クラスの共有と配送に基づく オブジェクト指向分散システムの設計と実現

塚本享治[†] 濱崎陽一[†]
音川英之[††] 西岡利博[†††]

ネットワークと計算機の普及によって，数多くの計算機がネットワークによって接続された分散環境が広く使われるようになってきた．しかし，分散環境で動作するアプリケーションの開発には，OS等の広範な知識が要求されるとともに，単一計算機上のアプリケーションにない，(1) データの移動によるデータとプログラムの所在の不一致，(2) 局所的な変更で相互運用性が損なわれる，(3) プログラムを共有・再利用することが難しいなどの問題がある．これらの問題を解決するために，オブジェクト指向技術を分散環境に採用し，オブジェクト指向分散環境 OZ++を開発した．本論文では，その基本部分，すなわち，(1) オブジェクトを共有し配送する機能，(2) 必要なクラスを自動的に取り寄せる機能，(3) クラスの新旧バージョンを混在させて開発・実行する機能，に焦点を絞り，その設計思想，実現方法，性能評価について述べる．クラスのインタフェースに着目したバージョン管理と新旧バージョンの混在利用に工夫がされている．

The Design and Implementation of an Object-Oriented Distributed System Based on Sharing and Transferring of Classes

Michiharu Tsukamoto,[†] Yoichi Hamazaki,[†]
Hideyuki Otokawa[††] and Toshihiro Nishioka[†††]

Distributed environments with many computers and networks are becoming popular nowadays. However, the development of application programs for a distributed environment is harder than for a single machine. The reasons are (1)programs which process data do not always exist where the data are transferred, (2)programs which are modified locally are not inter-operable with others, (3)sharing or reusing programs is almost impossible, etc. To solve these problems, we developed an object-oriented distributed environment OZ++. The topics of this paper are the design, implementation and preliminary evaluation of the OZ++ system. This paper describes in detail the basic features of OZ++: (1)objects are shared and transferred among machines, (2)classes are delivered on demand to where they are required, (3)classes of different versions can be developed and executed at the same time. Version management based on interfaces of classes is one of outstanding features of OZ++.

1. まえがき

ネットワークの普及と計算機のダウンサイジング化により，数多くの計算機がネットワークを介して接続された分散環境が広く普及している．また，WWW/Mosaic の普及によってインターネットを使った情報の発信・共有・流通が急速に広まってきた．このような状況にあって，分散環境で動作する分散アプリケーションプログラムの需要はますます大きくなっている．

しかし，分散アプリケーションプログラムは1台の計算機で動作するプログラムに比べて複雑であり開発が難しい．また，広く利用されるプログラムほど，バグ修正，機能拡張，性能向上等が頻繁に行われるにもかかわらず，これを周知徹底するのが難しい．さらに，離れた場所で複数の利用者が使用する環境では，様々なバージョンのプログラムで作られたデータが混在するが，それらに必要なバージョンのプログラムを取り

† 電子技術総合研究所
Electrotechnical Laboratory
†† シャープ株式会社
Sharp Corporation
††† 株式会社 三菱総合研究所
Mitsubishi Research Institute, Inc.

そろえておくのも難しい．

データとプログラムを一体として扱うことができ，しかもプログラム部品の再利用に適し，さらに分散システムにも適しているとされるオブジェクト指向技術を採用して，次のようなシステムを実現すれば，上記の問題が解決できるものと考えた．

(1) ネットワーク上に存在するプログラム部品（クラス）を相互に共有・再利用して，新たなプログラムを開発する．
(2) 開発したクラスは，インタフェースとセマンティックスが同じならば，そのクラスを利用するクラスを再コンパイルしないで利用できる．
(3) 受信したオブジェクトに必要なクラスを所持していなければ，ネットワークを使って取り寄せる．
(4) クラスが変更拡張されると様々なバージョンができるが，それらを混在して動作させることができる．
(5) クラスを遠隔から取り寄せても，セキュリティ上の問題が発生しない．

この思想を実現するものが，本論文で述べる「オブジェクト指向分散環境 OZ++」である．OZ++システムは，オブジェクトを実行する環境，クラスを含むオブジェクトを管理する環境[1]，プログラミング言語と開発環境[2]，等から構成されているが，本論文では，基本部分である，ネットワーク上でオブジェクトとクラスを共有し配送する方法，それを実現した実行システムと付随する管理システム，および性能評価について述べる．セキュリティに関しては本論文では触れない．

以下，上記の思想を具体化するためのモデルを，分散システムのためのオブジェクトモデル（2章），分散システム上におけるクラスの共有と配送のための管理モデル（3章），に分けて述べる．4章以降は，その実現に関するものであり，4章でシステムの全体概要，5章でオブジェクト配送とそれにともなうクラス配送，およびそれに付随した管理を行う実行システムの実現について述べ，6章で性能評価，7章で関連研究に触れる．

2. 分散オブジェクトモデル

分散システムの構成要素であるオブジェクトとその実行モデルについて述べる．

2.1 オブジェクトモデル

分散システムをオブジェクトで構成する際，構成単位であるオブジェクトの性格づけに関して，機能記述に密着した粒度の尺度と，実行方法に密着した構成の尺度がある[3]．オブジェクトの内部構成や実行方法でなく，提供する機能だけに着目して，システムを構成し利用できるようにするのが望ましい．そこで，実装の観点から能動オブジェクトと受動オブジェクト[3]に分けるのでなく，機能の観点から粒度を工夫する．

2.1.1 粒度の異なるオブジェクト

粒度が大きければ，オブジェクトのメソッドの実行によって比較的大量の命令が実行され，他のオブジェクトとの相互作用は比較的小さなものとなる．また，システムのデータに対する制御と保護もしやすい．粒度を小さくし，Smalltalk-80 のように数や文字までオブジェクトとするとその数が膨大なものとなるので，それらはオブジェクトとして扱わないこととした．一方，サービスを提供する大粒度のオブジェクトだけに限定すると，CORBA のようにオブジェクト間で交換されるものがオブジェクトでなくなり，オブジェクト指向の長所が生かせない．そこで，大粒度オブジェクトと，大粒度オブジェクト間で交換され大粒度オブジェクトの部品となる中粒度オブジェクトの2種類を採用した．

大粒度オブジェクトは分散システム上で一意に区別する必要があるので，グローバルオブジェクトと呼び，分散システム上で一意な識別子 OID（Object ID）を付す．一方，グローバルオブジェクトの部品である中粒度オブジェクトはローカルオブジェクトと呼び，OID を持たない．これは，ローカルオブジェクトは，いずれか1つのグローバルオブジェクトに属して，複数のグローバルオブジェクトに属さず，しかも，ポインタで直接参照され，他のグローバルオブジェクトおよびそれに属するローカルオブジェクトからは直接参照されることはないためである．1つのグローバルオブジェクトとそれに属するローカルオブジェクト群を合わせたものをセルと呼ぶ（**図1**）．セルの内部は密接な相互作用ができることを前提としており，これが細分されて分散配置されることはない．

