Design patterns for state management in Flutter

Author

Steffen Nowacki · PartMaster GmbH · www.partmaster.de

Abstract

This article presents the adoption of two design patterns that decouple the UI and logic of a Flutter app from the state management framework used and make the code more concise.

State management frameworks are used to separate UI from logic. They often have the side effect of infiltrating UI and logic, creating unwanted dependencies. This can be counteracted with a combination of the "State Reducer" and "Humble Object" design patterns and the concept of functional programming with immutable state objects. The building blocks used in the implementation of the design patterns `Reducer`, `Reducible` and `Callable` as well as `Binder`, `Builder`, `Props` and `Transformer` are explained in this article. The resulting code structure is easier to read, maintain, and test, and is compatible with popular state management frameworks, such as Riverpod or Bloc. These advantages come at a cost: Compared to the direct use of a state management framework or the use of mutable state objects, there is an additional abstraction layer and more boilerplate code.

If you want to stay flexible when using state management frameworks, or if you want to make your widget tree code structure clearer, or if you just want to get an overview of available state management frameworks, this article might be interesting for you.

Part 1
Responsibilities
in the counter demo app.

Flutter [1] describes itself with the phrase "Almost everything is a widget" [2]. What is meant by this is that most features are implemented in the form of widget classes that can be plugged together like Lego bricks. This is a great feature. But there is also a small downside: if you are not careful, responsibilities easily get mixed up in the resulting widget trees.

Let's take the well-known Counter demo app as an example:

```
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),
      ),
    );
  }
}
```

The `_MyHomePageState` class holds various responsibilities:

1. Layout

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

2. Rendering

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

3. Gesture detection

```
      onPressed:
```

4. App state storage

```
  int _counter = 0;
```

5. App state change operations

```
  void _incrementCounter() {
```

6. Widget rebuilds after app state changes

```
    setState(() {
```

7. Conversion of app state to display properties

```
            '$_counter',
```

8. Mapping gesture callbacks to app state change operations.

```
      onPressed: _incrementCounter,
```

Conclusion on responsibilities of the counter demo app

The `_MyHomePageState` class of the Counter demo app is an example of a class with many responsibilities. For a demonstration of basic Flutter features, it is understandable to omit separation of concerns in favor of compact presentation. As an example for clean code [3] the counter demo app is rather unsuitable in my opinion.

The principle of separation of concerns [4] has long been known. Nevertheless, its enforcement, especially within UI code, is always a challenge in my experience. UI code has the peculiarity that it is tightly bound to its runtime environment, the UI framework, and therefore basically already has an initial complexity. Since this complexity of UI code is inherent and unavoidable, the only goal is to increase it as little as possible. And solutions that cause some boilerplate code [5] can also be helpful here.

Part 2
Applying the Humble Object Pattern

A design pattern that fits this problem exactly is the Humble Object pattern [6] by Micheal Feathers.

The definition of the Humble Object pattern

The summary of the Humble Object pattern is:

> If code is not well testable because it is too tightly coupled with its environment, extract the logic into a separate, easily testable component that is decoupled from its environment.

The following two diagrams in Figs. 1 and 2 illustrate the situation before and after applying this design pattern:

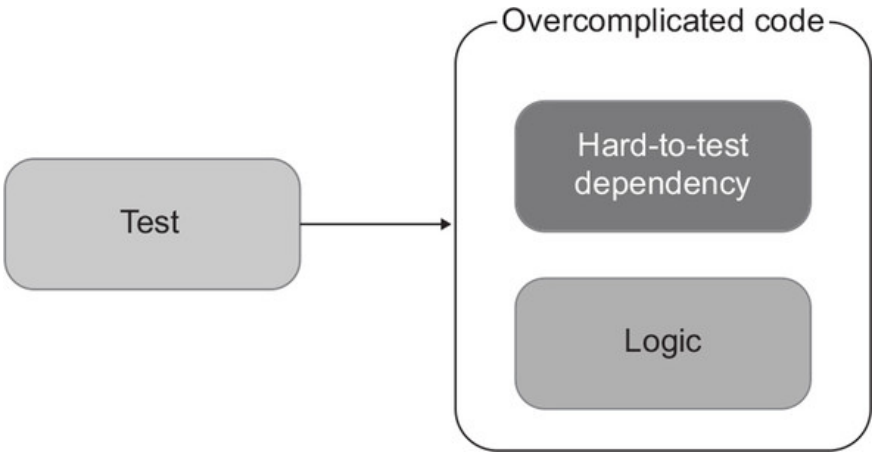

Fig. 1: Situation before applying the Humble Object pattern (image source: manning.com [7]).

