プログラミング－言語・基礎・実践－ 18－22
（１９９４．７．２２）

OZ++ コンパイラによるクラスの版管理

新部 裕[1*] 音川 英之[2*] 西岡 利博[1*] 石川 正基[1*]
中川 祐[3*] 濱崎 陽一[4] 塚本 享治[4]

1: 三菱総合研究所 2: シャープ
3: 富士ゼロックス情報システム 4: 電子技術総合研究所
*: 情報処理振興事業協会「開放型基盤ソフトウェア研究開発評価事業」研究員

オブジェクト指向分散環境 OZ++ は分散アプリケーションの開発と稼働を容易にする開発基盤を目指している.
OZ++ は広域ネットワーク環境で, ユニークなクラス ID に基づくクラスの配送システムと, クラスのインターフェースに基づくバージョン管理を提供することで, 高機能なソフトウェア資源の共有の枠組を与える.
本論文は, OZ++ における, コンパイル時に行なうクラスの継承関係, メソッドのシグネチャ, および変数の型のチェックに基づくクラスの版管理方式とその実現方法について述べる.

The Version Management Mechanism of OZ++

Yutaka Niibe[1*] Hideyuki Otokawa[2*] Toshihiro Nishioka[1*] Masaki Ishikawa[1*]
Yu Nakagawa[3*] Youichi Hamazaki[1] Michiharu Tukamoto[1]

1:Mitsubishi Research Institute, Inc. 2:Sharp Corporation
3:Fuji Xerox Information Systems, Co., Ltd. 4:Electrotechnical Laboratory

*: Researcher of Research, Development and Evaluation of Open Fundamental Software Technology in Information-Technology Promotion Agency, Japan

OZ++ is a programing environment that makes easy to develop and to execute distributed application programs.
It provides a version management mechanism of classes and a demand-driven class transfer mechanism over wide-area networks. These mechanisms make possible to share and update classes flexibly and efficently. Versions of classes are managed by the distibuted class management system according to class interfaces.
This paper presents a design policy and an implementation of version management mechanism of OZ++ .

1 はじめに

OZ++ はオブジェクトの交換と共有に基づく分散処理環境である.

OZ++ が提供する分散プログラミング環境とは, ネットワーク上に広く分散したコンピュータにおいて,

- オブジェクトを相互に利用し合うこと
- クラスを相互に利用し合うこと

を可能とする枠組である [2].

OZ++ ではオブジェクト指向の枠組を分散ネットワークプログラミングに採り入れている. これによって, ネットワーク上のサービスおよびソフトウェア資源を安全に利用し合うこと, また, 拡張が容易で自由度の高い分散システムを構築することを目指している. OZ++ では目的を異にする 複数の利用者が協調できる環境を想定した上で, ソフトウェアのさまざまなバージョンが並立することを前提としている.

本稿では, OZ++ におけるクラスと, そのバージョン管理について述べ, その利点について考察する.

2 節で分散システムにおけるソフトウェア資源の共有とソフトウェアの更新について考察する. 3 節でこの問題に対しての OZ++ における設計方針を述べる. 4 節で OZ++ のクラス管理システムについて述べ, 5 節で OZ++ におけるインターフェースの取り扱いについて述べる. 6 節で、実際に動作するバージョンの決定を遅延させることによる、ソフトウェアの柔軟な実行環境の枠組ついて述べる。 7 節で検討課題について述べる.

2 分散システムにおけるソフトウェア資源の共有とソフトウェアの更新

本節では, 分散システムにおいて, ソフトウェア資源の共有にはどのような特性があり, その有効利用のためにはどのような機構がシステムに要求されるかについて考察する.

2.1 分散透過なソフトウェア資源

プログラム開発とその利用には, 以下の資源が使われる.

- プログラム開発時に必要となるライブラリとそのインターフェース
- 実行時に必要となるプログラム
- プログラム利用のための説明書

ここでは, これらをソフトウェア資源と総称する.

