

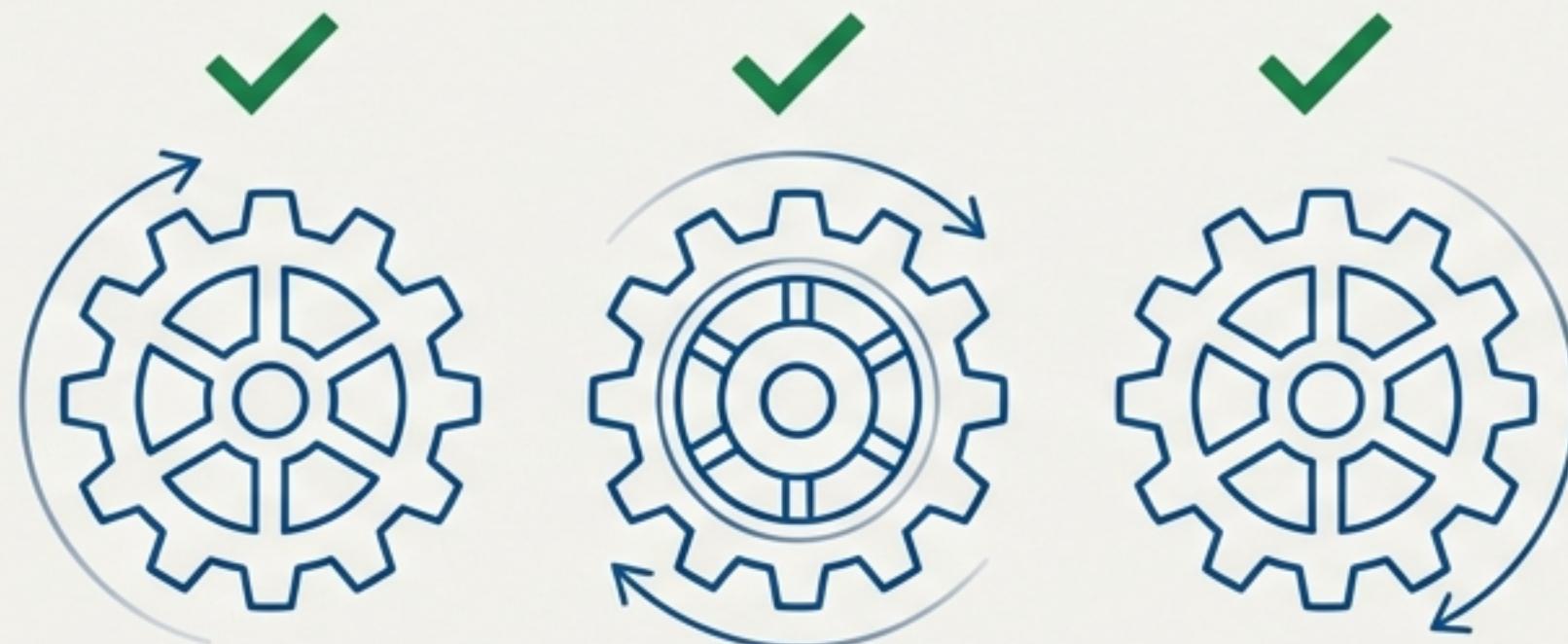
なぜ、統合 (Integration) テスト を行うのか？

単体テストの先へ：堅牢なソフトウェアと優れた設計を実現するためのガイド

- 統合テストの役割
- テスト・ピラミッドについてのさらなる考察
- 価値ある統合テストの書き方

単体テストはパスする。しかし、システムは壊れている。

単体テスト (Unit Tests)



システム全体 (System)



