クリーンアーキテクチャとは

1. クリーンアーキテクチャについて解説
2. 具体例から更に深掘りしていく

クリーンアーキテクチャってなに？ 1

そもそもアーキテクチャとは？

コンポーネントの構成やシステム全体の構成のことを指します。

全体を俯瞰してみた設計のことをアーキテクチャと言います。

つまり、家やモノなどの形や見た目、レイアウトのことです。

クリーンアーキテクチャとは？

将来的な変更に強いアプリケーションを生み出すための基本的な考え方で、

主に関心ごとを分離し、依存関係を整理する目的で使用されます。

クリーンアーキテクチャってなに？２

使うとどんなメリットがあるの？

システムやアプリケーションの開発や保守、運用にかかるコストを最小減にすることが出来ます。

つまり開発の効率を上げ、完成した後も品質を保つことが出来るということ。

#プラスα

「設計、設計っていって全然開発進まないより、思い思いに開発した方が効率がいいんだが？」って、
特に学生さんはそう考える方が多いと思います。（私も学生ですが、、、）
が結論を言うと、設計を捨てて得た開発力は持続しません。
膨大なバグやエラーの修正、そして修正後のテストで悲惨なことになり、結局帳消しになります。

クリーンアーキテクチャってなに？３

しかし設計をこだわるとデメリットもある？

開発者に見合わない設計をしてしまうと、逆に手間が増え開発力が落ちる。

人が見やすいコードということは処理速度が出にくいため、ゲームには不向きな場合もある。

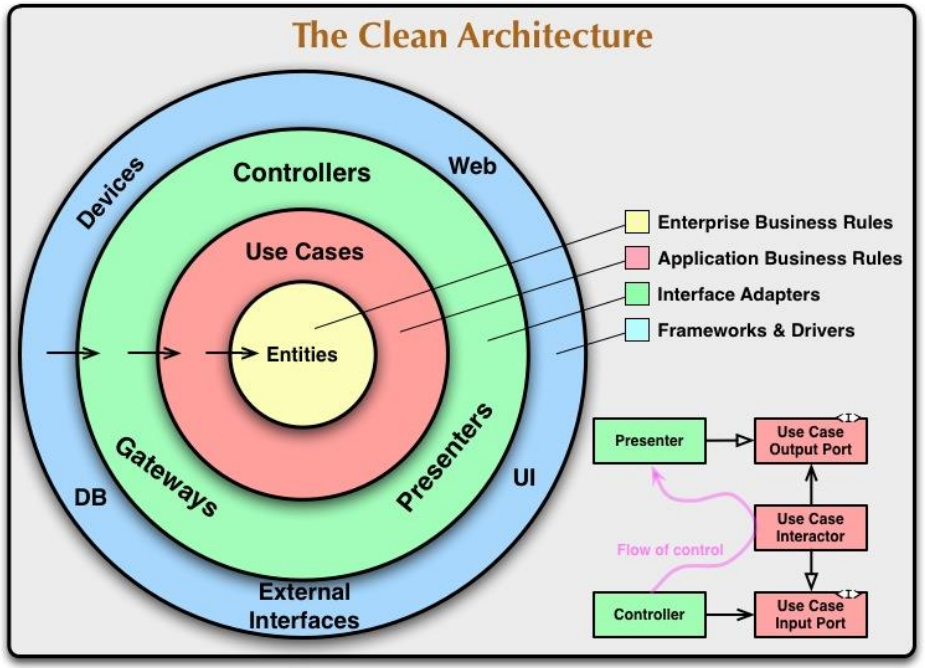
#プラスα

実際にプロジェクトに導入する際は、処理速度やチームメンバーの技術力を考えた上で、

「優先度としてどれを取るのか？」という考えをもって検討する必要があると思います。

本当は一番設計をとりたいのが本音ですが、設計ばかりではなく妥協も必要であるということ。

クリーンアーキテクチャを順番に理解する 1



クリーンアーキテクチャと言えば、
この図が有名だと思います。

ですが、
これだけ見ても理解出来ません。

なので次から分割して、
順番に説明していきたいと思います。

クリーンアーキテクチャを順番に理解する２

まずは色分け（レイヤー）の理解からです。

Enterprise Business Rules は主に、
ビジネスロジックを表すオブジェクトが所属します。

#プラスα

ビジネスロジックとは、

具体的な処理の方法や流れをもつ部分です。

プレイヤーなんかもビジネスロジックだと思います。

入力に対して「ジャンプ」といった固有処理を行いますよね？

クリーンアーキテクチャを順番に理解する３

Application Business Rules は主に、
「システムが何が出来るのか？」を表現します。

#プラスα

「何が出来るか？」だけを表現しているので、
実態を持たないインターフェイスクラスだと考えてもらって
問題ありません。
C++で言ったら、純粋仮想関数だけが定義されたクラスです。

