第5章
低凝集

高凝集な設計を目指そう

この章の目的

低凝集な構造を避けて，高凝集な構造を設計しよう.

◆凝集度について考える

- ■凝集度とは，モジュール内における，データとロジックの関係性の強さを表す指標.
- ■モジュールはクラス，パッケージ，レイヤーなど様々な粒度の解釈がある
- ■高凝集な構造は，変更に強い，望ましい構造.

内容

内容とキーワード

何度も紹介してきましたが…

◆コードが散らかっていると，何がどこにあるか分からなくなる

何度も紹介するということはかなり重要！

staticメソッドの誤用

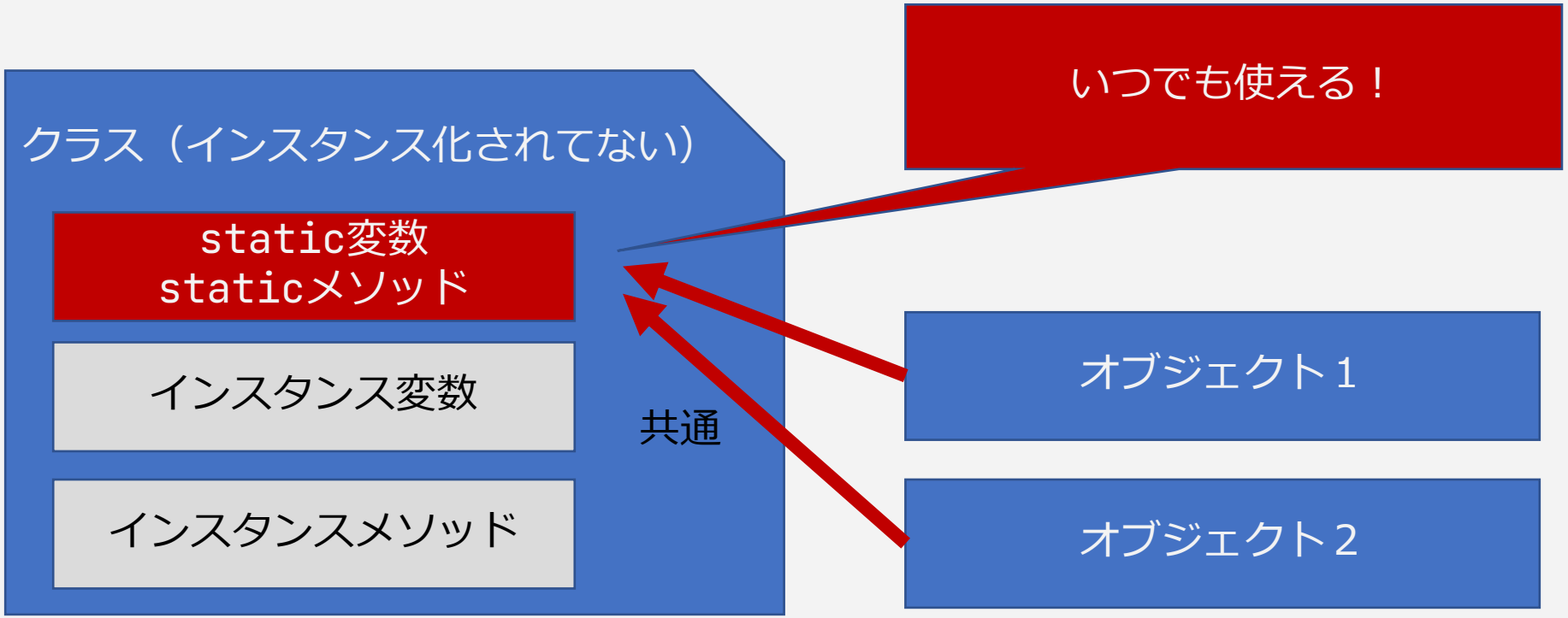
staticメソッドの誤用

説明に入る前に…

◆まてまてstaticなモノってなんだ？

■インスタンス化しなくても使えるメソッドや変数のこと

■インスタンス共通の変数，メソッドになる



staticメソッドの誤用

こうしたstaticなメソッドはデータクラスに対して用いられる傾向がある.

◆例.

