INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

Bachelor Thesis

IU University of Applied Sciences

Study program: Computer Science

State management in Flutter applications: a comparative study

Sergei Ulvis

Enrolment number: 32113004

Vidnoye, Moscow oblast, Russia

Date of submission: 26 March 2025

Abstract

Flutter, a popular cross-platform framework developed by Google, enables developers to build high-performance applications for Android, iOS, macOS, Windows, Linux, and the web using a single codebase, significantly reducing development time and costs. Its declarative UI paradigm relies heavily on widgets which render based on the application’s state, making state management a pivotal aspect of Flutter development. Effective state management influences an application’s architecture, scalability, and performance, yet Flutter’s flexibility in offering multiple approaches - without mandating a specific solution - presents both opportunities and challenges for developers.

This thesis targets Flutter developers seeking evidence-based guidance on selecting state management solutions, with a primary focus on mobile applications, though findings may extend to web and desktop contexts. It systematically compares six prominent state management approaches - Provider, Riverpod, BLoC, GetX, Redux, and MobX - using ISO/IEC 25010 quality criteria, such as performance efficiency and maintainability. To facilitate this comparison, a prototype application simulating an e-commerce scenario is implemented for each approach. Empirical evaluation leverages automated testing and static code analysis to assess key metrics, providing quantitative and qualitative insights into each solution’s strengths and trade-offs.

The results aim to equip developers with practical recommendations for choosing state management strategies tailored to their project needs, enhancing decision-making and fostering scalable, maintainable Flutter applications. This study contributes to the Flutter community by offering a data-driven perspective on a critical yet often debated aspect of cross-platform development.

Keywords: Flutter, state management, cross-platform development, software quality.

Table of contents

1. Introduction

Two mobile operating systems - iOS and Android – are the dominating mobile platforms last years (StatCounter Global Stats, n.d.). This makes it necessary to develop a program for both platforms when creating mobile applications. Since developing two similar applications leads to significantly greater development effort just by technical reasons (need to write platform-specific code) - cross-platform technologies are widely used in development for last several years. Main target of such technologies is to cover multiple platforms with the development of a single code base being compiled or somehow transferred to different target platforms.

Last years Flutter has become one of the leading frameworks for developing cross-platform applications. Created by Google, it allows developers to build high-performance, natively compiled apps for mobile, web, and desktop using a single codebase (Google LLC, n.d., *Supported platforms*). With its reactive framework, extensive widget library, and hot-reload feature, Flutter has gained popularity among developers looking to create visually engaging and responsive applications. However, as apps become larger and more complex, managing state - the data that controls the app’s behavior and appearance - becomes a significant challenge.

State management plays a crucial role in application development, as it directly influences an app’s scalability, maintainability, and overall performance. In Flutter, the framework itself does not impose a specific state management solution, giving developers the freedom to select an approach that best fits their application's needs (Google LLC, n.d., *List of state management approaches*). This flexibility, however, also introduces complexity, as different state management solutions vary in their structure, learning curve, and trade-offs.

Flutter developers can choose from a wide range of state management approaches, each offering different ways to handle state efficiently. Some solutions, like Provider and Riverpod, are lightweight and easy to implement, making them suitable for smaller applications or simpler use cases. On the other hand, more structured, pattern-driven approaches like BLoC (Business Logic Component) and Redux offer a well-defined architecture but require a steeper learning curve and additional boilerplate code (Google LLC, n.d., *List of state management approaches*).

Each state management approach follows its own design principles and comes with advantages and limitations, making the selection process a non-trivial decision. Choosing the right solution depends on several factors, including the size and complexity of the application, developer experience, performance considerations, and maintainability requirements. Therefore, understanding the strengths and weaknesses of these state management methods is essential for making informed architectural decisions in Flutter development.

The wide range of state management options in Flutter has created a fragmented ecosystem, making it difficult for developers to choose the right solution for their specific needs. The decision depends on various factors, such as application size, state complexity, team familiarity with a library, and the need for scalability and maintainability. However, since Flutter does not enforce a standard approach to state management, there is ongoing debate within the developer community about which solution is "best," often without clear guidance or solid data to support these claims.

This challenge is further complicated by the rapid growth of the Flutter ecosystem. New state management libraries and updates to existing ones are frequently introduced, each claiming to offer better performance, ease of use, or additional features. For example, Riverpod was developed as an improved version of Provider, addressing some of its limitations, while BLoC continues to be favored for its strict separation of business logic and UI (Google LLC, n.d., *List of state management approaches*). In this constantly evolving landscape, developers often rely on community feedback, trial and error, or personal experience to decide which state management approach best fits their application.

1.1 Problem definition

The motivation for this study comes from the need to help developers choose the right state management approach in Flutter based on real data rather than guesswork. With so many options available – about 20 packages listed (Google LLC, n.d., *List of state management approaches*) as possible state management in Flutter official documentation - it can be difficult to understand which one is best for a given project. This research compares most popular libraries by analyzing their strengths and weaknesses using key software quality factors from the ISO 25010 standard. These factors include performance efficiency, interaction capability, maintainability, flexibility (ISO/IEC, n.d.), as well as documentation volume and quality.

The main goal of this study is to provide practical insights that help developers make informed decisions about state management. By highlighting the trade-offs of different approaches, this research aims to reduce confusion and improve efficiency in the selection process. Additionally, the findings will contribute to the Flutter community, encouraging more data-driven discussions about state management and helping developers build applications that are scalable, maintainable, and high-performing.

In short, this study is driven by the increasing complexity of state management in Flutter and the need for a systematic, side-by-side comparison of available solutions. By addressing these challenges, the research aims to enhance developer experience and improve the overall quality of Flutter applications.

1.2 Objectives and research questions

The primary aim of this study is to address the challenges developers face when selecting state management solutions for Flutter applications. By conducting a comparative analysis of popular state management libraries, this research seeks to provide actionable insights and recommendations that can guide developers in making informed decisions. To achieve this aim, the study is guided by a set of defined objectives and research questions, which are outlined below.

The objectives of this research are as follows:

- identify and describe state management solutions: to provide a comprehensive overview of the most widely used state management libraries in Flutter, including their architecture, features, and use cases. This will serve as a foundation for understanding the strengths and weaknesses of each solution.
- develop prototype applications with consistent functionality using different state management solutions. This will enable a fair and objective comparison of the libraries in a real-world context.
- evaluate and compare solutions based on key metrics derived from the ISO 25010 standard. This will provide empirical evidence to support the selection process.
- propose best practices and recommendations to synthesize the findings from the evaluation and provide actionable recommendations for developers. These recommendations will help developers choose the most suitable state management solution based on their application's requirements, team expertise, and long-term goals.

To achieve the above objectives, the study addresses the following research questions:

- What are the most widely used state management solutions in Flutter, and what are their key architectural paradigms, features, and use cases? This question aims to provide a detailed understanding of the state management landscape in Flutter, including the strengths and weaknesses of most popular solutions.
- How do different state management solutions perform in terms of performance efficiency, interaction capability, maintainability and flexibility, as metrics defined by the ISO 25010 standard? This question focuses on evaluating the state management libraries based on standardized software quality metrics, ensuring a rigorous and objective comparison.
- What are the best practices for implementing and maintaining state management in Flutter applications, and how can developers avoid common pitfalls? This question focuses on providing actionable recommendations to help developers implement state management effectively and sustainably.

1.3 Structure

This thesis is organized into several key chapters to systematically address the defined objectives and research questions:

- Introduction: this chapter provides an overview of the research, outlining the significance of state management in Flutter applications, the motivation for the study driven by the complexity and variety of available solutions, and the specific objectives and research questions guiding the work. It sets the stage for the subsequent analysis and evaluation.
- Literature review: this section establishes the theoretical foundation by reviewing existing literature and concepts critical to the study. It covers Flutter's architecture, widget system, automated testing, and a detailed exploration of state management approaches, including built-in methods and popular third-party solutions (Provider, Riverpod, BLoC, GetX, Redux, and MobX). It concludes with comparative insights drawn from the literature.
- Research methodology and implementation: this chapter outlines the approach taken to evaluate state management solutions. It details the literature review process, the development of a prototype application for each approach, the experimental setup, and the evaluation criteria based on ISO/IEC 25010 standards. It also documents the implementation process, including requirements analysis, prototype design and testing procedures.
- Research findings: his section presents the results of the comparative evaluation, followed by a compilation of findings to highlight key performance and maintainability insights. Due to thesis`s main text length restriction, detailed assessments of each state management approach (Provider, BLoC, Riverpod, GetX, Redux, and MobX) based on the defined criteria is moved to Appendix, in "Evaluations details" section.
- Conclusion: the final chapter consolidates the research outcomes, offering an overall assessment of the evaluated state management approaches. It provides practical recommendations for Flutter developers based on the findings, tailored to factors like application complexity and team expertise, and identifies potential areas for future research to further enhance state management practices.
- References: a comprehensive list of all sources cited throughout the thesis, adhering to academic standards, ensuring traceability and credibility of the research.
- Appendix: this section contains supplementary materials, including detailed evaluation results for each state management approach, code listings, the Maintainability Index calculation process, Flutter-specific technical details, and an analysis of ISO/IEC 25010 quality characteristics applied to this study. It concludes with the Declaration of Authenticity.

This structure ensures logical progression from problem identification to solution evaluation, finalizing with actionable insights for the Flutter development community.

1.4 Methodology

This study employs a mixed-method approach to evaluate state management solutions in Flutter applications, combining a comprehensive literature review with empirical analysis through prototype development and testing. The methodology is designed to address the research objectives and questions systematically, ensuring robust and reproducible findings.

The process begins with an extensive literature review to analyze and synthesize existing state management strategies in Flutter. This review identifies key characteristics, strengths, and limitations of both built-in methods and third-party libraries (Provider, Riverpod, BLoC, GetX, Redux, and MobX). From this foundation, evaluation criteria are derived, based on selected ISO/IEC 25010 software quality characteristics, tailored to the context of state management.

In the implementation phase, a prototype application is developed for each of the six selected state management approaches. Each prototype implements identical UI and functionality - representing a simple e-commerce scenario with user authentication, product listing, and cart management - to ensure a fair comparison. Requirements for the prototypes are established through a combination of implicit needs (e.g., efficient state updates) and explicit specifications (e.g., feature consistency across implementations), derived from the literature review and practical use cases.

Evaluation is conducted using a mix of qualitative and quantitative methods. Qualitative assessments focus on criteria such as learnability, modifiability, testability and documentation, judged through expert analysis of code structure and documentation. Quantitative measures include performance metrics (e.g., widget rebuild counts via automated tests) and modularity (e.g., Maintainability Index calculated using the dcm tool). An experimental setup is documented, specifying hardware (e.g., consistent development machine) and software (e.g., Flutter version, developing environment information) configurations to ensure reproducibility.

Automated testing plays important role, with widget tests assessing UI responsiveness for each implementation. Performance benchmarks are executed using script (e.g., flutter test test/efficiency_benchmark_test.dart) to measure time behavior, while maintainability is quantified through static code analysis. All findings are recorded and analyzed to provide empirical evidence supporting the comparison of state management approaches.

2. Literature review

This chapter addresses the foundations for the subsequent evaluation. It includes an analysis of the relevant literature on the topic as well as an explanation of concepts through examples.

2.1 Literature search strategy

Following vom Brocke et al. (2009), this study adopts a rigorous, transparent literature search process to reconstruct the knowledge base on Flutter state management, ensuring replicability and validity. The search process aligns with their framework (Figure 3), emphasizing scope definition, topic conceptualization, and a structured search, tailored to address the research objectives (Section 1.2): to identify widely used state management solutions, evaluate their quality and propose usage guidence.

- Phase I: scope definition
  Based on Cooper's taxonomy (1988, cited in vom Brocke et al., 2009, Figure 4), the review's scope is defined as:
  - Focus: research outcomes (features of state management solutions) and methods (evaluation via ISO/IEC 25010).
  - Goal: synthesize existing knowledge and integrate findings to support prototypes development and analysis.
  - Organization: conceptual, grouping sources by Flutter architecture, state management paradigms, and quality metrics.
  - Perspective: neutral, presenting an objective comparison of solutions.
  - Audience: Flutter developers (especially in early steps of professional development).
  - Coverage: representative, targeting widely adopted solutions (e.g., >50k downloads on pub.dev) and seminal works on Flutter and software quality, rather than an exhaustive review, given the field's rapid evolution since Flutter's 1.0 release in 2018.

- Phase II: topic conceptualization

Key concepts were identified to guide the search: "Flutter", "state management", "cross-platform development", "software quality", and "ISO/IEC 25010". Synonyms (e.g., "reactive programming", "architecture patterns") and specific solutions (Provider, BLoC, Riverpod, GetX, Redux, MobX) were mapped using initial consultations of Flutter's official documentation (flutter.dev) and seminal texts (e.g., Flutter source code comments, pub.dev package descriptions), per Baker (2000, cited in vom Brocke et al., 2009).

- Phase III: literature search process
  A systematic search was conducted, adapting vom Brocke et al.'s sub-phases (Figure 5):
  - Sub-phase 1: source identification: focused on high-quality IS and technical sources, including:

        - academic databases: Google Scholar, IEEE Xplore, ACM Digital Library for peer-reviewed articles on software engineering and mobile frameworks (accessing via IU Library interface and EBSCO service).
        - technical repositories: pub.dev for package metadata (e.g., description, downloads, versioning), GitHub for community insights (e.g., issues, discussions).
        - official documentation: flutter.dev for Flutter’s architecture and built-in state management.
    - Sub-phase 2: database selection: these sources were chosen to balance scholarly rigor (IU library, IEEE, ACM) with practitioner relevance (pub.dev, GitHub), reflecting Flutter’s dual academic and industry context.
    - Sub-phase 3: keyword search: queries combined terms with Boolean operators:
        - "Flutter" AND "state management" (broad scope).
        - "Flutter" AND ("Provider" OR "BLoC" OR "Riverpod" OR "GetX" OR "Redux" OR "MobX") (specific solutions).
        - "software quality" AND "ISO 25010" AND ("mobile" OR "Flutter") (quality evaluation).
        - variations: "reactive programming" AND "Flutter", “Flutter” AND "state management" AND "performance".
    - Sub-phase 4: backward and forward search:
        - Backward: reviewed references in similar works (e.g. Slepnev`s “State management approaches in Flutter”, Hoang`s “State Management Analyses of the Flutter Application”).
        - Forward: forward searches were not conducted due to time constraints, focusing instead on primary keyword results.
    - Evaluation: sources were assessed for relevance via abstracts (academic) or metadata (technical), retaining:
        - inclusion criteria: published1995 – February 2025, English language, addressing Flutter state management (and describing packages with >50k downloads, as per Section 2.4.3) or software quality.
        - exclusion criteria: non-English, unrelated to Flutter, or packages with <50k downloads or packages marked “Dart 3 incompatible” as outdated.
    - Outcome: approximately 50 sources were identified, refined to about 30 core references (listed in References) based on authority (e.g., peer-reviewed, official docs) and relevance to research questions.

This process ensures representative coverage of Flutter state management literature, balancing depth and practicality, as recommended by vom Brocke et al. (2009). Detailed documentation -

databases, keywords, and criteria - enables readers to replicate or extend the search, addressing reliability and validity concerns.

2.2 Flutter overview

In mobile app development, it is common practice to create two separate applications for the two dominant operating systems, iOS and Android. An analysis by Appfigures reveals that in the Apple App Store 78% of the analyzed apps are based on Swift, which is used for native iOS development, while in the Google Play Store, 77% of the analyzed apps are based on Kotlin, the counterpart to Swift for Android, with Flutter being on second positions in both stores, with 13% in AppStore and 19% in Google Play (Appfigures, n.d.).

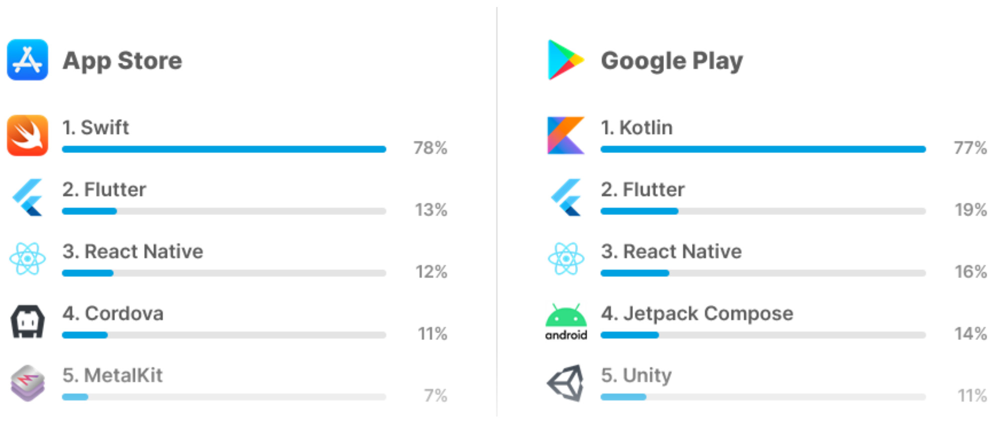


Figure 1. The most popular development SDKs (Appfigures, n.d.)

This highlights the growing popularity of Flutter as a viable alternative to native development, enabling developers to build apps for several platforms using a single codebase.

During development, Flutter apps run in a VM that offers stateful hot reload of changes without needing a full recompile. For release, Flutter apps are compiled directly to machine code, whether Intel x64 or ARM instructions, or to JavaScript if targeting the web. The framework is open source, with a permissive BSD license, and has a thriving ecosystem of third-party packages that supplement the core library functionality.

Flutter is designed as an extensible, layered system. It exists as a series of independent libraries that each depend on the underlying layer. No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable (Google LLC, n.d., *Flutter architectural overview*).

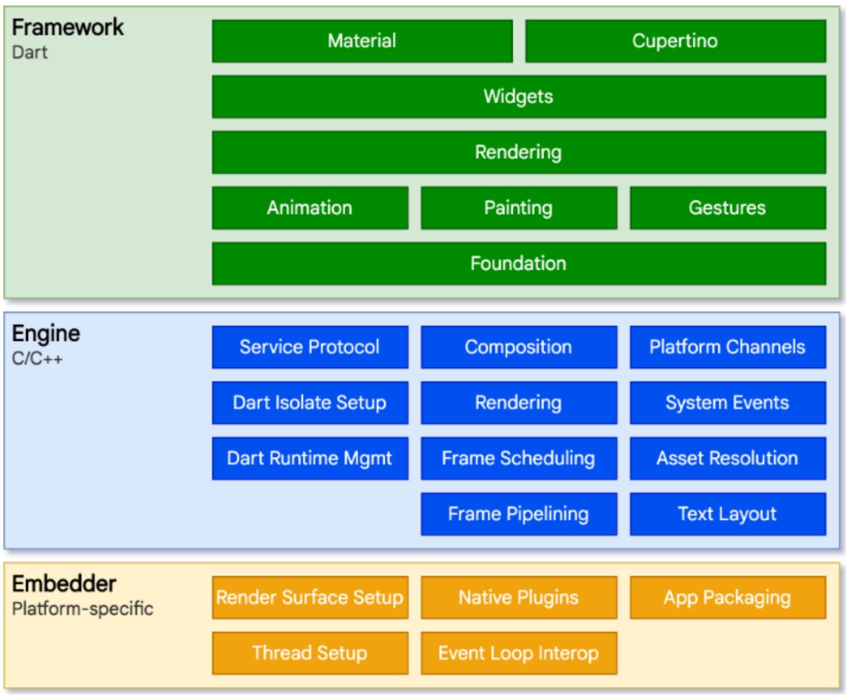


Figure 2. Architectural layers (Google LLC, n.d., *Flutter architectural overview*)

Regarding the underlying operating system, Flutter applications are packaged in the same way as any other native application. A platform-specific Embedder layer provides an entry point; coordinates with the underlying operating system for access to services like rendering surfaces, accessibility, and input; and manages the message event loop. The embedder is written in a language that is appropriate for the platform: currently Java and C++ for Android, Swift and Objective-C/Objective-C++ for iOS and macOS, and C++ for Windows and Linux. Using the embedder, Flutter code can be integrated into an existing application as a module, or the code might be the entire content of the application. Flutter includes several embedders for common target platforms, but other embedders also exist.

At the core of Flutter is the Flutter engine, which is mostly written in C++ and supports the primitives necessary to support all Flutter applications. The engine is responsible for rasterizing composited scenes whenever a new frame needs to be painted. It provides the low-level implementation of Flutter's core API, including graphics (through Impeller on iOS, Android, and desktop (behind a flag), and Skia on other platforms), text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain (Google LLC, n.d., *Flutter architectural overview*).

The Framework layer, at the top, builds on the two lower layers and provides the interfaces needed for app development. It also includes an extensive catalog of standard widgets, which are inspired by the design of both iOS and Android. These widgets are categorized into the Cupertino catalog for iOS-like designs and the Material catalog for Material Design is often used on Android.

The concept of widgets, which has been mentioned several times so far, will be explored in more detail in the following section.

2.3 Widgets in Flutter

One of the most important components of the Flutter framework is the concept of widgets. The phrase "Everything is a widget" (Payne, 2024, p. 27) is often used in the literature to emphasize that Flutter utilizes the widget concept for a wide range of use cases. A widget can take on various tasks, such as rendering a UI component, handling animations, or arranging other widgets.

To render the user interface, Flutter uses compositions of multiple widgets. For example, as shown in Listing 1, a Column widget can be used to display multiple widgets in column view, row by row.

