UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

REPUBLIC OF CAMEROON

PEACE-WORK-FATHERLAND

COURSE TITLE:

INTERNET PROGRAMMING (J2EE) AND MOBILE PROGRAMMING

COURSE CODE:

CEF440

REPORT ON TASK 1

GROUP 20

TIOKENG SAMUEL	FE19A110
KUE GILLES	FE22A235
LUM BLESSING N.	FE22A239
NWETBE LORDWILL	FE22A285
TALLA TIZA	FE22A304

COURSE INSTRUCTOR

DR. NKEMENI VALERY

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

Mobile App Types Comparison :

Native Apps :

Native apps are built specifically for a single platform, like iOS (using Swift or Objective-C) or Android (using Kotlin or Java), with tools provided by the platform's ecosystem (e.g., Xcode for iOS, Android Studio for Android).

Performance: They're fast and responsive because they're optimized for the specific OS and hardware, leveraging native APIs directly.

Access: Full access to device features (camera, GPS, accelerometer, etc.) without restrictions.

User Experience: Smooth, platform-specific UI that aligns with iOS or Android design guidelines (e.g., Material Design for Android). Offline Capability: Works well offline with local storage.

Development Cost/Time: Separate codebases for each platform mean higher costs and longer timelines.

Maintenance: Updates must be tailored and deployed separately for iOS and Android.

Use Case: Best for apps needing top performance or deep hardware integration—like games (e.g., PUBG Mobile) or complex tools (e.g., Adobe Photoshop Express).

Progressive Web Apps (PWAs) :

PWAs are web apps built with HTML, CSS, and JavaScript that run in a browser but mimic native app behavior. They use service workers for offline functionality and can be "installed" on a device via a browser point.

Cross-Platform: One codebase works across all devices with a modern browser, slashing development time and cost.

No App Store: Users access them via URLs, avoiding app store fees and approval processes.

Updates: Instant updates without user intervention.

Lightweight: Smaller footprint than native apps, often just a few MB.

Limited Access: Restricted access to some device features (e.g., advanced Bluetooth or full push notifications on iOS).

Performance: Slower than native apps due to browser overhead.

Discovery: Harder to find since they're not in app stores (though SEO helps).

Use Case: Ideal for businesses wanting broad reach with minimal investment—like Starbucks' PWA for ordering or news sites like Forbes.

Hybrid Apps:

Hybrid apps blend native and web approaches, using a single codebase (often with frameworks like Ionic, React Native, or Flutter) wrapped in a native shell. They run web content (HTML, CSS, JS) inside a native container via WebView.

Cost-Efficient: One codebase for iOS and Android reduces development effort.

Faster Development: Frameworks like React Native speed up coding with reusable components.

App Store Presence: Can be distributed via app stores, unlike PWAs.

Some Native Access: Plugins (e.g., Cordova) allow use of device features, though not as seamlessly as native apps.

Performance: Slower than native apps due to the WebView layer; animations and heavy computations can lag.

User Experience: May not fully match platform-specific UI/UX expectations. Complexity: Bridging native and web code can introduce bugs or delays.

Use Case: Good for apps needing moderate device access and quick deployment—like Instagram (early versions) or simple business apps.

Comparison Snapshot :

Aspect	Native	PWA	Hybrid
Performance	High	Moderate	Moderate
Development cost	High	Low	Medium
Device access	Full	Limited	Partial
Distribution	App stores	Web	App stores
Offline support	Strong	Strong(w/efffort)	Moderate
Maintainance	complex	simple	moderate

Bottom Line :

- Choose Native if you need peak performance, full device integration, or a premium UX—and budget isn't a constraint.
- Choose PWAs for broad accessibility, low overhead, and rapid deployment, especially if app store presence isn't critical.
- Choose Hybrid for a balance—faster development

Question 2: **Review and compare mobile app programming languages**

Comparison of Mobile Programming Languages:
Mobile development relies on several programming languages, which vary depending on the platform (iOS, Android, or cross-platform). Here is a comparison of the main languages used.