扱う変数はこのクラスにはない

```
// 注文を管理するクラス
class OrderManager {
    static int add(int moneyAmount1, int moneyAmount2) {
        return moneyAmount1 + moneyAmount2;
    }
}
```

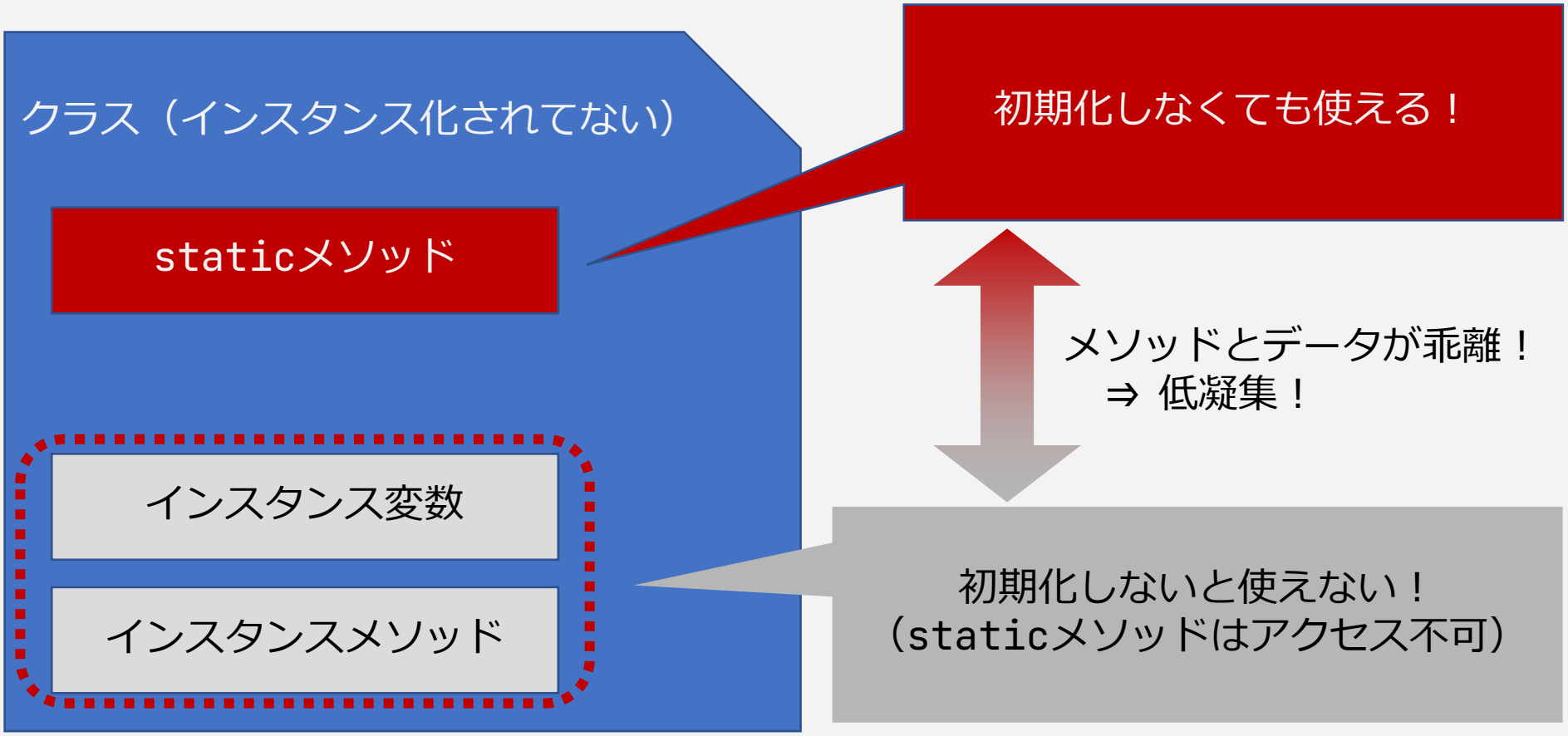
別のクラスの変数を変更している！
（低凝集！）

```
// moneyData1,moneyData2はデータクラス
moneData1.amount = OrderManager.add(
                                moneyData1.amount,moneyData2.amount
                                );
```

staticメソッドの誤用

staticメソッドを使うとメソッドとデータが乖離して低凝集になりがち.

◆staticはインスタンス変数を使えない



staticメソッドの誤用

◆インスタンス変数を使う構造に作り変えよう.

```
// 注文を管理するクラス
class OrderManager {
    static int add(int moneyAmount1, int moneyAmount2) {
        return moneyAmount1 + moneyAmount2;
    }
}
```

```
class Money {
    final int amount;
    // コンストラクタ省略
    Money add(final Money other) {
        return new Money(amount + other.amount);
    }
}
```

同じクラスのインスタンス変数を使用するようになった.
(高凝集)

staticメソッドの誤用

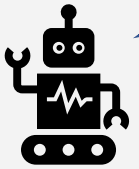
◆インスタンスメソッドのふりをしたstaticメソッド.

```
class PaymentManager {
    // 割引率（インスタンス変数）
    private int discountRate;

    // 省略
    int add(int moneyAmount1, int moneyAmount2) {
        return moneyAmount1 + moneyAmount2;
    }
}
```

インスタンス変数を使っていない！

staticメソッドはインスタンス化が不要で
お手軽に使われがち！

これstaticつけても問題ないよ？
⇒ 実質staticメソッド



IDE

staticメソッドの使い方

ではいつstaticメソッドをつかうのか？

◆staticメソッドの適切な使用例

- ■ログ出力用のメソッド
- ■フォーマット変換用メソッド
- ■ファクトリメソッド（後述）

これらは凝集度とは関係ないためstaticでも問題無い.

