REPUBLIC OF CAMEROON

PEACE-WORK-FATHERLAND

MINISTER OF HIGHER EDUCATION

UNIVERSITY OF BUEA

REPUBLIQUE DU CAMEROUN

PAIX-TRAVAIL-PATRIE

MINISTERE DE L'ENSEIGNEMENT SUPERIEURE

UNIVERSITE DE BUEA

EEF 440: Internet Program and Mobile Programming

TASK 1

Group 18

Course Instructor: ***Dr. NKEMENI Valery***

April 2024

Table of content

Major types of mobile apps

1. Native Apps:

Native apps are developed specifically for a particular platform or device using platform-specific programming languages and tools. For example, iOS apps are typically developed in Swift or Objective-C for Apple devices, while Android apps are developed in Java or Kotlin for Android devices.

Examples: WhatsApp, Facebook, waze, Spotify.

Some advantages of Native Apps

- **Performance**: Native apps tend to offer the best performance and user experience because they have access to all the device features and APIs (Application Programming Interface) provided by the platform.
- **Offline Functionality**: They can work offline and utilize device-specific features like camera, GPS, and accelerometer seamlessly.
- **Enhanced Security**: Native apps often benefit from built-in security measures provided by the platform, reducing the risk of vulnerabilities.

Some disadvantages of Native Apps

- **Platform Dependency**: Developing separate apps for different platforms (iOS, Android) can be time-consuming and costly.
- **Updates**: Updates need to be pushed through app stores, which can sometimes result in delays in reaching users.

2. Progressive Web Apps (PWAs):

Progressive Web Apps are web applications that use modern web technologies (HTML, CSS, JavaScript) to provide a native app-like experience within a web browser. : They can be installed on the user's device and appear like native apps, but they are accessed through a web browser.

Examples: Twitter, Alibaba, Uber.

Some advantages of Progressive web Apps

- **Cross-Platform**: They are cross-platform by nature, meaning they can run on any device with a web browser making them accessible across multiple platforms without the need for separate development.
- **Offline Functionality**: PWAs can work offline or in areas with poor internet connectivity by caching content and utilizing service workers.
- **Easy Installation**: They can be installed directly from the browser without going through an app store, providing a frictionless user experience.

Some disadvantage of Progressive web Apps

- **Limited Device Access**: PWAs have restricted access to certain device features compared to native apps, which may limit their functionality in some cases.
- **Browser Compatibility**: The full potential of PWAs may not be realized on older or less feature-rich browsers, impacting user experience.

3. Hybrid Apps:

Hybrid apps combine elements of both native and web applications. They are developed using web technologies like HTML, CSS, and JavaScript, then wrapped in a native container that allows them to be installed and run like a native app.

Examples: Instagram, Gmail.

Some advantage of Hybrid Apps

- **Cross-Platform Development**: Hybrid apps allow developers to write code once and deploy it across multiple platforms, reducing development time and cost.
- **Access to Device Features**: They can access native device features through plugins or APIs provided by the hybrid development framework, offering more functionality compared to PWAs.
- **Faster Development**: Using web technologies like HTML, CSS, and JavaScript accelerates the development process, especially for teams with web development expertise.

Some disadvantage of Hybrid Apps

- **Performance**: While not as performant as native apps, hybrid apps can achieve good performance for many use cases.
- **Dependency on Frameworks**: Hybrid apps rely on third-party frameworks like Apache Cordova or Ionic, which may introduce additional complexity and limitations.

Differences between the various mobile apps

1. **Development Approach**:

    - **Native Apps**: Developed using platform-specific programming languages and tools (Swift or Objective-C for iOS, Java or Kotlin for Android).
    - **PWAs**: Developed using web technologies like HTML, CSS, and JavaScript, and accessed through a web browser.
    - **Hybrid Apps**: Developed using web technologies but wrapped in a native container using frameworks like Apache Cordova or Ionic.

