デザインパターンを活用する

2002 年 10 月
中谷　多哉子

１．はじめに

パターンは，問題解決の知見を文書化したものです．
パターンは，オブジェクト指向の様々な開発現場で使われるようになってきました．ライブラリクラスのように，そのままの形でプログラムの中に取り込むことはできませんが，開発に適用することはできます．「適用」とは，

・実際の開発現場で直面している具体的な問題に対して，パターンが解決している問題とを対応付け，
・パターンの解決手段を参考にしながら，
・実際の問題を解決する方法を見つけることを指します．

本稿では，パターンの中でも特に設計で使われるデザインパターンに着目して，その利点を，具体的なプログラム例とともに紹介します．また，デザインパターンを使いこなすために必要な留意点も示していきます．

２．パターンとは

2.1 パターンが重視される理由

オブジェクト指向は，もともと再利用性を向上させるための技術として注目され，導入されてきました．オブジェクト指向で再利用される対象には，クラス，フレームワーク，そしてコンポーネントなど，様々な形態の部品があります．しかし，残念ながら，再利用は期待したほどの効果をあげていません．再利用を難しくする原因は，部品が見つからないだけでなく，再利用可能な部品を入手した後，どの部分にその部品が使えるかが分かり辛い点にもあります．というのは，部品の再利用では，事前に次のことを調査しなければならないからだと考えられます．

・部品が提供している機能，性能，信頼性など．
・その部品の使い方．同時に利用すべき他の部品，メッセージのインタフェースなど．
・部品の実装方針．
・部品の拡張方法．
・使用に際して新たに考慮すべき事項．例外処理への対応など．
・部品の利用実績．信頼性の保証．

言語が提供しているライブラリクラスを再利用するときにも，開発者がライブラリのソースコードを読み下さなければならない場合があります．このとき，開発者は，そのクラスが提供する機能と自分が必要な機能とを比較し，さらに，そのクラスを利用することによって新たに開発すべき事項と，再利用によって生じる新た

な問題を討して，再利用するか否かを決定します．このような作業で得られる知見は，やがて開発者に修得され，いずれは，簡単に再利用部品を選択し，開発を進めることができるようになります．

このような知見を持った技術者を「熟練者」などと表現しますが，熟練者と初心者との違いは何なのでしょうか．

再利用が思ったほどの効果を上げていないという教訓から，実際の開発現場では，部品の再利用よりも知見を使いこなす技術の方が重要であると考えられるようになりました．これが，現在のパターンに関する議論に繋がっています．

パターンとは，開発時に発生する様々な問題とその解法を文書化し，開発者同士で共有できるようにしたものです．一般的な再利用部品はプログラムですが，パターンは，分析や設計，プログラミング，テスト，開発管理，その他の様々な業務の知見を集めた文書です．

2.2 C．Alexander のパタンランゲージ

パターンの歴史は，1970 年代後半に，C．Alexander が街や建物を造るときの 253 のパターンをパタンランゲージとしてまとめたことから始まります．例として，C．Alexander の横断歩道パターンを紹介しましょう．

> 例：横断歩道（[1]より抜粋）
>
> 歩行路が道路を横断する地点では，たとえ歩行者に法的優先権があろうとも，車には歩行者を脅迫し，服従させる力がある．これは歩行路と道路が同一平面上にあるとつねに生じる問題である．（中略）横断者が道路を快適で安全と感じるのは，横断歩道が障害物になり，車がスピードを落とし，歩行者に道を譲ることを物理的に保証される場合のみである．（中略）横断歩行路を道路より 15-30cm 高くし，道路がそこに向かって盛り上がるようにすればよい．勾配が 1/6 以下であれば車には安全だし，しかも確実に車はスピードダウンする．遠方から横断を見やすくし，またそこでの歩行者の権利に重み付けをする意味でも，道路際に縁を設けて歩行路を明示することもできよう．（中略）問題の道路に，1 日数回，異なった時間に出かけてみること．その度に，何秒待てば横断できるか測ること．待ち時間の平均が 2 秒以上であれば，このパターンを適用する方がよい．（後略）

日本の横断歩道は白線が描かれているだけですが，スイスには，横断歩道パターンに準拠して作られたような横断歩道があります．パターンとして表現されている知見は，多岐に渡り，また特殊化されています．パターンには，解決する問題を限定し，さらに解決手段を選択する幅も狭め，特定の状況のもとで，さらに特定の制約のもとで選択される解決手段を定義してあります．この点は，開発手法や方法論のように問題解決手段を一般化したものと異なる点です．ですから，手法や方法論のように，それを使うことが半ば強制されるものではなく，個々人がパターンを適用できる状況か否かを，自分で判断する必要があります．

