Entwurfsmuster für das State-Management in Flutter

Autor

Steffen Nowacki · PartMaster GmbH · www.partmaster.de

Abstract

Hier wird die Anwendung von zwei Entwurfsmustern vorgestellt, die UI und Logik einer Flutter-App vom verwendeten State-Management-Framework entkoppeln. State-Management-Frameworks dienen der Trennung von UI und Logik. Sie haben oft die Nebenwirkung, UI und Logik zu infiltrieren und dadurch ungünstige Abhängigkeiten zu erzeugen. Mit einer Kombination der Entwurfsmuster "State Reducer" und "Humble Object" sowie dem Konzept der Funktionalen Programmierung mit unveränderlichen Zustandsobjekten kann dem entgegengewirkt werden. Die in der Umsetzung der Entwurfsmuster verwendeten Bausteine AppState, Reducer, Reducible und Callable sowie Binder, Builder, Props und Transformer werden im Folgenden erklärt. Die entstehende Code-Struktur ist einfacher lesbar, testbar, skalierbar und kompatibel zu verbreiteten State-Management-Frameworks, wie Riverpod ¹ oder Bloc ².

Diese Vorteile haben ihren Preis: Gegenüber der direkten Verwendung eines State-Management-Frameworks bzw. der Verwendung von veränderlichen Zustandsobjekten ensteht eine zusätzliche Abstraktionsschicht und mehr Boilerplate-Code.

Wer beim Einsatz von State-Management-Frameworks flexibel bleiben will oder wer seine Widget-Baum-Code-Struktur übersichtlicher gestalten will, oder wer sich einfach nur einen Überblick über verfügbare State-Management-Frameworks verschaffen will, für den könnte der Artikel interessant sein.

Teil 1

Anwendung des Humble-Object-Pattern

Flutter ³ beschreibt sich selbst mit dem Spruch "Fast alles ist ein Widget" ⁴. Damit ist gemeint, dass alle Features in Form von Widget-Klassen implementiert sind, die sich wie Lego-Bausteine aufeinander stecken lassen. Das ist eine großartige Eigenschaft. Es gibt

aber auch eine kleine Kehrseite: Wenn man nicht aufpasst, vermischen sich in den resultierenden Widget-Bäumen leicht die Verantwortlichkeiten.

Verantwortlichkeiten in der Counter-Demo-App

Nehmen wir als Beispiel die wohlbekannte Counter-Demo-App:

```
class _MyHomePageState extends State<MyHomePage> {
 int _counter = 0;
 void _incrementCounter() {
    setState(() {
      _counter++;
   });
 }
 @override
 Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
   );
 }
}
```

Die Klasse _MyHomePageState trägt die verschiedensten Verantwortungen:

1. Layout

```
mainAxisAlignment: MainAxisAlignment.center,
```

2. Rendering

```
style: Theme.of(context).textTheme.headline4,
```

3. Gestenerkennng

onPressed:

4. App-Zustands-Speicherung

```
int _counter = 0;
```

5. App-Zustands-Änderungs-Operationen

```
void _incrementCounter() {
```

6. Widget-Rebuilds nach App-Zustands-Änderungen

```
setState(() {
```

7. Konvertierung des App-Zustands in Anzeige-Properties

```
'$_counter',
```

8. Abbildung von Gesten-Callbacks auf App-Zustands-Änderungs-Operationen

```
onPressed: _incrementCounter,
```

Das Prinzip der Trennung von Verantwortlichkeiten ist lange bekannt. Trotzdem ist seine Durchsetzung, vor allem innerhalb von UI-Code, nach meiner Erfahrung immer eine Herausforderung. UI-Code hat die Besonderheit, dass er eng an seine Ablaufumgebung, das UI-Framework, gebunden ist und deswegen schon eine Komplexität besitzt. Da diese Komplexität des UI-Codes inhärent und nicht vermeidbar ist, bleibt als Ziel nur, sie möglichst nicht noch zu erhöhen.

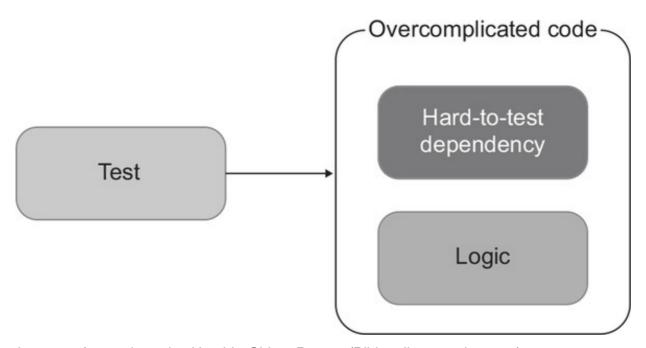
Ein Entwurfmuster, das genau auf diese Problemlage passt, ist das Humble-Object-Pattern ⁵ von Micheal Feathers.

