リファクタリング超入門

2000年4月7日

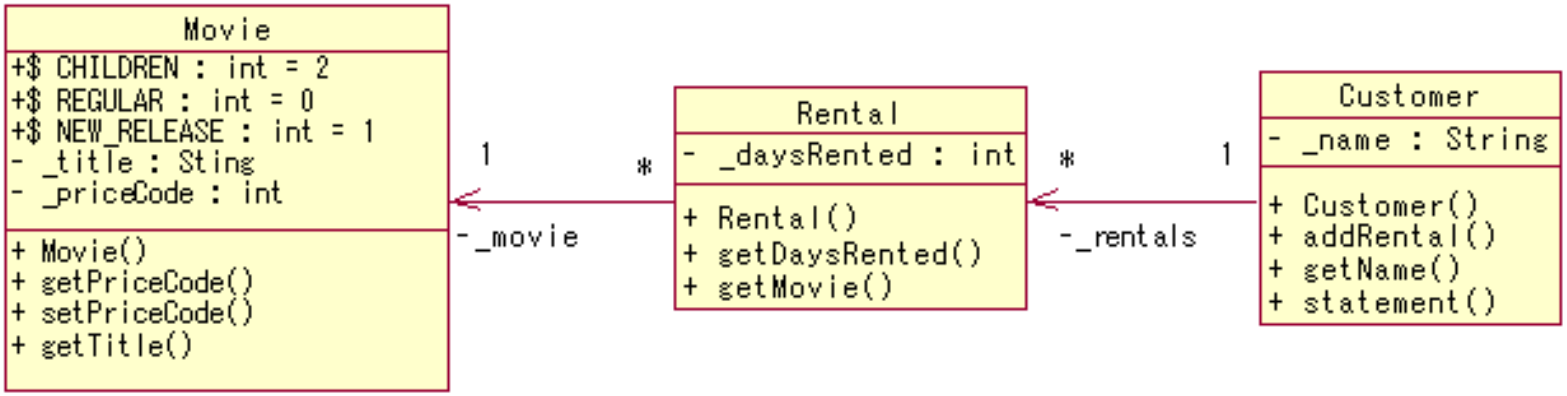
株式会社オージス総研
オブジェクトテクノロジー・ソリューション部
平澤　章

アジェンダ

- ビデオレンタル店のコード例
- リファクタリングの原則
- コードの不吉な匂い
- リファクタリングカタログ
- 金言集
- 最後に

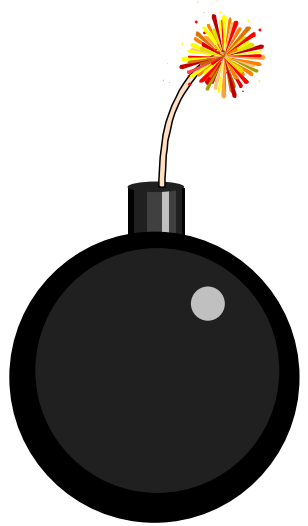
ビデオレンタル店のコード例

⌘このコードの良い点、悪い点を考えてください。



仕様追加！

- 次のような仕様が追加されることがわかりました。
  - Web対応
    - レシートをHTMLで出力して、Webブラウザで見ることができるようにしたい
  - 映画の区分方法の変更
    - Regular, New Release, Children の他にも Music, Discountなどを追加したい
  - さて、どうしますか？
- さらに以下のような変更も予想されます。
  - 料金計算ルールの変更
  - レンタルポイントの計算方法の変更



アジェンダ

- ビデオレンタル店のコード例
- リファクタリングの原則
- コードの不吉な匂い
- リファクタリングカタログ
- 金言集
- 最後に

リファクタリングとは

- 外部仕様を変更せずに、理解しやすく修正が容易になるように、ソフトウェアの内部構造を改善すること

リファクタリングを行う目的

- ソフトウェアのデザインを向上させる
  - 仕様追加や変更を繰り返しても、構造を劣化させない
  - コードの重複を排除する
- ソフトウェアを理解しやすくする
  - 将来、別の担当者がそのコードを修正するかも知れない。
  - 自分自身でさえ、3ヶ月後にそのコードを覚えている保証はない！

リファクタリングを行う目的（続き）

- バグを見つける
  - コードが理解しやすくなると、バグを見つけやすくなる
- より速くプログラミングをする
  - すべての結果として、より速くコーディングができる
    - 元のコードの理解に時間を取られない
    - 重複コードの修正に時間を浪費しない

いつリファクタリングをすべきか？

- 必要になった時点で随時行う
  - 3回、同じコードを書きそうになった時
  - 機能追加をする時
    - もし設計がこうだったら、機能追加がずっと楽なのに！
  - バグ修正をする時
    - コードを理解するためにリファクタリングを行う
  - コードレビューをする時
    - よいデザインやコーディング作法を教育する

