UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

REPULBLIC OF CAMEROON

PEACE-WORK-FATHERLAND

COURSE TITLE: INTERNET PROGRAMMING AND MOBILE PROGRAMMING

COURSE CODE: CEF 440

COURSE INSTRUCTOR: Dr. Valery NKEMENI

REPORT ON TASK 1

GROUP 17

GROUP MEMBERS:

SN	NAME	MATRICULE
1	AKENJI FAITH SIRRI	FE22A142
2	DYL PADARAN AMBE MUNJO	FE22A193
3	KONGNYU DESCHANEL	FE22A234
4	NDI BERTRAND	FE22A252
5	NDUKIE EBOKE BLANDINE	FE22A254

Table of Content:

1) Comparison of Major Types Of Mobile Apps And Their Differences(Native, Progressive web apps and Hybrid apps)

Mobile apps are categorized into three main types: **Native Apps, Progressive Web Apps (PWAs), and Hybrid Apps**. Each has distinct characteristics, advantages, and limitations.

1.1 Native Apps

Definition:
Native apps are applications built specifically for a particular operating system (iOS or Android) using platform-specific programming languages.

Technologies Used:

- **Android:** Java or Kotlin
- **iOS:** Swift or Objective-C

Key Features:

- Fully optimized for the OS, leading to **high performance and smooth user experience**.
- Access to **device features** like GPS, camera, microphone, and push notifications.
- **Downloaded from app stores** (Google Play Store, Apple App Store).

Advantages:
- Best performance & speed
- Full access to device features
- Better security
- Works offline

Limitations:
- Higher development costs (separate apps for iOS & Android)
- Longer development time
- Requires app store approval

1.2 Progressive Web Apps (PWAs)

Definition:
PWAs are web applications that function like native apps but run in a web browser. They don't need to be installed from an app store.

Technologies Used:

- HTML, CSS, JavaScript
- Frameworks: React, Angular, Vue.js

Key Features:

- **Runs in a web browser** (no need for installation).
- Can be added to the home screen.
- Works offline using service workers.
- Provides push notifications.

Advantages:
- Lower development cost (one version for all platforms)
- No app store approval required
- Fast updates (no need for downloads)

Limitations:
- Limited access to device features (e.g., Bluetooth, NFC)
- Slightly lower performance than native apps
- Not available in traditional app stores

1.3 Hybrid Apps

Definition:
Hybrid apps combine elements of both native apps and web apps. They are built using web technologies but run inside a native container.

Technologies Used:

- Ionic, Flutter, React Native, Apache Cordova

Key Features:

- Can be downloaded from app stores.
- Uses a **single codebase** for multiple platforms.
- Web content is displayed inside a native wrapper.

Advantages:
- Faster development than native apps
-Lower cost (one app for both iOS & Android)
- Can access some device features

Limitations:
- Performance not as good as native apps
- UI might not feel as smooth as native
- More dependency on third-party frameworks

Summary

- For best performance & user experience → Native Apps
- For cost-effectiveness & quick deployment → PWAs
- For a balance between performance & cost → Hybrid Apps

2. Mobile App Programming Languages

Mobile applications are developed using various programming languages depending on the platform, performance requirements, and development preferences. Below is a detailed overview of the most commonly used programming languages for mobile app development.

2.1 Swift

- **Platform**: iOS (Apple devices)
- **Description**: Swift is Apple's official programming language for developing iOS applications. It is designed to be fast, modern, and safe.
- **Use Cases:** iOS-exclusive apps requiring hardware access (e.g., ARKiT, Core ML)
- **Advantages**:
    - High performance and optimized for Apple devices
    - Easy to read and maintain
    - Strong community and Apple support
- **Disadvantages**:
    - Limited to the Apple ecosystem
    - Less flexibility for cross-platform development

2.2 Kotlin

- **Platform**: Android
- **Description**: Kotlin is Google's preferred language for Android development, offering concise syntax and interoperability with Java.
- **Advantages**:
    - Modern and concise
    - Safer than Java (null safety feature)

    - Fully supported by Google
- **Disadvantages**:
    - Slightly steeper learning curve compared to Java
    - Limited adoption outside Android development
    - Requires separate codebase for iOS

2.3 Java

- **Platform**: Android
- **Description**: Java has been the traditional language for Android development and is still widely used.
- **Advantages**:
    - Strong community support
    - Platform-independent and widely adopted
    - Scalable and robust
- **Disadvantages**:
    - More verbose compared to Kotlin
    - Slower compared to newer languages like Kotlin

