Better Coding with Laravel

目次

---

1. はじめに〜前提知識の共有

1.1. 前提

はじめにお断りしなければならないことは、本講義でお伝えすることは、私の経験上役に立った知識や経験の中から一部を抜粋したものですが、他のすべての方の役に立つかどうかはわかりません。ソフトウェアエンジニアと一口に言っても、多様な仕事、多様な働き方がありますし、「良い」の基準も時代とともに移り変わるものがあります。みなさんのやりたい仕事、目指す働き方、チームの同僚や会社の方針などによって、求められるものは変わります。できるだけ、そうした状況に左右されないようなことも含めたつもりですが、なによりも皆さんには、自分自身で「良い」の基準をつくり、それに向かって進歩し続けてほしいと思います。

さて、まず前提として、ここでは、ある程度複雑なドメイン（ソフトウェアで扱う対象）を持つアプリケーションをチームで開発し、ある程度の期間作り変えられながら運用されていくような状況を想定しています。

ソフトウェア開発の現場で「即戦力」として求められる技術は多岐に渡りますが、その中で設計スキルは、自律的に働くために持っていてほしいスキルのひとつです。したがって、本テキストでは、Laravel の実践的な使い方よりも多くの分量を設計に関する内容に割きました。

要件定義が「なにを作るか」を決めるフェーズであれば、設計は「どう作るか」を決めるフェーズです。また、設計は実装と切り離すことはできず、設計と実装を行ったり来たりしながらソフトウェアを作っていくことが求められます。

求められる設計スキルは作るものによって変わってきますが、基本的な考え方や取り組み方はこれからお伝えすることから大きく外れることはないと思います。ですが、どれくらい深く知っておくべきか、という点については「いまはなんとも言えない」とお答えするしかありません。みなさんの置かれた（これから置かれる）状況次第です。

この先のみなさんのキャリアにおいて、少しでも本講義の内容が礎になれば幸いです。

1.2. 「より良い」の定義

ソフトウェアにとって「より良い」とはどういう状態か？

⇒「**こっちより**良い」（現在における他の選択肢との比較）、「**前より**良い」（過去との比較）

- 複数のアイデアを比較できる？
- 過去の間違いに気付ける？
- 「良い」「悪い」の基準を持っている？

ソフトウェアにとって「良い」とはどういう状態か？

動く＜正しく動く＜正しく動き続ける

- 動くこと＝価値が提供できること
- 正しく動くこと＝不具合がないこと
- 正しく動き続けること＝変更しても壊れないこと

ソフトウェアにとって「正しい」とはどういう状態か？

- 顧客の求めているもの
- 意図したとおりに動くもの
- 意図を理解しやすい���の

なぜソフトウェアを「正しく」作ることが求められるのか？

- 顧客の求めているものを作ることで、顧客に価値を最大限に提供できるから
- 意図したとおりに動くものを作ることで、不具合をなくすことができるから
- 意図を理解しやすいものを作ることで、メンテナンスしやすくなるから

ソフトウェアを「正しく」作るとなにがうれしいのか？

- お客さんにいいものが出来たと喜んでもらえるとうれしい
- 不具合が少なくなればトラブルが減って平和になるのでうれしい
- 他人が書いたコードが読みやすいとどういう動きになっているか悩まなくていいのでうれしい

1.3. 「正しく動き続ける」ソフトウェアをつくるために必要なこと

ソフトウェアが「正しく動き続ける」ために必要なこと

- 改善され続ける開発プロセス
- ビジネスをソフトウェアに写すための技術
- 変更しやすいコード

ソフトウェアが「正しく動き続ける」ことを阻むもの

- 正しくない要求
- エントロピー増大の法則
- スキル不足

ソフトウェア「正しく動き続ける」ことを阻害する要因を排除するために必要なこと

- 要求開発
- リファクタリング
- 学習する組織

もう一つの大事なこと：「素早くつくる」

- 素早く世に出して市場の反応を見る
- 素早くつくり変えて顧客の要望に応える
- 素早く直して顧客の信頼を保つ

1.4. 本講義で取り上げるトピックに関する概要

本講義で取り上げる3つのトピック

1. ビジネスを写すソフトウェアをつくるコツ〜ドメインモデルのより良い表現
2. 変更容易なコードを書くコツ〜より良いコーディングを支える開発プロセス
3. フレームワークを使いこなすコツ〜フレームワークとのより良い協調

ここまでのまとめ

本講義における良いソフトウェアとはどういうものか、を順を追って説明しました。

「顧客の求めているものを意図したとおりに動くように作り、その意図が他の人に理解しやすいもの」

という定義で、このあともお付き合いください。「良い」や「正しい」の指すものは他にもあるでしょう。みなさんにとって、「良い」「正しい」ものがどういうものであるか、少し考えてみてください。

最後に、「Clean Code」という本から、著者が著名なソフトウェアエンジニアの何人かに、クリーンコードとはなにか？と聞いて得られた回答の中から一部引用します。

デイブ・トーマス（OTIの創設者、Eclipsestrategyの後見人）

> *クリーンコードとは、原作者以外の人にも読むことができ、そして拡張できるコードのことです。そこには単体テストと受け入れテストがあります。そこには意味を持った名前があります。1つのことをするのに、いくつもの方法を提供するのではなく、ただ1つの方法を提供します。依存性は最低限で、それは明確に定義され、そして明快で最低限のAPIが提供されます。コードは文芸的でなければなりません。なぜなら言語によりますが、すべての必要な情報がコードだけで明確に表現できるわけではないからです。*
>
> *— Robert C. Martin*
> *花井 志生. Clean Code　アジャイルソフトウェア達人の技*

マイケル・フェザーズ（『WorkingEffectivelywithLegacyCode』の著者）

> *私がクリーンコードの中に見つけた品質に関する項目を、1つ1つここで挙げていくこともできるでしょうが、その中でも、包括的で、すべてを先導するものが1つあります。クリーンコードは常に誰かが気配りを持って書いているように見えます。コードをよりよくするのに、すぐにわかるような明白なものは存在しません。こうしたことがらはすべて、コードの作者が考えるのです。改善について思いを馳せると、あなたは、あなた自身が座っている場所へといざなわれます。そこであなたは、誰か（同じ仲間のために、仕事に深い気配りができる誰か）が残してくれたコードを前に感謝を捧げているのです。*
>
> *— Robert C. Martin*
> *花井 志生. Clean Code　アジャイルソフトウェア達人の技*