Swift (for iOS):
- Modern language designed by Apple.

- Clear and expressive syntax.

- Excellent performance on iOS.

- Enhanced security due to strict type management.

- Exclusive to the Apple ecosystem.

Kotlin (for Android):
- Official language for Android development, created by JetBrains.

- Modern and concise syntax, interoperable with Java.

- Reduces boilerplate code compared to Java.

- Adopted by Google, ensuring continuous evolution.

- Mainly dedicated to Android.

Java (for Android and other platforms):
- Object-oriented language, widely used.

- Historically used for Android development.

- Large community and extensive documentation.

- Proven reliability and robustness.

- More verbose syntax compared to Kotlin.

JavaScript / TypeScript (for hybrid frameworks):

- Used in frameworks like React Native or Ionic.

- Allows sharing code across multiple platforms.

- Large ecosystem and numerous development tools.

- Lower performance compared to native languages.

- Depends on plugins to access advanced hardware features.

Dart (for Flutter):

- Developed by Google, used with Flutter.

- Object-oriented and highly performant language.

- Enables cross-platform application development.

- Excellent performance close to native.

- Smaller community compared to other languages.

Comparison Table:

Language	Platform	Advantages	Disadvantages
Swift	iOS	High performance, modern syntax	Exclusive to Apple
Kotlin	Android	Modern, interoperable with Java	Mainly for Android
Java	Android, Server	Reliable, large community	Verbose syntax
JavaScript	Cross-platform	Single codebase for multiple platforms	Lower performance
Dart	Cross-platform	Near-native performance, modern UI	Less widespread

Conclusion:

The choice of language depends on the project requirements:
- For iOS applications, Swift is the recommended choice.
- For Android, Kotlin is preferred for its modernity.
- For cross-platform applications, Dart (Flutter) or JavaScript (React Native) offer good alternatives.

Each language has its strengths and weaknesses, so it is essential to choose based on constraints and objectives.

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.**

- ❖ **Introduction:**

A development framework is a structured foundation or platform that provides tools, libraries, and conventions to help developers build applications efficiently by handling common tasks, allowing them to focus on unique features.

Mobile app development frameworks help developers create apps for smartphones and tablets. We will review and compare some popular frameworks based on their main features.

- ➢ Review of mobile app development frameworks

1. React Native:

A cross-platform UI framework that allows developers to build native-like apps for Android and iOS with a single codebase.

- Key Features:

Language: Uses JavaScript and JSX.

Performance: Runs almost as fast as native apps.

Cost & Time to Market: Faster to develop because you can reuse parts of the code, which saves money and time.

UX & UI: Offers a smooth experience with customizable components.

Complexity: Medium difficulty; you need to know JavaScript and React.

Community Support: Large community with many resources and libraries.

- Use Cases:

Great for cross-platform apps, especially for startups.

2. Flutter:

A framework that uses JavaScript to build native mobile apps for Android and iOS, leveraging a large and active community.

- Key Features:

Language: Uses Dart.

Performance: Very fast because it compiles directly to native code.

Cost & Time to Market: Quick development with one codebase for both iOS and Android.

UX & UI: Many widgets available to create beautiful designs.

Complexity: learning Dart can take some time.

Community Support: Growing community with lots of plugins.

- Use Cases:

Perfect for apps that need great performance and design, like e-commerce or games.

3. Xamarin

A cross-platform framework that allows developers to build native apps for Android and iOS using C# and XAML.

- Key Features:

Language: Uses C#.

Performance: Good performance, but not as fast as React Native or Flutter.

Cost & Time to Market: Can be slower and more expensive if you need lots of native features.

UX & UI: Uses native UI components for a familiar look.

Complexity: Medium to high; you need to know C# and the .NET framework.

