データベースシステム　95－1
（１９９３．　９．　９）

RDBMS設計に対するオブジェクト指向技法の適用

赤間 浩樹　武田 英昭

ＮＴＴ情報通信網研究所

今回我々は、RDBMSの拡張性と再利用性を向上させるため、カーネライズとライブラリ化を目的として、オブジェクト指向技法をRDBMS設計に適用することを試みた。

オブジェクト指向設計では、継承とポリモルフィズムを活用するようなオブジェクト構成を決定することが重要である。しかし、従来のオブジェクト指向技法で頻繁に使用される特殊化を重視した技法（差分プログラミング）はライブラリの構築に関して問題がある。そこで我々は、ポリモルフィズムをライブラリ構築に利用する方法をRDBMSの設計に適用した。

本稿では、オブジェクト指向技法を適用するにあたって行った検討の経緯と我々の採った手法、および、その効果を示す。

Application of object-oriented techniques to design RDBMS

Hiroki AKAMA　Hideaki TAKEDA

NTT Network Information Systems Laboratories

1-2356, Take, Yokosuka-shi, Kanagawa, 238-03 JAPAN

We applied the object-oriented techniques to designing RDBMS for improving extendibility and reusability. In the object-oriented design it is important to decide the object structure by inheritance and polymorphism. Especially the inheritance is usually used as specialization (difference-programming). However, This use of inheritance has many problems about the construction of reusable library. Therefore, we applied the inheritance to constructing the library for interface of replaceable object. This paper describes our considerations about the techniques, the object structure of our RDBMS, and effects of our design.

1　はじめに

現在、システム設計に対してオブジェクト指向技法の適用を試みることは、ごく当然のことになっている。しかし、現状のオブジェクト指向技法自体は、期待されているほどには万能ではなさそうである。

今回、我々はRDBMSの設計/実装に対してオブジェクト指向技法の適用検討を行った。

そこで本稿では、オブジェクト指向技法を適用するにあたって行った検討の経緯と我々の採った手法、および、その効果を示す。

1．1　いくつかの疑問

これまで、オブジェクト指向技法はGUIの分野において大成功をおさめたと言える。それはウィンドウ関連のクラスライブラリの充実が物語っている。しかし、それを他の分野に応用した場合に本当に上手くいっているのであろうか。初期に期待されたほどにはクラスライブラリが蓄積されていないのは何故だろうか。実際にオブジェクト指向の御利益を享受している設計者はどのくらいいるのであろうか。我々はオブジェクト指向に何を期待していたのだろうか。オブジェクト指向のメリットとは何なのだろうか？

さらに、各種のオブジェクト指向分析/設計(OOAD)手法が溢れ始めている。我々はどの手法を使用したら良いのだろうか。OOADを使うことによって何がどう良くなるのだろうか。本当に良くなるのだろうか？

また、開発言語としてC++を使用したという報告は多い。しかし、オブジェクト指向言語(OOPL)を使えば自然にオブジェクト指向プログラミング(OOP)になるのだろうか。我々はOOPすることを目的とするのではなく、目的のためにOOPを使用したいのである。メリットの明確なOOPとはなんなのだろうか？

1．2　いくつかの混乱

（1）オブジェクト

オブジェクトという用語は非常に定義が曖昧である。「オブジェクトは機能ではなく、その対象となるモノの方だ。」と言われて半分納得しながらも、「××機能はオブジェクトにはならないのか？」と問えば、「そういう機能を提供するモノ（たとえば××サーバ、××マネージャ、××エージェント）と考えれば良いのだ。」などと言われてしまったりする。

また、「オブジェクト指向では現実世界をモデル化すればいいのだ。」といわれていながら、概念や性質のオブジェクト化が重要だったりする。

結局、オブジェクトは何でもよく、全く掴み所がなく、それは何も言っていないのと同じなのである。

（2）オブジェクト指向分析/設計

オブジェクト指向技法はソフトウェア開発の各種の過程に適用されており、以下のようなものが存在する。

- オブジェクト指向分析(OOA)
- オブジェクト指向設計(OOD)
- オブジェクト指向プログラミング(OOP)

しかし、この段階分けは従来の構造化設計の思想に（無理矢理？）オブジェクト指向技法をあてはめたもので、オブジェクト指向における螺旋(spiral)状の開発と親和性が悪い。（フィードバックがかけにくい。）

さらに、これらのOOAやOODといった設計方法論は言語とは独立であるように主張されていることもあるが、実は個々のOOAD手法は特定のOOPLに強く依存している。これは、オブジェクト指向自体が明確にならないまま多くの言語が作られ広まったことにより、現状のオブジェクト指向プログラミング言語(OOPL)の共通部分が思ったより小さいためである。（たとえば、SmalltalkとCLOSでは設計思想が全く異なっている。） よって、OOADが各種のオブジェクト指向技法を包含すると考えるのは間違いである。

（3）オブジェクト指向プログラミング

OOPに限定すれば、抽象データ型、継承、ポリモルフィズムという特徴で定義できる。ここでよくある混乱は、オブジェクトが抽象データ型さえあれば実現できることから、単純に、OOP＝抽象データ型の使用、と考えてしまうことである。もちろん、抽象データ型は非常に有用なのであるが、それ自体はオブジェクト指向が登場する以前から既に有効性が実証されていた技術である。