グローバル/ローカルはクラスの性質ではなく，むしろオブジェクトの性質であるので，クラスに対してではなく，インスタンス変数の属性としてグローバル/ローカルを指定することにした（図3，Class C）．グローバル/ローカルの違いによって，実行モデルが異なるので，グローバル/ローカルの違いは，言語上では別の型として扱っている．

2.1.2 メッセージとしてのオブジェクト

オブジェクトはメソッドが呼ばれると実行して値を戻す．メソッドの引数および戻値に現われうるものを，

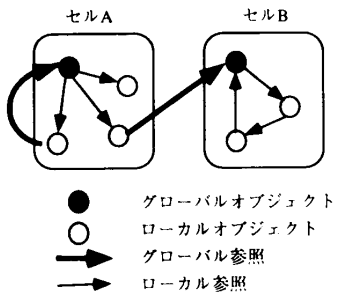

図1 オブジェクトとセル
Fig. 1 Object and cell.

文字，数，ストラクチャ，配列などの非オブジェクト，あるいはオブジェクトへの参照（ポインタあるいはOID）だけに限ると，オブジェクトは生成された場所にとどまるため，たとえEmeraldのように直接参照しているオブジェクトを移動（call by move）[4),5)]しても，分散システム上では効率の良い実行は望めない．また，ローカルオブジェクトへの参照が他のセルのグローバルオブジェクトに渡されると先に述べたローカルオブジェクトはどれか1つのグローバルオブジェクトに属するという性質も満たさなくなる．

そこで，引数と戻値に現われるオブジェクトは次のように渡すことにした．ローカルオブジェクトを引数や戻値として受渡しする場合，宛先がローカルオブジェクトであれば参照渡し（call by reference）とする．また，宛先がグローバルオブジェクトであれば，渡されるローカルオブジェクトとそこから参照されているローカルオブジェクトをすべて，参照関係を保ったまま，ポインタを再帰的にたどってコピーし，値渡し（call by value）する．一方，グローバルオブジェクトを受渡しする場合は，そのOIDを渡す（call by reference）．これにより，ローカルオブジェクトはいずれか1つのグローバルオブジェクトが代表するセルにしか属さないことが保証される．

2.2 実行モデル

2.2.1 スレッドとプロセス

オブジェクトのメソッドはスレッドによって実行する．ローカルオブジェクトのメソッド呼び出しは呼び出す側のスレッドで実行する．一方，グローバルオブジェクトのメソッド呼び出しが発生すると，戻値または例外が戻されるまでスレッドの実行をブロックし，宛先のグローバルオブジェクトにスレッドを生成してそのメソッドを実行する．このように，メソッド呼び出しは宛先がグローバルオブジェクトかローカルオブジェクトかにかかわらず，一連の連鎖となる．このメソッド呼び出しの連鎖をプロセスと呼ぶ．

言語上現われる計算の制御対象は，実行の単位であるスレッドでなく，計算の流れの単位であるプロセスである．そのため，プロセスを扱うためのプロセス派生型，およびそれに対する演算子として，fork（プロセスの生成），join（プロセスの終了を待って結果を得る），detach（プロセスの終了を待たないようにする），kill（プロセスの終了要求）を言語に導入した．

```
class ProducerConsumer {
  constructor: New;
  public: Produce,Consume,Consumed;
  condition empty,filled; /* condition var */
  int value,valid;
  void New() {
    valid = 0;
    consumed = 0;
  }
  int Consume() : locked {
    int v;
    while(!valid){ /* wait until filled */
      wait filled;
    }
    v = value;
    valid = 0;
    signal empty;
    return v;
  }
  void Producer(int v) : locked {
    while(valid) { /* wait until empty */
      wait empty;
    }
    value = v;
    valid = 1;
    signal filled;
    return;
  }
  int Consumed { /* not locked method */
    return !valid;
  }
}
```

図2 生産者消費者プログラム例
Fig. 2 Sample of a producer-consumer problem.

2.2.2 同期と相互排除

1つのオブジェクトが同時に呼ばれるので，スレッド間の同期と相互排除の機構が必要である．OZ++では，同期と排他制御の機構としてモニタ[6)]を採用した．OZ++言語による生産者消費者問題のプログラムを図2に示す．モニタでは，クラスを単位として指定するため，そのすべてのメソッドにlockedが指定されるのと同じである．一方，OZ++では，排他制御の必要なメソッドごとにlockedを指定する．これにより，排他制御が不要なメソッドの実行の並列度が高められる．lockedメソッドでは，そのオブジェクトの実行権を獲得したのちメソッドの実行が始まり，実行終了後に実行権を解放する．

スレッド間の同期は条件変数によって行う．条件変数に対する操作としては，スレッドを待ち状態にするwait 操作と，スレッドの待ち状態を解除する signal 操作がある．これらの操作は locked メソッドでのみ実行可能である．wait 操作を行ったスレッドは，獲得している実行権を解放し，その条件変数に対応する実行待ち状態となる．一方 signal 操作を行うと，その条件変数で待ち状態になっているスレッドの実行待ち状態を解除する．解除されたスレッドが実際に動作を再開するのは，次にスケジュールされ，しかもそのオブジェクトの実行権が獲得されたときである．

多重継承を行う場合には，同期と相互排除に関する設計は慎重に行う必要がある[7]．たとえば，それぞれ独自に排他制御を実装している 2 つのクラスを多重継承してクラスを作ると，親クラスのモニタが子クラスで共有されるので，親クラス間の独立性が損なわれる．OZ++ では，このような場合に，サービスのインタフェースを規定するクラスと，そのサービスを実現するクラス（そこでは相互排除の方針も実現されている）とを分離して設計することを推奨している．インタフェースだけを継承することで，それぞれの親クラスで実現されていた相互排除の方針が互いに影響を与えることなく，融合したインタフェースを提供できる．

2.2.3 例外処理

プロセスにおける大域脱出の手段として例外処理を導入した．プロセスを形成するメソッド呼び出しの連鎖で実行中のメソッドが例外を発生すると，その連鎖内にあるメソッドで，その例外を補足すると宣言している，最も近いメソッドに制御が移る．

3. クラス管理モデル

1 章に掲げた (1)～(4) を実現するモデルを本章でのべる．最初に，以後使用するプログラム例を**図 3** に示す．このプログラムは，クラス A をクラス B が継承し，クラス C において A と B を利用する恣意的なものである．

