Chapter 9:
State management

Plan

- **Ephemeral states & app states.**
- **Why State Management is important ?**
- **State Management approaches:**
  - ✓ **Provider.**
  - ✓ **BLoC.**

Plan

- **Ephemeral states & app states.**
- **Why State Management is important ?**
- **State Management approaches:**
  - ✓ **Provider.**
  - ✓ **BLoC.**

Ephemeral state & App state

- **Ephemeral state :** is the state contained in a single widget. It is called, **UI state** or **local state**.

- **Examples :**
  - ✓ state of a checkbox,
  - ✓ state of a Textfield,
  - ✓ …

➢ If no other part of the app needs to access ➔ use a Statefull widget and its setState () function to handle such local states.

Ephemeral state & App state

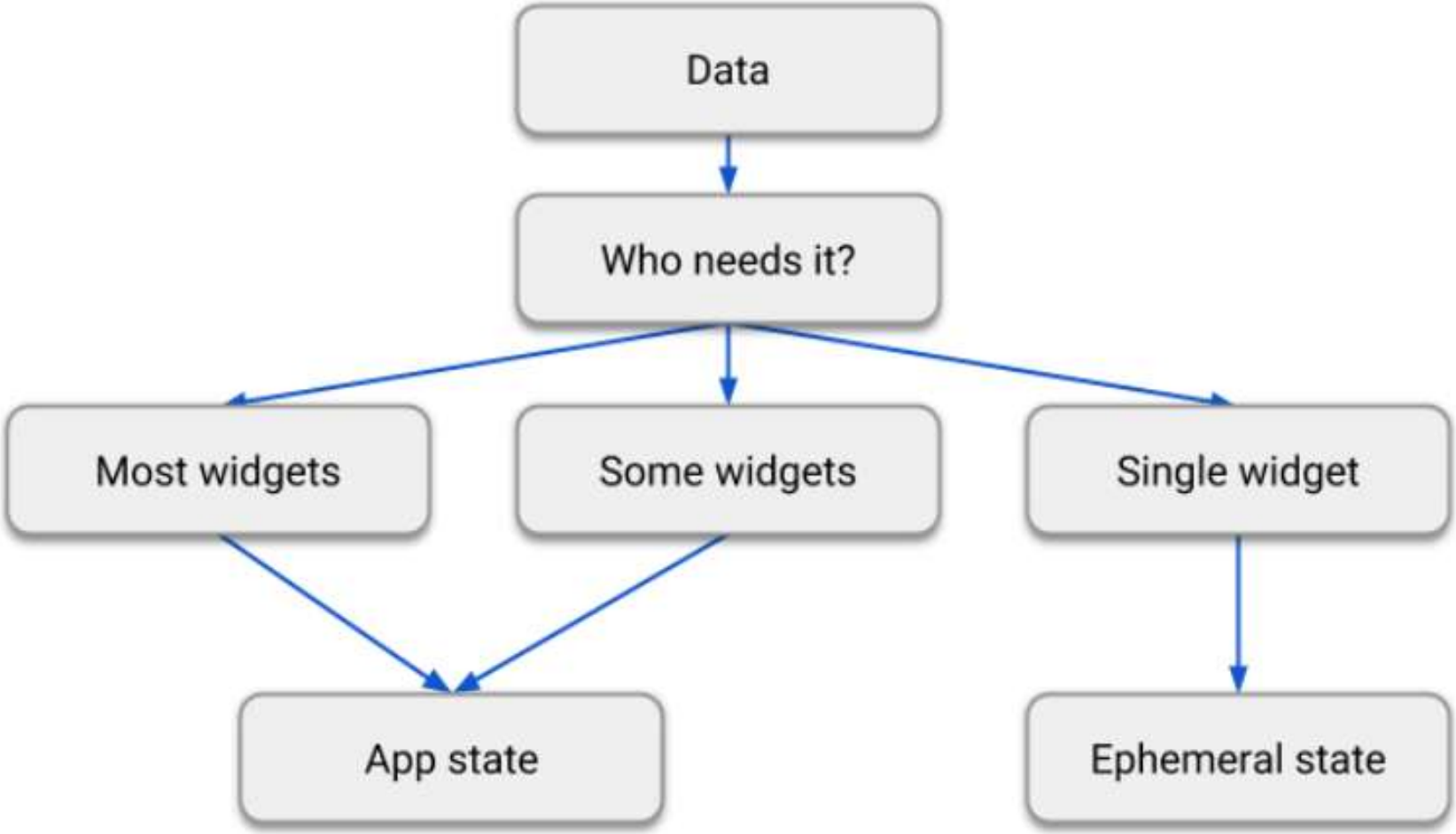
- **App state :** is the state used across many parts of the app. It is called **shared state**.

- **Examples :**

  ✓ Login info,
  ✓ Notifications in a social networking app,
  ✓ The shopping cart in an e-commerce app, …

➢ To handle such shared states ➔ use ***State Management*** approaches.

Ephemeral state & App state



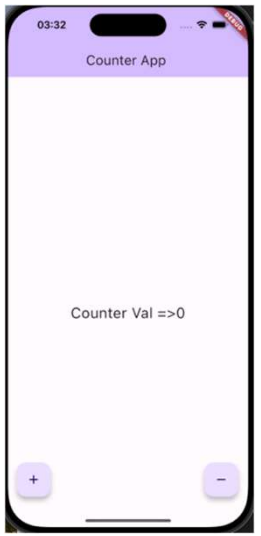
Plan

- Ephemeral states & app states.
- **Why State Management is important ?**
- State Management approaches:
  - ✓ Provider.
  - ✓ BLoC.

Why State Management ?

Illustrative example:

Basic Counter App



MyApp

MaterialApp

CounterPage

Counter = 0

CounterState

Build(context)

SetState()

Scaffold

appBar

body

floatingButton

AppBar

Center

Row

SetState(() {
++Counter; })

title

child

children

Column

FloatingButton

FloatingButton

Text

children

child

child

Text(CounterVal=>$Counter)

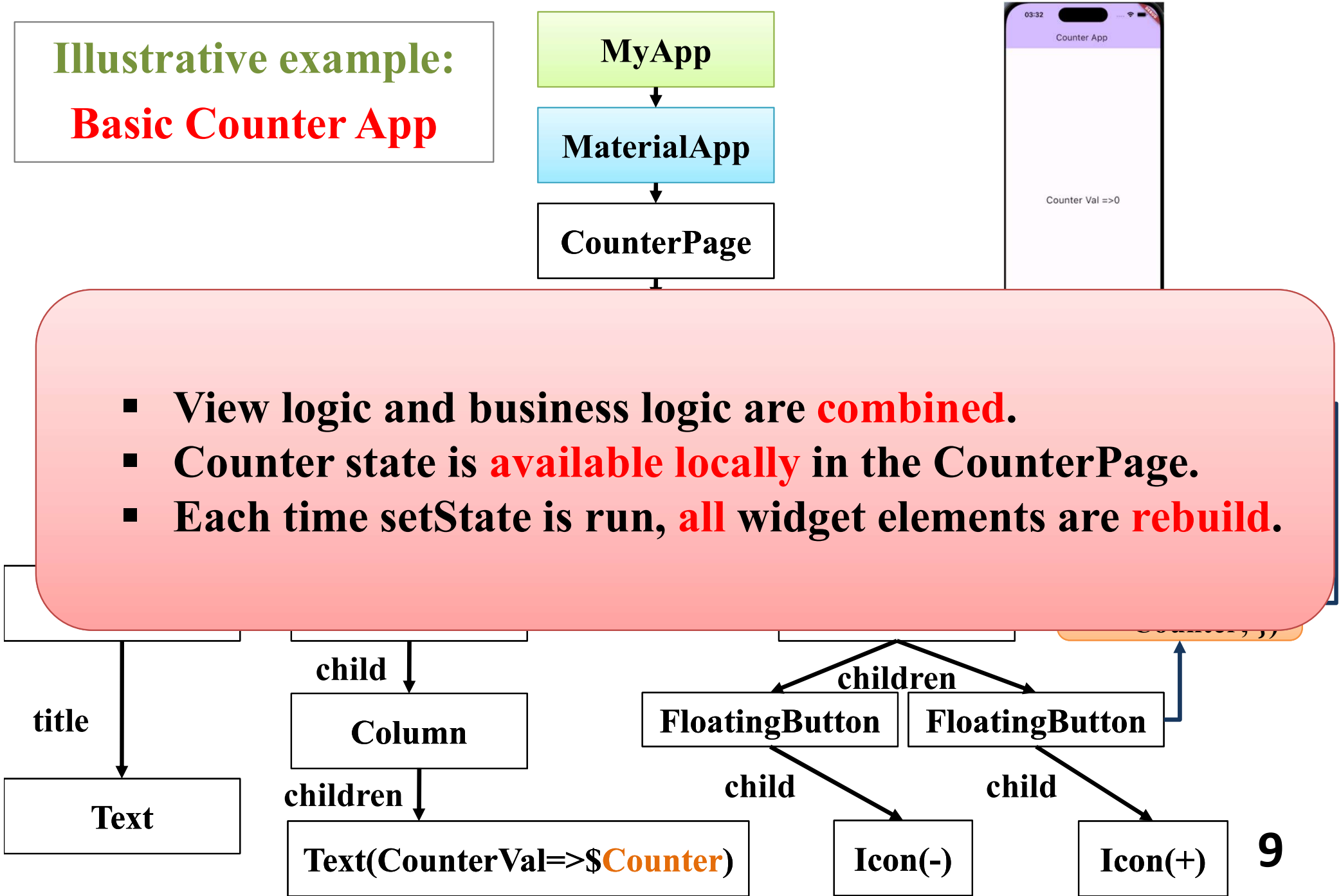
Icon(-)

Icon(+)

Why State Management ?

Illustrative example:

Basic Counter App



- View logic and business logic are **combined**.
- Counter state is **available locally** in the CounterPage.
- Each time setState is run, **all** widget elements are **rebuild**.

Why State Management ?

- It is used to respond to ephemeral state (Statefull) **issues**. It allows:

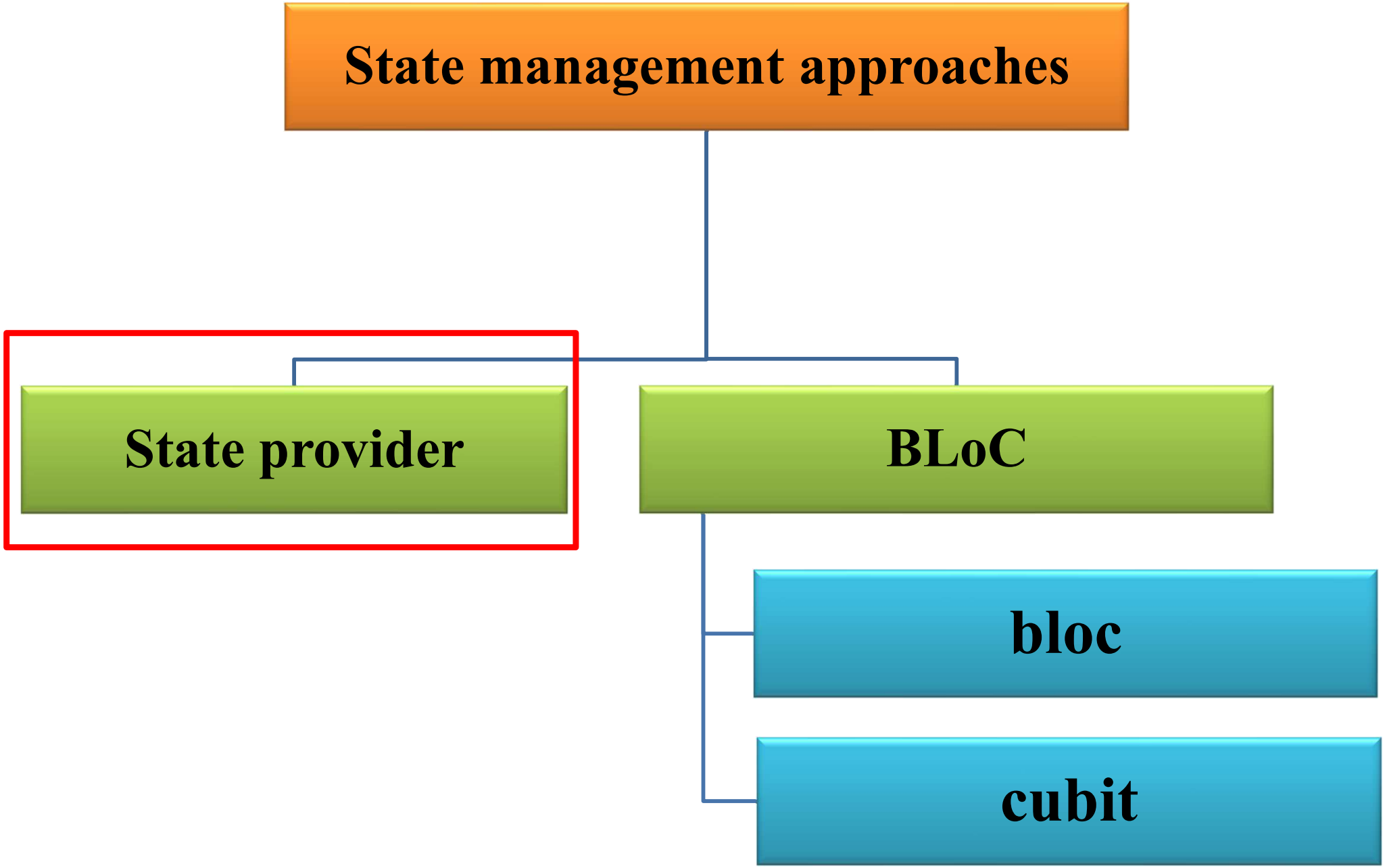
1) Separation between the **UI** and the **business logic**.

2) Maximum possible improvement of the **performance** (rebuild **only** widgets that need to be update => **fast** & **optimized**).

3) **Centralization** of the state to be access from different app parts.

Plan

- Ephemeral states & app states.
- Why State Management is important ?
- **State Management approaches:**
  - ✓ **Provider.**
  - ✓ **BLoC.**

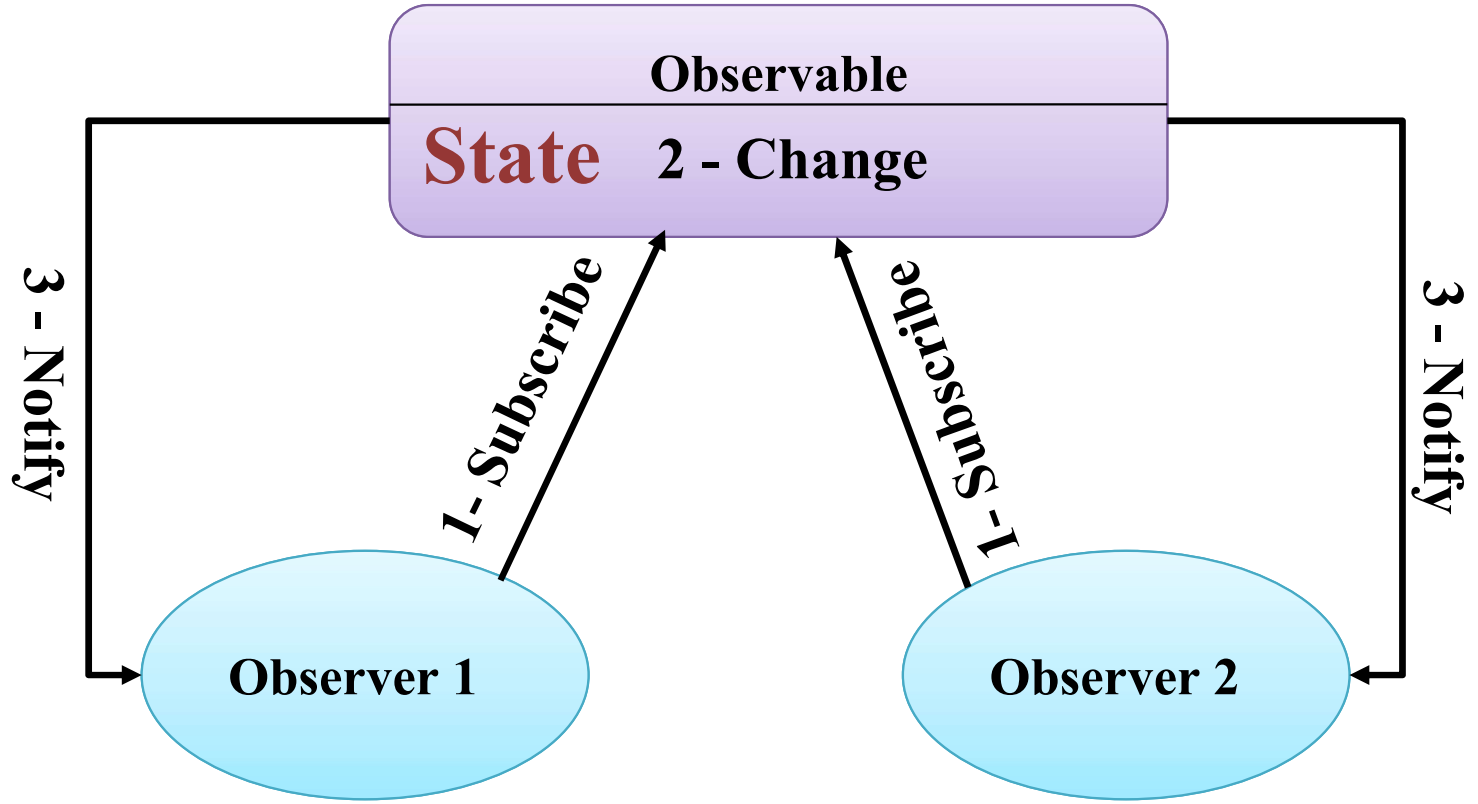

State management approaches
State provider
BLoC
bloc
cubit

Plan

- Ephemeral states & app states.
- Why State Management is important ?
- **State Management approaches:**
  - ✓ **Provider.**
  - ✓ **BLoC.**

State Management approaches: Provider

- **Provider:** is used to centralize a State within an App. It is based on the **Observer** pattern.



