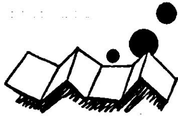
解 説



オブジェクト指向の指向するもの[†]

竹 内 彰 一[††]

1. はじめに

オブジェクト指向の考えが最初に論文に発表されてから，20年近くになる．最近ではその意味するところも十分に理解され，他のプログラミング言語（手続き型，関数型，論理型など）にもオブジェクト指向プログラミングを可能にする言語要素が加えられたり，データベース，オペレーティング・システム，プログラミング・システム，ハードウェア設計，アプリケーション・ソフトウェアの設計などにも大きな影響を与えつつある．

オブジェクトを指向することの利点は

- 頭の中のモデルと相似なモデリングを容易にするクラスという概念
- オブジェクトへメッセージでアクセスするという単一で普遍的メタファ
- データと手続きを組にしたクラス，さらにクラスの階層構造がもたらすモジュラリティ
- オブジェクトによる情報の構造化

などである．これらの利点の普遍性ゆえに計算機科学の他の分野においてオブジェクトを指向する動きが盛んである．

オブジェクト指向は問題へのアプローチの仕方の一つである．それはプログラミング・パラダイム，計算モデルであるだけでなく，人間と計算機との対話のモデルでもあり，また記憶のモデル（計算機の記憶装置のモデルではなく，より一般の情報を整理し，蓄積し，検索する機構のモデル）でもある．Simula, Actor, Smalltalk などのオブジェクト指向の端緒となった各プロジェクトは目標とするゴールはそれぞれ別にありながら，それぞれのゴール達成の過程で新しい普遍的なアプローチを生んだ．それぞれのゴールはオブジェクト指向の利点ほどには良く知られていないが，それと同じくらいに計算機文化に影響を与えるものではないだろうか．

本稿では「オブジェクト指向が指向するもの」について考察する．次章では，オブジェクト指向誕生の契機ともなった Simula, Actor, Smalltalk という三つの言語の提示した動機を言語の概要とともに振り返る．**3.** ではオブジェクト指向をプログラミング・パラダイムや計算モデルとしてだけではなく，視野を拡げて対話のモデル，記憶のモデルとしても眺め，その意味するところについて考察する．

2. オブジェクト指向言語

2.1 Simula

Simula 設計の背景は文献[1]で伺い知ることができる．以下はその要約である．

> 近代科学の発展の重要な側面は人間と機械の複雑に絡み合ったシステムの広範な使用である．たとえば，空港や港における交通管制システム，広域のチケット予約システム，大企業のデータバンクなどにその典型的姿を見ることができる．時代の大きな流れとして，これからますます多くの構成要素をもち，ますます高度な相互作用をもったシステムが増えていくであろう．計算機はこういった分野での問題の解析のための強力なツールであり，生産，輸送，通信システムなどにおいて重要な役割を果たしてきた．これら既存のシステムを理解，調節，制御することができることは，新しいシステムの設計や実現と同様に重要である．このことは新しい計算機言語の需要を生み出す．その言語ではシステムを望みの精度で記述できると同時に，計算機に思いどおりのことをするよう指示することもできる．Simula はこの要求に答える言語である．

Simula はシミュレーション言語であると同時に，プログラミング言語でもあるように設計された．シミュレーション言語としては，実在するシステムや想像上のシステムを望みの精度で記述して，外界からの刺激があったときや逆に孤立して置かれたときにシステ

† On What Object Oriented Programming Orients by Akikazu TAKEUCHI (Central Research Lab., Mitsubishi Electric corp.).

