ソフトウェア工学 112－6
（1996．11．15）

デザインパターンの適用自動化手法

直田繁樹　　堀田勇次
（株）富士通研究所 マルチメディア研究所
{suguta,hotta}@flab.fujitsu.co.jp

我々はデザインパターンの適用を省力化するために，選択されたパターンに対応するC++プログラムスケルトンを生成する手法を開発した．これは従来開発してきたC++クラスに対するイディオム生成機能を拡張したものである．本論文では，従来方式での問題点とそれを解決した新しい生成方式を説明する．ここでは，複数クラスを対象とする生成パターン記述の導入，生成処理への意味情報を与える手段の一般化等を行なった．それらによる自動適用の実例を示す．また，Gammaらのカタログ中のパターンを，我々のアプローチによる自動化という観点から分類したので，その結果についても述べる．

A Program Generation Approach for Applying Design Patterns

Shigeki Suguta　　Yuuji Hotta
Fujitsu Laboratories Ltd., Multimedia Systems Laboratories

To support applying design patterns, we have developed a new program generation system which produces C++ program skeltons based on design patterns. This is an enhancement of the C++ idiom generator which we developed before. In this article, we first describe the problems of the old generator to automate the application of design patterns. We introduced the new rule description which can process multiple classes and introduced generalized directions which give semantic information to the generator. Then we show an example of design pattern automation. We also discuss about the classification of Gamma's design patterns from the view point of our program generation approach.

1　はじめに

オブジェクト指向開発におけるデザインパターンの考え方は，ソフトウェアの再利用性を向上させる有用なアプローチとして近年大きく注目されている．Gammaらのデザインパターンカタログ [1] を始め，多くの設計ノウハウがパターンという形で蓄積されてきており，ソフトウェア再利用に新しい局面を開くものと期待される．

しかしながら，これらのデザインパターンを実際のプログラムに適用するのは，必ずしも容易ではない．我々の行なった簡単なプログラムでの実験では，一つのパターンあたり10数%のコード行数の増加が見られた (C++で基本処理部800行に対して1パターンで100～150行程度増加，4パターン適用)．増加した部分は，比較的定型的な処理であり，手作業で適用するのは煩わしいものであった．こうしたことから，我々はパターンの適用をサポートするツールを開発

することが必要であると考えた.

一方, 我々は, 分散環境での受動的オブジェクトについて, その一貫性管理に問題が生じることに着目して, [2] において, 一つのクラス定義から派生的な定義を自動生成することで, この問題を解決する手法を提案してきた. また, この手法が, C++のイディオム [3] の生成にも適用でき, C++ でのプログラミングの問題を軽減するのに役立つことも示した.

今回, この手法を, Gammaらのデザインパターンなど, 複数のクラスを用いる定型的な処理を自動生成できるように拡張したので, その結果を報告する.

まず, 従来の生成システムについて簡単に説明し, それをデザインパターンの自動化に用いる場合の問題点について述べる. 次に, その問題を解決するために導入した拡張について解説し, 全体のシステム構成を示す. 次に, 新しい方式での自動適用の例を示す. また, この適用実験の結果から, 自動化の観点でGammaらのデザインパターンを分類したので, それについても説明する. 最後に, 関連研究との比較を行なった上で, 今後の課題について述べる.

2 従来の生成システムの概要

2.1 生成システムの機能

我々が [2] において示した手法では, コメントにより意味情報を補ったC++のクラス定義を入力として, それを構文解析した結果に, 生成ルールを適用して, 種々の定義を生成するものであった.

この手法により, 以下のような定義を生成できることを確認している.

- 分散システムでの通信インタフェース
- DB スキーマとアクセス関数
- C++特有の補助関数 (イディオム)

これらの内容は, ルールを用いて記述され, ユーザによるカスタマイズが容易である. このことは, 種々のミドルウェアを利用しなければならない複合的な分散環境を考える場合, 重要な機能であった.

生成ルールの記述形式 生成ルールは, 関数生成ルールとメンバ操作ルールからなる. その記述中では, 入力クラスを構文解析した結果得られるクラスやデータメンバの構造情報が利用できる.

