〈報文〉

機械系におけるOOPSの利用に関する調査

岐阜職業能力開発短期大学校　　五　藤　三　樹

Resarch of OOPS for Mechanical Enginnering Departent

Miki GOTŌ

要約　現在、OOPS（Object Oriented Programing System）がパーソナルコンピュータ上におけるプログラミング環境として広く普及しつつある。本研究は機械システム系においてプログラミング教育を行う上でのOOPSの取り組みについて調査研究を行った。その結果、機械システム系に適したOOPSとしては、過去のＣ言語の資産を継承しているという点などを考え、Ｃ＋＋が最も適しているという結果を得た。

Ⅰ　はじめに

コンピュータは便利な道具である。もはや機械制御の分野においても無くてはならない道具となっている。最初は８ビットの単純なCPUをアセンブラでプログラミングを行って簡単な制御を行うところから始まった制御分野におけるコンピュータの利用も、いまや16ビットCPUは当たり前で32ビットCPUを使ったものも珍しくは無くなってきている。64ビットCPUが一般的になるのもそれほど遠い先のことでは無いであろう。このようにハードウェアの進歩は日進月歩であるが、翻ってソフトウェアの進歩を見ると、あまり大幅な進歩を遂げているとは思えない。

本研究では最近のソフトウェア進歩の中で一つの大きな流れであるObject Oriented Programing System（OOPS）について調査研究を行った。

Ⅱ　OOPSへの流れ

８ビットCPUが主流だった頃、もちろんいまでも広く使われているCPUではあるが、プログラミングといえばアセンブラ言語を使ったものが主流であり、コード表を見ながらのハンドアセンブルも珍しくはなかった。当時のCPUの能力を考えれば、それでも大きな問題には無らなかったのである。CPUが16ビット、32ビットと発達してハードウェアの性能があがると、処理させたい仕事量もそれにともなって増えてくる。当然、ソフトウェアは大規模化してくるため、アセンブラ言語ではいろいろな問題が起きてくる。

プログラムを行う道具としてはアセンブラ言語とともにコンパイラがある。この、アセンブラ言語とコンパイラ、どちらもソースプログラムを機械語に翻訳するための道具であるが、この翻訳行程に一つの根本的な違いがある。アセンブラのソースコードと、それから生成されるオブジェクトプログラムとは常に一対一に対応しているということである。したがってプログラマは、常にCPUがどのように動作しているかを、CPUの身になって考えながら、プログラミングを行わなければならないのである。ところでプログラムを作成するのは何らかの目的があって行うのであって、とくに制御技術の分野においては、何らかの制御をおこなうための道具としてコンピュータを使うわけである。故にプログラミングに費やす労力は可能な限り少ない方がよいわけで、プログラミングを行うときにCPUの身になって考えるということは、そういう意味では本来の作業の中では余分な工程といえる。

ここで良いプログラムの条件を上げてみる。いろいろな意見があるが次の５つは一般的にいわれていることである。

1）正当性　仮定通りの動作をおこなう

論文受付日　1993.7.15

2）頑強性　異常な状況に対応できる
3）拡張性　仕様の変更に容易に対応できる
4）再利用性　全体または一部を新しい応用または別の応用に利用できる
5）互換性　いろいろなプログラムを組み合わせて目標を達成できる

さらに次のような条件をあげることもできる。

6）移植性　プログラムを他のコンピュータやＯＳに移し変えられる
7）効　率　そのプログラムが走る環境（ハード、ソフト）を最大限利用できる
8）検証可能性　プログラムが正しく走ることの確認のしやすさ
9）信頼性　そのプログラムにエラーの無いことのわかりやすい尺度
10）有用性　実用的に使えること
11）可読性　ソースコードの読み易さ
12）自然さ　プログラムの構造が、コンピュータで解こうとしている問題の論理に沿っていてること。オブジェクト指向プログラミングの目指すものである。

