第8章（後半）
密結合と責務

この章の目的

◆前章では密結合が引き起こす問題について学んだ
◆本章ではさらに各事象にいつて深掘りする

内容

内容とキーワード

- ◆継承に絡む密結合
- ◆委譲
- ◆インスタンス変数毎に分割可能なロジック
- ◆何でもpublicで密結合
- ◆privateメソッドだらけ
- ◆高凝集の誤解から来る密結合
- ◆スマートUI
- ◆その他やばいクラス

継承に絡む密結合

継承に絡む密結合

継承はよっぽど注意して扱わないと危険であり、継承は推奨しない。

◆継承、使ってますか？

■継承はオブジェクト指向の重要な機能として紹介されり

■入門書でもさらっと紹介されがち

■ただし、継承は問題視や危険視する意見もある

- それをこれから考えていく

継承に絡む密結合

継承には思わぬ問題が潜んでいる。

◆スーパークラス依存
- ■ゲームを例にします
- ■単体攻撃と2回連続攻撃の仕様を決定する

```
class PhysicalAttak {
    int singleAttackDamage() {
        // 単体攻撃のダメージ値を返す
    }

    int doubleAttackDamage() {
        // 2回攻撃のダメージ値を返す
    }
}
```

攻撃のダメージ値について
定義している

継承に絡む密結合

◆スーパークラス依存

■さらに格闘家（職業）の攻撃力はそれぞれに攻撃力を加算する

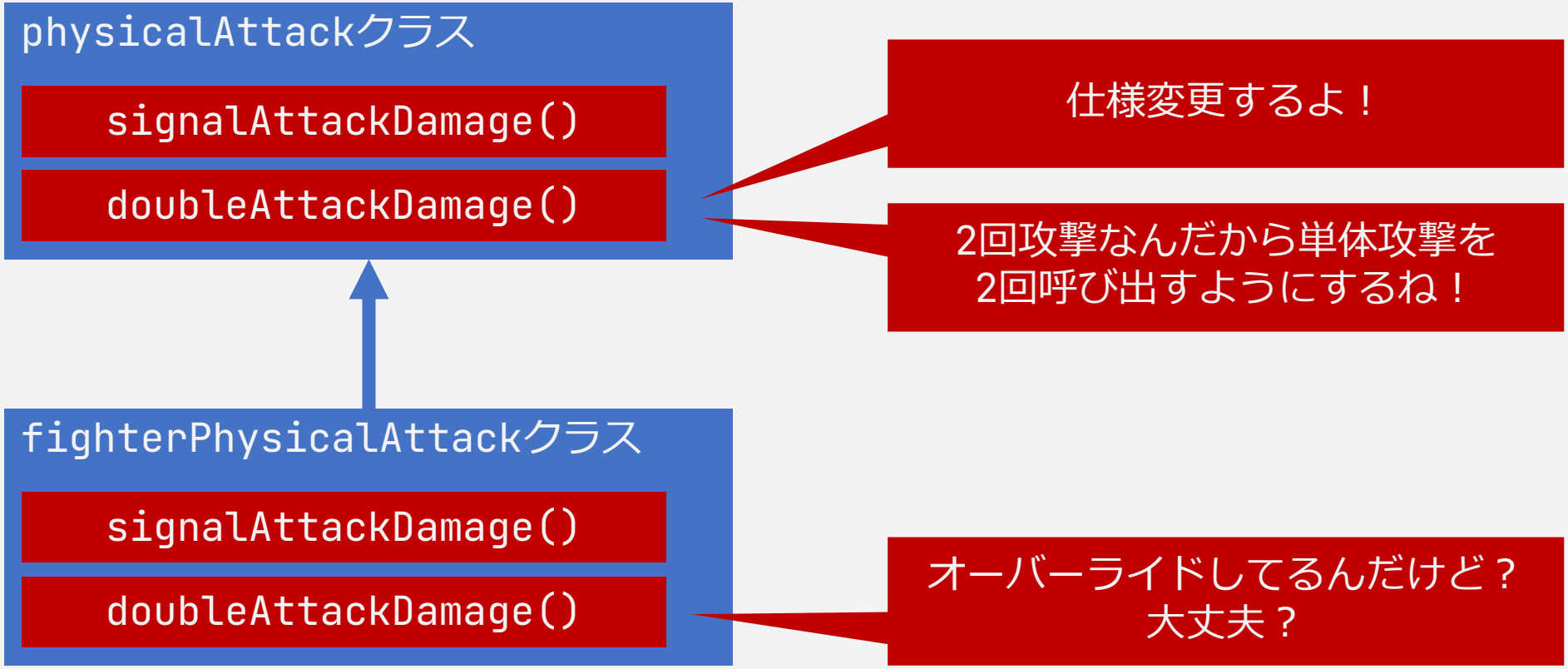
```
class FighterPysicalAttack extends PhysicalAttak {
    @Override
    int singleAttackDamage() {
        return super.signalAttackDamage() + 20;
    }

    @Override
    int doubleAttackDamage() {
        return super.doubleAttackDamage() + 10;
    }
}
```

