Republic of Cameroon

Peace -Work-Fatherland

Ministry of Higher Education

University of Buea

République du Cameroun

Paix-Travaille-Patrie

Ministère de l'enseignement superieure

Université de Buea

Faculty of Engineering and Technology

Faculty of Engineering and Technology

Department of Computer Engineering

CEF 440: Internet and mobile programming task 1

By: Group 1

ARREY-TABOT PASCALINE	FE22A151
TANUI NORBERT TANGIE	FE22A306
OROCKTAKANG MANYI	FE22A293
NGATTA GEORGE TABOT	FE22A259
SIDNEY FOMECHE	FE22A218

2024/2025 Academic year

Table of contents

.1 Major types of mobile apps and their differences

A mobile app is a software application designed to run on mobile devices such as smartphones and tablets. It provides specific functionalities, such as communication, entertainment, productivity, or business services, and can be developed on platforms like iOS and Android. Mobile apps can be native (built for a specific OS), web-based, or cross-platform.

1. **Native apps:**

-These apps are built specifically for a particular mobile operating system (OS), such as iOS or Android.
-They utilize the OS's native programming languages (e.g., Swift/Objective-C for iOS, Kotlin/Java for Android).
Characteristics
Performance: Generally, offer the best performance due to direct access to device hardware. **USER Experience (UX):** Provide a seamless and intuitive UX that aligns with the OS's design guidelines.
Hardware Access: Full access to device features like the camera, GPS, and sensors.
Offline Capabilities: Can often function offline.
Development: Requires separate development efforts for each OS, leading to higher costs.
It should be noted native apps are tailored for optimal device performance

2. **Progressive web apps (PWA):**

PWAs are web applications that use modern web capabilities to deliver a native app-like experience to users.
They're built using web technologies (HTML, CSS, JavaScript) and run within a web browser.
Characteristics:
Cross-Platform: Work on any device with a modern web browser.
Accessibility: Accessible through a URL, no app store download required.
Offline Functionality: Can work offline or in low-connectivity conditions using service workers.
Push Notifications: Can send push notifications to users.
Cost-Effective: Typically, less expensive to develop than native apps. Also, PWAs prioritize accessibility and cross-platform reach

3. **Hybrid Apps**:

Hybrid apps are a combination of web and native app elements.
They're built using web technologies (HTML, CSS, JavaScript) and then wrapped in a native container.
Frameworks like React Native and Ionic are used to create hybrid apps.
Key Characteristics:
Cross-Platform: Can run on multiple OSs from a single codebase.
Development: Faster and less expensive than native app development.
Performance: Performance is generally good, but may not match native app performance. **Hardware Access**: Can access some device features, but may have limitations.

.2 Mobile app programming languages

1. **Native Languages**:

>Swift (iOS):
Developed by Apple, Swift is the modern, preferred language for iOS, macOS, watchOS, and tvOS development.
Advantage
-High performance.
-Modern, clean, and safe syntax.
-Strong type safety, reducing errors.
-Growing community and strong Apple support.
Disadvantage
-Limited to the Apple ecosystem.
-Relatively smaller developer pool compared to some other languages.

>Kotlin (Android):
Officially endorsed by Google for Android development.
Advantage
Excellent interoperability with Java.
Concise and expressive syntax.
Null safety, preventing common errors.
Growing community and strong Google support.
Disadvantage
Relatively newer than Java, though rapidly gaining popularity.

>Java (Android):
Historically the primary language for Android development.
Advantage
Large and established community.
Extensive libraries and frameworks.
"Write once, run anywhere" capability.
Disadvantage
More verbose syntax compared to Kotlin.
Can lead to larger app sizes.
Can be slower than Kotlin.

2. Cross-Platform Languages:

>Dart (Flutter):
Developed by Google, used with the Flutter framework.
Advantage
Single codebase for iOS and Android.
Hot reload feature for fast development.
Rich set of customizable widgets. Good performance.
Disadvantage
Relatively newer compared to native languages.
Larger app sizes than native applications.

>JavaScript (**React Native**):
Used with the React Native framework, developed by Facebook.
Advantage
Single codebase for iOS and Android.
Large and active JavaScript community.
Relatively fast development.
Native like performance.
Disadvantage
Performance can sometimes be slightly lower than native apps.
Relies on native bridges, which can introduce complexities.

>C# (**Xamarin/.NET MAUI**):
Used with the Xamarin and now .NET MAUI frameworks, developed by Microsoft.
Pros:
Cross-platform development.
Strong integration with the .NET ecosystem.
Mature and well-supported by Microsoft.
Cons:
Performance can vary.
Framework dependencies can introduce complexity.

