ソフトウェア工学 82－5
（1991. 12. 3）

円卓：オブジェクトベースのメソッド独立機能

横山岳浩
ソニー株式会社　コンピューター＆マルチメディア開発本部

概要

システムの処理対象データに加え、入出力装置、周辺装置をもオブジェクトととらえ、そのメソッドをアプリケーションと完全に独立したプロセスに持たせる方式を試みた。アプリケーションは、立ち上げ時に、固有の構成データを渡し、以後の処理を管理系に委ねてしまう。構成データの中にはアプリケーションの処理記述も含まれる。

この結果例えば、処理結果を保存し、ついで同じものを画面に書き出す、という論理的に重複した記述をする必要がなくなる。また、メソッドの保守に際しても、アプリケーションの処理のカスタマイズに際しても、再リンク、再配布を要しない。

Entaku(Round Table) : Exactly-isolated method handler of Object-management system

Yokoyama Takehiro
Sony Corporation, Computer and Multimedia Development Group

Abstract

This paper presents an Object-management system that runs and executes methods as an exactly isolated process from application. In this paper "object" means not only transaction data of the application, also system resources like input-output or peripheral devices. Each application passes its construction data at start-up and entrust to the Object-management system. This construction data include the process description of the application.

This results that application developers need not describe some logically-duplicate description, e.g. saving the result of process and writing the same thing to the display. Also results that they need not re-link or re-distribute the application codes as maintaining the methods or customizing application scripts.

1．はじめに

筆者は[1]において、オブジェクトベースを中心とした業務システム開発支援系を提案した。この中で、従来のデータベースとオブジェクトベースを区別する特徴として、メソッドをアプリケーションの側でなくオブジェクトベース側に付随させる、という点を挙げた。ここでいうメソッドには、入出力装置、周辺装置などの計算機資源をオブジェクトと見て、これを制御するものも含む。

今般、この考えに基づいて、メソッドの管理・実行機能を、完全にアプリケーションと切り離されたプロセスにもたせる、という方式の実験を行ったので報告する。

2．現在流布しているオブジェクト指向データベースの問題点

本稿では、「オブジェクトベース」という言葉を用いて、あえて「オブジェクト指向データベース」と区別している。

それは、現在流布しているオブジェクト指向データベースが、主記憶上に確保した変数に対する操作と同様のプログラム記述で、二次記憶装置上のオブジェクトを操作できる、ということに重点を置いているように見受けられるからである。この場合でも、どの変数をデータベース管理の対象とするかの指示は必要となる。また、プログラムの中でポインターを使って表現しているリンク関係をどうデータベースに格納するか、が問題となる。さまざまなレヴェルのクラスタリングを施す、メモリー上の構造をそのまま二次記憶装置に格納する、などの手段で解決している。いずれにしても、まったく同様の記述という訳にはいかない。

これについては、テラバイト単位の広大なアドレス空間を管理できるＣＰＵを用い、不揮発性の大容量記憶装置をアドレス空間上にマップする技術を考えるほうが、本質的な解決に結び付く。

もう少し具体的にいうと次のようになる。まず、二次記憶装置全体を（仮想）アドレス空間のある領域に固定的にマップする。管理系は、直接特定のアドレスを指定してデータへのアクセスを試みる。このアドレスはメモリー管理機構によって物理アドレスに変換される。物理アドレス上に存在しなければ、一旦物理メモリー上のある領域を確保し、ここに読み込んでから再びアクセスする。

つまり、ページ例外を検出してページインする機構と全く同じことを行うのである。

ポインターは仮想空間上のアドレスで表現されているわけであるから、リンク関係はなんら変換することなくそのままスワップアウトすれば完全に保存される。

近似的には、常に同じアドレスにロードできるファイルと考えれば、実験的に実現できる。

逆にオブジェクトベースは、オブジェクト、つまりデータとメソッドの組を、単一の管理系で管理する、という点に重きを置いている。ここでいうオブジェクトは、二次記憶装置上に保存される処理対象情報ばかりでなく、入出力装置などの計算機資源も含む。そして、これらのオブジェクト間の整合性もここで管理する。