関数生成ルールは, さらに宣言生成ルールと定義生成ルールに分かれ, それぞれ, クラス内の関数宣言とクラスの外に記述する関数定義を生成するのに用いる. それらは, クラスの構造の情報などを使って, 関数の先頭部分を生成したのち, メンバ操作ルールを呼び出し, 個々のメンバ毎の操作を生成する.

以下に, 代入演算子関数の場合の関数定義生成ルールを示す.

```
!!DEF_FUNCDEF_RULE  assign_op_gen
  !!BEGIN
     !!PRINT_BEGIN
const $class &
$class ::operator=(const $class & src) {
   // 自分自身への代入になっていないか調べる
    if ( &src != this ){
     !!PRINT_END
     # メンバ操作ルール assign の呼出し
     !!EXEC_MEM_RULE(assign)
     !!PRINT_BEGIN
       }   return *this ;  }
     !!PRINT_END
  !!END
```

ここで, $で始まるシンボルはルール中の変数である. 入力クラスを解析した結果から得られる情報で, クラス名 ($class) やデータメンバ名 ($mem) とその型 ($mem_type) などはあらかじめ設定されている.

また, !!で始まる大文字の語は, ルール記述のキーワードである.

!!PRINT_BEGIN と !!PRINT_END の間の記述は, 変数置換を受けた上で出力される.

!!EXEC_MEM_RULE() は, 引数で与えられたメンバ操作ルールを全てのデータメンバに適用するよう呼び出すものである.

他のキーワードは通常のプログラミング言語とほぼ同様の意味をもつ.

また, ルール中のコメントは, #以降, 行の終りまでであり, 生成されるC++中のコメントは //以降である.

一方, メンバ操作ルールは, 個々のデータメンバを入力として受けとり, その型毎にどのように変換処理を行なうかを定義する.

代入演算子などの補助関数を定義する場合には, データメンバがポインタ型がある場合, ポインタの意味が has-a 関係にあるかどうかで変換処理を変えなければならない. しかし, そのような情報は自動検出は不可能であるので, ここでは, 以下のように, //!! で始まる特殊なコメント (生成指示コメント) をメンバ記述に付加することにした. これは構文解析時に検出され, 対応するメンバの属性情報として生成ルールで利用できるようになる.

クラス定義と生成指示コメントの例

```
class Part {
 int   code ;
 char  * name ;  //!! has-a → has-a関係の指定
 Supplier * supplier;
};
```

以下に, 代入演算子関数の場合のメンバ操作ルールを示す. ここでは, データ型毎に判定を行ない, ポインタの場合には, has-a 関係であれば, 代入以前の値の削除とコピーの処理を生成している.

```
!!DEF_MEM_RULE assign
 !!BEGIN
   # char * の場合
   !!ELSIF ( !!is_char_ptr($mem_type) )
     !!BEGIN
         !!IF ( !!has_a($mem) )
           !!BEGIN
              !!PRINT_BEGIN
     delete [] $mem;   // 以前のデータの削除
     if (src.$mem == (char *)0) {
         $mem= (char *)0;
      }
     else { // データがある場合
       $mem= new char[strlen(src.$mem)+1];
       strcpy($mem, src.$mem);
      }
              !!PRINT_END
           !!END
       !!ELSE
           !!BEGIN
             !!PRINT_BEGIN
     $mem = src.$mem;
             !!PRINT_END
           !!END
       !!END
   # int など固定長のデータ型の場合
   !!ELSIF ( !!is_fixed_size($mem_type) )
      ...  以下略
```

ここで, !!で始まり, () で引数を取る名前は, 組込関数の呼出しである. メンバの型判定やコメントから得られる has-a 関係などを判定できる機能を提供している.

2.2 従来方式の問題点

このシステムをデザインパターンの自動化に適用しようとした場合は, 以下に述べるような問題があった.

複数クラスの処理 まず, 一つのクラスのみを入力として変換する方式であることが問題である. Gamma らのパターンでは, 一つのクラスのみを扱っているのは, Singleton パターンだけであり, 他は全て複数のクラスの相互作用を定義したものである. このため, 従来の生成ルールの形式をそのまま適用することはできなかった.