State Management approaches: Provider

- **In provider:**

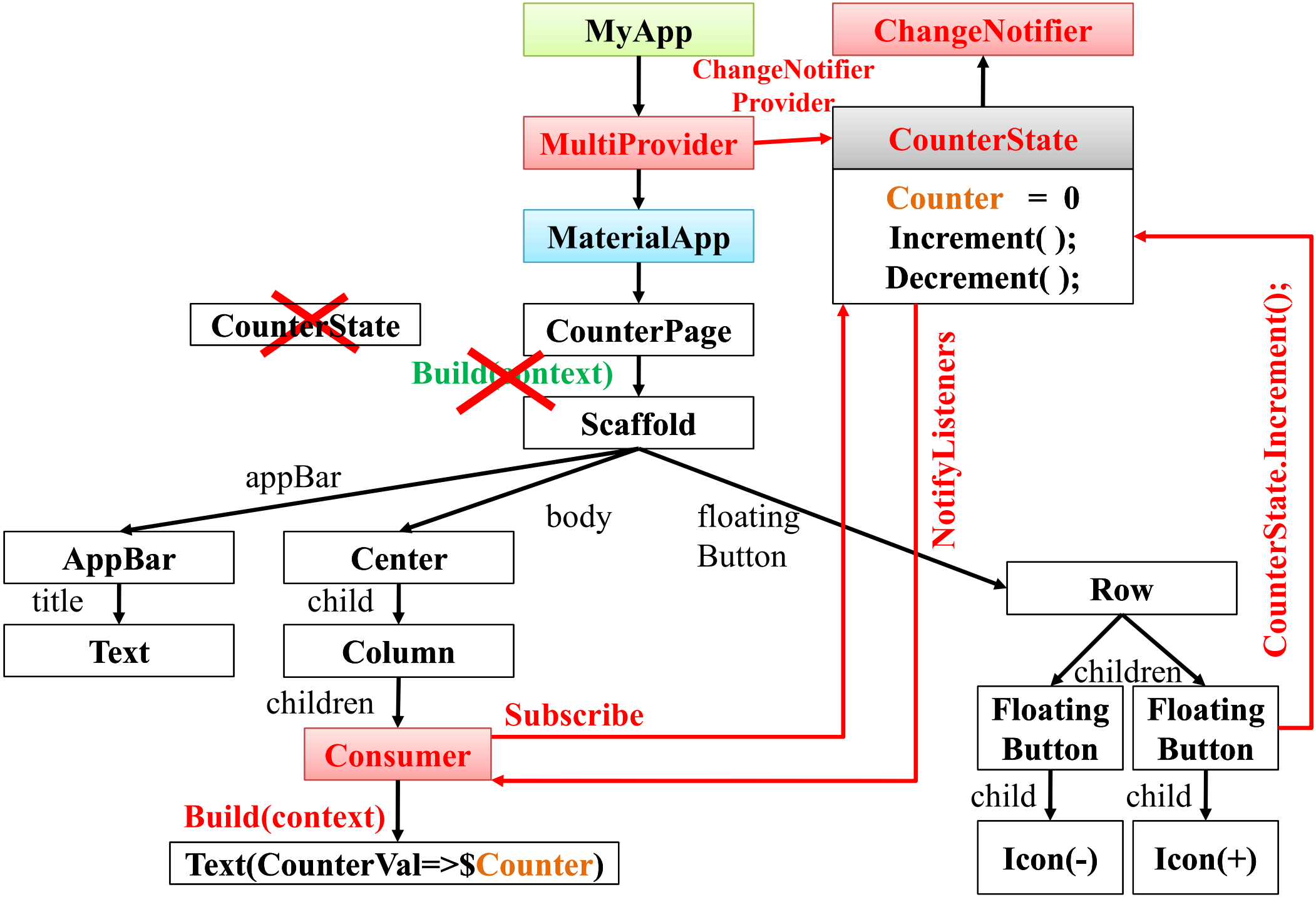
➢ The State is an object of a class that extends **ChangeNotifier** (**observable**) class.

➢ **notifyListener** method is invoked to inform listener widgets about the state change.

➢ **MultiProvider** widget is used to make State available to all app components.

➢ **ChangeNotifierProvider** is used to provide an instance of **ChangeNotifier** (instance of the state).

➢ **Consumer** (**observer**) widget allows a widget to subscribe to listen to a State.


MyApp
ChangeNotifier
ChangeNotifier Provider
MultiProvider
CounterState
Counter = 0
Increment( );
Decrement( );
MaterialApp
CounterState
CounterPage
Build(context)
Scaffold
NotifyListeners
CounterState.Increment();
appBar
body
floating Button
AppBar
title
Text
Center
child
Column
children
Row
children
Floating Button
Floating Button
child
child
Icon(-)
Icon(+)
Subscribe
Consumer
Build(context)
Text(CounterVal=>$Counter)

State Management approaches: Provider

- **Implementation steps:**

  - Adding the provider package dependency.
  - State creation.
  - Expose the State.
  - The use of the State: Consumer (listener)
    & Provider.of (not listener).

State Management approaches: Provider

- **Implementation steps:** adding provider package.

➢ Run the command : *flutter pub add provider*

➢ Or, add the line : *dependencies: provider: ^6.1.1* to the *pubspec.yaml* file.

➢ Run: *flutter pub get* if needed.

➢ Import the package from the App:

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

State Management approaches: Provider

- **Implementation steps:** state creation.

```
import 'package:flutter/material.dart';
class CounterState extends ChangeNotifier {
        int _counter = 0; // private  variable representing the state

        void increment() {
                + +_ counter;
                notifyListeners();
        }

        void decrement() {
                - - _ counter;
                notifyListeners();
        }
        // Getter method
        get valueCounter => _counter;
}
```

State Management approaches: Provider

- **Implementation steps:** making the state available.

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

@override
Widget build (BuildContext context) {

return MultiProvider(
            providers: [ChangeNotifierProvider (create: (context) => CounterState())],
            child: MaterialApp ( theme: ThemeData(useMaterial3: true,),
                                    home: ProviderCounterPage(),
                                ), //MaterialApp
                        ); // MultiProvider
            } // build
} // MyApp
```

State Management approaches: Provider

- **Implementation steps:** use of the state (**Consumer**).

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'provider_counter.dart';

class ProviderCounterPage extends StatelessWidget {

@override
Widget build (BuildContext context) { // This build method is executed only one time
return Scaffold(
        appBar: AppBar( backgroundColor: Colors.pink,
                        title: Text('Counter App with Provider'),
                ), // AppBar
        body: Center( child: Column(children: <Widget>[
        Consumer<CounterState> ( builder: (context, CS , child) {
                        return Text( 'Counter Val =>${CS.valueCounter}',);
                                }, // builder function
                        ), // Consumer
                ],  // children   ), // Center
```

State Management approaches: Provider

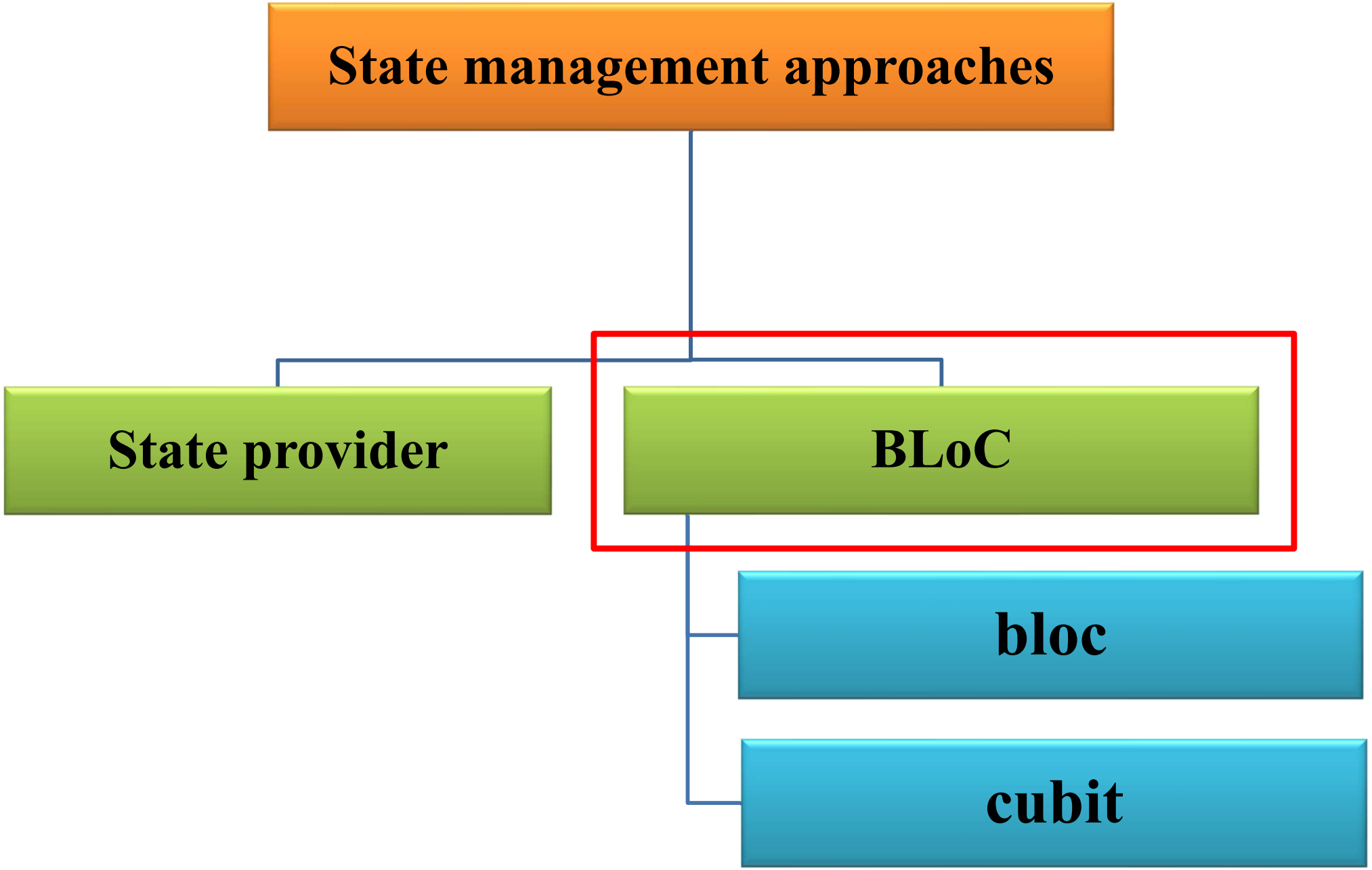
- **Implementation steps:** use of the state (**Provider.of**).

```
floatingActionButton: Row(   children: [

                    FloatingActionButton(
                            child: const Icon(Icons.add),
                            onPressed: () {
Provider.of<CounterState> (context, listen: false).increment(); // Access to the State
                            }, ), // FloatingActionButton

                    FloatingActionButton(
                            child: const Icon(Icons.remove),
                            onPressed: () {
Provider.of<CounterState> (context, listen: false).decrement(); // Access to the State
                            }, ), // FloatingActionButton

                    ], // children )  // Row
            ); //  Scaffold
} // build
```

It's compulsory to avoid rebuilding the entire widgets tree.

Plan

- Ephemeral states & app states.
- Why State Management is important ?
- **State Management approaches:**
  - ✓ Provider.
  - ✓ **BLoC.**


State management approaches
State provider
BLoC
bloc
cubit

State Management approaches: BLoC

- **BLoC** (**B**usiness **Lo**gic **C**omponent): a state management solution created by *Google* based on the concept of ***blocs***, which are small, isolated pieces of state (data) that can be easily managed.
- It's used to manage the state of **individual** screens or components (widgets), and the state of the **entire** application.
- A ***bloc*** is used to deal with the business logic independently of the view logic.

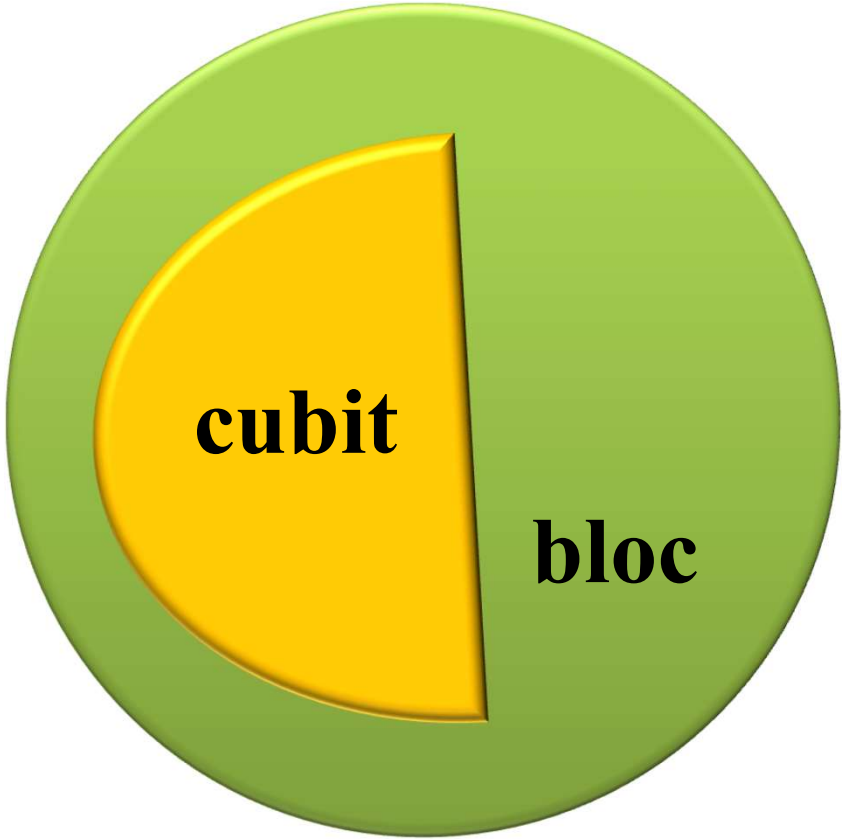
State Management approaches: BLoC

- **Bloc package** provides two implementations: **BLoC** & **Cubit**.

- **Cubit** : another state management solution that follows the principles of the **BLoC**.

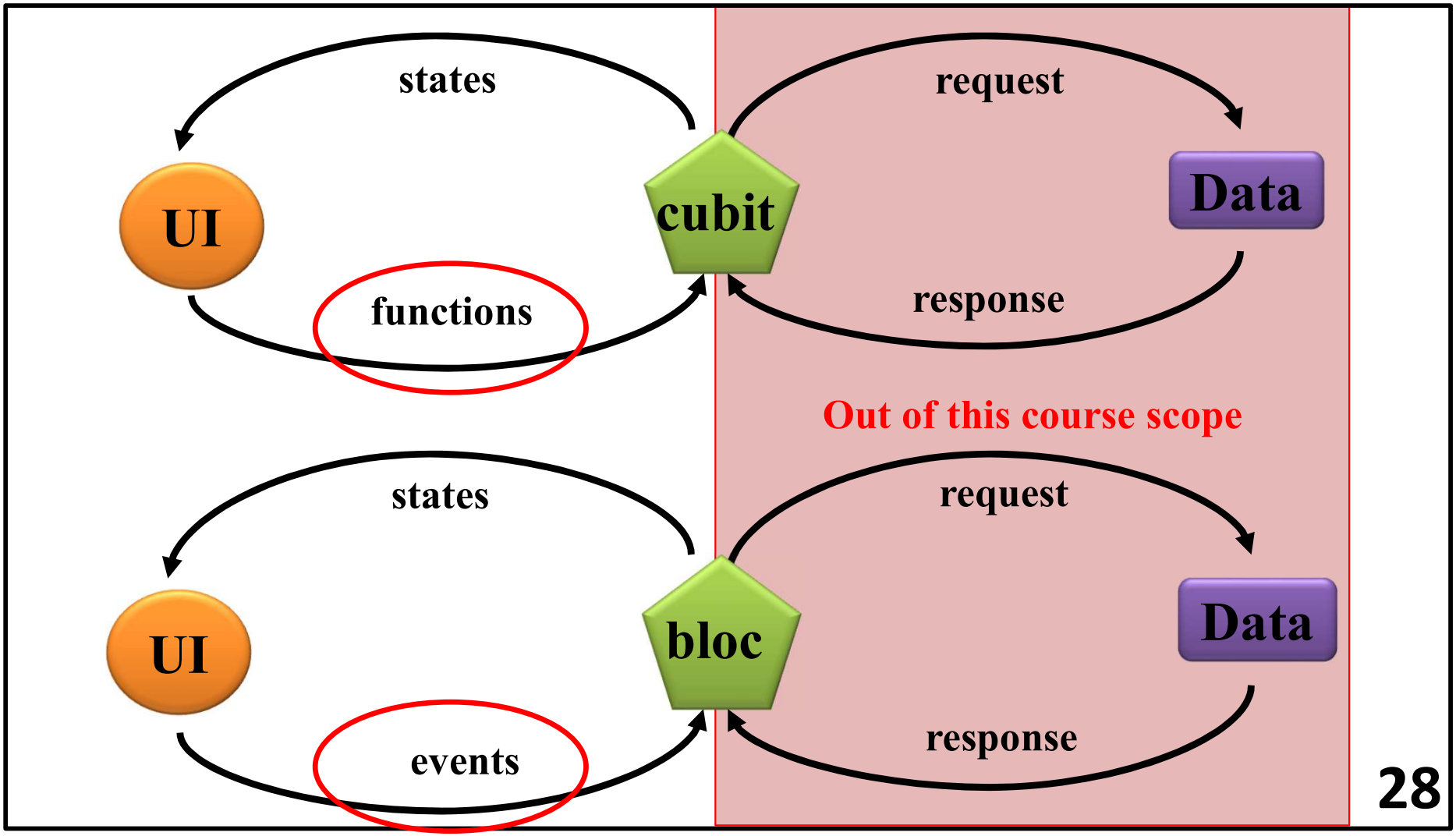
State Management approaches: BLoC

- **BLoC VS Cubit:** cubit is a **minimal** version of bloc i.e. bloc is an **extension** of cubit.