簡単な例を挙げる。現在、ほとんどのアプリケーションは、一連の処理結果を論理的な情報とし

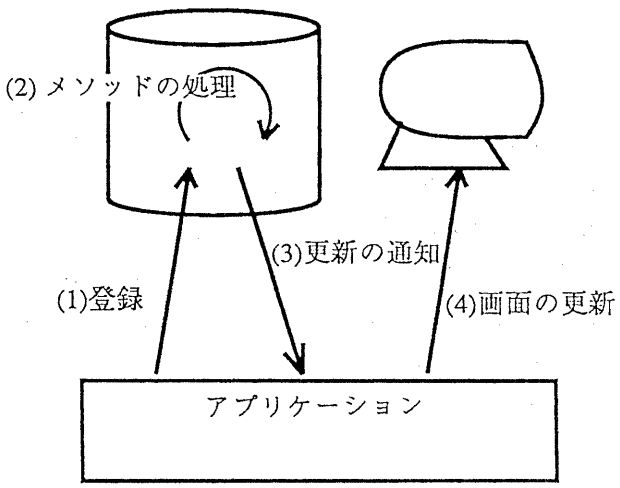

図１　更新処理の重複

て（例えばファイルに）格納し、同時に画面に書き込む。論理的な中身は同じものを、書き出す相手の素性に合わせて幾通りにも書き出さなければならない。

同じ中身ならば、一度の手間だけですべて済んでしまえば、アプリケーションの処理はそれだけ簡単になり、誤りが入り込む余地も少なくなる。

さて、オブジェクト指向データベースは、データの操作をデータベース側に用意されているメソッドで行おうとする。図1のように、アプリケーションは、例えば「登録」メソッドを用いてある情報を登録する(1)。データベースは適当なメソッドを検索し、登録の処理を行う(2)。このメソッドは、単に登録するばかりではない。なんらかの検査の結果、登録をしないかも知れないし、関連する別のデータを処理するかも知れない。そこで、実際にデータが変わったのか、あるいは変わらなかったのかの通知を受け(3)、画面に反映する(4)。

一般にオブジェクト指向データベースは画面の面倒までは見ないから、(3)(4)の処理はアプリケーションの責任である。

例えばGemStoneやO2では、データベースモニタープロセスを独立させている[2]。クライアントごとにセッションを用意し、要求を処理する。しかし、メソッドの実行の結果をアプリケーションに通知する(3)の機能が欠けているので、画面に反映することもできない。結局、メソッドに持たせるべき機能をアプリケーションが持たなければならない。

3．メソッド独立機能の概念

古くから、頻繁に使われる機能をサブルーチンとして独立させ、いろいろなプログラムで利用するということが行われてきた。オブジェクト指向言語を導入すると、クラスの概念、継承の概念を用いて、系統的にプログラムの再利用ができるようになった。

一方、処理対象データをプログラムから独立させ、データベースとして管理することによって、例えばデータ項目の追加が、プログラムと関係なくできるようになった。

しかし、データベースにおいても、格納されているデータの「意味」を規定するのは、それを処理するプログラムの側である。ある数値データを年齢と解釈して、健康保険料を算出するために使うことも、同じデータを価格と解釈して、売上高を計算することもできてしまう。ということは、一つのデータベースを利用する多くのプログラムがあるとき、たった一つのプログラムが誤った更新処理を行っても、その影響は非常に大きなものになりうる。