クリーンアーキテクチャを順番に理解する４

Interface Adapters は主に、
入力、永続化、表示を担当します。

入力は Application Business Rules に伝えるための
データの加工のことです。

永続化はDB（データベース）やメモリ上への
データの保存のことです。

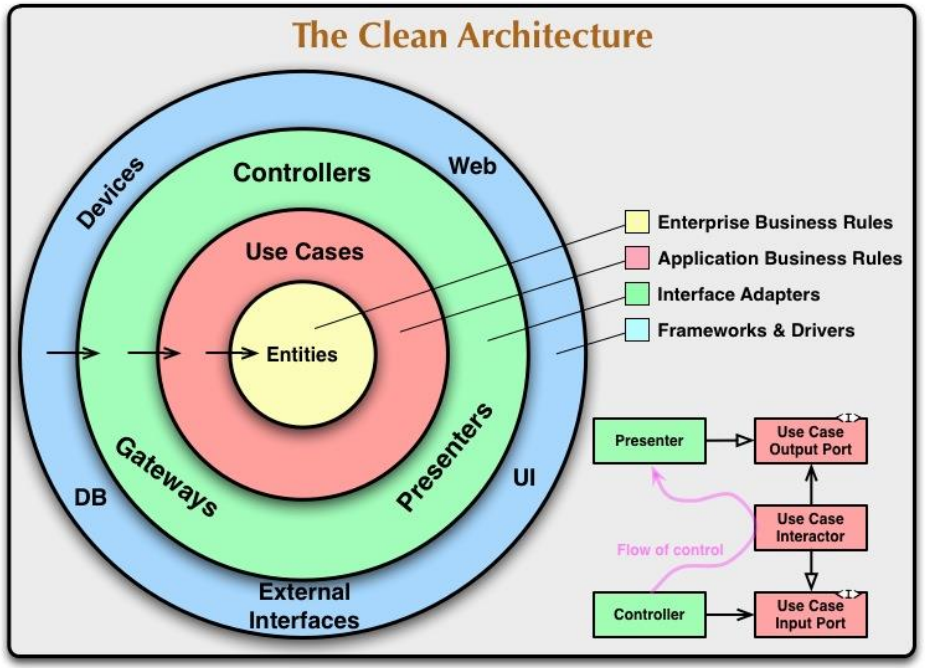
表示はそのままで結果を表示することです。

クリーンアーキテクチャを順番に理解する５

Frameworks & Drivers は主に、
Webフレームワークやデータベース操作クラス、
UIが所属します。

めちゃくちゃマニアックなコード達のことです。

クリーンアーキテクチャを順番に理解する６

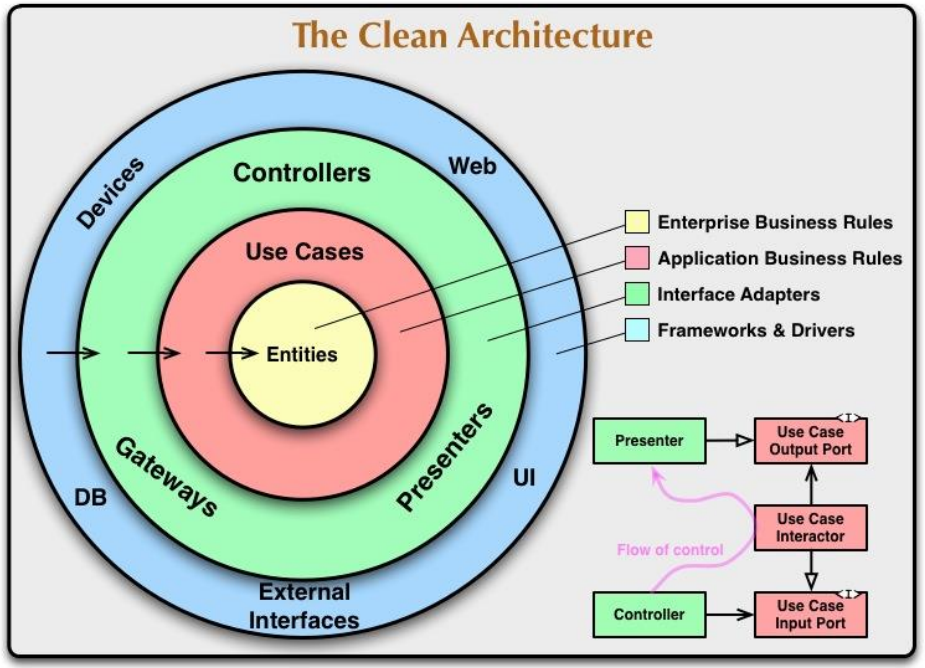


ここまでレイヤーについて説明してきましたが、
クリーンアーキテクチャでは「こうしろ」とは一つも書かれていません。

つまりレイヤーは4つに分割しろとも、
関心をここに書かれたControllers、UseCases、
Entities、などに分割しろとも言われてません。

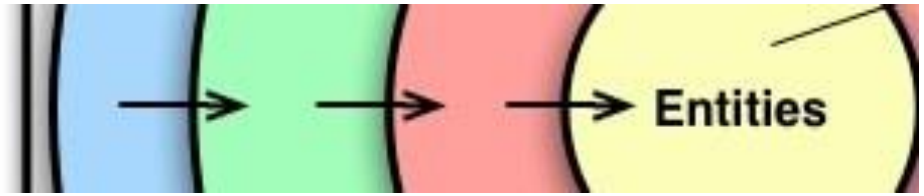
あくまで例え話の図なのです。

クリーンアーキテクチャを順番に理解する 7



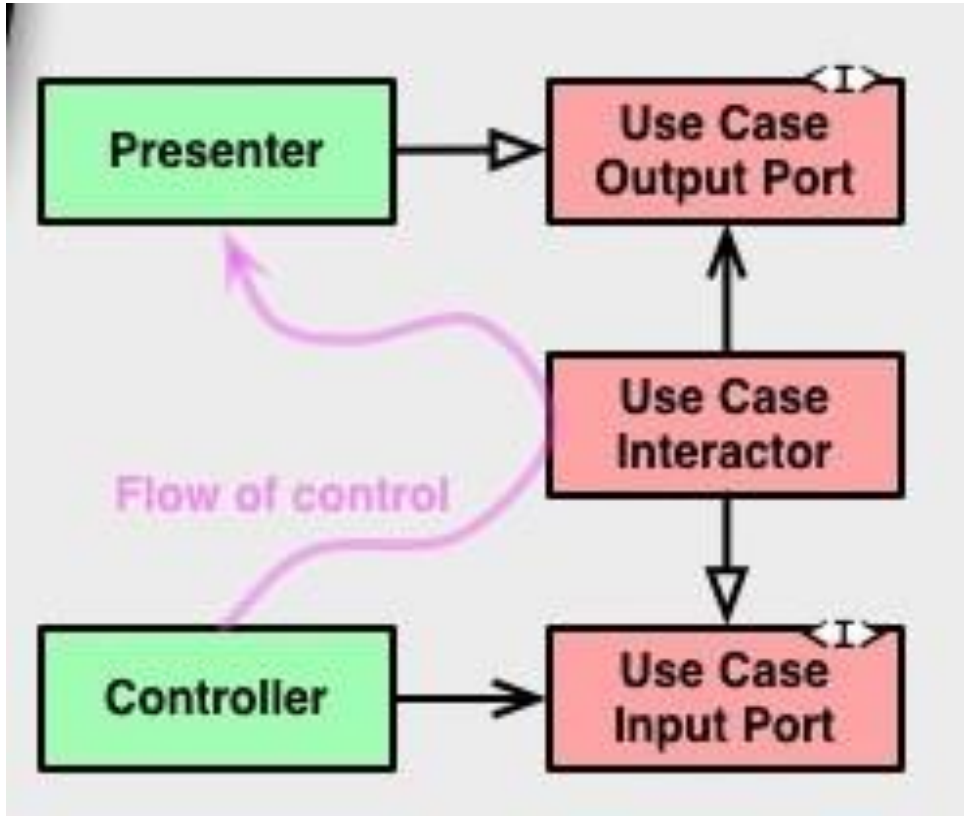
