Expertise
and insight
for the future

Ly Hong Hoang

State Management Analyses of the Flutter Application

Metropolia University of Applied Sciences

Bachelor of Engineering

Mobile Solutions

Bachelor's Thesis

17 November 2019

Author Title Number of Pages Date	Ly Hong Hoang State management analyses of the Flutter application 39 pages + 15 appendices 17 November 2019
Degree	Bachelor of Engineering
Degree Program	Information Technology
Professional Major	Mobile Solutions
Instructors	Kari Salo, Principal Lecturer

The purpose of the thesis is to analyze three of the most popular state-management systems: Redux, Scoped Model and BLoC. Another purpose is to recommend which of the three is the most suitable for Flutter. As Flutter is a new development tool, the best practice for state-management has not been analyzed yet. Therefore, the thesis provides the first step in finding the best practice when choosing a state-management system for a Flutter project.

The study was conducted by gathering different properties of multiple GitHub projects. The properties were gathered by analyzing the size of each project as well as evaluating the amount of projects existing in certain size group. The data was then used to create graphs that illustrate the results of the study.

In conclusion, the study shows that BLoC is the most suitable state-management system for Flutter, since BLoC is highly customized for Dart, which is Flutter's own programming language. In addition, the findings of the study could be used as the starting point for future application architects when deciding which state-management to use. Furthermore, the study can be built upon to develop the concept into a generally accepted best practice for the development of Flutter.

Keywords	Scoped Model, Redux, BLoC, Flutter, state-management system

Contents

1 Introduction 1

2 Background 2

2.1 Discussion 2

2.1.1 BLoC 2

2.1.2 Redux 8

2.1.3 Scoped Model 14

2.2 Conclusion 20

3 State Management in Practice 22

3.1 Overview 22

3.2 Discussion 22

3.2.1 Introduction 22

3.2.2 Data Gathering Process 24

3.2.3 Results 26

4 Result Summary 30

5 Discussion 32

6 Conclusion 35

References 37

List of Abbreviations

BLoC	Business logic component. A state-management system that was introduced by Google at the 2018 annual Google developer conference named Google I/O 2018 and is currently recommended by Google for the Flutter application.
DBMS	Database management system. Software for maintaining, querying and updating data and metadata in a database.
OS	Operating system. A set of software products that manages a computer system's hardware and software resources while providing the user with common services.

1 Introduction

The purpose of this document is to analyze three stage-management systems recommended for Flutter by the Flutter Development Team as well as the developer community (Angelov, 2019). Another purpose is to give a recommendation on which system is preferable for most Flutter developers.

At the time of writing this document, the tool is relatively new to the developer community: The first version of Flutter was released on 4th December 2018 (Google Development Team, 2018), two months prior to the initial writing of this document. Thus, suitable and reliable long-term real-case examples of a working Flutter application implementing any of the three recommended state management systems was lacking. Therefore, this document aims to provide an academic perspective as well as simple studies upon this new subject.

This document is by no means a scientific paper or report. The results and suggestions given in this paper are not facts. The thesis simply provides discussion upon the subject and provides a small study to support the author's conclusions. Therefore, readers should thoroughly consider other sources as well before deciding their course of actions.

2 Background

2.1 Discussion

As a state-management system is, at its core, a theory, there have been many state-management systems developed by hobbyists and professionals alike prior to the creation of Flutter. However, at the time of writing this thesis, the official Flutter website and blog posts seems to suggest that the Flutter development team along with the developer community have identified three viable state-management systems that Flutter developers can choose from: BLoC, Scoped Model and Redux (Angelov, 2019). Based on the suggestions from the Flutter team and its community, the thesis will solely be focused on these three systems despite having many other options.

2.1.1 BLoC

2.1.1.1 Theory

The Business Logic Component, or BLoC, was created by Google and announced at Google I/O 2018, which was a developer conference held by Google in California, the United States in 2018. It is a new concept, and to comprehend it, developers must first understand the theory behind the system as well as the basic concepts that BLoC will utilize.

In a nutshell, the BLoC acts as a middleman between the data layer and the UI layer. The BLoC is where all the business logics of an application resides. The basic functionality of the BLoC is to receive data/events from sources of information (i.e. data from a backend or events created by the user's interactions from other UI elements), apply business logic dictated by the developers, which is usually in the form of mapping these data/events to the application's states, and finally publish these states to the UI elements that are interested in these changes (see figure 1). (Opia, 2018).

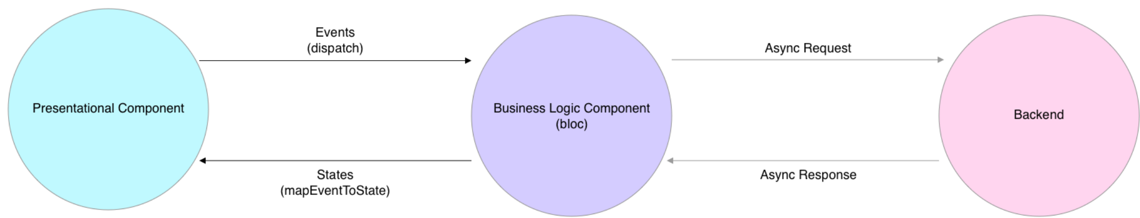

Figure 1. BLoC Architecture (Angelov, 2019)

2.1.1.2 Example

To further demonstrate how the BLoC pattern works with Flutter in practice, the thesis will use a simple example: a common Counter App.

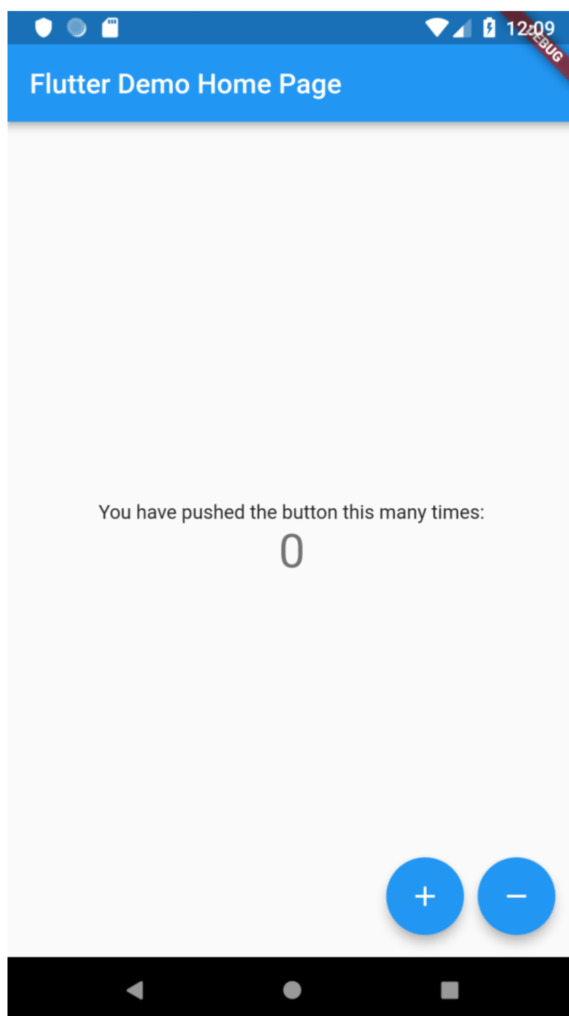

Figure 2. A Counter App

The idea of the Counter App is simple. The users of the application will be given two floating action buttons; the plus button will increment the number in the middle while the minus will decrement the number (see figure 2).

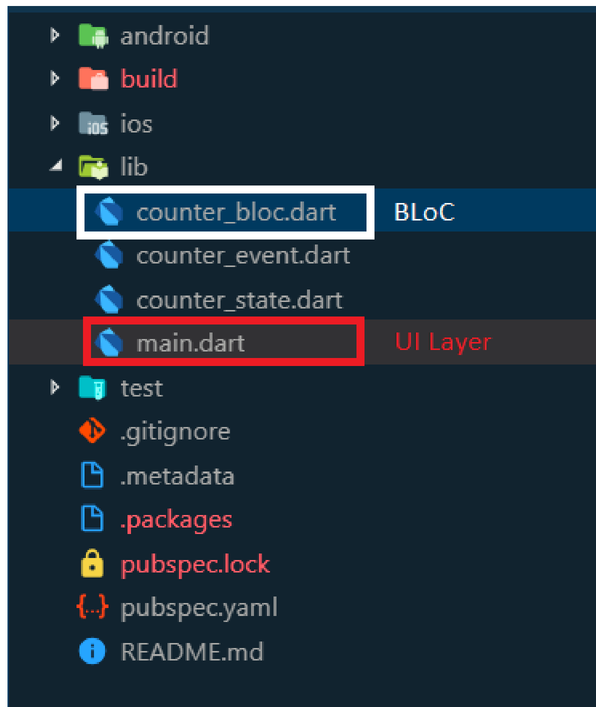

Figure 3. Counter App BLoC Architecture

Mirroring the theory part, the Counter App when implementing the BLoC pattern, will consist of two main components: the BLoC and the UI layer (since it is a simple application, there will be no backend). The BLoC will receive increment or decrement events generated by the UI layer and change the counter number state accordingly. Afterwards the BLoC will publish this state change to the UI layer. The UI layer, on the other hand, will receive interaction events from the users generated by either the plus button or the minus button, and notify BLoC of these events. Furthermore, the UI layer also constantly observes the counter number state exposed by BLoC to re-render the correct UI component as soon as there are any new changes. In addition, besides the two main parts, the application's events and states are also represented by Dart classes to increase clarity (see Appendix 1).