3.1 バージョン管理

3.1.1 インタフェース

オブジェクト指向プログラミングにおけるクラスは，インタフェースとそれに対する実装の 2 つの機能を持つと考えられる．このうちプログラミングで直接利用するのはクラスが公開しているインタフェースのみである．そのため，利用しているクラスに何らかの変更があっても，セマンティックスとインタフェースが変わらなければ，プログラミング上は影響を受けないようにすることができる．ここで，図 3 のクラス B のようにA を継承して利用する場合 (b) には，B は A のパブリックインタフェースとプロテクティッドインタフェースを利用できる．一方，クラス C のように A のインスタンスを参照し，メソッドを起動するだけの場合 (c) には，A のパブリックインタフェースのみが利用できる．さらに OZ++では，あるクラスから利用できない部分が変更されても影響を受けない．すなわち，A を継承している B では，A のパブリックおよびプロテクディッドインタフェース以外の部分（すなわち，実装）が変更されても再コンパイルの必要はない．同様に，C では，A のパブリックインタフェース以外の部分が変更されても再コンパイルの必要はない．

```
class A {
  public: foo;                          /* (a) */
  protected: bar,valA;                  /* (a) */
  int valA; /* protected instance var */
  int foo() { /* public method */
       ...
  }
  void bar(int x,int y){ /* protected method */
       ... /* slow algorithm */
  }
}
class B : A { /* B inherits A */  /* (b) */
  public foo;
  int m1(int x) {
    bar(x + x , x + valA + valA);
    return foo();
  }
}
class C {
  constructor New;
  global B b; /* instance var of global B*/
  A a;        /* instance var of local A */
  New(){
    a=>New();
    b=>New();
  }
  int m2() {
    return a->foo();                    /* (c) */
  }
}
```

図 3 クラスのインタフェース例
Fig. 3 Sample of class interfaces.

3.1.2 バージョン

インタフェースと実装のバージョンを管理し，しかも新旧のバージョンを混在して利用できるようにすることで，柔軟なソフトウェア開発が可能となる．図 3 のクラス A とクラス B を並行して開発する場合，まず A のプロテクティッドインタフェースが決まれば，B の開発者はそのインタフェースを利用して独立に B の開発を進めることができる．その間，A の開発者は，

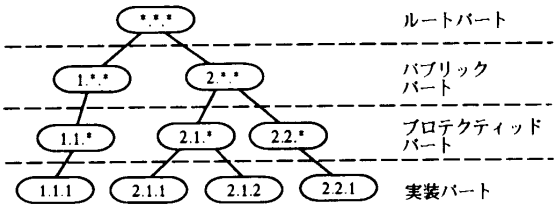

図4 クラスのバージョン木
Fig. 4 Version tree of a class.

インタフェース以外の部分を自由に変更して，Aの開発を進めることができる．またAを利用するクラスCの開発のためにAのインタフェースを変更しなければならない事態が生じた場合でも，Aの開発者はインタフェースのバージョンを更新することにより，以前のインタフェースを利用しているBの開発者に影響を与えないことができる．

OZ++では各クラスごとに，パブリックインタフェース，プロテクティッドインタフェース，および実装のバージョン管理を行い，それぞれをクラスのパブリックパート，プロテクティッドパート，実装パートと呼ぶ．1つのパブリックパートに対して複数のプロテクティッドパートが存在し，また1つのプロテクティッドパートはプロテクティッドインタフェースが共通な複数の実装パートを持つ．したがってこれらの3つのパートは図4のような上下関係を持つ．クラスのバージョンは，パブリック，プロテクティッド，実装，の各パートでふられる通番からなる3つ組数で表現され，図4の3つ組数はどこまで確定しているかを表す（*は未確定）．パブリックパートの上位のパートにはルートパートがあり，これがクラスのすべてのバージョンを代表するものである．

3.2 コンフィギュレーション管理

3.2.1 コンフィギュレーションとは

クラスの実装パートのバージョンの確定をそのクラスのインスタンスの生成時点まで遅延させることもできる．図3の例で，クラスBのインスタンスを生成する場合，コンパイル時にはBのパブリックパートのバージョンだけが確定しており，またそのBのバージョンがコンパイルされた時点ではBが継承しているクラスAに関してはプロテクティッドパートのバージョンが確定している．したがって，Bのインスタンスを生成するにはBとAの2つの実装のバージョンを確定しなければならない（図5）．

このようにクラスのインスタンス生成時には，そのクラスとそのすべての祖先クラスにわたって，どの実装のバージョンを使用するかを確定しなければならない．そこで，あるクラスとその祖先クラスが使用する

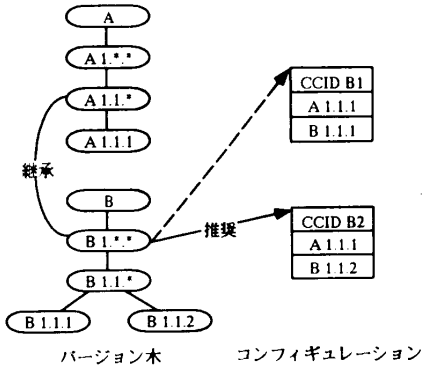

図5 バージョンとコンフィギュレーション
Fig. 5 Version and configulation of classes.

実装のバージョンの組をあらかじめ作成しておき，これをパブリックパートのバージョンとともにクラスに関する情報として管理する．これをコンフィギュレーションと呼ぶ（図5）．

インスタンス生成時にはコンフィギュレーションの指定を行う．一方，コンフィギュレーションを指定しないことも可能であり，そのような場合には，推奨されているコンフィギュレーション（推奨フィギュレーション）が使用される．この推奨コンフィギュレーションはその時点で開発者が最適と考えるバージョンの組合せを指定するものである．この設定はクラスのバージョンの更新とともに変更される可能性がある．

3.2.2 コンフィギュレーションセット

推奨コンフィギュレーションを利用することは柔軟であるが，反面，ソフトウェアの動作保証が完全に行えない．動作保証を行いたい場合には，特定の実装のバージョンからなるコンフィギュレーションを利用することが必要である．そこで，ソフトウェアごとにその実行中のインスタンス生成で利用されるコンフィギュレーションの集合（コンフィギュレーションセットと呼ぶ）を設定することを可能にした．

コンフィギュレーションセットは1つのセル内で生成されたインスタンス間で共有する．他のグローバルオブジェクトのメソッド呼び出しや新たにグローバルオブジェクトを生成する場合には，コンフィギュレーションセットをコピーして伝播させる．

