FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

Group 18

CEF440: Internet Programming and Mobile Programming

BY:

Enjema Ndive Peace	**FE22A203**
Njamutoh Shalom Brenda Tasah	**FE22A271**
Ngwinkem Ketty Nerita	**FE22A269**
Tabi Atem Rogan Essembion	**FE22A300**

Supervisor

Dr Nkemini Valerie

30/03/2025

Task 1

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 analyse user requirements for a mobile application (Requirement Engineering)
6. Study how to estimate mobile app development cost.

Question 1: Review and Compare of Major Mobile App Types: Native, Progressive Web Apps (PWAs), and Hybrid Apps

Mobile applications come in different forms based on how they are developed, deployed, and function across different devices. The three major types are Native Apps, Progressive Web Apps (PWAs), and Hybrid Apps. Below is a comprehensive comparison of these types, detailing their architecture, performance, user experience, advantages, disadvantages, and best use cases.

A. Native App

Native apps are software applications developed specifically for a particular mobile operating system, such as iOS (using Swift/Objective-C) or Android (using Kotlin/Java). These apps are downloaded from app stores (Google Play Store, Apple App Store) and installed directly on the device.

Characteristics

- Built using platform-specific programming languages.
- Access full hardware features (camera, GPS, sensors, etc.).
- Installed through official app stores.
- Requires platform-specific development, leading to multiple codebases for iOS and Android.

Advantages

- **High Performance:** Optimized for the specific OS, providing the best speed and efficiency.
- **Full Hardware Access:** Can use device-specific features like biometrics, GPS, Bluetooth, and push notifications.
- **Better User Experience (UX):** Seamless interaction, smooth animations, and adherence to platform guidelines.
- **Offline Functionality:** Works without an internet connection, depending on the app's requirements.
- **Security**: More secure as they follow platform security standards and can use encryption.

Disadvantages

- **High Development Cost & Time:** Separate development for iOS and Android increases time and budget.
- **Maintenance Complexity:** Updates and bug fixes require changes in multiple codebases.
- **App Store Dependency:** Requires approval from Apple's App Store and Google Play Store before release, which can delay deployment.

Best Use Cases

- Gaming apps (e.g., PUBG, Call of Duty) requiring high performance.
- Finance & banking apps needing top security.
- Social media apps (e.g., Instagram, Snapchat) that heavily use device capabilities.
- Productivity tools requiring smooth offline usage.

B. Progressive Web Apps (PWAs)

Progressive Web Apps (PWAs) are web applications that leverage modern web technologies to provide an app-like experience on a browser. They do not require installation from an app store and can be accessed via a URL.

Characteristics

- Built using web technologies (HTML, CSS, JavaScript).
- Works across different platforms using a single codebase.
- Can be installed as a web app shortcut on a home screen.
- Uses Service Workers to enable offline capabilities and push notifications.

Advantages

- **Cross-Platform Compatibility**: A single codebase works across mobile and desktop devices.
- **Low Development & Maintenance Costs:** No need for separate iOS and Android versions.
- **No App Store Approval Needed:** Easily updated and deployed like a website.
- **Fast & Lightweight:** Smaller file size than native apps, leading to quicker load times.
- **SEO-Friendly:** Can be indexed by search engines for better discoverability.

Disadvantages

- **Limited Hardware Access:** Cannot fully access device features like NFC, Bluetooth, advanced GPS features, or biometrics.
- **Performance Limitations:** Slower compared to native apps, especially for heavy tasks like gaming or 3D graphics.
- **Browser Dependency:** Requires modern browsers like Chrome or Edge for full functionality; some features may not work on Safari.
- **Limited Offline Functionality:** While Service Workers allow some offline capabilities, it is still not as robust as native apps.

Best Use Cases

- E-commerce websites (e.g., AliExpress, Starbucks PWA).
- News platforms & blogs needing fast access (e.g., Forbes, Twitter Lite).
- Service-based apps (e.g., ride-sharing, food delivery).
- Enterprise tools where installation restrictions apply.

C. Hybrid Apps

Hybrid apps combine elements of both native apps and web apps. They are built using web technologies (HTML, CSS, JavaScript) and wrapped in a native container (like Apache Cordova, Ionic, or React Native) to function like a native app.

