# Eine vorteilhafte Code-Struktur für Flutter-Apps

### **Autor**

Steffen Nowacki · PartMaster GmbH · www.partmaster.de

## **Abstract**

Hier wird eine Code-Struktur vorgestellt, die durch die Anwendung von Entwurfsmustern <sup>1</sup> die Trennung der Verantwortlichkeiten <sup>2</sup> im Code von App-Projekten mit Flutter <sup>3</sup> befördert. Für die Code-Struktur werden die Bausteine Binder, Builder und Props sowie AppState, Reducer und Reduceable verwendet, die im Folgenden erklärt werden. Die Code-Struktur ist gut testbar, skalierbar und vereinbar mit verbreiteten App-Zustands-Verwaltungs-Lösungen, wie Riverpod <sup>4</sup> oder Bloc <sup>5</sup>. Und sie kann auch für eigene Lösungen auf Basis der eingebauten Flutter-Mittel StatefulWidget und InheritedWidget eingesetzt werden.

# **Einleitung**

Source Code scheint dem 2. Gesetz der Thermodynamik zu folgen und zur Aufrechterhaltung der Ordnung der ständigen Zuführung von äußerer Energie zu bedürfen. Flutter-App-Projekte sind da keine Ausnahme. Ein typisches Symptom sind build-Methoden mit wachsenden Widget-Konstruktor-Hierarchien, die von App-Logik infiltriert werden. Mit App-Logik meine ich die Fachlogik der UI und ihres Zustands im engeren Sinn - in Abgrenzung zur Fachlogik einer Anwendungsdomäne, die oft in Datenbanken, Backends oder externen Bibliotheken implememntiert ist. Viele Flutter-Frameworks wurden und werden entwickelt, um eine saubere Code-Struktur zu unterstützen. Dabei geht es hauptsächlich um das Verwalten des Zustandes der App als Voraussetzung für eine Trennung der Verantwortlichkeiten zwischen App-Logik und UI.

Bei einem unreflektierten Einsatz solcher Frameworks besteht die Gefahr, dass sie neben ihrer eigentlichen Aufgabe, der Trennung der Verantwortlichkeiten, die App-Logik und die UI infiltrieren. Weil es so viele Frameworks gibt (in der offiziellen Flutter-Dokumentation sind aktuell 13 Frameworks gelistet <sup>6</sup>) und die Entwicklung sicher noch nicht abgeschlossen ist, kann es besonders für große und langlebige App-Projekte zur Herausforderung werden, zwischen Frameworks migrieren oder verschiedene Frameworks intergrieren zu müssen.

Im Folgenden wird eine Code-Struktur für Flutter-Apps vorgestellt, die solche unerwünschten Infiltrationen vermeidet und so die Qualität von App-Logik- und UI-Code zu verbessert. Dabei geht es ausdrücklich nicht um die Einführung eines weiteren Frameworks sondern um die abgestimmte Anwendung von zwei Entwurfsmustern, Humble Object Pattern <sup>7</sup> und State

Flutter beschreibt sich selbst mit dem Spruch "Alles ist ein Widget" <sup>9</sup>. 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 mit einer kleinen Kehrseite: Wenn man nicht aufpasst, vermischen sich in den resultierenden Widget-Bäumen schnell die Verantwortlichkeiten.

Die Verantwortlichkeiten um die es in diesem Dokument geht, sind einerseits die klassischen UI-Aufgaben:

- 1. Layout,
- 2. Rendering,
- 3. Gestenerkennung

und andererseits Aufgaben die sich auf den App-Zustand beziehen:

1. Benachrichtigung über App-Zustands-Änderungen,

Reducer Pattern <sup>8</sup>, auf den Flutter-App-Code.

- 2. Konvertierung des App-Zustandes in Anzeige-Properties,
- 3. Abbildung von Gesten auf App-Zustands-Operationen.