The Counter App will be implementing the flutter_bloc library and therefore will introduce two new concepts: BlocProvider and BlocBuilder. BlocProvider provides the BLoC to its

children using the current context while BlocBuilder handles automatic re-rendering of UI elements when a new app state is provided (Angelov, 2019).

The easiest way to understand how the Counter App works is by simply following the flow of data/events. The flow starts when the user decided to press one of the two buttons from the UI layer.

```
FloatingActionButton(onPressed: () =>
      BlocProvider.of<CounterBloc>(context). onIncrement ()
…
FloatingActionButton(onPressed: () =>
      BlocProvider.of<CounterBloc>(context). onDecrement ()
```

Listing 1. Floating action buttons receiving user's inputs

The two callbacks, onIncrement() and onDecrement(), as showed in listing 1 will then dispatch new events which will notify the BLoC of incoming user interactions as shown below.

```
void onIncrement() {
      dispatch (IncrementEvent());
      }

void onDecrement() {
      dispatch (DecrementEvent());
      }
```

Listing 2. Floating action buttons' callbacks dispatching events

The BLoC will then map these events to the application state by applying any business logic and mutate any inner-state necessary to provide the UI with a new correct application state as shown in listing 3.

```
Stream<CounterState> mapEventToState(
      CounterState currentState,
      CounterEvent event,
      ) async* {
            if (event is IncrementEvent) {
                  yield CounterState(counter: currentState.counter + 1);
            } else if (event is DecrementEvent) {
                  yield CounterState(counter: currentState.counter - 1);
            }
      }
```

Listing 3. Events being mapped

After the new application state has been returned by the BLoC, the UI layer will listen to the changes and automatically re-render the correct component accordingly using flutter_bloc library's BlocBuilder widget (see listing 4).

```
body: BlocBuilder (
      bloc: BlocProvider.of<CounterBloc>(context),
      builder: (context, CounterState state) {
            return Center (
                  child: Column (
                        mainAxisAlignment: MainAxisAlignment.center,
                        children: <Widget> [
                              Text (
                                    'You have pushed the button this many times:',
                              ),
                              Text (
                                    '${state.counter}',
                                    style: Theme.of(context). textTheme.display1,
                              ),
                        ],
                  ),
            );
      })
```

Listing 4. UI update according to state

After the re-rendering is completed, the UI will await further interaction from users and begin the data cycle again.

2.1.1.3 Discussion

Table 1. The BLoC was made by Google, the same developers that have developed Flutter. It was heavily recommended by Google as the go-to state-management system for Flutter (Google Developers, 2018). However, the BLoC is not perfect; there are advantages and disadvantages of using this pattern, which can be seen in table 1.

Summary of advantages and disadvantages of the BLoC

Advantages	Disadvantages
• BLoC was built for Flutter (Hracek, 2018) • Pure functions are often implemented inside the BLoC (Hracek, 2018) • Freedom of methods to introduce BLoC to the UI layer (Hracek, 2018) • Reusability across different platforms (Coca, 2018)	• Large application required many BLoCs (Coca, 2018)

There are many advantages to using the BLoC pattern. Firstly, as mentioned above, the BLoC was built by the same developers that built Flutter and, therefore, the BLoC and Flutter are compatible. Flutter is a declarative and reactive Framework, which means Flutter built its UI to reflect the current state of the application (declarative part). When the state changes, the UI will get rebuilt (reactive part) (Hracek, 2019). Following this principle, BLoC leverages Dart two of the most powerful features: streams and asynchronous functions to make sure the state is reactive with the declarative UIs and to embrace the asynchronous nature of UIs. The second advantage of using the BLoC is in the way the BLoC is built: it is mostly comprised of pure functions. Pure functions are functions that have their output value influenced by and only by their input. Thus, the readability / maintainability of the BLoC increases, as the component is easy to debug, test and follow. The third advantage of the BLoC lies in its flexibility in injection. Developers can choose which methods to use when introducing the BLoC to the UI layers to best suit the current architecture, let it be through Flutter's inherited widget, third party BLoC providers or simple constructor pass-on. Finally, since BLoC uses Dart language, it can be used across different Dart applications, which means it is not just Flutter specific.

Despite many advantages, there is one considerably important disadvantage that the BLoC has: due to the nature of the pattern, most of the application's business logics are incorporated into the BLoC. Therefore, bigger applications with many business logics will also have many BLoCs if the application uses the BLoC as the application state management system. As the number of BLoCs grow alongside the application size, it will quickly become difficult to keep track of which BLoC is in charge of which UI component. Hence for bigger applications, the BLoC is more suitable for handling the local state,

rather than an application wide state. In addition, performance can be an issue, as the core concept of BLoC revolves around streams and asynchronous operation. With a bigger project, this could lead to a performance heavy application that hinders user experience with loading screens and waiting time. In summary, it is easy to over-engineer the architecture of a large application if using the BLoC. It is not necessarily a pit-fall for developers, but overall is a disadvantage that a developer must be aware of when deciding to use the BLoC pattern.

2.1.2 Redux

2.1.2.1 Theory

At its core, Redux is similar to the BLoC: an event created by the user's interaction will dispatch an action, which mutates the inner state of the component. The UI components will listen to some part of this inner state and will change accordingly. The main difference between Redux and the BLoC is that the BLoC is a more customized version of Redux since BLoC can leverages Flutter's stream and sink feature to increase performance.

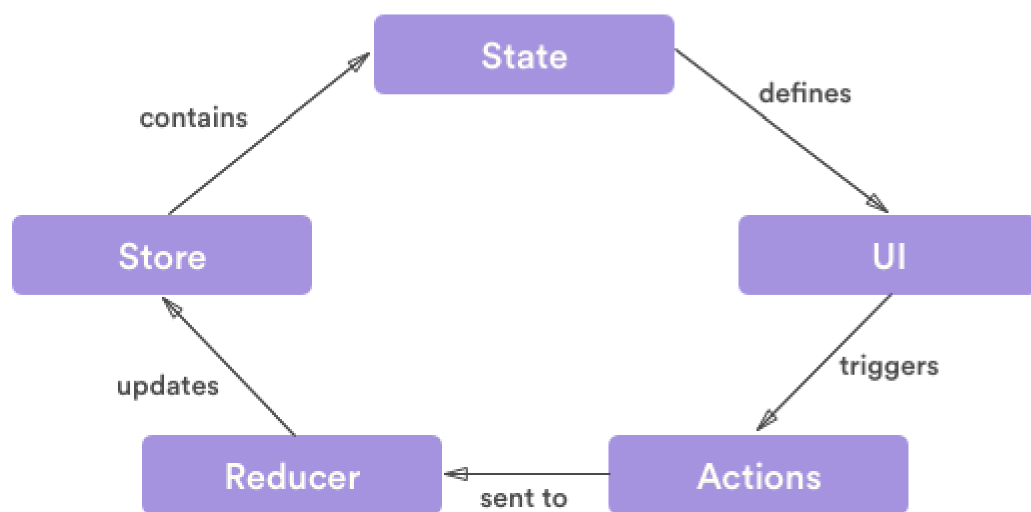

Figure 4. Redux architecture (Tahir, 2018)

Figure 4 illustrates a more detail representation of a Redux pattern. Firstly, the UI component will receive an interaction from a user, which in turn triggers an action to be sent to a reducer where this action is interpreted. In general, the purpose of a reducer is to interpret an action, mutate and return a new inner state according to the action, and finally update the store, which contains all the states of the application. Finally, the UI component that is registered to the specific state will be change accordingly. At the end of this cycle, the UI component will be idle and await the next user interaction to trigger a new action, thus continue the cycle.

2.1.2.2 Example

As with the BLoC example, the same simple Counter App will be used to demonstrate how Flutter uses Redux to manipulate the application's state.