2.3 パターン適用におけるフォースとは

図１に，私たちがよく直面する問題状況とその解決手段の関係を表してみました．図１に示すように，ひとつの問題状況には，それを解決する手段は無数に考えることができます．たとえば，東京から大阪へ移動しなければならないという問題状況では，その移動手段として，いくつかの候補を考えることができます．

パターンでは，知見を「あの問題を解決するときに，特定の状況のもとでは，その解決策のみが選択される

べきである」という知識と捉えています．解決しなければならない問題に対して，選択すべき解決策を限定させるような，特定の状況の制約は，「フォース」と呼ばれています．たとえば，東京から大阪へ実際に移動するときには，図 1 に示した移動手段の中から，いずれかひとつを解決策として選択することになりますが，「飛行機」を選択したとしたら，そこには，その選択肢が選ばれた根拠があるはずです．この根拠がフォースに相当します．ですから，フォースは，解決策を選択するために与えられる制約と言うこともできます．

図 1 フォースの役割

横断歩道パターンでは，問題状況，解決策，フォースは，以下のように定義されていると考えられます．

・問題：横断歩道を造るとき，歩行者に安心感を与え，安全を守らねばならない．
・解決策：横断歩行路を道路より 15-30cm 高くし，勾配を付けて車の走行に配慮する．
・フォース：２秒以上待たないと横断できないような道路．

３．パターンの歴史

C. Alexander のパターンランゲージは，街と建物を作るためのパターンを集めたものですが，この形式は，ソフトウェア開発の知見の文書化にも使える要素を網羅していると言えます．1991 年から，パターンに関するワークショップが OOPSLA（Conference on Object Oriented Programming Systems Languages and Applications.オブジェクト指向に関する国際会議．毎年数千名の参加者があります．）で始まりました．1995 年に発行された E．Gamma らのデザインパターンは，この活動の成果の一つです．システム開発のパターンには，そのほかにも，データパターン，アナリシスパターン，アンチパターンなど様々なものがあり，総称してソフトウェアパターンと呼ばれて実用されています．

パターンの研究は，現在，大きく二つの方向に分かれています．一つ目の方向は，パターンを文書化するための形式として，パターンテンプレートを検討する活動です．そして，もう一つの方向は，パターンそのものを収集して体系化し，特定の分野向けのパターン集をパタンランゲージとして定義する活動です．現在では，パターンを検討するワークショップが世界中で開催されています．そこで提案された J．コプリンによるパターンテンプレートを紹介しましょう．

- **パターン名(Pattern name)**：
- **問題状況(Problem)**：パターンが解決する問題の記述．
- **問題背景(Context)**：パターンを使うことが想定されている利用者を限定する背景の説明．これで，問題領域を特定する．
- **フォース(Force)**：問題の状況のもとで，このパターンの解決策が有効となる状況や制約．問題解決策を選択するために，さらに問題領域を限定する．
- **解決策(Solution)**：
- **問題解決後の状況(Resulting Context)**：パターンを適用して問題を解決した後に，新たに発生する可能性のある問題やそれを解決するパターンの紹介．
- **根拠(Rationale)**：解決策が合理的であるという著者の経験に基づく根拠や主張．
- **事例(Known Uses)**：このパターンが使われた事例の紹介．
- **関連するパターン(Related Patterns)**：このパターンに関連する他のパターンの紹介．
- **スケッチ(Sketch)**：設計や分析のパターンであれば，ここにクラス図を提示する．
- その他：**パターンの別名**，**著者**，**日付**，**参考文献**，**キーワード**，**サンプルソースコード**など

設計のパターンには，上記の他に，自然言語による解決策の解説と，UML(Unified Modeling Language)によるクラス図とプログラムソースコードの例が添えられます．

名前はパターンを指示するために必要です．問題状況，問題の背景，フォースが，このパターンの解決策が使われる状況の定義となります．解決策が重要なのは言うまでもありません．さらに，適用後に得られる良い成果と新たな問題とを知ることも重要です．これらの項目が，そのパターンを使うか使わないかを判断する情報となります．

L．Rising が最近提案しているパターンテンプレートでは，問題の説明の前に典型的な事例（Known Uses）が，C．Alexander の街や家の写真の代わりとして使われています．

