永続プログラミング言語INADAにおけるビューの実現

有次正義[†] 天野浩文[††] 牧之内顕文[†]

永続オブジェクトは，複数のアプリケーションで共有されるものである．しかし，永続オブジェクトの見方や扱われ方は各アプリケーションで異なる．そのため，各アプリケーションでは必要な情報のみを永続オブジェクトから抽出し，操作することを可能にする機構が必要になる．ビューの機構がそれである．ビューはデータを，さもユーザが要求した形であるかのようにユーザに提供するだけでなく，データの機密保護の機能をも提供するものである．本稿では永続プログラミング言語INADAでのオブジェクト指向ビューの実現に関して論じる．INADAは大量の永続データを扱う応用プログラムを記述するための高機能で拡張性の高い言語である．INADAのデータモデルはC++のそれを拡張したもので，オブジェクトの集合論的検索と一括操作の機能を提供している．INADAでは，ビューはオブジェクトの仮想集合として実現され，ビューの操作は実際に存在するオブジェクト集合操作へと変換される．さらに仮想集合属性の概念を提案する．

Implementation of Views in the Persistent Programming Language INADA

MASAYOSHI ARITSUGI,[†] HIROFUMI AMANO [††] and AKIFUMI MAKINOUCHI [†]

Persistent objects may be shared among many application programs, but they may be handled in different ways. Only needed attributes are selected from persistent objects and processed in each application. View is the mechanism that allows users to deal with data as they like. This paper describes how to implement object-oriented views in INADA. INADA is an object-oriented persistent programming language for developing data-intensive applications. INADA has rich functions and high extensibility. INADA borrows the object model of C++ and extends it to provide the facility of processing queries on sets of objects. In INADA, views are implemented as virtual sets of objects, and manipulation on views are translated into manipulation on sets of actual existing objects. The concept of virtual set attributes is also proposed.

1. まえがき

我々は大量の永続データを扱う複雑な応用の容易な記述を可能とする永続プログラミング言語INADAを研究・開発中である[5)~7)]．INADAはC++[10)]を拡張し，大量の永続オブジェクトを揮発オブジェクトと同様に操作することを可能とした言語である．

INADAでは(1)永続性とクラスは直交である[8)]．永続オブジェクトと揮発オブジェクトは同一のクラスから生成することができる．クラスから生成されるインスタンスオブジェクトは，それが永続・揮発に関わらず，同じデータ構造を持つ．(2)クラスは型(タイプ)である．INADAにおけるクラスはC++のクラスと同じくオブジェクトのコンテナではない．(3)永続・揮発ともに，INADAでは集合オブジェクトを定義・操作できる．これは，ユーザにより定義されたC++クラスまたはシステムが提供している集合クラステンプレートのインスタンスオブジェクトである．この集合オブジェクトがオブジェクトのコンテナとなる．集合オブジェクトのクラスはINADA処理系が正しく集合の要素にアクセスできるように，あらかじめ規定された標準インタフェースを持たなければならない．本稿では'集合'といった場合，オブジェクトの集まりを指す．つまり，'集合'とは計算機科学でのマルチセット[14)]を指す．これは整列されたものでも，されてないものでも良い．集合のこのような性質は，それを定義するユーザ定義のクラスに依存する．(4)オブジェクトの集合指向操作を提供する．(5)任意の永続オブジェクトは複数の型を持ち得る．永続オブジェクトには任意の時点で任意の型を付加・削除できる．このような性質を持ったオブジェクトをマルチタイプオブジェクト

† 九州大学工学部情報工学科
Department of Computer Science and Communication Engineering, Faculty of Engineering, Kyushu University
†† 九州大学大型計算機センター
Computer Center, Kyushu University

と呼ぶ．これにより実世界の実体が持つ動的に変化する性質のモデル化[1),11),18),22)]が可能となる．(6)ビューは仮想集合として実現される．ビューに対する問い合わせは，その基底集合への問い合わせへと変換される．

永続オブジェクトは複数のアプリケーションで共有されるという性質がある．ところが，永続オブジェクトの見方や扱われ方は各アプリケーションで異なる．そのため各アプリケーションでは，永続オブジェクトから応用に必要な情報のみを抽出し，操作することになる．そのため永続プログラミング言語でもビューの機構が必要となる．

データベースにおけるビューの考え方は非常に重要である．特に関係データベース(RDB)におけるそれは情報の別の見方を提供するばかりでなく，機密保護の機構の一つとしても大切な役割を持っている．オブジェクト指向データベース(OODB)[8)]では，これらの役割の他に文献2)で述べられているように，データの再構築や柔軟性の高いモデル化のためにもビューの機構が有用な手段となる．

本稿ではINADAにおける，C++のオブジェクトモデルにそったビューの実現と，INADAが提供しているビューを記述する機能について具体例を示しながら論じる．本稿の構成は以下の通りである．2章ではINADAについて，特に本稿に関連する必要最小限の事項を紹介する．3章ではマルチタイプ機構を集合オブジェクトに適用することによって選択ビューと射影ビューが実現できることについて論じ，4章ではこれらのビューの例を実装して得られた実験結果の評価を行う．5章ではビューに対する操作について考察する．6章では仮想集合属性の概念を提案する．7章で本稿のまとめを述べる．