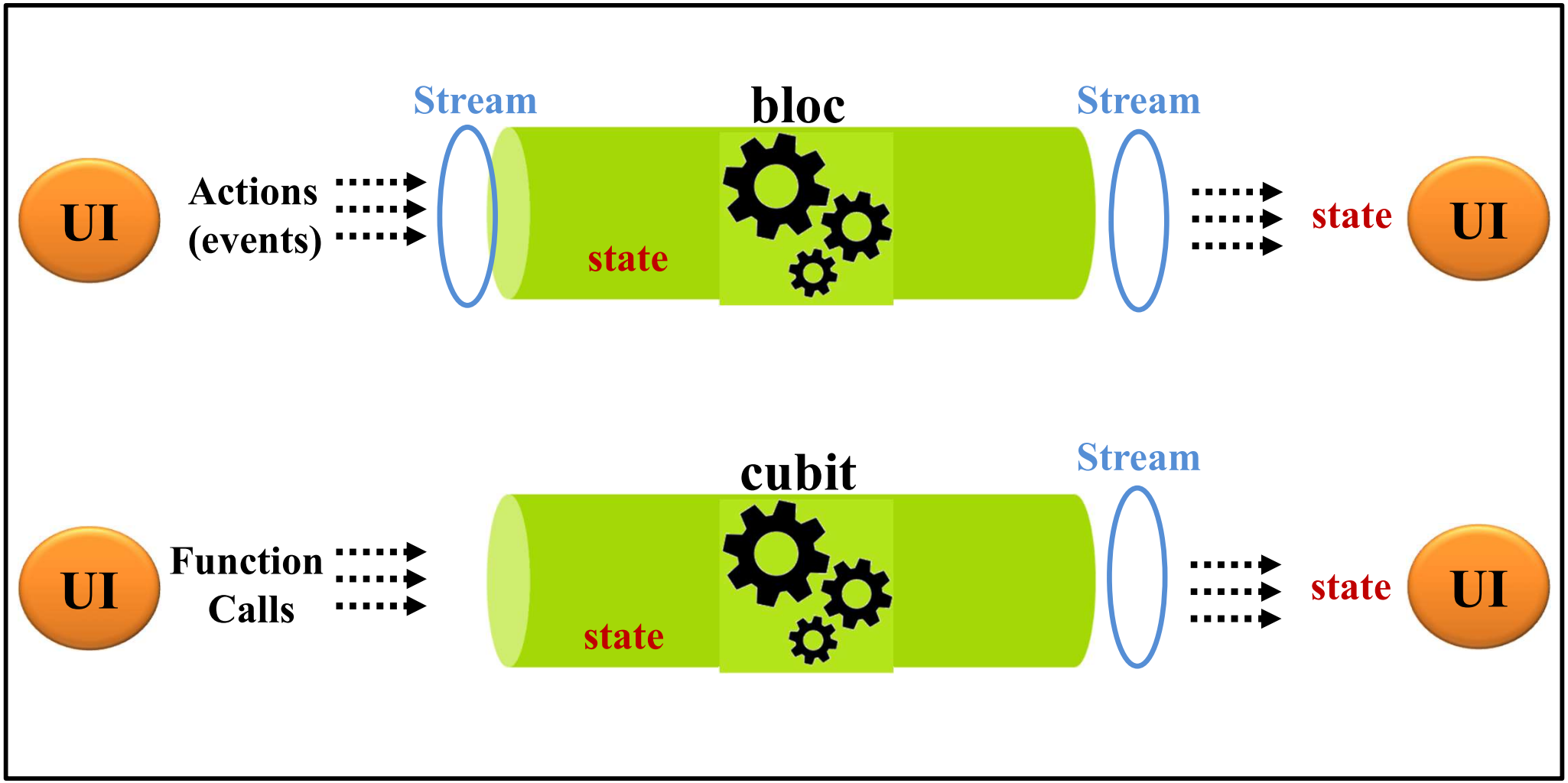
State Management approaches: BLoC

- **BLoC VS Cubit:** both **bloc** and **cubit** can communicate with the outer data layer.



State Management approaches: BLoC

- **BLoC VS Cubit:** overview.

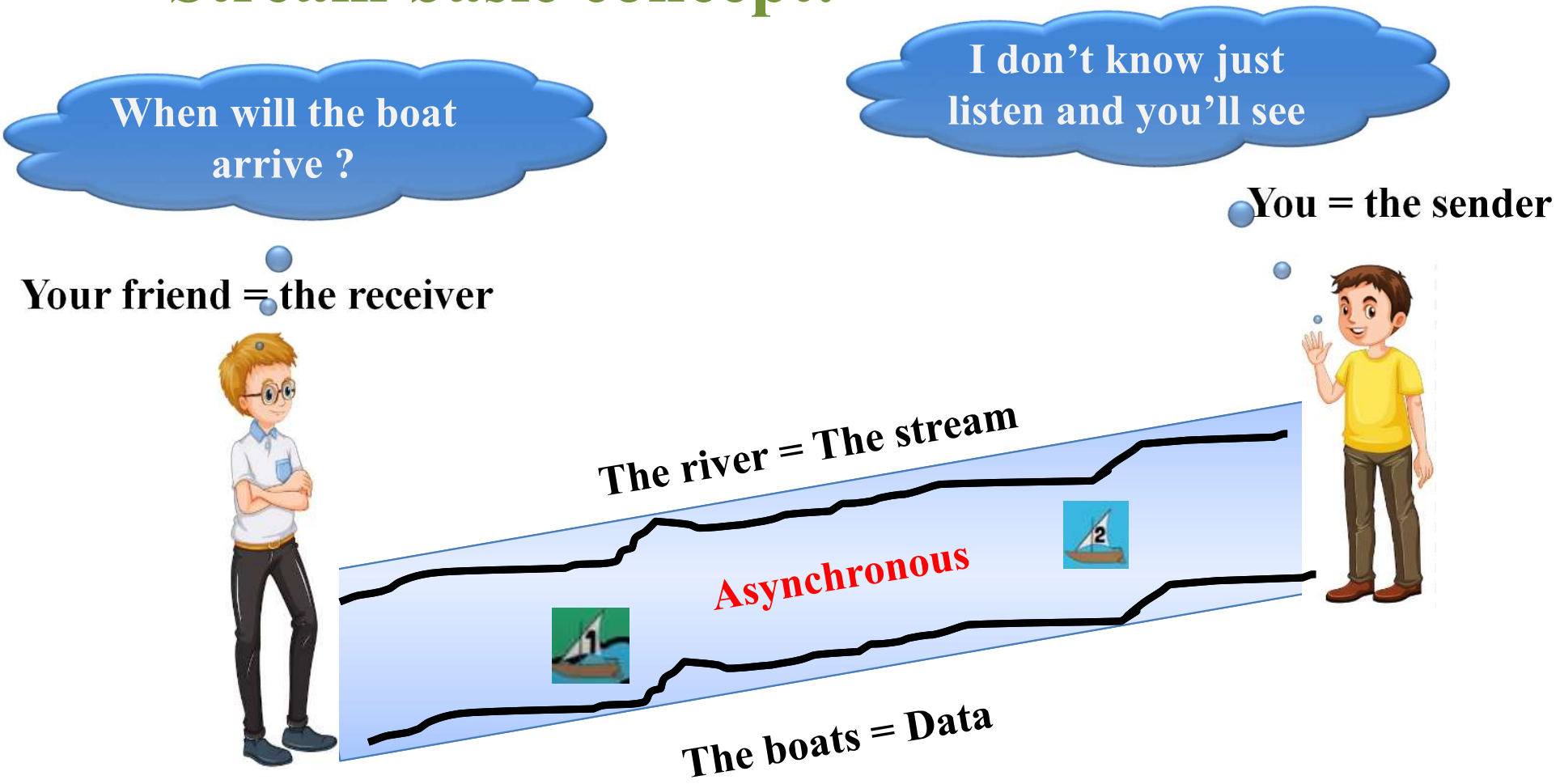


State Management approaches: BLoC

- **Stream basic concept:**

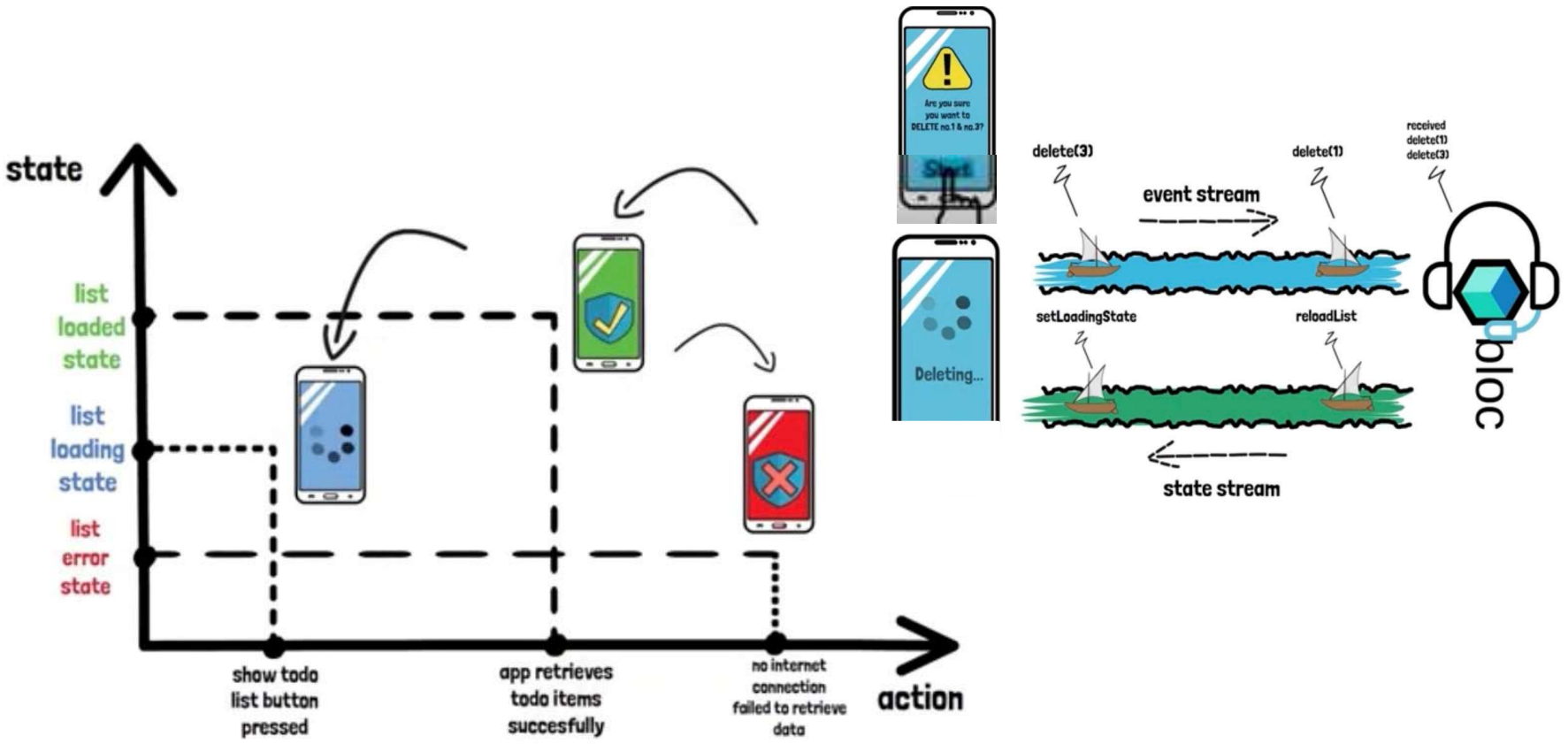
State Management approaches: BLoC

- **Stream basic concept:**



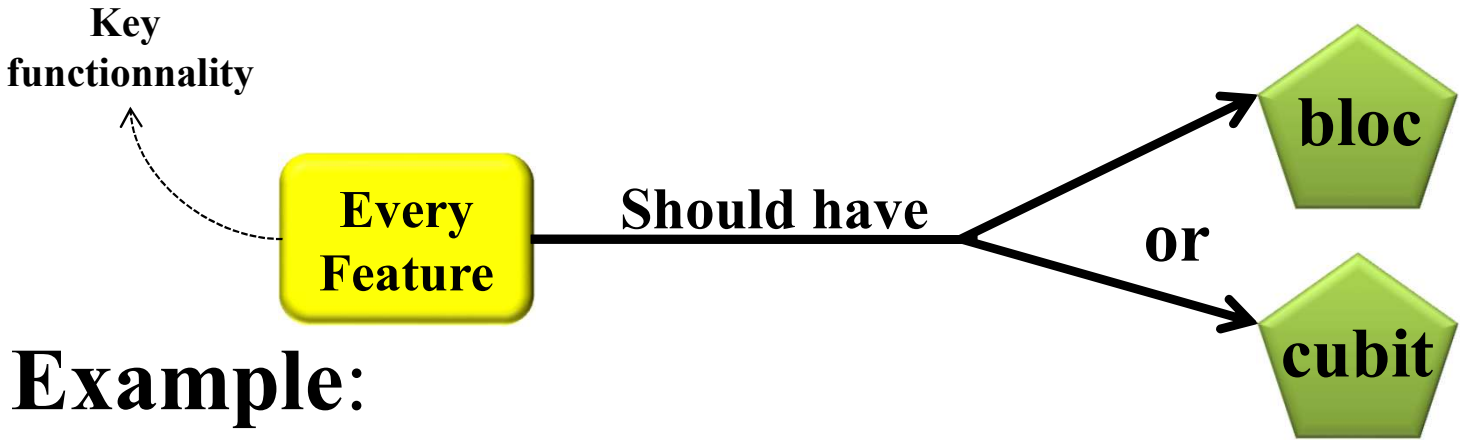
State Management approaches: BLoC

- **Example:**

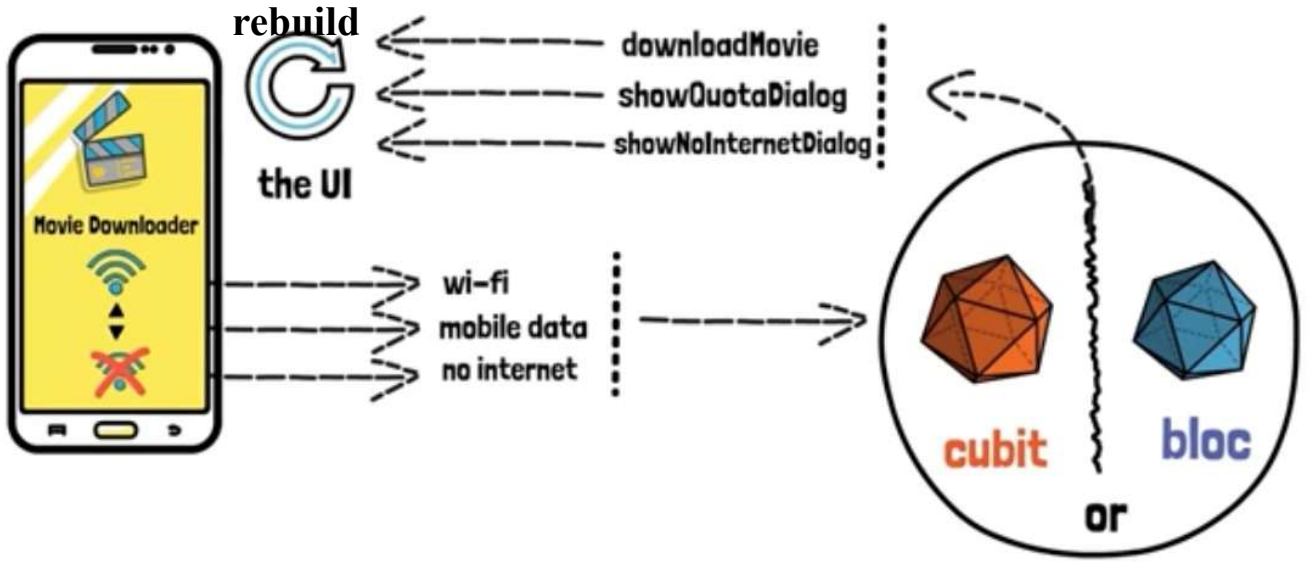


State Management approaches: BLoC

- **When BLoC or Cubit is needed ?** when should I use a bloc/cubit in my app ?

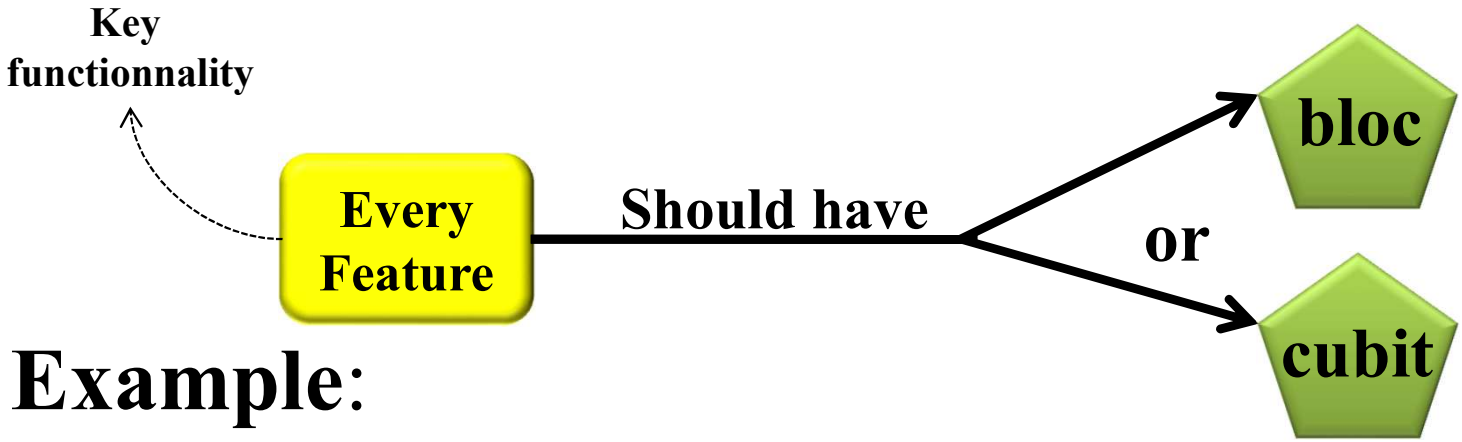


➢ **Example**:

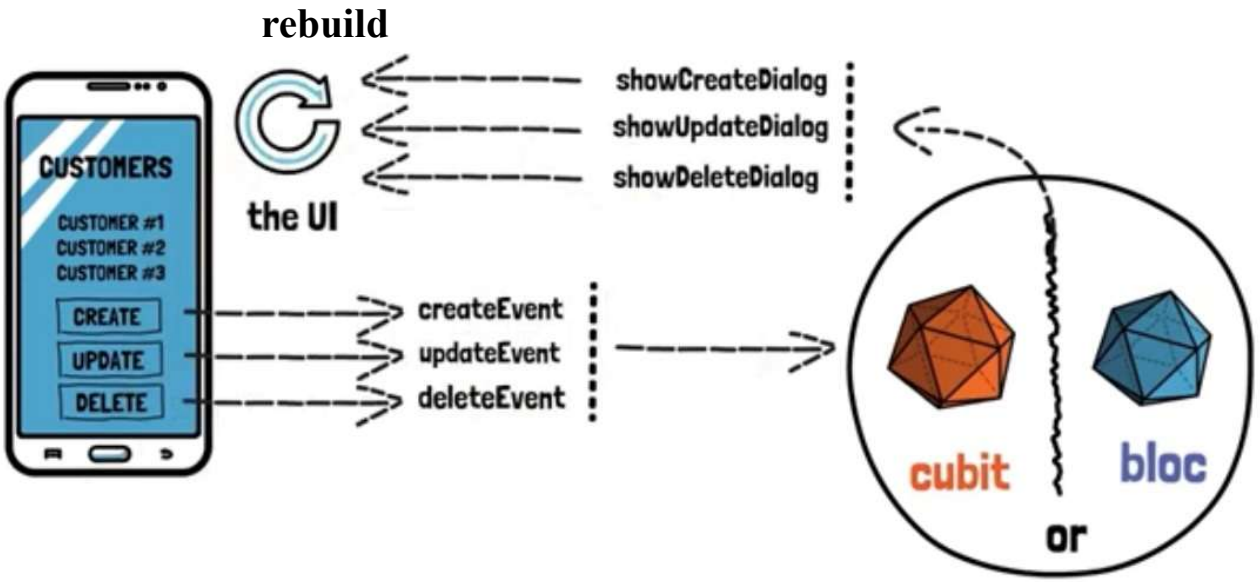


State Management approaches: BLoC

- **When BLoC or Cubit is needed ?** when should I use a bloc/cubit in my app ?



