C++クラスライブラリに対応したソースジェネレータの開発

納富 一宏[1] ・ 亀田 茂男[2] ・ 石井 博章[1]

[1] 情報工学科

[2] 工学部情報工学科4年

Development of Source Program Generator for C++ Class Library

Kazuhiro NOTOMI [1)], Shigeo KAMEDA [2)], Hiroaki ISHII [1)]

Abstract

This paper describes a development of source program generator for C++ Class Library. We are developing a new source program generator, which has a skeleton-program database. This database has two types of record(class and method), which are extracted from some all-purpose program sources. And already recorded classes are used for generating of new skeleton program. Target program sources are generated with classes and methods that are this database. If you use this system, you can develop various programs with low cost, and utilize existing program sources.

Key Words: Skeleton, Generate, Class Library, C++

1．はしがき

現在のコンピュータ事情は Microsoft 社の OS「Windows」を主体にしているのが現状であり、それにともないソフトウェアの開発では Windows 対応のアプリケーションを作ることが必要となってきた。しかし、Windows 上でのプログラミングは旧態依然のプログラミングに比して大変複雑な作業をプログラマに強いるものであると言える。たとえ、CUI(Character based User Interface)プログラミングのスペシャリストであってもGUI(Graphical User Interface)プログラミングを習得することは困難である。この理由は、前者が逐次処理方式をベースにしたプログラミングであるのに対し、後者はイベントドリブン方式であるためである。また、GUI 環境は初心者にとってはとても使いやすく親しみやすい環境となるが、逆にそれを開発する側にとってはウィンドウを一つ作るだけでも大変手間のかかる作業であり、実際の処理をする部分を記述することはなおさら困難である。

インターネットの爆発的な人気によりパソコンが急速に家庭レベルで普及しているが、ユーザがパソコンを使うことに慣れてくればインターネットにだけパソコンを利用するとは考えにくい。そうなればユーザからの要求は必然的に高くなりWindows 対応のソフトウェアもその需要に合わせて作らなければならない。しかし、前述したように Windows 上でのプログラミングというのは大変複雑でありプログラミングに慣れることにも非常に時間がかるため、ユーザの需要が増大すれば1960 年代後半に話題となった「ソフトウェア危機」と同様なプログラマ不足という問題も起こり得るのではないかと考えられる。だが、現在では Visual Basic や Delphi をはじめとする Visual 系言語が開発され、初心者でも Windows 上でのプログラミングが比較的容易になり、プログラマ不足は危惧するほどではなくなったと言える。

C++言語での Windows アプリケーション開発環境には Visual C++や Borland C++などがある。これらのアプリケーショ

ンではコードジェネレータなどがプログラム開発のサポートを行っている。

しかし、これらのコードジェネレータは Windows 対応のアプリケーションの開発を行うための機能であり、Windows アプリケーション開発用のクラスライブラリを継承した形のコードしか生成できない。そのため生成されるコードは実際の処理部分を排除したコードとなる。

そこで本研究では、ユーザ定義の汎用クラスや以前ユーザが作成したプログラムソースなどを利用できるコードジェネレータの開発を目的として、汎用プログラムソースをクラスとメソッドに分割して階層化し、データベースの作成を行った。そして、データベースに登録されているクラスを結合してプログラムソースの自動生成を行った。

2. ソフトウェア開発におけるコードジェネレータ

2.1 コーディング

コーディングをする際に求められることはわかり易さである。プログラムをコーディングした人だけでなく、第三者がプログラムソースを見た際に、どのような処理をしているかがわかるようにコーディングしなければならない。そのためには変数、関数、クラスなどに意味のある名前をつけることや、プログラムソースを見渡した際に各ブロックごとにインデントをつけるなどの見た目のわかり易さなどが重要である。しかし、それだけでは処理部分を見た際にわかりづらくなってしまうので、プログラム中の重要な変数や関数などにコメントを入れておく必要がある。

特に集団によるソフトウェア開発(プロジェクト)や大規模ソフトウェア開発などの場合はなおさらコーディングで注意しなければならい。このような場合、各処理ごとにチームに別れてコーディングすることが多く、プログラムソースは作成者のみが見るわけではない。この時にプログラムソースがわかりにくくコーディングされていたり、デバッグする際のことを考えずにコーディングされていると、開発効率の低下の原因になり得る。

