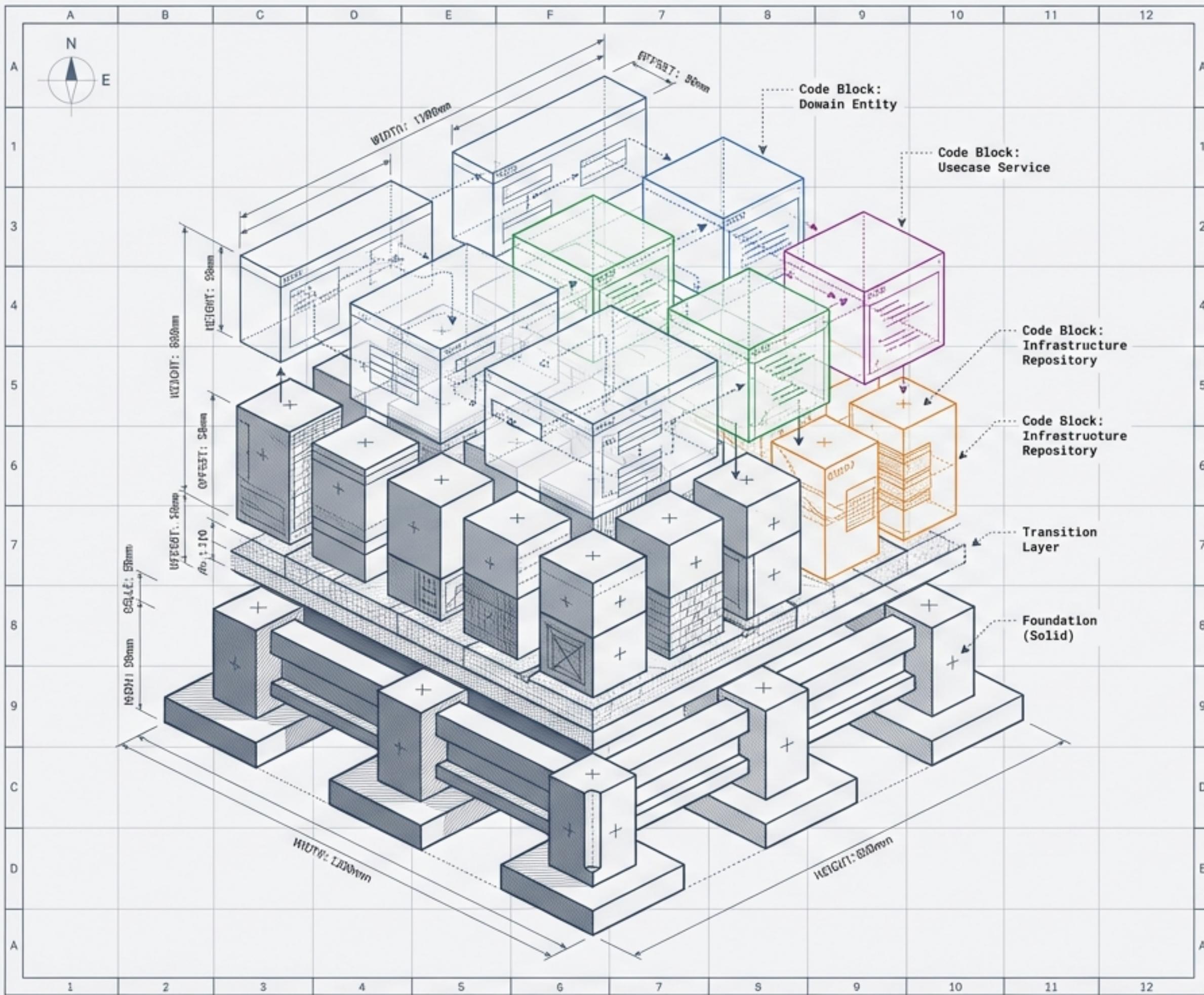




DDD AI Sample

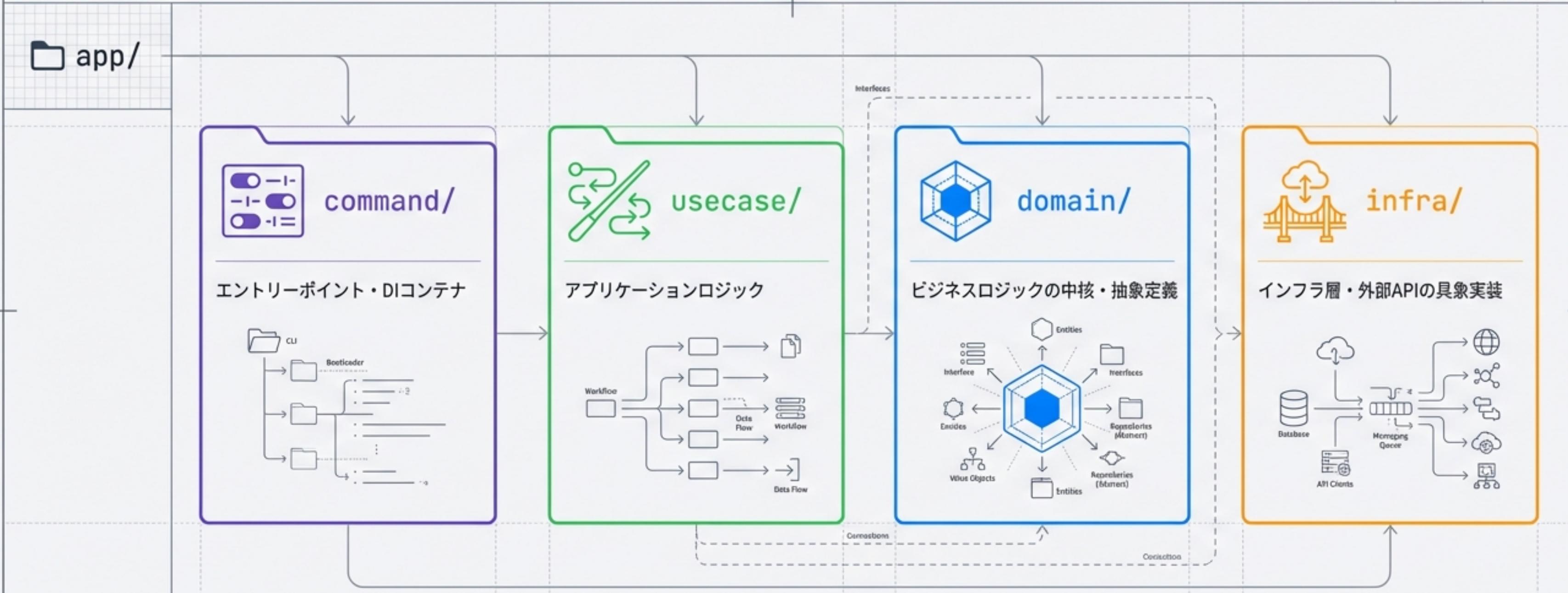
詳細設計ガイド

アーキテクチャ、依存ルール、
実装パターンの徹底解説



