIsAlwaysMet;
14
15     static bool _conditionIsAlwaysMet(Set<Choice> choices) => true;
16 }

```

Listing 8.3.: Der Klasse *Choices* wird die Instanzvariable *condition* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/base/choice.dart](#)

Sollte die übergebene Bedingung 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 Instanzvariable `condition` den Wert `null` erhält. Da der Ausdruck

rechts von dem `??` nicht *null* sein kann, so kann auch der gesamte Ausdruck der vorliegenden *if-null Expression* nicht zu *null* evaluieren. Damit ist es möglich, die Instanzvariable `condition` ohne das Suffix `?` als Variable ohne Null-Zulässigkeit zu deklarieren (Z. 8). Die Instanzmethode `conditionMatches` ruft die übergebene Funktion für die Bedingung über die Methode `call` auf (Z. 10). Das erlaubt es, den Ausdruck vereinfacht darzustellen. Der Ausdruck `wald.condition(priorChoices)` kann daraufhin durch die explizitere Schreibweise `wald.conditionMatches(priorChoices)` ersetzt werden.

8.2. Hinzufügen der Momentaufnahme aller ausgewählten Optionen im gesamten Formular

Die Menge der bisherigen Auswahloptionen 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 8.4).

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 *Subjekte* gehorcht. Dies wird durch die Funktion `combineLatest` des Pakets `RxDart` ermöglicht (Z. 24). Sie erlaubt die Übergabe einer Kollektion von *Streams*. In diesem Fall sind das alle *Subjekte* 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. 30-38). 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 *Subjekte* integriert werden (Z. 31-37). Das *collection if* Statement schließt dabei jeweils den Wert `null` aus (Z. 32-36). Somit tauchen keine Null-Werte in der Menge auf und damit kann der *Stream* mit dem Typargument `Set<Choice>` ohne Null-Zulässigkeit deklariert werden (Z. 24). Sind alle Auswahlfelder nicht belegt, so ist die Menge leer. Doch der kombinierte *Stream* `choicesStream` liefert immer nur die neuen Ereignisse und speichert nicht den zuletzt

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

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

übermittelten Wert zwischen. Deshalb wird das `BehaviorSubject` `priorChoices` verwendet. Die Methode `listen` horcht auf Änderungen des `choicesStream`-Objektes und fügt das übertragene Ereignis immer `priorChoices` hinzu (Z. 40). 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`.

```

20 BehaviorSubject<Set<Choice>> priorChoices =
21     BehaviorSubject<Set<Choice>>.seeded({});
22
23 MassnahmenFormViewModel() {
24     Stream<Set<Choice>> choicesStream = Rx.combineLatest([
25         foerderklasse,
26         kategorie,
27         zielflaeche,
28         zieleinheit,
29         hauptzielsetzungLand,
30     ], (_) {
31         return {
32             if (foerderklasse.value != null) foerderklasse.value!,
33             if (kategorie.value != null) kategorie.value!,
34             if (zielflaeche.value != null) zielflaeche.value!,
35             if (zieleinheit.value != null) zieleinheit.value!,
36             if (hauptzielsetzungLand.value != null) hauptzielsetzungLand.value!,
37         };
38     });
39
40     choicesStream.listen((event) => priorChoices.add(event));
41 }

```

Listing 8.4.: Das *BehaviorSubject* *priorChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

8.3. 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 8.5, Z. 143).

```

140 builder: (field) => SelectionCard<ChoiceType>(
141     title: allChoices.name,
142     allChoices: allChoices,
143     priorChoices: vm.priorChoices,
```

Listing 8.5.: Dem Konstruktor der *SelectionCard* wird das *BehaviorSubject* *priorChoices* übergeben, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Die Klasse `SelectionCard` deklariert die `priorChoices` als Instanzvariable (Listing 8.6, Z. 19) und initialisiert sie direkt bei der Übergabe im Konstruktor, ohne sie zu modifizieren (Z. 28). 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 Auswahloptionen übergeben werden, so wäre diese eine Kopie. Diese Kopie hätte den Zustand einer Momentaufnahme aller bisherigen Auswahloptionen zum Zeitpunkt der Konstruktion der Selektionskarte. 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 *StatefulWidget* und Zustandsänderungen über die *setState*-Methode würde dies das Neuzeichnen des gesamten Formulars bedeuten.

Performer 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 8.7, Z. 52) und damit auf die Änderungen aller Auswahlfelder.

Vor der Konstruktion der Karte wird nun überprüft, ob eine 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

```

15 class SelectionCard<ChoiceType extends Choice> extends StatelessWidget {
16   final String title;
17   final BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel;
18   final Choices<ChoiceType> allChoices;
19   final BehaviorSubject<Set<Choice>> priorChoices;
20   final OnSelect<ChoiceType> onSelect;
21   final OnDeselect<ChoiceType> onDeselect;
22   final String? errorText;
23
24   SelectionCard(
25     {required this.title,
26     required Iterable<ChoiceType> initialValue,
27     required this.allChoices,
28     required this.priorChoices,
29     required this.onSelect,
30     required this.onDeselect,
31     this.errorText,
32     Key? key})

```

Listing 8.6.: Die Klasse *SelectionCard* erhält die Instanzvariable *priorChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/widgets/selection_card.dart](#)

`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 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.

```

51 return StreamBuilder(
52   stream: priorChoices,
53   builder: (context, snapshot) {
54     final selectedChoices = selectionViewModel.value;
55     final bool wrongSelection = selectedChoices
56       .any((c) => !c.conditionMatches(priorChoices.value));
57
58     return Card(
59       child: Column(
60         crossAxisAlignment: CrossAxisAlignment.start,
61         children: [
62           ListTile(
63             focusNode: focusNode,
64             title: Text(title),
65             subtitle: Text(
66               selectedChoices.map((c) => c.description).join(", "),
67             trailing: const Icon(Icons.edit),
68             onTap: navigateToSelectionScreen,
69             tileColor:
70               wrongSelection || errorText != null ? Colors.red : null,

```

Listing 8.7.: Die *SelectionCard* reagiert auf Änderungen des Streams *priorChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/widgets/selection_card.dart](#)

8.4. Reagieren des Auswahlbildschirms auf die ausgewählten Optionen

Der Auswahlbildschirm wird im Folgenden um zwei weitere Funktionalitäten erweitert:

- 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 (Listing 8.8). 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 und nicht selektiert sind. Eine Schleife iteriert über alle verfügbaren Optionen, welche der Selektionsbildschirm anzeigt (Z. 98-105). Sollte die Option in den selektierten Optionen enthalten (Z. 99) oder aber mit den Selektionen aller anderen Auswahlfelder kompatibel sein (Z. 100), 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 Auswahloptionen 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 nun noch ein weiterer Vergleich, ob die Option mit den ausgewählten Optionen aller anderen Auswahlfelder inkompatibel ist (Z. 111). Das Ergebnis des Vergleiches wird in der lokalen Variablen `selectedButDoesNotMatch` gespeichert (Z. 110).

Sollte diese Variable wahr sein, so erscheint das `CheckboxListTile`-Element mit einem rot eingefärbten Hintergrund (Z. 118). 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), 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. 134-141). Solche unselektierbaren Auswahloptionen werden schlicht als `ListTile`-Element statt als `CheckBoxListTile` 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 (Z. 139). `Icons.close` zeichnet ein Kreuz-Symbol, um zu signalisieren, dass diese Option nicht anwählbar ist.

```

90 body: StreamBuilder(
91   stream: selectionViewModel,
92   builder: (context, snapshot) {
93     final selectedChoices = selectionViewModel.value;
94
95     Set<ChoiceType> selectedAndSelectableChoices = {};
96     Set<ChoiceType> unselectableChoices = {};
97
98     for (ChoiceType c in allChoices) {
99       if (selectedChoices.contains(c) ||
100         c.conditionMatches(priorChoices.value)) {
101         selectedAndSelectableChoices.add(c);
102       } else {
103         unselectableChoices.add(c);
104       }
105     }
106
107     return ListView(children: [
108       ...selectedAndSelectableChoices.map((ChoiceType c) {
109         bool isSelected = selectedChoices.contains(c);
110         bool selectedButDoesNotMatch =
111           !c.conditionMatches(priorChoices.value);
112
113         return CheckboxListTile(
114           key: Key(
115             "valid choice ${allChoices.name} - ${c.abbreviation}"),
116           controlAffinity: ListTileControlAffinity.leading,
117           title: Text(c.description),
118           tileColor: selectedButDoesNotMatch ? Colors.red : null,
119           value: isSelected,
120           onChanged: (bool? selected) {
121             if (selected != null) {
122               selectionViewModel.value =
123                 selectionViewModel.value.rebuild((b) {
124                   b.replace(isSelected ? [] : [c]);
125                 });
126             if (selected) {
127               onSelect(c);
128             } else {
129               onDeselect(c);
130             }
131           }
132         );
133       }).toList(),
134       ...unselectableChoices.map((Choice c) {
135         return ListTile(
136           key: Key(
137             "invalid choice ${allChoices.name} - ${c.abbreviation}"),
138           title: Text(c.description),
139           leading: const Icon(Icons.close));
140       }).toList()
141     ]);

```

Listing 8.8.: Der Selektionsbildschirm in Schritt 4, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/widgets/selection_card.dart](#)

8.4.1. 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 Maßnahme 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 8.9).

```

121 return FormField(
122   validator: (_) {
123     Iterable<Choice> choices = {
124       if (selectionViewModel.value != null) selectionViewModel.value!
125     };
126
127     if (choices.isEmpty) {
128       return "Feld ${allChoices.name} enthält keinen Wert!";
129     }
130
131     bool atLeastOneValueInvalid =
132       choices.any((c) => !c.conditionMatches(vm.priorChoices.value));
133
134     if (atLeastOneValueInvalid) {
135       return "Wenigstens ein Wert im Feld ${allChoices.name} ist fehlerhaft!";
136     }
137
138     return null;
139   },
140   builder: (field) => SelectionCard<ChoiceType>(

```

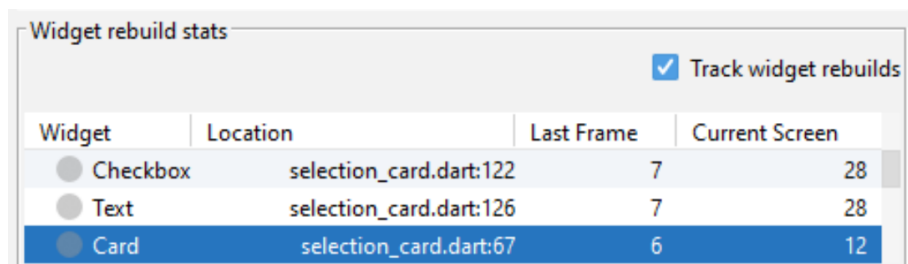
Listing 8.9.: Die `validator` Funktion von `FormField` in Schritt 4, Quelle: Eigenes Listing, Datei: [Quelldatei/Schritt-4/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Sollte auch nur eine der selektierten Optionen `choices` die ihr hinterlegte Bedingung nicht erfüllen (Z. 132), so speichert die lokale Variable `atLeastOneValueInvalid` den Wert `true` ab (Z. 131). In diesem 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 gewählten Optionen die jeweils hinterlegten Bedingungen erfüllen, so wird die `validator`-Funktion jeweils `null` statt einer Fehlermeldung zurückgeben (Z. 138). Nur dann kann eine Maßnahme mit dem Status *abgeschlossen* gespeichert werden.

9. Schritt 5 - Laufzeitoptimierung

Im letzten Schritt wurde das primäre Problem der Formularanwendung gelöst: Auswahloptionen sollen nur dann anwählbar sein, wenn sie die hinterlegten Bedingungen 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. 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*. Dieser zeigt, dass sich bei jeder Auswahl einer Option sechs *Card*-Elemente aktualisieren (Abb. 9.1). Das ist der Fall, da es im Formular in Summe sechs Selektionskarten mit einem darin befindlichen *Card*-Widget gibt.



Widget	Location	Last Frame	Current Screen
Checkbox	selection_card.dart:122	7	28
Text	selection_card.dart:126	7	28
Card	selection_card.dart:67	6	12

Abbildung 9.1.: Das *Card-Widget* wird sechsmal neu gezeichnet, Quelle: Eigene Abbildung

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 Selection-Card* den `StreamBuilder` in der `build`-Methode zurückgibt, wird der *Stream* `validityChanged` erstellt (Listing 9.2, 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 (Z. 53). Sollte es sich beispielsweise bei der aktuellen Selektionskarte um das Auswahlfeld der *Zieleinheit* handeln, so könnte der ausgewählte Wert *ha* sein.

Mit dem Aufruf `.any((c) => !c.conditionMatches(choices))` wird nun überprüft, ob der ausgewählte Wert – im Fall eines Einfachauswahlfeldes – oder die ausgewählten Werte – bei einem Mehrfachauswahlfeld – mit der neuen Momentaufnahme der Selektionen im Formular kompatibel sind. Für die *Zieleinheit ha* gelten folgende Bedingungen: Für die *Zielfläche* dürfen die Option *keine Angabe/Vorgabe* und *bitte um Unterstützung* nicht gewählt sein (Listing 9.1, Z. 166-167). Das bedeutet im Umkehrschluss, dass nur die Optionen *AL*, *GL*, *LF*, *DK/SK*, *HFF*, *Landschaftselement/Biotop o.Ä.* oder *Wald/Forst* gewählt sein dürfen.