Die Definition des Humble-Object-Pattern

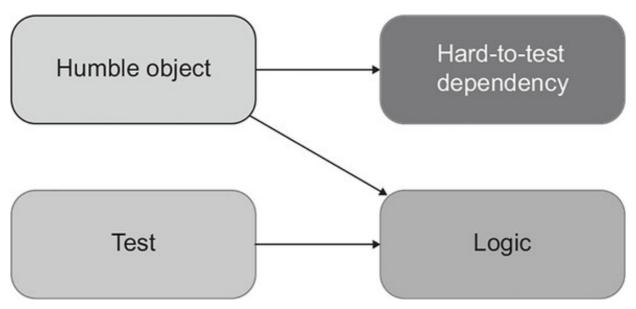
Die Zusammenfassung des Humble-Object-Pattern lautet:

Wenn Code nicht gut testbar ist, weil er zu eng mit seiner Umgebung verbunden ist, extrahiere die Logik in eine separate, leicht zu testende Komponente, die von ihrer Umgebung entkoppelt ist.

Die folgenden zwei Grafiken illustrieren die Lage vor und nach der Anwendung dieses Entwurfsmuster:



Lage vor Anwendung des Humble-Object-Pattern (Bildquelle: manning.com)



Lage nach Anwendung des Humble-Object-Pattern (Bildquelle: manning.com)

Counter-Demo-App refactored

Für eine kleine Demo App, wie die Counter-Demo-App, ist es angemessen, das so viele Verantwortlichkeiten in einer einzigen Klasse zusammengefasst sind.

Trotzdem habe ich diese App ausgewählt, um an ihr das Humble-Object-Pattern anzuwenden und anhand des Ergebnisses die Brauchbarkeit des Patterns für Flutter-Widget-Bäume zu initial zu bewerten.

Hier nun das Ergebnis der Anwendung des Pattern in Form der neuen Klassen, in die die verschiedenen Verantwortlichkeiten (oder Logik-Bestandteile) aus der ursprünglichen Klasse MyHomePageState extrahiert wurden, sowie die verbleibende Humble-Object-Klasse MyHomePageBuilder.

In den extrahierten Klassen habe ich eine Abstraktion für das State-Management-System, bestehend aus den Interfaces Reducible, Reducer und Callable, der Klasse ReducerOnReducible sowie den Funktionen wrapWithProvider und wrapWithConsumer verwendet, die ich später vorstelle.

App-Zustands-Speicherung

Für die Speicherung des App-Zustands habe ich zwei Konstrukte vorgesehen: Eine Klasse MyAppState für den eigentlichen App-Zustand, Eine Klasse MyAppStateBinder, die den initialen Wert des App-Zustands festlegt und an die Funktion wrapWithProvider übergibt.

Die Funktion wrapWithProvider abstrahiert das verwendete State-Management-Framework und sorgt dafür, dass es für die nachfolgenden Widgets im Widget-Baum zugreifbar wird.

MyAppState

Um den App-Zustand speichern zu können, habe ich das Property counter in eine Klasse MyAppState ausgelagert und das Property title hinzugefügt, obwohl das Property nie geändert wird.

MyAppStateBinder

Die Klasse MyAppStateBinder bindet die spezifische App-Zustands-Klasse MyAppState mit der Methode wrapWithProvider an eine App-Zustands-Verwaltiungs-Instanz.

App-Zustands-Änderungs-Operationen

Die Counter App hat nur eine einzige App-Zustands-Änderungs-Operation.

IncrementCounterReducer

Die Methode call der Klasse IncrementCounterReducer erzeugt einen neuen MyAppState -Wert in welchem das Property counter gegenüber dem als Parameter state übergebenen MyAppState -Wert inkrementiert wurde.

Die Basisklasse Reducer definiert die Signatur der call -Methode für alle App-Zustands-Änderungs-Operationen: MyAppState call(MyAppState state); und wird später erläutert.

```
class IncrementCounterReducer extends Reducer<MyAppState> {
  const IncrementCounterReducer._();

  static const instance = IncrementCounterReducer._();

  @override
  call(state) => state.copyWith(counter: state.counter + 1);
}
```

Widget-Rebuilds nach App-Zustands-Änderungen

Für die Benachrichtigung über einen notwendigen Rebuild nach App-Zustands-Änderungen habe ich die Funktion wrapWithConsumer vorgesehen.

Die Funktion wrapWithConsumer abstrahiert das verwendete State-Management-Framework und sorgt dafür, dass der übergebene builder bei jeder Änderung des App-Zustandes aufgerufen wird. Der übergebene transformer transformiert dabei den eigentlichen MyAppState in den vom builder erwarteten Parameter-Typ.

