一言で言うと、これを読むことでプログラミングが楽しくなります。
プログラマー必読といってもよいのでは？
「動くコードはさわるな」とよく言われますけど、動くコードの振る舞いを変えることなく構造を「きれいに」する指針を与えてくれます。
今は結構 IDE がサポートしてくれてるので、重宝してます。
個人的にもっとも影響を受けた (仕事上) 本の一冊です。

まぁ実際問題として、「動いているコードをさわる」勇気はないんですけど・・・

以下、確認用メモ。

リファクタリング原則

定義

外部から見たときの振る舞いを保ちつつ、理解や修正が簡単になるように、ソフトウェアの内部構造を変化させること。
一連のリファクタリングを行って、外部からみた振る舞いを変えずにソフトウェアを再構築する。

コードの不吉なにおい

・重複したコード
・長すぎるメソッド
・巨大なクラス
・多すぎる引数
・変更の発散 (変更箇所が特定できない)
・変更の分散 (変更を行うたびにあちこちのクラスが少しずつ書き換わる)
・属性、操作の横恋慕 (あるメソッドが自分のクラスより、他のクラスに興味を持つ)
・データの群れ (複数個のデータがグループで出現)
・基本データ型への執着
・スイッチ文
・パラレル継承 (新たなサブクラスを定義するたび、別の継承木にもサブクラスを追加しなければいけない)
・怠け者クラス (十分な仕事をしないクラス)
・疑わしき一般化 (いつかこの機能が必要になるさ)
・一時的属性 (インスタンス変数の値が、特定の状況でしかセットされない)
・メッセージの連鎖 (連鎖の構造に依存)
・仲介人 (メソッドの大半が委譲しているだけのクラス)
・不適切な関係
・クラスのインターフェース不一致
・未熟なクラスライブラリ
・データクラス
・接続拒否 (継承した操作、属性が利用されず混乱が引き起こされている)
・コメント (非常にわかりにくいコードを補うためのコメント)

リファクタリングカタログ

■メソッドの構成

メソッドの抽出

1. まとめられるコードの断片がある
2. 断片をメソッドにして、目的をあらわすような名前をつける

メソッドのインライン化

1. メソッドの本体が、名前をつけるほどではなく明らか
2. メソッド本体をコール元にインライン化

一時変数のインライン化

1. 簡単な式によって一度だけ代入される一時変数があり、他のリファクタリングの障害となっている
2. 一時変数への参照を式で置き換え

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000);
```

```
return (anOrder.basePrice() > 1000);
```

問い合わせによる一時変数の置き換え

1. 一時変数を使って式の結果を保持している
2. 式をメソッドに抽出する。一時変数への参照を式へ置き換える。新たなメソッドが他のメソッドでも利用できるようになる。

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

```
if (basePrice() > 1000)
      :
double basePrice() {
  return _quantity * _itemPrice;
}
```

説明用変数の導入

1. 複雑な式がある
2. 式の結果または部分的な結果を目的を説明する名前をつけた一時変数に代入する

```
if ( (platform.tuUpperCase().indexOf("MAC") > -1 ) &&
     (browser.toUpperCase().indexOf("IE") > -1) {
     :
}
```

```
final boolean isMacOs = platform.tuUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
if (isMacOs && isIEBrowser) { ... }
```

一時変数の分離

1. 複数回代入される一時変数があるが、ループ変数でも、一時変数を集めるものでもない
2. 代入ごとに一時変数を分ける

```
double tmp = 2 * (_height * _width);
System.out.println(tmp);
tmp = _height * _width;
System.out.println(tmp);
```

```
final double perimeter  2 * (_height * _width);
System.out.println(perimeter);
final double area = _height * _width;
System.out.println(area);
```

パラメータへの代入の除去

1. 引数への代入が行われている
2. 一時変数を使う

オブジェクトによるメソッドの置き換え

1. 長いメソッドで、メソッドの抽出を適用できないようなローカル変数の使い方をしている
2. メソッド自身をオブジェクトとして、ローカル変数をそのオブジェクトのフィールドとする。

```
class Order...
  double hoge(){ ... }
  double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    /* 長い処理 */
      :
    primaryBasePrice * hoge();
}
```

```
class Order...
  double price() {
    return new PriceCaluculator(this,
                                primaryBasePrice,
                                secondaryBasePrice,
                                tertiaryBasePrice
                                ).compute();
  }
}
class PriceCaluculator...
  private Order order;
  private double primaryBasePrice;
```

