REPUBLIC OF CAMREOON
Peace - Work – fatherland
MINISTRY OF HIGHER EDUCATION
UNUVERSITY OF BUEA

REPUBLIQUE DU CAMEROUN
Paix – Travail – Patrie
MINISTERE L'ENSEIGNEMENT SUPERIER
UNIVERSITE DE BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

CEF440: INTERNET PROGRAMMING(J2EE) AND MOBILE PROGRAMMING PROJECT REPORT

PRESENTED BY:

NAME	MATRICULE
KENFACK SAMEZA VICTORIN-JOY	FE21A213
MBUNGAI GEORGE BERINYUY	FE21A234
FOZAO JARID NZOLEFACK	FE20A042
POKAM NGOUFFO TANEKOU	FE21A299
MBUA SEDRICK GOBINA	FE21A233

COURSE COORDINATOR
DR. NKEMENI VALERY

TASK 1

Question 1 :
Review and compare the major types of mobile apps and their differences (native, progressive web apps, hybrid apps).

1. **Native Apps:**
   Native apps are developed specifically for a **particular mobile operating system**, such as iOS or Android. They are built using **platform-specific programming languages and tools**, such as Swift or Objective-C for iOS and Java or Kotlin for Android.

Advantages:

- High performance: Native apps are optimized for the specific platform they are built for, resulting in faster and smoother performance.
-
- Access to device features: They can utilize the full range of device capabilities, such as camera, GPS, push notifications, and more.
-
- Better user experience: Native apps can provide a seamless and consistent user experience that matches the platform's design guidelines.

Disadvantages:

Development time and cost: Building separate apps for different platforms requires more development effort and resources.

Maintenance: Updates and bug fixes need to be implemented separately for each platform.

Examples of native apps:

1) Whatsapp
2) Facebook

2. **Progressive Web Apps (PWAs):**
   PWAs are web applications that are designed to look and function like native apps. They are **accessed through a web browser** but can be installed on a user's home screen for quick access.

Advantages:

- Cross-platform compatibility: PWAs can run on any device with a modern web browser, regardless of the operating system.
-
- Easy updates: Since they are web-based, updates to PWAs can be deployed instantly without requiring users to download and install updates.
-
- Offline functionality: PWAs can work offline or with limited connectivity, allowing users to access certain features and content without an internet connection.

Disadvantages:

Limited access to device features: PWAs have restricted access to certain device capabilities compared to native apps.

Performance limitations: Although PWAs have made significant improvements, they may still have performance limitations compared to native apps.

Limited discover-ability: PWAs are not listed in app stores, which can make it harder for users to discover and install them.

Examples: Weather.com web app, Pinterest.

3. **Hybrid Apps:**
   Hybrid apps are a combination of native and web apps. They are built using web technologies (HTML, CSS, JavaScript) and then wrapped in a native container that allows them to be distributed through app stores.

Advantages:

Cross-platform development: Hybrid apps can be developed once and deployed on multiple platforms, reducing development time and cost.

Access to device features: Hybrid apps can leverage device capabilities by using frameworks like Apache Cordova or React Native.

Easier maintenance: Updates and bug fixes can be applied to all platforms simultaneously, reducing maintenance efforts.

Disadvantages:

Performance limitations: Hybrid apps may not achieve the same level of performance as native apps, especially for complex and resource-intensive tasks.

Dependency on third-party frameworks: Hybrid apps rely on frameworks, which may introduce limitations or compatibility issues.

User experience inconsistencies: Hybrid apps may not perfectly match the native user interface, leading to slight inconsistencies across platforms.

Examples: Instagram.

Question 2: Review and compare mobile app programming languages

1. **Swift:**
   - Developed by Apple for iOS, mac-OS, watch-OS, and tvOS app development.
   - Known for its safety features, concise syntax, and modern language design.
   - Provides robust support for iOS frameworks and libraries.
   - Offers features like optionals, type inference, and memory management.
   - Swift is considered more beginner-friendly compared to Objective-C.

2. **Objective-C:**
   - The primary language used for iOS and mac OS app development before Swift.
   - Offers a dynamic runtime, allowing for powerful runtime manipulations.
   - Supports broader compatibility with older Apple frameworks and libraries.
   - Objective-C code can coexist with Swift in the same project.
   - Considered more verbose and complex compared to Swift.