4. OZ++システムモデル

4.1 システム構成の概要

OZ++システムは，オブジェクトの実装とメソッドの実行を行うオブジェクト実行系，オブジェクトによって実現された分散管理系[1]，OZ++言語で記述されたプログラムをコンパイルしてクラスを生成する言語処理系[2]からなる．

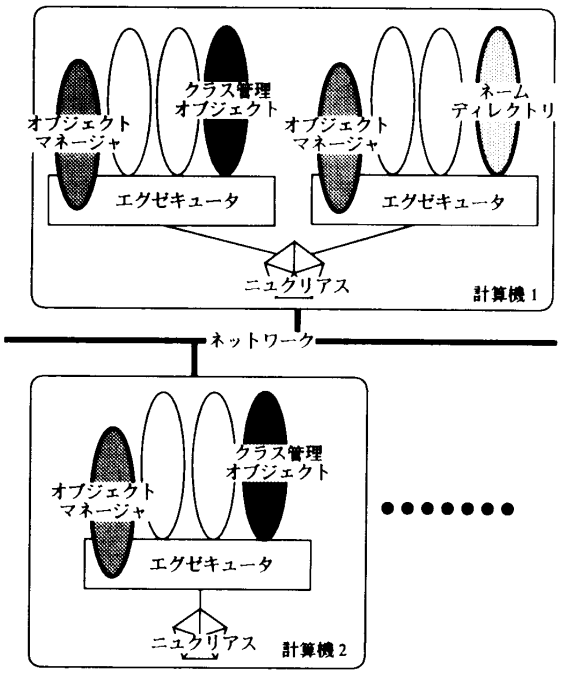

図 6 システムの概要
Fig. 6 Overview of OZ++ system.

オブジェクト実行系はメカニズムのみを提供し，各種の管理ポリシーは管理オブジェクトとして実現する方針で設計した．これにより，オブジェクト実行系を小さくでき，システムの変更・拡張が容易になった．

オブジェクト実行系は分散配置され，オブジェクトの保持，メソッドの実行，オブジェクト間のメソッド起動ならびに引数や戻値のオブジェクトの転送などの機能を実現するエグゼキュータと計算機ごとにエグゼキュータを管理するニュクリアスからなる．

分散管理系はオブジェクトやクラス，名称などの管理を行うオブジェクト群からなり，これらを管理オブジェクトと呼ぶ．管理オブジェクトには，エグゼキュータ上のオブジェクトを管理するオブジェクトマネージャ，クラスを管理するクラス管理オブジェクト，オブジェクトの名称を管理するネームディレクトリやトレーディングディレクトリ，公開されたソフトウェアを管理するカタログなどがある．

各エグゼキュータには，オブジェクトマネージャが1つ存在し，アプリケーションのオブジェクトおよび管理オブジェクトがその上に搭載されて実行される（**図 6**）．

4.2 システム機能の概要

OZ++では，クラスはシステム全体で共有し，必要に応じてネットワークを介してクラスの配送を受ける環境を提供している．そのためにクラスにはシステム全体でユニークな ID（CID: Class ID）をコンパイル時に付与し，CID によってクラスは識別される．CID は実装ばかりではなく，インタフェースの各バージョンやコンフィギュレーションにも付与される．

プログラムの作成時には，ネットワーク上に存在するクラスを親クラスあるいはインスタンス変数として利用し，差分部分を付け加えることにより所望のクラスを作ることができる．OZ++はこのようなクラスの共用・再利用を促進するシステムである．また，オブジェクトを計算機間で転送する場合，受信側に転送したオブジェクトのクラスがあるかどうかどうかを送信側で知ることは困難である．しかし，OZ++では必要に応じてクラスをネットワークを介してロードすることができるので，どこにオブジェクトが転送されても実行することができる．これは，オブジェクトの転送につれて計算機間でそのオブジェクトのクラスが自然に共有されていくことを意味している．

5. 実行システムの実現

実行システムは，オブジェクト実行系と管理オブジェクトによって実現されている．オブジェクトの基本的な動作のうち，クラスの管理などかなりの部分をオブジェクトにより実現している．オブジェクトマネージャは，一般のオブジェクトとオブジェクト実行系との間の仲介を行う．エグゼキュータは，オブジェクトマネージャにランタイムルーチンの機能を提供する．また，オブジェクトマネージャはエグゼキュータからのフォールトを受けて，他の管理オブジェクトと連携しながら要求に応える．

5.1 オブジェクトの構造

インスタンス生成時にそれを構成するクラスの実装が確定するが，継承したクラスのプロテクティッドインスタンス変数を操作するプログラムの実装はコンパイル時に確定しなければならない．そのため，プロテクティッドインスタンス変数とプライベートインスタンス変数を分離して配置し，実装が変わってもプロテクティッドインスタンス変数へのアクセス方法が変化しない構造とした．

オブジェクトはヘッダ部とボディ部からなる．ヘッダ部は全体を構成するクラス数やコンフィギュレーション ID を持つ全体のヘッダと，各クラス部分のパブリックパートの ID やインスタンス変数格納部分であるボディ部へのポインタを持つヘッダからなる．ボディ部は，インスタンス変数領域の大きさ等のアロケーション情報，プロテクティッドインスタンス変数とプライベートインスタンス変数とからなる（**図 7**）．クラス Object はすべてのクラスが暗黙のうちに継承してい

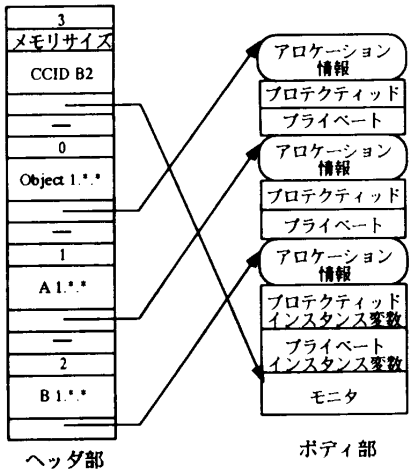

図7 オブジェクトの構造
Fig. 7 Memory structure of an object.

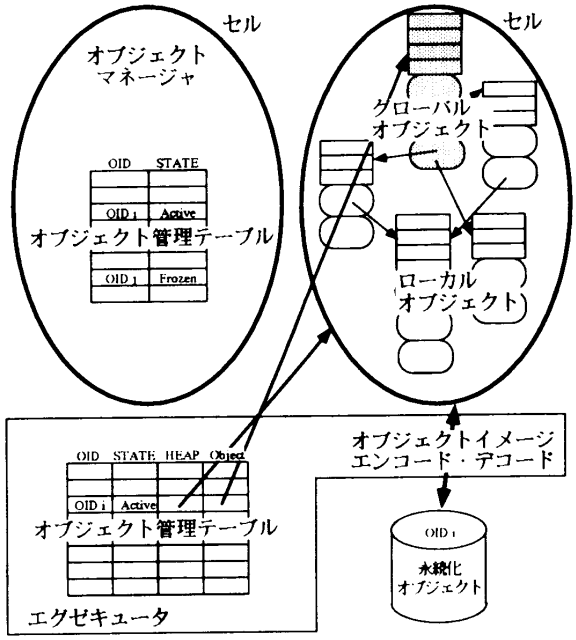

図8 オブジェクト管理情報
Fig. 8 Information for object management.