---

2. ビジネスを写すソフトウェアをつくるコツ〜ドメインモデルのより良い表現

2.1. はじめに

2.1.1. この章の概要

ソフトウェア開発において「ビジネスを写す」行為に相当するのは「要件定義」と「設計」です。アプリケーション（＝ウェブサービス）がビジネスそのものであるケースや、大きなビジネスの一部であるケース、自社開発や受託開発など、様々な要素が絡み合って、要件定義や設計は複雑になりがちです。それでも、そうした複雑さをできるだけ明瞭に保つためにできる工夫があります。この章ではそうした工夫の一部をご紹介します。

2.1.2. この章の目的

「前提知識」のところで挙げた以下の2点を思い出してください。

なぜソフトウェアを「正しく」作ることが求められるのか？

- 顧客の求めているものを作ることで、顧客に価値を最大限に提供できるから

ソフトウェア���「正しく動き続ける」ために必要なこと

- ビジネスをソフトウェアに写すための技術

顧客の求めているものをソフトウェアにする過程で、できるだけ正確で明瞭な言葉や概念を定義し、それらをできるだけそのままコードに落とし込んでいくことができれば、要件定義から設計、実装、テストまで、認識の齟齬や不明瞭な解釈といった不具合の原因になりがちな要素を少しでも多く排除することができます。

この章では、そうした言葉や概念の定義を正確に表現することの大切さを学んでいただければ、と思います。

2.2. ドメインモデルとは

以下の記事を参照してください。

[ドメインモデル、ドメインロジックとは何かをコードを交えて考えてみる - Qiita]

一言で説明するのは難しいんですが、

> *ソフトウェア開発におけるドメインは、そのソフトウェアがなにをするためのものなのか、という定義のうち、ウェブとかデータベースとかメールとか、そういう外部のソフトウェアや決まりごと（HTTPとかSQLとかSMTPとか）の無関係な部分*
>
> — *ドメインモデル、ドメインロジックとは何かをコードを交えて考えてみる - Qiita*

という理解でいったんはいいのかな、と思っています。上記の記事では、実際にコードを用いて「ドメインモデル」を表現した例も載せてありますので、そこを読むとさらに理解が深まると思います。

2.3. 概念の定義と表現

2.3.1. 概念の定義

> *思考において把握される、物事の「何たるか」という部分。抽象的かつ普遍的に捉えられた、そのものが示す性質。*
>
> — *実用日本語表現辞典*

ソフトウェアは何らかの目的を達成するためにつくられるもので、とりわけ仕事で携わるウェブアプリケーションソフトウェアは、何らかのビジネスの一端を担う目的があることがほとんどです。そうしたビジネスの「何たるか」をコードで表現することが、プログラミングの大事な要素のひとつです。

2.3.2. 概念の投影としてのコード

プログラムは**書いたとおりにしか動かない**ものなので、プログラマが「概念」を曖昧にしか理解していないと、その概念を具現化したコードも不正確になってしまう危険があります。

日常で何気なく使っている言葉でも、それをソフトウェアで表現する際には、厳密に表現する必要があります。

概念の例1）商品

ECサイトで販売しているものは商品ですが、実体を持っているわけではなく、そのサイトで販売しているものが洋服であれば（洋服もまた概念）、洋服一着一着が実体であり、それらの集合に名前をつけたものが「洋服」であり「商品」です。また、「商品」も、同じ名称で異なる色やサイズによって分かれていることがあり、一口に「商品」と言っても、それが指す概念を統一させるのは難しいので、細かく定義する必要があります。

```php
/**
 * 商品
 */
class Item
{
    private string $name;
    private int $price;
    private string $brand;
    // ...
}
```

概念の例2) VIP会員

あるECサイトでは特定の条件を満たす会員を「VIP会員」と定め、優遇することにしました。その条件は、サイト運営者が決めた概念です。条件や優遇措置は、「VIP会員」という言葉からイメージできないので、細かく定義する必要があります。

```php
/**
 * 会員
 */
class Member
{
    private bool $isVip;
    // ...
}

/**
 * VIP会員
 */
class VipMember extends Member
{
    // ...
}

/**
 * VIP会員の条件を満たしているかどうか
 */
class VipMemberSpecification
{
    public function isSatisfied(Member $member): bool {...}
}

/**
 * VIP会員の優遇措置
 */
class VipMemberPreferentialTreatment
{
    public function discount(int $price): int {...}
    public function shippingFee(int $fee): int {...}
    // ...
}
```

要件定義や設計で抽出した概念をコードで表現するためには、名前付けが非常に重要です。人間は概念を言葉で理解するので、仕様を明確にしたり、複数人で認識を合わせたりするために、言葉に細心の注意を払う必要があります。次のセクションでもう少し詳しく説明します。

��習1

1. 「赤信号」とはどういうものか説明してください
2. 「じゃんけん」のルールを説明してください
3. 「親近感」を別の言葉で表現してください（思いつく限り挙げてください）

2.4. より良い名前付け

概念の抽出ができたら、次に行うことはその概念に名前を付けることです。ほとんどの場合、プログラムは英語をベースにして書かれますので、概念は日本語、表現は英語、となり、間に翻訳が挟まることになります。日英辞典、英英辞典を使って翻訳しますが、日本特有の概念は英語化することが難しいので、複数の候補から選んで決めます（最近は https://codic.jp/ みたいな便利なサービスもありますので使うのもいいでしょう）。チーム開発の場合は、何らかの方法でコンセンサス（この概念にはこの語を充てますという共通理解）を取ることが必要です。

2.4.1. 変数名

Bad

```
$x = 0;
```

変数名を見て、データの中身や用途がイメージできない。

Good

```
$countLiked = 0;
```

いいねされた数、という具体的なイメージができる。

2.4.2. 関数名

Bad

```
$sum = func(1, 2);

function func($a, $b) {
    return $a + $b;
}
```

関数名を見て、どんな処理をするのかイメージできない。

Good

```
$sum = add(1, 2);

function add(int $a, int $b): int {
    return $a + $b;
}
```

2つの数値を足す、という具体的なイメージができる。

2.4.3. クラス名

Bad

```
class Util
{
    public static function formatDateTime(DateTimeInterface $dateTime): string {...}
}

echo Util::formatDateTime($article->publishedAt);
```