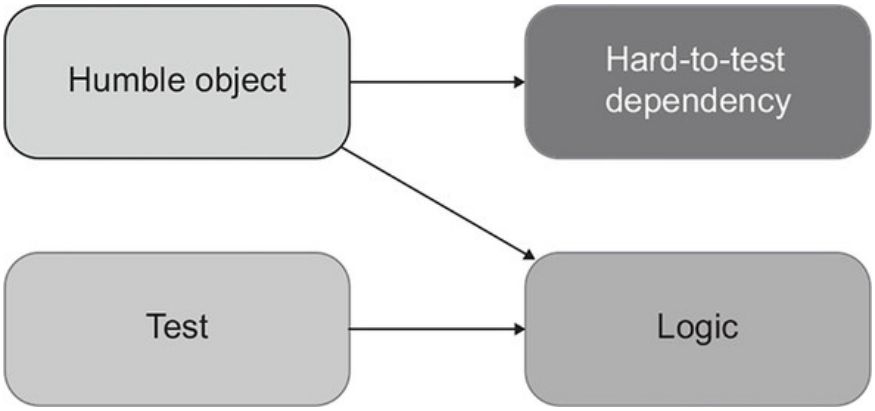

Fig. 2: Situation after applying the Humble Object pattern (image source: manning.com [8]).

Counter demo app refactored

For a small demo app like the Counter demo app, it's fitting that so many responsibilities are grouped into a single class.
Nevertheless, I chose this app to apply the Humble Object pattern to and use the result to evaluate the usefulness of the pattern for Flutter widget trees.

Here is the result of applying the pattern in the form of the new classes in which the various responsibilities (or pieces of logic) have been extracted from the original class `_MyHomePageState` and the remaining Humble Object class `MyHomePageBuilder`.

INotel I---I I In the extracted classes, I used an abstraction for the state management framework consisting of the `Reducible`, `Reducer`, and `Callable` interfaces, the `ReducerOnReducible` class, and the `wrapWithScope`, `wrapWithProvider`, `registerReducible`, and `wrapWithConsumer` functions, which I discuss later. I

App state storage

I have provided two constructs for storing the app state: A class `MyAppState` for the actual app state, A class `MyAppStateBinder` which, depending on the state management framework used

1. either with the function `wrapWithScope` creates a state management scope
2. or with the function `wrapWithProvider` creates a `Reducible` and sets the initial value of the app state
3. or use the `registerReducible` function to create a `Reducible` and set the initial value of the app state.

The `wrapWithScope`, `wrapWithProvider`, and `registerReducible` functions abstract the state management framework used and make it accessible to subsequent widgets in the widget tree. For each abstraction of a state management framework, exactly one of these three functions is provided.

MyAppState

To be able to save the app state, I moved the property `counter` to a class `MyAppState` and added the property `title`, although the property is never changed.

```
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

The `MyAppStateBinder` class binds the specific app state class `MyAppState` to a state management instance using the `wrapWithProvider` function.

```
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 State Change Operations

The Counter demo app has only one app state change operation.

IncrementCounterReducer

The `call` method of the `IncrementCounterReducer` class creates a new `MyAppState` value in which the property `counter` is compared to the `MyAppState` value passed as parameter `state` has been incremented.

The base class `Reducer` defines the signature of the `call` method for all app state change operations:
`MyAppState call(MyAppState state);` and is explained later.

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

  static const instance = IncrementCounterReducer._();

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

Widget rebuilds after app state changes

I have provided the function `wrapWithConsumer` for notification of a necessary rebuild after app state changes.

The `wrapWithConsumer` function abstracts the state management framework used and ensures that the passed `builder` is called whenever the app state changes. The passed `transformer` transforms the actual `MyAppState` into the parameter type expected by the `builder`.

MyHomePageStateBinder

The class `MyHomePageStateBinder` specifies that the widget `MyHomePageBuilder` is built and that the function `MyHomePagePropsTransformer.transform` is used to set the required constructor parameter for fir class `MyHomePageBuilder`.

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

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

Conversion of app state to display properties

When creating the widget tree, some widget constructor properties depend on the current app state. These are combined into their own property classes. In addition, a `transform` function is defined for each property class, which can transform the app state into an instance of the property class.

MyHomePageStateProps

The properties needed in the build method of `MyHomePageBuilder` are grouped in the class `MyHomePageProps`.

```
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,
      );
}
```

Mapping of gesture callbacks to app state change operations

Flutter widgets provide callback properties for gesture processing and similar purposes. We'll treat callback properties the same as the display properties discussed earlier and add them to the same properties class `MyHomePageProps`. The `transform` function generates the value for the `onIncrementPressed` callback property from the app state operation `IncrementCounterReducer`.

To do this, a convenience method `get incrementCounterReducer` is defined, which binds the app state operation to the state management instance using the `ReducerOnReducible` class.

get incrementCounterReducer

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

Layout, rendering and gesture detection

I couldn't separate out the remaining responsibilities of layout, rendering and gesture recognition because they can hardly be separated from the UI framework. They remain in the resulting Humble object in the `MyHomePageStateBuilder` class.