```
  private double secondaryBasePrice;
  private double tertiaryBasePrice;
  public PriceCaluculator(Order order,
                          double primaryBasePrice,
                          double secondaryBasePrice,
                          double tertiaryBasePrice ) {
    this.order = order;
    this.primaryBasePrice   = primaryBasePrice;
    this.secondaryBasePrice = secondaryBasePrice;
    this.tertiaryBasePrice  = tertiaryBasePrice;
  }
  public double compute() {
       :
    primaryBasePrice * order.hoge(); /* 元クラスへの特性へのコール */
       :
  }
}
```

アルゴリズムの取替え

1. メソッドの本体を新たなアルゴリズムで置き換える

■オブジェクト間での特性の移動

メソッドの移動

1. あるクラスでメソッドが定義されているが、そのクラスの特性よりも他のクラスの特性の方が、そのメソッドの使用、被使用頻度が高い
2. 同様の本体を持つメソッドを多用するクラスにも作成する。元のメソッドは委譲とするか、取り除く

フィールドの移動

1. あるクラスに定義されているフィールドが、他のクラスから使用されることの方が多い
2. 移動先クラスに新たなフィールドを作って、利用先側をすべて変更する

クラスの抽出

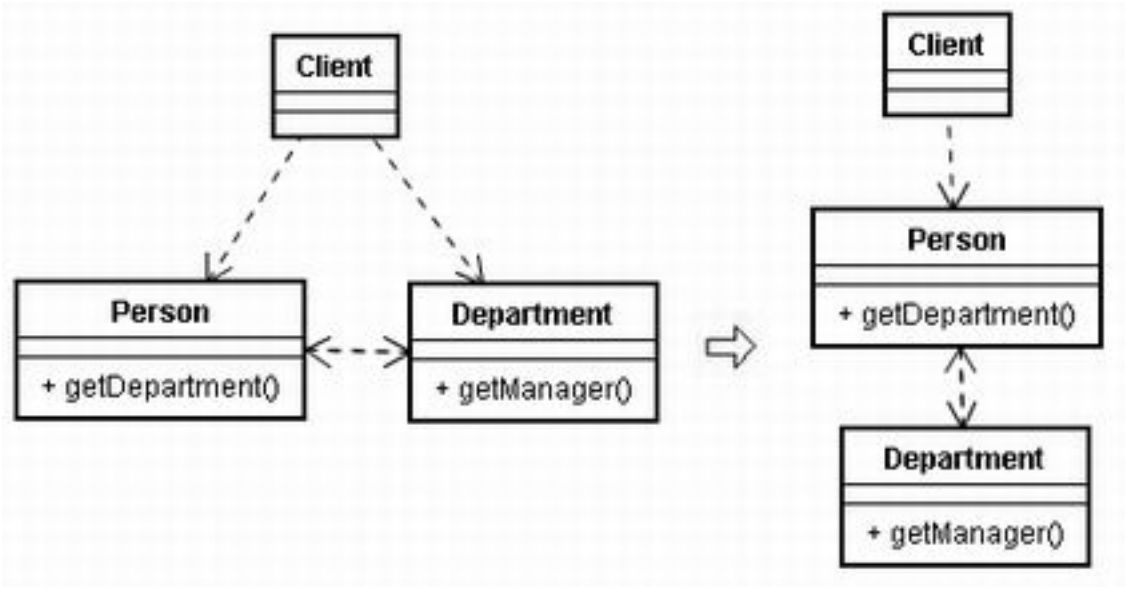
1.2 つのクラスでなされるべき作業を 1 つのクラスで行っている
2. クラスを新たに作成し、適当なフィールドとメソッドを移動する

クラスのインライン化

1. クラスのやっていることはたいしたことではない
2. 別のクラスに特性を移動し、削除する

委譲の隠蔽

1. クライアントがあるオブジェクトの委譲クラスをコールしている
2. サーバーにメソッドをつくって委譲を隠す



仲介人の除去

1. クラスがやっていることは単純な委譲だけ
2. クライアントに委譲オブジェクトを直接コールさせる

委譲の隠蔽の逆

外部メソッドの導入

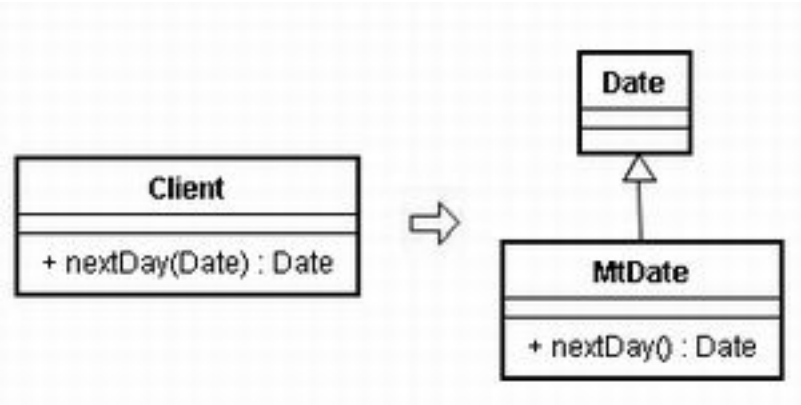
1. 利用中のサーバクラスにメソッドを追加する必要があるが、そのクラスを変更できない
2. クライアントクラスに、サーバクラスのインスタンスを第 1 引数にとるメソッドを作る

