P.O BOX 63
Buea, Cameroon
Phone No:
(237)233322134/
233322690
Fax: (237)2 33322272
Email: info@ubuea.cm

REPUBLIC OF CAMEROON
PEACE-WORK-FATHERLAND

MINISTER OF HIGHER EDUCATION

UNIVERSITY OF BUEA

DEPARTMENT OF COMPUTER ENGINEERING

Course Code: CEF440

Course name - Internet Programming And Mobile Programming

Course Instructor - Dr Nkemeni Valery

Mobile App Development Process

Task 1

Task Report

Prepared by :

GROUP 6

Prepared on:
Sunday 30th March 2025

Group Members

NAME	MATRICULE
AJIM NEVILLE BEMSIBOM	FE22A141
EBUA CINDY SANGHA	FE22A195
KINGO KINGSLEY KAAH	FE22A233
NGONG ODILO BERTILA DUFE	FE22A263
NGOUNOU NGNINKEU JOEL JONATHAN	FE22A265

Table of Content

Introduction

The mobile application landscape has evolved significantly over the years, driven by advancements in technology and changing user needs. This report aims at understanding the tools, technology and materials needed through the process of mobile app development. It also reviews the major types of mobile apps, programming languages, development frameworks, architectures, and cost estimation strategies essential for successful mobile app development.

Major Types of Mobile Applications [1]

There are basically 3 major types of mobile applications:

- Web Apps
- Native Apps
- Hybrid Apps
- Progressive Web Apps (PWAs)

1. Web Apps/Mobile Websites

They are built using standard web technologies including HTML, CSS and JavaScript. They run inside a browser which could be Safari, Chrome, Firefox etc. They are built like any regular website or web app on the internet. The only real difference is that they are designed to look good and function well on mobile devices; meaning that they are responsive, and they are designed with the mobile first approach. That is, they start with focusing on the mobile view but also make it work for desktops as well. Some examples of web applications include Gmail and office 365

1.1 <u>Advantages of Web Apps</u>

- ✓ They are easy to build (HTML/CSS/JavaScript).
- ✓ Easy to maintain.
- ✓ Use any technology/language.
- ✓ Cheaper than native and hybrid apps.
- ✓ Can be used to build a single app for all platforms (iOS, Android) as long as it has a browser.

1.2 <u>Disadvantages of Web Apps</u>

- ❖ Need to run in a browser
- ❖ Slower than Native apps
- ❖ Less interactive and less intuitive
- ❖ Cannot be submitted to app stores
- ❖ No icon on desktop
- ❖ Cannot interact with device utilities

2. Native Apps

It is the most common type of app, and it is built for specific platforms. Android apps are coded in Java, and it uses the SDK for that platform. Same thing with iOS, it's written in Swift and/or sometimes in C and it's written for the iOS platform. Some examples of Native apps include WhatsApp, Pokémon Go and Spotify.

2.1 <u>Advantages of Native Apps</u>

- ✓ Very fast
- ✓ Built to run on Specific Platforms
- ✓ Easily distributed in app stores
- ✓ Interactive and intuitive
- ✓ Easily interact with device utilities

2.2 <u>Disadvantages of Native Apps</u>

- ❖ Built for single platform
- ❖ Harder languages to learn
- ❖ Very expensive
- ❖ Hard to maintain

3. Hybrid Mobile Apps

It is a combination of native and web apps. It uses HTML, CSS and JavaScript. It also runs inside some kind of container or web view usually through a framework. So, on the surface, it can be perceived as a native app. Some examples of hybrid apps include Instagram, Evernote, Uber.

3.1 <u>Advantages of Hybrid Apps</u>

- ✓ Easy to build (HTML/CSS/JavaScript)
- ✓ Much cheaper than Native app
- ✓ Uses a single app for all platforms
- ✓ No browser needed
- ✓ Can usually access device utilities using an API
- ✓ Faster to develop than native apps

3.2 <u>Disadvantages of Hybrid Apps</u>

- ❖ Slower than Native Apps
- ❖ More expensive than web apps
- ❖ Less interactive than native apps

4. Progressive Web Apps (PWAs)

Progressive Web Apps (PWAs) are web applications that leverage modern web technologies to deliver an app-like experience to users. They combine the best features of web and native apps, offering offline functionality, fast loading times, and the ability to be installed on a device's home screen without requiring an app store download. PWAs are built using standard web technologies (HTML, CSS, and JavaScript) but incorporate service workers, web app manifests, and other advanced features to enhance performance and usability. Some examples of PWAs include Pinterest, Twitter and Flipkart.

