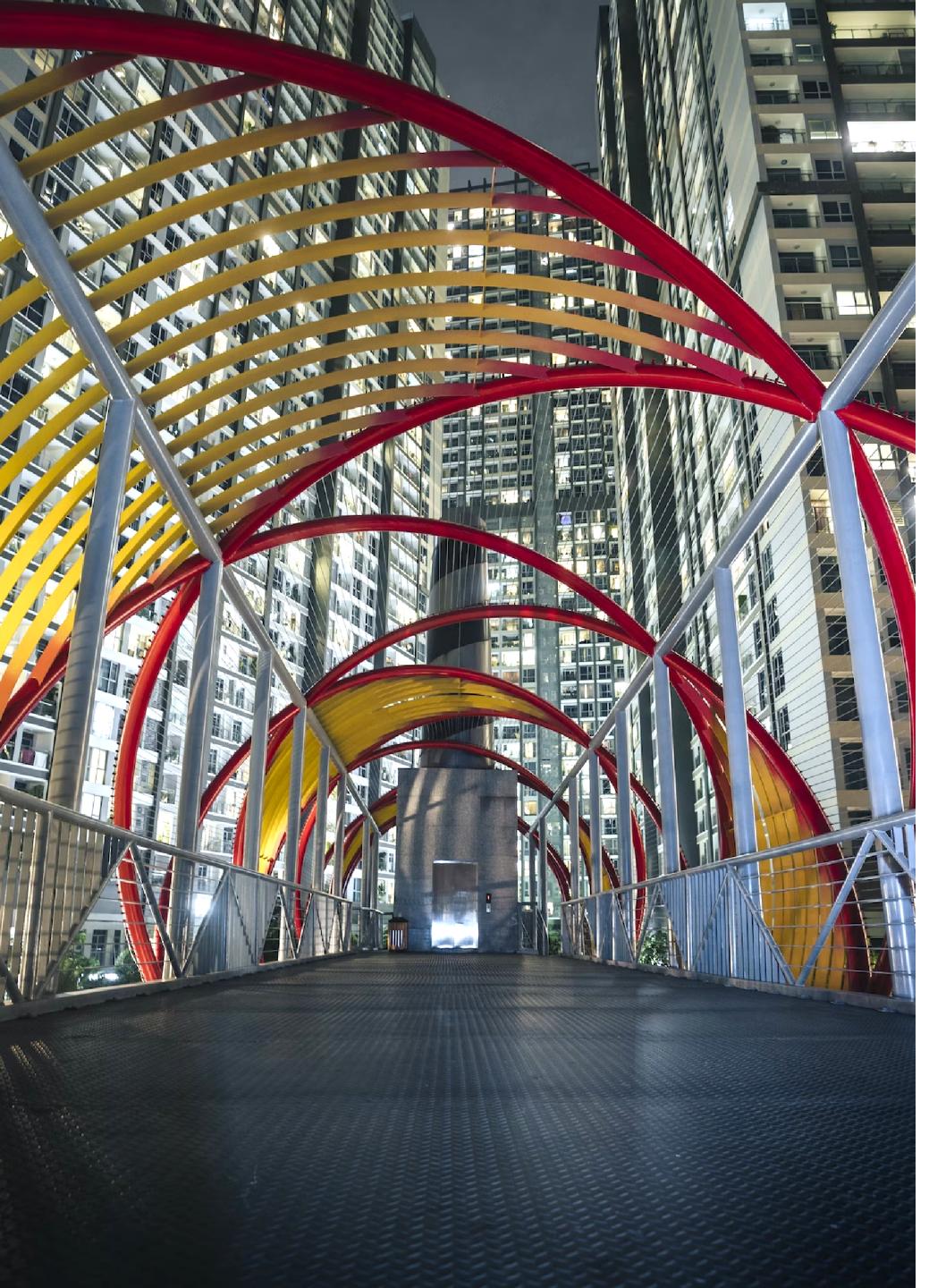


ドメイン駆動設計入門

@Kiyo_Karl2



自己紹介

- エンジニア歴は2年ちょい
- PHP/Laravelとか、インフラ周りの設計・構築、CI/CDとか
- 趣味はプログラミング以外だと車弄りとか

サマリー

- ドメイン駆動設計とは何か
- 値オブジェクト
- エンティティ
- ドメインサービス

ドメイン駆動設計とは？

Dmain Driven Design

ドメインの知識に焦点をあてた設計手法

ドメインとは？

- ソフトウェア開発において、「プログラムを適用する対象となる領域」。
- 重要なのはドメインが何かではなく「ドメインに何が含まれるか」ということ。
 - 会計システムのドメイン
 - 金銭、帳票
 - 物流システムのドメイン

