クリーンアーキテクチャ入門

変更に強い設計思想

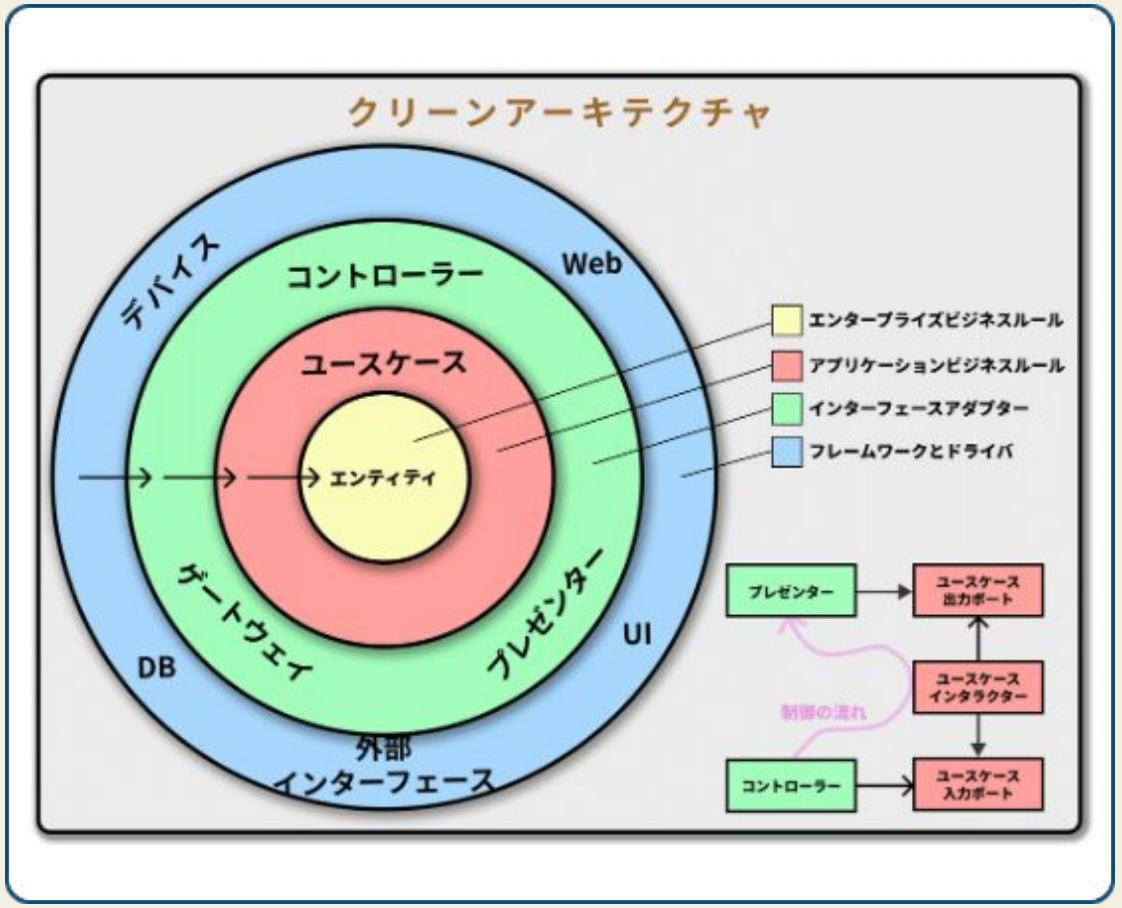
全体像 - 依存性のルール

クリーンアーキテクチャのルールはただ一つです。

「依存性の矢印は、常に内側へ向かう」

- **Entities (中心):** アプリケーションの核となるビジネスルール。
- **Use Cases:** アプリケーション固有の操作。
- **Interface Adapters:** データを変換する層。
- **Frameworks & Drivers (外側):** DBやWebフレームワーク。

ポイント: 内側の層（ビジネスロジック）は、外側の層（DBやUI）が何であるかを知りません。DBがMySQLからPostgreSQLに変わっても、内側の層は影響を受けません。



