REPUBLIC OF CAMEROON

MINISTRY OF HIGHER EDUCATION

Peace-Work-Fatherland

REPUBLIC DU CAMEROUN

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR

Paix-Travail-Patrie

FACULTY OF ENGINEERING AND TECHNOLOGY

CEF440

INTERNET PROGRAMMING AND MOBILE PROGAMING

GROUP 23: TASK 1

Names	Matriculation Number
NDIFON LEMUEL ASHU-MBI	FE21A247
NGWASIRI RYAN TANIFIORM ONGA	FE21A266
NJIDDA SALIFU	FE21A272
ASONGNKWELLE COURAGE AYIM	FE21A142
NKWO BRAINIE NGONDA	FE21A282

SUPERVISOR:

Dr. Nkemeni Valery

MARCH, 2024

Tasks

1.Review and compare the major types of mobile apps and their differences (native, progressive web apps, hybrid apps)
2. Review and compare mobile app programming languages
3. Review and compare mobile app development frameworks by comparing their key features (Language, performance, cost & time to market, UX & UI, complexity, community support) and where they can be used.
4. Study mobile application architectures and design patterns
5. Study how to collect and analyses user requirements for a mobile application (Requirement Engineering)
6. Study how to estimate mobile app development cost.

Table of Contents

I. Mobile App Types and Comparisons

Introduction:

In today's digital age, mobile applications have become an integral part of our daily lives. From ordering food to managing finances, there's an app for almost everything.

However, for new developers entering the realm of internet applications, understanding the different types of mobile apps and their differences is crucial. In this paper, we will review and compare the major types of mobile apps: native, progressive web apps (PWAs), and hybrid apps, providing insights to help developers make informed decisions when embarking on app development projects.

a. Various Mobile App Types

i. Native Mobile Apps:

A native app is a software application designed specifically for a particular operating system, like Android or iOS for phones and tablets, or macOS or Windows for computers.

- ***Definition*:** Native mobile apps are developed specifically for a single platform using platform-specific programming languages and tools, such as Swift for iOS or Java/Kotlin for Android.

Characteristics:

- *Performance:* Native apps typically offer the best performance and responsiveness because they are optimized for a particular platform.
- *User Experience:* They provide a seamless user experience by leveraging platform-specific UI elements and guidelines.
- *Access to Device Features:* Native apps have full access to device features like camera, GPS, and push notifications.
- *Distribution:* They are distributed through platform-specific app stores (e.g., Apple App Store, Google Play Store).

Examples: Instagram (iOS), Google Maps (Android), Snapchat (iOS and Android).

ii. Progressive Web Apps (PWAs):

***Definition*:** PWAs are web applications that use modern web technologies to provide a native app-like experience within a web browser.

Characteristics:

- *Cross-Platform Compatibility*: PWAs work across different platforms and devices, including desktops, tablets, and smartphones.
- *Offline Functionality:* They can function offline or with a poor internet connection by caching resources and data.
- *Responsive Design:* PWAs are designed to be responsive and adaptive to various screen sizes and orientations.
- *Built with web technologies:* PWAs are created using common web technologies like HTML, CSS, and JavaScript, allowing them to run on any device with a modern web browser.

- *Discoverability:* PWAs are discoverable through search engines and can be easily shared via URLs.

Examples: Twitter Lite, Flipkart, Starbucks.

iii. Hybrid Mobile Apps:

Definition: Hybrid mobile apps are built using web technologies like HTML, CSS, and JavaScript, then wrapped in a native container for deployment.

Characteristics:

- *Cross-Platform Development:* Developers can write code once and deploy it across multiple platforms (iOS, Android, etc.).
- *Access to Native Features:* Hybrid apps can access certain native features through plugins or APIs, although performance might be impacted.
- *Faster Development:* They offer faster development cycles compared to native apps, as developers can reuse code across platforms.
- *Distribution:* Hybrid apps are distributed through app stores like native apps.