焦点を当てる？とは

- ソフトウェアを適用する領域(ドメイン)と向き合い、ここに渦巻く知識に焦点をあてるということ。
 - よく観察し、よく表現すること。これは当たり前だが難しい。ドメイン駆動設計の指針はこれを補佐するものである。ドメイン駆動設計は当たり前を当たり前に実践するための開発手法である。

ドメインモデル

モデルとは

- 現実の事象や概念を抽象化したもの。
 - 例 ペン
 - 小説家: 文字が書けることが大事
 - 文房具店: 文字が書けることよりも値段が重要視される(利益を出す必要がある)
 - 例: トランク
 - 物流システム: 「荷運びができること」が表現できればよい。
 - エンジンキーを回すとエンジンがかかるといったことまで表現する必要はない

対象が同じものであっても、何に重きを置くかは異なる。

モデリング

- 事象や概念を抽象化する作業をモデリングと言い、その結果得られるのがモデル。
- ドメイン駆動設計ではこのモデルのことをドメインモデルと呼ぶ。
- ドメインの世界の住人は、ドメインの知識はあるが、ソフトウェアにとって重要な知識がどれかは不明。逆に開発者はソフトウェアにとって重要な知識はわかるが、ドメインについての知識がない。
 - 両者は協力する必要がある。(戦略的DDD)

知識をコードで表現するドメインオブジェクト

- ドメインモデルだけでは抽象化されただけでなんの問題解決にもならないので、ソフトウェアで動作するモジュールとして表現する必要がある。これがドメインオブジェクトである。
- 当然ソフトウェアは変化していく。
 - ドメインの概念→ドメインモデル→ドメインオブジェクト
 - 上記3つは互いに影響し合う

ドメイン駆動設計のパターン

- 知識を表現する
 - 値オブジェクト
 - エンティティ
 - ドメインサービス(今回発表するのはここまで)
- アプリケーションを実現するためのパターン
 - リポジトリ
 - アプリケーションサービス
 - ファクトリ
- 知識を表現する、より発展的なパターン
 - 集約
 - 仕様

値オブジェクト(Value Object)

- システム固有の値を表現するために定義されたもの。

値オブジェクトの性質

- 不変である
- 交換が可能である
- 等価性によって比較される

工業製品番号

工業製品は、ロットやシリアル、製品番号など製品を識別するためにさまざまな意味を持たせた番号が存在している。

これをプリミティブな値のみで表現してみる。

```
$modelNumber = 'a20421-100-1';
```

なんかよくわかんないけど、数字が羅列されている...

値オブジェクトを使ってみる

```
class ModelNumber
{
    public function __construct(
        private readonly string $productCode,
        private readonly string $branch,
        private readonly string $lot
    ) {
        if (!$this->isValidBranch()) {
            throw new \InvalidArgumentException("branchは3桁である必要があります");
        }
    }

    public function productCode(): string
    {
        return $this->productCode;
    }

    public function branch(): string
    {
        return $this->branch;
    }

    public function lot(): string
    {
        return $this->lot;
    }

    public function __toString(): string
    {
        return "{$this->productCode}-{$this->branch}-{$this->lot}";
    }

    private function isValidBranch(): bool
    {
        return strlen($this->branch) === 3;
    }
}

$number = new ModelNumber("a20421", "100", "1");
echo $number->productCode() . PHP_EOL;// a20421
echo $number->branch() . PHP_EOL;// 100
echo $number->lot() . PHP_EOL;// 1
echo $number . PHP_EOL;// a20421-100-1
```

わかりやすい！

- 値をオブジェクトとして表現してあげることで、表現力が増す。
- 仕様がひとめでわかる。
 - ドキュメント的な
- 不正な値を存在させない。(コンストラクタ)

値オブジェクトはデータを保持するコンテナではない。

- ふるまいをもつことができる。

お金

```
class Money
{
    public function __construct(private readonly float $amount, private readonly string $currency)
    {
    }

    (略)

    // お金を加算する
    public function add(Money $money): Money
    {
        if ($this->currency !== $money->currency) {
            throw new \InvalidArgumentException("通貨単位が異なります");
        }

        return new Money($this->amount + $money->amount, $this->currency);
    }
}

$myMoney = new Money(100, "JPY");
$yourMoney = new Money(200, "JPY");
$result = $myMoney->add($yourMoney);
echo $result->amount(); // 300

$myMoney = new Money(100, "JPY");
$yourUSDMoney = new Money(2.5, "USD");
Fatal error: Uncaught InvalidArgumentException: 通貨単位が異なります
$result = $myMoney->add($yourUSDMoney);
```

