'reduced' - eine Abstraktion für das State Management in Flutter

Autor

Steffen Nowacki · PartMaster GmbH · www.partmaster.de

Abstract

Hier wird die Abstraktion 'reduced' vorgestellt, die UI und Logik einer Flutter-App vom verwendeten State Management Framework entkoppelt. 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. Dem soll die Abstraktion entgegen wirken. 'reduced' basiert auf dem Konzept der Funktionalen Programmierung mit unveränderlichen Zustandsobjekten sowie der Kombination der Entwurfsmuster "State Reducer" und "Humble Object" und verwendet die Bausteine AppState, Reducer und Reducible sowie Binder, Builder, Props und Transformer, die im Folgenden erklärt werden. Die entstehende Code-Struktur ist einfacher lesbar, testbar, skalierbar und kompatibel zu verbreiteten State Management Frameworks, wie Riverpod [^4] oder Bloc [^5].
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
Das Humble-Object-Pattern für Flutter

Flutter [^3] beschreibt sich selbst mit dem Spruch "Fast alles ist ein Widget" [^9]. 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.

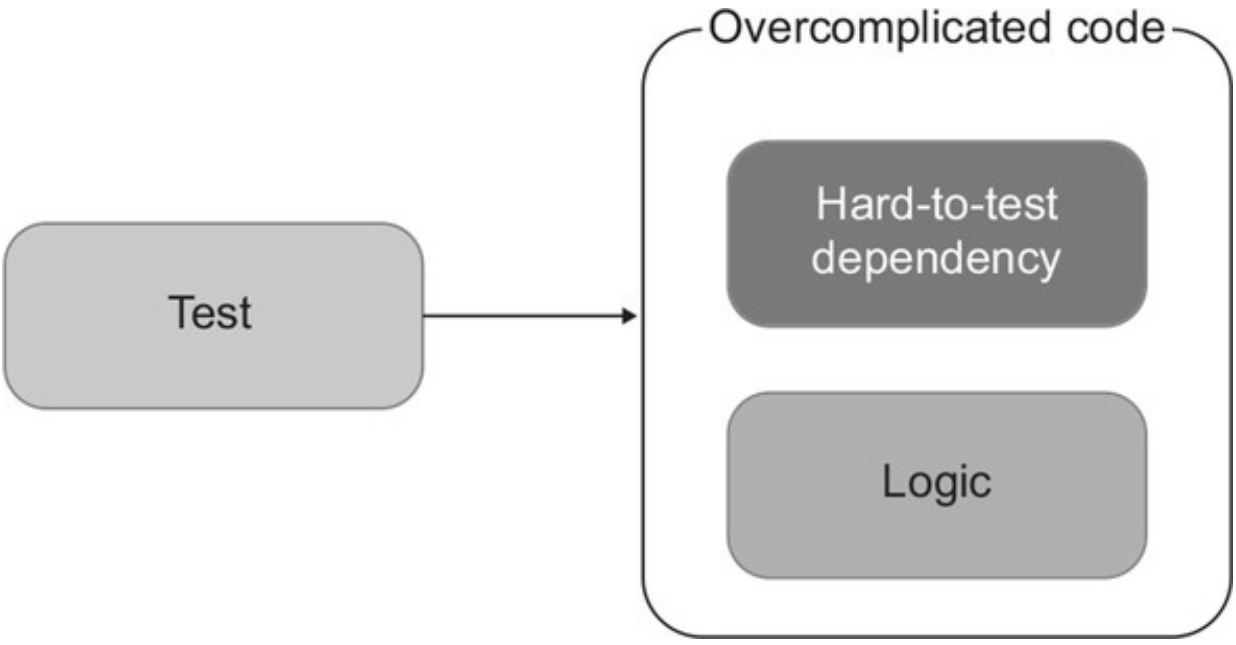
Ein Entwurfmuster, das genau auf diese Problemlage passt, ist das Humble-Object-Pattern [^1] von Micheal Feathers.