しかし, 入力クラスの解析結果を利用して種々の変換操作を定義するという基本的な考え方は有効であるので, ルールの構成を変更することで対処する.

出力先ファイルの指定 単一クラスを扱う場合は, 出力先ファイルを特に考慮する必要はなかったが, デザインパターンの適用では, 複数の入力クラスを加工し, 入力以外のクラスも新しく生成する. これらをどのようにファイルに分割していくかを指定する必要がある. 特に C++においては, クラス及びファイルの構成によるコンパイル時の依存性は大きな問題であるため, 対応が必要である.

生成指示コメントの特殊性 従来の生成指示コメントでは, あらかじめ組込の機能として, has-a,unique-key などのデータメンバの属性を決めておき, それをコメントで指定するというものであった. これでは, 一般性がなく, パターン毎に異なる意味情報を与えるのには不十分であった.

3 パターン自動化のための拡張

3.1 自動化に必要な情報

複数クラス間の関係を自動生成するに当たっては, それらの入出力の関係を明確にしなければならない. 一般には, デザインパターンの記述テンプレートにおいて, 構成要素 (Participants) に現れるクラスが, 生成処理の入力と考えられる. しかし, その中には, 本来の処理を実現するクラスでパターン向けに加工されているものとパターンを構成するために導入されたクラスとがある. また, 構造 (Structure) の中で, それらのクラスのデータメンバやメンバ関数として現れるものについても, 同様のことが言える.

そこで, 我々は, 本来の処理を実現する要素については, あらかじめクラス定義しておき, それに生成指示コメントでその役割を指定したものを入力とし, パターンとして必要な加工を行なう, という方式を取ることにした. 一方, パターン独自の部分は, 名前のみを入力パラメタとしておき, 実体は自動生成させるようにした.

以上のことから, パターンの適用自動化のために記述する情報を次のように整理した.

- 生成パターン定義
  従来のルール定義を拡張したものに構成要素の

宣言, ルールの出力先定義などを付け加えてたものであり, 生成内容を定義する.

- 入力クラス
  入力になるC++クラスの定義. パターン用に加工される. このため, 必要に応じて生成指示コメントを与えておく.
- ユーザ生成指示
  個別の生成に必要な情報を指定する. 生成パターンの選択, パターン中のユーザ入力要素の値の指定などを行なう.
- 補助データ定義
  生成のための補助データを与える. 一定の範囲での共通性はあるが, 汎用ではない情報を変数としておき, 生成パターン定義やユーザ生成指示とは別に与える.

以下では, 従来の生成ルールの拡張である生成パターン定義と生成指示コメントの一般化, 及びユーザインタフェースの概略について説明する.

3.2 生成パターン定義

パターン定義の内容は以下のように定めた.

- パターンの名前
- パターンの説明及び説明図ファイルの指定
- パターン構成要素の宣言
- 生成ルールの定義
- 補助関数／述語の定義
- 出力先ファイルの定義

パターン構成要素の宣言 パターンのパラメタとその入力方式を指定する. ここで宣言した名前はルールの中で変数として使用できる. 入力方式には, user(ユーザ生成指示), infile(入力クラスファイル), extra(補助データ) の3種類の指定がある. また, 構成要素には, 複数のクラス／メンバに対応するものがあるため, リスト変数 (@で始まる) や連想配列変数 (%で始まる) も構成要素として宣言することができる.

生成ルールの定義 従来のルールに加えて, 処理対象クラスを特定しない一般生成ルールを用意した. また, 従来の関数生成の宣言生成ルールと定義生成ルールは, それぞれクラス定義編集ルール, クラスルールと名称を変更し, 関数以外の情報の生成にも用いる汎用的なものにした. メンバ操作ルールには, データメンバ用とメンバ関数用のルールを用意した. メンバ操作ルールの呼出しには, クラス名を指定して呼び出す構文を追加し, メンバ操作ルールを一般生成ルールからも呼び出せるようにした.

出力先ファイルの定義 ルール毎にファイル名を決定する方式を指定する. 入力のファイル名や上記の構成要素の値を利用できる.