お金の加算

- 計算処理にルールを記述し、それにそぐわない操作を弾くようにしてシステムチックにバグを防止できる。
- 値オブジェクトはただのデータの構造体ではなく、オブジェクトに対する操作をふるまいとしてまとめることで、自身に関するを語るドメインオブジェクトらしさを帯びてくる。
- 逆に上記に定義されないことで、それができないということも示している。

エンティティ

- ドメインモデルを実装したドメインオブジェクト。
- 値オブジェクトを真逆にしたもの

可変である

```
<?php

class User
{
    public function __construct(private string $name)
    {
    }

    //getter
    public function name(): string
    {
        return $this->name;
    }

    public function changeName(string $name): void
    {
        if (mb_strlen($name) < 3)) {
            throw new InvalidArgumentException("名前は3文字以上です");
        }

        $this->name = $name;
    }
}

$user = new User("John");
$user->changeName("Elisa");
echo $user->name(); // Elisa
```

基本的にオブジェクトは不变にすべき

- 同じ属性であっても区別される。
- 例: 氏名が同じだったら人間は同一人物である? → 違う。
 - 人間は属性では区別されない。もっと別のところで評価される。まさにエンティティとしてふさわしい
 - 何で区別するか? → 識別子。等価性ではなく、同一性によって識別される必要がある。

```
<?php

class UserId
{
    public function __construct(private readonly int $value)
    {
    }

    public function value(): int
    {
        return $this->value;
    }
}

class User
{
    public function __construct(private readonly UserId $id,private string $name)
    {
    }

    public function name(): string
    {
        return $this->name;
    }

    public function changeName(string $name): void
    {
        if (mb_strlen($name) < 3) {
            throw new InvalidArgumentException("名前は3文字以上です");
        }

        $this->name = $name;
    }
}
```

エンティティの同一性

- 識別子は同一性の実体なので、`readonly`をつける → 読み取り専用にして値が不変であることを担保。
- エンティティでは属性を表す識別子だけが比較の対象となる。属性が変化してもエンティティの同一性は変わらない

```
class User
{
    public function __construct(private readonly UserId $id, private string $name)
    {
    }

    //getter
    public function name(): string
    {
        return $this->name;
    }

    public function changeName(string $name): void
    {
        if (mb_strlen($name) < 3) {
            throw new InvalidArgumentException("名前は3文字以上です");
        }

        $this->name = $name;
    }

    public function equals(object $other): bool
    {
        //同一オブジェクトであればtrueを返す
        if ($this === $other) {
            return true;
        }

        if (gettype($obj) !== gettype($this)) {
            return false;
        }

        return $this->id->value() === $other->id->value();
    }
}

$user = new User(new UserId(1), "test");
echo $user->name(). PHP_EOL;
$user->changeName("test2");
echo $user->equals(new User(new UserId(1), $user->name())) ? "true" : "false" . PHP_EOL;
```

エンティティにする判断基準

- ライフサイクルが存在するか
 - ユーザーは登録があり、途中で名前が変わったり、退会することがある

ドメインオブジェクトを定義するメリット

コードのドキュメント性

無口なコード

```
class User
{
    public string $name;
}
```

饒舌なコード

```
class UserName
{
    public function __construct(private readonly string $value)
    {
        if (mb_strlen($value) < 3) {
            throw new InvalidArgumentException("名前は3文字以上です");
        }
    }

    public function value(): string
    {
        return $this->value;
    }
}
```

コードを饒舌に

- コードを饒舌にする努力を怠らなければ、開発者はコードを手掛かりにしてそこに存在するルールを確認できる。
 - 本来はドメインモデルをつくり、これからドメインオブジェクトとして実装する
- 無口なコードだと、そこにあるルールがすべて守られているかはすべてのコードを洗い出し調べる必要がある。これはバグのもとになる。

コードを饒舌にする努力を怠るな！

ドメインサービス

サービスとは

- クライアントのために何かを行うオブジェクト
 - ドメインのためのサービスとアプリケーションのためのサービスがある。
 - 前者をドメインサービス、後者をアプリケーションサービスと呼ぶ

ドメインサービスとは

- 値オブジェクトやエンティティに記述すると不自然なものはドメインサービスに書く。

不自然なふるまいとは

例

```
class User
{
    public function __construct(private readonly UserId $id, private string $name)
    {
    }

    (省略)

    public function exists(User $user): bool
    {
        //重複確認処理
        return true;
    }
}

$userId = new UserId(1);
$userName = new UserName("Taro");
$user = new User($userId, $userName);

$duplicateCheckResult = $user->exists($user); //自身に重複の問い合わせ？
```

エンティティに定義すると不自然

- 重複確認のメソッドはExistsがあるので、自身に対して重複を問い合わせるのは不自然。
これはメソッドではtrueを返すかfalseを返すのが正解かわかりにくい。
- このことから、この重複判定メソッドをUserエンティティに記述するのは不自然なふるまいっぽい

不自然さを解決するオブジェクト

ドメインサービス

```
class UserService
{
    (省略)

    public function exists(User $user): bool
    {
        //重複確認処理
    }
}

$userService = new UserService();

$userId = new UserId(1);
$userName = new UserName("Taro");
$user = new User($userId, $userName);
```

ドメインサービスの濫用に注意!