Examples: Instagram (older version built with React Native), LinkedIn, UberEATS.

b. Comparison of Mobile App Types

- **Performance**
  Native apps generally offer the best performance, followed by hybrid apps, while PWAs might lag behind due to limitations of browser environments.
- **Development Time**
  PWAs and hybrid apps typically have shorter development cycles compared to native apps, as they allow code reuse across platforms.
- **Access to Device Features**
  Native apps have the most comprehensive access to device features, followed by hybrid apps, while PWAs have limited access.
- **Distribution and Reach:**
  Native and hybrid apps are distributed through app stores, whereas PWAs are accessible via web URLs, potentially reaching a wider audience.

II. Review and Comparisons of Mobile App programming languages

i. Review of Mobile App programming languages

(a) Java:

I. ***Platform*:** Android

II. ***Overview:*** Java has been the primary language for Android app development since the inception of the platform.

III. Strengths:

- ✓ *Robust Ecosystem:* Java boasts a vast ecosystem of libraries, tools, and frameworks for Android development.
- ✓ *Performance:* Apps written in Java typically offer good performance and responsiveness.
- ✓ *Community Support:* There's a large community of Java developers providing support and resources.

IV. Weaknesses:

- ✓ *Verbosity:* Java code tends to be verbose compared to other modern languages, which can lead to longer development cycles.
- ✓ *Memory Management*: Manual memory management in Java can lead to memory leaks and performance issues if not handled properly.

(b) Kotlin:

I. **Platform:** Android

II. **Overview:** Kotlin is a modern programming language developed by JetBrains, officially supported by Google for Android app development.

III. Strengths:

- ✓ Conciseness: Kotlin offers concise syntax compared to Java, resulting in shorter and more readable code.
- ✓ Interoperability: Kotlin is fully interoperable with Java, allowing developers to leverage existing Java libraries and frameworks.
- ✓ Safety Features: Kotlin includes features like null safety and immutability by default, reducing the likelihood of runtime errors.

IV. **Weaknesses:**

- ✓ Learning Curve: While Kotlin's syntax is generally considered more modern and intuitive, there may be a learning curve for developers transitioning from Java.

(c) Swift:

I. **Platform:** iOS, macOS, watchOS, tvOS

II. **Overview:** Swift is a powerful and intuitive programming language developed by Apple for iOS and macOS app development.

III. Strengths:

- ✓ Safety: Swift includes modern features like optional and type inference, enhancing code safety and reducing errors.
- ✓ Performance: Swift is optimized for performance, offering fast execution and low memory footprint.
- ✓ Expressiveness: Swift's concise syntax and expressive features enable developers to write clean and maintainable code.

IV. **Weaknesses:**

- ✓ Limited Platform Support: Swift is primarily used for iOS and macOS development, limiting its applicability to other platforms.

(d) JavaScript (and Frameworks like React Native):

I. **Platform:** Cross-platform (iOS, Android, Web)

II. **Overview:** JavaScript, along with frameworks like React Native, enables cross-platform mobile app development.

III. Strengths:

- ✓ Cross-Platform Compatibility: JavaScript allows developers to write code once and deploy it across multiple platforms.
- ✓ Large Community: JavaScript has a vast and active developer community, providing ample resources and support.
- ✓ Rapid Development: Frameworks like React Native offer fast development cycles by enabling code reuse and hot reloading.

IV. Weaknesses:

- ✓ Performance: Cross-platform apps built with JavaScript may suffer from performance issues compared to native apps.
- ✓ Dependency on Frameworks: Developers using frameworks like React Native may face limitations or dependencies on framework-specific features.

ii. Comparison of Mobile App programming languages

- ✓ **Platform Specificity:** Java and Kotlin are primarily used for Android development, while Swift is exclusively for iOS and macOS. JavaScript with frameworks like React Native allows for cross-platform development.

- ✓ **Performance:** Swift and native languages (Java/Kotlin) generally offer better performance compared to JavaScript-based solutions.

- ✓ **Community and Ecosystem:** Java, Kotlin, and JavaScript have large and active developer communities, offering extensive libraries, tools, and resources.

- ✓ **Learning Curve:** Kotlin and Swift are considered more modern and user-friendly compared to Java, which might have a steeper learning curve for beginners.

