ソフトウェア工学 107－7
（1996．1．18）

オブジェクト指向設計を支援する設計モデルからのコード生成機能

野田夏子†, 岸 知二†, 前川佳春†, 山之内徹‡

†NEC マイコンソフト開発環境研究所
‡NEC C&C 研究所

ソフトウェア開発においては, ソフトウェアアーキテクチャの決定が重要である. 我々は, 早期に実環境で動作させてこの決定の妥当性を確認することができるように, 部品を利用して実装メカニズムとの対応付けを表現した設計モデルから実行可能なコードを生成する技術を検討している. 実装メカニズムとの対応付けは, 環境, 資源, ドメイン等により変化するものであるから, このコード生成においては用いる部品や部品の展開方法をカスタマイズできなければならない.

本稿では, 部品の展開方法をユーザ自身が容易にカスタマイズできるコード生成方式を提案する.

Natsuko NODA†, Tomoji KISHI†, Yoshiharu MAEGAWA†, Toru YAMANOUCHI‡
{n-noda,kishi,maekawa}@ccs.mt.nec.co.jp,yamano@swl.cl.nec.co.jp

†Microcomputer Software Development Laboratories, NEC Corporation
Igarashi building, 11-5, Shibaura 2-chome,
Minato-ku, Tokyo, 108, JAPAN

‡C&C Research Laboratories,NEC Corporation
1-1, Miyazaki 4-chome,
Miyamae-ku, Kawasaki, Kanagawa, 216, JAPAN

In software development, the decision on software architecture is very important. We are examing the software development approach, in which fully executable code is generated from design model to verify the correctness of the architectural decision. In order to generate the code, we use design parts to specify the mechanism by which the design model is realized on the target environment. As the mechanisms necessary to realize the design model depend on the target environment or target domain, design parts and generated code patterns must be customizable.

In this paper, we propose the code generation technique for our design approach. We also discuss on our prototype generator.

1 はじめに

近年，オブジェクト指向開発への注目が高まり，様々な分野に適用されている．オブジェクト指向開発においては，設計段階でのソフトウェアアーキテクチャの決定が重要である．この決定が正しく行われないと，最終的なソフトウェアに性能面などで問題が生じる．しかし，ソフトウェアアーキテクチャの決定の妥当性を設計モデルのレビュー等の手段だけで確認するのは困難なことも多く，開発の終了間際に全体を動作させて初めて問題が顕在化することもまれではない．そこで我々はソフトウェアアーキテクチャの決定に基づく設計モデルを短時間で構築し，実環境で動作させて妥当性を確認する開発アプローチ Crossover の検討を進めている [3]．本アプローチの実現のためには，設計モデルから実行可能なコードを生成することが必要である．

ソフトウェアアーキテクチャの決定は，設計モデルにおける実装メカニズムとの対応付けに制約を与える．どのような実装メカニズムに対応付けるかは，環境や利用できる資源，アプリケーションのドメインによって変化する．また同じ実装メカニズムでも，それをどのようなコードパターンで実装するかはユーザ毎に異なる．したがって，コード生成においては，定形的なコードパターンに変換することはできず，生成の仕方はユーザによってカスタマイズ可能でなければならない．

本稿では，Crossover の実現のための設計モデルからのコード生成において，ユーザによるカスタマイズに適した生成方式を提案する．

2 ソフトウェアアーキテクチャと設計モデル

本章では，コード生成のベースになる設計モデルについて説明する．特に，ソフトウェアアーキテクチャを表現するモデルとそのモデルの制約を受けて設計モデルがどのように構築されるかについて述べる．

2.1 ソフトウェアアーキテクチャモデル

ソフトウェアアーキテクチャとは，ソフトウェアの全体構造を表すものであり，ソフトウェアを構成するコンポーネントとその間のインタラクションという形で表現することが多い [1]．ソフトウェアアーキテクチャはそのソフトウェアの設計に様々な制約を課す [5]．

設計段階では，分析結果に基づき様々な設計判断を加えて設計モデルを構築する．例えば，オブジェクト間の関連をポインタを用いて実装する，等の実装メカニズムとの対応付けを行う．この対応付けを制約するのが，ソフトウェアアーキテクチャである．例えば，アーキテクチャの決定としてプロセス分割が決まると，プロセス間をまたがるオブジェクト間の関連をポインタを用いて実装することはできなくなる．