るクラスである．

オブジェクトはクラスのヘッダのアドレスで参照され，どのクラスのヘッダを参照するかでポリモルフィズムを実現している．図7の例では，オブジェクトがクラスAのオブジェクトとして参照される場合にはクラスAのヘッダが，クラスBのオブジェクトとして参照される場合にはクラスBのヘッダが参照される．グローバルオブジェクトの場合，メモリアドレスではなくOIDで参照される．そのために，実行時にナローイングすることとし，パブリックパートのIDをクラスのヘッダに持たせた．

クラスBのプログラムで親クラスであるクラスAのインスタンス変数にアクセスする際には，1つ上のヘッダからクラスAのボディを求め，その中でプロテクティッドインスタンス変数にアクセスする．継承するクラスは実装にかかわらず同じプロテクティッドバージョンを持ちプロテクティッドインスタンス変数のメモリ上の配置は同じであるので，コンパイル時に親クラスの実装が確定しなくてもアクセス方法を確定することができる．

5.2 オブジェクトの管理

5.2.1 オブジェクト識別子

グローバルオブジェクトを識別するOIDは64 bitであり，サイト識別番号（16 bit），エグゼキュータ識別番号（24 bit），オブジェクト識別番号（24 bit）からなる．階層的に，エグゼキュータ識別番号はサイト内で，また，オブジェクト識別番号はエグゼキュータ内で，それぞれユニークに付番する．さらに，OIDを再利用しないので，OIDをシステム全体でつねに一意にすることができる．オブジェクト識別番号が0であるOIDは，エグゼキュータの識別子として使われる．CIDもOIDと同じく64 bitであり，同様な方法で付番する．

5.2.2 グローバルオブジェクトの管理

グローバルオブジェクトとローカルオブジェクトの構造は同じであるが，セルの代表として参照されるか否かだけが異なる．エグゼキュータはグローバルオブジェクトを管理するオブジェクトテーブルを持っており，グローバルオブジェクトとOIDの対応付けや割り当てられたヒープの管理などを機械的に行う．オブジェクトマネージャもオブジェクト管理テーブルを持つが，これはオブジェクトの状態管理に用いられ，オブジェクトの状態はエグゼキュータの持つテーブルに反映される（図8）．

グローバルオブジェクトには，永続化された二次記憶上にしか実体のない凍結状態，メモリと二次記憶間で推移中の解凍状態，メモリ上で動作可能な活性状態および停止状態などの状態があり，その状態遷移はオブジェクトマネージャが管理する．エグゼキュータは，凍結状態などメソッドの起動が即座にはできない状態のオブジェクトに対するメソッド起動の要求を受付けると，オブジェクトマネージャにフォールトを伝え，オブジェクトが永続化オブジェクトファイルから読み込まれ活性状態になるまでメソッドの起動を遅らせる．

グローバルオブジェクトの生成は，オブジェクトマネージャに対してクラスを指定して要求することにより行い，その際にOIDの付与や新たなヒープの割当を行う．グローバルオブジェクトの削除は，オブジェ

クトマネージャに要求することにより行い，その際にはヒープも解放する．永続化を指定されたグローバルオブジェクトは，オブジェクトマネージャが要求されない限り削除されることはない．

5.2.3 ローカルオブジェクトの管理

ローカルオブジェクトは，それが属するグローバルオブジェクトのヒープ上に生成され，その状態はグローバルオブジェクトに従う．ローカルオブジェクトは，相互にメモリアドレスで参照しているが，その参照がセルの境界を越えることはない．この性質を使って，セル単位でローカルオブジェクトのゴミ集め（Gabage Collection: GC）が行われる．なお，GCの開始の指示はオブジェクトマネージャによりなされる．

5.3 クラスの配送

5.3.1 エグゼキュータが必要とするクラスの情報

エグゼキュータは，オブジェクトの生成やメソッドの実行の際に，種々のクラスの情報を必要とする．その情報としては，インスタンス生成に必要なコンフィギュレーションテーブル，オブジェクトの実装の詳細情報である実行時クラス情報，各クラスのメソッドの実装である実行可能コード（バイナリコード）などがある（図9）．

コンフィギュレーションテーブルは，パブリックパートのCIDとコンフィギュレーションのCIDの表である．インスタンス生成の際にコンフィギュレーションのCIDを得るのに用いられる．

実行時クラス情報は，コンフィギュレーションごとに生成され，構成するクラスのパブリックパートおよび実装パートのCIDとメソッドディスパッチテーブルを持つ．インスタンスの生成およびメソッドの実行の際に参照される．

メソッドディスパッチテーブルは，メソッドがどの実装パートの何番目の関数として実装されているかを示すテーブルであり，スロット番号に対応する実行可能コードのエントリを得るために用いられる．メソッドディスパッチテーブルが実行時クラス情報に含まれるのは，メソッド定義のオーバーライドの有無は実装の組が確定しないと決定できないからである．図3のプログラムで，クラスBでクラスAのメソッドbarをオーバーライドした場合の実行時クラス情報を図10に示す．メソッドbarはクラスAで宣言されるが，その実装はB 1.1.2の最初の関数である．

5.3.2 クラスのロード

クラスの情報は必要となったときに，クラス管理オブジェクトから直接あるいはファイルを介してオブジェクトマネージャやエグゼキュータにロードされる．

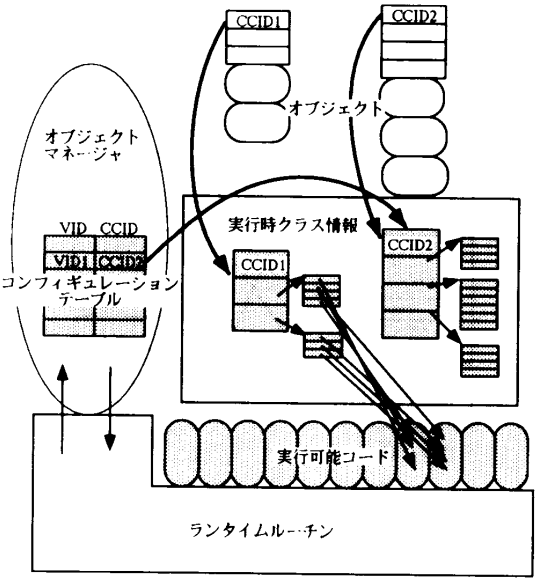

図9 実行機構の持つクラス情報
Fig. 9 Class information in an execution system.