初期化ロジックの分散

初期化メソッドの分散

以下は，ギフトポイントを表現するクラスです．一見良さそうに見えますが…?

```
class GiftPoint {
    private static final int MIN_POINT = 0;
    final int value;

    GiftPoint(final int point) {
        if(point < MIN_POINT) {
            throw new IlligalArgumentException(“不正なポイントです．”);
        }
        value = point;
    }

    GiftPoint add(final GiftPoint other) {
        return new GiftPoint(value + other.value);
    }
    // 以降，インスタンスメソッドが続く．
}
```

初期化メソッドの分散

◆初期化メソッドが別実装になることがある.

※外部のコード

標準会員は3000ポイントね！

```
GiftPoint standardMemberShipPoint = new GiftPoint(3000);
```

プレミアム会員は10000ポイントね！

```
GiftPoint premiumMemberShipPoint = new GiftPoint(10000);
```

コンストラクタを公開すると,
外部に関連ロジックが実装される可能性がある！

初期化メソッドの分散

初期化の方法を外部に実装したことで，初期化メソッドとクラスが乖離し低凝集となる.

◆外部に実装すると低凝集を招く.

※外部のコード

```
GiftPoint standardMemberShipPoint = new GiftPoint(3000);
```

```
GiftPoint premiumMemberShipPoint = new GiftPoint(10000);
```

なんか外部でポイントが定義されてない？

GiftPointクラス

ファクトリメソッド

先ほどの初期化メソッドの分散を防ぐには，コンストラクタをprivateにし，目的別のファクトリメソッドを用意すればよい．

```
class GiftPoint {
    private static final int MIN_POINT = 0;
    private static final int STANDARD_MEMBERSHIP_POINT = 3000;
    private static final int PREMIUM_MEMBERSHIP_POINT = 10000;
    final int value;
    private GiftPoint(final int point) {
        if(point < MIN_POINT) {
            throw new IlligalArgumentException(“不正なポイントです．”);
        }
        value = point;
    }
    static GiftPoint forStandardMembership() {
        return new GiftPoint(STANDARD_MEMBERSHIP_POINT);
    }
    static GiftPoint forPremiumMembership() {
        return new GiftPoint(PREMIUM_MEMBERSHIP_POINT);
    }
}
```

ここで初期化メソッドを定義

ファクトリメソッド

◆ファクトリメソッドを使うことで初期化メソッドも関連するクラスに実装され，凝集度が高まった.