また、バージョン管理も重要である。作成したプログラムに追加や仕様変更などが生じた場合、今まで作成したプログラムソースを破棄するのではなく、再利用できる形にコーディングされていることが望ましい。

2.2 プログラムソースの自動生成

人が一からコーディングを行うとスペルミスなどが起こり、デバッグなどに時間がかかり作業効率が低下する。また、コーディング自体に大変時間がかかる。そこで、間違いのないプログラムソースを自動生成すれば、作業効率があがる。さらに、プログラムソースを雛形から自動生成することにより、信頼性も向上する。

本稿ではプログラムソースの自動生成をジェネレート(Generate)、自動生成する際に使用するプログラムソースの雛形をスケルトン(Skeleton)と呼ぶことにする。

2.3 スケルトン

スケルトンとはプログラムソースをジェネレートする際に一番元となるプログラムソースである。スケルトンからジェネレートされたプログラムソースは、ユーザが必要な処理を付け加えることにより独自のプログラムを作成するために使用される。スケルトンが複数用意されていれば、各々のスケルトンは違う目的のために使用されると考えるのが自然である。どのようなスケルトンを選択するかによってジェネレートされるプログラムは全く異なったものとなる。つまり、スケルトンの数が増加すれば、ジェネレートされるプログラムソースも増加する。

また、スケルトン自体にユーザの望む処理がすでに記述されているならば(エディタ作成のスケルトンやペイントツールのためのスケルトン)、ジェネレートされたプログラムソースに変更の必要はない。

2.4 コードジェネレータ

2.3節にも記述したようにスケルトンの数が増加すれば作成されるプログラムソースの数も増加する。しかし、スケルトンは可能な処理がすでに決定されているプログラムソースであり、ジェネレートされるプログラムソースも選択されたスケルトンによって決定されてしまう。

そこで、スケルトンを一つのプログラムソースとしてではなく、クラスや汎用関数に分けてデータベース化し、個々のクラスや汎用関数を選択して結合することによって、スケルトンと同じ形だけではなくユーザ独自の新たなプログラムソースをジェネレートすることが可能となる。

また、汎用的なプログラムソースをスケルトンとしてクラスなどに分割しデータベースに登録することによって、ユーザが過去にコーディングしたプログラムソースをスケルトンにして、新たにプログラムソースをジェネレートすることも可能である。汎用クラスや汎用関数などを結合させてコードをジェネレートする際にも、ユーザがスケルトンを追加できることにより、ジェネレートできるプログラムソースの数は増加する。

これによってユーザが要求する処理が過去にコーディングしたプログラムソースの中に使用されていた場合、その処理部分をまたコーディングし直すことなく新たなプログラムソースに追加することができ、プログラムソースの再利用が簡単になる。

3．データベース利用によるソースジェネレータ

図 3.1 に作成した本システムの構成図を示した。以下、各処理の実現方法を述べる。

3.1 データベースのシステム構成

図 3.1 からわかるように本システムのデータベースは一つのスケルトンを一レコードとし、一レコードは三階層にわかれている。一階層目は各スケルトンを区別するために、スケルトンの基本情報