➢ **Example**:



State Management approaches: BLoC

- **Implementation of bloc (e.g. Counter app):**

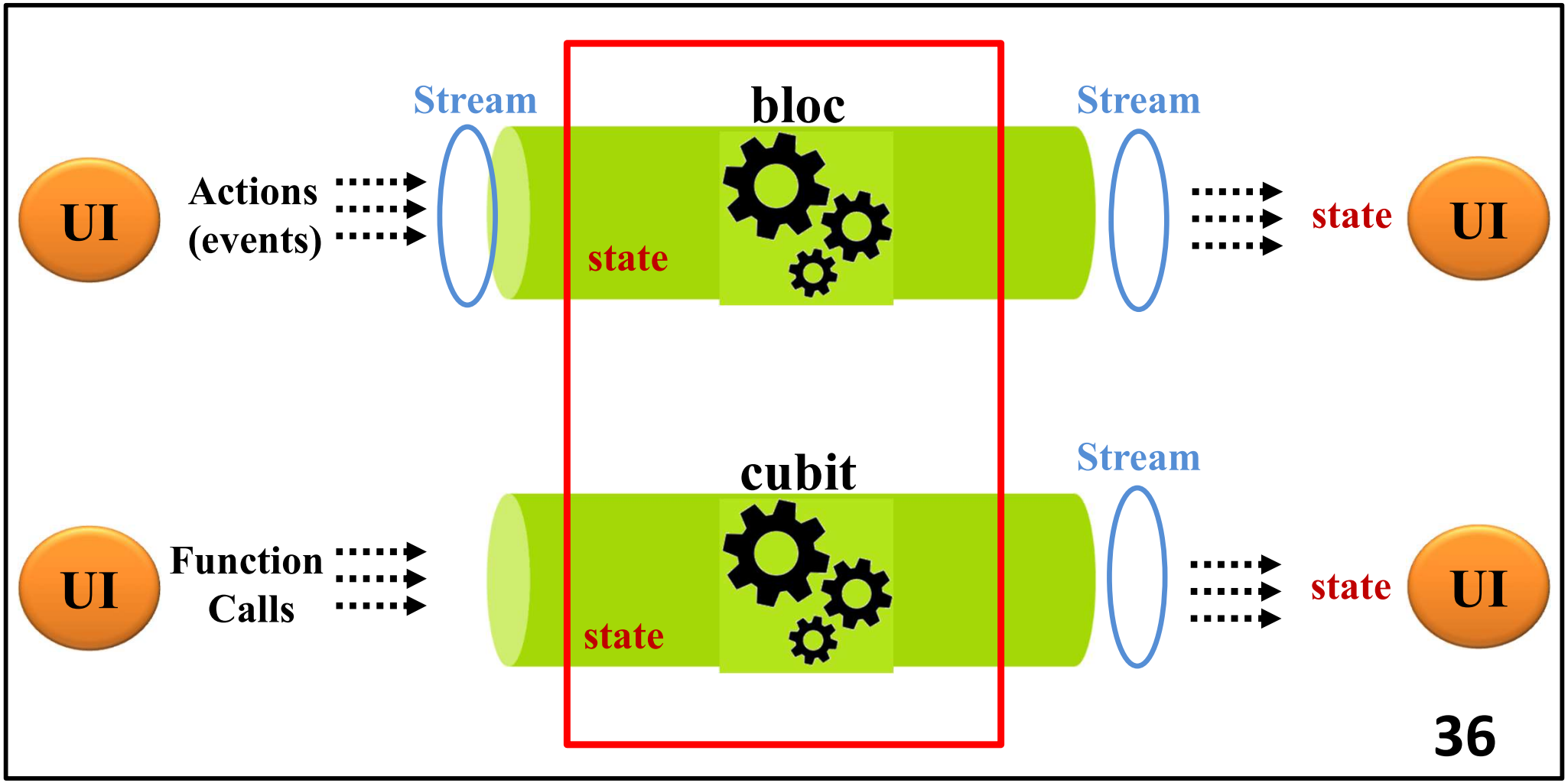
1. What is the initial state ? **It's 0.**

2. What are the possible events that can occur for the feature ?
   Click on one of the + or – buttons (increment event or decrement event).

3. What states can result from these events ? **the counter value is changed (increased or decreased).**

State Management approaches: BLoC

- **Implementation of bloc (e.g. Counter app):** declaring the **bloc** component.



State Management approaches: BLoC

- **Implementation of bloc (e.g. Counter app):** declaring the **bloc** component.

```
// Define the stream of events
enum CounterEvents {increment, decrement} // enum is used for simple events

class CounterBloc extends Bloc < CounterEvents , int> {
        CounterBloc( ):super(0) {

on< CounterEvents > ((event, emit) {
        if (event = = CounterEvents.increment ) {
                emit (state+1); // The state value is automatically provided by the Bloc
class and represents the current state managed by CounterBloc
                }
        else  if (event = = CounterEvents.decrement ) {
                emit (state-1);
                }
            } );
      }
}
```

State Management approaches: BLoC

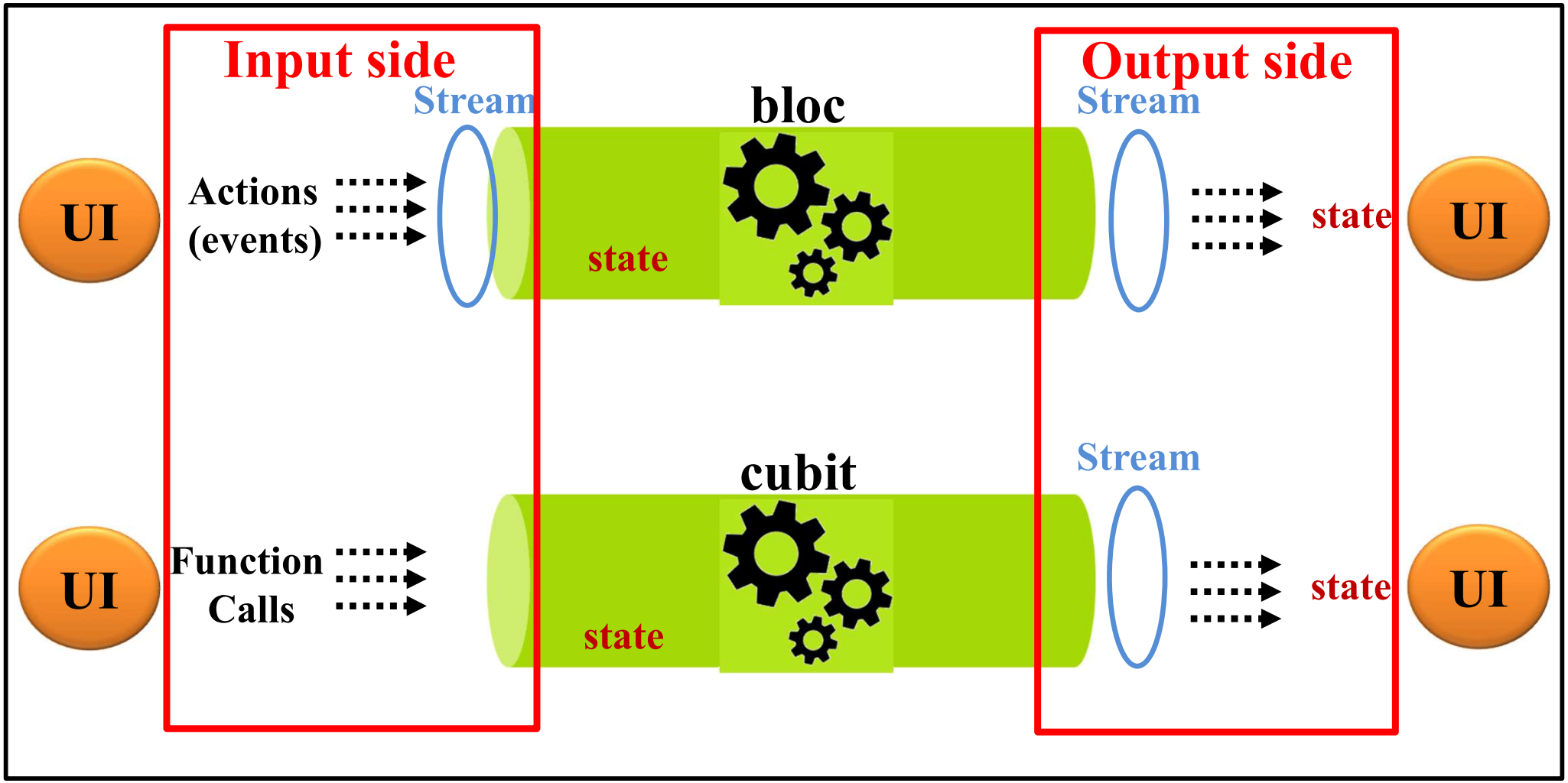
- **Implementation of bloc (e.g. Counter app): OR** declaring the **cubit** component.

```
class CounterCubit extends Cubit <CounterState> {

    CounterCubit( ):super( CounterState (counterValue: 0 ));

    void  increment () => emit (CounterState (counterValue: state.counterValue+1));

    void  decrement () => emit (CounterState (counterValue: state.counterValue-1));

}
```

```
class CounterState { int counterValue;
            CounterState ( {required this.counterValue});
            }
```

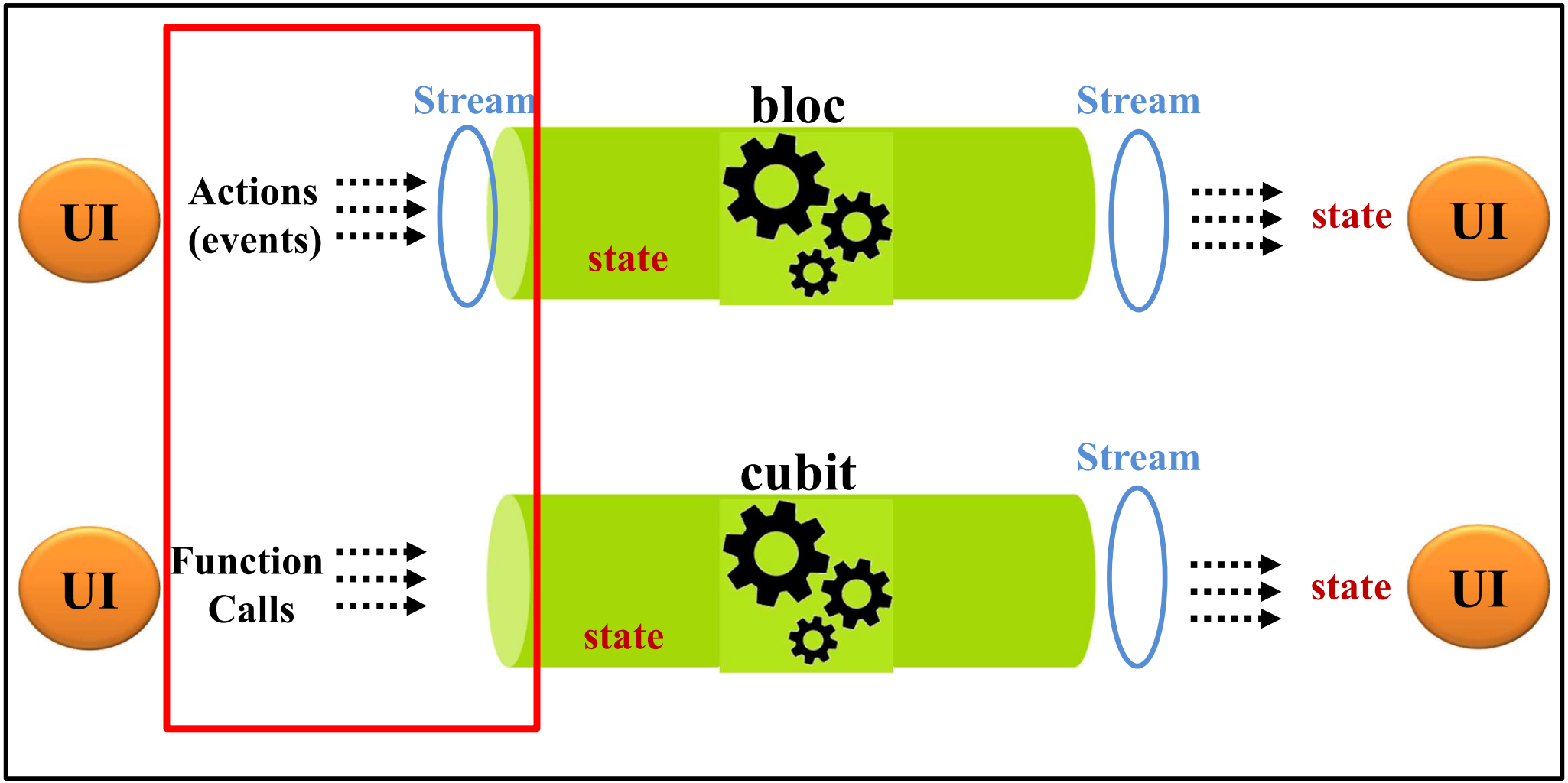
State Management approaches: BLoC

- **Implementation:** the use of Bloc/Cubit.



State Management approaches: BLoC

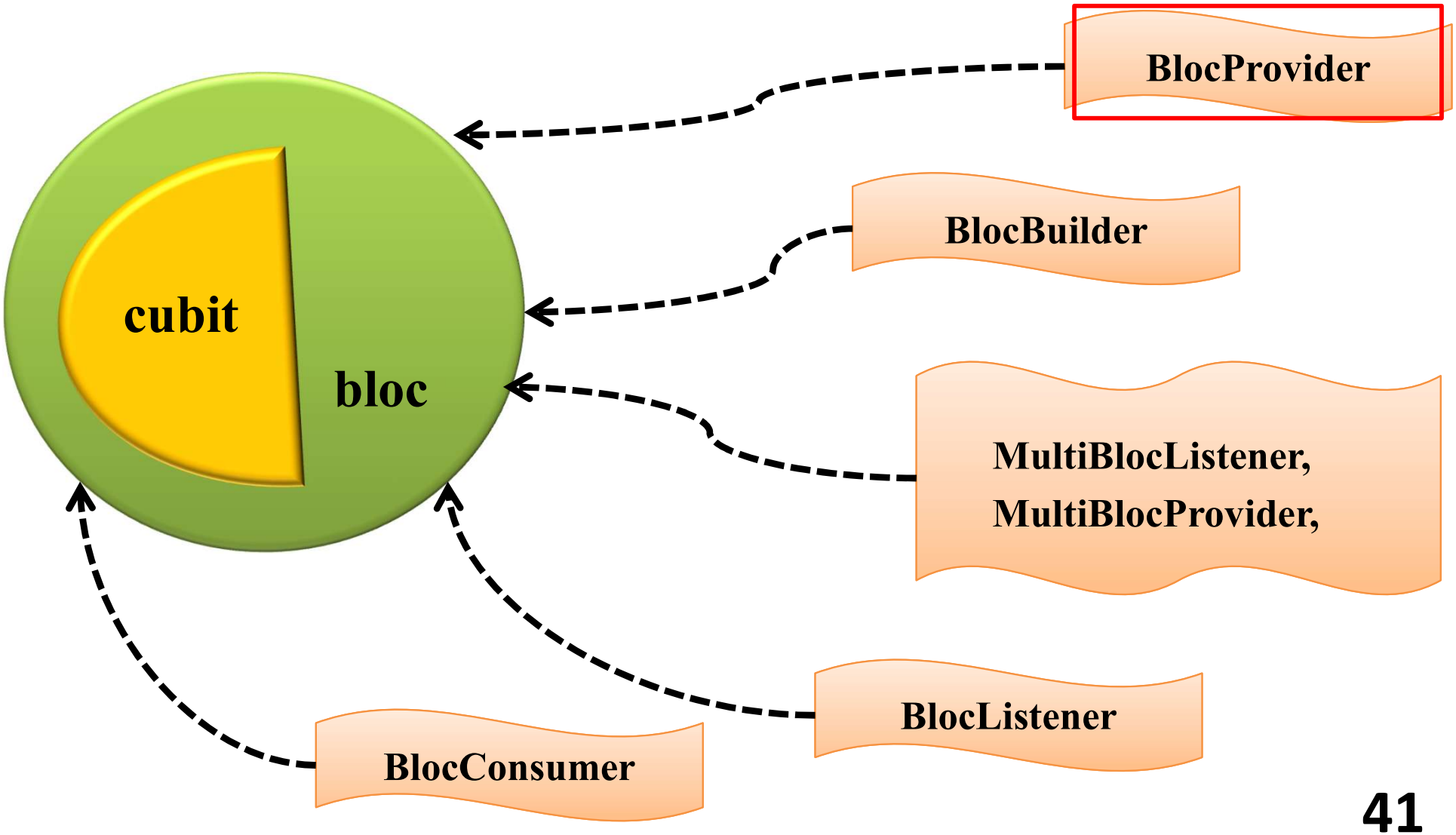
- **Implementation:** the use of Bloc/Cubit.