MyHomePageStateBuilder

In the converted Counter demo app, the `MyHomePageBuilder` class makes up the Humble object and is responsible for layout, rendering and gesture detection.

```
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),
        ),
      );
}
```

Conclusion on using the Humble Object pattern

Applying the Humble Object pattern to a Flutter widget class that produces a widget tree and has app state dependencies consists of the following five steps:

1. If the widget class has both UI tasks and solves app state tasks, then this widget class is split into a Builder class, a Binder class, a Props class, and a transform function to create props instances.
2. The Builder class is a `StatelessWidget`. It gets the props instance with ready-made properties and callbacks from the binder class in the constructor and creates a widget tree from layout, renderer and gesture detection widgets in the `build` method.
3. At the state management instance, the Binder class selectively listens for changes to 'its' Props and returns a Builder class widget in the build method.
4. A Props class is defined for the ready-made properties and callbacks of the Builder class - a pure data class with only final fields.
5. A transform function is defined for the Props class, which generates the values for the properties from the current app state and the values for the callbacks from the Reducer implementations.

The diagram in Fig. 3 shows the interaction of the components when implementing the Humble Object Pattern with Binder, Builder, Props and Transformer.

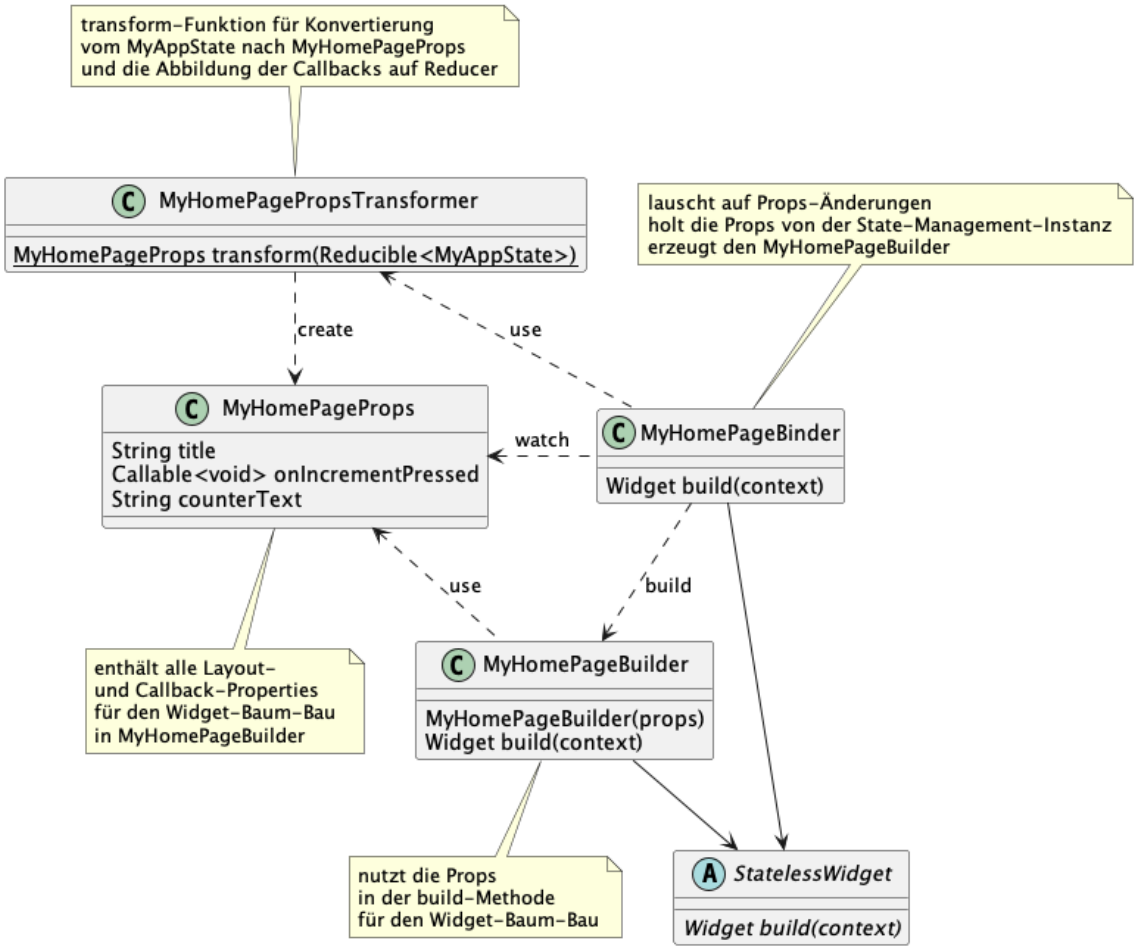

Fig. 3: Implementation of the Humble Object Pattern with Binder, Builder, Props and Transformer