4.1 <u>Advantages of PWAs</u>

- ✓ Fast and responsive performance than web app
- ✓ Easy maintenance
- ✓ Cheaper than Native and Hybrid apps
- ✓ Works on all platforms
- ✓ Can work offline

4.2 <u>Disadvantages of PWAs</u>

- ❖ Limited hardware and software support
- ❖ Technical options such as Bluetooth cannot be used.

Mobile App Programming Languages [2]

With advancements in technology, mobile app development has evolved significantly. Today, developers have a variety of programming languages and frameworks at their disposal to build mobile applications. These programming languages can be categorized into native and cross-platform languages, each with distinct advantages and disadvantages. Similarly, mobile app development frameworks can also be classified into native and cross-platform frameworks. This document explores these programming languages and frameworks, highlighting their principles, strengths, and weaknesses, along with the programming languages on which they are built.

Native Mobile App Programming Languages

1. Swift (iOS Development)

Swift is Apple's official programming language for iOS development, offering a modern, fast, and safe coding experience. The iOS SDK, built primarily on Swift and Objective-C, provides the necessary tools to build native iOS applications.

1.1 Strengths of Swift:

- ✓ Optimized for performance on iOS devices
- ✓ Strong type safety and error handling
- ✓ Interoperability with Objective-C
- ✓ Regular updates and strong support from Apple

1.2 Weaknesses of Swift:

- ❖ Exclusive to Apple platforms (iOS, macOS)
- ❖ Limited third-party libraries compared to other languages

2. Objective-C (iOS Development)

An older language for iOS development, built on the C language, allowing object-oriented programming. It is used within the iOS SDK for maintaining legacy applications.

2.1 Strengths of Objective-C:

- ✓ Mature language with extensive documentation
- ✓ Supports dynamic runtime capabilities
- ✓ Compatible with C and C++

2.2 Weaknesses of Objective-C:

- ❖ More complex syntax compared to Swift
- ❖ Slower development speed

3. Kotlin (Android Development)

A modern programming language officially supported by Google for Android development. The Android SDK, built primarily on Java and Kotlin, offers tools and libraries for developing native Android applications.

3.1 __Strengths of Kotlin:__

- ✓ Concise syntax and improved readability
- ✓ Fully interoperable with Java
- ✓ Enhanced null safety reducing runtime crashes

3.2 __Weaknesses of Kotlin:__

- ❖ Slower compilation speed compared to Java
- ❖ Smaller developer community compared to Java

4. Java (Android Development)

It's a widely used object-oriented language for Android development. It is a primary language for the Android SDK and is used extensively for Android application development.

4.1 __Strengths of Java:__

- ✓ It has a large developer community and extensive libraries
- ✓ It's Platform-independent with strong backward compatibility

4.2 __Weaknesses of Java:__

- ❖ Verbose syntax compared to Kotlin
- ❖ Slower performance than some modern languages

Cross-Platform Programming Languages

1. Dart

Dart is an object-oriented, class-based language optimized for UI development. Flutter, built on Dart, is a Google-backed UI framework that allows the development of cross-platform mobile applications.

1.1 __Strengths of Dart:__

- ✓ Optimized for mobile UI performance
- ✓ Fast development with Hot Reload

1.2 __Weaknesses of Dart:__

- ❖ Smaller ecosystem compared to JavaScript
- ❖ Less adoption outside of Flutter

2. JavaScript

A scripting language commonly used for web and mobile development. React Native and Ionic, built on JavaScript, allow developers to create cross-platform applications using web technologies.

2.1 __Strengths of JavaScript:__

- ✓ Extensive libraries and community support
- ✓ Code reusability for web and mobile apps

2.2 __Weaknesses of JavaScript:__

- ❖ Performance issues compared to native languages

- ❖ Limited access to native device features

3. **C#**

A Microsoft-backed language used for cross-platform mobile and desktop applications. Xamarin, built on C#, allows developers to create mobile applications with native performance and UI.

