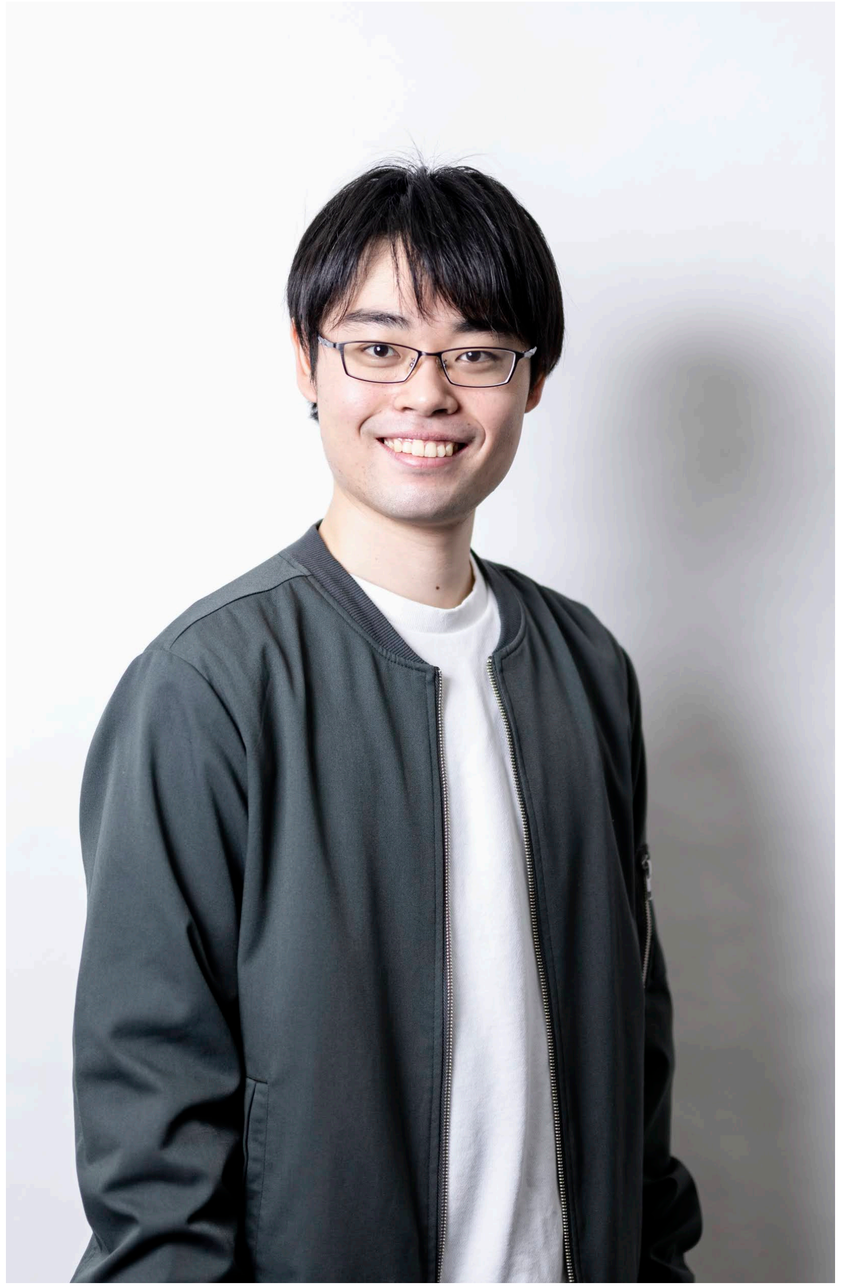
良いコードとは何か

2021年度 株式会社サイバーエージェント エンジニア 新卒研修

株式会社CyberZ 森 篤史



森 篤史

2019年度 新卒入社

Androidアプリエンジニア

未踏スーパークリエータ

@at_sushi_at

Mori-Atsushi

CYBER OPENREC.tv

リアーキテクチャプロジェクト

新卒1年目からAndroidアプリの**アーキテクチャ移行**を主導

- Java → Kotlin
- MVC → MVVM + Clean Architecture

機能開発と並行した段階的リアーキテクチャ

現在も進行中

CA BASE CAMP 2020 セッション

プロジェクトを進行する上で感じたこと

良いコード何かを知らなければ、

たとえ書き直したとしても

より**良くならない**

🚀 今回のゴール

良いコードとは何かについて深く考える

Contents

CyberAgent

1

品質とスピードは
トレードオフか

CyberAgent

今は急いでるので、

品質より速度優先で！

疑問①

品質とスピードは
トレードオフである？

その前に、品質とは？

- 外部品質
  - ユーザに見える
  - 仕様通り動く、バグがない、高速に動作する
- 内部品質
  - ユーザからは見えない
  - 保守性や柔軟性、可読性、一貫性

その前に、品質とは？

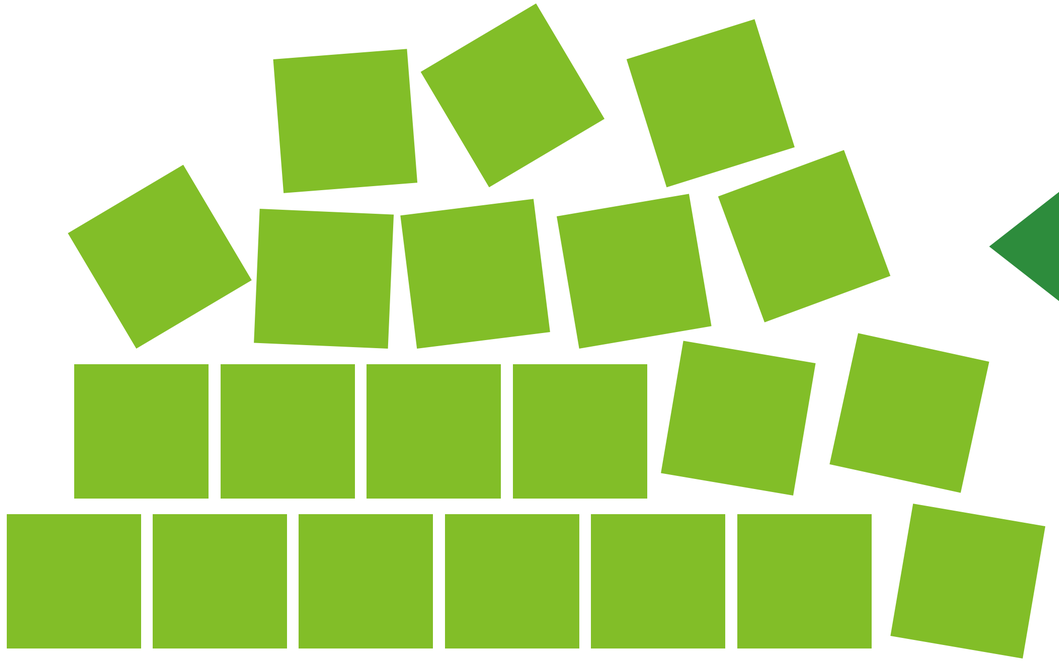
- **外部品質**
  - ユーザに見える
  - 仕様通り動く、バグがない、高速に動作する
- **内部品質**
  - ユーザからは見えない
  - 保守性や柔軟性、可読性、一貫性

