Chapter-3 柔軟なコードが必要になる実例

　このチャプターでは、私の経験から柔軟なコードが必要になった実例として、スマートフォンゲームエンジンの開発の話を紹介します。

3.1 iOS、Android、WindowsOS対応のスマートフォンゲームエンジン

　私は、スマートフォンゲームの黎明期にiOS、Android、WindowsOSで動作するスマートフォンゲームエンジンの開発を行ったことがあります。そのゲームエンジンでは、ゲームプログラマがC++でゲームのコードを書けるようにすることを目的としたエンジンでした。
　当時はちょうどUnityの黎明期で、私もこのエンジンの開発の前にUnityを利用したスマートフォンアプリの開発を行っていました。しかし、著者自身のUnityに対する理解の浅さと、Unityの黎明期ということで、今のように色々な最適化のノウハウもまだまだ足りていなかったという側面もあって、UnityとUnityで使えるC#という言語について、「パフォーマンスが出せない」、「メモリ管理を自由に行えない」といったネガティブな印象を持ってしまいました。そこで、次の開発に向けて、この二つの問題を解決するために、C++でコードを書けるiOS、Android向けのスマートフォンゲームエンジンを開発する流れになったのです。
　幸いにして、iOSはC++のスーパーセット言語のobjective C++、AndroidはJava言語を利用するのですが、C++でネイティブコードを開発することができるAndroid NDKというものが用意されていたため、また、グラフィックスAPIはともにOpenGLESを採用していたため、C++でゲームを作成することは可能でした。しかし、C++で記述できるとはいえ、iOSはobjective C++、AndroidはJavaという大きな環境の違いがあったため、まったく同じコードで両者で動くゲームを開発できるわけではありませんでした。
　また、当時のスマートフォン向けのゲーム開発の開発環境は貧弱なものしかなく、まともにスマートフォンゲームのデバッグができるようなものではありませんでした。iOSは幾分ましでしたが、Androidに至ってはC++で書かれたコードをまともにデバッグできるものはありませんでした。そこで、普段はVisualStudioという優秀なIDEを使うことができる、Windows上で開発を行えるようにするという目標も生まれました。
　このような背景から、iOS、Android、WindowsOSで動作するスマートフォンエンジンの開発がスタートしました。

3.2 クリアすべき課題

　まず、クリアしなくてはいけない課題として、iOS、Android、WindowsOSでゲームアプリ―ケーション層のプログラムを統一できるようにする必要がありました。前節で解説したように、iOSもAndroidもWindowsOSもC++で開発することができます。しかし、すべての処理を同じコードで記述することができるかというと、それは不可能でした。
　例えば、タッチパネルへのタッチの検出ですが、iOSとAndroidではタッチパネルの操作を取得するためのAPIがそもそも違います。また、WindowsOSに関してはタッチパネルでの操作は想定していないため、マウス操作をタッチパネルの操作に変換する必要がありました。その他にもファイル入出力、スレッド関係の処理、通信などなど、iOS、Android特有のAPIを利用するコードは軒並みNGです。次のコードはファイル入出力処理のiOS、Androidの疑似コードです。
[iOS]

```
// test.csvから1行読み込み
NSString *filePath = @"/Users/test/Desktop/test.csv";
NSString *text = [NSString stringWithContentsOfFile:filePath
encoding:NSUTF8StringEncoding error:nil];
```