このように，ソフトウェアアーキテクチャは設計モデル構築に大きな影響を持つものであるから，これを直接的かつ容易に定義し得るモデルがあることが望ましい．我々は，[5] においてオブジェクト指向設計のためのソフトウェアアーキテクチャモデルを提案した．本モデルは，OMT[7] のオブジェクト図を拡張した拡張オブジェクトモデル [5] の上に，コンポーネントとコネクタを配置して記述する．コンポーネントとは，アーキテクチャ表現上の基本単位であり，プロセスやタスクといったものがこれに相当する．コネクタは，コンポーネント間のインタラクションを規定するもので，例えば関数呼び出しやソケットでのコネクションなどがあげられる．図 1は，ソフトウェアアーキテクチャモデルの記述例である．

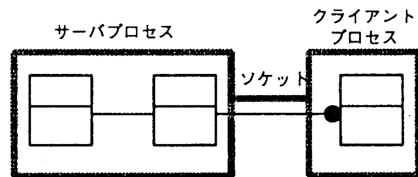

図 1: ソフトウェアアーキテクチャモデルの記述例

2.2 設計モデル

2.2.1 設計モデルの基本概念

Crossover アプローチを実現するために，我々はコード生成に適した設計モデルとその内部表現 Xover 言語を設定した [4], [5].

本設計モデルの基本概念を以下に示す．

- クラス
- モジュール: クラスもしくはモジュールの集合であり，名前のスコープになる．モジュールは他のクラスを利用するために，そのクラスを含むモジュールをインポートできる．
- 関係: OMT の関連や集約等に対応する．

2.2.2 部品

実装メカニズムとの対応付けを設計モデル上で表現するために, 以下の部品を利用する [4].

- クラス部品, モジュール部品: 既存クラスはクラス部品として利用することができる. またクラス部品の集合をモジュール部品ととらえることができる. 他のモデルのモジュールは, モジュール部品として利用できる. また既存コード, クラスライブラリ, フレームワーク等がモジュール部品に対応する. 既存資産を利用してモデルを構築するために, クラス部品, モジュール部品は不可欠である.

- 関係部品: 関係の実装方法を示すためのものである. 一般に設計モデル上で表現された関係の実装方法は一意に決まるものではないので, 相手側クラスの識別方法やリファレンスの管理手段等の実装パターンを関係部品として用意し, これを用いて関係の実装方法を指定する. 例えば, ポインタを用いて関係を実装するなら, ポインタで相手側クラスのリファレンスを管理する関係部品 Pointer を用いる.

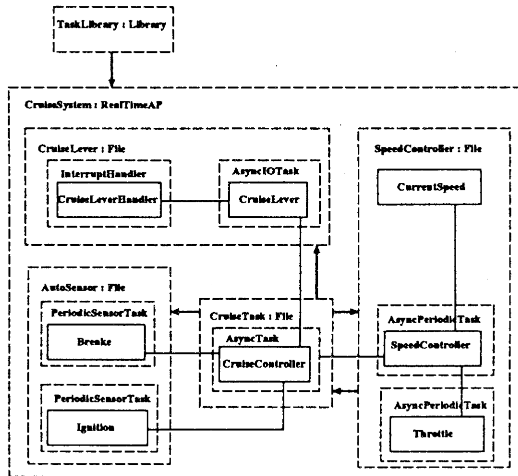

図 2: 設計モデルの例 (クルーズコントローラ)

2.2.3 設計モデルの構築

設計モデル上の概念に部品を対応づけることにより, その実装メカニズムを指定する. 前述したように, こうした対応付けはアーキテクチャの決定によって制約される.

このようにして構築された設計モデルから, コードを生成する. 図 2に設計モデルの例を示す[1]. 図 3にモデルの構築からコード生成に至る全体像を示す. なお, クラスの実装部分の生成については既存技術の適用が可能であるので, 本稿では議論しない.