†† 三菱電機(株)中央研究所

ムがどう反応するかを予測したり，想像上のシステムが予測したとおりの動作をするかどうかを調べたり，システムについてのわれわれの理解を改善するために用いられる．一方，プログラミング言語としては，その主要な用途として座席予約システムなどのシステム記述（構築）が考慮されている．シミュレーション言語であると同時にプログラミング言語でもあるという要求は Simula 設計において重要な影響を与えたと設計者らは語っている．

プログラミング言語としての Simula は Algol 60 をモデルとする逐次型の手続き型言語である．Simula を Algol 60 と大きく隔てるものは手続き・ブロックのインスタンスという概念の拡張である．Algol 系の言語においては，手続きとブロックはともに局所的なデータや手続きを備えた一つの計算単位を定義する．プログラム・テキスト中の手続き（ブロック）と実行中の手続き（ブロック）との間には明確な区別がある．すなわち実行中の手続き（ブロック）はテキスト中の手続き（ブロック）のインスタンスである．実行時には複数のインスタンスがネスト構造をなし，計算は最も内側のインスタンスについて行われ，それが終了するとそのインスタンスは消滅し，その外側のインスタンスへと計算は移る．すなわち，Algol 系の言語においては手続きやブロックのインスタンスの寿命はそれを呼び出した文の実行中（インスタンスの実行中）だけである．

Simula はそのインスタンスがそれを呼び出した文の実行を越えて生き残れるようなクラスという言語要素を導入した．クラスの構造は手続きと似ており以下の形式をしている．

```
⟨class declaration⟩ ::=
    class ⟨class identifier⟩
      ⟨formal parameter part⟩;
      ⟨specification part⟩;
      ⟨class body⟩
```

⟨class body⟩ にはクラス内で局所的な変数や手続きなどが定義できる．クラスの例を**図-1** に示す．

クラスのインスタンスは，new という手続きにより生成される．

例：new histogram(A, 7)

Simula では reference 変数という特殊な変数だけにインスタンスを代入できる．reference 変数は静的に宣言する必要があり，そのときにクラス名を明示しなくてはならない．

```
class histogram(X, n); array X; integer n;
  begin integerN; integer array T[0: n];
    procedure tabulate(Y); real Y;
      begin integer i; i: =0;
          while (if i<n then Y<X[i+1] else false)
              do i: =i+1;
          T[i]: =T[i]+1; N: =N+1
      end of tabulate;
    real procedure frequency (i); integer i;
      frequency: =T[i]/N;
    integer i;
    for i: =0 step 1 until n do T[i]: =0; N: =0
  end of histogram;
```

図-1 Simula のクラスの例[2)]

例：ref(histogram) X

reference 変数はクラスのインスタンスが保有する局所変数や局所手続きを参照する手段を提供する．たとえばクラス histogram のインスタンスを X という reference 変数がもっているときにそのインスタンスの N という局所変数を参照するときや tabulate という局所手続きを呼び出すときはそれぞれ

例：$X.N$

X.tabulate(W)

とする．

Simula はまたクラスを階層的に定義するためのクラスの結合 (concatenation) という機構をもっている．今クラス A が次のように定義されているとする．

```
class A(a1, a2, ...);
  ...specification of a's...;
  begin...attributes of A...;
      ...statements of A...
  end;
```

このとき A に新たな処理を付け加えるクラス B の定義は次のように書くことができる．

```
A class B(b1, b2, ...);
  ...specification of b's...;
  begin...attributes of B...;
    ...statements of B...
  end;
```

この定義は以下の定義と等価である．

```
class B(a1, a2, ..., b1, b2, ...);
  ...specification of a's...;
  ...specification of b's...;
  begin...attributes of A...;
    ...attributes of B...;
    ...statements of A...;
    ...statements of B...
```

```
class tree(val); integer val;
  begin ref(tree) left, right;
    procedure insert(x); integer x;
      if x<val then
          begin if left==none then left:- new tree(x)
                                  else left. insert(x)
          end
    else if right==none then right:- new tree(x)
                          else right.insert(x);
    ref (tree) procedure find(x); integer x;
      if x=val then this tree
    else if x<val then
            (if left==none then none
                                else left.find(x))
    else if right==none then none
                          else right.find(x);
end of tree;
```

