Flutter Engineering

Copyright 2024 Majid Hajian

Sample Digital Version

Contents

Contributors and Reviewers **5**

Preface **7**

Acknowledgment **11**

I Foundation of Flutter Engineering 13

1 Flutter Engineering: Core Concepts **15**

1.1 Engineering Software with Flutter 15
1.2 Unpacking the Core Principles 17
1.3 Lifecycle of Flutter Development 29
1.4 Flutter Engineering vs. Programming 31
1.5 Flutter's Position in Tech Evolution 32
1.6 Conclusion 33

2 Custom Painters and Shaders **35**

2.1 The Art of CustomPainter 35
2.2 Exploring Shaders 48
2.3 Using Shaders in Flutter 52
2.4 Conclusion 60

Closing Words **63**

Contributors and Reviewers

Simon Lightfoot, Mangirdas Kazlauskas, Roman Jaquez, Erick Zanardo, Carlo Lucera, Marco Napoli, Alessio Salvadorini, Pooja Bhaumik, Dominik Roszkowski, Oleksandr Leushchenko, Anna Leushchenko, Tomá Soukal, Danielle Cox, Manuela Sakura Rommel, Verena Zaiser, Cagatay Ulusoy, Mike Rydstrom, Muhammed Salih Guler and Renan Araujo.

Your commitment to detail has ensured the accuracy and clarity of the content.

Cover Design:

The cover art of this book is an outstanding collaboration between human creativity and artificial intelligence. It features an engineered butterfly that represents Flutter Engineering. Thanks to Shirin Sameiee for being a creative artist in bringing this visionary concept to life.

Digital Version:

This book is available in various formats, including PDF, EPUB, and MOBI. Visit Flutterengineering.io for more information and continuous updates.

Preface

My journey in programming has been diverse, from backend and frontend roles to full-stack development, software architecture, and developer relationships. Each step has enriched my insight into exceptional software engineering.

As someone naturally drawn to logic and analysis, I have often considered what makes a great software engineer and how to plan an app and manage its development effectively. “Flutter Engineering” aims to answer these questions and more. This book lays the foundation for a broader understanding of app development, offering knowledge, architectural insights, and advanced Flutter-specific content. This project is your gateway to look at software engineering concepts in Flutter. It is designed to guide, open doors, and help you explore.

During my year-long effort, I approached Flutter from an engineering perspective rather than merely a programming skill. As a Flutter engineer, building an app requires more than just coding skills. The “Flutter Engineering” project (flutterengineering.io) and this book were created to share this knowledge. Welcome to “Flutter Engineering.”

0.0.1 Who is this book for?

This book is intended for readers with a basic understanding of Flutter and some experience with Dart programming and Flutter. It is unnecessary to be an expert, but if you have no prior experience with Flutter and want to learn it from scratch, this book may not suit you, as I have this assumption while explaining concepts.

This book is meant to help beginner and professional Flutter developers level up their skills in software engineering.

0.0.2 What you will learn

This book covers various software engineering topics in Flutter and is divided into five parts, each addressing specific areas of interest and expertise.

- Part 1 of this book introduces software engineering concepts specifically relevant to Flutter. It starts by explaining the fundamental principles of Flutter and how they work and then moves on to demonstrate how these concepts

can be applied within the Flutter framework. This section also covers coding design patterns that help readers develop a foundational understanding of the subject matter.

- The second part of this material centers on architecture, beginning with basic principles and progressing onto different styles and patterns in architectural design. It covers important concepts such as concurrency, parallelism, dependency injection, and state management. Additionally, this section explores the architectural factors involved in developing offline apps, guiding readers on strategic thinking and decision-making in software architecture.
- Part 3 of this book covers the important software development processes necessary for building a successful app. It covers rules and guidelines, continuous delivery and integration, testing methodologies, and effective documentation practices in a comprehensive manner. This section gives readers the knowledge needed to streamline and efficiently develop apps.
- Part 4 covers important aspects of security and privacy in software development, specifically related to Flutter. It discusses the OWASP Top 10 security risks, privacy standards, and accessibility considerations. The section emphasizes creating inclusive and secure applications that value user privacy.
- Part 5 delves into advanced user interface (UI) concepts, including adaptive and responsive design strategies. It also explores sophisticated topics like custom painting, shaders, internationalization, localization, and effective theming in Flutter. This section enhances the reader's skills in creating visually appealing and user-friendly interfaces.

Whether a beginner or an experienced developer, this book can offer you something to learn. Although the book covers a wide range of topics, discussing each topic in detail could deserve a dedicated book. It is impossible to cover all the details and edge cases in one book, but I have done my best to cover the most important details. I plan to discuss topics I could not detail enough and cover edge cases on flutterengineering.io[1], so subscribe to the newsletter for free access to additional content.

0.0.3 How to use this book

This book has been designed to allow you to read each chapter independently. You are free to start from any part of the book. However, reading an entire part from start to end is always better. Certain fundamental concepts must be understood before proceeding to subsequent chapters. Therefore, I do not recommend starting with chapters that follow the previous chapter. Usually, in the introduction of each chapter, I mention if you should have read another chapter before starting it. Pay attention to this detail to read and understand the book better.

As you progress through this book, you'll encounter various conventions to enhance

[1] http://flutterengineering.io/

your reading and learning experience. Knowing these will help you understand and navigate the content more effectively.

1. **Bold Titles and Keywords**: Key concepts, titles, and important keywords are emphasized in **bold** font. This highlights essential information and makes it stand out for easy reference.
2. *Italicized Queries and Comments*: To differentiate, queries or comments within the text are presented in *italic*. This formatting helps distinguish them from the main content, providing additional insights or raising questions for contemplation.
3. Inline `Code`: I have tried to use inline code where it's possible to highlight the difference between regular text and a piece of code within the paragraphs.
4. **Formatted Code Blocks**: The Dart programming language formats the code examples for clarity. The blocks are structured to reflect the appropriate syntax and style of the language.
5. **Pseudo-code Examples**: It's important to note that most code examples are presented as pseudo-code. When integrated into your projects, they are simplified and may require additional coding or imports. This approach aims to focus on the key parts of the code, minimizing distractions and enhancing understanding. In some instances, critical lines within these code blocks are further emphasized in **bold**. Please pay attention to these lines, as they often represent core ideas.
6. **Code Explanations and Comments**: You will find step-by-step explanations following many code blocks. These are intended to clarify the code and guide you through the logic and functionality. While comments are included within the code blocks to aid comprehension, they are kept brief to avoid overloading the code with annotations.
7. **Access to Complete Code Examples**: To view the full working examples, please visit the accompanying website at flutterengineering.io[2]. There, you find where to access and clone the book's example repository, which contains the complete and functional code samples referenced in the book.

By familiarizing yourself with these conventions, you can navigate the book more efficiently and better understand the material. Enjoy your journey into Flutter engineering!

0.0.4 Found a bug?

Every chapter of this book has gone through a complete review process. At least two people have read each chapter, and an editor has also reviewed them. Before publication, the book's examples were tested, and multiple feedbacks were addressed to ensure their accuracy.

[2] http://flutterengineering.io/

However, despite all these efforts, some errors may still exist. You may encounter some grammar or spelling mistakes (which I hope you don't), or you may notice technical errors such as code errors or wrong explanations. If you disagree with any part of the book and have a valid reason, please don't hesitate to contact me. Your feedback and review are valuable to me, and I will consider and revise the book to ensure that the next person who reads it experiences different issues.

I appreciate your help in this process.

0.0.5 Contact me

If you would like to contact me, which I highly encourage, especially for sending me feedback and reviews of my book, you can do so in a few different ways. You can subscribe to my website newsletter or email me directly at majid@flutterengineering.io[3]. Alternatively, you can connect with me directly on social media. I am active on LinkedIn (linkedin.com/in/mhadaily) and Twitter (x.com/mhadaily). You can also find me on YouTube (youtube.com/mhadaily) and GitHub (github.com/mhadaily).

[3] mailto:majid@flutterengineering.io

Acknowledgment

I extend my heartfelt gratitude to everyone who has supported and contributed to the creation of this book. First and foremost, I would like to thank my family for their tireless support and patience throughout this journey. Their support and faith in my dreams have been the foundation of my drive.

Thanks to the Flutter Team for doing such amazing work, particularly to Craig Labenz, Leigha Reid, Eric Windmill, Kevin Moore, Kate Lovett, and John Ryan and many more engineers in Flutter team, whose content inspired me.

I also acknowledge the contribution of the technical reviewers, whose keen eyes and expert knowledge have greatly enhanced the quality of this book. Thank you Simon Lightfoot, Anna Leushchenko, Oleksandr Leushchenko, Mangirdas Kazlauskas, Roman Jaquez, Erick Zanardo, Carlo Lucera, Marco Napoli, Alessio Salvadorini, Pooja Bhaumik, Dominik Roszkowski, Tomá Soukal, Danielle Cox, Manuela Sakura Rommel, Verena Zaiser, Cagatay Ulusoy, Mike Rydstrom, Muhammed Salih Guler and Renan Araujo. Your commitment to detail has ensured the accuracy and clarity of the content.

I am grateful to my friends Taha Tesser, Argel Bejarano, Nilay Coskun, Elliot Hesp, Mike Diarmid, Andrea Bizzotto, Jaime Blasco, Bettina Carrizo, Tomas Piaggio, Enzo Conty, Gonçalo Palma, Chris Swan, Esra Kadah, Randal Schwartz, Frank van Puffelen, Mark O'Sullivan, Anthony Prakash, Lukas Klingsbo, Leo Farias, Abhishek Doshi, Sasha Denisov, Pascal Welsch, Swav Kulinski, Pawan Kumar, Scott Stoll, Remi Rousselet, Filip Hráek, Felix Angelov, Ahmed Alabd and many more which if I want to name I have to write a book only for that; your passion and dedication to the field have inspired me constantly.

I want to express my gratitude to Codemagic and Martin Jeret for being the pioneers in supporting this book. I am also grateful to Invertase, especially Elliot Hesp and Mike Diarmid, for their support in various aspects and to Sergiy Yakymchuk from Talsec, who has been an amazing supporter. Thank you all very much.

Lastly, I am thankful to the readers and the Flutter community. Your willingness to learn and grow continually drives me to share my knowledge and experience. This book is for you, and I hope it serves as a useful guide in your Flutter development journey.

Part I

Foundation of Flutter Engineering

CHAPTER 1

Flutter Engineering: Core Concepts

Reviewers: Anna Leushchenko, Oleksandr Leushchenko

Welcome to the exciting world of Flutter engineering! This chapter explores the fundamental principles and concepts that form the basis of successful software development using Flutter. Through this exploration, you will gain valuable insights into the unique perspectives and approaches that distinguish Flutter engineering from conventional programming, equipping you with the knowledge and understanding to craft impactful and enduring applications.

1.1 Engineering Software with Flutter

Throughout my career in software engineering, embracing Flutter has marked a significant evolution in my approach to technology. More than just acquiring a new skill, it has involved adopting a comprehensive strategy that spans the entire software development lifecycle, from design and development to testing and maintenance.

My diverse background in various technologies has helped me gain a better perspective on Flutter, which I see as both a technical tool and a way to promote innovation and creativity in software development. Flutter Engineering takes a holistic approach that carefully balances user experience, efficient time management, scalability considerations, and the trade-offs required to create impactful software.

Flutter's multi-platform architecture enables developers to concentrate on creating an exceptional user experience instead of getting into the nitty-gritty of platform-specific details. Unlike native development, which focuses on adhering to platform guidelines, Flutter prioritizes branding and user experience. This approach encourages developers to prioritize universal usability over platform constraints, leading to a more user-centric mindset.