Die UI-Aufgaben sind eng an eine Umgebung gebunden, in der User Interfaces ablaufen können, wohingegen die Logik in den App-Zustands-Aufgaben nicht unbedingt an eine UI-Ablaufumgebung gebunden ist.

## **Builder, Binder und Props**

Das erste Ziel ist, den Flutter-Code so zu strukturieren, dass im Widget-Baum die Ul-Aufgaben streng von den App-Zustands-Aufgaben separiert werden. Um das zu erreichen, wird das Humble Object Pattern von angewandt. 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.

Auf eine Widget-Klasse bezogen heißt das:

- 1. Wenn die Widget-Klasse sowohl UI-Aufgaben als auch App-Zustands-Aufgaben löst, dann wird diese Widget-Klasse in eine Builder-Klasse und eine Binder-Klasse geteilt.
- 2. Die Binder-Klasse lässt sich über App-Zustandsänderngen benachrichtigen, konvertiert den aktuellen App-Zustand in die Properties für die Builder-Klassse, stellt für die Nutzergesten die Callback-Objekte mit den App-Zustands-Operationen zur Verfügung und liefert in der build-Methode ein Widget der Builder-Klasse zurück.
- Die Builder-Klasse ist ein StatelessWidget. Sie bekommt von der Binder-Klasse im Konstruktor die vorkonfektionierten Properties und Callbacks und erzeugt in der build-Methode einen Widget-Baum aus Layout-, Renderer und Gestenerkennungs-Widgets.
- 4. Für die vorkonfektionierten Properties und Callback wird eine Props-Klasse definiert eine reine Datenklasse mit ausschließlich finalen Feldern und einem const Konstruktor.

# AppState, Reducer und Reduceable

Das zweite Ziel ist, den Code so zu strukturieren, dass die App-Logik streng vom eingesetzten Zustands-Verwaltungs-Framework und von Flutter insgesamt separiert wird. Um das zu erreichen, wird das State Reducer Pattern von angewandt. Die Zusammenfassung des State Reducer Pattern lautet  $\frac{10}{10}$ :

Anstatt sich auf Repository-Klassen zu verlassen, die ständig wechselnde Statuswerte enthalten, sind Reducer reine Funktionen <sup>11</sup>, die eine Aktion und einen vorherigen Status aufnehmen und einen neuen Status auf der Grundlage dieser Eingaben ausgeben.

#### 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 App-Zustands-Verwaltung sellt eine get-Methode für den aktuellen App-Zustand zur Verfügung.
- 2. Die App-Zustands-Verwaltung stellt eine reduce-Methode zur Verfügung, die einen Reducer als Parameter akzeptiert. Ein Reducer ist eine reine synchrone Funktion, die eine Instanz der AppState-Klasse als Parameter bekommt und eine neue Instanz der AppState-Klasse als Returnwert zurückgibt. Beim Aufruf führt die reduce-Methode den übergebenen Reducer mit dem aktellen App-Zustand als Parameter aus und speichert den Returnwert des Reducer-Aufrufs als neuen App-Zustand ab.
- 3. Die App-Zustands-Verwaltung stellt eine Möglichkeit zur Verfügung, sich über Zustandsänderungen benachrichtigen zu lassen. In der minimalen Variante recht es aus, wenn als Benachrichtigung in einem Widget ein setState oder markNeedsBuild ausgelöst wird. Diese Benachrichtigung sollte auch eine selektiv nur für ausgesuchte

Änderungen möglich sein.