往々にして
犠牲になるのはこっち

内部品質を一時的に下げると起きること

内部品質を犠牲に

内部品質を一時的に下げると起きること



その後
崩壊し続ける

割れ窓理論

窓ガラスを割れたままにしておくと、
その建物は十分に管理されていないと思われ、
ごみが捨てられ、やがて地域の環境が悪化し、
凶悪な犯罪が多発するようになる、という犯罪理論

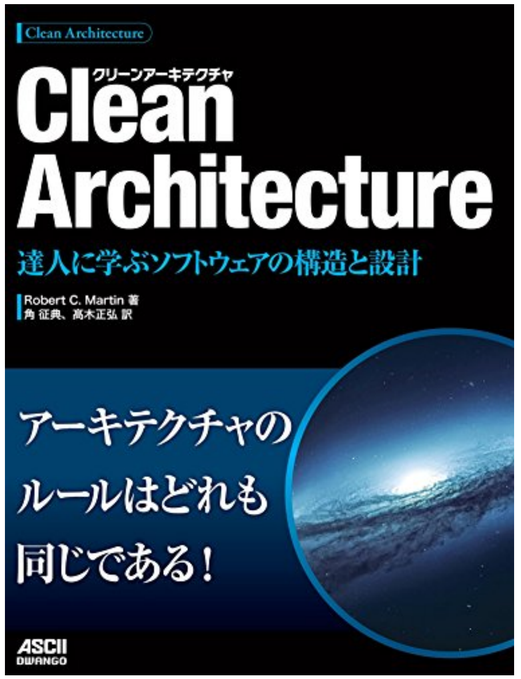
割れ窓理論とは - コトバンク https://kotobank.jp/word/%E5%89%B2%E3%82%8C%E7%AA%93%E7%90%86%E8%AB%96-665951

思考：後でキレイにしよう

内部品質を犠牲に

その後では来ない

- 市場からのプレッシャーは止まらない
- 競合他社に追い抜かれないためには、これからも走り続けるしかない
- 崩壊が始まり、生産性はゼロに近づいていく



Robert C.Martin (2018), Clean Architecture 達人に学ぶソフトウェアの構造と設計, KADOKAWA

品質とスピードはトレードオフである？

- ごく短期的にはスピードは上がるかもしれない
- 中期的には逆効果である
- 長期的には致命的になる

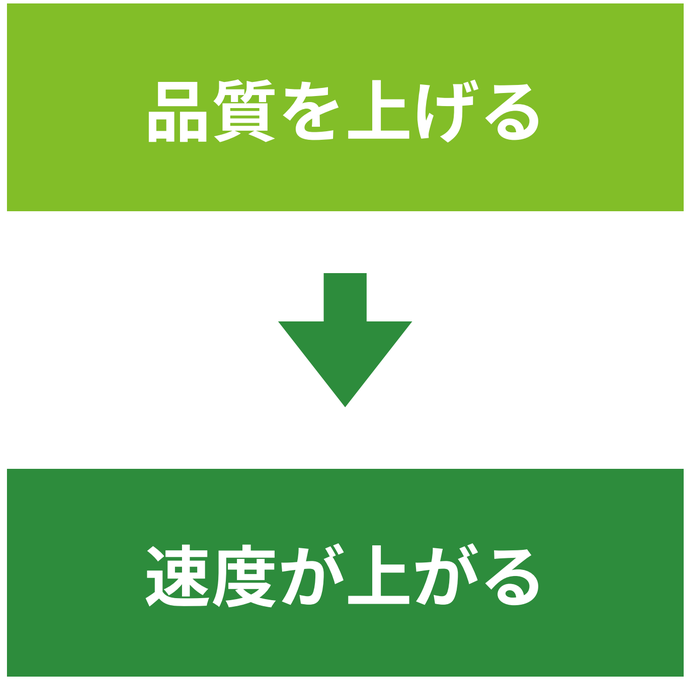
どれくらい短期なのか？

- 数週間（1ヶ月以内）と言う人もいる
- 数日で違いが出ると言う人もいる

- 個人的な 経験則から言えば、1週間で問題が発生する
- 再びそのコードもしくは関連するコードに触れた際、直ちに問題になる

コードの品質を高く保つからこそ速い

- 品質を上げることで
  - コードの変更速度が上がる
  - 手戻りが減る
  - 学びのループが早くなる
  - 市場での競争力が上がる



疑問②

時間をかければ
良いコードが書けるのか？

不要に時間をかけても品質は上がらない

品質とスピードはトレードオフではない

=

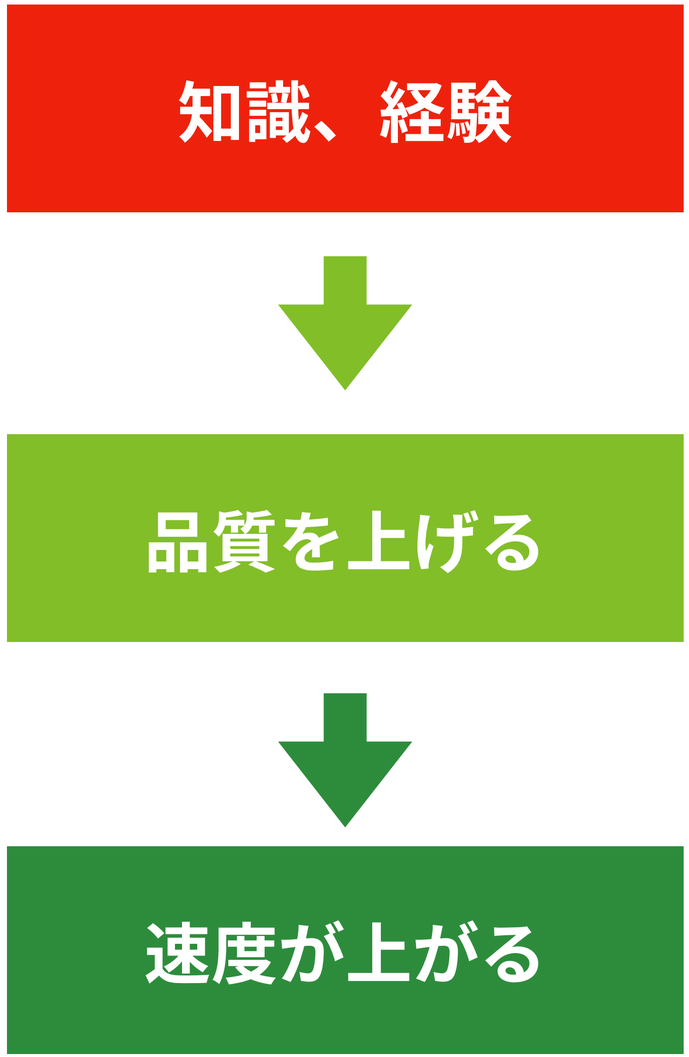
品質を下げても速度は上がらない

=

時間をかけても品質は上がらない

品質を上げるために必要なのは知識、経験

- 知識、経験が品質を向上させる
  - 習得するには時間がかかる
  - 学び続ける必要がある



2

技術的負債の発生と解消

CyberAgent

疑問③

品質を上げるために必要な
知識・経験とは？

品質について考えるために

品質が下がっている状態

=

技術的負債が溜まっている状態

技術的負債の発生理由

	意図的な負債 ⚡	変化による負債 🏃	学びによる負債 💡
技術知識の問題	持っている技術知識を 正しく使わなかった	当初は存在しなかった 技術が登場した 使っていた技術が 古くなった	当初は知らなかった 技術を学んだ
ドメイン知識の問題	ドメイン知識を 正しくコードに 反映しなかった	当初とはドメイン 知識が変わった	当初は知らなかった ドメイン知識を学んだ より良いドメイン表現を 見つけた