Characteristics

- A single codebase is used across multiple platforms.
- Installed from app stores like native apps.
- Uses web-to-native bridges to access some device features.
- Can work offline using local storage.

Advantages

- **Faster Development:** One codebase for iOS and Android reduces effort.
- **Lower Cost:** A cost-effective alternative to native development.
- **Access to Some Native Features**: Can use plugins to access GPS, camera, and push notifications.
- **Easier Maintenance:** Updates are applied universally without multiple codebase changes.

Disadvantages

- **Performance Issues**: Slower than native apps due to reliance on web views.
- **UI/UX Inconsistencies:** May not feel completely native, leading to a subpar experience.
- **Limited Native Capabilities**: Cannot fully utilize all hardware features.
- **App Store Approval Still Required:** Unlike PWAs, hybrid apps still need approval from Apple and Google.

Best Use Cases

- Content-based apps (e.g., news apps, blogs).
- Enterprise applications that do not require high performance.
- Simple utility apps (e.g., budget calculators, to-do lists).
- Prototype apps before investing in full native development.

Differences between Naive apps, PWA and Hybrid Apps

Feature	Naive Apps	PWA	Hybrid Apps
Performance	High	Medium	Medium
Development	High	Low	Medium
Maintenance	High	Low	Medium
User experience (UX)	Excellent	Good	Moderate
Offline Support	Full	Limited	Moderate
Installation	App Store	URL/Browser	App Store
Hardware access	Full	Limited	Moderate
Security	High	Moderate	Moderate
App Store	Required	Not Required	Required
Best For	High performance, feature-rich apps	Light weight, cross-platform experiences	Cost effective, multi-platform apps

Choosing the best App

The choice between native, PWA, and hybrid depends on factors like budget, performance, needs, and development time.

- **A Native App:** If we need the best performance, full hardware access, and top-notch UX (e.g. gaming, banking).
- **PWAs:** If we want a cost-effective, web-based solution with offline support (e.g. e-commerce, news).
- **Hybrid Apps:** If we need a balance between development cost and native-like features (e.g., business and utility apps).

Question 2: Review and compare mobile app programming languages

Mobile app development requires choosing the right programming language based on platform compatibility, performance, and development efficiency. Below are popular mobile app programming languages, categorized into native development, cross-platform development, and web-based mobile applications.

1. **Native Mobile App Programming Languages:** Native development provides high performance and better access to device features.

a. Swift (iOS)

Pros:

- Optimized for Apple devices.
- High performance and security.
- Strong community and official Apple support.

Cons:

- Limited to the Apple ecosystem.
- Smaller talent pool compared to other languages.

b. Kotlin (Android)

Pros:

- Modern, concise, and safer than Java.
- Officially recommended by Google for Android.
- Interoperable with Java.

Cons:

- Android-only (though Kotlin Multiplatform tries to bridge the gap).
- Slower adoption than Java.

c. Java (Android)

Pros:

- Strong legacy support for Android.
- Large developer community.
- High stability.

Cons:

- More verbose than Kotlin.
- Can be slower and less modern.

2. Cross-Platform Mobile App Programming Languages Flutter (Dart): Cross-platform development allows building apps for both Android and iOS from a single codebase.

a. **Flutter (Dart)**

Pros:

- Native-like performance.
- Rich UI with customizable widgets.
- Backed by Google.

Cons:

- Dart is less widely used than other languages.
- Large app size.

b. React Native (JavaScript)

Pros:

- Large ecosystem due to JavaScript popularity.
- Reusable components and fast development.
- Strong community support.

Cons:

- Performance lags behind native.
- Limited access to certain native modules.

c. MAUI / Xamarin (.NET, C#)

Pros:

- Strong Microsoft ecosystem support.
- Code sharing with .NET applications.

Cons:

- Less popular than Flutter or React Native.
- Some UI inconsistencies across platforms.

d. Kotlin Multiplatform (KMP)

Pros:

- Can share logic between iOS and Android while keeping native UI.
- Official support from JetBrains.