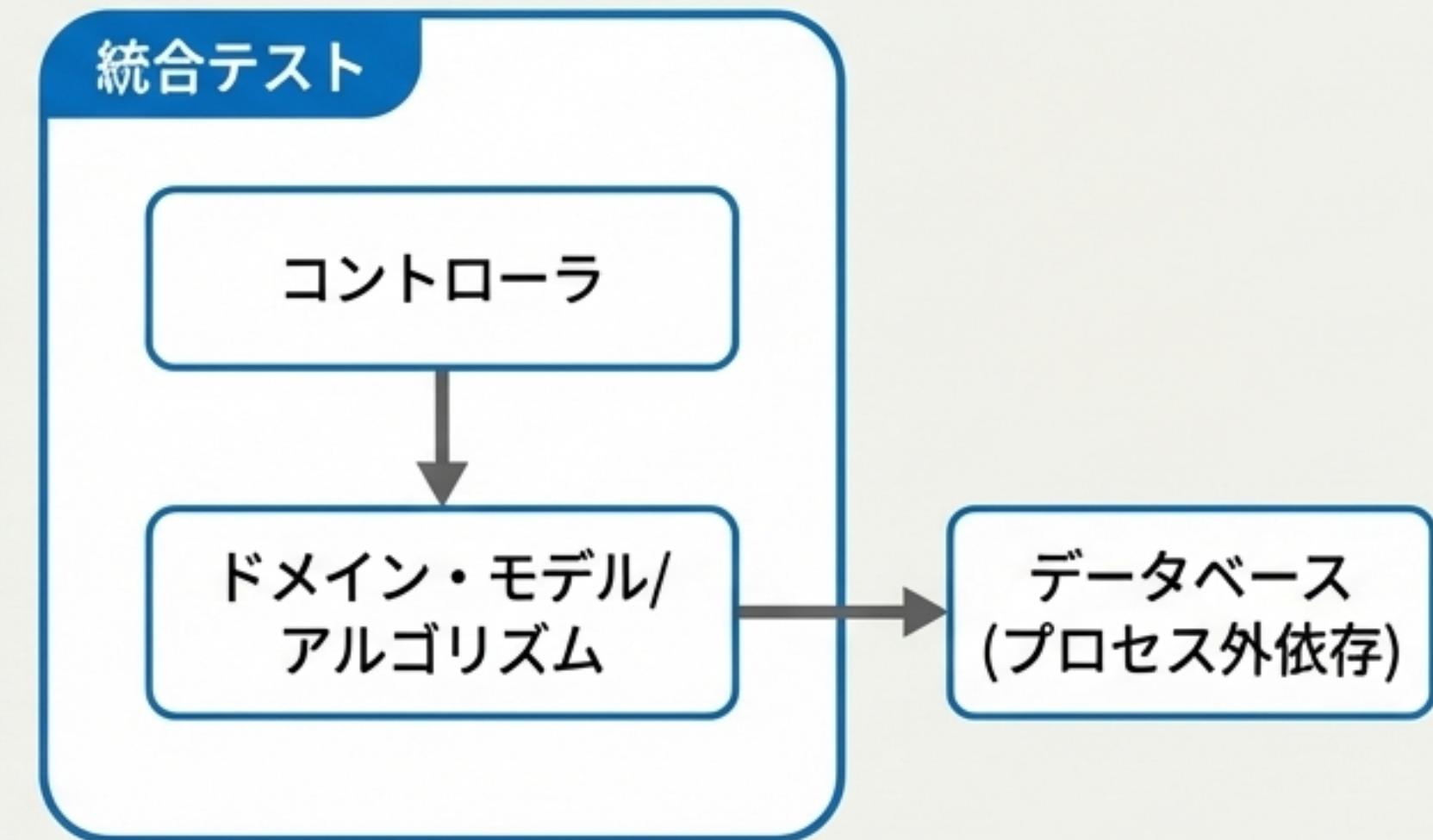
なぜ、単体テストだけでは不十分なのか？

- ・ 単体テストは、個々のビジネス・ロジックを独立して検証するには優れている。
- ・ しかし、**システム全体**の振る舞いは、個々の部品の**総和以上**のものである。
- ・ 重要なビジネス・ロジックは、複数のコンポーネント（コントローラ、ドメイン・モデル、データベース、メッセージ・バスなど）の「**協調**」によって実現される。
- ・ 単体テストでは、このコンポーネント間の**連携**が正しく機能するかどうかを確認できない。