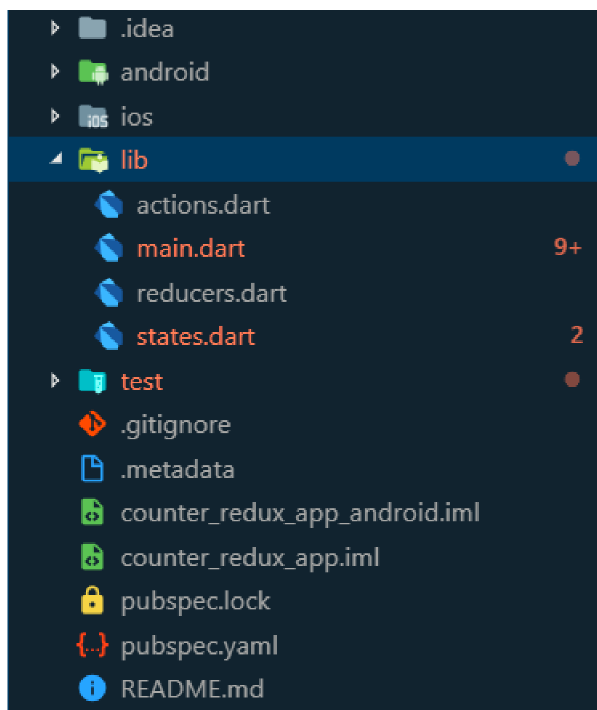

Figure 5. Flutter Redux Architecture

Before analyzing the Redux Architecture, a small explanation of what a reducer is, is needed. In a redux application, a reducer is nothing more than a pure function that receive an old state and an action as its parameters. The reducer will mutate this old state based on the action it receives and return the new state.

As seen in Figure 5, in practice Redux used in Flutter mirrors almost perfectly the Redux theory. Firstly, the UI components that are stored in main.dart will react to the user's interaction, triggering actions that are defined in actions.dart. Afterwards, the action will get dispatched to a reducer inside reducers.dart, where the old application state will get mutated and returned. Finally, after a new state has been returned and stored within states.dart, the UI components inside main.dart that are observing the states will change accordingly.

The example above will be implementing a third-party library called flutter_redux, which will reduce boilerplate code and increase efficiency. However, the new library will also introduce two new concepts: StoreProvider and StoreConnector. Similar to BLoCProvider, StoreProvider is a base widget that will pass Redux's store, or in other words, states to all of the StoreProvider descendant widgets. The descendant widgets will use StoreConnector to request the state from the closest StoreProvider. StoreConnector also handles subscriptions, which in turn, means the widget will automatically get notified and updated by StoreConnector when a new state from StoreProvider is returned. (Egan, 2019)

To further understand how Redux architecture works inside Flutter, it is easiest to follow the dataflow of a user's interaction.

```
floatingActionButton: new StoreConnector<CounterState, OnCounterChanged>(
        converter: (store) {
          return (count) => store.dispatch(IncrementAction(count));
        },
        builder: (context, callback) {
          return new FloatingActionButton(
            onPressed: () => callback(2),
            tooltip: 'Increment',
            child: new Icon(Icons.add),
          );
         }
      )
```

Listing 5. FloatingActionButton dispatching an event to the store

As seen in listing 5, when a user presses the floatingActionButton, a callback will dispatch an IncrementAction to the store. The action will carry the current state of this specific widget to the store, where it is used to mutate the inner state (see listing 6).

Due to the simplicity of this demonstration application, IncrementAction is nothing more than a class with a variable and a constructor.

```
class IncrementAction {
  int count;

  IncrementAction(this.count);
}
```

Listing 6. A simple action class

Afterwards, the store will use reducers to identify the action and mutate the inner state accordingly.

```
CounterState counterReducer(CounterState previousState, dynamic action) {
  if (action is IncrementAction) {
    return CounterState(previousState.count + action.count);
  } else if (action is DecrementAction) {
    return CounterState(previousState.count - action.count);
  } else {
    return previousState;
  }
}
```

Listing 7. The reducer mutating an old state into a new one

In listing 7 above, a reducer is a pure function, checking the action that is passed in and mutating the old state accordingly. The reducer will return a new state if there is any mutation, or the old state, if it cannot find any action that matched.

The CounterState class as seen in listing 7 is the application state, the state that the store will be holding (see listing 8 below):

```
class CounterState {

  static var empty = CounterState(0);

  int count;

  CounterState(this.count);
}
```

Listing 8. The application's state

After a new state is returned, StoreProvider will notify all of it descendant widgets that subscribe to the store via StoreConnector (see listing 9), and the widget will re-render

itself automatically. Afterwards, the widget will become idle and await a new user's interaction.

```
body: new Center(
        child: new Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new Text(
              'You have pushed the button this many times:',
            ),
            new StoreConnector<CounterState, String>(
              converter: (store) => store.state.count.toString(),
              builder: (context, viewModel) {
                return new Text(
                  viewModel,
                  style: Theme.of(context).textTheme.display1,
                );
              },
            ),
          ],
        ),
      )
```

Listing 9. The Text widget subscribing to the store will be re-rendered once a new state is returned

Using the flutter_redux library, most of the boilerplate codes have been hidden from the developer. The library hides the most important component of all, the store component, which in charge of holding the application's state, receiving the reducers, mutating the state, dispatching actions and handling UI subscriptions. Therefore, the developer only needs to define the application state, the reducer which dictates how to mutate the application state, the actions and finally which widget should subscribe to the store, As a result, the library dramatically reduces the codes needed to build the Redux architecture.

2.1.2.3 Discussion

The discussion will briefly explain the main advantage and disadvantage of Redux compared to the BLoC. This is specifically due to the fact that the two state management systems have similar ways of handling states. BloC was built from Redux. Thus, in many ways, Redux is similar to the BLoC. Despite the similarities, Redux does not has the performance advantage that the BLoC has, since Redux is not customized for Flutter like the BLoC, However, since Redux was built as an all-purpose state management system,

it benefits from being flexible and compatible with the Flutter application that is large and complex.

The table below will further showcase the advantages and disadvantages of using Redux in detail.

Table 2. Advantages and disadvantages of using the Redux architecture

Advantages	Disadvantages
• Redux is used by many developers, there is a lot of community support (Coca, 2018) • Data is centralized, only one source of truth (Boelens, 2019)	• Redux architecture produces many files (Boelens, 2019) • There are performance issues due to a lot of code execution (Boelens, 2019) • It is highly complex (Boelens, 2019)

There are two main advantages of using Redux. Firstly, Redux is an old concept. It was initially created in 2015 by Dan Abramov and Andrew Clark (Abramov, 2015). As with any old concept that has withstood the test of time, Redux accumulates a large community of developers. As a result, there are many resources online, tutorials, guides and forums, which make finding a solution much easier. Furthermore, a large and lively community also means Redux will change and evolve from time to time into a better version of itself, as many developers within this community also become contributors to Redux's library. Secondly, there are three principles that Redux has. The first principle is single source of truth, the second is state is read-only and finally the third is changes can only be made with pure functions. These principles mean the data of a Redux architecture is centralized: there is only one version of the data at any given moment. In addition, the data cannot be changed, instead a copy of the old data will be changed. This new copy is what will be returned. Furthermore, only a pure function can change the data... As a result of these three principles, there is only one version of the data at any given time. Therefore, the risk of unexpected behavior will be reduced and the application's scalability increase.

However, there are still disadvantages of using Redux. As everything is encapsulated and separated, a Redux project generates a large number of files and directories, thus making file management a daunting task. Furthermore, the increasing number of files

also means a larger application size, which is especially a problem if it is a native application. Consequently, this would lead to performance issues, as there is more code to be executed, more files to run and more logic to be calculated. Finally, as the project has many files and directories, it grew in complexity, making the maintenance process slow and troublesome.

In the end, most of the disadvantages also stem from the three principles that Redux proposes, as with the advantages. The three principles that lie at the core of every Redux application are double edge swords, rewarding developers with flexibility and scalability if used correctly, or becoming a burden to the development process if misused.

2.1.3 Scoped Model

2.1.3.1 Theory

Before going into detail of what the Scoped Model architecture pattern is, it is beneficial to understand the history behind the pattern. At the Google annual developer conference in 2019, , Google announced a new project they have been working on, a new operating system called Fuchsia. It is an open-source operating system that is considered by Google as “experiments and investments” (Li, 2019). As the OS is open source, Fuchsia exposed its repository to the community of developers, and as a result, the code base was analyzed and understood by the community. However, developers started to notice a pattern appearing repeatedly across the code. It was a state management pattern that Google has been using throughout the development process of Fuchsia. Furthermore, since Fuchsia used the native Flutter widgets throughout its UI development, the newly found state management pattern was also highly compatible with Flutter. Subsequently, some talented members of the community decided to isolate and build a library for Flutter that made use of this pattern. This pattern is called Scoped Model. (Tensor programing, 2018)

Scoped Model was the basis for building the Flutter Redux library (Tensor programing, 2018). Many similarities can be found between the two state management patterns. However, as Scoped Model is the parent pattern that Redux inherits from, the concepts that

Scope Model proposes are more general compared to Redux. Scoped Model consists of three components: Model, ScopedModel and ScopeModelDescendant. Firstly, a Model is relatively similar to a state in Redux: it holds the state variables that the view will be using. However, differ from the BLoC or Redux, the Models also holds the business logics of the application (for the BLoC the business logics were inside of it dedicated BLoC component while for Redux the business logics were inside reducers). Secondly, a ScopedModel function similar to StoreProvider for Redux: it is a widget wrapper. The ScopedModel wraps all widgets that require access to a specific Model instance. Finally, the widget can subscribe to the Model changes by using ScopeModelDescendant, similar to how UI components can use StoreConnector to communicate with the application state in Redux’s store.