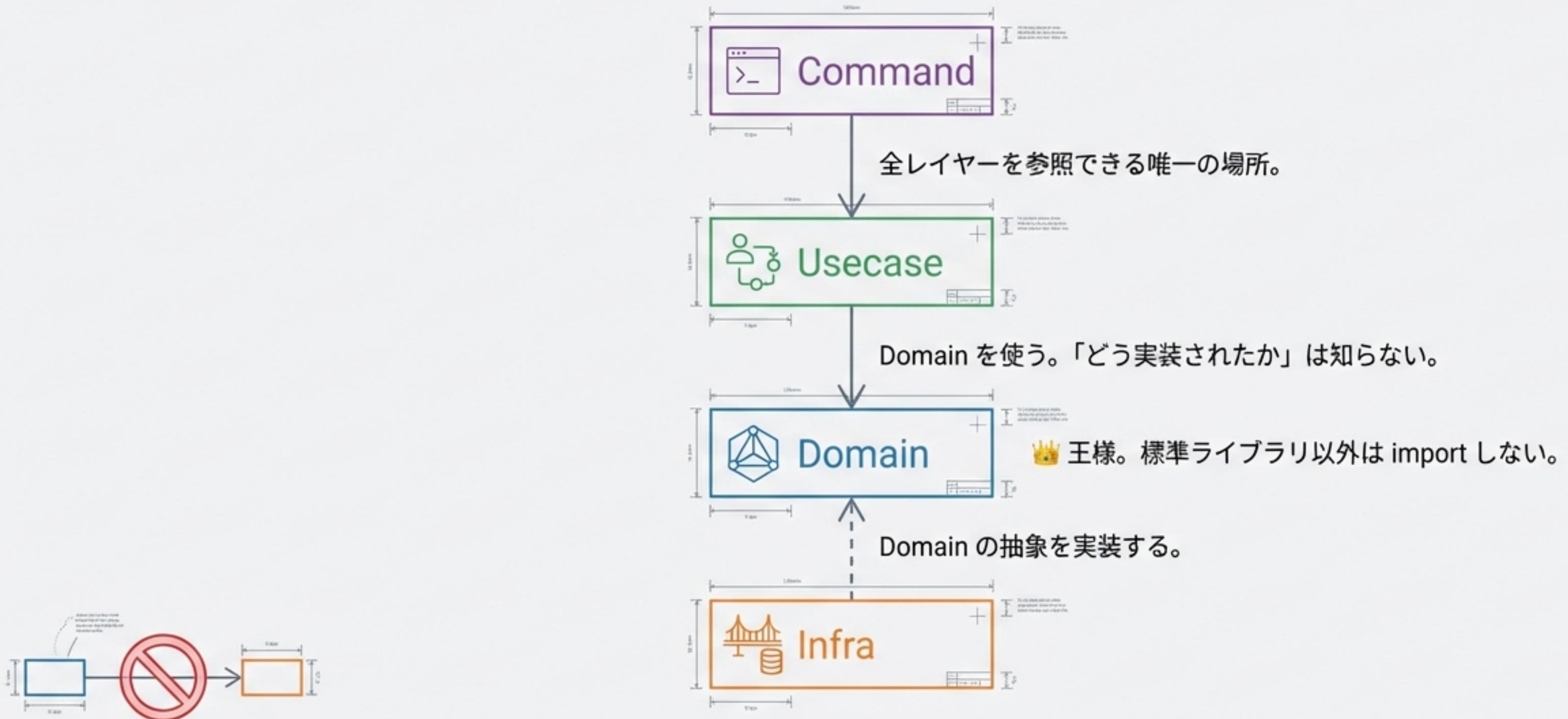
	目的 (OBJECTIVE)	プロジェクト参加者が「どこに何を書くのか」を迷わないための羅針盤。
	対象 (SCOPE)	レイヤー構造 / 依存性の注入(DI) / テスト戦略 / DTO配置