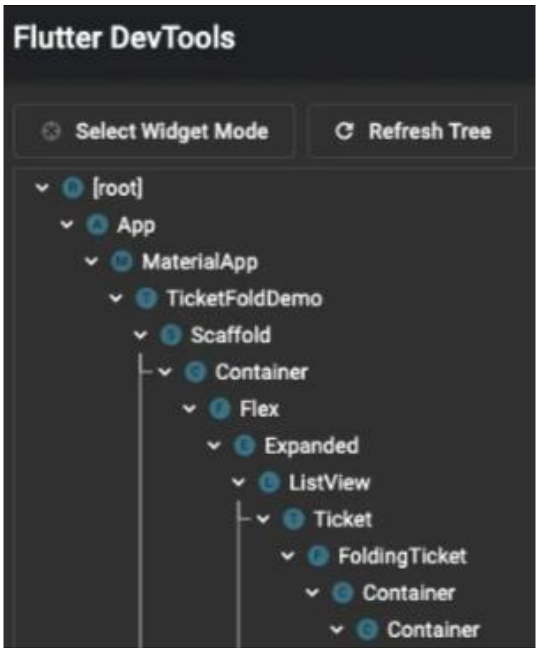
A significant difference compared to traditional declarative UI frameworks is that widgets only contain the instructions for building a user interface, but not the current state of the widget. In detail, this means that a widget's build() method should not have any side effects and therefore is expected to execute quickly. Conceptually, a widget can be seen as a function that receives the current state and generates the corresponding user interface. A widget receives the data to be displayed and generates the corresponding user interface (Google LLC, n.d., *Introduction to widgets*).

In Flutter, widgets are fundamentally distinguished between StatelessWidget and StatefulWidget.

A *StatelessWidget* fundamentally does not have any mutable state. This means that all class variables should be immutable. In Dart, this is indicated using the final modifier. It is implied that all information about the widget's state must come from the values passed into the constructor. Thus, the widget is only rebuilt when changes occur to the constructor's values due to modifications in the widget hierarchy above it (Google LLC, n.d., *Introduction to widgets*).

A *StatefulWidget*, on the other hand, consists of two classes: one for the StatefulWidget itself and another for its associated State. These widgets combine the capabilities of a StatelessWidget with the ability to change their own state. This is particularly relevant, for example, in the case of a counter that needs to be incremented. State changes are performed using the setState method, as shown in Listing 2, so that the change is communicated to the framework. Internally, this marks the widget as "dirty," causing the method of building the State to be called again by the framework. It is crucial that the State remains persistent, even if, for example, the constructor parameters of the StatefulWidget change (Google LLC, n.d., *Introduction to widgets*).

By combining various widgets, a widget tree is formed, as shown in Figure 3. This tree can also be traversed using auxiliary tools, but it is fundamentally designed for unidirectional data flow. This means that widgets should only be able to change the state of their child widgets. Consequently, parent widgets cannot be altered - at least not directly.



```
[root]
  App
    MaterialApp
      TicketFoldDemo
        Scaffold
          Container
            Flex
              Expanded
                ListView
                  Ticket
                    FoldingTicket
                      Container
                        Container
```

FIGURE 3. FLUTTER WIDGET TREE

Unlike native development, the user interface here is programmed declaratively. During development, it is defined how each element should appear in each state without needing to describe the control flow.

As conclusion, the state and the management of widget state are crucial parts of a Flutter application's lifecycle. This indicates early on that a solution must be found which can be applied to various scenarios.

2.4 Automated testing

To automatically test Flutter applications, Flutter employs three test categories: unit testing, widget testing, and integration testing (Google LLC, n.d., *Testing Flutter apps)*.

A unit test tests a single function, method, or class, unit tests are used for the automated testing of individual functions or classes. These tests are constructed as lambda functions. Using the provided tools, values can be verified to see if they match the expected outcome (Google LLC, n.d., *Introduction to unit testing*).

A widget test (in other UI frameworks referred to as component test) tests a single widget. Widget tests assess one or more widgets to determine if they show the desired behavior. These tests can be executed within Flutter without the need for an iOS or Android simulator. For this purpose, various tools exist in the flutter_test library that allow for the creation of widgets or the verification of specific widget properties (Google LLC. (n.d.). *Introduction to widget testing*).

An integration test tests a complete app or a large part of an app. Integration tests examine the application as a whole to ensure that the individual components function correctly together. However, this type of test is not relevant for the evaluation and will therefore not be discussed in further detail (Google LLC, n.d., *Testing Flutter apps)*.

2.5 State management in Flutter

Flutter's layered architecture and platform-agnostic design make it a powerful framework for building cross-platform applications. However, as developers begin to create more complex apps, they quickly encounter the challenge of managing the application's state - the data that drives the UI and determines how the app behaves. While Flutter's engine handles rendering, animations, and platform integration, it is up to the developer to decide how to structure and manage the app's state effectively.

The declarative nature of Flutter's UI framework means that the UI is a direct reflection of the app's state. When the state changes, the framework rebuilds the UI to match the new state. This approach simplifies UI development but also places a significant responsibility on developers to ensure that state changes are handled efficiently and consistently. Poor state management can lead to performance bottlenecks, unnecessary widget rebuilds, and a codebase that is difficult to maintain.

At its core, state management in Flutter is about organizing and synchronizing the data that drives the app's behavior and appearance. This includes everything from simple UI state, like the text in a text field, to complex app state, like user authentication or data fetched from an API. Understanding how to manage these different types of state is crucial for building scalable and maintainable Flutter applications.

State management is a critical aspect of developing Flutter applications, as it determines how data flows through the app and how the user interface (UI) reacts to changes in that data. In Flutter, the UI is built using a declarative approach, meaning that the UI is a function of the application's state. When the state changes, the framework rebuilds the UI to reflect the new state. This reactive paradigm makes state management a central concern for developers, especially as applications grow in size and complexity.

In the broadest possible sense, the state of an app is everything that exists in memory when the app is running. This includes the app's assets, all the variables that the Flutter framework keeps about the UI, animation state, textures, fonts, and so on. While this broadest possible definition of state is valid, it's not very useful for architecting an app (Google LLC, n.d., *Differentiate between ephemeral state and app state*).

Developers don't manage some states (like textures). The framework handles those for developers. So, a more useful definition of state is "whatever data developers need in order to rebuild UI at any moment in time". The other kind of state that developer do manages - can be separated into two conceptual types: ephemeral state and app state (Google LLC, n.d., *Differentiate between ephemeral state and app state*).

- *Ephemeral state*: short-lived state that is local to a single widget or screen, such as the current page in a PageView or the text in a TextField. This type of state is typically managed using Flutter's built-in State and setState mechanism.
- *App state:* persistent state that needs to be shared across multiple widgets or screens, such as user authentication status, app theme, or data fetched from an API. Managing app state requires more sophisticated solutions, as it involves sharing and synchronizing data across the app.

Effective state management is crucial for building scalable, maintainable, and performant Flutter applications. Poor state management can lead to:

- Spaghetti code: unorganized and tightly coupled code that is difficult to understand and maintain.
- Performance issues: unnecessary widget rebuilds, leading to sluggish app performance.
- Bugs and inconsistencies: inconsistent UI behavior due to improper handling of state changes.

2.5.1 Built-in(“vanilla”) methods and widgets

Unlike some frameworks that enforce a specific state management paradigm, Flutter offers a flexible environment where developers can choose from a range of approaches, depending on their application’s requirements. Flutter provides several built-in techniques for state management, often referred to as “vanilla” methods, which do not require any external libraries. These methods form the foundation upon which more advanced solutions are built.

setState in StatefulWidgets: the simplest and most direct way to manage state in Flutter is by using the setState method within a StatefulWidget. This approach is ideal for managing local state within a single widget, as shown in Listing 3 as an example.

In this example, the UI is rebuilt each time _incrementCounter is called, reflecting the updated state. *InheritedWidget:* for sharing state across multiple widgets in the widget tree, Flutter’s InheritedWidget can be used. This method enables the propagation of data down the tree without the need for explicit parameter passing at every level, as shown in Listing 4.

Widgets that need access to the shared state can call MyAppState.of(context) to retrieve the current counter value and the increment function. This approach is more scalable than using setState alone when dealing with multiple levels of the widget hierarchy. Widgets that read counter using .of(context) - will automatically rebuild when updateShouldNotify returns true - in this case, when the counter changes. This pattern is particularly useful for sharing common data among many widgets without having to pass parameters explicitly through every intermediate widget. Full code with comments is provided in Appendix, Listing 5.

Flutter also provides other tools that are part of its core framework for state management:

ValueNotifier and *AnimatedBuilder:* ValueNotifier is a simple way to manage and listen to changes in a value, while AnimatedBuilder can rebuild parts of the widget tree in response to changes in the notifier. Full code with comments is provided in Appendix, Listing 6 (example of ValueNotifier and AnimatedBuilder usage).

StreamBuilder: this widget listens to a stream of data and rebuilds when new events occur. It is particularly useful when working with asynchronous data sources. Full example code of StreamBuilder usage with comments is provided in Appendix, Listing 7.

While Flutter's built-in state management techniques - such as setState, InheritedWidget, ValueNotifier, and StreamBuilder - work well for simple or localized scenarios, they can become problematic as applications grow in size and complexity. In larger projects, managing the state manually with these methods can lead to several issues:

- Frequent and broad use of setState may trigger unnecessary rebuilds of widget subtrees, potentially impacting the application's performance. In complex applications, this can lead to sluggish behavior and a less responsive UI.
- While InheritedWidget allows state sharing across the widget tree, it often requires additional boilerplate and careful planning. In complex scenarios, the manual propagation of state can lead to tightly coupled code, making it difficult to modify or extend the application without introducing bugs.
- ValueNotifier is a simple way to manage state changes, but in larger applications, relying heavily on multiple ValueNotifiers can lead to fragmentation of state management. Tracking and coordinating updates across several notifiers can become cumbersome, and improper disposal of ValueNotifiers may even lead to memory leaks. Moreover, if many widgets listen to various ValueNotifiers, it can result in unpredictable rebuild patterns and performance bottlenecks.

These challenges underscore the need for more advanced state management solutions which offer better separation of concerns, more predictable state flows, and improved scalability for complex applications.

2.5.2 Definition of popular non-built-in approaches

Flutter provides developers with a wide range of state management options, as outlined in the official Flutter documentation (Google LLC, n.d., *List of state management approaches*). The extensive list includes both built-in methods and third-party libraries, each offering unique features and trade-offs. To identify the most popular state management solutions, we analyzed the downloads metric for each package on pub.dev, the official packages repository for Dart and Flutter. That metric serves as an indicator of community adoption and satisfaction, helping us narrow down the most widely used and trusted solutions.

Below is the list of state management approaches from the Flutter documentation, along with their downloads count; data valid as of extraction date 15th February 2025 (Google LLC, n.d., *List of state management approaches),* several available packages are excluded by following criteria:

- packages with "Dart 3 incompatible" mark not included as technically outdated
- as per Coverage mentioned in Section 2.1, packages with less than 50k downloads as possibly immature, small community (comparing with packages having at least 90-100k downloads minimum, up to 1-3M downloads).

Gathering steps are:

- going by list of approaches top to down, open each approach page on pub.dev
- check if package falls under exclusion criteria, if not – add it into Table 1 below with downloads count.

Packages sorted by Downloads count top to down, same sorting to be applied in further orderings.

Package	Downloads	Description and package link on pub.dev
Provider	3.8M	A wrapper around InheritedWidget to make them easier to use and more reusable. https://pub.dev/packages/provider
BLoC	2.7M	A predictable state management library that helps implement the BLoC (Business Logic Component) design pattern. https://pub.dev/packages/bloc
Riverpod	2M	A reactive caching and data-binding framework. Riverpod makes working with asynchronous code a breeze. https://pub.dev/packages/riverpod
GetX	668k	Open screens/snackbars/dialogs without context, manage states and inject dependencies easily with GetX. https://pub.dev/packages/get
Redux	252k	Redux is a predictable state container for Dart and Flutter apps https://pub.dev/packages/redux

Package	Downloads	Description and package link on pub.dev
MobX	97.7k	MobX is a library for reactively managing the state using the power of observables, actions, and reactions to supercharge your Dart and Flutter apps. https://pub.dev/packages/mobx

Table1. List of state management approaches (Google LLC, n.d., *List of state management approaches*, retrieved 15 February 2025)

These solutions represent a mix of lightweight, easy-to-use libraries (e.g., Provider, Riverpod) and more structured, pattern-driven approaches (e.g., BLoC, Redux). By focusing on these widely adopted solutions, this study aims to provide practical insights that are relevant to most Flutter developers.

2.5.3 Provider

Provider is a library based on the InheritedWidget approach (described in part 2.5.1). It simplifies the use and creation of classes that manage state and combines the InheritedWidget concept with the ChangeNotifier class, among other things (Rousselet R., n.d.).

The abstract class ChangeNotifier provides the ability to register event listeners for the inheriting class and notify them of changes by calling a corresponding method, as shown in Listing 8 (Google LLC, n.d., *ChangeNotifier class*).

This combination is also offered for other concepts and classes. For example, Provider can be used with Listenables (objects that can emit events), ValueListenables (Listenables that notify about changes to a single variable), Streams or Futures.

It can be concluded that Provider first provides a form of Dependency Injection (DI) and, in the second step, is linked to a class that notifies about changes, thereby enabling it to be expanded into a state management solution.

To use a ChangeNotifier as a state management solution with Provider, it is necessary to integrate the respective state classes (often called stores) into the widget tree. This is done as shown in the ProvidingWidget class in following listing, by creating a widget that does not render its own user interface but only passes widgets “downward.” As result that data from the state class can be accessed by all widgets that reside in one of the downstream branches of the injecting widget within the widget tree, as shown in Listing 9.

The downstream widgets can retrieve the state class through a simple, generic method call provided by the Provider library. By using InheritedWidget within the library, it is ensured that when data in the state classes changes, the widgets that reference them are notified of the change and, if necessary, rebuilt.

2.5.4 BloC

BLoC, which stands for Business Logic Component, is a design pattern rather than a library or widget. Originally developed by Google, the BLoC pattern was designed to enable code reuse across web, mobile, and backend applications. Its adoption by the Flutter community has made it synonymous with state management, particularly for complex applications. The BLoC pattern treats data as streams and manages state reactively, ensuring a clear separation between the UI (user interface) and business logic. This separation offers several advantages (Bloc Library, n.d., *Why Bloc*):

- Maintainable: clear separation of concerns makes the codebase easier to navigate and update.
- Testable: business logic can be tested independently of UI interactions, improving code reliability.
- Scalable: the pattern is well-suited for handling complex app states, making it ideal for large-scale applications.

BloC pattern was designed with three core values in mind (Bloc Library, n.d., *Why Bloc*):

- Simple: easy to understand and accessible to developers of varying skill levels.
- Powerful: enables the creation of complex applications by composing smaller, reusable components.
- Testable: facilitates comprehensive testing of every aspect of an application, ensuring confidence during development and iteration.

The BLoC pattern revolves around the use of streams and sinks to manage state. Key components include:

- StreamControllers: control the flow of data streams.
- StreamTransformers: process data as it flows through the streams.
- StreamBuilders: rebuild the UI when new data arrives in the stream (Payne, 2024, p. 120).

BLoC components receive state events from widgets via sinks and propagate updated state back to the widgets via streams. This reactive approach ensures that the UI remains in sync with the underlying state.

The BLoC pattern adheres to three non-negotiable rules that were defined in Paolo Soares's presentation (Soares, 2018, 23:45):

1. Inputs and outputs are simple Streams/Sink only: BLoC components must communicate with widgets exclusively through streams and sinks. This ensures a clean separation between the UI and business logic, as widgets cannot directly call methods or access variables within the BLoC.

2. Dependencies must be injectable and platform agnostic: BLoC components must not depend on the user interface or platform-specific code. This ensures that the business logic remains reusable across different platforms (e.g., web, mobile, desktop).

3. No platform branching allowed: BLoC components must not contain platform-specific logic. This ensures that the same BLoC can be used across multiple platforms without modification

The first rule means that BLoC must not expose methods or variables to the outside but only communicate with widgets through streams and sinks. In Flutter, a stream is an asynchronous flow of data or events. Widgets can subscribe to this event flow and will be updated when it changes. A sink is also a type of stream internally, but with the special feature that new events can be added from the outside. Data and events can be passed to the BLoC through this sink.

The second rule states that BLoC must not have any dependencies on the user interface. Even importing Flutter libraries into these files is forbidden. This ensures that BLoCs are completely platform-independent, meaning that the entire user interface could theoretically be replaced without having to modify the logic.

The third rule stipulates that within BLoCs, no distinctions may be made between operating systems or platforms. The internal structure of a BLoC is not explicitly prescribed; its purpose is to process the data and events received through the sinks and then propagate the new state back to the widgets via the streams. In practice, techniques and tools from reactive programming, such as RxDart, are often employed for this. Each screen should be assigned exactly one BLoC. For widgets to access this BLoC, they must be injectable - that is, shared among multiple widgets.

The BLoC pattern has evolved to include a simplified version called Cubit. While both BLoC and Cubit are part of the same ecosystem, they differ in their implementation and complexity (Bloc Library, n.d., *Bloc Concepts*).

Feature	BLoC	Cubit
State management	Uses events and states to manage state.	Uses methods to emit state changes directly.
Complexity	More complex due to the use of events and states.	Simpler and more lightweight, with less boilerplate code.
Use case	Ideal for complex applications with intricate state transitions.	Suitable for simpler applications or when a lightweight solution is needed.

Reactivity	Highly reactive, with explicit state transitions triggered by events.	Reactive, but state changes are triggered by method calls.
Learning curve	Steeper learning curve due to the need to define events and states.	Easier to learn and implement, making it more beginner-friendly.

Table 2. BloC and Cubit implementation and complexity

Main BLoC functions loop (Bloc Library, n.d., *Bloc Concepts*):

1. Events: widgets send events to the BLoC via a sink. These events represent user actions or other triggers that require a state change.
2. State transitions: the BLoC processes the events and maps them to state transitions. Each event triggers a new state, which is emitted via a stream.
3. UI updates: widgets listen to the BLoC's stream and rebuild themselves whenever a new state is emitted. This ensures that the UI always reflects the current state of the application.

The BLoC pattern is a powerful and scalable state management solution for Flutter applications, particularly for complex projects requiring a clear separation of concerns and high testability. Its reactive programming model and platform independence make it a versatile choice for multi-platform development. However, its complexity and boilerplate code may make it less suitable for smaller projects or teams with limited experience in reactive programming. For simpler use cases, Cubit offers a lightweight alternative within the same ecosystem, providing many of the benefits of BLoC with reduced complexity (Arshad, 2022, p. 62).

2.5.5 Riverpod

Riverpod extends the foundational concepts of the Provider library, introducing enhanced functionalities that offer greater flexibility in state management. Unlike Provider, Riverpod does not require integration into the widget tree for its providers. This allows for multiple providers of the same class type to be distinguished, not by type but by a variable name. This independence from the widget tree means developers can choose how to access these variables, such as through global variables or using dependency injection solutions like get_it. Consequently, Riverpod operates without dependencies on the Flutter framework, functioning as a pure Dart library (Riverpod, n.d.).

Riverpod key points:

- ProviderScope: this widget defines a scope for all providers. The entire application or any part of it must be wrapped in a ProviderScope to utilize Riverpod's features (Riverpod, n.d.).
- Provider: at its core, a provider in Riverpod is a class that supplies values to other classes. There are several types of providers tailored for different use cases (Riverpod, n.d.).

- Consumer: this widget listens for changes in providers and rebuilds itself accordingly. Riverpod supports Consumer and ConsumerWidget, allowing widgets to react to state changes (Moore et al., 2024, p. 399).
- Ref: a Ref is a reference used to access other providers. It's available in both providers and ConsumerWidgets, facilitating dependency injection and state access (Moore et al., 2024, p. 399).

Riverpod's structure avoids the need for nested providers due to its ability to read other providers during their creation, which is a significant advantage over Provider, where such nesting is necessary for dependent providers (Riverpod, n.d.).

For the state classes themselves, Riverpod provides its own structure. The state classes inherit from the generic class StateNotifier (Listing 10). The generic parameter specifies the data type of the state. This state is then made available to the class as an inherited variable. When the state changes, it is sufficient to simply update the contents of this variable, as shown in previous Listing 10.

The state class is only responsible for providing methods to modify this state. In addition to these state classes, Riverpod also accepts event emitters such as ChangeNotifier, which were introduced in the Provider chapter (section 2.4.3). As mentioned earlier, since Riverpod is not a Flutter library and does not use InheritedWidget like Provider to update referencing widgets, it does not inherently support usage within a widget. Riverpod introduces ConsumerWidget, which replaces or supplements StatefulWidget and StatelessWidget (Joshi, 2023, p. 228). By extending this class, the widget's build method is provided with a new variable through which the state of Riverpod providers can be accessed, as shown in Listing 11. The watch method used here ensures that the widget is rebuilt whenever the corresponding provider changes.

2.5.6 GetX

GetX stands out as a multifaceted library for Flutter, integrating state management, dependency injection, and route navigation into a singular, efficient package. It's crafted with an emphasis on simplicity, speed, and productivity, enabling developers to bypass the usual verbosity of app development (GetX, n.d.).