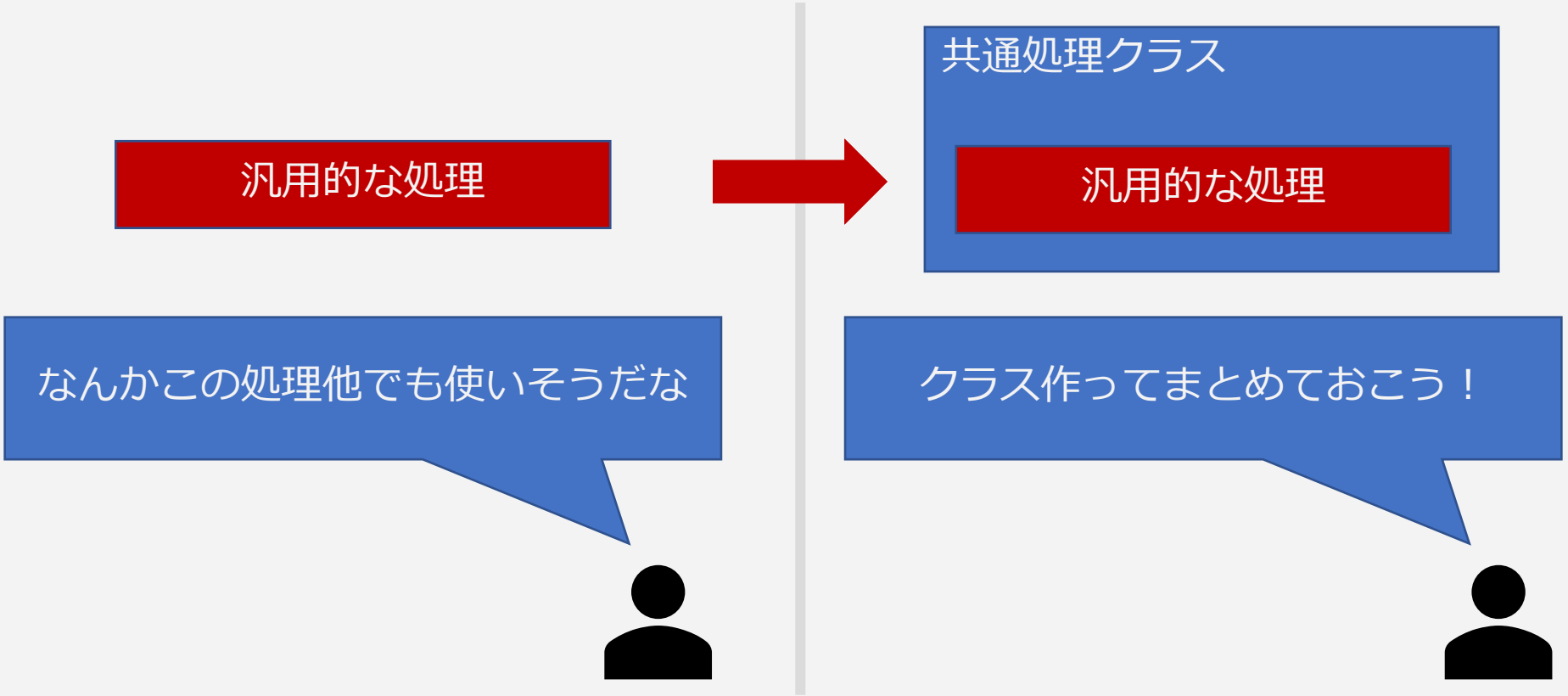
◆生成ロジック（初期化メソッド）が増えすぎた場合はさらに生成専門のファクトリクラスを作ることを検討しよう.

※ファクトリクラスもデザインパターン

共通処理クラス (Common/Util)

共通処理クラス

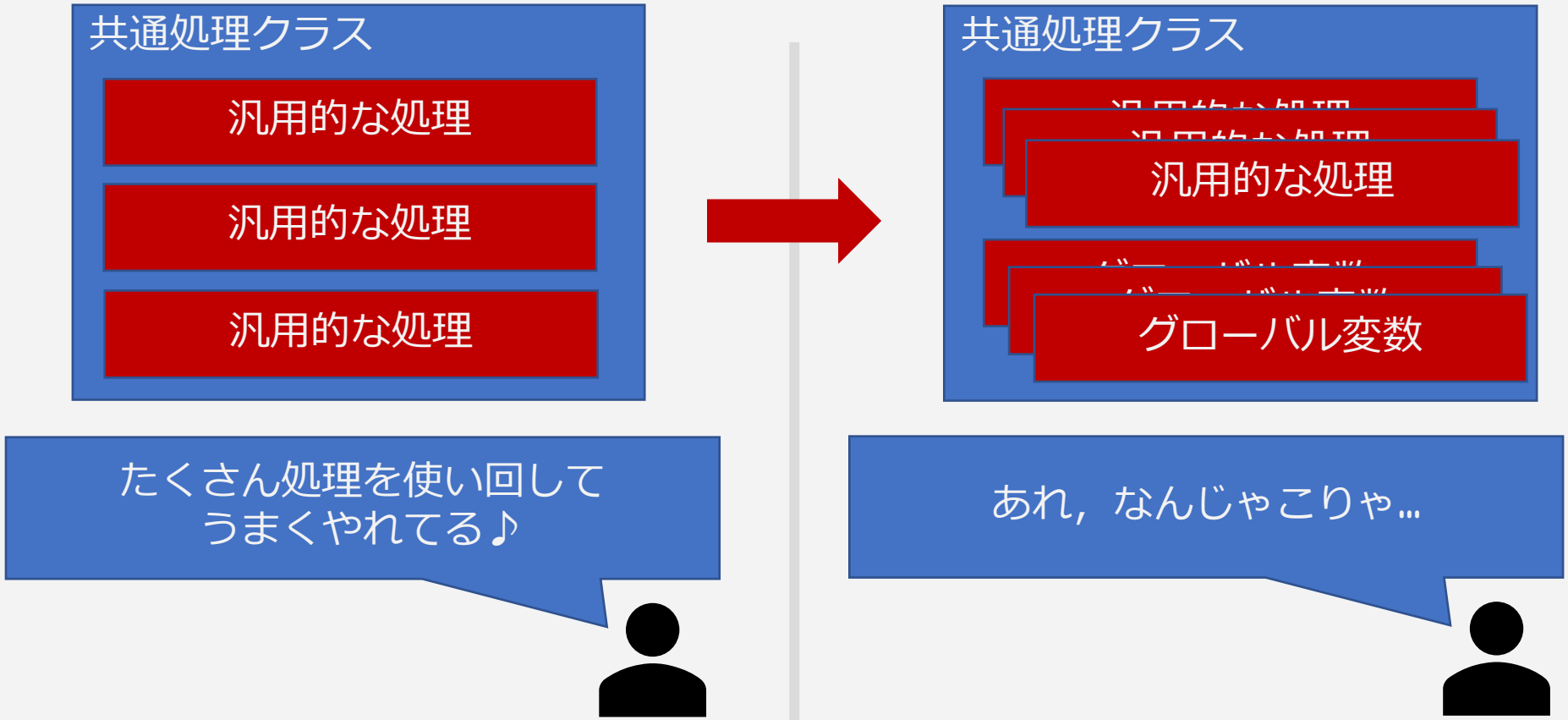
◆共通処理の置き場所として用意されたクラス.



