Entwicklung einer Formularanwendung mit Kompatibilitätsvalidierung der Einfach- und Mehrfachauswahl-Eingabefelder

Vorgelegt von:

Alexander Johr

Meine Adresse

Erstprüfer:	Prof. Jürgen Singer Ph.D.
Zweitprüfer:	Prof. Daniel Ackermann
Datum:	02.11.2020

Hochschule Harz
Fachbereich Automatisierung und Informatik

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

Teil I

Implementierung

1 Schritt 1 - Formular in Grundstruktur erstellen

1.1 Integrations-Test zum Test der Oberfläche

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

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

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

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

Damit der integration_test Ordner für die Quellcode-Generierung der Mocks integriert wird, muss ein entsprechender Eintrag in der Build-Konfiguration vorgenommen werden. Damit das Paket „source_gen“ die entsprechenden Dateien analysiert, müssen sie in der Rubrik `sources` angegeben werden (Listing. 2 Z. 3-8). Wird der Ordner integration_test darin eingefügt (Z. 8), bezieht „source_gen“ den Ordner in der Quellcode-Generierung mit ein. Zusätzlich dazu muss die Rubrik `generate_for` von dem `mockBuilder` des „mockito“-Pakets (Z. 11-13) um die gleiche Angabe des Ordners ergänzt werden (Z. 13).

```
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 1: Initialisierung des Integrations Tests, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/build.yaml

Anschließend kann mit der Annotation and generate mocks (Listing. 2 Z. 20) ein Mock für `MassnahmenJsonFile` angefordert werden. In der Kommandozeile ist `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 2: Initialisierung des Integrations Tests, 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 22 bis 25 wird der Testtreiber initialisiert. Ref. Anschließend wird ein Objekt der generierten Klasse `MockMassnahmenJsonFile` erstellt. Wenn das Model nun während der Applikation versucht, aus der Json-Datei zu lesen, soll der Mock eine leere Liste von Maßnahmen zurückgeben (Z. 28). Dazu wird die entsprechende Methode `when` verwendet. Als erster Parameter wird die Methode `readMassnahmen` des Mocks übergeben. Im darauffolgenden Aufruf `thenAnswer` kann angegeben werden, welche Rückgabe die Methode liefern soll.

Über den `tester` kann mit Hilfe der Methode `pumpWidget` ein beliebiges Widget in der Test-Ausführung konstruiert werden. In diesem Fall ist es die gesamte Applikation, die getestet werden soll. Dementsprechend ist hier erneut der komplette Haupteinstiegspunkt angegeben (Listing 3). Doch der Konstruktor von (Z. 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 3: Initialisierung des Widgets für den Integrations Tests, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart

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

Der Funktion `tabSelectionCard` (Listing 5) 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 4: Die Hilfsmethode tabSelectionCard, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart

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

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

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

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

Wurde das Text-Element gefunden, so muss noch den Vater gesucht werden, der vom Typ `Card` ist (Z. 53). Das kann mit `find.ancestor` erfolgen. Über den Parameter `of` erhält er den `Finder` des Kind-Elements und der Parameter `matching` erhält als Argument die Voraussetzung, die vom Vater-Objekt erfüllt werden soll, als weiteren `Finder`. `find.byType(Card)` sucht also alle Elemente vom Typ `Card`. `find.ancestor` sucht anschließend alle Entsprechungen, in der eine `Card` ein Vater des `Finder textLabel` ist. Wiederum überprüft die Funktion `expect`, dass die Karte gefunden wurde. Doch dieses Mal muss es genau ein Widget sein, welches mit dem „Matcher“ `findsOneWidget` verifiziert werden kann (Z. 54). Sollte mehr als nur eine Karte gefunden werden, so wäre nicht klar, welche geklickt werden soll.

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

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

`tabConfirmButton` funktioniert ähnlich (Listing 5). Das Finden des Buttons ist jedoch einfacher, da es nur einen Button zum Akzeptieren auf jeder Oberfläche gibt. Der Button enthält keinen Text, lässt sich aber auch über seinen Tooltip lokalisieren (Z. 62). Die Hilfsfunktion klickt den Button (Z. 63) und wartet dann erneut auf Vollendung aller angestoßenen Animationen (Z. 64).

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

Listing 5: Die Hilfsmethode tabConfirmButton, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart

Ist der Integrationstest aktuell in dem Auswahlbildschirm, so sorgt `tabOption` dafür, dass Auswahloptionen gewählt wird (Listing 6). 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 6: 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 Vater-Element - das `CheckboxListTile` - ebenfalls getroffen. Der `tester` holt es in den sichtbaren Bereich `71`, klickt es `72` und wartet auf Abschluss aller Animationen (Z. 73). Sollte der optionale Parameter `tabConfirm` auf `true` gesetzt sein (Z. 75), so wird der Auswahlbildschirm anschließend direkt wieder geschlossen, nachdem die Option ausgewählt wurde (Z. 76).

Schließlich kann mit der Hilfsfunktionen `fillTextFormField` ein Formularfeld über dessen Titel gefunden und der entsprechende übergebende Text eingetragen werden (Listing 7). Sie findet das `TextFormField`, indem es zunächst nach dem Titel mit `find.text(title)` und anschließend dessen Vater-Element vom Typ `TextFormField` sucht (Z. 83). Sollte sowohl der Hinweistext als auch der Titel den gleichen Text enthalten, so kann es sein, dass zwei solche Elemente gefunden werden. In Wahrheit ist es aber zwei Mal dasselbe `TextFormField`. Mit `.first` wird lediglich das erste Element geliefert (Z. 85). Nachdem feststeht, dass das Element existiert (Z. 85) und es in den sichtbaren Bereich gescrollt wurde (Z. 87), gibt der

Integrationstest den gewünschten Text in das Eingabefeld ein (Z. 88). Anschließend wird erneut auf Abschluss aller Animationen gewartet (Z. 89).

