Design And Implementation Of Mobile Applications

Theory

Christian Rossi

Academic Year 2024-2025

Abstract

The course is organized into five distinct parts.

The first part sets the stage by framing the problem and exploring the numerous opportunities that mobile devices present. It also provides a brief overview of various alternatives and competing solutions in the field.

In the second part, we focus on mobile application design, aiming to identify guidelines and recurring patterns that can facilitate the design process and contribute to producing high-quality solutions.

The third part introduces key innovations and features of important frameworks, specifically Flutter and React Native, which are essential for developing cross-platform applications.

The fourth part delves into Android development, covering how to create applications for a range of Android devices, including phones, tablets, watches, and TVs, using Kotlin and Jetpack Compose.

Finally, the fifth part addresses the development of applications for iOS-based devices, utilizing Swift and SwiftUI to create efficient and effective mobile applications.

Contents

CHAPTER 1

Mobile applications

1.1 Introduction

The history of mobile devices began in 1973 when Martin Cooper at Motorola made the first-ever mobile phone call using a prototype, marking a pivotal moment in telecommunications. Over the decades, mobile devices have evolved from simple communication tools to highly advanced, multifunctional systems equipped with a wide array of sensors. Modern smartphones now integrate technologies such as accelerometers, gyroscopes, digital compasses, GPS, barometers, ambient light sensors, and proximity sensors, enabling a range of applications from navigation to augmented reality.

Alongside hardware advancements, mobile programming has grown to support a variety of platforms, each with its own preferred development languages. Key programming languages used in mobile app development include:

- *Objective-C and Swift*: primarily for iOS development.
- *Java and Kotlin*: widely used for Android development.
- *C#*: common for cross-platform development, particularly with frameworks like Xamarin.
- *HTML5 and JavaScript*: popular for web-based and cross-platform apps, especially with frameworks like React Native.
- *Python and other languages*: increasingly used for cross-platform solutions through frameworks such as Kivy or BeeWare.

This diverse ecosystem of languages supports the ever-expanding capabilities of mobile devices and the broad range of applications they power.

1.2 Design principles

Effective mobile app design revolves around several core principles that prioritize usability, clarity, and user engagement. These principles guide the creation of intuitive and enjoyable user experiences:

1. *Keep it brief*: present information concisely to avoid overwhelming the user.

2. *Pictures are faster than words*: visual elements communicate more efficiently than text, improving comprehension and engagement.

3. *Decide for me but let me have the final say*: offer smart defaults and automation to simplify tasks, while still allowing users to override settings if needed.

4. *I should always know where I am*: ensure users have a clear sense of navigation and context within the app, reducing confusion and frustration.

To successfully implement these principles, designers should adopt the following approaches:

- *Mobile mindset*: design with a mobile-first approach, focusing on creating experiences that are streamlined, unique, and user-centered. Apps should be functional, visually appealing, and tailored to mobile usage patterns.

- *Identify different classes of users*: clearly define and understand the needs of your target audience. Users can be categorized into groups such as those who are bored, busy, or lost, each with distinct behaviors and expectations.

- *First Impressions Matter*: capture user interest within the first few seconds of interaction. To make a lasting impact:

  - Provide minimal or no help text, relying instead on intuitive design.
  - Create a distinctive and captivating look and feel.
  - Ensure the app quickly conveys its purpose and value.

CHAPTER 2

Flutter

2.1 Introduction

Flutter has emerged as a crucial framework for mobile and web development, offering a range of advantages that make it a popular choice for cross-platform applications:

1. *Widespread adoption*: Flutter has the highest number of users among cross-platform frameworks, making it a leading solution for multi-platform development.
2. *Google support*: as an open-source framework backed by Google, Flutter benefits from regular updates, community contributions, and long-term stability.
3. *Dart language*: Flutter uses Dart, a programming language similar to Java, making it easy to learn and adopt for developers familiar with object-oriented languages.
4. *Extensive platform support*: Flutter provides a wide codebase solution, supporting development across multiple devices and platforms, including mobile (iOS, Android), web, and desktop.