クラス名を見て、どういう処理やデータがまとまっているクラスなのかイメージできない。

Good

```
/**
 * 記事の公開日
 */
class PublishedAt
{
    public function __construct(DateTimeInterface $dateTime) {...}
    public function longFormat(): string {...}
    public function shortFormat(): string {...}
}

echo (new PublishedAt($article->publishedAt))->longFormat();
```

「公開日」という日時型のデータを持つクラスであることがイメージできる。

演習2

1. 「連続した計算の途中の結果」を保持する変数名を考えてください
2. 「2つの場所の距離を算出する」関数名を考えてください
3. 「開始日と終了日をデータとして持ち、与えられた日時がその範囲に含まれているかどうかを判定する関数」を持つクラス名を考えてください

2.5. より良い構造化

2.5.1. 「構造化」とは

> *構造化プログラミング（こうぞうかプログラミング、英: structured programming）とは、コンピュータプログラムの明瞭化を目的にした手法であり、一般的には順接、分岐、反復の三つの制御構造（control structures）によって、処理の流れ���記述することであると認識されている。コードブロックとサブルーチンも加えられることがある。*
>
> — *Wikipedia*

以下の5つがプログラムを「構造化」するための要素です。

- 順接 (sequence)
- 分岐 (selection)
- 反復 (repetition)
- ブロック (block)
- サブルーチン (subroutine)

順接 (sequence)

処理を順番に実行していくこと

```php
$n = 1 + 2;
$m = $n * 2;
echo $m;
```

分岐 (selection)

条件によって実行する処理を分けること (if/else, switch)

```php
if ($n > $max) {
    $max = $n;
}
```

反復 (repetition)

繰り返し処理を実行すること (for, foreach, while)

```php
$totalPrice = 0;
foreach ($items as $item) {
    $totalPrice += $item->price;
}
```

ブロック (block)

処理の集まり（PHP では `{}` で囲む）

（現代のプログラミングでは、以下のようなブロックはほぼまったく使わないので、覚えなくて大丈夫です）

```php
$n = 0;

first: {
    $n = 1;
    echo '$n=', $n, PHP_EOL;
    // $n=1
}

second: {
    $n = 2;
    echo '$n=', $n, PHP_EOL;
    // $n=2
}

echo '$n=', $n, PHP_EOL;
// $n=2
```

サブルーチン (subroutine)

処理の集まりに名前を付けたもの（言語によって呼び方は異なるが、PHP では関数やメソッドが該当する）

```php
function fizz_buzz(int $n): string {
    if ($n % 3 === 0 && $n % 5 === 0) {
        return 'FizzBuzz';
    }
    if ($n % 3 === 0) {
        return 'Fizz';
    }
    if ($n % 5 === 0) {
        return 'Buzz';
    }
    return (string)$n;
}

for ($i = 1; $i <= 100; $i++) {
    echo fizz_buzz($i), PHP_EOL;
}
```

2.5.2. 「良い構造化」とは

順接

意味のある順番になっているか

Bad

```php
$x = 0;
$y = 0;

$x += 1;
$y += 1;

$x *= 2;
$y *= 2;

echo "$x, $y", PHP_EOL;
```

Good

```php
// x に関する処理をまとめる
$x = 0;
$x += 1;
$x *= 2;

// y に関する処理をまとめる
$y = 0;
$y += 1;
$y *= 2;

echo "$x, $y", PHP_EOL;
```

分岐

Bad

```php
if (isset($params['value'])) {
    if ($params['value'] >= 10) {
        // ここに
        // 長い
        // 処理が
        // 入る
        return $result;
    } else {
        return $params['value'];
    }
} else {
    return 0;
}
```

Good

```php
// 例外的な処理は早期リターンを使う
if (!isset($params['value'])) {
    return 0;
}
// 計算が必要ないようなケースも早期リターンが使える
if ($params['value'] < 10) {
    return $params['value'];
}
// 長い処理を関数にすればもっと良い
return doSomething($params['value']);
```

反復

Bad

```php
// メンバーが全員同じ組織に属していることと3科目のスコアが揃っていることを確認する

$members = [
    ['id' => 1, 'organization_id' => 1],
    ['id' => 2, 'organization_id' => 1],
    ['id' => 3, 'organization_id' => 1],
];
$scores = [
    ['member_id' => 1, 'values' => ['数学' => 90, '国語' => 80, '英語' => 70]],
    ['member_id' => 2, 'values' => ['数学' => 80, '国語' => 70, '英語' => 60]],
    ['member_id' => 3, 'values' => ['数学' => 85, '国語' => 75, '英語' => 65]],
];
$organizationId = null;
$expectedKeys = ['数学', '国語', '英語'];
sort($expectedKeys);
for ($i = 0; $i < count($members); $i++) {
    if ($organizationId !== null && $members[$i]['organization_id'] !== $organizationId) {
        return false;
    }
    $organizationId = $members[$i]['organiztion_id'];
    $keys = array_keys($scores[$i]['values']);
    sort($keys);
    if ($keys !== $expectedKeys) {
        return false;
    }
}
return true;
```

Good

```php
// メンバーが全員同じ組織に属していることと3科目のスコアが揃っていることを確認する

$members = [
    ['id' => 1, 'organization_id' => 1],
    ['id' => 2, 'organization_id' => 1],
    ['id' => 3, 'organization_id' => 1],
];
$scores = [
    ['member_id' => 1, 'values' => ['数学' => 90, '国語' => 80, '英語' => 70]],
    ['member_id' => 2, 'values' => ['数学' => 80, '国語' => 70, '英語' => 60]],
    ['member_id' => 3, 'values' => ['数学' => 85, '国語' => 75, '英語' => 65]],
];

// 一度にひとつだけのことをやる
$first = array_shift($members);
// foreach が使えるところは for の代わりに foreach を使う
foreach ($members as $member) {
    if ($member['organization_id'] !== $first['organization_id']) {
        return false;
    }
}

// ここでも一度にひとつだけのことをやる
$expectedKeys = ['数学', '国語', '英語'];
foreach ($scores as $score) {
    // 自作の関数に置き換えて、ブロックの中をできるだけ簡潔にする
    if (!Arr::hasSameKeys($score['values'], expectedKeys)) {
        return false;
    }
}

return true;
```

サブルーチン

Bad

```php
/**
 * リクエストデータを処理する
 *
 * @param array $data
 * @param bool $flag true のときは object に変換して返す
 * @return array|object
 */
function processData(array $data, bool $flag = true)
{
    // ...
}
```

