ソフトウェア基礎論　38－4
プログラミング言語　28－4
（1991. 3. 8）

オブジェクト指向言語Koolaにおけるクラス定義

渡守武　和記　　西山　保
松下電器産業(株)　半導体研究センター

オブジェクト指向言語Koolaにおける3種類のクラス定義について、オブジェクト指向的なC言語インターフェースという観点から説明する。C言語をベースにしたオブジェクト指向言語のプログラムで、C言語で書かれた既存のモジュールの関数や基本データタイプに関するライブラリ関数の関数コールのようなC言語とのインターフェースに関する部分はオブジェクト指向的ではない。Koolaには、このようなオブジェクト指向的でない部分をオブジェクト指向的に記述するための3種類のクラス定義が備わっている。これらを利用することによって、アプリケーションプログラムではプログラム全体の完全なオブジェクト指向記述を実現できる。

Class Definitions for the C-Language Interface in the Object-Oriented Language Koola

Kazunori TOMOTAKE　　Tamotsu NISHIYAMA

Semiconductor Research Center
Matsushita Electric Industrial Co., Ltd.

3-15, Yagumo-Nakamachi, Moriguchi, Osaka 570 Japan

This paper describes the functionality and syntax of the three types of class definitions in the object-oriented language Koola from the point of view of an object-oriented interface to the C language. In an object-oriented program written in a C-based object-oriented language, its interface parts to C such as function calls of libraries or modules are not object-oriented expressions. Koola provides three types of class definitions that make it possible to describe such non-object-oriented parts in object-oriented style. By using these class definitions, application programs can be written in comlete object-oriented style.

1 はじめに

新しい言語を開発する場合、まず考慮しなければならないのは、従来から使用している言語とのインターフェースの問題である。モジュールやライブラリという形で蓄積してきたソフトウェア資産を、新しい言語から自由に使用できなければいけない。また従来の言語自体の機能も利用できた方がよい。しかし逆に、従来言語とのインターフェースをとることによって、新しい言語の長所が損なわれるようなことがあってもならない。

オブジェクト指向プログラミングは多くの長所をもったプログラミングスタイルである。特に、C言語をベースとしたオブジェクト指向言語は、C言語とのインターフェースを簡単にとることができるという点で、最近非常に重要性が増している。

Koola (Kernel Object Oriented Language)[1)2)]はC言語をベースとしたオブジェクト指向言語であるが、Koolaでは独自の3種類のクラス定義を導入し、C言語とのインターフェースにオブジェクト指向的アプローチを行って、プログラム全体のオブジェクト指向性を向上させている。今回は、そのオブジェクト指向的なC言語とのインターフェースという観点から、Koolaにおけるこれら3種類のクラス定義について説明する。

2. オブジェクト指向的なC言語とのインターフェースについて

2.1 オブジェクト指向プログラミングにおける記述の重要性

プログラミング言語を評価する場合、機能と記述という2つの面でとらえることができる。機能面とは、そのプログラミング言語が採るプログラミングの概念のどれが実現できているかということであり、記述面とは、その機能をどう表現するのかということである。

オブジェクト指向言語おいては、記述面は特に重要である。記述方法つまりシンタックスが、オブジェクト指向の概念をうまく表現できているかということが、その言語の記述性·可読性を左右する。オブジェクト指向の概念の把握が容易であるか、オブジェクト指向に見えるか、オブジェクト指向らしいか、そして書き易く読み易いかということが記述面で重要である。これらの点が満足されるとき、その記述方法はオブジェクト指向的であるといえる。

例えば構造化プログラミングでは、「while」という単語を使用した記述が重要である。もちろんwhile文の機能はif文とgoto文とで記述できるのではあるが、そうではなく「while」という単語を使っているというところに、記述面での構造化プログラミンが成立しているといえる。

オブジェクト指向プログラミングでも同じであり、オブジェクト指向的な記述をして初めて、オブジェクト指向のプログラムと言える。具体的には、オブジェクトがどれかすぐに分かるか、メッセージを送っているように見えるかということが左右される、クラス定義·メソッド定義·メッセージ式などのシンタックスが重要になってくる。