Flutter's architecture can be understood through three distinct abstraction layers, each playing a key role in the app development process:

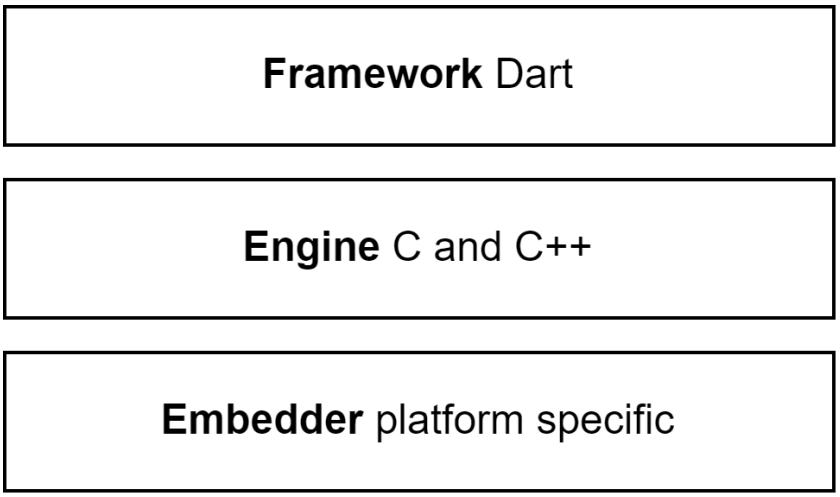

Figure 2.1: Flutter architecture

The architecture consists of the following layers:

- *Embedder*: the embedder is responsible for integrating Flutter's source code into specific platforms, such as iOS, Android, or desktop environments. This layer, developed by Google, ensures that Flutter can operate seamlessly across diverse platforms.
- *Engine*: the engine manages the execution of the application on various devices, handling the compilation process and ensuring that the code is optimized for each target platform.
- *Framework*: written in Dart, the framework is the layer that developers interact with the most. It provides the tools and libraries necessary to create the application's user interface, manage state, and implement core functionality.

2.2 Dart

Dart is an object-oriented programming language developed by Google, designed to be easy to learn and highly efficient for building applications across various platforms. Key features of Dart include:

- *Object-oriented design*: Dart is a class-based language with single inheritance, utilizing a familiar C-style syntax that makes it accessible to developers with backgrounds in languages like Java or C++.
- *Rich type system*: Dart supports interfaces, abstract classes, generics, and both optional and strong typing, providing flexibility while ensuring type safety.
- *Everything is an object*: in Dart, everything is treated as an object, including numbers, functions, and even null values. This uniformity simplifies programming and enhances code consistency.
- *Optional type annotations*: while Dart supports type annotations, they are optional, allowing developers to choose how explicitly they want to define types.
- *Function and variable declarations*: Dart allows for the declaration of top-level functions as well as functions associated with classes. Similarly, it supports both top-level variables and class-bound variables.
- *Private identifiers*: any identifier that starts with an underscore is considered private to its library, encapsulating functionality and enhancing modularity.

These features make Dart a powerful choice for developers working with Flutter, enabling them to create robust and efficient applications.

2.3 Dart applications

Creating a Flutter application can be done efficiently using the command line interface. To start a new application, run the following command:

```
flutter create app_name
```

This command generates a directory structure containing essential subdirectories and files, some of which are automatically created for different platforms, while others are critical for development. The most important components include:

- `lib` contains all your handwritten code, including the main Dart files that define your app.
- `pubspec.yaml` file: acts as the app's configuration file, managing external libraries and dependencies.

Running the application Once the app is created, you can run it using:

```
flutter run
```

This command allows you to select the platform on which to run the app. If the required platforms are not available on your device, Flutter will emulate the target device for testing.

2.3.1 Application structure

The `lib` directory initially contains a `main.dart` file, which includes the following minimal code:

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

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

You can choose between two styles: `material.dart` for material design or `cupertino.dart` for iOS-styled components. The `runApp` function initializes the application, setting a widget as the root of the widget tree. This tree typically consists of widgets like `Center` and `Text`, and Flutter ensures the root widget covers the entire screen.