全体像：4つの主要ディレクトリ



コードを読むときは、このディレクトリ構造がそのままアーキテクチャの階層を表している。

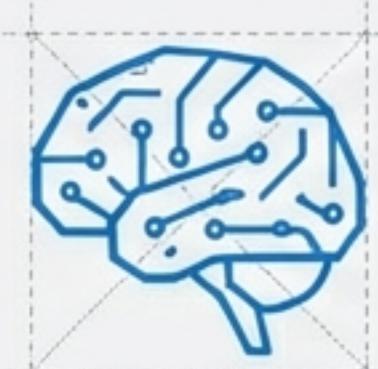
依存ルール：誰が誰を import できるか



「依存の方向を一方向に保つこと」が DDD の根幹である。

Domain 層：ビジネスロジックと抽象

純粋なビジネスロジック。外部サービスへの直接依存は禁止。



Brain

DTO

(Data Transfer Objects)

- Prompt
- LLMResult
(Pydantic models)

抽象インターフェース

- LLMClientBase
(ABC - Abstract Base Class)

ドメインロジック

- QuestionAnswerPrompt
(純粋な関数やクラス)

```
app/domain/llm_client/llm_client_base.py
```

1 from abc import ABC, abstractmethod
2
3 class LLMClientBase(ABC):
4 @abstractmethod ← 契約のみ定義
5 def predict(self, prompt: Prompt) -> LLMResult:
6 """
7 具体的なAPI名 (Gemini/OpenAI) は一切書かない
8 純粋な Python コードのみ
9 """
10 ...

Domain DTO vs ドメインロジック

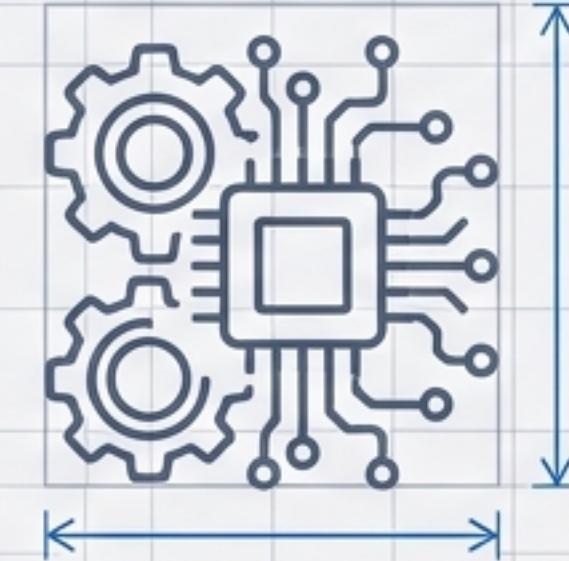
DTO (Data Transfer Object)



- Examples:
 - `Prompt`
 - `LLMResult`
- Characteristics:
 - `pydantic.BaseModel` を継承
 - イミュータブル (Immutable)
 - 「ビジネス概念」としてのデータ

```
class Prompt(BaseModel):  
    text: str
```

Logic (Prompt Builder)