.3 Mobile app development frameworks and their key features

React Native is an open-source mobile app framework created by Facebook. It allows developers to build cross-platform apps using JavaScript and React, enabling code reuse across iOS and Android.
Performance: Good but slightly slower than fully native apps
Cost & Time to Market: Faster development with reusable components
UX & UI: Good but requires extra optimization for a native feel
Complexity: Moderate
Community Support: Strong, backed by Facebook
Best Use Cases: Cross-platform apps that require a near-native experience

Flutter is an open-source UI toolkit developed by Google for creating natively compiled applications for mobile, web, and desktop from a single codebase. It uses Dart programming language and offers a rich set of pre-built widgets, enabling highly customizable and smooth UI experiences.
Performance: Excellent, almost native-level speed
Cost & Time to Market: Fast with "Hot Reload" feature for quick testing
UX & UI: Excellent, highly customizable UI
Complexity: Moderate to High
Community Support: Growing rapidly, backed by Google
Best Use Cases: UI-heavy apps, startups, and MVPs

Xamarin is a cross-platform app development framework owned by Microsoft. It uses C# and .Net to build apps that run on both iOS and Android. Xamarin allows code sharing across platforms, reducing development time while still maintaining a native-like experience through Xamarin.iOS and Xamarin.Android APIs.
Performance: Good, but UI rendering is not always smooth
Cost & Time to Market: Moderate, some code reuse possible
UX & UI: Decent but not as fluid as native apps
Complexity: Moderate
Community Support: Declining as Microsoft shifts focus
Best Use Cases: Business applications for companies using .NET

Swift (iOS Development) is a powerful, modern programming language developed by Apple for iOS, macOS, watchOS, and tvOS development. It is optimized for performance and safety, offering features such as memory management and strong type safety. Native iOS applications built with Swift have the best possible performance and seamless integration with Apple's ecosystem.
Performance: Excellent, fully native
Cost & Time to Market: Slower due to separate iOS-only development
UX & UI: Best for iOS apps
Complexity: High
Community Support: Strong, backed by Apple
Best Use Cases: High-performance iOS applications

Kotlin (Android Development) is a modern, statically typed programming language officially supported by Google for Android development. It is fully interoperable with Java and improves upon it with concise syntax, null safety, and improved code readability. Kotlin is used to create high-performance native Android applications.
Performance: Excellent, fully native
Cost & Time to Market: Slower due to separate Android-only development
UX & UI: Best for Android apps
Complexity: High
Community Support: Strong, backed by Google
Best Use Cases: High-performance Android applications

Each framework has strengths and weaknesses. Flutter and React Native are best for cross-platform development, while Swift and Kotlin provide the best native performance.

.4 Mobile app architectures and design patterns

Mobile applications architectures and design patterns improve scalability, maintainability, and performance in mobile app development.

>Mobile Application Architectures

1. Monolithic Architecture
The entire application is built as a single unit where all components are tightly connected.
Advantages: Simple to develop and deploy.
Disadvantages: Difficult to scale and update as the app grows.

2. Layered Architecture
The app is divided into layers (e.g., Presentation, Business Logic, and Data) that separate concerns.
Advantages: Easier to manage and maintain, modular code.
Disadvantages: Can introduce performance overhead due to multiple layers of communication.

3. Microservices Architecture
The app is broken down into small, independent services that communicate via APIs.
Advantages: Highly scalable and flexible, allows independent deployment of features.
Disadvantages: More complex to manage, requires careful API design.

4. MVVM (Model-View-ViewModel) Architecture
The app is structured into three components:
Model: Handles data and business logic.
View: Represents the UI.
ViewModel: Acts as a bridge between Model and View.
Advantages: Improves maintainability and testability, reduces UI-business logic coupling.
Disadvantages: Requires more initial setup, increases complexity.

>Design Patterns in Mobile Development

1. Singleton Pattern
Ensures only one instance of a class exists and provides a global access point.
Example: Managing a database connection in an app.

2. Factory Pattern
Creates objects without specifying their exact class, making the code more flexible.
Example: Generating different types of UI components dynamically.

3. Observer Pattern
Defines a dependency between objects so that when one changes, all dependent objects are notified.
Example: Implementing real-time notifications or event-driven apps.

4. Repository Pattern

Abstracts data access logic, creating a clean separation between business logic and data sources.
Example: Managing data from multiple sources (API, database, cache) efficiently.

5. Dependency Injection
Instead of creating dependencies inside a class, they are "injected" from outside, improving testability and flexibility
Example: Providing services like network calls or database access without hardcoding them into classes.

.5 How to collect and analyze user requirements for a mobile application (Requirement Engineering).

Importance of Requirement Engineering
Before development begins, clearly defining user requirements ensures the app meets business and user needs, reduces scope creep, minimizes development rework, and aligns stakeholders (developers, designers, clients, and end-users). It also leads to more accurate cost and timeline estimates.

Steps in Requirement Engineering

1. Requirement Elicitation (Gathering)

To gather requirements effectively, multiple techniques can be used. **Stakeholder interviews** involve discussions with clients, business owners, and end-users to identify pain points, business goals, and key features. **User surveys and questionnaires**, conducted via tools like Google Forms or SurveyMonkey, help collect quantitative and qualitative data from potential users.