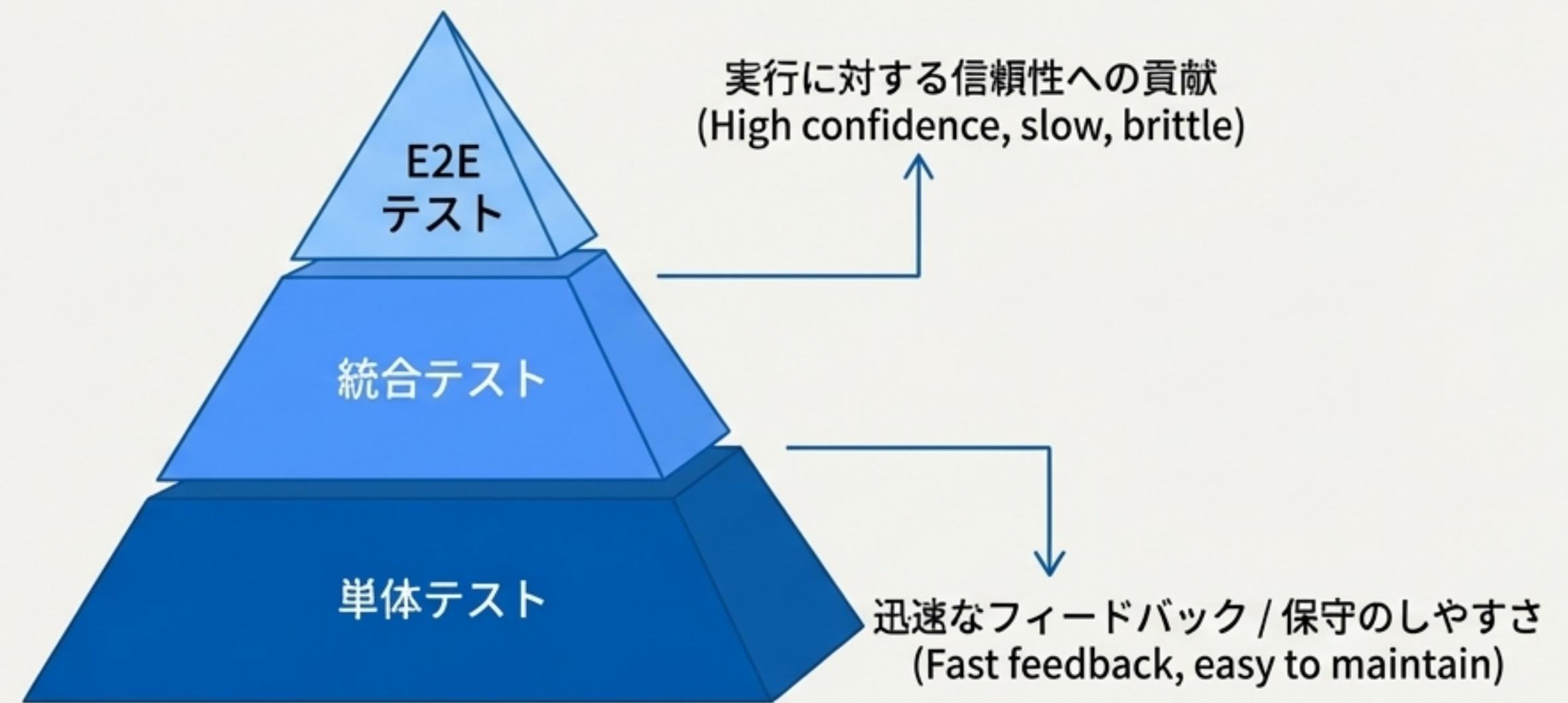
統合テストの真の目的：コンポーネント間の「協調」を検証する

定義: 統合テストとは、システムがプロセス外依存と統合した状態で、その振る舞いを検証するテストである。

- **単体テスト:** 1単位の振る舞い (a unit of behavior) を検証する。コードを分離してテストする。
- **統合テスト:** 複数のコンポーネントが連携する際の振る舞いを検証する。コードを統合してテストする。

