

第6章（後半）

条件分岐

この章の目的

- ◆前回に引き続き条件分岐について学ぶ
- ◆ポリシパターンとはなにかにつて学ぶ
- ◆インターフェースを使いこなそう

内容

内容とキーワード

- ◆ポリシーパターン
- ◆リスコフの置換原則
- ◆フラグ引数

前回までの内容

- ◆ Switch文を使って条件分岐を制御すると、重複コードが爆発的に増える可能性がある.
- ◆ Switch文の代わりにインタフェースでの設計を行うことで処理をまとめることができた.

条件分岐の重複とネスト

条件分岐の重複とネスト

インターフェースはswitch文の重複解消以外にも、多重にネストした分岐の解消にも役に立ちます。

◆例えば以下のコードについて考えてみる。

以下の条件を満たす人をゴールド会員とする

1. これまでの購入金額が10万円以上
2. 一ヶ月あたりの購入頻度が10回以上
3. 返品率が0.1%以内である

今までは条件分岐のネストは早期returnで解消！でしたね

※ゴールド会員かを判断するメソッド

```
boolean isGoldCustomer(PurchaseHistory history) {  
    if(100000 <= history.totalAmount) {  
        if(10 <= history.purchaseFrequencyPerMonth) {  
            if(history.returnRate <= 0.001) {  
                return true;  
            }  
        }  
    }  
    return false;  
}
```



ごちゃごちゃ...

条件分岐の重複とネスト

インターフェースはswitch文の重複解消以外にも、多重にネストした分岐の解消にも役に立ちます。

◆次のコードについても考えてみる。

以下の条件を満たす人をシルバー会員とする

1. 一ヶ月あたりの購入頻度が10回以上
2. 返品率が0.1%以内である

これも早期returnで解決...
でもなんかその前にコードが
重複しているような...?

※シルバー会員かを判断するメソッド

```
boolean isSilverCustomer(PurchaseHistory history) {  
    if(10 <= history.purchaseFrequencyPerMonth) {  
        if(history.returnRate <= 0.001) {  
            return true;  
        }  
    }  
    return false;  
}
```

ゴールド会員と一部条件が同じ
⇒ 重複コードが生まれた

ポリシーパターン

ポリシーパターン

条件を部品化し、部品化した条件を組み替えてカスタマイズを可能にする設計パターン。

◆条件をインターフェースで表現してみる

```
interface ExcellentCustomerRule {  
    Boolean ok(final PurchaseHistory history);  
}
```

ルールをインターフェース化したもの。
このインターフェースを実装するクラスは、
購入履歴のインスタンス (PurchaseHistory) を受け取り、
条件を満たすならtrueを返すようなメソッドを持つことを強制。

ポリシーパターン

先ほどのインターフェースを実装したルールをそれぞれクラスとして表現してみる。

```
class GoldCustomerPurchaseAmountRule
implements ExcellentCustomerRule {
    public boolean ok(final PurchaseHistory history) {
        return 100000 <= history.totalAmount;
    }
}
```

ゴールド会員の購入金額ルール

```
class PurchaseFrequencyRule implements ExcellentCustomerRule {
    public boolean ok(final PurchaseHistory history) {
        return 10 <= history.purchaseFrequencyPerMonth;
    }
}
```

購入頻度のルール

```
class ReturnRateRule implements ExcellentCustomerRule {
    public boolean ok(final PurchaseHistory history) {
        return history.returnRate <= 0.001;
    }
}
```

返品率のルール

ポリシーパターン

先ほどクラス化したルールを使って優良顧客の方針を表現するクラスを実装する。

◆優良顧客の方針を表現するクラス

```
class ExcellentcustomerPolicy {  
    private final Set<ExcellentCustomerRule> rules;  
  
    ExcellentCustomerPolicy() {  
        rules = new HashSet();  
    }  
    void add(final ExcellentCustomerRule rule) {  
        rules.add(rule);  
    }  
    boolean complyWithAll(final PurchaseHistory history) {  
        for(ExcellentCustomerRule each : rules) {  
            if(!each.ok(history)) return false;  
        }  
        return true;  
    }  
}
```

ルールを追加するメソッド

追加されたルールをすべて満たすか
検査するメソッド

ポリシーパターン

先ほどクラス化したルールを使ってゴールド会員、シルバー会員を表現するクラスを実装する。

◆ゴールド会員をクラスで表現する

```
class GoldCustomerPolicy {  
    private final ExcellentCustomerPolicy policy;  
  
    GoldCustomerPolicy() {  
        policy = new ExcellentCustomerPolicy();  
        policy.add(new GoldCustomerPurchaseAmountRule());  
        policy.add(new PurchaseFrequencyRule());  
        policy.add(new ReturnRateRule());  
    }  
  
    Boolean complyWithAll(final PurchaseHistory history) {  
        return policy.complyWithAll(history);  
    }  
}
```