意図的な負債 ⚡

- 持っている技術知識を正しく使わなかった
- ドメイン知識を正しくコードに反映しなかった

- 「品質より速度優先」で発生する
- 必ず避けられる

- コードの品質を犠牲にするのではなく、
  ターゲットを削るかリリース日を延長すべき

変化による負債 

- 当初は存在しなかった技術が登場した
  使っていた技術が古くなった
- 当初とはドメイン知識が変わった

- 予測しにくい技術的負債
- 時流を読み、想像することは一種のスキル

- 放置していくと、銀行の利息のように膨らんでいく
- 早期に回収することが望まれる

学びによる負債 

- 当初は知らなかった技術を学んだ
- 当初は知らなかったドメイン知識を学んだ、
  より良いドメイン表現を見つけた

- こういった負債が発生すること自体は良いことである
- 変化による負債と同様に、早期に回収することが好ましい

- 学ぶことで事前に防げる可能性はある

Ward Cunningham(ウォード・カニンガム)による説明

- たとえ理解が不完全だとしても、目の前の問題に対する**現時点での理解**を反映するコードを書くことには賛成です。
- そのソフトウェアに**現時点での理解**を**可能な限り反映させる**ことが重要です。
- そうしておけば、いざリファクタリングをするときが来たなら、コードには当時何を考えていたかが明快に残っているので、**現在の理解**に合わせてリファクタリングするのも**容易**になります。

【翻訳】技術的負債という概念の生みの親 Ward Cunningham 自身による説明 - t-wadaのブログ, https://t-wada.hatenablog.jp/entry/ward-explains-debt-metaphor
Debt Metaphor - Youtube, 2009, https://youtu.be/pqeJFYwnkjE

当初は知らなかった技術を学んだ

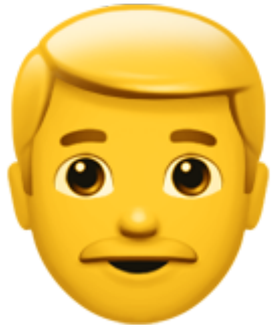
- 言語やライブラリの使い方を誤っていた、より良い使い方があった
- より適切なツールがあることを知らなかった
- より良いコードとは何か知らなかった

ここについて深堀りします

3

凝集度と結合度

CyberAgent



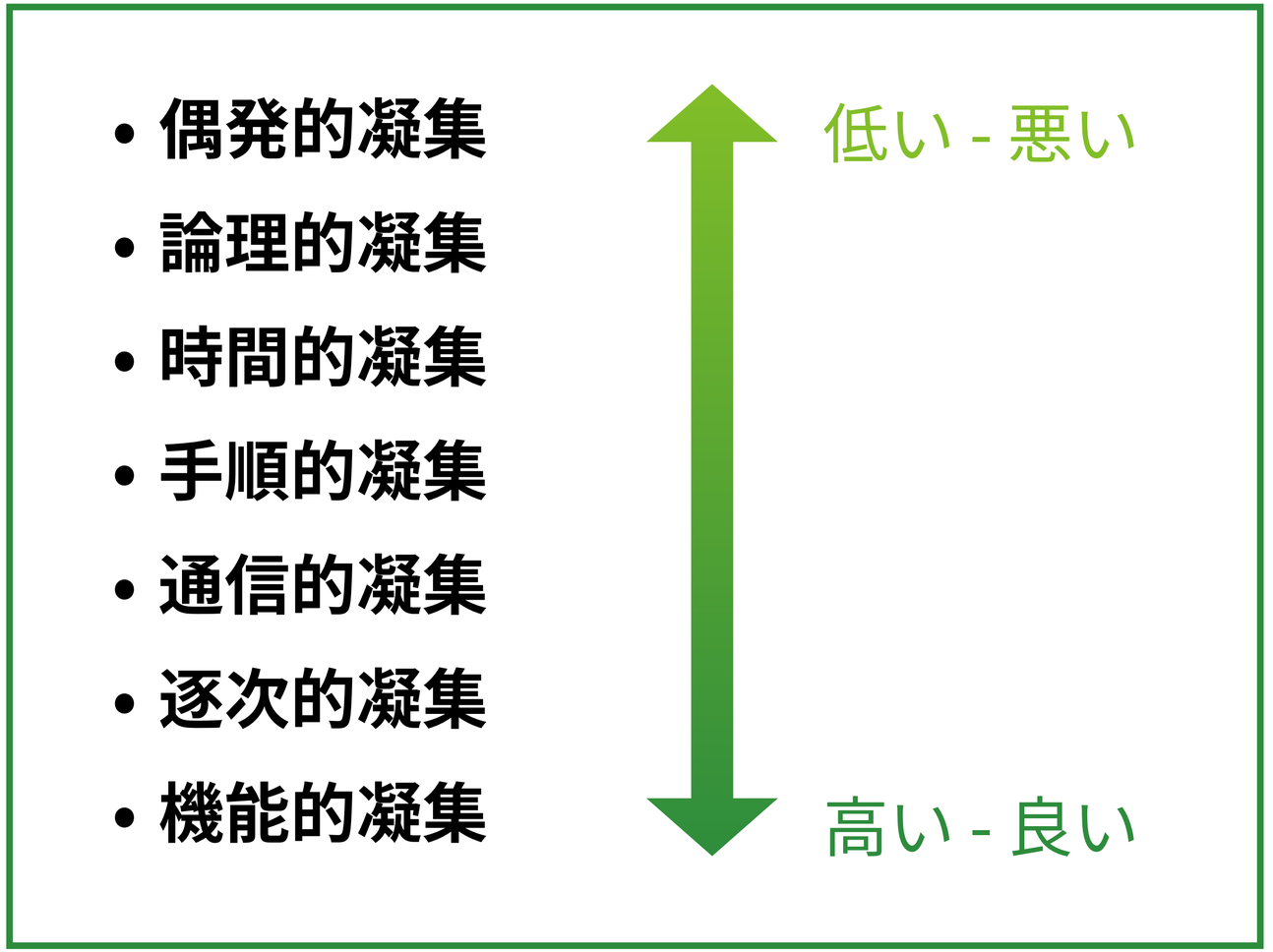
このコードは
こちらの書き方のほうが
好ましいと思います。

疑問④

よりよいコードであるという根拠は？

凝集度という一つの指標

- モジュール内の協調度を示す
  - パッケージ
  - クラス
  - メソッド
- 凝集度の高いモジュールは、
  堅牢性、信頼性、再利用性、
  読みやすさなどの点で好ましい
- 単一責任の原則（SOLID）