つまり、OOPは抽象データ型を活用したプログラミングではなく、継承とポリモルフィズムを活用したプログラミングなのである。ところが、その継承とポリモルフィズムを有効活用するためのオブジェクトの設計方法が未だ確立していない。

1．3　オブジェクト指向に期待するもの

オブジェクト指向技法の利用によるメリットはなんだろうか。一般には、生産性/保守性/拡張性/再利用性などが言われ、たとえば、以下のような効果が期待されている。

- 現実の世界を容易にモデル化する手法を与えることで、生産性が向上する。
- 機能ではなくモノに着目することでシステムの変更に対する柔軟性を高める。
- オブジェクト間の独立性を実現することで、生産性や保守性を高める。
- 作成したクラスをクラスライブラリ化し、再利用によって生産性を高める。

ところで、これらの生産性/保守性/拡張性/再利用性

という用語は、一部に重複する意味を持っている。我々は、ソフトウェア開発において、モデル化の容易性よりもシステム完成後の拡張性や資産の再利用性の向上（ゴミを増やさないこと）が問題であると考え、本稿では主に拡張性と再利用性に着目することにする。

そこで、用語を以下のように定義する。

【拡張性が良い】

●機能拡張/修正容易
　既存のシステムに対する機能の拡張/修正が容易であること。

●データ拡張/修正容易
　既存のシステムに対するデータ／データ型の拡張/修正が容易であること。

【再利用性が良い】

●ライブラリ利用容易
　既存ライブラリの利用が容易にできること。

●ライブラリ・カスタマイズ容易
　既存ライブラリのカスタマイズが容易にできること。

●ライブラリ構築容易
　再利用可能な汎用のライブラリの構築が容易にできること。

また、どのソフトウェア開発段階を対象とするかという問題もある。これは以下の理由により本稿での主な対象はOOP段階（設計もOOPLで行う）とする。

- 現在のソフトウェア開発において、分析や設計の拡張/再利用のニーズよりも、プログラムコードの拡張/再利用のニーズが圧倒的に多いと考えること。
- オブジェクト指向の開発法は螺旋状であり、分析/設計/コード化が本来一つの表現/言語で行われるべきであること。
- 近代的な言語はそのコードが直接設計情報に対応すること。（OOPLとOOAD専用の記法とは単にグラフィカルか否か程度の差しかない。）

2　設計対象の概要と方針

2．1　システムの概要

今回、オブジェクト指向技法をリレーショナル型データベース管理システム(RDBMS)の設計/実装に適用した。

設計対象のRDBMSは特定分野向けの専用DBMSであり、以下のような特徴を持っている。

（1）高速処理向けDBMS
（2）言語仕様は高速処理向けにサブセット化したSQL
（3）言語処理方式はコンパイラ方式
（4）APは埋め込みSQLを含むC言語

実装にはAT&T C++ v2.1を使用した。

2．2　設計目標

今回の設計では拡張性と再利用性の実現のために、DBMSのカーネライズとDBMSクラスライブラリ構築を目標とした。

従来のクラスライブラリが主に低レベルのクラスライブラリ（LIST,TREEなど）の整備を行ってきたのに対し、今回のDBMSライブラリは複数のライブラリを組み合わせて使用する比較的大きめ（AP依存部分）のライブラリ化を狙うところが若干異なる。

なお、SQL言語処理部分（プリコンパイラを含む）の設計/実現にはyacc/lexを使用したことにより、文法と実装の分離が可能になった。よって、言語処理部分については既に目標の拡張性や再利用性が実現されたと考え、無理なオブジェクト指向設計は行わなかった。

2．3　設計上の問題

（a）クライアント-サーバ構成と高速性の問題

現在のオブジェクト指向とネットワークの融合の流れを考えると、DBMSをカーネル部分と機能単位のサーバ群に分割し、サーバプロセス間のメッセージ通信によって処理を行うクライアント-サーバ構成が適当に思える。

しかし、汎用DBMSではともかく、本稿で対象とする高速処理向けDBMSにおいては、プロセス間通信は処理が重すぎて採用することができなかった。

（b）カーネライズとオブジェクト指向の相性の問題

一般的なオブジェクト指向ではモノに着目したクラス構成群が作成されるのに対し、一般的なカーネライズでは機能単位のモジュール群が作成される。

よって、カーネライズとオブジェクト指向は相性が悪いという問題があった。

（c）直感的なモデル化に基づく設計の問題

RDBMS内に存在する代表的なオブジェクトを中心にした簡単なオブジェクト指向分析の結果を図1に示す。

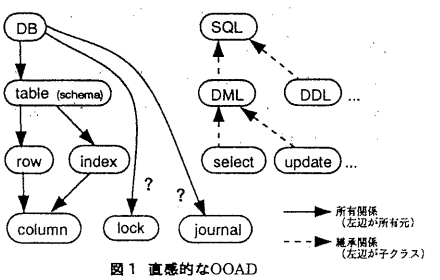

図1　直感的なOOAD

このモデルは非常に単純で直感的である。オブジェクト指向の特徴の一つにモデル化が容易という点があげられるが、まさに問題領域のモデル化を素直に行った結果である。

しかし、以下のような理由により、我々はこのモデル、

または、この方向の分析に満足できない。

- 拡張性や再利用性への効果が不明である。このモデルにより何がDBMSカーネルで、どれがDBMSのクラスライブラリとなるのかが分からない。
- オブジェクト抽出を直感に頼ったため、既存のRDBMSの所有構造がモデルの中心となる。ところが、所有構造が拡張性やライブラリの汎用化を阻害するのではないかという不安がある。
- ポリモルフィズムはDB/Table/Rowに対して同一メッセージ（select等）を送ることで利用できそうである。しかし、それが何の役に立つのかが分からない。