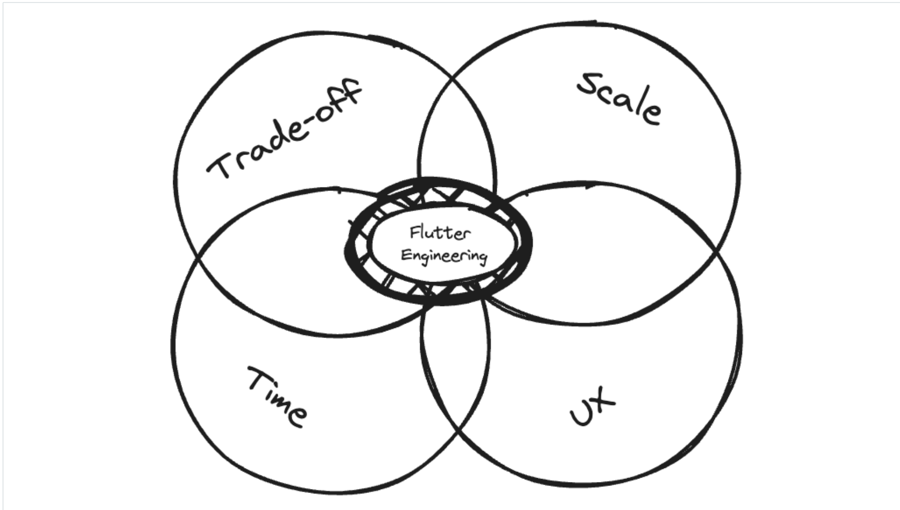

Figure 1.1: Flutter Engineering Pillars

User experience (UX) in Flutter engineering is a critical lens through which every project *must be* analyzed. I constantly ask myself, "*What user will think of this feature? Does it enhance or complicate their experience?*" For example, when creating a Flutter-based educational app, answering these questions helps make the design intuitive and engaging for learners; balancing aesthetics with functionality results in a delightful experience. The challenge is to align this user-centric focus with the technical capabilities of Flutter, ensuring the app is as pleasant to interact with as it is functional.

Time is a vital resource in software development. It is a finite constraint that needs to be managed efficiently to meet project deadlines and deliver value to stakeholders. Time management in software development goes beyond simply hitting milestones. Time is a multifaceted and dynamic element in the realm of Flutter development. It prompts me to frequently ask, "*What is the expected lifespan of my Flutter code? How long this app is going to stay? Year or decade? What is our deadline to deliver?*" These questions go beyond meeting project deadlines and delve into future-proofing the application.

For example, when I work on a Flutter-based smart home app, I focus not only on its immediate launch but also on its adaptability to future IoT trends and technological evolutions. This approach ensures the app remains relevant and functional over time, adapting to user behavior and technology changes. It also reminds me to incorporate the mechanism to upgrade Flutter and other third-party dependencies.

The concept of **scale** in Flutter engineering is complex and thought-provoking. When embarking on a new project, I often consider, "*How many people are involved, and what roles do they play in the development and maintenance of the project? How many end-users will use this application later?*" These questions

become particularly relevant in larger-scale projects like a comprehensive logistics application developed with Flutter. Here, the challenge lies in managing a robust codebase and orchestrating a team with diverse expertise, ensuring cohesive and efficient development across different platforms and devices.

Trade-offs in Flutter engineering involve making strategic decisions that balance various aspects of the project. For instance, I often face decisions like, "*Should I implement an advanced, resource-intensive feature that could enhance the user experience but also affect the app's performance on some devices?*" One example is choosing between high-resolution graphics and smooth performance in a gaming app. Another is deciding between implementing an advanced animation that enhances the user experience and maintaining a lean, fast-loading application, which exemplifies the kind of strategic decision-making that defines Flutter engineering. These decisions are not merely technical but also align with the broader objectives of the project and the expectations of its users.

In my experience, engineering software with Flutter is a detailed process of crafting adaptable, scalable solutions that resonate with end-users. It involves a blend of technical skills, strategic planning, and creative problem-solving, all directed toward building functional but also engaging and sustainable applications in the dynamic world of digital technology.

1.2 Unpacking the Core Principles

To fully grasp these concepts, let's explore Flutter app development through the lens of core software engineering principles.

1.2.1 Development Paradigms

In software development, diverse ideologies and methodologies guide the construction of systems. These guiding principles, or development paradigms, offer distinct lenses through which developers approach and shape software.

Different programming languages are often associated with specific paradigms, and the language choice can influence how developers think about and solve problems. Some languages, like Dart, support multiple paradigms.

Across the history of computing, several well-known paradigms have emerged, each leaving a significant imprint on the field. These include Procedural Programming, Object-Oriented Programming (OOP), Functional Programming, Agile Development, Event-Driven programming, Imperative and Declarative programming, etc.

These paradigms, however, seldom exist in isolation. Flutter embraces a **multi-paradigm** programming environment, utilizing various programming techniques where their strengths are most beneficial. Let's explore some of the key threads in this multifaceted approach:

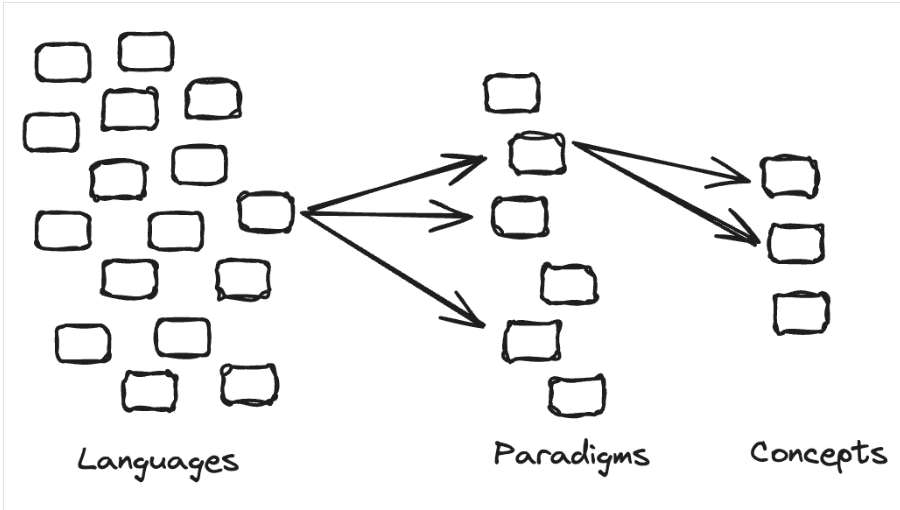

Figure 1.2: Development Paradigms and Concepts

1.2.2 Constraint and Composition Programming

The heart of Flutter's design lies in its use of composition. This approach involves building complex widgets by combining simpler ones. An example is the `TextButton` widget, a composition of other widgets like `Material`, `InkWell`[1], and `Padding`[2].

Imagine your app as a giant Lego masterpiece. Each **widget**, a small, specialized piece (text, buttons, images), snaps together, building complex screens. This method of aggressive composition results in a highly customizable and flexible UI. You will learn more about this in chapter 2.

In Flutter, the layout system employs a form of constraint programming to set the geometry of UI elements. Constraints regarding size, such as minimum and maximum width and height, are passed from parent widgets to their children. The child widgets then adjust their sizes to meet these constraints, allowing Flutter to efficiently lay out the entire UI, often in a single pass. This approach ensures a responsive and consistent layout across different devices.

1.2.3 Imperative and Declarative Programming

In Flutter, imperative programming is applied in scenarios requiring direct, step-by-step operations control. Mobile app business logic frequently involves sequences

[1] https://api.flutter.dev/flutter/material/InkWell-class.html
[2] https://api.flutter.dev/flutter/widgets/Padding-class.html

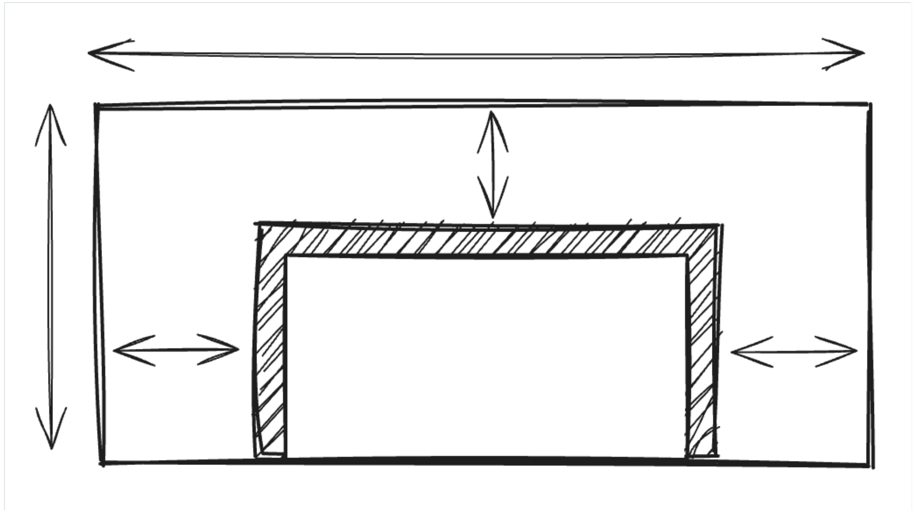

Figure 1.3: Constraints go down. Sizes go up. Parent sets position.

of steps, conditions, and loops. Imperative programming allows developers to express these sequences naturally, making writing and maintaining the logic easier.

Here is a simple example of an imperative-style function with conditional statements:

```
bool isPositive(int x) {
  if (x > 0) {
    print('x is positive');
    return true;
  }
  print('x is negative or zero');
  return false;
}
```

Another common example of imperative programming in Flutter can be seen in unit tests:

```
testWidgets('CustomButton displays a label', (WidgetTester tester) async {
  // Describe the situation under test
  await tester.pumpWidget(MaterialApp(home: CustomButton(label: 'Test')));

  // List the invariants the test must match
  expect(find.text('Test'), findsOneWidget);

  // Advance the clock or insert events if necessary
  await tester.tap(find.byType(CustomButton));
  await tester.pump();
});
```

Declarative programming is a key aspect of Flutter's framework, prominently seen in how widgets are constructed. In Flutter, the UI is typically defined using Dart's declarative syntax, where the build methods of widgets consist of single expressions with nested constructors.

Consider the `ListView` widget:

```
ListView(
  children: [
    ListTile(title: Text('Item 1')),
    ListTile(title: Text('Item 2')),
    // Additional list items
  ],
)
```

In this example, `ListView` and its children are defined concisely and expressively.

This approach allows developers to describe what the UI should look like rather than how to construct it step by step, as in imperative programming. The declarative style in Flutter simplifies the process of building complex UIs and enhances the readability and maintainability of the code. Additionally, this method can be seamlessly combined with imperative programming for scenarios where a pure declarative approach might be limited, offering the flexibility to build more dynamic and interactive UIs.