Good

```php
/**
 * リクエストパラメータを最適化して配列で返す
 *
 * @param array $params リクエストパラメータ
 * @return array
 */
 // process とか data とか flag とか抽象的すぎる名前はできるだけ使わない
 // bool な引数はできるだけ使わない
 // 一度にひとつだけのことをやるように関数を分ける
 // 戻り値の型はできるだけひとつにする（型宣言が使えるように）
function optimizeRequestParamsToArray(array $params): array
{
    //...
}

/**
 * リクエストパラメータを最適化してオブジェクトで返す
 *
 * @param array $params リクエストパラメータ
 * @return object
 */
function optimizeRequestParamsToObject(array $params): object
{
    $optimized = optimizeRequestParamsToArray($params);
    return (object)$optimized;
}
```

演習3

1. 以下のコードをより良いと思える形に書き直して、理由を説明してください

合計1000円以上購入で20%、10個以上購入で25%オフ

テストコード

```php
<?php

namespace Tests\Unit;

use App\Models\PriceCalculator;
use PHPUnit\Framework\TestCase;

class PriceCalculatorTest extends TestCase
{
    public function testCalculate()
    {
        $fruitPrices = ['banana' => 300, 'apple' => 200, 'orange' => 250];
        $itemCounts = ['banana' => 2, 'apple' => 4, 'orange' => 5];

        $discountedPrice = (new PriceCalculator())->calculate($fruitPrices, $itemCounts);

        $this->assertSame(1590, $discountedPrice);
    }
}
```

プロダクションコード

```php
<?php

namespace App\Models;

class PriceCalculator
{
    public function calculate(array $prices, array $counts): int
    {
        foreach ($prices as $fruitName => $price) {
            $totalPrice += $price * $counts[$fruitName];
            $totalCount += $counts[$fruitName];
        }

        if ($totalPrice >= 1000) {
            $discountRate = 0.8;
            if ($totalCount >= 10) {
                $discountRate = (float)bcmul($discountRate, '0.75', 1);
            }
        } elseif ($totalCount >= 10) {
            $discountRate = 0.75;
        } else {
            $discountRate = 1;
        }
        return (int)bcmul($totalPrice, $discountRate);
    }
}
```

2. 以下のコードをより良いと思える形に書き直して、理由を説明してください

ユーザーが a, b, c の中から選択した文字に対応するコマンド名を取得する

テストコード

```php
<?php

namespace Tests\Unit;

use App\Models\Command;
use PHPUnit\Framework\TestCase;

class CommandTest extends TestCase
{
    public function testFindCommand()
    {
        $param = 'c';
        $command = (new Command())->findCommand($param);
        $this->assertSame('continue', $command);
    }
}
```

プロダクションコード

```php
<?php

namespace App\Models;

class Command
{
    public function findCommand(string $option): string
    {
        $commandOptions = ['a', 'b', 'c'];
        $commands = ['abort', 'break', 'continue'];
        switch ($param) {
            case $commandOptions[0]:
                $command = $commands[0];
                break;
            case $commandOptions[1]:
                $command = $commands[1];
                break;
            case $commandOptions[2]:
                $command = $commands[2];
                break;
            default:
                throw new \InvalidArgumentException('invalid parameter');
        }
    }
}
```

3. 以下のコードをより良いと思える形に書き直して、理由を説明してください

```php
/*
 $filter['key'] が status のときは status が $filter['value'] に一致するレコードを $tasks から抽出し、
 $filter['key'] が priority のときは priority が $filter['value'] に一致するレコードを $tasks から抽出する
*/

$tasks = [
    ['id' => 1, 'status' => 'doing', 'priority' => 'high'],
    ['id' => 1, 'status' => 'doing', 'priority' => 'normal'],
    ['id' => 1, 'status' => 'open', 'priority' => 'low'],
];

$filter = ['key' => 'status', 'value' => 'open'];

$filteredTasks = [];
if ($filter['key'] === 'status') {
    foreach ($tasks as $task) {
        if ($task['status'] === $filter['value']) {
            $filteredTasks[] = $task;
        }
    }
} elseif ($filter['key'] === 'priority') {
    foreach ($tasks as $task) {
        if ($task['priority'] === $filter['value']) {
            $filteredTasks[] = $task;
        }
    }
}
assert(count($filteredTasks) === 1);
assert($filteredTasks['id'] === 1);
```

ここまでのまとめ

コードによる表現方法はさまざまです。実装しながらあっちがいい、いやこっちがいいかも、と試行錯誤を繰り返しながら、また、コードレビューを通して他の人と議論したりしながら、改善していきますが、できるだけ早く最初から無駄のない、シンプルなコードを書けるようになるためにできる訓練があります。

概念をコードに落とし込む訓練、複数のコードを見比べてどちらが良いか判断する訓練

こういったことを訓練するには、他にもっと良い書き方がないかな、という探究心が必要です。

---

3. 変更しやすいコードを書くコツ〜より良いコーディングを支える開発プロセスと技術

3.1. はじめに

3.1.1. この章の概要

アジャイル開発では、設計と実装を切り離さず（時として要件定義もしながら）、行ったり来たりしながら開発していきますが、そのためコードは頻繁に変更されます（仕様変更によるもの、不具合修正によるもの、リファクタリング、など）。昔は「動いてるコードに触るな」という戒律のあるチームも多かったですが（自動テストがない、不具合発生時のコストが大きい、影響範囲がわからない、などの理由で）、現代では少なくなってきています。頻繁に変更していく開発スタイルの場合、変更に強いシステムでなければなりませんし、開発プロセスもそれに応じたものにしなければなりません。この章では、継続的な改善活動を支えるプロセスや思想の一部をご紹介します。

3.1.2. この章の目的

「前提知識」のところで挙げた以下の項目を思い出してください。

なぜソフトウェアを「正しく」作ることが求められるのか？

- 意図を理解しやすいものを作ることで、メンテナンスしやすくなるから

ソフトウェアが「正しく動き続ける」ために必要なこと

- 改善され続ける開発プロセス
- 変更しやすいコード

3.2. コーディング規約とIDE

PHP には PSR（PHP Standard Recommendations）という、PHP の様々な領域において標準を規定するものがあります。ここでは詳しくは述べませんが、その中にコーディング標準に関するものもあります（最新は PSR-12）。コーディング規約は、いくつも存在するコーディング標準の中から「このやり方に従ってコーディングしよう」と取り決めることです。現在の PHP を使った開発では、この PSR が主流です。様々なチームやフレームワークで開発することになっても PSR-12 に従ってコーディングすることで、コードのフォーマットや命名規則にある程度の統一性をもたらすことができるようになり、だれが書いても同じようなコード（の見た目）になるため、読みやすいコードになります。

