オブジェクト指向プログラミング
第6回レポート課題

24336488 本間三暉

2024年6月10日

1　課題 1：ソースコードのリファクタリング

課題 2 までのリファクタリングを行ったソースコードをソースコード 1 に示す.

ソースコード 1: main

```
1  public class Main {
2    public static void main(String[] args) {
3        // 入力
4        double x = 2;
5        // 出力用変数の定義///
6        double fx=0;
7        // 重み
8        double w10 = -4.0, w11 = 3.0;
9        double w20 = 1.0, w21 = 1.5;
10       // バイアス
11       double beta0 = -1.0, beta1 = 1.5, beta2 = -2.0;
12       // 基底関数の計算
13       double u1 = calculateU(x,w10,w11);
14       double u2 = calculateU(x,w20,w21);
15       System.out.printf("u1␣=␣%.1f,u2␣=␣%.1f\n", u1,u2);
16       // 活性化関数の適用
17       double g_u1 = 0, g_u2 = 0;
18       g_u1 = calculateActivation(u1);
19
20       g_u2 = calculateActivation(u2);
21
22       System.out.printf("g(u1)␣=␣%.1f,g(u2)␣=␣%.1f\n", g_u1,
             g_u2);
23       // 出力の計算
24       fx = calculateB(beta,0.0) + calculateB(beta1,g_u1) +
             calculateB(beta2,g_u2);
25       // 結果の表示
26       System.out.printf("f(x)␣=␣%.1f\n", fx);
27   }
28
29   public static double calculateActivation(double u){
30       if(u > 0){
31           return u;
32       }else{
33           return 0;
34       }
35   }
36
37   public static double calculateU(double x,double w0,double
         w1){ //
```

```
38          return w0 + w1 * x;
39      }
40
41      public static double calculateB(double beta,double g_u){
            // バイアスをかける
            関数
42          return beta*g_u;
43      }
44  }
```

課題 1 は示したとおりである.

2 課題 2：次のリファクタリングとして考えられる事を提案し、コード例を示しなさい。

課題 1 に続くリファクタリングとして，ソースコード 1 に示す calculateB 関数のような関数を実装をすると良いと考える.

3 課題 3：課題 2 以外で、考えられるリファクタリングを項目として 2 つ以上提示しなさい。

リファクタリング項目として以下のようなものが考えられる.

- コメントアウトの整理
- 変数の定義部を一箇所にまとめる
- 定数の利用
- 変数名の改善

4 課題 4：課題 3 のリファクタリングは行うべきか否か立場を明らかにした上で、理由を述べてください。

課題 3 で挙げたリファクタリング項目について以下のように考える.

4.1　コメントアウトの整理

明らかに行うべきである．コメントを書くことで未来の自分や他の人がコードを見たときに可読性が上がり，構造を掴むことに集中できる．しかし，書きすぎても逆効果で，自明なコードにコメントを追加するのではなく，ある程度難解な処理に対して簡潔でわかりやすいコメントを付ける必要がある．

4.2　変数の定義部を一箇所にまとめる

今回は 50 行にも満たないのでどこに変数定義を書いても読めるので無理に行う必要はない．必要になったときに変数を定義していると今回の 10 倍や 100 倍もあるようなコードでその変数を同じコード内で複数回使う場合に変数を探すところから始めることになってしまい，著しく可読性が下がり無駄な時間がかかってしまう．

4.3　定数の利用

今回のような，値が一度しか利用されない場合や変更される可能性が少ない場合は行うべきでない．しかし，重みやバイアスなど明らかな固定値があり，処理によって値を変更する事がある場合は定義の段階で定数として定義し意味のある名前をつけると可読性が上がると考える．

4.4　変数名の改善

今回はしなくても問題ない．長いコードの場合，意味のある変数名をつけることで，一目で変数がどのような意味で使われているかわかるようになり可読性が上がると考える．しかし，三単語以上の変数や長い英単語，あまりメジャーでない英単語を用いた変数名では逆に可読性が下がるため，ある程度の妥協も必要である．