1803b 版

アジャイル概要（アーキテクチャーとテスト）

2020 年 5 月 17 日～5 月 22 日

（補助資料）

Microsoft CERTIFIED Solutions Developer
Web Applications
Windows Store Apps Using C#

Microsoft CERTIFIED Professional Developer
Windows Azure Developer
Windows Phone Developer
Windows® Developer 4
Enterprise Application Developer 3.5

2020 年 5 月 8 日

ニュートラル株式会社
林 広宣
http://www.neut.co.jp/
hayashi.hironori@neut.co.jp

（このページが、表紙の裏になるように印刷をお願いします。）

ご意見・ご質問は下記まで

ニュートラル株式会社　林広宣

目　　次

（空白ページ）

第1章. アプリケーションアーキテクチャーについて（参考）

1.1. アプリケーションアーキテクチャーに関する補足事項

1.1.1.レイヤーアーキテクチャーの欠点

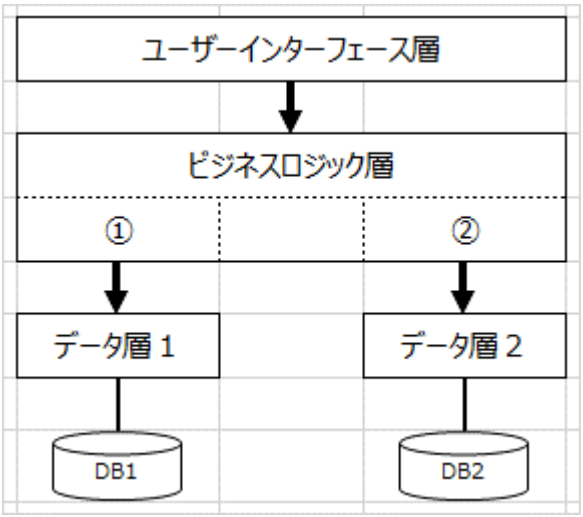
レイヤーアーキテクチャーは「処理の流れる方向とレイヤー（またはオブジェクト）の依存関係の方向を同一」と考えるアーキテクチャーです。

「処理の流れ」とは、「ユーザーインターフェース層がビジネスロジック層を呼び出し、ビジネスロジック層がデータリンク層を呼び出す」という流れのことです。

「レイヤーの依存関係」とは、「ユーザーインターフェース層はビジネスロジック層に依存し、ビジネスロジック層はデータリンク層に依存する」ということです。

ここでいう「依存関係」とは、ソースコード上では、クラス名や名前空間がどちらの側に登場するかということです。
ユーザーインターフェース層のコードににビジネスロジック層のクラス名や名前空間が登場し、ビジネスロジック層のコードにデータリンク層のクラス名やコードが登場するということです。

従ってレイヤーアーキテクチャーでは、下図の太い矢印は「処理の流れ」を示すものでもあり「依存関係」を示すものでもあります。



ここで問題となるのは、「DB1 を DB2 に置き換えたい」とか「本番のデータベース（DB1）設計が完了していないのでテスト用のオブジェクト（DB2 いわゆるモック）でしばらく開発しなければならない」というような場合です。
そのような場合は、ビジネスロジック層のコードを①から②へ書き替えなければならないか、①と②の両方を記述しなければならない、ということになります。

ビジネスロジック自体に変更はないのにもかかわらず、ビジネスロジック層のコードを書き替えなければならないというのは筋が通っていない話です。
また、ビジネスロジック層にテスト用（モック用）のコードが存在するのは不自然です。

