PHPではじめる CQRSっぽいやつ

@dnskimox

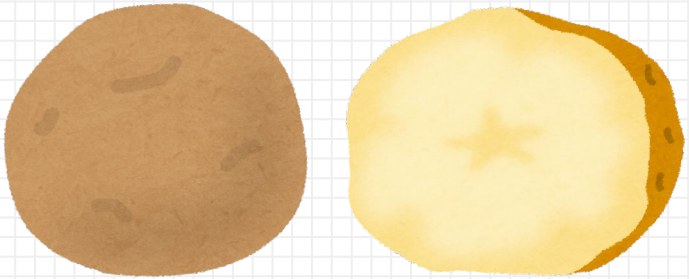
自己紹介

- HN：男爵
- dnskimo
- dnskimox
- ソフトウェアエンジニア
- 北海道在住

- アルプ株式会社
- サブスク管理SaaS開発

Alp, Inc.

Scalebase



今日話すこと

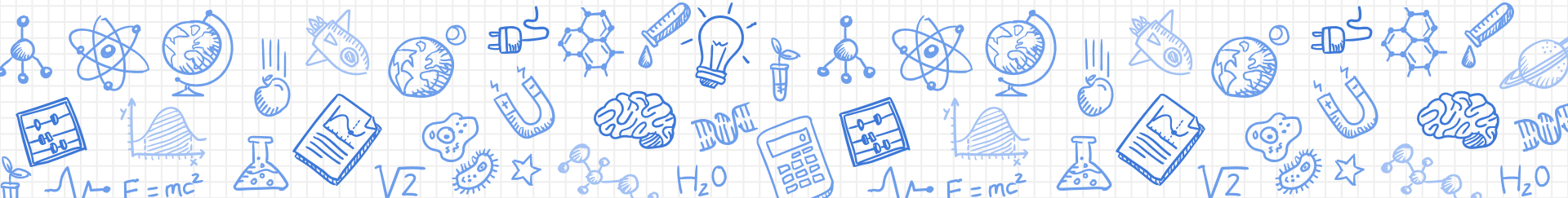
- ✘ コマンドクエリ分離原則（CQS）
- ✘ コマンドクエリ責務分離（CQRS）
- ✘ PHPでCQRSっぽいことを実践してみた

今日話さないこと

- ✘ イベントソーシングと結果整合性
- ✘ 更新用DBと参照用DBの分離と同期
- ✘ RDB以外のミドルウェアを使った事例

コマンドクエリ分離原則

命令と問い合わせを分離する



バートランド・メイヤーのコマンドクエリ分離原則（CQS）

- クラス設計の原則
- あらゆるクラスの特性は**コマンド（命令）**と**クエリ（問い合わせ）**に分けられる
- 両者を明確に区別することで、単純で読みやすいソフトウェアを作り出し、**信頼性**、**再利用性**、**拡張性**を飛躍的に向上させることができる

https://en.wikipedia.org/wiki/Bertrand_Meyer

コマンドクエリ分離原則（CQS）

コマンド

- ✘ オブジェクトの状態を書き換える
- ✘ 返り値をもたない

クエリ

- ✘ オブジェクトに関する情報を返す
- ✘ **副作用**をもたらしてはならない

副作用（side effect）＝　クエリにおいて、「問い合わせに答える」という本来の目的に付随する変更

副作用（side effect）≒ オブジェクトの状態の変更や、システムの外界に与えられる影響

```php
<?php

$character = new Character(...);

$character->getLevel(); // 1    クエリ？

if ($character->canLevelUp()) {  クエリ？
        echo “This character can level up!”;
}

$character->getLevel(); // 2        いつの間にか回答が変化している！
```

質問をすることで回答を変化させてはならない

```php
<?php
class Character
{
    private $level = 1;
    private $exp = 0;

    // コマンドの例
    public function gainExp(int $exp): void
    {
        assert($exp > 0);
        $this->exp += $exp;

        // 経験値100毎にレベルアップ
        while ($this->level < floor($this->exp / 100) + 1) {
            $this->level++;
        }
    }
}
```

```php
<?php
class Character …

    private $level = 1;
    private $exp = 0;

    // クエリの例
    public function getLevel(): int
    {
        return $this->level;
    }

    // クエリの例
    public function getExpForNextLevel(): int
    {
        return 100 - $this->exp % 100;
    }
}
```

```php
<?php

$character = new Character(...);

$before_level = $character->getLevel();  クエリ
$character->gainExp(100);コマンド

if ($before_level < $character->getLevel()) {クエリ
        printf(“Level up! Next exp is %d”, $character->getExpForNextLevel());
}
```

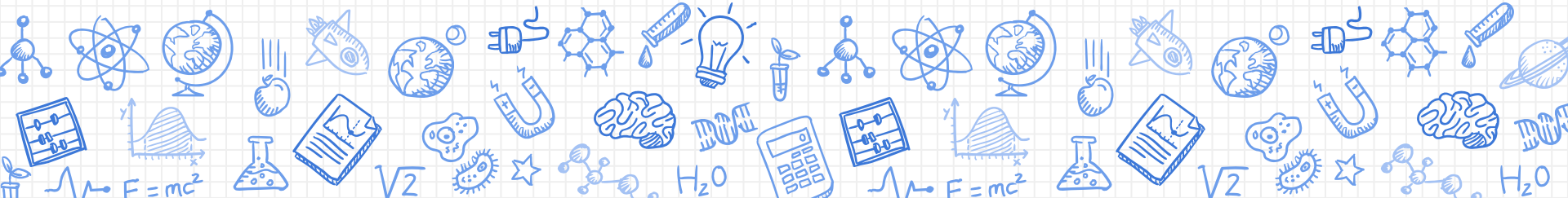
クエリの性質

- ✘ クラスの不変条件を壊す心配がない（信頼性）
- ✘ 副作用がないので様々な用途に使える（再利用性）
- ✘ 状態変化による複雑さがないので、変更を加えやすい（拡張性）