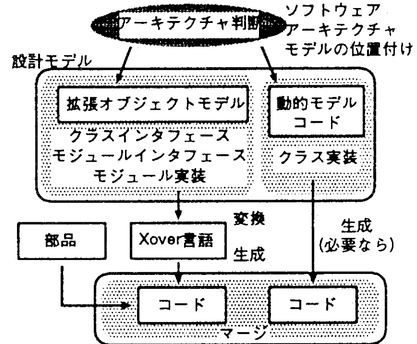

図 3: 設計モデルの構築とコード生成の全体

3 コード生成

3.1 コード生成の要件

前章で述べたように, 設計モデル中には部品が用いられている. 用いる部品は, ユーザ毎に異なる. また部品がどのようなコードパターンに展開されるかについても, ユーザによって異なる. したがって, この設計モデルからのコード生成では, どのようなコードを生成するかがカスタマイズ可能でなければならない.

また, 同じユーザにおいても, 用いる部品を追加したいことがある. 部品が実際に示すファイルやライブラリを変更したり, 展開した時のコーディングスタイルを変更したい場合もある. このような部品の追加や変更は, ユーザの実行環境に依存するものである. したがって, 生成の仕方をユーザがカスタマイズできることが望ましい.

3.2 部品の展開

設計モデル, 部品, 変換ルールからコードが生成される (図 4). 変換ルールは, モデルをどのようなファイルやコードに変換するのかを記述したルール

[1] クラスを囲む破線は, 親クラスとして指定するクラス部品を表す. 例えば, CruiseController クラスはクラス部品 AsyncTask から継承している. またモジュール間の矢印はインポートを表し, クラスを包含していないモジュールはモジュール部品である. 部品の図表現の方法については本稿の趣旨において本質ではなく, ここでは一例をあげるにとどめる.

である [6]. 部品や変換ルールが変われば生成されるコードは違ったものになる.

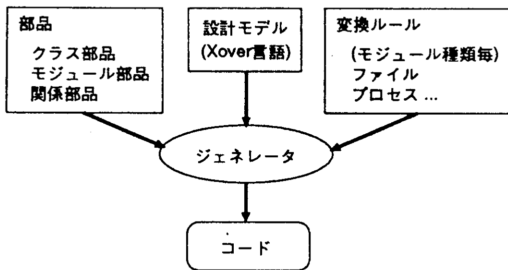

図 4: 生成に必要なもの

設計モデル上で部品が指定されると, 生成されるコード上に部品に対応する情報が展開されなければならない. 次のような展開を行う.

- クラス部品からの継承: クラス部品が実際に表すクラスの名前が親クラス名として展開される. さらにそのクラスから継承するために必要なマクロ宣言等があればそれらの宣言がなされる.
- モジュール部品のインポート: 実際に用いるファイルのインクルード宣言をする.
- 関係部品: 相手側クラスの識別ができるように属性を宣言する.

部品をコードに展開するのは, 部品展開の方法を変換ルールとして記述することにより実現できる. 例えば, モジュール部品 A,B,C があり, これらのモジュール部品のインポートをファイルのインクルード宣言に展開するには, 以下の 3 つの変換ルールで実現できる.

- ルール 1: モジュール部品 A のインポートは, #include "A.h" と展開
- ルール 2: モジュール部品 B のインポートは, #include <B.h> と展開
- ルール 3: モジュール部品 C のインポートは, #include </lib/C.h> と展開

しかし, 変換ルールの実際の記述方法はジェネレータの実装に依存したものであり, ユーザに分かりやすいものであるとは限らない. したがって, 変換ルールでの記述はユーザによるカスタマイズに適さない.

3.3 生成方式

本節では, 変換ルールを修正せずに部品の展開をユーザがカスタマイズできる生成方式を提案する.

3.3.1 基本的なアイデア

変換ルールを修正せずに部品の展開方法をカスタマイズするには, 部品名と部品がコードにどう展開されるかの対応関係をユーザが自由にカスタマイズしやすい形で表現し, 変換ルールとは別に管理しなければならない. このために, 入出力情報対応テーブルとテンプレートを用いる.