① スケルトン名

② スケルトンプログラム内にあるクラス数

③ クラス情報

をメンバとしている。

二階層目は一階層目のクラス情報を元に

① 各クラス名

② クラスID

③ クラス内にあるメソッド情報

をメンバとしている。

三階層目はメソッド情報を元に

① メソッド名

② メソッドID

をメンバとしている。

クラスIDとメソッドIDは対応する定義ファイルを参照するために使用される。

このようにデータベースを階層化することにより、必要なデータにアクセスすることが可能となる。

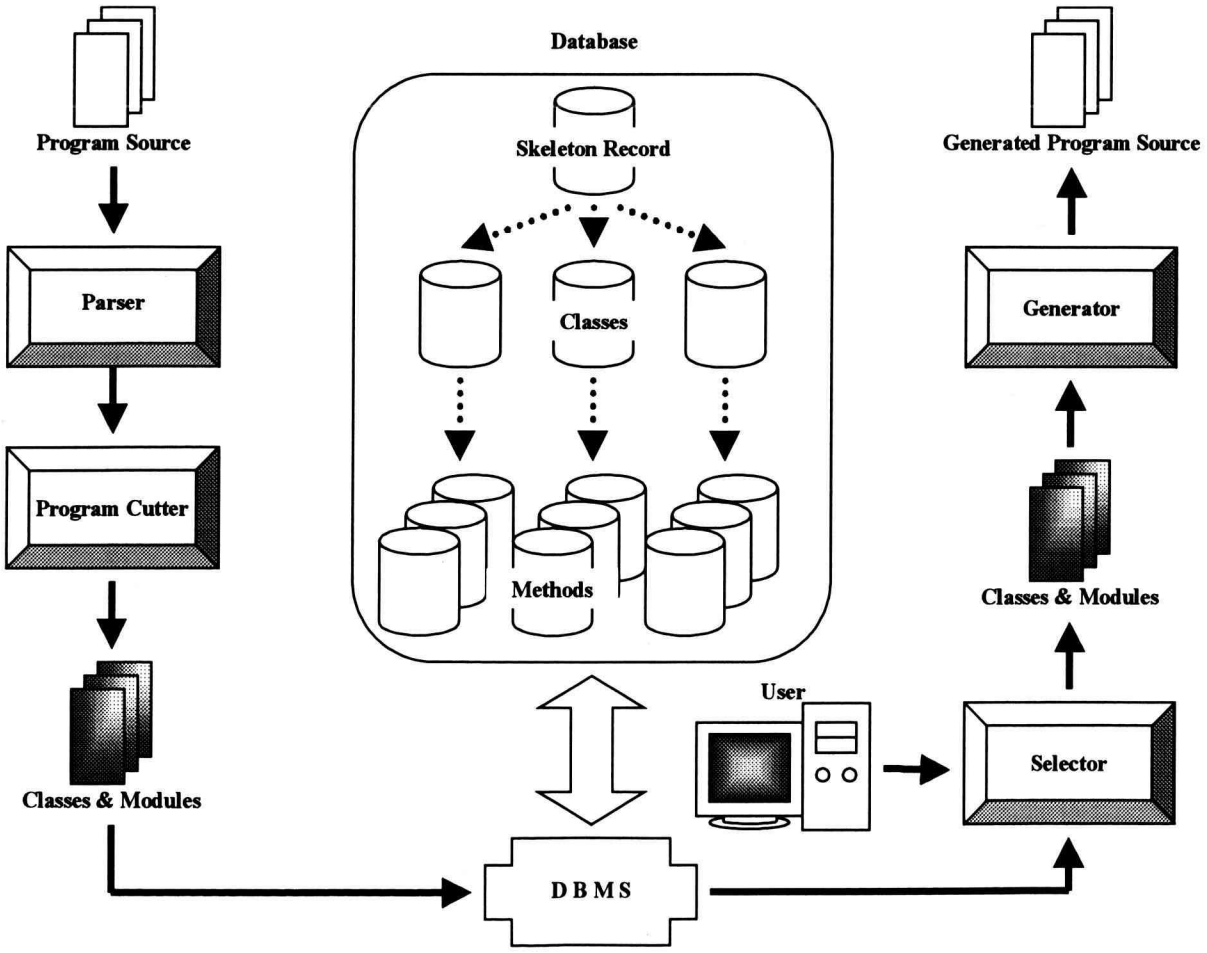

図 3.1 システム構成図

3.2 プログラムのジェネレート

登録されているスケルトンをユーザが選択すると、データベースファイルに格納されている目的のスケルトン情報が格納されているレコードを取り出すことができる。また、このレコードに記述されている各クラス情報を参照し、クラス定義ファイルをジェネレートする。その後、クラスに対応するメソッド情報を参照し、メソッド定義ファイルをジェネレートする。この作業を繰り返し、目的とするジェネレートプログラムを作成する。

また、スケルトン内にあるクラス名はユーザが望むクラス名に変更可能である。

3.3 汎用的なプログラムソースからのスケルトンの作成

3.3.1 プログラムソースのパースとカット

汎用的なプログラムソースをデータベースにスケルトンとして登録する前処理として、パース(Parse)とプログラムカット(Program Cut)行われる。

処理内容は次の手順となる。

① プログラムソースの余剰なホワイトスペース(White Space)を削除し、改行文字への置き換えを行い、トークン(Token)単位に分けたファイルを作成する。