2.2 C言語とのインターフェースにおけるオブジェクト指向的記述の重要性

C言語をベースとしたオブジェクト指向言語におけるC言語とのインターフェースには、次の2つの側面がある。

- ソフトウェア資産の活用
- C言語自体がもつ機能の活用

前者は、モジュールやライブラリという形で蓄積してきたC言語プログラムを使用するということであり、後者は、C言語のもつデータタイプや関数コール·制御文などを使用するということである。

C言語とのインターフェースをとるということは、こういった従来のC言語によるプログラムやプログラミングスタイルをオブジェクト指向の中に組み入れるということである。つまり、オブジェクト指向的でないものをオブジェクト指向の中に混在させるということになる。

もともとC言語ベースのオブジェクト指向言語というのは、C言語の中にオブジェクト指向を取り入れたという性格の強いものである。従って、インターフェースの部分をC言語の記述のままでプログラム中に散在させていてもかまわない。

しかし、一度オブジェクト指向でプログラミングしだすと、逆にオブジェクト指向の中に、いかに従来のものを溶け込ませるかというのが重要になってくる。というのは、一般によく言われて

いるように、オブジェクト指向でプログラミングする時、従来のプログラミングスタイルから完全に頭を切り替えて、オブジェクト指向的な思考になっていなければ、良いオブジェクト指向のプログラムは作成できないからである。

これには、C言語とのインターフェース部分もオブジェクト指向的に記述できた方が良いことになる。つまりインターフェースの部分を含めて全体をオブジェクト指向的記述に統一した方が、オブジェクト指向的な思考ですべてを考えることができ、プログラムも書き易く読み易くなる。

2.3 KoolaにおけるオブジェクトC言語インターフェース

Koolaは、C言語ベースのオブジェクト指向言語である。式や制御文などのシンタックスはC言語のままで、クラス定義やメッセージなどのオブジェクト指向に関する部分のシンタックスをSmalltalk-80[4]にならっている。Smalltalk-80にならったのは、Smalltalk-80の記述方法が、オブジェクト指向を表現するのに適していると考えたからである。

Koolaでは、オブジェクト指向的なC言語とのインターフェースをとるために、いわゆるオブジェクト指向の機能を実現するクラスとは別のクラス定義を導入した。これにより、ソフトウェア資産の活用はもちろんのこと、C言語自体のもつ機能の活用においても、基本データタイプと構造体とに関する処理をオブジェクト指向的に記述できるようになっている。

3. Koolaにおけるクラス定義の概要

3.1 クラス定義の分類

Koolaにおけるクラス定義は機能的に図1に示すような種類に分類される。

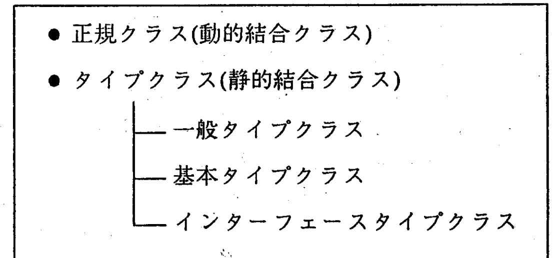

図1 Koolaにおけるクラス定義の分類

一般にメッセージとメソッドとのバインディングには、動的結合と静的結合とが存在する。オブジェクト指向の本質の一つは動的結合にあるが、場合によっては、効率の良い静的結合も必要になってくる。

Koolaでは、メッセージとメソッドとを静的に結合するクラスを設け、それを「タイプクラス」と呼んでいる。それに対し、通常の動的に結合するクラスを「正規クラス」または単に「クラス」と呼んでいる。

タイプクラスは、定義の方法や機能により、さらに細かく3つの種類に分類されている。それらを「一般タイプクラス」・「基本タイプクラス」・「インターフェースタイプクラス」と呼ぶ。

Koolaでは、これら3種類のクラスが、C言語とのインターフェースとしての役割を果たす。

表1 各クラス定義の特徴

特徴	正規クラス	タイプクラス		
		一般タイプクラス	基本タイプクラス	インターフェースタイプクラス
バインディング	動的	静的		
メソッドの継承	する	する		
メソッドのマクロ定義	不可	可		
インスタンス変数	有り・無し	有り	無し	無し
インスタンス変数の継承	する	する	−	−
インスタンスの大きさ	変数＋1ワード	変数だけ	−	−
用途	一般のオブジェクト	構造体のオブジェクト化	基本データタイプのオブジェクト化(システムに装備)	既存モジュールのオブジェクト化(ユーザが定義)