Cons:

- Requires more configuration than Flutter or React Native.
- Still evolving.

3. **Web-Based Mobile App Development (Hybrid):** Hybrid apps use web technologies and run inside a WebView.

a. Progressive Web Apps (PWAs) (HTML, CSS, JavaScript)

Pros:

- Works on any device with a browser.
- No need for App Store approval.

Cons:

- Limited access to device features.
- Not as fast or responsive as native apps.

b. Ionic (JavaScript, TypeScript)

Pros:

- Large library of pre-built UI components.
- Works with Angular, React, or Vue.

Cons:

- Depends on WebView, leading to lower performance.

Comparison Table

Language/Framework	Platforms	Performance	UI/UX	Learning curve	Community Support
Swift	iOS	High	Excellent	Medium	Strong
Kotlin	Android	High	Excellent	Medium	Strong
Java	Android	Medium-High	Good	Medium-High	Strong
Flutter (Dart)	Cross-Platform	High	Excellent	Medium	Growing
React Native (JS)	Cross-Platform	Medium	Good	Easy	Strong
Xamarin(C#)	Cross-Platform	Medium	Good	Medium	Moderate
Kotlin Multiplatform	Cross-Platform	High	Native	High	Growing
PWAs	Web	Low-Medium	Limited	Easy	Growing
Ionic	Hybrid	Low-Medium	Decent	Easy	Moderate

Choosing the right programming language depends on several factors that is:

- For iOS apps: Swift is the best choice.
- For Android apps: Kotlin is recommended (Java for legacy projects).
- For cross-platform apps: Flutter is ideal for UI-heavy applications, while React Native is good for JavaScript developers.
- For Microsoft-based apps: Xamarin is suitable for .NET developers.
- For lightweight web-based applications: PWAs and Ionic are viable options.

Each programming language has its strengths and trade-offs, and the choice should align with project requirements, budget, and developer expertise

Question 3: Review and compare mobile app development frameworks by comparing their key features

Mobile app development frameworks simplify the process of building apps by providing tools, libraries, and components. Below is a comparison of some of the most popular frameworks based on key features:

1. React Native (by Meta)

- **Language**: JavaScript (with React)
- **Performance**: Near-native performance, though slightly slower than fully native apps.
- Uses a bridge to communicate with native components.
- **Cost & Time to Market:** Faster due to code reusability between iOS and Android.
- **UX & UI:** Supports native components, giving a more authentic feel on both platforms.
- **Complexity:** Moderate; easier if you're familiar with React.
- **Community Support:** Excellent, with extensive documentation and numerous libraries.
- **Use Cases**: Social media apps (like Facebook, Instagram), e-commerce, and simple to moderately complex apps.

2. Flutter (by Google)

- **Language**: Dart
- **Performance**: High performance with native compilation. Uses Skia for rendering, achieving 60 FPS.
- **Cost & Time to Market:** Efficient due to a single codebase for multiple platforms.
- UX & UI: Highly customizable with its own rendering engine, allowing consistent UI across platforms.
- **Complexity:** Steeper learning curve due to Dart and its unique structure.
- **Community Support:** Growing rapidly with solid backing from Google.
- **Use Cases:** Apps requiring rich UI and animations (e.g., Alibaba, Google Ads).

3. Xamarin (by Microsoft)

- **Language**: C# and .NET
- **Performance**: Close to native performance, especially with Xamarin Native.

- **Cost & Time to Market:** Reduced for apps targeting both Android and iOS.
- **UX & UI:** Uses native UI components, delivering a native look and feel.
- **Complexity:** Moderate to high, especially if integrating with existing .NET applications.
- **Community Support:** Strong, with support from Microsoft and enterprise developers.
- **Use Cases:** Enterprise applications, apps requiring deep integration with the Microsoft ecosystem.

4. SwiftUI (for iOS) and Jetpack Compose (for Android)

- **Language:** Swift (SwiftUI) / Kotlin (Jetpack Compose)
- **Performance:** High performance as they are native frameworks.
- **Cost & Time to Market:** Slower if building for both platforms since they are platformspecific.
- **UX & UI:** Smooth, native interfaces with responsive design capabilities.
- **Complexity:** Low for simple apps but higher for complex UI and animations.
- **Community Support:** Good but relatively new compared to established frameworks.
- **Use Cases:** Platform-specific applications that require the best native performance.