４．GoF のデザインパターンの使い方

E．Gamma らのデザインパターンは，これからオブジェクト指向設計を始める技術者が最低限身につけていなければならない知見でしょう．E．Gamma らが出版したデザインパターンに掲載されている 23 種類のパターンは，著者の４人を指して GoF（Gang of Four の略:ゴフと読む）のデザインパターンとも呼ばれています．

本稿では，その中から，State パターンを例として取り上げて解説します．

4.1 State パターンとは

State パターンは，振る舞いに対応づけた個々の状態を独立したクラスにすることによって，状態変化に依存する振る舞いを，条件分岐を用いずに切り替えることができるようにした設計のパターンです．これを

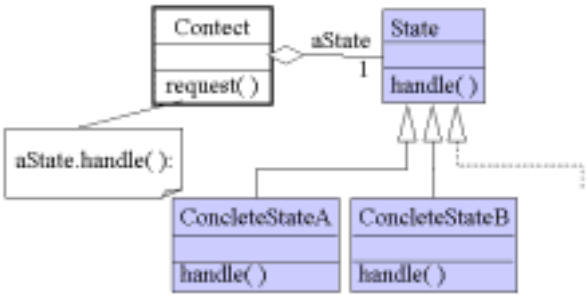

図 2　State パターンが提供する問題解決のためのクラス図

このパターンを用いることによって,状態数の増加に依存してプログラムの条件分岐が多様化することを防ぐことができます．状態数が増えた場合，図 2 の State のサブクラスを増やします．この作業は，他のクラスに全く影響を与えません．また，State パターンには，オブジェクト指向の継承や，図 2 の handle()にみられるようなメッセージの多相性，状態を Context クラスの中にカプセル化するといった，オブジェクト指向の特徴的な概念が活用されている点も注目に値します．さらに，状態という，現実世界に存在しているとは考えにくいものをオブジェクトとしている点にも面白さがあります．これらの点だけでも，オブジェクト指向設計の技術者が学ぶ価値のあるパターンと言えます．

もちろん，State パターンは多くのプログラムで使えるパターンです．

4.2 問題を含んだ設計の例

実際にパターンを適用する前と，適用後とで，ソースコードがどのように変わるのでしょうか．

図 3 に，State パターンを適用する前のクラスを UML のクラス図で示しました．

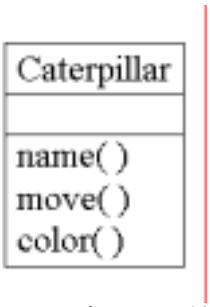

図 3　State パターンを適用する前のクラスの設計

それでは，このクラス図が何を表しているかを見ていきましょう．

図 3 は，パターンを適用する前のクラス図です．パターン適用前のクラス Caterpillar は，芋虫のクラスを表しています．このクラスのインスタンスは，name(),move(),color()という 3 つのメッセージを受けることができます．また，この芋虫は，成長するにつれ，蛹(pupal)，蝶(butterfly)へと状態を変え，それぞれの状態によって name()，color()，move()というメッセージに対する応答は，次のように変わります．

・芋虫のときは，これらの問いかけに対して，「CATERPILLAR」「GREEN」「I CAN WALK.」と応答します．
・蛹になると，「PUPAL」「BROWN」「I DON’T MOVE.」と応答します．

・蝶になると，「BUTTERFLY」「YELLOW」「I CAN FLY.」と応答します．

図 3　State パターンを適用する前のクラスの設計に定義されているクラス Caterpillar では，芋虫の状態や，状態によって振る舞いが変わることをクラス図から知ることはできません．これらの性質は，3 つのメソッドの中に実装されていますが，情報隠蔽されているのです．では，State パターンの適用が必要なプログラムを見てみましょう．これを図 4 に示しました．

```
class StateExample1{
   public static void main (String args[]) {
      Caterpillar myCaterpillar = new Caterpillar();
      for (int i=1; i<7; i++){
         myCaterpillar.name( );
         myCaterpillar.color( );
         myCaterpillar.move( );
         myCaterpillar.transformState();
      }
   }
}

class Caterpillar {
   protected int aState;
   public Caterpillar(){
       init();
   }
   public void init(){
       aState = 0;
   }
   public void transformState(){
      aState = (aState+1) % 3;
   }
   public void name( ){
      if (aState == 0){
        System.out.println("CATERPILLAR");
      } else if (aState == 1){
        System.out.println("PUPAL");
      } else {
        System.out.println("BUTTERFLY");
      }
   }
   public void color( ){
      if (aState == 0){
        System.out.println("GREEN");
      } else if (aState == 1){
        System.out.println("BROWN");
      } else {
        System.out.println("YELLOW");
      }
   }
    public void move( ){
```

```
      if (aState == 0){
        System.out.println("I CAN WALK.");
      } else if (aState == 1){
        System.out.println("I DON'T MOVE.");
      } else {
        System.out.println("I CAN FLY.");
      }
   }
}
```