これは、ひとえにビジネスロジック層が外部環境（データベースなど）に依存するような設計になっているのが原因です。

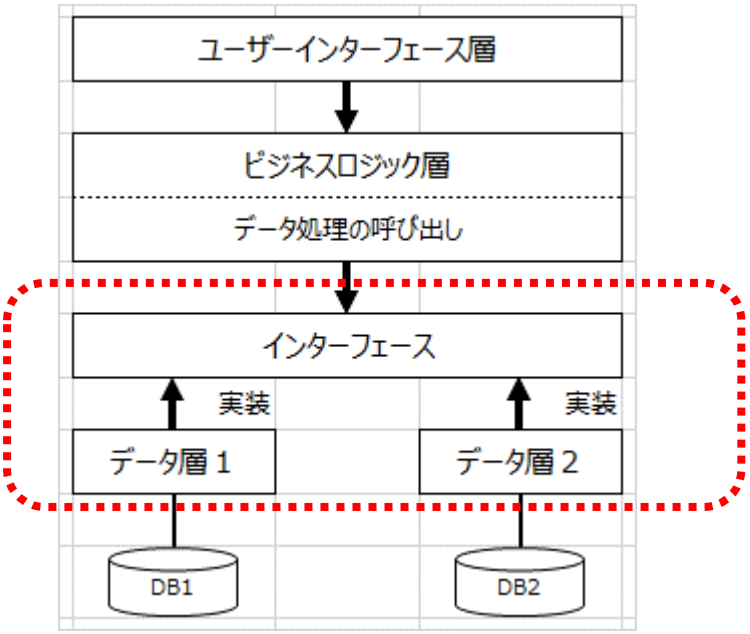
1.1.2.クリーンアーキテクチャーの考え方

クリーンアーキテクチャーなどのアーキテクチャーでは、「内部は外部に依存すべきではない」「抽象が詳細に依存すべきではない」「変化の緩やかなものが変化の激しいものに依存すべきではない」等と考えます。

今回の場合「内部」「抽象」「変化の緩やかなもの」はビジネスロジック層に相当し、「外部」「詳細」「変化の激しいもの」はユーザーインターフェース層とデータリンク層に相当します。

これを実現するためには、処理の流れの方向は変えずに、依存関係の方向を逆方向に変える必要があります。
これを「依存性の逆転 Dependency Inversion」と言います。

依存性を逆転させるためには、逆転させたい部分にインターフェースを介在させるようにします。
そして「外部」「詳細」「変化の激しいもの」等に属するオブジェクトは、そのインターフェースを実装するようにします。
一方、「内部」「抽象」「変化の緩やかなもの」に属する側は、インターフェースを使用するようにします。(下図)



1.1.3.依存性の逆転（Dependency Inversion）と依存性の注入（Dependency Injection）

このように依存性を逆転させると、ビジネスロジック側のソースコードはインターフェースを呼び出すことになりますが、インタフェース自体はインスタンスではないので、本当に必要なオブジェクトをインスタンスとして生成しておき、それをインタフェースとして参照できるようにする処理が、別途、必要です。

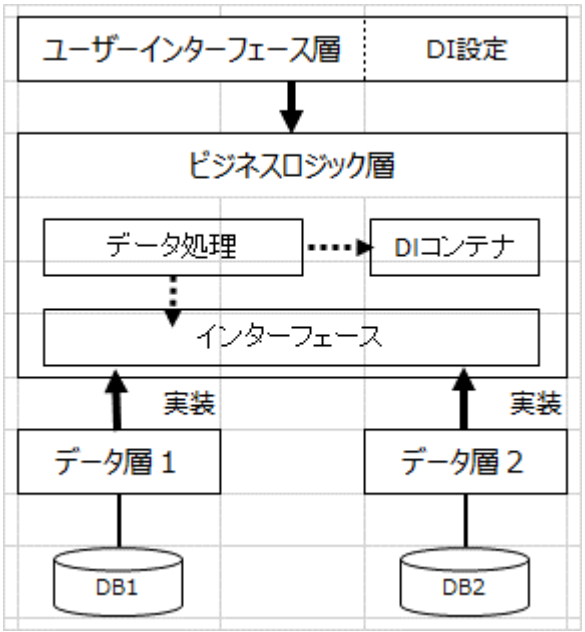
これを「依存性の注入（Dependency Injection)」と呼んでいます。
「依存性の注入」とは、インタフェースに対し、実際の実装オブジェクト（インスタンス）を参照できるようにすることです。

今回の例では、アプリケーション実行時に「データ層１」のコンポーネントを使用するのか「データ層２」のコンポーネントを使用するのかを事前に決定しておき、それをインタフェースから参照できるようにしておくことになります。

具体的には、ビジネスロジック層（またはビジネスロジックと同程度のレベルの層）に DI コンテナ（Dependency Injection Container）というオブジェクトを定義し、アプリケーション開始時に DI コンテナに対し、今回使用するインスタンス（インタフェースを実装したオブジェクト）を代入しておきます。