Send events into the stream

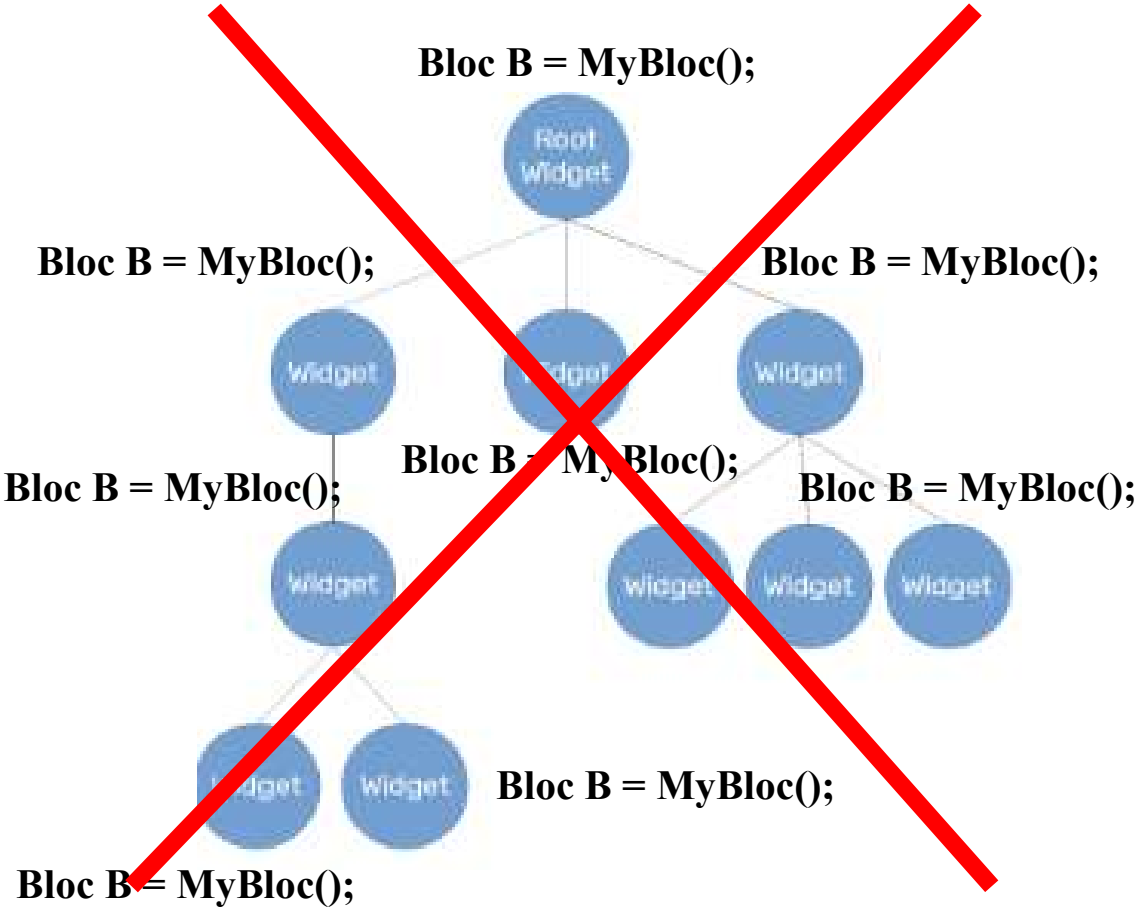
State Management approaches: BLoC

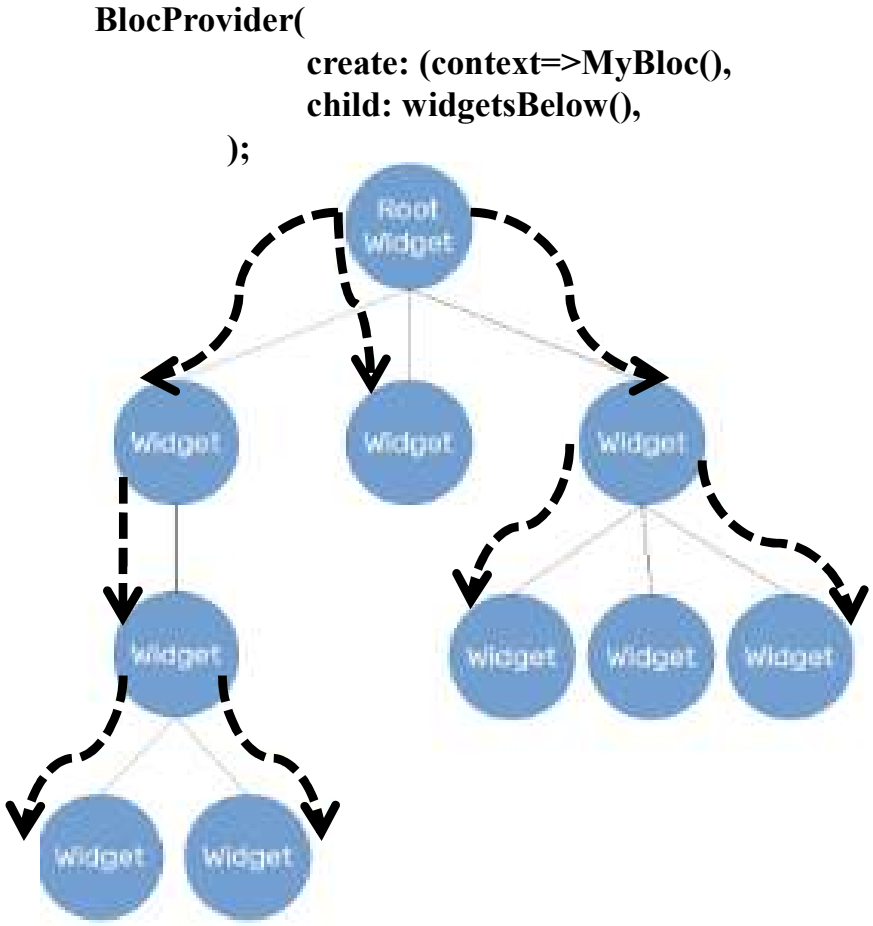
- **The use of BLoC :** how do we link the BLoc to the UI ? More concepts are needed.



State Management approaches: BLoC

- The use of BLoC : **BlocProvider** widget.



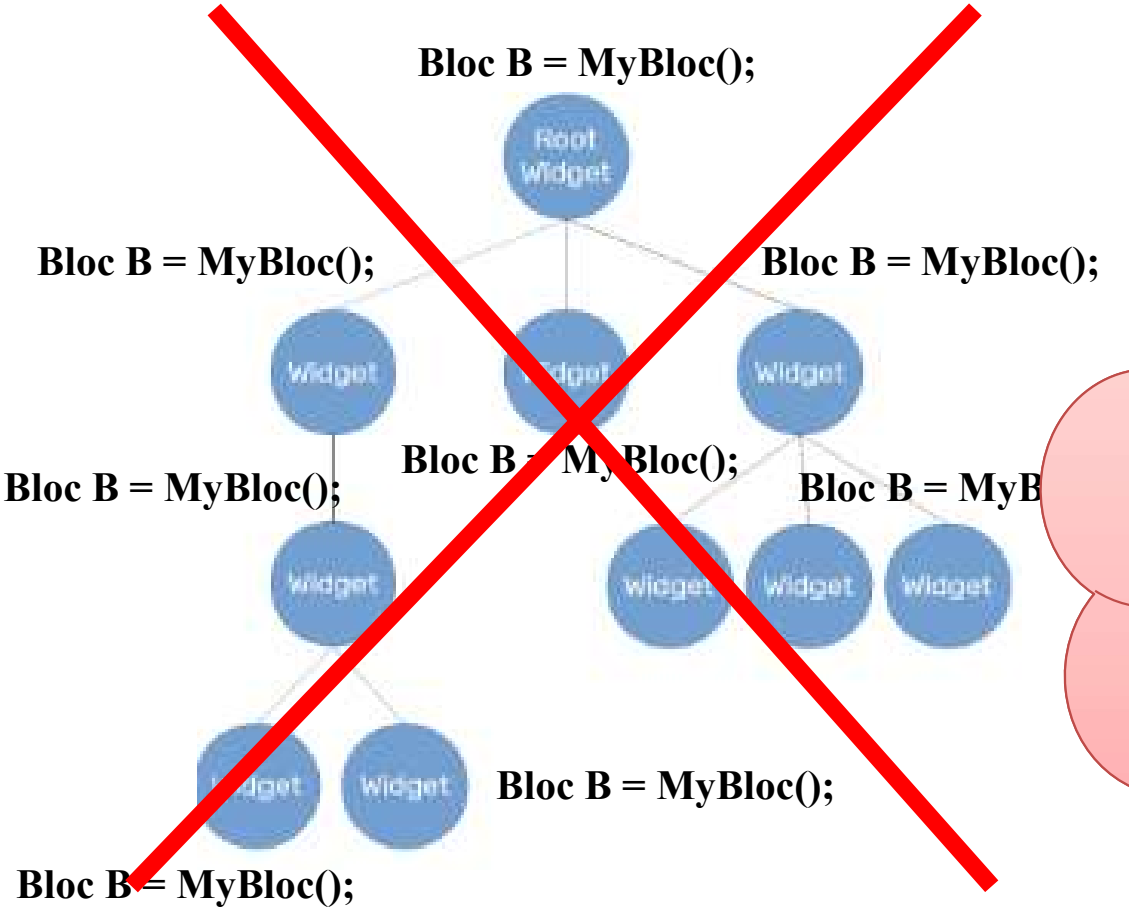


Extremely WRONG

The correct way

State Management approaches: BLoC

- The use of BLoC : **BlocProvider** widget.




```
BlocProvider(
        create: (context=>MyBloc(),
        child: widgetsBelow(),
);
```

BlocProvider provides a SINGLE INSTANCE of a BLoc to the subtree below it.

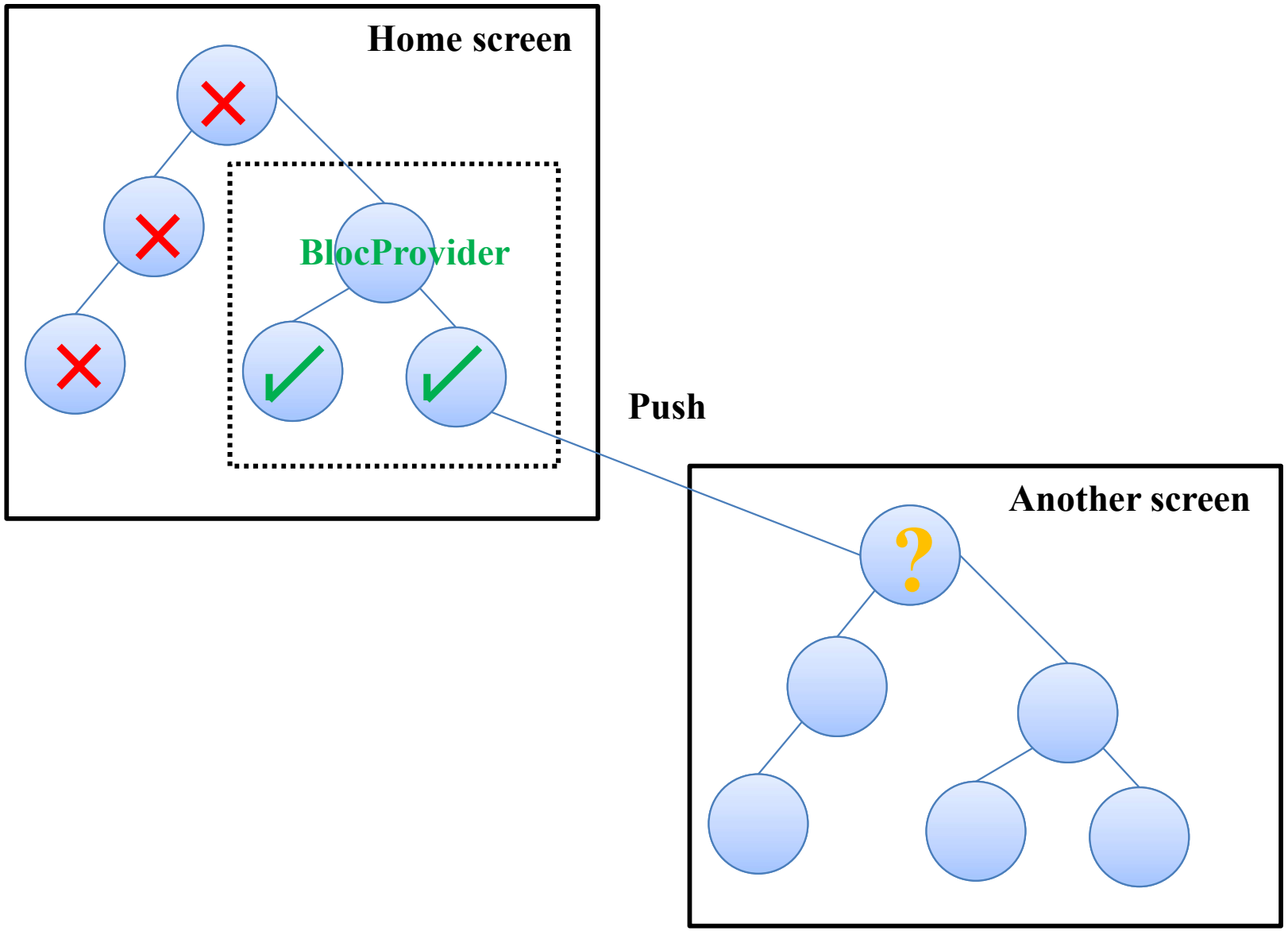
And closes automatically the blocs.

Extremely WRONG

The correct way

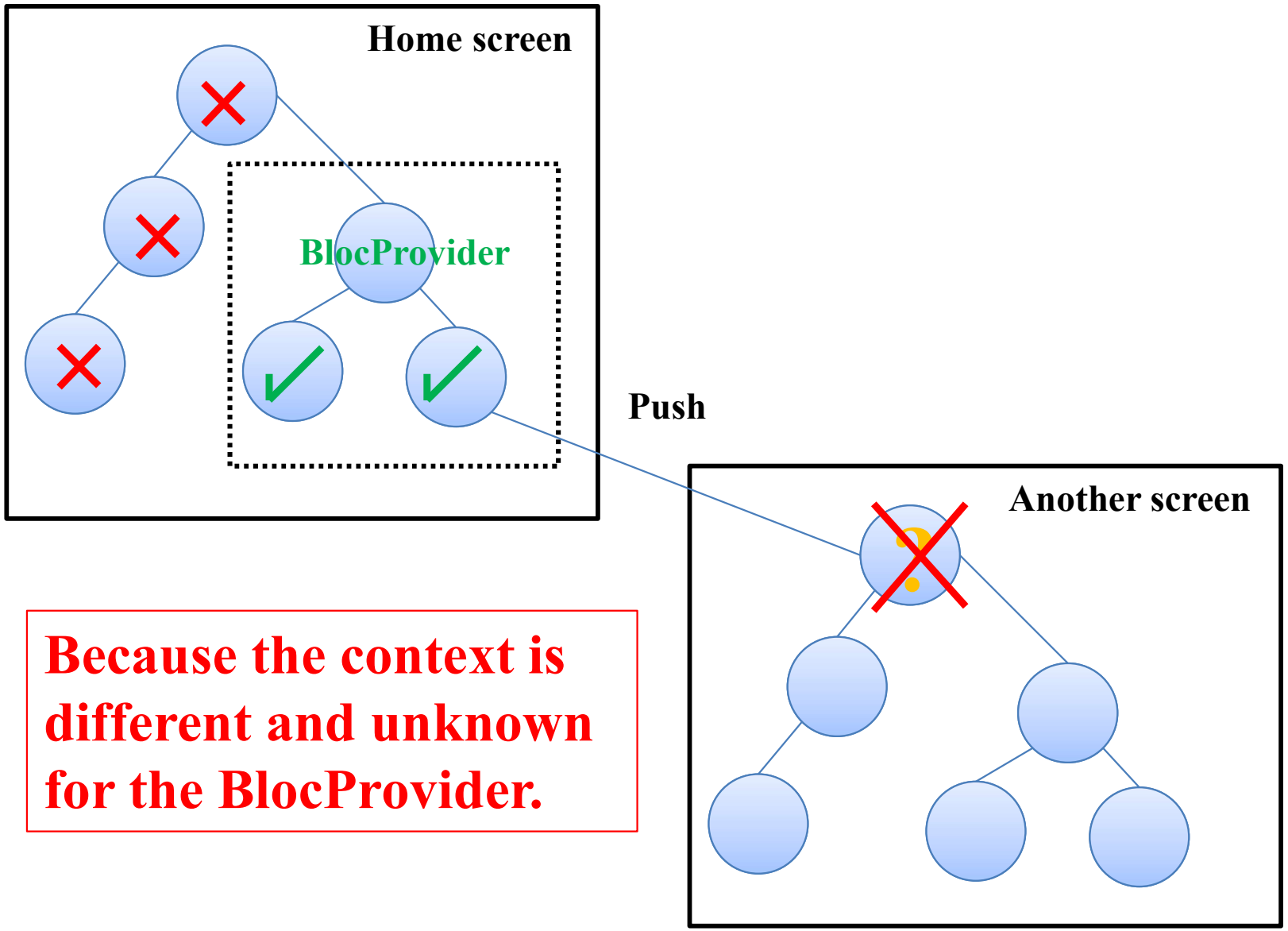
State Management approaches: BLoC

- The use of BLoC : **BlocProvider** availability.



State Management approaches: BLoC

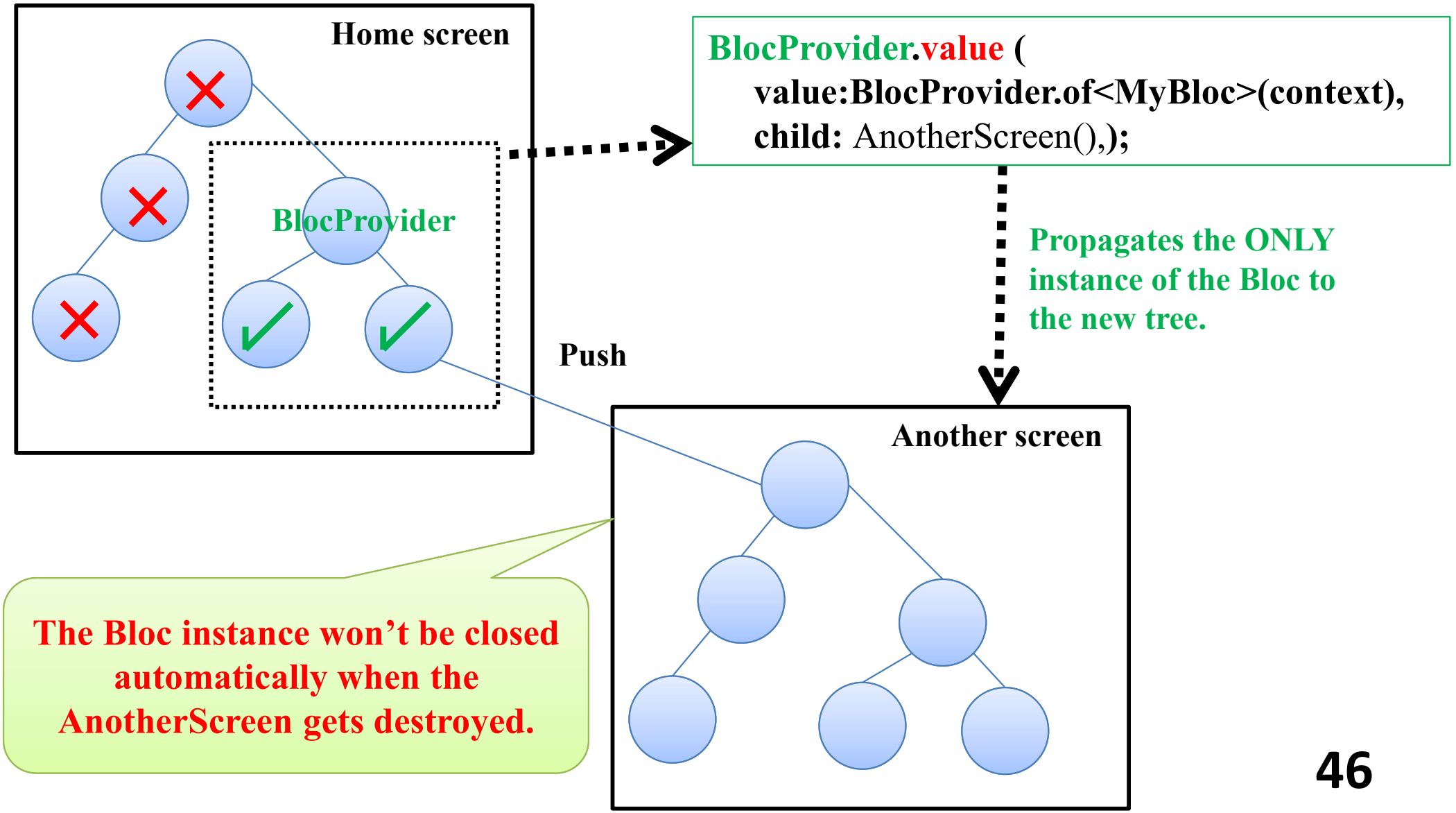
- The use of BLoC : **BlocProvider** availability.