② プログラムソースをスケルトンとしてデータベース登録する際に使用する一時ファイルを作成する。

③ プログラムソースをクラス定義ファイルとメソッドファイルに分割する。

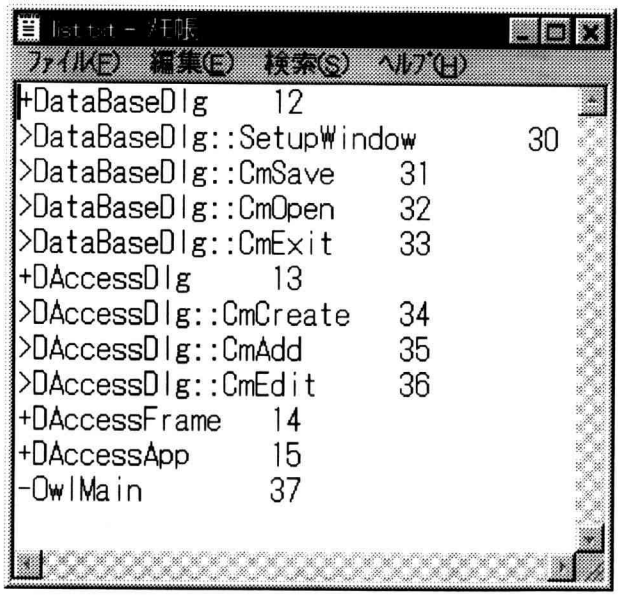

図 3.2 一時ファイル

②の処理では、図 3.2 に示したように、クラスの場合にはクラスプレフィックス記号'+'、クラス名、クラスIDの順に、メンバ関数にはメンバプレフィックス記号'>'、メンバ関数名、メソッドIDの順に、そして汎用関数には汎用関数プレフィックス記号'-'、関数名、メソッドIDの順にそれぞれ一行づつファイルに書き込み作成する。このファイルにはスケルトンとするプログラムソースは記述せず、どのようなクラスとメソッドが存在するかが記述されているのみである。

③の処理では、前者の①で作成されたファイルを基にして、プログラムソースをクラス定義部分とメソッド部分に分割する処理が行われる。分割を行う際にトークンを最大五つずつ用い、クラスの定義はどこからどこまでなのか、またはメンバ関数であるのか汎用関数であるのかなどの解析を行う。クラス定義ファイルはクラスIDを六桁にし、拡張子を.h というファイル名で保存する。メソッドファイルも同様にメソッドIDを六桁にし、拡張子を.met というファイル名で保存した。

3.3.2 データベースへの登録

3．1節の②で作成された一時ファイル(図 3.2 を参照)を基にしてデータベースのレコードを作成する。レコードはユーザから指定されたスケルトン名、スケルトンに含まれているクラス数、そしてクラス名とクラスIDからなる汎用クラス(ClassData class)のオブジェクトによって構成されている。各クラスはクラスIDを六桁にし、拡張子が.cla というファイルに、対応するメソッド情報を格納している。.cla ファイルにはメソッド名とメソッドIDからなる汎用クラス(MethodData Class)のオブジェクトが格納されている。また、スケルトンに含まれる汎用関数はクラス名をNormal とした仮想的なクラスのメンバ関数にすることによって格納している。このように一つのプログラムソースをクラス定義とメソッドに細分化しツリー構造にすることによって、各クラスやメソッドは個別に情報を引き出すことができ、別スケルトン同士のクラスやメソッドの結合が可能となる。

3.4 データベースを利用したスケルトンの作成

3．1節ですでに述べたように、登録されているスケルトン内のクラスとメソッドには順を追ってアクセスできる。そこで、データベースにすでに登録されているクラスを使用して新たにスケルトンを作成する。

登録されているスケルトン内のクラスを個別に集め、どのスケルトンのクラスなのかという情報を保持しておく。そして、ユーザからの作成指示が出た際に、各クラスのスケルトンから対応するクラス情報を引き出し、新たなデータベースのクラス情報に追加する。また、各クラスのメソッドはクラス情報から引き出されるため、登録は不要である。

