@wolfmagnate (進捗ゼミ)

どうしてあなたの共通化は間違っているのか：第1章「単一責任原則の有用性」

リファクタリング 設計 デザインパターン アーキテクチャ SOLID原則

最終更新日 2024年03月12日 投稿日 2024年03月10日

どうしてあなたの共通化は間違っているのかの目次はこちら

はじめに

この記事では、ソフトウェアにおける共通化についての指針を提供する「単一責任原則」について、現在語られている一般的な内容がどの程度役立つかを考察します。一般的な内容の復習が多く含まれているため、気楽に読んでください。
この記事では、単一責任原則についての一般的な説明として @MinoDiven さんの単一責任原則で無責任な多目的クラスを爆殺する を多く参照するため、単に元記事と呼びます。

いいね・ストックが励みになります！

一般的な単一責任原則の説明

さて、一般的に単一責任原則は、次のように説明されます。

> クラスが不正な動作をしないように責任をもって管理するべき関心事は単一にするべきである

例えば、元記事の例では、次のようなコードが「複数種類の割引の管理という責任をひとつのメソッドに任せているため、単一責任ではない」というように解説しています。

```
class DiscountManager {
  static int getDiscountPrice(int price, boolean isSummer) {
    if (isSummer) {
      int discountPrice = price - 300;
      if (discountPrice < 0) {
        discountPrice = 0;
      }
      return discountPrice;
    }

    return (int)(price * (1.00 - 0.04));
  }
}
```

この解説は正しく、割引する処理から2つのメソッドを切り出すことにより、メソッドの内容をよりシンプルにすることが出来ます。

- 適用する割引を判定する
- 通常割引処理
- 夏季割引処理

※元の記事では複数のクラスに分割していましたが、ここで説明したいのは分割すると良いという点であるため、簡潔さを重視してメソッドにしています。

```
class DiscountManager {
  static int getDiscountPrice(int price, boolean isSummer) {
    if (isSummer) {
        return getSummerDiscount(price)
    }

    return getRegularDiscount(price);
  }
```

```
        if (discountPrice < 0) {
          discountPrice = 0;
        }
        return discountPrice;
    }

    static int getRegularDiscount(int price) {
        return (int)(price * (1.00 - 0.04));
    }
}
```

限界1.「単一」なんてどうとでも言えてしまう

しかし、ここで疑問があります。元の関数も、責務を「割引後の値を取得すること」だと考えれば、元の関数も単一責任原則を満たすと言えるのではないでしょうか。
つまり、**責務が単一であることは単に使われている状況から見た後知恵であって、どのような設計に対しても責務の内容を強弁すれば単一責任原則を満たすと言えてしまうので、設計指針にはなりえない**のではないか、という問題です。

実際、**ビジネスに関心がないと単一責任設計が困難である**ことは元記事でも強調されています。しかし、技術的な観点を一切持たずに「ビジネスを知って、増えなさそうな処理をビジネス上範囲が明確そうな名前を使ってクラスにしよう」という観点で設計する場合、そもそも単一責任原則は不要で、ドメイン知識だけ知っていれば自然と設計できるのではないでしょうか？

単一について具体的な判定法やどのような単一でない場合があるのかについて一般的な判定方法を一切与えずに、単にドメイン知識をもっていい感じの範囲にしろ、という説明に終始するのであれば、それは単に「いい感じによろしく〜」と言ってるのと変わらないのではないでしょうか？

限界2. 単一責任原則とDRYの乱用

元記事では、単一責任原則を無視してDRYの名のもとに設計上問題を引き起こす共通化を行ってしまう危険性を解説しています。
しかし、逆もまたしかりであり、単一責任原則の名のもとにDRYが破壊される危険性もあります。

他にも単一責任原則と共通化については、単一責任原則をあまり理解せずに乱用するとコピペが増えてしまうという問題があります。例えば、元記事では通常割引と夏季割引において、メソッドの内容がたまたま300円引きで一致していたとしても分離するべきだ、という主張がなされています。しかし、どちらも「メモリ上のキャッシュから商品の値段ごとの割引率を計算、不足していた場合はDBアクセスして特定のカラムの値を利用」のような複雑な処理を含んでいるとき、単一責任原則を守るためにはそれを2回書くしかないのでしょうか？