MyHomePageStateBinder

Die Klasse MyHomePageStateBinder legt fest, dass das Widget MyHomePageBuilder gebaut wird, und dass die Funktion MyHomePagePropsTransformer.transform verwendet wird, um den benötigten Konstruktor-Parameter für fir Klasse MyHomePageBuilder zu erzeugen.

Konvertierung des App-Zustands in Anzeige-Properties

Bei der Erzeugung des Widget-Baums sind einige Widget-Konstruktor-Properties vom aktuellen App-Zustand abhängig. Diese werden zu eigenen Property-Klassen zusammengefasst. Außerdem wird für jede Property-Klasse eine transform -Funktion definiert, die den App-Zustand in eine Instanz der Property-Klasse transformieren kann.

MyHomePageStateProps

Die in der build-Methode von MyHomePageBuilder benötigten Properties werden in der Klasse MyHomePageProps zusammengefasst.

```
class MyHomePageProps {
  const MyHomePageProps({
    required this.title,
    required this.counterText,
    required this.onIncrementPressed,
  });

final String title;
  final String counterText;
  final Callable<void> onIncrementPressed;

@override get hashCode => ...

@override operator ==(other) => ...
}
```

MyHomePageStatePropsTransformer

```
class MyHomePagePropsTransformer {
   static MyHomePageProps transform(Reducible<MyAppState> reducible) =>
        MyHomePageProps(
        title: reducible.state.title,
        counterText: '${reducible.state.counter}',
        onIncrementPressed: reducible.incrementCounterReducer,
        );
}
```

Abbildung von Gesten-Callbacks auf App-Zustands-Änderungs-Operationen

Flutter-Widgets stellen für die Gestenverarbeitung und ähnliche Zwecke Callback-Properties zur Verfügung. Wir behandeln Callback-Properties genauso wie die bereits besprochenen Anzeige-Properties und fügen sie zur gleichen Properties-Klasse MyHomePageProps hinzu. Die transform -Funktion erzeugt aus der App-Zustands-Operation IncrementCounterReducer den Wert für das onIncrementPressed -Callback-Property.

Dazu wird eine Convenience-Methode get incrementCounterReducer definiert, die die App-Zustands-Operation mittels der Klasse ReducerOnReducible an die State-Management-Instanz bindet.

Layout, Rendering und Gestenerkennung

Die restlichen Verantwortlichkeiten Layout, Rendering und Gestenerkennung konnte ich nicht mehr herauslösen, weil sie sich kaum vom UI-Framework trennen lassen. Sie verbleiben im resultierenden Humble-Object.

MyHomePageStateBuilder

In der umgewandelten Counter-Demo-App bildet die Klasse MyHomePageBuilder das Humble-Object und ist für Layout, Rendering und Gestenerkennung zuständig.

```
class MyHomePageBuilder extends StatelessWidget {
  const MyHomePageBuilder({super.key, required this.props});
  final MyHomePageProps props;
  @override
  Widget build(context) => Scaffold(
        appBar: AppBar(
          title: Text(props.title),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>
              const Text(
                'You have pushed the button this many times:',
              ),
              Text(
                props.counterText,
                style: Theme.of(context).textTheme.headlineMedium,
            ],
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: props.onIncrementPressed,
          tooltip: 'Increment',
          child: const Icon(Icons.add),
        ),
      );
}
```

Fazit zur Anwendung des Humble-Object-Pattern

Die Anwendung des Humble-Object-Pattern auf eine Flutter-Widget-Klasse, die einen Widget-Baum erzeugt und Abhängigkeiten vom App-Zustand hat, besteht aus folgenden fünf Schritten:

- 1. Wenn die Widget-Klasse sowohl UI-Aufgaben als auch App-Zustands-Aufgaben löst, dann wird diese Widget-Klasse in eine Builder-Klasse, eine Binder-Klasse, eine Props-Klasse und eine Transformfunktion zur Erzeugung von Props-Instanzen geteilt.
- 2. Die Builder-Klasse ist ein StatelessWidget. Sie bekommt von der Binder-Klasse im Konstruktor die Props-Instanz mit vorkonfektionierten Properties und Callbacks und erzeugt in der build-Methode einen Widget-Baum aus Layout-, Renderer und

Gestenerkennungs-Widgets.

- 3. Die Binder-Klasse lauscht bei der State-Management-Instanz selektiv auf Änderungen 'ihrer' Props und liefert in der build-Methode ein Widget der Builder-Klasse zurück.
- 4. Für die vorkonfektionierten Properties und Callbacks der Builder-Klasse wird eine Props-Klasse definiert eine reine Datenklasse mit ausschließlich finalen Feldern.
- 5. Für die Props-Klasse wird eine Transform-Funktion definiert, die aus dem aktuellen App-Zustand die Werte für die Properties und aus den den Reducer-Implementierungen die Werte für die Callbacks erzeugt.