継承を用いて
攻撃力を追加

一見妥当な継承に見えるか…

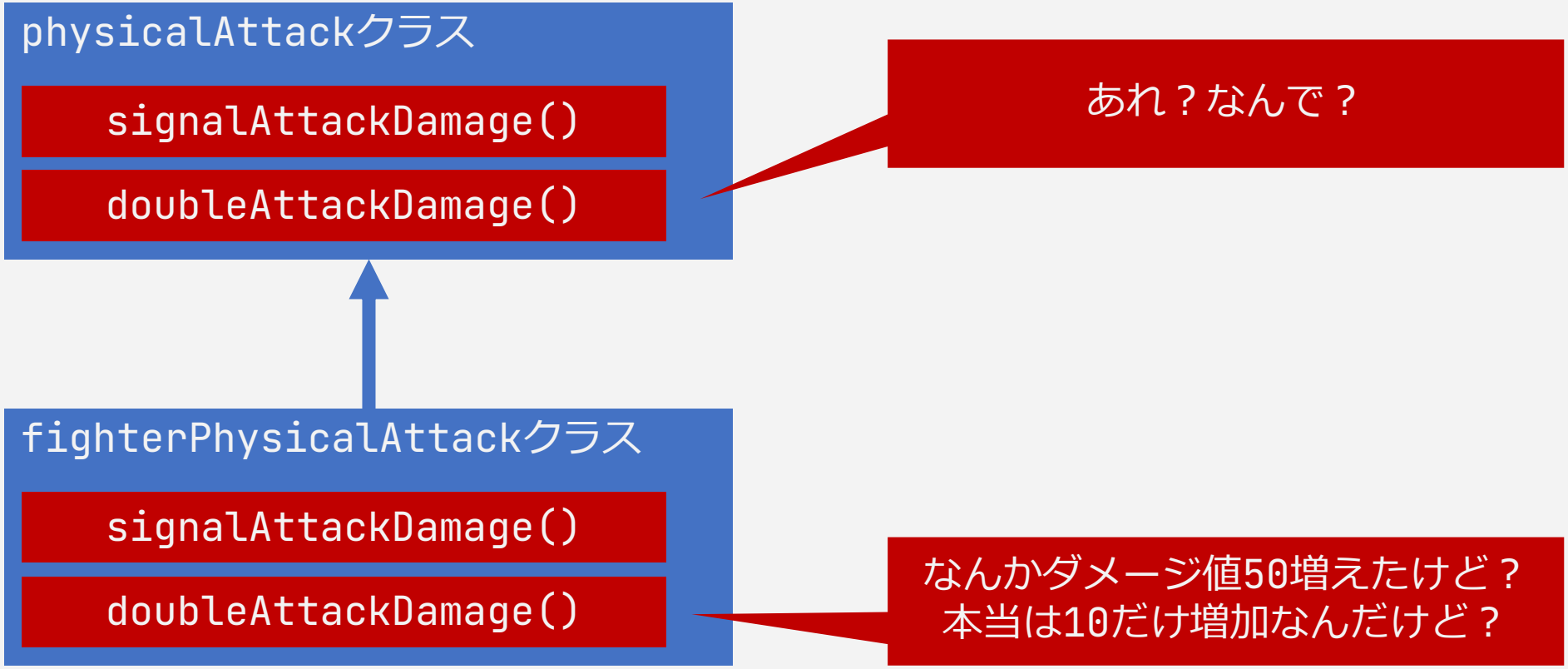
継承に絡む密結合

◆ここでスーパークラスであるPhysicalAttakクラスの仕様を変更しました



継承に絡む密結合

◆するとサブクラスのFighterPhysicalAttackクラスに影響が及んでしまった



継承に絡む密結合

◆スーパークラス依存

■これはスーパークラスに大きく依存しているため起きた

```
class FighterPysicalAttack extends PhysicalAttak {
    @Override
    int singleAttackDamage() {
        return super.signalAttackDamage() + 20;
    }

    @Override
    int doubleAttackDamage() {
        return super.doubleAttackDamage() + 10;
    }
}
```

2回呼びだしたため40加算された

さらに10加算し結果50加算された

スーパークラスの変更がサブクラスにまで影響している

委譲

委譲

スーパークラス依存による密結合を避けるため、継承より委譲が推奨されている。委譲とはコンポジション構造にすることである。

◆継承を使おうとすると、スーパークラスが共通ロジックの置き場として利用されがち

◆継承で無理に処理を共通化しようとして結果密結合となることが多い

◆継承の代わりに委譲を行うことができないか検討してみよう

委譲

◆先ほどの例を委譲で実装してみよう

```
class FighterPysicalAttack {
    private final PhysicalAttack physicalAttack;

    int singleAttackDamage() {
        return physicalAttack.signalAttackDamage() + 20;
    }

    int doubleAttackDamage() {
        return physicalAttack.doubleAttackDamage() + 10;
    }
}
```