これらの実際の形式は第5節の事例において説明する. なお, 生成処理は, 後述するようにPerl Version 5を用いて実装しているため, パターン記述中の式や変数などの表現はPerl5の形式を多く流用している.

3.3 生成指示コメントの一般化

従来のhas-a指定のようなメンバの特性を指定する形式に加えて, 生成指示コメントの一般形を導入した.

これは, パターン毎に導入した変数 (入力形式をinfileと指定した構成要素) にプログラムの構成要素を対応付けるものである. 実際には, 変数にコメントを付加されたプログラム要素名を設定するものであり, その変数を生成ルール中で使えるようにするものである. 以下に例を示す.

```
int i ; //!!pattern(Pat, $elm)
```

このように, 生成指示コメントを与えた場合, パターンPatの構成要素を表す変数$elmに, データメンバ名 'i' が登録される.

但し, 関数の場合は, 関数多重定義に対応するために, その型 (シグネチャ) の文字列を用いる. 例えば, 'int f(int)' などである.

また, 使用できる要素は単純変数のみでなく, リスト変数, 連想配列変数もある. 例を以下に示す.

```
int f ();  //!!pattern(Pat, @list)
class A ;  //!!pattern(Pat, %table = val)
```

ここでは, リスト@listの一要素に'int f ( )'を追加し, 連想配列%tableに'A'をキーとして値valを登録する. また, 次のように複数の要素を対応させることもできる.

```
int i ; //!!pattern(Pat, $var, @list)
```

3.4 生成処理のユーザインタフェース

生成のユーザインタフェースとしては, GUI インタフェースとバッチ型インタフェースを用意した.

GUI インタフェースでは, パターン定義で与えた説明を表示する機能をもち, GUI でのメニュー選択等により, ユーザ生成指示に相当する情報をを与えることができるようにしている.

一方, 大量に一括生成を行なう場合のために, バッチ型インタフェースも用意した. これはデザインパターンのように一般化されていないが, 実際の開発で現れる定型的処理を, 一括して自動生成する場合を意図したものである. この場合は, ユーザ生成指示はファイルに記述して与えることになる.

4 システム構成

図 1にシステムの構成を示す.

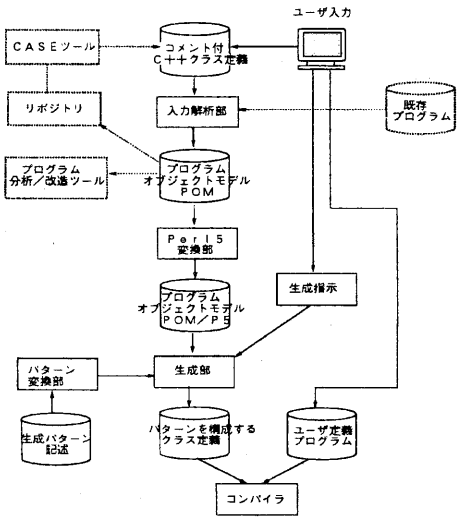

図 1: 自動生成システムの構成

ここで, 入力解析部では, ユーザの入力した C++ プログラムを解析して, その構造を表現するプログラムオブジェクトモデル (POM) を生成する. この部分は, C++と yacc を用いて実装している. 点線で示しているように, 将来 C++コードを生成する CASE ツールなどと連携させることを考えている. また, 関連の種々のツールのベースになるように汎用的な設計を行なっている.

生成部は, 生成時のパターンマッチング処理の容易さとデータ構造記述能力の強力さから, Perl Version 5 を用いて実装している. このため, プログラムオブジェクトモデルは, Perl 5 変換部によって, Perl 5 でデータとして読み込める形式 (POM/P5) でファイルに出力される.

パターン変換部は, 生成パターン記述を Perl5 の関数定義やデータ定義に変換する. この結果は, ユーザからの生成指示に従って生成部に読み込まれる.

生成部は, POM/P5 のデータを読み込み, さらに生成指示コメントの情報から種々の変数を設定する. 以上の前処理を行なった上で, 生成ルールの種別毎にルールが実行される. そこでは, まず出力先ファイルを決定した上で, 対応する Perl 関数が呼び出され, 各パターンに対応したプログラムが生成される.