```

164 static final ha = ZieleinheitChoice("ha", "ha",
165     condition: (choices) =>
166         !choices.contains(ZielflaecheChoice.ka) &&
167         !choices.contains(ZielflaecheChoice.contact));

```

Listing 9.1.: Der Klassenvariablen *ha* des Typs *ZielflaecheChoice* wird eine Bedingung hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-5/conditional_form/lib/choices/choices.dart](#)

Wurde also beispielsweise bei der neuen Selektion in der *Zielfläche* die Option *keine Angabe/Vorgabe* ausgewählt, so würde die Option *ha* invalide werden, da sie nicht mit den *Zieleinheit*-Optionen *keine Angabe/Vorgabe* bzw. *bitte um Unterstützung* kompatibel ist.

Die Methode `map` (Listing 9.2, Z. 52) 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 Selektionskarte valide sind. Ist er dagegen *false*, so ist wenigstens eine der ausgewählten Optionen mit den restlichen Selektionen der anderen Auswahlfelder im Formular inkompatibel.

Der resultierende *Stream* wird weiter transformiert: Durch die Funktion `distinct` (Z. 54) werden nur Ereignisse gesendet, sofern sie sich von dem letzten Ereignis unterscheiden. Der *Stream* `validityChanged` sendet also immer genau dann Ereignisse, wenn sich etwas an der Validität der Auswahloptionen der aktuellen Selektionskarte ändert.

```

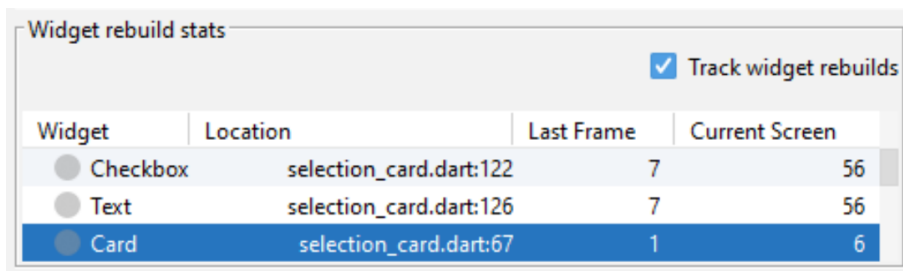
51 final validityChanged = priorChoices
52   .map((choices) =>
53     selectionViewModel.value.any((c) => !c.conditionMatches(choices)))
54   .distinct();
55
56 final needsRepaint = BehaviorSubject.seeded(true);
57 validityChanged.listen((value) => needsRepaint.add(true));
58 selectionViewModel.listen((value) => needsRepaint.add(true));
59
60 return StreamBuilder(
61   stream: needsRepaint,
62   builder: (context, snapshot) {
63     final selectedChoices = selectionViewModel.value;
64     final bool wrongSelection = selectedChoices
65       .any((c) => !c.conditionMatches(priorChoices.value));
66
67     return Card(
68       child: Column(
69         crossAxisAlignment: CrossAxisAlignment.start,
70         children: [
71           ListTile(

```

Listing 9.2.: Der *Stream* `validityChanged` in Schritt 5, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-5/conditional_form/lib/widgets/selection_card.dart](#)

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. Es muss aber eine Aktualisierung stattfinden, damit der neue Wert in der Selektionskarte abgebildet wird. 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 `true` initialisiert. Dafür ist unerheblich, welcher Wert in dem *Stream* aktuell gespeichert ist. Wesentlich ist nur, dass ein neues Ereignis hinzugefügt wird, um die Aktualisierung der Oberfläche auszulösen. 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 *Subjekt* `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 deren Validität ändern.

Ein Beispiel: Für die *Zielfläche* ist *AL* und für die *Zieleinheit* ist *ha* ausgewählt. Beide Optionen sind miteinander kompatibel. Nun erfolgt eine weitere Selektion: Für *Zielfläche* wird nun die Option *GL* gewählt. Auch sie ist mit der *Zieleinheit* *ha* kompatibel. Durch die Selektion hat sich der Wert der Selektionskarte der *Zielfläche* geändert, weshalb sie neu gezeichnet werden muss. Alle anderen Auswahlfelder im Formular sind aber nicht betroffen. Im *Flutter Performance*-Tab ist zu beobachten, dass das *Widget Card* nur einmal neu gezeichnet wurde (Abb. 9.2).



Widget rebuild stats

☒ Track widget rebuilds

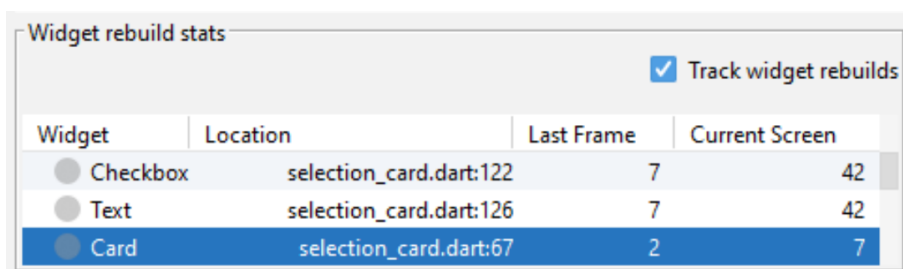
Widget	Location	Last Frame	Current Screen
Checkbox	selection_card.dart:122	7	56
Text	selection_card.dart:126	7	56
Card	selection_card.dart:67	1	6

Abbildung 9.2.: Das *Card-Widget* wird einmal neu gezeichnet, Quelle: Eigene Abbildung

Durch eine weitere Selektion für die *Zielfläche* soll nun provoziert werden, dass die Auswahl der *Zieleinheit* invalide wird. Deshalb wird für die *Zielfläche* nun *keine Angabe/Vorgabe* selektiert. Die *Zieleinheit ha* ist damit nicht kompatibel. Deshalb müssen sich nun zwei Auswahlfelder aktualisieren:

- die Selektionskarte *Zielfläche*, weil sich der darin ausgewählte Wert geändert hat und
- die Selektionskarte *Zieleinheit*, da sie zuvor valide war und nun invalide ist.

Der *Flutter Performance*-Tab reflektiert dies, da sich das *Widget Card* nun zweimal neu zeichnet (Abb. 9.3).



Widget rebuild stats

☒ Track widget rebuilds

Widget	Location	Last Frame	Current Screen
Checkbox	selection_card.dart:122	7	42
Text	selection_card.dart:126	7	42
Card	selection_card.dart:67	2	7

Abbildung 9.3.: Das *Card-Widget* wird zweimal neu gezeichnet, Quelle: Eigene Abbildung

10. Schritt 6 - Hinzufügen von Mehrfachauswahlfeldern

In diesem Schritt soll das Formular um Mehrfachauswahlfelder erweitert werden. Im Speziellen handelt es sich um das Auswahlfeld *Nebenziele* (Abb. 10.1).

Abbildung 10.1.: Die Eingabemaske in Schritt 6, Quelle: Eigene Abbildung

Es beinhaltet die gleichen Auswahloptionen wie das Auswahlfeld *Hauptzielsetzung* (Abb. 10.2).

Abbildung 10.2.: Im Selektionsbildschirm für die *Nebenziele* können mehrere Optionen gewählt werden, Quelle: Eigene Abbildung

10.1. Integrationstest erweitern

Zunächst wird der Integrationstest um die Auswahl der *Nebenziele* erweitert (Listing 10.1).

```

118 await tabSelectionCard(hauptzielsetzungLandChoices);
119 await tabOption(ZielsetzungLandChoice.biodiv, tabConfirm: true);
120
121 await tabSelectionCard(nebenzielsetzungLandChoices);
122 await tabOption(ZielsetzungLandChoice.bsch);
123 await tabOption(ZielsetzungLandChoice.klima, tabConfirm: true);
124
125 var saveMassnahmeButton = find.byTooltip(saveMassnahmeTooltip);
126 await tester.tap(saveMassnahmeButton);
127 await tester.pumpAndSettle(durationAfterEachStep);

```

Listing 10.1.: Der Integrationstest klickt die Karte für die *Nebenziele* und selektiert darin 2 Optionen, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/integration_test/app_test.dart](#)

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 Selektionsbildschirm, in welchem die Option *Bodenschutz* (Z. 122) und anschließend die Option *Klima* (Z. 123) gewählt werden. Mit Auswahl der letzten Option und durch die damit verbundene Übergabe des Arguments `true` für den optionalen Parameter `tabConfirm` wird der Selektionsbildschirm umgehend wieder geschlossen. Anschließend erfolgt erneut das Speichern der Maßnahme (Z. 125-126).

Anders als bei den bisherigen Schlüssel-Werte-Paaren innerhalb des Objektes mit dem Schlüssel `'massnahmenCharakteristika'` kann der Wert der *Nebenziele* nicht als einzelner *String* gespeichert werden (Listing 10.2). 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).

```

136 var expectedJson = {
137   'letzteBearbeitung': {'letzterStatus': 'fertig'},
138   'identifikatoren': {'massnahmenTitel': massnahmeTitle},
139   'massnahmenCharakteristika': {
140     'nebenziele': [
141       'bsch',
142       'klima',
143     ],
144     'foerderklasse': 'aukm_ohne_vns',
145     'kategorie': 'extens',
146     'zielflaeche': 'al',
147     'zieleinheit': 'ha',
148     'hauptzielsetzungLand': 'biodiv'
149   },
150 };

```

Listing 10.2.: Der Integrationstest überprüft im *JSON*-Dokument den Schlüssel *nebenziele*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/integration_test/app_test.dart](#)

10.2. 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 10.3, Z. 223-224).

```

219 final hauptzielsetzungLandChoices = Choices<ZielsetzungLandChoice>(
220     _zielsetzungLandChoices,
221     name: "Hauptzielsetzung Land");
222
223 final nebenzielsetzungLandChoices =
224     Choices<ZielsetzungLandChoice>(_zielsetzungLandChoices, name: "Nebenziele");

```

Listing 10.3.: Die Menge *nebenzielsetzungLandChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/choices/choices.dart](#)

10.3. Aktualisierung des *Models*

Um die Liste der *Nebenziele* im Wertetyp *MassnahmenCharakteristika* einzufügen, kann der Datentyp *BuiltSet* verwendet werden (Listing 10.4, Z. 77). Die *Getter*-Methode *nebenziele* bedarf keiner Null-Zulässigkeit, da das Nicht-Vorhandensein von Werten darüber erreicht werden kann, dass die Menge leer ist.

```

68 abstract class MassnahmenCharakteristika
69     implements
70         Built<MassnahmenCharakteristika, MassnahmenCharakteristikaBuilder> {
71     String? get foerderklasse;
72     String? get kategorie;
73     String? get zielflaeche;
74     String? get zieleinheit;
75     String? get hauptzielsetzungLand;
76
77     BuiltSet<String> get nebenziele;

```

Listing 10.4.: Die *Nebenziele* werden dem Wertetyp *MassnahmenCharakteristika* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/data_model/massnahme.dart](#)

10.4. Aktualisierung der Übersichtstabelle

Für das Einfügen der Überschrift in die Übersichtstabelle gibt es keine Unterschiede zum bisherigen Vorgehen. Die Überschrift wird nach der Spaltenüberschrift für die *Hauptzielsetzung* eingefügt (Listing 10.5, Z. 28).

```
27 _buildColumnHeader(const Text("Hauptzielsetzung Land")),
28 _buildColumnHeader(const Text("Nebenziele")),
```

Listing 10.5.: Die *Nebenziele* werden dem Tabellenkopf hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/widgets/massnahmen_table.dart](#)

Die Anzeige der Werte in den Tabellenzellen ist dagegen unterschiedlich (Listing 10.6, Z. 44-50). Dieses Mal handelt es sich um die Aufzählung von mehreren Werten, weshalb ein *Column-Widget* die einzelnen Einträge untereinander auflistet (Z. 46-50). Jedes Element des *BuiltSet* `nebenziele` (Z. 47) wird über die Methode `map` jeweils in ein Element des *Widgets* `Text` konvertiert (Z. 48).

```
42 _buildSelectableCell(m,
43   Text(m.massnahmenCharakteristika.hauptzielsetzungLand ?? "")),
44 _buildSelectableCell(
45   m,
46   Column(
47     children: m.massnahmenCharakteristika.nebenziele
48       .map((n) => Text(n))
49       .toList(),
50   )),
```

Listing 10.6.: Die *Nebenziele* werden dem Tabellenkörper hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/widgets/massnahmen_table.dart](#)

10.5. Aktualisierung des *ViewModels*

Die *Nebenziele* werden – erneut mit dem Datentyp `BuiltSet` – dem *ViewModel* hinzugefügt (Listing 10.7). Der benannte Konstruktor `seeded` initialisiert die Instanzvariable mit einer leeren Menge (Z. 20-21). 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.

```
18 final hauptzielsetzungLand =
19   BehaviorSubject<ZielsetzungLandChoice?>.seeded(null);
20 final nebenziele = BehaviorSubject<BuiltSet<ZielsetzungLandChoice>>.seeded(
21   BuiltSet<ZielsetzungLandChoice>());
```

Listing 10.7.: Die *Nebenziele* werden dem *ViewModel* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

10.5.1. Aktualisierung der *Setter*-Methode

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 10.8). 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).