Key features of GetX:

- State management: GetX is adopting a reactive programming model, so the widgets automatically update when the underlying data changes. Observables are used to track state changes, ensuring that only the necessary parts of the UI rebuild, which significantly boosts performance in large applications.
- Dependency injection: GetX simplifies dependency management with its context-free dependency injection system. This feature allows developers to inject services or controllers

into widgets without passing them through constructors or using BuildContext. This not only reduces code complexity but also enhances modularity and testability of components.

- Route management: navigation becomes straightforward with GetX, as it eliminates the need for BuildContext in routing. Developers can manage app navigation with ease, creating smooth transitions between screens with less boilerplate, which is particularly beneficial for apps with complex navigation patterns.
- Reactive programming: at the heart of GetX's efficiency is its use of Rx (Reactive Extensions), which facilitates reactive programming. This paradigm allows for the creation of responsive UIs that reflect state changes in real-time, offering a more seamless user experience by minimizing manual state updates.
- Utility functions: beyond its core functionalities, GetX provides additional utilities like internationalization support, theme management, and storage management, making it a comprehensive toolkit for app development (GetX, n.d.)

Listing 12 shows how to set up a simple counter using GetX for state management. We define a CounterController that extends GetxController to manage the state. The count variable is made observable with .obs, so any change to it will notify listeners. The CounterWidget uses Get.put() to either create or fetch an instance of CounterController, and Obx to reactively update the UI when the counter changes.

Listing 13 shows how GetBuilder can be used for state management to receive more control over when the UI updates. The CounterController still manages the state, but now count is a simple integer, and we manually call update() to notify the UI to rebuild. GetBuilder listens for these update() calls to know when to refresh the UI (Arshad, 2022, p. 85).

GetX reduces the amount of boilerplate code, leading to more maintainable and cleaner codebases. Its performance benefits from optimized state handling, where only necessary UI components are rebuilt. The API's simplicity enhances productivity by making state management, dependency injection, and navigation more intuitive.

While the basics are easy to grasp, mastering the reactive aspects of GetX can be challenging for those unfamiliar with reactive programming. Furthermore, although the GetX community is expanding, the support in terms of documentation and community might not yet match that of more established libraries, potentially impacting advanced usage or troubleshooting.

2.5.7 Redux

Redux is a predictable state container for JavaScript applications, which has been adapted and widely used in Flutter through the flutter_redux package. It's inspired by the architectural patterns offering a centralized store for managing application state. Redux is particularly beneficial for applications where state management needs to be strict, predictable, and manageable across multiple components (Redux, n.d.).

Key features of Redux:

- Single source of truth: Redux centralizes the state in one store, simplifying state management as all changes must go through this store, ensuring data consistency across the app (Redux.js, n.d.).
- State mutations are explicit: changes to the state are made through actions dispatched to reducers, which must be pure functions, making the flow of state changes predictable and traceable for debugging purposes (Redux.js, n.d.).
- Separation of concerns: encourages a clear distinction between business logic and UI, improving modularity and testability of the code (Redux.js, n.d.).

Redux has four major concepts (Arshad, 2022, p. 66):

- Store – holds the state object of the whole application
- Reducer – updates the state based on the action it receives
- State – the current snapshot of the application's UI
- Actions – the instructions to update the states

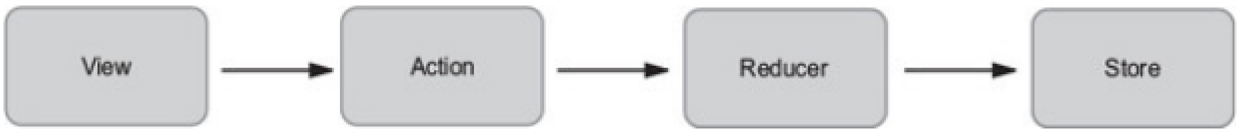


Figure 4. Data flow through Redux application (Garreau, 2018, p. 15).

The principle of “Single source of truth” is implemented by having a single, consolidated state for the entire application. This means that there is no division into various sub-states, for example, for individual pages. This global state, as described in the state mutations approach, is immutable. To change or replace the state - it is necessary to emit actions. In Flutter, actions are simple objects that can also include parameters in the form of variables. As illustrated in Figure 4, this action is then applied to the global state (i.e., store) via a reducer. A reducer is a simple function that receives the current state and the action as parameters and returns a new state, which replaces the global state (Garreau, 2018, p. 16).

Listing 14 shows implementation of simple counter application using Redux in Flutter. Code explanation:

- AppState defines the structure of our application's state, here it's just a counter.
- Actions are defined as an enum, providing a clear, type-safe way to describe state changes.
- The reducer is a pure function that specifies how the state changes in response to actions.

- The Store acts as the single source of truth for the application's state, initialized with the reducer and an initial state.
- StoreProvider makes the store available to all descendants in the widget tree.
- StoreConnector connects the UI to the Redux store, allowing for state reading and action dispatching.

Redux's design philosophy supports large-scale applications where state management needs to be rigorous and centralized. Its predictability and the ability to test individual parts of state management make it ideal for complex applications needing scalability and maintainability. The overhead of setting up Redux, particularly the additional boilerplate for actions, reducers, and store setup, can be seen as a drawback for smaller applications or those with less complex state requirements. The learning curve can also be steeper due to its unique paradigms compared to more straightforward state management solutions.

2.5.8 MobX

MobX is a state management solution for Flutter, inspired by reactive programming concepts, which aims to simplify the management of application state by automatically updating the UI when the state changes. It leverages the power of observables, actions, and reactions to manage state reactively, allowing developers to focus more on business logic than on manually updating the UI (MobX, n.d.).

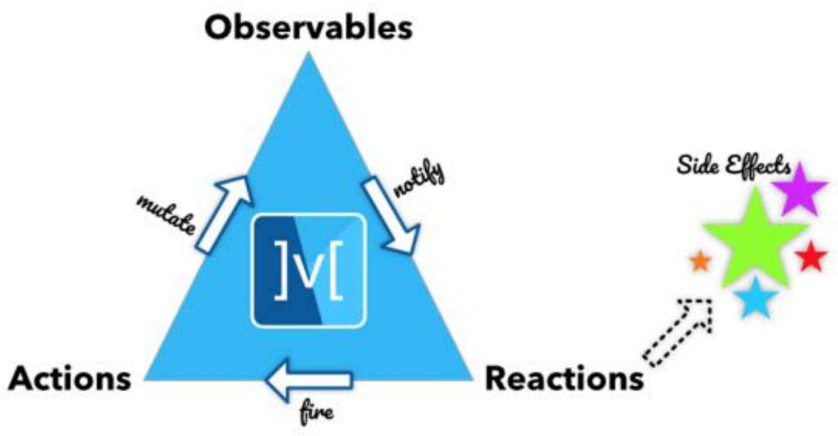


Figure 5. Core concepts of MobX

Key features of MobX:

- Reactive programming: MobX uses observables to track state changes automatically. Whenever an observable state changes, all reactions dependent on that state are re-run, ensuring the UI reflects the latest state without manual intervention.
- Actions: these are used to modify the state. Actions encapsulate the logic for state changes, ensuring that all changes are atomic and predictable.
- Reactions: reactions allow you to respond to state changes, typically used for side effects like logging, network calls, or updating the UI.

- Code generation: MobX can generate boilerplate code for observers and actions, reducing the amount of code developers need to write manually, thus enhancing productivity (Slepnev, 2020).

Listing 15 contains example of counter application using MobX. Code explanation:

- Counter class defined as observable
- The part 'counter.g.dart'; indicates that this class will be complemented by generated code.
- Counter is a class that uses MobX's store mixin, making value an observable variable.
- @observable annotation makes value detectable for changes.
- @action annotation on increment ensures that changes to value are done within an action, which helps in managing the state mutation lifecycle.
- Observer widget listens for changes in the Counter state and rebuilds its child when value changes.
- The increment method is called directly from the UI, which modifies the observable state.

MobX reduces the boilerplate code, especially with its code generation capabilities, making state management more intuitive. By only updating the necessary parts of the UI based on what's being observed, it can be very performant for complex UIs with many reactive components Although simpler for basic use, understanding the full potential of MobX might require learning about reactive programming and how MobX handles state mutations. While MobX simplifies state management, debugging reactive chains can sometimes be challenging if not set up correctly. Also, there is a lack of built-in dependency injection framework, need to use Provider to resolve that (Slepnev, 2020).

2.6 Comparative insights from literature

Prior studies (e.g., Arshad, 2022) suggest Provider and Riverpod can be used for almost any sort of application that is not so complex, having a decent backend that returns data in the form of JavaScript Object Notation (JSON) responses. Applications related to banks, productivity, tasks, to-do applications, and so on can be built using Provider and Riverpod. Since Provider is not reactive, it sometimes gets difficult to manage when your application starts to get bigger in terms of the code base.

Distinctive approach such as GetX is usually used in specific cases where developer want to have ease of use and spend less time building the architecture of the application. Scaling applications or adding/removing features from application using these techniques might not be very feasible (Arshad, 2022, p. 212).

GetX can be used by the developers who are new to Flutter and haven’t found the most comfortable state management approach yet but want to have the benefits of reactive programming while keeping the things simple (Slepnev, 2020, p. 70).

If developer is building an application where exists a single source of data for the state in application - Redux can be a good choice since it's unidirectional and it keeps its state all at a single place, as it is usually done in the React framework. Chat applications can be built using Redux as well (Arshad, 2022, p. 213).

Redux is a good solution for large applications due to the benefits of testability, scalability, ease of debugging and predictability that it provides. However, it requires some special knowledge and the ability to understand its core concepts which are not easy to grasp. For smaller applications, the advantages of Redux usually do not outweigh the disadvantages. Also, React developers are usually familiar with Redux, so this state management approach can be used by developers with React background coming to Flutter (Slepnev, 2020, p. 89).

If there is a lot of JSON data coming in and going out of application through application programming interfaces (APIs) – MobX could be used for code autogeneration. This will surely help to create functions automatically for converting the JSON data to Dart classes and vice versa (through the build_runner package) (Arshad, 2022, p. 213).

The best use case for MobX is creating prototypes. It is very easy to learn and to implement, it has almost no boilerplate code and can fit into different architectures. Developer don’t need to actually think what is happening behind the scenes, since the framework is very abstracted. However, the freedom offered by MobX results in the fact that developer need to be really careful when the app grows because the scalability and testability depends on your actual implementation (Slepnev, 2020, p. 81).

BLoC is the go-to technique for almost any sort of application, ranging from entertainment, ride-hailing, and chat applications to email applications, and so on. Since BLoC uses streams, it might be a good choice for games as well (Arshad, 2022, p. 217).

BLoC can be considered an advanced approach, because it is impossible to start using it right away like Provider or GetX, and the concepts which lie behind it are quite complicated. However, for the price of these complexity and a lot of boilerplate code, developers can write the code that is easily testable, reusable, scalable. This approach is also suitable for clean architecture. (Slepnev, 2020, p. 77).

As per Slepnev (2020), if simplicity is the main criteria - we should consider Provider, GetX and MobX. If we need good scalability - BLoC, Redux, Provider and GetX are the options.

This thesis builds on such observations by empirically testing these claims using standardized criteria.

3. Research methodology

In the previous chapter, we explored various state management approaches available in Flutter.

This chapter outlines the methodology used to evaluate state management solutions in Flutter, with a central focus on developing, implementing, and testing a prototype application. The prototype serves as the primary tool for empirical comparison, enabling a controlled assessment of each approach against standardized criteria derived from ISO/IEC 25010 and additional metrics (e.g., documentation quality). The methodology integrates qualitative and quantitative techniques to address the research questions outlined in Section 1.2, ensuring a rigorous and reproducible evaluation.

3.1 Overview and role of the prototype

The prototype application is the cornerstone of this study, designed to simulate a realistic Flutter app scenario for comparing state management solutions (Provider, BLoC, Riverpod, GetX, Redux, MobX). Its role is to implement the following tasks:

- Real world context: the prototype mirrors common app features (e.g., user login, product listing, cart management), ensuring findings are relevant to Flutter developers (RQ1, RQ3).
- Empirical testing: by implementing identical functionality across six versions of the prototype (one per solution), it provides measurable data (e.g., widget rebuild counts, Maintainability Index) to assess performance efficiency and maintainability (RQ2).
- Comparative analysis: it enables a side-by-side evaluation of architectural paradigms and trade-offs, directly supporting the study's objective to propose best practices (RQ3).

3.2 Literature review and requirements analysis

An extensive literature review (Section 2) fulfilled the identification of state management approaches. Key requirements derived from ISO/IEC 25010 (e.g., Performance Efficiency, Maintainability) and supplemented by Flutter-specific needs (e.g., documentation), detailed in Sections 3.6–3.7 of the thesis. These requirements are shaping the following prototype's design and evaluation criteria (Sections 3.8-3.11).

3.3 Prototype development process

The prototype was developed in six parallel implementations, each using a different state management solution, to ensure consistency and fairness in comparison:

- Development approach: a modular architecture was adopted, with shared UI components (e.g., ProductListScreen, CartScreen) and solution-specific state logic (e.g., controllers, stores). Code was versioned in a Git repository (Ulvis, 2025), with branches for each implementation (e.g., branch=provider, branch=getx).
- Features, Use Case and Design (Sections 3.11): the prototype simulates an e-commerce app with:

    - User authentication (login toggle).
    - Product listing with dynamic pricing (e.g., 20% discount for signed-in users).
    - Shopping cart with quantity updates and total price calculation.
- Implementation details: each version used the latest stable package versions (e.g., provider 6.1.2, get 4.7.2) as of February 2025. Shared Dart files (e.g., product_service.dart) ensured identical backend logic, isolating state management as the variable (Section 3.12).

3.4 Experimental setup and testing

The prototype was tested in a controlled environment to generate empirical data:

- Hardware/Software: detailed in section 3.14 Experimental setup
- Quantitative metrics:
    - Time behavior: widget rebuild counts measured via flutter test test/efficiency_benchmark_test.dart (e.g., cartButton rebuilds, detailed in section 3.14).
    - Modularity: Maintainability Index calculated using DCM, detailed in 3.8.3 and Appendix ("Maintainability Index calculation" part).
- Qualitative metrics: learnability, modifiability, testability, and documentation assessed through structured analysis (e.g., code readability, dependency complexity), detailed in Section 3.8 and Appendix ("Analysis of the ISO/IEC 25010 quality characteristics" part).
- Process: each implementation was benchmarked identically ensuring reproducibility (as described in Section 3.14), with results compiled in Section 4.7.

3.5 Contribution to research questions

The prototype directly addresses the study's research questions:

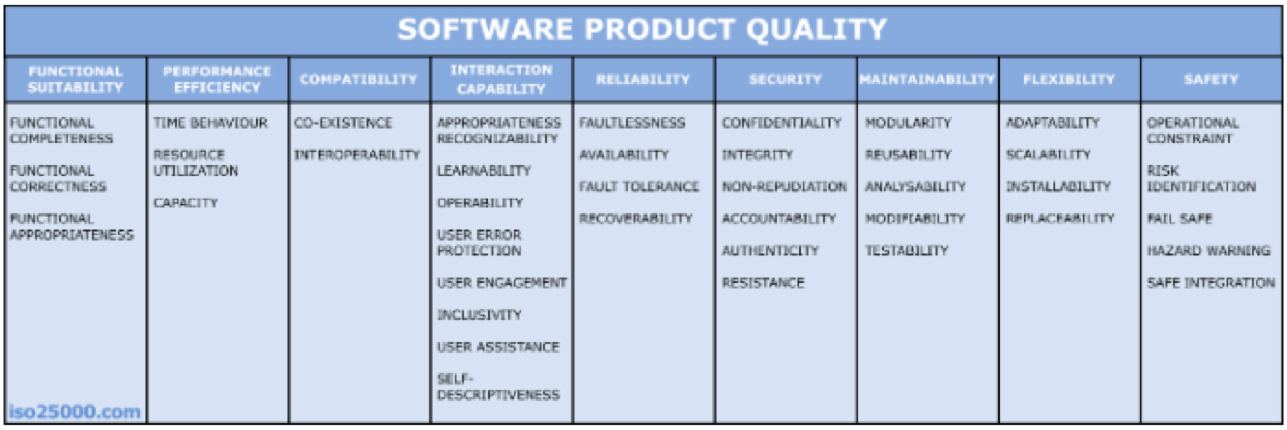
- RQ1 (widely used solutions): its implementation of six popular solutions (selected per Section 2.5.2) validates their features and paradigms empirically.
- RQ2 (performance and quality): metrics from prototype testing (e.g., rebuild counts: GetX=1, Redux=8 for cartButton) quantify ISO 25010 criteria, revealing trade-offs.
- RQ3 (best practices): comparative results leading to recommendations (e.g., GetX for efficiency, Riverpod for scalability), based on prototype-derived evidence.

3.6 Requirements analysis

The fundamental requirement for state management systems is that they must be able to manage the state of an application and certain parts of the application, such as a widget or an entire page. All the solution approaches presented so far can fulfill this requirement with varying degrees of effort. However, besides whether a system can implement this requirement, it is also important to consider how and with what quality it is implemented.

3.6.1 General requirements

The choice of a state management system significantly determines the architecture of an application. Therefore, the requirements for good architecture or good software design are also partially transferable to state management systems. To construct requirements from this, it is necessary to examine what constitutes good software design. Various requirements and criteria have been developed to evaluate software quality, which were standardized with the ISO standard 25010.



SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	PERFORMANCE EFFICIENCY	COMPATIBILITY	INTERACTION CAPABILITY	RELIABILITY	SECURITY	MAINTAINABILITY	FLEXIBILITY	SAFETY
FUNCTIONAL COMPLETENESS	TIME BEHAVIOUR	CO-EXISTENCE	APPROPRIATENESS RECOGNIZABILITY	FAULTLESSNESS	CONFIDENTIALITY	MODULARITY	ADAPTABILITY	OPERATIONAL CONSTRAINT
FUNCTIONAL CORRECTNESS	RESOURCE UTILIZATION	INTEROPERABILITY	LEARNABILITY	AVAILABILITY	INTEGRITY	REUSABILITY	SCALABILITY	RISK IDENTIFICATION
FUNCTIONAL APPROPRIATENESS	CAPACITY		OPERABILITY	FAULT TOLERANCE	NON-REPUDIATION	ANALYSABILITY	INSTALLABILITY	FAIL SAFE
			USER ERROR PROTECTION	RECOVERABILITY	ACCOUNTABILITY	MODIFIABILITY	REPLACEABILITY	HAZARD WARNING
			USER ENGAGEMENT		AUTHENTICITY	TESTABILITY		SAFE INTEGRATION
			INCLUSIVITY		RESISTANCE			
			USER ASSISTANCE					
			SELF-DESCRIPTIVENESS					

iso25000.com

Figure 6. ISO/IEC 25010 Software product quality characteristics (ISO/IEC, 2011)

Attributes such as efficiency, complexity, understandability, reusability, and testability/maintainability are considered significant (Rosenberg et al, 1997, p.1), they can be transferred to the described evaluation in context of using the ISO/IEC 25010 standard, which provides a structured framework for evaluating software quality. This standard outlines a set of quality characteristics and sub-characteristics that are crucial for assessing software products, including those related to state management systems in Flutter applications.

The analysis of the ISO/IEC 25010 quality characteristics (detailed analysis can be found in Appendix, section " Analysis of the ISO/IEC 25010 quality characteristics") reveals that, while some attributes like Performance efficiency and Maintainability are highly relevant for evaluating state management systems in Flutter, others such as Functional suitability, Compatibility, Reliability, Security, and Safety offer limited differentiation due to their focus on end-user or system-level concerns rather than developer-centric tools. Characteristics like Security and Reliability were excluded as they pertain to end-user systems rather than developer tools (see Appendix for details).

Flexibility shows partial relevance, primarily through Scalability, similar as Interaction capability contributes through sub-characteristic like Learnability. The specific metrics for these relevant characteristics - such as Time behavior, Learnability and Analyzability, Modularity, Modifiability, Testability, and Scalability - will be further defined and detailed in the subsequent sections of this chapter to guide the empirical evaluation of the state management solutions.

3.6.2 Specific requirements

In addition to general quality requirements, state management systems in Flutter (e.g., Provider, Riverpod, BLoC, GetX, Redux, MobX) have special needs that go beyond broad software standards, with Documentation standing out as a critical factor. This refers to both the volume - meaning a large amount of helpful resources like detailed tutorials, clear examples, and full API explanations - and the quality - ensuring these materials are simple, accurate, and easy to follow. Having lots of well-written documentation is essential because it allows developers to quickly learn how to set up and work with the system, cutting down on confusion and mistakes. This is especially important in fast-moving projects where developers need reliable guidance to keep up with changes and make the most of the tools (Slepnev, 2020, p. 26).

3.7 Requirements summary

Performance efficiency	*Interaction capability*	*Maintainability*	*Flexibility*	Documentation
Time behavior	Learnability	Modularity	Scalability	
		Analyzability		
		Modifiability		
		Testability		

Table 3. Attributes combination

When combining the general and specific requirements, the following list of requirements is obtained:

- Time behavior
- Learnability and Analyzability

In thesis, Learnability and Analyzability are merged as one metric because they both measure how well and easily developers can understand state management systems in Flutter, so they overlap in helping developers start using and fix issues in state management applications, matching the focus on developer experience and simplifying the ISO 25010 analysis.