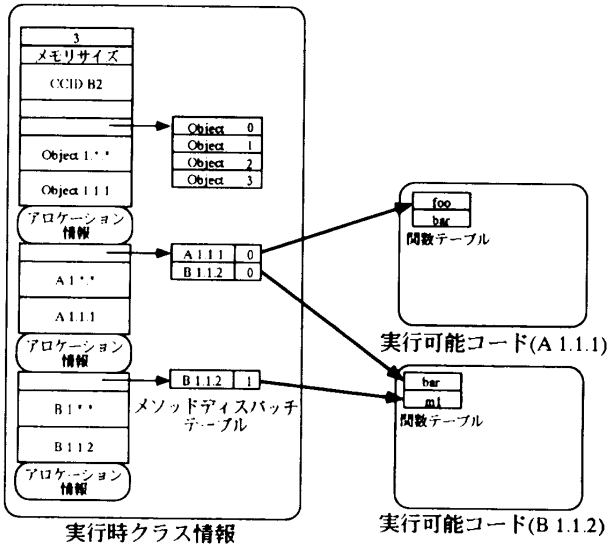

図10 実行時クラス情報
Fig. 10 Execution-time class information.

メソッドの実行中に必要な情報がないことをエグゼキュータが検出すると，オブジェクトマネージャにフォールトを伝え，その実行を中断する．エグゼキュータからのフォールトを受けたオブジェクトマネージャは，クラス管理オブジェクトに要求を伝える．クラス管理オブジェクトは情報が存在するファイルを答え，それがオブジェクトマネージャを経由してエグゼキュータに伝えられる．エグゼキュータは必要な情報をロードし，中断した実行を再開する．実行可能コードはロードされると，エグゼキュータのランタイムルーチンにダイナミックリンクされる．

クラス管理オブジェクトは，要求されたクラスの情報を持っていないとき，ブロードキャストを用いてサイト内の全クラス管理オブジェクトに問い合わせる．問い合わせを受けると当該情報を管理しているクラス管理オブジェクトは返事を返す．返事を返したクラス管理オブジェクトの1つにクラス転送の要求を出し，エグゼキュータ間のファイル転送機構を利用して必要なクラスの情報のファイルを入手する．

5.4 オブジェクトの実行

5.4.1 メソッドの実行

エグゼキュータはマルチスレッドによる並行実行環境を提供し，エグゼキュータのプログラムも，オブジェクトのメソッドもスレッドで実行される．

アプリケーションプログラムは，プログラム起動用ツールであるランチャにより，その起源となるオブジェクトを生成し，起動ルーチンを呼び出すスレッドを生成（フォーク）することにより実行が始まる．

5.4.2 グローバルオブジェクトのメソッド実行

グローバルオブジェクトのメソッドの実行は，新たに生成されたスレッドにより行い，呼び出した側のスレッドは戻値あるいは例外が返されるまでブロックする．2つのスレッドの同期と引数や戻値の受渡しのためにチャネルと呼ぶ構造体によりスレッドを関係付ける．同一エグゼキュータ内ではメモリ上に，異なるエグゼキュータ間ではネットワークによるオブジェクトの転送を介して，チャネルは構成される[8]．チャネルにより連結された一連のスレッドがプロセスを形成する．

他のエグゼキュータ上のグローバルオブジェクトのメソッド呼び出しは，まずニュクリアス間の通信によりエグゼキュータの通信アドレスを得て，エグゼキュータ間の通信によりメソッド呼び出しや引数，戻値の受渡しを行う．オブジェクトを転送する際には，オブジェクトのメモリ上の表現に近いフォーマットでエンコードする light-weight encoding の手法を用い，ワンパスでエンコードを行うようにした．これにより，大きなオブジェクト群を転送する場合にもエンコードとパケットの転送がパイプライン的に行える[8]．

5.4.3 オブジェクトの生成

オブジェクトを構成するクラスの実装の確定は，先に述べたようにオブジェクトの生成時に行われる．実行可能コード中にはコンパイラによりパブリックパートの CID が指定されている．インスタンスの生成時にそのバージョンで推奨されているコンフィギュレーションを得て，それにしたがってオブジェクトを生成する．パブリックのバージョンから推奨されているコンフィギュレーション CID（CCID）に変換する情報もクラス管理オブジェクトが管理しているが，効率化のためにそのキャッシュをオブジェクトマネージャが保持しており，通常それが使われる．コンフィギュレーション CID から実行時クラス情報を得ることができ，それにしたがってインスタンスが生成される．また，インスタンス生成後コンストラクタが実行される．

5.5 システムの起動

実行システムが動作するためにはオブジェクトマネージャなどの管理オブジェクトが必要である．これらのオブジェクトの生成は，実行システムが動作していないので，特別な手段で作り出さなければならない．そのため，クラス管理オブジェクトが提供するはずの情報をファイルの形で実現するブート用コンパイラ，永続化オブジェクトのイメージを生成するイメージビルダなどを開発した．

実行システムの起動には，実行システムに必須なオブジェクトマネージャ，クラス管理オブジェクト，ランチャなどの永続化オブジェクトイメージを生成し，それらの実行に最低限必要となるクラスのリストを準備する．エグゼキュータは，これらのファイルを順次読み込んでオブジェクトおよびクラスの情報をメモリ上に作り，オブジェクトマネージャを起動する．オブジェクトマネージャは初期化などの処理を行い，他の管理オブジェクトを起動する．クラス管理オブジェクトは，初めて起動される際にブート用コンパイラが生成したファイルを読み込むことにより，ブート用コンパイラによってコンパイルされたクラスについても管理できるようになる．

6. 性能測定

以上のシステムの基本的な性能測定を行った．測定条件は，**表1**のとおりである．スケジューリングタイムスライスは20 msec であり，GC を避けて測定した．ただし，マルチユーザ OS 上で，ネットワークを利用している状況で測定したので，厳密な測定ではない．

6.1 メソッド起動

引数と戻値を持たないメソッドの起動/復帰1回の消費時間を計測した（**表2**）．グローバルメソッド起

表1 測定条件
Table 1 Measurement environment.

計算機	Sun SparcStation 2
OS	SunOS 4.1.3
OZ++動作中の平均負荷	1.5 プロセス未満
ネットワーク	Ethernet
ネットワーク容量	10 Mbps
平均コリジョン	0.3% 未満
C/C++	gcc 2.5.8 (-O4 オプション)

表2 メソッド起動の性能
Table 2 Performance of method invocation without arguments and results.