3.2 各クラス定義の特徴

表1にこれらのクラス定義の特徴を示す。

タイプクラスが正規クラスと機能的に大きく異なるのは、メソッドをマクロ定義できることである。マクロ定義とは、メソッドの処理内容をインラインで展開してしまうことであるが、これが可能なのは、タイプクラスのメッセージ・メソッド結合が静的であることに所以する。

C言語とのインターフェースにおける各タイプクラスの役割は表1の用途の欄に記されているが、これらを、C言語とのインターフェースにおける2つの側面から見ると、表2のようになる。

表2 C言語とのインターフェースにおける役割

側面	タイプクラス		
	一般	基本	インターフェース
ソフトウェア資産の活用		○	○
C言語自体がもつ機能の活用	○	○	

3.3 クラス定義のシンタックス

ここで、Koolaにおけるクラス定義のシンタックスについて簡単に説明する。

正規クラスの定義例を図2に示す。後に示す各タイプクラスの定義も基本的にはこれとほとんど変わらない。

クラス定義全体のフォーマットは、Smalltalk-80に関する言語解説書4)にあるフォーマットに準拠している。このフォーマットは、クラス定義を1つのファイルにまとめ、記述のし易さ、読み易さ、および管理のし易さという点で非常に優れている。

クラス定義はClassNameからEndOfClassの11個のキーワードで10個のセクションに分かれている。これらの内、必要なセクションだけを記述すれば良い。なお、ReferenceClassesセクションで宣言されている名前は、このクラス定義で使用している他のクラスの名前である。

メッセージ式のシンタックスもSmalltalk-80と同じで、オブジェクトの後にセレクタとパラメータとを並べる方式である。但し、全体を[ ]で囲むようになっている。この記述方法は、オブジェクトが一目瞭然であり、また正にメッセージを送っているように見えるという点で、記述性・可読性に優れている。

メソッドの定義は、C言語の関数定義とほとんど同じである。ただ、関数名の代わりに、メッセージのパターンをセレクタと仮引数とで記述するところが異なっているだけである。

このようにKoolaのシンタックスは、C言語の書き易さとSmalltalk-80の分かり易さとを取り入れた形になっているが、どちらかというと、C言語よりもオブジェクト指向の色彩の方が強いと言える。

```
ClassName
    List
Superclass
    Collection
ReferenceClasses
    Object
    Instance
    Int
    Memory
GlobalVariables
ClassVariables
    typedef struct  Cell     Cell;
    struct  Cell    {
            Cell        *next;
            Object      *object;
    };
InstanceVariables
    Cell            *firstCell;
    Cell            *lastCell;
    Int             numberOfCells;
ClassMethods
InstanceMethods
    Instance        [ add: object ]
    Object          *object;
    {
    Cell            *cell;

        cell = [ Memory new: sizeof(Cell) ];
        cell->object = object;
        if ( lastCell )
            lastCell = lastCell->next = cell;
        else
            firstCell = lastCell = cell;
        numberOfCells++;
        return self;
    }

    Instance        [ includes: object ]
    Object          *object;
    {
    Cell            *cell;

        for ( cell=firstCell; cell; cell=cell->next )
            if ( [ object is: cell->object ] )
                return cell->object;
        return NULL;
    }
PrivateMethods
PrivateFunctions
EndOfClass
```

図2 正規クラスの定義例

4 各タイプクラスの機能とクラス定義

4.1 一般タイプクラス

4.1.1 機能

一般タイプクラスは、C言語の構造体をオブジェクト化するためのクラスである。

一般タイプクラスは、インスタンス変数を持つ点で正規クラスと同じである。しかしメッセージとメソッドとが静的に結合されるため、インスタンスの構造は非常に簡単になっている。図3に、正規クラスと一般タイプクラスのインタタンスの構造比較を示す。