- Modularity
- Modifiability and Scalability

In this thesis, Modifiability and Scalability are merged into a single metric because they both assess how effectively a state management system in Flutter can adapt to changes and growth, overlapping in their focus on maintaining system integrity and performance as complexity increases. Modifiability evaluates how easily the system's code or structure can be altered to accommodate new features or adjustments, while Scalability measures the system's capacity to handle an expanding scope, such as larger state sizes or increased interconnections between states. These concepts are closely related, as a system that scales well must inherently support modifications without disproportionate effort, and a modifiable system is better equipped to grow efficiently. Combining them aligns with the emphasis on developer experience and simplifies the ISO 25010 analysis by focusing on their shared goal: ensuring the system remains flexible and manageable under evolving demands.

- Testability
- Documentation

3.8 Evaluation criteria definition

Now that the requirements for state management systems have been established, criteria for evaluating these systems and their implementation can be defined. The aim is to establish meaningful and objective evaluation criteria. These criteria should directly address the respective requirements.

3.8.1 Time behavior

The efficiency of a state management system is evaluated by its ability to minimize widget rebuilds after state changes, ensuring only affected widgets are updated. Time behavior is assessed by tracking rebuild frequency at two key points in the application. A standardized automated test sequence will simulate consistent user actions across all systems. Rebuilds will be quantified using counters, the goal is to achieve low rebuild counts while maintaining functionality.

3.8.2 Learnability and Analyzability

Assessing how readable and understandable the source code of a state management system is can be tricky to measure with numbers, so this study will use a thoughtful qualitative approach instead, guided by a set of specific questions. For learnability, a big plus is when a state management system builds on ideas developers already know from Flutter. This lets newcomers to the system lean on their existing skills, making it quicker and easier to get started. For analyzability, the focus is on whether the code has a clear, easy-to-follow structure that developers can trace without getting lost. However, how much the state management system itself shapes this structure can differ, so we'll need to weigh its impact carefully during the evaluation.

In Flutter, as explained in Section 2.3, widget trees are built using constructor calls, which can stack up into deep layers of nesting. This nesting often makes code harder to read and follow, especially as the application grows. A good state management system should tackle this problem by offering ways to keep the code cleaner and less tangled - maybe through better organization or tools that simplify how widgets connect to state.

To judge these qualities, we'll use a straightforward scale with three levels: "low" (if the system does little to help with learning or clarity), "medium" (if it helps somewhat but still has gaps), and "high" (if it really shines in making things easy to learn and analyze). This scaling lets us fairly compare how each system supports developers in understanding and working with it, based on real, practical observations.

3.8.3 Modularity

Modularity is quantified via the Maintainability Index (MI), which assesses code complexity and cohesion, with higher scores indicating better maintainability.

To assess modularity, a quantitative approach can be applied to evaluate how well-separated and independent the components of a state management system are within a Flutter application. The literature offers various methods and metrics for this purpose (Kurt et al, 1997). The Maintainability Index (MI), as shown in Equation 1, combines multiple metrics with appropriate weighting.

$$\text{Maintainability index} = 171 - 5{\cdot}2 \times \ln(\text{aveV}) - 0{\cdot}23 \times \text{aveVG2} - 16{\cdot}2 \times \ln(\text{aveLOC})$$

Equation 1.

it incorporates the average Halstead volume (aveV), average cyclomatic complexity (aveVG2), and average number of source code lines (aveLOC) (Kurt et al, 1997, p. 133). This blend of metrics aims to produce a single-value score that reflects the degree of modularity in the system, indicating how easily its parts can be modified independently (Kurt et al, 1997, p. 129).

To compute this metric for each state management system, the tool Dart Code Metrics will be utilized, which analyzes Dart source code to provide relevant metrics. The resulting value, slightly adjusted as shown in Equation 2, will be scaled from 0 to 100, with 100 being the highest level of modularity achievable, with 85+ indicating good maintainability per industry norms (DCM team, n.d.).

$$MI = \max(0, (171 - 5.2 * \log(HALVOL) - 0.23 * \log(CYCLO) - 16.2 * \log(SLOC)) * 100 / 171)$$

Equation 2.

This metric will be used as the evaluation benchmark for modularity.

3.8.4 Modifiability and Scalability

To determine whether a state management system is scalable or easily modifiable, it is necessary to examine how it handles an increasing state size. A key criterion here is how effectively different states can be interconnected. As applications grow, it is expected that the coupling between individual states will also increase.

For this evaluation criterion, a qualitative rating scale will be used, assessing modifiability/scalability with the values "low”, “medium”, and “high”.

3.8.5 Testability

Several perspectives are relevant for assessing testability. First, it should be examined to what extent the business logic implemented with the state management system can be tested. For example, this can involve checking whether individual state-altering functions can be tested without modifying the original source code, ideally using automated unit tests as a foundation.

Another perspective is the testability of widgets that access shared states. Here, it should be evaluated whether these shared states can be easily replaced with placeholders (mocks) for widget testing purposes.

Based on the assessment of these properties, an evaluation will be conducted using the ratings "low”, “medium”, and “high”.

3.8.6 Documentation

For documentation to be effective in aiding developers with state management systems, it needs to be practical and thorough. First, we’ll check if documentation exists for each system. If it does, we’ll look at whether it clearly explains the system’s core concepts and how it works, giving developers a solid starting point. Another important aspect is whether it includes plenty of helpful examples that show how to use the system in real coding situations - these examples are key to making ideas easier to grasp and apply. Developers community volume and third-party tutorials and examples are also have impact.

Using these points, we’ll evaluate the documentation with a simple rating: "low" if it’s too weak to be useful, "medium" if it covers some things but isn’t complete or clear enough, and "high" if it’s detailed and does everything developers need. This straightforward method keeps the analysis focused and reliable for the thesis.

3.9 Evaluation criteria summary

In summary, the requirements and evaluation criteria result in an assessment matrix, as shown in Table 4, which will be applied to a prototype for each state management system in the following section.

Characteristic	Metric	Measurement value	Value type
Time behavior	quantitative	widgets rebuild count	N:N
Learnability and Analyzability	qualitative	low / medium / high	selection
Modularity	quantitative	Maintainability index	0-100
Modifiability and Scalability	qualitative	low / medium / high	selection

Testability	qualitative	low / medium / high	selection
Documentation	qualitative	low / medium / high	selection

Table 4. Criteria definition.

3.10 Description of the prototype development approach and application requirements

Now that the requirements for state management systems have been defined, it’s time to set up an experiment to evaluate them. To make the evaluation as practical as possible, it makes sense to test them using a sample application. This approach has the advantage of putting the state management systems under realistic conditions, which boosts the evaluation’s relevance. However, it also carries the risk that other variables, not part of the evaluation, might skew the results. To keep this risk in check, steps will be taken to ensure that only the effects of implementing a state management system are being assessed.

Before building such a sample application, it’s necessary to define its requirements and then plan a design based on those.

3.10.1 Implicit requirements.

The primary requirement for the sample application is to enable a thorough evaluation based on the criteria defined in Table 4. To ensure this, each evaluation criterion must be reviewed to determine whether it places specific demands on the application’s design and functionality. Criteria that do not introduce unique requirements will not be elaborated further in this section, keeping the focus on those that shape the application’s structure.

For modifiability and scalability, the application must incorporate a noticeable degree of state coupling. This means designing the app so that different states - such as user preferences, UI settings, or data displays - interact or depend on one another to some extent. This setup allows us to test how effectively each state management system manages interconnected states, revealing its ability to handle changes and growth as the app scales.

To evaluate testability, the application’s widgets need to be crafted with automated testing in mind. This involves structuring the code so that testing tools, like Flutter’s flutter_test package, can easily interact with it without running into obstacles caused by overly complex or rigid implementations. Beyond that, the app should include practical use cases that require state processing - for example, updating a counter based on user input or toggling a display mode. These scenarios will let us implement and test basic business logic within each state management system, ensuring we can assess how well it supports unit and widget testing.

For time behavior, the application must provide specific measurement points to track performance. This requires including widgets that can serve as reliable indicators of efficiency, such as those that update in response to state changes. Widgets positioned outside of list views are particularly valuable here, as their rebuild behavior won't be tied to the complexities of list rendering (e.g., scrolling or lazy loading). This independence ensures cleaner data on how state changes affect rebuild frequency performance metric.

Finally, to examine learnability and analyzability in the context of deep nesting, the application needs to intentionally create this situation. In Flutter, nesting happens when widget trees grow deep due to repeated constructor calls, which can make code harder to read and understand. To test how state management systems handle this, the app should feature pages with multiple independent, page-wide states - like a settings page with separate controls for theme, notifications, and user profile data. By forcing this layered structure, we can see how each system helps (or hinders) developers in keeping the code clear and manageable despite the complexity.

These implicit requirements ensure the sample application aligns with the evaluation goals, providing a realistic and controlled environment to compare the state management systems effectively.

3.10.2 Explicit requirements

Now that the implicit requirements have been outlined, it's time to establish additional requirements for the sample application to account for other important factors. This subsection will define explicit requirements to ensure the evaluation is as practical as possible, creating a realistic scenario for testing the state management systems.

The sample application should include multiple pages to allow testing of state management behavior during navigation. It's crucial that data remains consistent and persistent across all pages in the app, without any discrepancies.

In mobile applications, users are usually given several ways to interact with and change the app's state. To reflect this, the sample application should include some form of interaction that alters its state, mimicking real-world use. Additionally, the app should load data for its state from external sources, such as an external server, to simulate real-world approach usually used.

3.11 Application use case, features and design

Having established the evaluation criteria and the implicit and explicit requirements for the sample application in the previous sections, the focus now shifts to defining the practical framework through which these criteria will be assessed. This step is crucial to ensure that the state management systems are tested under realistic conditions that reflect both the defined requirements and common Flutter development scenarios. By specifying the application's structure and functionality, this section sets the stage for the subsequent implementation and evaluation phases, providing a clear blueprint for comparing the performance efficiency, maintainability, flexibility and interaction capability of each state management approach.

In this part, we will first introduce the selected use case, detailing how it aligns with the evaluation goals. Next, we will describe the application's key features, linking them directly to the implicit requirements such as state coupling, testability, and efficiency measurement. Finally, we will present the initial design of the user interface, ensuring it supports the functional scope and facilitates the intended analysis. Together, these elements form the foundation for a consistent and reproducible evaluation across all state management systems under investigation.

3.11.1 Use case

The chosen use case for the sample application is the product selection and cart view of an online shop. This scenario has the advantage of fulfilling all the defined requirements while also providing a realistic context for testing.

The foundation is a screen with list of various products loaded from the internet, from which users can add a certain quantity of one or more products to a shopping cart. Second screen is a cart view that displays all the products currently in the cart along with a total price. This setup meets all the explicit requirements.

3.11.2 Features

Building on use case from previous point, the individual functions of the sample application will now be described in relation to the established requirements.

To test the coupling of different states, the sample application will include a feature allowing users to log in and out as registered users. This will be implemented through a toggle switch that reflects the current login state. The coupling comes into play because registered users receive a 20% discount, creating a dependency between the calculation of the cart's total price and the login state. This setup ensures that the state management system's ability to handle interconnected states can be evaluated effectively.

At the same time, this feature meets the requirement for testing deep nesting. The login state, discount application, and cart total are independent states that coexist on a single page, requiring a layered widget structure. This complexity will allow us to assess how well each state management system manages readability and analyzability in a nested environment.

The calculation of the cart's total price also fulfills the testability requirement. It provides a practical use case for implementing business logic within the state management system - such as applying the discount based on login status - which can then be validated through automated unit and widget tests. This ensures the system's support for testing business logic is thoroughly examined.

To enable efficiency measurement, a button will be added to display the number of items in the cart. Beyond showing this count, the button will also navigate to the cart's detail view when tapped. Positioned outside the product list display, this button remains independent of list rendering changes (e.g., scrolling or item updates). This placement makes it an ideal candidate for tracking rebuild frequency and resource utilization, as its behavior isolates the impact of state changes from list - related overhead.

3.11.3 Design

Now that the functional scope of the application has been outlined, the next step is to design a user interface based on it. Before implementing this in Flutter, a preliminary design created using wireframes. As illustrated in Figure 7, these wireframes depict the individual pages (screens) of the app, showcasing navigation options between them, indicated here by arrows.

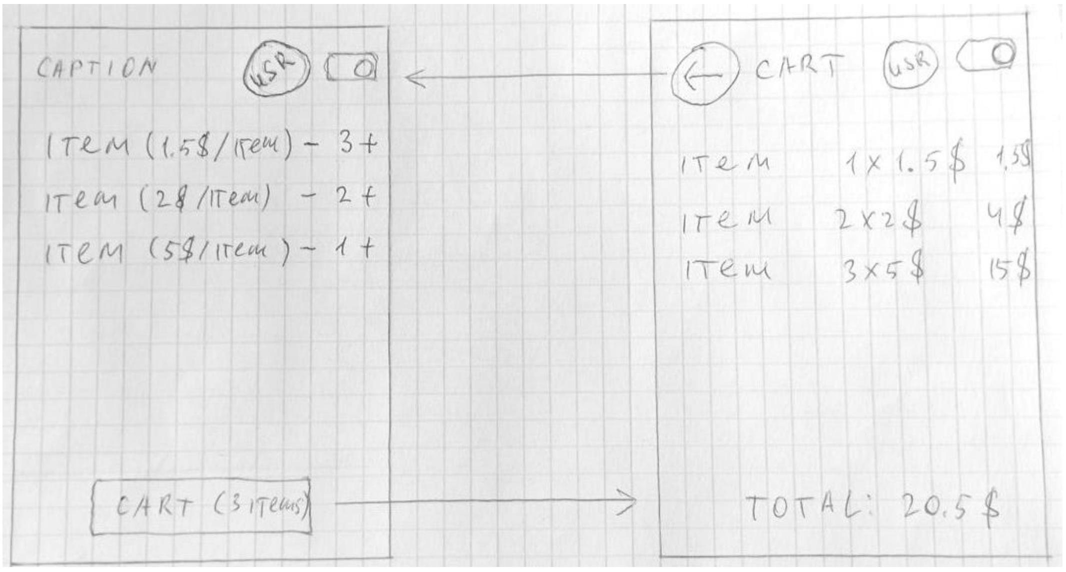


Figure 7. Application wireframes.

3.12 Implementation approach

To appropriately implement this sample application for the evaluation, a detailed description of the implementation approach is necessary to ensure the subsequent experimental setup can be understood and replicated.

Separate applications to be developed for each state management system under evaluation. To achieve that, the Git version control system will be utilized. The individual applications for each state management system will branch off from a common working branch, implementing their specific adaptations within their respective branches.

Additionally, the common working branch (main) will include the preparatory setup for efficiency measurement, ensuring that the same measurement procedure can be applied consistently across all state management systems.

Entire source code publicly available in GitHub repository (Ulvis, 2025).

3.13 Implementation result

The application was developed using Flutter version 3.29.1 using Material Design. The design deviates only slightly from the design conceptualized in the wireframes, as shown in Figure 8. Screenshots made using implemented application, same design for each version. The functions, aside from navigation, have not yet been implemented, as this will be carried out through the implementation with the state management systems, as described.

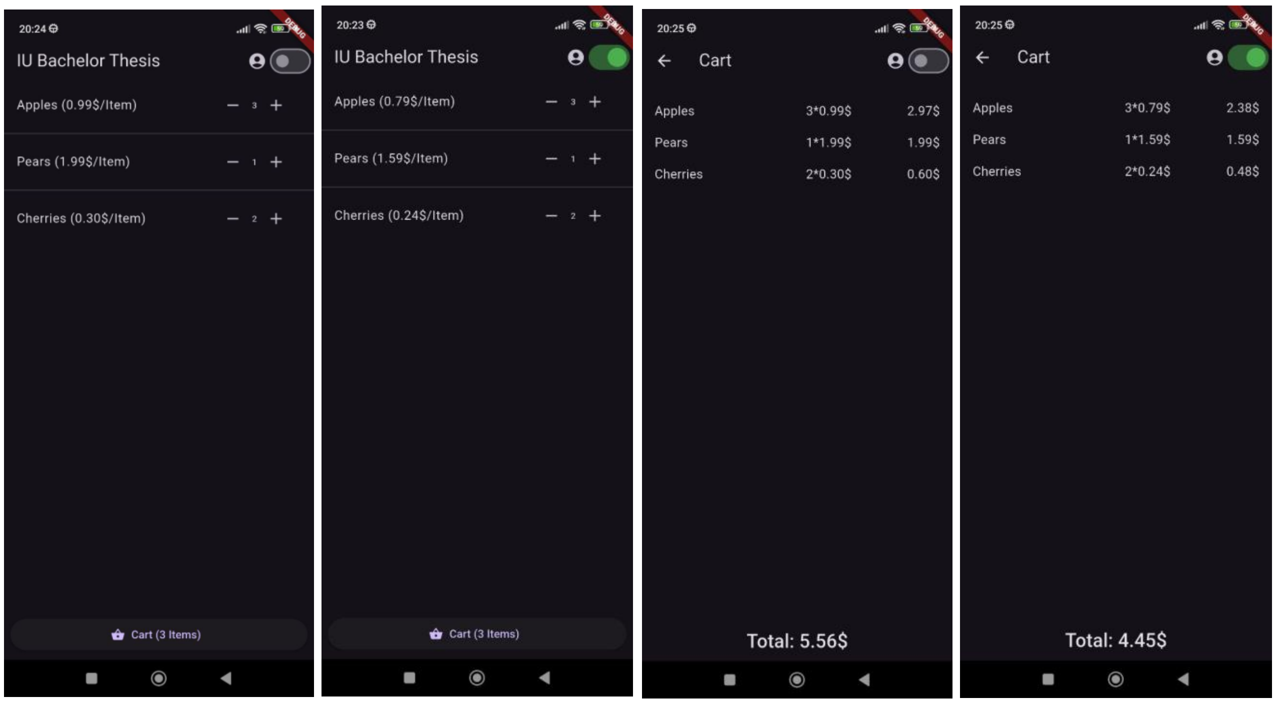


Figure 8. Application screenshots.

3.14 Experimental setup

Infrastructure established to conduct measurements for performance efficiency and modularity includes desktop PC and Android OS phone with following specifications.

Desktop:

Processor	RAM	Hard drive	OS
AMD Ryzen 5 2500X	16Gb DDR4	SSD LiteOn 256 Gb	MS Windows 10 Enterprise Edition 64-bit (Build 19043)

Android OS phone: Redmi 10 2022 (model 22011119UY)

Processor	RAM	Hard drive	OS
Helio G88	4Gb DDR4	eMMC 5.1 64 Gb	Android 13 TP1A.220624.014

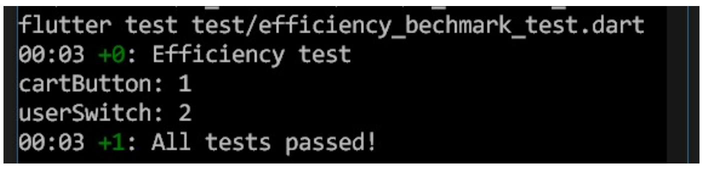
The Flutter version, as well as additional info is received by “flutter doctor -v” command and can be found in “Flutter info” section in Appendix of this thesis.

The measurement of modularity is based on the Maintainability Index metric. This value is calculated using the tool Dart Code Metrics (DCM). The individual execution steps are documented in the Appendix, specifically in the “Maintainability Index calculation” section. The complete test results are described in the Appendix of this thesis.

For the measurement of time behavior, the frequency of widget rebuilds is tracked at two specific measurement points. A counter is incremented each time the “build” method of the respective widget is called. This is evaluated using an automated test (project file test/efficiency_benchmark_test.dart), which performs a series of actions: adding and removing multiple products from the cart, triggering user login and logout, and finally navigating to the cart view.

The two measurement points are placed as follows: the first measurement point, cartButton, is triggered when rendering the cart button, and the second measurement point, userSwitch, is triggered when rendering the login/logout toggle switch. These widgets were selected because they are affected by state changes but not by all state changes in the application, and they are positioned outside of any list view.

The following result was determined as baseline when executing the test for application without state management, Figure 9:



```
flutter test test/efficiency_bechmark_test.dart
00:03 +0: Efficiency test
cartButton: 1
userSwitch: 2
00:03 +1: All tests passed!
```

Figure 9. Frequency of widget rebuilds without state management

These results represent the minimum values for the test.

4. Research findings

Now that the basic structure of the sample application and its requirements have been defined, the state management approaches can be evaluated based on the established criteria. The application will be fully implemented for each state management system, followed by the execution of tests and analyses for the evaluation.

Details of the evaluation process and criteria outputs definitions can be found in Appendix, in “Evaluations details” section. Each evaluation contains comments on implementation specifics, followed with evaluation details by each criteria with clarifications about the reasons of rating set.

4.1 Provider evaluation result

Implementation and evaluation details provided in Appendix, in “Provider evaluation details” section.

Resulting table:

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 3 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	medium
Modularity	quantitative	Maintainability index	88.02
Modifiability and Scalability	qualitative	low / medium / high	medium
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	medium

4.2 BLoC evaluation result

Implementation and evaluation details provided in Appendix, in “BLoC evaluation details” section.

Resulting table:

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 8 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	low
Modularity	quantitative	Maintainability index	88.35
Modifiability and Scalability	qualitative	low / medium / high	high
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	high

4.3 Riverpod evaluation result

Implementation and evaluation details provided in Appendix, in “Riverpod evaluation details” section.

Resulting table:

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 3 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	high

Modularity	quantitative	Maintainability index	87.72
Modifiability and Scalability	qualitative	low / medium / high	high
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	high

4.4 GetX evaluation result

Implementation and evaluation details provided in Appendix, in “GetX evaluation details” section.