Das Humble-Object-Pattern

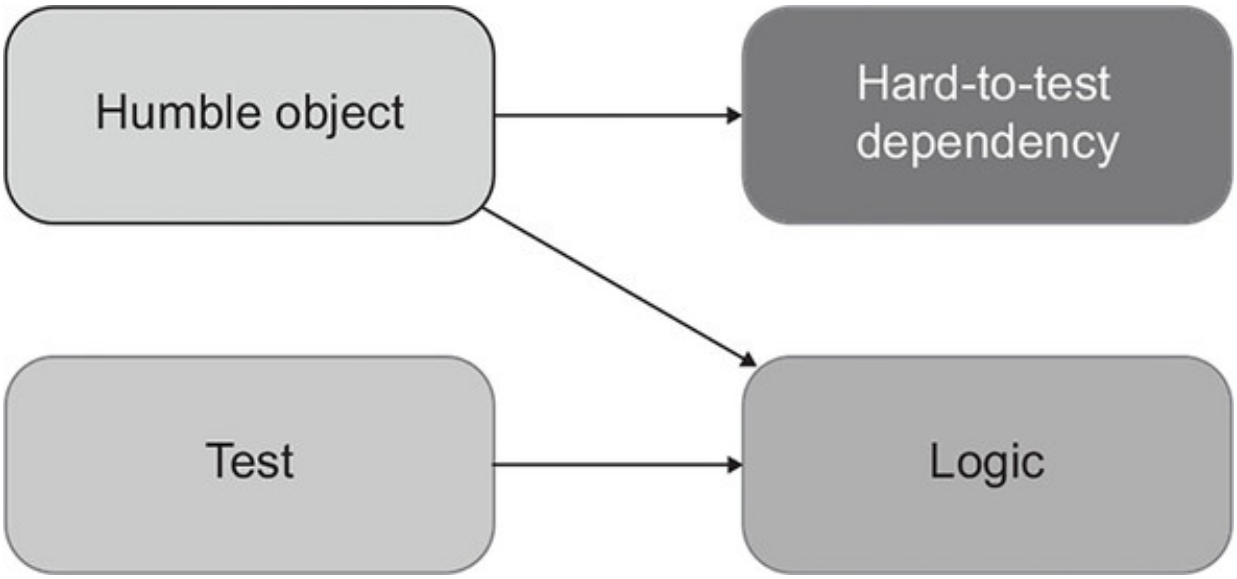
Die Zusammenfassung des Humble-Object-Pattern lautet:

> 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 App-Zustands-Verwaltungs-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.

```
class MyAppState {
  const MyAppState({required this.title, this.counter = 0});

  final String title;
  final int counter;

  MyAppState copyWith({String? title, int? counter}) => MyAppState(
        title: title ?? this.title,
        counter: counter ?? this.counter,
      );

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

MyAppStateBinder

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

```
class MyAppStateBinder extends StatelessWidget {
  const MyAppStateBinder({super.key, required this.child});

  final Widget child;

  @override
  Widget build(context) => wrapWithProvider(
        initialState: const MyAppState(title: 'flutter_bloc'),
        child: child,
      );
}
```

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.

```
class MyHomePageBinder extends StatelessWidget {
  const MyHomePageBinder({super.key});

  @override
  Widget build(context) =>
      context.bloc<MyAppState>().wrapWithConsumer(
            builder: MyHomePageBuilder.new,
            transformer: MyHomePagePropsTransformer.transform,
          );
}
```

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 werden `transform`-Funktionen definiert, die den eigentlichen App-Zustand in die jeweiligen Property-Klassen transformieren können.

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 App-Zustands-Verwaltungs-Instanz bindet.

```
extension IncrementCounterReducerOnReducible
    on Reducible<MyAppState> {
  ReducerOnReducible get incrementCounterReducer =>
      ReducerOnReducible(this, IncrementCounterReducer.instance);
}
```

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),
        ),
      );
}
```

Zwischenfazit

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 App-Zustands-Verwaltung 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-Änderungens-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 App-Zustands-Verwaltungsssystem mit den bereits erwähnten Interfaces `Reducible`, `Reducer` und `Callable`, der Klasse `ReducerOnReducible`, den Funktionen `wrapWithProvider` und `wrapWithConsumer` sowie weiteren Artefakten vorstellen.

Teil 2

Eine Abstraktion für die App-Zustands-

Verwaltung

Bei allen fünf mittels des Humble-Object-Pattern extrahierten Verantwortlichkeiten handelt es sich um App-Zustands-Verwaltungs-Verantwortlichkeiten.
In der Counter-Demo-App wird die App-Zustands-Verwaltung mit einem StatefulWidget implementiert. Das StatefulWidget und das InheritedWidget sind die beiden von Flutter bereitgestellten Bausteine für die App-Zustands-Verwaltung. Diese beiden Bausteine sind Low-Level-Bausteine. Nicht-triviale Apps benötigen meist eine höherwertige Lösung für die App-Zustands-Verwaltung. In der Flutter-Community sind viele Frameworks enstanden, um diesen Bedarf zu decken. In der offiziellen Flutter-Dokumentation sind aktuell 13 solcher App-Zustands-Verwaltung-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.
Die wesentlichen Bestandteile so einer 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 App-Zustands-Verwaltungs-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 App-Zustands-Verwaltungs-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 App-Zustands-Verwaltungs-Instanz zur Verfügung stellt

6. 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

Nach dieser Übersicht folgen nun die Details zu den Bestandteilen der Abstraktion für State Management Frameworks.

Interface Reducer

zur Definition von Operationen zur Änderung des App-Zustandes

Interface Reducible

zum Lesen und Aktualisieren des App-Zustands in einer App-Zustands-Verwaltungs-Instanz

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

Klasse ReducerOnReducible

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

Funktion wrapWithProvider

zum Einpacken eines Widgets in ein sogenanntes Provider-Widget, welches im Widget-Baum den Zugriff auf eine App-Zustands-Verwaltungs-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

Zweites Zwischenfazit

Da diese Abstraktion 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.

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