イケてるコードを書くためのステップ

Acroquest Technology
谷本 心

今回は総集編として、今までの連載を振り返りながら、特に重要なポイントをピックアップしていきます。イケてるコードを書くために覚えておきたいことを順番に説明します。

そもそもイケてるJavaコードとは?

連載のタイトルでもあるJavaの「イケてるコード」とは、何を指すのでしょうか。「読みやすい」「修正しやすい」「無駄な処理時間が掛からない」「メモリーを無駄遣いしない」など様々な評価基準があります。ですが、筆者は“読みやすさ”が一番重要だと考えています。読みにくいコードは間違いに気づきにくいだけではなくバグも混入しやすく、修正する際にミスをしてしまう可能性が高くなります。読みやすいコードは、その逆だと言えるでしょう。

実際に例をみてみましょう。**リスト1**は、連載第1回で紹介したラインカウンタのソースコードから抜粋したものですが、かなり読みにくいコードです。このコードを例にして読みやすいコードの書き方を紹介しましょう。ポイントは3つあります。

1 良い変数名を付ける

読みやすいコードには、誰にでもわかりやすい変数名やメソッド名を付けることが重要です。さらに、一般的な名前を付けるのではなく、具体的な意味を付与すると、もっとわかりやすくなります。

リスト1のポイント1で記述している変数名は「flag」「flag2」です。一般的な変数名なので、この2つの変数はフラグを表しているのだな、ということはわかります。しかし、何を表すフラグなのかは、この変数名では全くわかりません。リスト1の例では、ある行にコードが存在するかどうかをflagで表し、コメントが存在するかどうかをflag2で表しています。読みやすいコードにするためにも、それぞれの変数名は、

```
boolean codeExisted = false;
boolean commentExisted = false;
```

のようにするべきでしょう。

2 ネストは極力少なくする

リスト1では、ifやfor、whileでなんと7階層にもネストしています。このようにネストを深くしてしまうと、インデントが多くなって読みにくくなりますし、カッコの対応付けもわかりにくくなるものです。

読みやすさを保つためには、ネストを浅くすることが重

リスト1●読みづらいJavaコードの例

```
public void count() {
  String file = "Sample.java";
  boolean nest = false;
  BufferedReader reader = null;
  String line = null;
  boolean comment = false;
  boolean flag = false;                    ← ポイント1
  boolean flag2 = false;
  int commentNest = 0;

  try {
    reader = new BufferedReader(new FileReader(file));
    while ((line = reader.readLine()) != null) {
      flag = false;
      flag2 = false;                       ← (1)
      for (int i = 0; i < line.length(); i++) {
        if (line.charAt(i) != ' ' && line.charAt(i) != '¥t') {
          if (comment) {
            flag2 = true;
            if (i + 1 < line.length() &&
                "*/".equals(line.substring(i, i + 2))) {
              if (nest) {                  ← ポイント2
                commentNest--;
                if (commentNest == 0) {
                  comment = false;
                }
  …
```

要です。例えば、筆者は1メソッドの中でのネストは3階層までというルールにしています。ネストを少なくするには、(A) ifやfor、while文のブロックの中身をそのまま別メソッドに切り出す、(B)条件判定を整理して不要なif文を削除する、という方法が有効です。この(B)の方法をリスト1の(1)の部分に適用してみましょう(**リスト2**)。条件判定を反転させ、条件が一致した場合に処理を抜けるように実装し、ネストを1段階浅くできました。

リスト2●ネストを浅くする一例

```
for(int i=0; i < line.length(); i++) {
  if (line.charAt(i) == '' || line.charAt(i) == '¥t') {
    continue;
  }
  if(comment) {
```

&&から変更

条件を整理してif文のネストを1つ浅くできた

3 適切にメソッド分割をする

コードを読みやすくするためには、1メソッドの行数をだいたい40行以内を目指すと良いでしょう。長くとも2画面ぶんの80行以内に抑えることが重要です。

長いメソッドを見つけた場合は、コードを別メソッドに分割して、1メソッドの行数を減らしましょう。メソッド分割のポイントは、ひとかたまりの処理を見つけることです。例えば、「ファイルを読み込む」「1行を解析する」などが、ひとかたまりの処理に当たります。ネストを少なくする部分でも少し触れましたが、ブロックの中身を別メソッドに分割するのもよいでしょう。

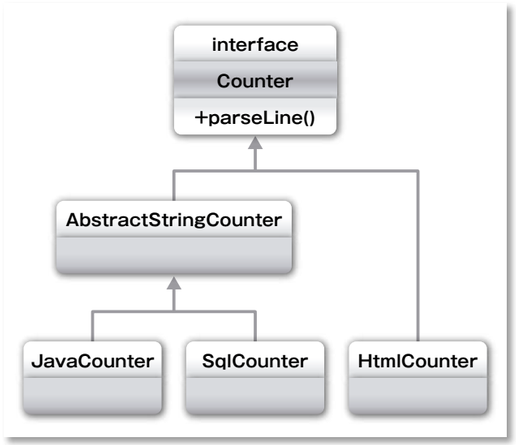
拡張しやすいことも重要

ここまで、「イケてるコード=読みやすいコード」とみなして説明してきました。しかし、現場で開発をしていると、ただ読みやすいだけではなく、機能を“拡張しやすい”こともイケてるコードの重要な要素になります。拡張しやすいコードとは、(1)拡張する仕組みがある、(2)拡張する際に修正個所が少ない、(3)コードの重複がない、の3つがポイントになります。