「じゃあ、この図は何ですか？」と思うと思います。
ですが、この図で大切なのは一つだけです。

クリーンアーキテクチャを順番に理解する８



それは、
依存関係は円の内側を向かなければ
いけませんよということです。

クリーンアーキテクチャを順番に理解する９



「じゃあ、右下のこれは何なのか？」というと、

これは内側から外側を呼び出す際の具体例を
示してくれています。

補足すると、
依存関係を内側に向けたまま、外側へ処理を流す
にはどうするのか？という例を示してくれています。

そして大切なのでもう一度言いますが、
「こうしろ」ということではありませんので、
処理の流れはこうじゃなくても良いです。

クリーンアーキテクチャを順番に理解する１０

クリーンアーキテクチャで大切なことを一旦まとめてみると、

有名なあの図はあくまでも「具体例」であり、
クリーンアーキテクチャで抑えるべき部分は下記の3っつだけ。

１：依存の方向はより上位レベルにのみ向けましょう。

２：処理の流れと依存関係は分離し、上手くコントロールしましょう。

３：上位レベルは相対的・再帰的なものであるということ。

クリーンアーキテクチャの具体例 1

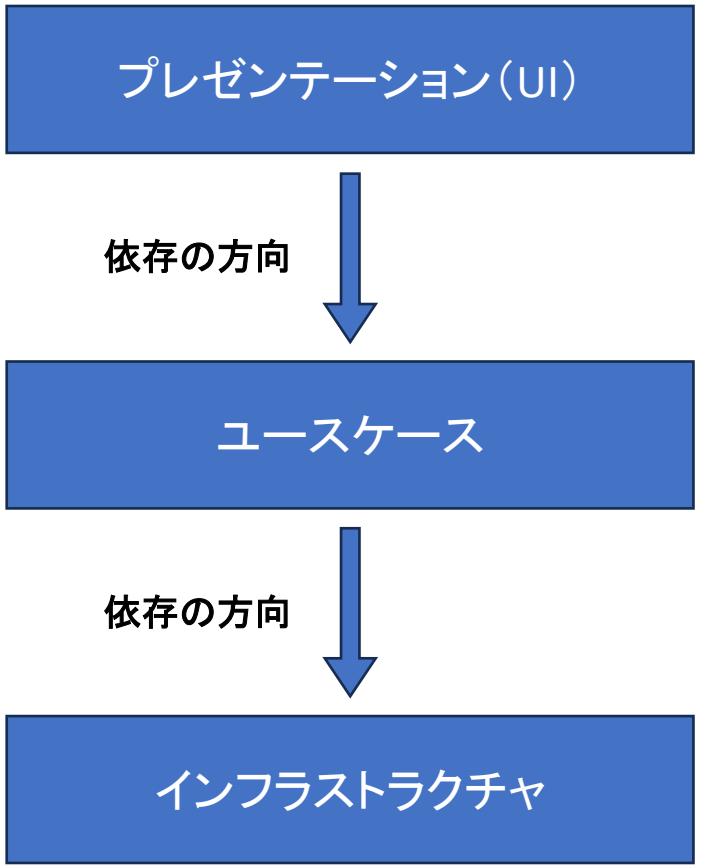
少しだけ、クリーンアーキテクチャについて理解した所で具体例から更に深掘りしていきます。