PSR-12: Extended Coding Style - PHP-FIG

現代のソフトウェア開発では、GitHub などの Git リポジトリホスティングサービスを使って、プルリクエストを作り、レビューを経てから、コードの変更がプロダクトに適用される、という開発プロセスが主流になりつつあります。まだコーディング標準が一般的になる以前は、コードレビューの中で、コードスタイルに対するレビューで議論が起こり、本質的でない部分に時間が浪費される、ということがときどきありました。コードの見た目やスタイルが統一されてないと、読み手の認知負荷が高まる（読むのに余計に時間と労力がかかる）ため、そうした部分をきれいに保つ、というのは重要なことです。けれども、プログラム自体の品質（意図したとおりに実装されているか、不具合はないか、といったこと）に比べると枝葉の部分であることは間違いありません。なので、そうした見た目やスタイルの部分はコーディング規約に従って、みんなが同じように書いて、コードレビューではそうした部分のレビューをする必要がないようにしましょう、というのが昨今の潮流です。

PhpStorm を始めとする IDE やエディタには、コードがこれらの規約に沿っているかどうかをチェックしてくれる機能やプラグインが存在しているので、必ず入れるようにしましょう。

PHP では PHP_CodeSniffer というツールが、コーディング規約に沿ったコードになっているかどうかチェックしてくれます。

squizlabs/PHP_CodeSniffer: PHP_CodeSniffer tokenizes PHP, JavaScript and CSS files and detects violations of a defined set of coding standards.

スタイルが統一されていない読みにくいコードの例

```
if($x===0){
  $n =1;
}
elseif ($x === 1)
{
    $n = 2;
} else
{
    $n = 3;
}
```

3.3. テスト駆動開発とリファクタリング

3.3.1. テスト駆動開発

「テスト駆動開発」は2002年にアメリカで出版された「Test-Driven Development By Example」という本の中で、著者の Kent Beck が提唱した開発手法です。

- プロダクションコードを書く前にテストコードを書く
- 仕様を満たしているかを確認しながらテストコードを書く
- テストコードを書きながら設計を洗練させていく

といった方針で、後述するリファクタリングとセットで、変更に強いソフトウェアを作ることを目的としています。

テスト駆動開発はまた、開発者が安心してコードを変更できる状況を作ります。テストが通りさえすれば、どんどんリファクタリングすることができます。

紹介した書籍は「テスト駆動開発」というタイトルで日本語訳もありますので、興味のある方は読んでみてください。

3.3.2. リファクタリング

1999年にアメリカで出版された「Refactoring」という本の中で、著者の Martin Fowler が提唱した、ソフトウェアの外部的振る舞いを保ちつつ、理解や修正が簡単になるように内部構造を改善していく開発手法です。

テスト駆動開発で行われるリファクタリングは実装しながら行うものですが、一度リリースしたものに対してリファクタリングを行うこともあります。ここでは詳しく解説しませんが、一般的に「技術的負債」と呼ばれる、継続的に生産性を落とすことになるコードの状態を定期的に見直してメンテナンスしていく作業です。リファクタリング自体は、プロダクトに新しい価値を与えるものではないため、リファクタリングだけをする期間を定期的に設けているチームは少ないようですが、開発速度が著しく低下して、累積的にコストが膨らむようなときはリファクタリングで解決できる可能性があります。技術的負債がもたらす負の側面について、もう少し経営層やマネジメント層の方々の理解を得ていかなければならないと感じています。

紹介した書籍は「リファクタリング」というタイトルで日本語訳もありますので、興味のある方は読んでみてください。

3.4. オブジェクト指向とドメイン駆動設計

3.4.1. オブジェクト指向

現在の PHP プログラミングでは、オブジェクト指向が不可欠になっている一方で、プログラムの複雑度を高める原因になってしまうこともあり、多くの人たちが頭を悩ませています。

詳しい歴史的な話は割愛しますが、Java、Ruby、Python などの他のプログラミング言語でも、オブジェクト指向スタイルが主流ですし、FORTRAN、COBOL、BASIC などの手続き型しかサポートしていない言語は（FORTRAN はオブジェクト指向を最近サポートしたようですが）、もう特定の分野（数値計算、金融、教育、など）だけでしか使われなくなっています。

[オブジェクト指向と20年戦ってわかったこと - Qiita]

書いてあることが小難しくてよくわからないかもしれませんが、オブジェクト指向と言ってもいろんな実装や考え方があるというのと、20年経ってもまだ戦い続けなくてはいけない相手である、ということだけ覚えておけばいいのではないか、と思います。

大事な部分は以下の点です。

> - *他のコードへの依存性が少ない/整理されていて、切り出しやすい*
> - *細かいロジックがうまく隠蔽されていて、少ないインタフェースコードで利用できる*
> - *レイヤーがうまく分かれている*
>
> *— オブジェクト指向と20年戦ってわかったこと - Qiita*

3.4.2. ドメイン駆動設計

2003年にアメリカで出版された「Domain-Driven Design」という本の中で、著者の Eric Evans が提唱した、「ドメインモデル」という、ソフトウェアが実現したい対象となるビジネスの構造や用語などを忠実に再現したモデルを元にし、インクリメンタルに改善を続けながら設計と実装をつなげていく開発手法です。

個人的にはすべてを理解する必要はなく、エッセンスを抽出して、うまく自分のプロジェクトに適用していけばいいかな、と思います。

- 小さいステップで繰り返し改善していきましょう
- 同じ言葉であってもコンテキスト（文脈）によって異なる概念となることがあるので分けましょう
- クラスをレイヤー化し、依存を一方向にしましょう

といったようなことは、ドメイン駆動でなくても、十分に有用なプラクティスになります。

上記の書籍は「ドメイン駆動設計」というタイトルで日本語訳もありますが、内容が難解で挫折する人が続出し、「実践ドメイン駆動設計」を始めとする「補助教材」が何冊も出ています。分厚い本も多いので、いきなり手を出すと損した気分になる可能性もあるので、興味のある方は手始めにこちらの記事を読むといいかもしれません。