ソフトウェア資源の共有を考えた場合, もっとも重要なことはソフトウェア資源の提供側と利用側とで共通の環境を持つことである. そして広域のネットワークでそれを実現することは難しい. あるソフトウェア資源が提供された時に, その都度, 利用側があらかじめ準備をするのではなしに, 必要に応じて自動的に利用できることが望ましい. このためには,

- 分散透過にソフトウェア資源が利用できる

ことが必要である. ただし, その実現には考慮が必要で, 効率の良い方法を考える必要がある. すべてのソフトウェア資源を転送し合う, あるいはすべてのソフトウェア資源をコピーするという実現は, ネットワークのバンド幅, コンピュータの計算時間, および二次記憶を大量に消費するので, これは避けなければならない.

2.2 モジュラリティ

広域のネットワークでは, ソフトウェア資源の提供側と利用側に密接な関係があり, 共通の目的を持っていると一律に仮定するのは望ましくない.

このとき, ソフトウェア資源を共有して有効に利用し合うには, 以下の特性が望まれる.

- 提供側と利用側の依存関係を明確に表現できる
- 安全に利用できる仕組みが提供される[1]

つまり, 明確なインターフェースを持つようにし, ソフトウェアのモジュラリティを向上させる必要がある.

2.3 ソフトウェアの更新

広域のネットワークでは, サービスに対するさまざまなニーズがあり, それらを反映してさまざまなソフトウェアが存在する. また, そのソフトウェアはニーズの変化に従って, 変更・拡張され, 新しいソフトウェアとなる.

ここでは, 以下の特性が望まれる.

- 既存のソフトウェアを利用し, これに新機能を追加するという形での拡張を行う仕組み
- 既存のソフトウェアを利用し, ある機能の実現を利用者固有の目的にかなったものにするためにカスタマイズする仕組み
- 機能拡張, バグフィックスなど, ソフトウェアが更新された場合, その機能を利用できる仕組み

[1] ここで言う "安全" は, secure ということではなく, プログラミングの際に表現された依存関係の枠からはみ出せないということである.

2.4 バージョン

2.3 節で述べたソフトウェアが更新されていく環境を考えた場合, そのソフトウェアに関わるすべてのデータおよびソフトウェアを更新に対応させることは, 分散システムにおいては非現実的である. 分散システムでは, それぞれの管理の範囲, 管理のポリシーがあり, ある判断がすべての場合に当てはまるわけではない.

そこで, 同じようなさまざまな新旧のデータ, 新旧のソフトウェアが混在することを念頭におくことが重要となる. このため,

- 新旧のソフトウェアを管理し, 矛盾のない運用を可能とするバージョンの管理の仕組み

が必要となる.

3 OZ++ の設計方針

前節に述べた, 分散透過なソフトウェア資源, モジュラリティ, ソフトウェアの更新とバージョンに対する要求を満たすために, OZ++ では

1. クラスをソフトウェア資源の単位とし, 多重継承をサポートしたモジュラリティの高いものとする
2. クラス管理システムを導入し, ネットワーク上のサービスとして, クラスに関する情報を提供する
3. クラス管理システムによって, バージョンを管理する

という方針を取った.

これは, 以下の判断による.

1. オブジェクト指向の枠組では, クラスをソフトウェア資源の単位とすることは自然でわかりやすい
2. オブジェクトがネットワーク上のサービスの単位であるという OZ++ の概念をそのまま用いて, 分散透過なクラスを実現するクラス管理システムを導入できる
3. バージョンの管理により, ソフトウェア資源の有効利用を推し進められる

3.1 ソフトウェア資源の単位としてのクラス

OZ++ ではソフトウェア開発のモジュールとしてクラスを導入している. そして, 同時に, これをソフトウェア資源の単位とし, 共有するようにしている. このために, OZ++ におけるクラスは全世界でユニークな ID を付与される. 言語処理系, および実行系はこの ID によりクラスを参照する.

オブジェクトには, このクラスの ID が埋め込まれ, 同一のクラスに属するオブジェクトはクラスを共有する.