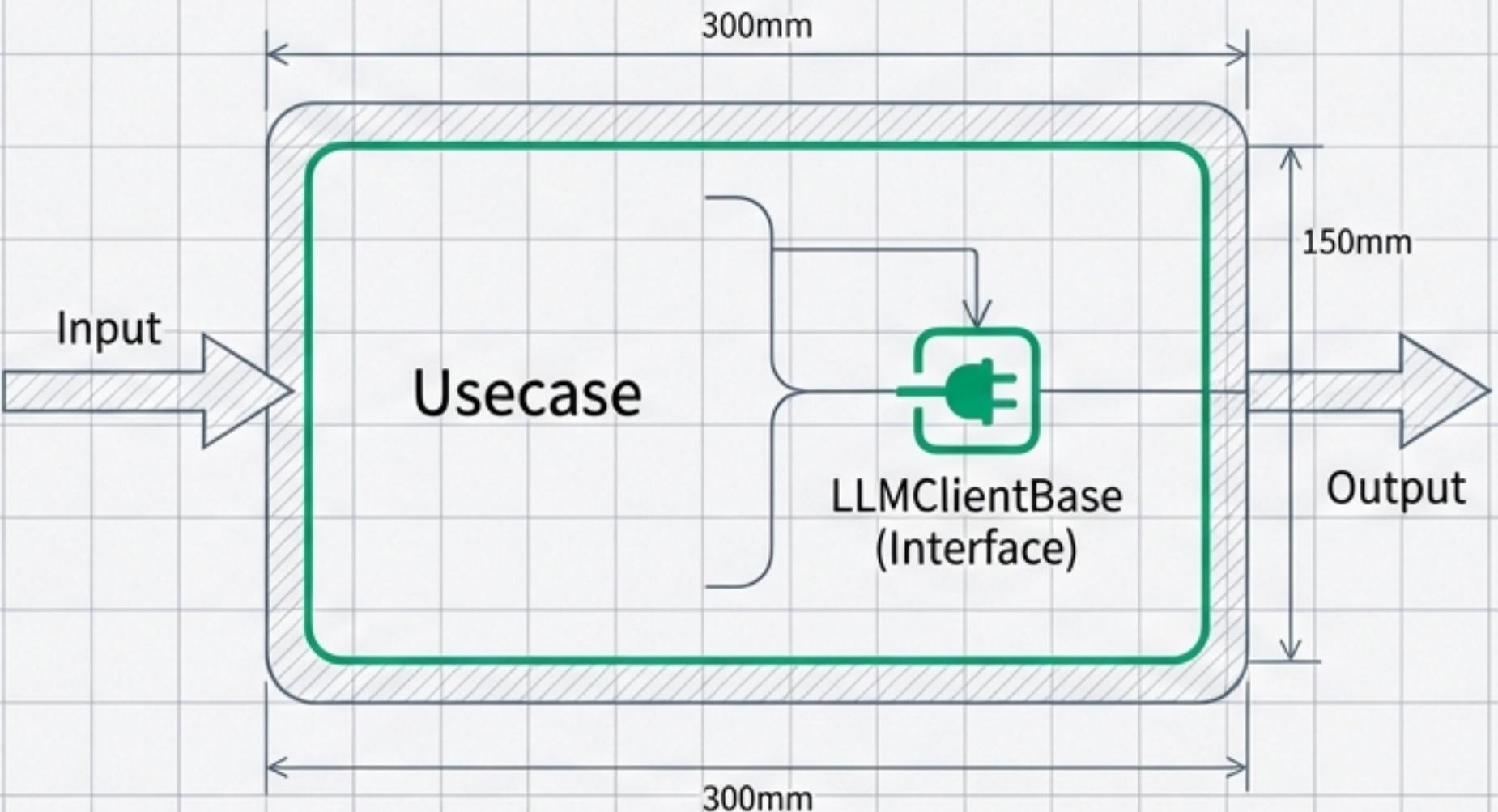


- Examples:
 - `QuestionAnswerPrompt`
- Characteristics:
 - 文字列操作や計算を行う
 - 特定のAPIには依存しない

“「データ」と「振る舞い」を明確に分ける”

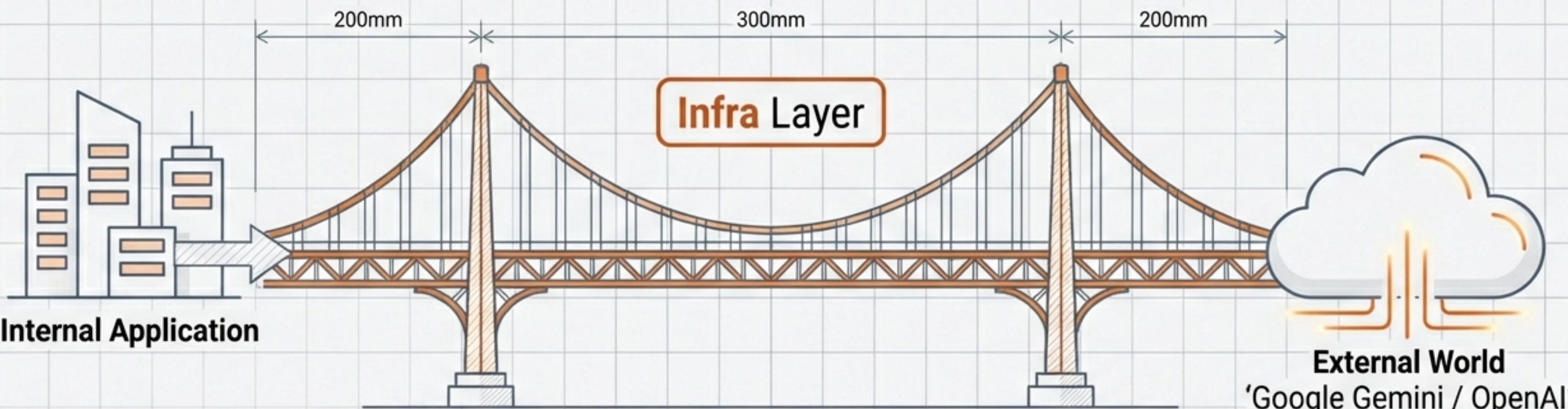
Usecase 層：アプリケーションの指揮者

- 役割 (Role)
 - Domain 層のオブジェクトを組み合わせて処理フローを実行する。
- DIP (Dependency Inversion)
 - コンストラクタで「抽象」を受け取る。具象クラス (Gemini等) は知らない。
- 専用 DTO
 - `QuestionAnswerResponse` (Domain の `LLMResult` とは区別する)



```
app/usecase/question_answer_usecase.py
+
class QuestionAnswerUsecase:
    # 抽象(LLMClientBase)を受け取る。GeminiかOpenAIかは知らない。
    def __init__(self, llm_client: LLMClientBase) -> None:
        self.llm_client = llm_client
```