Diese Schritte, angewandt auf die Counter-Demo-App, bringen folgendes Ergebnis:

Drei Verantwortlichkeiten aus der Klasse _MyHomePageState verbleiben im Humble Object:

- 1. Layout
- 2. Rendering
- 3. Abbildung von Gesten-Callbacks auf App-Zustands-Änderungens-Operationen

Fünf Verantwortlichkeiten wurden aus der Klasse _MyHomePageState in eigene Klassen bzw. Funktionen extrahiert:

- 1. App-Zustands-Speicherung
- 2. Bereitstellung von Operationen für App-Zustands-Änderungen
- 3. Widget-Benachrichtigung nach App-Zustands-Änderungen
- 4. Konvertierung des App-Zustands in Anzeige-Properties
- 5. Abbildung von Gesten-Callbacks auf App-Zustands-Änderungs-Operationen

In den extrahierten Klassen und Funktionen ist viel Boilerplate-Code entstanden und es wurde eine Abstraktion für das State-Management-Framework verwendet.

Für wen der Boilerplate-Code ein zu hoher Aufwand für das erreichte Ergebnis bedeutet, der kann jetzt aus dem Artikel aussteigen.

Für die weiter Interessierten will ich nun die verwendete Abstraktion für das State-Management-System mit den bereits erwähnten Interfaces Reducible, Reducer und Callable, der Klasse ReducerOnReducible, den Funktionen wrapWithProvider und wrapWithConsumer sowie weiteren Artefakten vorstellen.

Teil 2

Anwendung des State-Reducer-Pattern

Bei allen fünf mittels des Humble-Object-Pattern extrahierten Verantwortlichkeiten handelt es

sich um State-Management-Verantwortlichkeiten.

In der Counter-Demo-App wird das State-Management mit einem StatefulWidget implementiert. Das StatefulWidget und das InheritedWidget sind die beiden von Flutter bereitgestellten Bausteine für das State-Management. Diese beiden Bausteine sind Low-Level-Bausteine. Nicht-triviale Apps benötigen meist eine höherwertige Lösung für das State-Management. In der Flutter-Community sind viele Frameworks enstanden, um diesen Bedarf zu decken. In der offiziellen Flutter-Dokumentation sind aktuell 13 solcher State-Management-Frameworks gelistet ⁶.

Nachdem fünf Verantwortlichkeiten mit Mühe (und Boilerplate-Code) aus der Abhängigkeit von der UI-Umgebung gelöst wurden, ist es nur konsequent, sie mittels einer geeigneten Abstraktion auch vor der Abhängigkeit von einem konkreten State-Management-Framework zu bewahren.

Ich nenne die Abstraktion in Anlehnung an das zugrunde liegende Pattern 'reduced'. Die wesentlichen Bestandteile der 'reduced'-Abstraktion wurden bereits aufgezählt:

1. Interface Reducer

zur Definition von Operationen zur Änderung des App-Zustandes

2. Interface Reducible

zum Lesen und Aktualisieren des App-Zustands in einer State-Management-Instanz

3. Interface Callable

für die Defintion von Klassen mit Wertsemantik, deren Instanzen an Callback-Properties von Flutter-Widgets zugewiesen werden können,

zur Verwendung in den im Kapitel über die Anwendung des Humble-Object-Pattern erwähnten Props-Klassen, so dass diese Klassen ebenfalls mit Wertsemantik definiert werden können

4. Klasse ReducerOnReducible

zur Verknüpfung einer App-Zustands-Operation mit der State-Management-Instanz, auf der sie ausgeführt werden soll

5. Funktion wrapWithProvider

zum Einpacken eines Widgets in ein sogenanntes Provider-Widget, welches im Widget-Baum den Zugriff auf eine State-Management-Instanz zur Verfügung stellt

6. Funktion wrapWithConsumer

zum Einpacken eines Widgets in ein sogenanntes Consumer-Widget, welches dafür sorgt, dass das eingepackte Widget bei Änderungen am App-Zustand neu gebaut wird

Die Definition des State-Reducer-Pattern

Nach der Übersicht folgt nun die detaillierte Beschreibung der Abstraktion 'reduced' für

State-Management-Frameworks. Die Abstraktion ist im Kern eine Anwendung das State-Reducer-Pattern, darum wird zunächst dieses Entwurfmuster vorgestellt.