2. INADAのデータ操作機能

INADAが提供する機能のうち，本稿に関連のあるデータ操作に関する事項に限定して述べる．

2.1 オブジェクトの生成とマルチタイプオブジェクト

INADAにおけるオブジェクトは，C++と同じくクラスに演算子'new'を適用することによって生成される．クラスはC++のそれと全く同じものである*．'new'により生成されたオブジェクトが揮発か永続であるかは，そのオブジェクトがどの領域(ヒープ)に割り付けられたかによる．オブジェクトの存在する領域が通常のヒープ領域であれば揮発，永続ヒープであれば永続となる．ここで永続ヒープとは，ファイルにマップされた仮想空間の部分をいう[24)]．'new'演算子は生成したオブジェクトの識別子(OID)を返す．もしオブジェクトが揮発の場合，そのOIDは従来のC++オブジェクトへのポインタ変数である．生成されるオブジェクトが永続の場合，OIDはシステム定義のポインタ変数，すなわち永続オブジェクトポインタ変数となる．INADAは，C++でいうポインタ変数と永続オブジェクトポインタ変数の大きさを同じにして実装している．永続ヒープをサポートするためINADAでは「永続ヒープファイルテーブル(Persistent Heap File Table：PHFT)」および「永続ヒープ管理オブジェクト(Persistent Heap Management Object：PHMO)」の二つのシステム定義のクラスを提供している．これらを用いてINADAは2次記憶上のファイルをユーザプログラムの仮想空間上へマップする．永続ヒープはPHMOクラスに'new'メッセージを適用することにより(複数個)作成できる．PHMOオブジェクトは(システム定義の)集合オブジェクトであり，後述の集合操作インタフェースを持つ．

オブジェクトは永続オブジェクトである集合に属す時永続になる．一般にタイプTのオブジェクトをある集合の要素として生成するには次のように記述する．

```
objref=new(setref)T(…);
```

ここではsetrefはポインタ変数である．もしsetrefが揮発集合を指すならobjrefの指すオブジェクトは揮発，永続集合を指すなら永続となる．生成されたオブジェクトはその集合の要素となる．オブジェクトが揮発の場合，objrefの定義・宣言はC++のポインタ変数の定義・宣言と全く変わりない．オブジェクトが永続の場合，objrefの定義・宣言の前にキーワードpersistentをつけなければならない．

オブジェクト指向プログラミングは，実世界の実体のモデル化に適しているといわれる．しかし時間的に性質が変化する実体のモデル化は容易ではない[1),11),18),22)]．オブジェクト指向永続プログラミング言語が扱うオブジェクトは長期にわたり存続するため，性質が変化する実体のモデル化に対処しなければならない．そのためには実体の性質に対応するオブジェクトの「型」を動的に変化させることが必要である．INADAは，オブジェクトの永続性をサポートするとともに，その永続オブジェクトの同一性を保ちながら，オブジェクトの持つ型を動的に変化させるための「マルチタイプオブジェクト」機能を提供する．すなわちINADAでは永続オブジェクトの持つ型を任意の時点

* ただし，後述するマルチタイプオブジェクトのためのクラスは，一部に永続オブジェクトであることを前提としたコードを含む場合がある．

で変更することが可能である．つまりINADAでは永続オブジェクトに型を自由に付加，削除することができる．これらの型は互いになんら関係がある必要はない．例えば'objref'によって参照された，型'AType'を持つオブジェクトに対して新たに型'BType'を付加するには次のいずれかの方法による．

(1) AType* objref as BType
=new(pset)BType(*initial_values*);

(2) BType* newobjref
=new(pset)BType(*initial_values*) transforms objref;

ここで'pset'は，ある永続集合オブジェクトを参照する永続オブジェクトポインタである．(1)では'objref'の値は変化しない．(2)で'objref'の値は変数'newobjref'へ代入される．(1)と(2)の違いは，'objref'により参照されるオブジェクトを'BType'により参照する場合に，(1)では必ず'objref as BType'として参照しなければならないのに対し，(2)では'newobjref'によって参照すればよいという点である．'OID as type-name'と記述することによって永続オブジェクトのもつ役割を，ある型を通して変更することができる(詳しくは文献6)を参照)。

2.2 集合オブジェクト

大量の永続オブジェクトを扱う場合，個々のアプリケーションに必要なオブジェクトのみを検索し操作することが重要である．そのために集合論的検索機能が必要とされる．しかし種々の応用に効率的検索を可能とする集合を一義的に実装することは不可能である．このためINADAでは，システム定義の集合を用意するのではなく，検索操作を可能とするために集合が持つべき標準インタフェースを規定する．プログラマは，このインタフェースを持つクラスを定義し，自らの応用に適した実装法で集合オブジェクトを実現することができる．システムの要求する標準インタフェースを満たすクラスを集合クラスと呼ぶ．

満たすべき標準インタフェースのうち，ここでは後述するビューの実現例の中に出てくるもののみ挙げる．文中の「探索子」とは，集合の要素を操作するために適切な要素を指している抽象的なオブジェクトのことである．

