# Object.create(null)

0703
2022

## 共通化すれば良いとは限らない

プログラミング　日記

ここのところ偶然なのか「共通化」という言葉を多く聞いているのですが, その言葉を聞くたびに身構えていることに気がついたので, この気持ちの出どころを共有しておきます.

なぜ身構えているかというと, 共通化が必ずしもコードを良い状態にするとは限らないにも関わらず, それ自体が目的になってしまっている (ように見える) ことが多いからです. この手のリファクタリングの目的はあくまでコードの改善のはずで, そのことを忘れて共通化するだけで満足してしまうと, 良くてリファクタリングの効果が半減, 悪ければ逆効果になってしまいます.

個人的にコードを共通化する上で注意してほしいと思っているのは以下の二つです.

- コードを共通化すべきでない場合もある
- 共通化されたコードは一般的な原則にしたがって設計されなければならない

似たようなことは歴史の中で何度も繰り返し言われていることだろうと思いますが, 改めて.

### コードを共通化すべきでない場合もある

共通化に関する原則として有名なものに DRY (Don't Repeat Yourself) がありますが, これはどのような場合にもコードを共通化すべきだというものではありません.

DRY 原則が述べていることは, 同じ知識を繰り返し書いてはいけないということです. ここでいう知識とは, プログラミングの文脈では状況・問題・解決手段の組 (ある状況の下である問題を解決するためにはどういった手段をとればよいか) と考えておけば良いと思います. これらが複数の場所に書かれていると, どれが (あるいはどれも) 正しいのかわからないですし, 変更は必ず同時に行う必要があるので大変です.

逆に異なる知識 (状況・問題・解決手段のいずれかが異なる) であれば DRY 原則の対象からは外れるため, 繰り返し書かないようにしてやる義理はありません. むしろ異なる知識同士を無理やり一つにまとめてしまうと, 状況や問題など知識の内容が曖昧になったり, 変更を加えようとすると互いに干渉してしまったりと, 扱いが非常に難しくなってしまうため避けるべきです.

ここで落とし穴になるのが, コードで表現されるのはふつう知識の全てではなく, 最も多いのは解決手段のみであるということです. この事実を無視して コード = 知識 のように捉えてしまうと, DRY 原則を適用したつもりで誤った共通化をしてしまう可能性があります.

とはいえ状況・問題・解決手段は互いに無関係ではないので, 多くのコードの重複は知識の重複であるのも事実です. すべてがそうでないからといって共通化を避けすぎてもいけません.

### 例. 問題や解決手段は同じでも状況が異なる場合

React による UI コンポーネントを例にします.

一つ目のコンポーネントは, 画面上部にありがちなユーザー設定へのリンクを表示するためのものです.

*TypeScript*

```
function UserSettingsLink({ user }) {
  return (
    <a href="/settings" class="user-settings-link">
      <img src={user.iconUrl} />
```

```
      <span>{user.screenName}</span>
    </a>
  );
}
```

もう一つは, ユーザーのリストの項目を表示するためのコンポーネントです.

TypeScript

```
function UserListItem({ user }) {
  return (
    <li class="user-list-item">
      <img src={user.iconUrl} />
      <span>{user.screenName}</span>
    </li>
  );
}
```

これらのコンポーネントのうち, ユーザーのアイコンと名前を表示する部分のコードは全く同じです. この部分は共通化すべきでしょうか?

私はこれらの場合は状況が異なるため共通化すべきではないと考えます. 状況が異なれば問題も変わる可能性があって, 例えばそれぞれ表示したい項目やスタイリングが変わるかもしれませんし, それに応じて解決手段であるマークアップも変わることがあるかもしれません. 大まかな目的は近いものの, 詳細まで同じになっているのは単なる偶然の一致で, なんらかの同一の知識を表しているわけではないと考えるのが良さそうです.

あるいはコードをそのまま共通化するのではなく, より一般的な状況や問題を想定した形にコードを抽象化するという方法も採れるかもしれません. が, この規模であれば得られるメリットよりも良い抽象を考えたり理解したりするコストの方が高くついてしまいそうですね.

## 共通化されたコードは一般的な原則にしたがって設計されなければならない

上に挙げた DRY 原則を含むプログラミングの一般的な原則に従うことは, コードの可読性・保守性・柔軟性・再利用性などを確保する上で非常に重要です.