ここでゴールド会員を定義

ゴールド会員かを検査

ポリシーパターン

先ほどクラス化したルールを使ってゴールド会員、シルバー会員を表現するクラスを実装する。

◆シルバー会員をクラスで表現する

```
class SilverCustomerPolicy {  
    private final ExcellentCustomerPolicy policy;  
  
    SilverCustomerPolicy() {  
        policy = new ExcellentCustomerPolicy();  
        policy.add(new PurchaseFrequencyRule());  
        policy.add(new ReturnRateRule());  
    }  
  
    Boolean complyWithAll(final PurchaseHistory history) {  
        return policy.complyWithAll(history);  
    }  
}
```

ここでシルバー会員を定義

シルバー会員かを検査

ここまでで解消された問題

- ◆ゴールド会員やシルバー会員の条件が集約されたので、それぞれの条件を変更する際はそれぞれのクラスを参照すれば良くなった。
- ◆ルールを再利用することができるようになり条件分岐が減った。

型チェックで分岐しない

条件分岐のネストは解消されたが...

せっかくインターフェースを使っても条件分岐が減らない良くないやり方が存在する。

◆例えば宿泊料金をインターフェースを用いて実装する例を考える。

```
interface HotelRates {  
    Money fee(); // 金額  
}
```

宿泊料金インターフェース

```
class RegularRates implements HotelRates {  
    public Money fee() {  
        return new Money(7000);  
    }  
}
```

通常宿泊料金クラス

```
class PremiumRates implements HotelRates {  
    public Money fee() {  
        return new Money(12000);  
    }  
}
```

プレミアム宿泊料金クラス

条件分岐のネストは解消されたが...

せっかくインターフェースを使っても条件分岐が減らない良くないやり方が存在する。

◆このとき、繁忙期などの割り増し料金を実装しようとする場合...

```
Money busySeasonFee;
```

通常宿泊料金なら3000円追加

```
if(hotelRates instanceof RegularRates) {  
    busySeasonFee = hotelRates.fee().add(new Money(3000));  
}  
else if (hotelRates instanceof PremiumRates) {  
    busySeasonFee = hotelRates.fee().add(new Money(5000));  
}
```

プレミアム宿泊料金なら5000円追加

せっかくインターフェースとクラスを用意して条件分岐を減らしたのに
外部に型判定による条件分岐を作っているのは意味が無い。

リスクの置換原則

リスコフの置換原則

クラスの基本型と継承型の間に成り立つ規律をリスコフの置換原則と呼ぶ。

◆リスコフの置換原則とは、「基本型（親クラス）を継承型（子クラス）に置き換えても問題無く動作しなければならない」とするものである。

先ほどの例では...

型判定をしなければRegularRatesクラスとPremiumRatesクラスを使い分けることができない

```
Money busySeasonFee;  
  
if(hotelRates instanceof RegularRates) {  
    busySeasonFee = hotelRates.fee().add(new Money(3000));  
}  
else if (hotelRates instanceof PremiumRates) {  
    busySeasonFee = hotelRates.fee().add(new Money(5000));  
}
```

つまり単純にhotelRates型として扱うことができないため、リスコフの置換原則に違反している

リスクの置換原則

繁忙期の料金もインターフェースで切り替えよう。

◆リスクの置換原則に則った設計をする。

```
interface HotelRates {  
    Money fee(); // 金額  
    Money busySeasonFee(); // 割り増し料金  
}
```

宿泊料金インターフェース

```
class RegularRates implements HotelRates {  
    public Money fee() {  
        return new Money(7000);  
    }  
  
    public Money busySeasonFee() {  
        return fee().add(new Money(3000));  
    }  
}
```

※プレミアム宿泊料金についても同様に実装する

通常宿泊料金クラス

リスコフの置換原則

◆先ほどの実装により、呼び出し先での型判定がなくなかった。

```
Money busySeasonFee = hotelRates.busySeasonFee();
```