リファクタリングの問題点、課題

- データベースの変更
  - データベースのスキーマ変更はリスクが多い
- インタフェース変更
  - リファクタリングは、しばしばインタフェースの変更を伴う
- リファクタリングを避けるとき
  - 全面書き直しの方が手っ取り早い
  - 期限が間近に迫っている

リファクタリングと設計

- 事前設計を重視する考え方
  - 設計こそが鍵で、プログラミングは機械的作業
  - しかし…

    設計時には順調に考えが進むが、それは隙だらけ！

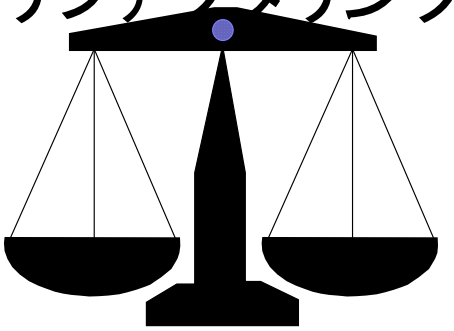
- Alistair Cockburn

リファクタリングと設計

- 事前設計を軽視する考え方
  - 事前の設計を一切行わず、思いつくままにコーディングして、ともかく動作させる。
    その後リファクタリングで形を整える。
  - しかし…
    もっとも効率的なやり方とは言えない。
    - XPを実践する場合でも、
      通常はある程度の事前設計を行う。

リファクタリングと設計 - まとめ

- 設計とリファクタリングのバランスを取る
  - 事前設計は行う。
    - しかし、唯一無二の完璧なデザインをする必要はない。
    - 先の先までの柔軟性を必ずしも考慮する必要はない。
  - 適切なときに、自信を持ってリファクタリングを行う。



リファクタリングとテスト

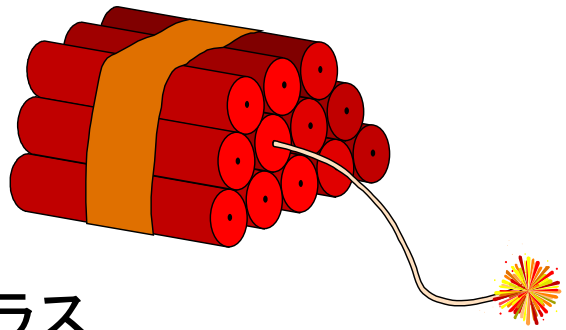
- テストは頼みの綱。
  - 細かい修正のサイクルでテストを頻繁に行う
    - 人間は間違いを犯すもの
    - テストの周期を短かければ、バグは見つけやすい
- 一連のテスト群を用意する。
  - 自己テストコードを含め、できる限り自動化する
- テストケースは重点主義で設定する。
  - 一番怪しい部分を先にテストする。
- 単体テストのためのフレームワーク - JUnit

アジェンダ

- ビデオレンタル店のコード例
- リファクタリングの原則
- コードの不吉な匂い
- リファクタリングカタログ
- 金言集
- 最後に

コードの不吉な匂い

- 「いつリファクタリングをすべきか？」を示す不吉な匂い（＝アンチパターン）
  - 重複したコード
    - 同じようなコードが2箇所以上に書かれている
  - 長すぎるメソッド
    - 長々と手続きが書かれたメソッド
  - 巨大なクラス
    - あまりに多くの仕事をしているクラス



コードの不吉な匂い（続き）

- 多すぎる引数
  - あまりに引数が多いと、1つ1つが何を意味しているか理解しづらい
- 怠け者クラス
  - あまり仕事をしていない、影の薄いクラス
- コメント
  - コメントを書かなければ理解できないようなコード
- *これらを含めて、全部で21種類の不吉な匂いが紹介されている。*

アジェンダ

- ビデオレンタル店のコード例
- リファクタリングの原則
- コードの不吉な匂い
- リファクタリングカタログ
- 金言集
- 最後に

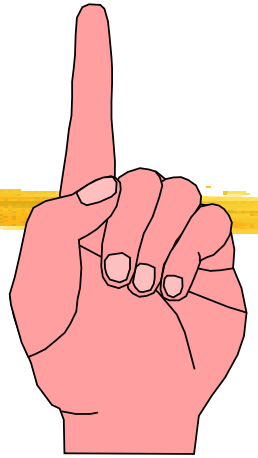
リファクタリングの記述形式

- 名前
  - 共通語彙を形成するための名前
- 要約
  - リファクタリングが必要な状況と解決法
- 動機
  - リファクタリングを行うべき理由
- 手順
  - リファクタリングを実施する簡潔な手順
- 例
  - 簡単なコード例を含む、リファクタリングの利用例

リファクタリングカタログ

- 7つのカテゴリ、72のリファクタリング
  - メソッドの構成(9)
  - オブジェクト間での特性の移動(8)
  - データの再編成(16)
  - 条件式の単純化(8)
  - メソッド呼び出しの単純化(15)
  - 継承の取り扱い(12)
  - 大きなリファクタリング(4)