- Iterator* OpenScan(Iterator* i)
  ポインタiが指す探索子が指す位置から集合を探索する探索子を指すポインタ(iが空の時は適当な位置の探索子を指すポインタ)を返す．
- Type* GetElement(Iterator* i)
  ポインタiが指す探索子の指す要素へのポインタを返す．ここで，返り値をポインタにしたのは，要素自身を直接扱うためである．
- Iterator* Next(Iterator* i)
  ポインタiが指す探索子の指す要素の次の要素を指す探索子へのポインタを返す．
- void CloseScan(Iterator* i)
  ポインタiが指す探索子を消去する．

ここで，Iterator，Typeはそれぞれ集合クラスの探索子，要素の型を総称的に表したものである．

2.3 オブジェクトの検索と処理

INADAでは，前述の集合オブジェクトの各要素に対する操作を記述するためにfor all文を提供している．これはベース言語であるC++の持つfor文を拡張したものである．for all文を使って，集合オブジェクトの要素のうちある条件を満たすオブジェクトに対する操作を記述することができる．domainを，Type型のオブジェクトを要素とする集合オブジェクトへのポインタとする時，

for all Type x in domain
such that ≪条件式≫
do ≪文≫

は，domainが指す集合オブジェクトの持つ要素のうち，≪条件式≫を満足する要素をxで表し，それに対する処理≪文≫を実行することを意味する．これは前述した集合クラスが持たなければならないインタフェースを用いた操作に展開される[7]．

3. ビューとその実現

RDBではデータの集合論的な操作と，単純だが強力な集合指向の問い合わせ言語を提供することに成功した．OODBにも集合指向の問い合わせの機能を導入しようとする試みがいくつかみられる[4),9),11),13)]．しかし，ビューについての議論がなされてはいる[2),16),19),20),23)]が，そのどれもビューの機能を実装するには至っていない．

関係ビューの実装には大きく二つの考え方がある．一つはビューを実体として持つ方法である．この方法では，データの変更は自動的にはビューに反映されないため，データの変更が生じた時にビューの値を正しく設定する機構が必要となる．もう一つは実体として存在しない関係として保持するビューである．この方法では，データの変更が生じても，ビューを評価すれば自動的にその変更は反映されることになる．この場合，ビューに対する質問は基底関係への問い合わせへと変換される．OODBの中に後者に基づくビュー機能を仮想クラスによって統合する試みがある[19),23)]．しか

し，クラスがオブジェクトのコンテナである場合，この統合は容易ではない．それらのオブジェクトモデルでは，一つのクラス階層の存在のみを許しているため，ビューを定義する仮想クラスを必ず既存のクラス階層の中に組み込まなければならない．クラス階層が属性の継承関係を決定しているため，問い合わせによって定義される仮想クラスをどこに位置付けるかが大きな問題となる[19),23)]．

一方，INADA はC++を拡張した言語であり，C++と同様にクラス階層は一つでなくてもよい．さらに，INADA ではオブジェクトのコンテナは集合オブジェクトである．集合オブジェクトは継承階層を形成するものではないため，この問題は生じてこない．しかし次の二つの問題がある．どのようにビューを定義し，どのようにしてそれを INADA の枠組の中で実現するかである．なぜなら集合オブジェクトはユーザが定義するクラスによって生成されるものであり，また集合の要素を操作するために提供する'for all' ループはプリミティブな構文にすぎないためである．

ここでは，ビューの具体例をいくつか考え，それを INADA でどう記述し，それがどのように展開され，実現されるかについて述べる．

先にも述べたが，ビューの実現法の一つに，ビューが評価される時に計算される仮想的な集合として実現する方法がある．この実現法はビューが評価されるたびに計算を行わなければならないが，メモリ上に保存されているデータの変更がビューに自動的に反映されるため，INADA ではこの方法を採用する．

INADA ではビューの定義のため，次のような定義文を提供する．

```
view ViewSet on BaseSet {
        a_i1 for a_j1 ;
        a_i2 for a_j2 ;
            :
        a_ik for a_jk ;
    where
            condition
    } ;
```

ここで，$a_{il}(l=1, 2, \cdots, k)$ はビューの持つ属性，$a_{jl}(l=1, 2, \cdots, k)$ は基底オブジェクトの持つ属性を意味する．最初にでてくるキーワード view は，C++の class がクラスの定義であるのを示すのと同じく，これがビューの定義であることを表現している．a_{il} for $a_{jl}(l=1, 2, \cdots, k)$ とは「ビューの要素が属性 a_{il} を持ち，それが基底オブジェクトの持つ a_{jl} に対応する」ことを意味する．a_{il} と $a_{jl}(l=1, 2, \cdots, k)$ はメンバ変数でもメソッドでも良いが，$a_{jl}(l=1, 2, \cdots, k)$ は public 定義されていなければならない．

関係モデルの選択演算および射影演算に対応するビュー(以下これらを各々選択ビュー，射影ビューと呼ぶ)について論じる．選択および射影ビューの実装のために，次のようなアプローチをとる．

(1) 射影ビューとして属性 $\langle a_{i1}, a_{i2}, \cdots, a_{ik}\rangle$ を持つC++のクラス View を定義する．このクラスのインスタンスオブジェクトは(2)で定義される ViewSet クラスのインスタンスオブジェクトの要素で，実体のない仮想的な要素である．ViewSet クラスのインスタンスオブジェクトは BaseSet クラスの基底集合のビューとなる．ここで BaseSet は別のところで定義してあるものとする．

