関数型アーキテクチャによるテスト容易な設計

課題：ひどいコードの例（特徴）

- 外部クラスの可変変数を関数内で直接参照している
- DBアクセスしている
- 複雑なロジックが組まれている（深いネストに複雑な条件式や大量の計算式）

→ **どうやってテストする？**

課題：ひどいコードの例

```
# ❌ ひどいコードの例
class DiscountConfig:
    rate = 0.1  # 外部クラスの可変変数

def calculate_order_total(order_id: int) -> float:
    conn = get_db_connection()  # DBアクセス
    rows = conn.execute(
        "SELECT price, qty, user_type FROM order_items WHERE order_id = ?",
        (order_id,)
    ).fetchall()

    total = 0.0
    for row in rows:
        price, qty, user_type = row
        if user_type == "premium":           # 深いネスト開始
            if qty >= 10:
                if price > 1000:
                    total += price * qty * (1 - DiscountConfig.rate) * 0.8  # 複雑な計算
                else:
                    total += price * qty * (1 - DiscountConfig.rate) * 0.9
            else:
                total += price * qty * (1 - DiscountConfig.rate)
        else:
            if qty >= 10:
                total += price * qty * 0.95
            else:
                total += price * qty

    return total
```

こうなってしまった場合の対処

- Linter → 設計までは面倒見てくれない
- **モックを使う** → DBやグローバル変数、ロジックが絡み合い、モックが仕込めない
- **厳格なコーディング規約** → ルールが多すぎると守られないし、アーキテクチャまでは縛れない

→ **アーキテクトの段階からテスト容易性を考える**

解決策：関数型アーキテクチャ

関数を純粋関数と副作用のある関数に分け、**副作用のある関数に複雑なロジックを持たせない**ようにするアーキテクチャ

基本思想：純粋関数と副作用のある関数の分離

- **純粋関数 (Pure Functions)**
  - ビジネスロジック・計算・条件分岐を担当
  - 外部状態にアクセスしない
  - 同じ入力に対して常に同じ出力を返す
- **副作用のある関数 (Impure Functions)**
  - DB操作・API通信・ファイルI/Oを担当
  - 複雑な実装はしない

※ ログ出力は厳密には副作用だが、テストを不安定にしないため純粋関数内でも許容する

２つの「複雑な実装」の定義

ブラックリスト方式（緩い規約） — いずれか1つでも含まれたら切り出す

- ネストした制御構文（深さ2以上）— `for` の中に `if`、`if` の中に `if`
- ビジネスルールの計算 — 金額・税率・割引率の計算、ステータス変更の条件判定
- 複数ステップのデータ加工 — リスト・辞書の生成・加工・フィルタリングが複数行

ホワイトリスト方式（より厳格な規約） — 以下**以外**が含まれたら切り出す

- インスタンス生成と関数の呼び出し
- データの単純な受け渡し
- ガード節（早期リターン: `if not user_id: return`）
- 例外のキャッチと変換（トップレベルの `try-except`）

マトリクスで考える

	簡単な実装	複雑な実装
副作用なし	単体テスト (or テスト不要)	単体テスト
副作用あり	単体テスト (Mock必要)	**テスト困難！**

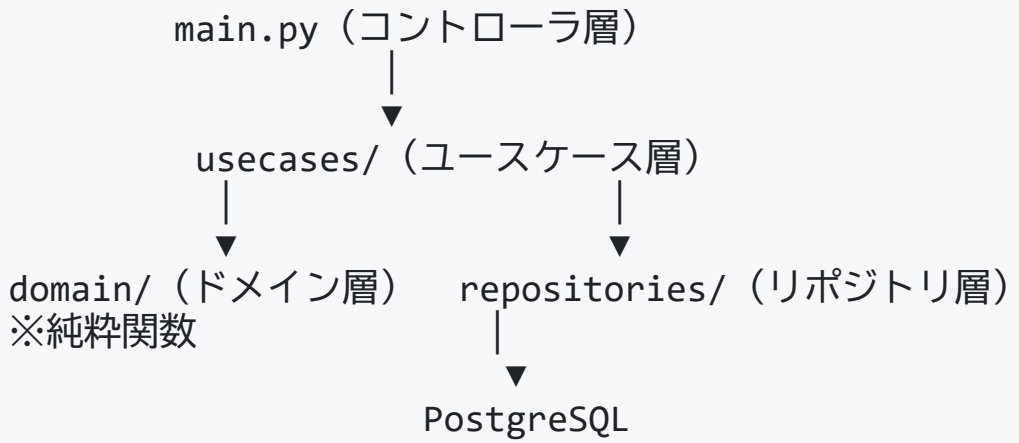
良いコードの例（純粋関数部分）

```python
# ✅ 良いコードの例
# 外部クラスの変数を引数として受け取ることで純粋関数化
def calc_item_total(
    price: float, qty: int, user_type: str, discount_rate: float
) -> float:
    if user_type == "premium":
        if qty >= 10:
            multiplier = 0.8 if price > 1000 else 0.9
        else:
            multiplier = 1.0
        return price * qty * (1 - discount_rate) * multiplier
    else:
        multiplier = 0.95 if qty >= 10 else 1.0
        return price * qty * multiplier

def calc_order_total(
    items: list[tuple[float, int, str]], discount_rate: float
) -> float:
    return sum(
        calc_item_total(price, qty, user_type, discount_rate)
        for price, qty, user_type in items
    )
```