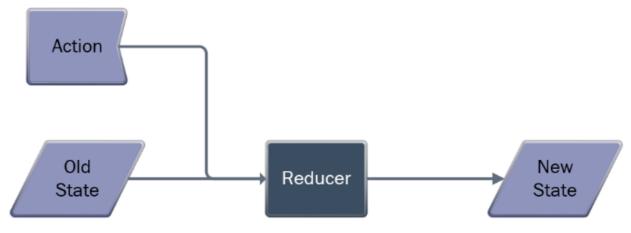
Einfach ausgedrückt beinhaltet das State-Reducer-Pattern die Forderung, dass jede Änderung am App-Zustand als atomare Operation mit dem aktuellen App-Zustand als Parameter und einem neuen App-Zustand als Resultat ausgeführt wird. Aktionen, die potenziell länger laufen (Datenbank-Anfragen, Netzwerk-Aufrufe, ..), müssen wegen dieser Forderung meist mit mehreren atomaren App-Zustands-Änderungen umgesetzt werden, z.B. eine am Beginn der Aktion und eine am Ende. Entscheidend ist, dass die App-Zustands-Änderung am Ende der Aktion nicht das App-Zustands-Resultat vom Anfang der Aktion als Parameter (wieder-)verwendet, sondern den dann aktuellen App-Zustand des State-Management-Frameworks. Das Pattern unterstützt diese Absicht, indem es dafür sorgt, dass man bei einer Änderung den aktuellen App-Zustand nicht selbst holen muss, sondern unaufgefordert geliefert bekommt.

Oder etwas analytischer ausgedrückt: Das State-Reducer-Pattern modelliert den App-Zustand als Ergebnis einer Faltungsfunktion ⁷ aus dem initialen App-Zustand und der Folge der bisherigen App-Zustands-Änderungs-Aktionen.

Dan Abramov und Andrew Clark haben dieses Konzept im Javascript-Framework Redux ⁸ verwendet und für den Kombinierungsoperator, der aus dem aktuellen App-Zustand und einer Aktion einen neuen App-Zustand berechnet, den Namen *Reducer* populär gemacht ⁹:

Reducers sind Funktionen, die den aktuellen Zustand und eine Aktion als Argumente nehmen und ein neues Zustandsergebnis zurückgeben.

Mit anderen Worten: (state, action) => newState.



Bildquelle: killalldefects.com

Auf App-Code bezogen heißt das:

- Für den App-Zustand wird eine AppState-Klasse definiert eine reine Datenklasse mit ausschließlich finalen Feldern und einem const Konstruktor. Die State-Management-API stellt eine get-Methode für den aktuellen App-Zustand zur Verfügung.
- 2. Die State-Management-API stellt eine reduce-Methode zur Verfügung, die einen

Reducer als Parameter akzeptiert. Ein Reducer ist eine reine 10 synchrone Funktion, die eine Instanz der AppState-Klasse als Parameter bekommt und eine neue Instanz der AppState-Klasse als zurückgibt. Beim Aufruf führt die reduce-Methode den übergebenen Reducer mit dem aktuellen App-Zustand als Parameter aus und speichert den Rückgabewert des Reducer-Aufrufs als neuen App-Zustand ab.

Nach dieser Vorstellung des State-Reducer-Pattern folgen nun die Details zu den Bestandteilen der Abstraktion 'reduced' für State-Management-Frameworks.

Interface Reducer

Basis-Interface für die Implementierungen von App-Zustands-Änderungs-Operationen das die Signatur der Methode für die Ausführung solcher Operation festlegt.

```
abstract class Reducer<S> {
   S call(S state);
}
```

Neben der Grundvariante des Interfaces gibt es weitere Varianten mit zusätzlichen Parametern für die App-Zustands-Änderungs-Operationen, z.B. mit einem Parameter:

```
abstract class Reducer1<S, V> {
   S call(S state, V value);
}
```

Interface Reducible

Basis-Interface für State-Management-Instanzen mit einem Getter get state für den AppState und einer Methode reduce zum Aktualisieren des AppState entsprechend dem State-Reducer-Pattern.

```
abstract class Reducible<S> {
   S get state;
   void reduce(Reducer<S> reducer);
}
```

Interface Callable

Basis-Interface für Implementierungen von Callbacks. Die Implementierung von Callbacks als Klassen und nicht als Funktionen erlaubt das Überschreiben von get hashCode und

operator == (other) für Wertsemantik.

```
abstract class Callable<R> {
   R call();
}
```

Neben der Grundvariante des Interfaces gibt es weitere Varianten mit zusätzlichen Parametern für die Callbacks, z.B. mit einem Parameter:

```
abstract class Callable1<R, V> {
   R call(V value);
}
```

Klasse ReducerOnReducible

Die Klasse implementiert das Interface Callable mit einem Reducer und einem Reducible indem bei Ausführung des Callbacks die Methode reduce des Reducible mit dem Reducer als Parameter ausgeführt wird.

```
class ReducerOnReducible<S> extends Callable<void> {
  const ReducerOnReducible(this.reducible, this.reducer);

final Reducible<S> reducible;
  final Reducer<S> reducer;

@override call() => reducible.reduce(reducer);

@override get hashCode => ...
  @override operator ==(other) => ...
}
```