Für die ersten beiden Anforderungen lässt sich leicht eine minimale Schnittstelle definieren:

- 1. eine get-Methode für den App-Zustand
- 2. eine reduce-Methode zum Ändern des App-Zustands

```
abstract class Reduceable<S> {
   S get state;
   Reduce<S> get reduce;
}

typedef Reduce<S> = void Function(Reducer<S>);

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

Die dritte Anforderung ist stark vom eingesetzten App-Zustands-Verwaltungs-Framework abhängig (insbesondere die selektive Benachrichtigung) und wird später für ausgewählte Lösungen (StatefulWidget/InheritedWidget, Riverpod, Bloc) diskutiert.

Mit Hilfe des vorgestellten Konzepts mit den Klassen AppState, Reducer, Reduceable sollte es möglich sein, die App-Logik komplett vom ausgewählten Zustands-Verwaltungs-Framework zu entkoppeln. Die App-Logik wird hauptsächlich in Form von verschiedenen Reducer-Implementierungen bereitgestellt. Hinzu kommen Konverter, die aus einem Reduceable und den Reducern die verschiedenen Props-Klassen für die Builder-Widgets aus dem vorherigen Kapitel und für die selektiven Benachrichtigungen aus diesem Kapitel konstruieren können.

# **Beispiel**

Nun soll die vorgestellte Code-Struktur an einem praktischen Beispiel illustriert werden.

## **Ausgangslage**

Als Vorlage wird das wohlbekannte Flutter-Counter-App-Projekt verwendet. In diesem Projekt spielt die Klasse \_MyHomePageState die zentrale Rolle:

```
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 und trägt die verschiedensten Verantwortungen. Hier einige Beispiele:

#### 1. Layout

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

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

3. Gestenerkennng

```
onPressed:
```

4. App-Zustands-Speicherung

```
int _counter = 0;
```

5. Bereitstellung von Operationen für App-Zustands-Änderungen

```
void _incrementCounter() {
```

6. Widget-Benachrichtigung nach App-Zustands-Änderungen

```
setState(() {
```

7. Konvertierung des App-Zustands in Anzeige-Properties

```
'$_counter',
```

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

```
onPressed: _incrementCounter,
```

## **Props**

Wir beginnen mit den Props, einer Klasse mit allen in der build-Methode zur Erstellung des Widget-Baums benötigten Properties. Die Property-Namen in der Props-Klasse werden so gewählt, dass eine leichte Zuordnung zu den Properties im Widget-Baum, denen sie zugewiesen werden sollen, möglich ist. Für die oben gezeigte Klasse \_MyHomePageState könnte die zugehörige Props-Klasse so aussehen:

```
class MyHomePageProps {
  final String title;
  final String counterText;
  final Callable<void> onIncrementPressed;
 MyHomePageProps({
    required this.title,
    required this counterText,
    required this.onIncrementPressed,
 });
  MyHomePageProps.reduceable(ReduceableMyAppState reduceable)
      : title = reduceable.state.title,
        counterText = '${reduceable.state.counter}',
        onIncrementPressed = VoidCallable(
          reduceable,
          IncrementCounterReducer(),
        );
  @override
  int get hashCode => hash3(title, counterText, onIncrementPressed);
 @override
  bool operator ==(Object other) =>
      other is MyHomePageProps &&
      title == other.title &&
      counterText == other.counterText &&
      onIncrementPressed == other.onIncrementPressed;
}
```

Um aus dem App-Zustand einen Props-Wert zu erzeugen, bedarf es einer Konvertierung. Diese Konvertierung ist in Form des Props-Konstruktors

MyHomePageProps.reduceable implementiert. In diesem Konvertierungs-Konstruktor wird festgelegt, wie App-Zustands-Werte für die Anzeige formatiert werden und auf welche App-Zustands-Operationen die Nutzereingaben abgebildet werden.

