リファクタリングは手戻りではない

ロン・ジェフリーズ著　金田忠士訳

http://www.xprogramming.com/xpmag/refactoringisntrework.htm

09/05/2002

概 要

十分に設計を行わないでコーディングを始めてしまうと、プログラムは行き詰まり、メンテナンスができなくなってしまうと恐れてしまいがちである。その結果、手戻りに多大なコストがかかり、最悪、プロジェクトは中止に追いやられてしまう。設計をシンプルに留めてリファクタリングを行う、というのは手戻りとは異なるものなのだ。これでびくびくせずにプロジェクトを進めることができるようになるであろう。

1　イントロ

最近、Yahoo グループで、コーディングを始める前にどれぐらい設計をすれば十分かという議論があった。その議論の中で、私はリファクタリングは決して手戻りではない、ことを明言し、Alistair Cockburn による下記の発言に反駁した。「えっと、リファクタリングは厳密に言えば手戻りである。もし、手戻りにあたるものがまだあるとするならば、リファクタリングがそれだ。」Alistair は正式には訂正をしている。「手戻り」とは、辞書に従えば、改訂という意味であり、おそらくリファクタリングとは確かに改訂も含めたある種のインクリメンタル開発のことだ。しかし多くの人々――ソフトウェア関係者やマネージャも含めて――にとって、手戻りはいけないことだと考えられており、避けるべきだとされている。私はこの言葉を Alstair によるもっと丁寧な定義とは異なり、そういう意味で使っている。

手戻りとは、もちろん、同じ作業を何度も行うことだ。プログラミングは、リファクタリングというテクニックを用いれば何度も繰り返し実施することができるが、簡単な設計からリファクタリングを繰り返すソフトウェア開発は厳密には手戻りとは異なるものである。

一言で言うと、プログラミングはタイピングとは異なる、ということだ。これは、マーチンファウラーから借りてきた言葉である。長い説明となると本論文がそれにあたる。

2　単純な問題

テストファースト、簡単な設計とリファクタリングを用いて小さなプログラムを実装する。ここでは、Push と Pop ができるスタックの実装を行う。我々のミッションは、明確になっている開発を観察し、プログラムを開発する中でどのぐらいの手戻りが発生するのかを見ることである。この点にフォーカスするために、これまでにやったことのない方法でスタックを実装しよう。

さて、もちろん本稿の件について考えたことがあるし、スタックについてもかなりの量の知識をもっている。というのは、スタックが考案された時のことをよく知っているからである。私がまだやっていないことは、どのようにこのバージョンを実施しているか、を詳細に見せることである。この中で私は以前に用いた設計のことは考えないように努めるつもりだ。本稿を読んでいる間はピンクの象さんについては考えないで読んでもらいたい。

3　まずはテスト

本稿で Ruby を用いるのは、簡単で無駄な時間を使わないからだ。私はこの原稿を、このホテルをチェックアウトするまでにあげたいと思っている。今日の最初のテストは、確かに Ruby プロジェクトを正しくセットアップできていることを確かめることだ。これを見て欲しい:

```
class TestStack < TestCase

  def testHookup
    assert 1==1
  end

end
```

4　最初の Pop

最初の本当のテストは、スタックを作って Pop することだ。私は空のスタックを定義し、Pop を実行して nil が返る。テストはこうなる:

```
def testPopEmpty
    s = Stack.new
    assert_equal(nil, s.pop)
  end
```

スタックはまだ定義されていないので、これはまだ完璧ではない。よってここでクラスを書く:

```
class Stack
end
```

これでもテストはまだ動かない。Pop が定義されていないからだ。そこで nil が返るように Pop 操作を定義する:

```
  def pop
    return nil
  end
```

これでテストがちゃんと動くと思う、うん、動いたね。

5 Push して Pop

Pop としてはこれではまだ十分ではない。そこでスタックに何かを格納してそれを取り出すような新しいテストを書くことにする。

```
  def testPushPop
    s = Stack.new
    s.push("a")
    assert_equal("a", s.pop)
  end
```