委譲（任せる）

これで影響が最小限になった

インスタンス変数ごとに分割可能なロジック

インスタンス変数ごとに分割可能なロジック

以下のコードはあるECサイトで用いられることを想定したコードである。

◆責務が異なるメソッドを詰め込みすぎると

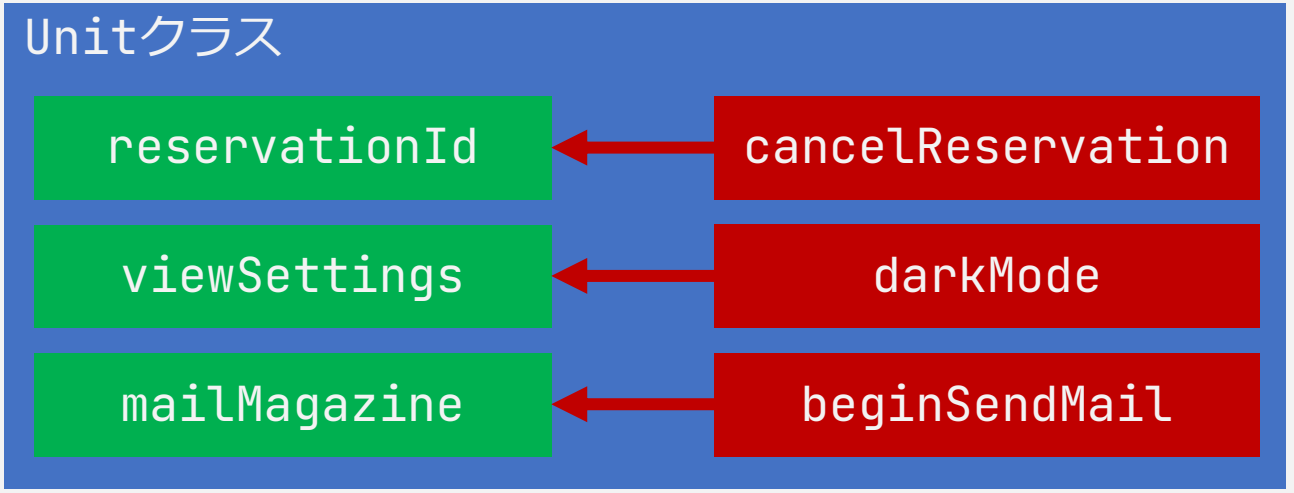
```
class Util {
    private int reservationId; // 商品の予約ID
    private ViewSettings viewSettings; // 画面表示設定
    private MailMagazine mailMagazine; // メールマガジン

    void cancelReservation() {
        // reservationIdを使った予約キャンセル処理
    }

    void darkMode() {
        // viewSettingsを使ったダークモード表示への変更処理
    }

    void beginSendMail() {
        // mailMagazineを使ったメール配信開始処理
    }
}
```

それぞれ責務の異なるメソッドが実装されている。

インスタンス変数ごとに分割可能なロジック

◆関係しないものが同じクラスに混在していないか？
  ■責務が異なるメソッドは同じクラスに実装すべきでない

◆先ほどのコードは以下の様にインスタンス変数とメソッドが1対1で対応している

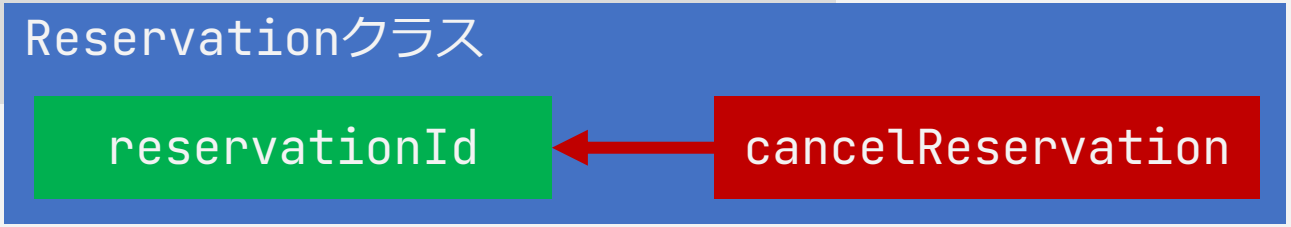


インスタンス変数ごとに分割可能なロジック

◆関係するもの同士でクラスに分類しよう

◆以下は先ほどのコードを分割したもの

```
class Reservation {
    private final int reservationId;
    // 中略
    void cancel() {
        // reservationIdを使った予約キャンセル処理
    }
}
```