DDDはオブジェクト指向を利用してどのようにメンテナブルなコードを書くか - little hands' lab

ここまでのまとめ

ウェブアプリケーション開発の世界で現在主流になっているアジャイル開発でも、最初から正しいものは作れないので、小さく試して素早く改善していこう、という方針にもとづいていて、「適応力」が求められます。問題を見つけ、解決策を試し、うまくいけば切り替え、うまくいかなければ  とに戻す、これらの小さなステップを頻繁に行います。

この章でご紹介した「テスト駆動開発」や「ドメイン駆動設計」も同じく継続的な改善を前提にしています。**最初から正しいものを作ろうとせず、間違っていたら素早く直す、そのために、あとから直しやすいようにプログラムを書く**、ということが必要になります。

ルールやプロセスが改善の足枷になっては本末転倒なので、より良いソフトウェアを作るために、柔軟で即応的な思考を持ち、より良い開発を阻むものを見つけたら、どんどん改善していきましょう。

---

4. フレームワークを使いこなすコツ〜フレームワークとのより良い協調

4.1. はじめに

4.1.1. この章の概要

Laravel を上手に使うためのコツをお伝えします。PHP を使って20年近くウェブアプリケーションを作ってきましたが、個人的には、Laravel を使うようになって、開発体験やプロダクト品質が飛躍的によくなったと感じていま

す。Laravel のどういう部分が優れているか、どうやって使えば高品質なアプリケーションになるか、といった点を説明します。

4.1.2. この章の目的

これまでは、どちらかというと概念的なことや基本となる考え方などについて述べてきましたが、実践的な問題として「プログラムをどう書くか」にフォーカスします。Laravel らしい書き方をひとつでも学んでいただければ、と思います。

4.2. フレームワークの作法に上手に従う

Ruby on Rails が後続のウェブアプリケーションフレームワークに与えた影響は大きくて、Laravel も例外ではありません。一方で「Rails Way」と呼ばれるような（「Laravel Way」はあまり言われませんね）、フレームワークの作法が、足枷になって生産性と品質を落としてしまうこともあります。Laravel では Rails に比べるとそうした作法が少ないとは思いますが、それでも、Eloquent や Event など、Laravel のパフォーマンスを最大限に引き出そうとすると利用せざるを得ないモジュールもありますので、フレームワークの作法に上手に従うためにどうすればいいか考えてみましょう。

4.2.1. Eloquent のクエリスコープメソッドを上手に使う

クエリスコープメソッドは、クエリビルダで構築されるロジックをひとまとめにしたり、別の形に置き換えたりして、インタフェースとロジックを切り離す仕組みです。

https://laravel.com/docs/7.x/eloquent#query-scopes

例として、ユーザーに対してなんらかの承認を行い、承認済みユーザーを区別したシチェーションを考えてみます。いくつかデータの持ち方が考えられます。

1. users テーブルに is_approved (TINYINT(1)) カラムを用意し、0=未承認、1=承認済みとする
2. users テーブルに approved_at (TIMESTAMP) カラムを用意し、NULL=未承認、NOT NULL=承認済みとする
3. user_approvals テーブルを用意し、レコードがなければ未承認、あれば承認済みとする

それぞれの実装は以下のようになるかと思います。

```
// 1
User::where('is_approved', true)->get();

// 2
User::whereNotNull('approved_at')->get();

// 3
User::whereHas('approval')->get();
```

「承認済み」という表現が変わるたびにコードを書き換えなければいけません。承認済みユーザーを取得する処理があちこちにあれば、このロジックが変わるとすべての箇所を探して直すことになり、変更漏れがあればそれが不具合になってしまいます。

一方、クエリスコープメソッドを使うと以下のようになります。

```php
class User
{
    public function scopeApproved(Builder $builder): Builder
    {
        // 1
        return $builder->where('is_approved', true);
        // 2
        return $builder->whereNotNull('approved_at');
        // 3
        return $builder->whereHas('approval');
    }
}

// 利用側
User::approved()->get();
```

こうしておくと、ロジックが変わっても変更箇所は一箇所で済みます。

ポイントは、ある概念（この場合は「承認済み」）に対して複数の実装方法が考えられるとき、選択された実装方法に依存するようなクエリを直接書かないことです。

4.2.2. 配列操作を上手に行う

不具合の元になる操作はいくつかありますが、配列操作もそのひとつです。あるはずの要素がないケースを想定してなかったり、あちこちに条件文が書かれて読みにくくなったりします。Laravel にはこうした配列操作をサポートするクラス Arr と Collection があります。これらを上手に使って、処理を見やすく安全にしましょう。

例として、いくつかよく使う操作を挙げます。

```php
//
// ネストした配列から下の下位層のデータを取り出す
//
$data = [
    'name' => 'John Doe',
    'address' => [
        'postal_code' => '100-0001',
        'prefecture' => '東京都',
    ],
];
// before
$postal_code = '';
if (isset($data['address'])) {
    $postal_code = $data['address']['postal_code'] ?? '';
}
// after
$postal_code = Arr::get($data, 'address.postal_code', '');

//
// 配列の中から特定のキーを持つ要素を取得する
//
$data = ['id' => 1, 'name' => 'John Doe', 'email' => 'john@people.example'];
// before
$keys = ['id', 'email'];
$extracted = [];
foreach ($data as $key => $value) {
    if (in_array($key, $keys, true)) {
        $extracted[$key] = $value;
    }
}
// after
$extracted = Arr::only($data, ['id', 'email']);

//
// 配列からキーバリューペアを構築する
$data = [
    ['id' => 1, 'name' => 'John Lennon', 'email' => 'john@beatles.example'],
    ['id' => 2, 'name' => 'Paul McCartney', 'email' => 'paul@beatles.example'],
    ['id' => 3, 'name' => 'George Harrison', 'email' => 'george@beatles.example'],
    ['id' => 4, 'name' => 'Ringo Starr', 'email' => 'ring@beatles.example'],
];
// before
$members = [];
foreach ($data as $record) {
    $members[$record['id']] = $record['name'];
}
// after
$members = collect($data)->pluck('name', 'id')->toArray();
```

4.2.3. 入出力に関する処理を上手に分離する

ウェブアプリケーションはユーザーからの入力を受け取って処理を行い、処理の結果を出力しユーザーに返す、というのが基本的な処理の流れになります。そのような入出力に関する処理を上手にまとめてメインの処理と分離してやることで、メインの処理がすっきりして読みやすくなります。