共通処理クラス

共通処理クラスはその特性上いろいろなメソッドが雑多に置かれがちで，低凝集となる．またグローバル変数が生まれてしまうことも．

◆共通処理の置き場所として用意されたクラス.



共通処理クラスを作るのはやめよう！

横断的関心事

様々なユースケースに広く横断する事柄を横断的関心事と呼ぶ.

◆横断的関心事に関する処理であれば，共通処理としてまとめても良い.

- ■ログ出力
- ■エラー検出
- ■デバッグ
- ■例外処理
- ■キャッシュ
- ■同期処理
- ■分散処理

```
try {
    shoppingCart.add(product);
} catch (IlligalArgumentException e) {
    Logger.report(“買い物かごに商品を追加できません. ”);
}
```

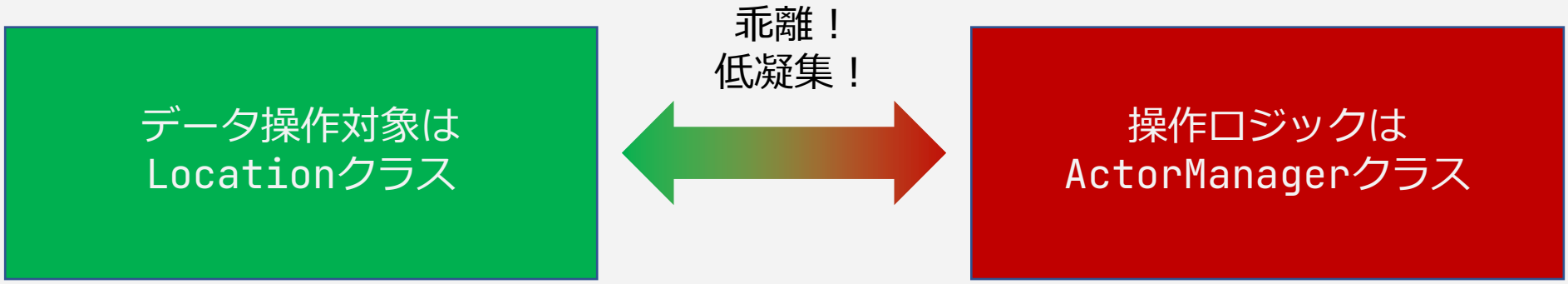
結果を返すために引数を使用しない

出力引数

出力として用いる引数を出力引数と呼ぶ.

◆以下のメソッドは出力引数を持ち，結果は返さない.

```
class ActorManager {

    void shift(Location location, int shiftX, int shiftY) {
        location.x += shiftX;
        location.y += shifty;
    }
}
```



出力引数

引数は入力値として扱いましょう．出力値として扱ってしまうと，引数が入力なのか出力なのかいちいち調べる必要が出てくる．

◆以下のメソッドは出力引数を持ち，結果は返さない．

```
class ActorManager {

    void shift(Location location, int shiftX, int shiftY) {
        location.x += shiftX;
        location.y += shifty;
    }
}
```

おいおいメソッド内部で変更してるのかよ！
いちいち調べてられるか！

出力引数

◆引数を変更しない構造へ改善しよう.

```
class Location {
    final int x;
    final int y;
    Location(final int x, final int y) {
        this.x = x;
        this.y = y;
    }
    Location shift(final int shiftX,final int shifty) {
        final int nextX = x + shiftX;
        final int nextY = y + shiftY;
        return new Location(nextX,nextY);
    }
}
```

多すぎる引数

多すぎる引数

intやfloat,boolean,String(※)といったプログラミング言語が標準で用意したデータ型をプリミティブ型と呼ぶ.