3.1 <u>Strengths of C#:</u>

- ✓ Strong integration with Microsoft technologies
- ✓ Code re-usability across multiple platforms

3.2 <u>Weaknesses of C#:</u>

- ❖ Heavier app size due to runtime dependencies
- ❖ Slower UI performance compared to native apps

Rust (Cross-Platform and Native Development)

Rust is a systems programming language that prioritizes memory safety and performance. It is increasingly being used for mobile development with frameworks like Droid, Slint, and Tauri.

<u>Strengths of Rust:</u>

- ✓ High-performance and memory safety without garbage collection
- ✓ Can be used for both native and cross-platform development
- ✓ Strong security features

<u>Weaknesses of Rust:</u>

- ✓ Smaller ecosystem for mobile development compared to Swift, Kotlin, and Java
- ✓ Steeper learning curve
- ✓ Limited tooling and libraries for mobile UI development

Mobile App Development Frameworks

Native Frameworks

1. **Android SDK (Built on Java & Kotlin)**

Strengths:

- ✓ Full access to Android APIs
- ✓ Optimized performance

Weaknesses:

- ❖ Requires extensive learning curve
- ❖ Limited code reusability

2. **iOS SDK (Built on Swift & Objective-C)**

Strengths:

- ✓ Best performance for iOS applications
- ✓ Strong Apple ecosystem integration

Weaknesses:

- ❖ Limited to Apple's ecosystem
- ❖ Higher development cost

Cross-Platform Frameworks

1. **Flutter (Built on Dart)**

Strengths:

- ✓ Single codebase for Android & iOS
- ✓ Fast development with Hot Reload

Weaknesses:

- ❖ Larger app size
- ❖ Limited third-party libraries

2. **React Native (Built on JavaScript)**

Strengths:

- ✓ Uses JavaScript (popular language)
- ✓ Large developer community

Weaknesses:

- ❖ Performance issues compared to native apps
- ❖ UI inconsistencies across platforms

3. Xamarin (Built on C#)

Strengths:

- ✓ Strong integration with Microsoft tools
- ✓ Single codebase for multiple platforms

Weaknesses:

- ❖ Heavy app size
- ❖ Slower UI rendering

Droid, Slint, Tauri (Built on Rust)

Strengths:

- ✓ High-performance and secure mobile development
- ✓ Rust's memory safety reduces security vulnerabilities
- ✓ Suitable for both native and cross-platform apps

Weaknesses:

- ❖ Smaller ecosystem for mobile compared to Flutter and React Native
- ❖ Steeper learning curve

Other Languages

C++:
Used in game development and high-performance mobile applications. Frameworks like Qt and Unreal Engine utilize C++ for mobile apps.

Python:
Though not commonly used for mobile development, frameworks like Kivy and BeeWare allow Python developers to build mobile applications.

Go (Golang):
Sometimes used for mobile backend development and experimental mobile frameworks.

Ruby:
With frameworks like RubyMotion, developers can build native iOS and Android applications.

Lua:
Mainly used in game development, with frameworks like Corona SDK (now Solar2D).

MOBILE APPLICATION ARCHITECTURES AND DESIGN PATTERNS [3]

In today's digital landscape, mobile applications play a critical role in various aspects of life, from communication and entertainment to finance, healthcare, and e-commerce. The effectiveness of these applications depends significantly on their underlying architecture and design patterns, which determine their scalability, maintainability, and overall performance.

A well-structured mobile application architecture provides a blueprint for organizing code, handling data flow, and ensuring seamless interaction between different application components. Choosing the right architecture helps developers create applications that are robust, scalable, and easier to maintain, reducing development time and effort while improving the user experience.

Similarly, design patterns serve as standardized solutions to common software development challenges. They help streamline code organization, enhance reusability, and promote efficient development practices. By adopting well-established design patterns, developers can write cleaner, more modular code, making applications easier to update, test, and debug.

This study explores the historical evolution of mobile application architectures and design patterns, their origin, components, how they function, comparisons, related examples and pros and cons of the various models.

The Historical Evolution of Mobile Application Architectures

The evolution of mobile application architecture has been shaped by advancements in hardware, software development methodologies, and user expectations. In the early days of mobile computing, applications were built as standalone, monolithic programs due to limited device capabilities and the absence of sophisticated development frameworks. As mobile technology matured, new architectural paradigms emerged to enhance scalability, maintainability, and user experience.