OZ++ のクラスは多重継承をサポートし, 既存のクラスを拡張して利用すること, あるいは部品として利用することを可能としている. オブジェクト指向の情報隠蔽の性質により, クラスの利用者と提供者の相互の依存部分を明確にすることが可能である. このようにして, ライブラリとしてのソフトウェア資源の有効利用をはかっている.

また, インターフェースを明確にし, 型の静的なチェックによりクラスの安全な利用をある程度保証している. これによって, ソフトウェア資源の再利用を推進する.

3.2 分散透過なクラスとクラス管理システム

クラスに全世界でユニークな ID を与え, クラスを分散透過とする, これが OZ++ の最大の特徴である.

この実現のために, クラス管理システムを導入し, ネットワーク上のサービスとして, クラスの情報を提供する. クラスの情報を利用するためには, ユニークな ID を示し, クラス管理システムに情報を要求する.

クラス管理システムは OZ++ のオブジェクトとして実現される. オブジェクトで実現することにより, クラス管理システムは, OZ++ の特徴を利用して, それ自身が拡張, カスタマイズが可能であるという特徴を持つ.

3.3 クラス管理システムと言語処理系, 実行系

OZ++ では, 図1に示すように, クラス管理系, 言語処理系, 実行系の 3 つのサブシステムがある. OZ++ の利用者はこれらを用いて, クラスを開発 / 利用する.

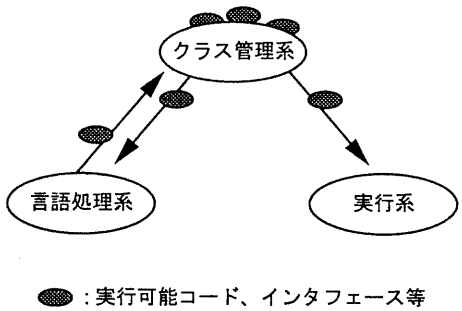

図 1: クラス管理システムと言語処理系, 実行系

クラス管理システムに対する言語処理系によるクラスの情報の登録と, 実行系によるクラスの情報の取得は以下の

ように分かれている.

1. 言語処理系はソースコードをコンパイルし, その結果であるインターフェース, メモリ配置情報, 実行可能コードなどをクラスに登録する
2. クラスはその情報をインターフェースに基づき, バージョンとして管理する
3. 実行系は必要となった時に, オブジェクトに埋め込まれたユニークなクラスの ID をもとにクラス管理系から情報を取得する

3.4 クラス情報取得の遅延

実行系によるクラス管理システムのアクセスと情報の取得について考えた場合, 以下のことがいえる.

- オブジェクトのメモリ配置情報の取得はオブジェクトの生成時まで遅延することができる
- 実行可能コードの取得は, メソッドの起動時まで遅延することができる

そして, OZ++ では, この遅延を積極的に行ない, オブジェクトのメソッドの実行, およびオブジェクトの生成に対して, あらかじめ実行可能コード, オブジェクトのメモリ配置情報を準備することなく, 必要となった時にクラス管理システムのサービスを利用して, 取得するという方針を取っている.

これにより, 不必要なコピー, 不必要な転送を行なうことなく, 分散透過なソフトウェア資源の共有を効率良く実現することを狙っている.

また, これは実装の特定がオブジェクトの生成時まで遅延されるということである. これにより, 利用者がその場に応じて, 実装の特定を変更できることを可能にしており, この機能を用いた柔軟な運用が期待できる.

3.5 安全な共用の保証

クラス管理システムによる分散透過なクラスの実現により, クラスを利用する枠組が提供される. さらに, この共有を有効に進めるためには, クラスを安全に共用する仕組みが必要である. 運用の矛盾をきたさないクラス管理が必要である.

OZ++ では, この問題に対し, コンパイル時にクラスの継承関係, メソッドのシグネチャ, および変数の型の静的なチェックを行ない, そのインターフェースに基づいてクラスの各バージョンを管理するというアプローチを取っている.

インターフェースが明確に定義され, それに基づいた管理が行なわれているため, 異なるインターフェースのクラスが, 誤って利用されることはない. よって, 新旧のクラスを用いたオブジェクトの混在を安全に実現することができる.

