ソフトウェア設計における進化パターンに向けて

下滝 亜里†

ソフトウェア設計におけるコードの進化（発展）や変化自体をパターンとして捉え，文章化することを提案し，その動機を述べる．

Towards Evolution Patterns in Software Design

ASATO SHIMOTAKI†

We propose evolution patterns in software design and discuss why such patterns are needed.

1. はじめに

ソフトウェア設計は，容易ではなく，過去の経験やそれによって得られた知識が重要な要因となる活動である．そのため，開発者の得た経験を共有化することは，重要である．デザインパターンやリファクタリングは，文章化することによって経験を共有化した代表例である．

本稿では，ソフトウェア設計におけるコードの進化（発展）や変化自体をパターン（以下，進化パターンと呼ぶ）として記述することを提案する．

2. 動機

本研究は，次の三つを動機としている．一つ目は，デザインパターンやリファクタリングよりも広い範囲で開発者の経験を文章化することが必要，ということである．二つ目は，プログラミング言語の特徴を比較するためのシナリオの収集にある．三つ目は，ソフトウェア設計の観点から，どのようにしてソフトウェアが進化（発展），あるいは，変化していくのかの理解にある．以下、これら三つの動機について詳しく述べる．

一つ目の動機は，デザインパターンやリファクタリングは有益であるが，ソフトウェア設計における経験や知識を広範囲に渡ってカバーしているとはいいがたい，ということにある．そのため，開発者の経験をより広い範囲で記述するような方法が必要である．

リファクタリングは，定義により，プログラムの振る舞いを変更することなくコードの質を向上させるための手法である．したがって，振る舞いを変更するようなコードの修正については取り扱っていない．また，現在の多くのリファクタリングでは，特定のライブラリ使用によるリファクタリングにはほとんど焦点を当てられていないように見える．一方，デザインパターンの記述の多くは，あるパターンが適用された後の記述に重点を置いているように見える．そのため，具体的なコード例を用いることによるパターン適用前後の比較はあまり行われていない．しかし，そのような視点からパターンを理解することは重要である[1].

二つ目の動機は，プログラミング言語の特徴を比較するためのシナリオの収集にある．たとえば，拡張された言語の特徴の使用による設計と，元々の言語による設計のどちらが優れているのかの判断はその状況に深く依存する．たとえば，通常の Java において Visitor パターンが適切な場合を考えてみる．AspectJ[3]は，Java のためのアスペクト指向の拡張であり，AspectJ を用いた Visitor パターンの実装方法が提案されている[4]．しかし，通常の Java による Visitor パターンが適切なのか，AspectJ による Visitor パターンが適切なのかの判断は具体的な状況が分かっていなければ難しい．同様のことが，他のパターン，たとえば Observer パターンにも言える．ソフトウェア設計では，基本的には，その状況において最も適切だと思われる設計上の選択肢を選ぶことが重要である．つまり，設計上のある選択肢（OO Visitor と AO Visitor）を比較する場合には，適切な状況（シナリオ）設定が重要である．デザインパターンやリファクタリングは，そのようなシナリオの記述を目的としていないため，新しい記述方法が必要である．

三つ目の動機としては，どのようにしてソフトウェアが進化していくのかの理解にある．ソフトウェア進化の理解に関連する研究はすでにいくつかあるが，それらは，ソフトウェア設計（コード）における進化には重点を置いていないように見える．ソフトウェアパターンの文献の多くは，開発者にとって役に立つことに重点を置いている．

†大阪産業大学
Osaka Sangyo University

したがって，ソフトウェア設計の観点からソフトウェアの進化を理解することは重要である．

3. パターンテンプレート

前述のように，進化パターンは，デザインパターンやリファクタリングとは異なる視点のため，新しいパターンテンプレートが必要である．

- パターン名：パターン名を表す．
- 文脈：コードが進化する状況を表す．
- 進化前のコード：変更前のコードとクラス図．
- 進化後のコード：変更後のコードとクラス図．
- 具体的な例：具体的な例を示すコード．
- 関連パターン：関連するリファクタリングやデザインパターン，進化パターン，その他のパターン．
- 進化パス：どのような進化を通ってこのパターンにたどり着くのか．あるいは，このパターンからどのような進化が考えられるのか．

4. 進化パターンの例

進化パターンの具体的な例としてライブラリ使用によるコードの進化の例を紹介する（その他の例については[2]を参照）．なお，スペースの都合のため，進化前と後のコードのみを述べる．

4.1. ArrayUtils[5]によるプリミティブ配列への変換

以下は Double[]をプリミティブ型の配列 double[]に変換するコードである．

```
double[] array = new double[values.length];
for(int i = 0 ; i < array.length; i++) {
   array[i] = values[i].doubleValue();
}
```

もし，プログラマが Commons Lang ライブラリ[5]の ArrayUtils の存在を知ったとしたら，上記のコードは，以下のように変更される．

```
import org.apache.commons.lang.ArrayUtils;
double[] array = ArrayUtils.toPrimitive(values);
```

4.2. ユーティリティクラス導入による外部ライブラリの隠蔽

```
import org.apache.commons.lang.ArrayUtils;
class Util {
   public static double[] toPrimitive(Double[] array)
      return ArrayUtils.toPrimitive (array);
 }}
```

また，そのような変換の必要な場面が多くあるとしたら，特定のライブラリへの依存を小さくするために，上記のようなユーティリティクラスを導入することが考えられる．

4.3. 議論

例からは，通常のリファクタリングと異なり，ライブラリを用いることによるコードの変化を表していることが分かる．また，従来のデザインパターンよりも，小粒であり，よりコードの変化に重点を置いていることが分かる．したがって、従来のデザインパターンやリファクタリングよりも広い範囲で開発者の経験を記述することができる．また，各進化パターンの粒度は小さいため，将来，新たなライブラリや言語拡張が現れた時でも，コードの進化の道筋を導入することは容易であると思われる．

進化パターンは，必ずしもデザインパターンに関連している必要はなく，またリファクタリングである必要はない．コードが変更された前後の状況自体が繰り返し起こるようなパターンであればよい．

5. まとめ

開発者の経験の共有は重要である．本稿では，その目的のために，ソフトウェア設計におけるコードの進化（発展）あるいは変化自体をパターンとして捉え，文章化することを提案した．

進化パターンは，デザインパターンやリファクタリングやよりも広い範囲で開発者の経験を記述することを目的としている．進化パターンの収集は，ソフトウェア設計におけるシナリオの作成に役立つと期待できる．また，十分な数の進化パターンが集まれば，ソフトウェア設計の観点から，ソフトウェア進化の理解に役立つと思われる．現在は，収集されたパターンは少ないため，今後はパターンの収集を行っていく必要がある．

謝辞

原田英明氏，徳光政弘氏，久保淳人氏には草稿を読んでもらい貴重なコメントをいただいた．感謝します．

参考文献

[1] Joshua Kerievsky, Refactoring to Patterns, Addison-Wesley, 2004
[2] http://noselab.ise.osaka-sandai.ac.jp/~asato/doc/evolution.html, 2004
[3] AspectJ, http://eclipse.org/aspectj/, 2004
[4] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. OOPSLA 2002
[5] http://jakarta.apache.org/commons/lang/, 2004