このような条件をみたすためにはどうしてもアセンブラ言語では問題が大きく、そのためコンパイラが開発された訳である。コンパイラはCPUのことを意識せず、より人間の思考に近いプログラミングを行い、それをコンパイラが機械語に一括「翻訳」するわけで、人間の作業の一部をコンパイラが代行してくれることになる。しかし、プログラムは年をおって拡大していき、それだけでは対応しきれなくなってきた。そのため各種のプログラミング技法が作られていく。その一つが構造化プログラミングである。

構造化プログラミングは、プログラムを一つの塊として見るのではなく、全体を機能分解して各機能をそれぞれソフトウェア的な部品として捉えることから始まる。現実の生産システムにおけるフォードシステムの様に、それぞれのソフトウェア的な部品を規格化することによって、ある目的に対して必要な部品を組み合せてプログラムを作成して行くというものである。この場合、重要なのはある目的に必要な機能を、どのように分解すればもっとも効率的に開発が行われるかを正しく見極めることにある。このように全体から底辺を分けて作って行くプログラミングスタイルをトップダウンと呼ぶ。これに対してプログラミングの細かいところから作り始めて最終的に一つの目的を達成するようなプログラミングスタイルをボトムアップと呼ぶ。

構造化プログラミングには数多くの利点がある。始めが正しく、機械的なミスをしなければ最終的に正しいプログラムが得られることが証明されており、簡単に教えられ、数学的思考に近くプログラムの検証（与えられたプログラムが正しく働くことを数学的に証明すること）の可能性も示唆した。

「構造化」という言葉は、プログラマーが一番最初の段階からプログラムを順次処理、繰返し処理そして条件判断からなる構造化を行うことを示しており、構造とは最終的には言語の制御文で表現される。どのようなプログラムも順次処理、繰返し処理そして条件判断の三つの制御構造だけで記述できることが1965年に証明されている（Bohm-Jacopini の定理）この結果コンパイラの仕様は極めて簡単なもので良い。

このように構造化プログラムの発展は、プログラム手法の大きな飛躍であり、プログラミングの生産性を一桁上げたことは確実である。しかし構造化プログラムにも欠点があり、次のようなものがあげられる。

①プログラマーに画一的な思考を課したこと
②永続性に乏しいことである。

構造化プログラミングの基本はいうまでもないが「構造化」であり、「トップダウン」である。よりよいプログラムを作成するためには、まずオブジェクトをよく理解し、それを構造化して部品に切り分けていく必要がある。問題はこの「トップ」を決定する時期が速いということである。そのため何らかの理由によってトップに変化があったとき、すなわち制御対象に仕様変更があったときなど、精密に作られた構造化プログラムほど構成全体が崩れ易いという問題がある。

また、コンパイラが作られた理由の一つが、より人間に近い思考を可能とするというものだったはずなのに、今度は全体を構造化してIF文、WHILE文に構造化することが必要となり、かえって人間の自然な思考から外れてきている。

そこで考えられたのが新しいプログラミング手法であり、次のとおりである。

(1)　データ構造を認識する
(2)　小さなプログラム（手続き、メソッド）を、しかるべき基本操作すべてについて書く
(3)　それらを組み合せてプログラムを作成する。

この考え方は構造化プログラミングの根幹となるトップダウン手法ではなく、ボトムアップに近い手法である。しかし、これがオブジェクト指向プログラム

の基本となる考え方なのであり、オブジェクト指向プログラミングとは、決して構造化プログラミングの先にあるものでは無い。

システムをボトムアップに、現実のオブジェクトを見据えながら設計すると、プログラマには扱う対象、操作する対象、そして考えるべき具体的な操作が見えてくる。もちろんそこには問題もある。

最終的な実行形式プログラムを考えたとき、同じ機能をするプログラムを作成するのに要する仕事量は、それほど大きくは変わらないはずである。問題は誰がその仕事量をするかであって、アセンブラでプログラムを作成する場合はその多くをプログラマー自らが行ってきた。しかしより進んだオブジェクト指向言語の場合は仕事の大半をコンピュータが行うはずである。これはなにを意味するかといえばより速いCPUと大容量のメモリを必要とするということである。しかし、ハードウェアは着実に進歩しており、そのハードウェアを十分に使いこなすためには、より進んだソフトウェアが必須である。そして最終的にはより良い制御が達成できるはずである。