例として、入力パラメータに検索条件と欲しいカラムのリストが送られてくるので、それに応じたデータの検索と出力を行う処理を考えてみます。

```
// 複数の検索条件にもとづいて検索を行い、検索結果を整形して返す
// before
public function index(Request $request): JsonResponse
{
    $query = Item::query();
    if ($request->price) {
        $query->wherePrice($request->price);
    }
    if ($request->name) {
        $query->where('name', 'LIKE', "%{$request->name}%");
    }
    // 他にも検索条件がたくさんある
    // ...

    // 詳細情報は別テーブルにあるが、レスポンスでは階層化しないで返したい
    $items = $query->with('detail')->get();
    $response = [];
    foreach ($items as $item) {
        $item['description'] = $item->detail ? $item->detail->description : '';
        $item['ranking'] = $item->detail ? $item->detail->ranking : '';
    }
    return response()->json($response);
}

// after
public function index(SearchRequest $request): JsonResponse
{
    // リクエストデータの操作をリクエスト側に移動、クエリビルダを組み立てる操作をモデル側に移動
    $items = Item::with('detail')->search($request->filters())->get();
    // レスポンスを組み立てる操作をリソース側に移動
    return ItemResource::collection($items);
}
```

いずれの例でも、処理が短くなっています。処理を分割して各々を短くすることで、読みやすく変更しやすい関数にすることができます。

4.3. フレームワークに任せる部分を見極める

フレームワークにできることをわざわざ自分で作ってしまうと、コード量が多くなったり、同じような処理があちこちに点在して不具合の原因になることがあります。Laravel には「フレームワークでこれくらいやってくれないかな」というような処理はたいていやってくれます。普段から公式ドキュメントやソースコードを読む習慣を持ち、まずはフレームワーク側でできないか検討してみてください。

4.3.1. ルートモデルバインディング

ある API で扱うモデルが予め決まっている場合、そのモデルのインスタンスを自動的に取得することができます。

例として、ECサイトで商品詳細データを返す API のケースを考えてみます。

```
// before
// routing
Route::get('/items/{id}', 'ItemController@show')->name('items.show');
// ItemController
public function show($id)
{
    $item = Item::findOrFail($id);
    // なにか間に処理があればここに書く
    return $item;
}

// after
// routing
Route::get('/items/{item}', 'ItemController@show')->name('items.show');
// ItemController
public function show(Item $item)
{
    // なにか間に処理があればここに書く
    return $item;
}
```

これくらい短い例だとあまり恩恵がわからないかもしれませんが、 before のケースで `findOrFail` ではなく間違って `find` を使った場合、 `$item` が null になる可能性があり、そのあとの処理で `$item` を参照するとサーバーエラーになってしまいます（ `findOrFail` を使えば 404 で返ります）。

4.3.2. 追加でバリデーションを行う

```
// before
class SomeController
{
    public function someAction()
    {
        $validator = Validator::make(...);

        $validator->after(function ($validator) {
            if ($this->somethingElseIsInvalid()) {
                $validator->errors()->add('field', 'Something is wrong with this field!');
            }
        });

        if ($validator->fails()) {
            //
        }

        // ここからメインの処理
    }
}

// after
class SomeRequest extends FormRequest
{
    public function withValidator(Validator $validator)
    {
        $validator->after(function ($validator) {
            if ($this->somethingElseIsInvalid()) {
                $validator->errors()->add('field', 'Something is wrong with this field!');
            }
        });
    }
}
```

before のコードは、公式ドキュメントに掲載されている例ですが、バリデーションの処理がリクエストクラスで完結せず、コントローラーで行っているために、わざわざ明示的に `Validator::fails()` メソッドを呼ばなくてはいけなくなっています。バリデーションが複雑になってくると `Validator::after()` に与えるクロージャが肥大化し、可読性が悪くなってしまいます。 after のコード例では、 `FormRequest::withValidator()` に追加のバリデーションが閉じ込められているので、バリデーションが増えてもコントローラーを変更する必要はありません。

4.3.3. エラーハンドリングを上手に行う

アプリケーションでエラーが発生したとき、どういった形でユーザーにエラー情報を伝えるか、というのは多くの場合難しい処理になりますが、Laravel ではアプリケーションでエラーが発生した場合に、例外の種類に応じてよしなにやってくれることがあります。

インストール直後の状態では、ModelNotFoundException, AuthenticationException, ValidationException などはアプリケーションのどこかでこれらの例外が発生すると自動的に、それぞれ 404: Not Found, 401: Unauthorized, 422: Unprocessable Entity のレスポンスを返します。アプリケーションの開発者は明示的に例外処理を記述したり、レスポンスデータを組み立てる必要はありません。

それを応用して、なんらかのデータの不整合が合った場合に、409: Conflict を返すようにしてみましょう。

```
// before
class SomeController
{
    public function __invoke(SomeModel $some)
    {
        $something = $some->doSomething();
        if (!$something->validState()) {
            return response()->json('なんかおかしい！', 409);
        }
        // 続き
    }
}

// after
class SomeController
{
    public function __invoke(SomeModel $some)
    {
        // どこかで例外が発生しても Laravel がちゃんとキャッチして適切なレスポンスを返してくれるので面倒を見なくていい
        $something = $some->doSomething();
        // 続き
    }
}

// app/Exceptions/Handler.php
class Handler
{
    protected function prepareException(Throwable $e)
    {
        $e = parent::prepareException($e);
        if ($e instanceof ConflictException) {
            $e = new ConflictHttpException($e->getMessage(), $e);
        }
        return $e;
    }
}

// app/Exceptions/ConflictException.php
class ConflictException extends RuntimeException {
}

// app/Models/SomeModel.php
class SomeModel
{
    public function doSomething()
    {
        if ($this->validState()) {
            throw new ConflictException('なんかおかしい！');
        }
        // 省略
    }
}
```

エラーの種類が増えてくると、どういう状態のときにどういうエラーを返せばいいか、その都度悩むことになりがちです。こういうケースではこういうエラーとあらかじめ決めておくことで、各コントローラーでエラーハンドリングをする必要がなくなるので、コード量が少なくて済みます。