◆プリミティブ型執着

- ■プリミティブ型を濫用したコードをプリミティブ型執着と呼ぶ.
- ■プリミティブ型を濫用するとデータのありかとデータを使って制御するロジックのありかがバラバラになりがち.(クラスではないのでロジックを持たない.)

- ■メソッドの引数も多くなりがち.

なんじゃこりゃあ！

多すぎる引数

intやfloat,boolean,String(※)といったプログラミング言語が標準で用意したデータ型をプリミティブ型と呼ぶ.

◆プリミティブ型執着

訳分からん…

```
int recoverMagicPoint(int currentMagicPoint,
                      int originalMaxMagicPoint,
                      List<Integer> maxMagicPointIncrements,
                      int recoveryAmount) {
    int currentMaxMagicPoint = originalMaxMagicPoint;
    for(int each : maxMagicPointIncrements) {
        currentMaxMagicPoint + each;
    }
    return Math.min(currentMagicPoint + recoveryAmount,
                    currentMaxMagicPoint);
}
```

魔法力を回復するメソッドを実装しました！
引数多くなりすぎたけど,
プリミティブ型使うならこうなるよね？

意味のある単位毎にクラス化しよう

プリミティブ型で受け取るのではなく，型を用意して型で受け取ろう．そのためには意味のある単位毎に変数をクラス化しよう．

```
class MagicPoint {
    private int currentAmount;
    private int originalMaxAmount;
    private final List<Integer> maxIncrements;

    // 省略
    int current() {
        return currentAmount;
    }
    int max() {
        int amount = originalMaxAmount;
        for(int each : maxIncrements) {
            amount += each;
        }
        return amount;
    }
    // 省略
}
```

これで魔法力に関するデータがひとつの型にまとまった．

さらに言えば，privateにすることで，他のクラスで関連するロジックを生成できなくしている．

多すぎる引数

型を用意したことで，関連するメソッドがクラスに凝集された．

```
int recoverMagicPoint(int currentMagicPoint,
                      int originalMaxMagicPoint,
                      List<Integer> maxMagicPointIncrements,
                      int recoveryAmount) {
    // 省略
}
```

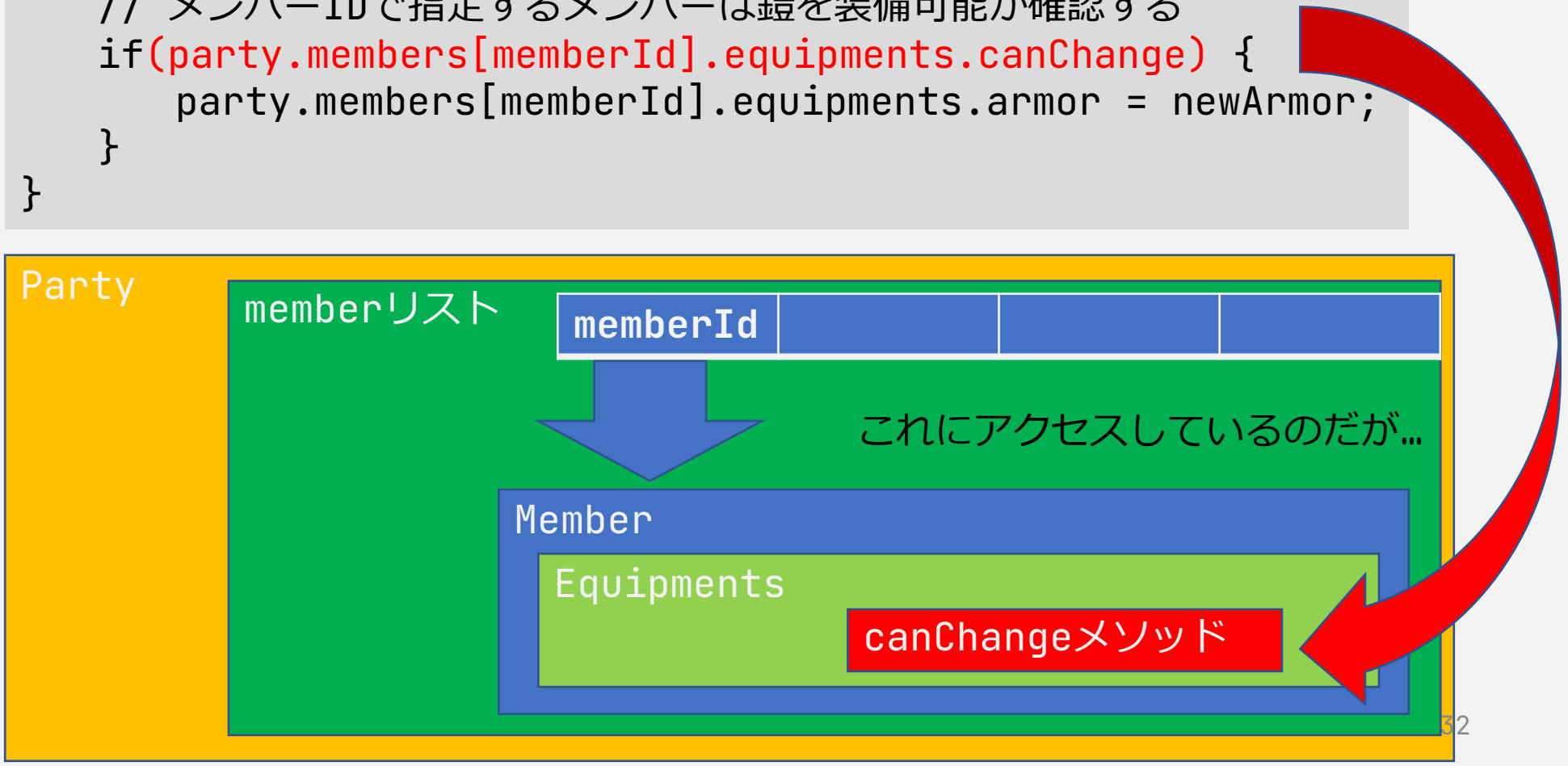
```
// 魔法力クラスに所属しているため，引数は回復量だけで良い

int recoverMagicPoint(final int recoveryAmount) {
    // 省略
}
```