Looking at this code, you may see the application with an AppBar and a centered text. It does not contain the logic that specifies **how** the UI will be constructed, but just the **declaration** of **what** the user will see:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Declarative Programming in Flutter'),
        ),
        body: Center(
          child: Text('Hello, Flutter!'),
        ),
      ),
    );
  }
}
```

1.2.4 Functional and Object-Oriented Programming

One of the core concepts of functional programming is called "pure function." Given the same input, it's a function that will always produce the same output

and has no observable side effects. The result of a pure function depends only on its input parameters, and it does not modify any external state, which significantly simplifies maintenance and opens doors for many optimizations.

Flutter also embraces functional programming, particularly in `StatelessWidgets`[3] that resemble pure functions. For instance, the `Icon`[4] widget can be viewed as a function mapping its parameters to visual output.

Flutter's emphasis on immutable data structures. The entire `Widget`[5] class hierarchy and supporting classes like `Rect`[6] and `TextStyle`[7] embrace this immutability, keeping your UI stable and reliable.

Dart's `Iterable`[8] API is another example of its functional programming characteristics. Remember those handy functions like `map`, `where`, and `reduce` your use in Dart? These are examples of the functional style frequently used to process lists of values in the framework.

Flutter's framework dances with both class inheritance and dynamic prototypes. Core APIs are built with class hierarchies, where base classes like `RenderObject`[9] define high-level functionalities that subclasses like `RenderBox`[10] specialize, adopting the Cartesian coordinate system for geometry. But it's not just static inheritance – the `ScrollPhysics`[11] class lets you chain instances dynamically at runtime, composing, for example, paging physics with platform-specific quirks, all without needing a pre-chosen platform. This blend of inheritance and dynamic flexibility gives Flutter apps the power to adapt and evolve like never before!

You will learn more about OOP in Dart in Chapter 3.

1.2.5 Abstraction and Encapsulation

Abstraction and encapsulation are fundamental principles in software engineering that Flutter effectively utilizes in its widget-centric architecture.

Abstraction is about simplifying complex systems into more manageable models, and Encapsulation involves grouping data and its associated operations within classes, protecting data integrity, and preventing improper access.

Abstraction simplifies complex UI elements into manageable widgets, focusing on essential attributes and functionalities. For example, a `ListView` widget in Flutter abstracts the complex functionalities of a scrollable list into an easy-to-use component.

[3] https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html
[4] https://api.flutter.dev/flutter/widgets/Icon-class.html
[5] https://api.flutter.dev/flutter/widgets/Widget-class.html
[6] https://api.flutter.dev/flutter/dart-ui/Rect-class.html
[7] https://api.flutter.dev/flutter/painting/TextStyle-class.html
[8] https://api.flutter.dev/flutter/dart-core/Iterable-class.html
[9] https://api.flutter.dev/flutter/rendering/RenderObject-class.html
[10] https://api.flutter.dev/flutter/rendering/RenderBox-class.html
[11] https://api.flutter.dev/flutter/widgets/ScrollPhysics-class.html

In the context of Flutter, encapsulation is applied to widget development, and the concept is evident in implementing the `Container` widget. The `Container` widget encapsulates various attributes or properties that define its appearance and behavior. These attributes include width, height, color, padding, margin, and more. Developers interact with the Container using a well-defined set of properties and methods. The encapsulation ensures that the internal details of how the `Container` manages these attributes are hidden from the outside world.

Together, abstraction and encapsulation in Flutter contribute to a framework where complex UI designs are simplified into manageable components, and internal widget states are well-guarded, enhancing usability and maintainability. You'll learn more about these topics in Chapter 3.

1.2.6 Event-Driven Programming

User interactions in Flutter are handled through an event-driven approach.

A prime example of this in Flutter is the use of the `Listenable`[12] class. This class serves as the foundation for the animation system in Flutter, where changes in the animation state are treated as events. `Listenable` provides a subscription model, enabling multiple listeners to register callbacks triggered in response to specific events. This mechanism ensures that various parts of the UI stay updated and in sync with the underlying data or state changes, reflecting the reactive nature of the framework.

In addition, widgets like `GestureDetector`[13] and state management tools utilize events to respond to user inputs, exemplifying event-driven programming in the framework. You will learn more about this in part 2 of this book.

1.2.7 Reactive Programming

In Flutter, reactive programming is a key concept that drives the dynamic nature of UI development. This paradigm is apparent in how widgets react to changes, updating their state and appearance in response to user interactions or internal data changes.

In Flutter's reactive system, any new input provided in a widget's constructor immediately triggers a rebuild of that widget, propagating changes down the widget tree. Conversely, changes in lower-level widgets can propagate up the tree through event handlers and state updates.

Flutter leverages Dart's support for streams to provide a reactive programming model, and StreamBuilder is a widget that plays a key role in this paradigm:

[12] https://api.flutter.dev/flutter/foundation/Listenable-class.html
[13] https://api.flutter.dev/flutter/widgets/GestureDetector-class.html