Resulting table:

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 1 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	high
Modularity	quantitative	Maintainability index	85.71
Modifiability and Scalability	qualitative	low / medium / high	high
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	high

4.5 Redux evaluation result

Implementation and evaluation details provided in Appendix, in “Redux evaluation details” section.

Resulting table:

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 8 (base: 1) userSwitch: 10 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	medium
Modularity	quantitative	Maintainability index	87.89
Modifiability and Scalability	qualitative	low / medium / high	low

Testability	qualitative	low / medium / high	medium
Documentation	qualitative	low / medium / high	high

4.6 MobX evaluation result

Implementation and evaluation details provided in Appendix, in “MobX evaluation details” section.

Resulting table:

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 4 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	high
Modularity	quantitative	Maintainability index	86.79
Modifiability and Scalability	qualitative	low / medium / high	high
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	high

4.7 Results compilation

The evaluation process has yielded a detailed set of quantitative and qualitative results for the six state management systems under scrutiny: Provider, BLoC, Riverpod, GetX, Redux, and MobX. These results, derived from the implementation of a consistent sample application across each system, are compiled in Table 5 below. This compilation serves as a centralized reference point, encapsulating the performance and characteristics of each approach based on the established evaluation criteria: time behavior, learnability and analyzability, modularity, modifiability and scalability, testability, and documentation.

	Provider	BLoC	Riverpod	GetX	Redux	MobX
Time behavior cartButton base: 1 userSwitch base: 2	3 4	8 4	3 4	1 4	8 10	4 4
Learnability and Analyzability	medium	low	high	high	medium	high

Modularity	88.02	88.35	87.72	85.71	87.89	86.79
Modifiability and Scalability	medium	high	high	high	low	high
Testability	high	high	high	high	medium	high
Documentation	medium	high	high	high	high	high

Table 5. Results compilation.

In next chapter - conclusion - the results of the evaluation are analyzed to assess the extent to which the study achieved its objectives. This is followed by recommendations for various use cases and an outlook on potential future research directions.

5. Conclusion

The evaluation assessed six prominent state management systems in Flutter: Provider, BLoC, Riverpod, GetX, Redux, and MobX. Nearly all evaluation criteria proved meaningful in distinguishing the systems' strengths and weaknesses. The exception was the Maintainability Index, which showed minimal variation (ranging from 85.71 to 88.35), limiting its ability to differentiate modularity across the systems. In contrast, the time behavior metric provided significant insights into efficiency, highlighting which systems minimize widget rebuilds effectively. Qualitative assessments further enriched the analysis by offering detailed perspectives on implementation nuances.

Riverpod, GetX, and MobX emerged as top performers across multiple criteria. Riverpod and GetX both achieved "high" ratings in learnability and analyzability, modifiability and scalability, testability, and documentation, with GetX excelling in time behavior (cartButton: 1, matching the baseline) due to its reactive optimization via Obx. MobX also scored "high" in most qualitative criteria and delivered balanced efficiency (cartButton: 4, userSwitch: 4), supported by its annotation-driven clarity and scalability. Riverpod's flexibility in provider design and GetX's minimal boilerplate make them compelling evolutions over Provider, which showed "medium" scalability and learnability due to widget nesting and runtime risks. Provider's simplicity remains viable for smaller scopes but lacks the robustness of Riverpod or GetX for growth.

BLoC demonstrated "high" scalability and testability, leveraging its stream-based architecture, but its "low" learnability and analyzability, alongside inefficiency (cartButton: 8), suggest it requires optimization and familiarity with reactive programming. Redux underperformed in Flutter, with the lowest scalability ("low") and worst efficiency (cartButton: 8, userSwitch: 10), indicating its centralized store struggles outside its React origins. These findings validate Riverpod as an enhancement over Provider, while GetX and MobX offer reactive alternatives with distinct strengths - GetX in performance and simplicity, MobX in structured reactivity.

The evaluation successfully provided a comprehensive overview of these systems' functionality, meeting the study's goal of delivering data-driven insights for Flutter developers. It identified key requirements and challenges in state management, such as balancing efficiency with complexity, and enabled a meaningful comparison to guide architectural decisions.

5.1 Recommendations

The evaluation of six prominent state management systems in Flutter - Provider, BLoC, Riverpod, GetX, Redux, and MobX - offers actionable insights into their performance across time behavior, learnability and analyzability, modularity, modifiability and scalability, testability, and documentation. These findings support tailored recommendations for different application scenarios, empowering developers to select the most suitable approach based on project size, complexity, team expertise, and maintenance goals. Below recommendations for GetX, Riverpod, and Provider are detailed, each aligned with specific use cases and substantiated by the evaluation results.

GetX is recommended for applications of all sizes where performance, simplicity, and rapid development are priorities, particularly excelling in medium to large projects with dynamic state interactions. Its standout time behavior (cartButton: 1, userSwitch: 4) matches or exceeds the baseline, leveraging Obx to minimize rebuilds with precision, as seen in its reactive response to numberOfProducts changes. The "high" learnability and analyzability rating reflects its alignment with Flutter’s declarative style and centralized controllers (e.g., CartController), reducing boilerplate and enhancing readability despite requiring some reactive programming knowledge. GetX’s "high" modifiability and scalability stem from its observable-based state coupling (e.g., “ever” linking ProductController and CartController), enabling seamless growth without widget-tree dependencies. Its "high" testability, supported by Dart-only logic and mockable controllers via Get.put(), ensures robust testing for both small and large codebases. With "high" - rated documentation bolstered by community resources, GetX suits projects like a real-time messaging app or e-commerce platform with frequent UI updates, offering a lightweight yet powerful alternative. Developers should note, however, that its lack of strict structure may require discipline in massive projects to avoid state sprawl.

Riverpod is recommended for medium-sized applications prioritizing flexibility, ease of use, and scalability without the rigid conventions of larger frameworks. Its time behavior (cartButton: 3, userSwitch: 4) is efficient, matching Provider’s performance while surpassing it in architectural design. The "high" learnability and analyzability rating highlights its intuitive ref-based state access and global providers, avoiding the deep nesting of Provider and complexity of BLoC. Riverpod’s "high" modifiability and scalability shine through its dynamic provider dependencies, free from ProxyProvider’s six-dependency limit, making it ideal for evolving apps like social media platforms with profiles and feeds. Its "high" testability, with overrideable providers for mocks, supports quality assurance in growing teams, while "high"-rated documentation ensures accessibility. Compared to GetX, Riverpod offers less aggressive reactivity but greater structural freedom, appealing to developers valuing adaptability over GetX’s performance edge or MobX’s annotations.

Provider is recommended for small applications where simplicity and quick setup outweigh scalability needs. Its time behavior (cartButton: 3, userSwitch: 4) is solid for basic use, leveraging InheritedWidget efficiently. The "medium" learnability and analyzability rating reflects its reliance on familiar Flutter concepts, ideal for novices or simple apps like to-do lists, though widget nesting and scattered logic hinder clarity. Its "medium" modifiability and scalability rating flags limitations - runtime errors from missing providers and ProxyProvider's cap - making it less suited for growth beyond prototypes. The "high" testability, with lightweight mocking, and "medium" documentation, supported by Flutter's resources, fit small teams needing minimal overhead. Provider lags behind GetX and Riverpod in scalability and readability, making it a practical choice only when future expansion isn't a concern.

While BLoC, Redux, and MobX were evaluated, they are secondary options. BLoC's "high" scalability suits complex, optimized projects but demands expertise to address its "low" learnability and efficiency (cartButton: 8). Redux's "low" scalability and poor efficiency (userSwitch: 10) limit its Flutter relevance. MobX, with "high" ratings across most criteria and balanced efficiency (cartButton: 4), is a strong alternative for large, structured apps, though GetX edges it out in performance and simplicity. Developers should align their choice with project demands and team skills, using this data to optimize performance and maintainability.

These recommendations fulfill the study's aim of guiding Flutter developers with empirical evidence, ensuring selections enhance application quality across diverse scenarios.

5.2 Outlook

This study establishes a robust framework for evaluating Flutter state management, yet several ways for future research emerge. First, the Maintainability Index's narrow range (85.71 - 88.35) suggests exploring alternative metrics, such as coupling and cohesion analysis, to better capture modularity differences. Second, including emerging libraries (e.g., June, Binder) or revisiting excluded ones could reflect Flutter's evolving ecosystem, broadening the study's scope. Third, a cross-platform comparison - e.g., contrasting Redux's poor Flutter performance (cartButton: 8, userSwitch: 10) with its React efficiency - could inform developers working across frameworks. Fourth, integrating GetX's reactive strengths (cartButton: 1) into a performance-focused study with larger datasets or real-world apps could validate its edge over MobX (cartButton: 4) and others. Finally, assessing developer experience through surveys on onboarding ease or satisfaction - particularly for GetX's reactive learning curve versus Riverpod's flexibility - would complement technical findings. These enhancements would refine the research, ensuring it remains a valuable resource for Flutter developers.

References

Appfigures. (n.d.). *Top SDKs: Development - All*. Retrieved February 10, 2025, from https://appfigures.com/top-sdks/development/all

Arshad, W. (2022). *Managing state in Flutter pragmatically*. Packt Publishing.

Bloc Library. (n.d.). *Why Bloc?* Retrieved February 18, 2025, from https://bloclibrary.dev/why-bloc/

Bloc Library. (n.d.). *Bloc Concepts.* Retrieved February 18, 2025, from https://bloclibrary.dev/bloc-concepts/

vom Brocke, J., et al. (2009). *RECONSTRUCTING THE GIANT: ON THE IMPORTANCE OF RIGOUR IN DOCUMENTING THE LITERATURE SEARCH PROCESS.* ECIS 2009 Proceedings. https://aisel.aisnet.org/ecis2009/161/

DCM Team. (n.d.). *DCM: Overview*. Retrieved February 24, 2025, from https://dcm.dev/docs/metrics/function/maintainability-index/

Garreau, M., & Faurot, W. (2018). *Redux in Action* (First Edition). Manning.

GetX. (n.d.). *GetX: Extra-light and powerful solution for Flutter*. Retrieved February 18, 2025, from https://pub.dev/packages/get

Google LLC. (n.d.). *ChangeNotifier class*. Retrieved February 16, 2025, from https://api.flutter.dev/flutter/foundation/ChangeNotifier-class.html

Google LLC. (n.d.). *Flutter architectural overview*. Retrieved February 11, 2025, from https://docs.flutter.dev/resources/architectural-overview

Google LLC. (n.d.). *Supported platforms*. Flutter. Retrieved February 9, 2025, from https://docs.flutter.dev/reference/supported-platforms

Google LLC. (n.d.). *List of state management approaches*. Flutter. Retrieved February 9, 2025, from https://docs.flutter.dev/development/data-and-backend/state-mgmt/options

Google LLC. (n.d.). *Testing Flutter apps*. Retrieved February 13, 2025, from https://docs.flutter.dev/testing/overview

Google LLC. (n.d.). *Introduction to widgets.* Flutter. Retrieved February 13, 2025, from https://docs.flutter.dev/get-started/fundamentals/widgets

Google LLC. (n.d.). *Introduction to widget testing*. Retrieved February 13, 2025, from https://docs.flutter.dev/cookbook/testing/widget/introduction

Google LLC. (n.d.). *Introduction to unit testing*. Retrieved February 13, 2025, from https://docs.flutter.dev/cookbook/testing/unit/introduction

Google LLC. (n.d.). *Differentiate between ephemeral state and app state*. Flutter. Retrieved February 10, 2025, from https://docs.flutter.dev/data-and-backend/state-mgmt/ephemeral-vs-app

Google LLC. (n.d.). *Testing in Flutter*. Retrieved February 12, 2025, from https://docs.flutter.dev/testing

Kokx, H. (2023). *Flutter for Jobseekers: Learn Flutter and take your cross-platform app development skills to the next level*. BPB Publications.

Kurt D. Welker, Paul W. Oman, Gerald G. Atkinson. (1997). *"Development and Application of an Automated Source Code".* Retrieved February 24, 2025 from https://onlinelibrary.wiley.com/doi/10.1002/(SICI)1096-908X(199705)9:3%3C127::AID-SMR149%3E3.0.CO;2-S

Lavallée, M., & Robillard, P. N. (2015). *Why Good Developers Write Bad Code: An Observational Case Study of the Impacts of Organizational Factors on Software Quality*. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 1, 677–687. https://doi.org/10.1109/ICSE.2015.83

Hoang, L. (2019). *State Management Analyses of the Flutter Application*. http://www.theseus.fi/handle/10024/267674

Joshi, D. (2023). *Building Cross-Platform Apps with Flutter and Dart: Build scalable apps for Android, iOS, and web from a single codebase*. BPB Publications.

Payne, R. (2024). *Flutter App Development: How to Write for iOS and Android at Once*. Apress. https://doi.org/10.1007/979-8-8688-0485-4

Redux. (n.d.). *Redux: A state management library for Dart and Flutter*. Retrieved February 18, 2025, from https://pub.dev/packages/flutter_redux

Redux.js. (n.d.). *Redux*. Retrieved February 19, 2025, from https://redux.js.org/

Redux.dart (n.d) *redux.dart.* GitHub. Retrieved 10 March, 2025, from https://github.com/fluttercommunity/redux.dart/tree/master

Rousselet, R. (n.d.). *Provider: A wrapper around InheritedWidget to make them easier to use and more reusable*. Retrieved February 16, 2025, from https://pub.dev/packages/provider

Rousselet, R. (2022). *Provider.* GitHub. Retrieved March 10, 2025, from https://github.com/rrousselGit/provider/blob/eac827630a5f330c86857e4e13113aacdca759bc/README.md#my-application-throws-a-stackoverflowerror-because-i-have-too-many-providers-what-can-i-do

Rosenberg, L. H., & Hyatt, L. E. (1997). *Software quality metrics for object-oriented environments*. Unisys Government Systems Software Assurance Technology Center, Goddard Space Flight Center. Retrieved February 21, 2025, from https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=fc8c16be91c43bb15b72e2c73728839c1a9468ef

Riaz, M., Mendes, E., & Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 367–377. https://doi.org/10.1109/ESEM.2009.5314233

Riverpod. (n.d.). *Riverpod: A Reactive Caching and Data-binding Framework.* Retrieved February 18, 2025, from https://riverpod.dev

Slepnev, D. (2020). *State management approaches in Flutter*. http://www.theseus.fi/handle/10024/355086

Soares, P. (2018). *"Flutter / AngularDart - Code sharing, better together"*. DartConf 2018. Retrieved February 16, 2025, from https://www.youtube.com/watch?v=PLHln7wHgPE

MobX. (n.d.). *MobX: Supercharge the state-management in your Dart apps with Transparent Functional Reactive Programming (TFRP)*. Retrieved February 19, 2025 from https://pub.dev/packages/mobx

MobX.dart. (n.d.). *MobX core concepts.* Retrieved March 13, 2025 from https://mobx.netlify.app/concepts

Moore, K. D., Ngo, V., Patterson, S., & Fallas, A. U. (2024). *Flutter Apprentice (Fourth Edition): Learn to Build Cross-Platform Apps*. Kodeco Inc.

Mobile Operating System Market Share Worldwide. (n.d.). StatCounter Global Stats. Retrieved February 9, 2025, from https://gs.statcounter.com/os-market-share/mobile/worldwide

ISO/IEC. (n.d.). *ISO/IEC 25010: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. ISO 25000. Retrieved February 9, 2025, from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Ulvis, S. (2025). *State management in Flutter applications: a comparative study*. GitHub. https://github.com/Wolfram-180/iu_bachelor_thesis

Appendix

Evaluations details

Provider evaluation details

Implementation

The Provider foundational concepts can be found in subsection 2.5.3. The latest version 6.1.2 of the Provider library was used for the implementation of state management. The state managed through four different stores (providers):

- Authentication state: managed using a ChangeNotifier with a corresponding ChangeNotifierProvider.
- Product loading: implemented using a FutureProvider, which provides the result of a Future (asynchronous operation) and can handle errors during the waiting process.
- Product list: managed with a ProxyProvider2, allowing the combination of two different providers. In this case, the authentication state and product loading state were combined to apply discounts dynamically.
- Shopping cart: implemented using a ChangeNotifierProxyProvider, which is a combination of a ProxyProvider and a ChangeNotifierProvider. This allows ChangeNotifier classes to listen for changes in other providers and update accordingly.

Integration into the UI: the providers are accessed within the UI using extension functions for BuildContext. The state can be either subscribed to (to receive updates) or retrieved once if no further updates are required. Additionally, partial state selection is supported, ensuring that a widget is only rebuilt when the specific selected part of the state changes, optimizing performance.

Evaluation

The following section evaluates the implementation using Provider (Ulvis, 2025, branch=provider) based on the defined evaluation criteria.

Modifiability & Scalability

The evaluation of modifiability and scalability in Provider presents an ambivalent picture, offering both advantages and limitations.

As advantages - Provider introduces a unified abstraction for state management, meaning developers can define state using Streams, Futures, or ChangeNotifiers interchangeably. This abstraction makes it easier to switch between state management approaches later without modifying external dependencies, enhancing flexibility in long-term development.

Disadvantages are - injection by type: providers are injected into widgets based on generic class types (Provider<T>), but the compiler does not validate whether the referenced state exists within the widget tree at compile time, which make risks to have runtime errors: if the widget hierarchy changes, unresolved references to missing providers can cause runtime errors that are difficult to detect during compilation.

Limitations identified in documentation -

- The official documentation states that using more than 150 providers may cause StackOverflowErrors (Rousselet, 2022).
- Limited state coupling: Provider's capability to combine multiple states is constrained. The ProxyProvider system, which allows dependency injections between state classes, is limited to six dependencies (up to ProxyProvider6, as per package source code).

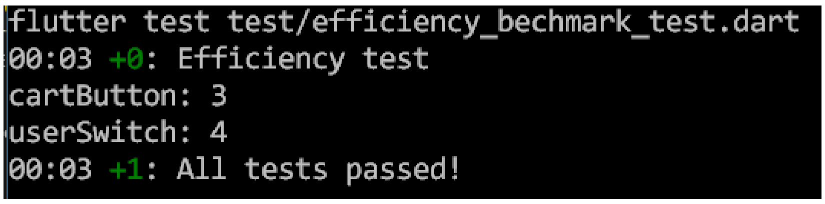
While these limitations may not often be relevant in practice, better architectural structuring (e.g., grouping related state objects) is usually recommended instead of exceeding this limit.

Due to Provider's ease of modifiability but limitations in compile-time validation and scalability, its modifiability and scalability are rated as “medium”.

Testability

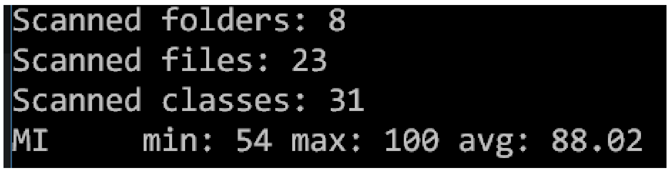
Testability is a key strength of the Provider approach in state management. Significant advantage is the clear separation of business logic and the user interface. This separation allows state management logic, such as ChangeNotifiers, Streams, or Futures, to be tested independently without requiring Flutter-specific dependencies. As a result, there’s no need to mock dependencies or rely on widget tests, which simplifies unit testing and makes it more straightforward. Additionally, when testing widgets that depend on state, mocking is kept minimal thanks to Provider’s abstraction. Specifically, "Value" providers, such as Provider<T>.value(), enable developers to inject simple mock classes that override the properties of the original state class. This method facilitates lightweight substitution of real state objects with test doubles, enhancing test flexibility. Due to Provider’s excellent testability across both business logic and UI layers, and because both widget tests and business logic tests can be implemented without requiring modifications to the source code, its testability is rated as “high”.

Time behavior: command “flutter test test/efficiency_bechmark_test.dart” returns output -



```
flutter test test/efficiency_bechmark_test.dart
00:03 +0: Efficiency test
cartButton: 3
userSwitch: 4
00:03 +1: All tests passed!
```

MI index: command “dcm calculate-metrics lib” returns output:


```
Scanned folders: 8
Scanned files: 23
Scanned classes: 31
MI     min: 54 max: 100 avg: 88.02
```

Learnability and Analyzability: evaluating Learnability and Analyzability for Provider involves considering several key factors. One notable aspect is that Provider introduces no new concepts, relying entirely on Flutter’s built-in widgets and state management components. It avoids external paradigms or complex architectural patterns, which makes it easier for developers already familiar with Flutter to adopt. However, this simplicity comes with drawbacks, particularly due to the lack of centralized business logic. With Provider’s use of InheritedWidgets, it becomes challenging to track where business logic resides. Rather than having a centralized state management layer, the logic is often dispersed across multiple StatefulWidgets, complicating efforts to trace it. This scattered approach also leads to boilerplate code overhead; furthermore, the increased widget nesting inherent in Provider’s design diminishes readability. Since InheritedWidgets are embedded within the widget tree and require explicit wrapping of child widgets, nesting is unavoidable, resulting in deeply nested structures. These complex hierarchies make the code harder to scan, modify, and debug. Due to Provider’s reliance on intricate widget structures, scattered logic, and mandatory nesting, its Learnability and Analyzability are rated as “medium”.