Neben der Grundvariante des Klasse gibt es weitere Varianten mit zusätzlichen Parametern für die Callbacks, z.B. mit einem Parameter:

```
class Reducer10nReducible<S, V> extends Callable1
const Reducer10nReducible(this.reducible, this.reducer);
final Reducible<S> reducible;
final Reducer1<S, V> reducer;

@override call(value) =>
    reducible.reduce(Reducer1Adapter(reducer, value));

@override get hashCode => ...
@override operator ==(other) => ...
}
```

Funktion wrapWithProvider

zum Einpacken eines Widgets in ein sogenanntes Provider-Widget, welches im Widget-Baum den Zugriff auf eine State-Management-Instanz zur Verfügung stellt

Funktion wrapWithConsumer

zum Einpacken eines Widgets in ein sogenanntes Consumer-Widget, welches dafür sorgt, dass das eingpackte Widget bei Änderungen am App-Zustand neu gebaut wird

Fazit zur Anwendung des State-Reducer-Pattern

Da die 'reduced'-API für jedes konkrete State-Management-Framework nur einmal implementiert werden muss, verursacht sie keinen zusätzlichen Boilerplate-Code, sondern nur eine zusätzliche Abstraktionsschicht. Aber auch jede Abstraktionsschicht verursacht Aufwände, die gegenüber dem Nutzen abgewogen werden sollten.

Die 'reduced'-API deckt nur den 'Standard-Teil' der APIs der State-Management-Frameworks ab. Jedes Framework biete über die 'reduced'-API hinaus noch individuelle Features. Um auch solche Features nutzen zu können, kann man die 'reduced'-API erweitern oder an der 'reduced'-API vorbei direkt mit der Framework-API arbeiten. Falls in einem Projekt die Notwendigkeit für direkte Nutzung der State-Management-Framework-API kein Ausnahmefall bleibt, dann ist es wahrscheinlich, dass die Nutzung einer Abstraktionsschicht für das State-Management-Framework in so einem Projekt ungünstig ist.

Teil 3

Implementierung der 'reduced'-API

Eine Implementierung der 'reduced'-API für ein konkretes State-Management-Framework besteht aus der Implementierung des Interfaces Reducible sowie den Implementierungen der Funktionen wrapWithProvider und wrapWithConsumer. Optional kann noch eine Extension für den BuildContext hinzukommen, die einen bequemen Zugriff auf die State-Management-Instanz bereitstellt.

'reduced'-API-Implementierung am Beispiel Bloc

Wie eine 'reduced'-Implementierung aussieht, soll beispielhaft anhand des Bloc-Frameworks gezeigt werden.

Reducible-Implementierung für Bloc

Das Framework Bloc implementiert State-Management-Instanzen mit der Klasse

Bloc<E, S>, wobei E der Typ-Parameter für State-Management-Ereignisse und S

der Typ-Parameter für die Zustands-Klasse ist. Wir verwenden als Ereignis-Typ das Interface

Reducer aus der 'reduced'-API. Da die Reducer ihre Operation auf dem App-Zustannd schon mitbringen, brauchen sie kein individuelles Dispatching, sondern sie können selbst ausgeführt werden. Die Methode S get state bringt die Klasse Bloc bereits mit und die Methode Reducible.reduce kann direkt auf die Methode Bloc.add abgebildet werden.

```
class ReducibleBloc<S> extends Bloc<Reducer<S>, S>
    implements Reducible<S> {
    ReducibleBloc(super.initialState) {
        on<Reducer<S>>((event, emit) => emit(event(state)));
    }

@override
    void reduce(Reducer<S> reducer) => add(reducer);

late final reducible = this;
}
```

Extension für dem BuildContext

```
extension ExtensionBlocOnBuildContext on BuildContext {
   ReducibleBloc<S> bloc<S>() =>
      BlocProvider.of<ReducibleBloc<S>>(this);
}
```

wrapWithProvider-Implementierung für Bloc

Die Funktion wrapWithProvider erzeugt das Widget BlocProvider .

```
Widget wrapWithProvider<S>({
   required S initialState,
   required Widget child,
}) =>
   BlocProvider(
     create: (_) => ReducibleBloc(initialState),
     child: child,
);
```

wrapWithConsumer-Implementierung für Bloc

Die Funktion wrapWithConsumer erzeugt das Widget BlocSelector. Die benötigte Reducible -Instanz wird implizit übergeben, indem wrapWithConsumer als Extension der Klasse ReducibleBloc definiert wird.

```
extension WrapWithConsumer<S> on ReducibleBloc<S> {
    Widget wrapWithConsumer<P>({
        required ReducibleTransformer<S, P> transformer,
        required PropsWidgetBuilder<P> builder,
}) =>
        BlocSelector<ReducibleBloc<S>, S, P>(
        selector: (state) => transformer(reducible),
        builder: (context, props) => builder(props: props),
        );
}
```