(2) 次のような INADA のクラス ViewSet を定義する．すなわち，そのインスタンスオブジェクトの要素の型は，選択ビューでは Base，射影ビューでは View となるクラスである．Base クラスは別のところで定義してあるものとする．ViewSet クラスの中のメソッドでは 'OID as Type' という役割の型の参照を用いて，BaseSet クラス内のメソッドが実行される．さらに選択ビューとしての ViewSet の持つ集合インタフェース内に選択条件(述語)condition を含む．

(3) ViewSet クラスのオブジェクトを BaseSet クラスの基底集合オブジェクトの別の型のオブジェクトとして生成する．つまり，BaseSet 型のオブジェクトに対し，ViewSet 型を付加し，マルチタイプオブジェクトを生成する．このマルチタイプオブジェクトが基底集合オブジェクトのビューとなる．

(4) ビューを操作する'for all' 文をその基底集合を操作する 'for all' 文へと変換する．これは INADA 処理系が要求している集合オブジェクトの持つべきインタフェースを BaseSet および ViewSet ともに満たしているため，それらを用いた'for all' 文の変換処理が自動的に行われる．

このビューの実現のオーバヘッドは，基底集合の別の型としてビューの集合オブジェクトを生成し，ビュー操作が終わって必要がなくなったときにそれを削除することである．ビュー集合の要素のクラスは上記の(1)のようにして定義されるが，実際にそのクラスのオブジェクトは生成されず，そのためのオーバヘッド

は生じない．このアプローチを明確にするため，次の例に基づいて具体的にビューを生成してみる．

[例1] ある会社の従業員の集合をモデル化する．各従業員を名前，年齢，所属部署番号と給料を属性として持つクラスEmployeeにより表現し，その永続オブジェクトが集合オブジェクトeset(クラスSet〈Employee〉の一つの永続インスタンスオブジェクト)内に多数存在するものとする．

必要以上に例を複雑化せず，後の議論のために最低限必要な部分のみを持つ簡単なコード例を図1に示す(メソッドの実装の細部その他の定義は省略してある)．クラスSetは，テンプレート機能を用いた実装例を挙げている．クラスEmployeeのコンストラクタではメンバ変数のデフォルト値を指定している．もしコンストラクタの呼び出し時にメンバの初期値が与えられない場合，ここで指定された値が設定される．メソッドChange_Dept()は異動に対する更新操作である．永続なEmployeeオブジェクトをSet〈Employee〉オブジェクトの要素として五つ生成している．phmoの指す永続ヒープは，ファイルEmpFileにマッピングされている．これらを概念的に表したのが図2である．

```
class Employee{
public:
    char name[40];
    int age;
    int deptno;
private:
    int pay;

public:
    Employee(char* n="No-Name",int a=0,int d=-1,int p=-1)
        { strcpy(name, n);
          age=a;
          deptno=d;
          pay = p;} // コンストラクタ
    ~Employee(); // デストラクタ
    char* Name(){ return name; }
    int Age(){ return age; }
    int DeptNo(){ return deptno; }
    int Pay(){ return pay; }
    Change_Dept(int d){ deptno=d; }
                // 異動のための更新メソッド
        :
};

template <class T>
class Set{
public:
    Set(); // コンストラクタ
    T* GetElement(iterator<T>* i);
    iterator<T>* Next(iterator<T>* i);
    iterator<T>* OpenScan(iterator<T>* i);
    void CloseScan(iterator<T>* i);
        :
};

main(){
    PHFT* phft = new PHFT();
    PHMO* phmo = new PHMO(phft, "EmpFile");
    persistent Employee *emp;
    persistent Set<Employee>* eset=new(phmo)Set<Employee>();
    emp=new(eset)Employee("Ari",26,4,1000);
    emp=new(eset)Employee("Kyu",50,2,3000);
    emp=new(eset)Employee("Csc",34,4,1800);
    emp=new(eset)Employee("Dat",28,1,1500);
    emp=new(eset)Employee("Kum",55,3,3800);
              :
}
```

図1 例1のクラス定義とインスタンスオブジェクトの生成
Fig. 1 Class definitions of example 1 and creation of instance objects.

以下例1を使って具体的にビューを定義してみる．

[ビュー1] 従業員の名前，年齢と部署を求め，給料を隠す(射影ビュー)．

これはINADAでは次のように記述する．

```
view Proview on Set 〈Employee〉 {
    char* Name() for Name();
    int AGE() for Age();
    int DeptNo() for DeptNo();
};
```

char* Name() for Name()とは「ビューが返り値の型char*のメソッドName()を持ち，それが基底オブジェクトの持つメソッドのName()に対応する」ということを意味する．また，int AGE() for Age()とあるように，ビューの持つメソッド名は基底オブジェクトの持つメソッドと同じ名前である必要はない．

これは図3のようなコードに展開される．クラスProview_Employeeにより給料のデータにはアクセスできない．これを要素の型とするクラスProviewにより射影ビューが実現される．