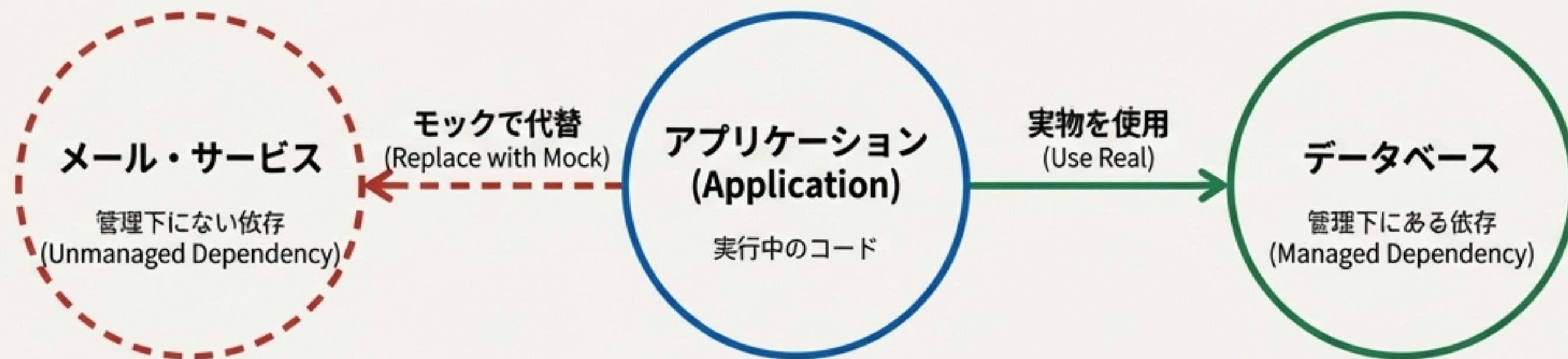


テスト戦略の要：テストピラミッドにおける統合テストの位置付け



- 統合テストは、単体テストの「速さ」とE2Eテストの「信頼性」のバランスを取る重要な中間層である。
- ビジネス・シナリオの大部分（ハッピーパスと主要なエッジケース）を単体テストよりも高い信頼性で検証し、E2Eテストの数を最小限に抑えることを目指す。

統合テストの核心的課題：プロセス外依存をどう扱うか？



すべてのプロセス外依存は、大きく次の2つに分類できる。

1. 管理下にある依存 (Managed Dependency)

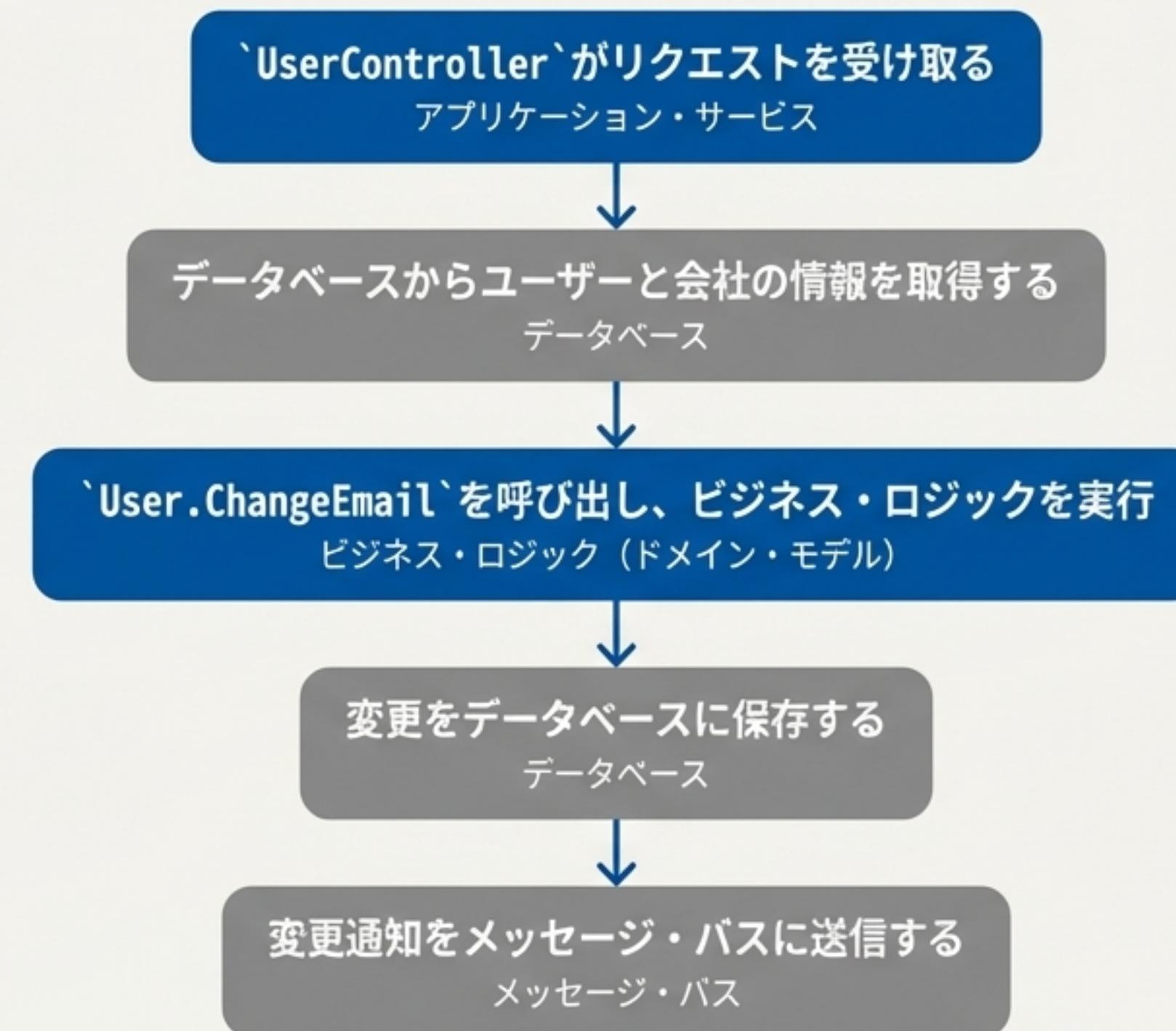
- テスト対象のアプリケーションが管理するプロセス外依存。
- 例: アプリケーションが直接所有し、スキーマを管理できるデータベース。
- アプローチ: テストでは実物のインスタンスを使用する。