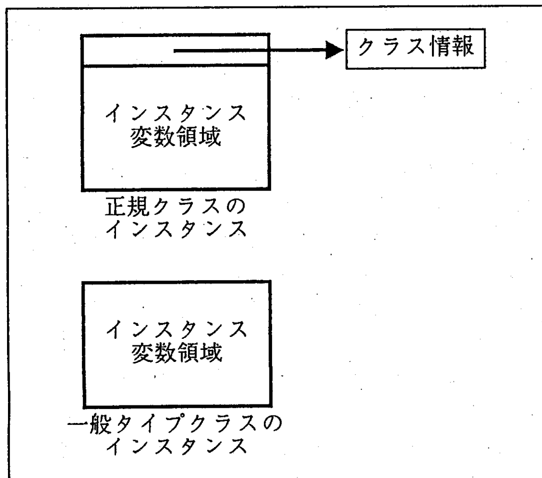

図3 インスタンスの構造比較

一般タイプクラスのインスタンスは、インスタンス変数をメンバとする構造体と同じと考えて良い。構造体と全く同じ使い方も可能である。

またメソッドを構造体の処理関数と見れば、一般タイプクラスはC言語におけるモジュールとよく似ている。しかし、インスタンス変数やメソッドの継承がある点で大きく異なっている。

一般タイプクラスのインスタンスは、図3に示したように、インスタンス変数として宣言された変数の領域だけから成るので、正規クラスよりも使用メモリの点で効率が良い。従って、動的結合を必要としないオブジェクトは、正規クラスではなく、一般タイプクラスで実現するのが普通である。

4.1.2 クラス定義

一般タイプクラスの定義フォーマットは、正規クラスとほとんど同じである。異なる点は、最初のキーワードがTypeClassNameであることと、ルートクラスがVoidであることだけである。

図4に、図2で示したクラスListの中で定義されていた構造体Cellを一般タイプクラスで定義した例を示す。

この例でも示したが、タイプクラスにおいてメソッドをマクロ定義するには、各メソッド定義の前に「#macro」を記述するだけで良い。メソッドがマクロ定義されていると、それに対応するメッセージ式はインラインで展開され、メソッドの呼出しにはならないため、実行スピード

```
TypeClassName
    Cell
Superclass
    Void
ReferenceClasses
    Object
    Instance
GlobalVariables
ClassVariables
InstanceVariables
    Cell            *nextCell;
    Object          *dataObject;
ClassMethods
    Instance        [ new: object ]
    Object          *object;
    {
    Cell            *instance;

        instance = [ super new ];
        [ instance object: object ];
        return instance;
    }
InstanceMethods
    #macro
    Instance        [ object: object ]
    Object          *object;
    {
        return (dataObject = object);
    }

    #macro
    Instance        [ next: cell ]
    Cell            *cell;
    {
        return (nextCell = cell);
    }

    #macro
    Instance        [ object ]
    {
        return dataObject;
    }

    #macro
    Instance        [ next ]
    {
        return nextCell;
    }
PrivateMethods
PrivateFunctions
EndOfClass
```

図4 一般タイプクラスの定義例

が速くなる。ただしC言語などと同じく、あまり複雑な処理のメソッドをマクロ定義しても効果は薄い。

図4のタイプクラスCellを用いて、図2で示したクラスListを記述し直したのが図5である。図2では構造体のメンバ参照であったのが、図5ではメッセージ式に置き換わっている。ただしこの例の場合、これらのメッセージに対応するメソッドがマクロ定義されているので、構造体のメンバ参照と全く同じ効率で実行されることになる。従って図5のクラス定義は、図2のクラス定義と全く同じ効率で且つよりオブジェクト指向的なクラス定義であると言える。

```
ClassName
    List
Superclass
    Collection
ReferenceClasses
    Object
    Instance
    Int
    Cell
GlobalVariables
ClassVariables
InstanceVariables
    Cell                *firstCell;
    Cell                *lastCell;
    Int                 numberOfCells;
ClassMethods
InstanceMethods
    Instance            [ add: object ]
    Object              *object;
    {
    Cell                *cell;

        cell = [ Cell new: object ];
        if ( lastCell )
            lastCell = [ lastCell next: cell ];
        else
            firstCell = lastCell = cell;
        numberOfCells++;
        return self;
    }

    Instance            [ includes: object ]
    Object              *object;
    {
    Cell                *cell;

        for ( cell=firstCell; cell; cell=[cell next] )
            if ( [ object is: [ cell object ] ] )
                return [ cell object ];
        return NULL;
    }
PrivateMethods
PrivateFunctions
EndOfClass
```