Because the context is different and unknown for the BlocProvider.

State Management approaches: BLoC

- The use of BLoC : **BlocProvider** availability.



State Management approaches: BLoC

- **The use of BLoC : BlocProvider** implementation.

```
BlocProvider(
//  use the function create to create a single instance of the bloc MyBloc
      create: (BuildContext context =>MyBloc( ),  //  Context provides information on
                                                    the position  of its widget in the widgets tree
      child: childX( ),
     );
```

- To access the provided instance of MyBloc from the widget subtree, use:

```
BlocProvider.of<MyBloc> (context);

        OR

context.read <MyBloc> ( );
```

State Management approaches: BLoC

- **The use of BLoC : BlocProvider** implementation.

Create a unique instance of CounterBloc/CounterCubit

```
@override
Widget build (BuildContext context){
return BlocProvider(
            cerate: (context) => CounterCubit ( ), // Or CounterBloc
            child: MaterialApp( title:"BlocProvider Demo",
                                        home: CounterPage( ););
            ); // BlocProvider
} // build
```

State Management approaches: BLoC

- **The use of BLoC : BlocProvider** implementation.

Access to the created instance of CounterBloc/CounterCubit

```
class CounterPage extends StatelessWidget {

@override
Widget build (BuildContext context) {
return Scaffold(
        body: Center( child: Column(children: <Widget>[
                FloatingActionButton (
                onPressed: () {
BlocProvider.of<CounterCubit >
                        (context).decrement();
                            }, // onPressed
                child: Icon (Icons.remove),
                ), // FloatingActionButton
        ],  // children   ), // Center
```

State Management approaches: BLoC

- **The use of BLoC : BlocProvider** implementation.

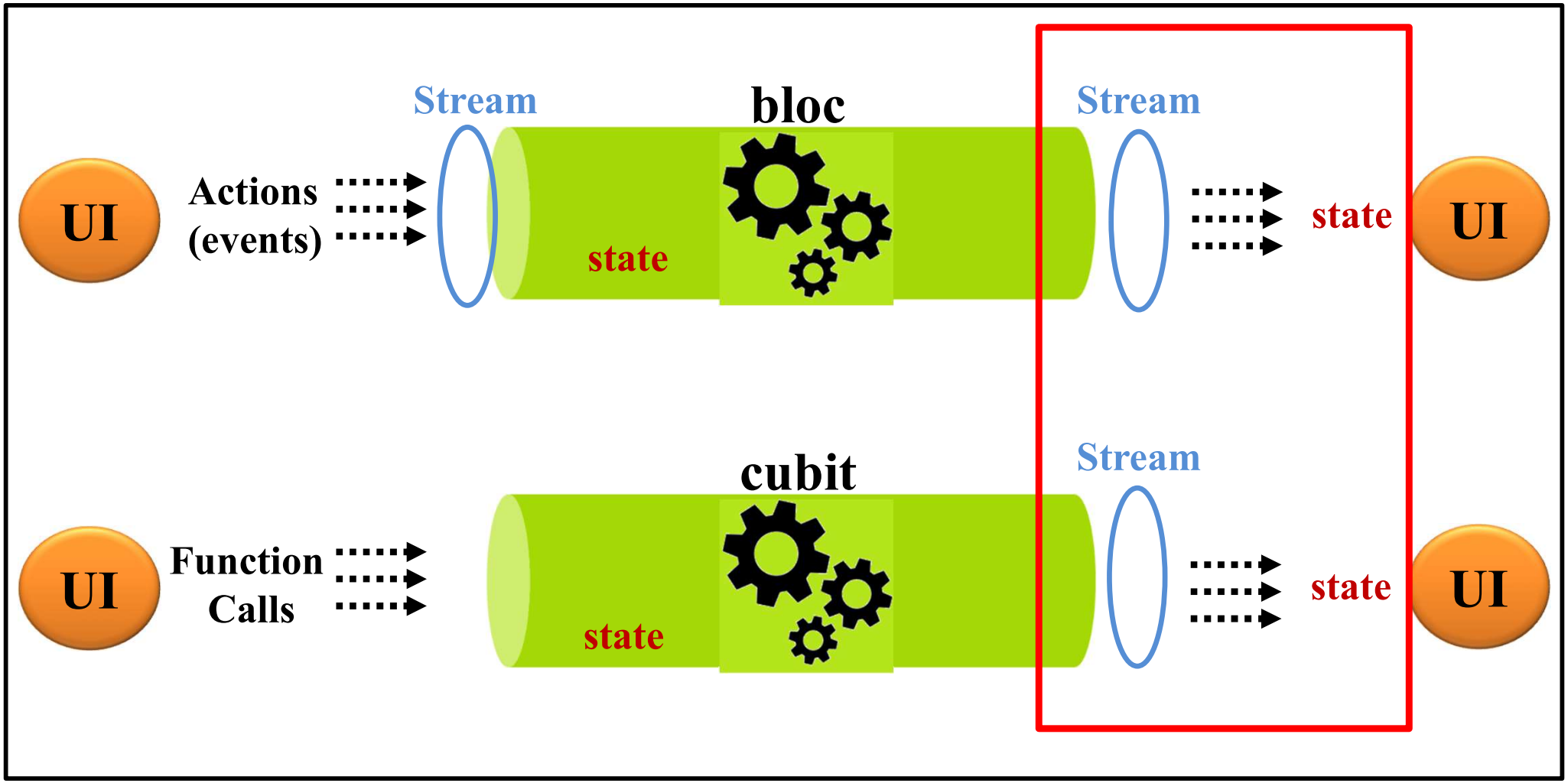
Access to the created instance of CounterBloc/CounterCubit

```
class CounterPage extends StatelessWidget {

@override
Widget build (BuildContext context) {
return Scaffold(
        body: Center( child: Column(children: <Widget>[
                FloatingActionButton (
                onPressed: () {
context.read<CounterBloc >().add(CounterEvent.decrement);
                }, // onPressed
                child: Icon (Icons.remove),
                ), // FloatingActionButton
        ],  // children   ), // Center
```

With Bloc

State Management approaches: BLoC

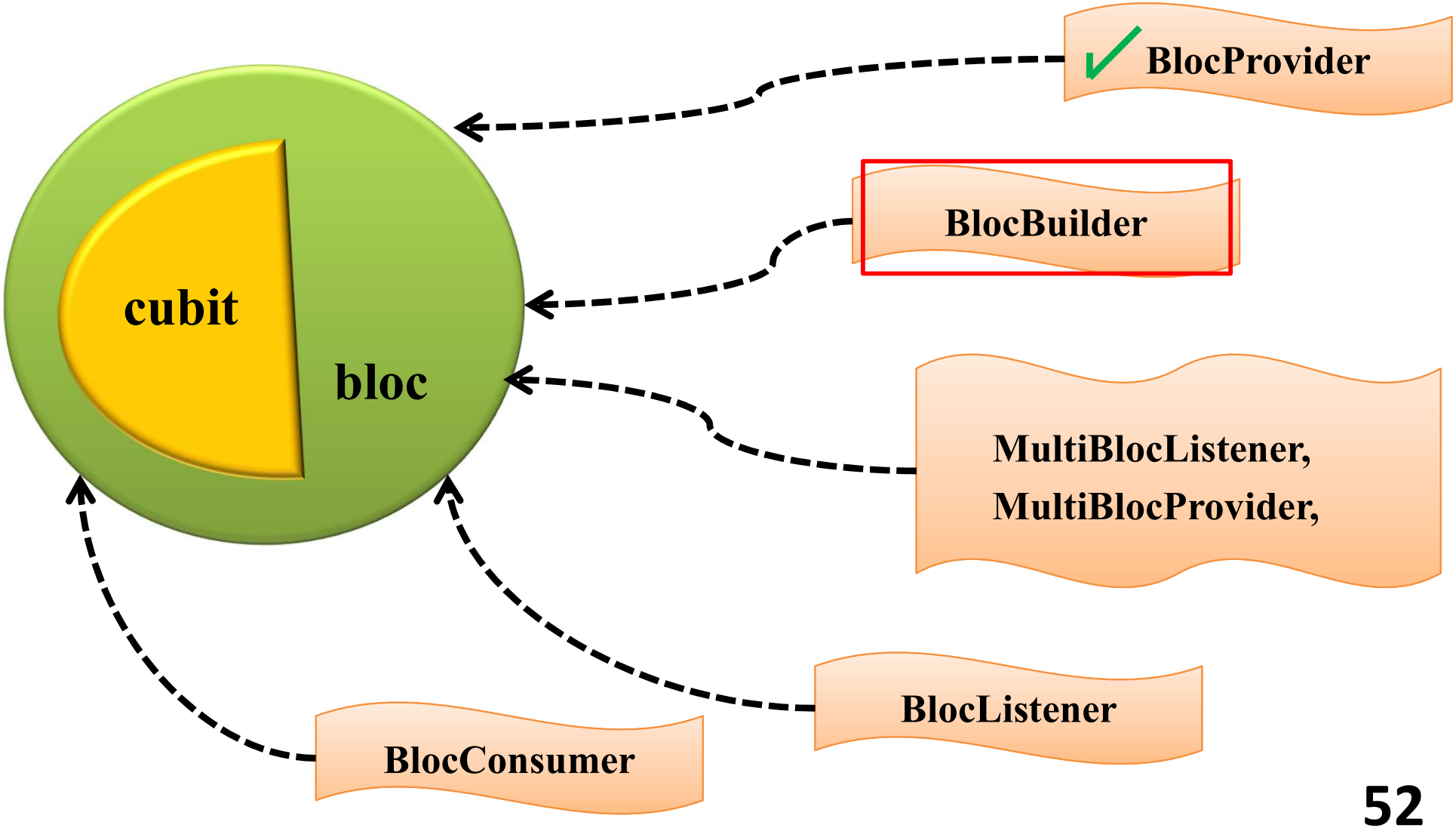
- **Implementation:** the use of Bloc/Cubit.



Consume states from the stream

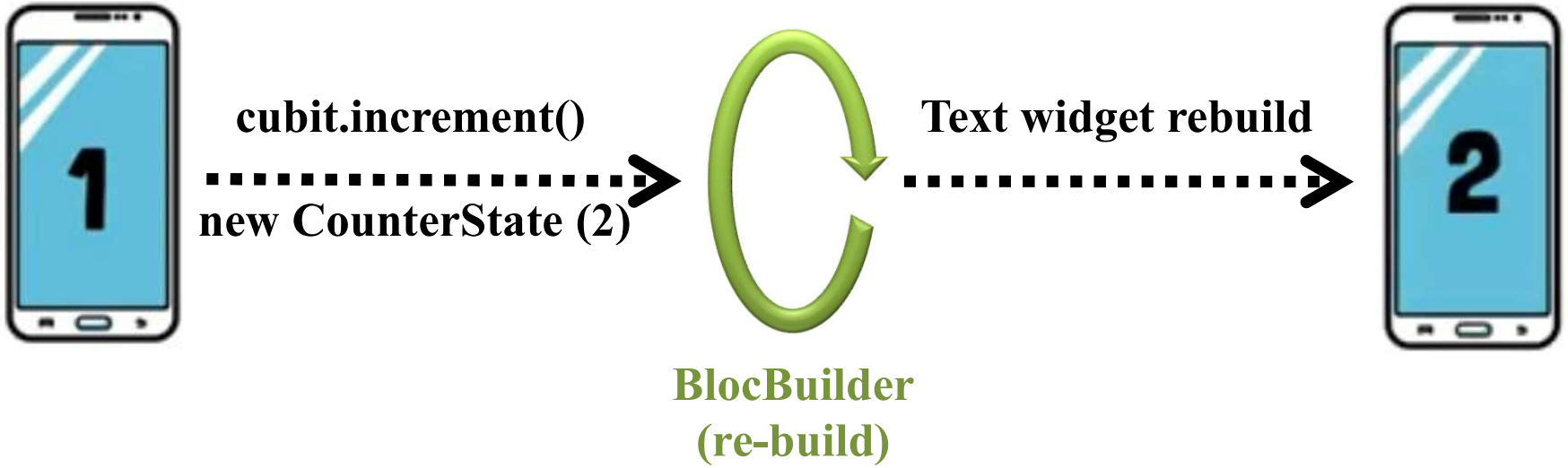
State Management approaches: BLoC

- **The use of BLoC : how to use the new value of the state in the UI ?**



State Management approaches: BLoC

- **The use of BLoC : BlocBuilder** widget.
- **BlocBuilder:** helps **re-building** the UI based on bloc **state changes**.



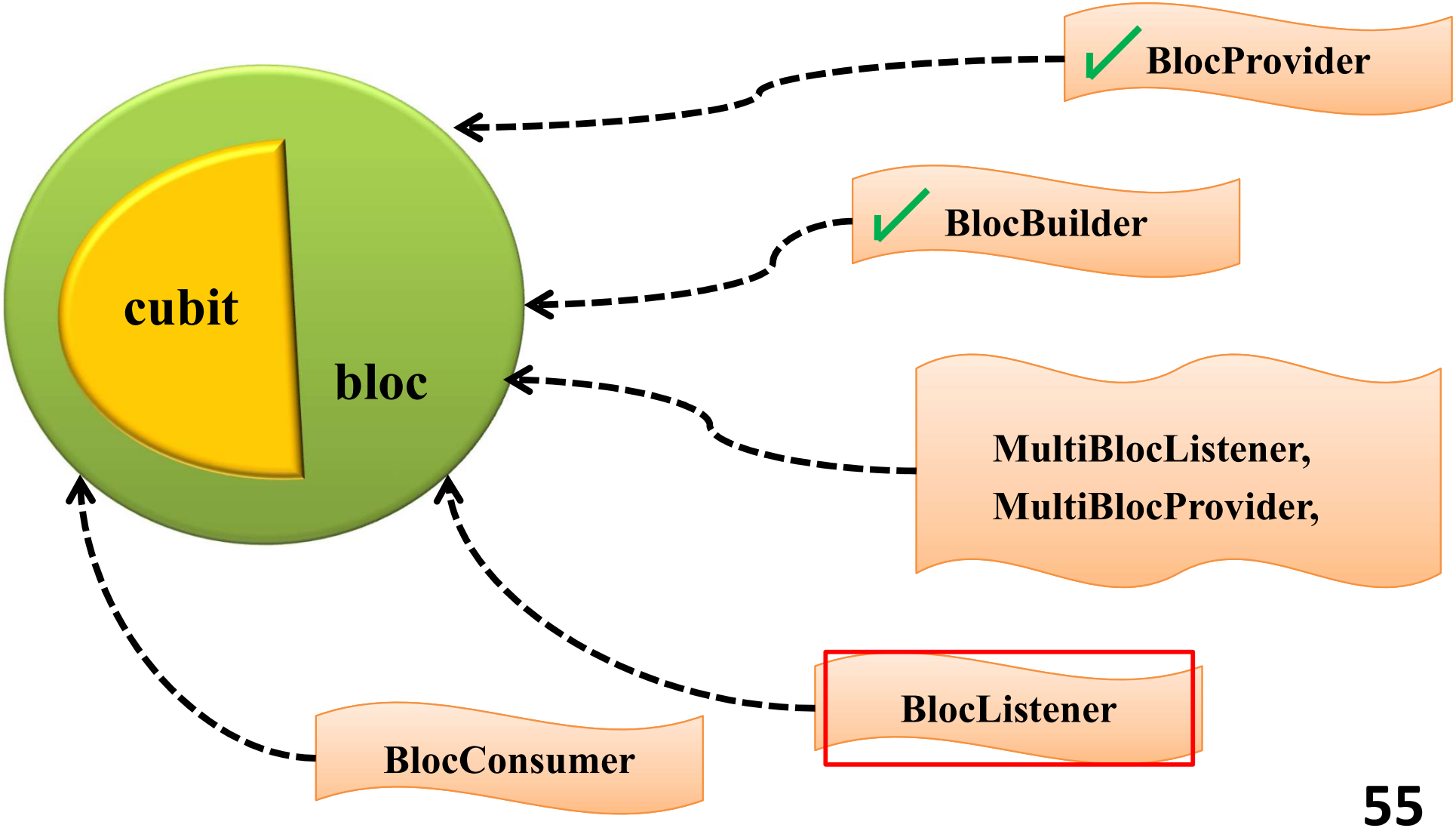
State Management approaches: BLoC

- **The use of BLoC : BlocBuilder** implementation- wrap the widget to rebuild with BlocBuilder.

```
BlocBuilder <CounterCubit, CounterState > (
// cubit: BlocProvider.of<CounterCubit> (context);
builder: (context, state){ // A pure function
// return widget based on CounterCubit state.
return Text(
"The current counter's value is $state.counterValue.toString();"
            ); //Text
}, // builder function
// return true/false to determine whether or not to rebuild the widget
with state.
buildWhen: (previousState, state){
                    if (previousState.value < state.value){ return false;}
), // BlocBuilder
```

State Management approaches: BLoC

- The use of BLoC :



State Management approaches: BLoC

- **The use of BLoC : BlocListener** widget.
- **BlocListener:** is used when the UI is fixed, but there are actions (e.g. navigation, show a popup) needs to be done when the state change.

```
// The same structure as BlocBuilder
BlocListener <CounterCubit, CounterState > (
        listener: (context, state){
                .... ... ...
                } // listener function is called only once per state which
                    makes the difference with BlocBuilder.

        listenWhen: .... // Determine listening or not for a state.

), // BlocListener
```

State Management approaches: BLoC

- **The use of BLoC : BlocListener** implementation- showing a ***snackBar*** displaying which button has been pressed.

```
BlocListener <CounterCubit, CounterState > (
        listener: (context, state){
                        if (state.wasIncremented = = true) {
                        Scaffold.of(context).showSnackBar
                        (SnackBar(content:Text ('INCREMENT'),
                                        duration:Duration(seconds:1),
                                ), // SnackBar
                        ); // showSnackBar } // if
                } // listener function
), // BlocListener
```