```
Data newStart = new Date(previousEnd.getYear(),
                         previousEnd.getMonth(),
                         previousEnd.getDate() + 1);
```

```
Date newStart = nextDay(previousEnd);
private static Date nextDay(Date arg) {
  return new Date(arg.getYear(), arg.getMonth(), arg.getDate() + 1);
}
```

局所的拡張の導入

1. 利用中のサーバクラスにメソッドをいくつか追加する必要があるが、クラスを変更できない
2. 追加されるメソッドを備えた新たなクラスを作る (サブクラス、ラッパー)



■データの再編成

自己カプセル化フィールド

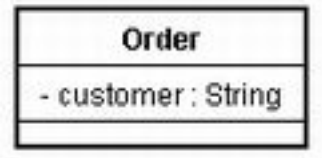
1. フィールドを直接アクセスしているが、結合関係が煩わしくなってきた
2.get メソッド、set メソッドを作成し、それだけを使ってアクセスするように変更する

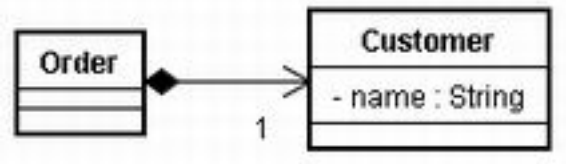
```
private int _low, _high;
boolean includes(int arg) {
  return arg >= _low && arg <= _high;
}
```

```
private int _low, high;
boolean includes(int arg) {
  return arg >= getLow() && arg <= getHigh();
}
int getLow()  { return _low;  }
int getHigh() { return _high; }
```

オブジェクトによるデータ値の置き換え

1. いくつかのデータや振る舞いが必要なデータ項目がある
2. そのデータ項目をオブジェクトに変える





値から参照への変更

1. 同じインスタンスが多数存在するクラスがあり、1 つのオブジェクトに置き換えたい
2. そのオブジェクトを参照オブジェクトに変える

```
/* 顧客クラス */
class Customer {
  public Customer(String name) { _name = name; }
  pulbic String getName()      { return _name; }
  private final String _name;
}
/* 注文クラス */
class Order...
  public Order(String customerName) {
      _customer = new Customer(customerName);
  }
  public void setCustomer(String customerName) {
      _customer = new Customer(custmerName);
  }
  public String getCustomerName() {
      return _customer.getName();
  }
  private Customer _customer;
}
/* クライアントコード */
private static int numberOfOrdersFor(Collection orders, String customer) {
  int result = 0;
  Iterator iter = orders.iterator();
  while (iter.hasNext()) {
    Order each = (Order) iter.next();
    if (each.getCustomerName().equals(customer)) result++;
  }
  return result;
}
```

```
class Customer...
  private Customer(String name) { _name = name; }
  static void loadCustomers() {
    new Customer("Lemon Car Hire").store();
          :
  }
  private void store() {
    _instandces.put(this.getName(), this);
  }
  public static Customer getNamed(String name) {
    return (Customer) _instances.get(name);
  }
  private static Dictionary _instances = new Hashtable();
}
```

顧客名毎に Customer オブジェクトが唯一存在するようにする

参照から値への変更

1. 小さくて、不変で、コントロールが煩わしい参照オブジェクトがある
2. 値オブジェクトに変える

オブジェクトによる配列の置き換え

1. 配列の各要素が、それぞれ異なる意味を持っている
2. その配列を要素ごとに対応したフィールドをもつオブジェクトに置き換える