副作用をコマンドに局所化し、クエリの世界を広げることにより、ソフトウェアの**信頼性・再利用性・拡張性**を向上させることができる

CQRSとはなにか？

アーキテクチャレベルの視点



グレッグ・ヤングのコマンドクエリ責務分離（CQRS）

- ✘ コマンドクエリ分離原則に基づいたアーキテクチャパターン
- ✘ システムのユースケースは**コマンド（更新）**と**クエリ（参照）**に分類できる
- ✘ コマンドとクエリにはそれぞれ**非常に異なるニーズ**があるので分離すべき

https://www.developerfusion.com/event/153843/special-guest-greg-young-on-tue-aug-13th/

WEBアプリケーションにおけるコマンド・クエリの分類

コマンド

- ✘ レコードの**追加・更新・削除**
- ✘ HTTP: PUT/POST/DELETE

クエリ

- ✘ レコードの**参照・集計・検索**
- ✘ HTTP: GET

異なるニーズ：一貫性

コマンド

- ✘ トランザクション処理によるアトミックな実行
- ✘ データを適切にロックする必要がある

クエリ

- ✘ データの不整合を起こす心配がない

異なるニーズ：データストレージ

コマンド

- ✘ **第3正規形**のテーブルを使うことが多い
- ✘ マスターDBを参照

クエリ

- ✘ **非正規化**したデータのほうが都合が良いが場合がある
- ✘ スレーブDBを参照

異なるニーズ：スケーラビリティ

コマンド

- ✘ 一般的にリクエストの割合が低い
- ✘ スケーラビリティは必ずしも重要ではない

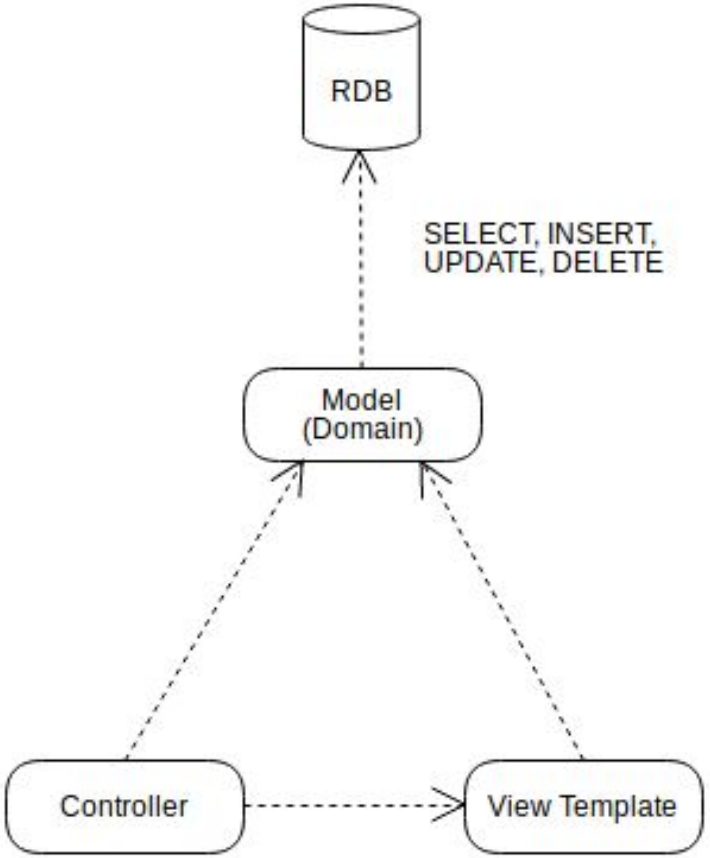
クエリ

- ✘ 一般的にリクエストの割合が高い
- ✘ スケーラビリティが非常に重要

伝統的なMVCアーキテクチャについて考える

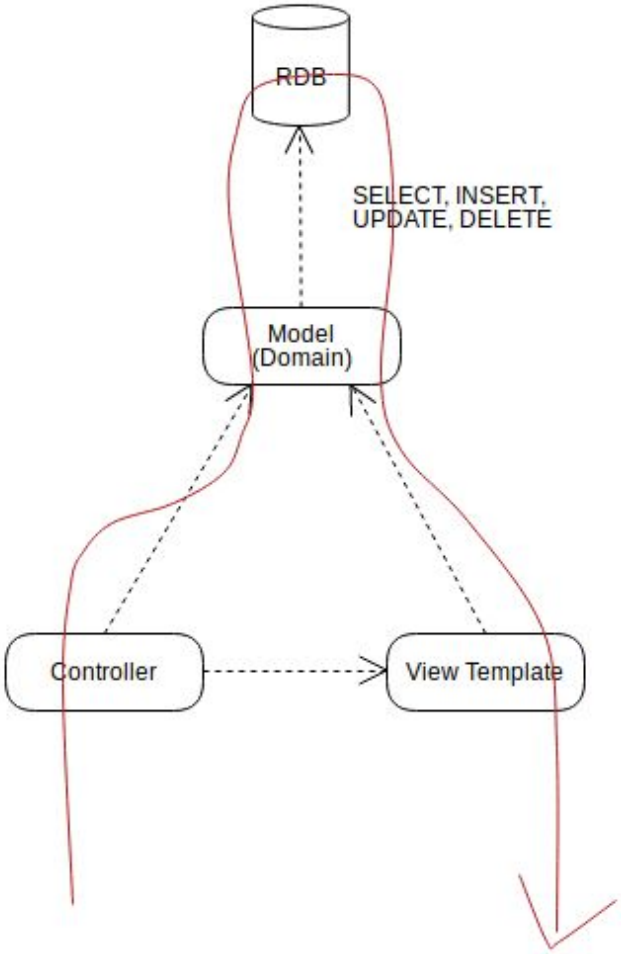
伝統的なMVCアーキテクチャの例

- ✘ **関心事**によってコードを複数のモジュールに分ける
- ✘ **Model**はDBアクセスと**ドメインロジック**を担当する（例：ActiveRecordパターン、DataMapperパターン）
- ✘ **View**はHTMLやJSON等、レスポンスの表現を定義する
- ✘ **Controller**はリクエストを読み取り、ModelとViewの助けを借りてレスポンスを作る