Documentation: documentation for Provider presents a mixed picture. One of its strengths lies in the fact that the individual Flutter widget types utilized by Provider, such as InheritedWidget and ChangeNotifier, are thoroughly documented in the official Flutter documentation. This provides a solid foundation for understanding the building blocks of the system. However, there are notable weaknesses, particularly the lack of comprehensive official documentation on how Provider functions as a cohesive state management system. While a few detailed examples exist to showcase real-world implementations of Provider-based state management, they are insufficient to fully clarify its application. Fortunately, this shortfall is partially offset by a wealth of third-party resources, including blogs, tutorials, and community guides, which explain how to use Provider and help bridge the documentation gap. Due to the well-documented nature of its individual components contrasted with the low volume of official guidance on Provider as a complete state management system, its documentation is rated as “medium.”

Resulting table:

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 3 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	medium

Modularity	quantitative	Maintainability index	88.02
Modifiability and Scalability	qualitative	low / medium / high	medium
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	medium

BLoC evaluation details

Business Logic Components are a widely used concept for state management. The fundamentals of this state management system were already explored in subsection 2.5.4.

Implementation

For the implementation of the app with BLoC, three Business Logic Components were created. These manage the login state, the contents of the shopping cart, and the available products. The BLoCs consist of multiple streams and sinks, which receive state changes from external sources and communicate them outward. In addition, public variables are provided for the streams, reflecting the current state. This serves with the initialData parameter, to ensure that widgets added to the widget tree later can receive the current state at the time of their initial creation, since streams can only communicate state changes.

All three BLoCs are passed to the user interface via a widget that functions as dependency injection. Other implementation variants, such as using the get_it service locator, would have been conceivable here as well. However, the chosen approach requires no additional library, thus minimizing any distortion of the results. The widgets in the user interface use StreamBuilder to stay informed about updated states.

Evaluation

The following section evaluates the implementation using BLoC (Ulvis, 2025, branch=bloc) based on the defined evaluation criteria.

Modifiability & Scalability

Regarding BLoC, it can be said that it is both scalable and modifiable. This statement is supported by the fact that communication exclusively via streams and sinks allows for the creation of a uniform interface to which any components can connect. Unlike widget-based approaches, BLoC approach can be used completely independently of the platform and is therefore well-suited for further scaling beyond the Flutter framework.

The coupling of different states can thus also be easily resolved by passing other BLoCs, for example, via the constructor or a dependency injection tool. Due to these properties, the rating “high” is assigned.

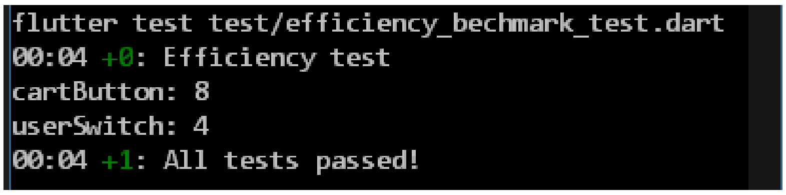
Testability

The business logic of individual BLoCs can be effectively tested, as the Flutter testing framework already provides tools for verifying streams, as used in the test in cart_bloc_test.dart. This allows for straightforward use of unit tests. However, if a BLoC has dependencies on other BLoCs or components, these must be mocked. Without an appropriate library, this introduces additional effort due to the need to implement mock classes.

Another aspect considered in testing is the testability of widgets that access BLoC. In this regard, it can be stated that replacing them with placeholders was possible without issues. However, this also depends on the injection system used. Additionally, it can be noted that to create a placeholder (i.e., mock) for a BLoC, the class of that BLoC must be fully mocked, several strategies can be utilized for this.

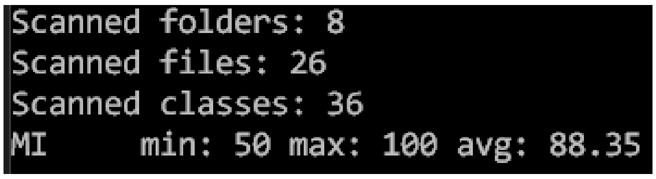
Due to the straightforward testability of the business logic and the ability to replace components with placeholders for widget tests, testability is rated as “high”.

Time behavior: command “flutter test test/efficiency_bechmark_test.dart” returns output -



```
flutter test test/efficiency_bechmark_test.dart
00:04 +0: Efficiency test
cartButton: 8
userSwitch: 4
00:04 +1: All tests passed!
```

MI index: command “dcm calculate-metrics lib” returns output -



```
Scanned folders: 8
Scanned files: 26
Scanned classes: 36
MI      min: 50 max: 100 avg: 88.35
```

Learnability and Analyzability: to use BLoC, it is necessary to engage with concepts of asynchronous programming to understand the functionality of streams and sinks. Therefore, it can be assumed that developers will first need to learn these concepts. The structure of the BLoC itself is difficult to follow, as the combination of streams, sinks, and variables is not immediately intuitive. Additionally, the interfaces to the user interface are not adapted to the natural flow of the language like other interfaces but are instead determined by the specific functions of streams and sinks. Moreover, this approach requires many classes to represent, for example, event arguments and states, and this further complicates readability.

Deep nesting of widgets, as seen in other approaches, is not observed with BLoC, since it does not rely on the widget system as its foundation. However, the use of streams requires the integration of StreamBuilder widgets in the user interface, as demonstrated in cart_button.dart. While this provides significant control over the processing of state changes on the one hand, it makes the source code harder to read on the other, especially when multiple StreamBuilder instances are nested within each other.

In summary, rating set as “low”.

Documentation for BLoC is challenging to evaluate since BLoC itself is merely a concept. However, there are libraries that enhance this concept with helper constructions, making the use of BLoC simpler. The documentation for these libraries includes extensive explanations and examples (Bloc Library, n.d., *Bloc Concepts*), which are also transferable to BLoC itself. Additionally, there is a large number of articles and other publications on the topic of BLoC. Due to the comprehensive documentation of the described libraries and the abundance of publications related to BLoC, its documentation is rated as “high”.

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 8 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	low
Modularity	quantitative	Maintainability index	88.35
Modifiability and Scalability	qualitative	low / medium / high	high
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	high

Riverpod evaluation details

Riverpod is an external state management library that builds upon and, according to its own claims, enhances the concept of Provider. The foundations for this are detailed in Subsection 2.5.5.

Implementation

Libraries riverpod 2.6.1 and flutter_riverpod 2.6.1 are used for implementation. The states are managed by various providers. For instance, the login state implemented using a StateNotifierProvider, which injects a StateNotifier class as explained in the fundamentals chapter. Additionally, a FutureProvider was employed to load the products, and a simple Provider was used to link the login state with the loaded products, thereby applying the discount. The cart state is represented through multiple providers. A StateNotifierProvider serves as the foundation for the cart, which is then supplemented by additional providers with further information, such as product data. All providers are made available to the user interface as global final variables.

Integration into the user interface was achieved using ConsumerWidget. To do this, the base classes of widgets that need to access states were changed to ConsumerWidget. This modification provides an additional ref parameter in the build method, allowing the providers to be accessed via their global variables.

Evaluation

In the following section, the implementation using Riverpod (Ulvis, 2025, branch=riverpod) is evaluated based on the defined evaluation criteria.

Modifiability and Scalability

Riverpod provides a uniform abstraction layer for states, capable of managing ChangeNotifier, StateNotifier, Futures, Streams, and simple values as states. This simplifies future modifications since only the respective states and providers need to be adjusted, without requiring changes to dependent states or other components.

Riverpod does not rely on injections via generics but instead uses variables. This approach offers the advantage that multiple states of the same data type can coexist, ensuring that a corresponding provider is always present within the managed overall state.

Dependencies between providers are specified using an array, indicating which other providers a given provider relies on. These states can then be subscribed to, modified, or read via a reference variable (ref) in the same way as when used in a ConsumerWidget. As a result, this coupling approach scales effectively with the growing number of states to be linked and is not constrained by generics, so due to that - scalability and modifiability rated as “fully met.”

Testability

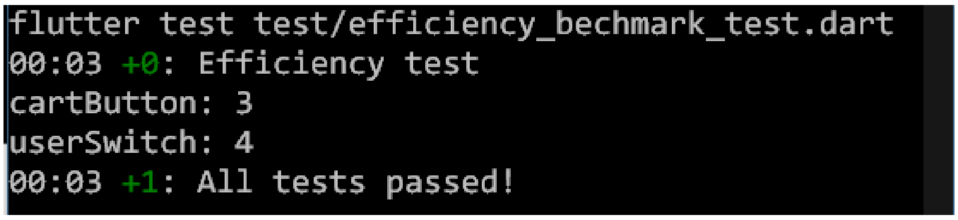
To assess testability, two aspects are examined: the extent to which the business logic of individual state classes can be tested, and the feasibility of replacing distributed states with placeholders during widget tests.

Testing the business logic poses no issues. These tests can be implemented without relying on Riverpod or Flutter, as the state classes retain their business logic independently of the state management system. Consequently, a straightforward unit test can be implemented to verify the functionality of a class or its methods.

In widget tests, Riverpod provides the ability to override any number of providers, either with fixed content or with other providers. The option to override with fixed content greatly simplifies replacing states with placeholders.

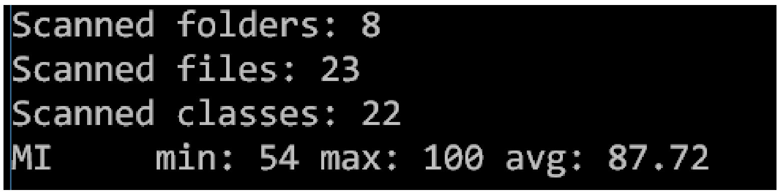
Therefore, testability is rated as “fully met.”

Time behavior: command “flutter test test/efficiency_bechmark_test.dart” returns output -



```
flutter test test/efficiency_bechmark_test.dart
00:03 +0: Efficiency test
cartButton: 3
userSwitch: 4
00:03 +1: All tests passed!
```

MI index: command “dcm calculate-metrics lib” returns output:



```
Scanned folders: 8
Scanned files: 23
Scanned classes: 22
MI      min: 54 max: 100 avg: 87.72
```

Learnability and Analyzability

Several aspects are considered in assessing understandability and readability.

Unlike Provider, Riverpod introduces a new syntax for injecting providers, which slightly deviates from the methodology commonly used in Flutter. As previously described, the base class of widgets accessing states must be changed to ConsumerWidget, and states are accessed via the reference variable ref. However, whether this alone is sufficient to claim that Riverpod introduces concepts divergent from Flutter is questionable. The use of the ref variable simplifies readability by eliminating the need for Provider.of<XY>() calls.

Using variables instead of generics to retrieve providers enhances understandability, as the variable name directly indicates its purpose. This also allows multiple providers of the same data type to coexist, eliminating the need for wrapper classes that increase complexity and hinder readability.

The newly introduced StateNotifier concept offers no clear advantages from a readability or understandability perspective. It separates state-changing functions and state content into different classes, and modifying the state requires setting a new instance of the state object (state), which can initially seem confusing. The benefits of StateNotifier lie more in performance improvements, as it checks whether the state variable itself has actually changed with each update.

Conversely, the ability to inject other states via the ref variable improves readability. Compared to ProxyProvider, it immediately clarifies why a particular provider needs to be read or subscribed to.

Riverpod avoids the issue of deep nesting by not requiring providers to be embedded in the widget tree. Additionally, alongside StateNotifier, it supports the use of ChangeNotifier as in Provider, mitigating the previously mentioned criticism of StateNotifier. As a result, this criterion is rated as “fully met.”

Documentation

The Riverpod documentation (Riverpod, n.d.) covers all fundamental concepts and includes various practical examples. It also documents how automated tests can be implemented. There are a lot of examples available as complete applications, including third-party. So the documentation is rated as “fully met.”

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 3 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	high
Modularity	quantitative	Maintainability index	87.72
Modifiability and Scalability	qualitative	low / medium / high	high
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	high

GetX evaluation details

GetX is a state management, dependency injection, and route management package for Flutter that simplifies application development with high-performance reactivity and minimal boilerplate code. The foundations for GetX are detailed in Subsection 2.5.6

Implementation

For the implementation of the sample application with GetX, the get package version 4.7.2 was used. The state is managed through three primary controllers:

- UserController: manages the login state using an observable boolean (isSignedIn). It provides a simple method (updateSignInStatus) to toggle the login state, which triggers reactive updates across the app.
- ProductController: handles the product list state using an observable productsState of type ProductsState. It integrates with the UserController to apply a 20% discount to product prices

when the user is signed in, leveraging reactive updates via the listen method. Products are fetched asynchronously from the ProductService.

- CartController: manages the shopping cart state with an observable map (_cart) tracking products and their quantities, alongside computed observables for totalPrice and numberOfProducts. It reacts to changes in the ProductController's productsState using the "ever" method to update prices dynamically (e.g., applying discounts).

Integration into the UI is achieved using GetX's reactive widgets (Obx and GetX<T>). Controllers are instantiated globally via Get.put() in the app's entry point (app.dart), enabling context-free access throughout the widget tree. The ProductListScreen uses Obx to display the product list reactively, while the CartScreen leverages Obx for the cart items and total price, and the ProductItem widget uses GetX<CartController> to update quantities reactively.

Evaluation

In the following section, the implementation using GetX (Ulvis, 2025, branch=getx) is evaluated based on the defined evaluation criteria.

Modifiability and Scalability

GetX offers significant advantages in modifiability and scalability due to its reactive and modular design. The use of observable variables (via .obs) allows state changes to propagate automatically, reducing the need for manual updates when adding new features. Controllers can be extended or replaced without altering the widget tree, as they are injected globally via Get.put() or lazily via Get.lazyPut(), supporting dependency injection without widget-level dependencies. This flexibility simplifies modifications, such as adding new state variables or integrating additional controllers.

Scalability is enhanced by GetX's ability to couple states efficiently. For instance, the CartController listens to the ProductController's productsState using "ever", enabling seamless updates when discounts change due to login status. This reactive approach scales well with increasing state complexity, as new controllers can be added and interconnected via observables without significant refactoring. However, as applications grow, the lack of strict architectural guidelines (unlike BLoC or Redux) might lead to less predictable state management in very large projects unless developers enforce their own structure.

Given GetX's ease of modification and strong scalability through reactive programming, modifiability and scalability are rated as "high".

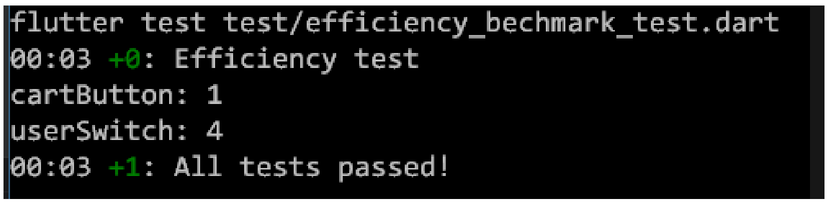
Testability

GetX's testability is robust for business logic but requires some consideration for widget testing. The controllers (UserController, ProductController, CartController) encapsulate business logic independently of the Flutter framework, relying solely on Dart and GetX's reactive system. This allows unit tests (e.g., cart_controller_test.dart) to verify methods like increaseQuantity or decreaseQuantity without Flutter dependencies, using standard Dart testing tools.

For widget tests, GetX provides utility like Get.put() to inject mock controllers, simplifying the replacement of real state with placeholders. However, the reactive nature of Obx and GetX<T> widgets means tests must account for asynchronous updates, potentially requiring await tester.pump() calls to ensure state propagation. While this is manageable, it adds slight complexity compared to non-reactive systems like Provider.

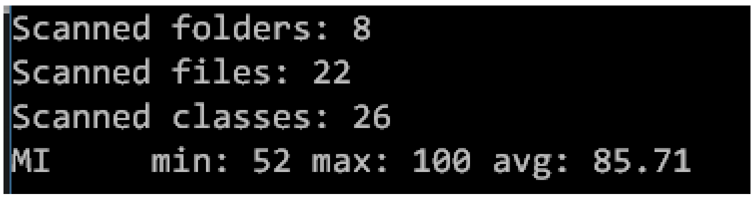
Due to the ease of testing business logic and the ability to mock states for widget tests with minimal overhead, testability is rated as “high”.

Time behavior: command “flutter test test/efficiency_bechmark_test.dart” returns output -


```
flutter test test/efficiency_bechmark_test.dart
00:03 +0: Efficiency test
cartButton: 1
userSwitch: 4
00:03 +1: All tests passed!
```

GetX demonstrates excellent time behavior, particularly in minimizing widget rebuilds. The cartButton rebuilds only once, matching the baseline, as Obx ensures it updates solely when numberOfProducts changes, avoiding unnecessary rebuilds from unrelated state changes (e.g., product list updates).

MI index: command “dcm calculate-metrics lib” returns output:


```
Scanned folders: 8
Scanned files: 22
Scanned classes: 26
MI      min: 52 max: 100 avg: 85.71
```

Learnability and Analyzability

GetX introduces reactive programming concepts (e.g., .obs, Rx, Obx), which may require developers unfamiliar with Rx paradigms to invest time in learning. However, it builds on Flutter’s widget system without mandating deep nesting or complex architectural patterns, aligning closely with Flutter’s declarative style. The use of controllers as centralized state holders enhances analyzability, as business logic is clearly separated from UI code (e.g., CartController handles cart logic independently of CartScreen).

The reactive syntax (e.g., Obx(() => ...)) simplifies state updates compared to manual setState calls, reducing boilerplate and improving readability. However, tracing state changes across multiple ever listeners or nested Obx widgets can become less intuitive in complex scenarios. Deep nesting is avoided, as GetX does not rely on widget-tree-based state injection, instead using global controller access. Overall, GetX balances simplicity with power, earning a “high” rating for learnability and analyzability, though mastery of reactive programming may pose an initial hurdle.

Documentation

GetX’s official documentation (GetX, n.d.) is comprehensive, covering state management, dependency injection, and routing with detailed explanations and examples. It includes practical snippets (e.g., counter apps) and API references, making core concepts accessible. However, some advanced reactive features (e.g., ever, debounce) lack in-depth tutorials, requiring developers to experiment or consult community resources. The growing GetX community supplements this with extensive third-party tutorials, blogs, and GitHub examples, though it doesn’t yet match the volume of resources for older libraries like Provider or BLoC.

Due to its thorough official documentation and strong community support, documentation is rated as “high”.

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 1 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	high
Modularity	quantitative	Maintainability index	85.71
Modifiability and Scalability	qualitative	low / medium / high	high
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	high

Redux evaluation details

Redux represents an external library for state management, built on approaches originating from the React ecosystem. The fundamentals of this are explained in subsection 2.5.7.

Implementation

The libraries redux version 5.0.0 and flutter_redux version 0.10.0 were utilized for the implementation with Redux. Instead of managing multiple states, a centralized state, referred to as the "store," was implemented. To make changes to this store, various actions were defined, such as an action for the login process. These actions are processed by reducers, which in this case always modify a portion of the centralized state. Middleware was implemented to fetch data for the product list. This middleware receives all actions from the library before they are handled by a reducer. When a corresponding action is triggered, the middleware performs the network requests and subsequently emits appropriate actions, which the reducers then use to transfer the data into the store. Middlewares are necessary because reducers are not intended to handle asynchronous operations; they must remain free of side effects and consistently produce the same output for the same input (Garreau, 2018, section 1.3.3). For integration into the user interface, two different approaches are employed. First, the StoreConnector selects a specific part of the centralized state and passes it to a widget. Second, StoreBuilder provides the entire store to a widget. The StoreConnector is primarily used in simpler scenarios that only require reading a variable from the store, while the StoreBuilder is used in more complex cases, such as those that also need to dispatch actions.

Evaluation

In the following section, the implementation using Redux (Ulvis, 2025, branch=redux) is evaluated based on the defined evaluation criteria.

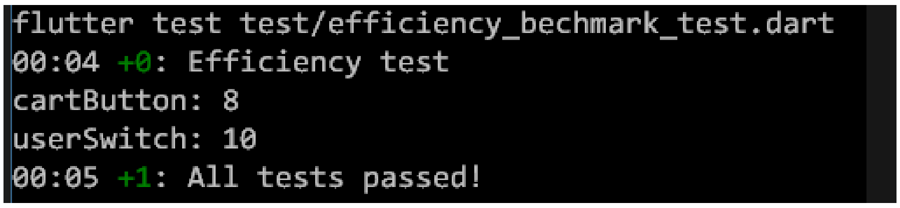
Modifiability and Scalability

Modifiability and scalability in Redux can be described based on various aspects. One challenge for scalability is the centralized state, which consists of a single class intended to represent the entire state of the app. In large applications, this concept could reach its limits, particularly when an application is composed of multiple modules. Examining the interconnection of multiple states is not applicable to Redux, as it relies solely on a single centralized state. However, it must be noted that this also means reducers have to take on many tasks that are not immediately obvious, stemming from the lack of an option to have multiple interdependent states. To better structure reducers, the library offers the ability to group reducers for specific aspects of the state, allowing them to address only a particular subset of the state. In summary, scalability and modifiability are rated as “low”.

Testability

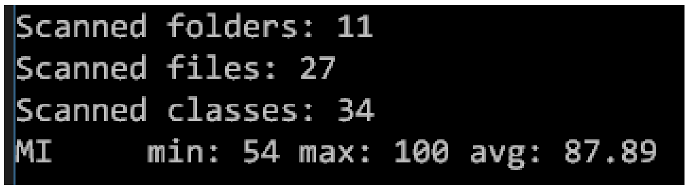
Testability is evaluated based on both tests of the business logic and tests of widgets that access the store. For testing the business logic, it is sufficient to manually create the store and dispatch the desired action, after which the central state can be checked to verify whether the expected outcome has occurred. However, testing the middlewares proves to be challenging, as they involve asynchronous, non-blocking actions. This would require waiting for the desired action without knowing whether it will ever actually be dispatched. The central state itself should not contain business logic and can therefore be easily replaced with a placeholder during widget tests without issues. Due to the difficulties in testing the middlewares, testability is rated as “medium”.