3. **Java:**
   - The main language for Android app development.
   - Offers a large ecosystem, extensive libraries, and tools.
   - Known for its platform independence, allowing Java code to run on multiple platforms.
   - Provides features like garbage collection, multi-threading, and exception handling.
   - Java has been widely adopted and has a vast developer community.

4. **Kotlin:**
- Introduced as a modern alternative to Java for Android development.
- Fully interoperable with Java, allowing seamless integration with existing code bases.
- Offers concise syntax, null safety, and enhanced readability.
- Supports functional programming concepts and co-routines for asynchronous programming.
- Kotlin has gained popularity due to its developer-friendly features and improved safety.

5. **C#:**
- Used for developing cross-platform mobile apps with Xamarin (Microsoft's framework).
- Xamarin allows code sharing between iOS and Android platforms.
- C# is known for its strong typing, garbage collection, and extensive framework support.
- Provides access to native APIs and performance optimizations.
- Xamarin.Forms enables building UIs with shared code across platforms.

NOTE:
When comparing these programming languages, factors to consider include platform support, performance, development tools and frameworks, community support, learning curve, and the specific requirements of your project. Each language has its strengths and weaknesses, and the choice often depends on the target platform, existing code-base, and personal preferences of the development team.

Question 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.

1. **React Native:**
- Language: JavaScript
- Key Features: Code sharing between iOS and Android, hot-reloading for faster development, large ecosystem of community-driven packages.
- Performance: Generally performs well, but can be slower than native apps for complex UI or heavy computations.
- Cost & Time to Market: Faster development due to code sharing, potentially reducing costs and time to market.
- UX & UI: Native-like look and feel, but may require additional work for pixel-perfect UI.

- Complexity: Moderate complexity, especially for complex UI or platform-specific features.
- Community Support: Large and active community with extensive resources and support.
- Usage: Cross-platform app development for iOS and Android.

2. **Flutter**:
- Language: Dart
- Key Features: Fast and customizable UI development, hot-reloading, extensive widget library, single code base for iOS and Android.
- Performance: High-performance apps due to the use of the Skia rendering engine and native-like UI components.
- Cost & Time to Market: Faster development with a single code base, potentially reducing costs and time to market.
- UX & UI: Offers a customizable UI with its own rendering engine, allowing for visually appealing and native-like interfaces.
- Complexity: Moderate complexity, especially for complex UI or platform-specific features.
- Community Support: Growing community with increasing popularity, along with official support from Google.
- Usage: Cross-platform app development for iOS, Android, web, desktop (experimental).

3. **Xamarin**:
- Language: C#
- Key Features: Code sharing between iOS and Android, access to native APIs, extensive libraries and tools.
- Performance: Provides native-like performance as the code is compiled to native binaries.
- Cost & Time to Market: Development time can be reduced by sharing code, but may require additional effort for platform-specific UI and features.
- UX & UI: Can achieve platform-specific UI and native-like experiences.
- Complexity: Moderate to high complexity, especially for complex UI and platform-specific features.
- Community Support: Active community with resources and support from Microsoft.
- Usage: Cross-platform app development for iOS and Android.

4. **Native-script:**
- Language: JavaScript, Typescript
- Key Features: Access to native APIs, UI components, and libraries, code sharing between iOS and Android.

- Performance: Native performance as the code is executed directly on the device.
- Cost & Time to Market: Development time can be reduced with code sharing, but platform-specific UI and features may require additional effort.
- UX & UI: Can achieve platform-specific UI and native-like experiences.
- Complexity: Moderate to high complexity, especially for complex UI and platform-specific features.
- Community Support: Active community with resources and support from Progress Software.
- Usage: Cross-platform app development for iOS and Android.

5. **Ionic:**
- Language: HTML, CSS, JavaScript, Typescript
- Key Features: Uses web technologies, code sharing across multiple platforms (including web), extensive UI components.
- Performance: Performance can be slower compared to native apps, particularly for complex or resource-intensive applications.
- Cost & Time to Market: Development time can be reduced with code sharing, potentially reducing costs and time to market.
- UX & UI: Uses web-based UI components, resulting in consistent but not fully native UI experiences.
- Complexity: Generally considered less complex due to web technologies, but may face limitations for complex native integration.
- Community Support: Active community with resources and support.
- Usage: Cross-platform app development for iOS, Android, and web.