4 OZ++ におけるインターフェースの取り扱いとバージョン

OZ++ ではクラスに明確なインターフェースを定義し, コンパイル時に十分な型のチェックを行なうことにより, クラスの一貫性を保持し, オブジェクトの動作を保証する.

本節では, OZ++ におけるインターフェースの取り扱いとバージョンについて述べる.

4.1 クラスの利用

クラスの利用には, 継承木の中での利用と, 外からの利用とがある. 継承木の中での利用とは, スーパークラスとして利用することであり, これを継承としての利用と呼ぶ. 継承木の外からの利用とは, インスタンス変数, 自動変数などとして利用することであり, これを外部参照としての利用と呼ぶ.

4.2 OZ++ 言語におけるアクセス制御

アクセス制御とは, クラスメンバを外部に公開するか否かの指定を行なうことであり, 情報隠蔽を支援する機能である.

クラスメンバのアクセス制御には次の 3 つがある.

- public
  クラスメンバのうちメソッドに対してのみ指定できる. 外部参照としての利用と, 継承としての利用に対してアクセスを許可する.
- protected
  クラスメンバのすべてに指定が可能である. 継承としての利用に対してアクセスを許可する.
- private
  クラスメンバのすべてに指定が可能である. 外部参照としての利用, 継承としての利用にいずれに対してもアクセスを許可しない.

4.3 クラスのインターフェースとそのバージョン

OZ++ 言語では, 上記のアクセス制御のうち, public および protected の部分をそのままインターフェースとして

扱っている [3]. private の部分は実装の詳細であるとし, インターフェースとはしない.

そして, そのそれぞれについてバージョンを与え, 管理を行なうシステムを採用している.

クラスのバージョンはこのパブリック・インターフェース, プロテクティッド・インターフェース, および実装の 3 つを表現する 3 組の数字により表現される. バージョンを表現するそれぞれの部分は, パブリックパート, プロテクテッドパート, インプリメンテーションパートと呼ばれる.

この言語の情報隠蔽の仕組に対応した 3 つの切口でバージョンを扱うことにより, あるクラスを用いているプログラムの再コンパイルの必要性を限定することができる [2].

クラスのバージョン構造は図2のようになる. また, これらすべてのバージョンには, 全世界でユニークな ID が付与される. OZ++ システムは, この ID から, それが示すバージョンに関する情報をアクセスする機構を提供している.

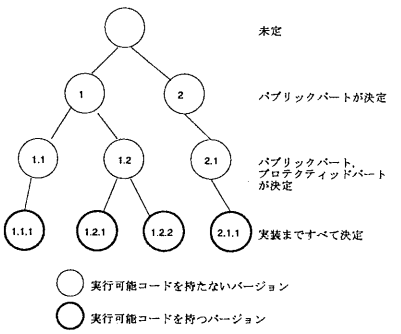

図 2: クラスのバージョン構造

このインターフェースにそって, クラスの開発は, 以下の三段階で行なわれる.

1. あるクラスのパブリック・インターフェースのバージョンを開発する.
2. パブリック・インターフェースのバージョンを指定して, そのプロテクティッド・インターフェースのバージョンを開発する.
3. プロテクティッド・インターフェースのバージョンを指定して, その実装のバージョンを開発する.

あるクラスが他のクラスを利用する場合, クラスの利用の仕方に従って, 利用するクラスのバージョンを指定する.

1. 外部参照としての利用の場合
   利用するクラスのパブリック・インターフェースのバージョンを指定する.
2. 継承としての利用の場合
   利用するクラスのプロテクティッド・インターフェースのバージョンを指定する.

   注意が必要なのは, 上記の開発の第一段階において, スーパクラスのプロテクティッドのバージョンまでが特定される必要があることである.

4.4 OZ++ コンパイラ

OZ++ プログラムを記述する場合, クラス名はプログラム開発者個人にとってのみ意味のある文字列として記述される. 開発者はコンパイラに, 次の指定を行なう.

