 **UNIVERSITY OF BUEA**

**REPUBLIC OF CAMEROON**

PEACE-WORK-FATHERLAND

Buea, South West Region

Cameroon

**FACULTY OF ENGINEERING AND TECHNOLOGY**

**DEPARTMENT OF COMPUTER ENGINEERING**

**Report on Task 1**

**Course Master: Dr Valery Nkemeni**

**CEF440:Internet Programming and Mobile Programming**

**GROUP 22**

|  |  |
| --- | --- |
| NAMES | MATRICULE |
| NGOUH KAMBI MARCBRYAN | FE22A264 |
| AMBASSA LISE-ASTRID | FE22A146 |
| EPIE-NGOLE MCCAULEY | FE22A445 |
| EPANDA RICHARD JUNIOR | FE22A206 |
| NKIANGWA NKEGOA KAREL | FE22A277 |

Table of Content

Introduction 3

Methodology 3

Review and comparison of the major types of mobile apps and their differences 3

Comparing Mobile App Programming Languages 4

**Mobile App Development Frameworks**  6

Mobile Application Architectures and Design Patterns 8

How to Collect and Analyze User Requirements for a Mobile Application…………………………………………………………………………12

How to Estimate Mobile App Development Cost………………………14

**Introduction**

This report evaluates the technical and strategic aspects of mobile application development, focusing on the application types, programming languages, frameworks, architectures, requirement engineering, and cost estiamtion.

Through comparative analysis, the trade-offs between native, hybrid, and progressive web apps, are highlighted. The suitability of languages like Swift, and Kotlin, and contrasts frameworks such as Flutter and React Native.

This report also shows methods for gathering user requirements and estimating development costs.

**Methodology**

This report was done through literature review, from various industry reports and case studies. The work was based on the criteria of Perfromance, cost, UX, Platform compatibility, and Community Support.

1. **Review and comparison of the major types of mobile apps and their differences**

**Comparison of Major Types of Mobile Apps**

**a. Native Apps**

* **Definition**: Apps built specifically for a platform (iOS or Android) using platform-specific languages (Swift for iOS, Kotlin/Java for Android).
* **Advantages**:
  + High performance and speed
  + Full access to device features (camera, GPS, etc.)
  + Better user experience and responsiveness
* **Disadvantages**:
  + Expensive and time-consuming development
  + Separate codebases for different platforms

**b. Progressive Web Apps (PWAs)**

* **Definition**: Web applications that behave like native apps but run in a browser.
* **Advantages**:
  + No need for installation (accessible via a URL)
  + Works across all devices and platforms
  + Faster and cheaper development
* **Disadvantages**:
  + Limited access to device features (e.g., Bluetooth, advanced sensors)
  + Performance may not match native apps
  + Requires an internet connection for full functionality

**c. Hybrid Apps**

* **Definition**: Apps that use web technologies (HTML, CSS, JavaScript) but are wrapped in a native container (e.g., using frameworks like Ionic, Cordova, React Native).
* **Advantages**:
  + Single codebase for multiple platforms
  + Faster development and cost-effective
  + Can access some native features
* **Disadvantages**:
  + Slower performance than native apps
  + Limited user experience due to web-based components

Key Points

* **Native Apps**: Best for high performance and full feature access but expensive.
* **PWAs**: Ideal for web-based experiences with low cost but limited device capabilities.
* **Hybrid Apps**: A balance between cost and performance, but may not match native app speed.

1. **Comparing Mobile App Programming Languages**

The common mobile app programming languages are Swift, Kotlin, Dart. Using any of these programming languages for mobile app programming, depends on certain factors, such as;

-the platform you are devloping for ie Android or iOS

Here you would want to use language that works well with the platform you are developing for, like swift for iOS, or maybe a language that works well for croos platform applications (Applications that work on either platform, Android or iOS)

-The learning curve of the language

The Learning curve of the language is another very important aspect that comes to play, when choosing a programming Language for a project. This has to do with how much it will affect the time of production, if the development team had to learn this language and use it in implementing their design for the mobile application.

Some Programming Languages are easy to learn, others moderate, and others seemingly hard.