NOTE:
Each framework has its unique strengths and considerations. The choice depends on factors such as project requirements, team expertise, performance needs, desired development speed, and budget. It's essential to evaluate these factors to make an informed decision for your specific app development needs.

Question 4: Study mobile application architectures and design patterns.

Mobile application architectures and design patterns play a crucial role in building scalable, maintainable, and robust mobile apps. Here are some common mobile application architectures and design patterns you should study:

1. **Model-View-Controller (MVC):**
   - One of the oldest and widely used architectures.
   - Separates the application into three components: Model (data and business logic), View (user interface), and Controller (handles user input and updates the model and view).
   - Helps maintain separation of concerns and facilitates code organization.
   - Often used in combination with other patterns to achieve a more modular and scalable architecture.

2. **Model-View-Presenter (MVP):**
   - Similar to MVC but emphasizes a clearer separation between the presentation and business logic layers.
   - The Presenter acts as an intermediary between the View and the Model, handling the UI updates and user interactions.
   - Facilitates test-ability, as the business logic can be decoupled from the UI components.
   - Popular in Android development, especially with frameworks like Moxy and Mosby.

3. **Model-View-View-Model (MVVM):**
   - Focuses on a clear separation between the View, View Model, and Model layers.
   - The View Model exposes data and commands to the View and handles the presentation logic.
   - Utilizes data binding to establish a connection between the View and the View Model.
   - Supports better test-ability and promotes the use of reactive programming and data-driven UI updates.
   - Widely used in frameworks like Xamarin, Flutter, and libraries like Jet-pack Compose.

4. **Clean Architecture:**
   - Emphasizes separation of concerns and independence of external frameworks or libraries.
   - Divides the application into different layers: Presentation, Domain (business logic), and Data (persistence and external services).
   - Enforces dependency inversion and dependency injection principles.
   - Helps achieve test-ability, maintainability, and scalability.
   - Enables easy replacement of components and adaptability to changing requirements.
   - Popularized by Robert C. Martin (Uncle Bob).

5. **Reactive Programming:**

- A programming paradigm focused on handling asynchronous data streams and propagating changes.
- Utilizes reactive extensions (Rx) libraries to compose and transform streams of data.
- Enables declarative and responsive programming.
- Commonly used in conjunction with other architectures like MVVM, MVP, or Clean Architecture.
- Prominent libraries include RxJava, RxSwift, RxJS, and ReactiveCocoa.

6. **Dependency Injection (DI):**
- A design pattern that promotes loose coupling and modular development.
- Dependencies are injected into a class rather than being created or managed by the class itself.
- Helps decouple components, improve test-ability, and facilitate code reuse.
- Popular DI frameworks include Dagger (Android), Swinject (iOS), and Koin (multi-platform).

NOTE:
These are just a few examples of mobile application architectures and design patterns. It's essential to study and understand them to make informed decisions when developing mobile apps. Each pattern has its strengths and considerations, and the choice depends on factors such as project requirements, team expertise, and scalability needs. Additionally, it's common to combine multiple patterns or adapt them to suit specific application needs.

Question 5: Study how to collect and analyse user requirements for a mobile application (Requirement Engineering)

Collecting and analyzing user requirements is a crucial step in the mobile application development process. Here are some key steps and techniques involved in requirement engineering for a mobile application:

1. **Elicitation:**
- Identify stakeholders: Determine the individuals or groups who have an interest in the mobile application, such as end-users, clients, business owners, and subject matter experts.
- Conduct interviews: Engage in one-on-one or group interviews with stakeholders to understand their needs, expectations, and pain points related to the mobile application.

- Perform surveys: Distribute questionnaires or online surveys to gather a broader range of feedback from a larger audience.
- Organize workshops: Conduct collaborative sessions or workshops with stakeholders to brainstorm ideas, gather requirements, and facilitate discussions.