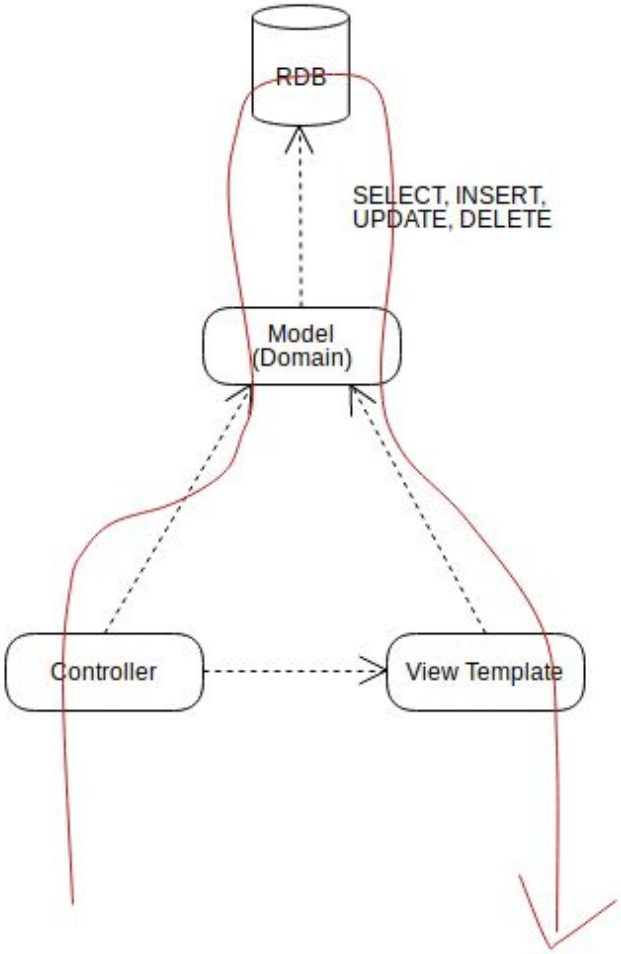


伝統的なMVCアーキテクチャ

何が問題か？



コマンドの大まかな流れ


RDB
SELECT, INSERT,
UPDATE, DELETE
Model
(Domain)
Controller
View Template

クエリの大まかな流れ

全く異なるニーズを持つ処理が同じ
モジュールで処理されている！

コマンドの実装で困ること

- ✘ Modelクラスに複雑な検索やページネーション等、参照のためのロジックが入り込んで**肥大化**する

クエリの実装で困ること

- ✘ ORMの検索メソッドで最適なSQLを発行するのは非常に困難
- ✘ 非正規形データはModelのオブジェクト構造と一致しない（**インピーダンスミスマッチ**）
- ✘ **N+1クエリ問題**への配慮が必要