図 4 パターン適用前のソースコード

図 4 のプログラムには，個々のメソッド内に同じ構造の条件分岐を発生させる if 文が使われています．この部分を赤い文字で表しました．このプログラムでは，たとえば，芋虫に「蛹になりかけの状態」，「羽化中」という状態を新たに追加したときには，赤い文字で表した全ての if 文に同じ条件分岐を新たに追加し，新しい振る舞いを定義しなければなりません．一般のプログラムでも，if 文はバグの温床になりやすいと言われていますが，この Caterpillar クラスは，保守の度に if 文の部分にプログラマの手が入る可能性があります．このようなプログラムは，好ましくありません．

4.3 問題を解決した設計の例

それでは，この問題を解決してみましょう．

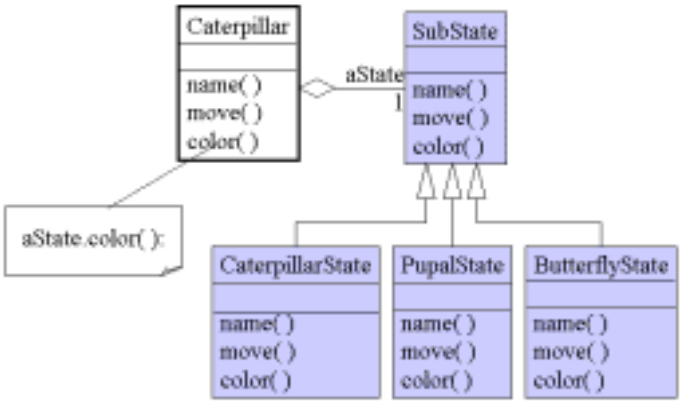

図 5　問題を解決した設計を表すクラス図

4.2 で紹介した問題は，状態という振る舞いが変化する軸があるのに，メッセージ名という変化しない軸でメソッドを分けている点に起因しています．オブジェクト指向では，同じメッセージ名でもその実装をクラスによって変えて，そのメッセージ名に多相性を持たせる設計が行えます．この問題を解決するために，この機構を用います．

まず，状態という変化の軸を用いて，芋虫の状態，蛹の状態，蝶の状態という，それぞれの状態に対応したクラスを定義します．更に，それらのクラスには，それぞれの状態のときに共通に受け取るメッセージ name()，color()，move()のメソッドを定義します．これで，状態という変化する軸をクラスに分け，変化しないメッセージ名に対して多相性という機構で対応したことになります．

この設計を用いた結果が，図 5 のクラス図です．図 3 のクラス図に比べて，クラス数が増えています．しかし，これが本質的にシステムを複雑にすることにはありません．なぜならば，Caterpillar クラスのメソッ

ドを実装する手段として，新しいクラスが追加されており，Caterpillar クラスにカプセル化されていると考えることができるからです．

このクラス図は State パターンを適用した後のクラスでもあります．図 2 の State パターンを表すクラス図と比較し，どこが同じでどこが異なるかを比較してみてください．クラス構造は同じですが，クラスの名前やメッセージの名前は異なっています．パターンを実際の問題に適用する際には，このようにパターンを実際の問題に合うように適合させる作業が必要になります．この点が，通常のライブラリクラスの使い方と異なる点です．

図 3 や図 4 のパターン適用前のクラス Caterpillar からの変更点は以下のとおりです．

・状態に依存して変わる振る舞いを定義するために，3 つのメッセージを持つインタフェース SubState を定義しました．

・このインタフェースを 3 つのクラス CaterpillarState，PupalState，ButterflyState で実装しています．それぞれのクラスに定義されている name()，color()，move()のメソッドには，芋虫，蛹，蝶という各状態のときに芋虫の振る舞いだけを定義します．

・プログラムの実行中，芋虫の状態が変わるごとに，Caterpillar の sState へのリンクがそれぞれの状態を表すクラスのインスタンスに張り替えられます．