```dart
final StreamController<int> _controller = StreamController<int>();
//
StreamBuilder<int>(
  stream: _controller.stream,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Center(
        child: Text('Data from stream: ${snapshot.data}'),
      );
    } else {
      return Center(
        child: Text('Waiting for data...'),
      );
    }
  },
)
```

Reactive programming is a programming paradigm that revolves around the propagation of changes and handling asynchronous data streams.

1.2.8 Generic Programming

Flutter uses generics to improve type safety and reduce errors. This is visible in widgets like `DropdownButton<T>` where T represents the type of data source, or classes like `State`[14]`<T>` and `GlobalKey`[15]`<T>`, where T represents the type of widget or state they are associated with.

1.2.9 Concurrent Programming

Concurrency in Flutter is handled through Dart's async features like `Future`[16]s[17] and `Stream`[18]s[19]. This is crucial in scenarios like fetching data from a network or working with long-running tasks.

You will learn more about Concurrency and Parallelism in chapter 8.

1.2.10 Cohesion and Coupling

In software engineering, **cohesion** and **coupling** are fundamental principles that can make or break a system's maintainability and efficiency.

[14] https://api.flutter.dev/flutter/widgets/State-class.html
[15] https://api.flutter.dev/flutter/widgets/GlobalKey-class.html
[16] https://api.flutter.dev/flutter/dart-async/Future-class.html
[17] https://api.flutter.dev/flutter/dart-async/Future-class.html
[18] https://api.flutter.dev/flutter/dart-async/Stream-class.html
[19] https://api.flutter.dev/flutter/dart-async/Stream-class.html

Cohesion describes the internal strength of a module and how tightly related its elements are to its core purpose. Ideally, modules exhibit high cohesion, with components that work together towards a single goal. Coupling, however, deals with the degree of interdependence between modules. Striving for low coupling ensures modules interact minimally, minimizing ripple effects when changes are made.

In Flutter's world, two fundamental principles define a maintainable masterpiece: **low coupling and high cohesion**. Let's break down their steps on the Flutter stage:

High Cohesion

Flutter achieves high cohesion by designing widgets that are focused on specific functionalities. For instance, the `Text` widget is solely responsible for displaying a text string with basic styling. Its responsibilities are clear and well-defined, making it highly cohesive. Another example is the `Image` widget, which is dedicated to displaying images and does not intertwine with non-image functionalities.

Low Coupling

Flutter maintains low coupling by allowing widgets to function independently with minimal reliance on each other. For example, the `Scaffold` widget, which provides the basic material design visual layout structure, operates independently of the `FloatingActionButton` widget used for action buttons. Modifications to a `FloatingActionButton`, such as changing its icon or color, do not affect the layout or functioning of the `Scaffold`, demonstrating low coupling between these components.

You may ask about the Theme now. Themes primarily affect visual styling, keeping functionality separate. Customizable themes at different levels reinforce low coupling, ensuring that changes don't tightly bind widgets.

Generally, widgets should rely on established communication channels like callbacks and events, minimizing cascading effects when one changes tune.

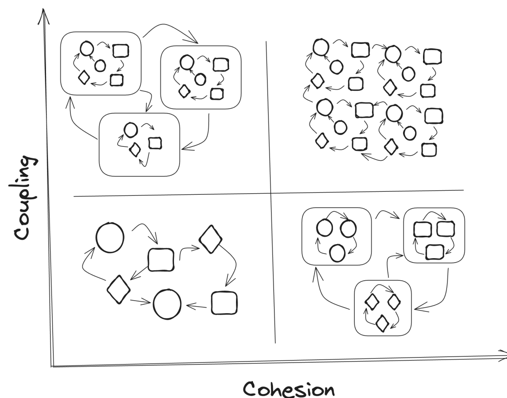

Figure 1.4: Coupling and Cohesion

When developing with Flutter, it's crucial to integrate the principles of "low coupling, high cohesion" for a robust app architecture. Create widgets that operate independently; for instance, a `PaymentProcessing` widget should not be intricately linked to a `UserDashboard` widget, demonstrating low coupling. Also, design each widget with a focused role, like a `ChatScreen` widget exclusively handling messaging features, ensuring high cohesion.

As you build, regularly ask yourself: "Does changing one widget unnecessarily impact others?" and "Is each widget's purpose and function well-defined and self-contained?" Reflecting on these questions will guide you in creating a more efficient, well-structured Flutter application.

1.2.11 Separation of Concerns and Modularity

Separation of Concerns (SoC) and Modularity are foundational concepts in software engineering that significantly improve code organization, maintainability, and scalability.

Separation of Concerns is a design principle that involves breaking down a software application into distinct sections, each addressing a specific aspect or concern. This approach helps simplify a program's complexity by allowing developers to focus on one area at a time without being overwhelmed by others. It aids in reducing interdependencies, which in turn makes the application more flexible and easier to maintain. **Modularity**, however, refers to dividing a software system into separate, interchangeable modules, where each module encapsulates a specific functionality. This design approach facilitates easier testing, debugging, and updating of individual components, leading to a more robust and adaptable system.

Flutter's widget-based architecture is inherently modular, each widget encapsulating a specific UI or functional aspect. This aligns with the SoC principle, where concerns like user interface, business logic, and data management are kept distinct.

As a Flutter developer, you can leverage these principles to create robust and efficient applications. For instance, in a Flutter-based to-do application, you can implement SoC by having a separate UI layer with widgets like `TaskListWidget` for displaying tasks. The business logic can be encapsulated in a `TaskManager` class that handles task-related operations. At the same time, data handling can be managed by a `DatabaseService` responsible for storing and retrieving task data. Modularity can be achieved by creating reusable components like a `LoginService` for user authentication. These can be used across different parts of your app or even in other projects, often residing within the `lib` folder or can be extracted and created as individual `pub` packages.

It's also good to know that in Flutter, the concept of Modularity often intersects with "package by feature" architecture, with a unique aspect being that modules can often take the form of widgets. This approach organizes the application into modules based on specific features, where each module, or in many cases, each widget, represents a distinct functionality of the app.

You will learn more about this in Part 2 of this book, where I delve into Architecture.

1.2.12 Design Patterns and Strategies

Design patterns in software engineering are established solutions to common design problems. They act as templates that can be applied to recurring problems in software design, such as managing object creation, facilitating communication between objects, and organizing complex interactions, allowing you to write code that is:

- **Reusable:** The patterns are reusable, saving time and effort while promoting consistency across your app.
- **Maintainable:** Code structured with patterns is easier to understand, modify, and extend in the future.
- **Flexible:** Patterns adapt to different contexts and requirements, making your code more versatile.

Flutter doesn't dictate specific patterns; its core features and architecture naturally lend themselves to various patterns. An excellent example of a design pattern used within the Flutter framework is the **Builder Pattern**. One everyday use of the Builder pattern in Flutter is the `ListView.builder` widget. This pattern is frequently employed in Flutter's widget creation process. The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

You will learn more about design patterns in Chapter 5.

1.2.13 Efficiency, Scalability, and Trade-offs

In software engineering, particularly in Flutter development, understanding and navigating the complexities of efficiency, scalability, and trade-offs is essential. These concepts focus on how applications utilize resources (efficiency), adapt to growth (scalability), and manage the delicate balance between competing needs (trade-offs). These choices aren't just about financial aspects but encompass various factors like resource allocation, personnel effort, etc.

Moving beyond the mindset of "*because everyone else is doing it*" and towards a consensus-driven approach that prioritizes well-reasoned, context-specific decisions is essential. This mindset is particularly relevant in Flutter when weighing options like state management techniques or integrating external packages.

For instance, choosing `setState` for its simplicity might lead to scalability challenges, whereas advanced methods like BLoC, though initially more complex, offer long-term benefits in scalability and maintainability. Similarly, using `cached_network_image` brings efficiency and enhanced user experience but introduces complexities such as added dependencies, which may affect long-term maintenance and compatibility with Flutter updates.

In my experience, "*It depends*" is particularly significant in software engineering, especially when working with Flutter. This highlights the importance of understanding the specific context of each technology choice. As a developer, I constantly balance factors like ease of use, scalability, and future maintainability. These decisions are more than just about immediate results; they shape the project's long-term health. This requires a deep level of critical analysis and foresight, stressing the need for well-informed and sustainable decisions in the rapidly evolving field of software development.

1.2.14 Verification, Validation, and "Shifting Left"

The Verification and Validation model in software engineering is a process used to ensure that a system meets all its specifications and fulfills its intended purpose. "Verification" involves checking whether the system is built correctly and meets the specified requirements. This is often referred to as "Static Testing." On the other hand, "Validation" checks whether the right system is built and meets the users' needs, known as "Dynamic Testing." This model is crucial for ensuring the quality and reliability of software systems.

In the Flutter context, the Verification and Validation (V&V) model could be tailored to its ecosystem as follows:

1. **Requirement Analysis**: Understanding what the app aims to achieve and the problem it solves for users.
2. **App Architecture**: Defining the overall structure of the app, including state management and navigation strategies.
3. **Feature Design**: Detailing each app feature's implementation plan, encompassing business logic and frontend interface.
4. **Unit Design**: Breaking down features into smaller, testable units, typically individual functions or widgets.

The corresponding testing phases are:

1. **Unit Testing**: Verifying the functionality of individual units or components, especially business logic.
2. **Widget Testing**: Ensuring that Flutter widgets render correctly and interact as expected so the overall widget composition as a feature works.
3. **Integration Testing**: Assessing the interaction between combined units or widgets within the app so the overall architecture works.
4. **User Acceptance Testing**: Validating the app against user requirements, often through manual testing, to ensure it meets their expectations.

The V&V model ensures that the Flutter app is developed correctly and meets its designed needs at each stage, from requirements to acceptance testing.

The "Shifting Left" concept in software development, particularly within Flutter, suggests that investing time in earlier stages of the development lifecycle—such as design, development, and initial testing—is more cost-effective than addressing

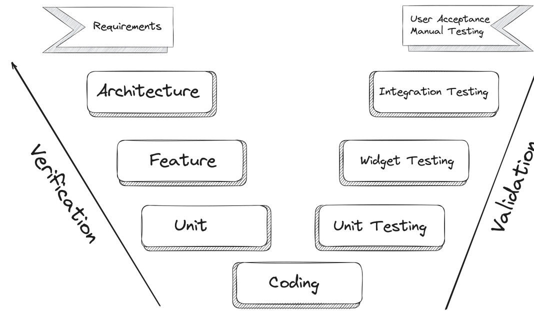

Figure 1.5: Flutter Verification and Validation Model

issues later in the pipeline. The closer an issue is discovered to its introduction (typically on the left side of the development timeline), the less costly it is to fix. This is because issues found during stages like staging or production (on the right side) can be significantly more expensive and time-consuming to resolve due to the complexity of debugging and the potential impact on the user experience.

"Shifting Left" in Flutter effectively means incorporating practices like static code analysis to catch syntactical errors and potential bugs early. Code reviews are essential to ensure quality and catch issues that automated tools might miss. Integrating automation within CI pipelines allows for consistently executing unit, widget, and integration tests, ensuring that new code additions meet quality standards before merging. Additionally, employing feature flags and A/B testing enables developers to test new features selectively in production environments, reducing the risk of widespread issues.

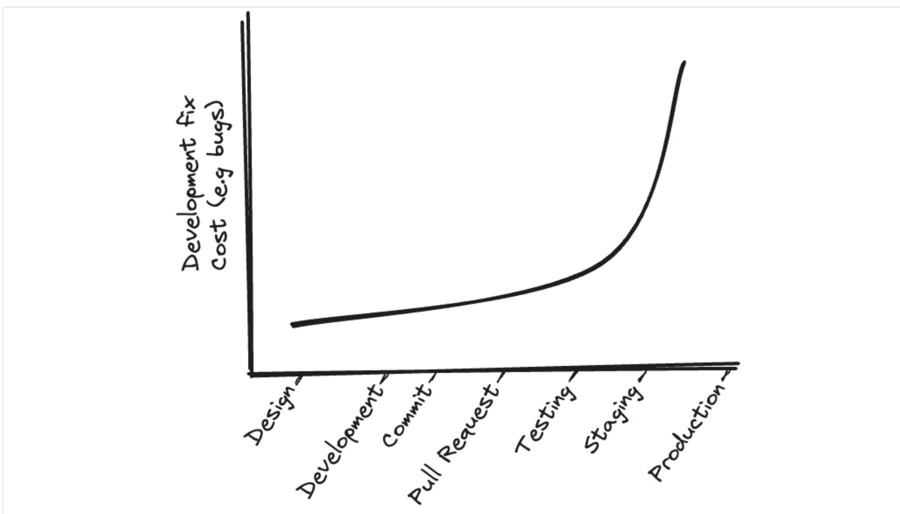

Figure 1.6: Shifting Left Concept in Flutter Development

By embedding these practices early and throughout the Flutter development process, teams can mitigate risks, reduce the cost of late-stage defect remediation, and deliver high-quality, robust applications efficiently.

The "Shifting Left" concept emphasizes integrating these processes early in the development cycle. For Flutter developers, it means conducting tests and quality checks right from the initial stages. This early intervention helps detect and fix issues promptly, reducing the cost and time typically associated with later-stage debugging. Implementing these practices in Flutter improves code quality and enhances the application's reliability and user experience.

1.2.15 Informed Decision-Making in Development

In Flutter and software development, informed decision-making often involves weighing quantifiable factors against more nuanced, non-quantifiable aspects. For example, a developer may need to choose between using state management solutions like BLoC, which offers scalability but with added complexity, versus more straightforward options like `setState`, which are easier to implement but may need to scale better for larger apps.

Additionally, decisions in Flutter are sometimes about something other than measurable elements. Consider implementing a custom widget versus using an existing third-party widget. The decision encompasses not just immediate functionality but also factors like long-term maintenance, the reliability of the third-party package, and its alignment with the app's evolving needs.

Balancing these aspects requires careful consideration of both the quantifiable impacts and the less tangible, yet equally important, long-term implications of development choices in Flutter.

1.3 Lifecycle of Flutter Development

Before adapting it for Flutter development, let's understand the Software Development Lifecycle (SDLC). SDLC is a structured framework that defines a series of stages for building and delivering software applications. It provides a roadmap for developers and stakeholders, ensuring quality, efficiency, and predictability throughout development.

There are various SDLC models, each with its specific stages and emphasis. Some popular models include:

- **Waterfall Model:** This linear, sequential model follows a strict stage-gate approach, where each stage must be completed before moving to the next. It is efficient for precise requirements and controlled environments.
- **Agile Model:** This iterative and incremental model emphasizes flexibility and adaptability. It breaks down development into smaller cycles (sprints), enabling continuous feedback and delivery of working software.

- **Spiral Model:** This risk-driven model combines Agile's iterative nature with Waterfall's control. It involves risk assessment throughout the development cycle, making it suited for high-risk projects.

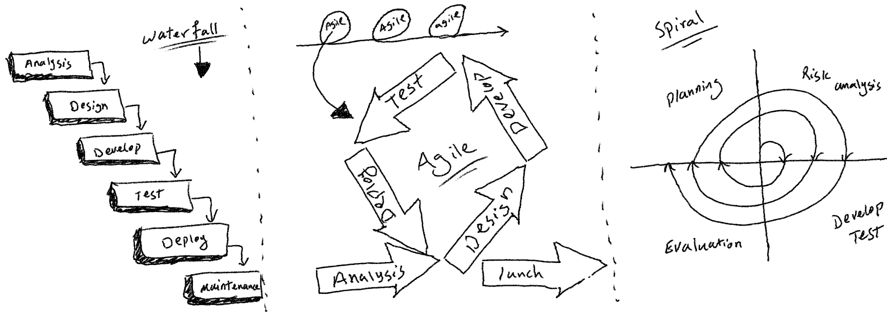

Figure 1.7: Waterfall vs. Agile vs. Spiral Models

Regardless of the chosen model, the core stages of an SDLC typically include:

1. **Analysis**: This phase involves understanding the specific needs and objectives of the Flutter app. It includes gathering detailed requirements from stakeholders and defining the scope of the app.
2. **Design**: Based on the requirements, the overall system architecture for the Flutter app is designed. This includes deciding on the app's navigation flow, state management approach, and overall UI/UX design.
3. **Development**: Here, the actual coding of the Flutter app takes place. Developers write Dart code to implement the defined functionalities, adhering to the design specifications. Developers should write unit and widget tests as part of their best practice in writing code.
4. **Testing**: In this critical phase, the Flutter app undergoes various tests to ensure quality and performance. This includes integration testing and potentially user acceptance testing to validate all aspects of the app.
5. **Deployment**: Once testing is complete and the app is bug-free, it is deployed to the appropriate platforms (e.g., Google Play Store, Apple App Store). This might involve setting up CI/CD pipelines for efficient deployment processes.
6. **Maintenance**: Post-deployment, the app enters the maintenance phase, where it is updated regularly, bugs are fixed, and new features are added as per user feedback or changing requirements. Monitoring the application is another part of this phase. Monitoring is related to crash reporting and bugs, analytics, performance measurement, etc.

In adapting the Software Development Life Cycle (SDLC) for Flutter development, a few specific considerations come into play to leverage the unique features of the framework. During the Requirement Analysis phase, a mobile-first approach is key, but with an eye on potential expansion to web and desktop, thanks to Flutter's

versatility. Flutter's hot reload feature facilitates rapid prototyping and iterative feedback, while performance requirements for animations and responsiveness are crucial for diverse device compatibility.

As the process moves into System Design, Development, Testing, and Deployment, selecting appropriate widgets and state management solutions tailored to the application's complexity becomes vital. Dart language features, like null safety and best practices in widget hierarchy and code organization, ensure clarity and efficiency. Testing, a critical phase, encompasses unit, widget, and integration testing to ensure stability and user-friendliness, with performance testing to optimize the app across devices. Finally, the deployment phase benefits from Flutter's ability to share codebases across platforms facilitated through CI/CD pipelines for efficient multi-platform releases.

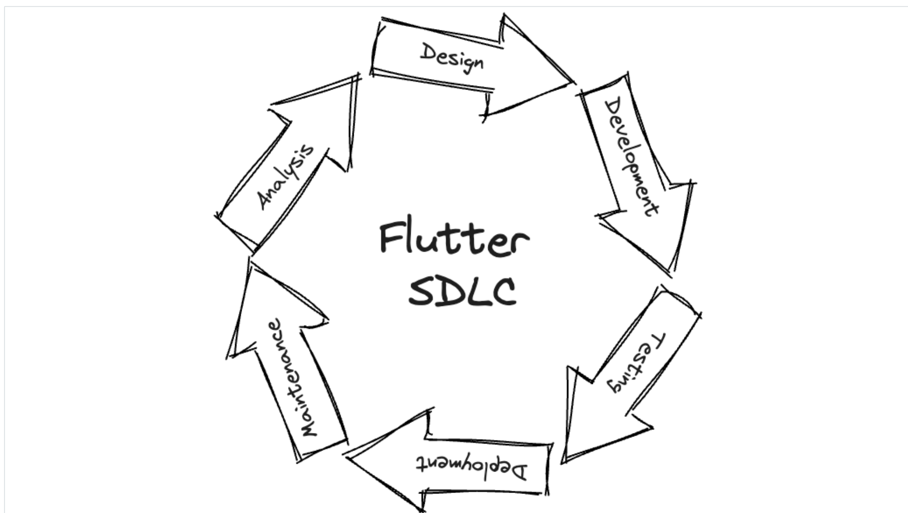

Figure 1.8: Flutter Software Development Life Cycle

Remember, your specific adaptations will depend on the project's size, complexity, and requirements. Choose the tools and practices that best suit your development team and application goals.

1.4 Flutter Engineering vs. Programming

Up to this point, we've explored various facets of software engineering, and at this stage, you should have gained a more comprehensive understanding of this topic. However, I'd like to elaborate further and share my perspective.

In software development, "Flutter engineering" and "programming" represent distinct roles and responsibilities within a project. Programming primarily involves writing code to implement specific functionalities, focusing on code implementation and problem-solving. Programmers are responsible for translating design and

requirements into executable code. In contrast, Flutter engineering encompasses a more comprehensive role. Flutter engineers write code, design the system architecture user interface, and make strategic project structure and scalability decisions. They focus on code quality, project management, and innovation, playing a pivotal role in the development process. Understanding these differences is essential for effectively managing a Flutter project and assembling the right team for success.

1.5 Flutter's Position in Tech Evolution

As we conclude this chapter, I'd like to express my perspective on the position of Flutter in the ever-evolving landscape of technology. Flutter occupies a truly unique and exciting position in the tech world.

1.5.1 Multi-platform Approaches

Flutter emerges as a true industry disruptor in an era where multi-platform development has gained paramount importance. Its capability to seamlessly empower developers to craft high-quality, visually captivating applications across diverse platforms represents a paradigm shift. The framework's focus on productivity, creativity, and efficiency has revolutionized our approach to application development, democratizing access for developers and businesses of all scales. The notion that "wherever there's a pixel, Flutter can be found" has fundamentally altered our perceptions of building multi-platform software from a single code base.

In this context, Flutter is a technological enabler and a catalyst for developer growth. It encourages developers to broaden their knowledge and skillsets across various platforms, each with unique attributes. This approach enhances the quality of apps, elevating the user experience and fostering the evolution of developers into seasoned professionals within their field.

1.5.2 Flutter's Role in Broader Tech Ecosystem

Furthermore, Flutter's role in the broader tech ecosystem holds significant importance. It streamlines the complexities of cross-platform development, catalyzing innovation by promoting the creation of visually appealing, responsive, and consistently exceptional user experiences. As we delve deeper into the world of Flutter development in the upcoming chapters, it becomes apparent that Flutter is not merely a tool; it's a driving force that pushes the boundaries of what is achievable in software development.

Flutter's innovative methods for multi-platform development allow developers to focus on creating user experiences rather than on specific technology or platforms. Additionally, the vibrant Flutter community plays a pivotal role in shaping the technology, fostering greater demand and innovation. Flutter is a remarkable role model to its peers and the broader technology industry as it continues to evolve and leave a lasting mark on the world of software development.

Flutter is relevant in the present and is set to shape the future.

1.6 Conclusion

This chapter has comprehensively explored the fundamental principles and unique philosophy that drive high-quality Flutter development. We've delved into critical paradigms, including abstraction, encapsulation, design patterns, and considerations of efficiency and scalability. The concept of "shifting left" through early verification and validation has been emphasized, setting the stage for informed decision-making throughout the development lifecycle.

Furthermore, we've underlined the distinction between programming and engineering, showcasing how Flutter promotes modularity, separation of concerns, and thoughtful trade-off analysis. We've also situated Flutter within the broader tech landscape, shedding light on its strengths and potential impact compared to other multi-platform approaches.

As we wrap up this chapter, we must ask the following question: How can we harness these foundational insights to create high-performance, maintainable, and aesthetically pleasing Flutter applications? This question will be our guiding star as we embark on the exciting journey of applying these principles in practice and crafting exceptional Flutter experiences in the following chapters.

CHAPTER 2

Custom Painters and Shaders

Reviewer: Renan C. Araújo

The world of custom painting and shaders in Flutter is vast and limitless. It offers remarkable flexibility that goes beyond conventional frameworks. The purpose of this chapter is not only to help you become a professional Flutter developer who can use canvas and shaders to create generative animated art but also to provide a comprehensive understanding of the engineering aspects of these features. We will explore when and how you can utilize these powerful tools in Flutter to enhance your applications while balancing technical ability with creative expression. It would require a book or more to thoroughly cover these topics, which could be the focus of my next book. Alternatively, please email me if you know these areas and would like to collaborate with me on writing about them.

Let's begin now.

2.1 The Art of CustomPainter

`CustomPainter` is a canvas for drawing custom designs in a Flutter application. The `CustomPaint` widget in Flutter is a gateway to creating visually stunning and unique user interfaces. At its core, `CustomPaint` is a widget that provides a canvas to draw custom graphics. It bridges the high-level world of Flutter widgets and the low-level operations of drawing and rendering.

But why and when should you use `CustomPainter`? The key lies in its flexibility and control. It's ideal for scenarios where you must create complex, custom graphics that can't be achieved with standard widgets. This includes scenarios like generating dynamic shapes, creating intricate animations, or implementing custom UI elements that must be visually distinct and interactive. `CustomPainter` shines in applications that require a high degree of customization in the UI, such as games, data visualization tools, or any app that wants to stand out with a unique visual identity.

Sometimes, you may need to use `CustomPainter` to optimize your application's UI. For instance, if you have a complex UI that can be created using standard

widgets, it might cause lags or consume a lot of energy and CPU. Using lower-level APIs in `CustomPainter` can help you optimize your app even more, though this may come at the cost of higher complexity in understanding and writing code. So, if you need to improve your app's performance, `CustomPainter` is an excellent option.

In short, CustomPainter is about painting points on the screen as desired. CustomPainter performs better than other widgets in Flutter because it bypasses the complex widget layout mechanism that Flutter usually uses. This allows the author to control what the canvas will do. A similar concept can be found in fragment shaders, where the author bypasses the Flutter framework and engine. But remember, as Winston Churchill once said: "Where there is great power, there is a great responsibility."

2.1.1 CustomPaint widget

To better understand CustomPaint in Flutter, let's review its fundamental structure and usage. The foundation is built upon a custom class that extends `CustomPainter`, as shown in the snippet:

