課題図書レポート-『Clean Code-アジャイルソフトウェア達人の技-』-

2023/1/30

本社システム課　町田

■本書を選定した理由

良いコードを書くために必要な観点や手法を理解するため。

■学んだこと(まとめ)

●意味のある名前

　ソフトウェア開発において、変数、関数、クラス、パッケージなど名前付けをする場面が多数ある。後にコードを読む人がそのコードについてなぜ存在するのか、何をするのか、どのように使用するのかを把握するためには、良い名前付けが重要である。本書で紹介されていた良い名前付けのルールの一部を以下に列挙する。

* 意図が明確な名前にする
* 発音可能な名前を使用する
  + 例えば、grepしてヒットするような命名にする。
* 検索可能な名前を用いる
* 1つのコンセプトには1つの単語を使用する
  + 例えば、fetch, retrieve, getといった名前のついたメソッドを同じ意味で提供すると混乱を招きやすいため、可能な限り1つの名前に統一すべきである。

●関数

　関数をうまく書くルールを以下に列挙する。

* 小さくする
  + 本書では、20行に達することなど、ほとんどないようにすべきであると紹介されていた。
* 1つのことを行う
  + 1つの抽象レベルにおけるいくつかのステップのみで表現される場合、その関数は1つのことをしているといえる。
* 引数について
  + 引数の数は、理想は0である。4つ以上は特別な理由がない限り避けるべきである。
  + フラグ引数は避ける。フラグ引数を使用した時点でその関数は2つ以上のことを行っていることになるため。(Trueの場合の処理とFalseの場合の処理)
* コマンド・照会の分離原則
  + 関数は、オブジェクトの状態を変更するか、オブジェクトの何らかの情報を返すかのいずれかを行うようにする。両方を同時に行わない。
* DRY原則
  + 重複は排除すべきである。同じ処理を4箇所に実装すると、その処理に変更が生じた際に4箇所すべてを修正する必要があり、修正漏れの確立も4倍になるためである。

●コメント

* 良いコメント
  + コードの意図を説明するコメントは良いコメントである。コードでは表現できない情報を提供するためである。
  + 警告をコメントに残すことも有用である。例えば、ある関数(もしくはクラス)を特定の条件下でのみ使用するよう指示する場合などが該当する。
* 良くないコメント
  + コードの処理内容をそのまま説明するコード。
  + コメントアウトされたコード。(絶対ダメ)

●オブジェクトとデータ構造

* オブジェクト
  + 裏にあるデータを隠して抽象化し、データを操作する機能を提供する。
* データ構造
  + データを公開し、意味を持った機能は何も提供しない。
* デメテルの法則
  + オブジェクトを使用する場合、そのオブジェクトの内部について知るべきでない、という原則。
  + 例えば、“変数.メソッド().メソッド().メソッド()”は、それぞれのメソッドの戻り値の情報を知ってしまっているため、この原則に違反する。(電車の衝突)

●エラー処理

* 例外を使用する
  + ビジネスロジックとエラー処理は扱う関心ごとが異なる。try-catch-finally文を用いることで、ビジネスロジックとエラー処理を分離しやすくなる。
* nullを返さない・渡さない
  + エラー処理においてnullを返すことは避けるべきである。nullを返すと、呼び出し元でnullチェックを行う必要が出てくるためである。1つでもnullチェックが漏れるとアプリケーションが動作しなくなるのは危険である。
  + null引数として関数に渡すことも避けるべきである。呼び出し元からnullが渡されてもうまく対処する手立てがあまりないためである。

●境界

　境界とは、例えば自分たちが実装するコードの領域とサードパーティのコードの領域などが該当する。サードーパーティのコードをビジネスロジックに分散的に実装すると、サードパーティコードの変更の影響を広範囲に受けることになる。サードーパーティ用のラッパークラスを用意したり、アダプタを用意するなどして境界を設けることで、サードーパーティの変更による影響を最小化すべきである。

