解　説

手続き型言語へのオブジェクト指向の導入†

上　村　　務††

1. はじめに

Smalltalk[7] によって代表されるオブジェクト指向の概念を他の言語と結合する試みは近年盛んに行われている．これはオブジェクト指向のもたらす次のような利点を積極的に活用しようとするものである．

a） オブジェクトを基盤にすることで，問題の構造をプログラムの構造に自然に反映することができる．

b） オブジェクトの自律性により，プログラムの各要素のモジュール化が促進される．

c） 継承の仕組みにより，プログラムの共有，再利用が積極的に行える．

d） 実行される手続き（オブジェクトのメソッド）が動的に決定されることを用いて，柔軟性に富むコントロールが実現できる．

これに対して従来の言語は Pascal, C, Modula-2, Ada などの手続き型言語が多く，それらは次のようないくつかの共通の基盤をもっている．

a） ハードウェアの処理を基本にして，それを抽象化し構造化した式の評価，代入文，条件文，くり返し文などを処理の基本にする．

b） データは型の概念を用いてその抽象化と構造化がなされる．

c） 関連のある処理とデータは，手続きや関数などのブロック構造としてグループ化され，プログラムはブロック構造によって組織化される．

手続き型言語とオブジェクト指向の結合は，こうしたオブジェクト指向の利点と手続き型言語の特徴をともに実現する目的をもっている．この結び付きにはいろいろな形が考えられようが，その背景としてやや異なった二つの立場があると思われる．

一つは，オブジェクト指向の利点を中心に考えて，それらをできるだけ多く実現しようとする立場である．ここではたとえば，オブジェクトのメソッドを手続き型の言語での処理のような形で与えることや，数や記号などはオブジェクトとはみなさず，通常のデータタイプとして利用するなどが考えられる．しかし，メソッドの動的な決定によるコントロールの柔軟性はオブジェクト指向の重要な利点と考えて Smalltalk のような形で完全に実現されることが多い．この立場ではプログラムの実行効率よりも柔軟性に重点が置かれることが多く，作られる言語はプロトタイピングの用途などを重視した言語と考えられる．

もう一つの立場は，手続き型言語を中心として，それにオブジェクト指向による利点を加えようとするものである．手続き型言語は上に述べた基盤をもっているが，ブロック構造による組織化だけでは大規模でかつ複雑なプログラムに対処するには不十分である．そのために多くの言語ではプログラムのモジュール化，各モジュールのインタフェースとインプリメンテーションの分離や分割コンパイルなどが実現されている．しかし，このモジュール化はプログラムを小さなプログラム要素の集まりに分割することを目的としているのに対して，オブジェクトを用いるプログラム作成では，問題の構造をオブジェクトとその集まりの構造に自然に反映させることができ，その結果としてプログラムの構造化及び分割化が期待できる．また強い型がついた手続き型言語では動的にデータの型が変わる計算は不可能かあるいは非常に制限されてしまうが，オブジェクト指向ではそれを積極的に利用する．手続き型言語にこうしたオブジェクト指向の利点がもたらす影響は大きく，言語をそうした形に拡張することは興味深いテーマを提供する．特に，手続き型言語の中にはたとえばCのように効率を非常に重視する言語も多く，実行効率とオブジェクト指向を両立させる問題などは重要でかつ難しいテーマである．

本解説では手続き型言語にオブジェクト指向を導入する場合に考慮すべき主要な概念と問題点，それらの解決法について概説する．

オブジェクト指向言語の設計において考慮すべき基

† Object-Oriented Extensions of Procedural Languages by Tsutomu KAMIMURA (IBM Research, Tokyo Research Laboratory).