メソッドチェイン

メソッドチェイン

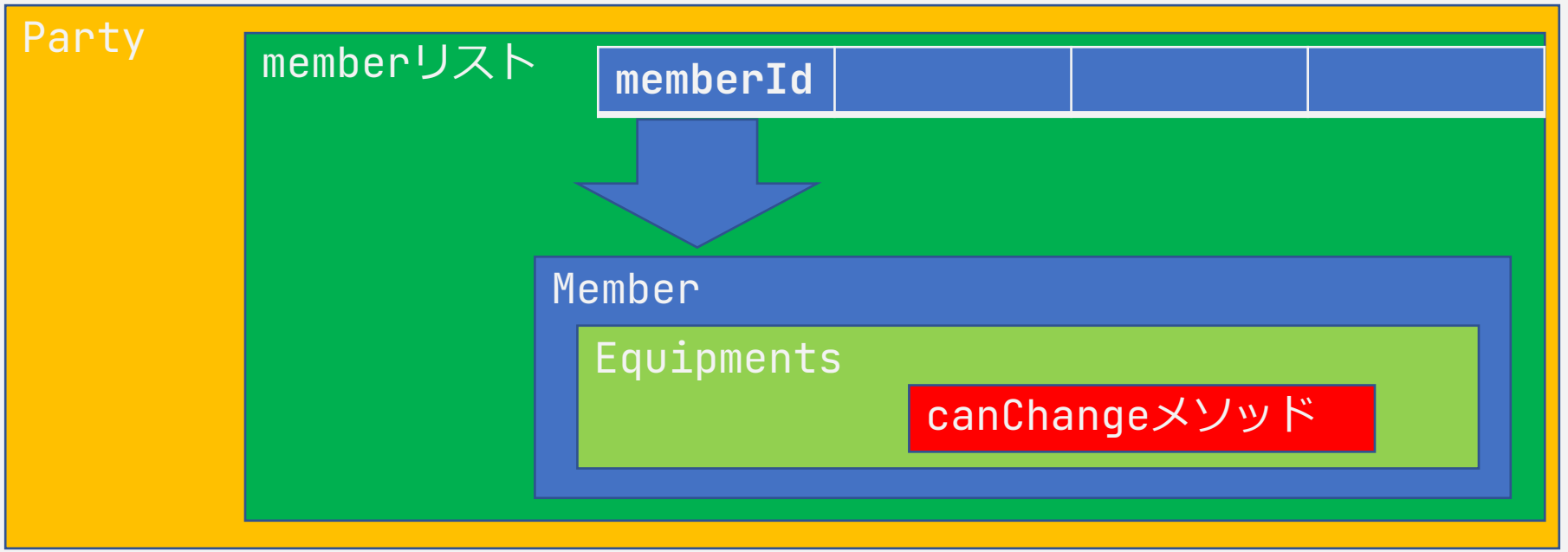
ドットで数珠つなぎにして戻り値の要素に次々アクセスする書き方をメソッドチェインと呼ぶ.

```
// 鎧を装備する
void equipArmor(int memberId, Armor newArmor) {
    // メンバーIDで指定するメンバーは鎧を装備可能か確認する
    if(party.members[memberId].equipments.canChange) {
        party.members[memberId].equipments.armor = newArmor;
    }
}
```



メソッドチェイン

ドットで数珠つなぎにして戻り値の要素に次々アクセスする書き方をメソッドチェインと呼ぶ.