●単体テスト

* TDD三原則
  + 失敗する探偵テストのコードを書く前に、製品のコードを書いてはならない。
  + コンパイルが通り、適切に失敗する単体テストができるまでは、次の単体テストを書いてはならない。
  + 現在失敗している単体テストが通るまで、次の製品コードを書いてはならない。
* F.I.R.S.T
  + 高速である(Fast)
    - テストは高速である必要がある。なぜなら、TDDではテストを頻繁に実行するためである。
  + 独立している(Independent)
    - テストは互いに関連すべきではない。なぜなら、あるテストの失敗が後続のテストの失敗につながり、テスト後半部の問題を隠してしまうためである。
  + 再現性がある(Repeatable)
    - テストはどんな環境でも再現可能である必要がある。なぜなら、動かあに環境があるということは、テストが失敗する言い訳をいつも抱えているということを意味するためである。
  + 自己検証可能(Self-Validating)
    - テストは成功か失敗かのどちらかを出力すべきである。なぜなら、テストが自己検証可能でないと、成功・失敗の判定が属人的になったり、長時間を要する手作業による検証が必要になる可能性があるためである。
  + 適時性がある(Timely)
    - テストは必要な時にすぐ書ける必要がある。なぜなら、製品コードの実装の後にテストを書こうとすると、テストが困難な設計になってしまっている可能性があるためである。

●クラス

　クラスは小さくすべきである。大きいクラスは、責務が多かったり、依存関係が多いなど、変更に弱くなる理由を持っている可能性がある。

* 単一責務の原則(SRP)
  + クラスやモジュールは変更の原因となるものが1つでなければならない。
* 凝集性
  + あるメソッドが操作するインスタンス変数が多ければ多いほど、凝集性が高いといえる。

●システム

　システムを使うことと、構築することを分離すべきである。

* mainの分離
  + 構築に関するすべての局面を、mainあるいはmainと名付けられたモジュール(いずれも開始処理)へ移動し、システム内の残りの部分は、すべてのオブジェクトが適切に生成され、関連付けられているという前提のもとに設計する手法。
* ファクトリ
  + 特定のオブジェクトを生成するタイミングをアプリケーションが制御することを可能とさせつつ、生成に関する詳細をアプリケーションコードから分離する手法である。
* 依存性注入
  + オブジェクトの利用と生成を分離する手法である。
  + 多くの場合は、セッタメソッドあるいはコンストラクタの引数を通して依存性を解決する。

●優れた設計を生み出す4つの原則

* 全テストを実行する
  + 設計作業を通して、意図通りに動作するシステムを生成する必要がある。そのためには、システムが意図通りに動作しているかを検証できるテストが必要である。
* 重複がない
  + 重複は、余計な作業、余分な危険、不必要な複雑さをもたらす。同じあるいは似ている処理を関数にくくりだす、多態性を利用するなど、重複の排除に尽力すべきである。
* プログラマの意図が表現されている
  + 変更を行う際にバグを埋め込む可能性を最小化するには、システムの動作を理解可能とすることが重要である。そのため、コードには書き手の意図が明快に表現されている必要がある。
* クラスとメソッドを最小化する
  + 必要最低限のクラスとメソッドを用意するよう尽力すべきである。ただし、この原則は他の原則よりも優先度は低い。

■重要だと思ったこと(業務で活かすこと)

　ソフトウェア開発において、関心事の分離と重複の排除が重要なのだと思いました。

　関心事の分離に関して、サードパーティコードとの境界を明確にすることや単一責務の原則など、関心事を局所化することにより、修正箇所を最小化する営みが変更に強いシステムの構築につながるのだと思いました。

　重複の排除が重要であることは先日のハッカソンにおいても実感しました。ほとんど同じような処理が分散して実装されていると、その処理に変更があった際に修正漏れが発生しやすくなると思います。そうした際には、メソッドに処理をくくりだすことで修正箇所を局所化できると思います。また、オブジェクト指向の三大要素の一つである多態性を利用して重複を排除することも重要だと思います。

　今後の業務における実装では、特に以下のことを意識します。

* 変数や関数の命名
  + 他人が読んで意図が伝わる名前や処理の流れになっているか
* 重複
  + 実装した内容が既存の実装と重複する部分がないか