図2は、従来のデータベースを用いて、同じデータを操作する複数のアプリケーションを動かしている様子である。

例えば、あるデータの妥当性を検査し、更新を行い、関連のある項目を同時に更新する、というようなコードを想定する。プログラム構成を工夫

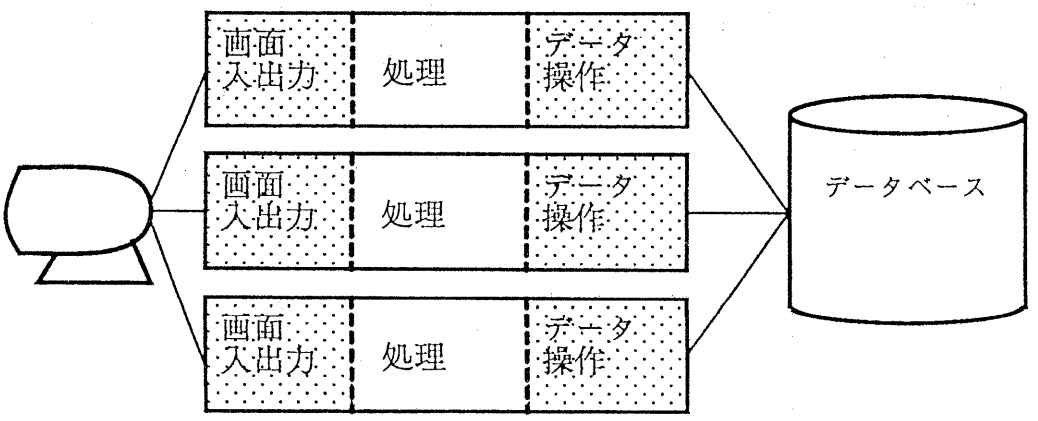

図2　実行イメージの重複

することにより、あるひとまとまりのトランザクション処理をサブルーチン化して、重複開発を避けることは可能である。しかしながら、実行コードを見ると、まったく同じものがいくつもディスクやメモリー中に展開されている。

図には入出力装置を管理するコードも明示してある。ファクシミリ送受信装置のような周辺機器についても同様である。各アプリケーションに、同じコードがいくつも展開される。場合によってはこちらの方が影響が大きいかも知れない。ウィンドウ管理をするコードは相当に大きく、実行コードの肥大化につながる。

これに対し図3は、共通部分は本当に1つにしてしまい、異なる部分だけを、必要に応じて切り替えながら実行する仕組である。

オブジェクトベースは、データの「意味」を規定するための「メソッド」を、プログラムと独立に管理するための機能を重視している。したがって、メソッドの部分がいかなる形にせよ個々のアプリケーションの実行イメージの一部として埋め込まれている、ということがあってはならない。そうではなくて、オブジェクト管理プロセスの中で、メソッドの管理・実行が閉じていなければならない。

上で見たように、入出力装置、周辺装置もオブジェクトと見て管理する。計算機システムの中での情報の流れを考えれば、これらも情報の蓄積庫として捉えることができるからである。

処理対象データを操作するメソッドは、制約条件式のように宣言的に記述できる場合と、処理手順を手続き的に記述したほうが扱いやすい場合とがある。宣言的に記述した場合でも、制約条件式が満たされていないときに、単に更新をとりやめるというのであればよいが、何らかの誤り処理を行うとすれば、手続き的な記述が必要になる。この場合、制約条件式は、メソッドというよりも、処理起動の「契機」として働いていることになる。いずれにしても、その記述はオブジェクト管理系の側に格納・管理し、解釈しなければならない。この機構は、「オブジェクトベース」クラスに属するメソッドとして、オブジェクト管理系に埋め込む。

入出力装置、周辺装置を操作するメソッドについても同じことが成り立つ。

例えば画像データは、単独ではビット列としか解釈のしようがなく、画面に表示したり紙に印刷したりして初めて人にとって意味のあるものとなる。「ディスプレィ」というクラスに画像表示メソッドがあってよいし、更に言えば、オブジェクトベースの「画像」クラスに、ディスプレィ・クラスと共同して表示を行うメソッドがあってよい。画像データの変化は、再表示の契機となりうる。個々のアプリケーションは、画像データの中身を一切見ることなく、表示を行うことができる。