実際の型が何かは意識しなくても良くなった。

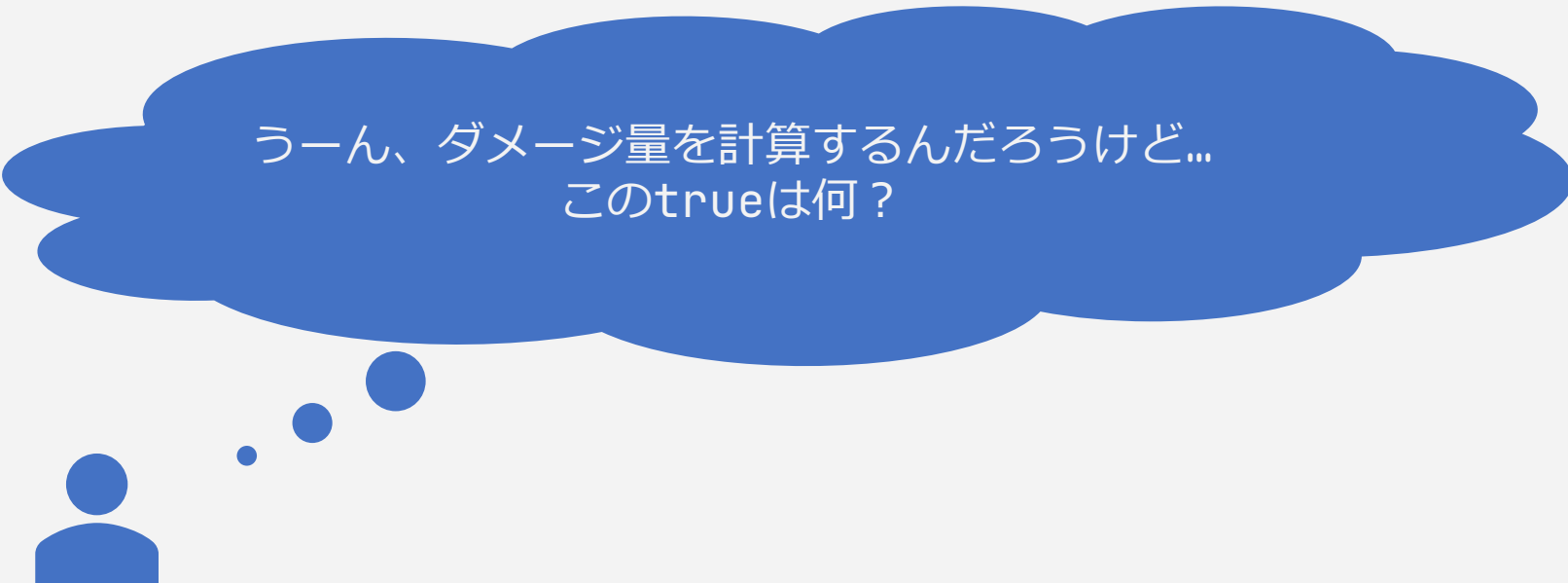
インターフェースを使いこなせるかが設計では重要！
分岐を書きそうになったらインターフェース設計を考えてみよう！

フラグ引数

フラグ引数

◆次のメソッドについて考えてみましょう。

```
damage(true, damageAmount);
```



うーん、ダメージ量を計算するんだろうけど...
このtrueは何？

フラグ引数

フラグ引数付きメソッドは、内部に複数の機能を持ち、フラグで切り替えている構造である。

◆メソッドの中身を見てみると...

```
void damage(Boolean damageFlag, int damageAmount) {  
    if(damageFlag == true) {  
        member.hitPoint -= damageAmount;  
        if(0 < member.hitPoint) return;  
        member.hitPoint = 0;  
        member.addState(StateType.dead);  
    } else {  
        member.magicPoint -= damageAmount;  
        if(0 < member.magicPoint) return;  
        member.magicPoint = 0;  
    }  
}
```

えー！第一引数でHPダメージかMPダメージかを切り替えてるじゃん！



フラグ引数

フラグ引数付きメソッドは、内部に複数の機能を持ち、フラグで切り替えている構造である。

◆int型変数で切り替える場合も同様

```
void execute(int processNumber) {  
    if(processNumber == 0) {  
        // アカウント登録処理  
    } else if (processNumber == 1) {  
        // 配送完了メール送信処理  
    } else if (processNumber == 2) {  
        // 注文処理  
    } else if (processNumber == 3) {  
        ... 以下分岐が続く  
    }
```



なんかたくさんの機能があるなあ...

フラグ引数

フラグ引数付きメソッドは機能毎に分離する。

◆メソッドを分離する

```
void hitPointDamage(final int damageAmount) {  
    member.hitPoint -= damageAmount;  
    if(0 < member.hitPoint) return;  
    member.hitPoint = 0;  
    member.addState(StateType.dead);  
}
```

```
void magicPointDamage(final int damageAmount) {  
    member.magicPoint -= damageAmount;  
    if(0 < member.magicPoint) return;  
    member.magicPoint = 0;  
}
```

機能を分けてふさわしい命名をすることで可読性UP！

ストラテジパターンを使ってみる

- ◆何らかの仕様により、さらに機能を切り替える場面が増えたときに、追加でbooleanによる切り替えを行っているという意味が無い。
- ◆ここで活躍するのがストラテジパターン

ストラテジパターンを使ってみる

◆ダメージを表現するインターフェースを実装

```
interface Damage {  
    void execute(final int damageAmount);  
}
```

ダメージインターフェース

```
class HitPointDamage implements Damage {  
    // 中略  
    public void execute(final int damageAmount) {  
        member.hitPoint -= damageAmount;  
        if(0 < member.hitPoint) return;  
        member.hitPoint = 0;  
        member.addState(StateType.deat);  
    }  
}
```

※魔法カダメージクラスも
同様に実装する

ヒットポイントダメージクラス

ストラテジパターンを使ってみる

◆Mapで処理を切り替える

```
enum DamageType {  
    hitPoint,  
    magicPoint  
}  
  
private final Map<DamageType, Damage> damages;  
  
void applyDamage(final DamageType damageType,  
                 final int damageAmount) {  
    final Damage = damages.get(damageType);  
    damage.execute(damageAmount);  
}
```

```
applyDamage(DamageType.magicPoint, damageAmount);
```

次の章に向けて

- ◆条件分岐はインターフェースで実装してみよう
- ◆ストラテジパターンやポリシーパターンを活用する
- ◆次の章ではコレクションについて学ぶ