1. クラス開発の段階
2. ソースプログラム
3. クラス名からインターフェースを特定するクラスのバージョン ID への対応表

コンパイラは3の対応表を利用して, ソースプログラム中のクラス名の出現を, インターフェースの適合性のチェックが可能となるクラスのバージョン ID に置き換える.

OZ++ コンパイラは上記の開発の段階に従って, 指定されたバージョンを用いて, 変数名による参照の適合性やメソッド呼び出しの適合性をチェックする.

5 クラス管理システムの管理する情報

クラス管理システムは, クラスをインターフェースに基づいてバージョンとして管理し, 以下の情報を扱う.

- 継承木
  言語処理系が生成し, 登録する. 言語処理系によりソースコードのコンパイル時に利用される. 実行系のコンフィギュレーション生成時, ディスパッチテーブル生成時に利用される.
- パブリックのメソッドシグネチャ
  言語処理系が生成し, 登録する. コンパイル時に利用される.
- プロテクティッドのメソッドシグネチャ
  言語処理系が生成し, 登録する. コンパイル時に利用される.

- オブジェクトのメモリ配置に関する情報
  実行系がオブジェクトを生成する際, オブジェクトをエンコード / デコードする際に利用する

- 実行可能コード
  言語処理系が生成し, 登録する. メソッドの実行時に利用される.

- 実行時クラス情報
  実行系がオブジェクトのメソッドを実行する際に必要となる情報である.

6 実行系とコンフィギュレーション管理

OZ++ の実行系は, 言語処理系によって生成された実行可能コードをオンデマンドでロードし実行する.

本節では, インスタンスの生成時にクラスの実装部分のバージョンを動的に決定し管理する機構について述べる.

6.1 コンフィギュレーション

前節で述べたように, OZ++ 言語で記述されたクラスはコンパイル時には, 継承としての利用のクラスの場合, プロテクティッドパートのバージョンまでが特定されているだけである. 外部参照としての利用のクラスの場合は, パブリックパートが特定されている. いずれの場合も, インプリメンテーションパートのバージョンはコンパイル時には特定されておらず, バージョンの特定はインスタンスの生成時に行なわれる.

図3のようなクラス階層のクラスを考えよう. このクラス D を利用する場合, インプリメンテーションパートのバージョンの特定は, D だけでなく A, B, C のすべてに対して行なう必要がある. A, B, C, D のすべてに対して特定して, 生成されたオブジェクトはそれらのバージョンの組をひとまとまりとして扱うことが必要である.

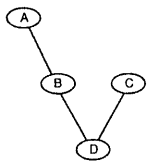

図 3: クラス階層の例

このように, あるクラスのすべてのスーパークラスとそれ自身についてインプリメンテーションパートまで特定されたバージョンの組をコンフィギュレーションと呼ぶ. またコンフィギュレーションに対して ID を付与し, これをコンフィギュアードクラス ID と呼ぶ.

コンフィギュレーションはクラスのバージョンに関する情報の一つであるため, クラスオブジェクトによって生成, 管理される. コンフィギュレーションは次の場合に生成される.

- 利用者がインスタンスを生成する前に, インプリメンテーション パートのバージョンとしてどれを利用するかを特定する場合.

- 利用者によるコンフィギュレーションの生成が行なわれずにインスタンスの生成が開始され, その場でコンフィギュレーションが必要となった場合

このようにコンパイル時にはクラスのインタフェース部分のバージョンだけを決定し, 実装部分についてはインスタンスの生成時にコンフィギュレーションによって動的に決定する.

この動的決定の仕組みを用いて, 利用者は生成されるインスタンスに対し, コンフィギュレーションを指定して, その挙動を選択することがある程度可能となる.

6.2 インスタンスの生成

あるクラスのインスタンスを生成する場合には, 実行系はデータ領域を決定したりコンストラクタを実行するために, インスタンスの構造を表現する情報やメソッドの実行のための情報が必要となる. ところがこれらの情報はクラスの実装部分のバージョンによって決定される部分である. したがってコンパイル時に付与されているインプリメンテーションパートが未確定なバージョンのバージョン ID (以後, コンパイル時バージョン ID と呼ぶ) に対するコンフィギュレーションを生成し, 実装部分のバージョンを特定しなければならない.