ところが特に既存のコードをリファクタリングする過程で共通化が行われた場合に起こりがちなことですが, 共通化されたコードは重複した部分を文字通りに共通化しただけのようになっており, 一般的な原則に従えていないことがしばしばあります. これでは折角のリファクタリングの価値が半減してしまいます.

例えば以下のようなコードのリファクタリングを考えてみましょう.

- 関数 A:
  - A 専用の処理
  - 重複した処理
- 関数 B:
  - B 専用の処理
  - 重複した処理

これらの関数の処理の重複をなくしたいという目的で, 以下のような共通化が行われることがあります.

- 関数 AB:
  - A のときは A 専用の処理
  - B のときは B 専用の処理
  - 共通の処理

直ちにこの形の共通化が良くないとは言えません. 例えば以下のような条件を満たしているのであれば, 妥当な共通化であると言えるでしょう.

- 処理のパターンは今後も A, B しか存在しないことが期待できる
- A, B それぞれの専用の処理は共通の処理と比べてごく小さい

一方でこれらの条件が満たされない場合には, 一般的な原則に基づいて問題点や改善案を考えることができます.

### 例 1. 開放・閉鎖原則に反する場合

まずは上に挙げた条件のうち, 特に前者の条件が満たされない場合を考えてみましょう.

- 処理のパターンは A, B 以外にも存在するかもしれない

例えば A, B とは別のパターン C に対応しなければいけなくなったとすると, 関数 AB は以下のように変更される必要があります.

- 関数 ABC:
  - A のときは A 専用の処理
  - B のときは B 専用の処理
  - C のときは C 専用の処理
  - 共通の処理

このように新たなパターンに対応できるように拡張をするときにそのコード自体を変更しなければならないというのは, 解放・閉鎖原則に反しています. 拡張のたびに共通の処理に手を入れなければいけないというのはコードの柔軟性や再利用性が低いと言えますし, 共通の処理にパターンごとの条件分岐が増えることで可読性や保守性にも影響を与えてしまいます.

拡張に対して開いた状態にリファクタリングするためには, 次のように共通の処理だけを関数に切り出し, 各パターンはそれを利用するような形にすると良いでしょう.

- 関数 X:
  - 共通の処理
- 関数 A:
  - A 専用の処理
  - 関数 X を呼び出す
- 関数 B:
  - B 専用の処理
  - 関数 X を呼び出す
- 関数 C:
  - C 専用の処理
  - 関数 X を呼び出す

## 例 2. 単一責任の原則に反する場合

続いて後者の条件が満たされない場合を考えてみます.

- A, B それぞれの専用の処理は共通の処理と比べて小さくない

もし A, B 専用の処理の規模が小さければ, 関数 AB の責任は共通の処理を実行することで, そこに小さなバリエーションが存在するだけであると考えられます. しかし専用の処理の規模が大きくなってくると, もはやバリエーションとしては捉えられず, 関数 AB が A の処理の実行と B の処理の実行という二つの責任を持っているように見えてきます.

このような状態は単一責任の原則に反していると言えます. 関数が複数の責任を持っている状態では, その関数が表している知識が理解しづらくなったり, ある一面に対する変更が他の面にも影響を与えてしまったりと, 可読性や保守性が低下してしまいます.

こちらの場合も, やはり共通の処理だけを関数に切り出すことで, それぞれの関数の責務を一つに限定するのが良いでしょう.

- 関数 X:
  - 共通の処理
- 関数 A:
  - A 専用の処理
  - 関数 X を呼び出す
- 関数 B:
  - B 専用の処理
  - 関数 X を呼び出す

## まとめ

ということで, 共通化は単にしておけば良いというものではありません.

- コードを共通化すべきでない場合もある
  - 知識 (状況・問題・解決方法) が重複している場合のみ共通化すべき
- 共通化されたコードは一般的な原則にしたがって設計されなければならない
  - 共通化されたコードについても可読性・保守性・柔軟性・再利用性などを確保すべき

コードを共通化するときにはこういったことを考えつつ, なぜ共通化するのか, なぜその方法で共通化するのかについて説明できるようにしましょう.

*by.Susisu (id:susisu)* 22日前