The Era of Monolithic Applications (1990s – Early 2000s)

In the late 1990s and early 2000s, mobile applications were primarily developed for feature phones using proprietary platforms such as **J2ME (Java 2 Micro Edition)** and **BREW (Binary Runtime Environment** for Wireless). These applications followed a monolithic architecture where all components (UI, business logic, and data storage) were tightly coupled into a single package. Due to the limited processing power and storage capacity of early mobile devices, applications were lightweight and designed for basic functionalities like messaging, games, and personal organization tools.

During this period, mobile applications had minimal interaction with external systems, and updates were rare because most applications were either pre-installed by manufacturers or sideloaded via wired connections. The lack of standardized app distribution channels further reinforced the monolithic approach, as developers had little incentive to design apps with modularity or scalability in mind.

The Rise of Client-Server Architecture and Web-Enabled Apps (Mid-2000s – Early 2010s)

With the launch of smartphones like the **iPhone (2007)** and **Android devices** shortly thereafter, the mobile landscape began shifting towards more complex architectures. Faster processors, larger storage capacities, and improved internet connectivity enabled mobile applications to communicate with remote servers, giving rise to client-server architectures.

In this architecture, mobile applications acted as clients that sent requests to remote servers, which handled data processing and storage. Web services and RESTful APIs became standard mechanisms for enabling communication between mobile clients and backend systems. During this period, many applications followed the layered (N-tier) architecture, dividing the app into distinct layers such as the presentation layer (UI), business logic layer, and data layer. This separation improved maintainability, as developers could update the server-side logic independently from the mobile client.

The emergence of hybrid applications, built using web technologies (HTML, CSS, JavaScript) and wrapped in native containers, further blurred the line between mobile and web applications. Platforms like Apache Cordova (formerly PhoneGap) allowed developers to write a single codebase that could run on multiple devices, reducing development time and cost. However, hybrid applications often suffered from performance issues, leading to the continued dominance of native applications.

The Shift to Modular and Component-Based Architectures (Mid-2010s – Present)

As mobile applications grew in complexity, the need for more scalable and maintainable architecture became evident. This led to the adoption of modular and component-based architecture such as Model-View-View Model (MVVM), Model-View-Presenter (MVP), and Clean Architecture. These architectures focused on separating concerns, making it easier to manage code, test individual components, and introduce new features without breaking the entire application.

During this period, backend development also evolved with the rise of microservices architecture. Instead of relying on a single monolithic backend, applications were now designed to communicate with multiple independent services. This shift was driven by large-scale companies such as **Netflix, Amazon, and Uber**, which needed scalable and resilient backend systems to handle millions of concurrent users. Mobile applications began integrating with cloud-based services and third-party APIs, enabling real-time data synchronization, push notifications, and machine learning-driven features.

The introduction of cloud-native technologies, serverless computing, and edge computing further transformed mobile architectures. Developers could now offload processing-intensive tasks to cloud platforms like Firebase, AWS Lambda, and Google Cloud Functions, allowing for lightweight mobile clients with enhanced functionality.

The Future: AI-Driven, Edge Computing, and Progressive Architectures

Looking ahead, mobile application architectures are likely to continue evolving toward AI-driven models, edge computing, and progressive web applications (PWAs). With the integration of artificial intelligence (AI) and machine learning (ML), applications can perform more advanced tasks such as real-time speech recognition, image processing, and predictive analytics directly on the device, reducing dependency on cloud-based services.

Additionally, edge computing is emerging as a way to bring computation closer to the user, reducing latency for applications that require real-time data processing, such as autonomous vehicles, IoT devices, and AR/VR applications. Progressive Web Apps (PWAs), which combine the best features of web and mobile applications, are also gaining traction, enabling a more seamless experience across different devices and platforms.

Mobile Application Architectures

1. Monolithic Architecture

Monolithic architecture is one of the earliest forms of software design, originating from traditional desktop applications. In this architecture, the entire application is built as a single, tightly coupled unit where all components, including the user interface, business logic, and database interactions, exist within the same codebase. This structure was popular in the early days of mobile app development when applications were relatively simple and did not require frequent updates or modularization.