Choosing a language with an easy learning curve make the development team produce faster, since much of the time used to understand the language, write and debug code, will bw used to implement other features. On the other hand, a language that seems hard to learn will only reduce the speed of the development team as much of their time maybe spent on understanding the complexities of the language, using it to impement features and debugging complex code.

-The Community

The community is very vital, in that, they help give feedback about the language and also solve any problems posed by developers using the language to solve a wide variety of problems.

A large community means you can easily get help on certain blockers, and the probability that someone in that commmunity has faced the same challenge as you, and is willing to help you come out of it, is very high. Unlike a small commmunity, where it may take longer to get a ressponse on a blocker and the probabalitity that a member could have experienced the same challenge as you is low.

Some Advantages and Dsiadvantages of Common Mobile Programming Languages

|  |  |  |
| --- | --- | --- |
| LANGUAGE | ADVANTAGE | DISAVANTAGE |
| Swift | -Modern Syntax  -Safe Memory Management | - iOS only |
| Kotlin | -Works well with java | -smaller community than java |
| Dart | -Cross Platform | -Less mature ecosystem |

Table of comparisons

|  |  |  |  |
| --- | --- | --- | --- |
| LANGUAGE | PLATFORM | LEARNING CURVE | COMMUNITY |
| Swift | iOS | Moderate | Large |
| Kotlin | Android | Moderate | Growing |
| Dart | Cross-Platform | Easy | Growing |

### ****Mobile App Development Frameworks****

### **Mobile app development frameworks** are tools and libraries that provide a structured environment for building mobile applications. They offer pre-written code, components, and functionalities to simplify development, enabling developers to create apps for iOS, Android, or both platforms efficiently.

### Some examples are listed below;

#### ****a. React Native****

A JavaScript framework by Facebook that allows developers to build cross-platform mobile apps using React, sharing a single codebase for both iOS and Android.

#### ****b. Flutter****

A UI toolkit by Google that uses the Dart language to build high-performance, cross-platform apps with a single codebase and custom UI components.

#### ****c. Swift (iOS) & Kotlin (Android)****

* **Swift**: Apple’s official programming language for developing fully native iOS apps.
* **Kotlin**: Google’s preferred language for building native Android apps with better performance and modern features.

#### ****d. Ionic****

An open-source framework that uses web technologies (HTML, CSS, JavaScript) to build cross-platform mobile apps, running inside a WebView.

#### ****e. Xamarin****

A Microsoft-backed framework that allows developers to create cross-platform apps using C# and .NET, compiling into native-like performance.

### ****Comparison of Mobile App Development Frameworks****

#### ****a. React Native****

* **Language**: JavaScript (React)
* **Performance**: Near-native but slower than fully native apps
* **Cost & Time to Market**: Faster and cost-effective (single codebase)
* **UX & UI**: Good, but depends on native components
* **Complexity**: Moderate (requires knowledge of React)
* **Community Support**: Strong (backed by Facebook)
* **Best For**: Cross-platform apps with a native feel

#### ****b. Flutter****

* **Language**: Dart
* **Performance**: Close to native, high performance
* **Cost & Time to Market**: Fast development (single codebase)
* **UX & UI**: Excellent, customizable UI with built-in widgets
* **Complexity**: Moderate to High (Dart learning curve)
* **Community Support**: Growing, backed by Google
* **Best For**: High-performance, visually rich apps

#### ****c. Swift (iOS) & Kotlin (Android)****

* **Language**: Swift (iOS), Kotlin (Android)
* **Performance**: Best (fully native)
* **Cost & Time to Market**: Expensive and time-consuming (separate codebases)
* **UX & UI**: Best experience with full OS integration
* **Complexity**: High (platform-specific development)
* **Community Support**: Strong (officially supported by Apple & Google)
* **Best For**: Performance-critical, platform-specific apps

#### ****d. Ionic****

* **Language**: HTML, CSS, JavaScript (uses WebView)
* **Performance**: Slower than native and hybrid frameworks
* **Cost & Time to Market**: Fast and affordable
* **UX & UI**: Decent but web-based (not truly native)
* **Complexity**: Low (easy for web developers)
* **Community Support**: Strong
* **Best For**: Simple cross-platform apps, PWAs