Infra 層：外部サービスの具象実装



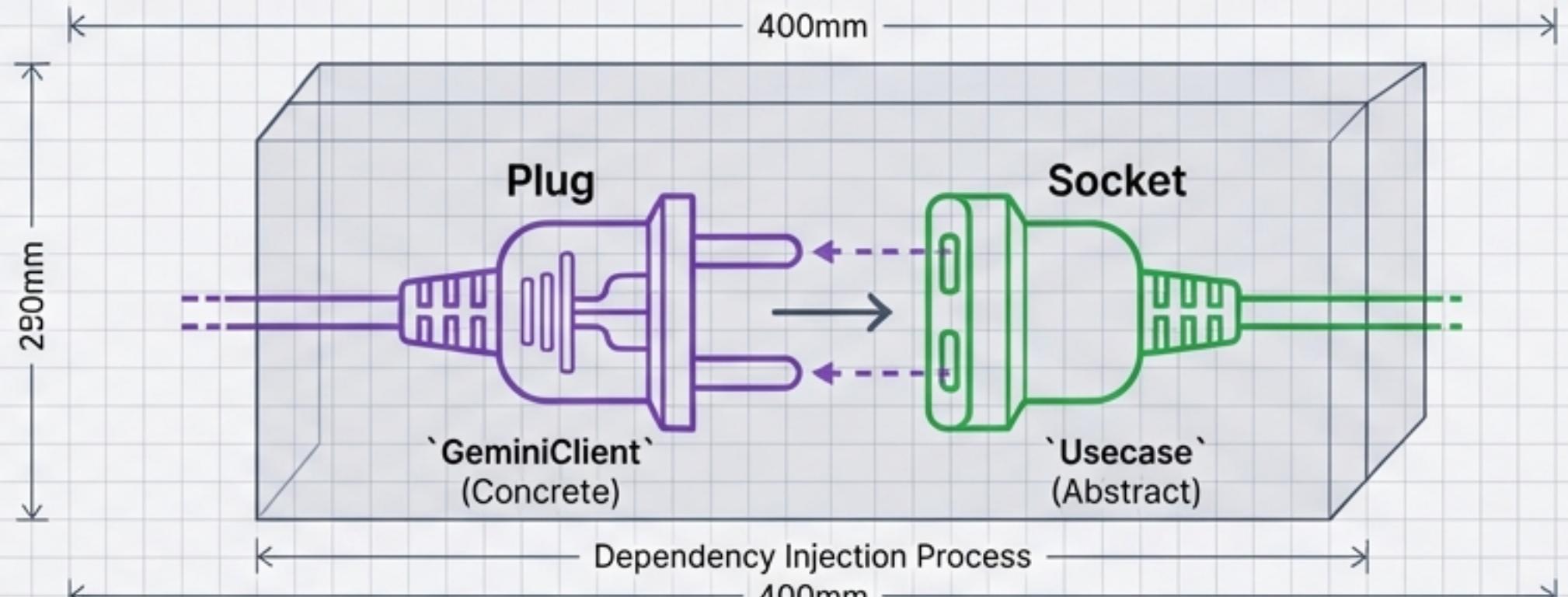
- **役割:** Domain 層で定義した契約 (`LLMClientBase`) を、実際の SDK を使って実装する。
- **カプセル化:** `google-genai`などの外部ライブラリは、この層に閉じ込める。
- **拡張性:** OpenAI に変更したい場合は、ここで `OpenAIClient` を作るだけ。Usecase は変更不要。

```
app/infra/llm_client/gemini.py
+
class GeminiClient(LLMClientBase): # 抽象を継承
    def predict(self, prompt: Prompt) -> LLMResult:
        # google.genai の処理はここだけ記述する
        ...
+
```

Command 層：エントリーポイントと依存性の注入

キーコード

- 特権 (Privilege): `infra` と `usecase` の両方を import できる唯一の層。
- DI (Dependency Injection): 「どの具象クラスを使うか」を決定し、Usecase に渡す。