このため, 実行系と管理システムの間で以下のやりとりが行なわれる (図 4).

1. 実行系はクラス管理システムに対してコンパイル時バージョン ID に対応するコンフィギュアードクラス ID を問い合わせる.

2. クラス管理システムは, 受けとったコンパイル時バージョン ID に対するコンフィギュレーションが存在すればそのコンフィギュアードクラス ID を返す.

3. コンパイル時バージョン ID に対するコンフィギュレーションが存在しない場合, 新たなコンフィギュレーションを生成し, そのコンフィギュアードクラス ID を返す.

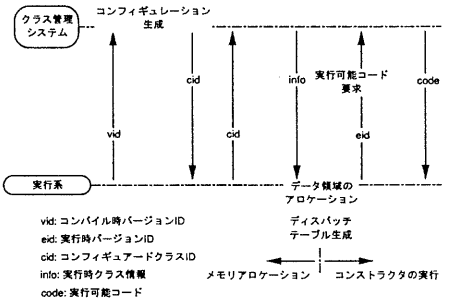

図 4: インスタンスの生成

次に実行系は次のようにデータ領域のアロケートとコンストラクタの実行を行なう.

1. コンフィギュアードクラス ID を用いてクラス管理システムから実行時クラス情報を取得する.

2. データ領域内のオブジェクトの管理情報にはコンフィギュアードクラス ID を格納し, そのオブジェクトが他の実行系にコピーされた場合でも, そこで実行時クラス情報を取得し次節で述べるようなメソッドの実行の処理が正しく処理されるようにする.

3. データ領域のアロケートの終了後, コンストラクタの実行を行ない, インスタンスの生成が完了する.

6.3 メソッドの実行

実行系は実行時クラス情報を元に図 5のようなクラス管理情報を作成する. この情報の中にはそのクラスを構成する各クラスの, インプリメンテーションパートが確定したバージョンのバージョン ID (以後, 実行時バージョン ID と呼ぶ) が格納されている. この実行時バージョン ID は実行可能コードをダイナミックロードする際に利用される.

実行系によるメソッドの実行は, 次のように行なわれる.

1. 実行系上にメソッドのディスパッチテーブルが存在しなければ, インスタンス中のコンフィギュアードクラス ID を元にクラス管理システムから実行時クラス情報を取得し, それを元にディスパッチテーブルを作成し, それによってロードすべき実行可能コードを決定する.

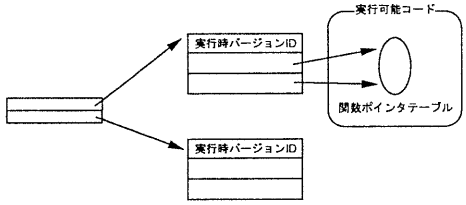

図 5: クラス管理情報

2. その実行可能コードが実行系上に存在しなければ, クラス管理システムにそれに対応するクラスの実行時バージョン ID を渡し, 実行可能コードを取得する. そして, 実行系のダイナミックローダを起動する.

3. 実行可能コードのロード後, コンパイル時にメソッドに割り当てられたセレクタを用いて, 実行すべき関数を選択し, その関数を実行する.

6.4 コンフィギュレーションの指定

図 6-(a) のようなクラス D を外部参照として利用するクラスではコンパイル時にはクラスのパブリックパートが決定されている. クラス D のコンパイル時には, スーパクラス, A, B, C のプロテクティッドパートまでのバージョンが決定されている.

クラス D のコンフィギュレーションの指定とは, A, B, C, D それぞれのインプリメンテーションパートまで特定したバージョンを決定することである.

各クラスには図 6-(b) のようにインプリメンテーションパートまで特定したバージョンが複数存在し, それらの中から一つずつを選択して図 6-(c) のようにバージョンの組合せを決定する.