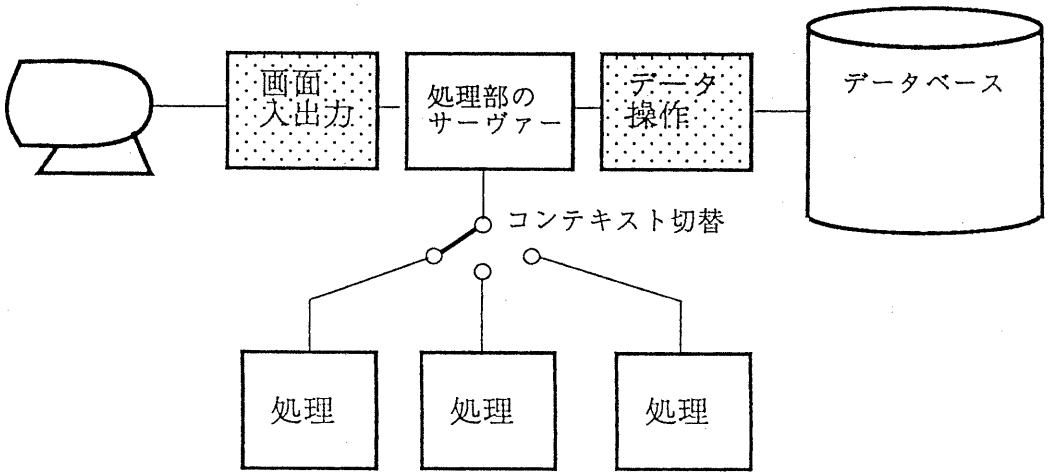

図3　重複の排除

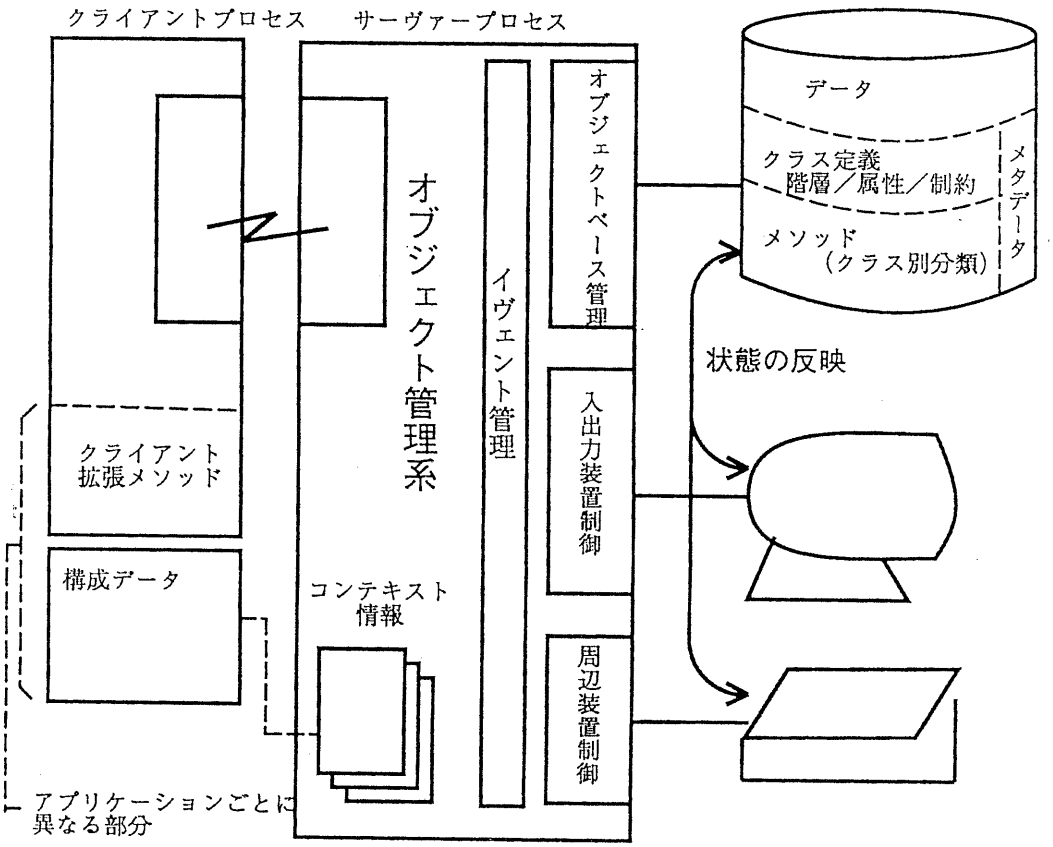

図４　システムの構成

但し、入出力装置、周辺装置の側では、データの論理的な操作は行わないようにした。この点で、オブジェクトベースとそれ以外の装置とでは、扱いに差がある。

例えば、キーボードから入力したデータを、入力装置の側で検査してしまう、ということはしない。データは即座にオブジェクトベースに伝達されるので、こちらの方で検査する。これによって、検査メソッドの重複を防ぐことができる。

４．メソッド独立機能の実現

メソッド独立機能を実現するために、図４のような構成を試みた。

4.1 個々のアプリケーションの構成

個々のアプリケーションは、オブジェクトベー

ス管理系とは独立のプロセスである。管理系から見ると、このプロセスは、計算機資源としてのオブジェクトの一つである。次のような機能を置く。

(a) アプリケーションの立ち上げ時に、アプリケーション固有の構成データを渡す機能。
但し、構成データとしては、
・スクリプト・ファイルの存在場所、
・アプリケーションに固有の制約条件式、
・入出力装置や周辺装置の設定、
・オブジェクトベース操作の権限設定に関する事項
などがある。

(b) オブジェクト管理系に対し、ある機能の起動を要求する機能。
クラス名称、メソッド名称、パラメーターを渡して起動を要求する。

(c) オブジェクト管理系の側で解決できないイヴェントが起こったとき、その旨の通知を受け入れる機能。
イヴェント名称とパラメーターが渡されるので、名称によって適当な処理を選択し、実行する。この中で(b)の機能を使うこともできる。