2. 管理下にない依存 (Unmanaged Dependency)

- テスト対象のアプリケーションが管理しないプロセス外依存。外部のチームやサードパーティが所有する。
- 例: サードパーティ製のAPI、メール配信サービス (SMTPサーバー)、メッセージ・バス。
- アプローチ: テストではモックに置き換える。

ケーススタディ：ユーザーのメールアドレス変更処理

シナリオ: ユーザーが企業ドメインのメールアドレスを、非企業の個人メールアドレスに変更するシナリオを考える。



最初の統合テスト：ハッピーパスをコードで検証する

```
public void Changing_email_from_corporate_to_non_corporate()
{
    // 準備 (Arrange)
    var db = new Database(ConnectionString);
    var user = CreateUser("user@mycorp.com", UserType.Employee, db);
    var messageBusMock = new Mock<IMessageBus>();
    var sut = new UserController(db, messageBusMock.Object);

    // 実行 (Act)
    var result = sut.ChangeEmail(user.Id, "new@gmail.com");

    // 検証 (Assert)
    Assert.Equal("OK", result);
    var userInDb = db.GetUserById(user.Id);
    Assert.Equal("new@gmail.com", userInDb.Email);
    Assert.Equal(UserType.Customer, userInDb.Type);

    messageBusMock.Verify(
        x => x.SendEmailChangedMessage(user.Id, "new@gmail.com"),
        Times.Once);
}
```