In a monolithic application, when a user interacts with the interface, their actions directly trigger the underlying business logic, which then interacts with the database before returning results to the UI. This straightforward approach makes development and deployment easy since everything is contained within one unit. However, because all components are interdependent, modifying one part of the application often requires redeploying the entire app, which can become inefficient as the project grows.

2. Layered (N-Tier) Architecture

Layered architecture, also known as the N-Tier architecture, emerged from enterprise software development and the client-server model. It was designed to introduce a structured approach to application development by separating concerns into different layers. The main goal of this architecture is to improve maintainability and scalability by dividing the application into distinct layers, typically the presentation layer (UI), business logic layer, and data layer.

Each layer in this architecture is responsible for a specific function. The UI layer interacts with users and collects input, which is then passed to the business logic layer that processes this information. The processed data is then stored or retrieved from the data layer, which typically consists of databases or external storage solutions. This separation of concerns makes it easier to modify or update individual components without affecting the entire system. However, the increased complexity and additional inter-layer communication can introduce performance overhead, making this approach more suitable for large-scale applications rather than lightweight mobile apps.

3. Microservices Architecture

Microservices architecture was pioneered by companies like Netflix and Amazon to address the limitations of monolithic and layered architecture. It is an evolution of Service-Oriented Architecture (SOA) and aims to create highly scalable and maintainable applications by breaking them into smaller, independent services that communicate through APIs. Each microservice is self-contained, handling a specific function such as authentication, payments, or notifications.

In this architecture, instead of having a single, monolithic codebase, an application is split into multiple microservices that can be developed, deployed, and scaled independently. For example, in a ride-hailing application, one microservice might handle user authentication, another service could manage ride requests, while a third one processes payments. This modularity enhances fault tolerance, as failure in one service does not necessarily bring down the entire system. However, managing multiple microservices requires careful orchestration, API management, and DevOps expertise to ensure seamless communication and data consistency.

4. MVVM (Model-View-ViewModel) Architecture

The MVVM (Model-View-ViewModel) architecture was introduced by Microsoft for Windows Presentation Foundation (WPF) applications and was later adopted in mobile development, especially in Android and iOS applications. It was designed to improve the separation of concerns and enhance testability in applications that rely heavily on UI interactions and data binding.

In MVVM, the application is structured into three primary components. The **Model** represents the data layer and business logic, fetching data from APIs or databases. The **ViewModel** acts as an intermediary between the Model and the View, containing logic for UI updates and ensuring that data is presented in a format the View can use. Finally, the **View** is responsible for displaying the UI and interacting with the user. This separation allows for more modular development and makes it easier to test individual components without being tightly coupled to the UI. However, implementing MVVM requires additional setup, and improper use of data binding can lead to performance issues in mobile applications.

5. Clean Architecture

Clean Architecture was conceptualized by **Robert C. Martin (Uncle Bob)** to create maintainable, scalable, and testable applications. It was developed to address the limitations of traditional architecture, ensuring that an application's core business logic is independent of frameworks, databases, and UI technologies.

The core principle of Clean Architecture is the division of an application into different layers, each with its own responsibility. At the center are Entities, which contain core business logic and rules. Surrounding this layer are Use Cases, which define the application's specific functionalities, followed by the Interface Adapters layer, which serves as a bridge between the use cases and external systems such as UI components, databases, or APIs. The outermost layer consists of Frameworks and Drivers, including third-party libraries, databases, and the operating system itself. By enforcing strict separation between these layers, Clean Architecture ensures that the business logic remains unaffected by changes in external technologies. However, its strict adherence to principles makes development more complex, and it may be overkill for small-scale applications.

Mobile Application Design Patterns

1. MVC (Model-View-Controller)

The Model-View-Controller (MVC) design pattern originated in the 1970s as part of **Smalltalk**, one of the earliest object-oriented programming languages. It was developed to provide a clear separation between an application's user interface and its underlying logic, making it easier to develop and maintain.

In MVC, the **Model** represents the data and business logic, the **View** is responsible for displaying information to the user, and the **Controller** acts as an intermediary, handling user input and updating the Model and View accordingly. This structure allows for modular development, where changes to the UI can be made without altering the core logic. However, in mobile development, MVC often leads to bloated Controllers, making it harder to manage large applications, which is why it has been largely replaced by MVVM and other patterns in modern app development.