偶発的凝集

- **適当（無作為）**に集められたものがモジュールとなっている。
- モジュール内の各部分には特に**関連性はない**

```
fun main() {
    val data = getData() // データの取得
    println("Hello World") // 出力
    calcPrimeNumber(10) // 素数の計算
}
```

- 「とりあえず動く」みたいな状況

論理的凝集

- **論理的に似た**ようなことをするものを集めたモジュール
- 例えば：フラグによって動作を変える

```
// sampleAとsampleBが似ている
fun sample(isA: Boolean) {
    if (isA) { // もし、フラグがAなら
        sampleA() // Aを実行する
    } else {
        sampleB() // Bを実行する
    }
}
```

時間的凝集

- 時間的に近く動作するものを集めたモジュール
- 実行順序を入れ替えても動作する
- 例えば：初期化処理や、UIのフォアグラウンド時の処理など

```
// 初期化処理
fun initApp(isA: Boolean) {
    initConfig() // 設定の初期化
    initLogger() // Loggerの初期化
    initDB() // DBの初期化処理
}
```

手続き的凝集

- 順番に実行する必要があるものを集めたモジュール
- 共通したデータは使わない
- 例えば：アクセス権を確認してファイルに書き込む

```
// アクセス権を確認してファイルに書き込む
fun outputFile(file: File) {
    checkPermission() // 権限確認
    writeFile(file) // ファイル出力
}
```

通信的凝集

- **同じデータ**を扱う部分を集めたモジュール

```
// 変更をABCに通知する
fun changeAll(data: Data) {
    changeA(data)
    changeB(data)
    changeC(data)
}
```

逐次的凝集

- ある部分の出力が別の部分の入力となるような部分を集めたモジュール
- 例えば：ファイルを取得して変換して保存する

```
fun sample() {
    val file = getFile() // ファイルを取得する
    val transfromed = transform(file) // ファイルを変換する
    saveFile(transfromed) // ファイルを保存する
}
```

機能的凝集

- **単一の定義されたタスク**を実現するモジュール
- 例えば：2点間の距離を計算する

```
// 2点間の距離を計算する
fun calcLength(a: Point, b: Point): Int {
    val dx = a.x - b.x
    val dy = a.y - b.y
    val length = sqrt(dx * dx + dy * dy)
    return length.toInt()
}
```

凝集度の使い分け

- 偶発的凝集
- 論理的凝集
- 時間的凝集
- 手順的凝集
- 通信的凝集
- 逐次的凝集
- 機能的凝集

必ず避けるべき

可能な限り避けるべき

可能な限り小さく保つ

理想的

例えば：時間的凝集の関数

```
// アプリ起動時に行う処理
fun initApp() {
    initLogger() // ログの初期化

    val config = getConfig() // 設定の取得
    setupConfig(config) // 設定の反映

    initDB() // データベースの初期化
    initLocalStorage() // ローカルストレージの初期化

    val data = getData() // 表示用データの取得

    renderHeader() // ヘッダのUI表示
    renderMenu() // メニューのUI表示
    renderMainContent(data) // メインコンテンツのUI表示
    renderFooter() // フッターのUI表示
}
```

時間的凝集

凝集度が低いモジュールを小さくする

```
// アプリ起動時に行う処理
fun initApp() {
    prepare()
    setupUI()
}

// 時間的凝集
fun prepare() {
    initLogger() // ログの初期化
    initConfig() // 設定の初期化
    initDB() // データベースの初期化
    initLocalStorage() // ローカルストレージの初期化
}
```

手続き的凝集

時間的凝集

```
// 逐次的凝集
fun initConfig() {
    val config = getConfig() // 設定の取得
    setupConfig(config) // 設定の反映
}

// 逐次的凝集
fun setupUI() {
    val data = getData()
    renderUI(data)
}

// 手続き的凝集
fun renderUI(data: Data) {
    renderHeader() // ヘッダのUI表示
    renderMenu() // メニューのUI表示
    renderMainContent(data) // メインコンテンツのUI表示
    renderFooter() // フッターのUI表示
}
```

逐次的凝集

逐次的凝集

手続き的凝集

例えば：論理的凝集

```
// 購入または復元
fun purchaseOrRestore(isRestore: Boolean) {
    val receipt = getReceipt() // レシートの取得
    sendReceipt(receipt) // レシートの送信

    if (isRestore) {
        onSuccessRestore() // リストア成功
    } else {
        onSuccessPurchase() // 購入成功
    }
}
```

論理的凝集

共通化せず、分ける

```
fun restore() {
    processReceipt() // レシートの処理
    onSuccessRestore() // リストア成功
}

fun purchase() {
    processReceipt() // レシートの処理
    onSuccessPurchase() //  購入成功
}

fun processReceipt() {
    val receipt = getReceipt() // レシートの取得
    sendReceipt(receipt) // レシートの送信
}
```