```

46 set model(Massnahme model) {
47   guid.value = model.guid;
48
49   letzterStatus.value = letzterStatusChoices
50     .fromAbbreviation(model.letzteBearbeitung.letzterStatus);
51   massnahmenTitel.value = model.identifikatoren.massnahmenTitel;
52
53   {
54     final mc = model.massnahmenCharakteristika;
55
56     foerderklasse.value =
57       foerderklasseChoices.fromAbbreviation(mc.foerderklasse);
58     kategorie.value = kategorieChoices.fromAbbreviation(mc.kategorie);
59
60     zielflaeche.value = zielflaecheChoices.fromAbbreviation(mc.zielflaeche);
61     zieleinheit.value = zieleinheitChoices.fromAbbreviation(mc.zieleinheit);
62     hauptzielsetzungLand.value =
63       hauptzielsetzungLandChoices.fromAbbreviation(mc.hauptzielsetzungLand);
64     nebenziele.value = BuiltSet(mc.nebenziele
65       .map((n) => hauptzielsetzungLandChoices.fromAbbreviation(n)));
66   }
67 }

```

Listing 10.8.: Konvertierung des *Models* in das *ViewModel* für die *Nebenziele*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart](#)

10.5.2. Aktualisierung der *Getter*-Methode

Ähnlich verhält es sich bei der Umwandlung des *ViewModels* in das *Model* (Listing 10.9). 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` erhält zu diesem Zweck erneut eine anonyme Funktion, welche die Abkürzungen der Auswahloptionen abfragt (Z. 81). Die resultierende Menge wird als Argument dem Konstruktor `SetBuilder` übergeben. Der `SetBuilder` wiederum kümmert sich um das Bauen des `BuiltSet`, sobald ein Objekt des Typs `Massnahme` gebaut wird.

```

69 Massnahme get model => Massnahme((b) => b
70   ..guid = guid.value
71   ..letzteBearbeitung.letzterStatus = letzterStatus.value?.abbreviation
72   ..letzteBearbeitung.letztesBearbeitungsDatum = DateTime.now().toUtc()
73   ..identifikatoren.update((b) => b..massnahmenTitel = massnahmenTitel.value)
74   ..massnahmenCharakteristika.update((b) => b
75     ..foerderklasse = foerderklasse.value?.abbreviation
76     ..kategorie = kategorie.value?.abbreviation
77     ..zielflaeche = zielflaeche.value?.abbreviation
78     ..zieleinheit = zieleinheit.value?.abbreviation
79     ..hauptzielsetzungLand = hauptzielsetzungLand.value?.abbreviation
80     ..nebenziele =
81       SetBuilder(nebenziele.value.map((n) => n.abbreviation).toList()));

```

Listing 10.9.: Konvertierung des *ViewModels* in das *Model* für die *Nebenziele*, Quelle: Eigenes Listing, Datei: `Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_form_view_model.dart`

10.6. Aktualisierung der Selektionskarte

Die Selektionskarte wird um die Instanzvariable `multiSelection` erweitert (Listing 10.10, Z. 17), deren Wert im Konstruktor übergeben wird (Z. 27), aber auch ausgelassen werden kann, da der Standardwert `false` angegeben ist.

```

15 class SelectionCard<ChoiceType extends Choice> extends StatelessWidget {
16   final String title;
17   final bool multiSelection;
18   final BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel;
19   final Choices<ChoiceType> allChoices;
20   final BehaviorSubject<Set<Choice>> priorChoices;
21   final OnSelect<ChoiceType> onSelect;
22   final OnDeselect<ChoiceType> onDeselect;
23   final String? errorText;
24
25   SelectionCard(
26     {required this.title,
27     this.multiSelection = false,

```

Listing 10.10.: Die Klasse *SelectionCard* erhält die Instanzvariable *multiSelection*, Quelle: Eigenes Listing, Datei: `Quellcode/Schritt-6/conditional_form/lib/widgets/selection_card.dart`

10.7. Aktualisierung des Selektionsbildschirms

Die Rückruffunktion `onChanged` des `CheckboxListTile` unterscheidet schließlich zwischen Mehrfach- und Einfachauswahl (Listing 10.11). Sollte `multiSelection` mit `true` gesetzt sein (Z. 133), so erstellt die Methode `rebuild` des `BuiltSet` eine Kopie des aktuellen `ViewModels` der Selektionen 135-141. In der anonymen Funktion, welche für die Manipulationen an der Kopie genutzt wird, wird in einer Fallunterscheidung überprüft, ob das angeklickte Element bereits selektiert ist (Z. 136). Sollte das der Fall sein, so wird diese bereits selektierte Option, die nun abgewählt wurde, mit der Methode `remove` des `Builder`-Objektes aus dem `BuiltSet` entfernt (Z. 137). Anderenfalls war die Option nicht selektiert, weshalb sie mit der Methode `add` hinzugefügt wird (Z. 139).

```

124 return CheckboxListTile(
125   key: Key(
126     "valid choice ${allChoices.name} - ${c.abbreviation}"),
127   controlAffinity: ListTileControlAffinity.leading,
128   title: Text(c.description),
129   tileColor: selectedButDoesNotMatch ? Colors.red : null,
130   value: isSelected,
131   onChanged: (bool? selected) {
132     if (selected != null) {
133       if (multiSelection) {
134         selectionViewModel.value =
135           selectionViewModel.value.rebuild((b) {
136             if (selectionViewModel.value.contains(c)) {
137               b.remove(c);
138             } else {
139               b.add(c);
140             }
141           });
142       } else {
143         selectionViewModel.value =
144           selectionViewModel.value.rebuild((b) {
145             b.replace(isSelected ? [] : [c]);
146           });
147       }
148       if (selected) {
149         onSelect(c);
150       } else {
151         onDeselect(c);
152       }
153     }
154   });

```

Listing 10.11.: Dem `CheckboxListTile` wird die Mehrfachselektion hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/widgets/selection_card.dart](#)

10.8. Aktualisierung der Eingabemaske

Unterhalb des Auswahlfeldes für das Hauptziel wird die Selektionskarte für die Nebenziele eingefügt (Listing 10.12). Allerdings handelt es sich dieses Mal um ein Mehrfachauswahlfeld, weshalb eine neue Methode namens `buildMultiSelectionCard` aufgerufen wird (Z. 215-217).

```

212 buildSelectionCard(
213     allChoices: hauptzielsetzungLandChoices,
214     selectionViewModel: vm.hauptzielsetzungLand),
215 buildMultiSelectionCard(
216     allChoices: nebenzielsetzungLandChoices,
217     selectionViewModel: vm.nebenziele),

```

Listing 10.12.: Der Aufruf von `buildMultiSelectionCard` für die Menge `nebenzielsetzungLandChoices`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

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

10.8.1. Auslagerung der Validierungsfunktion

Innerhalb der bereits vorhandenen Methode `buildSelectionCard` wird die Routine, welche für die Validierung des Formulars genutzt wird, in die neue Methode `validateChoices` (Listing 10.13, 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 G in Listing G.1 auf Seite 203 zu finden.

```

119 Widget buildSelectionCard<ChoiceType extends Choice>(
120     {required Choices<ChoiceType> allChoices,
121     required BehaviorSubject<ChoiceType?> selectionViewModel}) {
122     return FormField(
123         validator: (_) => validateChoices(
124             name: allChoices.name,
125             choices: {
126                 if (selectionViewModel.value != null) selectionViewModel.value!
127             },
128             priorChoices: vm.priorChoices.value),
129     builder: (field) => SelectionCard<ChoiceType>(

```

Listing 10.13.: Die Methode `buildSelectionCard` mit dem Aufruf der ausgelagerten Funktion `validateChoices`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

10.8.2. Konfiguration der Selektionskarte für die Mehrfachauswahl

Für die Erstellung der Mehrfachauswahlfelder ist die Methode `buildMultiSelectionCard` zuständig (Listing 10.14). Das übergebene `selectionViewModel` unterstützt mit dem Typargument `BuiltSet` die Auswahl von mehreren Optionen (Z. 146). Bei `selectionViewModel` handelt es sich bereits um eine Menge. Für die Validierung (Z. 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 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 mithilfe von `add` hinzuzufügen (Z. 160) bzw. um 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 Liste mit den gewünschten Änderungen. Deshalb erfolgt eine Zuweisung der Kopie zum Wert des `BehaviorSubject`-Objektes `selectionViewModel`, was wiederum das Auslösen eines neuen Ereignisses bewirkt (Z. 158, 161).

```

144 Widget buildMultiSelectionCard<ChoiceType extends Choice>(
145     {required Choices<ChoiceType> allChoices,
146     required BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel}) {
147     return FormField(
148         validator: (_) => validateChoices(
149             name: allChoices.name,
150             choices: selectionViewModel.value,
151             priorChoices: vm.priorChoices.value),
152         builder: (field) => SelectionCard<ChoiceType>(
153             title: allChoices.name,
154             multiSelection: true,
155             allChoices: allChoices,
156             priorChoices: vm.priorChoices,
157             initialValue: selectionViewModel.value,
158             onSelect: (selectedChoice) => selectionViewModel.value =
159                 selectionViewModel.value
160                 .rebuild((b) => b.add(selectedChoice)),
161             onDeselect: (selectedChoice) => selectionViewModel.value =
162                 selectionViewModel.value
163                 .rebuild((b) => b.remove(selectedChoice)),
164             errorText: field.errorText,
165         ));
166 }

```

Listing 10.14.: Die Methode `buildMultiSelectionCard`, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

11. Schritt 7 - Benutzerdefinierte Validierungsfunktionen

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 *Nebenziel* gewählt wird. In diesem Fall kommen wiederum nur die Werte *keine Angabe/Vorgabe* oder *bitte um Unterstützung* infrage.

Sollte dagegen ein *Hauptziel* gesetzt sein, so darf das *Nebenziel* nicht die gleiche Option enthalten. Ist also beispielsweise für das *Hauptziel Biodiversität* ausgewählt (Abb. 11.1), so darf die Option im Selektionsbildschirm für die *Nebenziele* nicht zur Verfügung stehen (Abb. 11.2).



Abbildung 11.1.: Im Selektionsbildschirm für das *Hauptziel* ist *Biodiversität* ausgewählt, Quelle: Eigene Abbildung

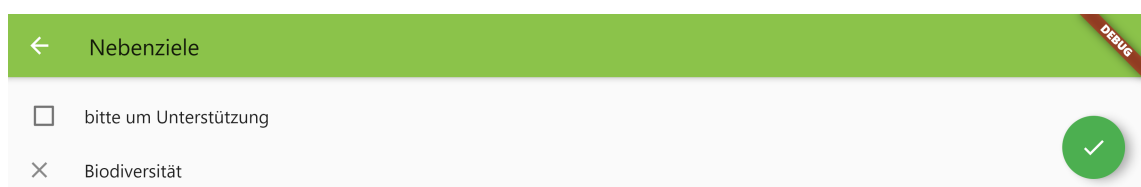


Abbildung 11.2.: Im Selektionsbildschirm für die *Nebenziele* kann *Biodiversität* nicht ausgewählt werden, Quelle: Eigene Abbildung

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. 11.3).

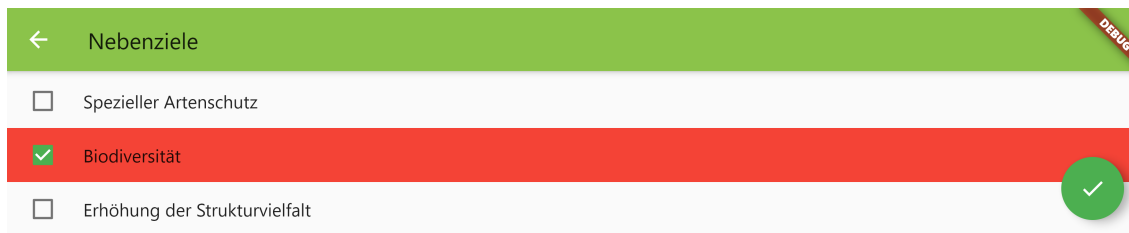


Abbildung 11.3.: Im Selektionsbildschirm für die *Nebenziele* wird die selektierte invalide Option *Biodiversität* rot gekennzeichnet, Quelle: Eigene Abbildung

11.1. Aktualisierung der Selektionskarte

Diese Bedingungen lassen sich nicht mit der 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 wurden, gibt aber keine Auskunft darüber, von welchem Auswahlfeld sie stammen. Sollte also die Auswahloption *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 11.1, Z. 31). Ein Parameter des gleichen Namens wird den Hilfsmethoden `buildSelectionCard` und `buildMultiSelectionCard` übergeben, welche ihn unverändert an den Konstruktor der Klasse `SelectionCard` weiterleiten. Die entsprechenden Listings sind in Anhang H auf den Seiten 205 und 206 zu finden.

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 Instanzvariablen `choiceMatcher` (Z. 46). Sollte der im Konstruktor übergebene Parameter nicht *null* sein, so wird er der Instanzvariablen zugewiesen. Ist er aber *null*, so sorgt die *if-null Expression* dafür, dass der Standardwert rechts von dem `??` zugewiesen wird: die Funktion `defaultChoiceMatcherStrategy`. Diese Funktion kapselt die Überprüfung der Abhängigkeiten – welche die Auswahloptionen 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