GUI の実装は Perl/Tk を用いて行なっているが, 一部の機能は WWW の CGI スクリプトとしても実装し, 分散環境への対応も行なっている.

5 適用例

Gamma らのカタログ中の Decorator パターンについて適用例を示す.

まず, Decorator パターンとは, クラスに複数の機能を追加する場合に, 機能の切替えが柔軟にできるようにする設計方法を示したものである. 図 2にそのオブジェクト構成を示す.

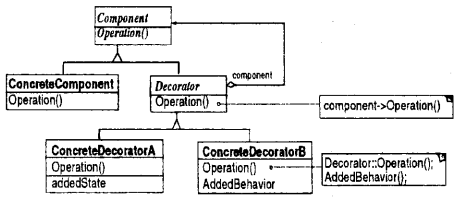

図 2: Decorator パターンのオブジェクト構成

この場合, Decorator クラスとそのサブクラスのオブジェクトは, component というメンバを使って, Operation の操作を委譲して行く. component として, 次々に Decorator のサブクラスのインスタンスを保持していけば, 機能の追加ができるようになっている. このため, 多重継承を用いる機能追加の方法に比べ, 動的に機能の追加/削除が行なえるという利点がある.

このパターンの適用を自動化する場合は，個別の操作対象(ConcreteComponent)と個別のデコレータ(ConcreteDecorator)に相当するクラスを入力として与え，それらの間に前記の図2のような関係を生成することになる．

このためには，生成パターン記述において，まず，構成要素を以下のように定義する．

```
!!PATTERN_ELEMENT
!!BEGIN
  $abst_comp  user   "操作対象の抽象クラスの名前"
  $abst_deco  user   "デコレータの抽象クラスの名前"
  $op_dcl     user   "共通操作の宣言形式"
  $op_call    user   "共通操作の呼出し形式"
  $comp_mem   user   "操作対象を保持するメンバ名"
  %cnc_comps infile "個別の操作対象を登録，1を設定"
  %cnc_decos infile "個別のデコレータを登録，1を設定"
!!END
```

個別の操作対象とデコレータに相当するクラスは，入力クラスファイルにおいて生成指示コメントにより指定しておく．それ以外の構成要素は，ユーザ生成指示で与える．

生成ルールは，一般ルールとして，操作対象抽象クラスの生成ルールとデコレータ抽象クラスの生成ルールの2つを，クラス定義編集ルールとして，個別操作対象クラスのクラス定義編集と個別デコレータクラスのクラス定義編集の2つを，それぞれ定義する．以下には，その一部を示す．

```
#  デコレータ（Decorator）抽象クラスの生成
!!DEF_RULE gen_deco
  !!BEGIN
    # includeするファイル名の取得
    $hdrfile = !!rule_file('gen_comp');
    # クラス定義の出力
    !!PRINT_BEGIN
#include  $hdrfile

class $abst_deco : publibc $abst_comp{
public:
  $abst_deco( $abst_comp* );//コンストラクタ
  virtual $op_dcl           //共通操作
    { $comp_mem->$op_call; };
private :
   $abst_comp * $comp_mem;
 };
   !!PRINT_END
 !!END

#  デコレータ用のクラス定義編集ルール
!!DEF_HEADER_ADD_RULE hdr_add_deco
  !!BEGIN
    #  個別のデコレータにベースクラスを追加
    !!IF ( $cnc_decos{$class}  )
      !!BEGIN
        # includeするファイル名の取得
        $hdrfile = !!rule_file('gen_deco');
        !!PRINT_BEGIN
#include  $hdrfile
        !!PRINT_END
      # ベースクラスを追加
       !!add_base_class($class,
            $abst_deco, 'public', 0 );
      # ベースクラスを追加後のクラスヘッダ部の出力
       !!print_class_head($class) ;
      # 操作メソッドの宣言をクラス定義に追加
       !!PRINT_BEGIN
public:
  virtual $op_dcl ;
        !!PRINT_END
    !!END
  !!END
```