（d）OOPテクニック上のオブジェクト指向の問題

全く逆に、単純な言語上のテクニックだけのオブジェクト指向技法の適用法としては以下のようなものが考えられる。

- 従来の設計においてidやポインタに相当するものをclassにする。同時に関連するメソッドを集める。
- 従来の設計においてstructに相当するものをclassにする。同時に関連するメソッドを集める。
- 各クラス間で構造やメソッドの共有部分があれば上位クラスとする。
- 従来の設計においてswitchに相当するものをポリモルフィズムで実現する。

この方法でもDBMSのカーネライズができるようにも思えないし、拡張性や再利用性が得られたという実感もない。

結局、オブジェクト指向技法を適用するのはいいのだが、そのメリットが明確でないため、何のためのOOPか分からないという印象を受けた。

3　継承の利用方法

前述の通り、OOPには３つの特徴があり、問題となるのは、継承やポリモルフィズムを活かすようなオブジェクト構成を決定することである。

そこで、以下ではOOPの最大の特徴である継承に着目し、DBMSのカーネライズとクラスライブラリ構築を目指したDBMS構成を検討する。

3．1　特殊化重視のオブジェクト決定（差分プログラミング）

3．1．1　メリット

継承の利用で一般に言われる最大の効果は差分プログラミングによるものであろう。

上位/類似のクラスを継承し、差分だけ記述（特殊化）すれば目的のクラスになるという差分プログラミングは、クラスの拡張やカスタマイズを容易にし、生産性や保守性を向上させる場合が多い。

3．1．2　デメリット

しかし、差分プログラミングは現実的には非常に問題の多い方法でもある。

【問題１】関連コードの分散をまねく。

差分プログラミングは修正個所の局所化（１カ所修正すれば関連するところがみんな変更される）という意味では一見保守性に優れる。

しかし、現実には差分化されることによって関連するコードが分散することも多く、局所性を崩す場合も多い。

これはプログラミング言語と環境の充分な支援がなければ逆に保守性を低下させることになる。

【問題２】クラスライブラリが蓄積されていない場合には効果が少ない。

問題領域に適したクラスライブラリが揃っていてはじめて効率的な差分プログラミングが可能になる。

しかし、現実にはクラスライブラリの整備は予想以上に遅れている。

【問題３】クラスのライブラリ化を阻害する。

これが差分プログラミングの最大の問題である。理想的なプログラミングではクラスライブラリの形成は以下のように行われる。

［step1］ 既存のライブラリを利用して新しいクラスを作る。

［step2］ その結果（クラス）を再利用可能なライブラリにする。

このサイクルが循環すれば部品資産の充実とその再利用によってソフトウェアの生産性は飛躍的に向上する。

しかし、多くのOOPではstep2の機能が上手く働かず、特定AP（問題領域）に強く依存したクラスが生成されてしまう。そして、そのようなクラスでは再利用可能なライブラリとしての資産価値があまりない（…か、全くない）。

単純に考えても、差分プログラミングがライブラリの汎用化を疎外する例は以下のような場面で発生する。

（例１）安易に差分を追加することが、ライブラリの汎用性を破壊する。

既存のクラスC1に機能または属性Sを加えれば目的のクラスC2ができるとする。ここで、安易にクラスC1を親とするクラスC2を作ることからクラスのライブラリ化ができなくなっていく。

たとえば、C1:円であり、C2:楕円である場合を考える。一般に最初から充分な汎化などは行われていないので、最初に作ったときはC1:円だけで充分だと思うことが多い。そして、いざ、C2:楕円が必要になった場合に、以

下のような理由によりC2:楕円をC1:円の子クラスとして作成してしまう。

- クラスC1とクラスC2の作成者が異なるため、C1クラスの修正は最低限にしたい。
- C1:円に比べてC2:楕円の方が属性数が多いので、差分プログラミングが活用できる。

しかし、ライブラリの汎用性を考えれば当然C1:円の上位クラスとしてC2:楕円を挿入しなければならなかったのである。ところが実際には、これは既存のクラスの上位クラスを設計/挿入することになってしまう（図2）。これは上位クラスの継承構造を意識する必要があるので、一般的にはかなり難しい。

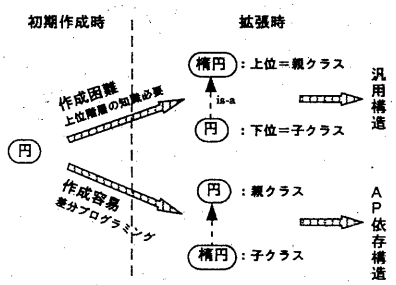

図2 クラスの拡張

同様に、C1:三角形でC2:四角形の場合も同様である。この場合には、C2:四角形の挿入のために、C3:多角形という上位クラスの挿入が必要になる。

（例2）属性は所有構造と継承のどちらでも表現できる。
よって、属性の実現方法が問題領域に依存し、汎用性が疎外される。
また、複数の属性でクラス分け（継承）するときにどちらの属性を重視するか（上位の継承にするか）も問題領域に依存する。