ビジネスロジック層のデータ処理ロジック部分では、DI コンテナから上記のオブジェクトを取得して入出力処理を行います。

この時、ビジネスロジック層ではインタフェース呼び出しだけでコードが記述できるので、データリンク層のオブジェクトに依存することはありません。



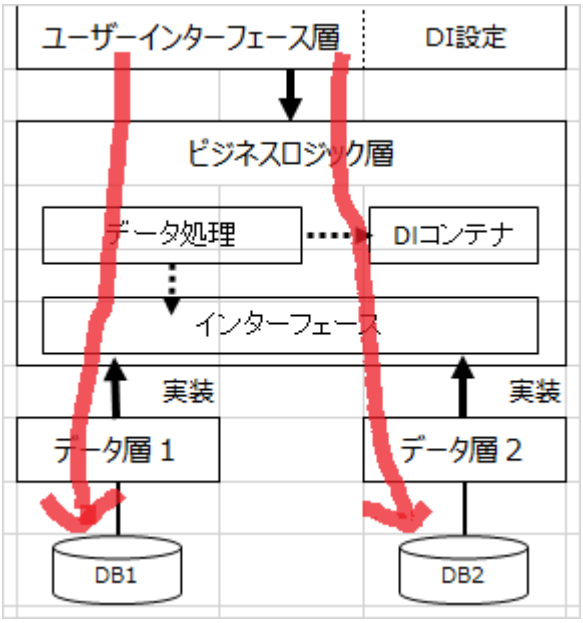
1.1.4.処理の流れ

DI（Dependency Inversion または Dependency Injection）を使用すると、依存性の矢印は逆転しますが、処理の流れは逆転しません。

つまり、ビジネスロジック層のオブジェクトがデータリンク層のオブジェクトのメソッドを呼び出すことになります。（下図のフリーハンドの矢印を参照）

処理の流れはそのままで、依存関係だけを逆転させていることが、クリーンアーキテクチャーの大きな特徴になります。

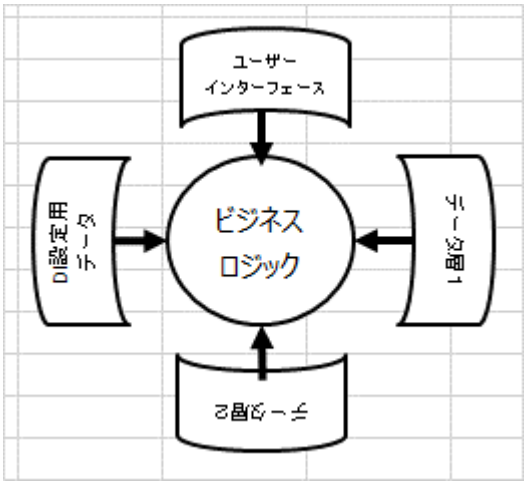
もちろん、外部から内部へ、詳細から抽象へ、変化の大きなものから変化の少ないものへ依存するように逆転させます。



1.1.5.まとめ

クリーンアーキテクチャーやオニオンアーキテクチャーなどのアーキテクチャーを示す図は、レイヤーアーキテクチャーのような「上下に並べる図」だと表現しにくいので、下図のように「同心円」となるような図で示されます。

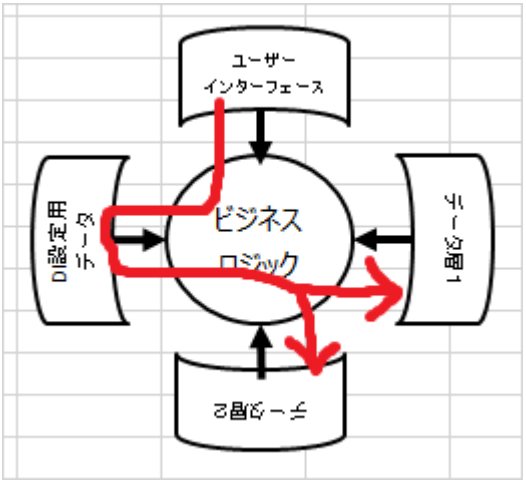
この場合、必ず「外側から内側」へ矢印が伸びるような依存関係となります。



一方、同心円は依存関係だけを表しており、処理の流れを示しているわけではないという点に注意する必要があります。

今回の場合、処理の流れは、下図のフリーハンドで書いた矢印になります。

処理の流れはレイヤー構造の図で示したほうがわかりやすいかもしれません。



第2章. テスト駆動開発（TDD）（参考）

2.1. テスト駆動開発（TDD）とは

2.1.1.概要

テスト駆動開発（Test Driven Development, TDD）は、開発手法、設計手法であり、テスト手法ではない。

よって、テストケースが完全であるか、あるいは漏れがあるかどうかはについては、それほど気にしない。（設計が正しいかどうかについては、気にする）

言い換えると、アプリケーションの品質という観点から見た場合、テスト駆動開発だけでなく、テスト手法（ドメイン分析テストなど）の考え方にも基づいて、テストケースに漏れがないかチェックする必要がある。