Damit die Props als Wert für selektive Benachrichtigungen über App-Zustands-Änderungen eingesetzt werden können, müssen die <a href="https://hashcode.com/hashcode">hashCode</a> und <a href="https://hashcode.com/ha

VoidCallable mit überschriebenen hashCode und operator== Methoden.

```
abstract class Callable<T> {
 T call();
}
class VoidCallable<S> extends Callable<void> {
 VoidCallable(this.reduceable, this.reducer);
  final Reduceable<S> reduceable;
  final Reducer<S> reducer;
 @override
 void call() => reduceable.reduce(reducer);
 @override
 int get hashCode => hash2(reduceable, reducer);
 @override
 bool operator ==(Object other) =>
      other is VoidCallable<S> &&
      reducer == other.reducer &&
      reduceable == other.reduceable;
}
```

#### **Builder**

Die Builder-Klasse ist für den Bau des Widget-Baums aus Layout-, Rendering- und Gestenerkennungs-Widgets verantwortlich. Dazu verwendet sie ein Property von der korrespondierenden Props-Klasse, das ihr im Konstruktor übergeben wird. Mit Hilfe der vorkonfektionierten Properties in den Props lässt sich nun leicht eine stateless Builder-Klasse für den Widget-Baum definieren.

```
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.headline4,
              ),
            ],
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: props.onIncrementPressed,
          tooltip: 'Increment',
          child: const Icon(Icons.add),
        ),
      );
}
```

#### **Binder**

Die Binder-Klasse ist für die Binding an den App-Zustand verantwortlich. Sie muss dafür sorgen, dass bei einer Änderung des App-Zustandes neue Props erzeugt werden und mit diesen Props eine neue Instanz der Builder-Klasse gebaut wird. Wie sie das umsetzt, hängt vom verwendeten App-Zustands-Verwaltungs-Framework ab. Für das Beipiel nutzen wir für den Anfang kein externes Framework wie Riverpod oder Bloc, sondern nur StatefulWidget und InheritedWidget. Die in der build-Methode von MyHomePageBinder verwendete InheritedValueWidget-Klasse, eine Ableitung der InheritedWidget-Klasse, wird im Abschnitt App-Zustandsverwaltung vorgestellt.

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

@override
Widget build(context) => MyHomePageBuilder(
     props: InheritedValueWidget.of<MyHomePageProps>(context),
     );
}
```

Die Binder-Klasse setzt also in der abgebildeten Implementierung voraus, dass das InheritedWidget InheritedValueWidget die aktuellen Props bereitstellt und dass ein StatefulWidget das InheritedWidget mit dem aktuellen App-Zustand versorgt. Unter diesen Voraussetzungen ist die Binder-Klasse seht einfach.

## **AppState**

Der AppState beinhaltet das Property counter. Da im Flutter-Counter-App-Projekt auch der title von außen an das \_MyHomePageState-Widget übergeben wird, habe ich es ebenfalls in den AppState mit aufgenommen, obwohl der title nie geändert wird.

```
class MyAppState {
  const MyAppState({required this.title, required this.counter});
  final String title;
  final int counter;
  MyAppState copyWith({String? title, int? counter}) => MyAppState(
        title: title ?? this.title,
        counter: counter ?? this.counter,
      );
 @override
  int get hashCode => hash2(title, counter);
  @override
  bool operator ==(Object other) =>
      other is MyAppState &&
      title == other.title &&
      counter == other.counter;
}
```

#### Reducer

Die Klasse \_MyHomePageState aus dem Flutter-Counter-App-Projekt hat für Änderungen

am App-Zustand nur die eine Methode incrementCounter. Da es sich hier um App-Logik in einer Klasse mit Ul-Aufgaben handelt, wird der Code in eine neue Klasse IncrementCounterReducer ausgelagert. Von der Basisklasse Reducer wird für alle Reducer-Implementierungen die reducer-Methode call mit der Signatur S call(S) vorgegeben. Als konkrete App-Logik wird in der Methode call in der Klasse IncrementCounterReducer das Property counter im AppState inkrementiert. Einige AppState-Verwaltungs-Frameworks entkoppeln die UI von der direkten Zuordnung eines Ereignisses (meist eine erkannte Geste) zu einer App-Logik-Operation, indem die UI Event-Instanzen versendet, um eine Anforderung über eine Operation am AppState auszulösen. In solchen Frameworks mit Event-Klassen können die Reducer-Instanzen als Events verwendet werden. Es gibt in der hier vorgestellten Code-Struktur trotzdem kein Kopplungsproblem, weil wir diese Entkopplung schon durch die Callback-Properties in den Props-Klassen erreichen. Wegen der potenziellen Verwendung als Events müssen die Reducer-Instanzen Wertsemantik haben. Wenn sichergestellt ist, dass es für eine Reducer-Klasse nur eine Instanz gibt, genügen die geerbten hashCode und operator== Methoden.