2. MVP (Model-View-Presenter)

The MVP (Model-View-Presenter) pattern evolved from MVC to address the issue of bloated Controllers. It was widely adopted in Android development before MVVM gained popularity.

In MVP, the **Model** contains the application's data and business logic, similar to MVC. The **View** still represents the UI but is kept as passive as possible, meaning it does not contain any business logic. Instead, all logic is handled by the **Presenter**, which acts as an intermediary between the Model and the View. The Presenter processes user inputs, updates the Model, and then modifies the View accordingly. This separation makes the application more testable and maintainable, but it also introduces additional complexity, especially in large applications with multiple Views and Presenters.

3. Singleton Pattern

The Singleton pattern is a design pattern that ensures a class has only one instance and provides a global point of access to that instance. It was first introduced as part of the Gang of Four (GoF) design patterns in the 1990s.

In mobile applications, Singletons are commonly used for managing shared resources such as database connections, network managers, or caching mechanisms. For example, an application might use a Singleton to manage API calls, ensuring that only one instance of the network manager exists throughout the app's lifecycle. While the Singleton pattern reduces memory consumption and prevents duplicate resource allocation, improper use can lead to issues like unintentional memory leaks and hidden dependencies, making unit testing more difficult.

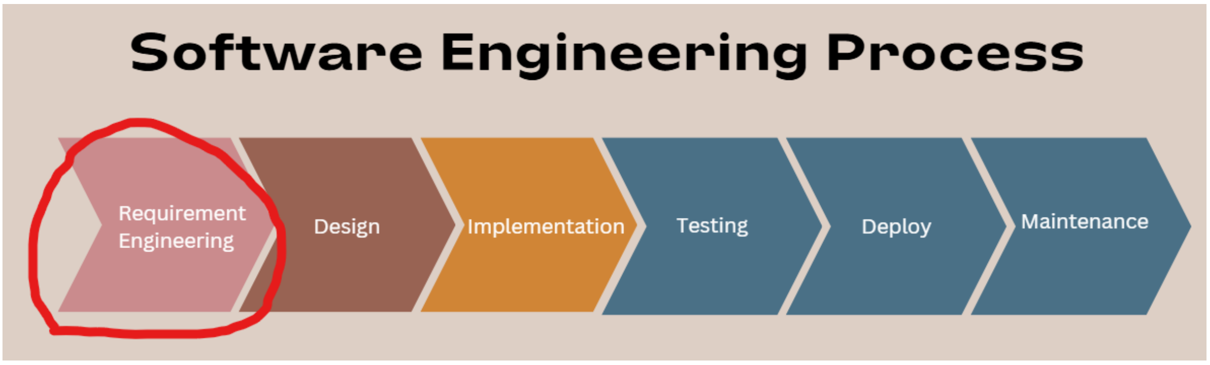
4. Factory Pattern

The Factory pattern, another design pattern from the Gang of Four (GoF), was developed to simplify object creation without exposing the instantiation logic. It provides a central method for creating instances of related objects, improving maintainability and flexibility.

In mobile applications, the Factory pattern is often used in dependency injection frameworks like Dagger and Hilt. Instead of manually creating objects, a Factory method provides an appropriate instance based on given parameters or configurations. This approach makes it easier to manage dependencies and switch between different implementations, but it also introduces an additional layer of abstraction that may complicate debugging.

Requirement Engineering

Requirement engineering is a phase in the software development lifecycle (SDLC) that focuses on identifying, documenting, and managing the requirements of a software system. It serves as the foundation for all subsequent phases of development, ensuring that the final product meets the needs and expectations of users and stakeholders.



What exactly are Requirements?

- ✓ What people want in a software system.
- ✓ What they need (they may not even know).
- ✓ Constraint (technical environment, software system, and operating environment constraint).

Importance of Requirement Engineering

1. **Clarity and Understanding**: It helps in clearly defining what the software should do, minimizing misunderstandings between stakeholders and developers.
2. **Stakeholder Satisfaction**: By involving stakeholders early in the process, requirement engineering enhances the likelihood of delivering a product that satisfies user needs.
3. **Cost Efficiency**: Early identification of requirements saves costs associated with rework and changes that often arise from misunderstandings or overlooked needs.
4. **Risk Management**: Proper requirement engineering helps in identifying potential risks early in the project, allowing for proactive management.