[ビュー2] 給料が2000以上ある従業員を求める(選択ビュー)．

このビューは例えば以下のように定義すればよい．

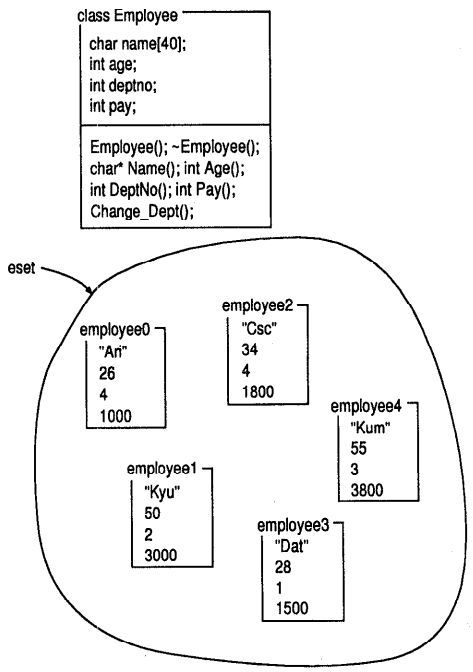


図2 例1の概念図
Fig. 2 A conceptual figure of example 1.

```
class Proview_Employee{
private:
    char name[40];
    int age;
    int deptno;
    int pay;

public:
    char* Name(){return ((Employee*)this)->Name();}
    int AGE(){return ((Employee*)this)->Age();}
    int DeptNo(){return ((Employee*)this)->Deptno();}
};

class Proview{
public:
    Proview(){}
    Proview_Employee* GetElement(iterator<Proview_Employee>* i){
        return (Proview_Employee*)
            (this as Set<Employee>->GetElement((iterator<Employee>*)i));}
    iterator<Proview_Employee>* Next(iterator<Proview_Employee>* i){
        return (iterator<Proview_Employee>*)
            (this as Set<Employee>->Next((iterator<Employee>*)i));}
    iterator<Proview_Employee>* OpenScan(iterator<Proview_Employee>* i){
        return (iterator<Proview_Employee>*)
            (this as Set<Employee>->OpenScan((iterator<Employee>*)i);}
    void CloseScan(iterator<Proview_Employee>* i){
        this as Set<Employee>->CloseScan((iterator<Employee>*)i);}
};
```

図3 射影ビュー
Fig. 3 A projection view.

```
view Selview on Set<Employee> {
    char* Name() for Name();
    int Age() for Age();
    int DeptNo() for DeptNo();
    int Pay() for Pay();
    where Pay()>=2000;
};
```

このビュー定義の例では，ビュー集合の要素の持つ属性をすべて挙げている．これはビューによって要素の持つ属性が同じであることを意味するが，ビュー定義中に属性の記述があると，図3にあるようにビュー集合の要素のクラスを展開することになり，無駄である．これを避けるには，次のような定義をすれば良い．

```
view Selview on Set<Employee> {
    where Pay()>=2000;
};
```

このように，メンバの記述がまったく無く，where句のみによるビューの定義を許す．この定義に対しては**図4**のような展開をする．図4の展開例のように，メンバの記述のないビュー定義に関しては基底オブジェクトの型をそのまま使うことを意味し，その展開は条件を満たす要素のみを返すビュー集合の定義のみとなる．クラスSelviewの持つ標準インタフェース中に条件が組み込まれている．

[ビュー3] 年齢50以上の従業員の名前を求め，その他のメンバを隠す．

これをINADAで表現すると以下のようになる．

```
view Combiview on Set<Employee> {
    char* Name() for Name();
    where
        Age()>=50;
};
```

ビューオブジェクトの持つメンバとwhere句の両方の記述のあるビュー定義である．これは選択と射影の二つを組み合わせたものであり，**図5**のように展開される．クラスCombiview_Employeeはビュー定義通り公開メソッドとしてはName()しか持たないものとなっている．条件'Age()>=50'はクラスCombiviewの標準インタフェース中に組み込まれている．

```
class Selview{
public:
    Selview(){}
    Employee* GetElement(iterator<Employee>* i){
        return this as Set<emplyee>->GetElement(i);}
    iterator<Employee>* Next(iterator<Employee>* i){
        iterator<Employee>* tmp
            =this as Set<Employee>->Next(i);
        if (!tmp){
            CloseScan(tmp);
            return 0;}
        if((GetElement(tmp)->Pay())>=2000)
            return tmp;
        else tmp = Next(tmp);
        return tmp;}
    iterator<Employee>* OpenScan(iterator<Employee>* i){
        iterator<Employee>* tmp
            =this as Set<Employee>->OpenScan(i);
        if(!tmp){
            CloseScan(tmp);
            return 0;}
        if((GetElement(tmp)->Pay())>=2000)
            return tmp;
        tmp = Next(tmp);
        return tmp;}
    void CloseScan(iterator<Employee>* i){
        this as Set<Employee>->CloseScan(i);}
};
```

図4 選択ビュー
Fig. 4 A selection view.