入出力情報対応テーブルとは, ジェネレータへの入力中の変数の値とそれによって定まる出力の値との対応関係をまとめた表である. すなわち, 変換のあるステップにおいて s(X1,...,Xn) が Y に変換され, {X1,...,Xn} の部分列 {Xi_1,...,Xi_k} (k は 1 以上 n 以下の自然数. Xi_1,...,Xi_k はそれぞれ X1,...,Xn の中のどれか) に対応して Y が定まる時, Xi_1,...,Xi_k が取る値の組み合わせと Y の対応関係を表にまとめたものである (表 1). このような表を用意すれば, 個別の変換ルールを記述しなくても, この表を検索するメタルールがあれば変換が実現でき, 表の内容を変えれば変換のカスタマイズができる. この, 入出力情報対応テーブルから入力に対応する出力を検索するメタルールを汎用テーブル検索ルールと呼ぶ.

{Xi_1,...,Xi_k}	Y
{Xi1_1,...,Xi1_k}	Y1
...	...
{Xij_1,...,Xij_k}	Yj
...	...

表 1: 入出力情報対応テーブル

前節のモジュール部品のインポートをインクルード宣言に展開する例では, モジュール部品名とインクルード宣言に用いるファイル名の対応関係を表す次のような入出力情報対応テーブルを用意すれば良い.

モジュール部品名	ファイル名
A	"A.h"
B	<B.h>
C	</lib/C.h>

表 2: モジュール部品名とファイル名との対応を与える入出力情報対応テーブル

なお, 入力仕様から出力プログラムへ変換する時, {Xi_1,...,Xi_k} に対応して定まる情報が複数あり, それぞれが変換の別のステップで用いられることがある. この時, 入出力情報対応テーブルを, 定まる情報毎に別々に作成しても良いが, 複数の列を持つ１つのテーブルとしても良い. このような入出力情報対応テーブルの例を表3に示す. この時, 汎用テーブル検索ルールは入力と入出力情報対応テーブルの何列目を検索するかが与えられると対応する出力の値を検索するルールになる.

{Xi_1,...,Xi_k}	Y1	Y2	Y3
{Xi1_1,...,Xi1_k}	Y1_1	Y2_1	Y3_1
...	...	...	...
{Xij_1,...,Xij_k}	Y1_j	Y2_j	Y3_j
...	...	...	...

表 3: 複数の列を持つ入出力情報対応テーブル

また, テンプレートとは, 入力中の変数の値によって出力のコーディングスタイルが定まる時, そのコーディングスタイルを表すものである. すなわち, 変換のあるステップにおいて s(X1,...,Xn) が Y に変換され, {X1,...,Xn} の部分列 {Xi_1,...,Xi_k} (k は 1 以上 n 以下の自然数. Xi_1,...,Xi_k はそれぞれ X1,...,Xn の中のどれか) に対応して Y が定まるが, この Y は u(X1,...,Xn) と表されるようなパターンである時, この Y の形態を定める一般的なパターン u(Φ1,...,Φn) (Φ1,...,Φn は変数) を {Xi_1,...,Xi_k} に対応するテンプレート[2]と呼ぶ. この時, 入出力情報対応テーブルは, {Xi_1,...,Xi_k} とテンプレートとの対応関係を管理することになる[3]. このようなテンプレートを用意すれば, どのようなコードに展開するのか個別に変換ルールに記述しなくても, このテンプレートを展開するメタルールがあれば変換が実現でき, テンプレートのカスタマイズで生成するコードのコーディングスタイルを変えることができる. テンプレート中の変数を入力中の変数で置き換えてテンプレートを展開するメタルールを汎用テンプレート展開ルールと呼ぶ.

クラス部品から継承したクラスを宣言する時, 親クラスとして指定するクラス部品によりクラス宣言の構造が変わる場合がある. 例えばクラス部品 ObjectA,ObjectB,ObjectC があり, それぞれを親クラスとして指定することにより, 属性 A, メソッド M を持つクラス S のクラス宣言の構造が次のようになるとする.

- クラス部品 ObjectA を親クラスとする場合:
  class S : public ObjectA {
  A
  M
  };
- クラス部品 ObjectB を親クラスとする場合:
  class S : public ObjectB {
  MACRO_1(S);
  A
  M
  };
- クラス部品 objectc を親クラスとする場合:
  class S : public ObjectC {
  A
  M
  MACRO_2(S);
  };

上記のようなクラス部品の展開は, それぞれ個別の変換ルールを記述するのではなく, 図 5に示すようなテンプレートと入出力情報対応テーブルを用意すれば良い.