```
class AwesomePainter extends CustomPainter {
  const AwesomePainter();

  @override
  void paint(Canvas canvas, Size size) {}

  @override
  bool shouldRepaint(
    covariant CustomPainter oldDelegate,
  ) =>
      false;
}
```

This `AwesomePainter` class grants access to a `Canvas` object, your playground, for custom drawing. The `paint` method is where all the magic happens. Here, you can draw anything from simple shapes to complex graphics on the canvas using various drawing methods provided by the `Canvas` API. The size parameter gives you the dimensions of the area you have for drawing.

The `shouldRepaint` method, the other method, is crucial for optimizing the widget's performance. It determines whether the `CustomPainter` should repaint itself. For instance, returning `false` means the canvas will not repaint unless explicitly told, which is beneficial for static graphics.

Now, to use this custom painter, you wrap it within a `CustomPaint` widget, like so:

```
CustomPaint(
  painter: const AwesomePainter(),
)
```

The `CustomPaint` widget integrates your custom drawing (`AwesomePainter` in this case) into the Flutter widget tree. When you use `CustomPaint` and pass in your `AwesomePainter`, Flutter knows to call the `paint` method of `AwesomePainter` whenever it needs to render the widget.

2.1.2 Drawing App

In this example, I want to ensure you understand how easy it is to use Canvas, even if it is your first use. Don't be afraid to try!

Step 1: Define the Custom Painter (`DrawingPainter`)

```
class DrawingPainter extends CustomPainter {
  List<Offset> points;

  DrawingPainter(this.points);

  @override
  void paint(Canvas canvas, Size size) {
    final pencil = Paint()
      ..color = Colors.black
      ..strokeWidth = 4
      ..isAntiAlias = true
      ..strokeCap = StrokeCap.round;

    for (int i = 0; i < points.length - 1; i++) {
      canvas.drawLine(
        points[i],
        points[i + 1],
        pencil,
      );
    }
  }

  @override
  bool shouldRepaint(
    DrawingPainter oldDelegate,
  ) => true;

  @override
  bool shouldRebuildSemantics(
    DrawingPainter oldDelegate,
  ) => false;

  @override
  bool? hitTest(Offset position) {
    return super.hitTest(position);
  }
}
```

`DrawingPainter` extends `CustomPainter` and is responsible for rendering the drawing on the canvas. The Constructor takes a list of `Offset` points, representing the positions where the user has touched the screen.

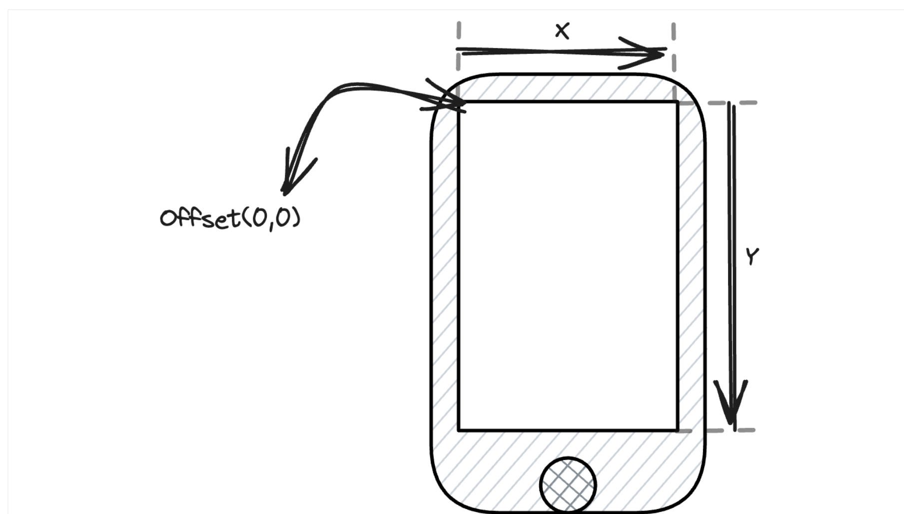

Figure 2.1: Start Point (0,0) Top Left Corner Screen

`paint` method iterates through the points, drawing a line between each consecutive point. The `Paint` object defines the appearance of the lines (color, stroke width, anti-aliasing, and stroke cap); you can think of it as your pencil. And finally, the `drawLine` method is used, which draws a line between the given points using the given paint.

`shouldRepaint` **method** returns `true`, ensuring the canvas repaints whenever the points list updates, and `shouldRebuildSemantics` and `hitTest` methods are related to accessibility and hit-testing, respectively. These work similarly to what you learned in Part 1 of the book while building custom `RenderObject`.

Step 2: Define the Widget (`DrawingPage`)

```
class DrawingPage extends StatefulWidget {
  const DrawingPage({super.key});

  @override
  DrawingPageState createState() => DrawingPageState();
}

class DrawingPageState extends State<DrawingPage> {
  List<Offset> points = <Offset>[];

  @override
  Widget build(BuildContext context) {
```

```dart
    return Scaffold(
      appBar: AppBar(
        title: const Text('DrawingPage'),
      ),
      body: GestureDetector(
        onPanStart: (details) =>
            points.add(details.localPosition),
        onPanUpdate: (
          DragUpdateDetails details,
        ) {
          points.add(details.localPosition);
          setState(() {});
        },
        onPanEnd: (DragEndDetails details) {
          points.add(Offset.infinite);
        },
        child: SizedBox(
          width: double.infinity,
          height: double.infinity,
          child: CustomPaint(
            painter: DrawingPainter(points),
            child: Container(),
          ),
        ),
      ),
    );
  }
}
```

This state class maintains a list of `Offset` points, updated whenever the user draws on the screen, which is given by `onPan` callbacks via `GestureDetector`.