In the end, even though the three concepts scoped model proposes almost mirror Redux, the main difference between Redux and Scoped Model is data centralization: it is possible to have many Models and many instances of the same Model which controls different parts of the widget tree. In contrast, with Redux there can only be one store and one source of truth. In other words, the Redux pattern is a specific instance of Scoped Model that uses only one Model to control the whole application. Consequently, scoped model leaves room for developers to be flexible in designing their own architecture. The flexibility will in turn make it possible to reduce the boiler plate code and simplify a smaller project’s architecture.

2.1.3.2 Example

As with other examples in this section, a simple Counter App will be used to explain how the Scoped Model works.

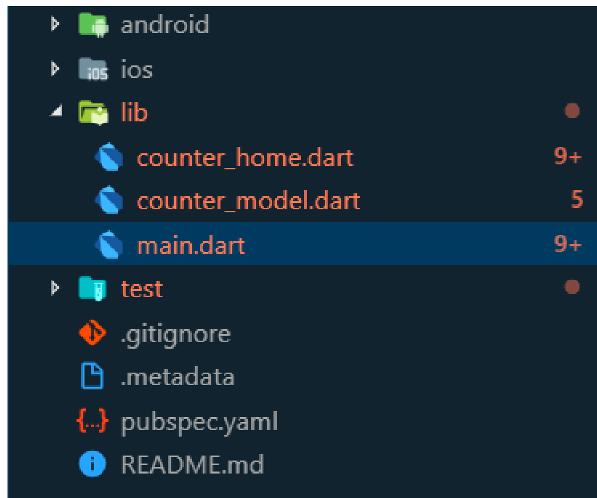

Figure 1. ScopedModel architecture

Figure 6 illustrates one way of using Scoped Model to handle the state of the simple Counter App. The bare minimum architecture required for Scoped Model to work, is simpler than both Redux and the BLoC.

As mentioned in the theory section, applications that used the Scoped Model pattern will be using the scoped_model third party library, which was extracted from the Fuchsia OS repository and developed separately by the community. The library has three main classes corresponding to the three main concepts: Model, ScopedModel and ScopedModelDescendant. The functionality of each class remains similar to what was discussed in the theory section.

Before going into further detail, it is vital to understand how an instance of the data can be passed down the widget tree. The main.dart file is the entry point of the application. It acts as a bridge between the UI elements defined inside counter_home.dart and the data that is stored inside counter_model.dart. Hence, we will find the ScopedModel class inside the main.dart file, Model classes inside the counter_model.dart file and the ScopedModelDescendant class inside counter_home.dart.

```
void main() {
  runApp(MyApp(
    model: CounterModel(),
  ));
}
```

```
class MyApp extends StatelessWidget {
  final CounterModel model;

  const MyApp({Key key, @required this.model}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    // At the top level of our app, we'll, create a ScopedModel Widget. This
    // will provide the CounterModel to all children in the app that request
it
    // using a ScopedModelDescendant.
    return ScopedModel<CounterModel>(
      model: model,
      child: MaterialApp(
        title: 'Scoped Model Demo',
        home: CounterHome('Scoped Model Demo'),
      ),
    );
  }
}
```

Listing 1. CounterModel initialization

As seen in listing 10, the application first creates an instance of CounterModel, which inherits from the scoped_model library's Model class (see listing 11). This instance then gets passed down to the root widget MyApp. As a result, the child widgets of MyApp will have reference to the instance of CounterModel. Afterwards, inside the CounterHome class, the instance of CounterModel is retrieved by using ScopedModelDescendant (see listing 12).

```
class CounterModel extends Model {
  int _counter = 0;

  int get counter => _counter;

  void increment() {
    // First, increment the counter
    _counter++;

    // Then notify all the listeners.
    notifyListeners();
  }
}
```

Listing 2. CounterModel class inherited from the Model class

```
floatingActionButton: ScopedModelDescendant<CounterModel>(
        builder: (context, child, model) {
          return FloatingActionButton(
            onPressed: model.increment,
            tooltip: 'Increment',
            child: Icon(Icons.add),
          );
        },
      )
```

Listing 3. Retrieving the CounterModel instance from the parent widget

By wrapping the parent widget with ScopedModel and wrapping the child widget that needs access to the model with ScopedModelDescendant, the application has successfully passed the instance of the model to other widgets down the widget tree while making sure that there is only one version of the instance exist at any given time.

```
body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('You have pushed the button this many times:'),
            // Create a ScopedModelDescendant. This widget will get the
            // CounterModel from the nearest parent ScopedModel<CounterModel>.
            // It will hand that CounterModel to our builder method, and
            // rebuild any time the CounterModel changes (i.e. after we
            // `notifyListeners` in the Model).
            ScopedModelDescendant<CounterModel>(
              builder: (context, child, model) {
                return Text(
                  model.counter.toString(),
                  style: Theme.of(context).textTheme.display1,
                );
              },
            ),
          ],
        ),
      ),
```

Listing 4. The Text widget wrapped by ScopedModelDescendant

Since the individual widget has access to the model instance, it is not difficult to understand how the data flow inside a Scoped Model application. For example, when the user clicks on the floating action button, a callback is fired. This callback connects directly to the increment method of the model (see listing 12), which in turn changes the inner variable of the model (see listing 11). Afterwards, the model will call a function notifying ScopedModel that there is a data change and that the child widgets need to be refreshed (see listing 10). Finally, the ScopedModel will notify all ScopedModelDescendant classes

of the new change, causing the text widget that is being wrapped by the ScopedModelDescendant to refresh, showing the new data (see listing 13). In the end, the widgets become idle again, awaiting new interaction from the user.

2.1.3.1 Discussion

As Scoped Model was initially developed by Google to be used for Fuchsia OS, there are no major disadvantages to this management system. However, it is not perfect. The table below summarizes the main advantages and disadvantages of using Scoped Model:

Table 1. Advantages and disadvantages of using Scoped Model (Boelens, 2019)

Advantages	**Disadvantages**
• Requires less file management • Is simple to use • Is flexible	• Model class stores both logic and data • There are performance issues due to all components getting refreshed when the model changes • Difficult to find when to notify widgets that that there is a data change

There are a few key advantages Scoped Model has compared to other state management systems. Firstly, as seen in the example section, the architecture of a Scoped Model application is simpler than that of a Redux or BLoC one. Scoped Model simply requires less to set up, less files to manage and, as a result, a cleaner project structure. As a result, a larger project that makes use of Scoped Model will be easier to understand and study for new developers that want to join the team. In addition, Scoped Model is also simpler to use. The state management system does not require understanding of Stream / Sink similar to the BLoC, or Events, Reducers, Store similar to Redux. The concept Scoped Model proposes is easy to understand and quick to apply. Thus, Scoped Model is the most beginner-friendly pattern out of the three patterns mentioned. Finally, as Scoped Model is the basis of Redux, and the BLoC is just a Flutter-customized version of Redux, Scoped Model is the most generic state management system compared to Redux and the BLoC. Hence, it is more flexible, and the developer can freely design an architecture that is suitable for their application based on Scoped Model.

Even though being generic is one advantage of Scoped Model, many disadvantages also stem from Scoped Model being too generic. While Scoped Model is simple and flexible, and it leaves room for more design capability, using the bare Scoped Model itself without any customization will cause problem cause problems. The most problematic part of using Scoped Model is how to handle the Model class. By default, the Model class stores both data and logic; thus, for a small to medium application where the Model class never grows much in complexity, it is perfectly acceptable and sometime preferable. However, as the application becomes big, the Model class becomes more complex. Without any customization from the developers in the form of functionality separation and encapsulation, the Model class will become unwieldy. In addition, as the Model class also handles data change notifications (see listing 11), complex Model classes create a problem with when and where to notify the widget listener. Incorrect data change notifications will result in a widget refresh that is populated with old data or no data at all. Finally, some performance issues also stem from the Model class. As it is the only class that holds data and logic, the Model class become the main data source to many components, not just UI widgets but service and such. Therefore, when a data change notification is being fired, all components that are tied to the Model class get refreshed. This could lead to performance issues, as in reality not all components need a refresh. In conclusion, without proper design and customization, a large application that implements Scoped Model will suffer more performance issues than those which implement the BLoC or Redux.

2.2 Conclusion

So far, the concepts of the BLoC, Redux and Scoped Model have all been explained. Based on the theory section, the three state management systems will be summarized using two key criteria: performance and scalability.