テスト駆動開発では、「インテグレーションテスト（受入テスト、結合テスト等と訳される）」と「ユニットテスト」に分けて記述する。
まず、インテグレーションテストを記述し、それを正しく成功させるために（正しい設計となるように）細かなユニットテストを記述する。

インテグレーションテストは、できる限り「エンドツーエンド（端から端まで）」なテストとする。
言い換えると、個々のメソッドレベルのテストではなく、「１つの機能全体」のテスト、あるいは「データの入力から、それに対する結果の出力まで」をカバーできるようなテストを記述する。
一方、ユニットテストは、メソッドレベルのテストである。

ポイント

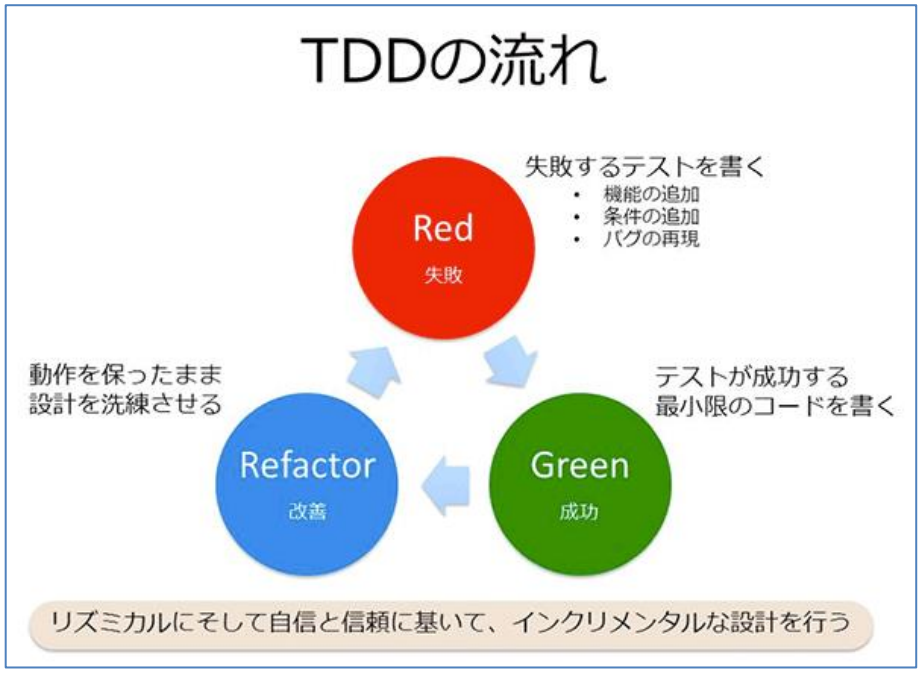
1. TDD は設計手法である
2. 受入テストと単体テストに分けて記述する

2.2. テスト駆動開発の流れ

2.2.1.説明

テスト駆動開発を用いた設計では、次の順序でテストと設計を繰り返す

1. 失敗するテストを記述する
2. テストが成功する最低限のコードを記述する
3. リファクタリングを行う



出典：https://thinkit.co.jp/story/2014/07/30/5097

2.3. 受入テストと単体テスト

2.3.1.説明

最近のテスト駆動開発では、「インテグレーションテスト（受入テスト、結合テスト等と訳される）」と「ユニットテスト（単体テスト）」に分けて記述するのがよいとされている。

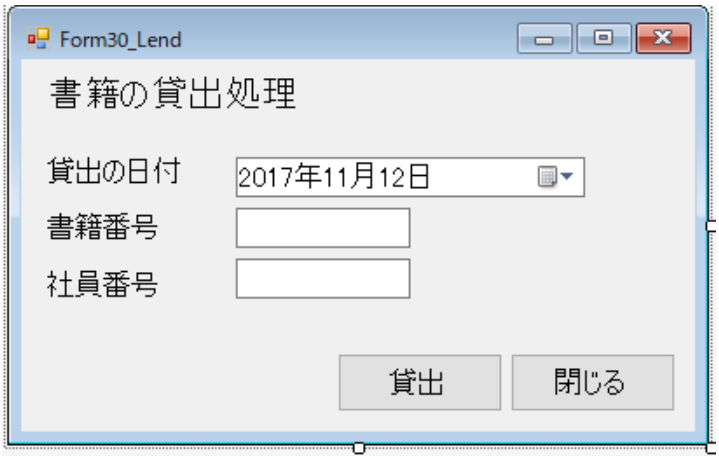
ユニットテストとは、個々のメソッドを対象にしたテストで、１つのメソッドに対し、カバレッジなどを考慮して、正常系と異常系のテストをいくつか記述したものである。