```
String[] row = new String[3];
row[0] = "Liverpool";
row[1] = "15";
```

```
Performance row  new Performance();
row.setName("Liverpool");
row.setWins("15");
```

観察されるデータの複製

1. ある GUI コントロールでのみ有効なドメインデータがあり、ドメインメソッドからもアクセスする必要がある
2. データをドメインオブジェクトにコピーして、それらを同期させるためのオブザーバを設ける

単方向関連の双方向への変更

1. 2 つのクラスが互いにその特性を使う必要があるが、単方向のリンクしかない
2. 逆ポインタを加えて、両方の集合を更新するように更新操作を変更する

双方向関連の単方向への変更

1. 双方向関連があるが、一方のクラスはもはや他方の特性を必要としていない
2. 不要になった関連の一方を削除する

シンボリック定数によるマジックナンバーの置き換え

1. 特別な意味を持った数字リテラルがある
2. 定数を作り、それにふさわしい名前をつけて、そのリテラルを置き換える

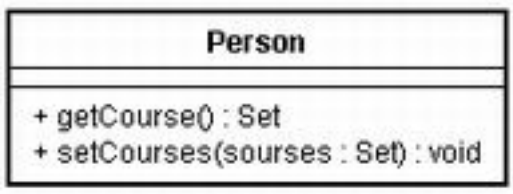
フィールドのカプセル化

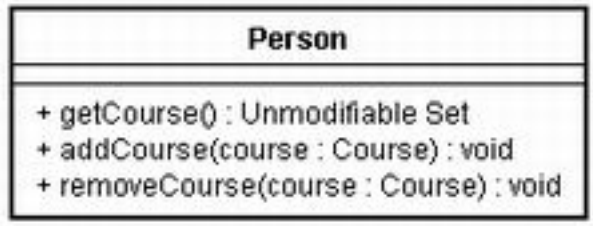
1. 公開フィールドがある
2. 非公開にして、アクセサを用意する

コレクションのカプセル化

1. メソッドがコレクションを返す
2. 読み取り専用のビューを返して、追加と削除のメソッドを提供する

```
java.util.Collections.unmodifiableSet
```



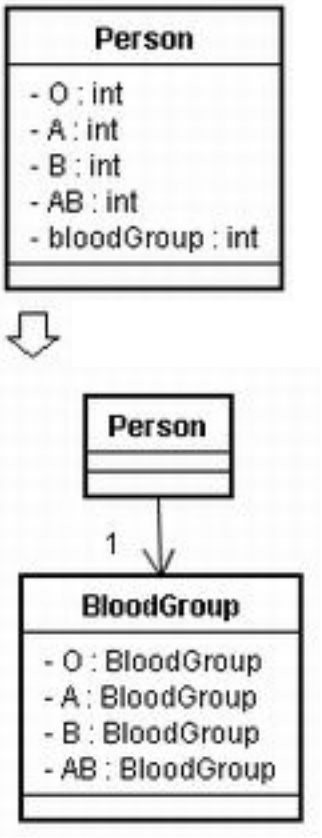


データクラスによるレコードの置き換え

1. 古いプログラミング環境のレコード構造とインターフェースをとる必要がある
2. そのレコード用に、振る舞いを持たないデータオブジェクトを作る

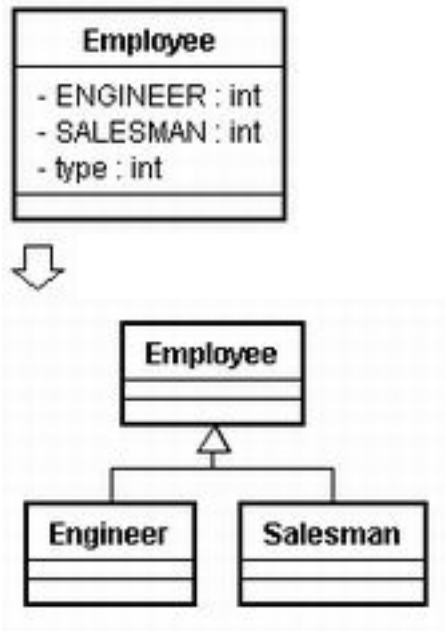
クラスによるタイプコードの置き換え

1. 振る舞いに影響しない数字のタイプコードを持つクラスがある
2. その数字を新しいクラスで置き換える



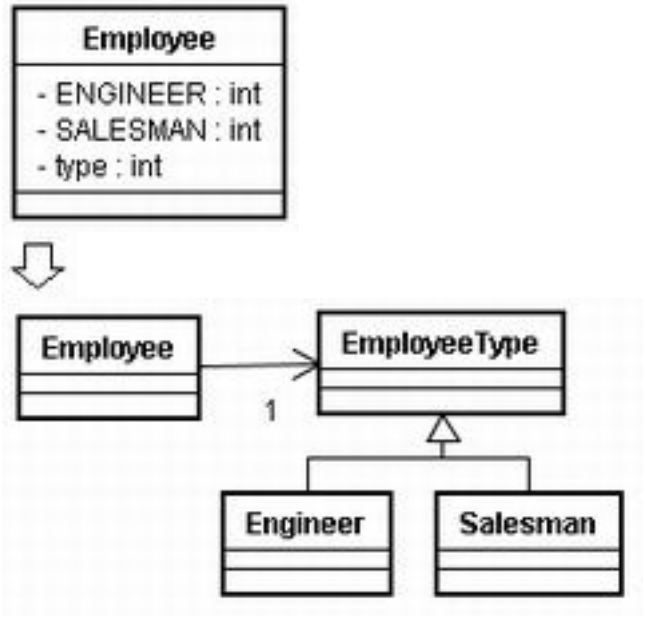
サブクラスによるタイプコードの置き換え

1. クラスの振る舞いに影響を与える不変のタイプコードがある
2. そのタイプコードをサブクラスに置き換える



State/Strategy によるタイプコードの置き換え

1. クラスの振る舞いに影響するタイプコードがあるが、サブクラス化はできない
2. 状態オブジェクトでタイプコードを置き換える



フィールドによるサブクラスの置き換え

1. 定数データを返すメソッドだけが異なるサブクラスがある
2. メソッドをサブクラスのフィールドに変更して、サブクラスを削除する

■条件記述の単純化

条件記述の分解

1. 複雑な条件記述 (if-the-else) がある
2. 条件記述部を then 部および else 部からメソッドを抽出する