```dart
GestureDetector(
  onPanStart: (details) => points.add(details.localPosition),
  onPanUpdate: (DragUpdateDetails details) {
    points.add(details.localPosition);
    setState(() {});
  },
  onPanEnd: (DragEndDetails details) {
    points.add(Offset.infinite);
  },
  // ... [CustomPaint]
)
```

`onPanStart` Listener adds a new point to the list when the user starts dragging. `onPanUpdate` Listener Continuously adds points to the list as the user moves their finger and calls `setState` to trigger a rebuild. `onPanEnd` Listener adds a special `Offset.infinite` point, signaling the end of a continuous stroke.

The current list of points is passed to `DrawingPainter`, who draws the lines on the canvas.

Figure 2.2: Simple Drawing App with Flutter

Nothing too fancy is happening here, just a few basic lines of code. And voila! You now have a canvas that can be used to paint anything on. I am showing you how to combine some basic elements to create an interesting concept using a powerful API in Flutter. Now, let's explore more advanced best practices.

2.1.3 Optimize

In the example provided, there are several key improvements and best practices that I would like to highlight for a more efficient implementation.

Firstly, let's consider the `CustomPainter` class in Flutter:

```
abstract class CustomPainter extends Listenable {
  const CustomPainter({Listenable? repaint})
      : _repaint = repaint;

  final Listenable? _repaint;

  // ...
}
```

In this snippet, it's noteworthy that `CustomPainter` can automatically manage repainting when provided with a `Listenable`. This feature enables us to simplify our widget structure by transitioning from a `StatefulWidget` to a `StatelessWidget`, utilizing a `ValueNotifier`:

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

  final pointsListenable = ValueNotifier<List<Offset>>([]);
  // ...
}
```

Within this stateless widget, changes to the drawing can be easily managed by updating the `ValueNotifier` in response to user gestures:

```
class DrawingPage extends StatelessWidget {
  //...
  body: GestureDetector(
    onPanStart: (details) => pointsListenable.value = [
      ...pointsListenable.value,
      details.localPosition
    ],
    onPanUpdate: (DragUpdateDetails details) {
      pointsListenable.value = [
        ...pointsListenable.value,
        details.localPosition
      ];
    },
    onPanEnd: (DragEndDetails details) {
      pointsListenable.value = [
        ...pointsListenable.value,
        Offset.infinite,
      ];
    },
    //...
  );
  // ... ;
}
```

Additionally, we can optimize the repaint process by passing this notifier to the custom painter and wrapping the painter with `RepaintBoundary`.

I suggest you review the official documentation and source code to learn about `RepaintBoundary`. Explaining the details would take a lot of pages. However, I can tell you that to display a colored border around each widget; you need to set the `debugRepaintRainbowEnabled` property to `true`. These borders will change color as the user scrolls through the app. To set this flag, add `debugRepaintRainbowEnabled = true;` as a top-level property in your app. If you see that static widgets rotate through colors after setting this flag, consider adding repaint boundaries to those areas.

```
main() {
  **debugRepaintRainbowEnabled = true;**
```

```
  runApp(MyApp());
}
```

RepaintBoundary ensures that the repainting is contained and does not affect other parts of the Flutter widget tree unnecessarily:

```
child: RepaintBoundary(
          child: CustomPaint(
            painter: DrawingPainter(pointsListenable),
            child: const SizedBox.expand(),
          ),
        ),
```

Furthermore, the implementation of the CustomPainter itself is crucial. Here's how it can be adapted to leverage the ValueNotifier:

```
class DrawingPainter extends CustomPainter {
  ValueNotifier<List<Offset>> points;

  DrawingPainter(this.points) : super(repaint: points);
  // ...
  @override
  void paint(Canvas canvas, Size size) {
    for (int i = 0; i < points.value.length - 1; i++) {
      canvas.drawLine(
        points.value[i],
        points.value[i + 1],
        pencil,
      );
    }
  }

  @override
  bool shouldRepaint(DrawingPainter oldDelegate) =>
      oldDelegate.points != points;
  // ...
}
```

This modification greatly enhances performance, especially in larger-scale applications. It's an important optimization technique for a more efficient and effective Flutter development experience.

This is the perfect moment to introduce you to the next section on best practices.

2.1.4 Best Practices

CustomPainter offers immense flexibility for crafting custom visuals in Flutter, but mastering its efficiency and maintainability requires thoughtful practice. Let's dive into essential best practices with code examples for a deeper understanding:

Minimize Rebuilds with `shouldRepaint`:

Imagine your CustomPainter draws a dynamic chart. Every data tweak triggers a complete redraw, impacting performance. `shouldRepaint` come to the rescue!

```
class ChartPainter extends CustomPainter {
  final List<double> data;

  ChartPainter(this.data);

  @override
  bool shouldRepaint(ChartPainter oldDelegate) {
    // Compare only data changes
    return !listEquals(
      data,
      oldDelegate.data,
    );
  }

  @override
  void paint(Canvas canvas, Size size) {
    // ... draw chart based on data
  }
}
```

With this code, the chart repaints only when the data changes, not for minor UI tweaks, boosting performance.

Cache with `PictureRecorder` for Smooth Animations:

Think of a complex animated scene with static background elements. Repainting them every frame is redundant. Enter `PictureRecorder`:

```
class AnimatedPainter extends CustomPainter {
  PictureRecorder recorder = PictureRecorder();
  Picture? picture;

  // ... other methods

  @override
  void paint(Canvas canvas, Size size) {
    if (picture == null) {
      return;
    }
    canvas.drawPicture(picture!);

    // ... draw dynamic animation elements on top
  }
```

```
  @override
  bool shouldRepaint(covariant CustomPainter oldDelegate) {
    // ...
  }
}
```

Here, the static background gets recorded once and reused repeatedly, making animations smoother and less resource-intensive.

Separate Logic with Mixins for Code Reuse:

Sometimes, your painting logic becomes complex. Mixing it with widget management can get messy. Mixins to the rescue!

```
mixin ShapePainterMixin on CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
    // ... common painting logic for drawing shapes
  }
}

class MyPainter extends CustomPainter
    with ShapePainterMixin {
  // ... other properties and methods
}
```

This mixin encapsulates reusable shape-drawing logic, keeping your `MyPainter` class focused on specific details and facilitating code reuse across other painters.

Modularize with Reusable Painters for Large Compositions:

Large, intricate visuals benefit from a divide-and-conquer approach. Enter modular painters:

```
class ChartPainter extends CustomPainter {
  // ... paint the chart
}

class GaugePainter extends CustomPainter {
  // ... paint the gauges
}

class LabelPainter extends CustomPainter {
  // ... paint the text labels
}

class DashboardPainter extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
    ChartPainter().paint(canvas, size);
```

```
    GaugePainter().paint(canvas, size);
    LabelPainter().paint(canvas, size);
  }
}
```

Here, individual painters handle specific elements; the `DashboardPainter` combines them seamlessly for a complete picture. This promotes modularity and simplifies maintenance.

Enhance Accessibility with Semantics:

Accessibility ensures everyone can use your app. Semantics help screen readers understand your custom visuals:

```
class ChartPainter extends CustomPainter {
  // ...

  @override
  void paint(Canvas canvas, Size size) {
    // ... draw chart elements

    for (int i = 0; i < data.length; i++) {
      final rect = Rect.fromLTWH(
        i * 10,
        0,
        10,
        data[i] * 50,
      ); // Example rect for data point
      **CustomPainterSemantics(
        rect: rect,
        properties: SemanticsProperties(
          label: 'Data point ${i + 1}',
          value: '${data[i]}',
        ),**
      ).paint(canvas, size);
    }
  }
}
```

With semantics, screen readers can interpret data points and values, making your custom charts accessible to everyone.

Reuse Paint Objects:

Instantiate `Paint` objects outside of the `paint` method and reuse them. This practice conserves resources as creating a new `Paint` instance on each repaint can be costly.

```
final paint = Paint()
  ..color = Colors.blue
```

```
  ..style = PaintingStyle.stroke;

void paint(Canvas canvas, Size size) {
  // use existing paint object
  canvas.drawLine(
    Offset.zero,
    Offset(size.width, size.height),
    paint,
  );
}
```

Handling High-DPI Screens:

Ensure your custom painting code scales correctly on high-DPI screens for a consistent visual experience across devices.

```
final pixelRatio = MediaQuery.of(context).devicePixelRatio;

void paint(Canvas canvas, Size size) {
  // Scale your drawing based on pixelRatio
}
```

Profiling and Optimization:

Regularly profile your custom painter code, especially when dealing with complex drawings or animations, to identify and optimize performance bottlenecks.

Utilizing RepaintBoundary for Performance Optimization:

`RepaintBoundary` is a widget that isolates its child from the rest of the widget tree in terms of painting. This can significantly improve performance, especially for widgets that are expensive to paint and only change sometimes. By using `RepaintBoundary`, you tell Flutter to handle the painting of this widget separately, reducing the overall repaint cost.

```
RepaintBoundary(
  child: CustomPaint(
    painter: ExpensivePainter(),
    isComplex: true,
    willChange: false,
  ),
)
```

This code wraps `ExpensivePainter ()` in a `RepaintBoundary`, optimizing its rendering performance. The `isComplex` and `willChange` properties further inform the rendering system about the nature of the painting operation, allowing for more efficient handling.

Vertices - Advanced Shape Rendering with high performance:

In graphics programming, vertices are the cornerstone of rendering shapes and models. A vertex is a point in either 2D or 3D space, defined by coordinates, and serves as the fundamental unit for constructing more complex geometrical shapes. Vertices are crucial in defining the outlines of polygons, particularly triangles, which are the basic building blocks for most graphical objects in two-dimensional and three-dimensional environments. These points can also carry additional attributes such as color, texture coordinates, and normals, essential for creating detailed and visually rich graphics.

In Flutter, the `Vertices` class is crucial for custom shape rendering, especially when dealing with intricate designs or the need for high-performance graphics. This class allows developers to define a series of points and how they should be connected and colored. When used with the `Canvas.drawVertices` method, it enables the drawing of complex shapes that are not achievable with standard widgets.

The shapes drawn using `Vertices` are primarily based on triangles, the simplest polygon in graphics programming, and can be combined to form any complex shape. The way these triangles are formed and rendered depends on the mode specified:

1. **Triangles:** Each set of three vertices forms an independent triangle.
2. **Triangle Strip:** Vertices are connected in a strip-like fashion, where each new vertex forms a triangle with the previous two.
3. **Triangle Fan:** All triangles share the first vertex, fanning out from this common point.

The `Vertices` class can be instantiated using either its default constructor or the `Vertices.raw` constructor for more direct control over the data. Here's a basic overview of how to use each:

```
Vertices(
  VertexMode mode,
  List<Offset> positions, {
    List<Color>? colors,
    List<Offset>? textureCoordinates,
    List<int>? indices,
})
```

This constructor is more straightforward and uses lists of `Offset` and `Color` objects, making it user-friendly but slightly less efficient.

```
Vertices.raw(
  VertexMode mode,
  Float32List positions, {
    Int32List? colors,
    Float32List? textureCoordinates,
    Uint16List? indices,
})
```

`Vertices.raw` is a more performance-oriented constructor using typed data arrays like `Float32List` and `Uint16List`. This approach is closer to the low-level data format the rendering engine uses. It is ideal for performance-critical applications or when working with data already in a raw format.