4．評　価

4.1 システム概要

作成したコードジェネレータのメインウィンドウを図 4.1 に示した。コンボボックス内に格納されているスケルトン名を選択すると、対応するクラス名をリストボックスに表示する。同じようにクラス名を選択すると、メソッド名を表示する。

以下に本システムを使用してスケルトンとプログラムソースを作成した時の用例を示す。

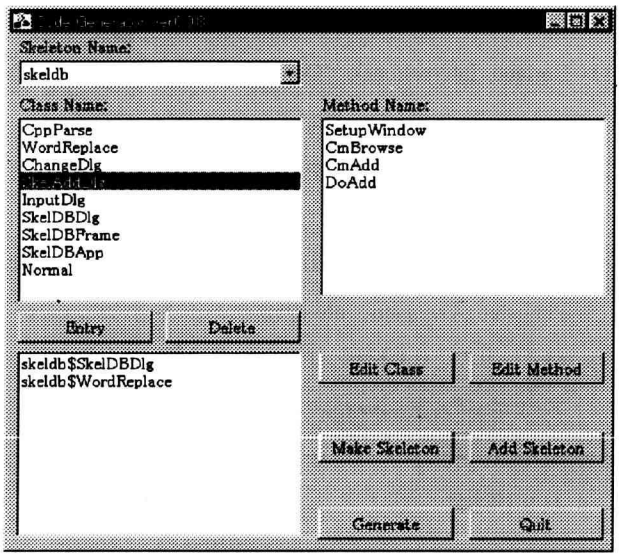

図 4.1 メインウィンドウ

4.2 プログラムソースの作成

スケルトンを選択した状態で、図 4.1 の Generate ボタンを押すと図 4.2 のダイアログが表示される。図 4.2 のリストボックスに表示されているクラス名は、選択したスケルトン内にあるクラスである。これらのクラス名は、テキストボックスに任意のクラス名を書き込んで Change ボタンを押すことにより変更可能である。

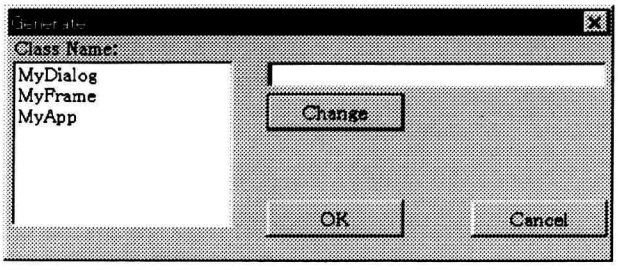

図 4.2 ジェネレートダイアログ

このシステムを利用して作成されたジェネレートプログラムと、それを実際に実行させた画面を図 4.3、図 4.4 に示す。

ジェネレートされるプログラムは図 4.3 のようにブロックごとにインデントされ、クラスの文頭にはコメントが入る。このプログラムソースを実際に実行させると図 4.4 のようなアプリケーションが作成される。

```
#include    <owl¥owlall.h>
#include    <stdio.h>
#include    "hoge2.rh"

//----------------------------------------------------------------------
//  class   :   MyDialog
//----------------------------------------------------------------------

class   MyDialog
        :   public  TDialog
{
public:
    TEdit*  Name;
    TEdit*  Path;
    TEdit*  Value;
    TEdit*  Up;

    int     DataNum;
    int     CurrentPoint;

public:
    MyDialog(TWindow* parent)
        :   TDialog(parent, MAINDIALOG)
    {
```

```
{
public:
    MyApp(char* title)
        :   TApplication(title)
    {
    }

    void    InitMainWindow()
    {
        SetMainWindow(new MyFrame(NULL, Name, new MyDialog(NULL)));
    }
};

//----------------------------------------------------------------------
//  関数    :   OwlMain
//----------------------------------------------------------------------

int     OwlMain(int argc, char** argv)
{
    MyApp   app("フレーム君 Ver1.02");

    return  app.Run();
}
```