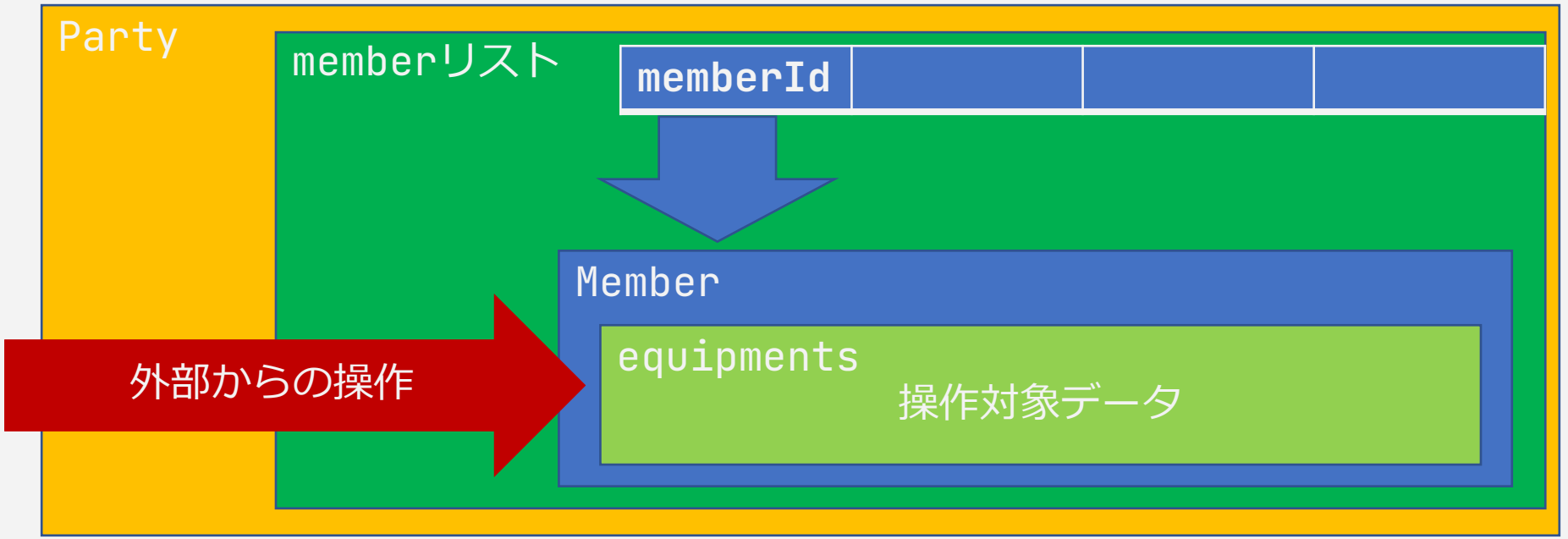



```
if(party.members[memberId].equipments.canChange)
```

この例ではメソッドチェインを用いて，改造構造になっているクラスのかなり奥深くの要素にアクセスしている.

メソッドチェイン

ドットで数珠つなぎにして戻り値の要素に次々アクセスする書き方をメソッドチェインと呼ぶ.




```
party.members[memberId].equipments.armor = newArmor;
```

かなり奥深くの要素を操作することは，操作とその対象のデータが乖離しており，これも低凝集を招く.

デメテルの法則

◆デメテルの法則とは？

■利用するオブジェクトの内部を知るべきではない

■「知らない人に話しかけるな」とも要約される

```
if(party.members[memberId].equipments.canChange)
```

ここまで奥深くの要素まで渡り歩いて
オブジェクトの内部詳細を聞き出そうとしている
⇒ デメテルの法則に反する

考えるな，命じろ

ソフトウェア設計には，考えるな，命じろという格言がある．

◆考えるな，命じろとは？

■他のオブジェクトの内部状態（変数）を尋ねたり，その状態に応じて呼び出し側が判断したりするのではない．

■呼び出し側はただメソッドを命ずるだけで，命令された側で適切な判断や制御するよう設計する．