4. 評　価

これまでにあげた例を実際に計算機上で動かしてみた．その結果を**表1**に示す．実験では10000個のオブジェクトを，ソートされていない単純な1次元リスト

```
class Combiview_Employee{
private:
    char name[40];
    char age;
    char deptno;
    int pay;

public:
    char* Name(){return ((Employee*)this)->Name();}

protected:
    int Age(){return ((Employee*)this)->Age();}
    friend class Combiview;
};

class Combiview{
public:
    Combiview_Employee* GetElement(iterator<Combiview_Employee>* i){
        return (Combiview_Employee*)
            (this as Set<emplyee>->GetElement((iterator<Employee>*)i));}
    iterator<Combiview_Employee>*
        Next(iterator<Combiview_Employee>* i){
            iterator<Employee>* tmp
                =this as Set<Employee>->Next((iterator<Employee>*)i);
            if (!tmp){
                CloseScan( (iterator<Combiview_Employee>*)tmp );
                return 0;}
            if((GetElement((iterator<Combiview_Employee>*)tmp)->Age())>=50)
                return (iterator<Combiview_Employee>*)tmp
            else tmp
                =(iterator<Employee>*)Next((iterator<Combiview_Employee>*)tmp);
            return (iterator<Combiview_Employee>*)tmp;}
    iterator<Combiview_Employee>*
        OpenScan(iterator<Combiview_Employee>* i){
            iterator<Employee>* tmp
                =this as Set<Employee>->OpenScan((iterator<Employee>*)i);
            if(!tmp){
                CloseScan( (iterator<Combiview_employee>*)i);
                return 0;}
            if((GetElement((iterator<Combiview_Employee>*)tmp)->Age())>=50)
                return (iterator<Combiview_Employee>*)tmp;
            tmp=(iterator<Employee>*)Next((iterator<Combiview_Employee>*)tmp);
            return (iterator<Combiview_Employee>*)tmp;}
    void CloseScan(iterator<Combiview_Employee>* i){
        this as Set<Employee>->CloseScan((iterator<Employee>*)i);}
};
```

図5 射影と選択を組み合わせたビュー
Fig. 5 A combination view of projection and selection.

表1 射影，選択，およびその混合ビューの評価(単位は秒)
Table 1 Evaluation of projection and selection and thier combination views (in seconds).

選択条件を満足する割合	基底集合	射影ビュー	選択ビュー	混合ビュー
0%	1.94	2.15	0.54	0.56
20%	1.95	2.24	0.66	0.67
40%	1.95	2.24	1.04	1.06
60%	1.97	2.24	1.45	1.46
80%	1.95	2.23	1.85	1.87
100%	1.95	2.24	2.24	2.26

による集合で実現し，10000個のオブジェクトのうち，選択条件を満足する割合を変化させた時の，集合の保持するすべての要素の持つ名前を表示させた．基底集合の操作に要する時間に対し，射影ビューは射影を施すだけ時間がかかった．一方選択ビューでは，選択条件の判断が容易なため，選択条件を満足する要素が少ないほど基底集合の操作よりも早く処理を行うことができた．

5. ビューの挿入・削除

ここでのビューの挿入および削除操作は，コンストラクタ new，デストラクタ delete による要素の挿入，削除に対応する．

挿入操作では，基底集合の要素のオブジェクトが生成され，挿入されることになる．ルールは次の通り．

- 選択ビューに対しては
  選択条件を満たさないオブジェクトの挿入は許さない．これは，処理系がコンストラクタ呼出時に選択条件を満足するかを調べることにより実現する．これにより，ビューからのより安全な挿入操作を提供することが可能となる．
- 射影ビューに対しては
  射影によって隠されたメンバ値は，基底集合オブジェクトの要素のコンストラクタ内で指定された

デフォルト値で補うことによって新たなオブジェクトが挿入される．

削除操作では，基底集合の要素オブジェクトが削除される．このときルールは次の通り．

●選択ビューに対しては

選択条件を満たすオブジェクトのみが削除の対象となる．deleteのオペランドとなるOIDが，ビューから得られたものであるため，操作は自明である．

●射影ビューに対しては

基底集合の要素の持つ，射影によって現れてこない属性値もすべて削除される．

例としてビュー1を次のように変更して考える．

[ビュー4] 従業員の名前，年齢と部署を求め，給料を隠す(射影ビュー)．また，ビューからの異動に対する更新操作と挿入操作を許す．

これはINADAでは以下のような記述となる．

```
view Proview on Set<Employee> {
    char* Name() for Name();
    int AGE() for Age();
    int DeptNo() for DeptNo();
    Change_Dept(int D) for Change_Dept
    (D);
    Proview(char* N, int A, int D)
        for Employee(N, A, D);
};
```

この定義にはコンストラクタと異動のための更新メソッドが含まれている．この定義の場合，図3の展開例のうち，クラスProview_Employeeの部分が**図6**の展開例へと変更される．

更新操作はそのまま基底オブジェクトの更新操作へと変換され，実行される．また，例えば

```
new(peset)Proview ("Mas", 26, 4);
```

という挿入操作が起こると

```
new(peset)Employee("Mas", 26, 4, -1);
```

が評価され，実行されるのと同じ結果となる．

6. 仮想集合属性

集合オブジェクトに対してマルチタイプオブジェクト機構を適用することによって選択および射影ビューを仮想集合として実現できることを示した．これらはともに一つの集合オブジェクトに対するビューである．ここでは複数の集合オブジェクトに対する仮想集合について論じる．

次のような例を考える．

[例2] ある会社の部署の集合をモデル化する．各部署を部署名，部署番号と部長名を属性として持つクラスDeptにより表現し，その永続オブジェクトが集合オブジェクトdset(クラスSet〈Dept〉の永続オブジェクト)内に多数存在するものとする．

```
class Proview_Employee{
private:
    char name[40];
    int age;
    int deptno;
    int pay;

public:
    char* Name(){return ((Employee*)this)->Name();}
    int AGE(){return ((Employee*)this)->Age();}
    int DeptNo(){return ((Employee*)this)->Deptno();}
    Change_Dept(int D){((Employee*)this)->Change_Dept(D);}
    Proview_Employee(char* N, int A, int D)
        {((Employee*)this)->Employee(N,A,D);}
};
```