Community Support: Strong support from Microsoft with good documentation.

- Use Cases:

Best for enterprise apps and projects that already use .NET.

4. Ionic:

A framework that uses web technologies (HTML, CSS, JavaScript) to build cross-platform apps.

- Key Features:

Language: Uses HTML, CSS, and JavaScript.

Performance: Performance can vary; mostly web-based, which might affect speed.

Cost & Time to Market: Quick to develop using web skills, which can lower costs.

UX & UI: Provides good UI options but may not feel as native.

Complexity: Easy; web developers can quickly learn it.

Community Support: Active community with many templates and plugins.

- Use Cases:

Ideal for hybrid apps, especially those that need to be developed quickly.

5. Native Development (Swift/Java/Kotlin):

- Key Features:

Language: Swift for iOS, Java/Kotlin for Android.

Performance: Best performance, fully optimized for each platform.

Cost & Time to Market: Higher costs and longer development time due to separate codebases.

UX & UI: Excellent user experience with full access to device features.

Complexity: High; requires expertise in platform-specific languages.

Community Support: Strong support for both Android and iOS.

- Use Cases:

Ideal for apps that need top performance and specific features, like games or complex apps.

Summary Table

Framework	Language	Performance	Cost & Time to Market	UX & UI	Complexity	Community Support	Use Cases
React Native	JavaScript	Almost native	Fast	Smooth, customizable	Medium	Large	Cross-platform apps, MVPs
Flutter	Dart	Very high	Quick	Beautiful, customizable	Medium to high	Growing	High-performance apps, e-commerce, games
Xamarin	C#	Good	Slower	Native look	Medium to high	Strong (Microsoft)	Enterprise apps, .NET projects
Ionic	HTML/CSS/JS	Variable	Quick	Good but not native-like	Easy	Active	Hybrid apps, content-heavy applications
Native	Swift/Java/Kotlin	Best	High	Excellent	High	Strong	Resource-intensive apps, games

Some other frameworks are; jQuery mobile, Agular, Solar 2D

❖ **Conclusion**

Choosing the right mobile app development framework depends on your project needs, budget, and team skills. React Native and Flutter are great for cross-platform apps, while Xamarin fits well with enterprise solutions. Ionic is good for quick hybrid apps, and native development is best for high-performance applications. Consider these factors to find the right framework for your project.

<u>Flutter (Google):</u>

Why it's used: Known for its fast development speed, rich UI component set, and excellent performance, making it suitable for complex applications.

O <u>Cross-Platform:</u> You can write the code once and run it on both iOS and Android.

O <u>Great UI</u>: It has lots of customizable widgets to create nice-looking apps.

O <u>Hot Reload:</u> You can see changes immediately, which speeds up development.

Compared to older frameworks: Flutter's focus on <u>performance</u> and a <u>modern UI approach</u> sets it apart from older frameworks like Xamarin, which, while also cross-platform, might have a less intuitive developer experience.

React Native (Facebook/Meta):

Why it's used: Favored for its large community, JavaScript-based development, and the ability to reuse web development skills.

- **Reusability**: You can use code from web apps, making it easier to share components.
- **Strong Community**: There are many libraries and tools available to help developers.
- **Native Performance**: It provides an experience close to native apps by using native components.

Compared to older frameworks: React Native's strong community support and JavaScript-based approach make it a good choice for teams with web development experience, while older frameworks like Xamarin might require developers to learn new languages and ecosystems.

Xamarin (Microsoft):

Why it was used: Xamarin's extensive ecosystem with C#, .Net, and Microsoft Visual Studio made it one of the most complete mobile app frameworks available.

Why it's less common: Xamarin has been superseded by .NET MAUI, a newer framework that offers similar functionality with a more modern approach.

Ionic:

Why it was used: Ionic facilitates the development of cross-platform mobile applications.

- **Web Skills**: It's easy for web developers to learn and use.
- **Quick Development**: It allows fast prototyping and deployment.