・Caterpillar が name(),move(),color()のいずれかのメッセージを受け取ったときは，sState のリンク先のオブジェクトに同じメッセージを委譲するように，各メソッドの内容を

```
sState.name();
sState.move();
sState.color();
```

と変えました．

以上の作業で，State パターンの適用は完了です．パターン適用前の Caterpillar クラスが if 文を用いて状態に依存した振る舞いを切り替えていたのに対し，適用後の Caterpillar は，状態が変わった時点で aState のリンクを張り替え，メッセージを受け取った時に適切に振る舞える準備をします．そして，メッセージを受け取ったら，そのメッセージを aState のリンク先の状態オブジェクトに委譲するのです．

図 4 のプログラムに State パターンを適用して変更したプログラムを図 6 に示します．

```
class StateExample2{
   public static void main (String args[]) {
      Caterpillar myCaterpillar = new Caterpillar();
      for (int i=1; i<7; i++){
         myCaterpillar.name( );
         myCaterpillar.color( );
         myCaterpillar.move( );
         myCaterpillar.transformState();
      }
   }
}

class Caterpillar {
   protected SubState aState;
   public Caterpillar(){
       init();
   }
   public void init(){
       aState = new CaterpillarState( );
   }
```

```
   public void transformState(){
      aState = aState.transformState( );
   }
   public void name(){
      aState.name( );
   }
   public void color(){
      aState.color( );
   }
   public void move(){
      aState.move( );
   }
}
interface SubState {
     public void name();
     public void color();
     public void move();
     public SubState transformState();
}
class CaterpillarState implements SubState {
  public void name( ){
        System.out.println("CATERPILLAR");
  }
  public void color( ){
        System.out.println("GREEN");
  }
  public void move( ){
        System.out.println("I CAN WALK.");
  }
  public SubState transformState(){
        return(new PupalState());
  }
}
class PupalState implements SubState {
  public void name( ){
        System.out.println("PUPAL");
  }
  public void color( ){
        System.out.println("BROWN");
  }
  public void move( ){
        System.out.println("I DON'T MOVE.");
  }
  public SubState transformState(){
        return(new ButterflyState());
  }
}
class ButterflyState implements SubState {
  public void name( ){
        System.out.println("BUTTERFLY");
```

```
  }
  public void color( ){
        System.out.println("YELLOW");
  }
  public void move( ){
        System.out.println("I CAN FLY.");
  }
  public SubState transformState(){
         return(new CaterpillarState());
  }
}
```

図 6　State パターン適用後のプログラム

このプログラムでは，図 4 のプログラムでバグの温床となると予想された if 文が使われていないところに注目してください．ただし，Caterpillar の状態は，芋虫，蛹，蝶を繰り返すようにプログラムされています．

オブジェクト指向では，プログラムの実行中にオブジェクトがメッセージを受け取った時に，起動するメソッドを決定するメソッドの動的束縛という機構があります．State パターンはこの機構を使った設計となっています．たとえば，name()というメッセージを受け取った芋虫オブジェクトが，実際にのどのメソッドを起動するかは，実行時に芋虫オブジェクトがリンクを結んでいる状態オブジェクトに依存して決定されるのです．SubState はインタフェースですから，そこに定義されている name()，color()，move()は，多相性を持つメッセージになっています．

芋虫に「蛹になりかけの状態」，「羽化中」という状態を新たに追加する場合は，新しい状態に対応するクラスを二つ作り，インタフェース SubState の実装クラスとして定義するだけです．このような追加は，他の部分に全く変更を及ぼしません．したがって，図 6 のプログラムは，図 4 が抱えていた問題を解決していると言えるでしょう．

4.4 State パターンを活用するために必要な技術

State パターンを適用する目的は，「**オブジェクトの内部状態が変化したときに，オブジェクトが振る舞いを変えるようにする．クラス内では，振る舞いの変化を記述せず，状態を表すオブジェクトを導入することでこれを実現する**」ことです．State パターンの著者は，「クラス内で振る舞いの変化を記述する」ことは問題だと考えたのです．パターンを適用するには，このようなことが問題であることに気づく技術が，開発者には必要です．問題であると気づかなければ，パターンを適用することを思いつくこともないでしょう．図 4 に示したプログラムのように，パターンを適用しなくてもプログラムは動作するのです．このプログラムを図 6 のように変更するという作業を始めるには，「状態の数が増えたとき，これ以上プログラムの複雑度を上げてはいけない」という意識が必要です．