図5 図4のタイプクラスCellを用いて定義したListクラスの定義例

以上のように、一般タイプクラスを利用すると、使用メモリ·実行速度の両面で全く効率を落とさずC言語の構造体をクラスにすることができ、プログラム全体のオブジェクト指向性を無理なく向上させることが可能である。

4.2 基本タイプクラス

4.2.1 機能

基本タイプクラスは、C言語の基本データタイプをオブジェクト化するためのクラスである。

C言語ベースのオブジェクト指向言語でオブジェクト指向性を高めようとする時に問題になるのは、C言語の基本データタイプのデータをどう扱うか、つまり、基本データタイプのデータが主体である処理をどう記述するかということである。

四則演算のように「+」や「−」などの記号で式として記述できるものは、そのまま記号を用いて記述した方が記述性·可読性において優れている。まさにC言語は、その記号的な記述性の高さゆえに広く使用されているとも言え、C言語ベースのオブジェクト指向言語としては、ここまではオブジェクト指向的な記述に置き換える必要はないと思われる。

しかし、記号で記述できない部分、つまり基本データタイプのデータを主たる引数とする関数コールは、そのデータをオブジェクトとするメッセージセンディングの記述にしたほうが、プログラム全体のオブジェクト指向性が向上する。

特にC言語には、文字列処理の関数や算術関数などのような、基本データタイプのデータを処理する関数がライブラリとして多数用意されている。C言語をベースとしてプログラミングする以上、これらの関数を使用しないわけにはいかない。従って、これらの関数の呼出しを、基本データタイプのデータをそのままオブジェクトとしたメッセージ式として記述することで、プログラムのオブジェクト指向性を高めることができる。

Koolaには、このような関数コールをメッセージ化する、言い換えれば、基本データタイプのデータをオブジェクト化するために、基本タイプクラスが存在する。基本タイプクラスは、基本データタイプをクラス化したものである。

基本タイプクラスとしては、C言語にあるすべての基本データタイプに対応するものが、処理系

の標準ライブラリとして用意されている。(もちろんユーザが独自にクラス定義することも可能である。)

このライブラリのクラス階層を図6に示す。クラスの名前は、基本データタイプの最初の文字を大文字にしたような形になっている。ユーザは、これらの基本タイプクラスを、C言語の基本データタイプの代わりにそのまま使用することができる。

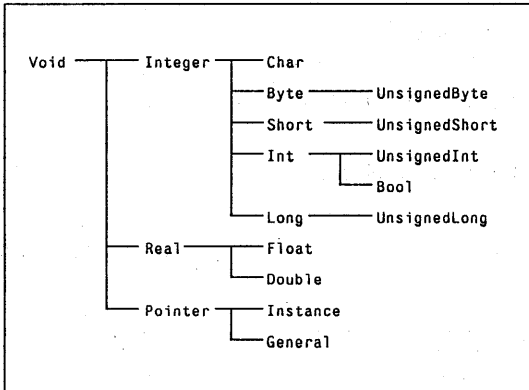

図6 基本タイプクラスのクラス階層

クラス階層において、クラスVoidは、全てのタイプクラスのルートクラスである。クラスInteger・Real・Pointerは抽象クラスであり、メソッド定義の階層化のために設けられている。クラスInstanceとGeneralとは、可読性を高めるために新たに導入されたクラスであり、どちらもC言語におけるvoid *と同じ機能である。Instanceは、インスタンスへのポインタを返すメソッドの型として、Generalは汎用のデータポインタ型として使用する。

4.2.2 クラス定義

基本タイプクラスの定義フォーマットは、一般タイプクラスとほとんど同じである。異なるのは、InstanceVariablesセクションに、基本データタイプとクラス名とを用いてtypedef文を記述するところである。