今回は、ポケモンGOのような位置情報を活用したソーシャルゲームを例に考えていきます。

このゲームの具体例を、まずは伝統的なレイヤードアーキテクチャで作成し、

その問題を解説し理解した上で、クリーンアーキテクチャで適用、解説していきます。

クリーンアーキテクチャの具体例２



プレゼンテーション（UI）では、

「ポケモンが出現する画面」を管理しています。

ユースケースでは、

「画面に表示するポケモンの一覧を取得する」

関数が用意されています。

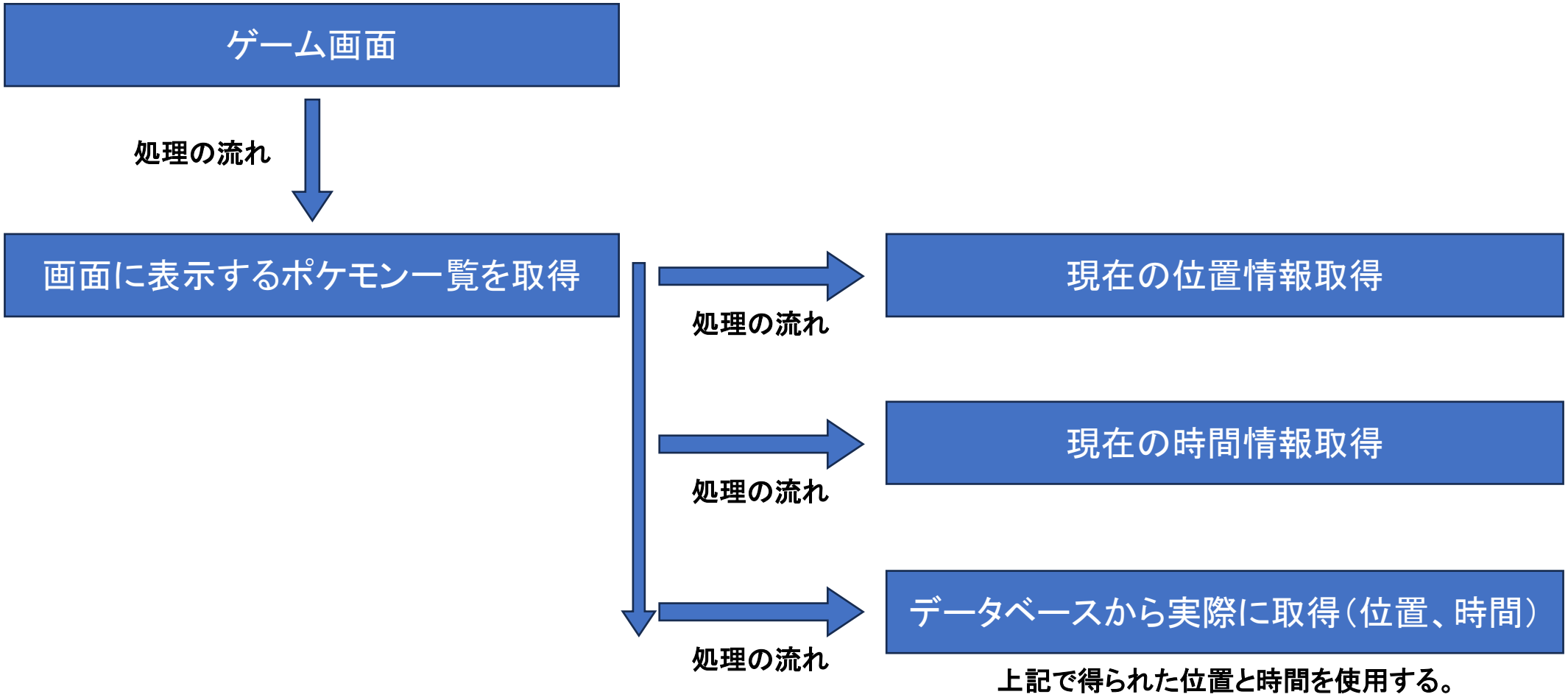
インフラストラクチャでは、

DBからポケモンの情報などを取得するApiクラスと、

位置情報を取得するLocationクラス、

時間を取得するTimeクラスがあります。

クリーンアーキテクチャの具体例３



クリーンアーキテクチャの具体例４

先ほど図だけでは少しわかりずらいと思いますが、

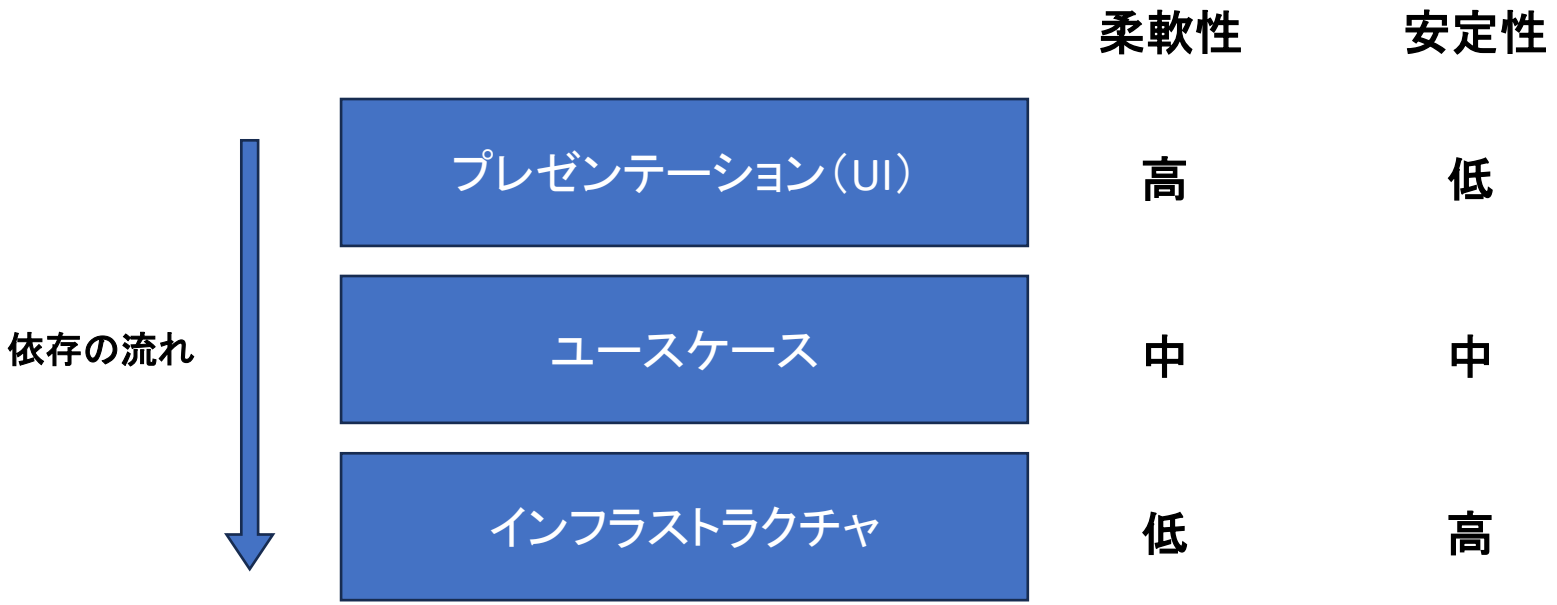
「何が問題なの？普通じゃない？」と思う人がほとんどだと思います。

しかし実際には大きな問題が隠されています。

クリーンアーキテクチャの具体例５

それは安定性と柔軟性のバランスに問題があります。

先ほどの例だと下記のようなバランスになります。



クリーンアーキテクチャの具体例６

先に「そもそも何故あのような表になるのか？」ということを解説します。

それは依存の関係について理解すれば分かると思います。

変更！！！

「ん？なんかあった？」

依存する側は変更を加える際、依存先を考慮せずに変更出来ます。

これは依存する側の変更が依存先には影響を与えないためです。

そのため、依存する側は変更が容易なため柔軟性が高くなります。

クリーンアーキテクチャの具体例 7

次は逆に依存される側が変更された場合を考えます。

「また修正かよ」

変更！！！

依存される側が変更される度、依存する側はそれに合わせて修正する必要があります。