These steps applied to the Counter demo app bring the following result:

Three responsibilities from the `_MyHomePageState` class remain in the Humble Object:

1. Layout
2. Rendering
3. Mapping of gesture callbacks to app state change operations

FFive responsibilities were extracted from the `_MyHomePageState` class into their own classes or functions:

1. App state storage
2. Providing operations for app state changes
3. Widget notification after app state changes
4. Conversion of app state to display properties
5. Mapping of gesture callbacks to app state change operations

The source code of the refactored counter demo app can be found here: github.com/partmaster/reduced/tree/main/examples/counter_app.

In addition to the counter demo app, I also refactored the example project Simple app state management from the official Flutter state management documentation. The result can be found here:github.com/partmaster/reduced/tree/main/examples/shopper_app.

In the classes and functions extracted after the Humble Object Pattern, a lot of boilerplate code was created and a state management abstraction was used. The abstraction consists of the interfaces `Reducible`, `Reducer` and `Callable`, the class `ReducerOnReducible` and the functions `wrapWithScope`, `wrapWithProvider`, `registerReducible` and `wrapWithConsumer`.

I hope this has piqued your interest, as I now want to present the abstraction used for the state management system.

Part 3
Application of the State Reducer Pattern

All five responsibilities extracted using the Humble Object Pattern are state management responsibilities.

In the Counter demo app, state management is implemented with a StatefulWidget. The StatefulWidget and the InheritedWidget are the two state management building blocks provided by Flutter. These two building blocks are low-level building blocks. Non-trivial apps usually require a higher quality state management solution. Many frameworks have emerged in the Flutter community to fill this need. The official Flutter documentation currently lists 13 such state management frameworks [9].

After having painstakingly (and with boilerplate code) removed five responsibilities from their dependency on the UI environment, it is only logical to use a suitable abstraction to protect them from being dependent on a specific state management framework.

I call the abstraction 'reduced' in reference to the underlying pattern. The essential components of the 'reduced' abstraction have already been listed:

1. Interface **Reducer**
   Definition of operations to change the app state.
2. Interface **Reducible**
   Reading and updating app state in a state management instance.
3. Interface **Callable**
   Basis for defining classes with value semantics [10] whose instances can be assigned to callback properties of Flutter widgets,
   for use in the Props classes mentioned in the chapter on using the Humble Object pattern , so these classes can also be defined with value semantics.
4. Klasse **ReducerOnReducible**
   Association of an app state operation with the state management instance on which it is to be executed.
5. Functions **wrapWithScope**, **wrapWithProvider** and **registerReducible**
   The functions `wrapWithScope` , `wrapWithProvider` and `registerReducible` make state management functionality accessible for subsequent widgets in the widget tree. Exactly one of the three functions is provided for each state management framework.
6. Funktion **wrapWithConsumer**
   The `wrapWithConsumer` function ensures that the rebuild of a widget is appropriately triggered by the state management framework.

The definition of the State Reducer pattern

The overview is followed by the detailed description of the 'reduced' abstraction for state management frameworks. At its core, the abstraction is an application of the State Reducer pattern, so this design pattern is presented first.

Put simply, the State Reducer Pattern requires that any change to the app state be performed as an atomic operation with the current app state as a parameter and a new app state as the result. Actions that potentially run longer (database queries, network calls, ..) usually have to be implemented with several atomic app state changes because of this requirement, e.g. one at the beginning of the action and one at the end. It is crucial that the app state change at the end of the action does not (re)use the app state result from the beginning of the action as a parameter, but the then current app state of the state management framework. The pattern supports this intention by ensuring that you do not have to fetch the current app status yourself when there is a change, but that it is delivered without being asked.

Or to put it more analytically: The state reducer pattern models the app state as the result of a Fold function [11] from the initial app state and the sequence of previous app state change actions.

Dan Abramov and Andrew Clark used this concept in the Javascript framework Redux [12] and popularized the name *Reducer* [13] for the combination operator that calculates a new app state from the current app state and an action:

> Reducers are functions that take the current state and an action as arguments and return a new state result.
> In other words: `(state, action) => newState` .