手続き的凝集

手続き的凝集

逐次的凝集

関数は分ければ良いものではない

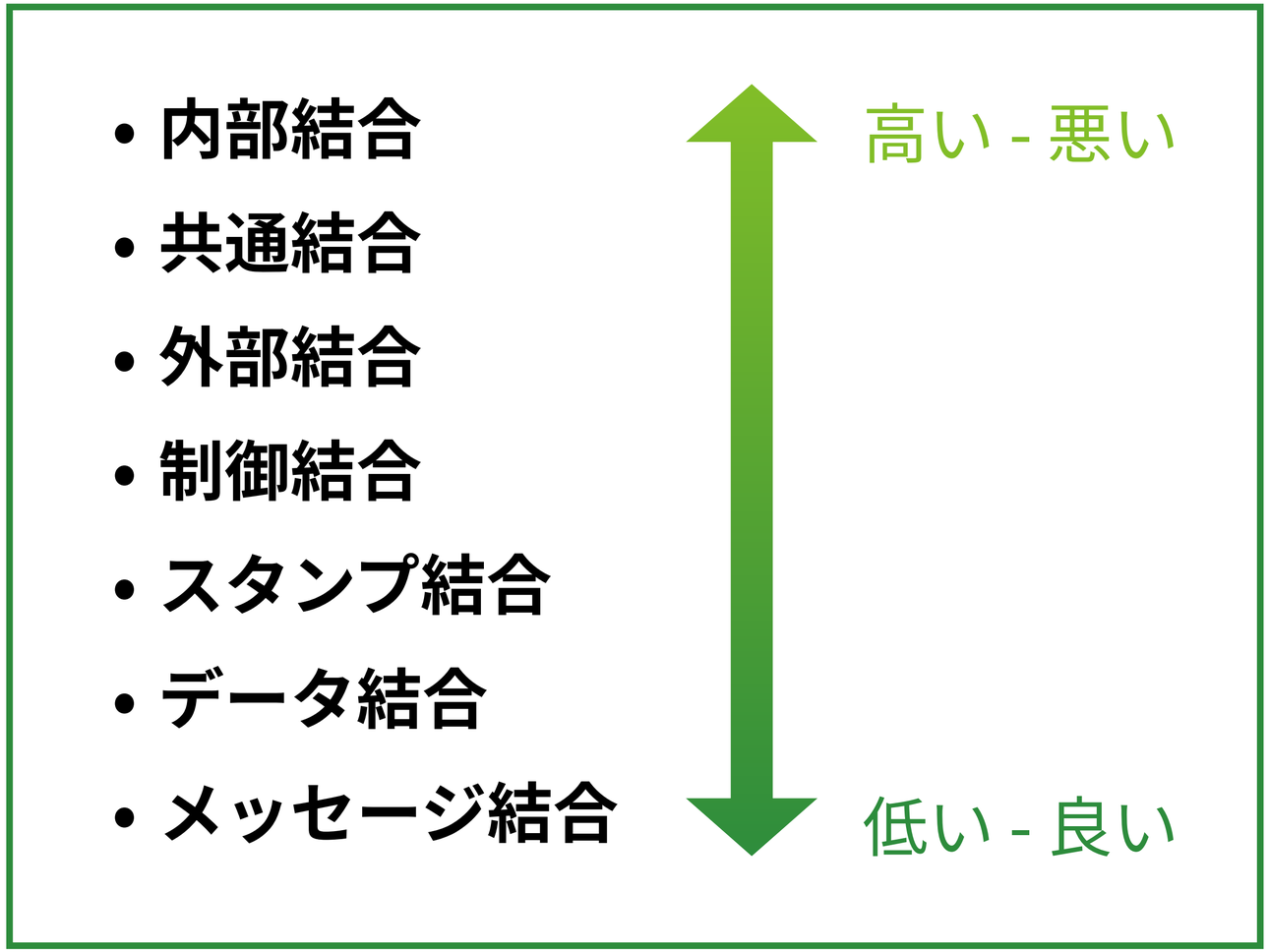
- 関数を分けることで、凝集度を高めることができる
- 関数が分かれると認知負荷は多少なりとも上がる
- 意味のわかる単位で区切ることが重要

▼ こっちのほうがわかりやすいかも…？

```
fun purchase() {
    val receipt = getReceipt() // レシートの取得
    sendReceipt(receipt) // レシートの送信
    onSuccessPurchase() // 購入成功
}
```

結合度という指標

- モジュール間の相互依存性の程度を示す
  - パッケージ
  - クラス
  - メソッド
- 結合度が低いと可読性と保守性が上がる
- 凝集度と相関が強く、凝集度が高くなれば結合度は低くなる



内部結合

- あるモジュールが別のモジュールの**内部動作**によって変化したり依存したりする
- 例えば：別のモジュールの内部データを**リフレクション等**で**直接参照**する

```
fun updatePrivateVariable() {
    val data = Data()
    val property = data::class.memberProperties
        .first { it.name == "value" } as KMutableProperty<*>
    property.isAccessible = true
    val updated = Value()
    property.setter.call(data, updated)
}
```

共通結合

- 複数のモジュールが同じグローバルデータにアクセスできる状態
- 変更が加えられると、予期しない副作用が発生する可能性がある

```
val data: Data = Data()

fun updateA() {
    data.value = "a"
}

fun updateB() {
    data.value = "b"
}
```

外部結合

- 標準化されたインタフェースをもつグローバルな状態を共有する
- 外部ツールや外部デバイスへの通信で発生する可能性がある
- 文献により解釈が一部異なる

```
fun function1() {
    Api.getData()
}

fun function2() {
    Api.update(Data())
}
```

制御結合

- あるモジュールに何をすべきかについての情報を渡すことで、
  別のモジュール処理の流れを制御する。
- **論理的凝集**が発生する

```
fun function1() {
    function2(true)
}

fun function2(flag: Boolean) {
    if (flag) {
        println("A")
    } else {
        println("B")
    }
}
```

スタンプ結合

- **構造体やクラス等の受け渡し**で結合されている
- 不必要にデータを渡す可能性がある

```
fun function1() {
    function2(User(name = "Mori Atsushi"))
}
```

データ結合

- **単純な引数**のやりとり
- 必要最小限のデータを渡す

```
fun function1() {
    function2(123, "abc")
}
```