```

13 typedef ChoiceMatcher<ChoiceType extends Choice> = bool Function(
14     ChoiceType choice, Set<Choice> priorChoices);
15
16 bool defaultChoiceMatcherStrategy(Choice choice, Set<Choice> priorChoices) {
17     return choice.conditionMatches(priorChoices);
18 }
19
20 const confirmButtonTooltip = 'Auswahl übernehmen';
21
22 class SelectionCard<ChoiceType extends Choice> extends StatelessWidget {
23     final String title;
24     final bool multiSelection;
25     final BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel;
26     final Choices<ChoiceType> allChoices;
27     final BehaviorSubject<Set<Choice>> priorChoices;
28     final OnSelect<ChoiceType> onSelect;
29     final OnDeselect<ChoiceType> onDeselect;
30     final String? errorText;
31     final ChoiceMatcher<ChoiceType> choiceMatcher;
32
33     SelectionCard(
34         {required this.title,
35         this.multiSelection = false,
36         required Iterable<ChoiceType> initialValue,
37         required this.allChoices,
38         required this.priorChoices,
39         required this.onSelect,
40         required this.onDeselect,
41         ChoiceMatcher<ChoiceType>? choiceMatcher,
42         this.errorText,
43         Key? key})
44         : selectionViewModel = BehaviorSubject<BuiltSet<ChoiceType>>.seeded(
45             BuiltSet.from(initialValue)),
46           this.choiceMatcher = choiceMatcher ?? defaultChoiceMatcherStrategy,
47           super(key: key);

```

Listing 11.1.: Der *choiceMatcher* wird der Klasse *SelectionCard* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/widgets/selection_card.dart](#)

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.

11.1.1. Strategie-Entwurfsmuster

Das *Strategie*-Entwurfsmuster ist ein Verhaltensmuster der *Gang of Four*. Es erlaubt Algorithmen zu kapseln und auszutauschen¹. Abbildung 11.4 zeigt das *UML*-Diagramm des *Strategie*-Entwurfsmusters.

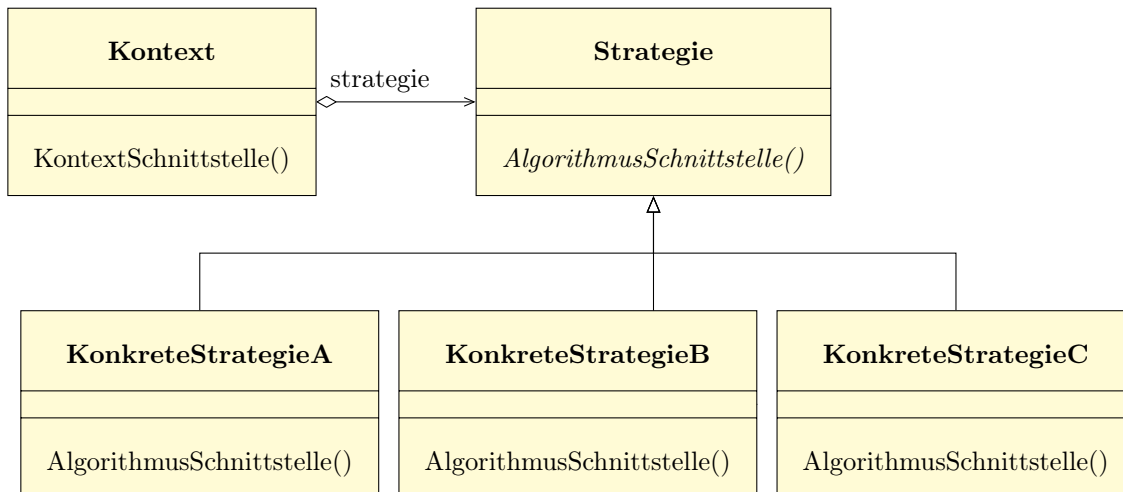


Abbildung 11.4.: *UML*-Diagramm des *Strategie*-Entwurfsmusters, Quelle: In Anlehnung an Gamma u. a. (*Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, S. 374)

Die Typdefinition `ChoiceMatcher` (Z. 13-14) 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 werden. Sollte der Parameter `choiceMatcher` gesetzt sein, so tauscht er die standardmäßig genutzte *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 11.5 zeigt das *UML*-Diagramm der konkreten Implementierung des *Strategie*-Entwurfsmusters für die *Strategie* `ChoiceMatcher`.

11.2. Aktualisierung der Eingabemaske

Im Diagramm ist ebenfalls der *View* `MassnahmenDetailScreen` enthalten, denn er verwendet die *konkrete Strategie* `defaultChoiceMatcherStrategy` für die Validierung (Listing 11.2). Sollte nämlich ein Argument für den Parameter `choiceMatcher` übergeben werden (Z. 122),

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

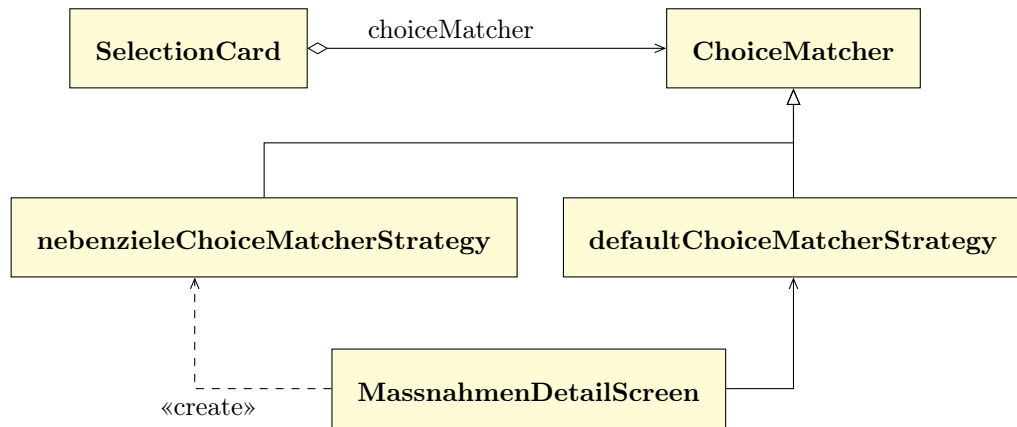


Abbildung 11.5.: UML-Diagramm des *Strategy*-Entwurfsmusters für den *ChoiceMatcher*, Quelle: Eigene Abbildung

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.

```

119 Widget buildSelectionCard<ChoiceType extends Choice>(
120   {required Choices<ChoiceType> allChoices,
121     required BehaviorSubject<ChoiceType?> selectionViewModel,
122     ChoiceMatcher<ChoiceType?> choiceMatcher}) {
123   return FormField(
124     validator: (_) => validateChoices(
125       name: allChoices.name,
126       choices: {
127         if (selectionViewModel.value != null) selectionViewModel.value!
128       },
129     priorChoices: vm.priorChoices.value,
130     choiceMatcher: choiceMatcher ?? defaultChoiceMatcherStrategy),
131     builder: (field) => SelectionCard<ChoiceType>(
132       title: allChoices.name,
133       allChoices: allChoices,
134       priorChoices: vm.priorChoices,
135       initialValue: {
136         if (selectionViewModel.value != null)
137           selectionViewModel.value!
138       },
139       choiceMatcher: choiceMatcher,
140       onSelect: (selectedChoice) =>
  
```

Listing 11.2.: Die Methode *buildSelectionCard* mit dem Parameter *choiceMatcher*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Außerdem erstellt *MassnahmenDetailScreen* die konkrete Strategie für die benutzerdefinierte Validierung der *Nebenziele*. Im UML-Diagramm wurde sie zum besseren Verständnis *nebenzieleChoiceMatcherStrategy* genannt. Im Listing 11.3 ist sie als anonyme Funktion zu sehen. Im Aufruf `buildMultiSelectionCard` für die *Nebenziele* wird sie dem Parameter `choiceMatcher` übergeben (Z. 224-238). In der ersten Fallunterscheidung wird überprüft, ob die gewählte Option ein tatsächliches *Nebenziel* ist (Z. 225). Das 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 es sich dagegen um ein tatsächliches *Nebenziel* 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. Sollte das *Hauptziel* keinen tatsächlichen Wert einer Zielsetzung enthalten, dann ist die Wahl eines oder mehrerer *Nebenziele* nicht sinnvoll. Waren beide bisherigen Bedingungen nicht wahr, so steht bereits fest, dass sowohl das *Hauptziel* als auch die *Nebenziele* gesetzt sind. Beide enthalten weder *keine Angabe/Vorgabe* noch *bitte um Unterstützung* als Wert. Nun soll eine letzte Fallunterscheidung überprüfen, ob das *Nebenziel* bereits im *Hauptziel* gesetzt ist (Z. 230-231). Das ist nicht erlaubt, weshalb `false` zurückgegeben werden soll (Z. 232). Anderenfalls sind alle Bedingungen erfüllt und `true` kann zurückgegeben werden (Z. 234).

An diesem Beispiel wird auch offensichtlich, welchen Nutzen die Generalisierung der Klasse *SelectionCard* über den Typparameter *ChoiceType* hat. Über eine Reihe von Methoden- und Konstruktoraufrufen gelangt das Typargument `ZielsetzungLandChoice` in die Klasse *SelectionCard* und somit auch zu der Instanzvariablen `choiceMatcher`. Zuerst wird es der Methode `buildMultiSelectionCard` übergeben (Z. 221). Mit dem Konstruktoraufruf `SelectionCard<ChoiceType>` wird es an die Selektionskarte weitergereicht (Siehe Listing H.2 in Zeile 157 in Anhang H auf Seite 206). Schließlich erhält die Instanzvariable das Typargument über die Deklaration `ChoiceMatcher<ChoiceType> choiceMatcher` (Listing 11.1, S. 161, 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 11.3, Z. 224) – um den Typ `ZielsetzungLandChoice`. Aus diesem Grund können die Methoden `hasRealValue` (Z. 225) und `hasNoRealValue` (Z. 228) auf dem Objekt `choice` aufgerufen werden, obwohl sie Teil der Klasse *ZielsetzungLandChoice*, aber nicht der Basisklasse *Choice* sind. Ohne Parametrisierung über den Typ müsste das Objekt `choice` in einen anderen Typ 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 Typarguments ist das Vorhandensein des richtigen Typs garantiert und keine Typumwandlung nötig.

```

221 buildMultiSelectionCard<ZielsetzungLandChoice>(
222   allChoices: nebenzielsetzungLandChoices,
223   selectionViewModel: vm.nebenziele,
224   choiceMatcher: (choice, priorChoices) {
225     if (choice.hasRealValue) {
226       if (vm.hauptzielsetzungLand.value == null ||
227         vm.hauptzielsetzungLand.value!
228           .hasNoRealValue) {
229         return false;
230       } else if (choice ==
231         vm.hauptzielsetzungLand.value) {
232         return false;
233       } else {
234         return true;
235       }
236     }
237     return true;
238   },
239 ),

```

Listing 11.3.: Der benutzerdefinierte *choiceMatcher* für die Menge *nebenzielsetzungLandChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

Die beiden neuen Methoden sind in Listing 11.4 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 den Wert (Z. 203).

```

185 class ZielsetzungLandChoice extends Choice {
186   static final ka = ZielsetzungLandChoice("ka", "keine Angabe/Vorgabe");
187   /* ... */
188   static final contact =
189     ZielsetzungLandChoice("contact", "bitte um Unterstützung");
190
201   bool get hasRealValue => this != ka && this != contact;
202
203   bool get hasNoRealValue => !hasRealValue;

```

Listing 11.4.: Die *Getter*-Methoden *hasRealValue* und *hasNoRealValue*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/choices/choices.dart](#)

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 11.5).

```

63 final validityChanged = priorChoices
64   .map((choices) =>
65     selectionViewModel.value.any((c) => !choiceMatcher(c, choices)))
66   .distinct();

```

Listing 11.5.: Der *Stream* *validityChanged* in Schritt 7, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/widgets/selection_card.dart](#)

Alle Vorkommnisse, die durch den neuen Ausdruck ersetzt werden, sind im Anhang H auf den Seiten 207 bis 209 zu finden.

Teil IV

FAZIT

12. Diskussion

12.1. 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 verwenden¹. 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, einen Mechanismus zu verwenden, der es erlaubt, nur Teile der Oberfläche neu zu zeichnen, die wirklich eine Aktualisierung benötigen.

Zu diesem Zweck empfiehlt Google das Paket *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 12.1 zu sehen ist. `ChoiceChangeNotifier`

```
1 class ChoiceChangeNotifier extends ChangeNotifier {  
2     BuiltSet<Choice> _choices = BuiltSet<Choice>();  
3  
4     BuiltSet<Choice> get choices => _choices;  
5  
6     set choices(BuiltSet<Choice> choices) {  
7         _choices = choices;  
8         notifyListeners();  
9     }  
10 }  
11 class LetzterStatusViewModel extends ChoiceChangeNotifier {}
```

Listing 12.1.: Verwendung der Klasse *ChangeNotifier*, Quelle: Eigenes Listing

verwaltet den internen privaten Zustand `_choices` (Z. 3) über die öffentlichen Schnittstellen

¹Vgl. Google LLC, *Flutter* / *Adding interactivity to your Flutter app*.

²Vgl. Google LLC, *Provider* / *A recommended approach*.

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 12.2, 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.

```

1 MultiProvider(
2   providers: [
3     Provider<MassnahmenFormViewModel>(create: (_) => MassnahmenFormViewModel()),
4     Provider<MassnahmenJsonFile>(create: (_) => MassnahmenJsonFile()),
5     Provider(
6       create: (context) => MassnahmenModel(
7         Provider.of<MassnahmenJsonFile>(context, listen: false)),
8     ChangeNotifierProvider(create: (context) => LetzterStatusViewModel()),
9   ],
10  child: MaterialApp(),
11 )

```

Listing 12.2.: Die *Widgets Provider*, *ChangeNotifierProvider* und *MultiProvider*, Quelle: Eigenes Listing

Dann ist der *ChangeNotifier* in dem *Widget*, welches dem Parameter `child` übergeben wird, und darüber hinaus in allen Kindelementen dieses *Widgets* verfügbar. Über einen `Consumer` kann in der Oberfläche auf Änderungen des *ChangeNotifier* reagiert werden (Listing 12.3).