（a）クラス tree

```
tree class stock item;
  begin integer qoh, price, reorder point;
    Boolean ordered;
    procedure reduce;
        begin if qoh=reorder point & ¬ ordered then
                        issue reorder;
                    qoh: =qoh-1
        end of remove;
  end of stock item;
```

（b）クラス stock item

図-2 Simula のクラス結合[2]

end;

例を**図-2** に示す．図-2（a）はクラス tree を定義している．この tree の各ノードは値を一つ (val) 保持できる．図-2(b)は tree を継承する stock item の定義である．stock item と tree の違いは各ノードが val 以外に qoh, price, reorder point, ordered という4つの値を保持でき，かつ reduce という手続きが追加されているということである．

以上のように Simula はクラス，インスタンス，クラス結合（継承）といったオブジェクト指向の重要な概念を最初に手続き型言語の中に導入した．Simula のクラスの局所変数，局所手続きなどは今日の Smalltalk などのオブジェクト指向言語のインスタンス変数，メソッドにそれぞれ相当している．実際 *X*. tabulate (*W*) という手続き呼び出しの式は *X* というインスタンスに "tabulate *W*" というメッセージを送ると読むことができる．またクラスの本体はインスタンスの初期化ルーチンに相当している．Simula は algol 60 に基づく手続き型言語であったが，しかし，シミュレーションを指向する過程で今日のオブジェクト指向言語の多くの基本概念を確立したといえる．

2.2 Actor

Actor は知識の自然で効率の良い手続き的表現 を研究する MIT の PLANNER プロジェクトの中で，種々の概念を統一するフォーマリズムのための基本概念として誕生した．Actor モデルは言語ではなく，計算モデルである．Hewitt らはそれを新しい普遍的な計算モデルとしてとらえ，データ構造，関数，セマフォ，モニタ，ポート，セマンティック・ネット，論理式，数，デーモン，プロセス，データベースなどはすべて，Actor の特殊形であり，それらに固有の動作はすべて「Actor にメッセージを送る」という単一の動作に還元できると主張した[3]．それを支えるのは制御フローとデータフローの統一と，自律的計算という二つの原理である．

Actor プロジェクトの目的は問題解決のための確固とした基盤を作ることであった．そのための第一の要求は実世界の問題解決知識を効率良く，自然に計算機に埋め込むためのモデルを確立すること，第二は高度な問題解決機構を構築するための柔軟で強力な言語を設計することである．Actor モデルはこの二つの要求の交わるところで生まれた．

Actor モデルでは計算は Actor の集合とその間でのメッセージの交換とにより表される．Actor は概念的にはメッセージを受け取ることにより能動化される手続き的な実体である．Actor はそれ自身の状態をもつこともできる．Actor という名は台本に従い役割を演じる能動的な主体という比喩からきている． Actor が最も強調したのはメッセージ交換による制御フローとデータフローの統一である．

通常手続き型言語では制御とデータは別々の流れを形成する．すなわち，計算は逐次的に進行するのに対し，計算の結果生成されたデータはいったん変数（メモリ）に格納され，後に必要になった時点で取り出されるように，データの流れは制御の流れと独立である．Actor の場合，メッセージ送信だけが唯一制御を他へ移す手段であり，またメッセージの宛先の Actor がその内部状態に蓄えていないデータを外から与えるための唯一の手段である．メッセージ送信はデータとともに制御を他へ移すという点で手続き呼出し，関数適用などとよく似ているが，メッセージ送信は起動された側（メッセージの受信者）が計算終了時に制御とその計算結果を起動した側（送信者）に返還しない片道（単方向）のものである点で大きく異なる．一般に call/return のような制御とデータの往復（双方向）の

```
(factorial≡                                        ; factorial という Actor の定義
  (≡⇒(request: [=n] (reply-to: =c))                ; 受信メッセージの形式記述
                                                     ; n の階乗を求めよ (request)
                                                     ; 答を継続 c へ送れ (reply-to)
    (rules n                                       ; n についての場合分け
      (⇒1                                            ; n=1 のとき
        (c⇐=(reply: 1)))                              ; c へメッセージ (reply: 1) を送る
      (⇒(>1)                                         ; n>1 のとき
        (factorial ⇐=                                 ; factorial に以下のメッセージを送る
          (request: [(n−1)]                            ; n−1 の階乗を求めよ
            (reply-to:                                 ; 答を以下の継続へ送れ
              (≡⇒(reply: =y)                            ; 継続の定義: 受信メッセージの形式
                (c⇐=(reply: (y*n))))))))))              ; c へメッセージ (reply: (y*n)) を送る
```