メッセージ結合

- 引数のないやりとり
- データのやり取りは存在しない

```
fun function1() {
    function2()
}
```

結合度の使い分け

- 内部結合
- 共通結合
- 外部結合
- 制御結合
- スタンプ結合
- データ結合
- メッセージ結合

必ず避けるべき

可能な限り避けるべき

一部ケースで注意が必要

理想的

4

Clean Architecture

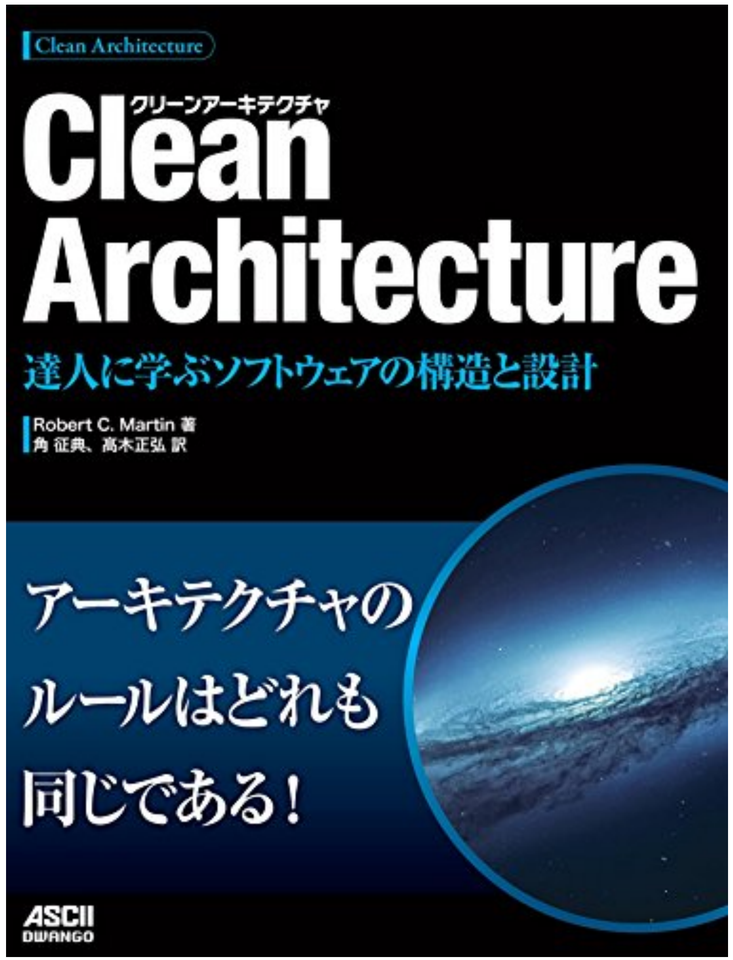
より良いコードとは

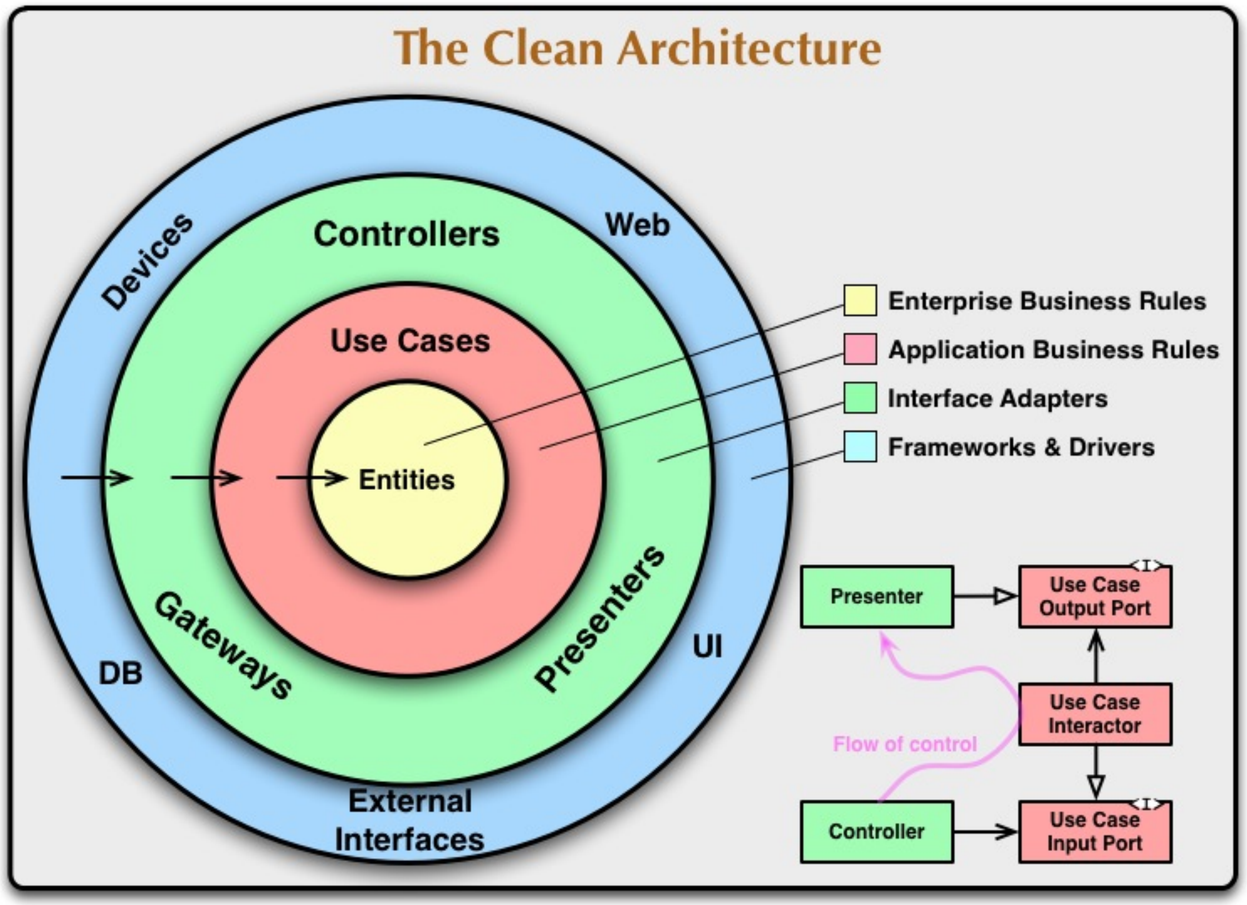
- より良いモジュール → 凝集度 

- より良いモジュール間の関係 → 結合度 

- より良いモジュール群 → ？

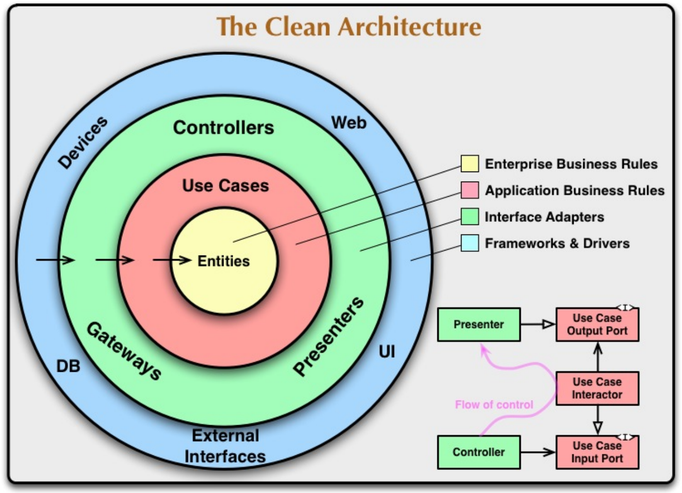
Clean Architecture



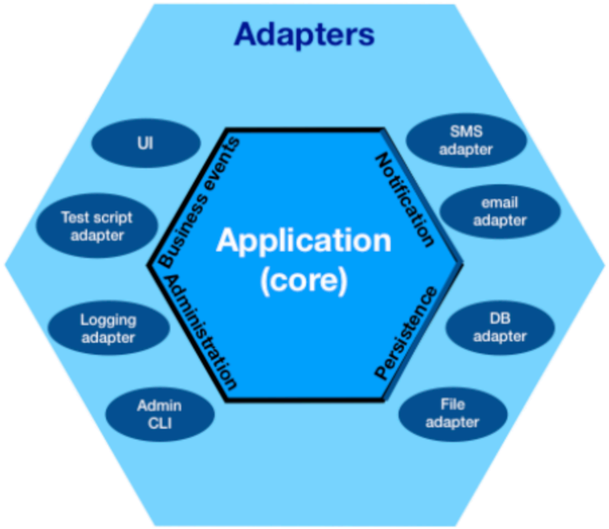


Robert C.Martin (2018), Clean Architecture 達人に学ぶソフトウェアの構造と設計, KADOKAWA
The Clean Architecture, https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