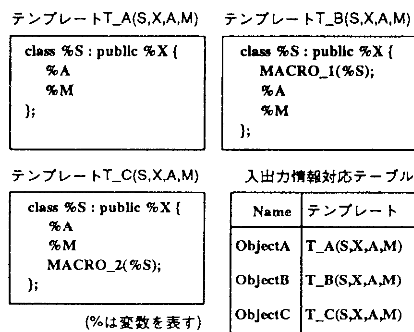

図 5: テンプレートと入出力情報対応テーブルの例

入出力情報対応テーブルとテンプレートを用いた生成方式の全体像を図 6に示す.

3.3.2 生成方式の具体例

3.3.1で, 部品の展開方法を入出力情報対応テーブルとテンプレートで与える方式を述べた. ここでは具体的に, クラス部品, 関係部品, モジュール部品の展開方法をどのような入出力情報対応テーブルとテ

[2]直感的には一種のパラメータつきマクロである.

[3]実際の対応関係の管理は, 入出力情報対応テーブルにテンプレートそのものを格納しても良いし, テンプレートへのポインタ(テンプレートの名前, 番号等) を格納しても良い. これはどう実装するかの問題である. この論文では見やすさのために, 入出力情報対応テーブルへはポインタを格納するような表記とする.

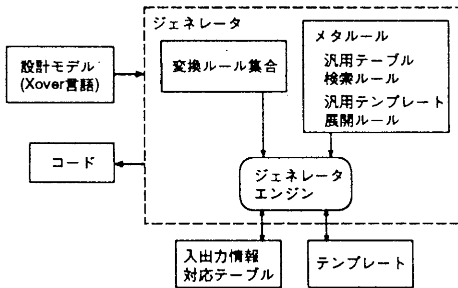

図 6: 生成方式の全体像

ンプレートで表現するかを説明する.

クラス部品

クラス部品は, 既存のクラスを表す部品であり, 親クラスの指定に使われる. これを展開するためには以下の情報が必要である.

- 実際のクラス名
- このクラス部品を用いるために必要なモジュール部品名: 他のモジュールにあるクラスを利用するためには, そのクラスが宣言されているモジュールをインポートしなければならない. したがってクラス部品を利用するためには, そのクラス部品が定義されているモジュール部品が必要である.
- クラス宣言の構造を表すテンプレート

入出力情報対応テーブル

Name	Class	Module	Template
C_1	Task	Tasks	T_A
C_2	Task2	TasksB	T_B
C_3	Task3	Tasks	T_C
⋮	⋮	⋮	⋮

テンプレートT_A

```
class %class_name : public %part_name {
   MACRO1(%class_name);
public:
   %public_attribute
   %public_operation
protected:
   %protected_attribute
   %protected_operation

   %relation_member_list
};
```

図 7: クラス部品の情報

以上の情報は, 部品名とそれに対応するクラス名, モジュール部品名, テンプレート名を示す入出力情報対応テーブルと, 実際のクラス宣言の構造を示すテンプレートで管理される. 図7に生成する対象言語がC++ である場合の入出力情報対応テーブルとテンプレートの例を示す.

なお, 関係部品はクラスのメンバに展開されるが, クラス部品のテンプレートには, 関係部品を展開して生成されたメンバが埋め込まれる位置の指定があるだけで, 実際のメンバは関係部品の展開により出力される.

関係部品

関係部品は, クラス間関係の実装パターンを示すものである. 関係を実装するためには, 以下の情報が必要である [6].

- 関係管理クラス:
  Set や List 等, 相手のリファレンスを管理する.
- 被管理クラス:
  相手側に対するリファレンスを直接持てない場合 (間接関係 [6]), 関係管理クラスにより管理され, 間接的に相手を示す.
- インクルードすべきファイルとそれに関連する宣言:
  関係管理クラスが定義されているファイル, また間接関係の場合, 被管理クラスから相手を参照する手段を提供するファイル, 及びそれらのファイルを利用するための宣言が必要になる.

関係管理クラス, 被管理クラスに関しては, これにしたがって関係に対応する属性が宣言される. また単純に属性として宣言するだけでなく, 非公開の属性として宣言し, その属性の取得や設定を行う操作を宣言する場合もある. したがって, これらはまとめてクラスのメンバ宣言へ展開するテンプレートとして与えることができる.