図-3 PLASMA による階乗の定義[4]

流れは起動する側と起動される側の間に主従関係という非対称な関係を課す．この点メッセージ送信の単方向性は，両者の関係が対等であるため，並列に行われている計算の間の相互作用を自然に記述できる．

Actor モデルではメッセージの到着をイベントと呼び（メッセージの送信をイベントと定義することもある），並列計算をイベントの半順序付き集合として表現することができる．このモデルの中で Actor は自律した能動的計算主体である．計算はそれを遂行する能力をもった，すなわち，必要なデータや手続きをもった Actor に完全に委託することにより行われる．

Actor モデルに基づいて，PLASMA という言語が MIT で設計されている[4]．PLASMA による階乗を計算する Actor の定義を**図-3** に示す．図中で継続 (continuation) と呼んでいるものは計算結果の送り先の Actor のことである．比較のために factorial の LISP による定義を以下に示す．

```
(defun factorial (n)
    (cond ((=n 1) 1)
        (t (* n (factorial (-n 1))))))
```

二つの定義の本質的相違は再帰的に現れた factorial に関する部分である．LISP の場合，(factorial (−n 1)) という式は評価されると $(n-1)!$ の値を返す，すなわちこの式と $(n-1)!$ の値とは等価である (referential transparency)．したがって (*n (factorial…)) により $n*(n-1)!$ が計算できる．一方，PLASMA の場合メッセージ送信は単方向であることに注意が必要である．factorial に $n-1$ の階乗の計算を再帰的に依頼するときに，単に依頼するだけだとその結果に n を乗じて $n!$ を計算することができない．そこでメッセージを送るときに，「値を受け取り，それに n を乗じてその結果を継続 c に送る」という動作をする Actor を定義し（最後の2行），それを継続としてメッセージの中に含めて送る．このようにすると，$n-1$ の階乗を計算した factorial はその答をこの継続 Actor へ送り，継続 Actor はそれに n を乗じて結果をもともとの継続 c へ送り，最後に c は $n!$ を受け取ることができる．

2.3 Smalltalk

Smalltalk は約 15 年にわたるプロジェクトの成果である．プロジェクトのゴールは Ingalls によれば，「すべての人の創造的活動に計算機による支援を与えること」[5] である．プロジェクトの重点的研究テーマは，ビットマップ・ディスプレイの視覚へのインパクト，高度に対話的なグラフィックス，ユーザがプログラムすることによりシステムを変えられるという高度な柔軟性などである．プロジェクトの成果は今日広く知られるところである．ウィンドウ，メニュー（テキスト表示，アイコン表示），スクロール・バーなど，Smalltalk とともに生み出された概念はディスプレイ画面の潜在的可能性を押し拡げるという意味で大きなインパクトを与えた．今日これらはマン・マシン・インタフェースの標準的なものとなり，Smalltalk とは独立に広く使われている．

Smalltalk に関して忘れてならないものは Dynabook プロジェクトである[6]．Dynabook とは当時 XEROX にいた Alan Kay の夢見た「すべての年代の人の情報に関連した行為を支援するノートぐらいの大きさの個人用ダイナミック・メディア」である．このDynabook プロジェクトにおいて，Smalltalk は人間とメディアとの対話言語として開発されたといわれる．すなわち，Smalltalk を生み，育てたプロジェクトは新しい創造的メディアの研究であった．この中で Smalltalk の役割は，頭の中のモデルと計算機の中のモデルとの円滑なインタフェースとして機能する記述言語（プログラミング言語）であること，人間の会話

システムを計算機のそれに適合させるような対話言語(ユーザ・インタフェース)であることの二つであった.