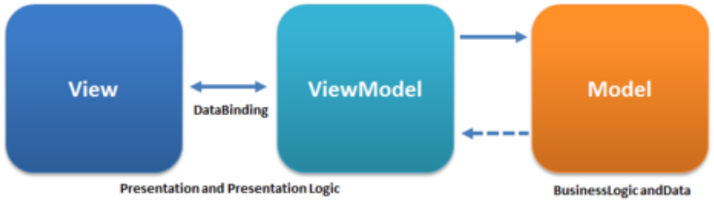
Q. どれがClean Architectureに準拠している？



Clean Architecture

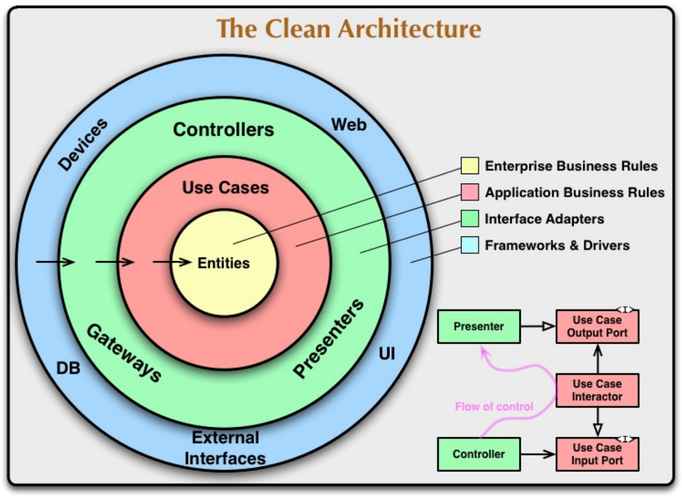


Hexagonal Architecture

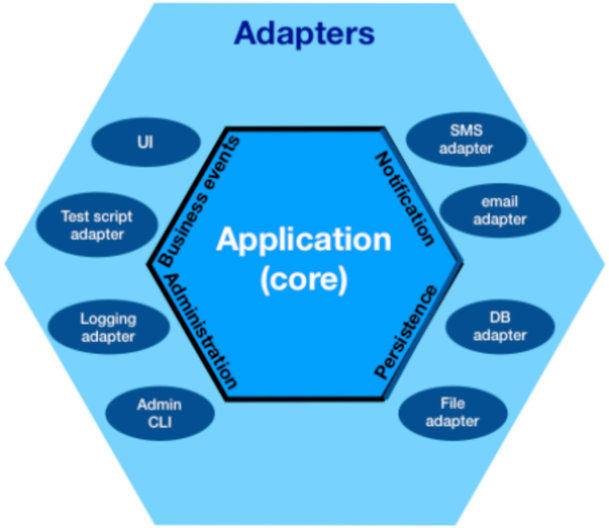


MVVM Architecture

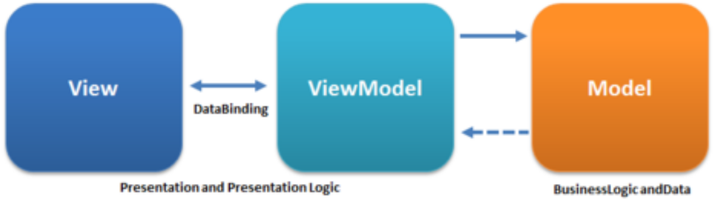
A. 全部



Clean Architecture



Hexagonal Architecture



MVVM Architecture

4つの円はあくまで例にすぎない

- Clean Architectureで主張されているルールは以下の2つだけである

① レイヤーに分離することで、関心事の分離を行う

② 依存性は内側だけに向かっていなければいけない

内側/外側とは？

- 外側
  - UI
  - データベース
  - 外部システム
  - フレームワーク
- 内側
  - ビジネスロジック
  - エンティティ

内側に近づくにつれ、ソフトウェアは抽象化され、

一般的なものになる必要がある

これらのルールに従うことで

- フレームワーク非依存
- テスト可能
- UI非依存
- データベース非依存
- 外部エージェント非依存

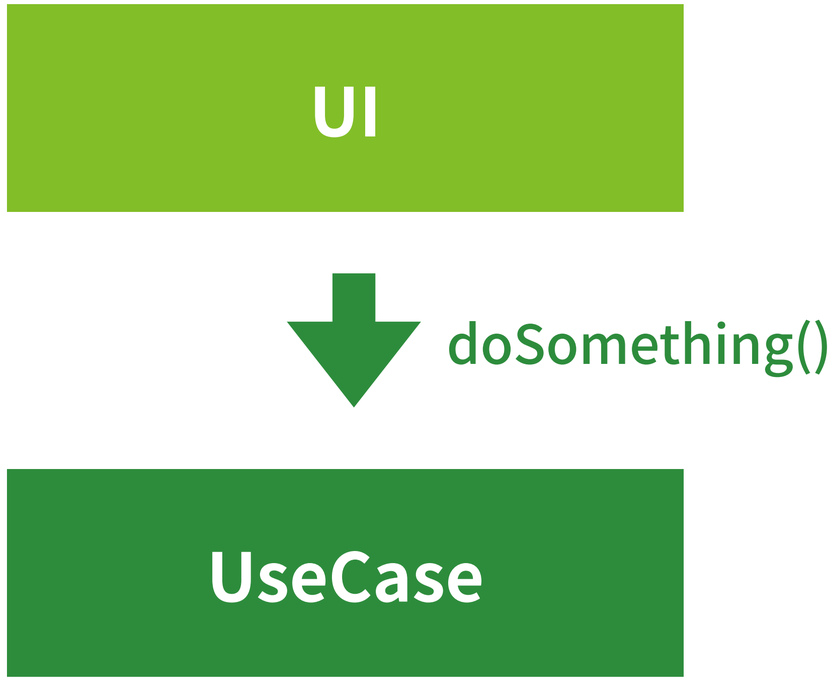
高い保守性を実現することができる

境界線を超える方法①

- **外側から内側**へのアクセスは、常に可能

```
class UI {
    val usecase: UseCase = UseCase()

    fun onClick() {
        usecase.doSomething()
    }
}
```

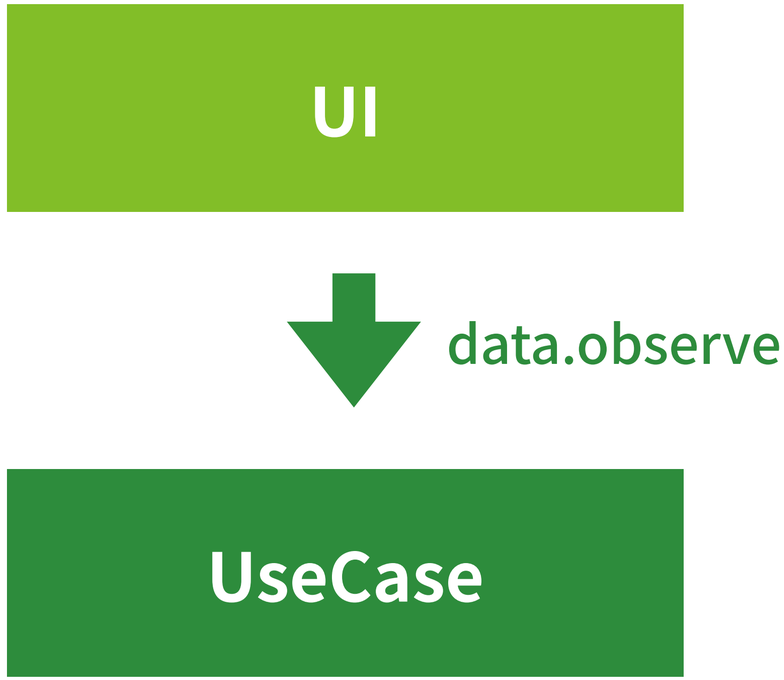


境界線を超える方法②

- ストリームを使って内側から外側にイベントを流す

```
class UI {
    val usecase: UseCase = UseCase()

    init {
        usecase.data.observe { value ->
            updateUI(value)
        }
    }
}
```