One example in the source code provided in this book allows you to navigate to `/lib/custompainters/snowfall.dart`. Once you run the application, you can turn `Vertices.raw` implementation on and off and fall back to standard canvas drawing to observe the difference in performance.

2.2 Exploring Shaders

Shaders are small but powerful programs that run on the graphics processing unit (GPU), responsible for rendering the intricate details of light and color that bring digital images to life.

Shaders operate within the graphics pipeline, a sequence of steps that a computer graphics system uses to render 3D objects onto a 2D screen. This pipeline includes stages like vertex processing, rasterization, and fragment processing, with shaders playing a vital role at various points. There are several types of shaders, each with its unique function:

1. **Vertex Shaders**: These process each vertex of a 3D model, transforming 3D coordinates into 2D screen coordinates and passing per-vertex data down the pipeline. They are essential for manipulating vertex positions and creating effects like animations.
2. **Fragment (Pixel) Shaders**: These calculate the final color of each pixel, factoring in lights, shadows, and textures. They are key to achieving detailed surface effects and realistic rendering of materials.
3. **Geometry Shaders**: Operating between vertex and fragment shaders, they can generate new vertices and shapes, adding complexity and detail to objects.
4. **Tessellation Shaders**: Used for adjusting the level of detail of 3D models, these shaders enhance efficiency and visual quality by adapting to the camera's distance.
5. **Compute Shaders**: These handle general-purpose computing tasks within the GPU, like physics simulations or post-processing effects, separate from the direct image rendering process.

The graphics pipeline is a conceptual framework describing the steps to render a 3D object onto a 2D screen. It starts with processing the 3D coordinates (vertex processing), then turning the object into pixels (rasterization), and finally coloring these pixels (fragment processing). Shaders are integral to this process, providing the flexibility to create complex visual effects.

2.2.1 Understanding Shader Language (GLSL)

OpenGL Shading Language (GLSL) is a high-level shading language used widely in computer graphics for writing custom shaders. Shaders are small programs that dictate how to graphically render each pixel, vertex, or geometry on the screen. They are executed directly on the graphics processing unit (GPU), making them incredibly efficient for graphics computations.

GLSL is an integral part of the OpenGL graphics API, a standard specification defining a cross-language, cross-platform API for rendering 2D and 3D vector graphics. The language closely resembles C and, in essence, Dart-like syntax, making it familiar to Flutter developers.

The primary types of shaders in GLSL include **vertex shaders**, which process vertex data, and **fragment shaders**, which determine each pixel's color and other attributes. Other types, like geometry and tessellation shaders, offer additional control over rendering. I will focus on Fragment shaders as Flutter only supports that.

A typical fragment shader, which usually has `.frag` or `.glsl` extension in GLSL, includes:

Version Declaration: Specifying the GLSL version. This is optional.

Output Variable: A variable to store the color output for the pixel.

```
out vec4 fragColor;
```

`vec4` is the most common output variable type used in fragment shaders. It represents a four-component vector corresponding to the color's RGBA (Red, Green, Blue, Alpha) components. Sometimes, you might encounter `vec3` if the alpha component is not needed or is handled separately. It represents a three-component vector for the RGB components of the color. In cases where only a single color channel or a grayscale output is needed, a `float` can be used.

Main Function: Where the color calculation happens.

```
void main() {
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);   // Red color
}
```

Here, `fragColor` is set to a static red color for every pixel.

Incorporate Uniform Variables: In GLSL, uniform variables are a type of variable that you can use to pass data from your main application (running on the CPU) to your shader program (running on the GPU). Uniforms are global and remain constant for all vertices and fragments processed during a single draw call. They are a key way to make your shaders dynamic and responsive to what's happening in your application.

For example, you can pass the time elapsed to create animations.

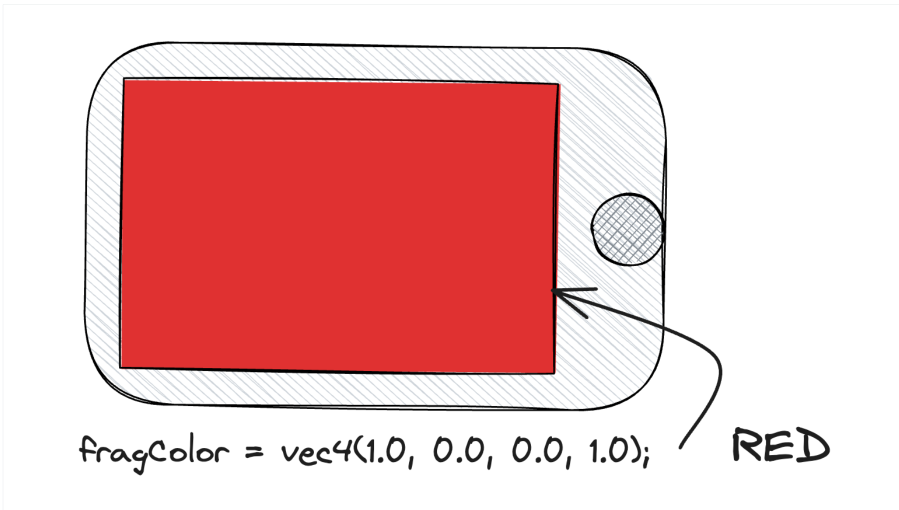

Figure 2.3: Demonstrate Red for Every Pixel

```
uniform float u_time;
void main() {
        // The **sin** function creates a wave-like pattern,
        // and **abs** ensure the value is positive.
    float red = abs(sin(u_time));
    fragColor = vec4(red, 0.0, 0.0, 1.0);
}
```

The fragment shader will produce a pulsating red color on the rendered object, with the intensity of red changing over time in a sinusoidal pattern[1].

Manipulate Coordinates: Use the coordinates of each pixel (`gl_FragCoord`) to create gradients or patterns. `gl_FragCoord` provides the coordinates of the current fragment (or pixel).

In the following example, these coordinates are divided by `u_resolution`, a uniform variable passed to the shader representing the resolution of the rendering window or texture. The result is a normalized coordinate `st` (with both x and y values ranging from 0 to 1) across the rendered surface.

```
uniform vec2 u_resolution;
out vec4 fragColor;

void main()
```

[1] https://www.shadertoy.com/view/XclSWr

```
{
    vec2 st = gl_FragCoord.xy / u_resolution;
    fragColor = vec4(st.x, st.y, 0.0, 1.0);
}
```

This shader will produce a gradient effect[2] across the rendered surface, smoothly transitioning in color based on the pixel's position. The gradient will blend from black at the bottom-left corner to red and green at the top-right corner.

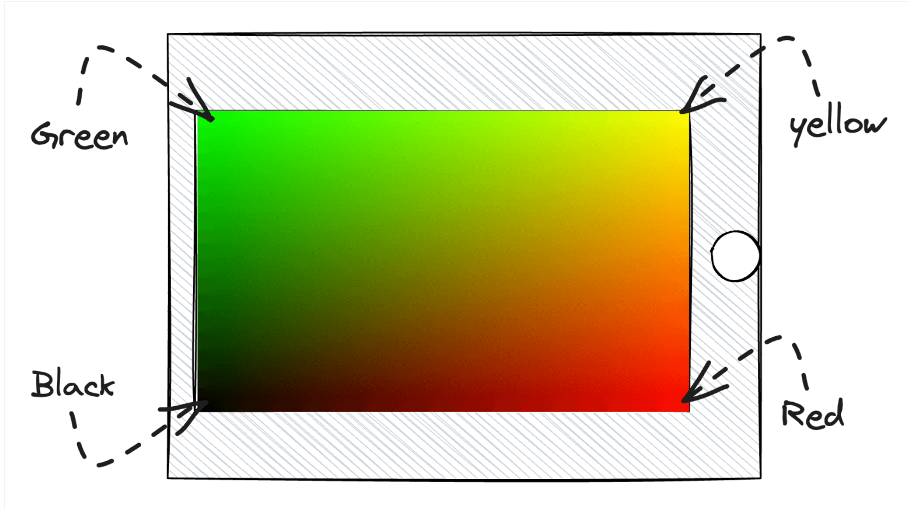

Figure 2.4: Manipulate Coordinates to Create Gradient Effect

Complex Lighting and Color Effects: Here, we're calculating the diffuse component of lighting based on a light direction and surface normal.

```
uniform vec3 u_lightDirection;
uniform vec3 u_normal;
void main() {
    float diff = max(dot(u_normal, u_lightDirection), 0.0);
    // Diffuse lighting effect
    vec3 diffuse = diff * vec3(1.0, 0.5, 0.3);
    fragColor = vec4(diffuse, 1.0);
}
```

So, as you can see, there are also available helper functions. You can also create a function to define `const` variables.

[2]https://shadertoy.com/view/4csSWr

Texture Sampling: In shaders, textures are images that add surface details like color, patterns, or bumps to 3D models.

In the example, A `sampler2D` uniform in a shader is used to pass a 2D texture from your application to the shader. The term "sampler" refers to the functionality in a shader that allows it to read or sample data from a texture. The shader then samples colors from this texture to apply to the rendered surface.

```
uniform sampler2D u_texture;
void main() {
    vec4 texColor = texture(u_texture, gl_FragCoord.xy);
    fragColor = texColor;
}
```

There are also other sampler types for different kinds of textures, like `sampler3D` for 3D textures and `samplerCube` for cube map textures, each used for specific rendering techniques.

This should give you an overview of how to read the shader's GLSL code. However, we have just touched the surface, and the best is to read a few shader examples, particularly fragment shaders, to get more familiar with the concept.

2.3 Using Shaders in Flutter

Now that you understand what shaders are and how they play a role in computer graphic programming let's explore how Flutter leverages them. Flutter incorporates Fragment shaders for enhanced visual effects. As a Flutter engineer, understanding how to integrate and use shaders can significantly elevate the visual appeal of your applications.

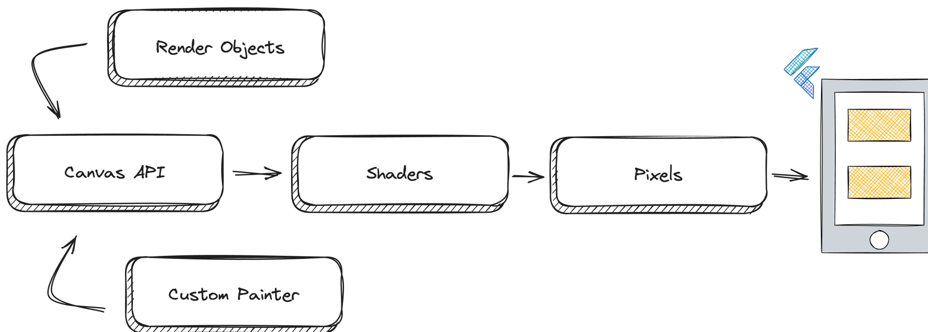

Figure 2.5: Widgets to Pixels Simple Pipeline in Flutter

2.3.1 Adding a Fragment Shader to Flutter

In Flutter, you primarily work with fragment shaders. Here's how you can add a `.frag` shader file to your Flutter project:

Creating the Shader File (`simple.frag`): Create a fragment shader file with your GLSL code. Here's an example shader that creates a gradient using Flutter's brand colors:

```
#version 460 core

#include <flutter/runtime_effect.glsl>

uniform vec2 u_surfaceSize;

out vec4 fragColor;

vec3 flutterBlue = vec3(5, 83, 177) / 255;
vec3 flutterNavy = vec3(4, 43, 89) / 255;
vec3 flutterSky = vec3(2, 125, 253) / 255;

void main() {
  vec2 st = FlutterFragCoord().xy / u_surfaceSize.xy;

  vec3 color = vec3(0.0);
  vec3 percent = vec3((st.x + st.y) / 2);

    color = mix(
    mix(flutterSky, flutterBlue, percent * 2),
    mix(flutterBlue, flutterNavy, percent * 2 - 1),
    step(0.5, percent));

  fragColor = vec4(color, 1);
}
```

- Any GLSL version from 460 to 100 is supported in Flutter, though some available features are restricted.
- `#include <flutter/runtime_effect.glsl>`: This line includes declarations for using Flutter-specific features in your shader.
- `FlutterFragCoord()`: A Flutter-specific function that provides the fragment coordinates. Unlike `gl_FragCoord` in traditional GLSL, `FlutterFragCoord` is adjusted for Flutter's coordinate system.
- This Shader needs two `floats` that define surface size, which we can pass from Flutter.