2.4 Dart (Flutter)

- **Platform**: Cross-platform (iOS & Android)
- **Description**: Dart is used with Google's Flutter framework to build cross-platform mobile applications with a single codebase.
- 2025 Trend: Dominates 68% of new cross-platform projects (Statista 2025)
- **Advantages**:
    - Fast development with hot reload
    - High performance (compiled to native code)
    - Excellent UI support with Flutter widgets
- **Disadvantages**:
    - Limited adoption outside Flutter
    - Smaller community compared to Java or Swift

2.5 JavaScript (React Native, Ionic)

- **Platform**: Cross-platform
- **Description**: JavaScript is widely used for building hybrid and cross-platform mobile applications using frameworks like React Native and Ionic.
- **Advantages**:

    - Single codebase for multiple platforms
    - Large developer community
    - Rapid development and flexibility
- **Disadvantages**:
    - Lower performance compared to native languages
    - Dependence on third-party plugins for native features

2.6 Python (Kivy)

- **Platform**: Cross-platform
- **Description**: Python can be used for mobile development with frameworks like Kivy and BeeWare, though it is less common.
- **Advantages**:
    - Easy to learn and highly readable
    - Strong AI and data science support
- **Disadvantages**:
    - Not optimized for mobile performance
    - Limited community and support for mobile development

2.7 C# (Xamarin)

- **Platform**: Cross-platform
- **Description**: C# is used with the Xamarin framework to build cross-platform mobile applications using a shared codebase.
- **Advantages**:
    - High performance
    - Strong support from Microsoft
    - Native-like UI experience
- **Disadvantages**:
    - Requires knowledge of .NET
    - Heavier applications compared to Flutter or React Native

summary

Languages	**Platforms**	**Performance**	**Ease of use**	**Community Support**
swift	iOS	High	moderate	strong

kotlin	Android	High	moderate	strong
java	Android	High	moderate	strong
Dart	cross-platform (flutter)	High	moderate	growing
JavaScript	cross-platform (React Native, Ionic)	Moderate	easy	strong
Python	cross-platform (kivy)	Moderate	easy	moderate
C#	cross-platform (Xamarin)	High	moderate	strong

3) Review & Comparison of Mobile Application Frameworks by Key Features

In the fast-evolving landscape of mobile application development, selecting the right framework can significantly impact project success. With various frameworks offering distinct advantages, developers must assess them based on performance, user experience, development efficiency, and cross-platform capabilities. This report provides a unique and in-depth comparison of the most popular mobile application frameworks to help developers and businesses make informed decisions.

3.1 Overview of Mobile Application Frameworks :

The leading mobile application frameworks include:

- **React Native** – Developed by Facebook, known for its JavaScript-based cross-platform capabilities.
- **Flutter** – Google's UI toolkit that enables natively compiled applications.
- **Xamarin** – A Microsoft-backed framework allowing C# developers to create cross-platform apps.
- **Swift** – Apple's powerful language for native iOS development.
- **Kotlin** – The official language for native Android app development.
- **Ionic** – A hybrid framework utilizing web technologies to create mobile apps.

3.2 Key Features Comparison: This section examines each framework based on critical features essential for mobile development.

3.2.1 Performance

- **React Native:** Near-native performance but dependent on JavaScript bridge, causing occasional lags.
- **Flutter:** Exceptional performance due to direct compilation with Dart and Skia rendering engine.
- **Xamarin:** Good performance but suffers from larger app sizes and dependency on .NET runtime.
- **Swift & Kotlin:** Unmatched native performance, as they are designed specifically for iOS and Android, respectively.
- **Ionic:** Performance bottlenecks due to WebView-based rendering, making it less suitable for complex applications.

3.2.2 UI/UX Capabilities

- **React Native:** Uses native components, ensuring a familiar look and feel.
- **Flutter:** Provides a highly customizable UI with its widget-based approach.
- **Xamarin:** Enables native UI access through Xamarin.Forms.
- **Swift & Kotlin:** Full control over UI elements, allowing precise customization.
- **Ionic:** Relies on web components, requiring additional effort to achieve a native-like experience.

3.2.3 Development Speed & Ease of Use

- **React Native:** Fast development with Hot Reload but may require native modules for complex tasks.
- **Flutter:** Hot Reload and comprehensive widget libraries make development seamless.
- **Xamarin:** Slower than Flutter and React Native due to Visual Studio dependencies.
- **Swift & Kotlin:** Native development requires more effort but offers high optimization.
- **Ionic:** Quick development but may face performance trade-offs.