Phases of Requirement Engineering

Requirement engineering typically involves several key phases:

1. **Requirements Elicitation**: This phase involves gathering requirements from users and stakeholders through interviews, surveys, workshops, and other methods. The goal is to understand the needs and expectations of users, and all possible constraints.
2. **Requirements Analysis**: Collected requirements are analysed for feasibility, consistency, and completeness. Conflicting requirements are resolved, and priorities are established.

3. **Requirements Specification**: This phase involves documenting the requirements in a clear and unambiguous manner. It's a document which demonstrate the understanding of technical team. Specifications can take various forms, including use cases, user stories, and formal specifications.

4. **Requirements Validation**: The documented requirements are validated with stakeholders to ensure they accurately reflect their needs. This may involve reviews, inspections, and prototyping.

Techniques for Requirement Elicitation

Below are some techniques we can employ to elicit requirements effectively:

- **Interviews**: Direct conversations with stakeholders to gather insights.
- **Surveys and Questionnaires**: Collecting quantitative data from a larger audience.
- **Workshops**: Collaborative sessions that bring stakeholders together to discuss and refine requirements.
- **Prototyping**: Creating prototypes allows users to interact with a preliminary version of the application. This hands-on experience can help identify usability issues and gather feedback on features before full development. Prototyping fosters user engagement and helps refine requirements.
- **Analysis of Existing System:** This technique involves examining current systems that users are already using. By understanding how these systems function, we can identify strengths and weaknesses. This analysis helps determine what features are essential and what improvements can be made for the new mobile application
- **Creativity Technique:** Using creativity techniques, such as brainstorming sessions, Morphological box or mind mapping, encourages innovative thinking. These methods can help generate new ideas and solutions that may not emerge through traditional requirements-gathering methods.
- **Observation of Users during Work / Site Visit:** Observing users in their natural environment can provide valuable insights into their habits and challenges. This technique allows for firsthand understanding of user interactions with existing systems and reveals pain points that the new application can address. The problem with this approach is a phenomenon known as the Hawthorne effect, where workers behave differently when they know they're being observed.

Challenges in Requirement Engineering

1. **Ambiguity**: Requirements can often be vague or poorly defined, leading to misunderstandings.
2. **Changing Requirements**: Stakeholder needs may evolve, requiring ongoing adjustments to requirements.

3. **Communication Gaps**: Miscommunication between technical and non-technical stakeholders can lead to incomplete or inaccurate requirements.
4. **Terminology:** Different people have different understanding of the same term.

Best Practices

To mitigate challenges in requirement engineering, consider the following best practices:

- **Involve Stakeholders**: Engage users and stakeholders throughout the process to gather diverse perspectives.
- **Use Clear Language**: Document requirements in simple, clear language to avoid ambiguity.
- **Prioritize Requirements**: Establish a priority for requirements based on stakeholder needs and project goals.
- **Regular Reviews**: Conduct periodic reviews of requirements to ensure they remain relevant and accurate.
- **Maintain Traceability**: Keep track of requirement changes and their impact on the project.

Estimating Mobile App Development Costs

With the increasing growth in the mobile app industry, businesses and entrepreneurs keep asking the question of "How much will it cost to build a mobile app?" Estimating mobile application development costs requires considering various factors, with varying cost ranges depending on the complexity, features, platform and development team. Whether you plan on building a simple prototype or a complex app, understanding the cost is very important.

The cost of a developing mobile app varies depending on multiple factors, but an estimate of how much the apps may cost based on the size of the app is as shown below:

- ✓ Basic App (Minimal features) – 2,000,000 to 5,000,000 francs
- ✓ Medium Complexity – 5,000,000 to 15,000,000 francs
- ✓ Feature-Rich, Large – Scale App - 15,000,000+ francs

This calculation could be as well calculated using an **App Cost Calculator.**

App Development Cost Breakdown

There are different phases in the app development process which have different estimated costs. The different phases and estimated cost per phase is as shown below:

- ✓ **Discovery Stage**: Involves market research, competitor analysis and defining core features. Estimated cost: 5,000,000- 15,000,000 francs
- ✓ **UX and UI Design**: It enhances the user engagement and costs range from 5,000,000 to 50,000,000 francs depending on complexity and animations.
- ✓ **Mobile App Development**: Costs depending on platform and technology used
  - Native Development (iOS/Android) – 4,000,000 to 20,000,000 francs
  - Hybrid Development (Flutter, React Native) – 3,000,000 to 15,000,000 francs
  - No-Code/Low-Code Solutions - 500,000 to 5,000,000 francs
- ✓ **App Testing and Deployment**: Quality Assurance (QA) ensures a bug-free app and costs from 500,000 to 2,500,000 francs.
- ✓ **Ongoing Maintenance and Updates**: Could potentially cost 15% - 20% of the initial development cost annually, covering bug fixes, updates and server costs.

Key Factors Affecting App Development Costs

1) **Feature Scope and Complexity**: Every additional feature adds to the total cost. For example, integrating real-time chat, AI capabilities or payment gateways increases development time and expenses.
2) **UI/UX Design Considerations**: Investing in high-quality design improves user engagement bust also adds to the budget. Factors such as custom animations, interactive elements and micro-interactions can increase costs significantly.

3) **Development Approach: Native vs. Cross Platform**:
    - Native Apps (iOS and Android separately) – Higher cost, better performance
    - Cross Platform (Flutter, React Native) – Cost-effective, faster development.
4) **App Maintenance Costs**: Ongoing: Ongoing costs for bugs fixes, security updates and scaling infrastructure can range from 15% - 20% of the initial development cost annually.
5) **Development Team Experience**: Experienced and diversely skilled app development teams often cost more but also deliver much higher initial quality and require less downstream rework. Global capability diversity balances speed and affordability.

Hidden App Development Costs

- ✓ Backend Development costs about 2,000,000 – 10,000,000+ francs
- ✓ SaaS and SDK Fees: when using third-party APIs like firebase, Stripe or Mongodb will incur monthly fees ranging from 10,000 - 500,000 francs
- ✓ Server Hosting Costs: Hosting an app on AWS, Google Cloud or Azure costs 100,000 – 1,000,000 francs per year.
- ✓ Project Management Expenses: Project managers cost 500,000 – 3,000,000 francs per project.
- ✓ App Marketing Costs: Marketing costs for user acquisition and app promotion range from 1,000,000 – 10,000,000+ francs.

How to Optimize App Development Costs

- **Research Product-Market Fit Before Development**

Avoid unnecessary features by validating market demand early.
Conduct user interviews, surveys, and competitor analysis to ensure your app solves real problems.

- **Use Prototyping & MVP Development**

Start with an MVP (~1,000,000 – 5,000,000frs) to test market response.
Prototyping helps refine features before full development, reducing costly rework.

- **Streamline Project Management with Agile Methodologies**

Agile helps reduce waste and control costs during development.
Frequent iterations allow for quick adjustments based on user feedback, saving time and resources.

- **Stick to a Defined Scope**

Avoid scope creep, which increases costs by 20-50%.
Set clear project milestones and use a priority-based approach to feature development.

Conclusion

The landscape of mobile application development is dynamic and multifaceted, shaped by technological advancements and evolving user expectations. This report has explored the various types of mobile applications—web, native, hybrid, and progressive web apps—each with its unique advantages and challenges. Understanding these distinctions is crucial for us in selecting the most suitable approach for our specific needs.

Furthermore, the analysis of programming languages and development frameworks highlights the importance of choosing the right tools to enhance efficiency and performance. As mobile applications continue to grow in complexity, adopting modern architectural patterns and design methodologies becomes vital in ensuring scalability and maintainability.

Effective requirement engineering stands out as a foundational element in the development process, fostering clarity and alignment between stakeholders and developers. Accurate cost estimation is equally essential, as it can significantly impact project feasibility and success.

Overall, the insights provided in this report will serve as a guide, equipping us with the knowledge needed to navigate the complexities of mobile application development successfully.

References

1. https://youtu.be/ZikVtdopsfY , https://youtu.be/a6jLO0BadtM
2. 14 Programming Languages for Mobile App Development | Buildfire
3. https://youtu.be/tm_paZsPsrI
4. https://youtu.be/qENBiYaAXNE
5. Anant Jain – How Much Does it Cost to Develop an App in 2025