コードスニペット

app/command/question.py

JetBrains Mono

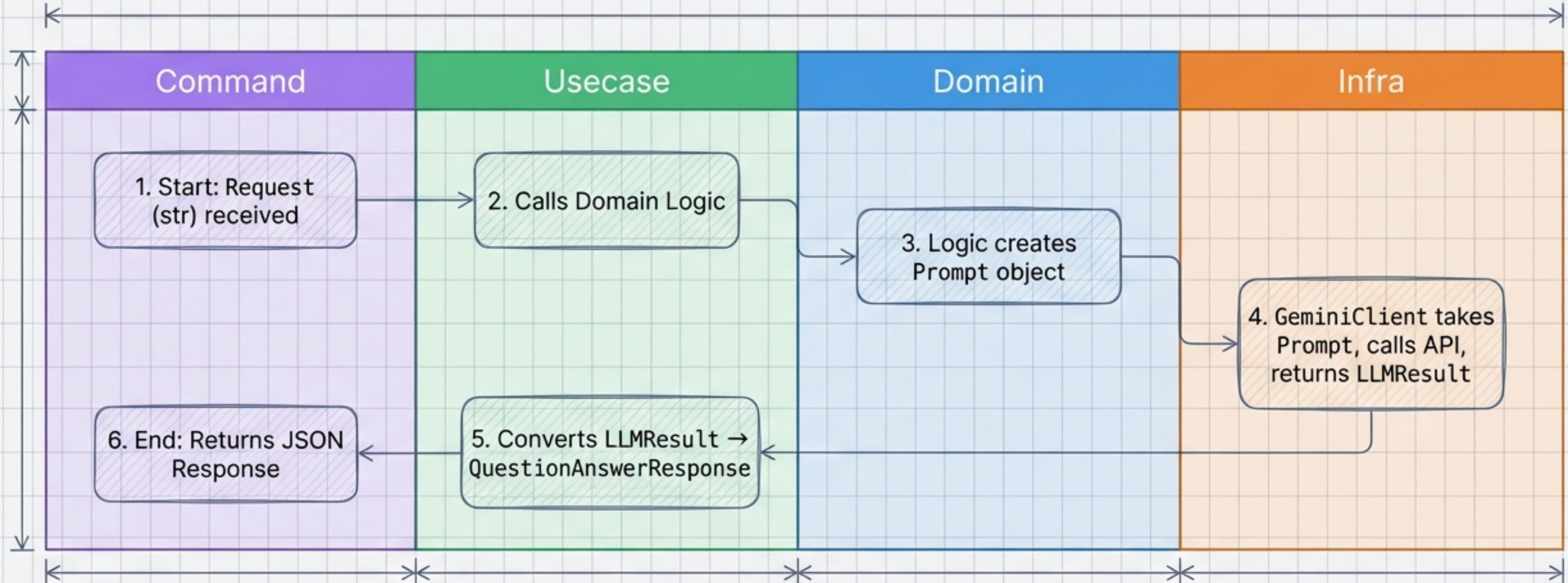
```
# app/command/question.py
```

```
# 1. 具象クラス(Infra)を生成  
client = GeminiClient(api_key="...") ← ここで具象を選択
```

```
# 2. Usecaseに注入(Injection)  
usecase = QuestionAnswerUsecase(llm_client=client)
```

```
# 3. 実行  
usecase.execute(request.question)
```

データの流れ：リクエストからレスポンスまで



Insight

データはレイヤーの境界を越えるたびに、そのレイヤーに適した形 (DTO) に変換される。

DTO マップ：データの所属と役割

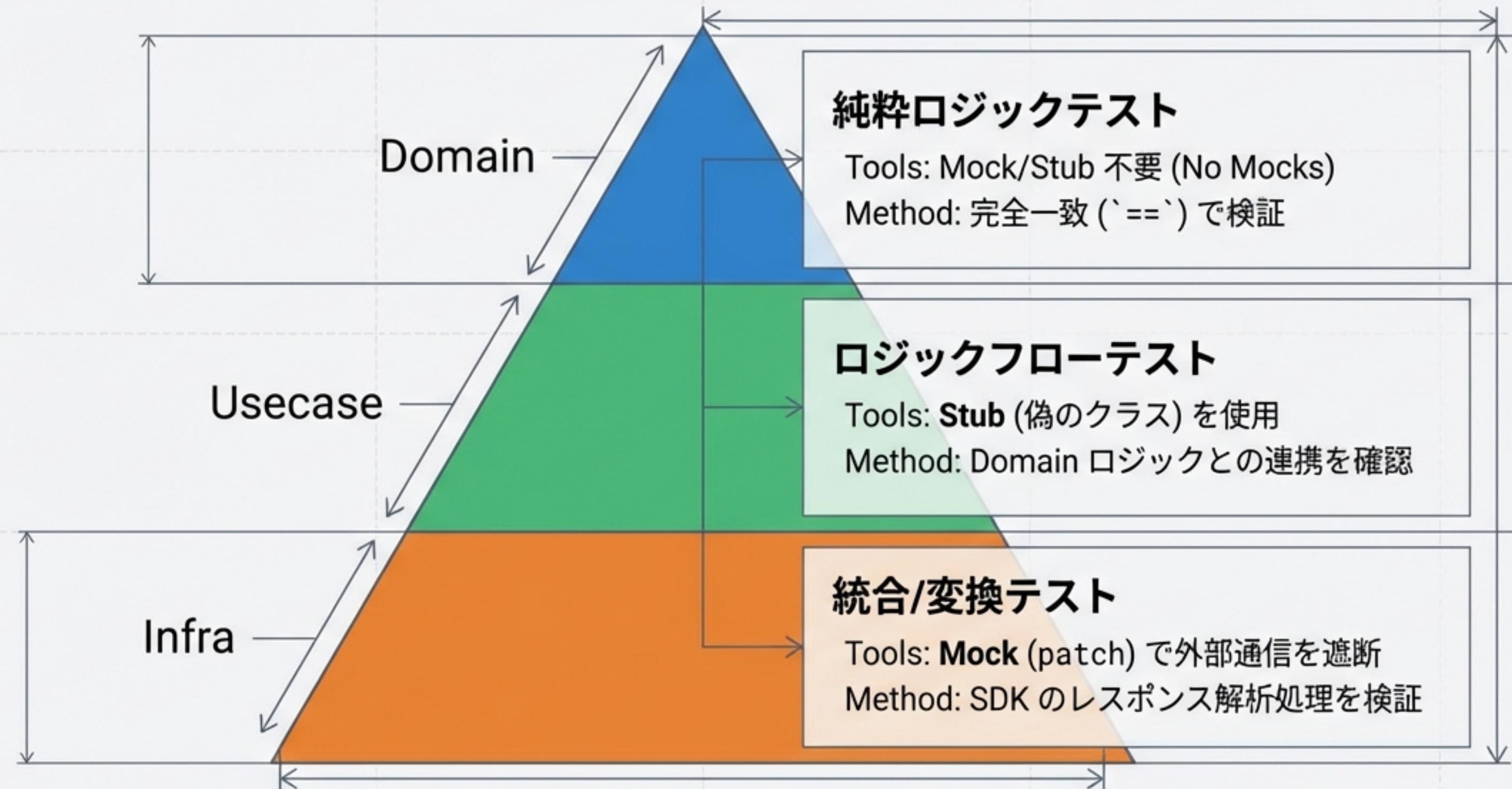
DTO Name	Location (Layer)	Role	User
Request / Response	command	API/CLI 用の入出力	
QuestionAnswerResponse	usecase	ユースケースの実行結果	
Prompt / LLMResult	domain	ビジネス上の LLM 入出力概念	