Market and competitor analysis involves studying similar apps to identify strengths, weaknesses, and gaps. Tools like App Store/Play Store reviews, Similar Web, and Sensor Tower can provide insights. **Focus groups and workshops** engage a small group of target users for in-depth feedback. If an existing app is being updated, analytics from tools like Firebase, Google Analytics, or Mixpanel can track user behavior to refine requirements.

The output of this phase includes a list of business objectives, user pain points, expectations, and initial feature ideas.

2. Requirement Analysis

Once requirements are gathered, they must be analyzed and structured. **Categorizing requirements** into **functional (**what the app should do, e.g., "Users should be able to log in via Google/Facebook") and **non-functional** (performance, security, scalability, e.g., "The app should load in under 2 seconds") is essential.

Prioritization follows, often using the **MoSCoW method**:

- **Must Have** – Core features without which the app fails.

- Should Have – Important but not critical for launch.

- **Could Have** – Nice-to-have features for future updates.

- Won't Have – Excluded from the current scope.

User stories and use cases help translate requirements into actionable tasks (e.g., "As a user, I want to reset my password so that I can regain access if I forget it"). **Persona development** creates fictional user profiles (e.g., "Tech-savvy millennials," "Senior citizens") to guide design and functionality.

The output of this phase includes a prioritized feature list, user stories, acceptance criteria, personas, and use-case diagrams.

3. Requirement Specification

This phase involves formalizing requirements into structured documentation. A **Software Requirements Specification (SRS)** is a detailed document outlining functional and non-functional requirements.

Wireframes and prototypes, created using tools like Figma, Adobe XD, or Balsamiq, visualize UI/UX before development begins. **Flowcharts and UML diagrams** (using tools like Lucid chart or Draw.io) illustrate user journeys and system interactions.

The output includes a finalized SRS document, clickable prototypes, and system architecture diagrams.

4. Requirement Validation

Before development starts, requirements must be validated. **Stakeholder reviews** ensure prototypes and SRS documents align with expectations**. Usability testing** (A/B testing, heuristic evaluations) helps refine the app's flow. Finally, a **formal sign-off** ensures all parties agree on the scope before coding begins.

.6 Estimating Mobile App Development Cost.

A. Key Cost Factors

Several factors influence mobile app development costs:

- App Complexity – Simple apps (basic UI, few screens) cost between $10,000 and $50,000. Medium-complexity apps (custom UI, API integrations) range from $50,000 to $100,000. Complex apps (advanced features, custom backend) exceed $100,000.

-Platform Choice– Native apps (iOS/Android) cost more due to separate codebases. Hybrid (Flutter, React Native) and Progressive Web Apps (PWAs) reduce costs but may compromise performance.

-UI/UX Design – Custom designs cost more than template-based solutions.

-Backend & APIs – Cloud services (AWS, Firebase), databases, and third-party integrations (payment gateways, maps) add to costs.

-Maintenance – Annual costs typically range from 15% to 20% of the initial development budget.

B. Cost Estimation Approaches

Three common methods help estimate costs:

1. Bottom-Up Estimation – The project is broken into small tasks (e.g., login screen, payment gateway), each estimated separately. The sum provides the total cost. This method is highly accurate and best suited for Agile teams.

2. Analogous Estimation – Costs are estimated by comparing them to similar past projects (e.g., "A food delivery app like UberEATS cost $80K, so ours might be similar"). This works well for quick ballpark estimates.

3. Parametric Estimation – A cost-per-screen or cost-per-feature model is used (e.g., basic screen: $1,000; complex feature: $5,000). This works best for standardized projects.

C. Detailed Cost Breakdown

Development costs are distributed across phases:

Planning & Research (5-10%) – Market research, competitor analysis, requirement documentation.

UI/UX Design (10-15%)– Wireframing, prototyping, branding.

Frontend Development (25-30%) – Native (Swift/Kotlin) or cross-platform (Flutter/React Native).

Backend Development (20-25%)– Server setup, APIs, databases (Firebase, AWS).

Testing & QA (10-15%)– Manual and automated testing (unit, UI, performance).

Deployment & Launch (5%) – App Store/Play Store submission.

Post-Launch Maintenance (15-20% annually)– Bug fixes, updates, feature additions.

D. Hidden Costs to Consider

Additional expenses often overlooked include:

- App Store Fees (Apple: $99/year, Google: $25 one-time).

- **Third-Party Services** (Payment gateways, SMS, Maps API).

- Security & Compliance (GDPR, HIPAA for sensitive data).

- Marketing & User Acquisition (ASO, ads, influencer partnerships).

E. Ways to Reduce Costs

To optimize budgets:

- Start with an MVP (Minimum Viable Product) – Launch core features first, then expand.

- Use Cross-Platform Frameworks – Flutter or React Native reduces costs vs. native development.

- Outsource Development – Eastern Europe and Asia offer cost-efficient talent.

- **Leverage Open-Source Tools** – Firebase for backend, TensorFlow for AI.