```
class IncrementCounterReducer extends Reducer<MyAppState> {
   IncrementCounterReducer._();
   factory IncrementCounterReducer() => instance;

static final instance = IncrementCounterReducer._();

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

## **App-Zustandsverwaltung**

Die Umsetzung der App-Zustands-Verwaltung für die Beispiel-App ist stark vom verwendeten App-Zustands-Verwaltungs-Framework abhängig. Zunächst soll eine Umsetzung mit StatefulWidget und InheritedWidget, also ohne externes Framework gezeigt werden. Für die Implementierung der App-Zustands-Verwaltung mit einer Kombination aus StatefulWidget und InheritedWidget werden die Hilfsklassen AppStateBinder und InhritedValueWidget eingeführt.

## **AppStateBinder**

Die Klasse AppStateBinder erbt von StatefultWidget und hält im Feld \_state den veränderlichen App-Zustand. Sie stellt eine get-state-Methode für den App-Zustand sowie eine reduce-Methode, um den App-Zustand von außen mit einem Reducer verändern zu können, bereit. Die beiden Methoden get state und reduce werden in ein

Reduceable verpackt und sind in der hier vorgestellten Code-Struktur die generelle Schnittstelle der App-Zustands-Verwaltung nach außen. Das Reduceable wird von AppStateBinder an den im Konstruktor hereingereichten builder zum Bau des child-Widgets übergeben. Dieser builder wird im später vorgestellten MyAppStateBinder dazu genutzt werden, InheritedWidget-Kinder zu erzeugen und sie dabei mit dem Reduceable zu versorgen.

```
typedef ReduceableWidgetBuilder<S> = Widget Function(
  Reduceable<S> value,
  Widget child,
);
```

```
class AppStateBinder<S> extends StatefulWidget {
  const AppStateBinder({
    super.key,
    required this.initialState,
    required this.child,
    required this.builder,
});

final S initialState;
final Widget child;
final ReduceableWidgetBuilder<S> builder;

@override
State<AppStateBinder> createState() =>
    _AppStateBinderState<S>(initialState);
}
```

#### InhritedValueWidget

Die Klasse InheritedValueWidget erbt von InheritedWidget. Sie bekommt im Konstruktor einen Wert value, speichert ihn in einem gleichnamigen finalen Feld und stellt diesen Wert nachfolgenden Kind- und Kindeskind-Widgets über eine of-Methode zur Verfügung.

```
class InheritedValueWidget<V> extends InheritedWidget {
  const InheritedValueWidget({
    super.key,
    required super.child,
    required this.value,
 });
  final V value;
  static V of<V>(BuildContext context) =>
      _widgetOf<InheritedValueWidget<V>>(context).value;
  static W _widgetOf<W extends InheritedValueWidget>(
          BuildContext context) =>
      context.dependOnInheritedWidgetOfExactType<W>()!;
  @override
  bool updateShouldNotify(InheritedValueWidget oldWidget) =>
      value != oldWidget.value;
}
```

## MyAppStateBinder

Die Klasse MyAppStateBinder kapselt in der hier vorgestellten Code-Struktur die App-Zustands-Verwaltung, bei der es sich im ein Framework, wie Riverpod oder Bloc, oder um eine eigene Implementierung handeln kann. Als Beispiel implementieren wir mit Hilfe von AppStateBinder und InheritedValueWidget selbst.

```
class MyAppStateBinder extends StatelessWidget {
  const MyAppStateBinder({super.key, required this.child});
  final MyAppState state = const MyAppState(
    title: 'Flutter Demo Home Page',
    counter: 0,
  );
 final Widget child;
  @override
  Widget build(context) => AppStateBinder(
        initialState: state,
        child: child,
        builder: (value, child) => InheritedValueWidget(
          value: MyHomePageProps.reduceable(value),
          child: InheritedValueWidget(
            value: MyCounterWidgetProps.reduceable(value),
            child: child,
          ),
       ),
      );
}
```