- 大事なのは不自然なふるまいに限定すること
- サービスになんでも定義してしまうと、結局エンティティにはゲッターとセッターしかのこらない.....

```
class User
{
    public function __construct(private readonly UserId $id, private string $name)
    {
    }

    public function name(): string
    {
        return $this->name;
    }
}
```

ドメインサービスは極力使わない

- どんな熟練の開発者でもこのセッターとゲッターだけのUserクラスからどんなふるまい
やルールが存在するのはを知ることは不可能。
- この状況をドメインモデル貧血症に落ちいっているという
- **まずエンティティや値オブジェクトに定義できないか考えて、それが不自然なときに
ドメインサービスの利用はとどめる。**極力利用しないようにする。

今までのを使ってコードを簡単な書いてみる

エンティティ、値オブジェクトとユースケースの組み立て

- ユースケースというのはいわばクライアントがどんな動作をするかということ。
 - 登録、編集、削除

例えばユーザーが登録をするという処理を考える

```
class Program
{
    // ユーザー登録
    public function createUser(UserName $userName): void
    {
        $userService = new UserService();

        $userId = new UserId(1);
        $userName = new UserName("Taro");
        $user = new User($userId, $userName);

        if ($userService->exists($user)) {
            throw new Exception("ユーザーは既に存在しています");
        };

        // ユーザー登録処理
        try {
            $dbHost = getenv('DB_HOST');
            $dbName = getenv('DB_NAME');
            $dbUser = getenv('DB_USER');
            $dbPassword = getenv('DB_PASSWORD');

            $dsn = "mysql:host=$dbHost;dbname=$dbName;charset=utf8";
            $pdo = new PDO($dsn, $dbUser, $dbPassword);
            $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

            // ユーザー登録用のSQLクエリ
            $sql = "INSERT INTO users (username) VALUES (:username)";
            $stmt = $pdo->prepare($sql);

            $stmt->bindParam(':username', $username, PDO::PARAM_STR);

            $stmt->execute();
        } catch (PDOException $e) {
            // エラー処理
            echo "Error: " . $e->getMessage();
        }
    }
}
```

ユーザーサービスの定義

```
class UserService
{
    public function exists($userName): bool
    {
        try {
            // 環境変数からデータベース接続情報を取得
            $dbHost = getenv('DB_HOST');
            $dbName = getenv('DB_NAME');
            $dbUser = getenv('DB_USER');
            $dbPassword = getenv('DB_PASSWORD');

            // データベースへの接続
            $dsn = "mysql:host=$dbHost;dbname=$dbName;charset=utf8";
            $pdo = new PDO($dsn, $dbUser, $dbPassword);
            $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

            // ユーザーが存在するかどうかを確認するSQLクエリ
            $sql = "SELECT * FROM users WHERE username = :username";
            $stmt = $pdo->prepare($sql);
            $stmt->bindParam(':username', $userName, PDO::PARAM_STR);
            $stmt->execute();

            $result = $stmt->fetch(PDO::FETCH_ASSOC);

            return $result !== false;
        } catch (PDOException $e) {
            echo "Error: " . $e->getMessage();
            return false;
        }
    }
}
```

多少は読みやすい。なんとなーくやっていることはわかる。

でもさ

なんか \$pdo とか \$stmt は 何？？？

データ操作により処理の本質がぼやける

- ユーザー名の重複を確認するのにもDBへ問い合わせが必要。
 - UserServiceはほぼほぼDB操作に終始してしまっている。
 - どちらもちゃんと動くが、柔軟性に乏しい。リレーションナルデータストアがNoSQLデータへ変更するとなったら、ユーザー作成処理の本質は変わっていなくともコードを変更する必要がでてくる。
- データを扱う以上データの保存や読み取りは避けられない。
 - ユーザー作成処理が特定のデータストアや、操作処理について関心も持つべきではない。
 - リポジトリというもので行う。(今回は割愛)

引用元

ドメイン駆動設計入門 ボトムアップでわかる! ドメイン駆動設計の基本

まとめ

コードを適切な場所に適切に書く

ドメインロジックを散見させるな！ドメインオブジェクトに集める！

コードを饒舌にする努力を怠るな！

ご清聴ありがとうございました

ヘッダータイトル