5. Ionic (with Capacitor or Cordova)

- **Language:** HTML, CSS, JavaScript (with Angular, React, or Vue)
- **Performance:** Lower than native due to WebView rendering.
- **Cost & Time to Market:** Fast for simple apps or prototypes.
- **UX & UI**: Uses web technologies, leading to a more web-like feel.
- **Complexity:** Low to moderate, especially for web developers.
- **Community Support:** Strong, with plugins and integrations.
- **Use Cases**: Progressive Web Apps (PWAs), cross-platform mobile apps with limited native functionality.

6. Apache Cordova

- **Language**: HTML, CSS, JavaScript
- **Performance**: Low to moderate, as it runs in WebView.•
- **Cost & Time to Market**: Fast for basic apps, but performance can be an issue with complex features.

- **UX & UI**: Limited native feel, more like a web app within a mobile shell.
- **Complexity**: Simple for web developers but complex when requiring native features.
- **Community Support**: Decent but declining as newer frameworks gain popularity.
- **Use Cases**: Simple apps, prototypes, or apps with minimal native interactions.

Below is a table showing the differences between the frameworks

Framework	Language	Performance	Cost & time to market	UI &UX	Complexity	Community support
React Native	JavaScript	High	Fast	Naive-Like	Moderate	Excellent
Flutter	Dart	Very high	Fast	Rich UI	Moderate to high	Strong
Xamarin	C#/.NET	High	Moderate	Native-like	High	Strong
SwiftUI/Jetpack compose	Swift/Kotlin	Very high	Slow (platform-specific)	Native	Low to high	Good
Ionic	HTML/CSS/JS	Moderate	Very Fast	Web-like	Low to moderate	Strong
Cordova	HTML/CSS/JS	Low	Fast	Web-like	Low	Decent

Consider the following when choosing the best framework:

- Choose React Native or Flutter if you need cross-platform apps with near-native performance.
- Opt for Xamarin if your team is skilled in .NET and you need strong enterprise integration.
- Go for SwiftUI or Jetpack Compose if you're building exclusively for iOS or Android and want maximum performance.
- Use Ionic or Cordova for lightweight apps or when leveraging web development skills

Question 4: Study mobile application architectures and design patterns

Design Patterns for Mobile Development

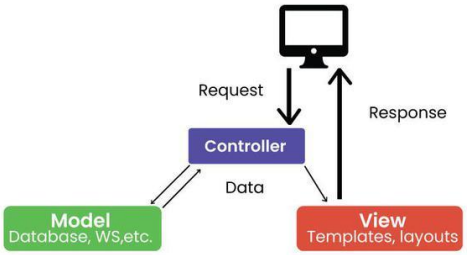
Design patterns are reusable solutions to common software development problems. They have had a significant impact on software development, including mobile app development. The implementation of mobile apps has established some proven models and standards to overcome the challenges and limitations of mobile app development.

Most mobile applications were built with low code and were not based on architecture. Mobile app development with the right design patterns can effectively integrate user interfaces with data models and business logic. This will affect the quality of your source code.

Below are mobile application Architrctures:

1) Model View Controller (MVC) Architecture

MVC is a design model that separates an application into three interacting parts: Model, View, and Controller. This separation allows for better code design and modularization



Model: Represents application data and business logic

View: Displays data to the user

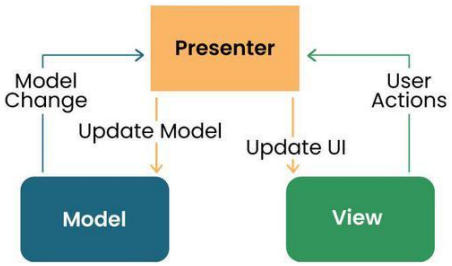
Controller: Processes user input and controls data flow between model and view

❖ For example

Imagine a mobile weather app, the Model stores weather information, the View displays it to the user and the Controller handles user interaction such as updating the displayed location or converting units (Celsius to Fahrenheit)

2) Model View Presenter (MVP) architecture

MVP is a new architecture that separates an architecture into 3 parts model,View and Presenter. This is similar to MVC but puts more responsibility on the teacher to manage the interaction between model and view.