Why it's less common: Ionic might not offer the same performance and native UI experience as Flutter or React Native, making it less suitable for complex applications.

Similarities:

- **Cross-Platform Development**:

React Native, Flutter, Xamarin, and Ionic allow developers to write code once and deploy it on both iOS and Android, reducing development time and costs.

- **Community Support**:

All frameworks have active communities that provide resources, libraries, and plugins, making it easier for developers to find help and tools.

- **UI Components**:

Each framework offers a variety of UI components to help developers create visually appealing applications. They focus on providing a good user experience.

- **Rapid Development**:

Most frameworks, especially Flutter and Ionic, emphasize fast development cycles with features like hot reload and quick prototyping.

Key Differences Explained base key factors:

- **Language**: Each framework uses different programming languages, with React Native using JavaScript, Flutter using Dart, Xamarin using C#, and Ionic leveraging web technologies (HTML/CSS/JavaScript). Native development relies on Swift for iOS and Java/Kotlin for Android.
- **Performance**: Native development offers the best performance, while Flutter is known for its high speed due to direct compilation. React Native provides near-native performance, whereas Ionics performance can vary since it relies on web technologies.

- **Development Speed**: Flutter and Ionic are designed for rapid development, while Xamarin can be slower due to the need for platform-specific code. Native development usually takes longer as it requires separate codebases for each platform.
- **UI/UX**: Flutter excels in providing rich and customizable UI components. React Native uses native components for a familiar look, while Ionic offers good UI but may not feel fully native. Native development provides the best user experience as it's fully optimized for the platform.
- **Complexity**: Ionic is the easiest to pick up for web developers, while native development is the most complex due to the need for deep knowledge of platform-specific languages.

Question 4: **Study mobile application architectures and design patterns**

Mobile Application Architectures:

Mobile application architectures provide structured frameworks for organizing code in mobile development, ensuring applications are maintainable, scalable, and testable. These architectures define interactions between key components—user interface (UI), data management, and business logic—which is critical for apps dependent on internet connectivity and real-time data. Below are four prominent architectures widely used in mobile development:

- MVC (Model-View-Controller): Common in iOS, MVC splits responsibilities into:
- Model: Handles data and business logic.
- View: Displays the UI.
- Controller: Manages input and updates the Model and View. Though simple, it can result in bloated controllers in complex apps.
- MVP (Model-View-Presenter):

  Popular in Android, MVP features a passive View, with the Presenter managing logic and updates. This separation improves testability by isolating logic from UI components.
- MVVM (Model-View-ViewModel):

  Leveraging data binding, MVVM connects the View to a ViewModel that exposes data and commands. Supported by frameworks like Android Jetpack and SwiftUI, it suits dynamic, data-driven UIs.
- Clean Architecture:

  This layered approach (Entities, Use Cases, Interface Adapters, Frameworks) ensures modularity and platform independence, ideal for large, scalable projects.

The choice of architecture hinges on project complexity, team skills, and specific needs—MVVM excels in UI-heavy apps, while Clean Architecture suits long-term scalability.

Design Patterns in Mobile Development:

Design patterns offer reusable solutions to recurring challenges in mobile programming, addressing issues in UI, data, networking, and asynchronous tasks. Their relevance to internet and mobile programming lies in tackling constraints like limited resources and variable connectivity. Key patterns include:

- UI Patterns:
- Adapter: Binds data to UI elements (e.g., Android’s RecyclerView, iOS’s UITableView).

- Decorator: Enhances UI components (e.g., adding animations) without altering their core.
- Data Management Patterns:
- Singleton: Ensures a single instance (e.g., database client) for consistency and efficiency.
- Repository: Unifies access to data sources (local or remote), supporting offline functionality.
- Networking Patterns:
- Retry: Reattempts failed requests, vital for unreliable networks.
- Circuit Breaker: Halts calls to failing services, boosting resilience.
- Asynchronous Patterns:
- Observer: Updates subscribers (e.g., via LiveData or Combine) on state changes.
- Promises: Simplifies async tasks (e.g., API calls) with future values. These patterns enhance robustness and efficiency, addressing mobile-specific demands like responsiveness and connectivity.