管理下にある依存：
実際のデータベースに接続

管理下にない依存：モックで代替

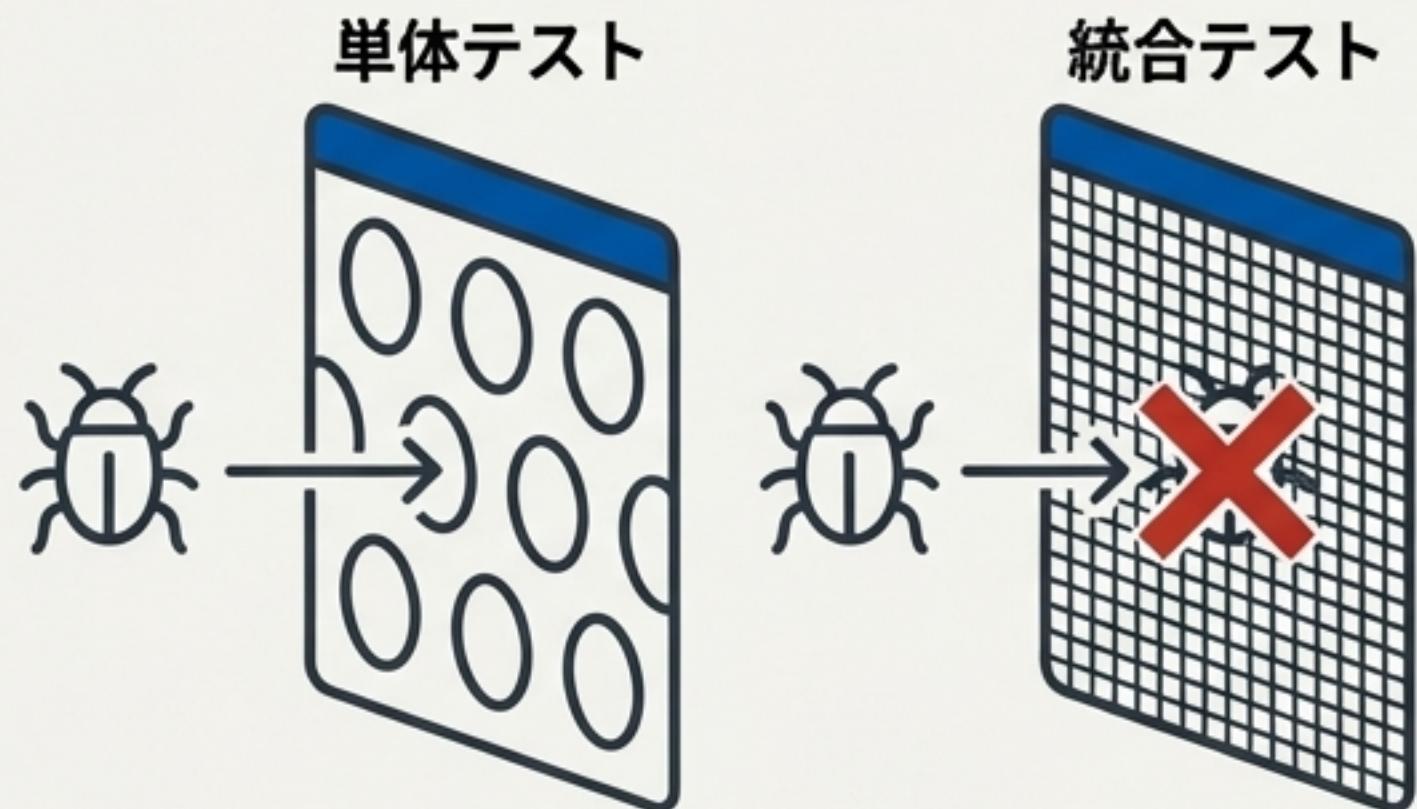
実行フェーズ (Act)

結果の検証①：データベースの状態変化を確認

結果の検証②：外部システムへの通知が行われたことを確認

早期失敗 (Fail Fast) の原則：単体テストが見逃すバグを素早く発見する

統合テストの価値は、ハッピーパスだけでなく、単体テストでは検証が困難な「異常ケース」を早期に発見できることにある。バグが後の工程（QAや本番環境）で発見されるのを防ぐ。



Example Scenario: 存在しない会社にユーザーを所属させようとする

- **単体テスト:** UserControllerの単体テストでは、companyがnullである場合の事前条件チェックはパスしてしまうかもしれない。
- **統合テスト:** 実際のデータベースとやり取りするため、database.GetCompany()がnullを返し、後続の処理でエラーが発生する。これにより、システム全体としてシナリオが失敗することを**早期に検知できる**。

Key Insight: バグは、存在すれば、すぐにそのことが分かるようにコードを書くべきである。早期失敗の原則は、統合テストの代わりに事前条件チェックを乱用する誘惑と戦うための道となる。

優れた設計への道標：インターフェースによる依存関係の分離

The Problem: コード内で依存関係を直接インスタンス化（例: `new Database()`）すると、コードが密結合になり、テストが困難になる。

The Solution: 依存関係逆転の原則 (DIP) とインターフェースを使用する。具象クラスではなくインターフェースに依存し、依存関係はコンストラクタ経由で注入 (Dependency Injection) する。