Adding Shader to `pubspec.yaml`: Include your shader file in the `pubspec.yaml` file of your Flutter project:

```yaml
flutter:
  shaders:
    - shaders/simple.frag
```

Loading Shaders at Runtime

One way to use shaders in Flutter is by loading them at runtime:

```
void loadMyShader() async {
  final program = await FragmentProgram.fromAsset(
    'shaders/snow.glsl',
  );
  final program2 = await FragmentProgram.fromAsset(
    'shaders/simple.frag',
  );
}
```

- `FragmentProgram.fromAsset`: Loads the shader from assets.
- You may see both `.frag` and `.glsl` extensions for fragment shaders.

Using Fragment Shaders with Canvas APIs

In Flutter, fragment shaders can be used with Canvas APIs by setting the `Paint.shader` property:

```
void paint(Canvas canvas, Size size) {
  shader.setFloat(0, size.width);
  shader.setFloat(1, size.height);
  **final pencil = Paint()..shader = shader;**

    canvas.drawRect(
    Rect.fromLTWH(0, 0, size.width, size.height),
    paint,
  );
}
```

- In this case, we are just cascading the shader with the `fragment shader` that must be passed down to the custom painter class.
- Shaders can be applied to various shapes and paths drawn on the canvas, offering versatile possibilities for custom graphics. Shaders, in combination with blend modes, can have effects, too!
- Last but not least, since we have to define surface size to float, now we can define it as the max with height—the `setFloat` sets the float uniform at [index] to [value].

Now, we can use everything together in our Flutter app.

```
void main() async {
  // 1
  final fragmentProgram = await FragmentProgram.fromAsset(
    'shaders/simple.frag',
  );
  // 2
```

```dart
  runApp(MyApp(shader: fragmentProgram.fragmentShader()));
}

class MyApp extends StatelessWidget {
  const MyApp({super.key, required this.shader});

  // 3
  final FragmentShader shader;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Simple Shader Demo',
      theme: ThemeData(
        colorSchemeSeed: Colors.blue,
        useMaterial3: true,
      ),
      home: MyHomePage(shader: shader),
    );
  }
}

class MyHomePage extends StatelessWidget {
  const MyHomePage({
    super.key,
    required this.shader,
  });

  final FragmentShader shader;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Simple Shader Demo'),
      ),
      // 4
      body: CustomPaint(
        size: MediaQuery.sizeOf(context),
        // 5
        painter: MyShaderPainter(shader: shader),
      ),
    );
  }
}

// 5
class MyShaderPainter extends CustomPainter {
```

```
  MyShaderPainter({required this.shader});
  // 6
  final FragmentShader shader;

  @override
  void paint(Canvas canvas, Size size) {
    // 7
    shader.setFloat(0, size.width);
    shader.setFloat(1, size.height);

    // 8
    final paint = Paint()..shader = shader;

    // 9
    canvas.drawRect(
      Rect.fromLTWH(0, 0, size.width, size.height),
      paint,
    );
  }

  @override
    bool shouldRepaint(
    covariant CustomPainter oldDelegate,
  ) =>
      false;
}
```

Let's break down what each numbered section in the code is doing:

1. **Shader Initialization**:
   - Loads the fragment shader from the asset `'shaders/simple.frag'`.
   - `FragmentProgram.fromAsset` asynchronously loads the shader program.
   - `fragmentProgram.fragmentShader()` creates a `FragmentShader` instance from the loaded program.
2. **Application Startup**:
   - `runApp` initializes and runs the Flutter application.
   - `MyApp` widget is created with the `shader` passed as a parameter.
3. **Shader Propagation in MyApp**:
   - The `MyApp` class takes a `FragmentShader` instance as a parameter.
   - This shader is then passed down to `MyHomePage`.
4. **CustomPaint Widget in MyHomePage**:
   - The `CustomPaint` widget is used to provide a canvas on which to draw.
   - `MediaQuery.sizeOf(context)` gets the current media (screen) size for the painting area.

5. **Painter for CustomPaint**:
    - `ShaderPainter` is set as the painter for `CustomPaint`.
    - This custom painter uses the provided shader for drawing.
6. **Shader in ShaderPainter**:
    - `ShaderPainter` takes a `FragmentShader` instance.
    - This shader will be used for painting.
7. **Setting Shader Uniforms**:
    - The shader's uniforms are set, which in this case are the width and height of the canvas.
    - These values are provided to the shader to control its behavior based on the canvas size.
8. **Creating Paint with Shader**:
    - A `Paint` object is created, and its shader is set to the provided `FragmentShader`.
    - This paint will be used to draw with the effects of the shader.
9. **Drawing on Canvas**:
    - A rectangle covering the entire canvas is drawn.
    - The `Paint` with the shader applied is used, so the shader effect is rendered across this rectangle.

This process can be way more simplified by using `flutter_shaders`. Let's refactor using this package.

```
void main() {
  runApp(const MyApp());
}

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: MyHomePage(),
    );
  }
}

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

```
      // 1
      **body: ShaderBuilder(
        assetKey: 'shaders/simple.frag',
        (context, shader, child) {
          return CustomPaint(
            size: MediaQuery.sizeOf(context),
            // 2
            painter: ShaderPainter(shader: shader),
          );
        },
      ),**
    );
  }
}
```

We have now cut a few steps and are simply now using `ShaderBuilder` builder widget to load the share for us. You can use other features from the package, such as the `SetUniforms` extension (look at flutter_shaders/lib/src/set_uniforms.dart), where technically, you can set your uniforms much easier and like a charm.

2.3.2 Converting from ShaderToy

Converting ShaderToy shaders to Flutter involves several steps to adapt the code to the Flutter environment. For example, let's consider converting a simple laser effect shader from ShaderToy, which can be found at this ShaderToy link[3].

```
//convert HSV to RGB
vec3 hsv2rgb(vec3 c)
{
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(
      K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y
    );
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 fragPos = fragCoord / iResolution.xy;
    fragPos.y -= 0.5f;

    vec3 color = hsv2rgb(
      vec3(fragPos.x * 0.5 - iTime * 3.0, 1.0, 1.0)
```

[3]https://www.shadertoy.com/view/4f2GRR

```
    );
    color *= (0.015 / abs(fragPos.y));

    color += dot(color, vec3(0.299, 0.587, 0.114));

    fragColor = vec4(color, 1.0);
}
```

ShaderToy shaders use a `mainImage` function, which differs from the standard `main` function used in Flutter's shaders. Additionally, ShaderToy automatically provides certain uniforms like `iResolution` and `iTime`, which you must manually declare in Flutter.

Key Differences and Adjustments

1. **Entry Point**:
    - ShaderToy uses `mainImage` function, while Flutter uses the standard `main` function.
2. **Fragment Coordinates**:
    - ShaderToy shaders use `fragCoord`, but in Flutter, you should use `FlutterFragCoord()`.
3. **Uniforms**:
    - ShaderToy provides `iResolution` and `iTime` uniforms automatically. In Flutter, these must be explicitly declared and passed from the Dart code.
4. **Shader Inclusion**:
    - The `#include <flutter/runtime_effect.glsl>` directive is specific to Flutter and necessary for using certain Flutter-specific features.
5. **Output Variable**:
    - The output color is stored in `fragColor` in both ShaderToy and Flutter.

```
#include <flutter/runtime_effect.glsl>

uniform vec2 iResolution;
uniform float iTime;
out vec4 fragColor;

vec3 hsv2rgb(vec3 c) {
    // ...
}

void main() {
    vec2 fragPos = FlutterFragCoord().xy / iResolution.xy;
    // ...
}
```

2.3.3 Setting Uniforms in Flutter

In your Flutter application, set the uniforms for the shader using `shader.setFloat`:

```
// In your custom painter class
void paint(Canvas canvas, Size size) {
  shader.setFloat(0, size.width);
  shader.setFloat(1, size.height);
  shader.setFloat(2, time);

  final paint = Paint()..shader = shader;

  canvas.drawRect(
    Rect.fromLTWH(0, 0, size.width, size.height),
    paint,
  );
}
```

Here, the `size.width` and `size.height` corresponds to `iResolution`, and `time` corresponds to `iTime`. These values must be passed to the shader to mimic the behavior of ShaderToy's built-in uniforms.

This example illustrates a straightforward shader conversion. However, there are more complex scenarios where additional steps are required. For instance, when dealing with 2D texture samplers, you should use an `AnimatedSampler` from the `flutter_shaders` package in Flutter and call `setImageSampler()` on the `shader` object.

```
// In Flutter
shader.setImageSample(0, image); // pass dart:ui Image to the shader
// in GLSL
uniform sampler2D uTexture;
```

For those eager to delve deeper into shaders and the GLSL language, a great resource is The Book of Shaders[4]. This online book offers an extensive and interactive learning experience, perfect for anyone seeking to expand their knowledge.

2.4 Conclusion

Exploring custom paintings and shaders in Flutter opens up a fascinating domain where creativity meets technology. These powerful APIs enhance your applications' visual appeal and offer an engaging and addictive playground for developers.

[4]http://thebookofshaders.com/

The ability to transform code into captivating visuals is not just a technical skill but an artistic expression, making the development process with Flutter an exciting and rewarding journey.

I encourage you to embrace this opportunity to experiment and innovate. Each raid into custom painting and shaders is a step towards mastering these tools, pushing the limits of what's possible in app design. So, dive in, play around, and let your creativity flourish. The more you explore and create, the more you learn, leading to applications that are not only functional but visually attractive masterpieces.

Closing Words

So, you have come to the end of this book. As you turn the final page, it's not just the conclusion of a chapter in your learning journey but a gateway to new possibilities. Throughout this book, we've navigated through complex concepts, topics, detailed code, and solutions. These pages have enriched your understanding and ignited a spark of curiosity and enthusiasm for the ever-evolving world of Flutter.

Flutter is a continuous journey of learning and discovery. The concepts and techniques you've encountered here are tools that can empower you to build, create, and innovate. Whether you're a student, a professional developer, or a hobbyist, the knowledge you've gained is a solid foundation upon which you can build amazing things.

As you step beyond this book, remember that every line of code you write reflects your creativity and problem-solving skills. The challenges you'll face in the real world might be complex, but the skills you've honed here will be your allies. Keep experimenting, keep learning, and most importantly, keep coding. Practicing makes perfect.

I hope this book will serve as both a guide and a step toward your success with Flutter. May your future code be efficient and elegant and reflect your potential.

Thank you for choosing this book as your companion in your Flutter journey. Here's to many more lines of code, breakthroughs, and innovations in your future endeavors.

Feel free to contact me through social media, email, and my flutterengineering.io newsletter. I am in a mission to write and release more content. I would appreciate your thoughts, feedback, and successes in your career and projects.

Remember: FLUTTER DEVELOPMENT IS FUN.

With best wishes for your continued success.

Majid Hajian