他のコードも同様に分割する

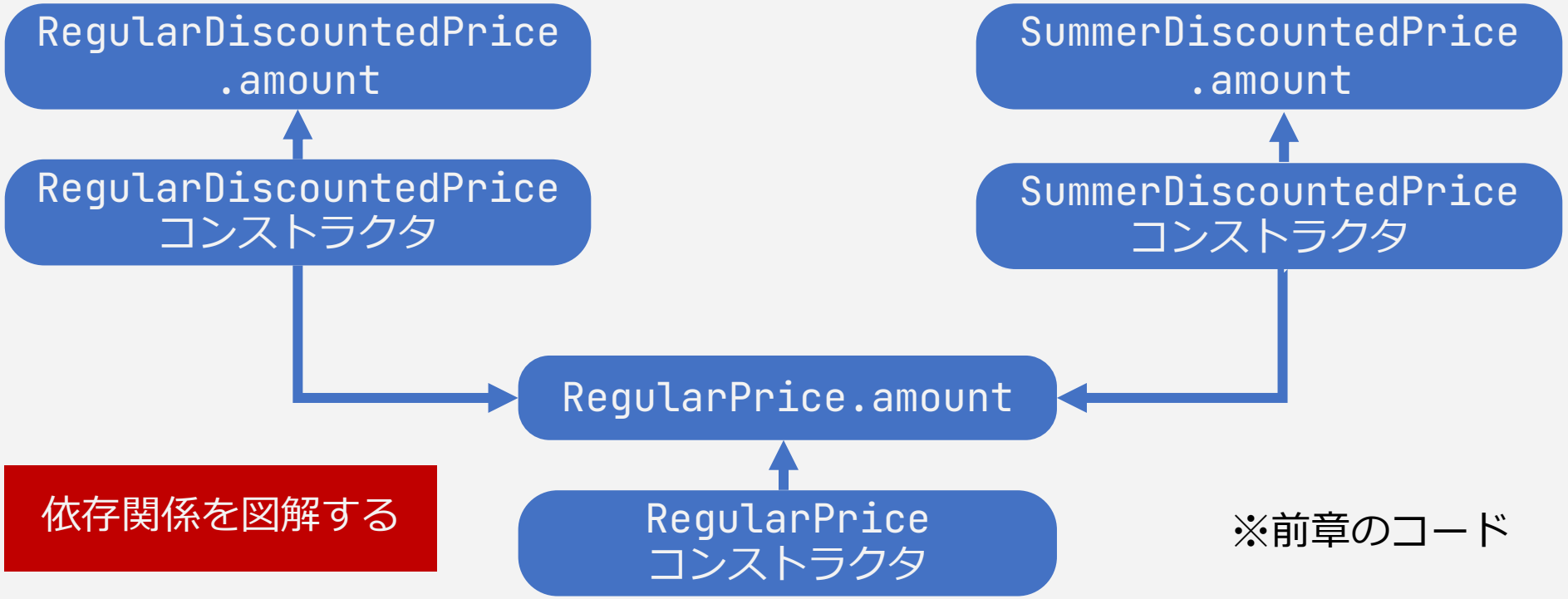
インスタンス変数ごとに分割可能なロジック

依存関係を調べるには影響スケッチを行うとよい。自分で書くのは難しいのでJigなどの解析ツールを使用するとよい。

◆先ほどのクラスは簡単に分類できたが…

■実際のコードは依存関係がもっと複雑である

■上手く分割するには影響スケッチを使用する



依存関係を図解する

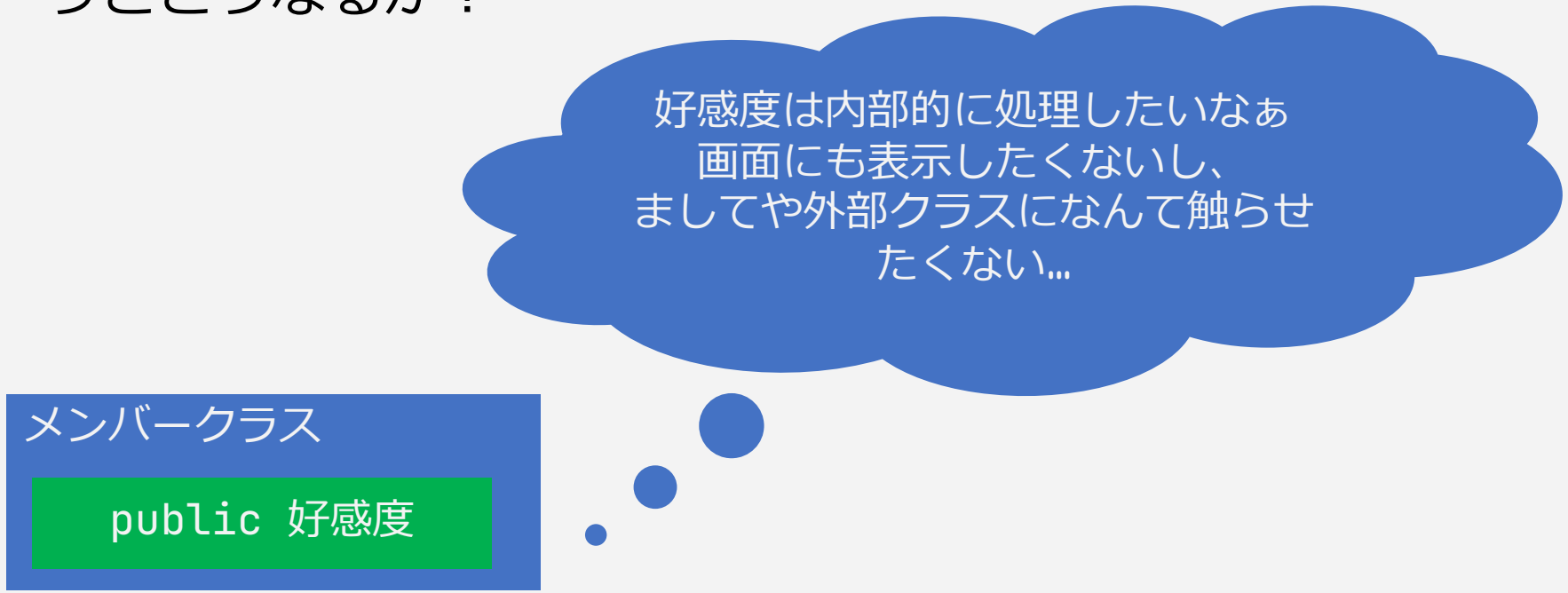
※前章のコード

なんでもpublicで密結合

何でもpublicで密結合

◆public修飾子を付けるとどこからでもアクセスが可能となるが…

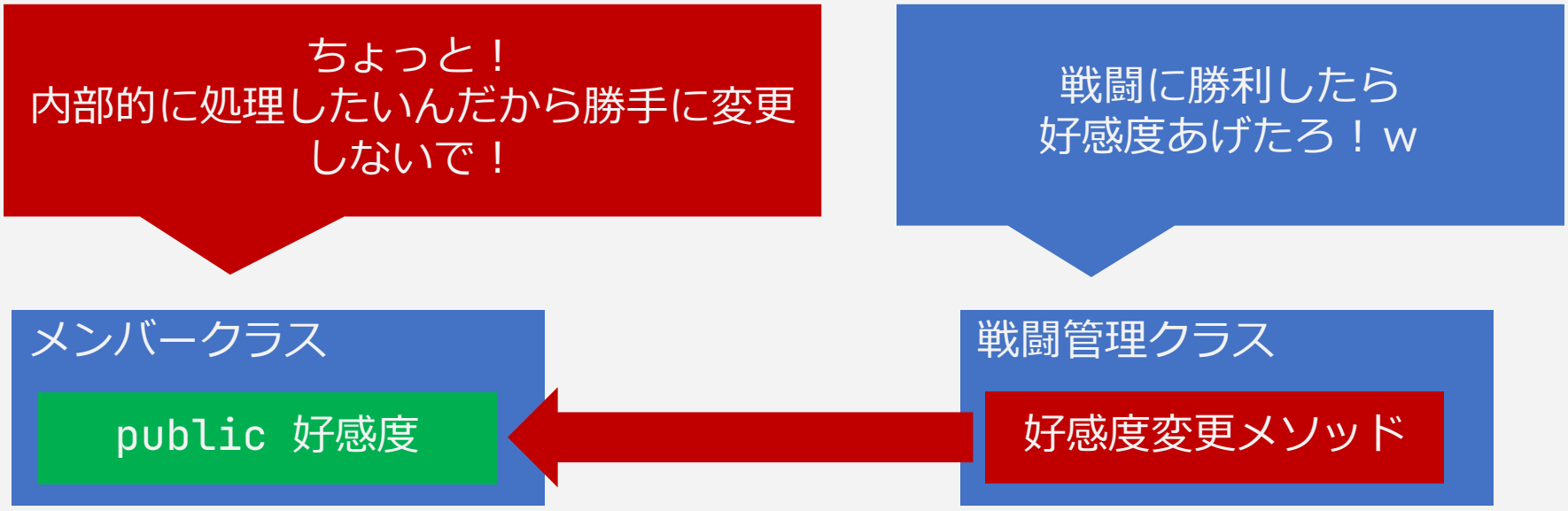