あるクラスC1への情報Sの追加を、構造的所有として表現するか、クラス分けするか、の判断はプログラマに任されており、問題領域に依存した構造を採ってしまう。

たとえば、C1:社員、S1:職種（営業、経理、企画、…）、S2:役職（アルバイト、役職なし、管理職、…）とした場合に、S1とS2のどちらをクラス分けの基準にするかによってクラス構造は変化する（図3(a),(b),(c)）。

また、S1とS2の両方をクラス分けの基準にする場合にも、どちらを優先するかが問題になる（図3(d),(e)）。

この例に限らず、どの属性/特徴に着目してクラス分類するかは非常に難しく（つまり個人差が大きい）、それは生物の分類体系の歴史を見てもわかる。

時々、「深い継承は良くない」などという経験則を見ることがあるが、上記の問題が影響していると考えられる。

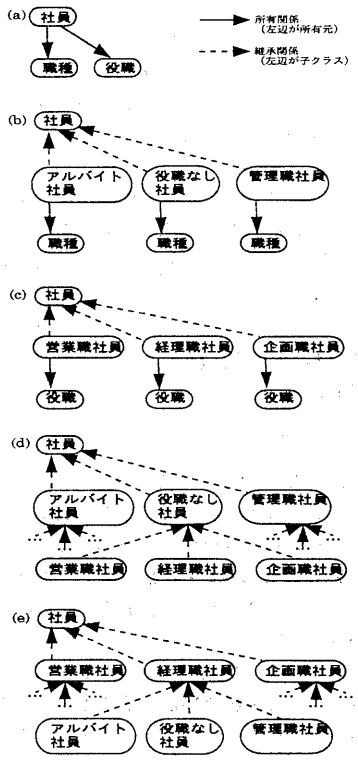

図3 属性の実現方法

【問題4】汎化階層に従っても有効なクラスライブラリが得られない。

通常の汎化階層では図4(a)のように整理されるに違いない。

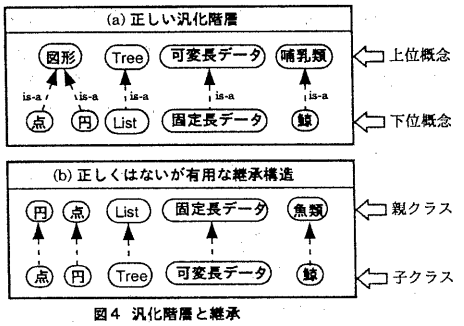

図4 汎化階層と継承

しかし、現実のクラスライブラリにおいては図4(b)のようにTREEはLISTの子クラスになっていないだろうか。可変長データは固定長データの子クラスになってはいないだろうか。

このように、鯨クラスを魚クラス下に置くのは、それなりに意味がある場合が多い。当然のことではあるが、差分プログラミングを有効に活用するクラス階層は、汎

化階層とは異なるのである。（これを汎化とOOPの継承ミスマッチ問題と呼ぶ） つまり、汎化階層が必ずしも有効なクラスライブラリを生成するとは限らず、慎重に汎化を行えば綺麗なクラスライブラリができるというのは幻想にすぎない。単に、使えないゴミを作っているだけなのかもしれないのだ。

結局、差分プログラミングは、汎化階層、構造継承、機能継承、制約継承などが混沌として存在する中で、オブジェクトの決定に関する指針を与えていないことが問題なのである。

３．２　性質合成を目指したオブジェクト決定（mixin）

３．２．１　mixin

前述（２．３節）のようにカーネライズとOOPはある部分で矛盾する。なぜなら、カーネライズはモノ単位の分解というよりも機能単位の分解そのものだからである。

しかし、最初から機能/性質をオブジェクトとしてとらえ、カーネライズとOOPを両立する可能性を持った手法(mixin)も存在する。

mixinではいくつかの機能をmixinクラスとして予め準備しておき、それらを多重継承することで目的のクラスを合成するといったプログラミングスタイルを採る。この場合の継承は機能の合成（重ね合わせ）を目的として使用される。たとえばread-streamとwrite-streamを重ねてread-write-streamを作成するようなイメージである。mixinはFlavors上で培われた実績を持ち、CLOS(Common Lisp Object System[5][6])上で広く用いられているプログラミング技法である。

３．２．２　メリット

mixinでは機能/性質単位のクラス作成を行うため、オブジェクトの決定が比較的容易である。また、各mixinクラスはメソッド結合の利用によりクラス間の独立性を高めることが可能なので、ライブラリ化を進め易い。

（１）mixinとライブラリ化

mixinのイメージを図５に示す。

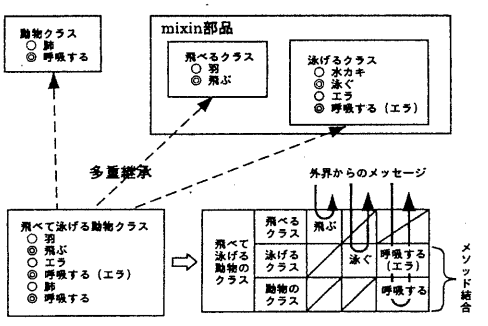

図５ mixinによるライブラリ化のイメージ

この例ではmixinの部品である「飛べるクラス」と「泳げるクラス」を多重継承して「飛べて泳げる動物のクラス」を合成している。このとき、「呼吸する」というメソッドが複数のクラスで重複しているためCLOSではメソッド結合が発生し（通常は）両方のメソッドを実行する。ここで、各mixin部品はある基本構造（たとえば動物クラス）を仮定しているものの、部品間は独立して存在するためライブラリ化には好都合である。