Firstly, the performance of the three state management systems are different. For the BLoC, as expected from a state management that is specifically customized to work with Flutter (Hracek, 2018), the performance is high. One of the most prominent features of the BLoC is that it makes used of Flutter’s Stream, which boost the speed in which events are fired and received. In addition, as BLoC provided flexibility by letting developers decide how to connect the business logic with the UI layer (Hracek, 2018), if configured

correctly, this flexibility could boost performance even more by reducing the amount of code needed to be executed. As for Scoped Model, it is difficult to say anything about the performance. Even though it is developed by Google for Fuchsia and thus some compatibility has already been considered, Scoped Model performance is influenced by the size of the application, and in most parts many customizations are required from the developers for it to perform well. The Model class if used straight from the library without any changes will create performance problems, especially for larger applications, due to the fact that too many functionalities are being carried by the Model class. Furthermore, as mentioned in previous section, since the Model class acts as the main data source for multiple components and services, when data change is notified, not every component requires a refresh. Therefore, it is safe to assume that the more the application grows, the worse the performance of Scoped Model. Finally, for Redux, as expected from a state management that has withstood the test of time, there are not a lot of performance issues. Even though it is not as high performance as BLoC, one can still argue that, since developers are more familiar with the concepts, it is easier and faster to develop a high-performance architecture using Redux than other state management. However, one pitfall developers need to be mindful about is complexity. Redux is not as refined for Flutter as BLoC and therefore could create many more files and folders. This will increase the amount of code needed to be executed as well as create severe file management problems for a larger size application.

Secondly, each state management system seems to work well with a specific application size. Since Scoped Model is the simplest and require the least amount of code to set up, a small to medium size application is favorable. However, when an application becomes big, Redux and BLoC seem to work better. For Redux, the complexity is a trade-off for scalability, as Redux can support complex architecture and, therefore, is suitable for a big application. BLoC, however, it is not as complex as Redux, but it is more specific than Scoped Model. Therefore, it does not excel in any type of application size but rather is useful for all sizes.

3 State Management in Practice

3.1 Overview

The theory section has provided a closer look at the three prominent state management systems, the advantages and disadvantages as well as how suitable each of the systems is for different project types. However, these are assumptions made based on guides and research documents. Therefore, in the practice section, these assumptions will be further analyzed through real-case examples.

Before beginning the practical part, it is worth noting about a change in the thesis content. Initially prior to the thesis, the practical section was planned to be an analysis of a medium size project made by the author. Observations done based on the analyses were to support the theory part. However, after finishing the theory part and having gathered more information, the author has learned considerably more about the topic and concluded that the initial plan was flawed, as it was unachievable: the author would have needed to develop multiple examples to verify each assumption which would have taken too much time. This made the project not feasible. Therefore, after reconsideration, the practical part was changed so that real-case examples were taken from an outside source instead, which would solve both problems.

3.2 Discussion

3.2.1 Introduction

As mentioned, multiple real-case projects taken from an outside source - specifically, GitHub, - were used as data to validate the assumptions made in the theory part, with each assumption being based on one specific characteristic. However, not all characteristics are demonstrable by the current data gathering method. Therefore, some assumptions were not verifiable.

To summarize, nine assumptions were made based on three characteristics which presented in the table below.

Table 2. Assumptions made based on the theory section

Characteristics	BLoC	Scoped Model	Redux
Complexity / Initial setup size	Medium in complexity, requires some setup	Low in complexity, simple and easy to setup	High in complexity, requires many initializations that will hinder performance if architecture not designed properly
Performance	High, made specifically for Flutter	Low, requires many custom optimizations	Medium to high if developers have more experience
Scalability / Flexibility	High, suitable for all application sizes, although prefers medium to large project	Low, not suitable for large size applications, a small to medium project is preferred	High, suitable for medium to large size applications, too complex for a small project

The three characteristics were not chosen by random. They were chosen by determining the final goals of multiple software design principles. The author conducted a small study to gather some of the most popular software design principles and to analyze what the final goal each principle aims to achieve is. For example, the final goal of DRY (Don't Repeat Yourself) is to increase performance as well as make it scalable while SOLID (Single responsibility, Open/closed/, Liskov substitution, Interface segregation, Dependency inversion) is a combination of multiple smaller principles that aim to increase performance, reduce code size and make the application scalable. However, analyzing what these design principles mean and how they affect software is a different topic and out of the scope of this document and therefore will not be addressed. Only the results of the study are reported. In the end, the three characteristics mentioned in table 4 come up repeatedly as the final goals of these design principles. As the design principles were presented as guidelines to develop a good software product, the characteristics should hold some high value toward creating quality software. Therefore, these characteristics were used to determine the quality of the state-management system as well.

In the next section, the author will attempt to demonstrate these assumptions using data from real-case projects. However, since it is too difficult to test the performance of multiple applications in a short amount of time, the performance characteristic was excluded

Metropolia University of Applied Sciences

from testing and the three assumptions correlating to the performance characteristics will be considered as correct.

3.2.2 Data Gathering Process

Before analyzing the data, the gathering process of the data must be discussed first. Since each characteristic required a different approach, the gathering process needs to be explained separately in order to understand the results best.

Firstly, for the complexity characteristic, the sample size was nine projects, taken randomly from the total repositories within GitHub, three for each of the state-management systems. Complexity here means how difficult it is for new developers to understand the code and how quick it is for new developers to utilize the already made state-management architecture to create a new UI. Hence, complexity in this situation dictates how easy to use the state-management is. In a nutshell, a good, easy-to-use library would hide most of its abstractions and concepts, only exposing some simple functions for the developer to interact with it. Therefore, the goal of this data gathering process is to return the percentage that represents the amount of code taken up by the state management development compared to the total amount of code for that project. The lower the percentage is, the better, as it means the library is efficient at hiding concepts difficult to understand, making the code less complex and more understandable. Initially, the property chosen to evaluate this percentage was the number of lines of code. However, this proved to be a mistake as different developers have different coding styles, using slightly different coding syntax. In addition, each project potentially has multiple developers and collaborators. As a result, the data is inconsistent and incorrect as it is impossible to take into account the coding style of each developer. Therefore, another property was chosen: the size of the file in kilobytes. Instead of calculating the number of lines the state management code has taken up, the file size of the whole state management folder was used, bypassing the developer's coding style and returning more consistent data. Finally, using a simple percentage formula, the amount of the project that the state management code has taken up was calculated. These final percentages will be used to conclude whether the assumption made in the theory part on the complexity characteristic is correct or not.

Secondly, for the scalability characteristic, the advance search API of GitHub was used to expand the sample size to all repositories existing in GitHub. The goal for this data gathering process is to return the percentage that represents the number of projects within a project size group (small, medium or large size project group) compared to the total number of projects. In order to retrieve these percentages, the process of gathering data was as follows: the total number of GitHub repositories were divided into many small groups based on the project size, with each group having the maximum size difference of 2,000 KB. The only exception from this rule is the beginning and end group, which will be “less than 1,000 KB” and “more than 19, 000 KB” respectively. Within each of these small groups, the total amount of repositories was registered and used to calculate the percentage of the number of projects this group has taken from the total amount of repositories on GitHub. Finally, these percentages will be used to judge the preciseness of the scalability characteristic

The data gathering process for the scalability characteristic was particularly difficult, encountering many problems along the way. Initially, the process of choosing the sample set was similar to the complexity characteristic, picking nine projects randomly from GitHub with three projects for each state management system. However, further research showed that it is possible to expand the sample size by using the GitHub advance search engine. Even though this method produced much more data, the only property that can be evaluated was the total size of the project. Therefore, it could not be used for the complexity characteristic’s data gathering process. Thus, this data gathering method was not brought over to be used for the complexity characteristic’s data. Another challenge occurring during the gathering process was determining the maximum difference in size for each small group. Initially, the amount was set to 5,000 KB, which proved to be incorrect. The data was not divided finely enough, and there was a big leap between data points, which made it impossible to draw any conclusions. The second attempt was to set the amount to 100 KB, which is incorrect also: the data was too finely divided, producing many data points. In this case, the difference between data points was too little which meant that the plotted graph was almost a horizontal line. The third attempt to set the amount to 2,000 KB was successful, and it properly showed the changes in the number of projects based on the project size, which is what the goal required. In the end, even though many failed attempts resulted in a delay with the data gathering process, the data points gathered are valid, and they enable drawing conclusions.

3.2.3 Results

After the data gathering process, the raw data can be summarized and roughly processed to make sense of the data. The two tables below summarize what was gathered.

For the complexity characteristic, the raw data is summarized in the third and fourth column in the table below.

Table 3. Results of data gathering on the complexity characteristic

Project type	Project link	Total code size (in KB)	Total code size for state management (in KB)	Complexity percentage
Scoped model	My Movies	15.7	5.1	32.48%
Scoped model	Flutter Products tutorial	23.8	4.84	20.33%
Scoped model	Flutter Flip	29.1	7	24.05%
Redux	inKino	79.7	28.9	36.26%
Redux	Flutter Shopping-cart	9.53	3.51	36.83%
Redux	Flutter Mobile	2920	655	22.43%
BLoC	Flutter Shopping-cart	5.32	1	18.79%
BLoC	Deer	232	44.98	19.38%
BLoC	Chillify	88.2	9.05	10.2%

Based on columns 3 and 4 in table 5, it is possible to calculate the complexity percentage of code dedicated to the development of state-management, which can be seen in column 5, by using the following formula:

$$complexity\ percentage = \frac{total\ code\ size\ for\ state\ managment}{total\ code\ size} \times 100$$

The complexity percentages are independent of the individual project and were used to analyze the results presented in the summary and discussion section.