図6 更新メソッドを持つ射影ビュー

Fig. 6 A projection view with a method to update.

```
class Dept{
    char name[40];
    int deptno;
    char mng[40];
public:
    Dept(); // Constructor
    char* Name();
    int DeptNo();
    char* Mng();
};

main(){
                :
    persistent Dept *dpt;
    persistent Set<Dept>* dset=new(phmo)Set<Dept>();
    dpt=new(dset)Dept("Design",4,"Ari");
    dpt=new(dset)Dept("Sale",1,"Dat");
                :
}
```

図7 例2のクラス定義とインスタンスオブジェクト

Fig. 7 Class definitions of example 2 and creation of instance objects.

これに対する簡単なコード例を**図7**に示す．図7の集合オブジェクトと，前述のEmployeeを要素とする集合オブジェクトを合わせて，「部署に所属する従業員を求める」ことを考える．今までの議論にそって，二つの集合オブジェクトo_1とo_2のマルチタイプオブジェクトとして生成することを考える．すなわち'OID as Type'の書式を使ってo_1とo_2のメソッドにアクセスできるように生成することを考える．しかし，INADAでは二つの独立のオブジェクトの型をマルチタイプオブジェクトで表現することはできない．これは，オブジェクト指向の概念が一つの実体の性質や振る舞いの表現には適しているが，二つの実体の関連を表現するには適していないためである．つまり，二つの実体の関連を操作しようとすれば，その関連を一つの個別の実体として認識しなければならない．

そこで，オブジェクトに仮想集合属性を持たせることによってこれを実現することを考える．オブジェクトの仮想集合属性は，その値として仮想の集合オブジェクトを持つ．その属性がアクセスされ評価される時，値は実体となる．

これは，クラスDeptに仮想集合属性Member()を持ったクラスExtDeptを例えば次のように記述すれば良い．

```
class ExtDept {
public:
    char* Name() for Dept::Name();
    int DeptNo() for Dept::DeptNo();
    char* Mng() for Dept::Mng();
    Set<Employee>* Member()
        where Employee::DeptNo()
            ==Dept::DeptNo();
};
```

このExtDept型をDept型のオブジェクトに付加し，その型を通してアクセスすれば，その部署に所属する従業員を求めることが可能である．

7. むすび

本稿では永続プログラミング言語INADAにより，オブジェクト指向の枠組みの中でどのようにビューを実現するかについて論じた．永続プログラミング言語の持つべき性質としてマルチタイプ機能が上げられる．それをうまくオブジェクトの集合に適用することによってオブジェクト指向のビューが実現できることを示した．

INADAはC++を拡張した永続(データベース)プログラミング言語である．INADAの持つ機能の多くは文献3)，4)，9)，15)，17)などに代表される他の多くの研究から大きな影響を受けている．しかしながら次の点において他のものと明確に異なる．(1) INADAはマルチタイプオブジェクトをサポートしている．(2) ビューは仮想集合として実現される．ビューは基底集合の一つの型として実現される．

OODBにおけるビューに関する多くの提案では，ビューを質問により導かれる仮想クラスと考える[12),19),20),23)]．それは，彼らのオブジェクトモデル内ではクラス階層は必ず一つしか存在できないという前提と，クラスがオブジェクトのコンテナとなっているためである．そのためにビュー機能のために非常に複雑なクラス階層を常に構築しなければならないという問題が生じる．

我々のとったアプローチはこれらとはまったく異なる．INADAではクラスはオブジェクトのコンテナでなく，ビューは仮想集合として実現される．このためデータベーススキーマであるクラスの構造そのものを変更する必要がない．

文献21)はOIDを保ちながらオブジェクトに複数のインタフェースを許すことによってビューを実現する試みである．ところが，ここではオブジェクトの集合についてはサポートしていない．

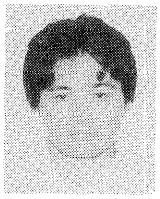
INADAはオブジェクトの集合の扱いを考え，それに対してビューを実現した．ビュー，すなわち仮想集合に対する操作はその基底集合への質問へと自動的に変換されることも示した．