ここで，!!rule_file() は引数で与えたルールの出力先ファイル名を得る関数の呼出しである．

!!add_base_class() は処理中のクラスに引数で与えたクラスをベースクラスとして追加するための関数である．

!!print_class_head() は，そのように加工したクラスのヘッダ部を出力するための関数である．

次に，ルールの出力先ファイルの定義は以下のように指定する．

```
#出力先定義
!!DEF_RULE_OUTFILE
!!BEGIN
  gen_comp      default = $abst_comp . '_gen.h'
  gen_deco      default = $abst_deco . '_gen.h'
  hdr_add_comp  default = $input_basename . '_gen.h'
  hdr_add_deco  default = $input_basename . '_gen.h'
!!END
```

ここでは，デフォルトのファイル名の規約を指定しておき，ユーザの生成指示によって置換えられるようになっている．

Gammaらの著書[1]では，Decoratorの例として，テキスト表示画面のクラス(TextView)に，スクロール機能(ScrollDecorator)や境界線付けの機能(BorderDecorator)を追加する例を示している．ここでも，その例を用いる．

入力として，継承関係を持たない単独の3つのクラスの定義ファイル(TextView.h, ScrollDecorator.h, BorderDecorator.h)を与える．これに対して，上記の関係を構成するクラス，Componentと Decoratorを生成し，それらとの継承関係を持たせるように入力のクラス定義を書き換えることになる．この入出力関係を含めたオブジェクト構成を図3に示す．

実際の入力ソースは以下に示す．個々のクラスの役割はコメントで指示している．

TextView.h:

```
class TextView
{ //!!pattern(Decorator, %cnc_comps = 1)
private:
  char * text;
};
```

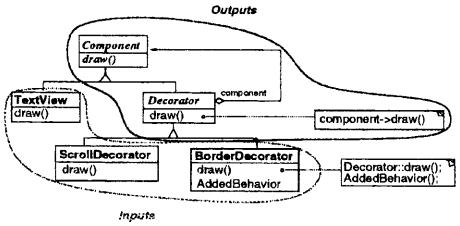

図 3: Decorator パターン自動適用の入力と出力

ScrollDecorator.h:

```
class Position;
class ScrollDecorator
{ //!!pattern(Decorator, %cnc_decos = 1)
private:
  Position * scroll_position ;
};
```

BorderDecorator.h:

```
class BorderDecorator
{ //!!pattern(Decorator, %cnc_decos = 1)
private:
  int width;
};
```

これに対する生成指示は以下のようになり, パターン要素のうちのユーザ指定部分を記述する.

```
!!USE  Decorator
  !!BEGIN
    $component = 'Component'
    $decorator = 'Decorator'
    $op_dcl    = 'void draw()'
    $op_call   = 'draw()'
    $comp_mem  = '_comp'
  !!END
```

出力のうち, 抽象デコレータと BorderDecorator の結果を示す. なお, この場合, 実際の処理の部分は, 手作業で書き足すことになる.

Decorator_gen.h:

```
#include  "Component_gen.h"
class Decorator : public Component {
public:
  Decorator( Component * ); //コンストラクタ
  virtual void draw()       //共通操作
    { _comp->draw() ; };
private :
  Component * _comp;
 };
```

BorderDecorator_gen.h:

```
#include  "Decorator_gen.h"
class BorderDecorator : public Decorator
{ //!!pattern(Decorator, %decorators=1)
public:
  virtual void draw() ;
private:
  int width;
};
```

6 デザインパターンと自動適用

Gamma の 23 のパターンのうち, 上記の Decorator に加えて, Observer, Visitor, Abstract Factory の 3 つについて生成パターンを記述し, 適用の自動化が可能であることを確認した. また, その他のパターンについても, 適用可能性の検討を行なった.

この経験から, 一概にデザインパターンといっても様々なレベルがあることが分かった. 単一の生成パターンで記述できるものもあれば, 多段階の生成パターンを用いるのが妥当なものもある. また, 自動化可能であっても, 手作業で比較的簡単に適用できるため, 自動化の有効性が低いものもある. さらに我々の手法では意味のある自動化を行なうのが困難なものもあった.

多段階の適用が必要になったのは, Abstract Factory の場合であった. これは, 単一の生成パターンで書き下すことは不適当で, 以下のような三段階に分割して適用するのが妥当であることがわかった.