III　オブジェクト指向プログラム言語の基本概念

コンピュータのプログラムは「データ構造＋アルゴリズム」である。標準的なプログラム言語は、データをアルゴリズムの作用を受ける受動的な実体として考えている。その上、データ構造とアルゴリズムは対になっているものなので、類似したデータ構造が複数あれば、その種類だけアルゴリズムを準備する必要がある。例えばC言語で絶対値を求めるアルゴリズム（関数）はint型データ構造に対してはabs関数、float型に対してはfabs関数と異なったものが要求される。

もし誰かに絶対値を求めて欲しいときは、その人にある数値と絶対値を求めたいという目的を伝えればよい。たとえその数値が整数型であろうが浮動小数点型であろうと問題無いはずである。このように、目的を伝えるだけで結果をえられることがオブジェクト指向プログラムの本質である。また、ある人にコミュニケーションを行う場合にその人の内部構造がどうなっているかを知る必要は基本的にはない。やはり、オブジェクト指向プログラムでも同様で、オブジェクトは情報を隠蔽するので、内部でどのような処理が行われているかは知る必要はない。ここでオブジェクトの基本概念を表した図を示す

すなわちObjectはDataに作用を与えるアルゴリズムにあたるMethodsを含有している。Objectは Messageを受け取ると、それを解釈して必要なMethodsを選び、Dataに対して必要な作用を行い、結果としてMessageを返すのである。OOPSの内部では、このObjectが相互にMessageをやり取りしながら処理を行っているわけである。さらに、このObjectはそれを継承して新しいObjectを作ることができるので、新しいObjectはその親のObjectの特質を継承することができるのである。

以下に、ある言語がオブジェクト指向言語と呼ばれるのに必要な条件を示す。

- 情報隠蔽

  オブジェクトの内部データが外部から直　接アクセスできない。特に指定が無い限り外部からのメッセージによってMethodsが内部データに作用を与える。

- データ抽象化

  オブジェクトが抽象データ型として実現される。つまりあるObjectにMessageを送る場合、送り側はそのMessageの型を意識する必要はない。

- 継承

  Objectは、他のObjectを拡張したものとして定義できる。この場合もとのObjectの属性を同時に受け継ぐことができる。

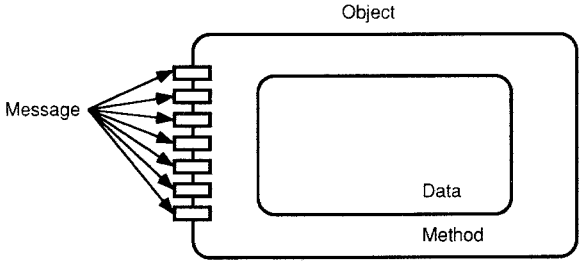

図1　オブジェクトの概念図

Objectはそれ一つで完結しており、従来のようにデータ構造とアルゴリズムが分離しているものより自然なプログラムの作成が可能である。

IV　各種オブジェクト指向言語の調査

現在各種オブジェクト指向言語が使用されている。一口にオブジェクト指向言語と言ってもそのために新たに開発された純粋なオブジェクト指向言語と、従来

からある手続き型言語にオブジェクト指向の拡張を行ったハイブリッド言語の2系統がある。

ここでは4つのハイブリッド言語、C++、Objective-C、Object-Oriented Pascal そして Common Lisp Object System および二つの純粋なオブジェクト指向言語、Smalltalk と Eiffel について調査を行った。以下にその特徴を簡単に述べる。