For the scalability characteristic, the raw data is presented in table 6.

Table 4. Result of data gathering on the scalability characteristic

Size / Type	Scoped Model (in number of projects)	Redux (in number of projects)	BLoC (in number of projects)
<1000kb	2643	896	7487
1000kb-3000kb	5203	2916	13999
3000kb-5000kb	2653	1371	4083
5000kb-7000kb	1568	702	3071
7000kb-9000kb	898	386	1742
9000kb-11000kb	505	252	695
11000kb-13000kb	310	146	634
13000kb-15000kb	200	92	349
15000kb-17000kb	119	58	187
17000kb-19000kb	91	50	38
>19000kb	316	116	393

To be able to compare the data of each column with each other, the data must be independent of each row and, therefore, must be converted into percentage. Thus, each data point can be further refined into percentage units by using the formula given below.

$$projects\ percentage\ for\ a\ group\ size = \frac{number\ of\ projects\ in\ state\ management\ group\ in\ the\ group\ size}{total\ number\ of\ project\ in\ the\ group\ size} \times 100$$

For example, to find the percentage of scoped model projects in group size <1000 KB that had the total number of projects 2643 + 897 +7487 = 11026, we have:

$$\frac{2643}{11026} \times 100 = 23.97\%$$

This process is repeated for all cells in table 6, and the results of the computation process are shown in table 7 below.

Table 5. Results of data gathering regarding the scalability characteristic (refined)

Size / Type	Scoped Model (in percentage)	Redux (in percentage)	BLoC (in percentage)
<1000kb	23.97%	8.12%	67.91%
1000kb-3000kb	23.52%	13.18%	63.3%
3000kb-5000kb	32.72%	16.91%	50.37%
5000kb-7000kb	29.35%	13.14%	57.51%
7000kb-9000kb	29.67%	12.75%	57.58%
9000kb-11000kb	34.77%	17.35%	47.88%

11000kb-13000kb	28.44%	13.39%	58.17%
13000kb-15000kb	31.2%	14.35%	54.45%
15000kb-17000kb	32.69%	15.93%	51.38%
17000kb-19000kb	50.8%	27.93%	21.27%
>19000kb	38.3%	14.06%	47.64%

The percentages in table 7 are independent of each row and were used to analyze the result in discussion section.

4 Result Summary

For the results of the complexity characteristic which was demonstrated in table 5, it is possible to refine the data even more by calculating the average complexity percentage of each state management system using the formula given below.

$$average\ complexity\ percentage = \frac{sum\ of\ complexity\ percentages\ of\ one\ state\ management}{number\ of\ projects\ of\ one\ state\ management}$$

The results of the calculation are illustrated in the table below.

Table 6. Average complexity percentage of each state management system

Scoped Model	Redux	BLoC
25.62%	31.84%	16.12%

Table 8 represents the data in its most concentrated form and will be used for the final discussion of the practical section as well as the conclusion of the thesis.

For the results of the scalability characteristic, it was initially planned that the data will be divided into three groups of projects depending on the size (small, medium and large size projects) of each group. The average percentage will be calculated and used to analyze the results. However, upon further research it is clear that the project size categorization is a wide topic on its own and that there are multiple variables such as the number of technologies, the number of developers and developers' various experiences (Borysowich, 2010). In other words, it is not just the physical byte size of the project that is important when calculating a project's size. Therefore, to simplify the summary process, the data was instead plotted into a graph and analyzed base on the graph's trend. The figures are illustrated in figure 7 below,

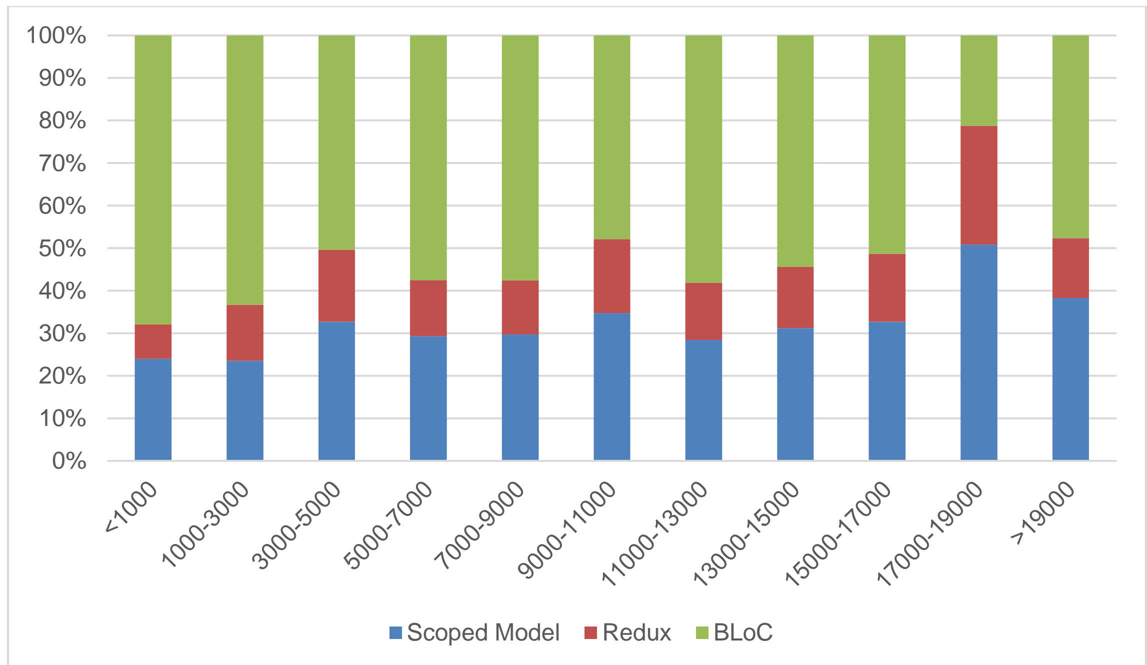

Figure 2. Difference in projects' state management usage per size group

Figure 7 demonstrates the popularity of each state management when the size of the project increases. This data will be used in the discussion section to analyze the validity of the assumptions made about the scalability characteristic in the theory part.

5 Discussion

Before analyzing the data, one should quickly be reminded of what assumptions need to be proven. To evaluate the viability of the three state-management systems, this document uses three characteristics: complexity / initial setup size, performance and scalability. However, the performance characteristic is difficult to be verified. Therefore, the performance assumption is considered to be correct. The performance assumption suggests that BLoC will have the most performance as it is a special version of Redux that takes advantages of Dart language's properties such as streams and asynchronous functions. Redux, as a generalized version of BLoC comes in second while Scoped Model, due to its extreme simplicity, comes in last. For the other two assumptions, complexity and scalability, these assumptions can be somewhat verified by gathering data of projects from GitHub, and therefore, it is the focus of this discussion section. The complexity and the scalability assumptions suggest that Redux is the most complicated to setup and maintain, even though the trade-off would be it being easily scalable. On the other hand, BLoC is a simpler version of Redux while still maintain the high scalability Finally, Scoped Model is the simplest state management system out of the three, but it is not as scalable.

For the complexity assumption, table 8 from the results summary section clearly demonstrates that it is true Redux is the most complex out of the three, coming in at approximately 32%. However, interestingly the second place is taken by Scoped Model, and not BLoC, with the complexity percentages being approximately 26% and 16% respectively. This could be due to the Scoped Model in its purest form is too simple to be used on any project that is not a prototype project. Therefore, developers must write additional code to back up Scoped Model, which in turn increases the complexity percentage. Lastly, BLoC seems to be the least complex, or at least it requires the smallest amount of setup for the project to run. Again, this could be due to BLoC being made specifically for Flutter. This means the code is cleaner, it requires less setup and it hides most of the complex logic behind libraries, exposing only simple functions for developers to interact with. In a nutshell, after evaluating the data, the complexity / initial setup assumption could be modified. Redux is the most complicated state management system while BLoC is the simplest and easiest to setup the state management system for Flutter.

Lastly, for the scalability assumption, figure 7 illustrates the popularity trend of the three-state management systems when projects grow. However, the data from two ends of the spectrum at <1,000 KB and >19,000 KB should be neglected due to these data being inconsistent. These data does not have the 2,000 KB size difference similar to the other data that the chart demonstrate. In addition, the data point at 17,000 KB-19,000 KB should be bypassed as well, since the data point creates too great of a difference between the rest of the data points and is therefore most likely an outlier in the graph. In the end, figure 7 can be modified to be figure 8, as shown below.