III. Exploring Mobile App Development Frameworks and Comparisons

Introduction:
Mobile app development frameworks play a crucial role in simplifying the app development process, allowing developers to create cross-platform applications efficiently. However, choosing the right framework requires careful consideration of various factors, including programming language, performance, cost, user experience, complexity, and community support. We will review and compare popular mobile app development frameworks based on these key features, providing insights to help new developers select the most suitable framework for their projects.

i. Mobile App Development Framework Types

a. React Native:

- ✓ **Language:** JavaScript
- ✓ **Performance:** React Native offers good performance, leveraging native components for optimal user experience.
- ✓ **Cost & Time to Market:** Development costs and time to market are relatively low with React Native due to code reuse across platforms.
- ✓ **UX & UI:** React Native enables the creation of native-like user interfaces with its component-based architecture and third-party libraries.
- ✓ **Complexity:** React Native simplifies cross-platform development but may require familiarity with JavaScript and React concepts.
- ✓ **Community Support:** React Native has a large and active community, providing extensive documentation, libraries, and support.
- ✓ **Where to Use:** Ideal for building cross-platform apps with a focus on user interface and performance.

b. Flutter:

- ✓ **Language:** Dart
- ✓ **Performance:** Flutter delivers high-performance apps with its fast-rendering engine and native-like performance.
- ✓ **Cost & Time to Market:** Development costs are relatively low with Flutter due to code reuse, and time to market is accelerated with its hot reload feature.
- ✓ **UX & UI:** Flutter offers customizable and expressive UI components, enabling developers to create visually appealing interfaces.
- ✓ **Complexity:** Flutter's declarative UI approach simplifies development, but developers need to learn Dart programming language and Flutter framework.

- ✓ **Community Support**: Flutter has a growing community with ample resources, documentation, and packages available.
- ✓ **Where to Use:** Suitable for building cross-platform apps with rich and interactive user interfaces.

c. Xamarin:

- ✓ **Language**: C#
- ✓ **Performance:** Xamarin provides native performance as it compiles to native code, offering excellent performance and responsiveness.
- ✓ **Cost & Time to Market:** Development costs may be higher with Xamarin due to its licensing fees, but time to market is accelerated with code sharing capabilities.
- ✓ **UX & UI:** Xamarin allows developers to create native user interfaces with platform-specific UI elements and customization options.
- ✓ **Complexity:** Xamarin development may require proficiency in C# and understanding of platform-specific APIs.
- ✓ **Community Support:** Xamarin has a strong community and is backed by Microsoft, offering extensive documentation, forums, and support.
- ✓ **Where to Use:** Ideal for enterprises and developers familiar with C# looking to build cross-platform apps with native performance.

d. Ionic:

- ✓ **Language:** HTML, CSS, JavaScript
- ✓ **Performance:** Ionic apps offer good performance, although not as fast as native apps due to reliance on web technologies.
- ✓ **Cost & Time to Market:** Development costs are low with Ionic, and time to market is accelerated with its rapid development approach.
- ✓ **UX & UI:** Ionic provides a variety of pre-designed UI components and themes, allowing for fast and visually appealing app development.
- ✓ **Complexity:** Ionic simplifies cross-platform development with web technologies, making it accessible to web developers.
- ✓ **Community Support:** Ionic has a large and active community, offering extensive documentation, plugins, and support.
- ✓ **Where to Use:** Suitable for building cross-platform apps with a focus on rapid development and cost efficiency.

ii. Mobile App Development Framework Comparisons

- **Language:** Frameworks use different programming languages such as JavaScript (React Native), Dart (Flutter), C# (Xamarin), and HTML/CSS/JavaScript (Ionic).

- **Performance:** Flutter and Xamarin offer native-like performance, while React Native and Ionic provide good performance but may not match native speeds.

- **Cost & Time to Market:** Development costs and time to market vary depending on the framework, with React Native and Ionic being cost-effective and fast.

- **UX & UI:** All frameworks enable the creation of visually appealing user interfaces, with Flutter and Xamarin providing more native-like experiences.

- **Complexity:** Frameworks like Flutter and React Native simplify cross-platform development, while Xamarin may have a steeper learning curve due to C#.

- **Community Support:** Each framework has its own community, providing resources, documentation, and support to developers.