(a) にいうスクリプトというのは、アプリケーションの処理を、オブジェクト管理系が解釈できる形式で記述したものである。なんらかのイヴェントに対する処理を一つの単位としてファイルに格納してある。そして、一旦スクリプトの存在場所を通知してからは、アプリケーションがこれを操作することはない。
オブジェクトベース管理系から見ると、スクリプトは「アプリケーション」クラスに属するメソッドである。
制約条件式は、指定した契機で評価し、条件が満たされないときに通知する。図1の(3)に相当する。
(b)(c) は、スクリプトで記述できないような処理を実現するための機構である。図4のクライアント拡張メソッドがこれに当たる。クライアントに結合された特殊なメソッドという意味でこのように呼ぶ。逆に、このようなメソッドを必要としない場合、改めてクライアント側のプログラムを用意しなくてもよいことになる。

4.2 オブジェクト管理系の構成

次に、オブジェクト管理系は、各アプリケーションに対しては、サーヴァープロセスとして振る舞う。次のような情報を管理する。

(a) 処理対象情報。
クラスごとに一意的なインスタンス識別番号を与え、消長を管理する。オブジェクト間のリンク関係は、クラス名と識別番号を設定することで行い、特別な機構を用意しない。リンクの張り替えは、識別番号を更新することであり、属性の更新と同じ操作である。
オブジェクトが永続的であるか否かは、クラスの定義による。否永続的なオブジェクトはデータキャッシュ内にのみ存在する。もともとこれはディスクアクセスの効率化を目的としているが、更新があってもディスクに書き出さないので、結果的に否永続オブジェクトとなる。どちらにしても操作を記述する方法は同じである。