Smalltalk の歴史は 1972 年に BASIC で書かれた Smalltalk-72 のインタプリタから始まる. この時点で Smalltalk は, メッセージによる計算とクラスの概念をもっていた. ただし, クラス自身はオブジェクトでなく, またクラス階層(継承)もなかった. 同年中に Alto に移植され, 以後 Smalltalk はビットマップ・ディスプレイとポインティング・デバイスをもった環境の中で成長していく. これらは Smalltalk に強い影響を与えた. このことは Simula や Actor と好対照をなす. これらのプロジェクトには言語にこれほどまで絡んだデバイスはでてこない. Smalltalk にとってこれらは体の一部であったということもできるが, むしろ Smalltalk もマウスもビットマップ・ディスプレイもすべて創造的メディアという体の一部と考えるべきであろう. この時期の Smalltalk のアプリケーションとして, すでにマウス駆動のエディタ, グラフィックス用の構造エディタ, アニメーション・システム, 音楽システムなどが作られていた. その後, Smalltalk-74 においてセマンティクスの再検討がなされ, クラス自身がオブジェクトに, そしてクラス階層(継承)が導入されたのは, Smalltalk-76 においてからである.

Smalltalk はクラス, インスタンス, クラス結合などの多くの概念を Simula から引き継ぎ, それらをオブジェクトを中心に据えるという観点から見直した. 言語としての Smalltalk の特徴は(1)すべてがオブジェクトである, (2)計算はメッセージ交換により行われる, の2点である.

Actor と同様に Smalltalk のデータ構造, 制御構造を含むすべての言語要素は基本的にオブジェクトとメッセージという概念で構成される. すべてのオブジェクトはなんらかのクラスのインスタンスである. Smalltalk のプログラムはクラス定義の集合である.

class name	Point
instance variable names	*x y*
methods	

```
x: xCoordlnate y: yCoordlnate||
    x←xCoordinate
    y←yCoordinate
x||
   ↑x
y||
   ↑y
    + aPoint|sumX  sumY|

    sumX← x + aPoint x.
    sumY← y + aPoint y.
    ↑Point newX: sumX Y: sumY

    −aPoint|differenceX differenceY|
    differenceX← x − aPoint x.
    differenceY← y − aPoint y.
    ↑Point newX: differenceX Y: differenceY

    *scaleFactor|scaledX scaledY|
    scaledX←x * scaleFactor.
    scaledY←y * scaleFactor.
    ↑Point newX: scaledX Y: scaledY
```

図-4 Smalltalk-80 のクラスの例 (1)[7]

class name	Point
superclass	Object
instance variable names	*x y*
class variable names	*pi*
class messages and methods	

```
instance creation
newX: xValue Y: yValue||
  ↑self new x: xValue
            y: yValue
newRadius: radius Angle: angle||
  ↑self new x: radius * angle sin
            y: radius * angle cos

class initialization
setPI||pi←3.14159
```

instance messages and methods

```
accessing
x: xCoordinate y: yCoordinate||
  x←xCoordinate.
  y←yCoordinate
x||↑x
y||↑y
radius||↑((x*x)+(y*y)) squareRoot
angie||↑(x/y) arctan

arithmetic
+ aPoint||↑Point newX: x + aPoint x
                   Y: y + aPoint y
− aPoint||↑Point newX: x − aPoint x
                   Y: y − aPoint y
* scaleFactor||↑Point newX: x * scaleFactor
                          Y: y * scaleFactor
circleArea |r|
  r← self radius.
  ↑pi*r*r
```