また, 生成する対象言語によっては (C++ 等), 関係の展開によって属性を宣言する際, 相手クラスのクラス型を用いるなら, 相手クラスが先に宣言されていなければならない. したがってこうした言語を生成する際には, 関係部品は, クラス宣言の順番に関する情報を持っていなければならない. 実装上は, この情報をフラグとして入出力情報対応テーブルに格納し, 属性宣言として相手クラスのクラス型を用いるならこのフラグを立てておくことにする.

インクルードすべきファイルとそれに関連する宣言は, 必要となるモジュール部品として管理される.

したがって関係部品は，部品名とそれに対応するモジュール名，テンプレート名，必要ならばクラス宣言の順番に関するフラグの対応を示す入出力情報対応テーブルと，メンバ宣言のテンプレートで管理される．図8にC++を生成する場合の関係部品の入出力情報対応テーブルとテンプレートの例を示す．

入出力情報対応テーブル

Name	Module	Template	Flag
R_1	Set	RT_A	Off
R_2	List	RT_B	Off
R_3	Assoc	RT_C	On
⋮	⋮	⋮	⋮

テンプレートRT_A

```
protected:
    Set<%targetClass *> %roleName;
public:
    void get_%roleName();
    void set_%roleName();
```

図 8: 関係部品の情報

このテンプレートにより展開されたクラスのメンバは，クラス部品の展開により生成されるクラス宣言に埋め込まれる．

モジュール部品

モジュール部品はクラス部品の集合であり，ファイルを指す．1つのモジュール部品で複数のファイルを指してもよい．これを展開するためには以下の情報が必要である．

- ファイル名 (複数でも良い)
- ファイルを利用する場合に必要な宣言

以上の情報を，部品名とそれに対応するファイル名，必要な宣言 (を示す文字列) を示す入出力情報対応テーブルで管理する．図9にC++を生成する場合の入出力情報対応テーブルの例を示す．

入出力情報対応テーブル

Name	File	Declaration
M_1	"Task.h"	enum Bool { FALSE,TRUE };
M_2	"String.h","Object.h"	typedef Collectable Object; #define MAXSIZE 64;
⋮	⋮	⋮

図 9: モジュール部品の情報

4 プロトタイプシステム

前節までに述べた方式に基づき，プロトタイプシステムを試作した．

4.1 システム構成

試作したプロトタイプシステムは，拡張オブジェクト図で表現した設計モデルからターゲット環境用のC++のコードを生成するものである．

システムは，図10に示すように，主に次のものから構成される:

1. Xover変換部: 拡張オブジェクト図をXover言語に変換する．
2. ジェネレータ: Xover言語表現された設計モデルと入出力情報対応テーブル，部品展開テンプレートが入力されると，C++コードを出力する．

なお，ジェネレータの実装にはSOFTEX/S[8], [9]を利用した．

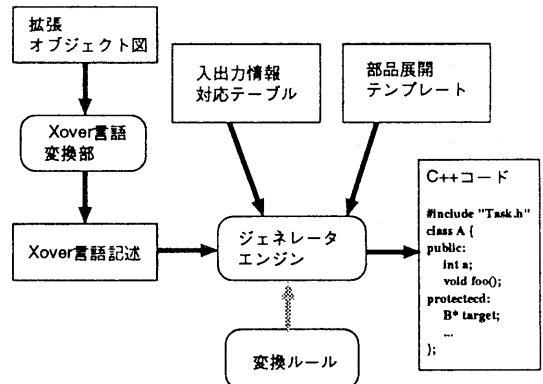

図 10: システム構成図

4.2 例題

今回の実験では，ターゲット環境としてリアルタイムOSを想定し，以下のような部品を用いた設計モデルからの生成を行った:

- クラス部品: 非同期タスク，非同期I/Oタスク，周期監視タスク，非同期周期タスク，割り込みハンドラ
- 関係部品: ポインタ
- モジュール部品: 割り込みハンドラ，タスク，周期タスク，タスクライブラリ