ここから，パターン活用の秘訣も見えてきたのではないでしょうか．

5 パターン活用の秘訣

パターンは，プログラム部品のように，そのまま使うことはできません．知見，すなわちパターンを活用するには，パターンを使おうという人が，自分の問題を発見し，その状況と，解決策を見つけるためのフォースを把握しなければなりません．さらに，パターンの適用という，自分の問題状況とパターンの問題状況の対応付けが必要となります．

先の横断歩道パターンでも，実際に横断歩道を設ける場所の選定や資材の調達は，パターンを適用する人が判断して行わなければならないのと同じです．

それでは，実際に開発現場でパターンを活用するための秘訣を，GoF のデザインパターンに掲載されている

パターンの概要から探ってみましょう.
AbstractFactory パターンというパターンがあります. このパターンを適用する目的は, 以下のとおりです.

「互いに関連したり依存し合うオブジェクト群を, その具象クラスを明確にせずに生成するためのインタフェースを提供する.」

確かに, 互いに関連するオブジェクト群の具象クラスの名前をそのままプログラム中に明示して, オブジェクトを生成するメソッドを書くことは問題ですね. 異なる組み合わせが発生する度にプログラムを修正しなければならないからです. **AbstractFactory パターン**は, 特に, 複数の環境のもとで複数のクラスのインスタンスを取り扱うという, 二重の変化の軸を持つ場合に有効なパターンです.

たとえば, OS ごとに異なる Window クラスと ScrollBar クラスのインスタンスを組み合わせてウィンドウを作る場合です. もし, このパターンを適用しないとしたら, プログラムは, OS を切り替える if 文と, それぞれの場合ごとに, 異なるクラスのインスタンスを生成して組み立てる処理を定義することになるでしょう. 新しい OS の追加が繰り返されたとき, プログラム中のインスタンス生成部分は, 巨大化していくに違いありません.

このように全てのパターンには, 解決すべき問題が示されています. 逆に考えると, ソフトウェア開発の初学者は, 開発者が問題と感じなければならない事項を, パターンから学ぶことができます. これは, パターンから問題解決手法を学ぶよりも重要なことです. 基本的な問題を学ぶことによって, 新たに直面するかも知れない問題を発見する勘を育てて貰いたいものです.

5 まとめ：パターン活用を始める技術者へのメッセージ

開発プロジェクトで問題が発生し, それを解決する都度, パターンを自分で定義してみることを勧めます. この作業を繰り返すと, 同じような問題にたびたび遭遇していることに気づくでしょう. そうやって, 特定の問題を解決した事例が集まってきたら, 仲間同士でその経験を持ち寄り, 共有してみましょう. 経験を仲間に説明するときには, パターンテンプレートに基づいて整理してみるとよいでしょう.

そうやって知見を獲得し, 実際に適用して開発を行ったときには, 後進の技術者のために, なぜそのパターンを適用したのか, 何が問題だったのか, パターン適用によってどのような結果を得ることができたのかを仕様書に明記しておきましょう. そのようなトレーニングを継続的に行うことこそがパターン活用の, そして, 開発エキスパートへの近道です.

参考文献

[1]C. Alexander: A Pattern Language, Oxford University Press, 1977.（平田翰那訳, パタン・ランゲージ, 鹿島出版会, 1984.
[2]D. Hay: Data Model Patterns, Dorset House, 1996.
[3]E. Gamma, R. Helm, R. Johnson and J. Vlissides: Desgin Patterns, Addison-Wesley, 1995.(本位田真一, 吉田和樹監訳：デザインパターン改訂版, ソフトバンク, 1999.)
[4]J. Coplien: Software Patterns,
http://www1.bell-labs.com/user/cope/Patterns/WhitePaper/SoftwarePatterns.pdf
[5]M. Fowler: Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997.(堀内一監訳, アナリシスパターン, アジソンウェスレイ, 1998.)
[6]細谷竜一, 中山裕子監訳, プログラムデザインのためのパターン言語（Pattern Languages of Programming Design 選集）, ソフトバンク, 2001.
[7]中谷多哉子, 青山幹雄, 佐藤啓太編著, ソフトウェアパターン, 共立出版, 1999.
[8]W. Brown, R. Malveau, H. McCormick and T. Mowbray: Anti Patterns, John Wiley & Sons, 1998.(岩谷宏訳, アンチパターン, ソフトバンク, 1999.)