1．C++

C++はC言語のスーパーセットであり、1980年代初期にAT&Tベル研究所の Bjarne Stroustrup によって書かれた。C言語から多くを継承しているため、現在もっとも一般に普及しているオブジェクト指向言語であると言える。パソコン用には Zortech C++や Borland 社から発売されている Turbo C++、Borland C++などが一般的である。

C++はX3J11 ANSI 規格のC言語をほぼ含有したスーパーセットであるので、その規格のC言語の文法が全て使用でき、また従来のC言語のプログラミングのうえに立って使うことができるのが特徴である。しかし、純粋なオブジェクト指向言語とちがいプログラマ自身で「オブジェクト指向なプログラミング」を心がける必要があるとともに、C++のもう一つの問題は言語仕様がいまだに ANSI に規定されていないことが問題で、コンパイラによって微妙に言語仕様が異なることがある。

2．Objectice-C

Objective-C は Productivity Products International のために Brad Cox によって開発された言語で、1983年にリリースされている。C++と同様に基本となっているのはC言語であり、Objective-C の場合はこれに Smalltalk の思想を加えたハイブリッド言語である。

C++に比べるとあまり広く使われているとは言えないが、NeXT コンピュータの開発言語として知られている。基本的なオブジェクト指向のパラダイムは備えており、多重継承が可能である。

C++に対する Objective-C の大きな利点はそのクラスライブラリと開発環境にある。ICpaks と呼ばれる Objective-C のクラスライブラリは豊富な Graphical User Interfase (GUI) クラスを備えている。また、「Software IC」を実現するための豊富なクラスも用意されている。

3．Object-Oriented Pascal

Object-Oriented Pascal は C++や Objective-C がC言語から派生したように Pascal にオブジェクト指向の拡張を行ったハイブリッド言語である。Apple のプログラミング環境では Object-Oriented Pascal は MacApp を通常同様使用することが可能である。

C++同様、純粋なオブジェクト指向言語に対して、従来の手続き型言語である Pascal のプログラムが実行可能であるという大きな利点をもっている。ただしこの利点が同時に欠点であるということも C++同様である。

4．Smalltalk

純粋なオブジェクト指向言語である Smalltalk の歴史は古く、1970年代初頭、当時ユタ大学の大学院生であった Alan Kay が考案したのが始まりである。その後、Xerox PARC に在籍した時、当初の Smalltalk のアイデアをワークステーション上に実現した。現在、パソコン上で動作する Smalltalk としては Digitalk から Smalltalk/V が IBM-PC、PC98、Macintosh 用にリリースされている。ちなみに/V は virtual のVで5ではない。

現在あるオブジェクト指向言語の多くはこの Smalltalk によって定義されたもので、object、class、method などの名前は全て Smalltalk によって定義されている。

Smalltalk はもはや単なる「言語」ではなく「オブジェクト指向開発環境」といえるシステムで極めて強力な開発環境であるといえる。しかしその分システムに要求されるパフォーマンスも大きく、同じ動きをするプログラムであれば他の手続き型言語に比べ実行速度が遅くなるという問題もある。また、純粋なオブジェクト指向言語であるため、従来のC言語などで行ってきたプログラミングスタイルを全て変える必要があるというのも、今一つ一般に普及していない理由の一つであろう。しかし今後、マシンインディペンデントでかつ複雑な Windows プログラミングのために強力な手段であることは間違いない。

5．Eiffel

Eiffel は Interactive Software Engineering Inc の Bertrand Meyer らによって開発された言語である。

Eiffel の特徴は強力な開発ツールと full typing 機能がそなわっており、しかも多重継承の機能もある。Eiffel は多重継承を実現するために name conflict を

解決する機能を持っている。

Eiffel の開発環境にはクラスライブラリとグラフィッククラスブラウザをもち、エディタが Eiffel の文法を理解している。また、Eiffel の C Package は Eiffel のソースコードから C 言語のソースコードを出力することができるので、C コンパイラがあれば、UNIX ベースのシステムでなくても実行可能プログラムを作成することができる。