ローカルメソッド	39 μsec
グローバルメソッド（同一エグゼキュータ内）	710 μsec
グローバルメソッド（同一計算機内）	8.4 msec
グローバルメソッド（計算機間）	6.6 msec

表3 引数と戻値を持つ場合のメソッド起動の性能
Table 3 Performance of method invocation with various size of arguments and results.

引数と戻値	同一エグゼキュータ内	計算機間
100 B 1 個	2.3 msec	10.3 msec
10 KB 1 個	18.9 msec	112 msec
144 B 70 個	93.5 msec	130 msec

表4 インスタンス生成の性能
Table 4 Performance of instantiation.

C++	0.17 μsec
ローカルオブジェクト	1.2 msec
グローバルオブジェクト（同一エグゼキュータ内）	14 msec
グローバルオブジェクト（同一計算機内）	29 msec
グローバルオブジェクト（計算機間）	56 msec

表5 クラス配送の性能
Table 5 Performance of class transfer.

ローカルメソッド起動（ファイルからロード）	0.11 sec
ローカルメソッド起動（クラス配送）	6.0 sec
ローカルインスタンス生成（ファイルからロード）	0.29 sec
ローカルインスタンス生成（クラス配送）	8.2 sec

動ではチャネルの生成/消滅，スレッドの生成/消滅，引数や戻値のコピーが生じる．またエグゼキュータが異なるとプロセス間通信あるいは計算機間通信が生じる．他のエグゼキュータへのメソッド起動では，OSのプロセス切替えがない分，他の計算機への起動の方が速い．

6.2 オブジェクトの転送

引数と同じオブジェクトを戻すメソッドの起動を，引数を変えて計測した（**表3**）．引数や返り値の受渡しのオーバヘッドはエグゼキュータ内の場合オブジェクトのトラバースとコピーのための領域確保などで約1.6 msec，計算機間の場合エンコード・デコードのための領域確保などで2 msecである．また，ネットワーク上のオブジェクトの転送速度は約200 KB/secである．また，大きなオブジェクト1つ（約10 KB）の場合と，ほぼ同じ総量で70個の参照関係にあるオブジェクトを転送した場合の差がポインタ操作のオーバヘッドと考えられるが，オブジェクトの転送に比べて影響は小さい．同一エグゼキュータ内では，セルごとに管理された2つのヒープ間で頻繁にコピーが起こり，メモリアクセスが分散するため，オーバヘッドが大きくなっていると考えられる．

6.3 インスタンス生成

十分小さなインスタンスを生成するための時間を測定した（**表4**）．C++が単にメモリ領域を確保するだけなのに対し，OZ++では動的にバージョンを確定するために，クラス管理オブジェクトにコンフィギュレーションを問い合わせ，実装パートを確定し，オブジェクトの構造を生成し，コンストラクタを実行する．グローバルオブジェクトの生成では，さらに新たなヒープの確保やOIDの付与等の処理が含まれる．

6.4 クラス配送

6.1～6.3節は，クラスの情報がすべてエグゼキュータ上に存在した場合の測定であるが，次にエグゼキュータ上に存在しない場合の測定を行った（**表5**）．エグゼキュータ上に存在しないがクラス管理オブジェクトがファイルとしてクラスの情報を保持している場合には，クラス管理オブジェクトからファイル名を得て，エグゼキュータにロードする．クラス管理オブジェクトがクラスの情報を持っていない場合には，他のクラス管理オブジェクトに問合せをし，ファイルの転送を受けなければならない．

メソッド起動時には実行時クラス情報および実行可能コードがロードされ，インスタンス生成時にはさらにコンフィギュレーションもロードされる．転送されるファイル数，全体の大きさとエグゼキュータにロードされるファイルの大きさ（括弧内）はそれぞれ実行時クラス情報が3ファイル1.5 KB（228B），実行可能コードが9ファイル38 KB（32 KB），コンフィギュレーションが2ファイル1 KB（303B）である．クラスの配送にはクラス1つあたり3秒弱かかっているが，そのうちファイル転送の前後でtarファイルの作成および展開をしている時間が全体の約8割を占めている．ファイル転送の性能は320 KB/secで，オーバヘッドは約0.2秒である．配送される他のファイルは，コンパイル時に必要な情報であり，現在はそれらも含めてそのCIDで管理されるすべてのファイルを配送する実装となっている．

7. 関連研究

7.1 分散オブジェクト指向言語

Eden[9]やArgus[10]はモニタ機能を有する抽象データ型言語を用い，ともに，オブジェクトは大粒度である．Edenではマーシャリング/アンマーシャリングのコードが自動生成されるが，Argusではその手続きをデータ型定義として記述しなければならず繁雑である．

Emerald[4),5)]も抽象データ型言語であり，大粒度から小粒度のオブジェクトまでが記述でき，同期・相互排除はモニタによって行う．オブジェクト間でオブジェクトが移動することが基本となっており，オブジェクトの移動阻止/解除，連携移動等の工夫がされている．必要なコードは自動的に取り寄せられる．

Guide[11]は単一継承言語であり，インタフェースと実装を分けて記述し，実装の記述において，同期・相互排除をガードとして記述する点に特徴がある．オブジェクト間で交換するものに，オブジェクトへの参照は含まれうるがオブジェクト自身は含まれない．CORBAのIDLと適合させることが強く意識されている．

Smalltalk-80の分散システム用拡張を試みたものに，Concurrent Smalltalk[12]，Distributed Smalltalk[13]等の先駆的研究がある．遠隔オブジェクトへのアクセス，分散オブジェクトの識別，等の工夫がされているが，単一ユーザ環境の拡張には無理がある[14]．

Java[15]はオブジェクト指向言語であり，HotJava[15]は中間コードを転送する機能を装備したブラウザである．セキュリティ実装が予定されている．WWWブラウザを補強する程度の小さなプログラムが想定されており，大規模な分散アプリケーション向きではない．

7.2 バージョン管理

クラスのバージョン管理については，様々な分散CASEツールがソフトウェア開発用に実現されてきた[16]が，これは最終的に1つのバージョンのものを開発するためである．また，GemStone[17]などのオブジェクト指向データベースで，スキーマエボリューションが実現されてきたが，これはスキーマが変わってもデータベースのオブジェクトが使えるようにするためである．一方，OZ++は，これらと違い分散システムに不可欠な新旧複数のバージョンが混在動作を可能にするものである．また，セマンティックスやインタフェースが変更になったときにバージョンも変わるという進化の枠組も提供している．