（２）RDBMSへの適用イメージ

これをRDBMSの構造に適用した場合には図６のようになると考えられる。

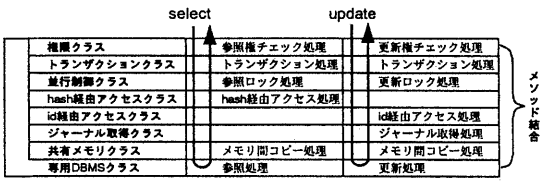

図６ mixinによるRDBMSライブラリ構成のイメージ

RDBMSはいくつかのmixinクラスを合成することで作成され、RDBMSへの問合せはメソッド結合で処理される。また、不要な部品は他に影響を与えることなく削除が可能で、たとえば、APが必ずシングルプロセスであるならば「並行制御クラス」を除くことで、専用のDBMSを生成することができるようになる。（逆に追加も可能）

ただし今回、mixinは実際に適用したわけではないのでこの図はあくまでイメージ図として捉えてほしい。

３．２．３　デメリット

ところが、mixinはOOPの標準的手法とは言えず、おそらく保守的なOOPの定義には納めることができない手法である。

現に、CLOSでは、他クラスのスロットに作用するためカプセル化は不完全、多重メソッド（総称関数）のためメソッドはクラスに属さない、といった主流のOOPLとは矛盾する特徴を持っているし、メソッド結合、メタオブジェクトといった通常のOOPLにはない特徴もたくさん存在する。特にC++ v2.1でのmixin実現はFORTRANで実現する継承にも似た不自然さ（拡張性の悪さ）を持つため、実用的には実現不可能である。また、CLOS上での高速処理向けDBMSの実現は現状では速度の面でも難しい。

３．３　取り替え口のためのオブジェクト決定

これまでの２つの使い方（特殊化、合成）の他に継承はポリモルフィズムの実現手段としての使い方がある。

３．３．１　ポリモルフィズム

ポリモルフィズムとは、同一のメッセージに対して、そのメッセージを受けたオブジェクト毎に反応が異なる

ことをいう。

型チェックの強い言語では、クラスC1とC2が同一の親クラスPを持つとき、それらは親クラスPと入れ替わることができる。このときポリモルフィズムが発生する。

ポリモルフィズムはOOPの３つの特徴のうち、その効果を一番理解しにくい概念であるため、OOADなどでも積極的には支援していない場合が多い。

３．３．２　メリット

ポリモルフィズムによる分岐では動的結合により分岐の隠蔽（switch文が消える）が可能になり、分岐元と分岐先の独立性の実現が容易になる。その例を図７に示す。

(a) 通常(非OOP)の分岐

```
01 void access_method_hash(...);
02 void access_method_btree(...);
03 void access(int type_access_method, ...)
04 {
05    switch( type_access_method ) {
06       case HASH:
07          access_method_hash(...); break;
08       case BTREE:
09          access_method_btree(...); break;
10       ...
11    }
12 }
...
20 access( HASH, ...);
```

(b) ポリモルフィズムによる分岐

```
01 class access_method_base {
02    virtual get_id(...) {}
03 };

04 class access_method_hash : public access_method_base {
05    get_id(...);
06 };

07 class access_method_btree : public access_method_base {
08    get_id(...);
09 };

10 class user_data {
11    access_method_base *access;
12    user_data( access_method_base *x, ...);
13    void get_id(...) {
14       access->get_id(...);
15    }
16 };
...
20 access_method_hash hash;
21 user_data COL1( &hash, ...);
22
23 COL1.get_id(...);
```

図７　分岐の比較

たとえば、動的ハッシュ(access_method_dyn_hash)を追加する場合に、従来の実装方法ではaccess関数の中のcase文の追加が必要であるのに対し、ポリモルフィズムによる分岐の場合にはaccess_method_baseを親クラスとする独立なクラスaccess_method_dyn_hashを追加し、user_dataの中にセットするだけでよい。

そこで、この独立性をライブラリの構築に利用することを考え、それを取り替え口の実現を目指したライブラリ化と呼ぶことにする。ポリモルフィズムを利用した取り替え口は以下の２種類のクラスによって実現される。

【抽象クラス】

- 入替え/拡張が予想される部分のプロトコルを記述したクラス。（図７(b)の01～03行）

【具体クラス】

- 抽象クラスに対応する具体的実装を記述したクラス。（図７(b)の04～09行）

このとき具体クラスは常に抽象クラスの子クラスとなり、継承は単段でしか使われない。

なお、取り替えの単位は機能に限らずデータに対しても可能である。（この場合は従来の抽象データ型である）

取り替え口を実現するときのプログラム構造は以下のようになる。

- 抽象クラスのみを用いて構成したプログラム（スケルトンと呼ぶ）。システムの中で拡張に対して安定な部分の構造がスケルトン（骨格構造）となる。この安定な部分が機能かデータかは問わない。
- 各抽象クラスに対応する具体クラス群。
- 各抽象オブジェクトと各具体オブジェクトのバインドを行うAP（問題領域）依存部分。