Das MyAppStateBinder -Widget bildet die Wurzel der Widget-Hierarchie der App und ist für die Bindung der App-Zustands-Verwaltung an die UI verantwortlich. Es stellt der nachfolgenden Widget-Hierarchie den Zugang zur App-Zustands-Verwaltung in Form des Interfaces Reduceable bereit. In diesem Fall haben wir die App-Zustands-Verwaltung einer Kombination aus StatefulWidget und InheritedWidget selbst implementiert und stellen das Interface Reduceable, wie bei InheritedWidgets üblich, mit einer statischen Methode T of<T>(BuildContext context) bereit.

#### main

In der main-Funktion werden Instanzen der Klassen MyAppStateBinder für die App-Zustands-Verwaltung und MyAppBuilder für die UI erzeugt, miteinander verknüpft und die App gestartet.

```
void main() {
  runApp(const MyAppStateBinder(child: MyAppBuilder()));
}
```

# **Zugabe**

Mit der main-Funktion könnte das Beispiel und dieser Artikel enden. Doch es gibt noch eine Verlängerung, und auf die passt der Titel des Films "Das Beste kommt zum Schluss":

- 1. Wir schreiben für die umgestellte Counter-Demo-App einen UI-Test und einen App-Logik-Test, um zu zeigen, wie die Trennung der Verantwortlichkeiten im Code für eine Vereinfachung der Tests genutzt werden kann.
- 2. Wir extrahieren aus der Klasse MyHomePageBuilder eine Klasse MyCounterBuilder, die nur die Anzeige des Zählerwerts enthält, und implementieren für beide Klassen individuelle selektive Bindungen an den App-Zustand, so dass die build-Methoden von MyHomePageBuilder und MyCounterBuilder nur ausgeführt werden, wenn sich ihre Props tatsächlich ändern, um zu zeigen, dass die neue Code-Struktur einfach skalierbar ist.
- 3. Wir portieren die umgestellte Counter-Demo-App von der eigenen App-Zustands-Verwaltungs-Implementierung nacheinander auf die Nutzung der Frameworks Riverpod und Bloc um zu zeigen, dass die neue Code-Struktur die Abhängigkeit vom gewählten Framework tatsächlich veringert und auf die Binder-Klassen beschränkt.

#### **Test**

t.b.d.

## Skalierung

t.b.d.

## **Portierung**

t.b.d.

## Referenzen

- Entwurfmuster

   en.wikipedia.org/wiki/Softwaredesignpattern ←
- Trennung der Verantwortlichkeiten en.wikipedia.org/wiki/Separationofconcerns ←
- 3. Flutter flutter.dev ←
- 4. Riverpod

#### <u>riverpod.dev</u> ←

5. Bloc

<u>bloclibrary.dev</u> ←

6. Flutter State Management Approaches

docs.flutter.dev/development/data-and-backend/state-mgmt/options ←

7. Humble Object Pattern

xunitpatterns.com/Humble Object.html ←

8. State Reducer Pattern

kentcdodds.com/blog/the-state-reducer-pattern ←

9. Alles ist ein Widget

docs.flutter.dev/development/ui/layout ←

10. Zusammenfassung des State Reducer Pattern <u>killalldefects.com/2019/12/28/rise-of-the-reducer-pattern/</u> ←

11. Pure Funktion

en.wikipedia.org/wiki/Pure function ←

12. Wertsemantik

en.wikipedia.org/wiki/Value semantics ←