#### ****e. Xamarin****

* **Language**: C# (.NET)
* **Performance**: Close to native but slightly slower
* **Cost & Time to Market**: Faster than native, but licensing can be costly
* **UX & UI**: Good, but some limitations in design flexibility
* **Complexity**: Moderate (requires .NET knowledge)
* **Community Support**: Good (Microsoft-backed)
* **Best For**: Enterprise apps, apps with .NET backend

**Key Points**

* **Best for Native Performance**: Swift (iOS), Kotlin (Android)
* **Best for Cross-Platform with Native Feel**: React Native, Flutter
* **Best for Web-Based Apps & PWAs**: Ionic
* **Best for .NET Developers**: Xamarin

1. **Mobile Application Architectures and Design Patterns**

Mobile application architecture forms the backbone of app development, dictating how components interact, manage data, and handle user input. Unlike desktop applications, mobile apps must operate under constraints like limited battery life, intermittent connectivity, and varying screen sizes. A well-chosen architecture ensures scalability, maintainability, and efficient resource utilization. Design patterns complement architectures by offering proven solutions to recurring challenges, such as state management or dependency resolution.

**Mobile Application Architectures**

Architectures structure an app into logical layers (presentation, business logic, data) to isolate concerns, simplify testing, and streamline collaboration. The choice of architecture depends on factors like project scale, team expertise, and platform requirements.

1. **MVC (Model-View-Controller)**

- Model: Manages data (e.g., API calls, database operations).

- View: Renders UI and captures user input.

- Controller: Processes input, updates the Model, and refreshes the View.

**Advantages**: Simplicity, quick prototyping.

**Disadvantages**:

- Tight Coupling: Controllers often handle business logic \*and\* UI updates, leading to "Massive View Controllers."

- Testing Challenges: Mixing logic in Controllers complicates unit testing.

- Use Case: Small iOS apps (e.g., legacy UIKit apps).

1. **MVP (Model-View-Presenter)**

- Presenter: Acts as a middle layer, fetching data from the Model and updating the View via interfaces.

- View: Passive, delegates logic to the Presenter.

**Advantages**:

- Clear separation of concerns.

- Easier testing (Presenter can be mocked).

**Disadvantages**: Boilerplate code for interface definitions.

- Use Case: Android apps (e.g., traditional Java/Kotlin codebases).

1. **MVVM (Model-View-ViewModel)**

- ViewModel: Exposes observable data streams (e.g., LiveData in Android, Combine in iOS).

- Data Binding: Automatically syncs View with ViewModel state.

**Advantages**:

- Decouples UI from business logic.

- Ideal for reactive UIs (e.g., real-time dashboards).

**Disadvantages**:

- Requires understanding of reactive frameworks (RxJava, SwiftUI).

- Overhead in simple apps.

- Use Case: Cross-platform apps (e.g., Xamarin, Jetpack Compose).

**d. Clean Architecture**

- Layers:

- Entities: Core business objects.

- Use Cases: Application-specific logic.

- Interface Adapters: Convert data between layers.

- Frameworks: UI, databases, external APIs.

- Dependency Rule: Inner layers never depend on outer layers.

**Advantages:**

- Platform-agnostic business logic.

- High testability (unit tests for Use Cases).

**Disadvantages:**

- Initial setup may be complex.

- Use Case: Enterprise apps (e.g., banking, healthcare).

**e. VIPER (View-Interactor-Presenter-Entity-Router)**

- Interactor: Handles business logic.

- Presenter: Formats data for the View.

- Router: Manages navigation.

- Entity: Data models.

**Advantages:**

- Strict single responsibility.

- Modularity for large teams.

**Disadvantages:**

- Verbose boilerplate.

- Steep learning curve.

- Use Case: Large iOS apps (e.g., e-commerce platforms).

**Mobile Application Design Patterns**

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blue prints that you can customize to solve a recurring design prob lem in your code.

Some common design Patterns include

* **Factory Method:** Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Use the Factory Method when you don’t know beforehand the exact types and dependencies of the objects your code should work with.