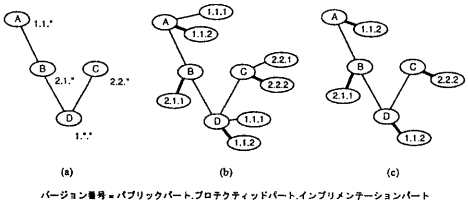

図 6: コンフィギュレーションの指定

7 課題

課題として, 以下のことがあげられる.

- スキーマエボルューション
  バージョンの管理が必要となるのはスキーマエボルューションの仕組みをシステムが提供しないからであり, スキーマエボルューションの仕組みが提供されれば, 異なる複数のバージョンのクラスを用いるオブジェクトは存在しなくなるのではないかと考えることもできる.

  しかしながら, 分散環境においては, それぞれ目的の異なる利用者が存在するので, クラスの更新により, 一斉に現存するオブジェクトのスキーマエボルューションを行なうことは現実的ではない. よって, 複数のバージョンのクラスが存在することはスキーマエボルューションによっては回避できない.

  一斉にすべての更新を行なうというのではなく, 個別の要求としてのスキーマエボルューションは必要であると考える. その実現を考えた場合, システムとして何を提供し, ユーザはどのように利用するかの切りわけが重要である.

  スキーマエボルューションのために, システムが一般的に提供できる機能よりも, クラスの設計者がそのクラスのために用意する機能のほうが実行の効率が良い. [1] このため, OZ++ ではスキーマエボルューションが必要であれば, ユーザが記述するという方向で, 検討をしている.

- 柔軟性
  言語における private をインターフェースとしていないが, これをインターフェースとし, private のインターフェースを同じくする実装を選択的に利用可能とすることによって, さらに柔軟な運用が期待できるであろう.

  この機能は, 特に開発途中でデバッグを行なう時に有効だと考えられるので, 導入を検討している.

- バージョンコントロール
  OZ++ のバージョン管理の仕組みは, クラスのインターフェースの識別とそれによる安全な運用を提供するものであり, 分散した環境における並行開発を支援するバージョンコントロールの仕組み (変更履歴の管理, バージョンの派生とその木の管理) は提供していない. これに関してはサポートすることが考えられる.

  ただし, これはシステムに組み込みの機能ではなく, OZ++ のアプリケーションとして実現可能である. 並行開発環境には必要な機能と思われるので, 開発支援システムとしての導入を検討している.

8 おわりに

オブジェクト指向分散環境 OZ++ について, クラスを分散透過とし, ソフトウェア共有の単位とすること, それを実現する機構であるクラス管理システム, およびインターフェースとコンフィギュレーションの実際について述べた.

現在, クラス管理システムを実装中であり, 実装後, 評価, 改良をしていく予定である.

謝辞

本研究を通じて, 熱心な討論をいただいている, 藤野晃延氏 (富士ゼロックス情報システム), 吉屋英二氏 (富士ゼロックス情報システム), 千葉滋氏 (東京大学理学部), および OZ++ プロジェクトの各メンバーに感謝する。

本研究は、情報処理振興事業協会「開放型基盤ソフトウェア研究開発評価事業」の一環として行われたものである。

参考文献

[1] D. Jason Peneny and Jacob Stein, "Class Modification in the GemStone Object-Oriented DBMS", In *OOPSLA '87*, pages 111-117, 1987.

[2] 塚本他, 「オブジェクト指向分散環境 OZ++ の基本設計」, 情報処理学会研究報告 93-OS-61-3(SWoPP 鶴の浦) Aug. 1993.

[3] 西岡他, 「オブジェクト指向分散環境 OZ++ の言語の基本設計」, 情報処理学会第 46 回全国大会, Mar. 1993.

[4] 吉屋他, 「オブジェクト指向分散環境 OZ++ のクラス管理方式」, 情報処理学会第 47 回全国大会, Oct. 1993.

[5] Graham D. Parrington, "Reliable Distributed Programming in C++: The Arjuna Approach", In *Proceedings of the 1990 Usenix C++ Conference*, Apr. 1990.