```
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 7: Die Hilfsmethode fillTextFormField, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart

Während der Integrationstest startet, öffnet sich als Erstes der Übersichts-Bildschirm. Zunächst wird gewartet, dass alle Widgets korrekt initialisiert wurden (Listing. 8 Z. 92). Es folgt der Klick auf den Button zum Erstellen einen neuen Maßnahme (Z. 95). Dazu wird der Button über den entsprechenden `Key` gefunden (Z. 94). Vor allem jetzt ist das Abwarten mittels `pumpAndSettle` (Z. 96) unablässig, denn es wird auf einen anderen Bildschirm navigiert. Angenommen der Test wartet nicht ab, so würden die Aktionen noch immer auf den Elementen des alten Bildschirms Anwendung finden.

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

Listing 8: Der Button zum Kreieren einer Maßnahme wird ausgelöst, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart

Der Integrationstest öffnet nun den Auswahl-Bildschirm, in dem die Selektions-Karte zum Setzen des letzten Statuses angewählt wird (Listing. 9 Z. 98). Anschließend fällt die Wahl auf die Option für „abgeschlossen“ (Z. 98). Dabei sorgt `tabConfirm: true` für die sofortige Rückkehr zum Eingabeformular nach der Auswahl.

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

Listing 9: 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 Maßnahmen-Titel überprüfen (Listing 10). Es erfolgt die Erstellung eines beispielhaften Titels anhand des aktuellen Datums und der aktuellen Uhrzeit (Z. 101, 102). Der erstellte Text dient als Eingabe für das Eingabefeld (Z. 104).