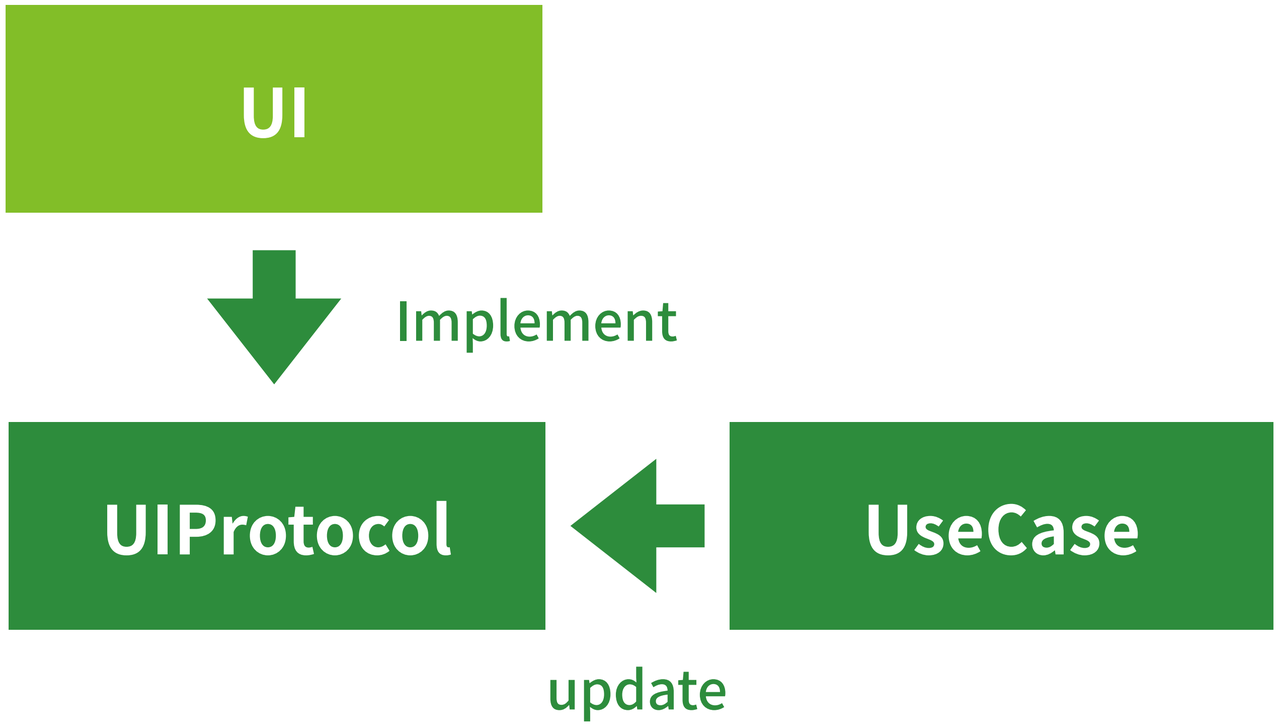
境界線を超える方法③

- **依存関係逆転の原則（DIP）**を使って**内側から外側**を呼び出す

```
interface UIProtocol {
    fun update(data: Data)
}

class UI: UIProtocol {
    override fun update(data: Data) {
        updateUI(data)
    }
}

class UseCase(
    val uiProtocol: UIProtocol
) {
    init {
        uiProtocol.update(Data())
    }
}
```



境界線を超えるデータ

- 単純なデータ構造が好ましい

  結合度を低くする

- 円の内側が外側について知るようなデータを渡してはいけない

  常に内側の知識のみで構成される

🚀 まとめ

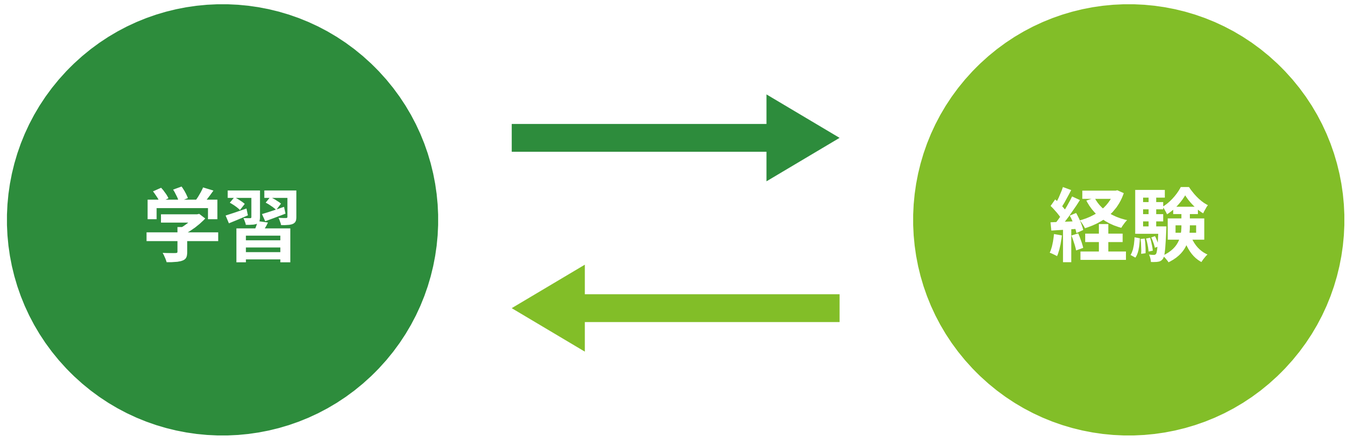
- **品質とスピード**はトレードオフではない
- 品質を上げるためには、**知識/経験**が必要
- **凝集度/結合度**の指標により、モジュールの評価をすることが可能
- **関心を分離**し、正しく**依存方向**を制御することで、
  クリーンなアーキテクチャを実現できる

品質を向上させるための知識は膨大

- 命名
- コメント
- コードレビュー
- オブジェクト指向
- 手続き型プログラミング
- 宣言的UI
- 自動テスト
- SOLIDの原則
- デザインパターン
- YAGNI
- KISSの原則
- 継承より移譲
- Dependency Injection
- Mutable / Imutable
- Null安全

最後に

- 今回の話にあんまりピンと来てない人もいるかもしれません
- **学習**と**経験**を繰り返すことで、初めて身についたスキルになります
- 数年後、**より深い理解**に到達することを期待しています



参考文献

- 質とスピード（2020秋100分拡大版） / Quality and Speed 2020 Autumn Edition, https://speakerdeck.com/twada/quality-and-speed-2020-autumn-edition
- Robert C.Martin (2018), Clean Architecture 達人に学ぶソフトウェアの構造と設計, KADOKAWA
- オブジェクト指向のその前に-凝集度と結合度/Coheision-Coupling, https://speakerdeck.com/sonatard/coheision-coupling
- 【翻訳】技術的負債という概念の生みの親 Ward Cunningham 自身による説明 - t-wadaのブログ, https://t-wada.hatenablog.jp/entry/ward-explains-debt-metaphor
- The Clean Architecture, https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- 世界一わかりやすいClean Architecture, https://www.slideshare.net/AtsushiNakamura4/clean-architecture-release
- Code readability, https://speakerdeck.com/munetoshi/code-readability