†† 日本アイ・ビー・エム東京基礎研究所

本的な概念については，たとえば，P. Wegner[27]によって分類されているが，オブジェクトとクラスの概念，そして継承の概念が基本的なものである．さらに手続き型言語との結び付きとして重要と考えられるのが型の概念である．以降では，まず次章で現在報告されている主な言語について簡単に紹介した後，これらの基本的な概念について解説を試みる．

2. 代表的な言語例

現在発表されているオブジェクト指向型の手続き型言語はかなりの数にのぼっているが，どういうわけかその主なものはCを中心にしたものが多い．その代表的なものは，Objective-C, C++それに Class C である．Objective-C[3),4)]と C++[21)~24)]はそれぞれ PPI, AT & T ベル研で作られた言語でともに良く知られている．

Class C[6)]は 1984 年に IBM のワトソン研究所で作られたもので，IBM 外部にはあまり紹介されていないが，内部では，CAD/CAM 用システムや PC 上のグラフィックスのツールキット作成などに使われている．Class C は Objective-C のようにメッセージとメソッドの結合を完全に動的に行い，メッセージのパターンをクラスとは独立した形で行うなどの特徴をもっている．

これらの言語のほかにCを基本にした言語としては，ブラウン大の Object C[13)]，三菱の Super C, CNS による Ctalk などが発表されている．

また筆者らは，やはりCを基本にした言語で COB (C with OBjects)という新しい言語を開発している[9)]．COB の特徴は，Cの効率をできるだけ落とさずオブジェクト指向を実現すること，ファイルごとの情報隠蔽により大規模なプログラミングをサポートすること，強い型づけの導入により信頼性の向上をはかることなどである．

こうしたCの拡張はプリプロセサによって実現されるものが多く，ソースプログラムと生成されるCのコードの対応が分かりにくく，デバッグなどが不便になりやすい．CMU の Andrew プロジェクトでは，この欠点をもたないようにヘッダファイルのみをプリプロセスしてオブジェクト指向を実現する Andrew Class System[15)]が考案されている．

C以外の言語を基本にしたものとしては，Trellis/Owl, Object Pascal, Neon, Eiffel などが知られている．

Trellis/Owl[16),17)]は DEC の Eastern Research Laboratory で作られた言語とその環境で，その基本を CLU に置いている．CLU の伝統である iterator や type generator などをもち，強い型付けによるオブジェクト指向プログラミングを目指している．

Object Pascal[18),25)]と Neon[18)]は Macintosh 上で使われている言語である．Object Pascal は名前のとおり Pascal 上に基本的なクラスの機能を導入したもので，Apple によって作られ，そのクラスライブラリは MacApp と呼ばれている．Neon は Kriya Systems によって開発された FORTH を基本にした言語である．

Eiffel[11),12)]は B. Meyer による Pascal ふうの独自の構文や強い型付け，アサーションによるプログラムの状態の検証などの機能をもった言語でCのプリプロセサとして実現されている．

このほかにオブジェクト指向に関連しては，Common Loops[1)], Common Objects[20)], Flavors[14)] などの LISP 型の言語や，言語ではないが Mesa のコーディング手法として考えられた Traits[5)] と呼ばれる手法などが報告されている．

ACM 主催の OOPSLA の 1986 年の会議における調査によると，この会議の出席者が使用しているオブジェクト指向言語の割合は，やはり Smalltalk が一番で全体の 44% を占め，次いで，Objective-C, Class C, C++などのCを基本にしたものが 23%，そして LISP 系の言語が 17% という結果がある．これからみても，手続き型言語とオブジェクト指向の結びつきは圧倒的にCを基本にしたものが多いことが分かる．本解説ではこうした背景，及び筆者の COB による経験を基に，Cを基本にした言語を中心に説明を行うことにする．しかしながら多くの議論は手続き型言語全般に共通するものである．

3. オブジェクトとクラス

オブジェクトとは，データとそれを操作するメソッド* と呼ばれる手続きをカプセル化した概念である．そしてクラスは同様の形をしたオブジェクトを共通に記述する．言語によっては SELF[26)]のようにクラスの概念をもたないものもあるが，クラスをもつ言語がはるかに一般的である．

* 言語によってはメソッドという名前を用いないものもあるが，ここではこの名前で統一する．

3.1 クラス

クラスについてまず問題となるのは，クラスをオブジェクトとみなすかどうかである．クラスをオブジェクトとみなすと，計算のモデルがオブジェクトのみで組み立てられ統一的になり，クラス変数やクラスメソッドの扱いが通常のオブジェクトでの扱いと同様になる．これは Smalltalk が導入したモデルで，Objective-C や Class C でも原理的にこうした扱いをしている．クラスをオブジェクトとみなすと，当然クラスのクラス，すなわちメタクラスが考えられ，たとえば Common Loops などではメタクラスをユーザが定義できることによって，メソッドの探索などの部分をも可変にする非常に柔軟な機構の実現などが考えられている．