クルーズコントローラの例題[2]を用いて，上記部品を用いて設計したモデルからのコード生成を行っ

た．例題の拡張オブジェクト図は2.2.3で示した図2である．このモデルから生成したプログラムの一部を図11に示す．

```
// CruiseLever.h
#include "InterruptHandler.h"
#include "CruiseTask.h"
enum Bool {FALSE, TRUE};
class CruiseController;
class CruiseLever : public AsyncIOTask {
public:
   DCL_TASKMAIN(CruiseLever);
   CruiseLever(TCB * tcb);
   void main();
private:
   int _position;
public:
   void position( int pos );
protocted:
   CruiseController * CruiseController_;
public:
   void getCruiseController_();
   void setCruiseController_( CruiseController * target);
};
...
```

図 11: 生成プログラム

5 議論

本生成方式では，部品の展開方法を個別に変換ルールに記述するのではなく，展開のための情報を入出力情報対応テーブルとテンプレートに記述することで，部品の展開を実現した．

一般に，変換ルールの記述方法はジェネレータの実装に依存するものであり，ユーザに分かりにくいことが多い．一方，入出力情報対応テーブルは，部品名と部品の持つ情報の対応関係を簡単な表形式にまとめたものであり，テンプレートは出力プログラムのコーディングスタイルをそのまま表現するものであるため，プログラム開発者が入出力情報対応テーブルやテンプレートを記述することは容易にできる．しかも，入出力情報対応テーブルとテンプレートの修正を行えば変換ルールの変更を一切することなしに，部品の展開方法を変えることができる．したがって，本生成方式ではユーザによるカスタマイズが容易に行える．

また，部品を展開した時のコーディングスタイルをそのままテンプレートに表現することができる．したがって，ユーザが得たいコーディングスタイルのコードをより容易に生成することが可能である．

Crossover アプローチの設計モデルで用いる部品は，実装メカニズムとの対応付けを行うものであり，どのような部品を用いるかはユーザ毎に異なる．また部品をどういうコードに対応付けるかも様々な選択肢がある．したがって，部品の展開方法のカスタマイズは重要な問題であり，ユーザによるカスタマイズを容易にする本生成方式は有効性が高いと考える．

6 おわりに

本稿では，部品を用いた設計モデルからのコード生成に際して，ユーザによるカスタマイズが容易な生成方式を提案した．

今後は，より多くの部品を整備した上での実験や，さまざまな言語をターゲットとした実験によって，方式の妥当性や有効性の検証を行いたい．

参考文献

[1] Allen, R. and Garlan, D. : *Formalizing Architecture Connection*, Proc. of the 16th ICSE, (1994).

[2] Gomaa, H : *Software Design Methods for Concurrent and Real-Time Systems*, Addison Wisley Publishing Company, (1993).

[3] 岸知二, 前川佳春, 池田健次郎, 野田夏子: オブジェクト指向開発アプローチ Crossover (1) - 設計モデル -, 情報処理学会第 49 回全国大会 (1994).

[4] 岸知二, 前川佳春, 池田健次郎, 野田夏子: オブジェクト指向開発アプローチ Crossover における設計モデル, 日本ソフトウェア科学会, 第 1 回ソフトウェア工学の基礎ワークショップ, (1994).

[5] 岸知二, 前川佳春, 池田健次郎, 野田夏子: オブジェクト指向設計のためのソフトウェアアーキテクチャモデル, 日本ソフトウェア科学会第 2 回ソフトウェア工学の基礎ワークショップ, (1995).

[6] 野田夏子, 前川佳春, 岸知二: オブジェクト指向開発アプローチ Crossover (2) - 生成系 -, 情報処理学会第 49 回全国大会 (1994).

[7] Rumbaugh, J., Blaha,M., Premerlani, W., Eddy,F. and Lorensen, W. : *Object-Oriented Modeling and Design*, Prentice-Hall, (1991).

[8] Sato, A., Tomobe, M., Yamanouchi, T., Watanabe, M. and Hijikata, M.: *Domain-Oriented Software Process Re-engineering with Software Synthesis Shell SOFTEX/S*, 10th Knowledge-Based Software Engineering Conference, pp. 97-104, (1992).

[9] Yamanouchi, T., Sato, A., Tomobe, M., Takeuchi, H., Takamura, J. and Watanabe, M.: *Software Synthesis Shell SOFTEX/S*, 7th Knowledge-Based Software Engineering Conference, pp. 28-37, (1992).