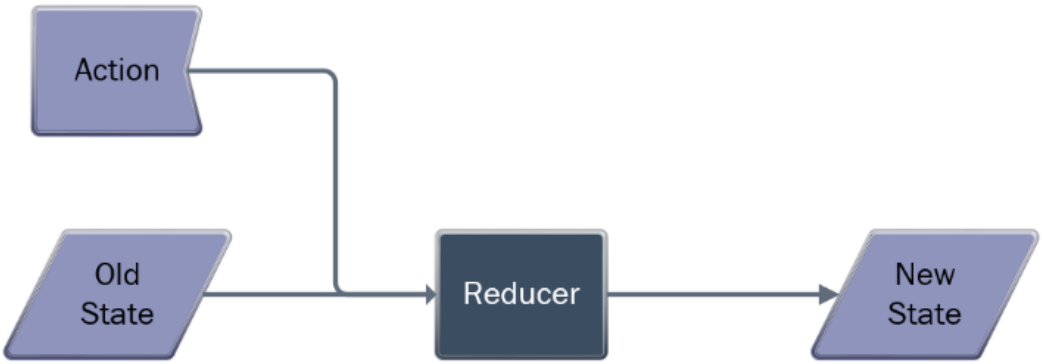

Fig. 4: Principle of the State Reducer Pattern (Image source: killalldefects.com [14])

In terms of app code, this means:

1. An AppState class is defined for the app state - a pure data class with only final fields and a const constructor. The state management API provides a get method for the current app state.
2. The State Management API provides a reduce method that accepts a reducer as a parameter. A reducer is a purely [15] synchronous function that takes an instance of the AppState class as a parameter and returns a new instance of the AppState class as . When called, the reduce method executes the passed reducer with the current app state as a parameter and saves the return value of the reducer call as the new app state.

After presenting the State reducer pattern, the details of the components of the 'reduced' abstraction for state management frameworks now follow.

Interface Reducer

Base interface for the implementations of app state change operations that specifies the signature of the method for performing such operations.

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

In addition to the basic variant of the interface, there are other variants with additional parameters for the app state change operations, e.g. with one parameter:

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

Interface Reducible

Basic interface for state management instances with a getter `get state` for the AppState and a method `reduce` to update the AppState according to the State Reducer pattern.

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

Interface Callable

Base interface for implementations of callbacks. Implementing callbacks as classes rather than functions allows overriding `get hashCode` and `operator==(other)` for value semantics.

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

In addition to the basic variant of the interface, there are other variants with additional parameters for the callbacks, e.g. with one parameter:

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

Class ReducerOnReducible

The class implements the `Callable` interface with a `Reducer` and a `Reducible` by calling the `reduce` method of the reducer with the reducer as parameter is executed.

```
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) => ...
}
```

In addition to the basic variant of the class, there are other variants with additional parameters for the callbacks, e.g. with one parameter:

```
class Reducer1OnReducible<S, V> extends Callable1<void, V> {
  const Reducer1OnReducible(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) => ...
}
```

wrapWithScope, wrapWithProvider, and registerReducible functions

The various state management frameworks differ in how state management functionality is exposed in the widget tree. Three concepts can be distinguished, which are so different that I have modeled them in the abstraction with three different functions:

1. **wrapWithScope**
   The state management frameworks Binder [16] and Riverpod [17] offer so-called 'scope' widgets with which nestable scopes for state management instances can be created.
   This is mapped with the `wrapWithScope` function.

2. **registerReducible**
   The state management frameworks GetIt [18] and GetX [19] do not use any integration of the state management instances in the widget tree but only an independent register of the instances.
   This is mapped with the `registerReducible` function.

3. **wrapWithProvider**
   The remaining state management frameworks examined offer so-called 'provider' widgets with which a state management instance that manages an app state can be created.
   This is mapped with the `wrapWithProvider` function.

Funktion wrapWithConsumer

The `wrapWithConsumer` function ensures that the rebuild of a widget is appropriately triggered by the state management framework.

The function always has a parameter `builder` of type Function with return type Widget. The complete signature of the function depends on the state management framework.

In the implementation, the passed `builder` is often packed into a so-called 'consumer' widget, which executes the `builder` exactly when relevant properties of the state management instance change. So, the `wrapWithConsumer` function makes it possible to selectively listen for state changes.

Conclusion on the application of the State Reducer pattern

Based on the state reducer pattern, a minimal API for state management frameworks was defined, covering the basic state management application scenarios. By reducing it to what is necessary, the API can be easily implemented for existing state management frameworks, as will be shown shortly. The source code for the API can be found here: github.com/partmaster/reduced

Since the 'reduced' API only has to be implemented once for each concrete state management framework, it does not cause any additional boilerplate code, just an additional abstraction layer. But every layer of abstraction also causes expenses that should be weighed against the benefit.
The 'reduced' API only covers the 'standard part' of the APIs of the state management frameworks. In addition to the 'reduced' API, each framework offers individual features. In order to be able to use such features as well, you can extend the 'reduced' API or work directly with the framework API, bypassing the 'reduced' API.
If in a project the need for direct use of the state management framework API does not remain an exceptional case, then it is likely that using an abstraction layer for the state management framework in such a project is unfavorable.