(b) クラス定義。
クラス名、属性、階層関係、属性ごとの制約条件などを管理する。

(c) メソッド・ファイル。
メソッドは中間コードにして、クラス別に分類して管理する。共通オブジェクト・ライブラリーの形にする方法もあるが、実行する計算機のアーキテクチャーなどに依存しないようにした。この中間コードのインタープリターが管理系に組み込まれている。

(d) アプリケーションごとのコンテキスト情報。
複数のアプリケーションが同時に動作し得るため、別々にコンテキストを保持し、切り替えながら処理を進める。

そして、基本的にイヴェント駆動の方式で動作する。
オブジェクトの生成・更新・削除はイヴェントの一例である。この場合、クラスに定義されている制約条件式、アプリケーションに固有の制約条

件式を検索し、評価する。

また、入出力装置、周辺装置からの割り込みもイヴェントである。

オブジェクト管理系は、イヴェントが発生したとき、まず当該アプリケーションに属するスクリプトを検索しようとする。検索の基準は、イヴェントの種類と、これを引き起こした主体である。みつからなかったときに限り、アプリケーションに対し、その旨を通知する。

このように、できるだけオブジェクトベース管理系側で処理を遂行し、プロセス間通信が起こらないようにしている。実際、イヴェントに対する処理を全てスクリプトとして記述した場合、通信はアプリケーションの立ち上げ時に一度起こるだけである。

あるデータ(例えばオブジェクトの属性値)が画面に表示されている、という場合、実質としては１つのものを、ディスプレイ制御の都合で別々に扱っている訳である。とすれば、常に値が一致するよう、互いに更新を反映しあうのは、オブジェクト管理系の役割である。

ところで、いくつかのメソッドは、サブルーチンの形でオブジェクト管理系の中に組み込んでしまっている。

画面入出力を含む、入出力装置の制御を行うコードは、ほとんどがサブルーチンの形になっている。たとえ中間コードの形にしても、その実体は画面操作ライブラリーを呼び出すだけのものになってしまい、冗長なだけだからである。同様に、周辺装置の制御を行う部分も、入出力部品の追加という形で組み込む。

5．メソッド独立機能による利点

このような構成により、次のような利点が考えられる。

(1) 保守性。

消費税の導入により売上げ計算の方法が変わった、というような場合に、メソッドの入れ替えで、全てのアプリケーションが新方式に移行できる。

また、新しいディスプレイ装置が追加された場合、これに対応した画像表示メソッド(装置の特徴を生かした高速なものかも知れない)を追加すると、全てのアプリケーションがその恩恵を被る。

従来のデータベースであれば、「税区分」という項目を追加する程度のことは可能であるが、計算式はアプリケーションに組み込まれてしまっており、多くの箇所を変更しなければならない。仮に単一のサブルーチンで実現していたとしても、関係する全てのアプリケーションについて、リンクを取り直し、再配布する必要がある。

(2) 容量。

個々のアプリケーションの実行イメージは、真にそのアプリケーションに固有の処理のみを含むため、全体として、容量を減らすことができる。

例えば、高度なウィンドウ処理を行うプログラムは、記憶装置をいたずらに浪費する傾向があり、効果が大きい。

(3) 処理内容のカスタマイズ。

アプリケーションの構成にもよるが、スクリプトを実行イメージと切り離したことにより、利用者によるカスタマイズを処理内容にまで及ぼすことが可能である。

このとき、あるデータの更新を自動的に画面表示に反映するなど、内部的な整合性を保つ処理を、明示的にスクリプトに記述しなくてもよい。

6．マルチメディア化に伴う問題

業務情報を扱う部署では、情報の重複ということが大きな問題となっている。例えば、出庫というひとつの事実に対応して、経理用、資材用など、複数のデータベースに情報が入力される。入力の手間という点でも問題だが、それ以上に、情報間の整合性が確保できない、という問題がある。月次の累計額を求めた結果、いくつもの異なる値が出てきて、結局手計算で求めたものが公式数値となる、という、ばかげた現象が起こっている。