Time behavior: command “flutter test test/efficiency_bechmark_test.dart” returns output -



```
flutter test test/efficiency_bechmark_test.dart
00:04 +0: Efficiency test
cartButton: 8
userSwitch: 10
00:05 +1: All tests passed!
```

MI index: command “dcm calculate-metrics lib” returns output:



```
Scanned folders: 11
Scanned files: 27
Scanned classes: 34
MI      min: 54 max: 100 avg: 87.89
```

Learnability and Analyzability

Learnability and analyzability are assessed based on the defined aspects. Redux introduces a completely new concept to the Flutter ecosystem, originating from the React ecosystem. Terms like reducers, actions, and middlewares, which play a minor role in Flutter outside of Redux, require developers to familiarize themselves with this concept before use, which does not aid comprehensibility. On the other hand, Redux provides a clear structure through its distinct separation into actions, reducers, middlewares, and state, making the source code readable due to relatively small methods. However, it may be questioned whether this readability can be sustained in the long term with a theoretically ever-growing central store, as this class would expand alongside the application. The issue of deep nesting does not apply to Redux, as it relies on a state management mechanism outside the widget tree, with the store being injected at a single central point in the widget tree. Ultimately, learnability and analyzability are rated as “medium”.

Documentation

The documentation (Redux.dart, n.d) for Redux outlines the fundamental concepts and refers to the Redux documentation for React for more detailed explanations. Additionally, it includes detailed application examples. Beyond the official documentation, there is a large number of publications on Redux for React, which are largely applicable to the Flutter implementation of Redux. Consequently, the documentation is rated as “high”.

Characteristic	Metric	Measurement value	Result / Amount
Time behavior	quantitative	widgets rebuild count	cartButton: 8 (base: 1) userSwitch: 10 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	medium
Modularity	quantitative	Maintainability index	87.89
Modifiability and Scalability	qualitative	low / medium / high	low

Testability	qualitative	low / medium / high	medium
Documentation	qualitative	low / medium / high	high

MobX evaluation details

MobX, much like Redux, is a library based on approaches from the React ecosystem. The fundamentals of this are covered in subsection 2.5.8.

Implementation

For the implementation with MobX, the libraries mobx version 2.5.0 and flutter_mobx version 2.3.0 were utilized. For state management, four so-called stores were implemented. These included a partial state, as well as computed values and actions to modify the state. Specifically, a store was implemented for the login state, the product list, the shopping cart, and the quantity of a product in the cart. The stores are supplemented by automatically generated source code using mobx_codegen library. The stores are passed into the widget tree via an InheritedWidget, although any other dependency injection solutions could also be used for this purpose. Access to the states is achieved within an Observer widget.

Evaluation

In the following section, the implementation using MobX (Ulvis, 2025, branch=mobx) is evaluated based on the defined evaluation criteria.

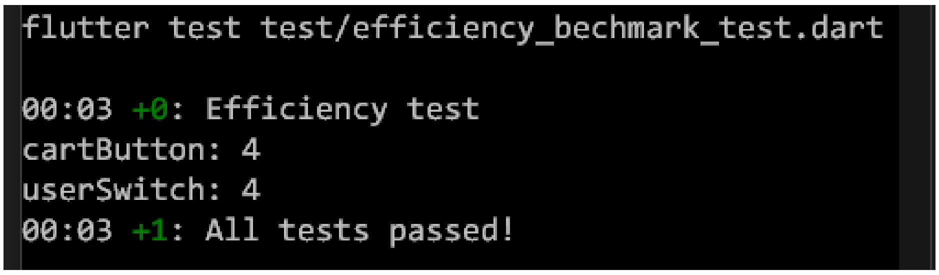
Modifiability and Scalability

It can be stated that no obstacles to scalability were identified. Multiple states can be coupled with ease, as simple access to observable variables is sufficient to receive state updates. As for modifiability, while there is no abstraction layer like that found in Provider or Riverpod, later changes can be balanced out using computed values, for example, thereby ensuring backward compatibility. Due to the straightforward coupling of states and its modifiability, rated here as “high”.

Testability

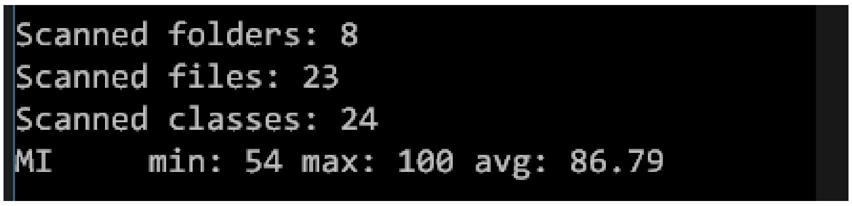
Testability is assessed based on two aspects: the extent to which the business logic can be tested and how well states can be replaced with placeholders during widget tests. The tests for the business logic could be implemented without additional measures, as externally there exists a class with variables and functions that can be tested using standard unit testing tools. For widget tests, it is necessary to replace the store class with new placeholder classes that inherit from the store classes. This approach allows stores to be successfully substituted with trivial placeholders. Consequently, testability is rated as “high”.

Time behavior: command “flutter test test/efficiency_bechmark_test.dart” returns output -



```
flutter test test/efficiency_bechmark_test.dart

00:03 +0: Efficiency test
cartButton: 4
userSwitch: 4
00:03 +1: All tests passed!
```

MI index: command “dcm calculate-metrics lib” returns output:



```
Scanned folders: 8
Scanned files: 23
Scanned classes: 24
MI     min: 54 max: 100 avg: 86.79
```

Learnability and Analyzability

It can be said that MobX pursues an approach here that contributes to readability, as the use of annotations like @observable, @computed and @action - supports the flow of language while reading. Additionally, the state classes only contain the truly necessary business logic, and procedures for state management are outsourced through source code generation. This makes the structure comprehensible and clear. However, source code generation can also pose a risk, as errors or issues in the generated source code are difficult to trace, and the generated source code itself is hard to read.

MobX primarily uses existing concepts from the Flutter ecosystem and merely extends them with the previously mentioned annotations, which, however, already provide insight into their function through their naming. Deep nesting is avoided in MobX by not relying on a widget-based approach for state management, thus eliminating the need for injection into the widget tree. Learnability and analyzability are rated as “high”, though the risk of source code generation should be taken into account when making a decision.

Documentation

The documentation (MobX.dart, n.d.) of MobX describes the basic concepts of the state management system and supplements them with extensive application examples. In addition to the official documentation, there are also various third-party publications on MobX, and the documentation for the JavaScript variant of MobX can also be partially adapted for Dart. The documentation is therefore rated as “high”.

Characteristic	Metric	Measurement value	Result / Amount

Time behavior	quantitative	widgets rebuild count	cartButton: 4 (base: 1) userSwitch: 4 (base: 2)
Learnability and Analyzability	qualitative	low / medium / high	high
Modularity	quantitative	Maintainability index	86.79
Modifiability and Scalability	qualitative	low / medium / high	high
Testability	qualitative	low / medium / high	high
Documentation	qualitative	low / medium / high	high

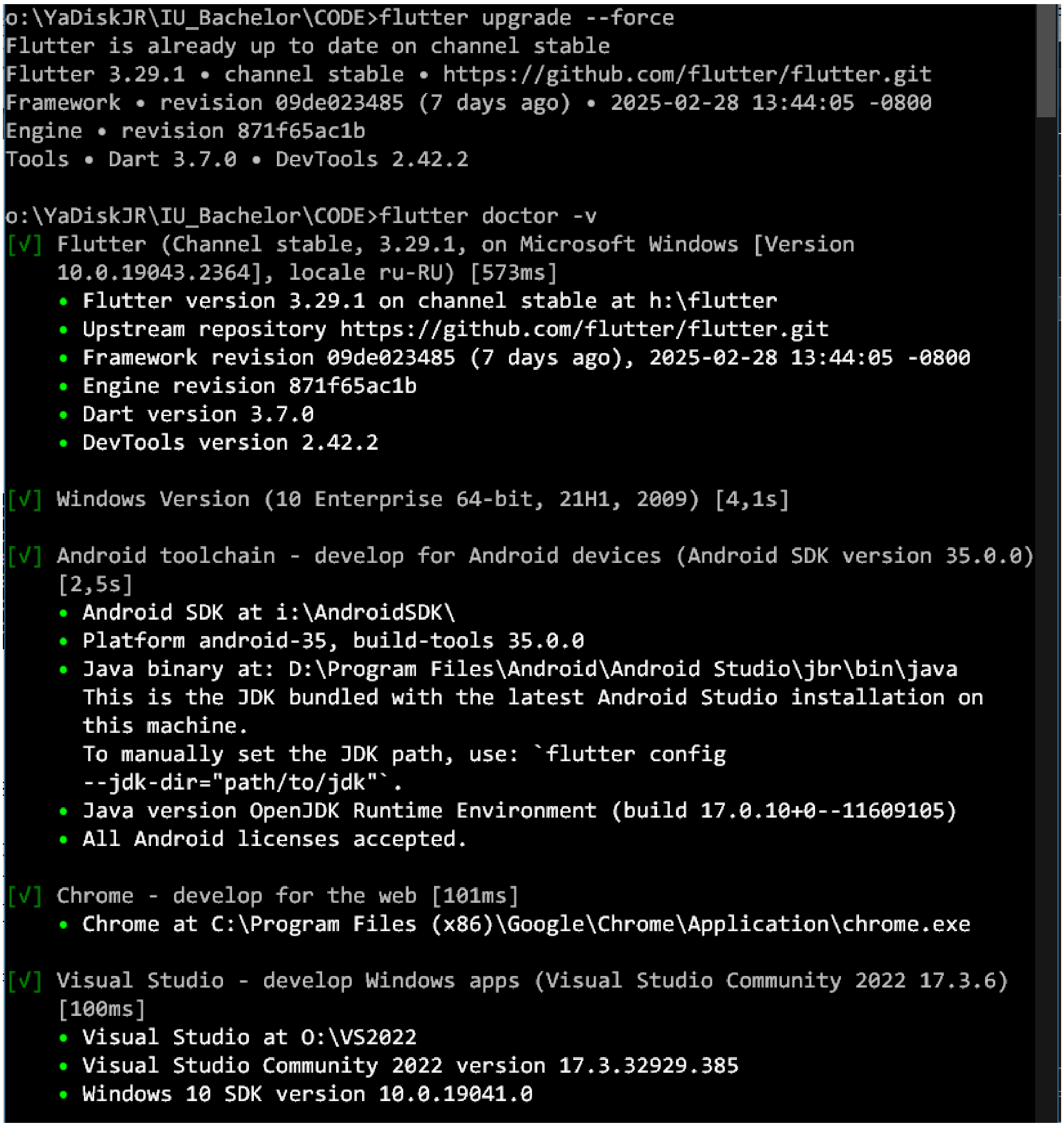
Maintainability Index calculation

Previously, the dart_code_metrics package was utilized for various analytical tasks in Flutter applications. Since June 2023 that package was marked it as outdated and the maintainer`s focus has shifted to the dcm.dev utility, which has different pricing options, including Free option. DCM installation requires CLI downloading and adding to PATH (on Windows), Free license obtaining and VS Code extension installation with license registration. Being installed and configured, terminal command “dcm calculate-metrics lib” in project root – returns console output as listed below:

```
======================

\iu_bachelor_thesis>dcm calculate-metrics lib

✓ Calculation is completed. Preparing the results: 1.1s

lib\app.dart:

  • method IUBachelorThesisApp.build (1 entry):

    HIGH     This method has a Halstead volume of 205.0, which exceeds the threshold of 150.0.

          halstead-volume : https://dcm.dev/docs/metrics/function/halstead-volume

lib\screens\cart\cart_item_list.dart:

  • method CartItemList.build (1 entry):

    HIGH     This method has a Halstead volume of 177.87213211613133, which exceeds the threshold of 150.0.

          halstead-volume : https://dcm.dev/docs/metrics/function/halstead-volume

lib\screens\product_list\product_item.dart:

  • method ProductItem.build (1 entry):

    HIGH     This method has a Halstead volume of 174.22857502740396, which exceeds the threshold of 150.0.

          halstead-volume : https://dcm.dev/docs/metrics/function/halstead-volume

lib\store\cart_store.dart:

  • method CartStore.updateProductList (1 entry):

    HIGH     This method has a Halstead volume of 222.90509710918678, which exceeds the threshold of 150.0.

          halstead-volume : https://dcm.dev/docs/metrics/function/halstead-volume
  • method CartStore._addAmount (1 entry):

    HIGH      This method has a Halstead volume of 293.43760004615405, which exceeds the threshold of 150.0.

              halstead-volume : https://dcm.dev/docs/metrics/function/halstead-volume

Scanned folders: 8

Scanned files: 23

Scanned classes: 31

MI    min: 54 max: 100 avg: 88.02

======================
```

From that console output – MI avg value is used as Maintainability Index metric value.

Flutter info



```
o:\YaDiskJR\IU_Bachelor\CODE>flutter upgrade --force
Flutter is already up to date on channel stable
Flutter 3.29.1 • channel stable • https://github.com/flutter/flutter.git
Framework • revision 09de023485 (7 days ago) • 2025-02-28 13:44:05 -0800
Engine • revision 871f65ac1b
Tools • Dart 3.7.0 • DevTools 2.42.2

o:\YaDiskJR\IU_Bachelor\CODE>flutter doctor -v
[√] Flutter (Channel stable, 3.29.1, on Microsoft Windows [Version
    10.0.19043.2364], locale ru-RU) [573ms]
    • Flutter version 3.29.1 on channel stable at h:\flutter
    • Upstream repository https://github.com/flutter/flutter.git
    • Framework revision 09de023485 (7 days ago), 2025-02-28 13:44:05 -0800
    • Engine revision 871f65ac1b
    • Dart version 3.7.0
    • DevTools version 2.42.2

[√] Windows Version (10 Enterprise 64-bit, 21H1, 2009) [4,1s]

[√] Android toolchain - develop for Android devices (Android SDK version 35.0.0)
    [2,5s]
    • Android SDK at i:\AndroidSDK\
    • Platform android-35, build-tools 35.0.0
    • Java binary at: D:\Program Files\Android\Android Studio\jbr\bin\java
      This is the JDK bundled with the latest Android Studio installation on
      this machine.
      To manually set the JDK path, use: `flutter config
      --jdk-dir="path/to/jdk"`.
    • Java version OpenJDK Runtime Environment (build 17.0.10+0--11609105)
    • All Android licenses accepted.

[√] Chrome - develop for the web [101ms]
    • Chrome at C:\Program Files (x86)\Google\Chrome\Application\chrome.exe

[√] Visual Studio - develop Windows apps (Visual Studio Community 2022 17.3.6)
    [100ms]
    • Visual Studio at O:\VS2022
    • Visual Studio Community 2022 version 17.3.32929.385
    • Windows 10 SDK version 10.0.19041.0
```

Code listings

Listing 1. Structure of a simple Flutter widget in Dart.

```
import 'package:flutter/material.dart';
class ExampleWidget extends StatelessWidget {
  const ExampleWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return const Column(
      children: [
        Text('First row'),
        Text('Second row'),
      ],    );  }}
```

Listing 2. Example of StatefulWidget.

```
class Counter extends StatefulWidget {
  const Counter({Key? key}) : super(key: key);
  @override
  State<Counter> createState() => _CounterState();
}
class _CounterState extends State<Counter> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++;
    });  }
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text("Count: $_counter"),
        IconButton(
          icon: Icon(Icons.add),
          onPressed: _incrementCounter,
        ),      ],    );  }}
```

Listing 3. Using the setState method within a StatefulWidget.

```
import 'package:flutter/material.dart';
class MyCounter extends StatefulWidget {
  @override
  _MyCounterState createState() => _MyCounterState();}
class _MyCounterState extends State<MyCounter> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++; // Update the state and trigger a rebuild.
    });  }
  @override
  Widget build(BuildContext context) {
    return Column(      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text('Counter: $_counter'),
        ElevatedButton(
          onPressed: _incrementCounter,
          child: Text('Increment'),
        ),      ],    );  }   }
```

Listing 4. Usage of InheritedWidget.

```
class MyAppState extends InheritedWidget {
  final int counter;
  final VoidCallback increment;
  const MyAppState({
    Key? key, required Widget child, required this.counter, required this.increment,
  }): super(key: key, child: child);
  // Static method to access the shared state from descendant widgets.
  static MyAppState? of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<MyAppState>();  }
  @override
  bool updateShouldNotify(MyAppState oldWidget) {
    return counter != oldWidget.counter;  }}
```

Listing 5. Example of InheritedWidget usage.

Code explanation:

1. Stateful root widget (MyApp):
   - The MyApp widget holds the mutable state (_counter) and the _incrementCounter function.
   - Its build method wraps the MaterialApp with an AppState widget, which is our custom InheritedWidget.
2. Custom InheritedWidget (AppState):
   - AppState takes two parameters: the current counter value and an increment callback.
   - The static method of(BuildContext context) allows descendant widgets to access the shared state.
   - The updateShouldNotify method returns true when the counter value changes, signaling Flutter to rebuild any widgets that depend on this inherited state.
3. Descendant widget (HomeScreen):
   - In the HomeScreen, we access the shared state via AppState.of(context).
   - The current counter value is displayed, and the ElevatedButton calls the increment method to update the state.

This example demonstrates how InheritedWidget can be used to propagate state information down the widget tree efficiently. Widgets that call AppState.of(context) will automatically rebuild when updateShouldNotify returns true - in this case, when the counter changes. This pattern is particularly useful for sharing common data among many widgets without having to pass parameters explicitly through every intermediate widget.

Code:

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

void main() {

  runApp(const MyApp());

}

/// The root widget of the application.

/// This is a StatefulWidget because it holds the state (counter) that will be shared.

class MyApp extends StatefulWidget {

  const MyApp({Key? key}) : super(key: key);

  @override

  _MyAppState createState() => _MyAppState();

}

class _MyAppState extends State<MyApp> {

  int _counter = 0;

  /// Method to increment the counter.

  void _incrementCounter() {

    setState(() {

      _counter++;

    });  }

    @override

  Widget build(BuildContext context) {

    // Wrap the MaterialApp with our custom InheritedWidget (AppState)

    return AppState(

      counter: _counter,

      increment: _incrementCounter,

      child: MaterialApp(

        title: 'InheritedWidget Demo',
        home: const HomeScreen(),
      ),    );  }}
/// An InheritedWidget that provides the shared state (counter) and the increment function.
class AppState extends InheritedWidget {
  final int counter;
  final VoidCallback increment;
  const AppState({
    Key? key,
    required this.counter,
    required this.increment,
    required Widget child,
  }) : super(key: key, child: child);
  /// Static helper method to easily access the AppState from descendant widgets.
  static AppState? of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType<AppState>();
  }
  /// Determines whether the widgets that depend on this InheritedWidget should rebuild
  /// when the data changes. In this case, we rebuild if the counter value has changed.
  @override
  bool updateShouldNotify(AppState oldWidget) {
    return counter != oldWidget.counter;
  }
}
/// The home screen widget that displays the counter and a button to increment it.
class HomeScreen extends StatelessWidget {
  const HomeScreen({Key? key}) : super(key: key);
    @override
```

```
Widget build(BuildContext context) {
  // Access the shared state using the static 'of' method.
  final appState = AppState.of(context);
  // Fallback values in case appState is null (should not happen in this example).
  final counter = appState?.counter ?? 0;
  final increment = appState?.increment;
  return Scaffold(
    appBar: AppBar(
      title: const Text('InheritedWidget Example'),
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Text(
            'Current Counter Value:',
            style: Theme.of(context).textTheme.headline6,
          ),
          Text(
            '$counter',
            style: Theme.of(context).textTheme.headline2,
          ),
          const SizedBox(height: 20),
          ElevatedButton(
            onPressed: increment,
            child: const Text('Increment Counter'),
          ),          ],        ),      ),    );  }}
```

Listing 6. Example of ValueNotifier and AnimatedBuilder usage.

Code explanation:

1. ValueNotifier initialization:
   - The ValueNotifier<int> counter = ValueNotifier<int>(0); initializes a notifier with an initial value of 0.
   - ValueNotifier is a special kind of ChangeNotifier that holds a single value. When you update counter.value, it automatically notifies its listeners about the change.
2. AnimatedBuilder Widget:
   - AnimatedBuilder is used to listen to the ValueNotifier (or any Listenable object) and rebuild its child widget when the notifier's value changes.
   - In the builder function, the current counter value is displayed using a Text widget. Each time counter.value changes, the builder function is triggered, updating the text.
3. FloatingActionButton:
   - The floating action button calls the _incrementCounter function when pressed.
   - The _incrementCounter function increments the counter’s value by updating counter.value++. This triggers the AnimatedBuilder to rebuild and update the UI with the new counter value.

ValueNotifier and AnimatedBuilder work together to create a reactive UI component without needing additional state management libraries.

Code:

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

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

class MyApp extends StatelessWidget {

  // Create a ValueNotifier that holds an integer value.

  final ValueNotifier<int> counter = ValueNotifier<int>(0);

  MyApp({Key? key}) : super(key: key);

  // Function to increment the counter.

  void _incrementCounter() {
    counter.value++;  }
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'ValueNotifier & AnimatedBuilder Example',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('ValueNotifier & AnimatedBuilder Example'),        ),
        body: Center(
          // AnimatedBuilder listens to the ValueNotifier (counter)
          // and rebuilds its child widget whenever the value changes.
          child: AnimatedBuilder(
            animation: counter,
            builder: (context, child) {
              return Text(
                'Counter: ${counter.value}',
                style: Theme.of(context).textTheme.headline4,
              );            },          ),        ),
        floatingActionButton: FloatingActionButton(
          onPressed: _incrementCounter,
          tooltip: 'Increment',
          child: const Icon(Icons.add),
        ),      ),    );  }}
```

Listing 7. Example of StreamBuilder usage.