【実践】各層とコードの対応 (1/2)

1. Entities (ドメイン層)

役割: アプリケーションの核となるビジネスルール。

該当コード:

- value_objects.py (禁止ワード)
- item.py (Entity, Repository IF)
- item_domain_service.py (重複チェック)

2. Use Cases (アプリ層)

役割: 具体的な操作(ユースケース)の実行。ドメイン層のロジックを使って処理を組み立てます。

該当コード:

- item_command_service.py (登録・更新・削除)
- item_query_service.py (参照)

【補足】CQRS（コマンド・クエリ責任分離 ）

✏️Command（コマンド）

「状態を変更する」責任を持つ操作です。

- 登録（Create）
- 更新（Update）
- 削除（Delete）

該当コード： item_command_service.py

特徴： データの整合性を保つため、トランザクション処理や複雑なバリデーションを伴うことが多いです。

🔍Query（クエリ）

「データを参照する」責任を持つ操作です。

- 一覧取得（Read List）
- 詳細取得（Read Detail）

該当コード： item_query_service.py

特徴： データを変更しないため、ロジックは比較的シンプル。読み取り専用のDBレプリカを参照するなど、性能最適化がしやすいです。

【実践】各層とコードの対応 (2/2)

3. Interface Adapters

役割: 外部とUse Case層の間でデータを変換します。

該当コード:

- item_api.py (Controller)
- item_schemas.py (DTO / データ変換)

4. Frameworks & Drivers

役割: DB、フレームワークなど技術的詳細。

該当コード:

- item_repository_adapter.py (DB操作)
- item_repository.py (IFの実装)
- database.py (DB接続設定)
- main.py (FastAPI起動)

【最重要】依存性逆転 (DIP)の仕組み

- **[Domain層]** ItemRepository という**「インターフェース（抽象）」**を定義。「データをどう保存するかは知らないが、こういうIFが必要だ」と宣言だけします。
- **[Application層]** 抽象的な ItemRepository 型にのみ依存。**具体的なDB実装（ItemRepositoryImpl）を一切知りません。**
- **[Infrastructure層]** Domain層で定義された ItemRepository を**「実装」**します（class ItemRepositoryImpl(ItemRepository)）。
- **[DI（依存性の注入）]** item_api.py (Controller) が、Application層にInfrastructure層の実装を**「外から注入」**します。
- **[結論]** 制御は外→内へ流れますが、**依存関係は内側が外側（ DB）を知らない状態** が完成します。

メリット①：テストが容易になる

Domain層のテスト

tests/test_domain_item.py

- ビジネスルール（例：禁止ワードチェック）をテストします。
- DBやWebフレームワークに一切依存しません。
- **Python単体で高速にテスト可能です。**

Application層のテスト

tests/test_item_service.py

- ユースケース（操作）のロジックをテストします。
- 本物のDB（インフラ層）を**「偽物（モック）」**に差し替えます。
- **DB接続なしで、ビジネスロジックの正しさだけを検証できます。**

メリット②：変更に強くなる

シナリオA：仕様変更

依頼:「商品名に『セール』も禁止してほしい」

対応: src/domain/value_objects.py のリストに追加するだけです。

結果: **ビジネスルールの変更が Domain層 だけで完結します。**

シナリオB：技術移行

依頼:「DBをMySQLからPostgreSQLに移行したい」

対応: src/infrastructure 配下のDB接続・実装コードのみを修正します。

結果: **application層やdomain層のビジネスロジックは修正不要です。**

まとめ

- クリーンアーキテクチャは「関心の分離」と「依存性逆転」を重視する設計思想です。
- **メリット:** テストが容易になり、仕様変更や技術移行に強くなります。
- **コスト:** ファイル数が増えたり、学習コストがかかったりする側面もあります。
- **結論:** 長期保守するシステムや、仕様変更が多いSI案件では、強力な武器となります。

おわり

ご清聴ありがとうございました。

資料

The Clean Architectureを読む｜みきふく【スキでモチベ UP!!】

https://github.com/kuriboo1002/python-api-sample/tree/feature/clean-architecture