本来クエリの実装は単純なはずでは？？

OOPの原則

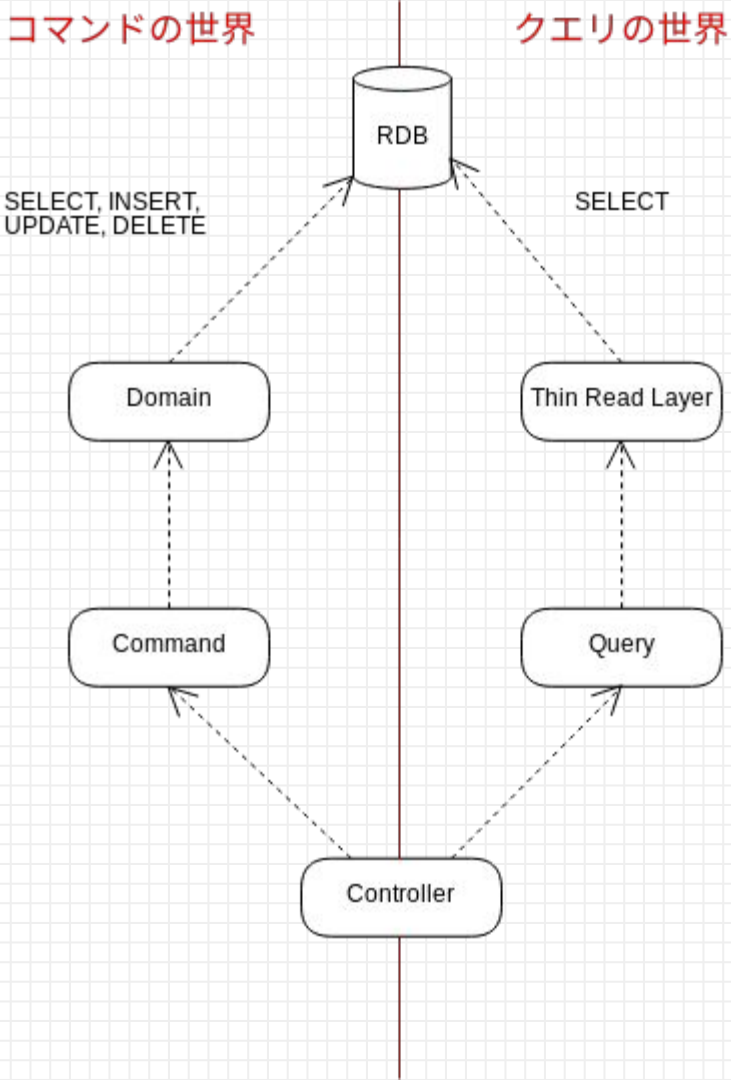
関心を分離せよ

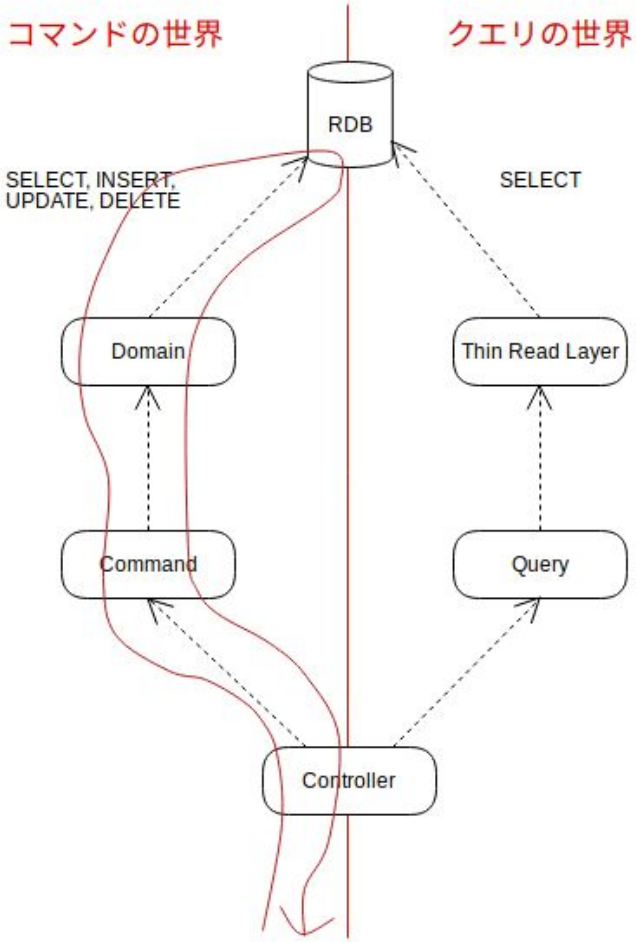
コマンドの責務を持つモジュール
と、クエリの責務を持つモジュール
に分離する

CQRS (Command Query Responsibility Segregation = コマンドクエリ責務分離)

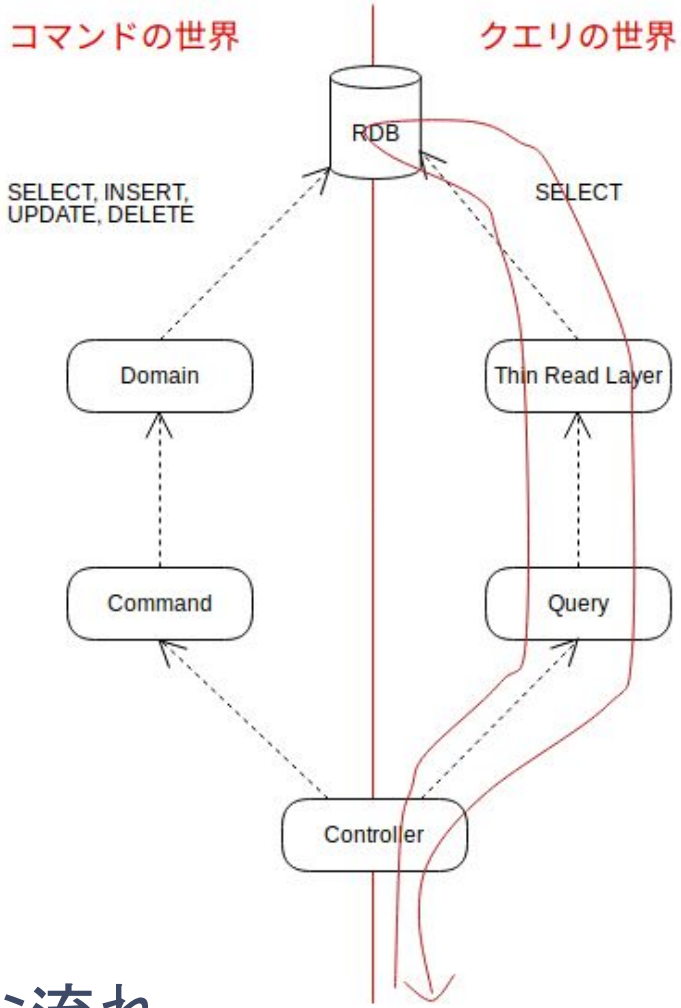
注意

これから話す内容は
”正式な”CQRSとは異なります


コマンドの世界
クエリの世界
RDB
SELECT, INSERT,
UPDATE, DELETE
SELECT
Domain
Thin Read Layer
Command
Query
Controller



コマンドの大まかな流れ


コマンドの世界
クエリの世界
RDB
SELECT, INSERT,
UPDATE, DELETE
SELECT
Domain
Thin Read Layer
Command
Query
Controller

クエリの大まかな流れ

クエリ側はドメインレイヤーを迂回する

そもそも何故ドメインレイヤーが必要か？

ドメインロジックを書くため

ドメインロジックの例

- ✘ キャラクターは経験値100毎にレベルが1上がる
- ✘ キャラクターは装備を10個まで装着できる
- ✘ キャラクターがモンスターを攻撃した際のダメージの計算式は以下である