判断基準

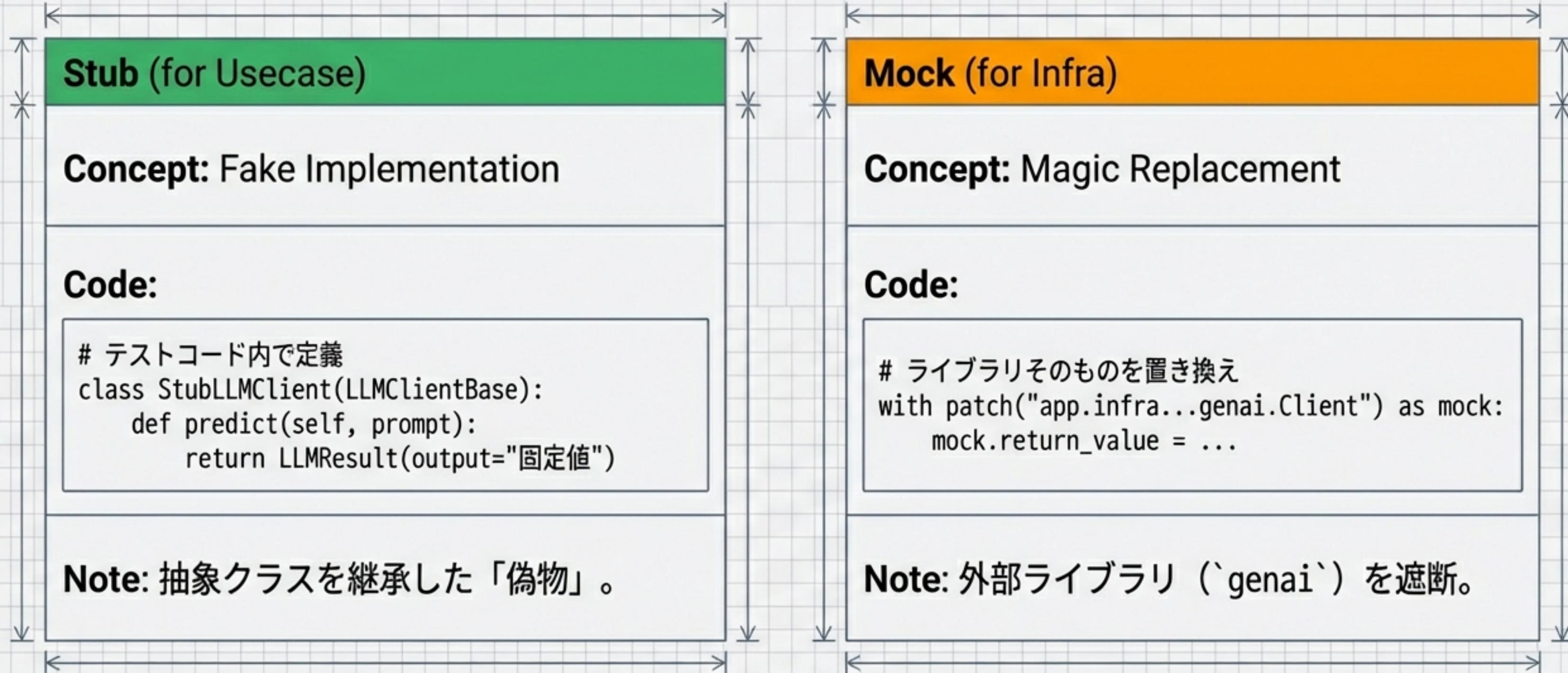
「誰のためのデータか？」

外部 API の都合なら Command。ビジネスロジックなら Domain。

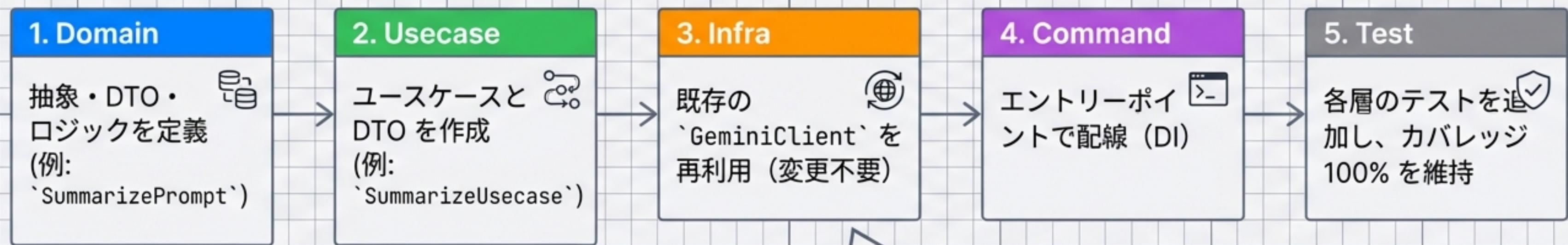
テスト戦略：レイヤー別のアプローチ



テスト戦術：Stub vs Mock



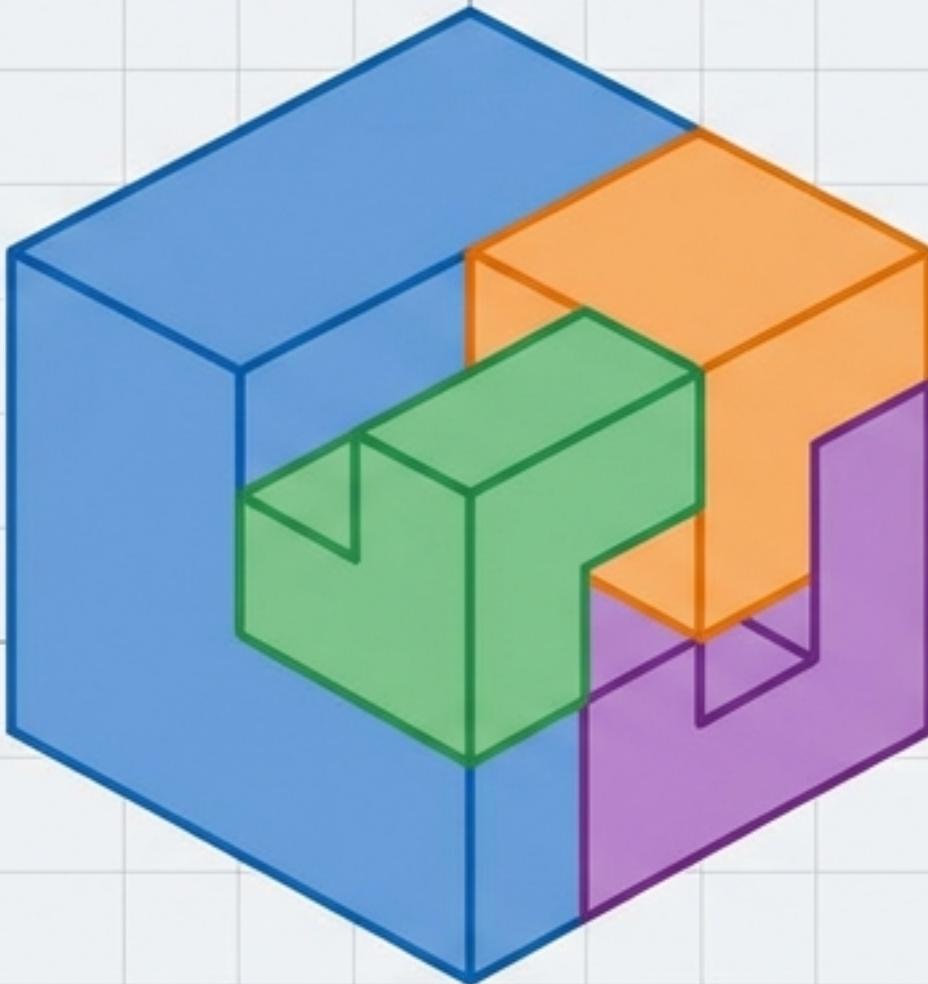
新機能追加のレシピ



Do's and Don'ts チェックリスト

	Domain 層で app.usecase を import している
	Usecase 層で app.infra を import している
	テストで実際の API を叩いている
	新規モジュールのカバレッジは 100%
	文字列検証は「部分一致」ではなく「完全一致」

結論：複雑さを制御する



DDD のルールは一見、厳格に見える。

しかし、関心事を分離することで、「外部APIの変更」や「仕様変更」に強いコードベースが育つ。

Next Step: リポジトリを `clone` し、`tests/` を実行して構造を体感してください。