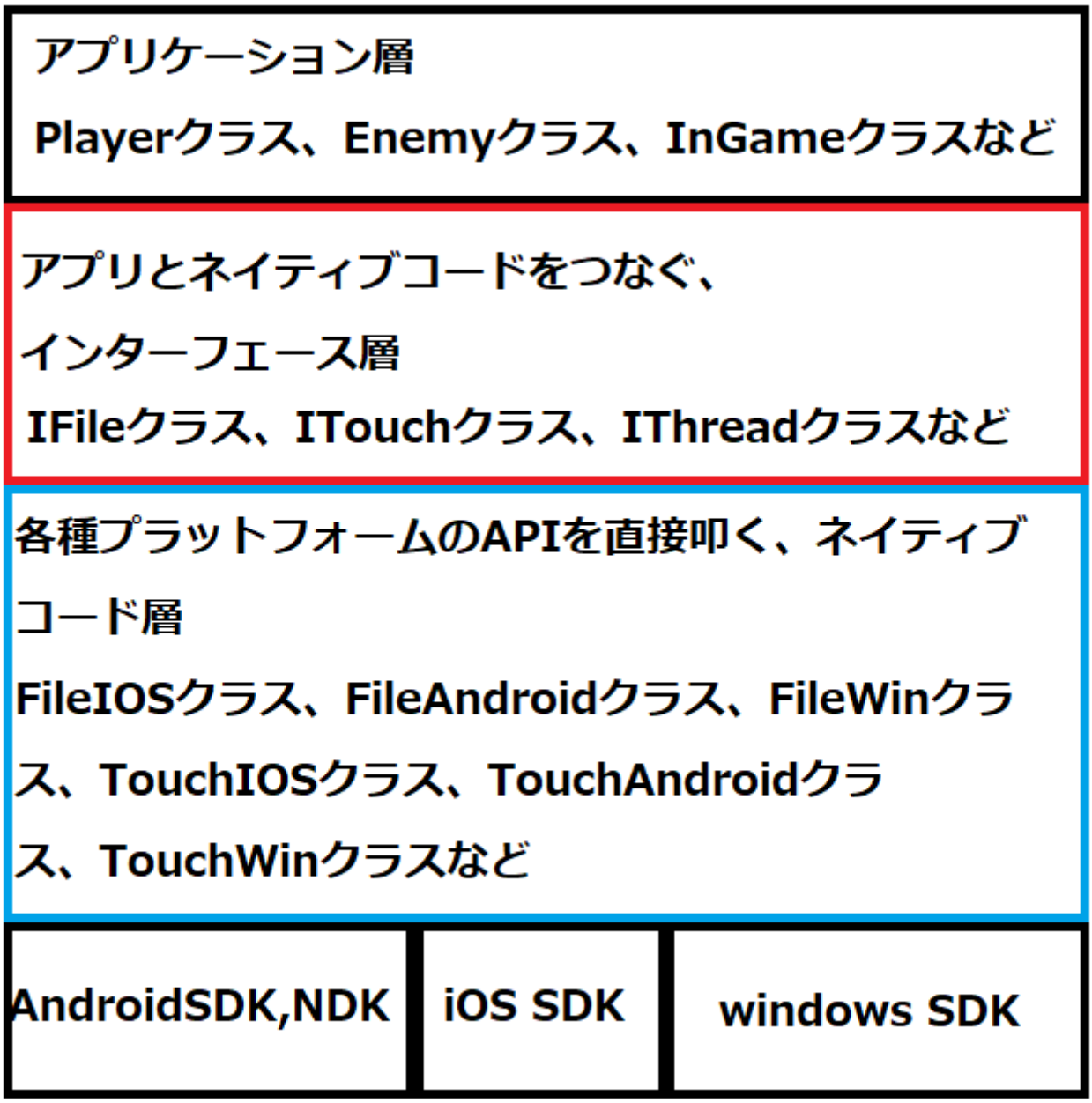
[Android]

```
char line[256];

FILE* fp = fopen("/sdcard/test.csv","r");
if(fp)
{
    fgets(line,256,fp);
    fclose(fp);
}
```

これらの違いを解決するために、私はiOS、Android、WindowsOSの専用APIを直接叩くネイティブコード層と、ゲーム側のプログラムを記述するアプリケーション層との間に、インターフェースクラス用意することにしました(図3.1)。

[図3.1]



インターフェース層を用意することで、変化が起きる部分(各種プラットフォームのAPIに依存する部分)を隠ぺい、カプセル化してしまい、交換可能にするという、まさにここまで説明してきたことを実現していたわけです。このエンジンの設計では、この後のチャプターで解説するデザインパターンがいたる箇所で使われています。アプリケーション側はインターフェースクラスを利用して、ゲームをプログラミングすることとなります。背景のプラットフォームに依存する実装は、見事にカプセル化されているため、プラットフォームが切り替わっても、ゲームのコードは大きな変更なしで動作します。