  モンスターの防御力 -（レベル × 10 + 武器の攻撃力）
- ✘ キャラクターのIDの表示フォーマットはPXXXXX（0埋め5桁）である

ドメインロジック ＝
ソフトウェアが扱う問題領域に固有のロジック

これはコマンドとクエリどちらの話か？

ドメインロジックの例

- ✘ キャラクターは経験値100毎にレベルが1上がる　コマンド
- ✘ キャラクターは装備を10個まで装着できる　コマンド
- ✘ キャラクターがモンスターを攻撃した際のダメージの計算式は以下である　コマンド

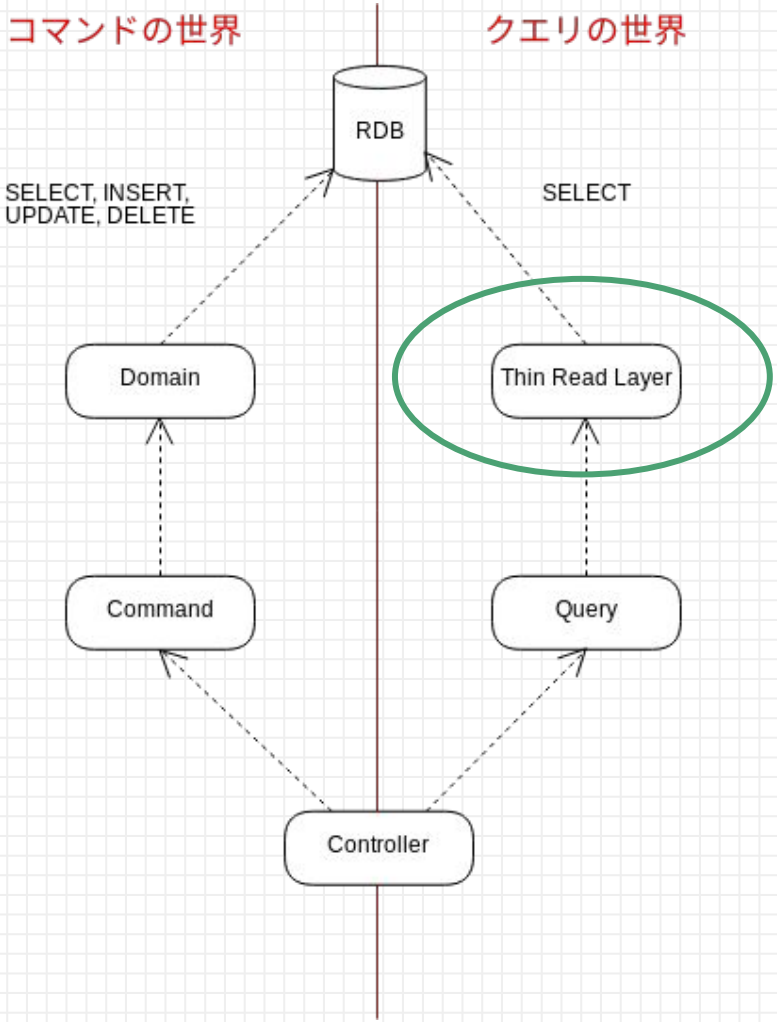
  モンスターの防御力 -（レベル × 10 + 武器の攻撃力）
- ✘ キャラクターのIDの表示フォーマットはPXXXXX（0埋め5桁）である　クエリ

（仮説）複雑なドメインロジックはコマンド側に多く存在する

（仮説）クエリ側はドメインレイヤーを実装する
メリットが少ない

代わりに何を実装するか？

Thin Read Layer（薄い読み取り層）


コマンドの世界
クエリの世界
RDB
SELECT, INSERT,
UPDATE, DELETE
SELECT
Domain
Thin Read Layer
Command
Query
Controller

薄い読み取り層とは？

- ✘ **DBに直接依存**するレイヤー（必要ならDBベンダーと結びついても良い）
- ✘ **小さな規約**をベースとしたマッピングユーティリティ
- ✘ ORMほどの機能性は要らない