State Management approaches: BLoC

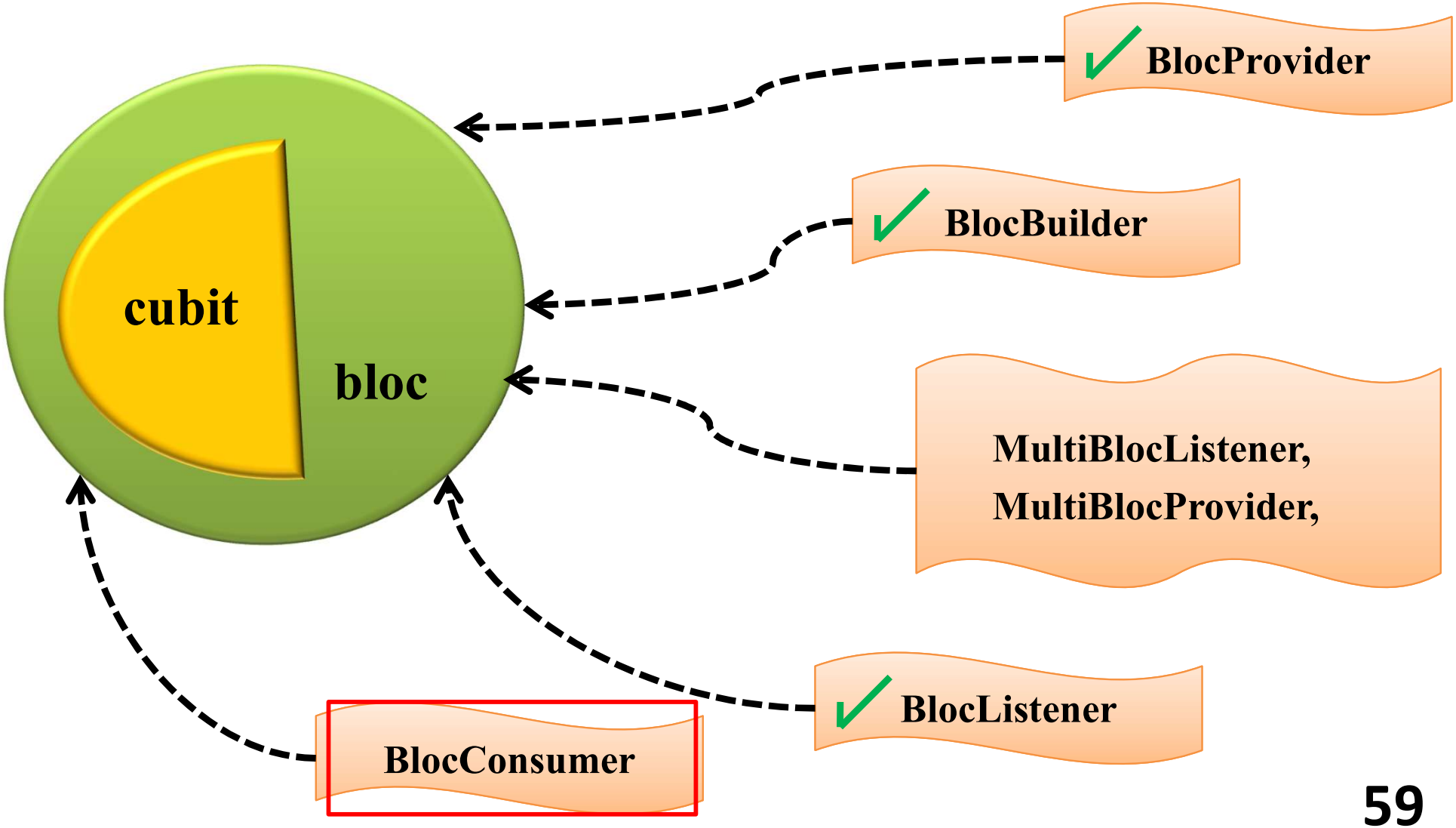
- **The use of BLoC : BlocListener** implementation- showing a ***snackBar*** displaying which button has been pressed.

```
class CounterState {
        int counterValue;
        bool wasIncremented;

        CounterState ( {@required this.counterValue, this. wasIncremented});
}
```

```
class CounterCubit extends Cubit <CounterState> {

    CounterCubit( ):super( CounterState (couterValue: 0 ));

    void  increment () => emit (CounterState (counterValue: state.counterValue+1,
                                              state.wasIncremented: true));
    void  decrement () => emit (CounterState (counterValue: state.counterValue-1,
                                              state.wasIncremented: false));}
```

State Management approaches: BLoC

- The use of BLoC :



State Management approaches: BLoC

- The use of BLoC : **BlocConsumer** widget.
- **BlocConsumer** = **BlocBuilder** + **BlocListener**

```
BlocConsumer <CounterCubit, CounterState > (
        listener: (context, state) {
                        // do stuff here based on CounterCubit state
                }
        builder: (context, state) {
                        // return widget here based on CounterCubit state

                }
), // BlocConsumer
```

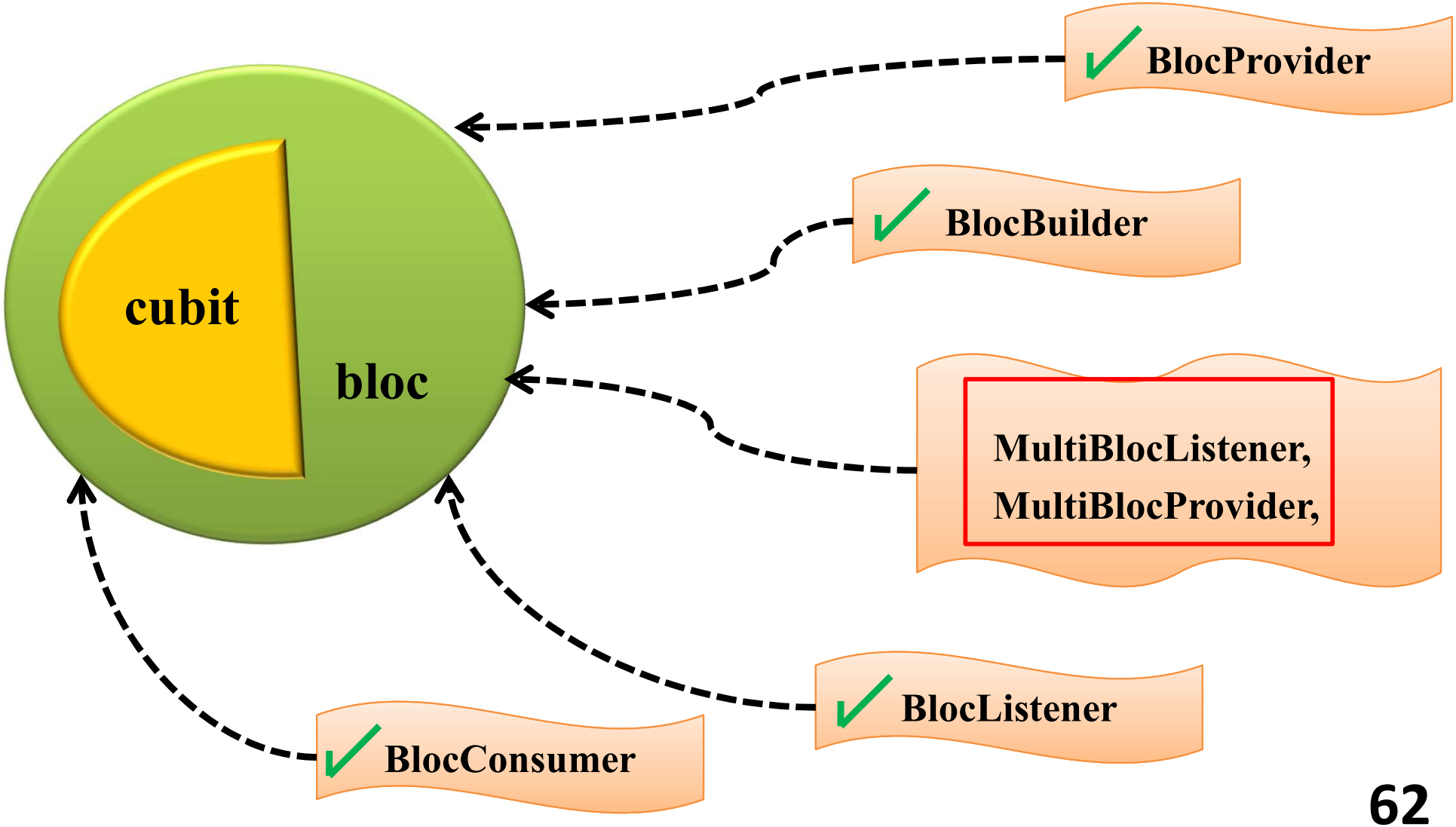
State Management approaches: BLoC

- The use of BLoC : **BlocConsumer** implementation.

```
BlocConsumer <CounterCubit, CounterState > (
        listener: (context, state) {
                if (state.wasIncremented = = true) {
                        Scaffold.of(context).showSnackBar
                        (SnackBar(content:Text ('INCREMENT'),
                            duration:Duration(seconds:1),
                                    ), // SnackBar
                        ); // showSnackBar } // if
                }
        builder: (context, state) {
        return Text( "The current counter's value is
                        $state.counterValue.toString();"); //Text
         }, // builder function
), // BlocConsumer
```

State Management approaches: BLoC

- The use of BLoC :



State Management approaches: BLoC

- The use of BLoC : multiple blocs/cubits

1 bloc BlocProvider

1 bloc BlocBuilder

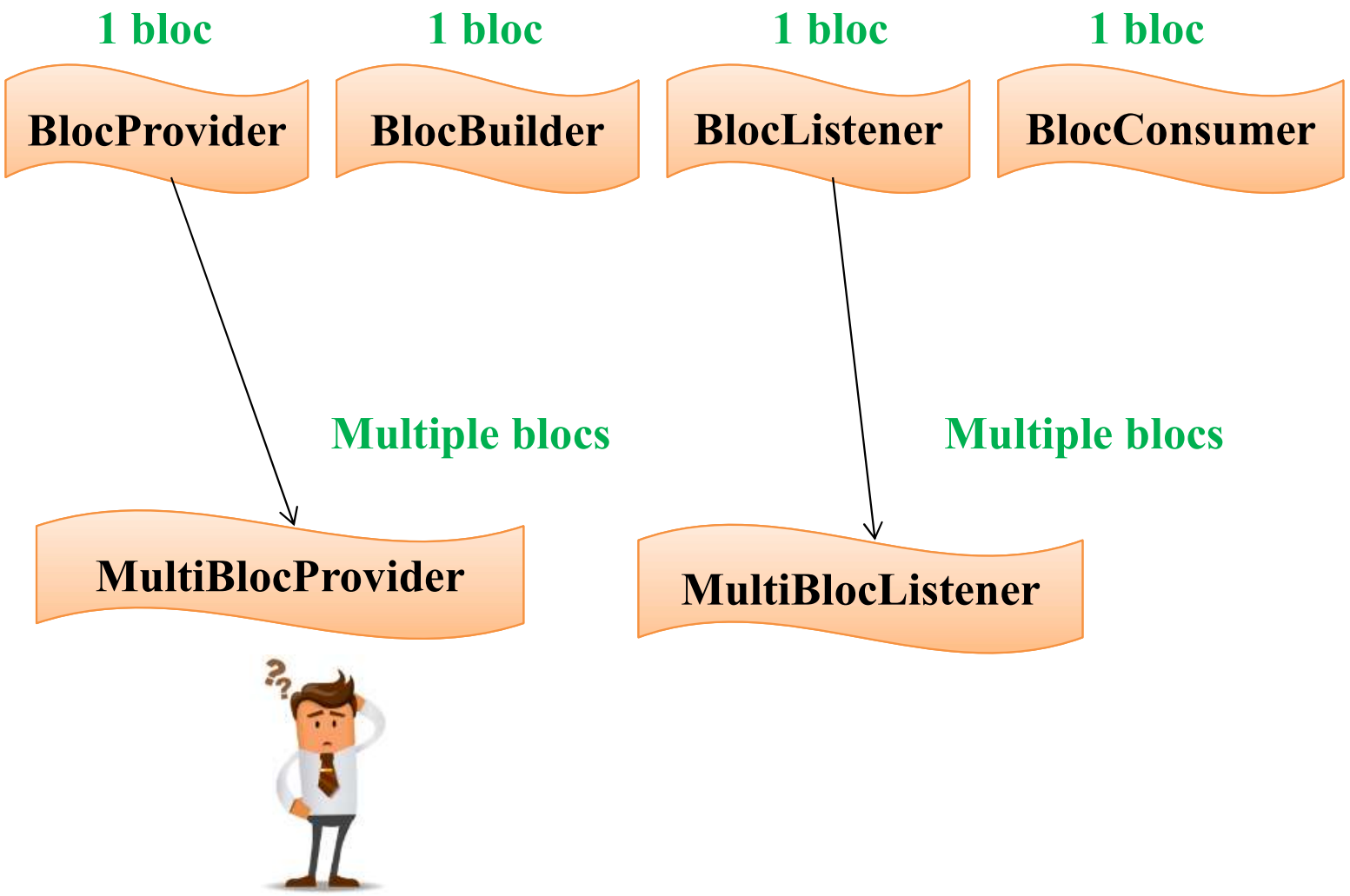
1 bloc BlocListener

1 bloc BlocConsumer

What about more rich app (app with multiple blocs ?

State Management approaches: BLoC

- The use of BLoC : multiple blocs/cubits



State Management approaches: BLoC

- The use of BLoC : multiple blocs/cubits

MultiBlocProvider

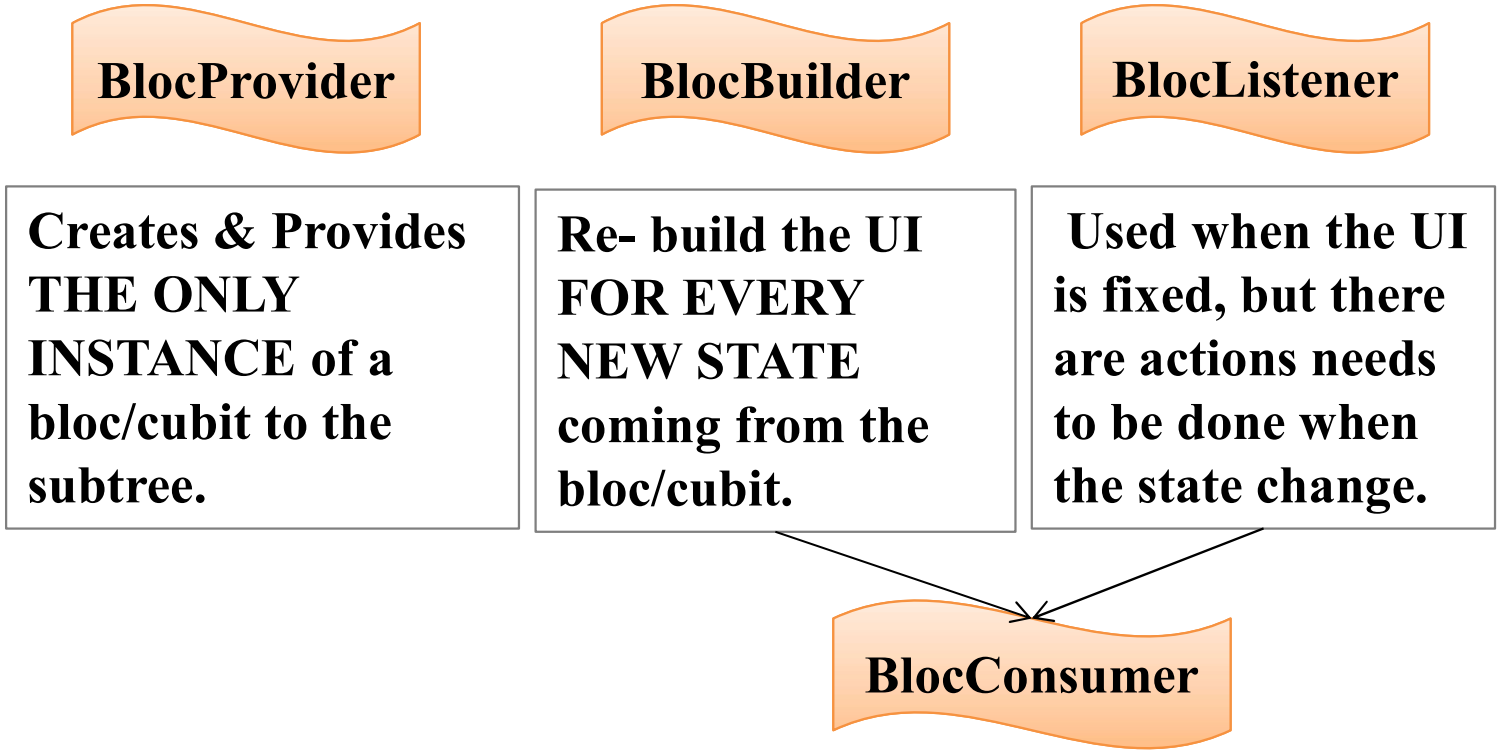
MultiBlocListener

```
MultiBlocProvider{
providers:[
BlocProvider <BlocA>
(cretae: (BuildContext
context ) => BlocA( ),),

BlocProvider <BlocB>
(cretae: (BuildContext
context ) => BlocB ( ),),
……
            ]
child: …
}
```

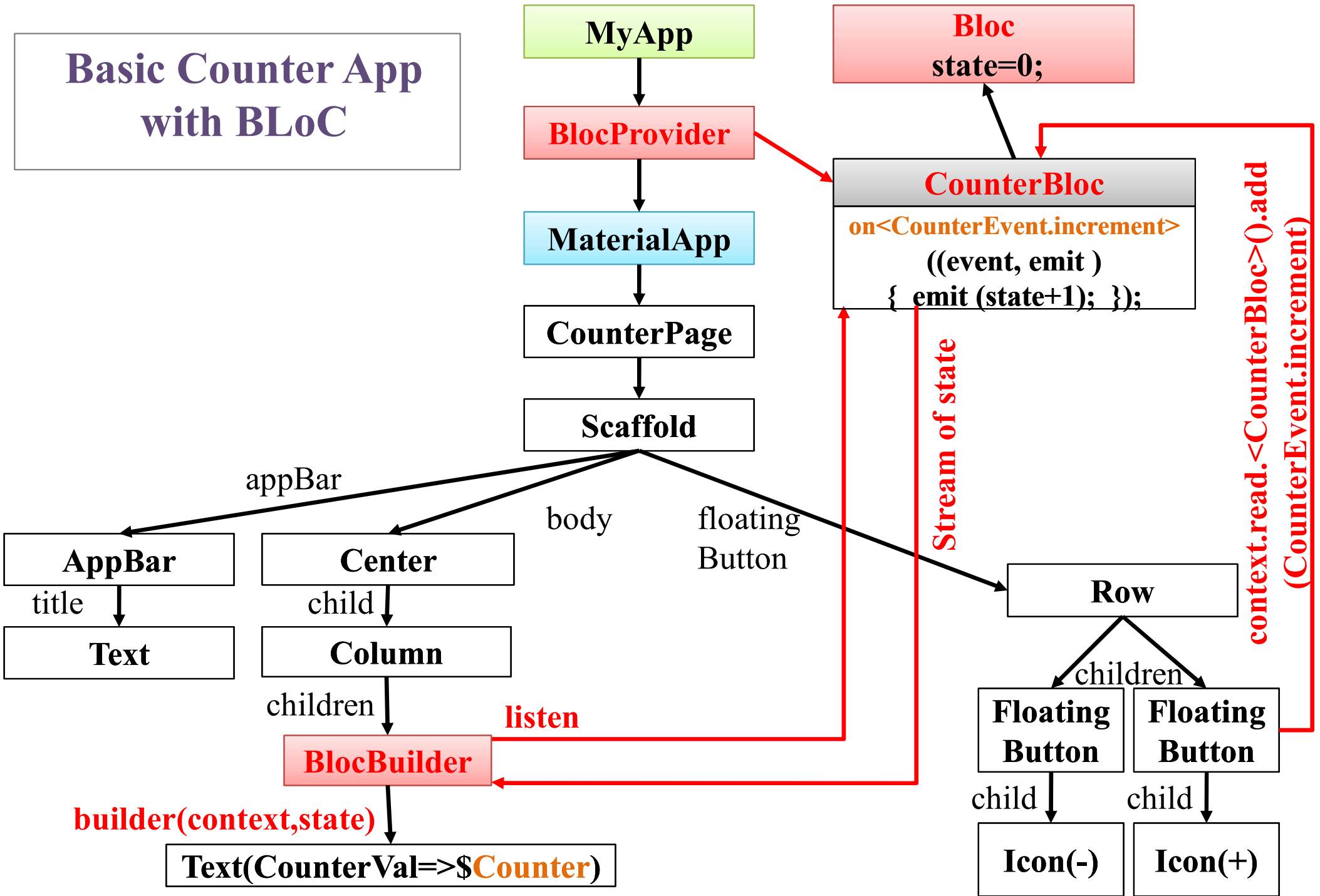
```
MultiBlocListener{
listeners:[
BlocListener <BlocA,
BlocAState>
(listener: (context, state){ }, ),

BlocListener <BlocB,
BlocBState>
(listener: (context, state){ }, ),
……
        ]
child: …
}
```

State Management approaches: BLoC

- **The use of BLoC : summary.**



State Management approaches: BLoC



State Management approaches: BLoC

- **Illustrative example :**

Implementing the basic ToDo app with BLoC

State Management approaches: BLoC

- **Implementation steps:**

1) Adding the "***flutter_bloc"*** package dependency.
2) Select the feature to implement with bloc.
3) Answer the key questions (determine ***events*** and ***states***).
4) State (bloc) implementation.
5) Expose the bloc (the state) by using ***BlocProvider***, ...
6) Link the bloc to the UI =>
   a) use the bloc (the state) by using ***BlocConsumer, BlocBuilder, BlocListener ...***
   b) Access to the bloc by triggering events.