* **Abstract Factory:** Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Use the Abstract Factory when your code needs to work with various families of related products, but you don’t want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.

* **Builder**: Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).

* **Prototype:** Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Use the Prototype pattern when your code shouldn’t depend on the concrete classes of objects that you need to copy.

* **Singleton:** Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for exam ple, a single database object shared by different parts of the program.

* **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.

UsetheAdapter classwhenyouwant tousesomeexisting class, but its interface isn’t compatible with the rest of yourcode.

* **Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some function ality (for example, if the class can work with various database servers).

1. **How to Collect and Analyze User Requirements for a Mobile Application**

It refers to the process of gathering insights and expectations from potential users of a mobile app to understand their needs and ptreferences. This process is *vital* for developing an application that meets user expectations and provides positive experience.

It uncovers your blinspot as you get closer to understanding the user amd their bahavioural patterns.

**Requirements Elicitation**

* Surveys and Questionaires:

Distribute surveys to a large audience to gather quantitative and qualitative data.

Use of multiple choice and open ended questions.

* Interviews:

Conduct one-on-one interviews with potential users to gain in depth insights into their experiences and expectations.

Focus on user goals, tasks and pain points.

Example questions; Asking questions of like, What do you think of the features in this app

* Focus groups:

Organize group dicsussions to explore their thoughts about mobile applications in general.

**User observations:**

Observe users interacting with existing apps to identify usability issues and feature preferences.

**Competitive analysis:**

Analyze competitor applications to identify best practices and potential differentiators.

**Requirements Analysis**

* Classification:

Categorize requirements into functional and non funtional requirements. Functional requiremnets describes what the apllication should do while the non-funtional requirements describes the quality attributes of the application.

* Prioritization:

The reqirements are then prioritized based on their importance and urgency.

This prioritization helps with planning and resource alloacation.

* Requirements Validation:

We verify that the requirements are accurate, complete and consistent.

Conducting reviews with stakeholders to ensure that the requirements meet their needs.

* Requirements Specification:

Document the requirements in a clear and concise manner.

Creating a Software Requirements Specification document which serves as a contract between the development team and the stakeholders.

1. **How to Estimate Mobile App Development Cost**

In this section, we analyze the key factors that influence the cost of developing a mobile application, with a specific focus on Cameroon. Our research takes into consideration local market conditions, development practices, and economic factors that affect software development within the region.  
  
**a. App Objectives and Features**  
Before estimating the cost of a mobile app, it is essential to clearly define its purpose and core functionalities. The complexity of an app is one of the main factors influencing development costs.   
- Simple Apps: These include basic informational apps, calculators, or small utility apps with minimal functionalities.  
- Moderate Complexity Apps: These feature additional functionalities such as social media integration, geolocation services, and in-app purchases.  
- Complex Apps: These include applications with real-time data processing, advanced animations, artificial intelligence components, and extensive backend development.  
  
**b. Development Platform**  
The choice of platform significantly impacts the development cost. In Cameroon, Android is the dominant platform due to its affordability and widespread usage. However, businesses looking to target iOS users may choose to develop for both platforms.  
- Single Platform (Android or iOS): Developing for one platform reduces initial costs.  
- Cross-Platform Development: Using frameworks such as Flutter or React Native can help develop an app for both platforms simultaneously, saving time and effort but increasing the initial cost.  
  
**c. Design and User Experience (UX/UI)**  
The design of a mobile app plays a crucial role in user engagement and overall success. The complexity of the design directly impacts cost.  
- Basic UI: Uses pre-built templates, making development faster and more affordable.  
- Custom UI: Requires specialized design work and animations, increasing development time and cost but improving user engagement.  
  
**d. Development Time and Team Costs**  
The time required to complete different phases of the project (planning, design, development, testing, and deployment) contributes to the overall cost. Software developers in Cameroon charge varying rates based on their experience and the scope of the project.   
- Freelance Developers: Generally more affordable but may take longer to complete the

**References**

- Dive into Design Patterns, *Alexender Schvets*

*-* Progressive Web Apps, *Tal Ater*

*-S*oftware Requirements, *Ian Sommerville*(chapter 4)