```
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);
```

条件記述の統合

1. 同じ結果を持つ一連の条件判断がある
2.1 つの条件記述にまとめてから抽出する

```
double disabilityAmount() {
  if (_seniority < 2)        return 0;
  if (_monthisDisabled > 12) return 0;
  if (_isPartTime)           return 0;
```

```
double disabilityAmount() {
  if (isNotEligibleForDisablity()) return 0;
}
```

重複した条件記述の断片の統合

1. 条件式のすべての分岐に同じコードの断片がある
2. それを式の外側に移動する

```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
} else {
  total = price * 0.98;
  send();
}
```

```
if (isSpecialDeal()) {
  total = price * 0.95;
} else {
  total = price * 0.98;
}
send();
```

制御フラグの削除

1. 一連の論理型の式に対して制御フラグとして機能する 1 つの変数がある
2. 代わりに break か return を使う

ガード節による入れ子条件記述の置き換え

1. メソッドに正常ルートが不明確な条件付振る舞いがある
2. 特殊ケースすべてに対してガード節 (メソッド内で、処理がメインロジックに到達するのを防ぐためのコード) を使う

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
    if (_isSeparated) result = separatedAmount();
    esle {
      if (_isRetired) result = retiredAmount();
      else result = normalPayAmount();
    }
  }
  return result;
}
```

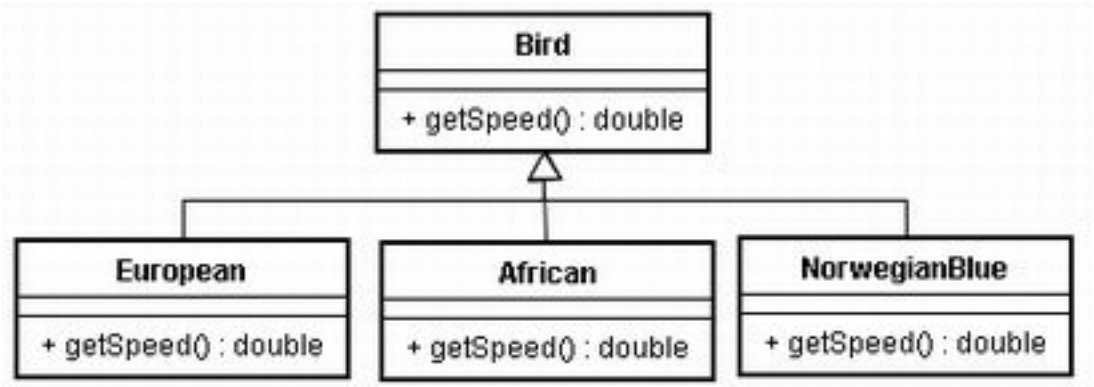
```
double getPayAmount() {
  if (_isDead)      return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired)   return retiredAmount();
  return normalPayAmount();
}
```

出口が 1 つだけというルールは本当は有益ではありません。1 つにすることでメソッドが明瞭になるならば出口は 1 つでいいが、そうでなければ使うべきではない

ポリモーフィズムによる条件記述の置き換え

1. オブジェクトのタイプによって異なる振る舞いを選択する条件記述がある
2. 条件記述の各アクション部をサブクラスのオーバーライドメソッドに移動する。元のメソッドは abstract にする

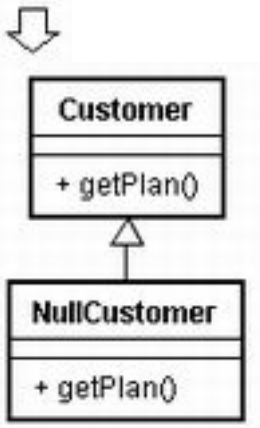
```
double getSpeed() {
  switch (_type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
    case NORWEGIAN_BLUE:
      return (_isNailed)?0:getBaseSpeed(_voltage);
  }
  throw new RuntimeException(" ここには来ないはず ");
}
```



ヌルオブジェクトの導入

1.null 値のチェックが繰り返し現れる
2.null 値をヌルオブジェクトで置き換える

```
if (customer == null) plan = billingPaln.basic();
else plan = customer.getPlan();
```