Integration with Other Development Aspects:

Architectures and patterns integrate with frontend, backend, and testing processes, ensuring a cohesive development workflow:

- Frontend Development:

  MVVM and Adapter patterns streamline dynamic UI creation, reducing manual updates and improving maintainability.

- Backend Integration:

  Clean Architecture and Repository patterns abstract data sources, enabling seamless local-remote switching for offline support.

- Testing:

  MVP and Clean Architecture isolate logic, simplifying unit testing, while Singleton ensures consistent resources in tests.

This integration ensures that all components—frontend, backend, and testing—align, enhancing the application's reliability and scalability

Question 5: **study how to collect and analyse user requirements for a mobile application (requirement engineering).**

User requirements refer to the specific needs, expectations, and constraints of the end users or stakeholders who will interact with the software system.

To build a successful mobile app that satisfies users, you need to effectively collect and analyse user requirements

1. **Define the Purpose of the App:**

This answers questions like:

- What problem does the app solve for users?
- Who are the primary users (age, gender, location, behaviour)?
- What are the key features the app must have?

2. **Methods of Collecting User Requirements:**

a. **Conducting interviews:**

Purpose: Get in-depth insights into user needs.

How: Conduct one-on-one interviews with potential or current users.

b. **Make use of surveys and questionnaires:**

Purpose: Gather data from a large group of users.

How: Distribute a set of questions, both closed (multiple choice) and open-ended (for detailed feedback).

c. **Focus Groups:**

Purpose: Encourage group discussions on app features and design.

How: Assemble a small group of potential users to discuss their expectations.

d. **User Personas:**

Purpose: Represent different user types based on research.

How: Create fictional profiles that represent typical users of the app.

e. **Observation (Usability Testing):**

Purpose: Understand how users interact with the app or similar apps.

How: Observe users performing tasks within the app or its competitors to identify challenges.

f. **Competitive Analysis:**

Purpose: Understand what works and what doesn't in similar apps.

How: Analyse features, performance, and user reviews of competitor apps.

3. **Documenting the Requirements:**

a. **Categorize Requirements:**

Which is divided into;

Functional requirements which describes features and functionalities such as login, notifications

Non-functional requirements which describes app's performance like speed security and scalability

UI/UX requirements which describes the design and usability expectations such as colour schemes, layout.

Security requirements which describes how user data is protected and privacy maintained.

b. **Prioritize Requirements:**

categorizes features such as

Must Have: Essential for the app to function.

Should Have: Important but not critical.

Could Have: Nice to have, but not necessary.

Won't Have: Not necessary in the current version.

Value vs. Effort Matrix: Prioritize features based on their value to users versus the effort required to implement them.

4. **Analyse and Validate Requirements:**

a. **Identify Conflicting Requirements:**

Some user needs may conflict, e.g., one group may want a complex UI with many options, while another group wants simplicity.

Prioritize which requirements are most important based on user personas and business goals.

b. **Use Case Development:**

Which comprises of detailed use cases that describe step-by-step how users will interact with the app and helps clarify functional requirements and ensures the app will meet user needs.

Example: "As a user, I want to receive a notification when a new message arrives, so I know when to check my inbox."

c. **Feasibility Study:**

Analyse whether the identified features are technically feasible and fit within the project constraints (time, budget, resources) and considering development tools, platform limitations, and scalability.

5. **Create Prototypes or Wireframes:**

The purpose is to visualize the app layout and flow.

It makes use of tools such as Figma, Sketch or adobe XD to create mock-ups of the user interface and app flow.

Helps stakeholders visualize the app and allow early-stage feedback.