■外部からアクセスされたくないものもpublicにしてしまうとどうなるか？



何でもpublicで密結合

◆何でもpublicにすると関係無いクラスから呼び出されてしまう危険性が高まる

◆これは密結合な状態である



publicなのでアクセス可能

◆パッケージ内では強く関連付くクラス同士を凝集させるが、関係の薄い外部のクラスとは疎結合とする

◆よく入門書などでpublic修飾子が付いた変数・メソッドをよく見かけるからか、public修飾子を気軽に使いがち

適切なアクセス修飾子を使おう

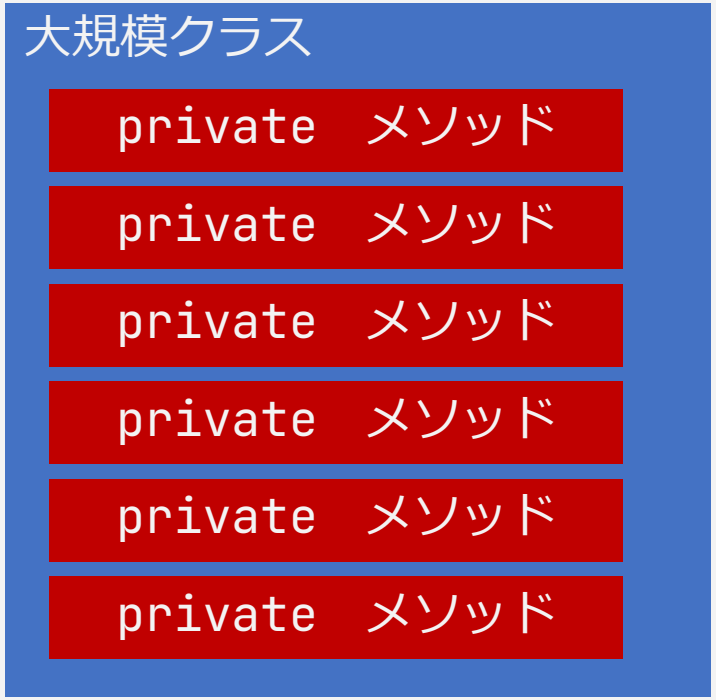
アクセス修飾子	どこからアクセス可能か
public	全てのクラスから
protected	同じクラスもしくは継承クラス
修飾子無し	同じパッケージのクラス
private	同じクラスのみ