Model: Manages data and business logic

View: represent the user interface

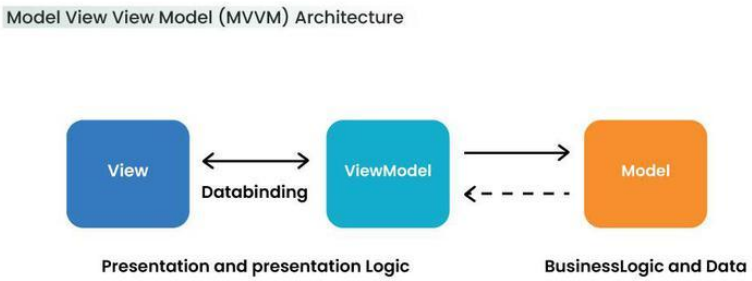
Designer: act as an intermediary processing user input and updating the View model

❖ For instance

In a note-taking app, the Model would store the text, the the View will display it and the provider would handle user input such as typing, editing or deleting the process.

3) Model View View Model (MVVM) Architecture

MVVM is a design model widely used in mobile development, especially in frameworks like Android's jetpack. It's purpose is to separate the application into 3 parts: model, View and viewmodel.



Model: Represents data and business logic

View: Represents the user interface

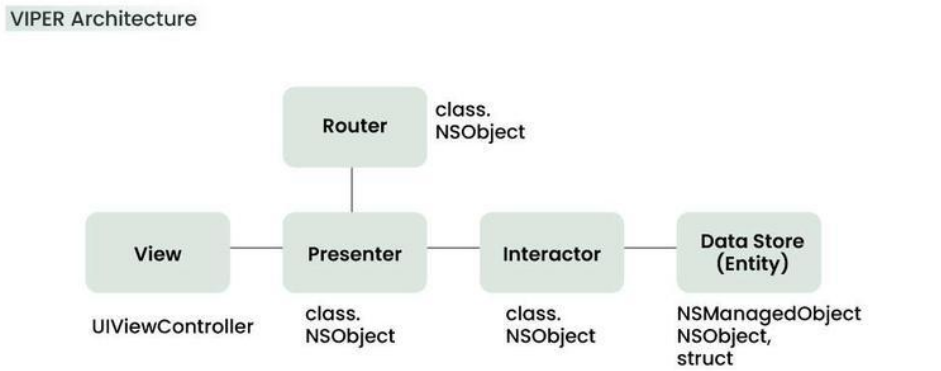
Viewmodel: acts as an interface between the model and the View which contains the reference logic

❖ For example

In an e-commerce application, the Model contains product data, the View displays product information and the Viewmodel manages interactions such as adding items to a chart

4) VIPER Architecture

VIPER stands for View, interactor, Presenter, entity and router. VIPER is primarily based at the clean Architecture ideas which propose to separate the concerns of the different layers of the utility Each layer has a single duty and communicates with different layers through a properly defined interfaces.



Below is a brief explanation of each element:

View: This is the consumer interface Layer, wherein the perspectives and Look at controllers are defined. The View is chargeable for showing the Information provided by way of the Presenter and forwarding the person Moves to the presenter

Presenter: This is the presentation Layer, where the good judgment for Formatting and imparting the records Is defined. The presenter is liable for Fetching the records from the Interactor, reworking it right into a Suitable layout for the view, and Updating the view hence. The Presenter additionally handles the Consumer movements acquired from The view and calls the router to Navigate to other screens

Interactor: This is the enterprise good Judgment layer, where the common Sense for manipulating the data and interacting with external services is Described. The interactor is Accountable for gaining access to the Facts from the service layer, acting any Vital operations on it, and returning it To the presenter. The interactor Additionally communicates with the Entity layer to store and retrieve the Information fashions

Entity: This is the information layer, Wherein the data models and systems are described. The entity is Responsible for representing the data In a constant and coherent manner Throughout the software. The entity Layer also can encompass records get Entry to gadgets (DAOs) or Repositories that summary the Information of records patience and retrieval.