6. **Gather Feedback and Iterate:**

It involves testing the prototypes with the users to gather feedback on the design, functionality and usability which helps the developers to refine the app's features and designs.

Continuously Improving the app by revisiting the requirements regularly based on user feedback and market changes.

To conclude, the process of collecting and analysing user requirements is crucial to the success of a mobile application. It involves:

- Understanding user needs and goals.

- Using multiple methods like interviews, surveys, and competitive analysis.

- Documenting and prioritizing requirements for better clarity and focus.

- Analysing the feasibility and identifying any conflicting requirements.

- Validating through prototypes and user feedback to ensure the final product meets users' expectations.

By following these steps, you ensure that your mobile app is user-centred, functional, and successful in the market.

Sources:

Comprehensive Guide to User Requirements for Software Success

https://www.requirement.com/requirements-gathering-and-management-for-mobile-apps/

https://thisisglannce.com/learning-centre/how-do-i-define-the-requirements-for-my-mobile-app

https://www.appivo.com/articles/7-key-steps-for-good-app-requirements/

https://www.geeksforgeeks.org/software-engineering-requirements-engeneeering-process/

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

The cost of developing a mobile application is determined based on the following factors:

1. **Complexity of the app:**
   The app's complexity defines whether it is a basic app, medium complex app or a complex app.

   a) **Basic app:**
   This app doesn't involve back end and uses simple UI designs. An example is notes. Usually requires the least time and takes the least cost. (usually takes 3-6 months)

   b) **Medium complex app:**
   This app includes features like database, authentication and API's. an example is an e-commerce app. Usually requires longer time as compared to basic apps. (usually takes 6-9months)

   c) **Complex apps:**
   This app involves AI, real time operations/services and high security. An example includes a messaging app like WhatsApp. It takes the longest time for completion and is the costliest. (usually takes 9-12 months+)

2. **Platform support:**
   This is the OS on which the app will run. Based on 3 criteria which are:

   a) **Native:**
   Which involves 2 separate code bases and usually takes the longest time and is costlier.
   b) **Cross-platform:**
   Apps are built using a single code base for both iOS and android. Takes lesser time as compared to native apps.

   c) **Hybrid platform:**
   Here, apps are built using web technologies (html, CSS and JavaScript) and wrapped in a native shell. Takes the least time and least cost.

3. **Functionality and Features:**
   Each feature like chat/messaging, location, payment, UI/UX design etc contributes to the time and cost.

4. **Development Team Experience :**
The size and hourly rates of app development teams influence project pricing significantly. A team with more experience will charge higher than a team with less experience.

5. **Testing Requirements:**
The level of functional testing and certification required before launch drives budgets higher as well.
6. **Maintenance Needs:**
The app's maintainace which includes significant care to address bugs, release enhancements per user feedback, and scale usage growth. Basically, an update.

7. **Developer's team cost:**
Which usually involves paying each developer an hourly wage. Here we have 3 categories:
freelancers: Lower cost, but require more management.

Small Agencies: Mid-range cost with balanced expertise.

Large Agencies: Higher cost but best quality and scalability.

8. **Additional cost:**
Which is cost for unforeseen circumstances.

the cost can be determined by using the formula below.
*Cost= (hourly wage * number of hours) + additional cost.*

To conclude, the complex app which requires vast functionalities and features requires more cost than a simple less featured app.

source:
Mobile App Development Cost in 2025 - (A Breakdown)
How Much Does it Cost to Develop an App in 2025? New Cost Breakdown | Blog
www.appdevelopmentcost.com.

Group member	Matricule	Contribution
TIOKENG SAMUEL	FE19A110	Question 1
KUE GILLES	FE22A235	Question 2, power point
NWETBE LORDWILL	FE22A285	Question 3, power point
TALLA TIZA	FE22A304	Question 4
LUM BLESSING N	FE22A239	Question 5, Question 6, report