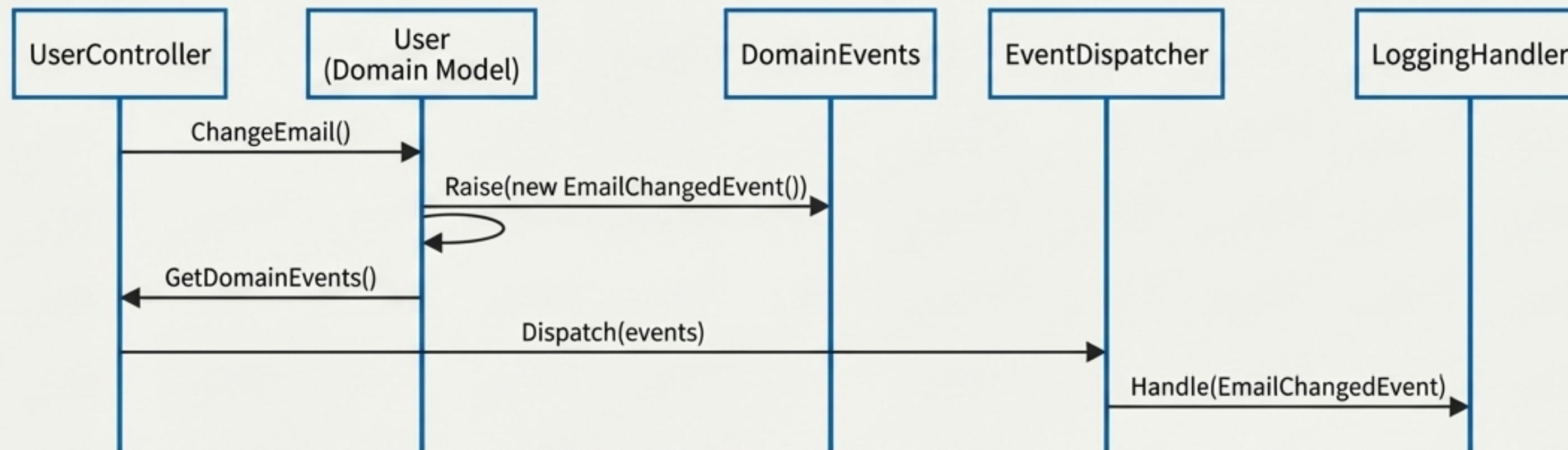
Before (密結合)	After (疎結合)
<pre>public class UserController { private readonly Database _database; public UserController() { _database = new Database(); // <-- Problem } }</pre>	<pre>public class UserController { private readonly IDatabase _database; public UserController(IDatabase database) { // <-- Solution _database = database; } }</pre>