4.4. フレームワークの使い方を統一する

Laravel は比較的自由度の高いフレームワークなので、ある処理を行うために、複数のやり方があることが多いです。どちらでもいいケースもありますし、こちらのほうがいい、というケースもあります。例えば前述のバリデーションにしても、コントローラーで書くかリクエストで書くか、選ばなくてはなりません。ここでは、複数の方法を示し、どちらの方法がいいか、理由を添えてコードで説明します。

4.4.1. バリデーションはリクエストに書く

バリデーションをコントローラーに書くかリクエストに書くか、というのは本質的にはどちらでも変わらないですが、前述の通り、追加のバリデーションが増えたりすると、コントローラーのコード量が増えますので（コード量の多いコントローラーはファットコントローラーと呼ばれ、避けられる傾向にあります）、強いてどちらかに統一するなら、リクエストに書きましょう。

参考）Laravel で Fat Controller を防ぐ 5 つの Tips - Qiita

4.4.2. シングルアクションコントローラーを利用する

シングルアクションコントローラーとは、コントローラークラスに `__invoke()` という PHP の特殊なメソッドを唯一の公開メソッドとし、単一のルートに割り当てられるコントローラーのことです。

ルーティング（URL）とコントローラーの対応が難しい場合があります。あるリソースに対する操作で、いわゆる CRUD に該当しないものが増えたとき、ひとつのコントローラーにメソッドがたくさんできてしまうことがあります。そういうときに備えて、以下のメソッド以外をシングルアクションコントローラーにする、というルールを設けておくといいでしょう。

同じコントローラーに書く処理

- index（リソースの一覧を取得する）
- create（リソースの新規作成用のビューを表示する）
- store（リソースを新規作成する）
- show（単一のリソースを表示する）
- edit（リソースの編集用のビューを表示する）
- update（リソースを更新する）
- delete（リソースを削除する）

これら7つのメソッドは、artisan コマンドでコントローラーを作成するときに、 `--resource` パラメータを付与して実行したときに自動的に作成されるものです。

```
# php artisan make:controller UserController --resource
```

それ以外の処理は以下のように、シングルアクションコントローラーにしましょう。例として、ユーザーの退会処理を行う API で考えてみます。退会は削除とイコールではなく、ユーザーが希望したタイミングでいつでも復会できるものとします。

```
// route
// シングルアクションコントローラーの場合、クラス名だけでいい
Route::post('/users/leave', 'Users\LeaveController')->name('users.leave');

// controller
class LeaveController extends Controller
{
    // このメソッドが自動的に呼ばれる
    public function __invoke()
    {
        // 退会処理を行う
    }
}
```

バリデーション同様、コントローラーをスリムに保つために分割します。

4.4.3. アクセサを使う

アクセサは Eloquent の機能で、特定のアトリビュートの中身を書き換えて返したり、データベースにカラムはない

がアトリビュートのように扱いたいときに使いますが、単なるメソッドで代用が可能です。アクセサにするとあたかもプロパティのように扱える反面、処理が追いにくくなるので、デメリットもあります。コードで比較してみましょう。

```
// メソッド
class Avatar
{
    public function getUrl(): string
    {
        if ($this->url) {
            return $this->url;
        }
        return '/images/no-image.png';
    }
}

// 利用側
<img src="{{ $avatar->getUrl() }}" alt="avatar">

// アクセサ
class Avatar
{
    public function getUrlAttribute(string $url): string
    {
        if ($url) {
            return $url;
        }
        return '/images/no-image.png';
    }
}

// 利用側
<img src="{{ $avatar->url }}" alt="avatar">
```

MPA のように、サーバサイドでレンダリングする場合はどちらでも可ですが、API レスポンスのように、最終的なデータを返さなくてはならないようなケースでは、アクセサを使ったほうがひと手間減らせるでしょう。

```
// メソッド
class Avatar
{
    public function getUrl(): string
    {
        if ($this->url) {
            return $this->url;
        }
        return '/images/no-image.png';
    }
}

// 利用側
$user = Auth::user();
// 変換の処理が必要
$user->avatar->url = $user->avatar->getUrl();
return $user;

// アクセサ
class Avatar
{
    public function getUrlAttribute(string $url): string
    {
        if ($url) {
            return $url;
        }
        return '/images/no-image.png';
    }
}

// 利用側
$user = Auth::user();
// レスポンスを生成する際に toArray() が呼ばれ url は自動で変換される
return $user;
```

Vue.js に算出プロパティというものがありますが、それと同じ感じで、プロパティを加工して呼び出し側に渡すようなシチュエーションでは、このようにアクセサを使ったほうが、モデルを呼び出す側がわざわざ変換しなくて済むので便利です（データを使う側はそれが算出プロパティかどうかを気にする必要はありません、ほとんどの場合は）。前述のとおり、処理の流れが追いにくいというデメリットはありますが、アクセサを使う方式で統一して

しまってもメリットがそれを上回ると思います。

ここまでのまとめ

Laravel は巨大なフレームワークなので、出来ること/覚えることがたくさんあります。最初からエキスパートになる必要はありませんが、コードを書く際の指針として、

まずは自分で実装する前に「これフレームワークでできないかな？」と調べてみること、複数書き方がある場合に「どっちがいいかな」と考えてみること

この2つが大事なのではないかと思います。

---

5. おわりに

Laravel でウェブアプリケーションをつくる際、より良いコーディングをするためのガイドとして、いくつかのトピックについてお話しました。

ただ PHP の書き方、Laravel の使い方を識るだけではなく、設計のスキルをプラスすることで、より正確で不具合の少ない、読みやすいコードになり、結果的に「正しく動き続ける」アプリケーションにできるのではないかと思います。

最後にもう一度、本講義で定義したより良いコーディングをする目的を書いておきます。

なぜソフトウェアを「正しく」作ることが求められるのか？

- 顧客の求めているものを作ることで、顧客に価値を最大限に提供できるから
- 意図したとおりに動くものを作ることで、不具合をなくすことができるから
- 意図を理解しやすいものを作ることで、メンテナンスしやすくなるから

ソフトウェアを「正しく」作るとなにがうれしいのか？

- お客さんにいいものが出来たと喜んでもらえるとうれしい
- 不具合が少なくなればトラブルが減って平和になるのでうれしい
- 他人が書いたコードが読みやすいとどういう動きになっているか悩まなくていいのでうれしい

お客さんのため、チームメンバーのため、なにより自分自身のために、良いソフトウェアを作っていきましょう！

Last updated 2020-06-30 17:27:59 +0900