図 4.3 ジェネレートプログラム

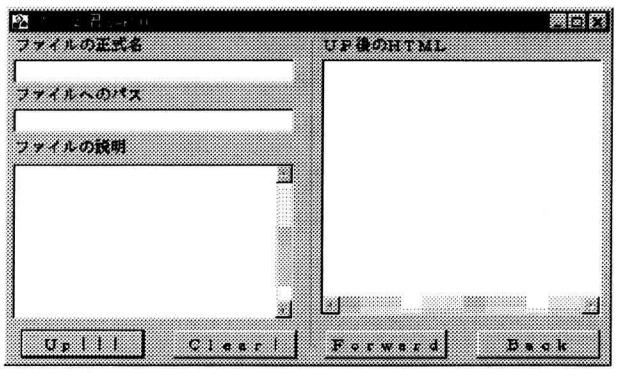

図 4.4 実行されたジェネレートプログラム

4.3 スケルトンの作成

図 4.1 の Add Skeleton ボタンを押すと図 4.5 のダイアログが表示される。Base File Path のテキストボックスにスケルトンに登録したいプログラムソースのパスを書き込む。そして、スケルトン名に任意の名前を書き込み Add Skeleton ボタンを押すと、汎用プログラムソースがスケルトンとして、データベースに登録される。スケルトンを登録した後は、メインウィンドウ（図 4.1）のスケルトン名に即座に反映され、登録したスケルトンを利用し

て、プログラムをジェネレートすることが可能である。

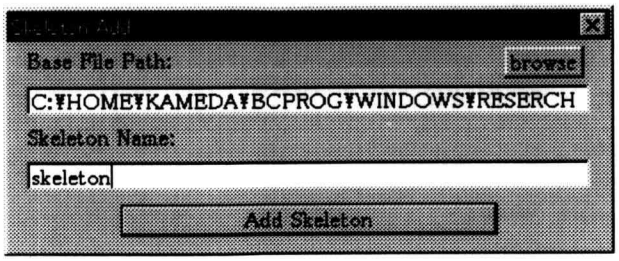

図 4.5 汎用プログラムソースからのスケルトン作成

すでに登録されているデータベース内のクラスを利用して新たなスケルトンを作成する場合、図4.1のスケルトン名を選び、要求するクラスをEnterボタンによって登録する。これは、登録されているすべてのスケルトンから任意に選択することが可能である。また、データベースに追加する前ならば、Deleteボタンにより、削除することが可能である。登録する際は、図4.1のMake Skeletonボタンを押すことにより、スケルトン名を入力するダイアログが表示され、任意のスケルトン名を入力すると、登録が完了する。

4.4 考察

本システムでは、スケルトンとして汎用プログラムソースを登録することができる。各スケルトンは、プログラムソースをクラスとメソッドに分割してデータベースに格納している。上記のようにすることにより、すでにデータベースに登録されているクラスを結合して、新たにスケルトンを作成することが可能である。また、Windows対応のアプリケーション作成のために使用されるクラスライブラリを継承したクラスだけではなく、ユーザ定義の汎用クラスもデータベースに登録でき、そのクラスを用いて新たなスケルトンを作成することも可能である。

プログラムソースをジェネレートする際は、データベースに登録されているスケルトンを選択するだけでジェネレートが可能である。スケルトンを選択した際に、メインウィンドウのリストボックスに対応するクラス名とメソッド名を表示する。

データベース内のクラスとメソッドは外部エディタを用いることにより、編集可能である。

5. むすび

本システムでは汎用プログラムをクラスとメソッドに分割することでスケルトン化を実現させているが、各クラスやメソッドには対応するドキュメント(Document)は存在しない。ユーザがスケルトンを選択する際に、各々のクラスがどのような処理を行えるクラスであるのかがドキュメントとして存在しているほうがわかりやすい。

また、本システムは一つのコンピュータに一つ存在していなくてはならず、データベースの拡張やジェネレートする際に障害となる可能性がある。この問題を解決するために、本システムをCGI(Common Gateway Interface)対応にし、インターネット上からスケルトンの追加、プログラムソースのジェネレートを可能にすることが今後の課題である。

参考文献

1) Borland International: "Object Windows Programmer's Guide", Borland International, August 1996
2) Microsoft:『Win32API オフィシャルリファレンス』、アスキー、1997年7月
3) Scott Meyers: "Effective C++", Addison-Wesley Publishing Company, July 1992
4) M.A.エリス、B.ストラウストラップ:『注釈 C++ リファレンス・マニュアル』、トッパン、1992年11月
5) B.ストラウストラップ:『プログラミング言語C++』、トッパン、1993年8月