参考文献

1) Albano, A., Bergamini, R. et al.: An Object Data Model with Roles, *Proc. the 19th VLDB Conf.,* Dublin, Ireland, pp. 39-51 (1993).
2) Abiteboul, S. and Bonner, A.: Objects and Views, *Proc. the 1991 ACM SIGMOD Int'l Conf. on Management of Data,* pp. 238-247 (May 1991).
3) Agrawal, R. and Gehani, N. H.: ODE (Object Database and Environment): The Language and the Data model, *Proc. the 1989 ACM SIGMOD Int'l Conf. on Management of Data,* Portland, Oregon, pp. 36-45 (May 1989).
4) Aoshima, M., Izumida, Y., Makinouchi, A. et al.: The C-based Database Programming Language Jasmine/C, *Proc. 16th VLDB Conf.,* pp. 539-551 (Aug. 1990).
5) Aritsugi, M. and Amano, H.: Views in an Object-Oriented Persistent Programming Language, *Proc. Int'l Symposium on Next Generation Database Systems and Their Applications,* pp. 18-25 (Sep. 1993).
6) 有次正義，天野浩文，牧之内顕文：永続プログラミング言語INADAのマルチタイプオブジェクト，情報処理学会，アドバンストデータベースシステムシンポジウム，pp. 93-100 (Dec. 1992).
7) Aritsugi, M., Teramoto, K., Amano, H. and Makinouchi, A.: Multiple Type Objects and Set Objects in an Object-Oriented Persistent Programming Language, *Technical Report CSCE-93-C02,* Dept. of Comp. Sci. and Comm. Eng., Kyushu University (Mar. 1993).
8) Atkinson, M., Bancilhon, F., DeWitt, D. et al.: The Object-Oriented Database System Manifesto, *The First Int'l Conf. on DOOD*, Kyoto, Japan, pp. 40-57 (1989).
9) Deux, O. et al.: The Story of O 2, *IEEE Trans. on Knowledge and Data Eng.,* Vol. 2, No. 1, pp. 91-108 (Mar. 1990).
10) Ellis, M.A. and Stroustrup, B.: *The Annotated C++ Reference Manual,* Addison-Wesley, Reading, Mass. (1991).
11) Fishman, D.H. et al.: Iris: An Object-Oriented Database Management System, *ACM Trans. on Office Information Systems,* Vol. 5,

No. 1, pp. 48-69 (Jan. 1987).
12) Heiler, S. and Zdonik, S.: Views, Data Abstraction, and Inheritance in the FUGUE Data Model, *Proc. the 2nd Int'l Workshop on Object-Oriented Database Systems (Lecture Notes in Computer Science 334)*, pp. 225-241 (Sep. 1988).
13) Kim, W., Garza, J.F. et al.: Architecture of the ORION Next-Generation Database System, *IEEE Trans. on Knowledge and Data Eng.*, Vol. 2, No. 1, pp. 109-124 (Mar. 1990).
14) Knuth, D.: *The Art of Computer Programming. Seminumerical Algorithms*, Addison-Wesley, Reading, Mass. (1969).
15) Lamb, C., Landis, G. et al.: The ObjectStore Database System, *Comm. ACM*, Vol. 34, No. 10, pp. 51-63 (Oct. 1991).
16) Mamou, J.-C. and Medeiros, C. B.: Interactive Manipulation of Object-oriented Views, *Proc. the 7th Int'l Conf. on Data Engineering*, Kobe, Japan, pp. 60-69 (Apr. 1991).
17) Richardson, J.E. and Carey, M.J.: Persistence in the E Language: Issues and Implementation, *Software—Practice and Experience*, Vol. 19, No. 12, pp. 1115-1150 (Dec. 1989).
18) Richardson, J. and Schwarz, P.: Aspects: Extending Objects to Support Multiple, Independent Roles, *Proc. the 1991 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 298-307 (May 1991).
19) Rundensteiner, E. A.: MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases, *Proc. the 18th VLDB Conf.*, Vancouver, British Columbia, Canada, pp. 187-198 (1992).
20) Scoll, M.H., Laasch, C. and Tresch, M.: Updatable Views in Object-Oriented Databases, *Proc. on the 2nd Int'l Conf. on DOOD*, Munich, Germany, pp. 189-207 (Dec. 1991).
21) Shilling, J. J. and Sweeney, P. F.: Three Steps to views: Extending the Object-Oriented Paradigm, *Proc. OOPSLA'89, ACM SGPLAN Notices*, Vol. 24, No. 10, pp. 353-361 (1989).
22) Stefik, M. and Bobrow, D. G.: Object-Oriented Programming: Themes and Variations, *The AI Magazine*, Vol. 6, No. 4, pp. 182-204 (1986).
23) Tanaka, K., Yoshikawa, M. and Ishihara, K.: Schema Virtualization in Object-Oriented Databases, *Proc. the 4th Int'l Conf. on Data Engineering*, pp. 22-29 (Feb. 1988).
24) Teramoto, K., Aritsugi, M. and Makinouchi, A.: Design, Implementation, and Evaluation of the Persistent Programming Language INADA, *Technical Report CSCE-94C-02*, Dept. of Comp. Sci. and Comm. Eng., Kyushu University (Mar. 1994).

(平成6年4月11日受付)
(平成7年1月12日採録)



有次 正義（正会員）

1991年九州大学工学部情報工学科卒業．同大学院工学研究科博士後期課程在学中（3年）．現在，文部省学生国際交流制度に基づく派遣留学生としてカナダのマギル大学に留学中．永続プログラミング言語，並列分散データベースなどに興味を持っている．

天野 浩文（正会員）

昭和61年九州大学工学部電子卒業．平成3年同大学院情報工学専攻後期課程修了．工学博士．同年同大学工学部助手．平成6年より同大学大型計算機センター助教授．データベース言語，並列プログラミング言語の研究に従事．ACM会員．

牧之内顕文（正会員）

昭和42年京都大学工学部電子卒業．昭和45年グルノーブル大学理学部応用数学科大学院修了（技術博士）．同年富士通(株)入社．(株)富士通研究所を経て，現在，九州大学工学部情報工学科教授．永続プログラミング言語，分散・並列オブジェクト指向データベースシステムの研究・教育に従事．