代表的なリファクタリング



- メソッドの抽出
- メソッドの移動
- オブジェクトによるデータ値の置き換え
- 条件記述の分解
- メソッド名の変更
- 委譲による継承の置き換え

メソッドの抽出(1)

- 状況
  - 長すぎるメソッド、コメントなしには理解できないメソッドがある。
- 解決法
  - コードの断片をメソッドとして抽出し、適切な名前をつける。
- 例
  - ビデオレンタル店の statement メソッドから、料金計算のコードを抜き出して、amountFor(Rental rental) とする。

メソッドの抽出(2)

- 手順
  - 適切な名前をつけた新しいメソッドを作る。
  - 抽出したいコード部分を新しいメソッドにコピーする。
  - そのコードが参照しているローカル変数があれば、メソッドの引数にする。
  - そのコードが変更する値が1つあれば、メソッドの戻り値にする。
  - コンパイルする。
  - 抽出元のコードを、新しいメソッドを呼び出すように修正する。
  - コンパイルしてテストする。

メソッドの移動

- 状況
  - あるクラスで定義されているメソッドが、そのクラスの特性（属性＋メソッド）よりも別のクラスの特性を（／から）多く使っている（／使われている）。
- 解決法
  - そのメソッドを、別のクラスに定義する。元のメソッドは単純な委譲とするか、取り除く。
- 例
  - ビデオレンタル店の料金計算メソッドを Customer クラスから Rental クラスへ移動する

オブジェクトによるデータ値の置き換え

- 状況
  - 追加のデータや特別な振る舞いが必要なデータ項目がある。
- 解決法
  - そのデータ項目をオブジェクトに変更する。
- 例
  - 電話番号を String から TelephoneNumber クラスに変更する。

条件記述の分解(1)

- 状況
  - 複雑な条件記述（if-then-else）がある。
- 解決法
  - その条件記述部と then 部および else 部から、メソッドを抽出する。
- 補足
  - 複雑な条件分岐は、プログラムの中でもっともわかりづらい箇所である。
  - 適切な名前をつけたメソッドを抽出することで、コードが何を／なぜ行っているのかを表現できる。

条件記述の分解(2)

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))
        charge = quantity * _winterRate + _winterServiceCharge;
else
        charge = quantity * _summerRate;
```

```
if (notSummer(date))
        charge = winterCharge(quantity);
else
        charge = summerCharge(quantity);
```

メソッド名の変更

- 状況
  - メソッドの名前が、そのメソッドの目的を正しく反映していない。
- 解決法
  - メソッドの名前を変更する。
- 補足
  - 不適切な名前がつけられたメソッドは、断固として変更すべき。
  - 引数の名前も同様。

委譲による継承の置き換え

- 状況
  - サブクラスが、スーパークラスの一部のインタフェースだけを使っている。
- 解決法
  - スーパークラス用のフィールドを作成し、メソッドをスーパークラスに委譲するように変更する。
- 例
  - JDK1.1のutilパッケージでは、Stack が Vector のサブクラスとして定義されていた。

デザインパターンを適用するリファクタリング

- State/Strategyによるタイプコードの置き換え
  - タイプコードにより振る舞いが変わるクラスに、State/Strategyパターンを適用する。
- Factory Methodによるコンストラクタの置き換え
  - オブジェクトを生成するときに、単純な生成以上のことをしたい場合にFactory Methodパターンを適用する。
- Template Methodの形成
  - 異なるサブクラスの2つのメソッドが、類似の処理を同じ順序で実行しているときに、Template Methodパターンを適用する。

アジェンダ

- ビデオレンタル店のコード例
- リファクタリングの原則
- コードの不吉な匂い
- リファクタリングカタログ
- 金言集
- 最後に

金言集

- 設計判断の変更
  - ある週に正しく適正であった設計判断も、次の週にはそうでなくなります。それが問題ではなく、それについて何もしないことが問題なのです。
- 事前設計の位置づけ
  - 事前設計は行うのですが、そこで唯一無二の、完璧な解決策を見いだす必要はありません。
- テスト
  - 不完全なテストは、実行できない完璧なテストよりもましです。

金言集

- コメント
  - コメントが丁寧に書かれていたのは、実はわかりにくいコードを補うためだったと言うことがよくあります。
- コードの目的
  - あなたの書くコードは、第一に人間のためのものであり、コンピュータはその次であることを忘れてはなりません。

アジェンダ

- ビデオレンタル店のコード例
- リファクタリングの原則
- コードの不吉な匂い
- リファクタリングカタログ
- 金言集
- 最後に

感想

- パターン、イディオムに続く、設計者同士の共通語彙
- 「リファクタリング」カタログは、リファクタリング以外の目的にも使えそう。
  - 事前設計のレビューとして
  - オブジェクト指向技術の教育用題材として
    - Javaコードを通じて、具体的にわかるオブジェクト指向

書籍情報

- 原書
  - Refactoring - Martin Fowler (with Kent Beck, John Brant, William Opdyke & Don Roberts)
  - 1999年発刊
- 日本語訳
  - ピアソンより近日出版予定。