Part 4

Implementation of the 'reduced' API

An implementation of the 'reduced' API for a concrete state management framework consists of the implementation of the interface `Reducible` and the implementation of the function `wrapWithConsumer` and one of the functions `wrapWithScope` ``, wrapWithProvider `or` registerReducible `. Optionally, an extension for the` BuildContext``` can be added, which provides convenient access to the state management instance.
The frameworks 'Bloc' [20] and 'Riverpod' [^17] will be used to show how a 'reduced' implementation looks like.

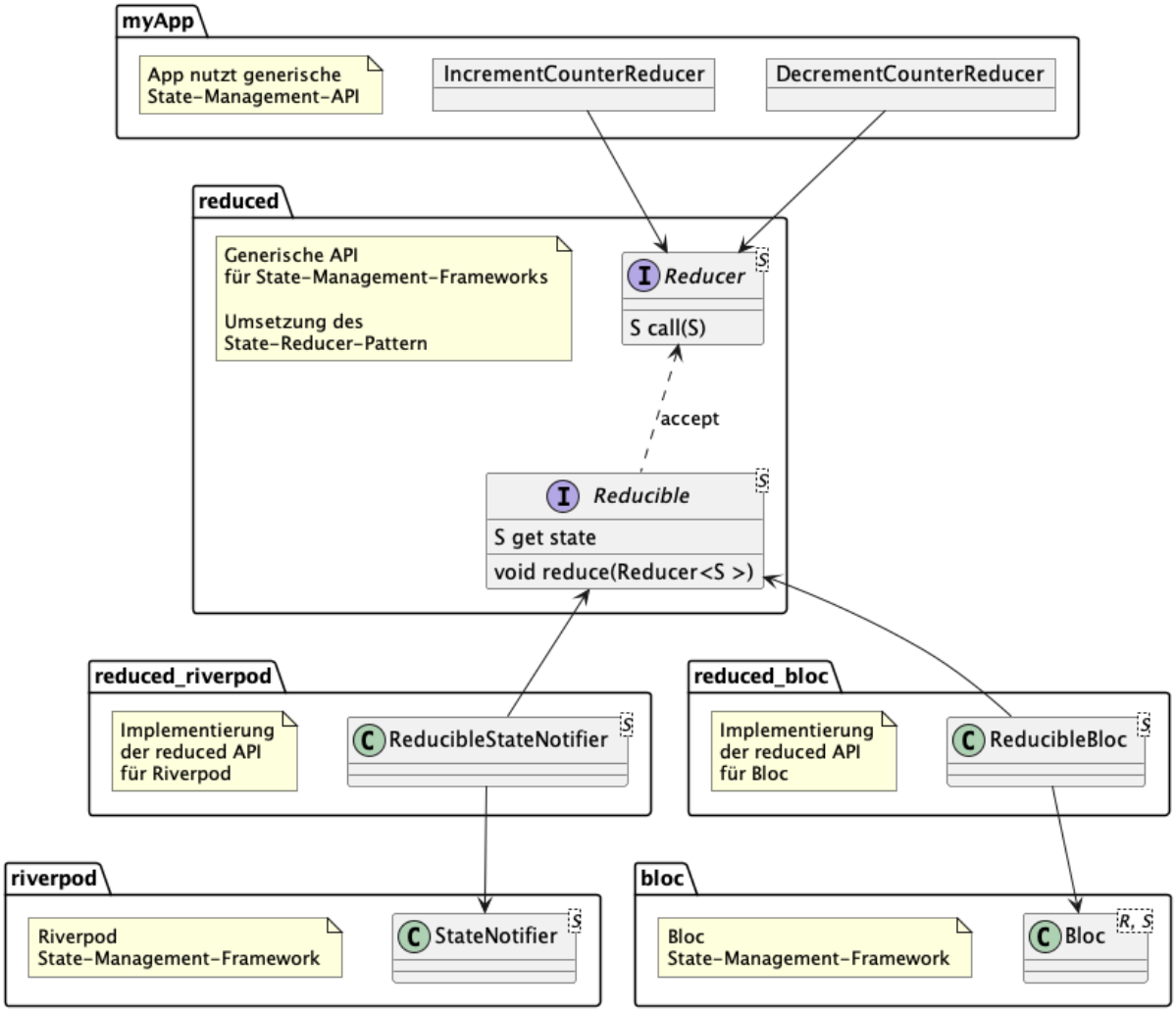

Fig. 5: Relations between 'reduced' API, API implementations and API usage

'reduced' API implementation for Bloc as an example

Felix Angelov's state management framework 'Bloc' [^16] is based on the Bloc pattern [21] by Paolo Soares and Cong Hui.