図7に、基本タイプクラスの1つであるCharクラスの定義例を示す。

この例で示されているように、基本タイプクラスでは、C言語のライブラリ関数をメッセージに置き換えるようにメソッドが定義されているものが多い。それらは一般にマクロ定義され、直接関数をコールするのと同じ効率で実行されるようになっている。

また、C言語のライブラリには存在しない便利な機能も多数定義されている。例えば図7のメソッド[save]は、文字列を別に確保したメモリに保存するメソッドである。

この基本タイプクラスCharを使用したプログラムの例を図8に示す。そして比較のために、Charを使用しないで、C言語の関数を直接使用した場合のプログラム記述例を図9に示す。

この例を見て明らかなように、基本タイプクラスを使用すると、基本データタイプに関する関数コールがすべてメッセージ式でされ、プログラム全体がオブジェクト指向記述で統一される。

```
TypeClassName
    Char
Superclass
    Integer
ReferenceClasses
    Int
    Bool
    Memory
GlobalVariables
    #include <string.h>
ClassVariables
InstanceVariables
    typedef char      Char;
ClassMethods
InstanceMethods
    #macro
    Bool              [ is: string ]
    Char              *string;
    {
        return (strcmp( self, string ) == 0);
    }

    #macro
    Int               [ length ]
    {
        return strlen( self );
    }

    #macro
    Char              *[ copyFrom: string ]
    Char              *string;
    {
        return        strcpy( self, string );
    }

    Char              *[ save ]
    {
    Char              *s;

        s = [ Memory new: [ self length ] + 1 ];
        return [ s copyFrom: self ];
    }
PrivateMethods
PrivateFunctions
EndOfClass
```
図7 基本タイプクラスの定義例

```
TextStream      *input;
SetOfString     *words;
Char            *word;

    while ( word = [ input string ] ) {
        if ( ! [ words includes: word ] )
            [ words add: [ word save ] ];
    }
```

図8 Charを使用したプログラム例

```
TextStream      *input;
SetOfString     *words;
char            *word, *save;

    while ( word = [ input string ] ) {
        if ( ! [ words includes: word ] ) {
            save = malloc( strlen( word ) + 1 );
            [ words add: strcpy( save, word ) ];
        }
    }
```

図9 Charを使用しない場合のプログラム記述例

4.3 インターフェースタイプクラス

4.3.1 機能

インターフェースタイプクラスは、既に存在するC言語のモジュールとインターフェースをとる時に使用するクラス定義である。

一般にC言語でのプログラム資産は、構造体とその処理関数群とからなるモジュールとして蓄積される。例えばX-WindowのXlibや各種ツールキット、もっと卑近な例としては、C言語に標準にある構造体FILEと関数fopen()やfprintf()などからなるストリーム入出力のためのモジュールがある。

C言語ベースのオブジェクト指向言語でプログラムを記述しているとき、これらの既に存在するモジュールを使用しなければならない場合、一般には次の2通りの方法が考えられる。

(1) インターフェースのためのクラスを1つ定義し、そのクラス内においてのみ直接それらの構造体·関数を使用するが、プログラムの他の部分では、そのインターフェースのクラスを使用する。

(2) プログラム全体において、直接それらの構造体·関数を使用する。

(1)の場合、プログラム全体としてはオブジェクト指向が貫徹されるが、モジュールへのアクセスに、インターフェースとなったクラスのインスタンスが1つ介在することになり、メモリ·速度の両面で効率が悪くなる。

(2)の場合、効率は落ちないが、プログラム全体の記述に、オブジェクト指向記述の部分とそうでない部分とが混在することになり、プログラムの可読性が低下する。

インターフェースタイプクラスは、これら両方式の長所を残し、問題点を解決した第3の方式である。つまり(1)の方法ではあるが、インターフェースのクラスのインスタンスは存在せず、もとの構造体が直接使用され、かつメッセージ式が関数コールに直接置き換わる方式である。従って、メモリ·速度の両面で効率の低下がない。