これに対してクラスをオブジェクトではなく，オブジェクトの型であると考えることもできる．C++，COB, Trellis/Owl, Object Pascal などはこうした立場である．クラスを型と考えることは型のついた言語においては自然であり，またオブジェクトとクラス，メタクラスの階層がなく仕組みが簡単になる．しかし，この場合はクラス変数やクラスメソッドをオブジェクトとは別に扱う必要がある．この場合にはそうした変数や手続きは COB のようにクラスの中の特定の個所で定義されるか，C++ や Trellis/Owl のように特別の構文によって定義される．

3.2 インタフェース

オブジェクトによるカプセル化により，オブジェクトへのアクセスは定められたインタフェースをとおしてなされる．このカプセル化の概念はすべてのオブジェクト指向言語で共通しているが，実際のインタフェースはかなり多様性をもっている．

Objective-C と Class C ではメソッドはCにはない独自の概念として定義され，オブジェクトへのアクセスはすべてメソッドを介して行われる．

これに対して C++, COB, Trellis/Owl, Object Pascal などでは，メソッドは通常の関数として定義される．さらに C++ と COB では関数だけでなく変数も public にできる．Trellis/Owl ではクラス*中でコンポーネントと呼ばれるデータを public にすることができるが，これはデータに対して get と put という関数が自動的に作られ，それらが public なものとみなされる．

オブジェクトのアクセスは，外部では public なものをとおしてのみ行うのがオブジェクト指向の基本であり，多くの言語ではこれが守られている．これに対して C++ と COB ではクラスの中で定義されている関数は自由にそのクラスの任意のオブジェクトの内部を参照でき，クラスがデータ抽象の色彩を強くもっている．

インタフェースとしては，内部用の private と外部用の public の区別が一般的である．この点で問題になるのは，継承がある場合にスーパクラスの内部への参照を許すかどうかである．Smalltalk ではスーパクラスで定義されるインスタンス変数はオブジェクト自身の一部とみなされ，それらへのアクセスは内部からのアクセスと考えて許される．Objective-C, Class C ではこの点で Smalltalk に忠実である．

しかし，これはクラスの情報隠蔽の機能を制限してしまう．つまり，クラスへの内部アクセスを目的に勝手にサブクラスを作れることになる．COB などの言語ではスーパクラスを特別に扱うことはせずにサブクラスからのアクセスは public なもののみにしてクラスによる情報隠蔽を守っている．Trellis/Owl や最近の C++ では，サブクラスからのアクセス用に private, public に加えて別の種類のインタフェースを用意している．Trellis/Owl ではさらにオブジェクトのアロケーションの場合にのみ用いる四番目のインタフェースがある．

COB には，異なったインタフェースの種類をふやす替わりに，クラスのインタフェースを各コンパイル単位ごとに変える機能がある．これはクラスビューといわれ，本来のクラスでの public な部分集合を作って別のコンパイル単位に供給するもので，コンパイル単位ごとのクラスの情報隠蔽の機能が強化される．これはコンパイル単位での変数の影響を制御し，再コンパイルを少なくするために用いられる．

3.3 オブジェクトの生成と管理

Smalltalk や Trellis/Owl, Eiffel, それに LISP を基本にしたシステムではオブジェクトは動的にヒープ上に生成され，ガーベジコレクションやリファレンスカウントによってオブジェクトのメモリ管理が行われる．

これに対してCなどによるシステムプログラミングの分野では不定期的に起こるガーベジコレクションのオーバヘッドは効率的に許容範囲外である場合が多い．実際Cを基本にしたオブジェクト指向言語では自動的なメモリ管理をしているものはない．しかし，常