- ✓ テスト容易性の向上: テスト時にモックやスタブを容易に注入できる。
- ✓ 柔軟性と保守性の向上: 実装の変更が容易になる（開放/閉鎖原則）。

発展：ロギングをビジネスロジックから分離するには？

Problem: ビジネスロジック内のログ出力コードは、ロジックを複雑化させ、テストを困難にする。

Solution: ドメイン・イベントを使用して、ビジネス上の決定（何をログ出力するか）と技術的な詳細（どうログ出力するか）を分離する。



Benefit: ドメイン・ロジックは永続化や通知の仕組みから完全に独立し、クリーンでテストしやすくなる。

統合テストが導く優れたアーキテクチャの原則

統合テストを正しく適用すると、コードベース全体の健全性が改善される。

1. ドメイン・モデルの境界を明確にする

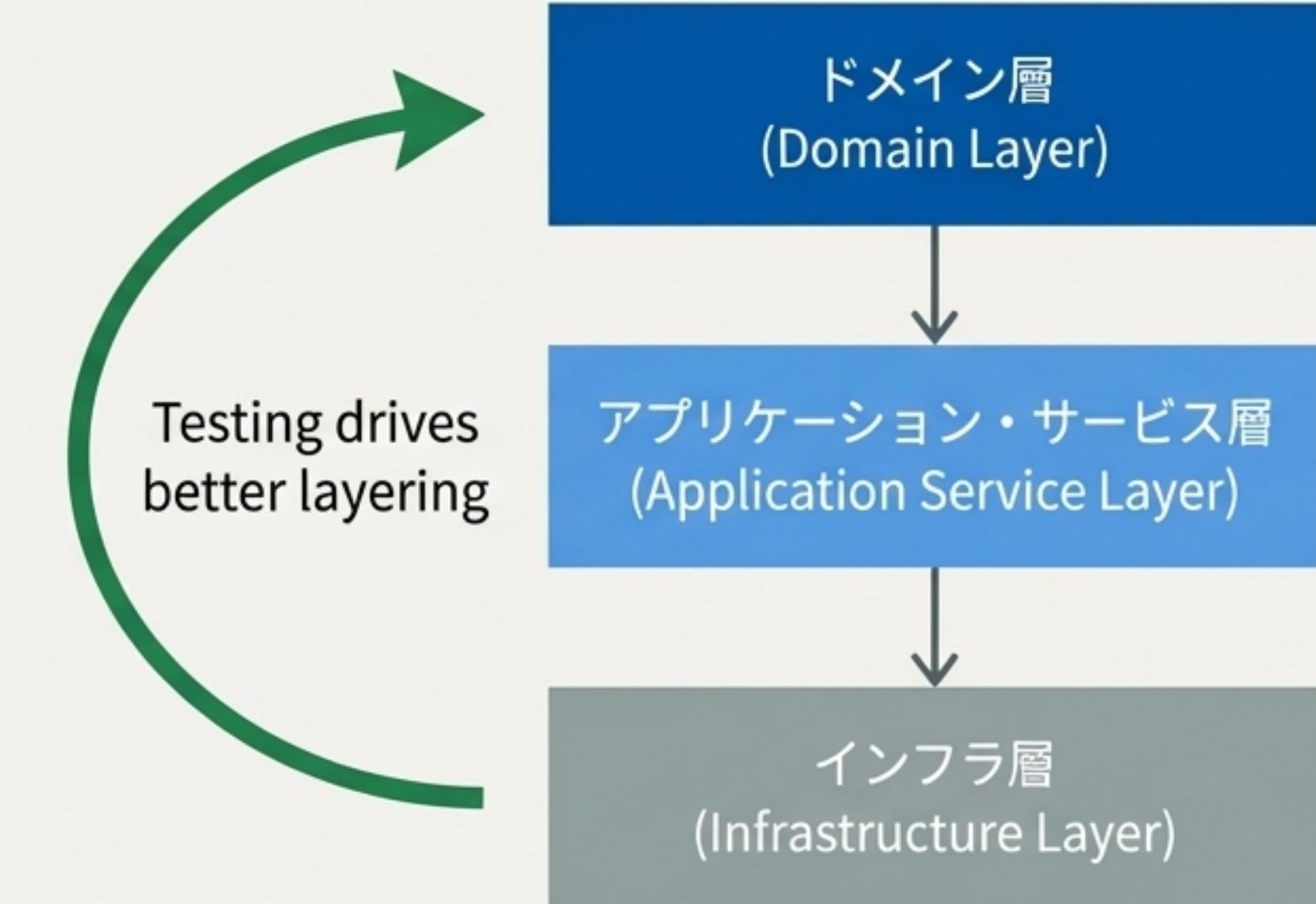
テストは、ドメイン・ロジックがインフラの関心事（DBアクセスなど）から分離されていることを強制する。

2. 不要な間接層を削減する

テストを書くことで、過剰な抽象化や不要なコードの層が明らかになり、シンプルな設計を促進する。

3. 循環的依存関係を排除する

統合テストを書くプロセスで、クラス間の循環的依存を解消せざるを得なくなり、より健全な依存関係グラフが生まれる。



まとめ：価値ある統合テストのための実践的ベストプラクティス

✓ 1テストケースにActフェーズは1つだけ

テストの焦点を明確に保ち、失敗時の原因特定を容易にする。

✓ 管理下にある依存とない依存を区別する

管理下にある依存（DBなど）は本番同様のコンポーネントを使い、管理下にない依存（外部APIなど）はモックを使う。

✓ ハッピーパスと主要な異常系シナリオを網羅する

単体テストでカバーしきれない、コンポーネント間の連携に起因する問題を発見する。

✓ 診断ログの出力はテストしない

ログ出力は外部システムへの副作用とみなし、ドメイン・ロジックのテストとは分離する。

統合テストは、書かれたコードを検証するだけの作業ではない。

それは、保守性が高く、堅牢で、理解しやすいコードへと導くための、**リファクタリングの指針**そのものである。