2. **Performance**:
    - **Native Apps**: Typically offer the best performance and user experience due to optimization for specific platforms and direct access to device features.
    - **PWAs**: Performance can vary but generally not as fast as native apps, especially for complex or graphics-intensive applications.
    - **Hybrid Apps**: Performance is generally between native apps and PWAs, with some overhead due to the additional layer of the hybrid development framework.
3. **Access to Device Features**:
    - **Native Apps**: Have full access to device features and APIs, including camera, GPS, push notifications, and offline functionality.
    - **PWAs**: Have limited access to device features compared to native apps, although this is improving with advancements in web standards.
    - **Hybrid Apps**: Can access native device features through plugins or APIs provided by the hybrid development framework, offering more functionality compared to PWAs but may not be as seamless as native apps.
4. **Distribution and Installation**:
    - **Native Apps**: Distributed through app stores (e.g., Apple App Store, Google Play Store) and installed directly on the device.
    - **PWAs**: Accessed through a web browser and can be installed directly from the browser, appearing as icons on the device's home screen.
    - **Hybrid Apps**: Distributed through app stores like native apps but may also be accessible through web browsers if designed as such.
5. **Cross-Platform Compatibility**:
    - **Native Apps**: Not inherently cross-platform; separate development is required for each platform (iOS, Android, etc.).
    - **PWAs**: Cross-platform by nature, running on any device with a compatible web browser.
    - **Hybrid Apps**: Offer cross-platform compatibility by writing code once and deploying it across multiple platforms, but may require adjustments for platform-specific features and optimizations.

Comparism between the various mobile apps

1. Development Approach:

- **Native Apps**: Developed using platform-specific languages and tools (Swift/Objective-C for iOS, Java/Kotlin for Android).
- **PWAs**: Developed using web technologies (HTML, CSS, JavaScript) and accessed through web browsers.
- **Hybrid Apps**: Developed using web technologies but wrapped in a native container using frameworks like Apache Cordova or Ionic.

2. Performance:

- **Native Apps**: Typically offer the best performance and user experience due to optimization for specific platforms and direct access to device features.
- **PWAs**: Performance can vary but generally not as fast as native apps, especially for complex or graphics-intensive applications.
- **Hybrid Apps**: Performance falls between native apps and PWAs, with some overhead due to the additional layer of the hybrid development framework.

3. Access to Device Features:

- **Native Apps**: Have full access to device features and APIs, offering seamless integration with hardware capabilities.
- **PWAs**: Have limited access to device features compared to native apps, although capabilities are expanding with evolving web standards.
- **Hybrid Apps**: Can access native device features through plugins or APIs provided by the hybrid development framework, offering a balance between native and web capabilities.

4. Distribution and Installation:

- **Native Apps**: Distributed through app stores and installed directly on devices, offering visibility and accessibility through established platforms.
- **PWAs**: Accessed through web browsers and can be installed directly from the browser, providing a frictionless installation process.
- **Hybrid Apps**: Distributed through app stores like native apps but may also be accessible through web browsers if designed as such.

5. Cross-Platform Compatibility:

- **Native Apps**: Not inherently cross-platform; separate development is required for each platform.
- **PWAs**: Cross-platform by nature, running on any device with a compatible web browser.

- **Hybrid Apps**: Offer cross-platform compatibility by writing code once and deploying it across multiple platforms, but may require adjustments for platform-specific features and optimizations.

Mobile app development programming language

1. **Java**: Primarily used for developing Android apps. It's a versatile, object-oriented language known for its performance and scalability.
2. **Kotlin**: Kotlin is now the preferred language for Android app development by Google. It's interoperable with Java, concise, and offers features like null safety and coroutines.
3. **Swift**: Swift is the main programming language for developing iOS and macOS apps. It's modern, fast, and offers safety features such as optionals and type inference.
4. **Objective-C**: Although less common now, Objective-C is still used for developing iOS and macOS apps, especially in legacy projects. It's a superset of the C programming language with object-oriented capabilities.
5. **JavaScript**: JavaScript is used for building hybrid mobile apps with frameworks like React Native and Ionic. It allows developers to write code once and deploy it across multiple platforms.
6. **HTML, CSS, and JavaScript**: These web technologies are used for developing Progressive Web Apps (PWAs), which are accessed through web browsers but can offer app-like experiences. Frameworks like Angular, React, and Vue.js are commonly used for PWA development.
7. **C#**: C# is used with the Xamarin framework for cross-platform mobile app development. Xamarin allows developers to write code in C# and compile it for both Android and iOS platforms, achieving native-like performance.
8. **Dart**: Dart is the programming language used with the Flutter framework for cross-platform mobile app development. Flutter compiles Dart code to native machine code, providing high performance and a rich set of customizable widgets.