Router: This is the navigation layer, Where the logic for routing and Transitioning among different monitors is defined. The router is chargeable for developing and providing the view controllers, passing any vital facts to them, and coping with any dependencies or configurations. The router also communicates with the presenter to get hold of the navigation requests and execute them.

Below are mobile application Design Pattern:

1) Singleton Method Design Pattern

The singleton policy ensures that there is only one instance of a class and provides global access.This is especially useful when you want to manage a single instance of an object or control access to a delayed object.

- For Example:

Singleton can be used to manage Player's score in mobile game There can only be one instance that is responsible for tracking scores and is updated throughout the game.

2) Factory Method Design Pattern

The Factory Method model defines an interface for creating an object but allows subclasses to modify the type of the created object. Especially useful when you need to create objects with a common interface but different functionality.

- For Example:

In a mobile app that supports multiple payment gateways, payments can be made using the Factory Method. Each payment gateway (e.g., PayPal, Stripe) is a small business and provides its services

3) Observer Method Design Pattern

The observer structure defines one to Many dependencies between objects, so when one object changes its state, all its dependents are automatically notified and updated. This is useful for scheduling distributed events.

- For Example:

In the reports app, many features (Observers) such as the title widget, the report feed view, and the notification provider (Themes) can subscribe to updates when new information arrives. The observer model ensures that they are created all registered parts report, and accordingly You can update it.

4) Dependency Injection (DI) method Design Pattern

Dependency Injection is a method of providing class dependencies from the outside, rather than creating them in the class. It improves code modularity and testability by making classes independent of their dependencies.

❖ For Example:

In an Android app, instead of creating a single database connection object in the class, you can place the database object externally, allowing you to easily test and modify database operations

5) Adapter Method Design Pattern

The adapter configuration allows you to use the interface of an existing class as a link to a new one. It is often used to work with others without modifying the source code of existing classes

❖ For Example:

If you want to use a third-party library that provides data in a different way, you can create an adapter that will convert the library's results to the format your app would expect and make sure that they are compatible meet without changing the library code.

6) Strategy Method Design Pattern

The strategy model defines a family of algorithms, contains each of them, and provides them with flexibility. It allows you to select the appropriate algorithm at runtime. This example is useful when you want to provide different options for a task.

❖ For Example:

In a weather application, you can use various methods to retrieve weather information, such as using a REST API, WebSocket, or local storage. The user can change these options, and the app adapts to his preferences.

7) Composite Method Design Pattern

A composite pattern allows you to arrange objects in a tree structure to represent a part-of-the-whole structure. This is helpful when you have to deal with individual objects and sets of objects accurately.

❖ For Example:

You can use Composite pattern to create complex shapes from simple shapes in the mobile drawing app complex designs can contain individual designs, allowing users to manipulate and categorize resources as needed.

Overall, Design processes play an important role in mobile app development by providing proven solutions to common software design challenges. Using this framework allows

developers to create maintainable, extensible, and efficient applications understanding when and how to apply these options can significantly improve the quality of your mobile app codebase. whether you’re working for Android, iOS, or any other mobile platform, a solid understanding of these design patterns will allow you to create robust, scalable mobile applications

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

In the development of a mobile application, understanding and analysing user requirements is crucial to ensure the final product meets the expectations of its intended users. This report explores the methodologies involved in collecting and analysing user requirements, a process known as requirement engineering.

Importance of User Requirements in Mobile Applications

Gathering accurate user requirements helps developers create user-friendly applications that enhance user satisfaction, reduce project costs, and minimize the need for extensive revisions post-launch.

Phases of Requirement Engineering

The requirement engineering process typically involves the following phases:

1. Requirement Elicitation: Gathering information from users and stakeholders.

2. Requirement Analysis: Examining and refining the collected requirements.

3. Requirement Specification: Documenting the requirements in a formal manner.

4. Requirement Validation: Ensuring the requirements meet the stakeholder's needs.

Techniques for Collecting User Requirements

Interviews: Conducting one-on-one or group interviews allows for in-depth discussions. This qualitative method helps in understanding user needs and expectations directly.