IV. Mobile Application Layers and Design patterns

i. Mobile App Architectures:

An architecture defines the overall structure of your app, including its components, their communication, and data flow. Here are some common mobile app architectures:

1. Model-View-Controller (MVC)

This is a classic architecture that separates the app into three parts:

- ✓ Model: Manages the app's data and business logic.
- ✓ View: Handles the user interface (UI) elements and presentation.
- ✓ Controller: Receives user input, updates the model, and instructs the view to update accordingly.
- ✓ MVC promotes loose coupling between components, making the app easier to maintain and test.

2. Model-View-Presenter (MVP)

An evolution of MVC, MVP introduces a Presenter layer between the View and Model.

- ✓ Presenter: Acts as an intermediary, receiving user input from the View, updating the Model, and formatting data for the View.

MVP improves code organization and testability compared to MVC.

3. Model-View-View Model (MVVM)

Another variation, MVVM uses a View Model that sits between the View and Model.

- ✓ ViewModel: Holds data relevant to the View and handles UI logic. It doesn't directly access the Model but fetches data through the controller or a dedicated data layer.
- ✓ MVVM is popular for complex UIs with data binding capabilities.

4. Layered Architecture

This architecture organizes the app into horizontal layers with well-defined functions. Common layers include:

- ✓ Presentation Layer: Handles UI elements and user interaction.
- ✓ Business Logic Layer: Implements core app functionalities.
- ✓ Data Access Layer: Manages data persistence and retrieval.

Layered architecture offers modularity and promotes separation of concerns.

ii. Mobile App Design Patterns

Design patterns are reusable solutions to common mobile app development problems. They help developers write clean, maintainable, and efficient code. Here are some commonly used design patterns:

- ✓ **Adapter**
  Allows incompatible interfaces to work together. (e.g., using a common interface for handling data from different APIs)
- ✓ **Singleton**
  Ensures only one instance of a class exists throughout the app. (e.g., a user session manager)
- ✓ **Observer:**
  Defines a one-to-many relationship where an object (subject) notifies its dependents (observers) about changes. (e.g., notifying multiple UI components about data updates)
- ✓ **Factory**
  Creates objects without specifying the exact type upfront, promoting flexibility. (e.g., a factory for creating different types of database connections)
- ✓ **Facade**
  Provides a simplified interface to a complex subsystem. (e.g., a login facade that handles all login-related logic)

Using design patterns effectively can improve your app's code quality, readability, and overall maintainability.

iii. Benefits of using Architectures and Design Patterns:

1. Improved Code Quality: Well-defined architectures and design patterns promote clean, modular, and reusable code.
2. Easier Maintenance: The app becomes easier to understand, modify, and extend due to its structured design.
3. Enhanced Scalability: The architecture can be adapted to accommodate future growth and new features.
4. Reduced Development Time: Leveraging existing patterns saves time for developers who don't need to reinvent the wheel.

V. Data Collection and Analyzation for Mobile App Development

Introduction:
In the realm of mobile application development, understanding and accurately capturing user requirements is essential for delivering successful and user-centric apps. This process, known as requirement engineering, involves systematically gathering, analyzing, and documenting user needs and expectations. In this paper, we will explore the fundamentals of collecting and analyzing user requirements for mobile applications, providing guidance for new developers to ensure their projects meet user needs effectively.

i. Understanding the Importance of User Requirements:

- ✓ **User-Centric Design:** User requirements form the foundation of user-centric design, ensuring that mobile apps fulfill user needs and expectations.
- ✓ **Minimizing Rework:** Accurately capturing user requirements minimizes the need for rework and iterations during the development process, saving time and resources.
- ✓ **Enhancing User Experience:** Apps built based on well-defined user requirements are more likely to provide a positive and intuitive user experience, leading to higher user satisfaction and retention.

ii. Techniques for Collecting User Requirements

a. Interviews: Conducting interviews with potential users, stakeholders, and domain experts to gather insights into their needs, preferences, and pain points.

b. Surveys and Questionnaires: Distributing surveys or questionnaires to a broader audience to collect quantitative data on user preferences and behavior.

c. Observations: Observing users in their natural environment or using similar existing apps to understand their workflow, challenges, and interactions.

d. Workshops and Focus Groups: Facilitating workshops or focus groups to brainstorm ideas, gather feedback, and collaboratively define user requirements.

e. Prototyping: Creating prototypes or mockups to visualize app concepts and gather early feedback from users before development begins.