良いコードの例（副作用のある関数部分）

```python
# 副作用のある関数：DBアクセスだけ、ロジックなし
def get_order_items(order_id: int) -> list[tuple[float, int, str]]:
    conn = get_db_connection()
    return conn.execute(
        "SELECT price, qty, user_type FROM order_items WHERE order_id = ?",
        (order_id,)
    ).fetchall()

def calculate_order_total(order_id: int) -> float:
    """エントリーポイント（副作用あり・ロジックなし）"""
    items = get_order_items(order_id)          # 副作用
    discount_rate = DiscountConfig.rate        # 外部状態の取得（副作用）
    return calc_order_total(items, discount_rate)  # 純粋関数
```

層の依存関係と呼び出しフローの例



モックの有無について

外部インターフェース（DB操作/IO）を実装の詳細と扱っていい場合：

- 現システム以外で更新されないDB
- テスト用のファイルが用意できるファイル操作

※ 名前的型付け言語（Java, C# 等）には必ずインターフェースを使うこと

まとめ

副作用のある複雑な実装をするな

	簡単な実装	複雑な実装
副作用なし	✅	✅
副作用あり	✅	❌

余談：Reactにおける関数型アーキテクチャ

同じ考え方はフロントエンド（React + Atomic Design）にも自然に当てはまる。

コンポーネントを「純粋関数」として捉える

- Atomic Designでは `Page` が最上位のコンテナ
- **Pageより下のコンポーネントはすべて「props → JSX」の純粋関数**

```
// ✅ 純粋関数コンポーネント — propsを受け取りDOMを返すだけ
type UserCardProps = { name: string; email: string; };

const UserCard = ({ name, email }: UserCardProps) => (
  <div>
    <p>{name}</p>
    <p>{email}</p>
  </div>
);
```

外部状態・通信・副作用を一切持たないので、Storybookやユニットテストで検証できる。

通信の副作用はPageに集約する

API通信はどうしても副作用になる。以下の3つに分割する。

関数の種類	責務	副作用
URLルーティング関数	エンドポイントの構築	なし（純粋）
デコーダー関数	レスポンスの型変換・バリデーション	なし（純粋）
カスタムhook	上2つを合成してfetchを実行	あり（単純）

通信の副作用はPageに集約する（コード例 1/2）

```
// URLルーティング関数（純粋関数）
const buildUserUrl = (userId: string): string => `/api/users/${userId}`;

// デコーダー関数（純粋関数）
type UserResponse = { id: string; name: string; email: string };

const decodeUser = (raw: unknown): UserResponse => {
  if (
    typeof raw !== "object" || raw === null ||
    !("id" in raw) || typeof (raw as Record<string, unknown>).id !== "string" ||
    !("name" in raw) || typeof (raw as Record<string, unknown>).name !== "string" ||
    !("email" in raw) || typeof (raw as Record<string, unknown>).email !== "string"
  ) {
    throw new Error("Invalid user response");
  }
  return raw as UserResponse;
};
```

通信の副作用はPageに集約する（コード例 2/2）

```
// カスタムhook（副作用あり・ロジックなし）
const useUser = (userId: string) => {
  const [user, setUser] = useState<UserResponse | null>(null);
  const [error, setError] = useState<Error | null>(null);
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    let cancelled = false;
    setLoading(true);
    fetch(buildUserUrl(userId))
      .then((res) => res.json()).then(decodeUser)
      .then((data) => { if (!cancelled) setUser(data); })
      .catch((err) => { if (!cancelled) setError(err); })
      .finally(() => { if (!cancelled) setLoading(false); });
    return () => { cancelled = true; };  // race condition防止
  }, [userId]);

  return { user, error, loading };
};

// Page — hookを呼ぶだけ、ロジックなし
const UserPage = ({ userId }: { userId: string }) => {
  const { user, error, loading } = useUser(userId);
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error.message}</p>;
  if (!user) return null;
  return <UserCard name={user.name} email={user.email} />;
};
```

React まとめ

レイヤー	テスト方法
`UserCard` （純粋コンポーネント）	propsを渡してレンダリング検証（Storybook / Testing Library）
`buildUserUrl`（純粋関数）	入出力の単体テスト
`decodeUser`（純粋関数）	正常系・異常系の単体テスト
`useUser`（副作用hook）	MSWでfetchをモックして統合テスト
`UserPage`	E2Eテストか、hookをモックして薄くテスト

副作用（fetch）は hookに閉じ込め、ロジックは純粋関数に追い出す
バックエンドの関数型アーキテクチャと同じ発想。

※ hookが肥大化していくので、実際はTanStack Queryを使うことが多い