1. 具体プロダクトから具体ファクトリの生成
2. 具体プロダクトから抽象プロダクトの生成
3. 抽象プロダクトから抽象ファクトリの生成

我々のアプローチによる自動化の可能性とその有効性という観点から, Gamma らのパターンは以下のように分類できる.

- 自動化可能かつ有効 (14 種)
  Abstract Factory, Builder,
  Factory Method, Prototype, Singleton,
  Adapter,Composite, Decorator,
  Memento, Observer, Strategy, State,
  Template Method, Visitor
- 自動化可能だが, 有効性が低い (4 種)
  Chain of Responsibility, Facade,
  Mediator, Proxy
- 自動化が困難 (5 種)
  - 入力の構造に依存する部分が多過ぎる
    Bridge, Iterator, Flyweight, Command
  - 概念が抽象的過ぎる
    Interpreter

ここで自動化困難と分類したものについては, どのようなサポートが有効であるかを今後検討していきたい.

7 関連研究

デザインパターンの収集については, 多くの努力がなされており, 国際会議も開催されている [4][5] . 一方, パターン適用の自動化／支援の研究はまだ始まったばかりであるが, Budinsky らの研究 [6] と山本らの研究 [7] がある.

[6] は, パターンカタログの内容を HTML 化し, それに加えて, 対応するクラス宣言を生成する機能を与えている.

[7] は, オブジェクト指向 CASE ツールの中にデザインパターンの適用支援機能を加えるという構想であり, そのためのパターンの記述形式を定めている.

我々のアプローチは, 入力として C++クラス定義を与え, それを構文解析した情報を元に生成処理を行なうことと生成内容をルールとしてユーザ定義可能にするという点で独自性がある. 入力は C++クラスに限定されているが, その分, C++特有のイディオムの生成との連携も容易であり, 自動化の比率を上げることができる. また, 既存の C++コードにデザインパターンを組み込んでいくことも可能になる.

8 おわりに

本稿では, デザインパターンの適用を簡略化するために, 選択されたパターンに対応する C++プログラムスケルトンを生成するシステムについて述べた. これは, 既開発の C++イディオム生成システムを拡張／一般化したものであり, そのルール記述と生成指示の一般化について解説し, 実際の適用例を示した. また, この経験から自動化の可能性という観点でデザインパターンの分類を行なった.

なお, この手法は, デザインパターンのようには一般的なものばかりでなく, 実際の開発で現れる定型的な処理パターンの自動生成にも利用できるものである.

今後の課題は数多くあるが, 当面は, 実用性を向上させる点を重視して, 以下の項目を中心に研究を進める予定である.

- より多くのパターンの記述と登録
- 実プロジェクトへの適用と評価
- カスタマイズ後の再生成への対応
- 既存コードとの整合性の検証機能

また, 図 1のシステム構成において示したように, 解析結果から得られるプログラムオブジェクトモデルをベースにして種々のツールとの連携を図ることも検討している.

9 謝辞

本研究に, 種々の御協力を頂いた, ソフトウェア研究部の毛利部長, 吉田主任研究員を始め, 関係者の皆様に感謝致します.

参考文献

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. 本位田, 吉田 監訳, デザインパターン, ソフトバンク 1995.

[2] 直田, 上原. 分散環境における受動的オブジェクトの実現／管理手法. オブジェクト指向シンポジウム'95 論文集, pp. 141–148, 1995.

[3] J. O. Coplien. *Advanced C++ Programming Style and Idioms.* Addison-Wesley, 1992. 安村, 大谷, 渦原訳, C++プログラミングの筋と定石 1994.

[4] J. O. Coplien and D. C. Schmidt, editors. *Pattern Language of Program Design.* Addison-Wesley, 1995.

[5] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Pattern Language of Program Design 2.* Addison-Wesley, 1996.

[6] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, Vol. 35, No. 2, pp. 151–171, 1996.

[7] 山本, 松本. ＣＡＳＥツールによるデザインパターン適用支援. 情報処理学会研究報告 96-SE-111, pp. 41–48, 1996.