これに対しては、全社的なデータ辞書を設定し、すくなくとも、論理的に同じ情報である旨の識別をつけられるようにしておこうという試みがある。しかし、ＥＲ(Entity-Relationship)のようなモデルに現実世界を当てはめようとしたとき、モデルがあまりにも融通無碍であるため、具体的に何を「実体」や「関係」ととらえるか、うまく定めることができなかった。「実体」とは何か、と

いった哲学的論議に終始した例もある。結局のところ、何を「実体」ととらえるかを一番よく決定できるのは、データベースを実際に利用する立場の人であって、システム部門ではない、というのが私見である。

情報の重複の問題は、扱う情報がマルチメディア化すれば、更に悪化する。それは、

(1) 各情報が大容量のものとなるため、重複部分が占有する記憶媒体のコストも増大する、

(2) 情報の記憶形式という、新たな情報重複の要因がある

ためである。

(2) についてもう少し述べてみよう。例えば画像情報についていうと、ＥＰＳ(Encapslated PostScript) 形式、TIFF形式その他、いくつもの表現形式があり、アプリケーションによって、扱える形式が違う。中には複数の形式のファイルを読み込めるものもあり、結局同じ内容でありながら格納形式の異なる複数のファイルが併存する。さて、内容を更新する場合、その結果を他の形式のファイルにも反映しておかなければならないはずであるが、なかなか実行が難しい。結局、内容も形式も異なるファイルが多くできてしまい、どれが正しい内容なのかもわからなくなってしまうのである。

これは、形式変換を各アプリケーションが行い、そして、そのアプリケーションに最も相性のよい形式で格納してしまうからである。

「画像検索」「画像保存」というメソッドを持つオブジェクトベースを導入してみる。アプリケーションは、画像の名前と、これをどんな形式で渡して欲しいかを指定して、「画像検索」メソッドを起動する。実際に保管されている形式と異なる場合、変換を実行するのはメソッドの側である。「画像保存」も同様であって、一時的なものを除き、ファイルに直接書き出すことはしない。

この構成は、変換プログラムをそれぞれのアプリケーションが持つ、という重複も排除する。

ただし、変換の結果情報が失われることがある、というのが難点である。

画像データベースを構築しようとする人にとって、関心があるのはその中身だけであって、さまざまな格納形式が併存することは、面倒以外のなにものでもない。変換の方法を隠すばかりでなく、変換が行なわれているということ自体を隠すことができる、オブジェクトベースの方式が有効であろうと考える。

7．結果

もともと、オブジェクトベースを導入しようとしたのは、利用者主導でアプリケーションを開発するための機構としてであった。そのために、

・カスタマイズ可能なスクリプト、

・明示を要しないオブジェクトベース入出力

といった特徴を盛り込むことを試みた。また、今回の話題の外であるが、

・入出力画面の定義を中心とした開発形態

をも取り入れてみた。

今回は、オブジェクトベースの管理部分と入出力装置の管理部分、周辺機器の管理部分を、渾然一体として構成した。これに対し、それぞれが別の機械で実行されることを考慮し、別々のプロセスとしておくべきという考えもある。通信量がどの程度になるかを評価する必要がある。

参考文献

[1] 横山岳浩：「円卓：オブジェクトベースを中心とした業務システム開発支援」、ソフトウェア工学研究報告 No.77

[2] Elisa Bertino, Lorenzo Martino : "Object-Oriented Database Management Systems : Concepts and Issues", IEEE Computer, April 1991.