3.2.4 Cross-Platform Support

- **React Native:** Supports both iOS and Android with a shared codebase.
- **Flutter:** Single codebase for Android, iOS, and experimental support for web and desktop.
- **Xamarin:** Covers iOS, Android, and Windows but is less popular.
- **Swift & Kotlin:** Platform-specific, requiring separate development.
- **Ionic:** Works across mobile and web platforms.

3.2.5 Community Support & Ecosystem

- **React Native:** Large, active community with extensive libraries.
- **Flutter:** Rapidly growing ecosystem backed by Google.
- **Xamarin:** Smaller community compared to React Native and Flutter.
- **Swift & Kotlin:** Strong support from Apple and Google, respectively.
- **Ionic:** Broad community but limited for large-scale apps.

3.2.6 Integration with Native Features

- **React Native:** Relies on third-party plugins for deeper native integration.
- **Flutter:** Uses platform channels for smooth interaction with native APIs.
- **Xamarin:** Direct access to native APIs but may require additional configurations.
- **Swift & Kotlin:** Seamless integration as they are native languages.
- **Ionic:** Limited native capabilities; depends on Cordova or Capacitor plugins.

Each mobile application framework offers unique benefits based on project needs. React Native and Flutter dominate cross-platform development, with Flutter excelling in performance and UI design. Xamarin remains relevant for Microsoft ecosystem integration, while Swift and Kotlin are indispensable for native app development. Ionic, though fast for hybrid apps, lacks the performance required for intensive applications.

3.3 Recommendation

- **For high-performance cross-platform apps:** Flutter is the best choice due to its fast UI rendering and smooth performance.
- **For cross-platform development with JavaScript expertise:** React Native offers a familiar ecosystem and flexibility.
- **For enterprise solutions and Microsoft integration:** Xamarin is a viable option, especially for C# developers.
- **For native iOS and Android apps:** Swift and Kotlin remain the gold standard.
- **For web developers seeking mobile compatibility:** Ionic provides a quick solution but is not ideal for performance-intensive applications.

4) Mobile Application Architectures

Mobile application architectures and design patterns are crucial for building scalable, maintainable, and efficient applications.

Some common architectures and design patterns used in mobile app development Include:

4.1 MVC (Model-View-Controller):

- **Model**: Represents the data and business logic.
- **View**: Displays the data (UI).
- **Controller**: Handles user input and updates the model and view.
- **Use Case**: Simple applications where separation of concerns is needed.

4.2 MVVM (Model-View-ViewModel):

- **Model**: Represents the data.
- **View**: The UI that displays the data.
- **ViewModel**: Acts as a bridge between the Model and View, handling the presentation logic.
- **Use Case**: Applications that require data binding and a clear separation of UI and business logic (popular in Xamarin and Android development).

4.3 MVP (Model-View-Presenter):

- **Model**: Represents the data.
- **View**: The UI that displays the data.
- **Presenter**: Handles the presentation logic and interacts with the Model and View.
- **Use Case**: Applications that require a more testable and maintainable structure than MVC.

Clean Architecture:

- Divides the application into layers (e.g., Presentation, Domain, Data) to achieve separation of concerns.
- Each layer is independent, making it easier to test and maintain.
- **Use Case**: Large applications that require scalability and maintainability.

VIPER (View, Interactor, Presenter, Entity, Router):

- A more modular architecture that separates responsibilities into five components.
- **Use Case**: Complex applications that benefit from high modularity and testability.

Flux/Redux:

- A unidirectional data flow architecture.
- **Flux**: A pattern that emphasizes a unidirectional flow of data.
- **Redux**: A library that implements Flux principles, often used in React Native applications.
- **Use Case**: Applications with complex state management.

Design Patterns

1. **Singleton**:
    - Ensures a class has only one instance and provides a global point of access to it.
    - **Use Case**: Managing shared resources like configuration settings or network connections.
2. **Observer**:

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.
- **Use Case**: Implementing event handling systems.