CQRSを適用すると何が起きるか？

コマンドの実装はこうなる

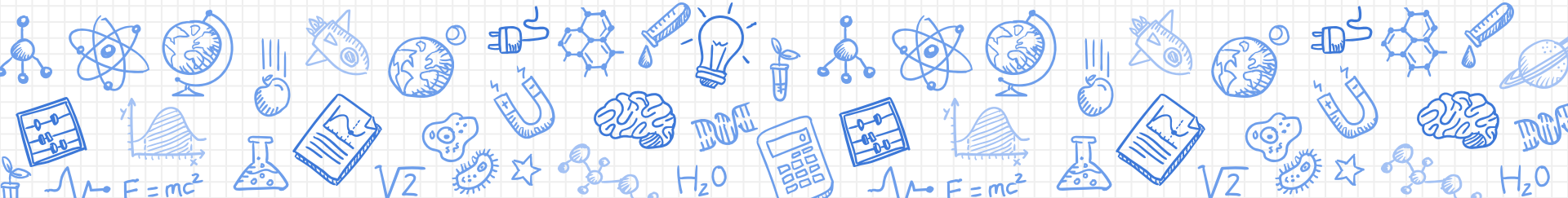
- ✘ ドメインレイヤーは更新系の複雑なドメインロジックに集中できる

クエリの実装はこうなる

- ✘ ドメインモデルに捕らわれること無く、欲しい情報を取得するための最適な方法を選択できる
- ✘ 副作用のない参照処理を扱うだけのシンプルなコードにできる

PHPでCQRSっぽいものを実践する

要点を踏まえて手軽な方法を考える

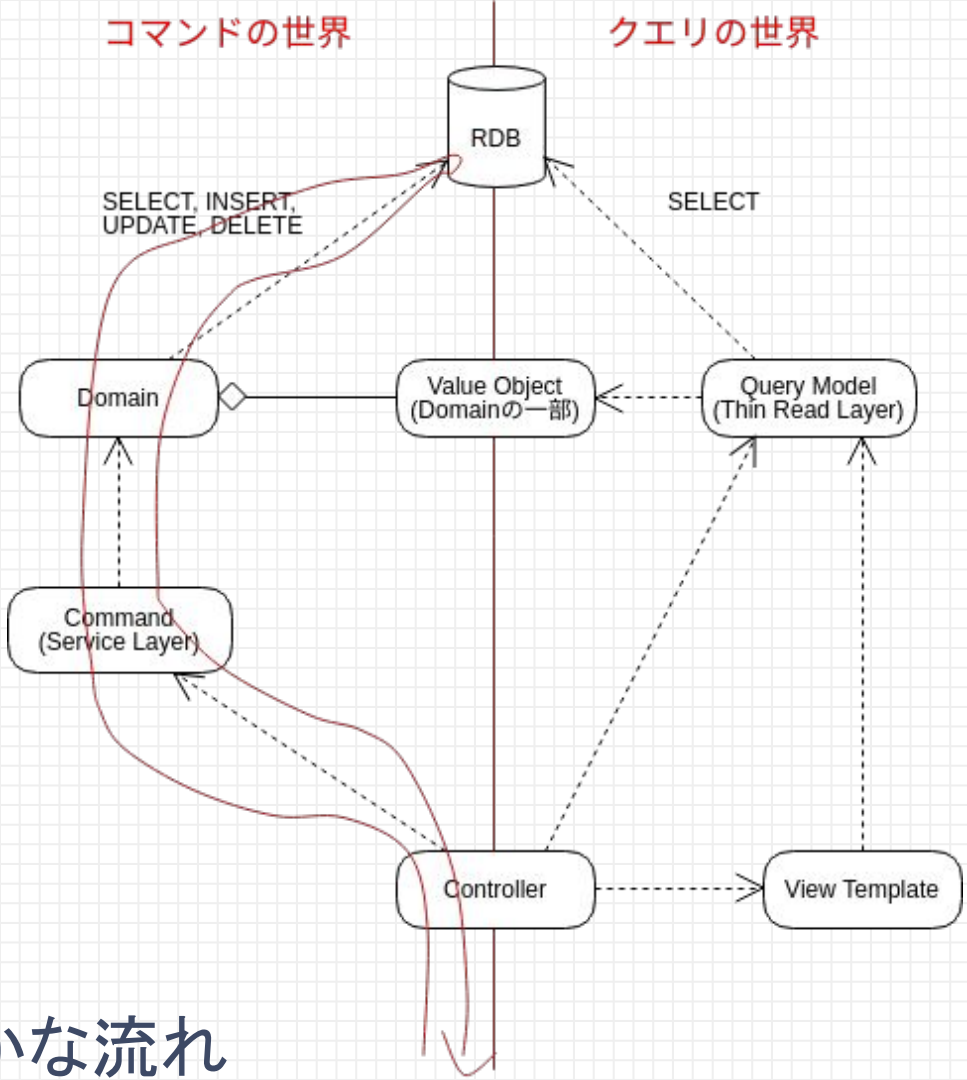


どのように実践するか？

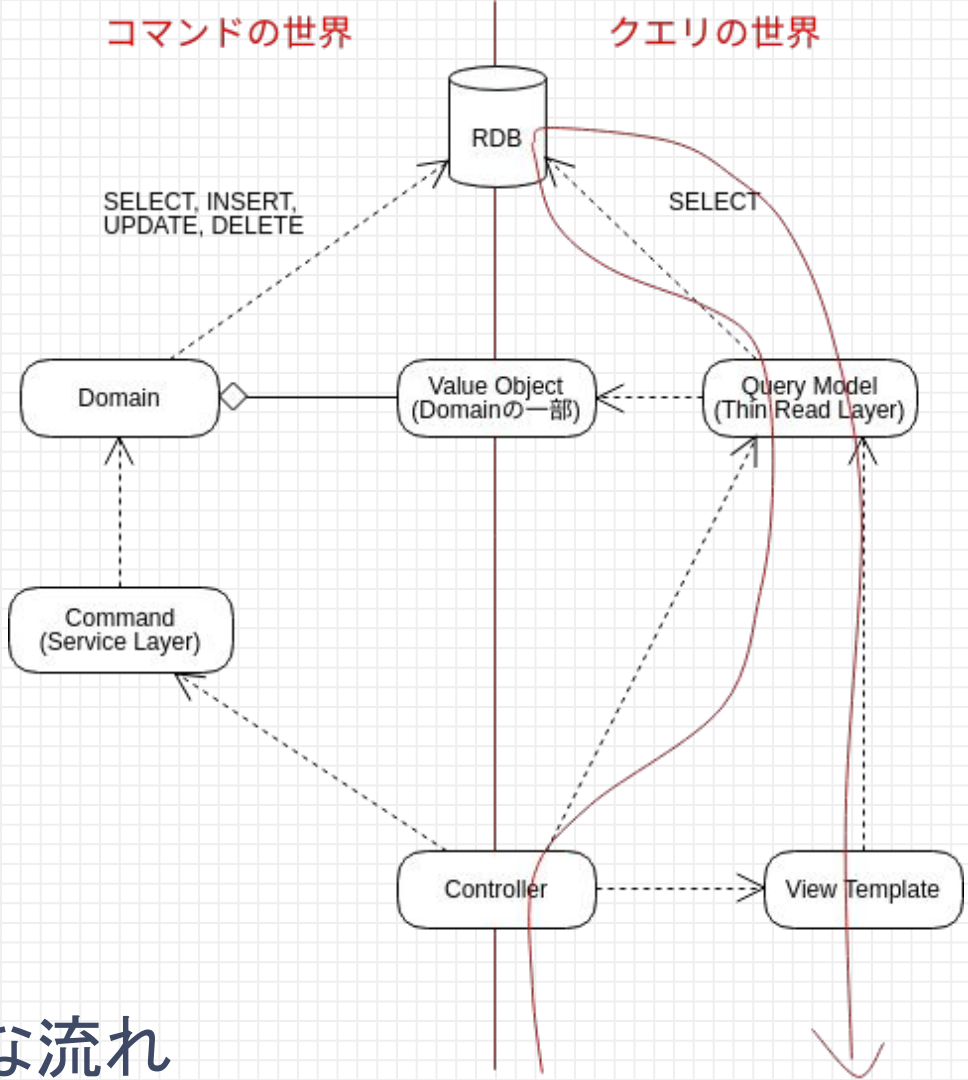
1. 伝統的なMVCを**変形**させる
2. **SQL**が持つ本来の力を引き出す
3. **Value Object**を使ってドメインロジックを共有する