```

1 Consumer<LetzterStatusViewModel>(
2   builder: (context, choiceChangeNotifier, child) {
3   },
4 )

```

Listing 12.3.: Das *Widget Consumer*, Quelle: Eigenes Listing

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

Damit unterstützt es ein breiteres Spektrum von Einsatzmöglichkeiten. Beispielsweise kann ein transformierter *Stream* übergeben werden, wie im Kapitel 8 gezeigt wird.

Die einzige fehlende Komponente dafür ist ein *Stream*, der den zuletzt übermittelten Wert speichert und allen neuen zuhörenden `StreamBuilder`-Elementen übermittelt. Deshalb wurde sich für das Package *RxDart* 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 Sullivan vorgestellt. Doch anstatt lediglich das `BehaviorSubject` für das `ViewModel` zu verwenden, sorgte die Präsentation durch den zusätzlichen – jedoch überflüssigen – Einsatz zweier weiterer `Stream`-Klassen für schwereres Verständnis (Listing 12.4)³.

```

1 class CartBloc{
2   final _cart = Cart();
3
4   Sink<Product> get addition => _additionalController.sink;
5
6   final _additionalController = StreamController<Product>();
7
8   Stream<int> get itemCount => _itemCountSubject.stream;
9
10  final _itemCountSubject = BehaviorSubject<int>();
11
12  CartBloc(){
13    _additionalController.stream.listen(_handle);
14  }
15
16  void _handle(Product product){
17    _cart.add(product);
18    _itemCountSubject.add(_cart.itemCount);
19  }
20 }

```

Listing 12.4.: Die Klasse `CartBloc`, Quelle: Google LLC, *Build reactive mobile apps with Flutter* (Google I/O '18) TC: 27:37

Der Quelltext setzt einige Grundlagen voraus. Der `Stream` ist als Teil der Datenübertragung zu identifizieren, der die Nachrichten empfängt. Über die Methode `listen` können die eintreffenden Ereignisse behandelt werden. Der `Sink` ist dagegen als der Teil zu sehen, welchem die Nachrichten zugestellt werden. Die Methode `add` erlaubt es, eine neue Nachricht zu senden. Ein `StreamController` verwaltet sowohl den Übermittler als auch den Empfänger und stellt beide über die *Getter*-Methoden `sink` und `stream` zur Verfügung. Durch den Einsatz von `BehaviorSubject` kann jedoch auf die drei Klassen verzichtet werden. Ein Objekt der Klasse `BehaviorSubject` vereint nicht nur das Senden und Empfangen von Nachrichten, sondern speichert überdies den zuletzt gesendeten Wert für neu dazukommende Zuhörer ab. Im Quelltext wurde ein Objekt des Typs `Sink` verwendet, um Ereignisse von dem `View` an das `ViewModel` senden zu können (Z. 4). Der dazugehörige `StreamController` wird in Zeile 6 erstellt. 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 bereitstellt (Z. 18). Er nutzt das

³Google LLC, *Build reactive mobile apps with Flutter* (Google I/O '18), TC: 27:37.

`BehaviorSubject` mit dem Namen `_itemCountSubject` (Z. 10), verwendet allerdings keine der für die Klasse einzigartigen Eigenschaften – wie zum Beispiel die *Getter*-Methode `value` für den zuletzt übermittelten Wert. Die Eigenschaften werden auch nicht anderen Klassen angeboten, da `_itemCountSubject` nicht öffentlich ist. Der *Stream* `itemCount` könnte genauso gut durch einen weiteren `StreamController` ersetzt werden.

Der gesamte Quellcode kann stark vereinfacht werden (Listing 12.5).

```

1 class CartBloc{
2   final _cart = BehaviorSubject<Cart>(seedValue: Cart());
3
4   addProduct(Product product) => _cart.value = _cart.value..add(product);
5
6   Stream<int> get itemCount => _cart.map((cart) => cart.itemCount);
7 }

```

Listing 12.5.: Die vereinfachte Klasse `CartBloc`, Quelle: Eigenes 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 auf den Wert, welcher durch das `BehaviorSubject` verwaltet wird, keinen Effekt, denn `Cart` ist ein Referenztyp und kein Wertetyp. Die Änderung an dem Wert war mit dem Methodenaufruf `..add(product)` bereits abgeschlossen, denn sie wurde am Original-Objekt durchgeführt, anstatt – wie im Falle eines Wertetypen – an einer Kopie des Objektes. Mit der Zuweisung `_cart.value = _cart.value` wird der *Getter*-Methode dieselbe Referenz zugewiesen, welche zuvor bereits gespeichert war. In diesem Fall wird sich zunutze gemacht, dass jede Zuweisung zur *Getter*-Methode `value` ein neues Ereignis auslöst, ungeachtet dessen, ob der Wert derselbe ist. Die Erstellung weiterer `StreamController` zum Senden der transitiven Eigenschaft `itemCount` ist nicht nötig. Sendet das `BehaviorSubject` `_cart` ein neues Ereignis (Z. 4), so wird auch die Methode `map` ausgelöst und ein transformiertes Ereignis gesendet (Z. 6).

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

12.2. Anzeige von fehlerhaften Teilkomponenten der Bedingungen von deaktivierten Auswahloptionen

Ein Wishkriterium für die Formularapplikation war es, bei der Auswahl von einer deaktivierten Option einen Hinweis zu erhalten, warum diese deaktiviert ist.

In Kapitel 8 ist die Umsetzung der Deaktivierung von Optionen beschrieben. Die Option validiert sich selbst und bekommt zu diesem Zweck eine Funktion übergeben. Diese Funktion überprüft die Kompatibilität mit allen anderen Feldern im Formular. Konjunktion, Disjunktion und Negation werden 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. Eine Lösung könnte die Nutzung eines *Und-Oder-Baums* sein. Diese Vorgehensweise könnte erlauben, die Bedingung einer Option in mehrere Teilbedingungen zu verzweigen und durch den dadurch aufgespannten Baum zu traversieren.

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

13. Schlussfolgerung-und-Ausblick

In dieser Arbeit wurde gezeigt, dass das Hauptproblem der Formularanwendung mithilfe 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 ein deklaratives Frontend-Framework ist. Die *Stream*-Transformationen aus der *Dart*-Standardbibliothek und aus *RxDart* haben ihre Äquivalente in der Bibliothek *RxJS*.

Die Wahl von *Flutter* für die Entwicklung war dennoch 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 Integrationstests stoppten die Entwicklung für einen Moment.

Hätte die Umsetzung in *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 Webportalen der Drittanbieter verstreut ist.

Auch das *Flutter*-Kochbuch bot die benötigten Rezepte für die Funktionalitäten wie etwa die Formularvalidierung und 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 12.1 beschrieben. Das ursprünglich von Google beworbene *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 Masterar-

beit bezüglich der reibungslosen Implementierung des Zustandsmanagements mit *RxDart* gesammelt wurden, sollen in Zukunft mit der *Flutter*-Community geteilt werden.

Das Wunschkriterium, dem 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.

Literatur

- Adobe Inc. *PhoneGap Docs | FAQ*. Aug. 2016. URL: <https://web.archive.org/web/20200806024626/http://docs.phonegap.com/phonegap-build/faq/> (besucht am 02.06.2021).
- *Update for Customers Using PhoneGap and PhoneGap Build*. Aug. 2020. URL: <https://web.archive.org/web/20200811121213/https://blog.phonegap.com/update-for-customers-using-phonegap-and-phonegap-build-cc701c77502c?gi=df435eca31bb> (besucht am 02.06.2021).
- Beck, Kent. *Test-driven development: by example*. Addison-Wesley Professional, 2003. URL: <https://archive.org/details/extremeprogrammi00beck/page/9>.
- Borenkraout, Matan. *Testing Library Docs | Native Testing Library Introduction*. Nov. 2020. URL: <https://web.archive.org/web/20210128142719/https://testing-library.com/docs/react-native-testing-library/intro/> (besucht am 02.06.2021).
- Bray, Brandon. *Async in 4.5: Worth the Await*. Apr. 2012. URL: <https://web.archive.org/web/20210702135551/https://devblogs.microsoft.com/dotnet/async-in-4-5-worth-the-await/> (besucht am 09.08.2021).
- Does redux-form work with React Native?* URL: <https://web.archive.org/web/20210602234346/https://redux-form.com/7.3.0/docs/faq/reactnative.md/> (besucht am 02.06.2021).
- Elliott, Conal und Paul Hudak. *Functional Reactive Animation*. In: *International Conference on Functional Programming*. 1997. URL: <http://conal.net/papers/icfp97/>.
- Facebook Inc. *The React Native Ecosystem*. URL: <https://web.archive.org/web/20210602191504/https://github.com/facebook/react-native/blob/d48f7ba748a905818e8c64fe70fe5b24aa098b05/ECOSYSTEM.md> (besucht am 02.06.2021).
- Formik Docs | React Native*. URL: https://web.archive.org/web/20210507005917if_/https://formik.org/docs/guides/react-native (besucht am 02.06.2021).
- Formik Docs API | <Formik />*. URL: https://web.archive.org/web/20210409184616if_/https://formik.org/docs/api/formik (besucht am 02.06.2021).
- Fowler, Martin. *Inversion of Control Containers and the Dependency Injection pattern*. Jan. 2004. URL: <http://web.archive.org/web/20210707041912/https://martinfowler.com/articles/injection.html>.
- *InversionOfControl*. Juni 2005. URL: <http://web.archive.org/web/20050628234825/https://martinfowler.com/bliki/InversionOfControl.html>.
- Gamma, Erich u. a. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Pearson Deutschland GmbH, 2009.

- GitHub-Nutzer nt4f04uNd. *GitHub | dart-lang | mockito | Mocks are not generated not in test folder*. URL: <http://web.archive.org/web/20210830132721/https://github.com/dart-lang/mockito/issues/429> (besucht am 30.08.2021).
- Google LLC. *Build a form with validation*. URL: <https://web.archive.org/web/20210122020924/https://flutter.dev/docs/cookbook/forms/validation> (besucht am 02.06.2021).
- *Build reactive mobile apps with Flutter (Google I/O '18)*. Juni 2021. URL: <https://youtu.be/RS36gBEp80I?t=1657> (besucht am 10.05.2018).
 - *built_value Changelog*. URL: https://web.archive.org/web/20210226045401/https://pub.dev/packages/built_value/changelog#680 (besucht am 29.08.2021).
 - *built_value_generator*. URL: https://web.archive.org/web/20210812042530/https://pub.dev/packages/built_value_generator (besucht am 28.08.2021).
 - *Dart - Language tour - Named constructors*. URL: <https://web.archive.org/web/20210726120223/https://dart.dev/guides/language/language-tour#named-constructors> (besucht am 29.08.2021).
 - *Dart | Effective Dart | Style | PREFER using _, __, etc. for unused callback parameters*. URL: https://web.archive.org/web/20210728114518/https://dart.dev/guides/language/effective-dart/style#prefer-using-_-__-etc-for-unused-callback-parameters (besucht am 08.08.2021).
 - *Dart | Language tour | spread operator*. Juli 2021. URL: <https://web.archive.org/web/20210625070139/https://dart.dev/guides/language/language-tour#spread-operator> (besucht am 08.07.2021).
 - *Dart Programming Language Specification 5th edition*. Apr. 2021. URL: <https://web.archive.org/web/20210702071617/https://dart.dev/guides/language/specifications/DartLangSpec-v2.10.pdf>.
 - *Dart: The platforms*. URL: <https://web.archive.org/web/20210719180726/https://dart.dev/overview#platform> (besucht am 09.08.2021).
 - *Flutter | Adding interactivity to your Flutter app*. URL: <https://web.archive.org/web/20210603051020/https://flutter.dev/docs/development/ui/interactive> (besucht am 16.08.2021).
 - *Flutter | Beautiful native apps in record time*. URL: <https://web.archive.org/web/20210630233338/https://flutter.dev/> (besucht am 30.06.2021).
 - *Flutter | Desktop support for Flutter*. URL: <https://web.archive.org/web/20210531034514/http://flutter.dev/desktop/> (besucht am 28.06.2021).
 - *Flutter | Hot reload*. URL: <http://web.archive.org/web/20210811184729/https://flutter.dev/docs/development/tools/hot-reload> (besucht am 28.08.2021).
 - *Flutter | InheritedWidget class*. URL: <http://web.archive.org/web/20210813022207/https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html> (besucht am 27.08.2021).
 - *Flutter | Introduction to widgets*. URL: <http://web.archive.org/web/20210603081649/https://flutter.dev/docs/development/ui/widgets-intro> (besucht am 03.06.2021).

- *Flutter | JSON and serialization*. URL: <https://web.archive.org/web/20210723055649/https://flutter.dev/docs/development/data-and-backend/json> (besucht am 22.08.2021).
 - *Flutter | StatelessWidget class*. URL: <http://web.archive.org/web/20210531051546/https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html> (besucht am 27.08.2021).
 - *Flutter | Web support for Flutter*. URL: <http://web.archive.org/web/20210506012158/https://flutter.dev/web> (besucht am 06.05.2021).
 - *Flutter | where method*. URL: <http://web.archive.org/web/20210829164822/https://api.flutter.dev/flutter/dart-core/Iterable/where.html> (besucht am 29.08.2021).
 - *Flutter Docs Cookbook | Forms*. URL: <https://web.archive.org/web/20201102003629/https://flutter.dev/docs/cookbook/forms> (besucht am 02.06.2021).
 - *Google Trends-Hilfe | Häufig gestellte Fragen zu Google Trends-Daten*. URL: <https://web.archive.org/web/20210813173858/https://support.google.com/trends/answer/4365533> (besucht am 29.08.2021).
 - *Provider | A recommended approach*. URL: <https://web.archive.org/web/20210729143240/https://flutter.dev/docs/development/data-and-backend/state-mgmt/options#provider> (besucht am 16.08.2021).
 - *source_gen*. URL: https://web.archive.org/web/20210812103702/https://pub.dev/packages/source_gen (besucht am 28.08.2021).
- Gosling, James u. a. *The Java[®] Language Specification Java SE 16 Edition*. Feb. 2021. URL: <https://web.archive.org/web/20210514051033/https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf>.
- Gossman, John. *Introduction to Model/View/ViewModel pattern for building WPF apps*. Okt. 2005. URL: <https://web.archive.org/web/20101103111603/http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>.
- JetBrains s.r.o. *Kotlin | High-order functions and lambdas | Underscore for unused variables*. URL: http://web.archive.org/web/20210331062820if_/https://kotlinlang.org/docs/lambdas.html#underscore-for-unused-variables (besucht am 08.08.2021).
- Johnson, Ralph E und Brian Foote. *Designing reusable classes*. In: *Journal of object-oriented programming* 1.2 (1988), S. 22–35.
- Johr, Alexander. *GitHub | dart-lang | mockito | Antwort auf “Mocks are not generated not in test folder”*. URL: <http://web.archive.org/web/20210830132721/https://github.com/dart-lang/mockito/issues/429#issuecomment-874963390> (besucht am 30.08.2021).
- Leach, Paul J., Rich Salz und Michael H. Mealling. *A Universally Unique Identifier (UUID) URN Namespace*. RFC 4122. Juli 2005. DOI: [10.17487/RFC4122](https://doi.org/10.17487/RFC4122). URL: <https://rfc-editor.org/rfc/rfc4122.txt>.
- Lynch, Max. *The Last Word on Cordova and PhoneGap*. März 2014. URL: <https://web.archive.org/web/20210413012559/https://blog.ionicframework.com/what-is-cordova-phonegap/> (besucht am 02.06.2021).

- MDN contributors. *MDN | async function - JavaScript*. Juni 2021. URL: https://web.archive.org/web/20210608034309/https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/async_function (besucht am 06.08.2021).
- MDN | *Promise - JavaScript*. Mai 2021. URL: https://web.archive.org/web/20210516053958/https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise (besucht am 06.08.2021).
- Nystrom, Bob. *Dart | Understanding null safety | Type promotion on null checks*. Juli 2020. URL: <https://web.archive.org/web/20210813113615/https://dart.dev/null-safety/understanding-null-safety#type-promotion-on-null-checks> (besucht am 21.08.2021).
- React Hook Form - API | register*. URL: <https://web.archive.org/web/20210406032209/https://react-hook-form.com/api/useform/register> (besucht am 02.06.2021).
- React Hook Form - Get Started*. URL: https://web.archive.org/web/20210523042601if_/https://react-hook-form.com/get-started/ (besucht am 02.06.2021).
- Redux Form - API | reduxForm*. URL: <https://web.archive.org/web/20210506221401/https://redux-form.com/7.4.2/docs/api/reduxform.md/#-code-validate-values-object-props-object-gt-errors-object-code-optional-> (besucht am 02.06.2021).
- Spolsky, Joel. *How Hard Could It Be?: The Unproven Path*. In: *inc.com* (Nov. 2008). URL: <http://web.archive.org/web/20081108094045/http://www.inc.com/magazine/20081101/how-hard-could-it-be-the-unproven-path.html> (besucht am 02.06.2021).
- Stack Exchange, Inc. *Stack Overflow Insights | Stack Overflow Annual Developer Survey*. URL: <http://web.archive.org/web/20210815173513/https://insights.stackoverflow.com/survey> (besucht am 29.08.2021).
- Thomsen, Michael. *Announcing Dart 2.12*. März 2021. URL: <https://medium.com/dartlang/announcing-dart-2-12-499a6e689c87> (besucht am 28.08.2021).

Teil V

ANHANG

A. Minimalistische *Flutter*-Formularanwendung