privateメソッドだらけ

privateメソッドだらけ

先ほどの項目でなんでもpublicにしないようにと言われたが…

◆publicにしないようにprivateを多用しよう！ってなるかもしれませんが…



他のクラスが使えないように
全部privateで実装したよ！
これで安全！

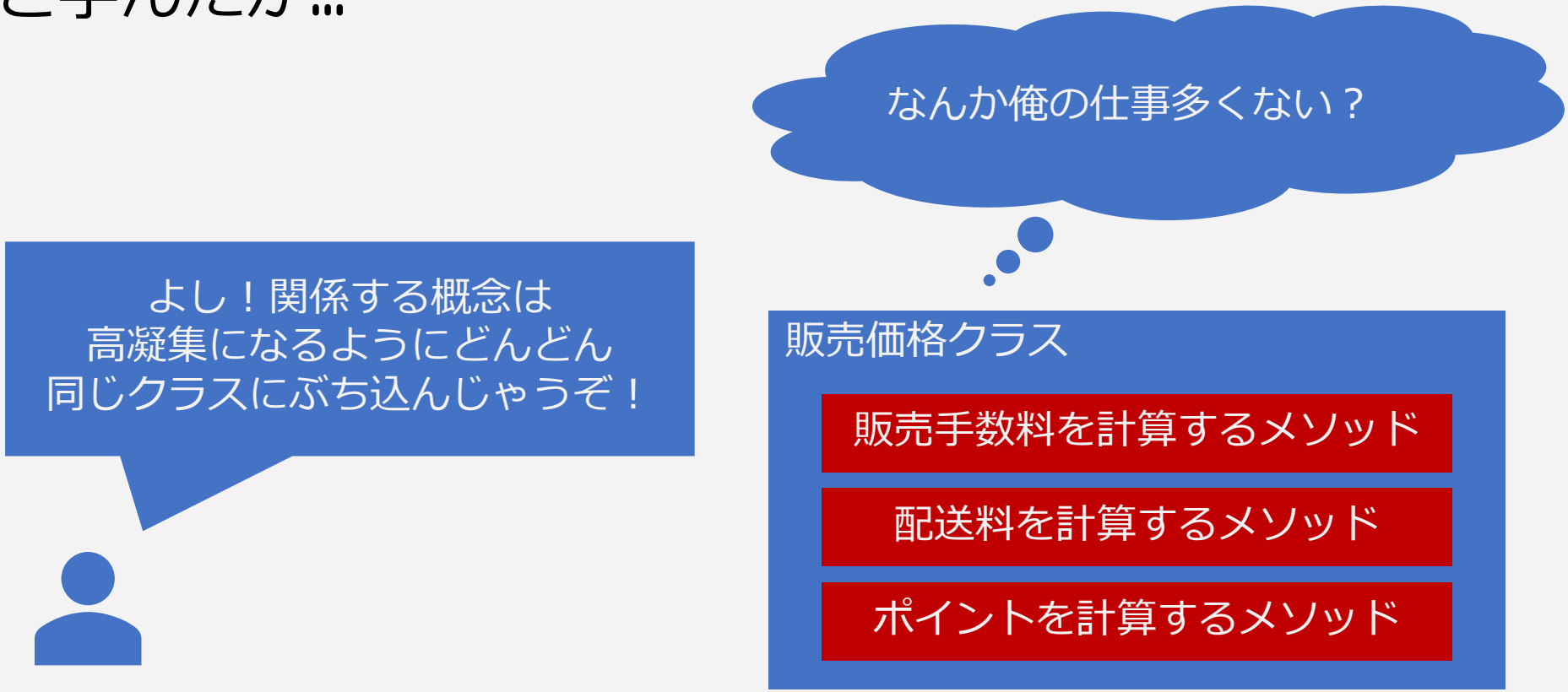
しかし…

たくさんのメソッドがあるということは
多くの責務を負いすぎていませんか？
単一責任の原則を見直しましょう。

高凝集の誤解から来る密結合

高凝集の誤解から来る密結合

◆第五章で凝集度について学び、高凝集の状態が良いと学んだが…

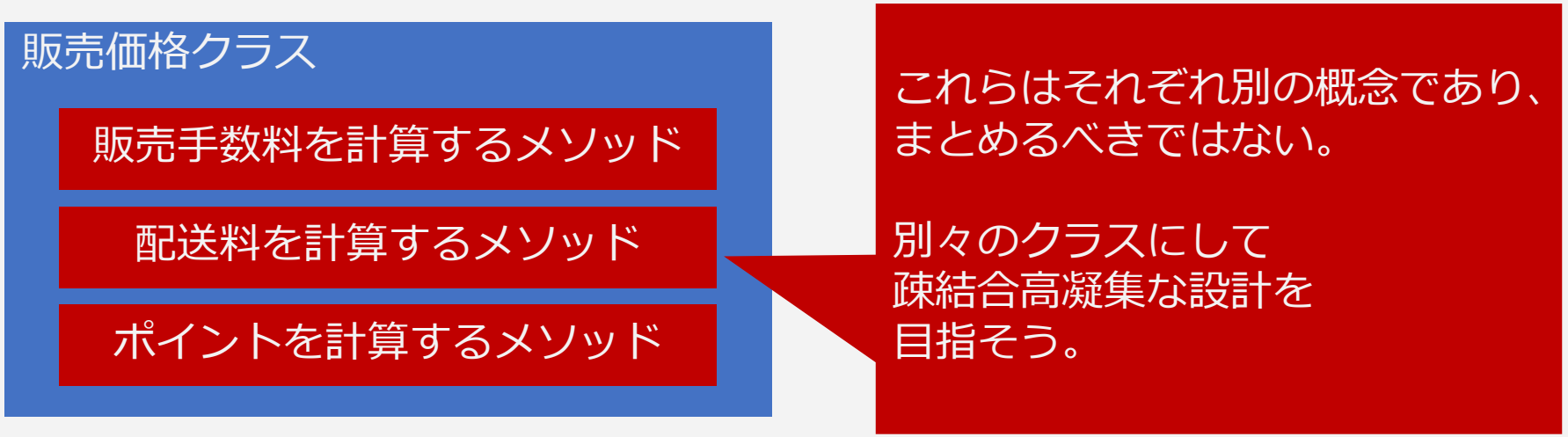


高凝集の誤解から来る密結合

高凝集を意識しすぎて、結果として密結合に陥っているケースは非常に多い。

◆第五章で凝集度について学び、高凝集の状態が良いと学んだが…

◆別の概念を混ぜてしまうと、どこになんのロジックが書かれているか理解が困難になる！

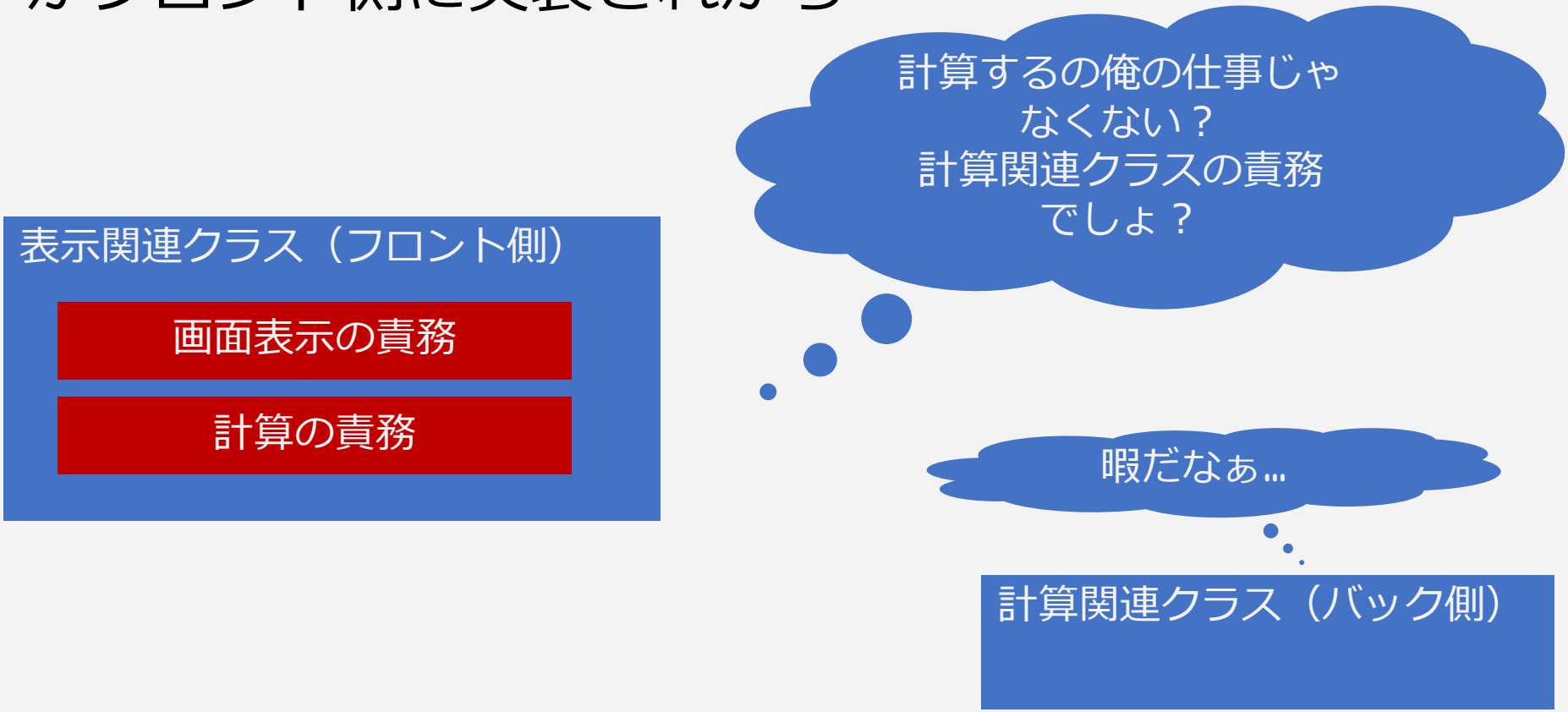


スマートUI

スマートUI

