

## 第3章

# コードの不吉な臭い

# 前回まで

- リファクタリングがどう役立つかの話
  - ソフトウェア設計の改善
  - ソフトウェアを理解しやすいものに
  - バグを見つけ出す
  - プログラミングの効率があがる
- リファクタリングとパフォーマンス
  - リファクタリング⇔パフォーマンス

# 3章では

- リファクタリングのタイミングについて  
– よくわからん

経験によってコードの臭いを嗅ぎ分けることができるようになるろう

- 不吉な臭い（兆候）について説明します  
– 紹介したようなコードが目についたらリファクタリングしたらどうでしょうか？

# 重複したコード

- 同じようなコードが複数個所にある

```
void A(double value){  
    this. func_y(value)  
    this. func_z(value)  
    System.out.println(value);  
}
```

```
void B(double v){  
    this. func_y(v)  
    this. func_z(v)  
    System.out.println(v);  
}
```

- メソッドの抽出（6章）やクラスの抽出（7章）を行うことで解決できる

# 長すぎるメソッド

- 理解が困難
- 短くするためには
  - 命名を適切におこなう
  - メソッドの抽出 (6章)
  - 問い合わせによる一時変数の置き換え (6章)
  - 条件記述の分解 (9章)

# 巨大なクラス

- インスタンス変数の持ちすぎなど、重複したコードが存在する可能性が高くなる
- クラスの抽出（7章）をおこなうことや、GUIの部分であればデータと振舞を別にするなどなどで解決する

# 長すぎるパラメータリスト

- ひとつひとつのパラメータが何を意味するのか理解しづらくなる

```
void checkTime(int Num,String s_Date,String e_Date,  
               Access access,List<Integer>flags){ . . .
```

- 非オブジェクト指向言語
  - グローバル変数が悪さをすることが多々あるので必要なデータはパラメータで渡す
- オブジェクト指向言語
  - オブジェクトに問い合わせればいい

# 変更の偏り・変更の分散

- クラス内で変更する理由が複数あると分かりづらい
- 例：同一クラス内において
  - データベースの変更には3つのメソッド
  - 入出力の変更には4つのメソッドのように変更する原因が複数あるときはクラスを2つに分けた方がすっきりする

変更要求と変更クラスが1対1の関係になるといい

# 特性の横恋慕

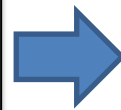
- 処理と処理に必要なデータを1つにまとめることがオブジェクト指向の概念の1つ
- 他のクラスに興味をもつような設計
  - 他のオブジェクトのgetメソッドの濫用
  - メソッド内で様々なクラスのデータを参照

```
void X(){  
    int value_ = 0;  
    value_ = c.sum(a.getValue(), b.getValue());  
    value_ = this. func_x(value_);  
}
```

# データの群れ

- データがグループとなって、クラスのフィールドやメソッドのシグニチャなど様々な個所に現れることがある
  - オブジェクトとして1か所にまとめるべき

```
void X(){  
    func_y(startDate,endDate);  
    func_z(startDate,endDate);  
}
```



```
void X(){  
    func_y(DateRange);  
    func_z(DateRange);  
}
```

# 基本データ型への執着

- データ型
  - Primitive型
    - integer, character, boolean などなど
  - レコード型
    - 配列, List, Map などなど（ただしここでは基本データ型の構造体のような扱い）
- データ型ではなくクラスを使おう！

# スイッチ文

- スイッチ文は重複したコードを生み出す問題児らしい
- ポリモーフィズムの検討
  - エレガントに解決できるらしい

# パラレル継承

- 例：スーパーカークラスとパトカークラスがすでに存在しているときに車クラスを作成すると
  - －スーパーカークラスとバスクラスの両方で車クラスを継承する必要がある
- クラス名の接頭辞が同じときは疑ってみては？
  - －スーパー**カー**とパト**カー**のように

# 怠け者クラス

- 十分な仕事をせず，見返りに合わないようなクラスは排除しよう
  - － 理解・保守のためのコストが無駄にかかる
- リファクタリングの過程で必要なくなることも

# 疑わしき一般化

- 「いつかこの機能が必要になるさ！」って誰かがいったものの・・・
- 大した働きが感じられない抽象クラスについては階層の平坦化（11章）等を試みて解決すべき

# 一時的属性

- インスタンス変数の値が、特定の状況でしか設定されていないオブジェクト
  - クラス内で定義された変数がほとんど使われないならクラスの構成要素としてまとめる

# メッセージの連鎖

- クライアントがあるオブジェクトにメッセージを送り，そのオブジェクトがさらに別のオブジェクトにメッセージを送る  
– 中間のオブジェクトの変更が大きく関わる

```
void X(){  
    func_y(a.getValue(b.getValue(c.getValue())));  
}
```

- 委譲の隠喩（7章）等を用いて解決する

# 仲介人

- 内部の詳細を外部からは一切見えないようにする（カプセル化）
- カプセル化が過剰となる場合
  - メソッドの大半が別のオブジェクトへの委譲
  - 仲介人の除去（7章）を考えてみる

# 不適切な関係

- クラス同士の仲が良すぎる
  - 双方向の参照関係，継承などで発生する
  - プライベートな部分を探しだすのも一苦労

```
void X() extends Y{  
    this.sum(y);  
    func_y(x);  
    .  
    .  
    .  
}
```

# その他①

- クラスのインターフェイス不一致
  - 処理が同じでシグニチャのみ違うもの
- 未熟なクラスライブラリ
  - 再利用しているライブラリ自体が未熟かも
- データクラス
  - `get`と`set`しかメソッドを持たないようなクラスはアクセスを過剰にうけるのでよくない

## その他②

- 相続拒否
  - 相続した機能の一部しか利用してないならメソッドの引き下げ・引き上げを行う（11章）
    - ただししない方がいいこともあるので注意
- コメント
  - コメント書こうね！
  - ただし，丁寧にコメントが書かれているところは分かりにくいコードを紛らわすためが多いから修正した方がいいかも

# 不吉な臭いを示すコード

- 重複したコード
- 長すぎるメソッド
- 巨大なクラス
- 長すぎるパラメータリスト
- 変更の偏り・変更の分散
- 特性の横恋慕
- データの群れ
- 基本データ型への執着
- スイッチ文
- パラレル継承
- 怠け者クラス
- 疑わしき一般化
- 一時的属性
- メッセージの連鎖
- 仲介人
- 不適切な関係
- クラスのインターフェイス不一致
- 未熟なクラスライブラリ
- データクラス
- 相続拒否
- コメント

この章をわかりやすく紹介してます

- <http://objectclub.jp/technicaldoc/refactoring/refact-smell>
- <http://p-monster.hatenablog.com/entry/2013/02/19/200414>