For optimal data collection, these 2 points of interest must be kept in mind

1. **Analyzing User Requirements:**

   a. **Prioritization:** Prioritize user requirements based on their importance, impact on the app's functionality, and alignment with business objectives.

   b. **Feasibility Assessment:** Assess the technical feasibility of implementing each requirement within the constraints of time, budget, and technology stack.

c. **Requirement Traceability:** Establish traceability between user requirements and system functionalities to ensure that each requirement is adequately addressed during development.

d. **Validation and Verification:** Validate user requirements through user acceptance testing and verify that the implemented features meet user expectations.

2. Documenting User Requirements:

- ✓ **Use Cases**: Describe typical user interactions with the app, including various scenarios, inputs, outputs, and system responses.
- ✓ **User Stories:** Write user-centric narratives describing specific tasks or goals that users want to accomplish with the app.
- ✓ **Functional Requirements:** Specify the functional capabilities and features that the app must deliver to meet user needs.
- ✓ **Non-Functional Requirements:** Document non-functional aspects such as performance, usability, security, and scalability criteria.

VI. How to estimating Mobile App Development Cost

Introduction:

Estimating the cost of mobile app development is a crucial step in planning and budgeting for app projects. However, accurately predicting development costs can be challenging due to various factors that influence the overall expenses. In this paper, we will explore the key considerations and methodologies involved in estimating mobile app development costs, providing insights to help new developers navigate this aspect of app development effectively.

i. Factors Influencing Mobile App Development Cost:

a. **Complexity of Features**

The number and complexity of features desired in the app significantly impact development costs. Features like user authentication, real-time messaging, and payment integration require more time and resources to implement.

b. **Platform Compatibility**

Developing for multiple platforms (iOS, Android) increases development costs compared to targeting a single platform. Each platform has its development requirements and may require separate teams or additional effort for cross-platform compatibility.

c. **Design Complexity**

High-quality and intricate designs contribute to development costs, as they require skilled designers and more time for implementation.

d. **Third-Party Integrations**

Integrating third-party services or APIs (e.g., maps, payment gateways) adds complexity to the development process and may incur additional costs.

e. **Backend Infrastructure**

The need for a robust backend infrastructure, including servers, databases, and APIs, adds to the overall development costs.

f. Maintenance and Updates

Ongoing maintenance, support, and updates post-launch should be factored into the cost estimation to ensure the app's long-term viability.

ii. Methods for Estimating Mobile App Development Cost:

a. **Bottom-Up Estimation**

Break down the project into smaller components (features, screens) and estimate the time and resources required for each component. Multiply by the hourly rate of developers to calculate the total cost.

b. Top-Down Estimation

Use historical data or industry benchmarks to estimate the overall cost based on similar projects' average costs.

c. Analogous Estimation:

Compare the current project to past projects with similar scope and complexity to estimate the cost based on previous experience.

d. Parametric Estimation

Use predefined parameters or models based on project characteristics (e.g., size, complexity) to estimate development costs.

iii. Cost Components in Mobile App Development

a. Development Costs: Includes expenses related to design, coding, testing, and debugging of the app.

b. Design Costs: Covers UI/UX design, wireframing, prototyping, and visual design elements.

c. Infrastructure Costs: Includes expenses for backend development, server hosting, databases, and APIs.

d. Maintenance and Support Costs

Estimates ongoing expenses for bug fixes, updates, server maintenance, and user support post-launch.

e. Marketing and Distribution Cost: Budget for marketing efforts, app store fees, and promotion to attract users and increase app visibility.

VII. References

- https://github.com/cis-projects/project_based_course_notes
- https://clutch.co/directory/mobile-application-developers/pricing
- https://www.goodfirms.co/resources/cost-to-develop-an-app
- https://www.atlassian.com/agile/project-management/user-stories
- https://uxdesign.cc/mobile-ui-13-basic-patterns-of-app-ui-design-to-know-about-d3f7c6176f13
- https://developer.android.com/topic/architecture
- https://www.linkedin.com/posts/nitish-kumar-619414154_activity-6980202868301594624-TG8q?trk=public_profile_like_view
- https://medium.com/@falsecrypt/microservices-architecture-integrating-mobile-clients-1e9b932cd971