2.3.2 Functions

n Dart, functions can have required positional parameters followed by optional named or optional positional parameters (but not both). Named parameters are optional unless explicitly marked as required. You can assign default values to both named and positional parameters using the `=` operator, but these must be compile-time constants. If no default value is specified, the default is `null`.

Commas When writing Dart functions, methods, or constructors, always include a trailing comma in the parameter list. This practice helps Flutter's code formatter add proper line breaks and maintain code consistency.

2.3.3 Widgets

Flutter applications are composed entirely of widgets, which are the building blocks of the user interface. Below is an example of Flutter's material design structure:

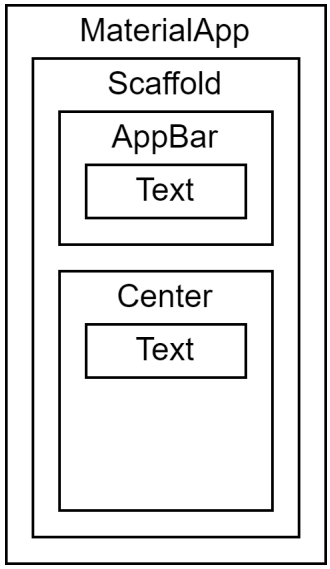

Figure 2.2: Material structure

Key characteristics of widgets include:

- *Immutability*: widgets are immutable; once created, they cannot be changed. To modify their appearance or behavior, you must create a new widget.
- *Composability*: widgets are composable, allowing developers to combine simpler widgets to create more complex interfaces.
- *Descriptive nature*: widgets describe the UI, providing a visual representation of the interface components.

Widget states Widgets can either be stateful or stateless:

- *Stateless widgets*: do not manage any internal state and are simple, static components of the UI.
- *Stateful widgets*: maintain state that can change during the app's lifetime (e.g., the position of a slider). Stateful widgets require two classes: one to extend `StatefulWidget` and another to manage the mutable state via the `State` object.

Stateful widgets use the `setState` method to notify Flutter that the state has changed, prompting the widget to be rebuilt and reflecting the updates in the UI. State management is crucial for dynamic user interfaces. When using the the `setState` method, Flutter knows that the state has changed and will re-render only the affected widgets, optimizing performance. Failing to call the `setState` when modifying state will leave the UI unchanged.

Commond widgets The most common widgets are:

- *Flexible*: resizable widgets that adjust their size according to their `flex` and `fit` properties.
- *Expanded*: forces a widget to take up all available space.
- *SizedBox*: specifies a widget's dimensions or adds empty space.
- *Spacer*: creates space between widgets using the `flex` property.

2.3.4 Assets

To include external assets in your project, you must declare them in `pubspec.yaml`. You can include all assets in a directory by specifying the directory name followed by a slash, e.g., `assets/`. For subdirectories, each must be listed separately.

Flutter's `AssetImage` class automatically selects the appropriate asset based on the device's pixel ratio, ensuring that the app displays optimally across different devices.

2.3.5 Navigation

Navigation in Flutter involves managing multiple screens (or routes). Routes represent screens, and you can navigate between them using:

- `Navigator.push`: adds a new route to the stack.
- `Navigator.pop`: removes the current route from the stack, returning to the previous screen.

To pass data between routes, use `Navigator.pushNamed` and access arguments using `ModalRoute.of`.

Asynchronous programming Dart uses `Future` objects to represent asynchronous operations. To define an asynchronous function, use the `async` keyword, and you can pause execution using the `await` keyword. This allows for handling asynchronous tasks such as network calls in a non-blocking manner.

Adaptive design For responsive layouts, use widgets like `LayoutBuilder` and `MediaQuery`. `LayoutBuilder` returns layout constraints from the parent widget, while `MediaQuery` gives the size of the app window. This distinction helps in creating adaptive interfaces that work across various screen sizes.

Advanced navigation For complex apps, especially web-based or multi-Navigator apps, using a `Router` allows for declarative navigation, integrating better with browser history and enabling advanced features like deep linking. This structure ensures efficient state management, adaptive design, and smooth navigation throughout the app development lifecycle in Flutter.