表明の導入

1. コードのある部分が、そのプログラムの状態について何らかの仮定を持っている
2. 仮定を表明を使って明示する

```
double getExpenseLimit() {
  /* 支出上限か優先プロジェクトか、どちらかを持つこと */
  return (_expenseLimit != NULL_EXPENSE) ?
          _expenseLimit:
          _primaryProject.getMemberExpenseLimt();
}
```

```
double getExpenseLimit() {
  Assert.isTrue(_expenseLimit != NULLEXPENSE || _primaryProject != null);
  return (_expenseLimit != NULL_EXPENSE) ?
                _expenseLimit:
                _primaryProject.getMemberExpenseLimt();
}
}
```

■メソッド呼び出しの単純化

メソッド名の変更

1. メソッドの名前がその目的を正しく表現できていない
2. メソッドの名前を変更する

引数の追加

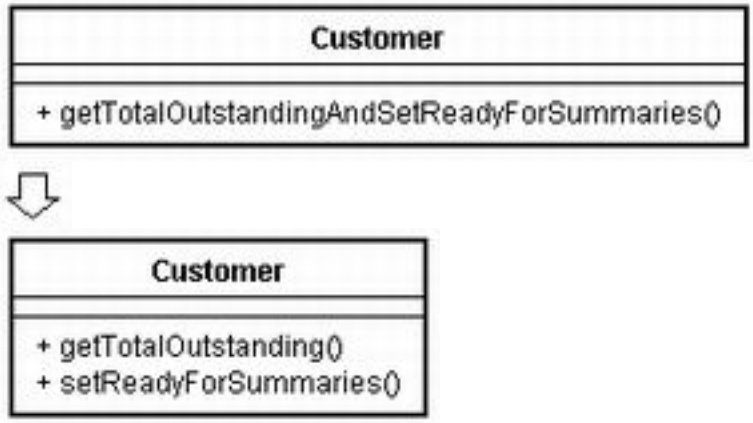
1. あるメソッドが、呼び出し元からより多くの情報を必要としている
2. 情報を渡すために引数を追加する

引数の削除

1. ある引数が、もはやメソッド本体から使われていない
2. 引数を削除する

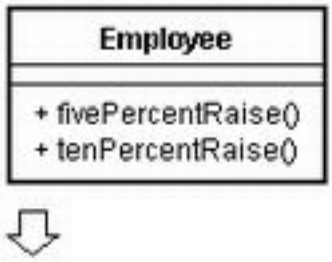
問い合わせと更新の分離

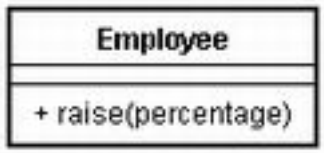
1. 1 つのメソッドが返す値と同時にオブジェクトの状態を変更している
2. 問い合わせようと行進用の 2 つのメソッドをそれぞれ作成する



メソッドのパラメータ化

1. 複数のメソッドが、異なる値に対してよく似た振る舞いをしている
2. 異なる値を 1 つの引数として受け取るメソッドを作成する





明示的なメソッド群による引数の置き換え

1. 引数の特定の値によって異なるコードが実行されるメソッドがある
2. 引数の値に値に対応する別々のメソッドを作成する

```
void setValue (String name, int value) {
  if (name.equals("height"))
    _heigth = vaue;
  else if (name.equals("width"))
    _width =  value
  else
    Assert.shouldNeverReacheHere();
}
```

```
void setHeight (int arg) {
  _height = arg;
}
void setWidth (int arg) {
  _width = arg;
}
```

オブジェクトそのものの受け渡し

1. あるオブジェクトから複数の値を取得し、それらの値をメソッド呼び出しの引数として渡している
2. オブジェクトそのものを渡す

```
int low  = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

メソッドによる引数の置き換え

1. あるオブジェクトがメソッドを呼び出し、その戻り値を別のメソッドの引数として渡している。受信側はそのメソッドを呼び出し可能である
2. 引数を削除し、受信側にそのメソッドを呼び出させる

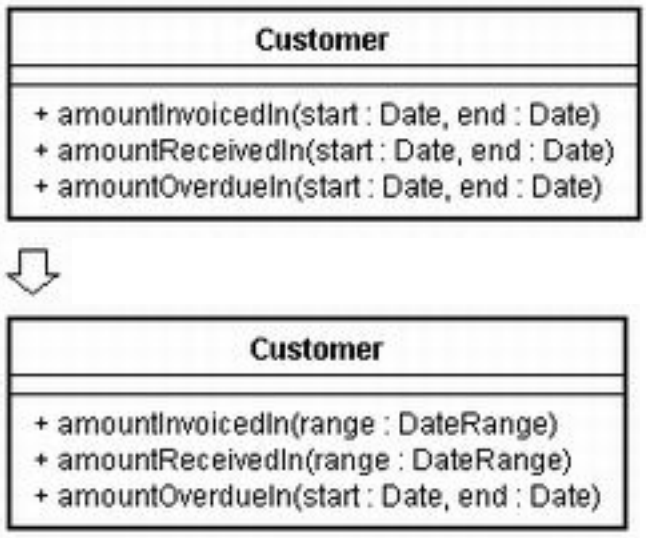
```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice, discountLevel);
```