これは依存される側の変更が依存する側へ影響を与えるためです。

そのため、依存される側は容易に変更が出来ないため相対的に安定性が高くなります。

クリーンアーキテクチャの具体例８

ではもう一度先ほどの図を見直してみます。

このゲームで一番重要である（安定しているはず）のユースケースが

インフラストラクチャに依存しているため問題があるということです。

クリーンアーキテクチャの具体例９

そしてここでクリーンアーキテクチャのまとめでもありましたが、

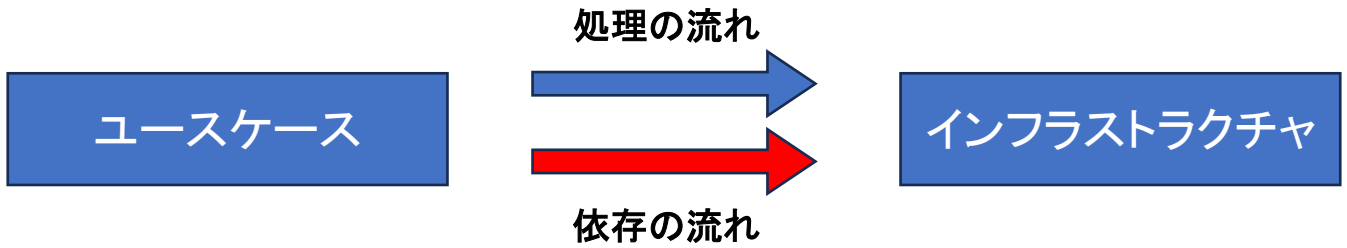
依存の方向はより上位レベルにのみ向けましょうというのが重要になります。

つまり、下記のような理想形にする必要があるということです。

クリーンアーキテクチャの具体例１０

先ほど理想形を書いていましたが、

一般的に処理の流れと依存の流れはイコールであることがほとんどです。



しかしクリーンアーキテクチャのまとめでもあったように、

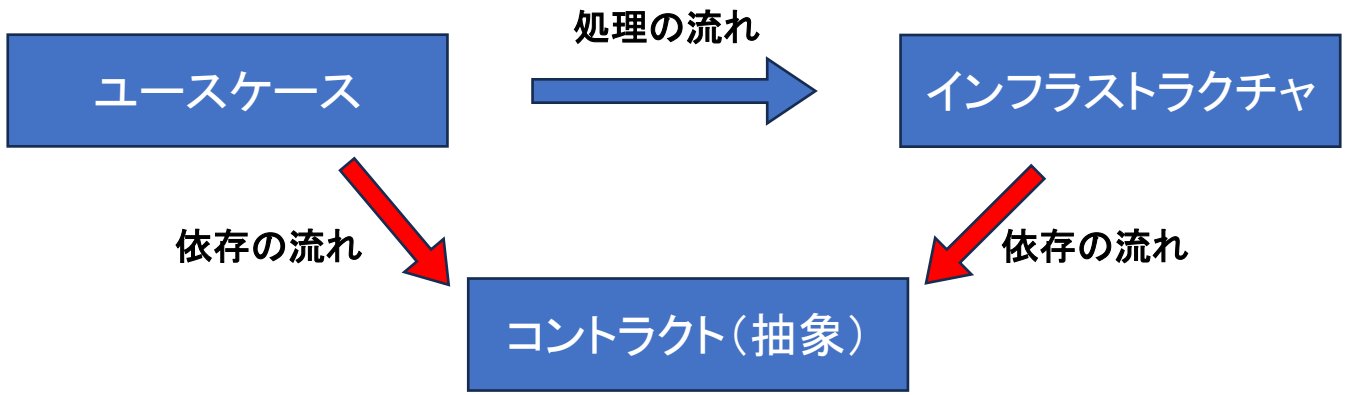
依存関係と処理の流れを分離しコントロールすることが可能です。

クリーンアーキテクチャの具体例 11

そのためには、

二つの関係の間にコントラクトと呼ばれるインターフェイスを挟む必要があります。

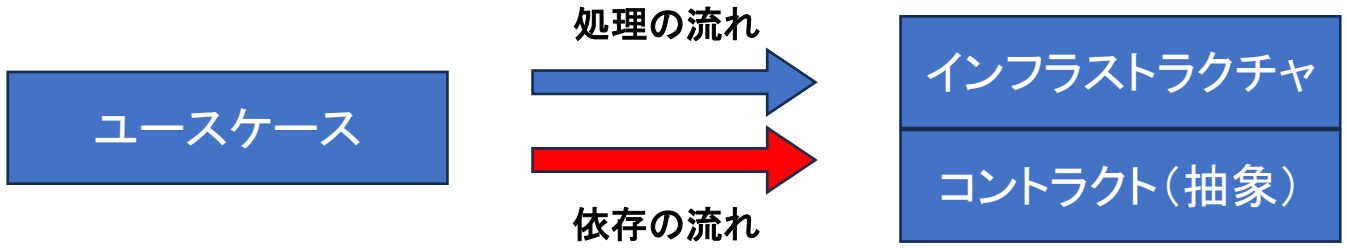
そうすることで依存と処理の流れをコントロール出来るようになります。



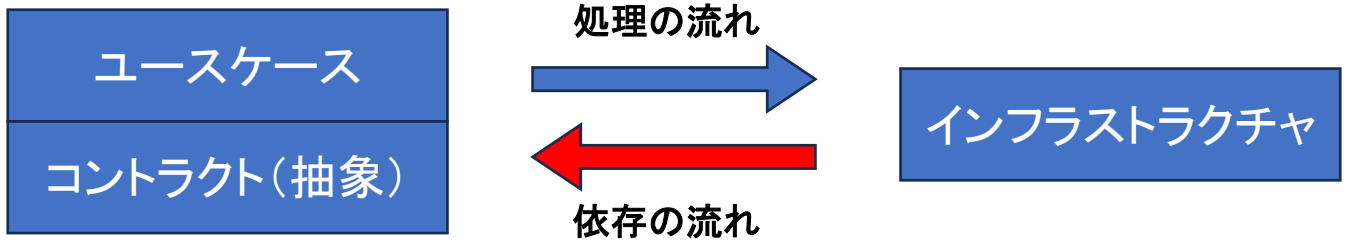
具体的にはコントラクト（抽象）を、

ユースケース、またはインフラストラクチャどちらが持つかで効果が変わります。

クリーンアーキテクチャの具体例 12



インフラストラクチャがコントラクトを持つ場合、流れは一般的な内容になります。



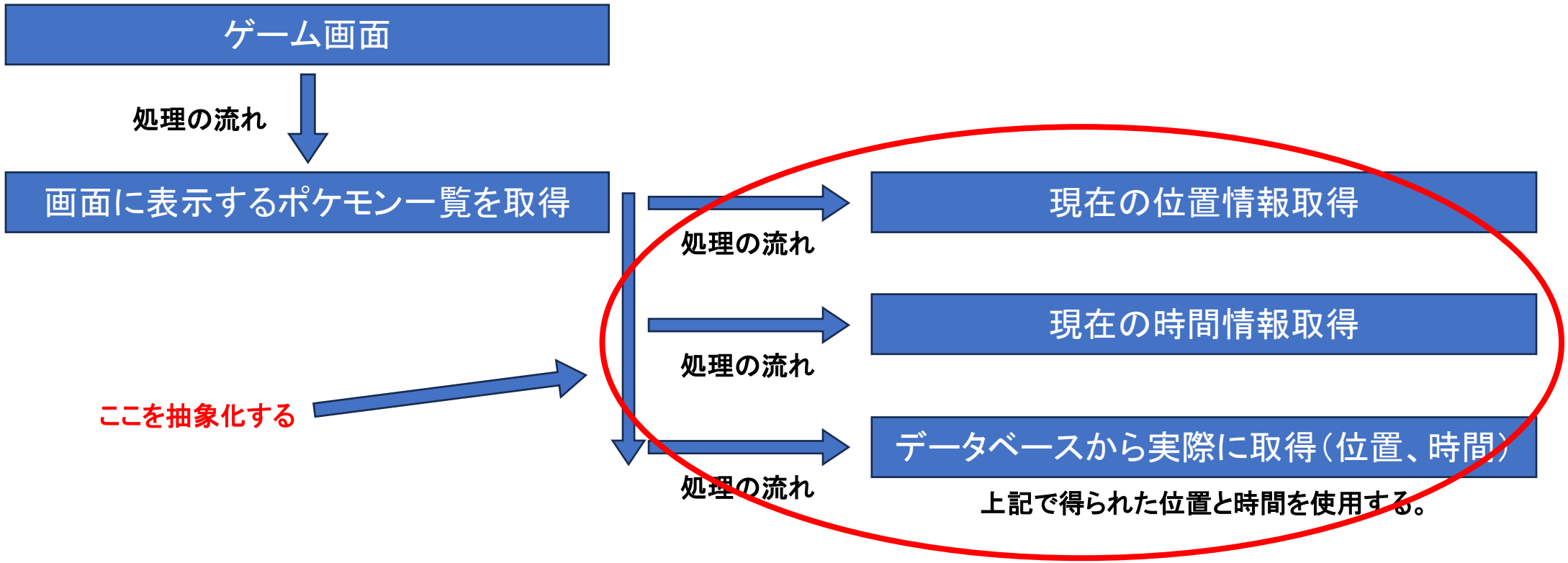
逆にユースケースがコントラクトを持つ場合、依存の流れが反転します。

これは**依存性逆転の原則**と呼ばれるものになります。

クリーンアーキテクチャの具体例 13

依存関係をコントロールする方法がわかった所で先ほど具体例を見直してみます。

そうすると先ほどの話から下記の図の赤丸部分を抽象化すれば良いことが分かると思います。



クリーンアーキテクチャの具体例 14

#仮のコード例

```cpp
class UseCase
{
public:

    std::vector<Pokemon> FindImpl(Location* location, Time* time, SearchApi* api)
    {
        // 戻り値として返すデータ配列
        std::vector<Pokemon> results;

        // 現在の位置、時間を取得
        auto nowLocation = location->GetLocation();
        auto nowTime = time->GetTime();

        // 実際に検索するが、戻り値はApi特有のデータ。
        auto searchObjects = api->Find(nowLocation, nowTime);

        // Apiの戻り値を使用する形に詰め直しをしている。
        for (auto& searchObj : searchObjects)
        {
            auto pokemon = Pokemon();
            pokemon.id   = searchObj.id;
            pokemon.name = searchObj.name;

            results.emplace_back(pokemon);
        }

        return results;
    }
};
```