Comparism between the various mobile application programming languages

1. **Features**:
    - **Java**: Object-oriented, platform-independent, robust standard library.
    - **Kotlin**: Modern, concise syntax, null safety, interoperability with Java.
    - **Swift**: Modern, expressive syntax, safety features like optionals and memory management.

- **Objective-C**: Object-oriented, dynamic messaging, legacy language for iOS app development.
- **JavaScript**: Dynamic, interpreted language, widely used for web development.
- **HTML, CSS, and JavaScript**: Standard web technologies for building user interfaces and adding interactivity.

2. **Popularity**:
    - **Java**: Widely used for Android app development.
    - **Kotlin**: Growing popularity for Android app development, officially supported by Google.
    - **Swift**: Primary language for iOS app development, developed by Apple.
    - **Objective-C**: Declining in favor of Swift but still used for maintaining legacy iOS projects.
    - **JavaScript**: Increasing popularity for mobile app development with frameworks like React Native and Ionic.
    - **HTML, CSS, and JavaScript**: Widely used for web development and increasingly for mobile app development with frameworks like Cordova and PhoneGap.
3. **Perfomance**:
    - **Java**: Generally good performance, but can have issues with memory management.
    - **Kotlin**: Comparable performance to Java, with some improvements in code readability and productivity.
    - **Swift**: High-performance language with low-level control, optimized for iOS and macOS platforms.
    - **Objective-C**: Generally good performance, but less optimized compared to Swift.
    - **JavaScript**: Performance can vary depending on the runtime environment and optimization.
    - **HTML, CSS, and JavaScript**: Performance can be slower compared to native languages due to the use of web views.
4. **Ecosystem**:
    - **Java**: Strong ecosystem with extensive libraries, frameworks, and community support.
    - **Kotlin**: Expanding ecosystem with support from Google and a growing community.
    - **Swift**: Strong ecosystem with extensive libraries, frameworks, and support from Apple.
    - **Objective-C**: Mature ecosystem with extensive libraries and frameworks, but declining community support.
    - **JavaScript**: Large ecosystem with numerous libraries, frameworks, and tools for web and mobile development.
    - **HTML, CSS, and JavaScript**: Extensive ecosystem with numerous libraries, frameworks, and tools for web and mobile development.

5. **Use cases**:
   - **Java**: Android app development, enterprise applications, web servers.
   - **Kotlin**: Android app development, server-side development, cross-platform development with Kotlin Multiplatform.
   - **Swift**: iOS, macOS, watchOS, and tvOS app development, server-side development with Vapor.
   - **Objective-C**: Legacy iOS and macOS app development, maintenance of existing Objective-C codebases.
   - **JavaScript**: Cross-platform mobile app development with frameworks like React Native and Ionic, web development.
   - **HTML, CSS, and JavaScript**: Cross-platform mobile app development with frameworks like Cordova and PhoneGap, web development.

Mobile app development frameworks

1. **React Native**:
   - **Language**: JavaScript
   - **Performance**: Generally good performance, though complex animations and UI interactions may suffer compared to native apps.
   - **Cost & Time to Market**: Reduces development time and cost by allowing code sharing across platforms.
   - **UX & UI**: Provides a native-like experience with access to native components and animations.
   - **Complexity**: Relatively easy to learn for web developers with knowledge of JavaScript and React.
   - **Community Support**: Large and active community with extensive libraries, components, and documentation.
   - **Where to Use**: Suitable for cross-platform mobile app development for both iOS and Android, particularly for applications requiring rapid development and code reuse.
2. **Flutter**:
   - **Language**: Dart
   - **Performance**: Excellent performance due to compilation to native code, enabling smooth animations and UI interactions.

- **Cost & Time to Market**: Can reduce development time and cost by allowing code sharing and fast development with features like hot reload.
- **UX & UI**: Offers a customizable and consistent UI across platforms with its widget-based architecture.
- **Complexity**: May have a steeper learning curve for developers new to Dart and the reactive programming model.
- **Community Support**: Growing community with a strong focus on documentation, packages, and tooling.
- **Where to Use**: Suitable for cross-platform mobile app development for both iOS and Android, especially for apps requiring high performance and a polished UI.

