Analyse der Aufgabenstellung und Vorgehensweise

Die Aufgabenstellung besteht darin, eine bestehende iOS-App, welche in Swift entwickelt wurde, auf Android mit Jetpack Compose zu portieren. Dabei soll – wo sinnvoll – Domain Driven Design (DDD) angewendet werden. Die gezeigten Codeausschnitte geben Einblick in die Struktur der iOS-App.

Analyse der Aufgabenstellung

Wesentliche Punkte der Aufgabenstellung:

- Die iOS-App ist in Swift entwickelt und nutzt verschiedene Common-Komponenten wie Network, Storage, Logging und Konfigurationsobjekte.
- Es gibt ein zentrales Loader-Modul (`CommonApplicationBaseModuleLoader`), das u. a. folgende Aufgaben übernimmt:
  - Initialisierung der Module
  - Synchronisation von Remote- und Local-Data
  - Fehler- und Statusbehandlung (Loading, Success, Error)
  - Reset-Mechanismen (Daten löschen, erneute Synchronisation)
- Der Prozess ist stark asynchron aufgebaut (async/await in Swift).

Empfohlenes Vorgehen bei der Portierung

1. Analyse der bestehenden Architektur:
- Verstehen der Domain-Logik (ModuleLoader, Sync, Reset, Status).
- Dokumentation der bestehenden Schnittstellen und Abhängigkeiten.

2. Übertrag auf Android-Welt:
- Jetpack Compose als UI-Framework.
- Nutzung von Kotlin Coroutines (statt Swift async/await).
- Verwendung von Android-spezifischen Services (z. B. Room für Storage, Retrofit/OkHttp für Network, DataStore für Konfiguration).

3. Domain Driven Design:
- Klare Trennung von Domain, Application, Infrastructure und UI.
- Abbildung der Business-Logik in einer `Domain`-Schicht, die unabhängig von UI und Frameworks bleibt.
- ModuleLoader als zentrales Aggregat in der Domain-Schicht.

4. Synchronisations-Logik portieren:

- Abbildung des Sync/Reset-Verhaltens in Kotlin.
- Behandlung von Edge-Cases (kein Netzwerk, Remote nicht erreichbar, keine lokalen Daten).

5. Architektur auf Android:
- MVVM oder Clean Architecture als Basis.
- ViewModels, die den `CommonApplicationBaseModuleLoader`-Ansatz abbilden.
- StateFlows oder LiveData für Statusänderungen (`loadingStatus`).

6. Testbarkeit sicherstellen:
- Dependency Injection (z. B. Hilt oder Koin).
- Unit-Tests für Domain-Logik (Sync, Reset, Fehlerbehandlung).
- UI-Tests für Compose-Oberflächen.

7. Wichtige Punkte zu beachten:
- Unterschiedliche Plattformparadigmen (Swift async vs. Kotlin Coroutines).
- Fehler- und Statusbehandlung muss konsistent übertragen werden.
- Möglichst viel Domain-Logik unabhängig von Frameworks halten.
- DDD nur dort anwenden, wo es Mehrwert bringt (nicht alles erzwingen).

Fazit

Die Portierung erfordert eine sorgfältige Analyse der bestehenden Architektur und eine saubere Übertragung in die Android-Welt. Jetpack Compose bietet für die UI eine moderne Grundlage, während Domain Driven Design für eine klare Struktur sorgt. Besonders wichtig ist die saubere Trennung der Schichten, die Wiederverwendbarkeit und die Sicherstellung von Testbarkeit.