実際には下記のようなコードになり、

抽象クラスへの依存のみに出来ると思います。

もちろん、

ユースケース側で実装される

LocationやTime、SearchApiクラスは、

純粋仮想関数のみを保持しているクラスです。

クリーンアーキテクチャの具体例 15

#仮のコード例

```cpp
class UseCase
{
public:

    std::vector<Pokemon> FindImpl(Location* location, Time* time, SearchApi* api)
    {
        // 戻り値として返すデータ配列
        std::vector<Pokemon> results;

        // 現在の位置、時間を取得
        auto nowLocation = location->GetLocation();
        auto nowTime = time->GetTime();

        // 実際に検索するが、戻り値はApi特有のデータ。
        auto searchObjects = api->Find(nowLocation, nowTime);

        // Apiの戻り値を使用する形に詰め直しをしている。
        for (auto& searchObj : searchObjects)
        {
            auto pokemon = Pokemon();
            pokemon.id   = searchObj.id;
            pokemon.name = searchObj.name;

            results.emplace_back(pokemon);
        }

        return results;
    }
};
```

↑まだ問題がある

「これで完成」と言いたいですが、

実際にはもう一つ問題が残っています。

それは、

Apiクラスから実際に検索をかけた後、

返される戻り値がApi専用のデータ型であるため、

ユースケースがApiに依存しているのです。

クリーンアーキテクチャの具体例 16

#仮のコード例

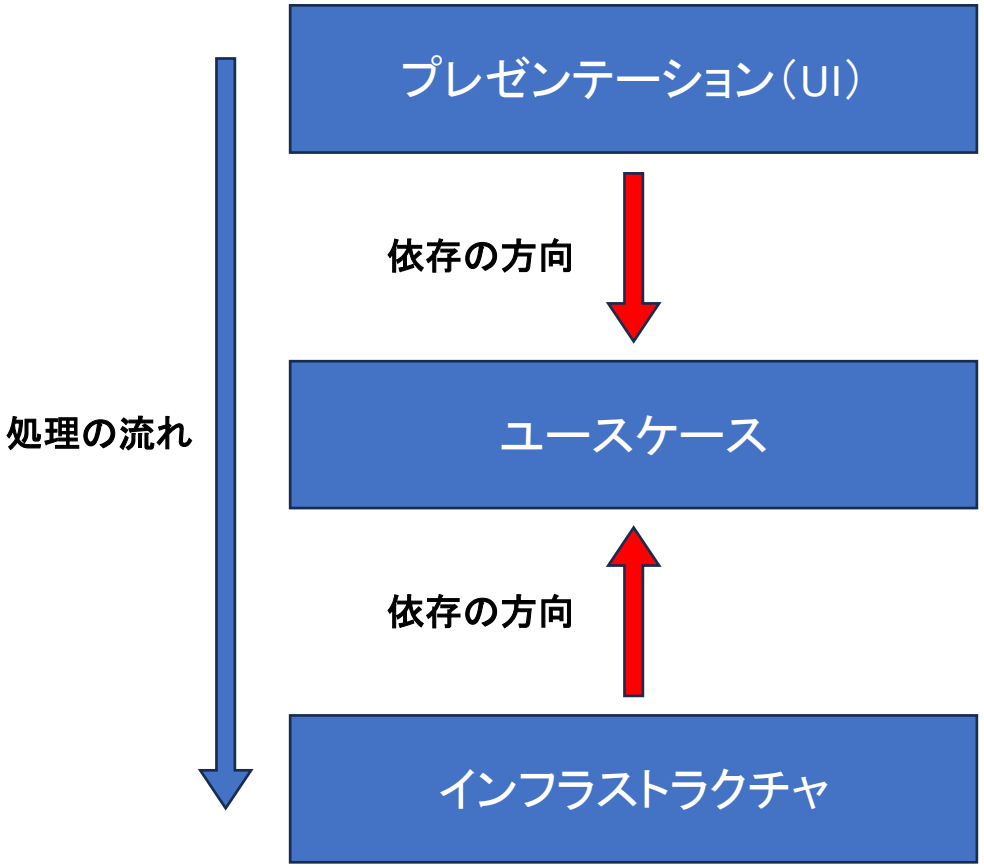
```
34  class UseCase
35  {
36  public:
37
38
39      std::vector<Pokemon> FindImpl(Location* location, Time* time, SearchApi* api)
40      {
41          // 現在の位置、時間を取得
42          auto nowLocation = location->GetLocation();
43          auto nowTime = time->GetTime();
44
45          // ApiがUseCaseの戻り値に依存している形にした場合。
46          return api->Find(nowLocation, nowTime);
47      }
48  };
```