```
1 final emailPattern = RegExp(  
2   r'^(([^<>()\\[\]\\. ,;:\s@"]+(\.[^<>()\\[\]\\. ,;:\s@"]+)*)|("[\s@"]+))@(\[[0-9]{1,3}\. _]  
3   ↪ [0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\]|([a-zA-Z-0-9]+\.[a-zA-Z]{2,}))$';  
4  
5 String? validateEmail(String label, String? value) {  
6   if (value == null || value.isEmpty) {  
7     return '$label is required';  
8   } else if (!emailPattern.hasMatch(value)) {  
9     return 'Invalid Email Format';  
10  } else {  
11    return null;  
12  }  
13  
14 String? validateNotEmpty(String label, String? value) {  
15   if (value == null || value.isEmpty) {  
16     return '$label is required';  
17   } else {  
18     return null;  
19   }  
20 }
```

Listing A.1.: Validierungs-Funktionen der minimalistischen *Flutter*-Formularanwendung, Quelle: Eigenes Listing, Datei: [Quellcode/Vergleich/form-in-flutter/lib/validation.dart](#)

```

6  void main() => runApp(MyApp());
7
8  class MyApp extends StatelessWidget {
9    @override
10   Widget build(BuildContext context) => MaterialApp(
11     home: Scaffold(
12       body: MyForm(),
13     ),
14   );
15 }
16
17 class MyForm extends StatelessWidget {
18   final formData = Map<String,String>();
19   final _formKey = GlobalKey<FormState>();
20
21   @override
22   Widget build(BuildContext context) => Form(
23     key: _formKey,
24     child: Padding(
25       padding: const EdgeInsets.all(8.0),
26       child: Column(
27         children: [
28           Padding(
29             padding: const EdgeInsets.all(30.0),
30             child: Image(
31               image: AssetImage('assets/logo_flutter.png'), height: 100),
32           ),
33           Padding(
34             padding: const EdgeInsets.all(8.0),
35             child: Text("Form in Flutter", style: TextStyle(fontSize: 30)),
36           ),
37           TextFormField(
38             decoration: const InputDecoration(labelText: "Name"),
39             validator: (String? value) => validateNotEmpty("Name", value),
40             onChanged: (value) => formData["name"] = value),
41           TextFormField(
42             decoration: const InputDecoration(labelText: "Email"),
43             validator: (String? value) => validateEmail("Name", value),
44             onChanged: (value) => formData["email"] = value),
45           TextFormField(
46             decoration: const InputDecoration(labelText: "Password"),
47             validator: (String? value) =>
48               validateNotEmpty("Password", value),
49             onChanged: (value) => formData["password"] = value),
50           Padding(
51             padding: const EdgeInsets.symmetric(vertical: 16.0),
52             child: ElevatedButton(
53               onPressed: () {
54                 if (_formKey.currentState!.validate()) {
55                   ScaffoldMessenger.of(context).showSnackBar(
56                     SnackBar(content: Text(jsonEncode(formData))));
57                 }
58               },
59               child: Text('Submit'),
60             ),
61           ),
62         ],
63       ),
64     ),
65   );
66 }

```

Listing A.2.: Haupteinstiegspunkt der minimalistischen *Flutter*-Formularanwendung, Quelle: Eigenes Listing, Datei: [Quellcode/Vergleich/form-in-flutter/lib/main.dart](#)

B. Minimalistische *React Native*-Formularanwendung

```
21 type FormData = {
22   name: string;
23   email: string;
24   password: string;
25 };
26
27 export default () => {
28   const { handleSubmit, register, setValue, errors } = useForm<FormData>();
29
30   const onSubmit = (data: FormData) => {
31     Alert.alert('data', JSON.stringify(data));
32   };
33
34   return (
35     <KeyboardAwareScrollView
36       contentContainerStyle={styles.container}
37       style={{ backgroundColor: '#181e34' }}>
38       <Hero />
39       <View style={styles.formContainer}>
40         <Form {...{ register, setValue, validation, errors }}>
41           <Input name="name" label="Name" />
42           <Input name="email" label="Email" />
43           <Input name="password" label="Password" secureTextEntry={true} />
44           <Button title="Submit" onPress={handleSubmit(onSubmit)} />
45         </Form>
46       </View>
47     </KeyboardAwareScrollView>
48   );
49 };
50
51 const styles = StyleSheet.create({
52   container: {
53     flex: 1,
54     justifyContent: 'center',
55     paddingTop: Constants.statusBarHeight,
56     backgroundColor: '#181e34',
57   },
58   formContainer: {
59     padding: 8,
60     flex: 1,
61   },
62   button: {
63     backgroundColor: 'red',
64   },
65 });
```

Listing B.1.: Haupteinstiegspunkt der minimalistischen *React Native*-Formularanwendung, Quelle: <http://dev.to/elazizyoussouf/forms-in-react-native-the-right-way-4d46>, Datei: Quellcode/Vergleich/form-in-react-native-the-right-way/App.tsx

```

4 export default () => {
5   return (
6     <View style={styles.container}>
7       <Image style={styles.logo} source={require('./assets/hero.jpg')} />
8       <Text style={styles.paragraph}>
9         Form in React Native, The right Way!
10      </Text>
11    </View>
12  );
13 }
14
15 const styles = StyleSheet.create({
16   container: {
17     justifyContent: 'center',
18     flex:1,
19   },
20   paragraph: {
21     margin: 24,
22     marginTop: 0,
23     fontSize: 34,
24     fontWeight: 'bold',
25     textAlign: 'center',
26     color: '#FFF'
27   },
28   logo: {
29     width: '100%',
30     height: 200
31   }
32 });

```

Listing B.2.: Logo der minimalistischen *React Native*-Formularanwendung, Quelle: <https://dev.to/elazizyoussouf/forms-in-react-native-the-right-way-4d46>, Datei: [Quellcode/Vergleich/form-in-react-native-the-right-way/Hero.tsx](#)

```

1 export default {
2   name: {required: {value: true, message: 'Name is required'}},
3   email: {
4     required: {value: true, message: 'Email is required'},
5     pattern: {
6       value: /^[^<>()\\[\]\\. ,;:\s@"]+(\.[^<>()\\[\]\\. ,;:\s@"]+)*|(".*")@ _]
7         ↪ ((\\[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3})|([a-zA-Z\\-0-9]+\\. _]
8         ↪ )+[a-zA-Z]{2,}))$/,
9       message: 'Invalid Email Format',
10    },
11  },
12  password: {
13    required: {value: true, message: 'Password is required'},
14  },
15 };

```

Listing B.3.: Validierungs-Funktionen der minimalistischen *React Native*-Formularanwendung, Quelle: <https://dev.to/elazizyoussouf/forms-in-react-native-the-right-way-4d46>, Datei: [Quellcode/Vergleich/form-in-react-native-the-right-way/validation.tsx](#)

```

5  interface ValidationMap {
6    [key: string]: ValidationOptions;
7  }
8
9  interface ErrorMap {
10   [key: string]: FieldError | undefined;
11 }
12
13 interface Props {
14   children: JSX.Element | JSX.Element[];
15   register: (
16     field: { name: string },
17     validation?: ValidationOptions
18   ) => void;
19   errors: ErrorMap;
20   validation: ValidationMap;
21   setValue: (name: string, value: string, validate?: boolean) => void;
22 }

```

Listing B.4.: Schnittstellen *Props* der minimalistischen *React Native*-Formularanwendung, Quelle: <http://dev.to/elaziziyoussouf/forms-in-react-native-the-right-way-4d46>, Datei: [Quellcode/Vergleich/form-in-react-native-the-right-way/components/Form.tsx](#)

```

24 export default ({
25   register,
26   errors,
27   setValue,
28   validation,
29   children,
30 }: Props) => {
31   const Inputs = React.useRef<Array<TextInput>>([]);
32
33   React.useEffect(() => {
34     (Array.isArray(children) ? [...children] : [children]).forEach((child) => {
35       if (child.props.name)
36         register({ name: child.props.name }, validation[child.props.name]);
37     });
38   }, [register]);
39
40   return (
41     <>
42       {(Array.isArray(children) ? [...children] : [children]).map(
43         (child, i) => {
44           return child.props.name
45             ? React.createElement(child.type, {
46               ...{
47                 ...child.props,
48                 ref: (e: TextInput) => {
49                   Inputs.current[i] = e;
50                 },
51                 onChangeText: (v: string) =>
52                   setValue(child.props.name, v, true),
53                 onSubmitEditing: () => {
54                   Inputs.current[i + 1]
55                     ? Inputs.current[i + 1].focus()
56                     : Inputs.current[i].blur();
57               },
58               //onBlur: () => triggerValidation(child.props.name),
59               blurOnSubmit: false,
60               //name: child.props.name,
61               error: errors[child.props.name],
62             },
63             )
64           : child;
65         }
66       )}
67     </>
68   );
69 };

```

Listing B.5.: Form-Komponente der minimalistischen React Native-Formularanwendung, Quelle: <https://dev.to/elaziziyoussouf/forms-in-react-native-the-right-way-4d46>, Datei: Quellcode/Vergleich/form-in-react-native-the-right-way/components/Form.tsx