Reducible-Implementierung für Bloc

Das Framework Bloc implementiert State-Management-Instanzen mit der Klasse

Bloc<E, S>, wobei E der Typ-Parameter für State-Management-Ereignisse und S

der Typ-Parameter für die Zustands-Klasse ist. Wir verwenden als Ereignis-Typ das Interface

Reducer aus der 'reduced'-API. Da die Reducer ihre Operation auf dem App-Zustannd schon mitbringen, brauchen sie kein individuelles Dispatching, sondern sie können selbst ausgeführt werden. Die Methode S get state bringt die Klasse Bloc bereits mit und die Methode Reducible.reduce kann direkt auf die Methode Bloc.add abgebildet werden.

```
class ReducibleBloc<S> extends Bloc<Reducer<S>, S>
    implements Reducible<S> {
    ReducibleBloc(super.initialState) {
        on<Reducer<S>>((event, emit) => emit(event(state)));
    }

@override
    void reduce(Reducer<S> reducer) => add(reducer);

late final reducible = this;
}
```

Extension für dem BuildContext

```
extension ExtensionBlocOnBuildContext on BuildContext {
   ReducibleBloc<S> bloc<S>() =>
        BlocProvider.of<ReducibleBloc<S>>(this);
}
```

wrapWithProvider-Implementierung für Bloc

Die Funktion wrapWithProvider erzeugt das Widget BlocProvider.

```
Widget wrapWithProvider<S>({
   required S initialState,
   required Widget child,
}) =>
   BlocProvider(
     create: (_) => ReducibleBloc(initialState),
     child: child,
   );
```

wrapWithConsumer-Implementierung für Bloc

Die Funktion wrapWithConsumer erzeugt das Widget BlocSelector. Die benötigte Reducible -Instanz wird implizit übergeben, indem wrapWithConsumer als Extension der Klasse ReducibleBloc definiert wird.

```
extension WrapWithConsumer<S> on ReducibleBloc<S> {
   Widget wrapWithConsumer<P>({
     required ReducibleTransformer<S, P> transformer,
     required PropsWidgetBuilder<P> builder,
}) =>
     BlocSelector<ReducibleBloc<S>, S, P>(
        selector: (state) => transformer(reducible),
        builder: (context, props) => builder(props: props),
     );
}
```

Tabelle der 'reduced'-API-Implementierungen

In der Flutter-Dokumentation sind aktuell 13 State Management Frameworks gelistet [^1]. Das Fish-Redux-Framework ist nicht Null-Safety und darum veraltet. Für die anderen 12 Frameworks wurde die 'reduced'-API exemplarisch implementiert. Die folgende Tabelle enthält Links zu diesen Frameworks und zu ihren 'reduced'-Implementierungen.

Name	Publisher	'reduced'-Implementierung	
<u>Binder</u>	romainrastel.com reduced binder		
Fish Redux	<u>Alibaba</u>		
Flutter Bloc	bloclibrary.dev	reduced bloc	
Flutter Command	<u>escamoteur</u>	reduced_fluttercommand	
Flutter Triple	flutterando.com.br	reduced fluttertriple	
Getlt	fluttercommunity.dev	reduced getit	
<u>GetX</u>	g <u>etx.site</u>	reduced getx	
<u>MobX</u>	dart.pixelingene.com	m reduced mobx	
<u>Provider</u>	dash-overflow.net	reduced provider	
Redux	brianegan.com	reduced_redux	
Riverpod	dash-overflow.net	reduced_riverpod	
<u>Solidart</u>	bestofcode.dev	reduced_solidart	
States Rebuilder	Mellati Fatah	reduced statesrebuilder	

Fazit zur Implementierung der 'reduced'-API

Die Ziel der 'reduced'-API ist eine minimale Abstraktionsschicht für State-Management-Frameworks.

Der geringe Code-Umfang und die direkten Abbildungern in den Implementierungen der 'reduced'-API zeigen, dass dieses Ziel erreicht wurde.

Bei den State-Management-Frameworks sind mir folgende Besonderheiten aufgefallen:

MobX - Verwendung eines Code-Generators - Transformation des App-Zustands in selektive Props-Klassen im Provider (statt im Consumer)

GetIt und **GetX** - Service-Locator für die State-Management-Instanz (statt einer Provider-Widget-Klasse).

Riverpod und **StatesRebuilder** - Verwendung globaler Referenzen für App-Zustand und selektive Props-Klassen (zur Vermeidung der Not-Found-Problematik)

Offene Enden

Grauzonen zwischen UI und App-Logik

Zur Implementierung einiger UI-Aktionen benötigt man einen [BuildContext]. Ein prominentes Beispiel ist die Navigation zwischen App-Seiten mit [Navigator.of(BuildContext)]. Die Entscheidung, wann zu welcher App-Seite navigiert wird, ist App-Logik. Die App-Logik sollte möglichst ohne Abhängigkeiten von der UI-Ablaufumgebung bleiben, und ein [BuildContext] repräsentiert quasi diese Ablaufumgebung.