一方、インテグレーションテストとは、個々のメソッドを対象にしたテストではなく、１つの機能全体をテストするものである。
インテグレーションテストは、なるべく、「エンドツーエンド（端から端まで）」なテストにすることが推奨される。

エンドツーエンドとは「データの入力から、それに対する結果の出力まで」をカバーできるようなテストとなる。

例えば、下図で［貸出］ボタンがあるが、このボタンの処理に対するテストが、インテグレーションテストに該当する。

この［貸出］機能は、１つのメソッド呼び出して実現されているかもしれないが、そのメソッドがさらに「データベースの参照」「データの整合性チェック」「新しいデータの出力」などの数多くのメソッドを呼び出している。



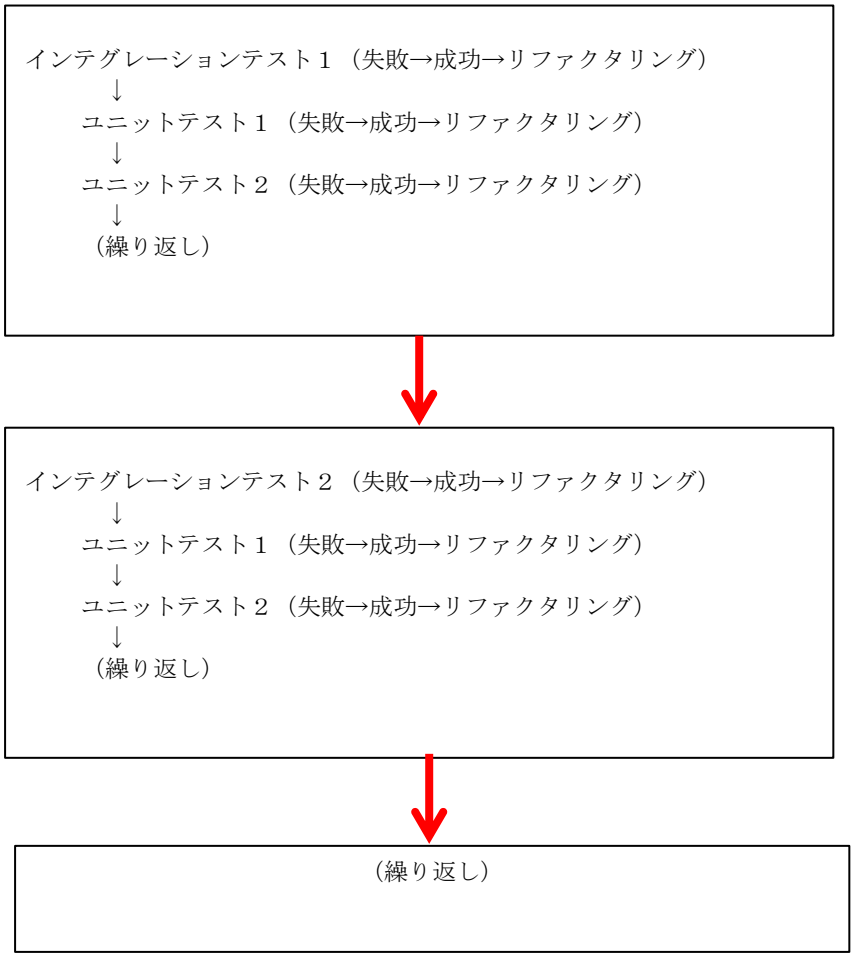
ユニットテストは、個々のメソッドが正しく動作することを保証してくれるが、システム全体として正しく動作するかどうかは不確定なテストとなる。

一方、インテグレーションテストは、システム全体が正しく動作することを保証するためのテストである。

2.3.2.手順

インテグレーションテストもユニットテストも、見かけ上は同じユニットテストとして記述されるが、異なった意味合いのものであることを念頭に記述することが重要である。

まず、インテグレーションテストを記述し、それを正しく成功させるために（正しい設計となるように）細かなユニットテストを記述する。



（空白ページ）