表示関連クラスの中に、表示以外の責務のロジックが実装されている構造をスマートUIと呼ぶ。

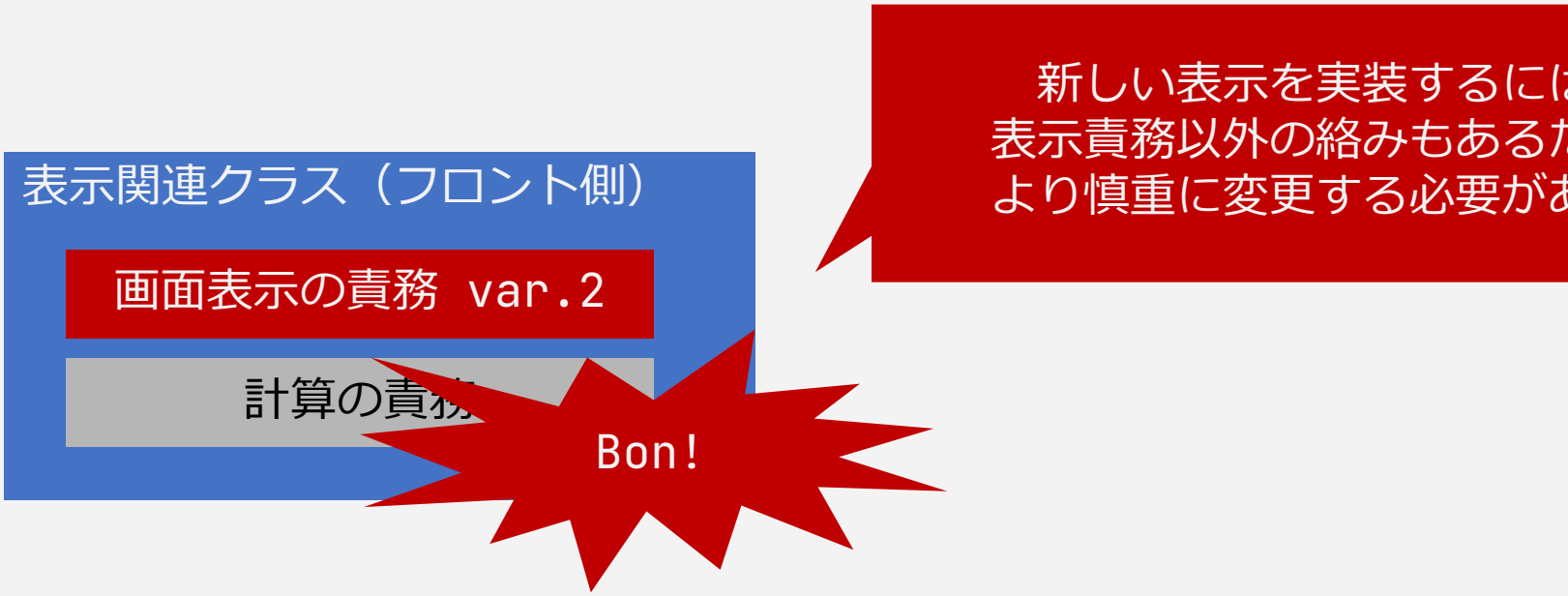
◆とにかくコードを書いていくと複雑な計算ロジックがフロント側に実装されがち



スマートUI

表示関連クラスの中に、表示以外の責務のロジックが実装されている構造をスマートUIと呼ぶ。

◆関係ないロジックが紛れていると、新しい表示に書き換えた際にバグが起こることがある



こうならないためにも表示責務とそれ以外の責務を分けよう。

その他のやばいクラス

巨大データクラス

大量のインスタンス変数を持つクラスを巨大データクラスと呼ぶ。

◆とりあえず変数を追加していったデータクラスはさまざまなデータな置き場となり、巨大化する。

たくさんの変数の置き場所になって便利でしょ！

しかし…

あらゆる場所からアクセスされ、グローバル変数の性質を帯びてくる

注文クラス

- 注文ID
- 注文日時
- 予約日時
- 発注者ID
- 注文状態
- 配送先
- 注文者一覧
- 予約ID
- その他大量の変数

そもそもデータクラスは良くない

トランザクションスクリプトパターン