C++などと違い、Eiffel は純粋なオブジェクト指向言語である。

6. Common Lisp Object System (CLOS)

CLOS は AI 言語である Common Lisp に対して、オブジェクト指向の拡張を行ったものである。CLOS は1986年、Xerox PARC、Symbolics、Lucid, Inc のグループによって定義され、ANSI 規格 X3J13の Common Lisp にの一部としてコミットされている。

CLOS は二つのオブジェクト指向に拡張された Lisp、Flavors と Common Loops、から派生している。AI プログラマーの間では評価のたかい CLOS であるが、一般的にはあまり使われていない。

CLOS と他のオブジェクト指向言語との大きな違いは、その言語仕様のカスタマイズに対する柔軟生である。言語仕様に対して何らかの操作が行われたとき、CLOS は自動的に自分自身の再定義をおこなう。しかしこの柔軟生は諸刃の剣である。

表1にそれぞれの言語におけるオブジェクト指向コンセプトの実現方法をまとめた。

これら言語の中で、制御技術という分野での使用を考えてみると、まず CLOS は AI プログラマのための言語であり、機械制御に使われることにはあまり向いているとは思えない。現在の制御の分野では C 言語が主流となっており、C 言語のプログラムを作成できる者は制御技術の分野にも多い。この点から考えると、C 言語を基本としているオブジェクト指向言語が望ましくなり、C++か Objective-C が推薦される。この二つを比べると言語としての優劣は双方付け難いが、パソコン上での言語処理系の豊富さという点で、C++の方が制御技術の分野では望ましいと思われる。

コンセプト	C++	Objective-C	Object-Oriented Pascal	Smalltalk	Eiffel	Common Lisp Object System
Object	Class	Object	Object	Object	Object	Instance
Class	Class	Factory	Object type	Class	Class	Class
Method	Member function	Method	Method	Method	Routine	Method
Instance Variables	Member	Instance variable	Object variable	Instance variable	Attribute	Slots
Message	Function call	Message expr	Message	Message	Applying a ro	Generic function
Subclass	Derived class	Subclass	Descendant	Subclass	Descendant	Subclass
Inheritance	Derivation	Inheritance	Inheritance	Inheritance	Inheritance	Inheritance

表1　各言語におけるコンセプトの実現

V　OOPS の実際

実際に C++を用いてクラスを一つ作成した。turtle という名前のこのクラスは、message に従って前進することと進行方向を変えることができる。また移動のときペンを下ろすよう Message を与えることによって自分の軌跡を描くことができる。以下にそのプログラムを示す。

プログラムは3つのファイルから構成され、turtle class の宣言を行う TURTLE. H、class の実現部 TURTLE. CPP それに CLASS の実行を行うための main 部分 TMAIN. CPP である。

最初に TURTLE. H を示す。ここではオブジェクトである class turtle の宣言を行う。クラスの宣言は C 言語の構造体に似ており、変数の宣言方法などは構造体と同じである。しかしクラス内部の private 変数には外部からアクセスできない。これによって情報隠蔽を実現しているわけである。そして public 以下に書かれたメンバ関数によって外部からの Message を受け取っている。

次にクラスの実現部 turtle. cpp を示す。ここでメッセージを解釈実行するメンバ関数の実体が記述される。細かい点を除けば普通の C 言語と大きく変わる所はないが、プロトタイピングが強力になっている。これはデータ抽象化を行うためで、C 言語のように引数

```
// Function of Turtle Graphics
// turtle.h

#define PI  3.141592654

class turtle
{
    private:
        int xPos, yPos;
        double  direct;
        int pen;

    public:
        turtle()    // constructor of turtle
        {
            xPos = 0;
            yPos = 0;
            direct = 0.0;
            pen = 0;
        };
        void    move( double d );   //forward the turtle
        void    moveto( int x_new, int y_new ); //move to turtle to (x,y)
        void    turn( double t );   //turen the turtle direction
        void    penup(void);
        void    pendown(void);
};
```