どのような場合にDRYを重視し、どのような場合に単一責任原則を行うべきかを理論的かつ明確に説明できずに「概念単位」程度のふんわりした言葉でしか説明できないのであれば、それはもはや感覚でいい感じに分けたり共通化したりしようね、程度の意味しか持ちません。

より解像度の高い理解に必要なもの

ドメイン駆動設計

元記事では、上記のような単一責任原則だけではどうしようもない現実に対応するために、ドメイン駆動設計を推奨しています。ドメイン知識を適切に理解し共有し、ドメインエキスパートから学び、それをコードにするまでの方法論を学ぶことが出来ます。そのため、単一責任原則に基づいた設計において重要な設計手法です。

設計原則へのより深い理解

より技術的・一般的な側面から、単一責任原則をどのように解釈するかについての設計上の諸問題を解消することを目指します。責務や関心事というどうとでもいえる曖昧な言葉に逃げず、なおかつ**ドメインについての具体的な知識を使わない汎用的な手法**を扱います。
このアプローチをとる場合、責任がどのように特徴づけられるのか、責任が単一でないことをどのように判定すればよいのか、どのような責任の扱いがどのような問題を引き起こすのか、種々の誤りをどのように解消すればいいのかを正確に理解しやすい形で理論的にまとめることが必要です。

これらの手法は、どちらか片方しか使えないというものではなく、相互に補完しあうものです。どのように補完しあうのかについて解説します。

設計理論の役割

汎用的で技術のことだけを考えている理論は、アルゴリズムのようなものです。ドメインの性質や複雑性が入力されたときに、役立つ設計を出力として提供します。
本連載ではこの理論について解説するため、もう少し詳しく設計理論の役割を整理してみましょう。
設計の理論の理解度には、おおよそ3つのレベルが存在します。これに応じてエンジニアの設計能力にもざっくり3つのレベルが存在します。

> **設計理論レベル0**
> 処理の内容、データの流れだけを気にしており、設計上何をしていようが、最終結果さえあっていれば気にしない

- このレベルのエンジニアにコーディングを任せると遠からず破滅する
- ChatGPTに全体の設計の流れを渡さずに「〇〇するコードを書いて」と頼んだときのレベル
- コードの仕様に比べると設計方針は文書化されにくいため、ChatGPTに設計方針のコンテキストを余さず伝えるためには、人間が設計上の内容を理解しておく必要がある。このため、レベル0のエンジニアはChatGPTを使ってもレベル0の設計がされたコードしか出力できない

> **設計理論レベル1**
> 自分が何を書いているのかを、コードの処理だけでなく設計の面から理解することが出来る

- どのようなコードがどのような効果を持つのかを理解する
- 特定の方法で書いた場合の辛い点、うれしい点を理解する
- 自分の書いているコードの設計パターンの名前を知り、どのような性質があるのかを理解する
- このレベルのエンジニアは、新しい処理を追加するときも既存の設計の意図を理解して、設計と矛盾しない形でコーディングすることが出来るため、全体の設計を破

設計理論レベル2
設計面での行動の選択のベストプラクティスを把握し、戦略的に何を書くか選ぶことが出来る

- どのような場合にどのような方法をなぜ適用するべきかを理解する
- しない場合のデメリットは何かを理解する
- あえてベストプラクティスを外れるようなエッジケースはあるかを知っている
- このレベルのエンジニアは、既存のアーキテクチャ全体の設計手法が現在の複雑性に対してどの程度対応しているかを把握できる
- このレベルのエンジニアは、現在のアーキテクチャにとらわれず、現在の複雑性に応じてアーキテクチャをより適した形に再構成できる

ドメイン理解の役割

実際の設計を行う場合は、設計自体についての方法論やパターンのうちどれをどのように適用すべきであるかを、設計対象のドメイン知識から得られる洞察によって決定します。そのため、どれだけ網羅的にどのような抽象的性質を持ったソフトウェアには何をするべきかを根拠を持って把握していたとしても、**現在対処している問題がどのような性質を持っているかを正確に把握することが出来なければ、何の意味もありません。**