1. 伝統的なMVCを変形させる


コマンドの世界
クエリの世界
RDB
SELECT, INSERT,
UPDATE, DELETE
SELECT
Domain
Value Object
(Domainの一部)
Query Model
(Thin Read Layer)
Command
(Service Layer)
Controller
View Template



コマンドの大まかな流れ


コマンドの世界
クエリの世界
RDB
SELECT, INSERT,
UPDATE, DELETE
SELECT
Domain
Value Object
(Domainの一部)
Query Model
(Thin Read Layer)
Command
(Service Layer)
Controller
View Template

クエリの大まかな流れ

クエリはORM（ActiveRecord）を迂回する

ActiveRecordやDataMapperがDBのレコードをオブジェクトにマッピングするのは何のためか？

Domain Modelパターンを実装するため

Domain Modelパターン

- ✘ ドメインロジックを記述する設計パターンの一つ
- ✘ データとドメインロジックの両方を持つオブジェクトを設計する

Transaction Scriptパターン

- ✘ ドメインロジックを記述する設計パターンの一つ
- ✘ ロジックを持たないオブジェクトを使って、手続き的にドメインロジックを記述する
- ✘ ドメインモデル貧血症

```php
<?php

class GainExpService
{
    public function execute(...)
    {
        $character = Character::findById(123);

        $character->exp += 100;
        while ($character->level < floor($character>exp / 100) + 1) {
            $character>level++;
        }
    }
}
```

OOPの原則

データとロジックを一体化する

ORMを使ってDomain Modelパターンを実装することで、OOPの力が最大化される

BUT

ORMを使うとSQLの力が制限される

2. SQLが持つ本来の力を引き出す

SQLの特性

- ✘ **検索・集計**が得意
- ✘ 非正規形のデータを簡単に作れる
- ✘ 集合関数、CASE式等を使って「何が欲しいか？」を**宣言的**に記述できる
- ✘ パフォーマンスチューニングしやすい（ORMに比べれば）

SQLは「集合」を扱う上で強力な言語である

とはいえ

SQLの結果を連想配列で扱うのは避けたい

SQLの結果とPHPオブジェクトの単純なマッパー（Thin Read Layer）を用意する

QueryModel（クエリのためのModel）

- ✘ PDOStatementをオブジェクトにマッピングするだけ
- ✘ 定義されていないプロパティへのアクセスは禁止（**静的解析可能**）
- ✘ プロパティの書き換えは出来ない（**イミュータブル**）

```php
<?php

/**
 * @property int $character_id
 * @property string $name
 * @property int $friend_count
 */
class CharacterSummary extends QueryModel
{
    protected static $properties = [
        'character_id' => ['type' => 'type'],
        'name' => ['type' => 'string'],
        'friend_count' => ['type' => 'int'],
    ];
}
```

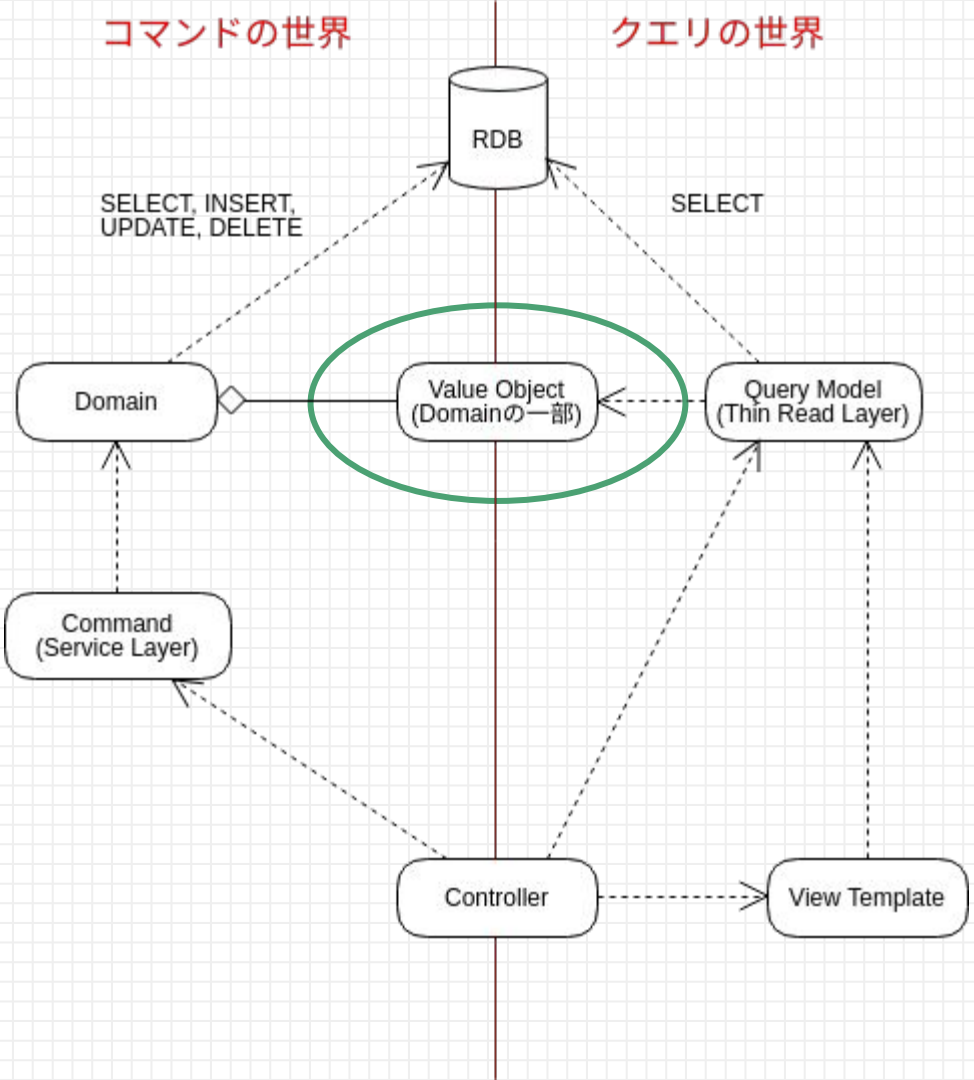
```php
<?php

class CharacterSummary...

    public static function findById(int $character_id): self
    {
        $result = QueryBuilder::table('character_tbl')
            ->select('character_tbl.character_id', 'name', 'COUNT(friend_id) AS friend_count')
            ->leftJoin('friend_tbl', 'USING', 'character_id')
            ->where('character_tbl.character_id', $character_id)
            ->group_by('character_id')
            ->get_one();

        return self::inflate($result);
    }
}
```