これは期待どおりには動かない。Push が定義されてないからだ。‥やっぱり通らない。そこで入力を格納する Push を定義し、格納したものを返すように Pop を修正しよう。ここでは本当にバカみたいな方法をとる。誓って言うが、教育的な目的のためにこうするのだ。

```
  def pop
    return @saved
  end

  def push (x)
    @saved = x
  end
```

このテストは全てちゃんと動く。しかしながら、Ruby は、@saved が初期化されていないとワーニングを出す。これは、最初のポップのテストに起因するものである。でも、テストはちゃんと動いているのでこのワーニングについては気にしないことにする。

テストが動くように、全く新しいメソッド”push”を書いたが、pop メソッドの中の return 行もいじった。return nil となっていたところが return @saved になっている。

これは手戻りだろうか?　私はそうではない、と声を大にして言いたい。というのは、プログラミングというのは決してタイピングとは違うのだから。私が最初にこのメソッドを書いたとき、ほとんど何も設計を行わず、「テストでは nil が欲しいので nil を与えよう」と考えた。2 番目のあまり賢くないバージョンでは、少しだけ設計を行った。私は、「push した入力を格納し、後でそれを返すこと」と考えた。この作業は新しい作業であり、手戻りではない。タイピングそのものは手戻りである。古い 3 文字を新しい 6 文字に置き換えたのだ。(プログラム中には全部では 299 文字あるので、手戻りがあったのは約 1 %である。)

6　Push、Push そして Pop、Pop

じゃぁ、ここでちょっとかき回してみよう。厳しいテストをやってみるのだ。値を 2 つ Push して、2 回 Pop を行って正しい値が正しい (逆の) 順序で取り出せることを確認する。

```
def testPushTwice
  s = Stack.new
  s.push(1)
  s.push(2)
  assert_equal(2,s.pop)
  assert_equal(1,s.pop)
  assert_equal(nil,s.pop)
end
```

ここで、私は極端に走って、スタックが空になったら nul を返すことを確かめることにした。もし私がこのテストを testPush メソッドで行おうと考えていたら、テストはそこで失敗し、このプログラムは違った方向へ発展していたことだろう。でも私はそうとは考えなかった。そして今がある。我々はこのテストが失敗することを期待しよう (詳しく書けば、1 を Pop するときに失敗することを期待する。)... さあ。

さて、ここでやらないといけないことがいくつかある。私は、このスタックをリンクリストを用いて実装しようと決めた。だからここで、コード中に私の意図を表明しておこう。ここで私は以下のように、コードを数行を追加しないといけない：

```
def push (x)
  @saved = ListElement.new(x,@saved)
end
```

私の考えは、X と@saved に既に存在するもの全てを含む ListElement(このクラスはまだ存在していない) を新たに生成し、そのリスト要素を@saved

に格納しようということである。今、スタックをポップするために、@saved の最初の要素に格納されているアイテムを返し、@saved に次の要素を格納しなければならない。これはこのように見える... いや、ちょっと待って欲しい。私は今テストを走らせ、ListElement に関するエラーを見つけて、最初にそのエラーを訂正しようと思う。この作業は次の段階として程よい大きさだし、私はこれ以上たくさんのことを一度にやるつもりはないので、今やってしまおう。

```
class ListElement
  def initialize(object, link)
    @object = object
    @link = link
  end
end
```

よし、できた。でも、2つほどテストが通らない (我々の期待どおりに)。2つのエラーは両方とも、Pop の結果として予期された答えの代わりに ListElement を得たと苦情を述べている。これは驚くに値しない。Pop メソッドは@saved を答えている。我々は Pop メソッドが@saved 内にある ListElement 中のオブジェクトを返し、そして ListElement のリンクを@saved に格納するように変更しないといけない。じゃコードを書こう。これは少しトリッキーになるけど、許容範囲だ。

```
  def pop
    result = @saved.object;
    @saved = @saved.link
    return result
  end
```