4.3.2 クラス定義

図10に、FILEとのインターフェースをとるためのインターフェースタイプクラスの定義例を示す。

```
TypeClassName
    File
Superclass
ReferenceClasses
    Int
    Char
    Instance
GlobalVariables
    #include <stdio.h>
ClassVariables
InstanceVariables
    typedef FILE        File;
ClassMethods
    #macro
    Instance            [ open: name mode: mode ]
    Char                *name, *mode;
    {
        return fopen( name, mode );
    }
InstanceMethods
    #macro
    Int                 [ close ]
    {
        return fclose( self );
    }

    #macro
    Int                 [ character ]
    {
        return fgetc( self );
    }

    #macro
    Int                 [ character: c ]
    {
        return fputc( c, self );
    }
PrivateMethods
PrivateFunctions
EndOfClass
```

図10 インターフェースタイプクラスの定義例

インターフェースタイプクラスの定義フォーマットは基本タイプクラスとほとんど同じである。InstanceVariablesセクションに、インターフェースをとりたい構造体名とクラス名とを組み合せたtypedef文を記述する。基本タイプクラスと異なるのは、スーパークラスの指定が普通必要ないという点である。もちろんメソッドの継承をうまく利用できるのであれば、スーパークラスを指定してもよい。

クラス定義では、あと構造体の処理関数を使用したメソッド定義を記述すれば良い。普通これらは、効率が落ちないようにマクロ定義されることになる。

図10に示したクラスFileを使用したプログラムの例を図11に示す。この例では、構造体FILEの代わりにクラス名Fileが使用され、処理がすべてメッセージ式で記述されている。このプログラムの場合、構造体FILEと、処理関数(fopen()など)とを使用して記述するのと、メモリ·速度の両面で全く効率が同じである。

```
File        *input, *output;
Char        *inputFileName, *outputFileName;
Int         c;

    input  = [ File open: inputFileName  mode: "r" ];
    output = [ File open: outputFileName mode: "w" ];
    while ( (c = [ input character ]) != EOF )
        [ output character: c ];
    [ input close ];
    [ output close ];
```

図11 インターフェースタイプクラスFileを使用したファイルコピープログラムの例

以上のように、インターフェースタイプクラスを利用すると、既存のモジュールを、全く効率を落とさずにオブジェクト指向の形で使用できるようになる。

5 おわりに

本論文では、Koolaにおける3種類のタイプクラスを、オブジェクト指向的なC言語とのインターフェースという観点から説明した。

これらのクラスの機能は、簡単には、関数コールの記述を局所化するものであるとも言える。しかも、そのための効率の低下が一切ない。

従って、アプリケーションプログラムのレベルにおいては、これらのクラスをうまく利用することにより、無理なくプログラム全体の完全なオブジェクト指向記述が可能になる。

しかし、これらのクラスは、単なるインターフェースという意味だけでなく、あらゆるものをオブジェクト指向でとらえるという積極的な意味を持っている。特に一般タイプクラスは構造体の代替ではなく、静的結合のオブジェクトと考えて使用するのが正しい方法である。また基本タイプクラスは、基本データタイプをオブジェクトとして考えるためのクラスである。

一般に、オブジェクト指向プログラミングでは大きなオブジェクトに注意を向けがちであるが、Smalltalkと同じように、基本的なデータからオブジェクトとして考えるための環境を、これらのクラスは提供しているのである。

現在KoolaでVLSI設計用のCADツールを開発しているが、基本データタイプや既存のモジュールをすべてオブジェクトとしてとらえ、プログラム全体を完全なオブジェクト指向記述でプログラミングしている。そのため、C言語という枠に全くとらわれずにオブジェクト指向で設計·コーディングができ、オブジェクト指向プログラムとしての完成度が高くなっている。

今後は、様々な分野のアプリケーションにKoolaを活用し、ソフトウェア開発の効率を上げてゆきたいと考えている。

参考文献

1) 渡守武 他: 多彩なメッセージ表現が可能なオブジェクト指向言語Koola, 情報処理学会第38回全国大会, 3P-1.

2) 渡守武 他: オブジェクト指向言語Koola -新しい概念と機能-, 情報処理学会第40回全国大会, 4G-2.

3) 岡﨑 他: オブジェクト指向言語Koola -グラフィックスライブラリ-, 情報処理学会第40回全国大会, 4G-3.

4) Adele Goldberg and David Robson: Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

5) Bjarne Stroustrup: The C++ Programming Language, Addison-Wesley, 1986.