Ein ähnliches Problem sind UI-Ressourcen wie Bilder, Icons, Farben und Fonts, die eine Abhängigkeit zur UI-Ablaufumgebung besitzen und deren Bereitstellung UI-App-Logik erfordern kann. Zwischen UI-Code und App-Logik-Code gibt es also noch Grauzonen, die in klare Abgrenzungen umgewandelt werden sollten. (Im Fall des [Navigator]s gibt es mit dem Property [MaterialApp.navigatorKey] einen möglichen Workaround für die Navigation zwischen App-Seiten ohne [BuildContext].)

Lokale bzw. geschachtelte App-Zustände

Der Ansatz, den kompletten App-Zustand als unveränderliche Instanz einer einzigen Klasse zu modellieren, wird bei sehr komplexen Datenstrukturen, sehr großen Datenmengen oder sehr häufigen Änderungsaktionen an seine Grenzen kommen ¹¹.

In der Redux-Dokumentation gibt es Hinweise 12, 13, wie man diese Grenzen durch eine

gute Strukturierung der App-Zustands-Klasse erweitern kann.

Letztlich kann man versuchen, Performance-kritische Teile aus dem globalen App-Zustand zu extrahieren und mit lokalen Zustands-Verwaltungs-Lösungen umzusetzen. Im State-Management-Framework Fish-Redux ¹⁴ ist es z.B. grundsätzlich so, dass (neben einen globalen App-Zustand) für jede App-Seite eine lokale Seiten-State-Management-Instanz existiert.

UI-Code-Strukturierung

In diesem Artikel wurden Code-Struktur und Entwurfsmuster für eine Trennung der Verantwortlichkeiten von UI-Code und App-Logik-Code diskutiert.

Eine separierte App-Logik kann man mit allen verfügbaren Architektur-Ansätzen und Entwurfmustern weiterstrukturiert werden.

Für den separierten Flutter-UI-Code werden allerdings für größere Projekte weitere Strukturierungskonzepte benötigt:

- Wie separiere ich das Theming, z.B. Light Mode und Dark Mode?
- Wie separiere ich die Layout-Adaptionen für verschiedene Endgeräte-Gruppen, z.B. Smartphone, Tablet, Desktop, Drucker?
- Wie separiere ich den Code für die Layout-Responsivness für Änderungen der App-Display-Größe zwischen den Adaptionsstufen ?
- Wie separiere ich den Code für Animationen?

Groß-Projekt-Erprobung

Die in diesem Artikel vorgestellte Code-Struktur ist ein Ergebnis meiner Erfahrungen aus kleinen und mittleren Flutter-Projekten. Eines davon ist das Projekt Cantarei - die Taizé-Lieder-App. Die App ist frei im Apple- ¹⁵ und im Google- ¹⁶ App-Store verfügbar, so dass jeder Interessierte selbst einen Eindruck gewinnen kann, für welche Projekt-Größen die hier vorgeschlagenen Konzepte bereits praxiserprobt sind. Ob sie sich, so wie sie sind, mit Erfolg auf größere Projekte anwenden lassen, muss sich erst noch erweisen.

Referenzen

[^1] Ist alles ein Widget? twitter.com/ulusoyapps/status/1484090230651162624/photo/1

1. Riverpod

```
<u>riverpod.dev</u> ←
2. Bloc
    <u>bloclibrary.dev</u> ←
 3. Flutter
    <u>flutter.dev</u> ←
 4. Alles ist ein Widget
    docs.flutter.dev/development/ui/layout ←
5. Entwurfmuster
    en.wikipedia.org/wiki/Softwaredesignpattern ←
6. Flutter State Management Approaches
    docs.flutter.dev/development/data-and-backend/state-mgmt/options ←
7. Faltungsfunktion
    www.cs.nott.ac.uk/~gmh/fold.pdf ←
 8. Redux
    redux.js.org/ ←
 9. Reducer Pattern
    redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers ←
10. Pure Funktion
    en.wikipedia.org/wiki/Pure_function ←
11. Reducer Pattern Nachteil
    twitter.com/acdlite/status/1025408731805184000 ←
12. App-Zustands-Gliederung
    redux.js.org/usage/structuring-reducers/basic-reducer-structure ←
```

13. App-Zustands-Normalisierung

redux.js.org/usage/structuring-reducers/normalizing-state-shape ←

- 14. Fish Redux <u>pub.dev/packages/fish_redux</u> ←
- 15. Die App Cantarei im Apple-Appstore apps.apple.com/us/app/cantarei/id1624281880 ←
- 16. Die App *Cantarei* im Google-Playstore <u>play.google.com/store/apps/details?</u>
 <u>id=de.partmaster.cantarei</u> ←