プログラム1 TURTLE.H

```
// main program of turtle
// tmain.cpp
//
#include <graphics.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include "turtle.h"
main()
{
    turtle  t;
    int i;
    double  len, degree;
    int graphdriver = DETECT, graphmode, errorcode;
    initgraph( &graphdriver, &graphmode, "c:\¥¥tc¥¥bgi");
        errorcode = graphresult();
    if (errorcode != gr0k)  /* エラーが発生したか？ */
    {
       cout << "Graphics error: " << grapherrormsg(errorcode);
       cout << "Press any key to halt:";
       getch();
       exit(1); /* エラーとして終了 */
    }
    t.moveto( (getmaxx() / 2), ( getmaxy() / 2) );
    t.pendown();
    degree = 330.0;
    len = 10;
    for( i = 0 ; i <= 30 ; i++)
    {
        t.move( len );
        t.turn( 120.0 );
        len += 5;
    }
    t.penup();
    for( i = 0 ; i <= 10 ; i++)
    {
        t.move( len );
        t.turn( 120.0 );
        len += 5;
    }
    t.pendown();
    for( i = 0 ; i <= 30 ; i++)
    {
        t.move( len );
        t.turn( 120.0 );
        len += 5;
    }
    getch();
    closegraph();
}
```

プログラム3 TMAIN. CPP

```
//
// Function of Turtle Graphics
// turtle.cpp
//

#include  <math.h>
#include  <graphics.h>
#include  "turtle.h"

void turtle::move( double d)
{
     int xNew, yNew;

     xNew = xPos + (int)( d * cos( direct * PI / 180.0 ));
     yNew = yPos + (int)( d * sin( direct * PI / 180.0 ));

     if( pen )
          line( xPos, yPos, xNew, yNew);
     xPos = xNew;
     yPos = yNew;
};

void turtle::moveto( int xNew, int yNew )
{
     if( pen )
          line( xPos, yPos, xNew, yNew );
     xPos = xNew;
     yPos = yNew;
};

void turtle::turn( double d)
{
     direct = (double)((int)( direct + d ) % 360 );
};

void  turtle::penup(void)
{
    pen = 0;
};

void  turtle::pendown(void)
{
    pen = 1;
}
```

プログラム2 TURTLR. CPP

のチェックを甘くしておくと関数多重定義がなりたたず、その結果としてAbstractionを実現することができなくなる。

最後にこのクラスを使用するための簡単なプログラムを示す。クラス名そのものを使ってturtle型のObjectを宣言する。そして宣言されたObject tに対して

t. move (len)；

とメッセージを与えるわけである。この場合は「距離lenだけ進め」というメッセージで、その時の進行方向などはObject内部に隠蔽されている。

VI 結論

このようにC++を使うことによって、従来のC言語によって行ってきたプログラミングに要する時間を節約することが可能である。

勿論、その分従来のC言語に比べて大きなマシンパワーを要求される。たとえばCPUはインテル製のものであれば80286以上のものを要求され、メモリも640 KBは最低必要条件であり、ハードディスクは当然備えられていることを前提とされている。また生成される実行形式ファイルのサイズも大きく、ROM化に適しているとは言い難い。

しかしながら、現在のハードウェアの開発スピードをみていると、これらの問題も遠からず解決されるものと信じている。何れにしても、目的（Object）を指向したプログラミングの方が従来の手続き型、及び構造化プログラミングより自然な形であることは言うまでもない。

参考文献

(1) Ann L. Winblad, Samuel d. Edwards, David R. King: Object-Oriented Software: - Addison Wesley, 1990

(2) Dusko Savic: Object Oriented Programing with SMALLTALK/V:（小林史典訳 ：オブジェクト指向応用プログラミング：トッパン、1991）

(3) Stephan C.Dewhurst, Kathy T.Stark: PRIGRAMMING IN C++：（小山裕司訳：C++言語入門：アスキー、1990）

(4) 五藤三樹：TURBO C++をチェックする：TURING MACHINE, SPEC, 1991.4