```
int basePrice =¯ _quantity * _itemPrice;
```

```
double finalPrice = discountedPrice(basePrice);
```

引数オブジェクトの導入

1. 本来まとめて扱うべき一連の引数がある
2. それらをオブジェクトに置き換える



set メソッドの削除

1. フィールドの値が生成時に設定され、決して変更されない
2. そのフィールドに対するすべての set メソッドを削除する

メソッドの隠蔽

1. メソッドが自分の定義されているクラス以外からはまったく使用されていない
2. メソッドを非公開にする

Factory Method によるコンストラクタの置き換え

1. オブジェクトを生成する際に、単純な生成以上のことをしたい
2. ファクトリメソッドを使って、コンストラクタを置き換える

```
Employee (int type) {
  _type = type;
}
```

```
static Employee create(int type) {
  return new Employee(type);
}
```

ダウンキャストのカプセル化

1. メソッドが返すオブジェクトが、呼び出し側によってダウンキャストされる必要がある
2. ダウンキャストをメソッド内に移動する

```
Object lastReading() {
  return readings.lastElements();
}
```

```
Reading lastReading() {
  return (Reading) readlings.lastElement();
}
```

例外によるエラーコードの置き換え

1. メソッドがエラーを示す特別なコードを返す
2. 代わりに例外を発生させる

条件判定による例外の置き換え

1. チェックされる例外を発生させているが、本来は呼び出し側が先にチェックすべきだる
2. 最初に条件判定をするように呼び出し側を修正する

```
double getValueForPeriod(int periodNumber) {
  try {
    return _values[periodNumber];
  } catch (ArrayIndexOutOfBoundsException e) {
    return 0;
  }
}
```