スケルトンにより具体的な機能やデータ構造を隠蔽したシステム設計が可能になる。そして、抽象クラスと具体クラスの分離により、機能やデータ型の変更/入れ替えに強い構造となる。さらに、具体クラスをライブラリとして蓄積することが可能になる。

３．３．３　デメリット

このポリモルフィズムを使うことで、動的結合/分岐が発生する。つまり、ソースコード上での静的な分岐追跡が不可能になり、（goto文のように）ソースコードの可読性を落とす可能性がある。これはTOOLによる支援が不可欠である。

また、動的結合のため実行速度は当然遅くなる。

３．４　各継承利用方法の比較

上記の３つの継承の使い方の拡張性と再利用性への効果をまとめると表１のようになる。

表１　継承の利用方法の比較

		特殊化（差分プログラミング）	合成（mixin）	取り替え口
object単位		・既存のクラスをカスタマイズしたもの	・合成して使用するための性質/特性/機能	・システム内で取り替えが発生する部分
拡張性	機能	・差分プログラミングによりクラス拡張	・mixin部品の追加/交換	・具体機能オブジェクトの追加/交換
	データ	・差分プログラミングによりクラス拡張		・具体データオブジェクトの追加/交換
再利用性／ライブラリ	利用	・汎用のライブラリのカスタマイズ	・mixin部品を多重継承して合成	・抽象クラスによりシステムを構成 ・抽象オブジェクトに具体オブジェクトをバインド
	カスタマイズ	・差分の記述によりクラス作成	・mixin部品の組合せを変更 ・mixin部品の作成	・具体クラスの作成
	構築	×汎用クラスの構築が困難	○メソッド結合により部品の独立化が可能	○抽象クラスに基づく具体クラスの整理が可能
速度		○悪化なし	×メソッド結合が遅い ×CLOS処理系が遅い	×動的結合が遅い

特殊化としての差分プログラミングは、ライブラリ等の蓄積を考えずに取り合えず動くシステムを高速に作る

（ラピッドプロトタイピング）という面では非常に有効である。

それに対し、合成や取り替え口として継承を使う場合は、動的な結合などの影響により実行速度が遅くなるものの、クラスライブラリの構築がより容易である。

4　アプローチと実装

今回我々が試作したRDBMSの構造を図８に示す。

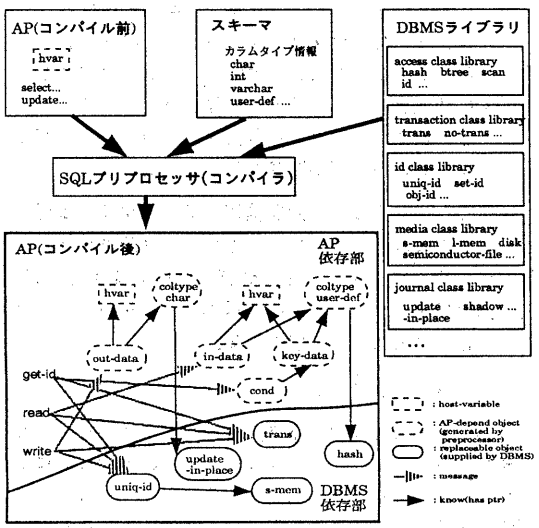

図８ 取り替え口を実現するRDBMSの構成

以下ではその構造に至った理由を示す。

4．1　継承の利用について

従来のオブジェクト指向プログラミングでは差分プログラミングが重視されてきた。しかし、我々の目指すところは再利用可能なライブラリの蓄積であり、領域依存クラスを最小とするようなRDBMS開発技法である。ところが差分プログラミングには、３．１．２節で示したような多くの問題がある。よって、差分プログラミングを重視した設計は望ましくないとの結論に達した。

また、今回のRDBMSでは高速性の実現も目指しているため、開発言語はCLOSではなくC++ v2.1としなければならなかった。よって、継承の利用方法は３．３節で示した「取り替え口の実現」をとった。

この方法では変更が予測される機能やデータに着目してオブジェクト化を行う。そこで我々は図８中のDBMSライブラリを抽象機能クラスとして選んだ。さらに、ホスト変数のオブジェクト化のためにin-data,out-data,key-dataというデータクラス、サブセットSQLの検索条件記述のためにcondという検索条件クラスをSQLプリプロセッサが生成するようにした。

4．2　所有構造の利用について

継承ほどには目新しさが無いものの、オブジェクト指向において所有構造(has-a)の決定は重要視されている。

オブジェクト指向の最大の特徴は現実世界のモデル化の容易性であった。よって、当然のように設計者は気軽に対象領域の所有構造をプログラムに反映してしまう。（多くのOOADでは積極的にそれを推奨している。）

しかし、拡張性や再利用性を考えれば、この所有構造の決定でも問題が生じる。たとえば、a:会社、b:社員、とするとき、以下のいずれがよりオブジェクト指向らしい構造なのであろうか。そして拡張性や再利用性への影響はどうなのであろうか？