3. **Factory**:
   - Provides an interface for creating objects in a superclass but allows subclasses to alter the type of created objects.
   - **Use Case**: Creating instances of classes without specifying the exact class of object that will be created.
4. **Adapter**:
   - Allows incompatible interfaces to work together by wrapping an existing class with a new interface.
   - **Use Case**: Integrating third-party libraries or legacy code.
5. **Decorator**:
   - Adds new functionality to an object dynamically without altering its structure.
   - **Use Case**: Extending the behavior of UI components.
6. **Strategy**:
   - Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
   - **Use Case**: Implementing different sorting algorithms or payment methods.
7. **Command**:
   - Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.
   - **Use Case**: Implementing undo/redo functionality.

Best Practices

- **Separation of Concerns**: Keep different aspects of the application (UI, business logic, data) separate to improve maintainability.
- **Dependency Injection**: Use dependency injection to manage dependencies and improve testability.
- **Reactive Programming**: Consider using reactive programming paradigms (e.g., RxJava, Combine) for handling asynchronous data streams.
- **Testing**: Write unit tests and UI tests to ensure the application behaves as expected.
- **Performance Optimization**: Optimize for performance, especially in mobile environments where resources are limited.

Choosing the right architecture and design patterns depends on the specific requirements of the application, including its complexity, scalability needs, and team expertise. Understanding these concepts will help you build robust mobile applications that are easier to maintain and extend over time.

5. Requirement Engineering for Mobile Applications

5.1 Collecting User Requirements

Requirement engineering involves gathering, analyzing, and documenting the needs and expectations of users to ensure a mobile application meets their needs. The following methods are commonly used to collect user requirements:

1. **Surveys and Questionnaires**
   - Online or paper-based forms to collect feedback from potential users
   - Useful for gathering large-scale data from diverse audiences
2. **Interviews**
   - One-on-one discussions with stakeholders (users, clients, developers)
   - Helps gather in-depth qualitative insights
3. **Focus Groups**
   - A moderated discussion with a group of target users
   - Helps identify common pain points and preferences
4. **Observation**
   - Studying user behavior to understand how they interact with existing applications
   - Provides real-world insights into user needs
5. **User Stories & Use Cases**
   - **User Stories**: Simple descriptions of features from a user's perspective
   - **Use Cases**: Detailed scenarios explaining how users interact with the app

5.2 Analyzing User Requirements

Once requirements are collected, they must be analyzed to identify key functionalities and prioritize them.

1. **Categorization**
   - Functional Requirements: Features that define the app’s functionality (e.g., login, payment processing)
   - Non-functional Requirements: Performance, security, usability, and scalability aspects
2. **Prioritization**

        - MoSCoW Method:
            - **Must-Have**: Essential features
            - **Should-Have**: Important but not critical
            - **Could-Have**: Nice-to-have features
            - **Won't-Have**: Features for future development
3. **Prototyping and Feedback**
    - Creating wireframes or mockups to visualize the app
    - Gathering feedback from stakeholders before actual development

5.3 Validating Requirements

Requirement validation ensures that the gathered requirements are accurate, complete, and feasible.

- Conducting **requirement review meetings** with stakeholders
- Using **prototypes** to demonstrate the functionality
- Performing **feasibility studies** to assess technical and economic viability

6) Estimating Mobile Development Costs

Estimating mobile app development costs is a process that depends on many variables influenced by your app's complexity, the range of features you want to include, the development process, team location, design requirements, and even post-launch maintenance.

6.1 Complexity & Features

- Simple apps (e.g., a flashlight or basic calculator) can cost little to nothing for a sole developer to $500 outsourcing. These typically include basic functionality such as user login, simple UI, and minimal interactivity.
- Medium complexity apps(for example, a social media or e-commerce app) might cost between $50,000 and $150,000. These apps include more advanced features like real-time updates, payment gateway integrations, push notifications, and custom user interfaces.
- Highly complex apps(such as those with AI, AR/VR, IoT, or enterprise-level features) can exceed $300,000. Complexity increases with additional integrations, advanced security requirements, and support for multiple platforms simultaneously.

6.2 Development Process & Cost Breakdown

The cost is not just about coding—the overall process is typically divided into several stages:

- Discovery & Planning: Involves market research, competitor analysis, and defining project scope. Data required may incur some costs to get as it maybe paywalled where there's usually little alternatives like Nature, Springer, ...
- UX/UI Design: Creating user journeys, wireframes, and high-fidelity designs. Depending on the number of screens and complexity
- Development (Coding): Often the most significant portion, usually accounting for 50–70% of the total cost. This phase includes both front-end and back-end development.
- Quality Assurance (QA) & Testing: Ensures that the app works seamlessly across devices and meets user expectations. Testing costs may also rely on paid beta testers and platforms designed for testing applications.
- Deployment & Maintenance: Publishing the app to stores and planning for ongoing updates and bug fixes. Maintenance may require spending around 15–25% of the initial development cost each year.