これは簡単に治せます。

抽象クラスで定義されている関数の、
戻り値自体をユースケース特有のものにすれば
良いのです。

つまり、
Apiの派生クラスの実装部分で先ほどの
詰め直しを実装するということです。

クリーンアーキテクチャの具体例 17



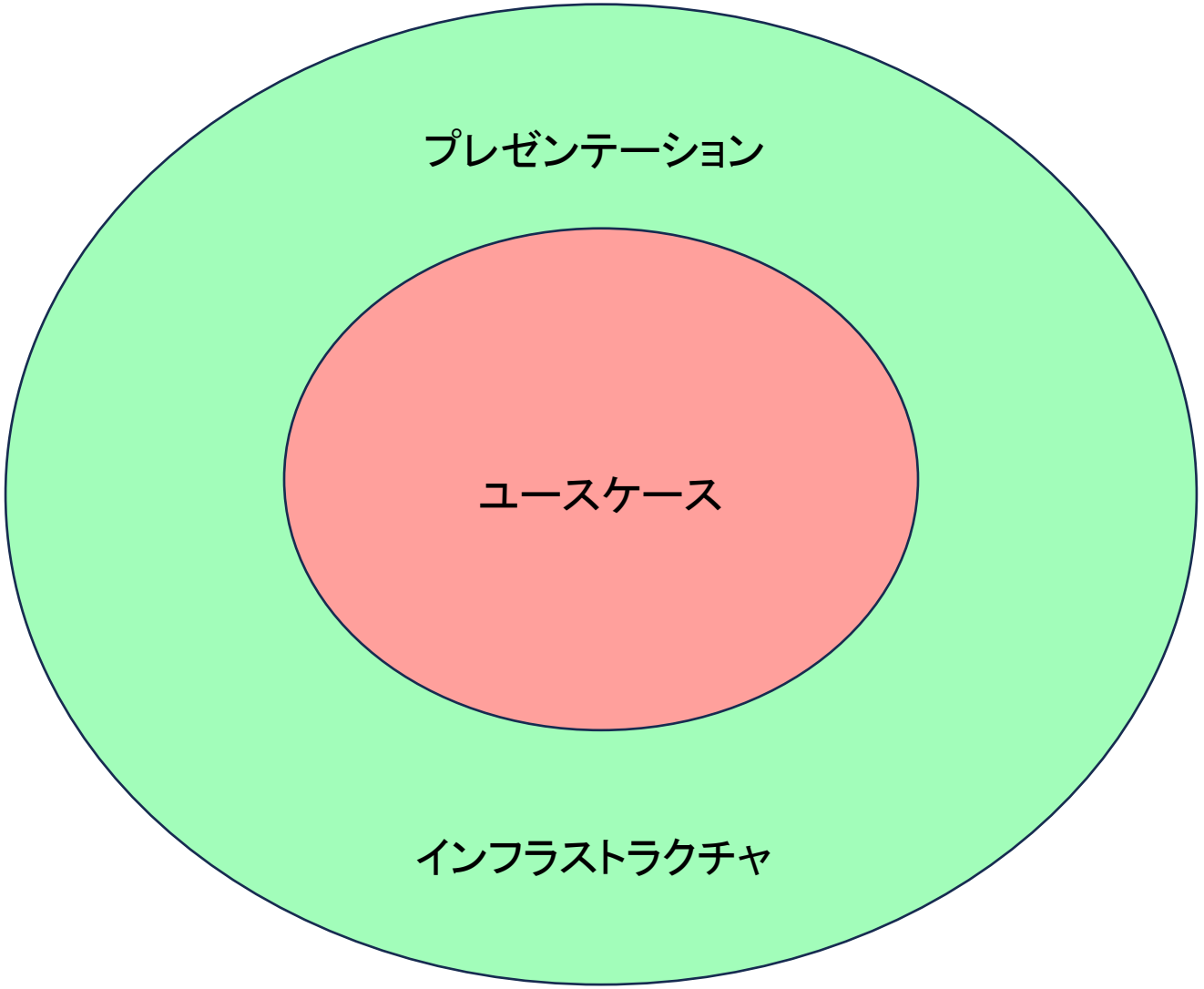
先ほどまでの話で、最初の具体例の図が

右のような形に変わっていると思います。

そして、

この図を円状に配置すると次のようになると思います。

クリーンアーキテクチャの具体例 18



私達が知っている図になりました。

今回の話で、

依存は上位レベルへのみ向ける。

依存と処理の流れは制御する。

ということが分かったのではないでしょうか？

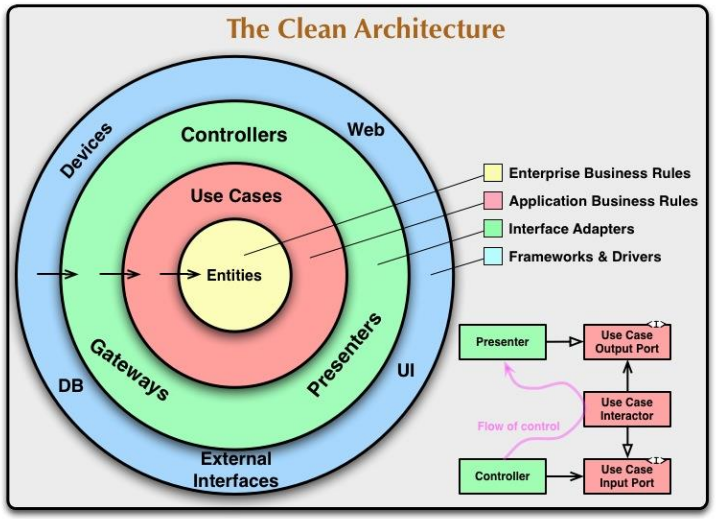
クリーンアーキテクチャの具体例 １９

今回は簡単な例でしたが例え今回のソーシャルゲームを、

Unityで開発しても、UEで開発しても、はたまたスクラッチで開発しても先ほどの図になると思います。

つまり上位レベルは相対的・再帰的なものであるということが言えると思います。

クリーンアーキテクチャまとめ



今回はクリーンアーキテクチャについて、
説明をしてきました。

そしてここまできて元も子もないことを書きますが、

この左の図はあくまで「たとえ」であり、
イイ感じに保守性が高いコードになるんじゃないか、
という指標に過ぎません。

なのでこの図に振り回されすぎないことが大切です。
※間違っても設計をしなくて良いと言ってるわけではない

今回クリーンアーキテクチャが理解できなかった人は、
先に設計原則等の知識を蓄えることをおすすめします。
※「クリーンアーキテクチャ」という本も21章までは全て設計原則の話です