7.3 OZ+

OZ++のベースは，筆者らが開発したOZ+[18]である．オブジェクトモデル，クラス/オブジェクトの共有・配送等に共通点が多い．OZ+からOZ++に移行するにあたって，次の点に大きな改良を加えた．

- クラスのモジュラリティを高め，継承によるクラスの再利用が自然に行えるように，単一継承から多重継承可能なものにした．
- 実行時の例外処理を減らし，コンパイル時に多くのエラー検出が可能なようにタイプ付言語にした．
- OZ+でもクラスにバージョンが付与できたが，バージョン更新の方針がユーザまかせであった．実行時のオーバヘッドも大きかった．OZ++では，3章で述べたように，クラスのインタフェースに着目したバージョン管理を導入し，バージョン更新の意味を明確にした．

8. おわりに

本論文では，ネットワーク環境でクラスを共有し必要に応じて配送することを特徴とする分散オブジェクト指向システムOZ++の設計思想，それを具体化する実行システムの実装，および性能評価について述べた．OZ++では，(1) ネットワーク上に存在するクラスを相互に共有・再利用して新しいクラス群を作ることができる，(2) 新たに作られたクラスのオブジェクトを他の計算機に送信しても受信側でその新しいクラスをネットワーク上から自動的にロードして実行できる，(3) バージョン管理により，新旧のバージョンを混在して使うことができる，(4) インタフェースに着目したクラスのバージョン管理を行うために，新しいバージョンのクラスを部品として積極的に使うことができる，ことなどが実現された．

クラス配送の性能がやや低いが，これはクラスのファイルへの格納形式を変更することにより大幅な改善が見込まれる．

現在開発中のセキュアなクラス配送機構により，ワールドワイドにクラスを共有・再利用できるオブジェクト指向分散環境が構築できるであろう．

なお，本論文で述べたOZ++システムは，紙面の都合で説明を割愛した開発環境も含めて，次のアドレスで公開されている．

http://www.etl.go.jp:8080/etl/bunsan/OZ/OZ.html

謝辞 本研究は，情報処理振興事業協会（IPA）が実施している「開放型基盤ソフトウェア研究評価事業」の一環として行われたものである．OZ++を共同開発してきた開放型基盤ソフトウェアつくば研究室の研究員諸氏，温かくご支援いただく電子技術総合研究所情報アーキテクチャ部太田公廣部長，IPA棟上昭男理事，ならびにOZ++関係各社の方々に感謝したい．

参考文献

1) 西岡ほか：オブジェクト指向分散環境OZ++のオブジェクト管理系の設計，情報処理学会マルチメディア通信と分散処理研究会，66-4 (1994).
2) 音川ほか：オブジェクト指向分散環境 OZ++のプログラミングパラダイム，情報処理学会 SWoPP'95 (1995).
3) Chin, R.S. and Chanson, S.T.: Distributed Object-Based Programming System, *ACM Computing Survey*, Vol.23, No.1, pp.91–124 (1991).
4) Black, A., et al.: Distribution and Abstract Type in Emerald, *IEEE Trans. Softw. Eng.*, Vol.SE-13, No.1, pp.65–76 (1990).
5) Jul, E. et al.: Fine-Grainded Mobility in the Emerald System, *ACM Trans. on CS*, Vol.6, No.1, pp.109–133 (a988).
6) Hoare, C.A.R.: Monitors: An Operating System Structuring Concept, *CACM*, Vol.17, No.10, pp.549–557 (1974).
7) Matsuoka, S., et al.: Highly Efficient and Encapsulated Re-use of Synchoronization Code in Concurrent Object-Oriented Languages, *Proc. OOPSLA '93*, pp.109–126 (1993).
8) Hamazaki, Y., et al.: The Object Communication Mechanisms of OZ++: An Object-Oriented Distributed Programming Environment, *Proc. ICOIN-9*, pp.425–430 (1994).
9) Almes, G.T., et al.: The Eden System: A Technical Review, *IEEE Trans. Softw. Eng.*, Vol.SE-11, No.1, pp.43–59 (1985).
10) Liskov, B.: Distributed Programming in Argus, *CACM*, Vol.31, No.3, pp.300–312 (1988).
11) Balter, R., et al.: The Guide Language, *Comput. J.*, Vol.37, No.6, pp.519–530 (1994).
12) Yokote, Y.: *The Design and Implementation of Concurrent Smalltalk*, World Science Co. Pte. Ltd. (1990).
13) Bennett, J.K.: The Design and Implementation of Distributed Smalltalk, *Proc. OOPSLA '87*, pp.318–330 (1987).
14) 所ほか：オブジェクト指向コンピューティング，岩波書店 (1993).
15) Gosling, J. and McGilton, H.: *The Java Language Environment - A White Paper*, Sun Microsystems Computer (1995).
16) 青山：分散開発環境：新しい開発環境像を求めて，情報処理，Vol.33, No.1, pp.2–13 (1992).
17) Penny, D.J. and Stein, J.: Class Modification in the GemStone Object-Oriented DBMS, *Proc. OOPSLA '87*, pp.111–117 (1987).
18) 塚本，濱崎：オブジェクト指向開放型分散システム：OZ+，情報処理，Vol.36, No.8, pp.715–720 (1995).

(平成7年10月3日受付)
(平成8年 2 月7日採録)

塚本 享治（正会員）

昭和24年生．昭和47年東大計数卒業．同年電子技術総合研究所入所，現在同分散システム研究室長．知能ロボット用計算機システムの研究の後，オブジェクト指向型分散システム，並列処理システム等の研究開発に従事．平成元年日本ロボット学会論文賞，平成6年科学技術長官賞研究功績賞．現本学会理事．

濱崎 陽一（正会員）

1957年生．1979年岡山大学工学部電気工学科卒業．1981年同大学大学院修士課程修了．同年電子技術総合研究所入所．マルチプロセッサシステムの開発，オブジェクト指向分散システムの開発などに従事．

音川 英之（正会員）

1991年大阪大学基礎工学部情報工学科卒業．同年シャープ（株）入社．1992年よりIPA開放型基盤ソフトウェアつくば研究室研究員として，分散オブジェクト指向システムの研究開発に従事．オブジェクト指向プログラミング，ユーザインタフェースに興味を持つ．

西岡 利博（正会員）

1965年生．1988年電気通信大学電気通信学部計算機科学科卒業．1990年同大学院電気通信学研究科情報工学専攻博士前期課程修了．同年，(株) 三菱総合研究所に入社．以後，オブジェクト指向知識処理，オブジェクト指向ソフトウェア工学，オブジェクト指向並列計算の研究に従事．IPA開放型基盤ソフトウェアつくば研究室研究員．e-mail: nishioka@mri.co.jp