3. **Ionic**:
    - **Language**: HTML, CSS, JavaScript (with Angular, React, or Vue.js)
    - **Performance**: Performance can be affected by the performance of the underlying WebView, but optimizations can be made for better performance.
    - **Cost & Time to Market**: Allows for fast development and code sharing across platforms, reducing time to market.
    - **UX & UI**: Provides a native-like experience with pre-designed UI components and themes.
    - **Complexity**: Relatively low complexity, especially for web developers familiar with HTML, CSS, and JavaScript frameworks.
    - **Community Support**: Large community with extensive documentation, plugins, and support for different JavaScript frameworks.
    - **Where to Use**: Suitable for cross-platform mobile app development, particularly for web developers looking to leverage their existing skills and build apps with web technologies.
4. **NativeScript**:
    - **Language**: JavaScript, TypeScript
    - **Performance**: Provides good performance with access to native APIs and components.
    - **Cost & Time to Market**: Can reduce development time by allowing code sharing across platforms, though learning curve may impact initial development time.
    - **UX & UI**: Offers a native-like experience with access to native UI components and animations.

- **Complexity**: Moderate complexity, especially for developers familiar with JavaScript and TypeScript.
- **Community Support**: Active community with comprehensive documentation, plugins, and support for JavaScript and TypeScript.
- **Where to Use**: Suitable for cross-platform mobile app development, particularly for developers familiar with JavaScript and wanting to build native-like apps.

5. **Xamarin**:

- **Language**: C#
- **Performance**: Offers good performance with compilation to native code, though performance may be slightly lower compared to fully native apps.
- **Cost & Time to Market**: Can reduce development time by allowing code sharing across platforms, though learning curve may impact initial development time.
- **UX & UI**: Provides a native-like experience with access to native UI components and animations.
- **Complexity**: Moderate complexity, especially for developers familiar with C# and .NET.
- **Community Support**: Active community with extensive documentation, libraries, and support for C# developers.
- **Where to Use**: Suitable for cross-platform mobile app development, particularly for enterprises and developers familiar with C# and .NET.

Mobile application architectures and design patterns

1. **MVC (Model-View-Controller)**:

- **Description**: Divides an application into three interconnected components: Model (data and business logic), View (user interface), and Controller (handles user input and updates the model and view accordingly).
- **Usage**: Widely used in both web and mobile app development.

2. **MVVM (Model-View-ViewModel)**:

- **Description**: Similar to MVC but separates the view from the model through the use of view models. View models expose data and actions to the view, decoupling the presentation logic from the view.
- **Usage**: Commonly used in modern mobile app development, especially with frameworks like React Native, Flutter, and SwiftUI.

3. **MVP (Model-View-Presenter)**:

   - **Description**: Similar to MVC but places a greater emphasis on separation of concerns by introducing a presenter layer responsible for handling user interactions and updating the view and model accordingly.
   - **Usage**: Used in Android development, particularly with older Android frameworks.

4. **Clean Architecture**:

   - **Description**: Divides an application into layers, with each layer having its own set of responsibilities and dependencies. The core principles include separation of concerns, dependency inversion, and testability.
   - **Usage**: Suitable for complex mobile applications, promoting maintainability, scalability, and testability.

5. **Flux Architecture**:

   - **Description**: Unidirectional data flow architecture for managing application state. Consists of actions, dispatchers, stores, and views. Actions trigger updates to the application state, and views render based on the current state.
   - **Usage**: Commonly used with React Native and other JavaScript frameworks for building scalable and predictable mobile apps.

6. **Adapter:**

   - **Description:** Adapter is a structural design pattern that allows incompatible interfaces to work together by wrapping an existing class with a new interface. It acts as a bridge between two incompatible interfaces, converting the interface of a class into another interface that a client expects.
   - **Usage:** Adapter pattern is commonly used in mobile app development for integrating legacy code or third-party libraries with modern codebases. It enables seamless interaction between components with different interfaces.

7. **Dependency Injection**:

   - **Description**: Design pattern used to reduce the coupling between components by externalizing the creation and management of dependencies. Dependencies are injected into components rather than being created internally.
   - **Usage**: Commonly used in Android development, particularly with frameworks like Dagger and Koin, to improve testability and maintainability.

8. **Singleton Pattern**:

  - **Description**: Creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. Often used for managing shared resources, such as database connections or network clients.
  - **Usage**: Widely used in mobile app development to manage global application state and resources.