2. **Documentation**:
- Capture requirements: Document the gathered information, including functional requirements (what the app should do) and non-functional requirements (performance, security, usability, etc.).
- Use cases and user stories: Create use cases or user stories to describe the interactions and expected behavior of the application from the perspective of different user roles.
- User personas: Develop user personas to represent different user types and their characteristics, needs, and goals.
- Prototypes: Create low-fidelity or high-fidelity prototypes to visualize and validate the requirements with stakeholders.

3. **Analysis and Prioritization**:
- Analyze requirements: Review and analyze the collected requirements for clarity, completeness, and consistency.
- Resolve conflicts: Identify and address any conflicting or ambiguous requirements by consulting with stakeholders and seeking clarification.
- Prioritize requirements: Assign priorities to requirements based on their importance, impact on the application, and business goals.
- Define MVP: Determine the Minimum Viable Product (MVP) by identifying the core features and functionalities that must be implemented in the initial version of the application.

4. **Validation and Verification**:
- Validate requirements: Validate the requirements with stakeholders to ensure they accurately represent their needs and expectations.
- Use prototypes: Utilize prototypes or mock-ups to demonstrate the proposed functionality and gather feedback from stakeholders.
- Review and feedback: Conduct regular review sessions with stakeholders to obtain their input and address any concerns or changes.
- Requirement traceability: Establish traceability between requirements, design elements, and test cases to ensure that the application meets the specified requirements.

5. **Documentation Management**:
- Maintain a requirements document: Keep a well-organized and up-to-date requirements document to serve as a reference throughout the development process.

- Change management: Establish a process for managing changes to requirements, including proper documentation, impact analysis, and obtaining stakeholder approval for modifications.

NOTE:

It's important to note that requirement engineering is an iterative process, and requirements may evolve and change over time. Regular communication and collaboration with stakeholders, as well as involving them in the validation and verification process, are crucial for successful requirement engineering.

By following these steps and techniques, you can effectively collect and analyze user requirements for a mobile application, ensuring that the final product meets the needs and expectations of its intended users.

Question 6: **Study how to estimate mobile app development cost.**

Estimating the cost of mobile app development involves considering various factors and aspects of the project. While the actual cost can vary significantly depending on the complexity and scope of the app, here are some key steps to help estimate the cost:

1. Define the Project Scope:
   - Determine the purpose and goals of the app.
   - Define the target platforms (iOS, Android, or both).
   - Identify the key features and functionalities required in the app.

2. Break Down the Development Phases:
   - Identify the major development phases, such as UI/UX design, front-end development, back-end development, testing, and deployment.
   - Estimate the effort and time required for each phase based on the project requirements and complexity.

3. Consider Design and User Experience:
   - Evaluate the complexity of the app's user interface (UI) and user experience (UX) requirements.
   - Determine if custom designs, animations, or complex interactions are needed.
   - Consider the need for responsive design to accommodate different devices and screen sizes.

4. Back-end Development and APIs:

- Assess the complexity of server-side development and integration with APIs or third-party services.
- Consider the need for user authentication, data storage, real-time communication, or external integrations.
- Evaluate the effort required to develop and maintain the back-end infrastructure.

5. Integration of Third-Party Services:
- Determine if the app requires integration with third-party services like payment gateways, social media platforms, mapping services, analytic tools, etc.
- Consider the effort and potential costs associated with integrating these services and any associated licensing or usage fees.

6. Testing and Quality Assurance:
- Estimate the time and effort required for testing the app across multiple devices, platforms, and scenarios.
- Consider the need for automated testing, performance testing, security testing, and user acceptance testing.

7. Project Management and Support:
- Account for project management activities, including communication, coordination, and ongoing support.
- Consider the need for updates, bug fixes, and future enhancements after the initial release.

8. Consider External Factors:
- Take into account the hourly rates of developers and the development team's location.
- Consider any additional expenses, such as licenses for development tools, cloud hosting, or marketing efforts.

9. Request and Compare Quotes:
- Reach out to multiple app development agencies or freelancers to obtain quotes based on the provided requirements and scope.
- Compare the proposed costs, timelines, and expertise of the development teams.

NOTE:
It's important to note that estimating the cost of mobile app development is inherently challenging, and unexpected complexities may arise during the development process. It's advisable to work closely with an experienced development team and keep room for contingencies in the budget.

Additionally, ongoing maintenance and updates should be considered as part of the overall cost beyond the initial development phase.