図-5 Smalltalk-80 のクラスの例 (2)[7]

Simula は手続き型言語であり，そのプログラムは手続きの定義やクラスの定義が混在していたが，Actor 同様，Smalltalk はそれをオブジェクトという単一概念で統一的に記述するという点が大きく異なる．

図-4 に Smalltalk のクラス定義の例を示す．Smalltalk のクラスの概念は Simula のクラス概念と基本的に等しい．クラスはインスタンス変数という局所変数およびメソッドという局所手続きをもつ．

すべてがオブジェクトであるからクラス自身もオブジェクトである．するとクラス・オブジェクトは何のインスタンスかということが問題になる．Smalltalk ではそのインスタンスがクラス・オブジェクトであるようなクラスのことをメタクラスと呼ぶ．メタクラスは一つのクラスに付き一つ存在し，その定義はクラス定義中に便宜上埋め込まれる．**図-5** にそのような例を示す．図において "instance messages and methods" とラベル付けられた箱の中味がクラス Point のインスタンスを記述している．そして，"class messages and methods" は Point というクラス・オブジェクトを記述する．言い換えると，"class messages and methods" は Point というクラス・オブジェクトをインスタンスとするようなメタクラスの "instance messages and methods" に等しい．Smalltalk は Simula のクラス結合に相当する概念としてクラス継承をもつ．図の superclass という箱にあるクラスが継承しているクラスを示す．

Smalltalk は逐次型言語であり，計算はすべてメッセージ交換によって行われる．すなわち，オブジェクトにメッセージを送り，受信オブジェクトが計算を行う．メッセージは起動すべきメソッド名とそこで使うデータからなる．データの指定方法としてはキーワード指定など便利な工夫があるが，そういった構文上の柔軟さを除けばその本質は手続き呼出しと同じである．つまり，Smalltalk のメッセージ送信は制御とデータを受信オブジェクトに渡し，受信オブジェクトの計算が終了すると必ず制御はメッセージ送信式の直後に評価すべき式に戻り，メッセージ送信式は計算結果で置き換えられる．このことは Actor のような制御とデータを渡すだけで戻りのないメッセージ送信と大きく異なる．Actor の場合，メッセージ交換による新しい計算モデルを狙ったのに対し，Smalltalk の場合はシミュレーションのようなオブジェクトの対話的操作の面からメッセージ交換という概念を導入したといえる．

3. 計算機からメディアへ

3.1 模倣と創造

Simula はシステムを領域として，シミュレーション言語であり，かつ，システム記述言語であることを目指した．Actor は人工知能を領域として，知識表現言語であり，かつ，知識を操作する問題解決言語のための統一的フォーマリズムであることを目指した．Smalltalk は新しい創造的メディアを領域として，情報や知識を表現する言語であり，かつ，それらを操作する対話言語であることを目指した．これらに共通するのは，実在するもの（物理的実体をもつものや概念としてのみ存在するもの）を計算機の中に表現し，シミュレートする模倣のための言語であること，同時に実在するかどうかにかかわらず，新しい世界（思考や概念的モデル）を表現することの可能な創造のための言語であることである．すなわち，模倣（シミュレーション）と創造（プログラミング）のための言語である．

模倣するという点において，オブジェクトを指向するということは，モデルを構成する基本的概念をプログラミング言語の基本単位であるクラスに自然に写像できるという点できわめて有効である．このような表現は世界を具象的に描き出すという上で便利な思考的枠組みを与えている．実際オブジェクト指向への最大の期待は頭の中のモデルを自然に計算機の中へ埋め込むことができることであり，これは Simula, Actor, Smalltalk に共通する目的でもあった．一方，創造するという点においては，オブジェクト指向プログラミングではおのおののクラスに比喩的な意味付けが容易にできるという意味で便利である．具体的に新しい世界を創造しつつも，個々の構成要素については実在するものとの間になんらかの比喩が存在することは，できあがった世界を人間が解釈・理解する上で親しみやすい．それ以上に比喩は比喩されているものと比喩しているものとの間で主要な性質が共通に存在することを示唆する．したがって比喩されているものが人間の馴染みの深いものであれば，比喩しているものの性質について多くのことを自動的に推測することができる．これにより，創造された世界を対話的に操作するときに，プログラムについて詳細な知識なしに常識だけを用いて多くのことができるようになる．