- a:会社はb:社員を持っているため、会社の中に社員リストを含めた構造とする。
- b:社員は自分の所属するa:会社を知っているため、各社員が自分の所属する会社に関する情報を持つ。（ここで「構造として持つ」ことはポインタを持つことと考えても構わない。）

同様の問題は、DB/Tableがデータを管理しているか、データが自分の所属するDB/Tableを知っているか、という構造決定の場合にも発生する。

RDBMSの現実の所有構造を直感的にモデル化すれば先の図１に示した構造になる。しかし、機能やデータの拡張では、その所有構造そのものが変化することが多い。つまり、この構造に基づいたプログラムは既存のRDBMSを再現する場合には良いが、今後発生するDBMS構造の変化に対応するには不十分である。また、ライブラリも構造に依存しすぎた構成になってしまう。

そこで、我々は現実をモデル化したhas構造より、抽象的なknow構造を重視し、その構造をポインタで所有することとした。つまり、個々のオブジェクトは自分が知っているべき情報（他のオブジェクト）へのポインタを持つ。このとき、相手のオブジェクトももちろん抽象化されており、相手はデータと機能のいずれでも構わない。

4．3　パラメータの利用について

取り替え口の実現では、機能の抽象化と共にデータの抽象化も重視する。つまり、従来のポリモルフィズム設計法ではメッセージ名の同一化のみを主張している場合が多いが、取り替え口のためには、同時にパラメータや戻り値の同一化も必要となる。

しかし、これらのオブジェクトを抽象化することで新たな問題が生じる。

たとえば、データのIDを取得するメソッドには、アクセス法、カラム情報、ホスト変数情報、検索条件、トランザクション管理法、データIDなどがパラメータとして必要になる。このとき、この中でデータを取得するメソッドを所有するオブジェクトはどのオブジェクトなのだ

ろうか。どのオブジェクトが所有すれば抽象機能として汎用化が図れるのだろうか？

各オブジェクト間には主従関係などないので、どれにしても良いようで非常に困ってしまう。

この問題を単純にすると以下のようになる。

【問題】２つのオブジェクトa,bがあり、aの内容をbに複写するメソッドをcopyとする。このとき、メソッドcopyはどのオブジェクトに属するべきか？

解答として以下の３つが考えられる。

（解１）aまたはbがメソッドcopyを持つ。
（解２）aとbの両方に持つ。
（解３）aとbとは独立に存在する。

このようにパラメータの抽象化を充分行うと、結局、パラメータと主オブジェクト（メソッドを所有するオブジェクト）の区別が単純にはできなくなり、１つのメッセージが複数のオブジェクトに対して送られ、それらが協調してメッセージを解釈するといった関係が見えてくる。

つまり、上の例では、オブジェクトaがgetメソッドを、オブジェクトbがsetメソッドを提供し、抽象的なcopyメソッドはそのgetとsetを用いて実体のコピーを行うといった感じになる。

我々の理想解は、CLOSの多重メソッドであったが、C++での実装を検討し、解３を採った。つまり図８におけるget-id, read, writeといったメソッドはそれ専用のオブジェクトが所有する構造となっている。（実際には、AP依存部のcond, in-data, out-dataがそれらを所有する。）

なお、パラメータの同一化のために、オブジェクトの差異は生成時に問題領域依存部分で積極的に吸収するようにしている。

５　考察

以下では、実際のRDBMSの実装結果をふまえ、今回のアプローチについての考察を示す。

（ａ）取り替え口はオブジェクト指向技法なのか？

この取り替え口の実現は現在のオブジェクト指向設計法の主流ではない。さらに、データより機能の抽象化を重視しているため、もしかしたらオブジェクト指向設計法ではないのかもしれない。

【差異】
- 問題領域の構造を素直に反映しない。
- モデルとしてメソッドはオブジェクトの外に存在することがある。
- 差分プログラミングによるクラスカスタマイズを使わない。

【共通点】
- 抽象化を目指そうとする方向
- クラスライブラリを蓄積しようとする方向
- ポリモルフィズムを活用しようとする方向

だが、我々の目的は拡張性や再利用性の向上であって、OOAD/OOPではない。よって、これがOOAD/OOPであるか否かが我々の興味の本質ではない。

（ｂ）何がカーネルなのか？

ライブラリは図８に示した。しかし、我々の設計ではカーネルは存在していないという方が正しい。つまり、システムの構成方法は、不変なカーネルの周りにライブラリを接続するという構成方法ではなく、抽象部品（抽象DBMS機能クラス）を組み合わせて領域依存のスケルトンを構成するという方法を採る。

（ｃ）拡張性は実現されたのか？

拡張性という点から見ると差分プログラミングは非常に強力であった。しかし、それは継ぎ接ぎだらけの保守困難なシステムを作る手法であった。

それに対し、我々の採った手法での拡張への対処は部品の組み合わせ方の変更と部品の交換である。この場合、全体構成を意識した拡張になるため、拡張に対し常に全体の整合性が保持可能な、より健全な拡張法であると考えている。

（ｄ）再利用性は実現されたのか？

再利用性については、差分プログラミングよりは遙かに向上していると考えられる。なぜなら、今回の方法は部品の抽象化を行うことを中心とする設計法であるためである。よって、スケルトンの再構成による再利用が容易である。また、もし機能の追加があって抽象クラス定義の変更があっても、抽象クラスは継承が浅いので、他への影響を深く考えずに修正することが可能である。