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

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

Listing 10: Der Maßnahmentitel wird eingegeben, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart

ermöglicht die Überprüfung auf irgendeine Übergabe und stellt die übergebenen Argumente über den Rückgabewert bereit.

```dart
var saveMassnahmeButton = find.byTooltip(saveMassnahmeTooltip);
await tester.tap(saveMassnahmeButton);
await tester.pumpAndSettle(durationAfterEachStep);

var capturedJson =
    verify(massnahmenJsonFileMock.saveMassnahmen(captureAny)).captured.last;

var actualMassnahme = capturedJson['massnahmen'][0] as Map;
actualMassnahme.remove("guid");
actualMassnahme["letzteBearbeitung"].remove("letztesBearbeitungsDatum");

var expectedJson = {
  'letzteBearbeitung': {'letzterStatus': 'fertig'},
  'identifikatoren': {'massnahmenTitel': massnahmeTitle},
};

expect(actualMassnahme, equals(expectedJson));
```

Listing 11: Validierung des Testergebnisses, Quelle: Eigenes Listing, Datei: Quellcode/Schritt-1/conditional_form/integration_test/app_test.dart

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

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

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

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

Teil II

Anhang

A Schritt 7 Anhang

Ich wurde zwischengeparkt Weil es auch möglich sein soll, das eine Selektions-Karte nicht nur direkt in der Eingabemaske sondern auch als Kind einer Option anderen Elementes auftaucht, ist ein optionaler Parameter ancestor hinterlegt 49. Ich wurde zwischengeparkt

Ich wurde zwischengeparkt Ist aber die selektions Karte Kind eines anderen Elementes so soll nur nach Elementen besucht werden, die Kind Elemente des angegebenen Vater Elementes sind. Dies kann mit feiner. Descendant erfolgen. Dazu wird das Vater Element dem Parameter of übergeben. Der Parameter matching erhält wiederum ein Feinde Objekt für das Kindelement. In diesem Fall ist dies erneut find. Text, welcher nach dem Titel der Selektion Skate sucht. Ich wurde zwischengeparkt

Ich wurde zwischengeparkt Um nun das Vater-Element zu finden wird nicht wie zuvor fein. Enzersdorf verwendet. Während der Erstellung dieser Arbeit wurde versucht lediglich Feinde Objekte zu benutzen. Doch je häufiger ein feinder Objekt in einem anderen verschachtelt wird, desto länger dauert die Suche nach dem gewünschten Element. Ohne Optimierungen dauerte das Finden des Elementes daher mitunter mehrere Sekunden. Deshalb wurde versucht so häufig es nur geht Alternativen zugfinder Objekten zu verwenden. So kann etwa mittels Methode element und dem Feinde object choice Label das tatsächliche visuelle Objekt gefunden werden. mittels Methode findAncestorWidgetOfExactType Kann über die Vater Elemente des Elementes iteriert werden. Sobald das elements mit dem gewünschten typ CheckboxListTile Gefunden wurde 80, speichert die lokale Variable listtilekey 78 den dem Element hinterlegten key ab. Der Hintergrund dafür ist, dass Methoden wie expekt, Tab, in schwissel und so weiter nur mit Feinde Objekten funktionieren. die Methode bikee konvertiert den Schlüssel in einen fein da, der ein Element über den Schlüssel sucht. sucht.de ist als Suche nach allen existierenden Textelementen und die anschließende Suche von checkboxlist Teil 1 Enten, die diesen Text enthalten. Mithilfe dieser Optimierung konnten die Tests in ihrer Laufzeit Geschwindigkeit deutlich verbessert werden.

Nachdem überprüft wurde, dass von dem listtile genau ein Element existiert 84, wird es in den sichtbaren Bereich gerückt 88, und anschließend angeklickt 87, sowie auf abschließen alle Animationen gewartet 88. Ich wurde zwischengeparkt

Ich wurde zwischengeparkt Damit jedoch auch Elemente innerhalb von des List teils gefunden werden können, wird der feinder, der nach dem key das ist teils sucht zurückgegeben, damit er gegebenenfalls wieder verwendet werden kann. Ich wurde zwischengeparkt