こうした比喩の強力さを用いることはオブジェクト指向にかかわらず，今ではアイコンやメニューなどヒ

ューマン・インタフェースの主要な技術となっている．しかし，こうした技術の起源はオブジェクト指向の，特に Smalltalk の初期の成果，にさかのぼることができる．実際 Smalltalk プロジェクト自身がヒューマン・インタフェースの開発も含んだ新しいメディアの開発でもあった．

既存のものをまねる模倣と新しいものを自分の考えで作り出す創造とは概念としては対立するものであるが，よくいわれるように人間の創造的思考においては創造と模倣とは互いに拮抗しつつもともに不可欠なものである．三つの言語はそれぞれの領域で人間の創造的思考を計算機を用いて増幅しようというゴールの下で模倣と創造のための言語の枠組みとしてオブジェクトを指向することを発見したといえる．一般化して言えば，オブジェクト指向が指向するものは模倣と創造を両輪として人間の創造的思考を計算機を用いて増幅することであったと言える．

オブジェクト指向は幅広く浸透している．最も尖鋭的にオブジェクト指向が展開しているのは，オブジェクト指向の指向する創造的思考の増幅であろう．これは新しい創造的メディアの研究である．

3.2 計算機からメディアへ

計算機を創造的思考のためのメディアとする見方は Alan Kay の Dynabook 構想にすでにその完全な姿が記述されている．計算機のメディアとしての潜在力を洞察する Kay の文章を引用しよう[6]．

> 「あらゆるメッセージはなんらかの意味である観念のシミュレーションである．それは具象的なものであることもあれば，抽象的なものであることもある．メディアの本質はこれらのメッセージを蓄え，変更し，眺めるその手段にある．計算機は元来算術計算を行うように設計されているが，それ自身をメディアとしてみたとき，詳細に記述されたものをシミュレートするその能力は，メッセージを蓄え，眺める手段を十分に用意すれば他のどんなメディアにでもなれるということを意味する．さらにこの新しい『メタメディア』は質問や実験の要求にも応じることができるという意味で能動的であり，その結果，メッセージは使う者を双方向の会話へと巻き込む．この特性は個人教師というメディアを除いてはいまだかつてなかったものである．われわれはこれの含意するところは深遠であり，人を動かさずにはおかないと考える．」

Kay のいうメッセージは情報や知識を伝達する記号のことである．メッセージは言葉や音楽や絵や匂いなどあらゆる形態をとる．また彼のいうシミュレーションとは観念的なものを視覚や聴覚の下で顕在化させるという意味である．したがって「メッセージは観念のシミュレーションである」とは，メッセージは観念的なものが視覚や聴覚などで認識できる記号へと顕在化させられたものであるということである．シミュレーションに対するこうした深い認識は，観念や思考を表現する道具としての計算機の新しい可能性を切り開いた．それは知識情報処理とも異なる新しい計算機の可能性である．

今日その構想にあったものと機能的にかなり近いものが大きさこそ若干大きいがパーソナル・コンピュータとして現れつつある．また昨今のワードプロセッサ，ゲーム・コンピュータ，パーソナル・コンピュータ，電子手帳など，驚異的普及をしているものはすべてこのメディアとしての方向を指向する計算機である．計算機のメディアとしての潜在力は測り知れなく，またその恩恵に浴するのが多くの人である以上，今後，このような形態の計算機はさらに大きく発展するだろう．これはまさに計算する機械としての計算機から，だれでもが情報・知識を表現し，蓄え，操作し，対話するというメディアへのシフトである．