```
double getValueForPeriod(int periodNumber) {
  if (periodNumber >= _values.length) return 0;
  return _values[periodNumber];
}
```

■継承の取り扱い

フィールドの引き上げ

1.2 つのサブクラスが同じフィールドを持っている
2. そのフィールドをスーパークラスへ移動する

メソッドの引き上げ

1. 同じ結果をもたらすメソッドがふくすうのサブクラスに存在する
2. それらをスーパークラスに移動する

コンストラクタ本体の引き上げ

1. 複数のサブクラスに内容がほとんど同一のコンストラクタがある

2. スーパークラスのコンストラクタを作成して、サブクラスから呼び出す

メソッドの引き下げ

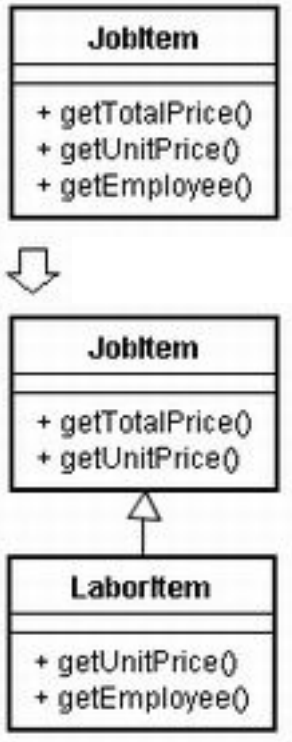
1. スーパークラスの振る舞いが、いくつかのサブクラスだけに関係している
2. そのメソッドをサブクラスに移動する

フィールドの引き下げ

1. フィールドがいくつかのサブクラスだけで使われている
2. そのフィールドを、サブクラスに移動する

サブクラスの抽出

1. あるクラス特定のインスタンスだけに必要な特性がある
2. その一部の特性を持つサブクラスを作成する



スーパークラスの抽出

1. 似通った特性を持つ 2 つのクラスがある
2. スーパークラスを作成して、共通の特性を移動する

インターフェースの抽出

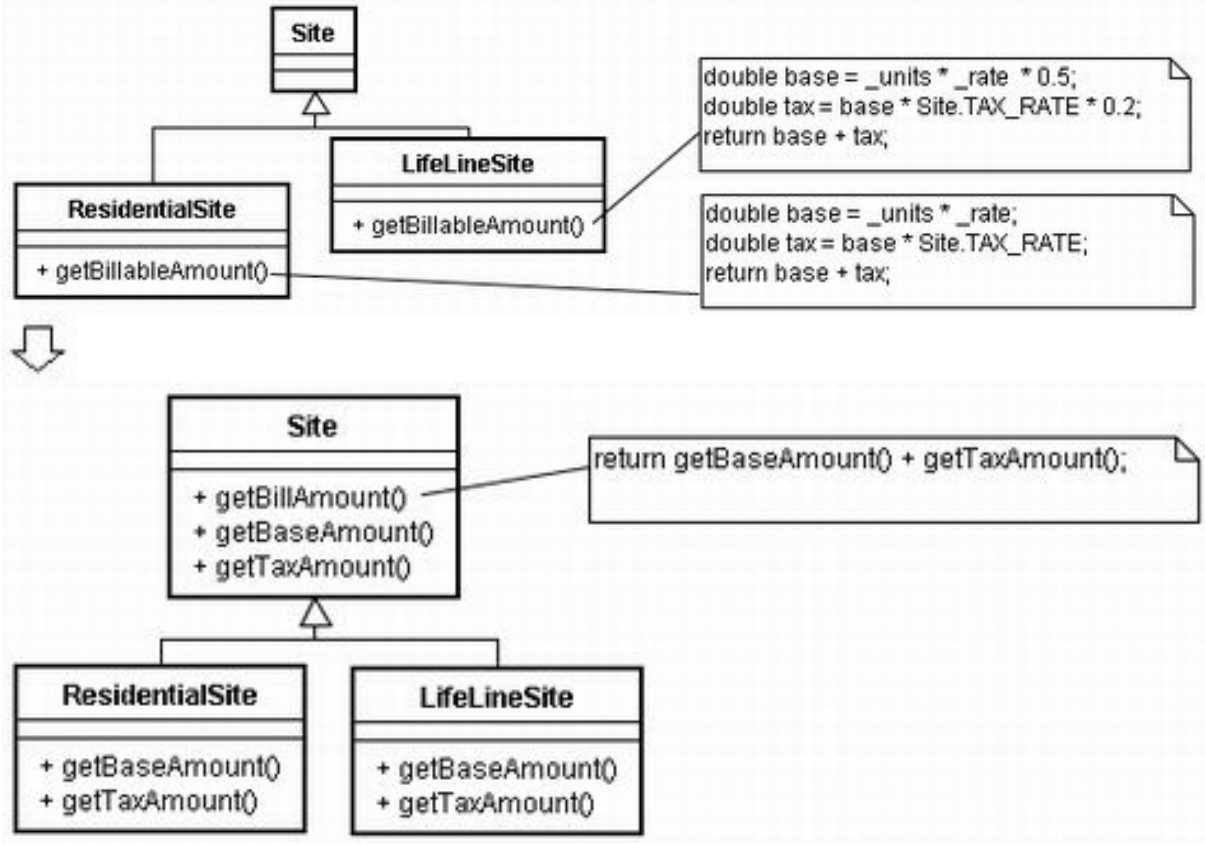
1. 複数のクライアントが、あるクラスのひとまとまりのインターフェースを使っている。または 2 つのクラス間でインターフェースの一部が共通である
2. 共通部分をインターフェースとして抽出する

階層の平坦化

1. スーパークラスとサブクラスにそれほど大きな違いがない
2. あわせてまとめてしまう

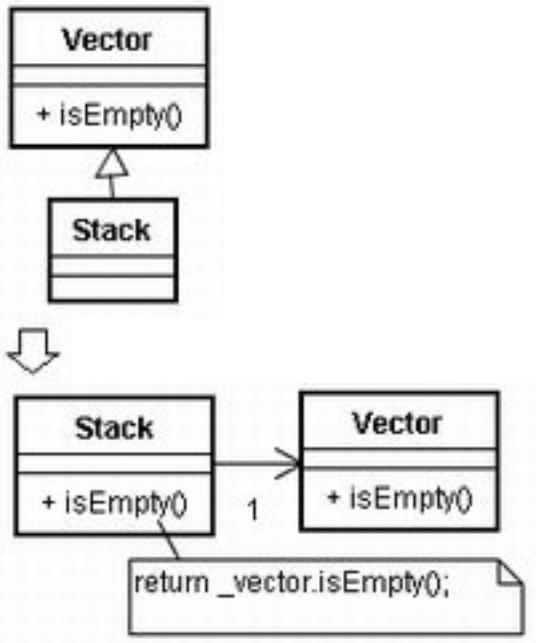
Templete Method の形成

1. 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が、各処理は異なっている
2. 元のメソッドが同一になるように、各処理を同じシグネチャを持つメソッドにする。そしてそれらを引き上げる。



委譲による継承の置き換え

1. サブクラスがスーパークラスの一部のインターフェースだけを使っている。あるいはデータを継承したくない
2. スーパークラス用のフィールドを作成して、メソッドをスーパークラスに委譲するように変更し、継承をやめる


Vector
+ isEmpty()
Stack
Stack
+ isEmpty()
1
Vector
+ isEmpty()
return _vector.isEmpty();