Reducible implementation for Bloc

The Bloc framework implements state management instances using the class `Bloc<E, S>` , where `E` is the type parameter for state management events and `S` is the type parameter for the state class. We use the `Reducer` interface from the 'reduced' API as the event type. Since the `Reducer` bring their operation on top of the app state, they don't need individual dispatching, they can be executed themselves. The `S get state` method already comes with the `Bloc` class and the `Reducible.reduce` method can be directly mapped to the `Bloc.add` method.

```
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 for the BuildContext

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

wrapWithScope implementation for Bloc

The wrapWithScope function creates the `BlocProvider` widget.

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

wrapWithConsumer implementation for Bloc

The `wrapWithConsumer` function creates the `BlocSelector` widget. The required `Reducible` instance is supplied implicitly by defining `wrapWithConsumer` as an extension of the `ReducibleBloc` class.

```
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),
      );
}
```

'reduced' API implementation for Riverpod as an example

Das State-Management-Framework 'Riverpod' [^17] von Remi Rousselet.

Reducible implementation for Riverpod

The Riverpod framework implements state management instances with the `StateNotifier<S>`` class, where `S` is the type parameter for the state class. The `S` get state `method already brings the` StateNotifier `class and the` Reducible.reduce `method can be easily mapped to the` `set state(S)`` method.

```
class ReducibleStateNotifier<S> extends StateNotifier<S>
    implements Reducible<S> {
  ReducibleStateNotifier(super.state);

  late final Reducible<S> reducible = this;

  @override
  reduce(reducer) => state = reducer(state);
}
```

Extension on the BuildContext

Riverpod does not need an extension on the BuildContext.

wrapWithProvider implementation for Riverpod

The function `wrapWithProvider` creates the widget `ProviderScope`.

```
Widget wrapWithProvider({required Widget child}) =>
    ProviderScope(child: child);
```

wrapWithConsumer implementation for Riverpod

The function `wrapWithConsumer` creates the widget `Consumer`. By means of the consumer, you get a `WidgetRef` with which you get the current Props at every change.

```
Widget wrapWithConsumer<S, P>({
  required StateProvider<P> provider,
  required PropsWidgetBuilder<P> builder,
}) =>
    Consumer(
      builder: (_, ref, __) => builder(props: ref.watch(provider)),
    );
```

Table of the 'reduced' API implementations

The Flutter documentation currently lists 13 state management frameworks. The Fish Redux framework [22] does not offer 'Null Safety' [23] and is therefore obsolete. For the other 12 frameworks, the 'reduced' API was implemented as an example. The following table contains links to these frameworks and their 'reduced' implementations.

Name	Publisher	'reduced' implementation
Binder	romainrastel.com	reduced_binder
Fish Redux	Alibaba	-
Flutter Bloc	bloclibrary.dev	reduced_bloc
Flutter Command	escamoteur	reduced_fluttercommand
Flutter Triple	flutterando.com.br	reduced_fluttertriple
GetIt	fluttercommunity.dev	reduced_getit
GetX	getx.site	reduced_getx
MobX	dart.pixelingene.com	reduced_mobx
Provider	dash-overflow.net	reduced_provider
Redux	brianegan.com	reduced_redux
Riverpod	dash-overflow.net	reduced_riverpod
Solidart	bestofcode.dev	reduced_solidart
States Rebuilder	Mellati Fatah	reduced_statesrebuilder

Conclusion on the implementation of the 'reduced' API

The goal of the 'reduced' API is to provide a minimal abstraction layer for state management frameworks.
The small code size and the direct mappings in the implementations of the 'reduced' API show that this goal has been achieved. The 'reduced' API fits very well with most frameworks.
However, with the state management framework MobX [24], in order to provide the `wrapWithProvider` and `wrapWithConsumer` functions, it was necessary to work against the framework: The 'reduced' API functions are designed to create and use state management instances with generic classes at runtime. In contrast, MobX uses specific state management classes that are already generated with a code generator during the build.

DBy using the 'reduced' API, independence from the state management framework is possible in addition to the separation of responsibilities: the refactored counter demo app runs with all 12 listed and available frameworks. In the file binder.dart, the state management framework used in the counter demo app can be switched by executing the corresponding `export` statement (stripped of the comment characters):