読者諸君もお気づきのように、これらの LinkElement のオブジェクトメソッドやリンクメソッドはまだ実装していないことは分かっている。これを確認するためにテストを走らせてみよう。(私は、@saved が nil だった場合に発生する別のエラーも期待している)。

果たして、” オブジェクト” メソッドが定義されていないというエラーがでた。じゃぁ、実装しよう：

```
class ListElement
  def initialize(object, link)
    @object = object
    @link = link
  end
  attr_reader :object
  attr_reader :link
end
```

attr_reader 文ではこのクラスのための read アクセサを定義する。このくらいにして次のエラーに進もう。期待通りに、次のエラーでは、nil が”object”へ返らない。でも驚かないよ。@saved が nil だったときに pop に入ることに関して何か対処する必要があるのだ。また、@saved が初期化されていないというワーニングがまだ出ている。1 個めの問題をまずは解決してしまおう。pop に 1 行追加だけだ:

```
  def pop
    return nil if @saved == nil
    result = @saved.object;
    @saved = @saved.link
    return result
  end
```

@saved が nil の場合をチェックする。もし nil だったらスタックが空だという意味なので nil を返す。テストを実行しよう。全部走らせるのだ。今度はうるさいワーニングの番だ。@saved を nil にする初期化メソッドを追加してフィックスしよう:

```
  def initialize
    @saved = nil
   end
```

そしてテストは全て合格した。

7　ファイナルアンサー?

我々はわずかな時間で、ゼロからスタックを実装した。(私が望んでいたほどわずかという訳ではない。私はアトランタから帰宅する途中、30,000 フィート上空でこれをやり終えた)。このスタックは単純な設計とリファクタリングを使ってテストファーストで作った。これが、そのプログラムとテストの全てだ：

```
class TestStack < TestCase

  def testHookup
    assert 1==1
  end

  def testPopEmpty
    s = Stack.new
    assert_equal(nil, s.pop)
```

```
  end

  def testPushPop
    s = Stack.new
    s.push("a")
    assert_equal("a", s.pop)
  end

  def testPushTwice
    s = Stack.new
    s.push(1)
    s.push(2)
    assert_equal(2,s.pop)
    assert_equal(1,s.pop)
    assert_equal(nil,s.pop)
  end
end

class Stack

  def initialize
    @saved = nil
   end

  def pop
    return nil if @saved == nil
    result = @saved.object;
    @saved = @saved.link
    return result
  end

  def push (x)
    @saved = ListElement.new(x,@saved)
  end
end

class ListElement
  def initialize(object, link)
    @object = object
    @link = link
  end
  attr_reader :object
  attr_reader :link
end
```

8　手戻り?それとも新しい手順?

手戻りはあった? だろ? 修正したコードはほとんど全部が、あちこち文字を書き換えているけど追記の類だ。それに考え直すようなこともほとんどなかった。最初の「簡単な」テストの際に、私はシステムの全体的な構成を組

み入れた。クラスやメソッド定義、ほとんどが空でテストを走らせるためだけのコードだった。そして、ListElement を実装し、使い、拡張するというテストで支援された小さなフェーズをこなすことでラストスパートをかけた。

ここで、私は、手戻りだとは思わず、追加、追加、さらに追加のようだったと言える。確かに、書き換えた部分も少しはあったけど、タイピングの大部分や思考過程は新しいものであり、手戻りではなかったのだ。

リビングルームをペンキで塗る場合における手戻りというのは、壁全体を塗り直さなければならない場合のことであって、2, 3 箇所を直す場合のことではない。学期末のレポートにおける手戻りというのは、ほとんど全てを書き直さないといけない場合であって、スペルチェッカを走らせたり、タイポや文法を修正することではない。そして、ソフトウェアにおける手戻りというのは、コード全体をいじり直す場合のことであって、新しいことを追加したり、コードを若干移動させたりすることではない。

リファクタリングは手戻りではない。修正だ。でも、何度も繰り返しやってはいけないよ。