コードの行数やコメントをカウントするラインカウンタのJavaアプリケーションを作ったとしましょう。もちろん最初はJavaのソースファイルのライン数をカウントすることを目指して開発していきます。しかし、アプリケーションを完成させて使っているうちに、機能を追加したくなるかもしれません。例えば、JavaだけではなくHTMLやSQLのファイルもカウントできるような機能が必要になるかもしれないのです。このような機能追加の要望が発生したときに、簡単に対応できることが拡張しやすいソースコードです。

Javaに用意されている「インタフェース」や「抽象クラス」を活用して設計することが、拡張しやすいソースコードを書くためのコツになります。**図1**はラインカウンタのクラス構成の一例です。どのように考えてこの構成にしたのかを順に説明していきましょう。

図1●ラインカウンタのクラス図



1 インタフェースで操作を定義する

JavaとSQL、HTMLでは、それぞれコメントの書き方が違いますし、カウントのロジックもわずかに異なります。「カウントのロジックは同じではないの?」と思った人もいるかもしれません。しかし、Javaでは

```
String str = "これは // コメントではありません。";
```

のような書き方をしてもコメントとはみなされません。一方、HTMLには上記の文字列表現はないため、Javaと同じカウントロジックを使っても正しくカウントできません。JavaとHTMLのカウントロジックは異なっているといえます。このように「振る舞い(ロジック)は異なるけど、同じ操作(カウント)がしたい」という場合に利用できるのが、Javaのインタフェースです。

インタフェースとは、クラスの原型に相当し、クラスで利用する定数やメソッドの定義のみを並べたものです。今回の例で言えば「Counter」というインタフェースを定義し、それを実装した「JavaCounter」「SqlCounter」「HtmlCounter」の3つのクラスができることになります。インタフェースを定義しておくことで、他の拡張子が増えた際にもインタフェースに実装するクラスを追加することで対応できるのです。

2 抽象クラスで似た処理をまとめる

先ほどJavaとHTMLのカウントロジックが異なっていると説明しましたが、SQLのカウントロジックはどうでしょうか。SQLのカウントロジックは、Javaと同じタイプで文字列表現が存在します。

```
select * from TEST
where COMMENT = 'これは /**/ コメントではありません'
```

のような書き方ができます。つまり、Javaのカウントロジックとの SQLのロジックは、コメントの種類や文字列表現の種類は異なりますが、振る舞い自体はよく似ているのです。

このような場合には、抽象クラスが利用できます。今回であれば「JavaCounter」と「SqlCounter」の共通の親クラスとして「AbstractStringCounter(文字列表現つきカウンタ)」という抽象クラスを定義して、共通した振る舞いであるカウントのロジックを文字列表現付きカウンタのメソッドとして記述します。これで、JavaCounterとSqlCounterにそれぞれ同じロジックを書く必要がなくなります。

このように考えて、最終的には図1のようなクラス構成になりました。ただインタフェースと抽象クラスを適切に使うだけではなく、「どこが拡張される可能性があるのか?」を適切に見切ることも拡張性を高めるためには欠かせないスキルです。自分が作ったアプリケーションについて「将来的にどう使ってほしいのか」「ユーザーはどのような機能追加を希望するか」を考えることになります。開発しながらも常に想像力を働かせて、どこが拡張されるかを考慮することが、拡張性を高めるための秘訣になると筆者は考えています。

次のステップは性能を考慮したJavaコードを書く

良いコードが書けるようになったら、次のステップはプログラムの性能を考慮することが大切になります。簡単なプログラムであれば、多少性能が悪くとも問題にはなりません。しかし、より複雑なプログラムになると、扱うデータ量が多くなったり、プログラムの動作時間が長くなったり[*1]するために、応答性が悪くなってしまったり、最悪の場合はプログラムが異常終了してしまったりします。ここでやってしまいがちな性能低下を招くソースコードの例を3つ紹介しましょう。

1 文字列を「+」で結合している

リスト3のようにStringを「+」演算子」で結合すると、毎回Stringのオブジェクトを生成してしまい、生成したオブジェクトの数だけ無駄にメモリーを使ってしまいます。Javaでは、StringBuffer(J2SE 1.4以前)や、StringBuilder(J2SE 5.0以降)を利用して、文字列結合を行なうことで消費するメモリーを節約できます(**リスト4**)。

リスト3●文字列を「+」演算子で結合するプログラム

```
String message = "こんにちは ";
message += name;
message += "さん、今回は ";
message += count;
message += "回目のアクセスですね!";
return message;
```

なるべく+演算子は使わない

リスト4●文字列をStringBuilderで結合するプログラム

```
StringBuilder message =
  new StringBuilder("こんにちは ");
message.append(name);
message.append( "さん、今回は ");
message.append(count);
message.append( "回目のアクセスですね!");

return message.toString();
```

2 いつでも線形検索をしている

リスト5のようなコードをよく見かけます。この処理は順番にすべての要素をチェックしていく処理で、「線形検索」と呼ばれています。最もシンプルな検索ですが、要素数が増えたぶんだけ処理に時間がかかってしまいます。

さらに、リスト5はもう1つ問題があります。期待する要

*1 常駐サービスなどの終了しないプログラムの場合を想定しています。