State Management approaches: BLoC

- **Implementation steps:** **1)** adding the flutter_bloc package.

➢ Run the command : *flutter pub add flutter_bloc*

➢ Or, add the line : *dependencies: flutter_bloc : ^8.1.2* to the *pubspec.yaml* file.

➢ Run: *flutter pub get* if needed.

➢ Import the package from the App:

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

State Management approaches: BLoC

- **Implementation steps: 2)** select the feature.

➢ **General feature** : Manage the **list of todos** (tasks).

Or consider a **particular feature** like :

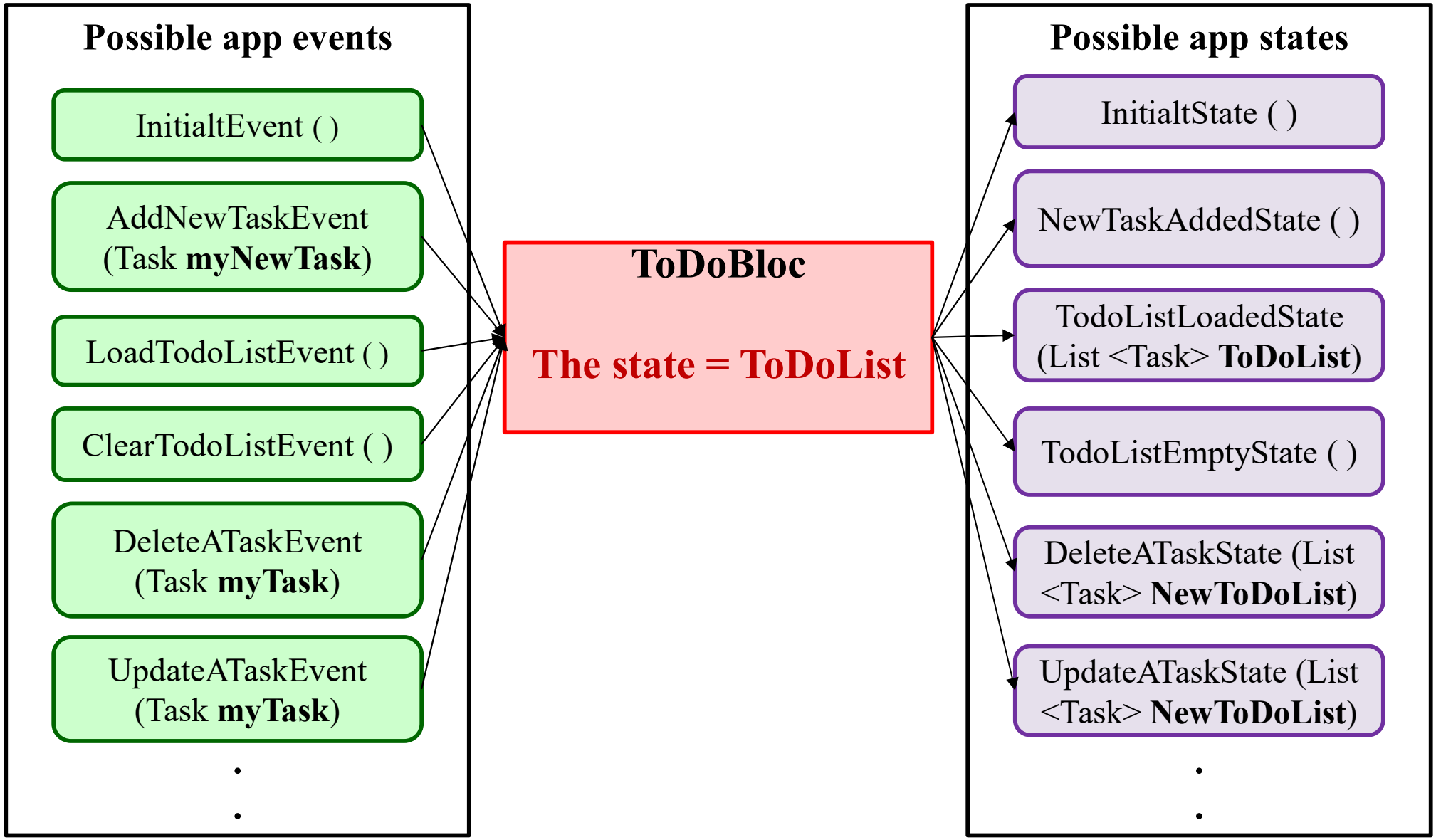
"add a new task to the list".

➢ For each feature do the following steps

→ → →

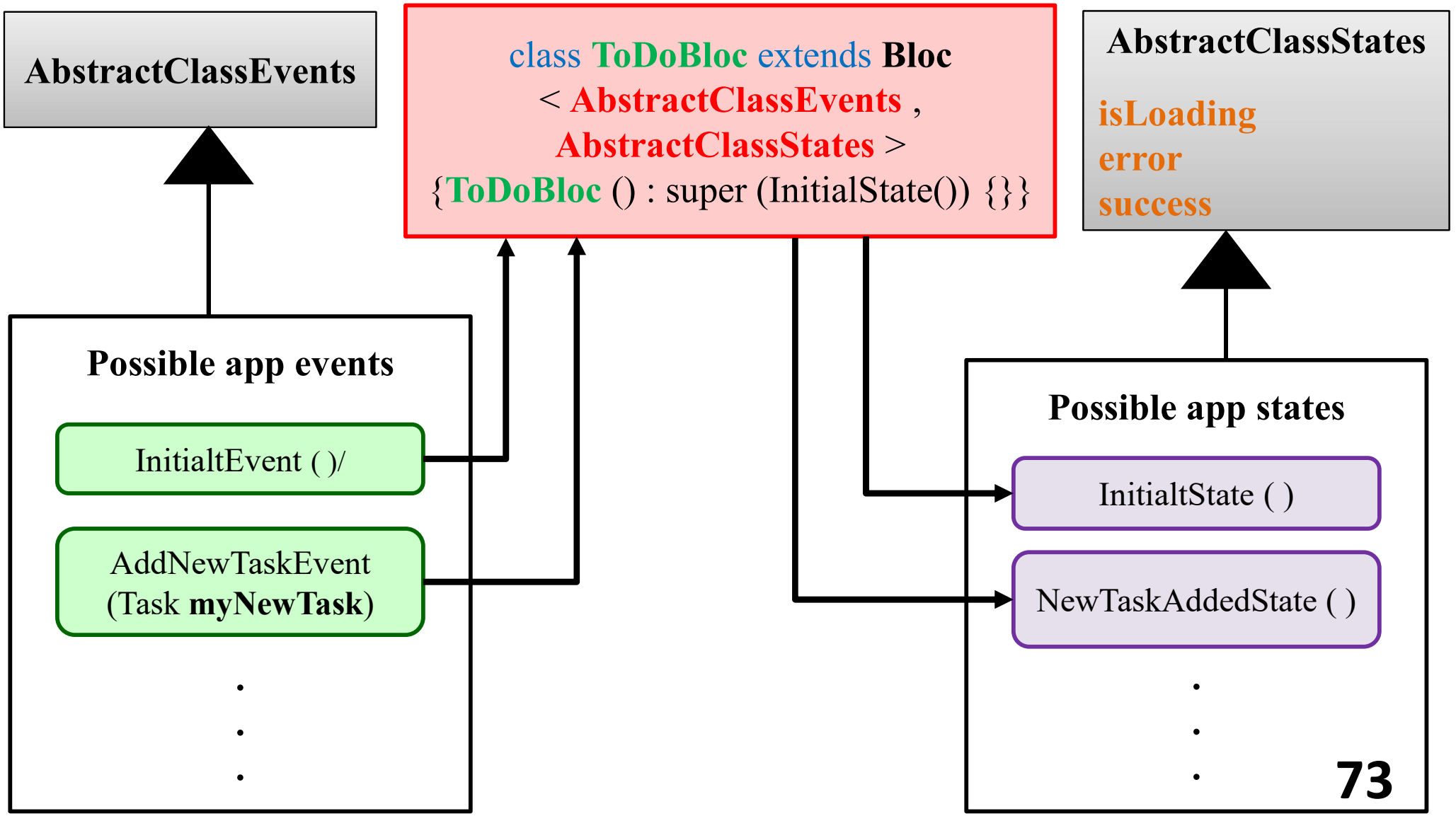
State Management approaches: BLoC

- **Implementation steps: 3)** determine ***events*** and ***states***.



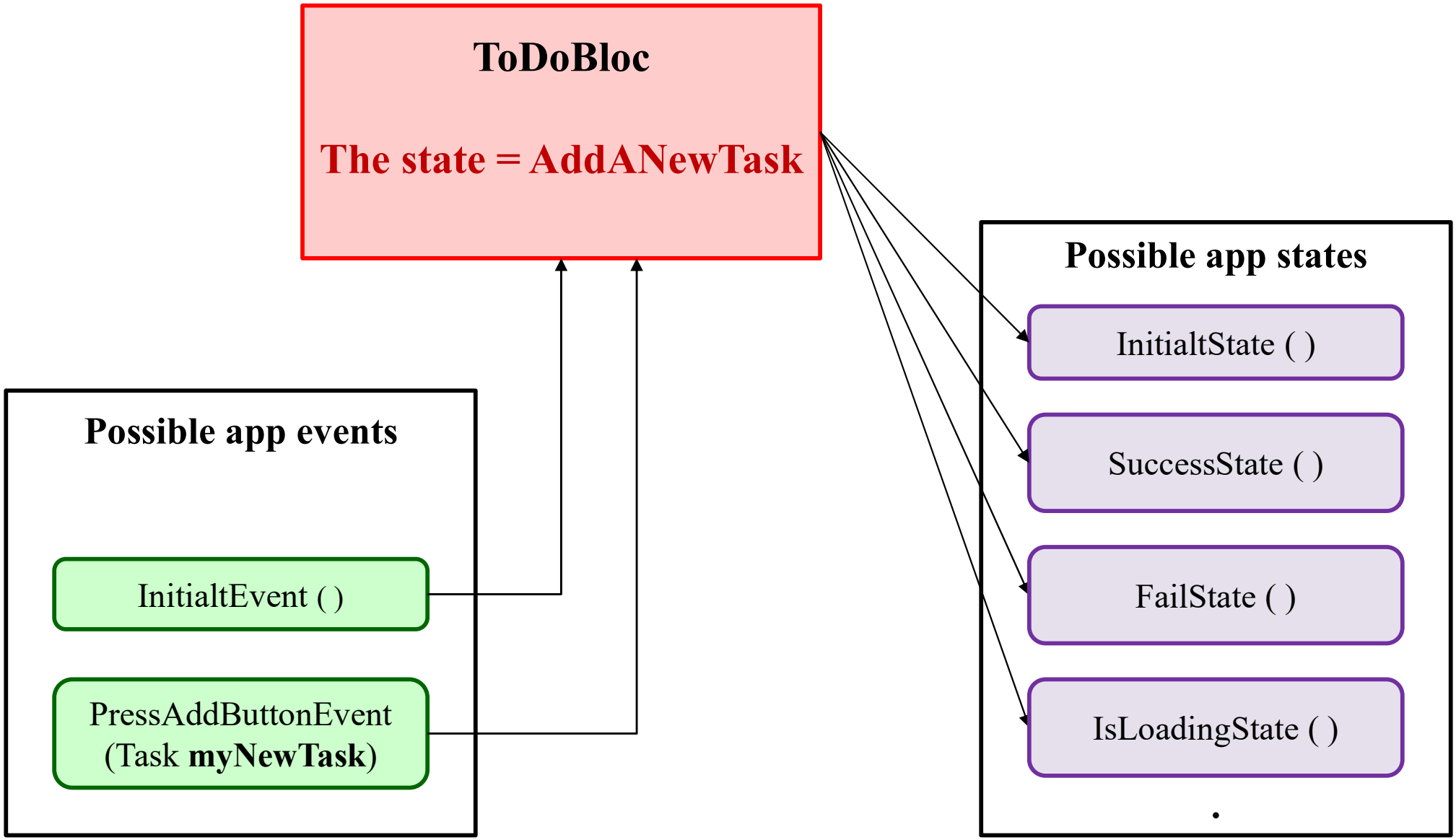
State Management approaches: BLoC

- **Implementation steps: 3)** determine ***events*** and ***states***.



State Management approaches: BLoC

- **Implementation steps: 3)** determine ***events*** and ***states***.



State Management approaches: BLoC

- **Implementation steps: 4) *bloc*** implementation.

ToDoBloc --- The state = ToDoList ---

```
on< InitialtEvent > ((event,emit){ emit (InitialtState()) });
**********************************************************************************************
on< AddNewTaskEvent > ((event,emit){
        AddTaskMethod (ToDoList, event.myNewTask); // processing the add operation
        emit (NewTaskAddedState()) });
**********************************************************************************************
on< LoadTodoListEvent > ((event,emit){
        ToDoList = LoadToDoListMethod ( ); // processing the load operation
        emit (TodoListLoadedState(ToDoList)) });
**********************************************************************************************
on< ClearTodoListEvent > ((event,emit){
        ToDoList = [ ]; // processing the clear operation
        emit (TodoListEmptyState( )) });
**********************************************************************************************
on< DeleteATaskEvent > ((event,emit){
        DeleteTaskMethod (ToDoList, event.myTask); // processing the delete operation
        emit (DeleteATaskState( )) });
```

State Management approaches: BLoC

- **Implementation steps: 4) *bloc*** implementation.

ToDoBloc --- The state = AddANewTask ---

```
on< LoadTodoListEvent > ((event,emit){

**********************************************************************************
 emit (TodoListLoadedState([ ], isLoading: true)) });
**********************************************************************************

Try{        // processing the load operation
            ToDoList = LoadToDoListMethod ( );
 emit (TodoListLoadedState(ToDoList , isLoading: false, success: true)) });
} **********************************************************************************

catch (e) {
 emit (TodoListLoadedState([ ], isLoading: false, error: e.toString() )) });
**********************************************************************************
}
```

State Management approaches: BLoC

- **Implementation steps: 5)** expose the ***bloc*** with ***BlocProvider***.

```
BlocProvider< ToDoBloc > (

        create: (BuildContext context ) => ToDoBloc ( ) . . add (InitialEvent ( )) ,
        child: MaterialApp(..... ),

);
```

```
MultiBlocProvider (
providers: [  BlocProvider< ToDoBloc > (
                 create: (BuildContext context ) => ToDoBloc ( ) . . add (InitialEvent ( )) ,
               ],
child: MaterialApp(..... ),
 );
```

State Management approaches: BLoC

- **Implementation steps: 6.a)** use the bloc (***BlocConsumer, BlocBuilder, BlocListener***).

```
BlocBuilder <ToDoBloc, AbstractClassStates > (
    builder: (context, state){

        if (state.isLoading){return Text("the state is loading");}
        else if (state.success) {return ListView.builder
                        (itemCount: state.ToDoList.length, itemBuilder: ... ); }
        else if (state.ToDoList.isEmpty) {return Text("No tasks found ");}
        } // builder

), // BlocBuilder
```

State Management approaches: BLoC

- **Implementation steps: 6.b)** access to the bloc by triggering events.

```
IconButton (
        icon: Icon(Icons.add),
        onPressed: (){
              context.read<ToDoBloc >().add(AddNewTaskEvent (Task myNewTask));
            }
);
```

```
IconButton (
        icon: Icon(Icons.add),
        onPressed: (){
              BlocProvider.of<ToDoBloc >(context).add(AddNewTaskEvent (Task
                                    myNewTask));
            }
);
```

Conclusion

- **BLoC** design pattern offers a solution to separate the business logic from the user interface.

- It is based on the concept of **streams** to transmit data asynchronously between the business logic and the user interface.

- Answering key questions such as : *the initial state*, *possible interactions*, and *functions available in Cubit,* … => structure the app in a **modular** way => reuse code and easier to maintain and scale the code.

Conclusion

- **BlocProvider** is the widget used to inject *BLoC* instances into widget tree.
- **BlocBuilder**, **BlocListener** & **BlocConsumer** are the widgets used to communicate with BLoC from the user interface to react to state changes and update the user interface in an efficient manner.
- **Cubit** is a minimal version of bloc that offers a simpler approach to certain features.

Conclusion

- **Go further with other state management solutions :**

  - **Riverpod,**
  - **GetX,**
  - **StateX,**
  - **….**