Surveys and Questionnaires: Surveys can reach a larger audience, providing quantitative data on user preferences. They are valuable for gathering demographic information and general feedback.

Focus Groups: Focus groups involve discussions with a diverse group of users. This method encourages interaction and can yield insights into user attitudes and perceptions.

Observation: Observing users in their natural environment can reveal unarticulated needs and behaviors, which might not be captured through interviews or surveys.

User Stories and Use Cases: User stories describe features from the user's perspective, while use cases outline specific interactions between users and the application. Both techniques help clarify requirements.

Requirement Categorization

Categorizing requirements into functional, non-functional, and domain-specific helps in organizing and managing them effectively.

Prioritization of Requirements

Not all requirements are equally critical. Prioritizing them based on user needs, project scope, and business goals ensures that essential features are developed first.

Validation and Verification

Validation ensures the requirements reflect user needs, while verification checks that the requirements are feasible and can be implemented within the project constraints.

Tools for Requirement Management

Several tools can aid in requirement management, including:

- JIRA: For tracking and managing requirements and issues.

- Trello: For organizing tasks and workflows visually.

- Confluence: For documentation and collaboration among team members.

- Lucidchart: For creating diagrams and flowcharts to visualize requirements.

Challenges in Requirement Engineering

- Ambiguity: User requirements can often be vague or unclear.

- Changing Requirements: Users' needs may evolve during the development process.

- Stakeholder Conflicts: Different stakeholders may have conflicting needs and priorities.

- Communication Gaps: Misunderstandings between users and developers can lead to incorrect implementations.

In collecting and analysing user requirements for a mobile app, Effective collection and analysis of user requirements are essential for the success of mobile applications. By employing various techniques and tools, developers can ensure that they understand the needs of their users and deliver applications that meet their expectations. Addressing challenges in requirement engineering can further enhance the quality of the final product.

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

Estimating the cost of mobile app development is essential for budgeting and project planning. The total cost depends on several factors, including platform choice, app complexity, development team, and additional costs such as maintenance and marketing.Below are key considerations and provides a structured approach to estimating mobile app development costs.

1. **Factors Affecting Mobile App Development Cost**

a. **Scope and Features:** The more features an app has, the higher the development cost. Common features include:

- User authentication (login, registration)
- Push notifications
- API integrations
- Payment gateway
- Geolocation services
- Chat/messaging
- Admin dashboard
- AI/ML features (e.g., chatbots, recommendations)

b. Platform Selection

- **iOS, Android, or Both:** Native development (Swift for iOS, Kotlin for Android) is generally more expensive than cross-platform development (Flutter, React Native).
- **Web Apps:** Often cheaper but may lack native functionality.

c. UI/UX Design Complexity

- **Simple UI**: Basic screens with minimal animations.
- **Medium UI:** Custom designs and moderate animations.
- **Complex UI:** Advanced animations, transitions, and multiple user flows.

d. Development Team Composition

Costs vary depending on the team's location and expertise:

- **Freelancers** (30,000frs-150,000frs per month)
- **Small Development Agency** (100,000frs -250,000frs per month)

- **Enterprise-Level Agency** (300,000+ monthly)

e. Development Time Estimation

Approximate development hours based on app complexity:

- **Basic App**: 500–800 hours
- **Medium Complexity** App: 800–1500 hours
- **Complex App:** 1500+ hours

2. Additional Cost Considerations

a. Backend Development

- API development, databases, and cloud services contribute to costs.

b. Maintenance and Updates

- Typically 15–20% of the initial development cost per year.

c. Marketing and Launch

- App Store and Play Store fees, advertisements, and promotions.

3. Estimated Cost Breakdown

Complexity Level	Estimated Cost
Basic App	70,000frs-180,000frs
Medium Complexity App	180,000frs-700,000frs
Complex App	750,000frs-100,000,000frs+

Overall, The cost of mobile app development depends on several variables, including features, platform, team, and ongoing maintenance. Proper planning and a clear project scope are essential for an accurate cost estimate. Businesses should also budget for post-launch updates and marketing to ensure long-term