おわりに

この記事では、設計理論を理解することへのモチベーションについて説明しました。次回以降の記事を読むことで、より根拠を持った設計の説明が出来るようになりましょう。

@wolfmagnate (進捗ゼミ)
C#が好きな情報系大学生 進捗を出したい

今日のトレンド記事

@DEmodoriGatsuO (Gatsuo De'modori)
2024年05月18日

Power Apps & GPT-4oを使って超高速で画像解析アプリを作る！

PowerApps AzureKeyVault PowerAutomate ChatGPT GPT-4o

♡ 34

@naoya_347 (なおや)
2024年05月18日

useSession？なにそれおいしいの？

React Next.js useSession

@kaku3
2024年05月18日

AIは仕事ではなく仕事力を奪う？

ポエム　教育　AI　pm　新人プログラマ応援

♡ 15

@mkt_hanada (Makoto Hanada)
2024年05月18日

【AI品質・AIテストまとめ】AIシステムの品質を高めるために

テスト　AI　データサイエンス　品質

♡ 17

@sanjushi003
2024年05月17日

VMware Fusion 環境 (macOS) に Windows Server 2022 仮想マシンを作成してみた

Mac　Broadcom　vmware　VMwareFusion　WindowsServer

♡ 7

トレンド一覧を見る

関連記事 Recommended by

どうしてあなたの共通化は間違っているのか：第5章「新・単一責任原則」
by wolfmagnate

単一責任原則で無責任な多目的クラスを爆殺する
by MinoDriven

どうしてあなたの共通化は間違っているのか：目次
by wolfmagnate

単一責任の原則（SRP）
by gomi_ningen

134　151

この記事は以下の記事からリンクされています

どうしてあなたの共通化は間違っているのか：第5章「新・単一責任原則」 からリンク　2 months ago

どうしてあなたの共通化は間違っているのか：第2章「抽象度と文脈」 からリンク　2 months ago

どうしてあなたの共通化は間違っているのか：目次 からリンク　2 months ago

コメント

この記事にコメントはありません。

T コメントする

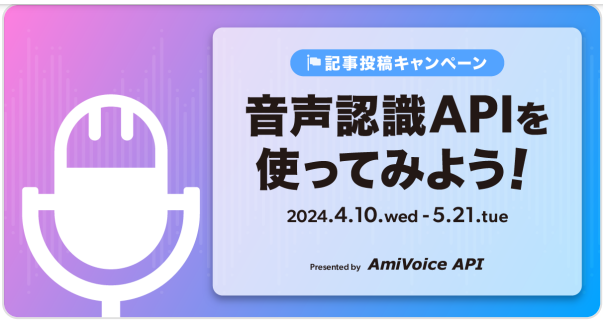
プレビュー

コミュニティガイドラインに基づき、良識ある内容を心がけましょう。

テキストを入力

0B / 100MB　投稿する

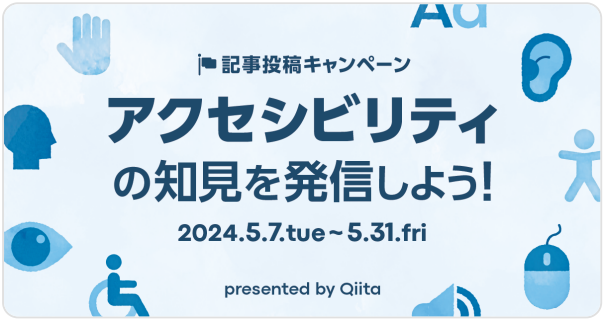
134　151



音声認識APIを使ってみよう！

2024/04/10~2024/05/21

詳細を見る



アクセシビリティの知見を発信しよう！

2024/05/07~2024/05/31

詳細を見る

すべて見る

How developers code is here.

ガイドとヘルプ

About

コンテンツ

リリースノート

SNS

Qiita（キータ）公式

134

151