3. Value Objectを使ってドメインロジックを共有する


コマンドの世界
クエリの世界
RDB
SELECT, INSERT,
UPDATE, DELETE
SELECT
Domain
Value Object
(Domainの一部)
Query Model
(Thin Read Layer)
Command
(Service Layer)
Controller
View Template

Value Object（値オブジェクト）

- ✘ 開発者が定義した**ある種の値**を表すオブジェクト
- ✘ プリミティブなデータ型を独自に作るようなもの
- ✘ 全てのメソッドがクエリである（**イミュータブル**）
- ✘ 参照系のドメインロジックを持たせることが可能

Value Objectの例

- ✘ Money（通貨、金額）
- ✘ MyDate（年、月、日）
- ✘ Address（都道府県、市町村、番地...）
- ✘ Status（HP、MP、ATK、DEF、MATK...）
- ✘ **CharacterId**（数値）

ドメインロジックの例

- ✘ キャラクターは経験値100毎にレベルが1上がる
- ✘ キャラクターは装備を10個まで装着できる
- ✘ キャラクターがモンスターを攻撃した際のダメージの計算式は以下である

  モンスターの防御力 -（レベル × 10 + 武器の攻撃力）
- ✘ キャラクターのIDの表示フォーマットはPXXXXX（0埋め5桁）である

```php
<?php

class CharacterId
{
    private $character_id;

    public function __construct(int $character_id)
    {
        $this->character_id = $character_id;
    }

    public function formatted(): string
    {
        return sprintf("P%05d", $this->character_id);
    }
}
```

```php
<?php

class Character extends Model…

    public function getId(): CharacterId
    {
        return new CharacterId($this->character_id);
    }
}
```

```php
<?php

class CharacterSummary extends QueryModel...

    public function getId(): CharacterId
    {
        return new CharacterId($this->character_id);
    }
}
```

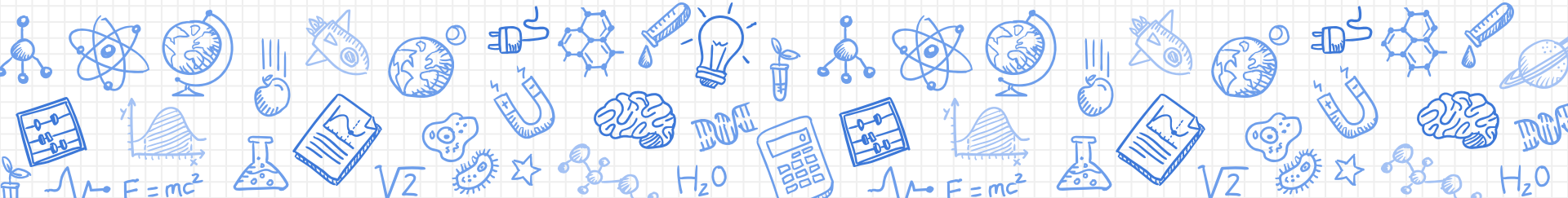
```php
<?php

$character = Character::findById(123);
$character->getId()->formatted(); // F00123

$character_summary = CharacterSummary::findById(123);
$character_summary->getId()->formatted(); // F00123
```

まとめ

PHPでCQRSっぽいやつのはじめかた



PHPでCQRSっぽいやつのはじめかた

- ✘ **コマンドクエリ分離原則**を意識してクラスを設計する
- ✘ アーキテクチャレベルで**更新と参照の責務を分離**する
- ✘ ドメインロジックを持つ**ValueObject**を見つける
- ✘ **SQL**本来の力を活用する

ご清聴ありがとうございました

参考資料

- ✘ Greg Young流CQRSの和訳版
- ✘ Greg Young流CQRS - Mark Nijhof
- ✘ 副作用を最小限に抑えるために必要なこと
- ✘ オブジェクト指向入門 第2版 方法論・実践
- ✘ Patterns of Enterprise Application Architecture
- ✘ 達人に学ぶSQL徹底指南書
- ✘ Domain Model- P of EAA Catalog
- ✘ Transaction Script - P of EAA Catalog
- ✘ ValueObject - Martin Fowler