Code explanation:

1. Creating the Stream:

    - The counterStream function is an asynchronous generator that yields an incremented integer value every second.
    - This stream continuously emits values which are then consumed by the StreamBuilder.
2. Using StreamBuilder:
    - The StreamBuilder<int> widget listens to counterStream().
    - Its builder callback receives an AsyncSnapshot<int> containing the current state of the stream.
    - Depending on the connection state:
        - If waiting for data (ConnectionState.waiting), a CircularProgressIndicator is shown.
        - If there is an error (snapshot.hasError), the error is displayed.
        - If data is available (snapshot.hasData), it displays the current counter value.
        - Otherwise, a fallback text is provided.
3. Reactive UI Updates:
    - Each time the stream emits a new value, the StreamBuilder rebuilds its child widget with the updated data, providing a reactive and seamless user experience.

This example demonstrates the power of StreamBuilder for creating dynamic UIs that respond to real-time data changes in Flutter applications.

Code:

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

import 'dart:async';

void main() {

  runApp(const MyApp());}

/// The root widget of the application.

class MyApp extends StatelessWidget {

  const MyApp({Key? key}) : super(key: key);

  // A simple stream that emits an increasing integer every second.

  Stream<int> counterStream() async* {
```

```
  int counter = 0;

  while (true) {

    await Future.delayed(const Duration(seconds: 1));

    yield counter++;

  }  }

@override

Widget build(BuildContext context) {

  return MaterialApp(

    title: 'StreamBuilder Example',

    home: Scaffold(

      appBar: AppBar(

        title: const Text('StreamBuilder Example'),        ),

      body: Center(

        // Use StreamBuilder to listen to the counterStream.

        child: StreamBuilder<int>(

          stream: counterStream(),

          builder: (BuildContext context, AsyncSnapshot<int> snapshot) {

            // Check the connection state and display appropriate widgets.

            if (snapshot.connectionState == ConnectionState.waiting) {

              // While waiting for data, show a loading indicator.

              return const CircularProgressIndicator();

            } else if (snapshot.hasError) {

              // If an error occurs, display the error message.

              return Text('Error: ${snapshot.error}');

            } else if (snapshot.hasData) {

              // If data is available, display the current counter value.

              return Text(
                'Counter: ${snapshot.data}',
                style: Theme.of(context).textTheme.headline4,                );
            } else {
              // Fallback widget in case there's no data.
              return const Text('No data available');
            }           },          ),        ),      ),    );  }}
```

Listing 8. Example of ChangeNotifier class usage.

```
class CounterModel with ChangeNotifier {
  int _count = 0;
  int get count => _count;
  void increment() {
    _count += 1;
    notifyListeners();  }}
…
FloatingActionButton(
          onPressed: _counter.increment,
          child: const Icon(Icons.add),        ),
…
          ListenableBuilder(
            listenable: counterNotifier,
            builder: (BuildContext context, Widget? child) {
              return Text('${counterNotifier.count}');    },
```

Listing 9. Use a ChangeNotifier as a state management solution with Provider

```
class AnimalStore extends ChangeNotifier {
  var animal;
  AnimalStore(this.animal);

  void updateAnimal(newAnimal) {

    this.animal = newAnimal;

    notifyListeners();

  }}

class ProvidingWidget extends StatelessWidget {

        final animal;

        const ProvidingWidget({Key? key, this.animal}) : super(key: key);

        @override

        Widget build(BuildContext context) {

                return ChangeNotifierProvider(create: (context) => AnimalStore(animal),

                        child: ConsumingWidget());   }}

class ConsumingWidget extends StatelessWidget {

        const ConsumingWidget({Key? key}) : super(key: key);

        @override

        Widget build(BuildContext context) {

                final animal = Provider.of<AnimalStore>(context).animal;

                return Text(animal!.nickName);        }}
```

Listing 10. State class in Riverpod

```
class AnimalStore extends StateNotifier<Animal> {

        AnimalStore(Animal initialState) : super(initialState);

        changeNickname(String newNickName) {

                state = Animal(newNickName);        }}
```

Listing 11. Consumer widget

```
final animalStoreProvider = StateNotifierProvider((ref) => AnimalStore());
```

```
class ConsumingWidget extends ConsumerWidget {
        @override
        Widget build(BuildContext context, WidgetRef ref) {
                final animalStore = ref.watch(animalStoreProvider);
                return Text(animalStore?.state.nickName ?? ‘’);     }}
```

Listing 12. Simple state management with GetX and Obx.

```
class CounterController extends GetxController {
  // Here, 'count' is defined as an observable variable with an initial value of 0.
  var count = 0.obs;
  // The 'increment' method increases 'count' by 1.
  void increment() => count++;
}
class CounterWidget extends StatelessWidget {
  // 'Get.put()' is used to create or retrieve an instance of CounterController,
  // managing its lifecycle automatically.
  final CounterController controller = Get.put(CounterController());
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        // 'Obx' listens for changes in 'count'. When 'count' changes,
        // the Text widget rebuilds to show the new value.
        Obx(() => Text('Count: ${controller.count.value}')),
        // This button, when pressed, calls the 'increment' method on the controller,
        // which in turn updates the observable 'count'.
        ElevatedButton(
          onPressed: controller.increment,
          child: Text('Increment'),
        ),      ],    );  }}
```

Listing 13. Using GetBuilder for more control.

```
class CounterController extends GetxController {
  // 'count' is a simple integer, not automatically observable.
  int count = 0;
  void increment() {
    // Increment 'count' and call 'update()' to manually notify listeners
    // to rebuild the UI.
    count++;
    update();
  }}
class CounterWidget extends StatelessWidget {
  // Similar to before, 'Get.put()' manages the instance of CounterController.
  final CounterController controller = Get.put(CounterController());
  @override
  Widget build(BuildContext context) {
    return GetBuilder<CounterController>(
      // 'GetBuilder' listens for 'update()' calls, which manually triggers
      // a rebuild of its child widgets.
      builder: (controller) => Column(
        children: [
          // Here, the Text widget directly accesses 'count' because
          // 'GetBuilder' manages the rebuild when 'update()' is called.
          Text('Count: ${controller.count}'),
          // Button press increments 'count' and triggers UI update via 'update()'.
          ElevatedButton(
            onPressed: controller.increment,
            child: Text('Increment'),
          ),        ],      ),    );  }}
```

Listing 14. Using Redux for simple counter application in Flutter.

```
import 'package:flutter/material.dart';
import 'package:flutter_redux/flutter_redux.dart';
import 'package:redux/redux.dart';
// Define the app state
class AppState {
  final int counter;
  AppState({this.counter = 0});
}
// Define actions
enum Actions { Increment }
// Reducer function to handle state changes
AppState reducer(AppState state, dynamic action) {
  if (action == Actions.Increment) {
    return AppState(counter: state.counter + 1);
  }
  return state;
}
void main() {
  // Create the store with initial state and reducer
  final store = Store<AppState>(
    reducer,
    initialState: AppState(),
  );
  runApp(MyApp(store: store));
}
class MyApp extends StatelessWidget {
  final Store<AppState> store;
  MyApp({Key key, this.store}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return StoreProvider<AppState>(
      store: store,
      child: MaterialApp(
        home: Scaffold(
          appBar: AppBar(title: Text('Redux Example')),
          body: Center(
            // Connect the UI to the store using StoreConnector
            child: StoreConnector<AppState, String>(
              converter: (store) => store.state.counter.toString(),
              builder: (context, count) => Text(
                'Counter: $count',
                style: Theme.of(context).textTheme.headline4,
              ),           ),          ),
          floatingActionButton: StoreConnector<AppState, VoidCallback>(
            converter: (store) {
              return () => store.dispatch(Actions.Increment);
            },
            builder: (context, callback) => FloatingActionButton(
              onPressed: callback,
              tooltip: 'Increment',
              child: Icon(Icons.add),
            ),          ),        ),      ),    );  }}
```

Listing 15. Using MobX for simple counter application in Flutter.

File 'counter.dart' :

```
import 'package:mobx/mobx.dart';
// This will be included in the generated file
part 'counter.g.dart';
class Counter = _Counter with _$Counter;
abstract class _Counter with Store {
  @observable
  int value = 0;
  @action
  void increment() {
    value++;
  }}
```

…using Counter in widget:

```
import 'package:flutter/material.dart';
import 'package:flutter_mobx/flutter_mobx.dart';
import 'counter.dart';
class CounterPage extends StatelessWidget {
  final Counter counter = Counter();
  @override
  Widget build(BuildContext context) {
```

```
return Scaffold(
  appBar: AppBar(title: Text('MobX Counter')),
  body: Center(
    child: Observer(
      builder: (_) => Text(
        '${counter.value}',
        style: Theme.of(context).textTheme.headline4,
      ),        ),      ),
  floatingActionButton: FloatingActionButton(
    onPressed: counter.increment,
    tooltip: 'Increment',
    child: Icon(Icons.add),
  ),    );  }}
```

Analysis of the ISO/IEC 25010 quality characteristics

Here, we will outline how ISO/IEC 25010 quality characteristics can – or cannot be and why - applied to evaluate the state management solutions discussed:

- Functional suitability: this characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions (ISO/IEC, 2011). This characteristic is composed of the following sub-characteristics:
    - Functional completeness: ensures that all necessary functions for state management are provided by the system.
    - Functional correctness: the state management system must correctly manage the state transitions without errors.
    - Functional appropriateness: the solution should fit the specific needs of the application, whether it's managing simple local states or complex global app states.

Applying this characteristic (or any of it`s sub-characteristics) to state management packages looks not meaningful because they do not provide end-user functions but instead act as mechanisms for managing and propagating state. Unlike application features that deliver business logic, state management libraries are general-purpose utilities that developers configure according to their needs. Since all state management solutions fulfill their intended role of handling state changes, evaluating them based on functional suitability would offer little differentiation.

- Performance efficiency: this characteristic represents the degree to which a product performs its functions within specified time and throughput parameters and is efficient in the use of resources (such as CPU, memory, storage, etc...) under specified conditions (ISO/IEC, 2011). This characteristic is composed of the following sub-characteristics:
    - Time behavior: measures how quickly state changes are reflected in the UI, including app responsiveness and rebuild efficiency.
    - Resource utilization: evaluates the memory usage and CPU load during state changes, ensuring that the solution does not lead to excessive resource consumption.

In the context of Flutter state management, performance efficiency is primarily evaluated by focusing on factors like widget rebuild frequency. Effective state management solutions should minimize unnecessary widget rebuilds to maintain smooth UI performance. Tools like Flutter DevTools can measure frame rendering times and jank (frame drops) to gauge the impact of state updates. While memory usage and CPU load are relevant, their variation across state management approaches is typically minimal, making detailed resource utilization tests (e.g., CPU/RAM profiling) less critical. Instead, we prioritize testing widget rebuild counts to assess time behavior, as this provides sufficient insight into computational efficiency and ensures state propagation remains responsive and scalable in complex applications.

- Compatibility: degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same common environment and resources (ISO/IEC, 2011). This characteristic is composed of the following sub-characteristics:
    - Co-existence assesses how well different state management libraries can work together or within the same application environment without conflicts.
    - Interoperability: looks at how easily the state management system can interface with other parts of the application or external systems.

In the context of Flutter state management, this characteristic is largely irrelevant because all state management solutions are built within the same Flutter ecosystem and run on Dart, ensuring inherent compatibility across platforms. Unlike APIs or external integrations that require interoperability with different technologies, state management libraries interact only with Flutter's widget tree and Dart runtime, making compatibility a non-issue. Since all major solutions work seamlessly within any Flutter project, evaluating them based on compatibility would provide no meaningful differentiation.

- Interaction capability: degree to which a product or system can be interacted with by specified users to exchange information via the user interface to complete specific tasks in a variety of contexts of use (ISO/IEC, 2011). This characteristic includes several sub-characteristics, however, in the context of evaluating state management systems in Flutter - where the focus is on developer tools rather than end-user interfaces - only certain aspects are applicable. Below the sub-characteristics are outlined with justification for their inclusion or exclusion:
    - Appropriateness recognizability: the degree to which users can recognize whether a product or system suits their needs. For state management systems, this pertains to end-users assessing an application's suitability, not developers evaluating a library's technical fit, making it less relevant here.
    - Learnability: how easy it is for developers to learn and implement the state management approach, considering the quality of documentation and community support. This sub-characteristic is directly applicable, as it addresses the developer experience, a key factor in choosing and effectively using state management solutions.
    - Operability: the ease with which developers can operate and maintain the state management within the application. While relevant to developers, Operability essentially mirrors Learnability, as both address the developer's ability to effectively

use and understand the state management system. Since Learnability is already included to evaluate how easily developers can adopt these tools - adding Operability separately would be redundant and unnecessary.

  - User error protection. the degree to which a system prevents users from making operational errors. This is user-centric, focusing on end-user interactions with the UI, not developer interactions with state management APIs.
  - User engagement: the degree to which a user interface presents functions and information in an inviting and motivating manner to encourage continued interaction. This applies to end-user UI design, not state management systems as developer tools.
  - Inclusivity: the degree to which a product can be used by people of diverse backgrounds (e.g., ages, abilities, cultures). This is an end-user-focused metric, irrelevant to state management libraries used by developers.
  - User assistance: the degree to which a product supports users with varying characteristics to achieve goals. This pertains to end-user help features, not developer-oriented state management.
  - Self-descriptiveness: the degree to which a product provides clear information to make its capabilities obvious without excessive external resources. While potentially relevant to developer documentation, it overlaps with Learnability and is more user-interface-specific, thus less critical here.

- Reliability: may be defined as the degree to which a state management system in Flutter performs its specified functions under defined conditions over a specified period (ISO/IEC, 2011). While this characteristic is essential for assessing software systems broadly, its sub-characteristics relevance to state management solutions in Flutter, where the focus is on developer tools rather than end-user-facing system uptime or hardware resilience, is to be reviewed:

  - Faultlessness: the extent to which a state management system performs its intended functions - such as updating or retrieving state - without errors during normal operation. This is not distinctly relevant in this thesis context, as preventing errors in state updates stands more with developer`s expertise and accuracy, rather than a unique reliability aspect of state management tools.
  - Availability: the degree to which the state management system remains operational and accessible when needed by the application. This sub-characteristic is less applicable, as state management systems are not standalone services requiring

uptime but are embedded libraries within the app. Their "availability" is tied to the overall app's runtime rather than independent operational status.

    - Fault tolerance: the capability of the system to function as intended despite hardware or software faults. This is largely irrelevant, as state management systems operate within the Flutter framework and do not directly handle hardware faults or low-level software failures (e.g., OS crashes), which are outside their scope and managed by the broader application or runtime environment.
    - Recoverability: the ability to restore affected data and return to the desired state after an interruption or failure. This is minimally relevant, as state management systems typically lack built-in mechanisms for data recovery post-failure (e.g., app crashes); such recovery is usually handled by persistence layers (e.g., databases) or app-level logic, not the state management tool itself.

- Security: in the ISO/IEC 25010 standard, Security is defined as the degree to which a product or system defends against attack patterns by malicious actors and safeguards information and data, ensuring that access is limited to authorized entities based on their types and levels of authorization (ISO/IEC, 2011). While Security is a critical attribute for many software systems, it is not directly relevant to the evaluation of state management systems in Flutter, as described below in sub-characteristics description:

    - Confidentiality: the degree to which a product or system ensures data are accessible only to those authorized. This is not relevant, as state management systems in Flutter manage in-memory application state rather than enforcing data access controls, which are typically handled by higher-level security mechanisms like authentication or encrypted storage.
    - Integrity: the degree to which a system prevents unauthorized modification or deletion of its state or data by malicious actions or errors. This is not distinctly applicable, as state management systems focus on state consistency within the app's runtime, not on protecting against external tampering, which falls under application-level or OS security.
    - Non-repudiation: the degree to which actions or events can be proven to have occurred, preventing later denial. This is irrelevant, as state management systems do not involve tracking or proving user actions for legal or audit purposes; such concerns are outside their scope and pertain to logging or transactional systems.
    - Accountability: the degree to which an entity's actions can be uniquely traced to that entity. This sub-characteristic is not applicable, as state management libraries do not manage user identities or action attribution, which is a concern for user authentication frameworks, not state handling.

    - Authenticity: the degree to which a subject or resource's identity can be verified as claimed. This is not relevant, as state management systems operate within the app's trusted environment and do not authenticate entities, a task reserved for security layers external to state management.
    - Resistance: the degree to which the system sustains operations under malicious attack. Not applicable as state management systems are not designed to resist external attacks directly; such resilience is managed by the Flutter framework, OS, or app-level security measures, not the state management logic itself.

- Maintainability: refers to the degree of effectiveness and efficiency with which state management systems in Flutter can be modified to enhance functionality, fix issues, or adapt to evolving requirements or environments (ISO/IEC, 2011). This characteristic is composed of the following sub-characteristics:

    - Modularity: the extent to which a state management system is built from independent components, allowing changes to one part to minimally affect others. Highly relevant, as modular state management (e.g., separating business logic from UI) enhances maintainability by isolating changes, reducing unintended side effects across the application.

    - Reusability: the degree to which a state management system can be reused across different systems or in building other components. Limited relevance, as state management solutions are typically tailored to specific application contexts in Flutter, and reusability across unrelated projects is not a primary focus of this evaluation

    - Analyzability: the effectiveness and efficiency with which developers can evaluate the impact of changes, diagnose issues, or identify parts needing modification in the state management system. Very relevant, as clear analyzability allows developers to understand and troubleshoot state-related problems efficiently, a critical factor for maintaining complex Flutter apps.

    - Modifiability: the degree to which a state management system can be altered effectively and efficiently without introducing defects or degrading quality. Highly relevant, as the ability to modify state logic (e.g., adding new features or fixing bugs) without compromising stability is essential for ongoing development and adaptation in Flutter projects.

    - Testability: the effectiveness and efficiency with which test criteria can be defined and tests can be conducted to verify the state management system meets its requirements. Extremely relevant, as testable state management systems enable developers to validate state behavior through automated tests, ensuring reliability and correctness, a key focus of this thesis.

- Flexibility: refers to the degree to which state management systems in Flutter can be adapted to changes in requirements, usage contexts, or system environments, as outlined by the ISO/IEC 25010 standard (ISO/IEC, 2011). While Flexibility is valuable in broader software contexts, its relevance to state management evaluation varies across its sub-characteristics. Below are the sub-characteristics with their applicability to this study:
    - Adaptability: the extent to which a state management system can be effectively and efficiently adjusted or transferred to different hardware, software, or operational environments. Moderately relevant, as adaptability matters for ensuring state management works across Flutter's multi-platform support (e.g., mobile, web, desktop). However, this is often more a function of Flutter's framework than the state management system itself.
    - Scalability: the degree to which a state management system can manage increasing or decreasing workloads or adjust its capacity to handle variability. Highly relevant, as scalability directly impacts how well a system handles growing application complexity (e.g., more states, users, or interactions), a key focus of this thesis for assessing performance and maintainability in larger Flutter apps.
    - Installability: the effectiveness and efficiency with which a state management system can be installed or uninstalled in a specified environment. Not relevant, as state management systems are libraries integrated via package managers (e.g., pub.dev) within a Flutter project, not standalone products requiring installation or uninstallation processes.
    - Replaceability: the degree to which one state management system can substitute another for the same purpose in the same environment. Limited relevance, as replacing a state management system typically requires significant refactoring due to architectural differences, making direct substitution impractical in most Flutter development scenarios.
- Safety: within the framework of this thesis, Safety, as defined by the ISO/IEC 25010 standard, represents the degree to which a product operates under specified conditions to prevent states that could endanger human life, health, property, or the environment (ISO/IEC, 2011). While Safety is a vital attribute in certain software domains, its relevance to state management systems in Flutter is negligible due to their role as developer tools rather than safety-critical systems. Below are the sub-characteristics with their applicability in this context:

- Operational constraint: the extent to which a state management system limits its operations to safe parameters when facing hazards. Not relevant, as state management systems in Flutter do not manage operational hazards affecting life or property; they handle application state within a controlled software environment.
- Risk identification: the degree to which a system can detect events or operations posing unacceptable risks to life, property, or the environment. Irrelevant, as these systems lack mechanisms to identify risks beyond software logic errors, and such risks are not within their scope, which is limited to managing UI state.
- Fail safe: the ability of a system to enter a safe mode or revert to a safe state upon failure. Not applicable, as state management systems do not define safety-critical fail-safe behaviors; failure handling (e.g., crashes) is managed by the Flutter framework or app-level logic, not the state management layer.
- Hazard warning: the degree to which a system warns of risks to sustain safe operations. Not relevant, as state management systems do not monitor or warn about operational hazards; their focus is on state consistency, not safety-related alerts.
- Safe integration: the extent to which a system maintains safety during and after integration with other components. Irrelevant, as safety in integration pertains to physical or critical systems, not the integration of state management libraries into Flutter apps.

Declaration of Authenticity