```

19 export default React.forwardRef<any, Props>(
20   (props, ref): React.ReactElement => {
21     const { label, labelStyle, error, ...inputProps } = props;
22
23     return (
24       <View style={styles.container}>
25         {label && <Text style={[styles.label, labelStyle]}>{label}</Text>}
26         <TextInput
27           autoCapitalize="none"
28           ref={ref}
29           style={[styles.input, { borderColor: error ? '#fc6d47' : '#c0cbd3' }]}
30           {...inputProps}
31         />
32         <Text style={styles.textError}>{error && error.message}</Text>
33       </View>
34     );
35   }
36 );
37
38 const styles = StyleSheet.create({
39   container: {
40     marginVertical: 8,
41   },
42   input: {
43     borderStyle: 'solid',
44     borderWidth: 1,
45     borderRadius: 5,
46     paddingVertical: 5,
47     paddingLeft: 5,
48     fontSize: 16,
49     height: 40,
50     color: '#c0cbd3',
51   },
52   label: {
53     paddingVertical: 5,
54     fontSize: 16,
55     fontWeight: 'bold',
56     color: '#c0cbd3',
57   },
58   textError: {
59     color: '#fc6d47',
60     fontSize: 14,
61   },
62 });

```

Listing B.6.: *Input-Komponente der minimalistischen React Native-Formularanwendung*, Quelle: <https://dev.to/elaziziyousseuf/forms-in-react-native-the-right-way-4d46>, Datei: [Quellcode/Vergleich/form-in-react-native-the-right-way/components/Input.tsx](#)

C. Schritt 1 Anhang

```
48 test('Storage with one Massnahme deserialises without error', () {
49   var json = {
50     "massnahmen": [
51       {
52         "guid": "test massnahme id",
53         "letzteBearbeitung": {
54           "letztesBearbeitungsDatum": 0,
55           "letzterStatus": "bearb"
56         },
57         "identifikatoren": {"massnahmenTitel": "Massnahme 1"}
58       }
59     ]
60   };
61
62   var expectedStorage = Storage();
63   expectedStorage =
64     expectedStorage.rebuild((b) => b.massnahmen.add(Massnahme((b) => b
65       ..guid = "test massnahme id"
66       ..identifikatoren.massnahmenTitel = "Massnahme 1"
67       ..letzteBearbeitung.update((b) {
68         b.letztesBearbeitungsDatum =
69           DateTime.fromMillisecondsSinceEpoch(0, isUtc: true);
70       })))));
71
72   var actualStorage = serializers.deserializeWith(Storage.serializer, json);
73
74   expect(actualStorage, equals(expectedStorage));
```

Listing C.1.: Unittest der Deserialisierung der Maßnahmenliste, Quelle: Eigenes Listing, Datei: [Quelldatei/Schritt-1/conditional_form/test/data_model/storage_test.dart](#)

D. Schritt 2 Anhang

```
5 class FoerderklasseChoice extends Choice {
6   static final oelb = FoerderklasseChoice("oelb", "Ökolandbau");
7   static final azl = FoerderklasseChoice("azl", "Ausgleichszulage");
8   static final ea = FoerderklasseChoice("ea", "Erschwerenausgleich");
9   static final aukm_nur_vns = FoerderklasseChoice("aukm_nur_vns",
10    "Agrarumwelt-(und Klima)Maßnahme: nur Vertragsnaturschutz");
11  static final aukm_ohne_vns = FoerderklasseChoice("aukm_ohne_vns",
12    "Agrarumwelt-(und Klima)Maßnahmen, tw. auch mit Tierwohlaspekten, aber OHNE
13    ↳ Vertragsnaturschutz");
14  static final twm_ziel = FoerderklasseChoice(
15    "twm_ziel", "Tierschutz/Tierwohlmaßnahmen mit diesem als Hauptziel");
16  static final contact =
17    FoerderklasseChoice("contact", "bitte um Unterstützung");
18  FoerderklasseChoice(String abbreviation, String description,
19    {bool Function(Set<Choice> choices)? condition})
20    : super(abbreviation, description);
21 }
22
23 final foerderklasseChoices = Choices<FoerderklasseChoice>({
24   FoerderklasseChoice.oelb,
25   FoerderklasseChoice.azl,
26   FoerderklasseChoice.ea,
27   FoerderklasseChoice.aukm_nur_vns,
28   FoerderklasseChoice.aukm_ohne_vns,
29   FoerderklasseChoice.twm_ziel,
30   FoerderklasseChoice.contact
31 }, name: "Förderklasse");
```

Listing D.1.: Die Menge *foerderklasseChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/choices/choices.dart](#)

```

33 class KategorieChoice extends Choice {
34     static final zf_us =
35         KategorieChoice("zf_us", "Anbau Zwischenfrucht/Untersaat");
36     static final anlage_pflege =
37         KategorieChoice("anlage_pflege", "Anlage/Pflege Struktur");
38     static final dungmang = KategorieChoice("dungmang", "Düngemanagement");
39     static final extens = KategorieChoice("extens", "Extensivierung");
40     static final flst = KategorieChoice("flst", "Flächenstilllegung/Brache");
41     static final umwandlg = KategorieChoice("umwandlg", "Nutzungsumwandlung");
42     static final bes_kult_rass = KategorieChoice(
43         "bes_kult_rass", "Förderung bestimmter Rassen / Sorten / Kulturen");
44     static final contact = KategorieChoice("contact", "bitte um Unterstützung");
45
46     KategorieChoice(String abbreviation, String description)
47         : super(abbreviation, description);
48 }
49
50 final kategorieChoices = Choices<KategorieChoice>({
51     KategorieChoice.zf_us,
52     KategorieChoice.anlage_pflege,
53     KategorieChoice.dungmang,
54     KategorieChoice.extens,
55     KategorieChoice.flst,
56     KategorieChoice.umwandlg,
57     KategorieChoice.bes_kult_rass,
58     KategorieChoice.contact
59 }, name: "Kategorie");

```

Listing D.2.: Die Menge *kategorieChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/choices/choices.dart](#)

```

61 class ZielflaecheChoice extends Choice {
62   static final ka = ZielflaecheChoice("ka", "keine Angabe/Vorgabe");
63   static final al = ZielflaecheChoice("al", "AL");
64   static final gl = ZielflaecheChoice("gl", "GL");
65   static final lf = ZielflaecheChoice("lf", "LF");
66   static final dk_sk = ZielflaecheChoice("dk_sk", "DK/SK");
67   static final hff = ZielflaecheChoice("hff", "HFF");
68   static final biotop_le =
69     ZielflaecheChoice("biotop_le", "Landschaftselement/Biotop o.Ä.");
70   static final wald = ZielflaecheChoice("wald", "Wald/Forst");
71   static final contact = ZielflaecheChoice("contact", "bitte um Unterstützung");
72
73   ZielflaecheChoice(String abbreviation, String description)
74     : super(abbreviation, description);
75 }
76
77 final zielflaecheChoices = Choices<ZielflaecheChoice>({
78   ZielflaecheChoice.ka,
79   ZielflaecheChoice.al,
80   ZielflaecheChoice.gl,
81   ZielflaecheChoice.lf,
82   ZielflaecheChoice.dk_sk,
83   ZielflaecheChoice.hff,
84   ZielflaecheChoice.biotop_le,
85   ZielflaecheChoice.wald,
86   ZielflaecheChoice.contact
87 }, name: "Zielfläche");
88
89 class ZieleinheitChoice extends Choice {
90   static final ka = ZieleinheitChoice("ka", "keine Angabe/Vorgabe");
91   static final m3 = ZieleinheitChoice("m3", "m³ (z.B. Gülle)");
92   static final pieces =
93     ZieleinheitChoice("pieces", "Kopf/Stück (z.B. Tiere oder Bäume)");
94   static final gve = ZieleinheitChoice("gve", "GV/GVE");
95   static final rgve = ZieleinheitChoice("rgve", "RGV");
96   static final ha = ZieleinheitChoice("ha", "ha");
97   static final contact = ZieleinheitChoice("contact", "bitte um Unterstützung");
98
99   ZieleinheitChoice(String abbreviation, String description)
100     : super(abbreviation, description);
101 }
102
103 final zieleinheitChoices = Choices<ZieleinheitChoice>({
104   ZieleinheitChoice.ka,
105   ZieleinheitChoice.m3,
106   ZieleinheitChoice.pieces,
107   ZieleinheitChoice.gve,
108   ZieleinheitChoice.rgve,
109   ZieleinheitChoice.ha,
110   ZieleinheitChoice.contact
111 }, name: "Zieleinheit");

```

Listing D.3.: Die Mengen *zielflaecheChoices* und *zieleinheitChoices*, Quelle: Eigenes Listing, Datei: [Quel
lcode/Schritt-2/conditional_form/lib/choices/choices.dart](#)

```

113 class ZielsetzungLandChoice extends Choice {
114   static final ka = ZielsetzungLandChoice("ka", "keine Angabe/Vorgabe");
115   static final bsch = ZielsetzungLandChoice("bsch", "Bodenschutz");
116   static final wsch = ZielsetzungLandChoice("wsch", "Gewässerschutz");
117   static final asch = ZielsetzungLandChoice("asch", "Spezieller Artenschutz");
118   static final biodiv = ZielsetzungLandChoice("biodiv", "Biodiversität");
119   static final struktkviel =
120     ZielsetzungLandChoice("struktkviel", "Erhöhung der Strukturvielfalt");
121   static final genet_res = ZielsetzungLandChoice("genet_res",
122     "Erhaltung genetischer Ressourcen (Pflanzen, z. B. im Grünland, und Tiere, z. B.
123     ↳ bedrohte Rassen)");
124   static final tsch = ZielsetzungLandChoice(
125     "tsch", "Tierschutz/Maßnahmen zum Tierwohl im Betrieb");
126   static final klima = ZielsetzungLandChoice("klima", "Klima");
127   static final contact =
128     ZielsetzungLandChoice("contact", "bitte um Unterstützung");
129   ZielsetzungLandChoice(String abbreviation, String description)
130     : super(abbreviation, description);
131 }
132
133 final _zielsetzungLandChoices = {
134   ZielsetzungLandChoice.ka,
135   ZielsetzungLandChoice.bsch,
136   ZielsetzungLandChoice.wsch,
137   ZielsetzungLandChoice.asch,
138   ZielsetzungLandChoice.biodiv,
139   ZielsetzungLandChoice.struktkviel,
140   ZielsetzungLandChoice.genet_res,
141   ZielsetzungLandChoice.tsch,
142   ZielsetzungLandChoice.klima,
143   ZielsetzungLandChoice.contact
144 };
145
146 final hauptzielsetzungLandChoices = Choices<ZielsetzungLandChoice>(
147   _zielsetzungLandChoices,
148   name: "Hauptzielsetzung Land");

```

Listing D.4.: Die Menge *hauptzielsetzungLandChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-2/conditional_form/lib/choices/choices.dart](#)

E. Schritt 3 Anhang

```
35 void saveRecord() {  
36   ScaffoldMessenger.of(context)  
37     ..hideCurrentSnackBar()  
38     ..showSnackBar(  
39       const SnackBar(content: Text('Massnahme wird gespeichert ...')));  
40  
41   model.putMassnahmeIfAbsent(vm.model);  
42 }
```

Listing E.1.: Die Methode *saveRecord*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-3/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

F. Schritt 4 Anhang

```
33 class KategorieChoice extends Choice {
34     static final zf_us = KategorieChoice(
35         "zf_us", "Anbau Zwischenfrucht/Untersaat",
36         condition: (choices) =>
37             choices.contains(FoerderklasseChoice.aukm_ohne_vns));
38     static final anlage_pflege = KategorieChoice(
39         "anlage_pflege", "Anlage/Pflege Struktur",
40         condition: (choices) =>
41             choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
42             choices.contains(FoerderklasseChoice.aukm_ohne_vns));
43     static final dungmang = KategorieChoice("dungmang", "Düngemanagement",
44         condition: (choices) =>
45             choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
46             choices.contains(FoerderklasseChoice.aukm_ohne_vns));
47     static final extens = KategorieChoice("extens", "Extensivierung");
48     static final flst = KategorieChoice("flst", "Flächenstilllegung/Brache",
49         condition: (choices) =>
50             choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
51             choices.contains(FoerderklasseChoice.aukm_ohne_vns));
52     static final umwandlg = KategorieChoice("umwandlg", "Nutzungsumwandlung",
53         condition: (choices) =>
54             choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
55             choices.contains(FoerderklasseChoice.aukm_ohne_vns));
56     static final bes_kult_rass = KategorieChoice(
57         "bes_kult_rass", "Förderung bestimmter Rassen / Sorten / Kulturen",
58         condition: (choices) => !choices.contains(FoerderklasseChoice.ea));
59     static final contact = KategorieChoice("contact", "bitte um Unterstützung");
60
61     KategorieChoice(String abbreviation, String description,
62         {bool Function(Set<Choice> choices)? condition})
63         : super(abbreviation, description, condition: condition);
64 }
```

Listing F.1.: Die Klasse *KategorieChoice* in Schritt 4, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/choices.dart](#)