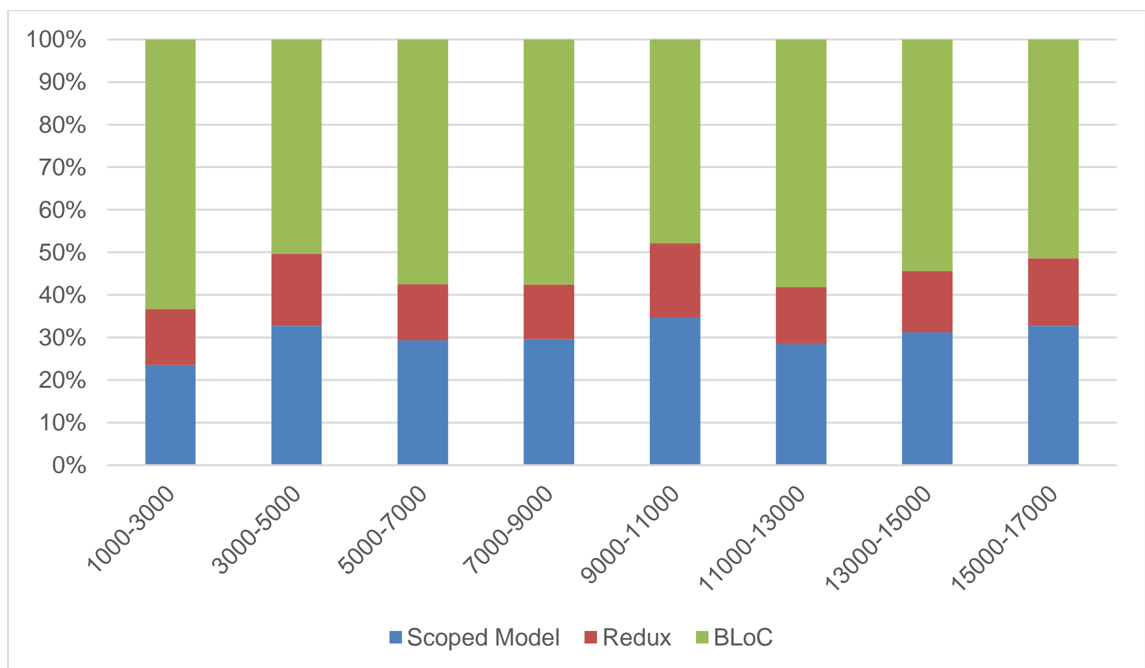

Figure 3. Difference in projects' state management usage per size group after removing outliers

The results summarized in figure 8 could be used to validate the scalability assumption by analyzing the popularity of the state management systems throughout the project size group. For the BLoC, it seems the assumption is correct: the BLoC maintains around 50% popularity throughout all groups, from small to large, meaning the BLoC is highly scalable and is the preferred choice of developers regardless of project size. However, when it comes to Redux, the data show the opposite of the assumption: Redux is the least popular state-management system throughout all group sizes. The reason behind this could be that Scoped Model is not as bad with scalability and that the simplicity Scoped Model brought to the project outweighs it disadvantages. The simple architecture

Scoped Model provided will result in easier collaboration and faster development speed for projects, which are two crucial aspects for open source projects, since these projects often have a long list of collaborators and must constantly evolve to suit the needs of the ever-growing community. Consequently, when choosing between Redux and Scoped Model, the most logical choice for GitHub developers, while considering which state management their open source project would take, is Scoped Model. Therefore, on paper Redux might be more scalable than Scoped Model code-wise; in reality when collaboration and speed is emphasized, Scoped Model wins over. However, as a result, due to the data source heavily influencing the data gathered, the results presented in the previous section cannot be used to validate the assumption anymore due to bias created by lack of generalization. Despite the data being invalidated for a general use case, the conclusion could still be drawn in the scope of open source projects. Therefore, the scalability assumption can be modified as follows: for open source projects, the BLoC has the highest scalability while Redux has the lowest scalability. This is due to the nature of open source projects: open source projects required a lot of collaborations and speed, both of which Scoped Model, with its simplicity, excel at.

In a nutshell, after analyzing the data and using it to validate the assumptions, the conclusion can be drawn: For open source projects, out of all state management systems, BLoC is the simplest and easiest to setup, with a high performance and the highest scalability to match. On the other hand, Redux is the most difficult and complex to setup, with marginally high performance and the lowest scalability. Lastly, Scoped Model requires medium setup with medium performance and medium scalability.

6 Conclusion

Initially, the goal of the thesis was to answer one question: between the three most common state management systems, BLoC, Redux and Scoped Model, what is the most suitable state-management system for Flutter? In the end, the results of the research managed to answer just that. While Redux is a high-performance model, it is too complex and therefore not ideal in modern software development where collaboration is emphasized. As for Scoped Model, the state-management system is easy to understand, and even though not too high in performance and scalability, Scoped Model's trade-offs are still mostly acceptable. Finally, BLoC, which is made for Dart, is a high-performance and scalable state-management system built with simplicity in mind; hence, it inherits all the advantages Scoped Model has with none of the disadvantages. Therefore, in conclusion, BLoC is the most suitable state-management system for Flutter at the time of writing this document.

As Flutter is relatively new in the software development community, not many best practices have been set yet. Thus, the conclusion above could potentially play an important role as it represents the first step in creating a new best practice. In the end, after multiple further studies have been conducted, the conclusion drawn based on the thesis could be developed even more until it becomes a mature concept and could be added as a new best practice for the development of Flutter. The productivity boost once the conclusion has been developed into a best practice could be tremendous, as knowing beforehand what the most ideal state-management system is could help an application architect design an application structure to be future-proof, which would save valuable development time. However, for the time being, as the conclusion drawn from the thesis is not yet a mature concept, developers should take the statement "BLoC is the most suitable state-management system for Flutter" with precautions.

In conclusion, it is recommended for future studies based on this document to further develop the conclusion into a mature concept. However, for future work, there are multiple aspects of the research that should be improved to ensure higher accuracy. Firstly, the sample size should be increased to include more projects: since time is restricted, current research conducted for this thesis could not include as big sample size as needed and therefore the final data could potentially be biased and contain more human error

than it should. Secondly, the project types should be expanded further beyond the scope of GitHub to include also open-source projects from other platforms as well as closed-projects made by senior developers. This will ensure the conclusion drawn will be universal and not tied to specific scope. Lastly, future research should be conducted throughout a longer period. As with any young development tools, Flutter's development trend could fluctuate more than the development trend of an older, more mature tools; hence, a longer data gathering process could average out this fluctuation.

References

Abramov, D., 2015. *Three Devs and a Maybe* [Interview] (6 November 2015).

Angelov, F., 2019. *BLoC package.* [Online]
Available at: https://pub.dev/packages/bloc
[Accessed 25th April 2019].

Angelov, F., 2019. *Dart Packages.* [Online]
Available at: https://pub.dartlang.org/packages/flutter_bloc
[Accessed 19 February 2019].

Angelov, F., 2019. *Flutter.* [Online]
Available at: https://flutter.io/docs/development/data-and-backend/state-mgmt/options
[Accessed 18 2 2019].

Boelens, D., 2019. *Didier Boelens.* [Online]
Available at: https://www.didierboelens.com/2019/04/bloc---scopedmodel---redux---comparison/
[Accessed 21 June 2019].

Borysowich, C., 2010. *Toolbox.* [Online]
Available at: https://it.toolbox.com/blogs/craigborysowich/project-size-and-complexity-calculation-form-template-060210
[Accessed 25 October 2019].

Coca, J., 2018. *Let me help you to understand and choose a state management solution for your app.* [Online]
Available at: https://medium.com/flutter-community/let-me-help-you-to-understand-and-choose-a-state-management-solution-for-your-app-9ffeac834ee3
[Accessed 26 February 2019].

Egan, B., 2019. *Dart Packages.* [Online]
Available at: https://pub.dev/packages/flutter_redux
[Accessed 17 June 2019].

Google Developers, 2018. *Build reactive mobile apps with Flutter (Google I/O '18).* [Online]
Available at: https://youtu.be/RS36gBEp8OI
[Accessed 26 February 2019].

Google Developers, 2018. *Technical Debt and Streams/BLoC (The Boring Flutter Development Show, Ep. 4).* [Online]
Available at: https://youtu.be/fahC3ky_zW0
[Accessed 26 February 2019].

Google Development Team, 2018. *Google Developers.* [Online]
Available at: https://developers.googleblog.com/2018/12/flutter-10-googles-portable-ui-toolkit.html
[Accessed 18 2 2019].

Hracek, F., 2018. *Build reactive mobile apps in Flutter—companion article.* [Online]
Available at: https://medium.com/flutter-io/build-reactive-mobile-apps-in-flutter-companion-article-13950959e381
[Accessed 26 February 2019].

Hracek, F., 2019. *Start thinking declaratively.* [Online]
Available at: https://flutter.dev/docs/development/data-and-backend/state-mgmt/declarative
[Accessed 26 February 2019].