9. **VIPER Architecture**:

- **Description**: VIPER stands for View, Interactor, Presenter, Entity, and Router. It's a design pattern commonly used for building iOS applications. Each component has a specific responsibility:
  - View: User interface elements and presentation logic.
  - Interactor: Business logic and data manipulation.
  - Presenter: Acts as a mediator between the view and interactor, handling user inputs and updating the view.
  - Entity: Represents the data model or business objects.
  - Router: Handles navigation between screens.
- **Usage**: VIPER promotes separation of concerns and facilitates unit testing by breaking down the application into smaller, more manageable components. It's suitable for large and complex iOS applications.

<u>Requirement engineering process</u>

It is a systematic process for eliciting, analyzing, documenting, and managing requirements throughout the software development lifecycle. It involves understanding stakeholder needs, defining system functionality, and ensuring that the final product meets user expectations.

1. **Requirements Elicitation**:
   - **Explanation**: Gathering and understanding the needs and expectations of stakeholders for a software system.
   - **Goal**: Obtain a comprehensive understanding of user needs and expectations to guide the development process.
   - **Methods**: Interviews, surveys, workshops, observation, and brainstorming sessions.
2. **Requirements Specification**:

- **Explanation**: Documenting the gathered requirements in detail, ensuring clarity, completeness, and consistency.
- **Goal**: Define the scope, features, and functionalities of the software system to provide a clear roadmap for development and validation.
- **Artifacts**: Requirement documents, user stories, use cases, wireframes, mockups, and prototypes.

3. **Requirements for Verification and Validation**:
    - **Explanation**: Defining criteria and methods for assessing whether the software system meets the specified requirements and user expectations.
    - **Goal**: Verify and validate the software system against defined requirements and acceptance criteria to ensure compliance and identify any discrepancies or defects.
    - **Activities**: Defining acceptance criteria, developing test plans, executing testing activities (unit testing, integration testing, system testing, user acceptance testing).
4. **Requirements Management**:
    - **Explanation**: Organizing, controlling, and tracking changes to requirements throughout the software development lifecycle.
    - **Goal**: Ensure that requirements are systematically managed, documented, and communicated to stakeholders, maintaining consistency and alignment.
5. **Requirement Communication:**
    - **Definition**: Process of effectively conveying and exchanging information about requirements among stakeholders.
    - **Objective**: Ensure shared understanding of requirements, goals, and constraints.
    - **Key Aspects**: Clear documentation, stakeholder engagement, communication channels, feedback mechanisms.
    - **Processes**: Change management, version control, traceability management.

How to estimate mobile app development cost

Estimating the cost of mobile app development involves considering various factors related to project scope, complexity, resources, and other variables.

1. **Define Project Scope**:
    - Determine the objectives, features, and functionalities of the mobile app. Clearly define what the app needs to accomplish and what problems it should solve for the users.

2. **Identify Requirements**:
    - Break down the project requirements into specific features and functionalities. Identify the must-have features and any additional enhancements or integrations.
3. **Choose Development Approach**:
    - Decide whether to build a native app (iOS, Android), hybrid app, or progressive web app (PWA). The choice of development approach can impact development time and cost.
4. **Select Development Platform**:
    - Determine the platforms supported by the app (iOS, Android, or both). Developing for multiple platforms will require additional development effort and cost compared to targeting a single platform.
5. **Estimate Development Time**:
    - Break down the project into development tasks and estimate the time required for each task. Consider factors such as UI/UX design, front-end development, back-end development, testing, and deployment.
6. **Consider Complexity**:
    - Evaluate the complexity of the app based on features, integrations, third-party APIs, user authentication, data storage, and any custom requirements. More complex apps will require more development time and resources.
7. **Calculate Development Cost**:
    - Estimate the development cost based on the time required for development tasks and the hourly rates of developers. Consider rates for designers, developers, testers, and other team members involved in the project.
8. **Include Additional Costs**:
    - Account for additional costs such as project management, quality assurance/testing, app store submission fees, third-party services (e.g., cloud hosting, analytics), maintenance, and ongoing support.
9. **Factor in Contingencies**:
    - Allocate a contingency budget to account for unexpected changes, scope creep, and revisions during the development process.
10. **Get Quotes from Development Agencies**:
    - Reach out to development agencies or freelancers to get quotes based on the project requirements and specifications. Compare quotes and services offered to make an informed decision.

11. **Review and Finalize Budget**:

    - Review the estimated costs, compare them with available budgets, and finalize the budget for the mobile app development project.

12. **Monitor and Manage Costs**:

    - Throughout the development process, track expenses, monitor progress, and manage costs to ensure the project stays within budget.