```
// export 'binder/binder_binder.dart';
// export 'binder/bloc_binder.dart';
// export 'binder/fluttercommand_binder.dart';
// export 'binder/fluttertriple_binder.dart';
// export 'binder/getit_binder.dart';
// export 'binder/getx_binder.dart';
// export 'binder/mobx_binder.dart';
// export 'binder/provider_binder.dart';
// export 'binder/redux_binder.dart';
// export 'binder/riverpod_binder.dart';
// export 'binder/setstate_binder.dart';
// export 'binder/solidart_binder.dart';
export 'binder/statesrebuilder_binder.dart';
```

Open ends

Grey zones between UI and app logic

To implement some UI actions, one needs a [BuildContext]. A prominent example is the navigation between app pages with [Navigator.of(BuildContext)]. Deciding when to navigate to which app page is app logic. The app logic should remain as free as possible from dependencies on the UI runtime environment, and a [BuildContext] virtually represents this runtime environment.

A similar problem is UI resources such as images, icons, colors, and fonts, which have a dependency on the UI runtime environment and may require UI app logic to be deployed. So there are still gray zones between UI code and app logic code that should be turned into clear boundaries. (In [Navigator]s case, the [MaterialApp.navigatorKey] property provides a possible workaround for navigating between app pages without [BuildContext]).

Local or nested app states

The approach of modeling the complete app state as an immutable instance of a single class will reach its limits when dealing with very complex data structures, very large data sets, or very frequent change actions [25].

There are hints in the redux.js documentation [26], [27] on how to extend these limits by structuring the app state class properly.

Finally, one can try to extract performance-critical parts from the global app state and implement them with local state management solutions. In the state management framework Fish-Redux [^19], for example, it is always the case that (in addition to a global app state) there is a local state management instance for each app page.

UI code structuring

In this article, code structure and design patterns for separating the responsibilities of UI code and app logic code were discussed.

A separated app logic can be freely further structured using all available architectural approaches and design patterns.

However, for the separated Flutter UI code, further structuring concepts are needed for larger projects:

- How do I separate the theming, e.g. Light Mode and Dark Mode?
- How do I separate the layout adaptations for different device groups, e.g. smartphone, tablet, desktop, printer ?
- How do I separate the layout responsiveness code for app display size changes between adaptation levels ?
- How do I separate the code for animations ?

Testing in practice

The code structure presented in this article is a result of my experience from small and medium flutter projects. One of them is the Cantarei project - the Taizé songs app. The app is freely available in the Apple [28] and Google [29] app stores, so anyone interested can get their own idea of the project sizes for which the concepts proposed here have already been field-tested. Whether they can be successfully applied, as they are, to many and especially to larger projects remains to be proven.

Closing words

In software development, over-motivation is unfavorable. Any abstraction to hide dependencies incurs costs and should yield enough benefits to justify the cost [30]. I hope the 'reduced' abstraction, especially in combination with the application of the Humble Object pattern, is worth the effort.

References

1. Flutter
   flutter.dev ↩
2. Everything is a widget
   docs.flutter.dev/development/ui/layout ↩
3. Clean Code
   de.wikipedia.org/wiki/Clean_Code ↩
4. Separation of concerns
   en.wikipedia.org/wiki/Separation*of*concerns ↩
5. Boilerplate Code de.wikipedia.org/wiki/Boilerplate-Code ↩
6. Humble Object Pattern
   xunitpatterns.com/Humble Object.html ↩
7. Before the Humble Object pattern livebook.manning.com/book/unit-testing/chapter-7/49 ↩
8. After the Humble Object pattern livebook.manning.com/book/unit-testing/chapter-7/51 ↩
9. Flutter State Management Approaches
   docs.flutter.dev/development/data-and-backend/state-mgmt/options ↩
10. Value semantics
    en.wikipedia.org/wiki/Value_semantics ↩
11. Folding function
    www.cs.nott.ac.uk/~gmh/fold.pdf ↩
12. Redux
    redux.js.org/ ↩
13. Reducer Pattern
    redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers ↩
14. Principle of State Reducer pattern (killalldefects.com/2019/12/28/rise-of-the-reducer-pattern/)[https://killalldefects.com/2019/12/28/rise-of-the-reducer-pattern/] ↩
15. Pure function
    en.wikipedia.org/wiki/Pure_function ↩
16. Binder pub.dev/packages/binder ↩
17. Riverpod
    riverpod.dev ↩
18. GetIt pub.dev/packages/get_it ↩
19. GetX pub.dev/packages/get ↩
20. Bloc
    bloclibrary.dev ↩
21. Bloc Pattern
    www.youtube.com/watch?v=PLHln7wHgPE ↩
22. Fish Redux pub.dev/packages/fish_redux ↩
23. Null Safety dart.dev/null-safety#enable-null-safety ↩
24. MobX mobx.netlify.app/ ↩
25. Reducer Pattern Nachteil

twitter.com/acdlite/status/1025408731805184000 ↩

26. Basic reducer structure
    redux.js.org/usage/structuring-reducers/basic-reducer-structure ↩

27. Normalizing state shape
    redux.js.org/usage/structuring-reducers/normalizing-state-shape ↩

28. The app *Cantarei* in the Apple Appstore apps.apple.com/us/app/cantarei/id1624281880 ↩

29. The app *Cantarei* in the Google Playstore play.google.com/store/apps/details?id=de.partmaster.cantarei ↩

30. Unfavorable abstractions twitter.com/remi_rousselet/status/1604603131500941317 ↩