```

77 class ZielflaecheChoice extends Choice {
78   static final ka = ZielflaecheChoice("ka", "keine Angabe/Vorgabe");
79   static final al = ZielflaecheChoice("al", "AL",
80     condition: (choices) => !choices.contains(KategorieChoice.zf_us));
81   static final gl = ZielflaecheChoice("gl", "GL");
82   static final lf = ZielflaecheChoice("lf", "LF");
83   static final dk_sk = ZielflaecheChoice("dk_sk", "DK/SK",
84     condition: (choices) => !choices.contains(FoerderklasseChoice.twm_ziel));
85   static final hff = ZielflaecheChoice("hff", "HFF");
86   static final biotop_le = ZielflaecheChoice(
87     "biotop_le", "Landschaftselement/Biotop o.Ä.",
88     condition: (choices) =>
89       (choices.contains(FoerderklasseChoice.azl) ||
90         choices.contains(FoerderklasseChoice.ea) ||
91         choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
92         choices.contains(FoerderklasseChoice.aukm_ohne_vns)) &&
93       (!choices.contains(KategorieChoice.zf_us) ||
94         !choices.contains(KategorieChoice.bes_kult_rass)));
95   static final wald = ZielflaecheChoice("wald", "Wald/Forst",
96     condition: (choices) =>
97       (choices.contains(FoerderklasseChoice.ea) ||
98         choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
99         choices.contains(FoerderklasseChoice.aukm_ohne_vns)) &&
100       (!choices.contains(KategorieChoice.zf_us) ||
101         !choices.contains(KategorieChoice.bes_kult_rass)));
102   static final contact = ZielflaecheChoice("contact", "bitte um Unterstützung");
103
104   ZielflaecheChoice(String abbreviation, String description,
105     {bool Function(Set<Choice> choices)? condition})
106     : super(abbreviation, description, condition: condition);
107 }

```

Listing F.2.: Die Klasse *ZielflaecheChoice* in Schritt 4, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/choices.dart](#)


```

121 class ZieleinheitChoice extends Choice {
122   static final ka = ZieleinheitChoice("ka", "keine Angabe/Vorgabe");
123   static final m3 = ZieleinheitChoice("m3", "m3 (z.B. Gülle)",
124     condition: (choices) =>
125       (choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
126         choices.contains(FoerderklasseChoice.aukm_ohne_vns)) &&
127       (choices.contains(KategorieChoice.dungmang) ||
128         choices.contains(KategorieChoice.extens)) &&
129       (!choices.contains(ZielflaecheChoice.ka) &&
130         !choices.contains(ZielflaecheChoice.contact)));
131   static final pieces = ZieleinheitChoice(
132     "pieces", "Kopf/Stück (z.B. Tiere oder Bäume)",
133     condition: (choices) =>
134       (choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
135         choices.contains(FoerderklasseChoice.aukm_ohne_vns) ||
136         choices.contains(FoerderklasseChoice.twm_ziel)) &&
137       (!choices.contains(KategorieChoice.zf_us) ||
138         !choices.contains(KategorieChoice.flst) ||
139         !choices.contains(KategorieChoice.umwandlg)) &&
140       (!choices.contains(ZielflaecheChoice.ka) &&
141         !choices.contains(ZielflaecheChoice.contact)));
142   static final gve = ZieleinheitChoice("gve", "GV/GVE",
143     condition: (choices) =>
144       (choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
145         choices.contains(FoerderklasseChoice.aukm_ohne_vns) ||
146         choices.contains(FoerderklasseChoice.twm_ziel)) &&
147       (!choices.contains(KategorieChoice.zf_us) ||
148         !choices.contains(KategorieChoice.anlage_pflege) ||
149         !choices.contains(KategorieChoice.flst) ||
150         !choices.contains(KategorieChoice.umwandlg)) &&
151       (!choices.contains(ZielflaecheChoice.ka) &&
152         !choices.contains(ZielflaecheChoice.contact)));
153   static final rgve = ZieleinheitChoice("rgve", "RGV",
154     condition: (choices) =>
155       (choices.contains(FoerderklasseChoice.aukm_nur_vns) ||
156         choices.contains(FoerderklasseChoice.aukm_ohne_vns) ||
157         choices.contains(FoerderklasseChoice.twm_ziel)) &&
158       (!choices.contains(KategorieChoice.zf_us) ||
159         !choices.contains(KategorieChoice.anlage_pflege) ||
160         !choices.contains(KategorieChoice.flst) ||
161         !choices.contains(KategorieChoice.umwandlg)) &&
162       (!choices.contains(ZielflaecheChoice.ka) &&
163         !choices.contains(ZielflaecheChoice.contact)));
164   static final ha = ZieleinheitChoice("ha", "ha",
165     condition: (choices) =>
166       !choices.contains(ZielflaecheChoice.ka) &&
167       !choices.contains(ZielflaecheChoice.contact));
168   static final contact = ZieleinheitChoice("contact", "bitte um Unterstützung");
169
170   ZieleinheitChoice(String abbreviation, String description,
171     {bool Function(Set<Choice> choices)? condition})
172     : super(abbreviation, description, condition: condition);
173 }

```

Listing F.3.: Die Klasse *ZieleinheitChoice* in Schritt 4, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-4/conditional_form/lib/choices/choices.dart](#)

G. Schritt 6 Anhang

```
276 String? validateChoices(  
277     {required String name,  
278     required Iterable<Choice> choices,  
279     required Set<Choice> priorChoices}) {  
280     if (choices.isEmpty) {  
281         return "Feld ${name} enthält keinen Wert!";  
282     }  
283  
284     bool atLeastOneValueInvalid =  
285         choices.any((c) => !c.conditionMatches(priorChoices));  
286  
287     if (atLeastOneValueInvalid) {  
288         return "Wenigstens ein Wert im Feld ${name} ist fehlerhaft!";  
289     }  
290  
291     return null;  
292 }
```

Listing G.1.: Die Funktion *validateChoices*, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-6/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

H. Schritt 7 Anhang

```
119 Widget buildSelectionCard<ChoiceType extends Choice>(
120   {required Choices<ChoiceType> allChoices,
121   required BehaviorSubject<ChoiceType?> selectionViewModel,
122   ChoiceMatcher<ChoiceType>? choiceMatcher}) {
123   return FormField(
124     validator: (_) => validateChoices(
125       name: allChoices.name,
126       choices: {
127         if (selectionViewModel.value != null) selectionViewModel.value!
128       },
129     priorChoices: vm.priorChoices.value,
130     choiceMatcher: choiceMatcher ?? defaultChoiceMatcherStrategy),
131     builder: (field) => SelectionCard<ChoiceType>(
132       title: allChoices.name,
133       allChoices: allChoices,
134       priorChoices: vm.priorChoices,
135       initialValue: {
136         if (selectionViewModel.value != null)
137           selectionViewModel.value!
138       },
139       choiceMatcher: choiceMatcher,
140       onSelect: (selectedChoice) =>
141         selectionViewModel.value = selectedChoice,
142       onDeselect: (selectedChoice) => selectionViewModel.value = null,
143       errorText: field.errorText,
144     ));
145 }
```

Listing H.1.: Der *choiceMatcher* wird in der Methode *buildSelectionCard* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

```

147 Widget buildMultiSelectionCard<ChoiceType extends Choice>(
148   {required Choices<ChoiceType> allChoices,
149   required BehaviorSubject<BuiltSet<ChoiceType>> selectionViewModel,
150   ChoiceMatcher<ChoiceType>? choiceMatcher}) {
151   return FormField(
152     validator: (_) => validateChoices(
153       name: allChoices.name,
154       choices: selectionViewModel.value,
155       priorChoices: vm.priorChoices.value,
156       choiceMatcher: choiceMatcher ?? defaultChoiceMatcherStrategy),
157     builder: (field) => SelectionCard<ChoiceType>(
158       title: allChoices.name,
159       multiSelection: true,
160       allChoices: allChoices,
161       priorChoices: vm.priorChoices,
162       initialValue: selectionViewModel.value,
163       choiceMatcher: choiceMatcher,
164       onSelect: (selectedChoice) => selectionViewModel.value =
165         selectionViewModel.value
166           .rebuild((b) => b.add(selectedChoice)),
167       onDeselect: (selectedChoice) => selectionViewModel.value =
168         selectionViewModel.value
169           .rebuild((b) => b.remove(selectedChoice)),
170       errorText: field.errorText,
171     ));
172 }

```

Listing H.2.: Der *choiceMatcher* wird in der Methode *buildMultiSelectionCard* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

```

50 Widget build(BuildContext context) {
51   final focusNode = FocusNode();
52
53   navigateToSelectionScreen() async {
54     focusNode.requestFocus();
55
56     Navigator.push(
57       context,
58       MaterialPageRoute(
59         builder: (context) =>
60           createMultipleChoiceSelectionScreen(context)));
61   }
62
63   final validityChanged = priorChoices
64     .map((choices) =>
65       selectionViewModel.value.any((c) => !choiceMatcher(c, choices)))
66     .distinct();
67
68   final needsRepaint = BehaviorSubject.seeded(true);
69   validityChanged.listen((value) => needsRepaint.add(true));
70   selectionViewModel.listen((value) => needsRepaint.add(true));
71
72   return StreamBuilder(
73     stream: needsRepaint,
74     builder: (context, snapshot) {
75       final selectedChoices = selectionViewModel.value;
76       final bool wrongSelection =
77         selectedChoices.any((c) => !choiceMatcher(c, priorChoices.value));
78
79       return Card(
80         child: Column(
81           crossAxisAlignment: CrossAxisAlignment.start,
82           children: [
83             ListTile(
84               focusNode: focusNode,
85               title: Text(title),
86               subtitle: Text(
87                 selectedChoices.map((c) => c.description).join(", "),
88               trailing: const Icon(Icons.edit),
89               onTap: navigateToSelectionScreen,
90               tileColor:
91                 wrongSelection || errorText != null ? Colors.red : null,
92             ),
93             if (errorText != null)
94               Padding(
95                 padding: const EdgeInsets.all(8.0),
96                 child: Text(errorText!,
97                   style:
98                     const TextStyle(fontSize: 12.0, color: Colors.red)),
99               )
100           ],
101         ),
102       );

```

Listing H.3.: Der *choiceMatcher* wird in der *build*-Methode der Klasse *SelectionCard* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/widgets/selection_card.dart](#)

```

106 Widget createMultipleChoiceSelectionScreen(BuildContext context) {
107   return Scaffold(
108     appBar: AppBar(
109       title: Text(title),
110     ),
111     body: StreamBuilder(
112       stream: selectionViewModel,
113       builder: (context, snapshot) {
114         final selectedChoices = selectionViewModel.value;
115
116         Set<ChoiceType> selectedAndSelectableChoices = {};
117         Set<ChoiceType> unselectableChoices = {};
118
119         for (ChoiceType c in allChoices) {
120           if (selectedChoices.contains(c) ||
121             choiceMatcher(c, priorChoices.value)) {
122             selectedAndSelectableChoices.add(c);
123           } else {
124             unselectableChoices.add(c);
125           }
126         }
127
128         return ListView(children: [
129           ...selectedAndSelectableChoices.map((ChoiceType c) {
130             bool isSelected = selectedChoices.contains(c);
131             bool selectedButDoesNotMatch =
132               !choiceMatcher(c, priorChoices.value);
133
134             return CheckboxListTile(
135               key: Key(
136                 "valid choice ${allChoices.name} - ${c.abbreviation}"),
137               controlAffinity: ListTileControlAffinity.leading,
138               title: Text(c.description),
139               tileColor: selectedButDoesNotMatch ? Colors.red : null,
140               value: isSelected,
141               onChanged: (bool? selected) {
142                 if (selected != null) {
143                   if (multiSelection) {
144                     selectionViewModel.value =
145                       selectionViewModel.value.rebuild((b) {
146                         if (selectionViewModel.value.contains(c)) {
147                           b.remove(c);
148                         } else {
149                           b.add(c);
150                         }
151                       });
152                   } else {
153                     selectionViewModel.value =
154                       selectionViewModel.value.rebuild((b) {
155                         b.replace(isSelected ? [] : [c]);
156                       });
157                   }
158                   if (selected) {
159                     onSelect(c);
160                   } else {
161                     onDeselect(c);
162                   }
163                 }
164               });

```

Listing H.4.: Der *choiceMatcher* wird in der Methode *createMultipleChoiceSelectionScreen* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/widgets/selection_card.dart](#)


```

298 String? validateChoices<ChoiceType extends Choice>(
299     {required String name,
300     required Iterable<ChoiceType> choices,
301     required Set<Choice> priorChoices,
302     required ChoiceMatcher<ChoiceType> choiceMatcher}) {
303     if (choices.isEmpty) {
304         return "Feld ${name} enthält keinen Wert!";
305     }
306
307     bool atLeastOneValueInvalid =
308         choices.any((c) => !choiceMatcher(c, priorChoices));
309
310     if (atLeastOneValueInvalid) {
311         return "Wenigstens ein Wert im Feld ${name} ist fehlerhaft!";
312     }
313
314     return null;
315 }

```

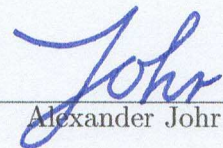
Listing H.5.: Der *choiceMatcher* wird in der Funktion *validateChoices* hinzugefügt, Quelle: Eigenes Listing, Datei: [Quellcode/Schritt-7/conditional_form/lib/screens/massnahmen_detail/massnahmen_detail.dart](#)

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 31.08.2021


Alexander Johr