メソッド内に一連の処理手順が長々と書き連ねているものをトランザクションスクリプトパターンと呼ぶ。

◆数百行以上の長さがあるメソッドは理解が困難

```
メソッド^^
    処理1
    処理2
    処理3
    処理4
    処理5
    処理6
    処理7
    処理8
    処理9
    処理10
    処理11
    処理12
    処理13
    処理14
    処理15
```

長すぎて読む気にもならない…

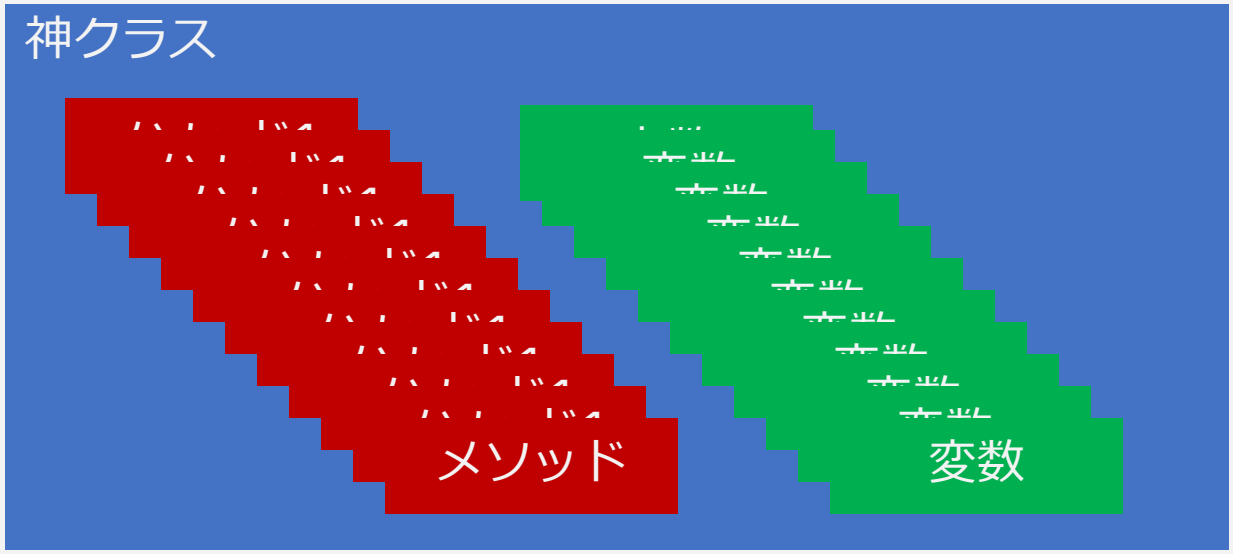
あまりにも長いメソッドは
分割できるはず

神クラス

1クラス内に何千、何万ものロジックを持ち、あらゆる責務のロジックが書き殴られているクラスのこと。

◆神（悪魔）は開発者の時間を奪い、多大な苦労を与えて疲弊させてしまう恐ろしい力を持っている。

言わずもがな存在してはいけない構造である



次の章に向けて

◆さまざまな密結合の事例が出てきましたが、対処法は同じです。

◆オブジェクト指向設計と単一責任の原則い基づいて丁寧に設計しましょう。

◆クラスはどんなに多くても200行までです。

◆10章で学ぶ目的駆動名前設計も大いに役に立ちます。