Li, A., 2019. *9TO5Google.* [Online]
Available at: https://9to5google.com/2019/05/09/what-is-google-fuchsia/
[Accessed 16 June 2019].

Opia, C. E.-G., 2018. *Medium Free Code Camp.* [Online]
Available at: https://medium.freecodecamp.org/how-to-handle-state-in-flutter-using-the-bloc-pattern-8ed2f1e49a13
[Accessed 19 February 2019].

Reso Coder, 2019. *Flutter BLoC library tutorial.* [Online]
Available at: https://github.com/ResoCoder/flutter-bloc-library-tutorial
[Accessed 19 February 2019].

Tahir, N., 2018. *hackermoon.* [Online]
Available at: https://hackernoon.com/lessons-learned-implementing-redux-on-android-cba1bed40c41
[Accessed 16 June 2019].

Tensor programing, 2018. *steemit.* [Online]
Available at: https://steemit.com/utopian-io/@tensor/managing-state-with-the-scoped-model-pattern-in-dart-s-flutter-framework
[Accessed 16 June 2019].

Appendix: Counter App code

Counter App code for BLoC

```
abstract class CounterEvent {}

class IncrementEvent extends CounterEvent {}

class DecrementEvent extends CounterEvent {}
```

counter_event.dart

```
class CounterState {
  final int counter;

  const CounterState({this.counter});

  factory CounterState.initial() => CounterState(coun-
ter: 0);
}
```

counter_state.dart

```
import 'package:bloc_library_tut/counter_bloc.dart';
import 'package:bloc_library_tut/counter_state.dart';
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
```

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home
Page'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;

  @override
  _MyHomePageState createState() => _MyHomeP-
ageState();
}

class _MyHomePageState extends State<MyHomePage> {
  final _counterBloc = CounterBloc();

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      bloc: _counterBloc,
```

```
      child: CounterWidget(widget: widget),
    );
  }

  @override
  void dispose() {
    _counterBloc.dispose();
    super.dispose();
  }
}

class CounterWidget extends StatelessWidget {
  const CounterWidget({
    Key,
    @required this.widget,
  }) : super(key: key);

  final MyHomePage widget;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: BlocBuilder(
          bloc: BlocProvider.of<CounterBloc>(context),
          builder: (context, CounterState state) {
            return Center(
              child: Column(
                mainAxisAlignment: MainAxisAlign-
ment.center,
                children: <Widget>[
                  Text(
```

```
                    'You have pushed the button this
many times:',
                  ),
                  Text(
                    '${state.counter}',
                    style: Theme.of(con-
text).textTheme.display1,
                  ),
                ],
              ),
            );
          }),
      floatingActionButton: Row(
        mainAxisAlignment: MainAxisAlignment.end,
        children: <Widget>[
          FloatingActionButton(
            onPressed: () =>
                BlocProvider.of<CounterBloc>(con-
text).onIncrement(),
            tooltip: 'Increment',
            child: Icon(Icons.add),
          ),
          SizedBox(width: 10),
          FloatingActionButton(
            onPressed: () =>
                BlocProvider.of<CounterBloc>(con-
text).onDecrement(),
            tooltip: 'Decrement',
            child: Icon(Icons.remove),
          ),
        ],
      ),
    );
  }
```

```
}
```

main.dart

```
import 'package:bloc/bloc.dart';
import 'package:bloc_library_tut/counter_event.dart';
import 'package:bloc_library_tut/counter_state.dart';

class CounterBloc extends Bloc<CounterEvent, Counter-
State> {
  void onIncrement() {
    dispatch(IncrementEvent());
  }

  void onDecrement() {
    dispatch(DecrementEvent());
  }

  @override
  CounterState get initialState => CounterState.ini-
tial();

  @override
  Stream<CounterState> mapEventToState(
    CounterState currentState,
    CounterEvent event,
  ) async* {
    if (event is IncrementEvent) {
      yield CounterState(counter: currentState.counter
+ 1);
    } else if (event is DecrementEvent) {
```

```
      yield CounterState(counter: currentState.counter
- 1);
    }
  }
}
```

counter_bloc.dart

Counter App code for Redux

```
import 'package:flutter/material.dart';
import 'package:redux/redux.dart';
import 'package:flutter_redux/flutter_redux.dart';
import 'reducers.dart';
import 'actions.dart';
import 'states.dart';

void main() {
  final Store<CounterState> store = new Store<Counter-
State>(counterReducer, initialState: Counter-
State.empty);
  runApp(new MyApp(store));
}

class MyApp extends StatelessWidget {
  final Store<CounterState> store;

  MyApp(this.store);

  @override
  Widget build(BuildContext context) {
    return new StoreProvider<CounterState>(
      store: store,
      child: new MaterialApp(
        title: 'Flutter Demo',
        theme: new ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: new MyHomePage(title: 'Flutter Demo Home
Page'),
      ),
    );
```

```
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;

  @override
  _MyHomePageState createState() => new _MyHomeP-
ageState();
}

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new Center(
        child: new Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new Text(
              'You have pushed the button this many
times:',
            ),
            new StoreConnector<CounterState, String>(
              converter: (store) =>
store.state.count.toString(),
              builder: (context, viewModel) {
                return new Text(
                  viewModel,
```

```
                  style: Theme.of(con-
text).textTheme.display1,
                ),
              },
            ),
          ],
        ),
      ),
      floatingActionButton: new StoreConnector<Coun-
terState, OnCounterChanged>( // () -> Unit
        converter: (store) {
          return (count) => store.dispatch(Incremen-
tAction(count));
        },
        builder: (context, callback) {
          return new FloatingActionButton(
            onPressed: () => callback(2),
            tooltip: 'Increment',
            child: new Icon(Icons.add),
          );
        },
      ),
    );
  }
}

typedef OnCounterChanged = Function(int count);
```

main.dart

```
enum Action {
  Increment, Decrement
```

```
}

class IncrementAction { // sealed
  int count;

  IncrementAction(this.count);
}

class DecrementAction {
  int count;

  DecrementAction(this.count);
}
```

actions.dart

```
import 'actions.dart';
import 'states.dart';

CounterState counterReducer(CounterState previ-
ousState, dynamic action) {
  if (action is IncrementAction) {
    return CounterState(previousState.count + ac-
tion.count);
  } else if (action is DecrementAction) {
    return CounterState(previousState.count - ac-
tion.count);
  } else {
    return previousState;
  }
```

```
}
```

reducers.dart

```
import 'package:meta/meta.dart';

@immutable
class CounterState {

  static var empty = CounterState(0);

  int count;

  CounterState(this.count);
}
```

states.dart

Counter App code for Scoped Model

```
import 'package:flutter/material.dart';
import 'package:scoped_model/scoped_model.dart';
import 'counter_model.dart';
import 'counter_home.dart';

void main() {
  runApp(MyApp(
    model: CounterModel(),
  ));
}

class MyApp extends StatelessWidget {
  final CounterModel model;

  const MyApp({Key key, @required this.model}) : su-
per(key: key);

  @override
  Widget build(BuildContext context) {
    // At the top level of our app, we'll, create a
ScopedModel Widget. This
    // will provide the CounterModel to all children
in the app that request it
    // using a ScopedModelDescendant.
    return ScopedModel<CounterModel>(
      model: model,
      child: MaterialApp(
        title: 'Scoped Model Demo',
        home: CounterHome('Scoped Model Demo'),
      ),
    );
  }
```

```
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:scoped_model/scoped_model.dart';

class CounterHome extends StatelessWidget {
  final String title;

  CounterHome(this.title);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('You have pushed the button this many
times:'),
            // Create a ScopedModelDescendant. This
widget will get the
            // CounterModel from the nearest parent
ScopedModel<CounterModel>.
            // It will hand that CounterModel to our
builder method, and
            // rebuild any time the CounterModel
changes (i.e. after we
```

```
            // `notifyListeners` in the Model).
            ScopedModelDescendant<CounterModel>(
              builder: (context, child, model) {
                return Text(
                  model.counter.toString(),
                  style: Theme.of(con-
text).textTheme.display1,
                );
              },
            ),
          ],
        ),
      ),
      // Use the ScopedModelDescendant again in order
to use the increment
      // method from the CounterModel
      floatingActionButton: ScopedModelDescend-
ant<CounterModel>(
        builder: (context, child, model) {
          return FloatingActionButton(
            onPressed: model.increment,
            tooltip: 'Increment',
            child: Icon(Icons.add),
          );
        },
      ),
    );
  }
}
```

counter_home.dart

```
import 'package:flutter/material.dart';
import 'package:scoped_model/scoped_model.dart';

// Start by creating a class that has a counter and a
method to increment it.
//
// Note: It must extend from Model.
class CounterModel extends Model {
  int _counter = 0;

  int get counter => _counter;

  void increment() {
    // First, increment the counter
    _counter++;

    // Then notify all the listeners.
    notifyListeners();
  }
}
```

counter_model.dart