コードの変更作業が楽で安全なものにならないのは きっと設計のせい

設計のまずいコードの特徴

世の中には2種類のコードがあります。設計のまずいコードとうまいコードです。では，設計のまずいコードとはどのようなコードでしょうか。特徴がいくつか挙げられます。

- **メソッドが長い**
- **クラスが大きい**
- **引数が多い**
- **記述の重複が多い**
- **関心事を詰め込みすぎている**

最初はだれもがうまく設計しようとします。しかし，小さな変更や修正，機能の追加など，一つひとつの小さな積み重ねが積み上がった結果，気がつけばどこになにが書いてあるかわからないような，手を入れるとどこでどのような副作用が起こるかわからないような，変更が大変なコードに変質していってしまうのです。

うまい設計は変更に強い

以下に，コードの設計を改良する例を示します。

```
int price = quantity * unitPrice;
if( price < 3000 )
    price += 500; //送料
price = price * taxRate();
```

このコードには，priceという1つの変数を3つの用途（数量×単価，それに送料を考慮した場合の値段，それをさらに税込みにした値段）で使いまわしているという問題があります。これは破壊的代入といって副作用を起こしやすい書き方です。これを防ぐために用途別にローカル変数を用意します。

```
int basePrice = quantity * unitPrice;

int shippingCost = 0;
if( basePrice < 3000 )  // 3000円未満なら
    shippingCost = 500; // 送料500円

int itemPrice = (basePrice + shippingCost) *
taxRate();
```

これでだいぶわかりやすくなりましたが，shippingCostに関してはif文や将来変更されうる固有のデータ（3000や500）を内包しています。これらはメソッドとして独立させます。

```
int basePrice = quantity * unitPrice;
int shippingCost = shippingCost(basePrice);
                                    //送料計算メソッド
int itemPrice = (basePrice + shippingCost)
* taxRate();
...
//メソッドに独立させた送料計算のロジック
int shippingCost(int basePrice) {
    if( basePrice < 3000 ) return 500;
    return 0 ;
}
```

送料計算に関するコードをshippingCost()メソッドに切り出しました。将来，送料の計算方法に変更が加わった場合は，このshippingCost()メソッド内だけ注目すればよい設計に生まれ変わりました。

このように，うまく設計されたコードは変更を加える場所が明確で影響範囲も限定されるため，変更が楽で安全なものになります。

書籍『**現場で役立つシステム設計の原則**』では，そのようなうまい設計のやり方を，コードからアプリ連携，システム全体の設計まで，わかりやすく解説しています。

ちょっとした変更のはずが大ごとになったり，あっちにもこっちにも飛び火しがちという現場の方に，一読をぜひおすすめします。

現場で役立つシステム設計の原則
変更を楽で安全にするオブジェクト指向の実践技法

増田 亨◉著
A5判・320頁　定価（本体価格2940円＋税）
ISBN 978-4-7741-9087-7