（ｅ）継承の扱いは混用できないか？

差分プログラミングは取り替え口の実現と混在が可能である。しかし、我々は今回ほとんど使用しなかった。

これらは場合によって使い分けるのが賢明な利用法なのかもしれないが、やはり、クラス構造を破壊し利用指針が明確でない差分プログラミングは極めて危険な技法である。よって、継承は主にポリモルフィズムの為だけに使用し、is-a構造の反映と言った問題領域構造のモデル化を目的とするのではなく、再利用可能な部品の整理に使うこととした。

これにより場当たり的な深い継承の発生によるクラスライブラリの迷宮化が防げたと考えている。

（ｆ）know構造は有効なのか？

今回、所有関係はknowに注目して設計を行った。しかし、このようにhas構造をknow構造にしても依存関係の存在は本質的には変わらない。それでも、know構造にしたことによる効果は以下の点に現れると考える。

- 現実構造とは別に、小さいもの（または、不変そうなもの）を中心に全体を組み立てることが可能になる。これにより、部品クラスの組み立て方といった問題領域依存の部分を、ライブラリから分離することができ、また、構造の変化にも対処しやすくなる。
- 大きなものがその中に含む小さなものまで面倒をみてしまうという依存関係が防げる。それに伴い、小さな情報でも自然にオブジェクト化され、自分のことは自分が知っているが、相手のことは相手に聞くという態度が明確になる。

（ｇ）速度はどうか？

取り替え口を実現すると、機能抽象化とデータ抽象化で動的結合が発生するので実行速度は遅くなる。

単純な検索処理において、どの程度のオーバーヘッドが生じるかを計測した結果を表２に示す。

表2 取り替え口実現によるオーバヘッド

	機能	処理時間比	
取り替え口実現前	検索部分		100.0
取り替え口実現後	インスタンス生成部分	14.4	120.5
	具体オブジェクトバインド部分	1.7	
	検索部分	104.3	

全体として20%以上の速度低下が見られる。
速度低下の原因には以下のような理由がある。

（１）インスタンス生成
（２）具体オブジェクトバインド
（３）動的結合

しかし、（１）、（２）はAP起動時に行えばいいだけの処理なので、これらの処理を最適化によって除くと4%程度の速度低下での実現が可能である。（３）の処理はオブジェクト指向の本質的な処理ではあるが、取り替え口の実現の方法では、一度抽象オブジェクトに具体オブジェクトがバインドされると以後のバインド変更は行われないことが多い。よって、この部分の高速化は今後の言語処理系の最適化に期待する機能である。

なお、今回の実装において、ロック機能は速度への影響が大きすぎるため取り替え可能とはしていない。

（ｈ）その他、周辺の問題/障害の存在

RDBMSだけをオブジェクト指向で設計すると、外部とのインピーダンスミスマッチが発生する。そのため、RDBMS内部に外界をオブジェクト化するインタフェース部分（たとえば、ユーザAPにおいてホスト変数やSQLCAのオブジェクト化や、OS資源（共有メモリやファイル）のオブジェクト化など）が必要になった。

６　まとめ

本稿では、オブジェクト指向技法の一つの適用例として、我々のRDBMS設計に対する適用法とその考察を述べた。

我々は拡張性と再利用性の向上のためのDBMSのカーネライズとライブラリ化を目的とし、はじめに特殊化（差分プログラミング）に基づくオブジェクト構造の設計を検討した。しかし、それはライブラリの汎用化に対して多くの問題を内在しており、結局、上手く活かしきることができなかった。

そこで我々は、それに代わるものとして、取り替え口の実現を中心としたオブジェクト指向技法をRDBMS設計に適用した。これによりRDBMS機能のクラスライブラリ化が可能になり、RDBMSの拡張性と再利用性が実現されたと考えている。

なお、本稿で示した我々の採ったアプローチの一部はC++ v2.1に依存している。また、高速性の実現のために強い制約を受けている。よって、取り替え口の実現を設計法の一般論にするには、さらに広い範囲への適用検討が必要である。

今後は、取り替え口の動的結合という特性を活かし、動的拡張性を備えた高速DBMSの構築技法を検討していきたい。

謝辞

本検討においてご指導いただいた当研究所ＤＢ部 田中ＧＬ、井上ＧＬに感謝します。

参考文献

[1] B.Meyer,"Object-Oriented Software Construction", Prentice-Hall, 1988 （邦訳 二木, オブジェクト指向入門, アスキー）

[2] 青木 淳,"オブジェクト指向システム分析設計入門", ソフト・リサーチ・センター, 1993

[3] T.Korson, J.D.McGregor, "Understanding Object-Oriented: A Unifying Paradigm", CACM, Vol.33, No.9, Sep.1990

[4] P.Wegner, "Concepts and paradigms of Object-Oriented Programming", ACM OOPS MESSENGER, Vol.1, No.1, Aug.1990

[5] 井田 昌之, 元吉 文男, 大久保 清貴,"Commom Lisp オブジェクトシステム -CLOSとその周辺-", bit別冊, 1989

[6] G.L.Steele Jr.,"COMMON LISP:The Language (2nd ed.)", Digital Press, 1990 （邦訳 共立出版）