6.3 Team & Regional Factors

The cost also significantly depends on:
- Team Structure: Whether you build an in-house team or outsource to an agency. Outsourcing can often lower costs due to access to a global talent pool.
- Geographical Location: Hourly rates vary widely:
  - In North America and Western Europe, rates may be between $80 and $170 per hour.
  - In regions like Eastern Europe, India, or Southeast Asia, hourly rates can be much lower , and much lower in African areas

6.4 Platform Considerations

- Native vs. Cross-Platform: Developing separate native apps for iOS and Android provides optimal performance and access to device features but is more expensive. Deployment to mobile play stores such as iOS store incurs an additional cost of $99 per month while $25 for each upload to the Google app store.
- Cross-platform frameworks (e.g., React Native or Flutter) allow for a single codebase and can reduce costs by 25–35%.
- Web or Hybrid Apps: Progressive Web Apps (PWAs) or hybrid solutions can be more cost-effective but may sacrifice some functionality or performance compared to native apps.

6.5 Estimation Techniques

Software engineering uses various methods to estimate costs:
- Parametric Estimation: Uses historical data and metrics such as Function Points or Story Points.
- Expert Judgment and Delphi Methods: Involve consensus from experienced developers.
- Models like COCOMO or SLIM: Which predict effort based on the estimated size (often measured in SLOC or function points).

6.6 Recurring Development Costs

Beyond the initial build, you must account for:
- Infrastructure costs: Servers, cloud services, databases,(hosting fees) and third-party API fees like a paymentAPI, datafeed API
- Security and Compliance: Additional costs if your app handles sensitive data or must comply with regulations.
- Marketing and User Acquisition: Essential for ensuring your app reaches its target audience, which is usually a considerable part of spending.
- Ongoing Maintenance & Updates: Apps require regular updates to support new OS versions, fix bugs, and add improvements—often estimated at 15–25% of the original development annual

EMPRICAL ESTIMATES FOR MOBILE DEVELOPMENT COSTS

1. COCOMO and Its Variants

Basic COCOMO Formula:

$$PM = a * (KLOC)^b$$

- PM is the effort in person-months.
- KLOC is the estimated thousands of lines of code.
- a and b are empirically derived constants. For example, for organic projects, $a \approx 2.4$ and $b \approx 1.05$.

There are also Intermediate and Detailed COCOMO models that adjust the basic formula with cost drivers (like team capability, product complexity, etc.), but they still follow a similar empirical power-law structure.

2. Function Point Analysis (FPA)–Based Estimation

Instead of using lines of code, you can size software based on its functionality. A common empirical approach is to compute Unadjusted Function Points (UFP) by counting inputs, outputs, inquiries, internal files, and external interfaces, and then adjust them using a value adjustment factor (VAF).

Once you have the total Function Points, you multiply by an empirical "hours per function point" factor—derived from historical data—to estimate the development effort in person-hours.

For example:

Effort (hours) = Total Function Points × (Average hours per Function Point)

The "hours per FP" value can range widely (say, 10–20 hours per FP) depending on the project and the team's experience.

3. Use Case Points (UCP)

If you have your system's use cases defined, you can estimate size via Use Case Points:

UCP = (Unadjusted Actor Weight + Unadjusted Use Case Weight) × TCF × ECF

- TCF is the Technical Complexity Factor.
- ECF is the Environmental Complexity Factor.

Then, similar to FPA, you multiply the UCP by an empirical productivity rate (hours per UCP) to get the effort estimate.

4. Empirical Data from Historical Projects

Many organizations derive their own "rules of thumb" based on historical data. For example, one source might suggest that a simple mobile app costs around $X per function point or requires Y person-hours per function point. Often, these numbers are calibrated from previous projects and can be used as a multiplier in one of the formulas above.

Each of these methods requires you to have some notion of the project size—whether in KLOC, function points, or use case points—and then apply an empirical multiplier (derived from past projects) to estimate effort and cost.

Conclusion

This report has provided an extensive review of mobile application development, covering types of mobile apps, programming languages, frameworks, architectures, requirement engineering, and cost estimation. Understanding these aspects is crucial for making informed decisions in mobile app development. Choosing the right combination of technologies, architectures, and cost estimation methods ensures the success of a mobile application in today's competitive market.