だれでも使えるメディアの基本技術は，情報・知識の表現の枠組み，蓄積の枠組み，操作・対話の枠組みである．オブジェクト指向はそれらに対してそれぞれ，クラス，オブジェクト単位の記憶管理，メッセージによるアクセスとビットマップという高解像度のディスプレイという解答を提示した．Smalltalk は元来このようなメディアの知識や情報の表現言語であると同時に，対話言語として設計された．またその実現については，オブジェクト単位の記憶管理に十分な注意が払われている．

Dynabook の方向は本や楽器やキャンバスなどの機能を包含した新しいメディアであるが，これは最近のマルチメディア・データベースの研究と共通するものが多い．実際マルチメディア・データベースにおいてもオブジェクト指向的な情報表現が研究されている．ただ両者は蓄積された多様な形態の情報・知識の利用される環境が異なる．マルチメディア・データベースはアプリケーション・ソフトウェアから独立なインタフェースを備えたデータ集合の確立を目的とするデータベースであるのに対し，Dynabook の方向は

蓄積されたものを人間が操作したり眺めたりするための対話的インタフェースを備えた環境をも目的としている．

Kay は Dynabook のことを「能動的メディア (Dynamic Media)」あるいは「対話的メモリ (Interactive Memory)」とも形容している．まさに柔軟な対話性こそがその本質である．Ingalls はこの柔軟な対話性を保証する条件として次の対応原理 (Reactive Principle) を提示している．

> 「ユーザからアクセス可能な要素はユーザがそれを観察したり，操作するときに常に意味のある仕方で自分を提示すべきである．」[5]

オブジェクト，メッセージという基本的枠組み，テキスト，メニュー，アイコンなどのオブジェクトの高解像度ディスプレイへの視覚化，キーボードやマウスなどのアクセス用装置はこの原理をよく支えている．オブジェクト指向は言語やアーキテクチャのレベルまで対話的インタフェースを徹底した最初の試みである．そしてそれはノイマン型計算機を生のままで見てメモリや割り込みを駆使したアルゴリズムを組み立てる時代から，人間の思考に近い形で計算機を利用する，あるいはだれでもが思考の自然な延長として計算機を使えるようにする時代の始まりでもある．

メディアとしての計算機を発展させるためにはオブジェクト指向は当然のことながら他にも多くの要因が必要である．ビジュアル・インタフェース，会話モデル，マルチメディアの提供などがあろう．オブジェクト指向はこうした新しい計算機の利用の仕方—文化—を開拓する上で大きな貢献をした．今，それが現実のものになり，商用機が多く出現するに至ると，あらためてメディアにとってオブジェクト指向という方法論が十分なものかどうかを問い直す必要があろう．それがオブジェクト指向をさらに発展させることになり，またオブジェクト指向に代わる何かの発見につながるかもしれない．

参考文献

1) Birtwistle, G. M. et al.: SIMULA BEGIN, Studentlitteratur (1979).
2) Dahl, O-J. and Hoare, C. A. R.: Hierarchical Program Structures, In Structured Programming, Dahl, O-J., Dijkstra, E. W., Hoare, C. A. R. (eds.), Academic Press, pp. 175-220 (1972).
3) Hewitt, C. et al.: A Universal Modular Actor Formalizm for Artificial Intelligence, Proceedings of IJCAI'73, pp. 235-245 (1973).
4) Hewitt, C.: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, Vol. 8, pp. 323-364 (1977).
5) Ingalls, D.: Design Principles Behind Smalltalk, Byte, pp. 286-298 (Aug. 1981).
6) Kay, A.: Personal Dynamic Media, Computer, pp. 31-41 (Mar. 1977).
7) The XEROX Learning Research Group: The Smalltalk-80 System, Byte, pp. 36-48 (Aug. 1981).

(昭和 62 年 10 月 19 日受付)