* Trellis/Owl ではクラスではなくタイプモジュールと呼ばれる．

にオブジェクトのメモリ管理をユーザに任せることはユーザの負担を重くするし，せっかくクラスによって情報隠蔽を行っても，クラスの供給側と使用側とでメモリ管理についての共通の理解が必要になり，情報隠蔽によるモジュール化の機能が制限されてしまう．こうした言語ではスタック上にもオブジェクトを作れるものが多く，スタック上のオブジェクトの管理はブロック構造による管理で行うことができる．しかし，ヒープ上とスタック上でのオブジェクトのメモリ管理についてユーザの理解が必要であることには変わりはない．

C++ ではユーザによるオブジェクトの管理を軽減するためにクラスのコンストラクタとディストラクタが用意されている．これらはそれぞれクラスのオブジェクトが作られるときと消滅するときに自動的に実行される関数である．コンストラクタによりオブジェクトの内部データの初期化を行うようにしておくと，スタック上のオブジェクトはブロックに入ると自動的に初期化されたオブジェクトが作れるなど便利である．コンストラクタ，ディストラクタを用いるとある程度のメモリ管理を行うことができ，リファレンスカウントを用いて counted pointers という方法が考えられている[24]．

そのほかのCを基本とした言語ではメモリ管理はすべてユーザに任せられ，スタック，ヒープ上のオブジェクトはともに関数呼び出しやメソッドの実行によって明示的に作られるものが多い．

3.4 メソッドの実行

メソッドの呼び出しと実行されるプログラムの結合の時点は言語によって異なる．より柔軟なコントロールやプログラムの修正の容易さを重視すれば実行時に行うことになるし，実行効率を高め，誤りをできるだけ早い時期に検出しようとすればコンパイル時に行うことになる．Smalltalk をはじめ，Objective-C や Class C は前者である．さらに，Class C では柔軟性をあげるためにメソッド呼び出しのパターンを変数を用いて実行時に決めることができる．また，Trellis/Owl や Object C, Andrew Class などでもこの結合は実行時に行われる．これらの言語ではメソッド呼び出しは関数呼び出しの構文を用いるが，ポインタを介した関数の間接呼び出しとして実現される．

C++ と COB では原則的にこの結合をコンパイル時に行う．しかし，すべてコンパイル時に行うと，冒頭で述べたオブジェクト指向の重要な利点が損なわれてしまう．このため，これらの言語では，仮想関数 (virtual function) と呼ばれる関数の呼び出しについてのみ，実行される関数への結合を実行時に行っている．このようにするとユーザが普通の関数と仮想関数を区別しなければならないが，普通の関数の呼び出しの実行効率を落とさずに済むことになる．

4. 継　　承

継承はオブジェクト指向における基本的な概念であり，一般的には種々の情報，資源の共有の機能と考えられる．

4.1 クラスと継承

継承はクラスの間にスーパクラス，サブクラスの関係を導入することで行うのが一般的であり，実際，2. で紹介した言語すべてはこの点で共通している．しかし，継承☆ は必ずクラスをとおしてなされるとは限らず，delegation[10]の概念によってオブジェクト間での継承の仕組みが考えられる．たとえば SELF ではクラスはなく，継承は delegation によっているし，最近の C++ では，オブジェクトへのポインタによる delegation が考えられている[22]．

さて，継承とはいったい何を継承するかをまず考える必要があろう．クラスはそのインタフェースとインプリメンテーションによって定義されることを考えると，クラスの継承にはインタフェースの継承とインプリメンテーションの継承が考えられよう．Smalltalk ではこの二つの継承が一致している．Objective-C や Class C などではこの点でも Smalltalk のモデルに忠実である．

この二つの継承を分離する利点は，インタフェースとインプリメンテーションの分離である．たとえば，あるクラスのメソッドを継承する場合にインタフェースとインプリメンテーションの継承を分離しておくと，インプリメンテーションの継承を変えた場合にもクラスのユーザには影響が出ない．しかし二つの継承を常に分離すると，同一の形の継承が多い場合には区別がわずらわしくなる．多くの言語ではインタフェースの継承はまず自動的にインプリメンテーションの継承を意味し，継承しないものについてはインプリメンテーションを新しく与えることができる．

C++ と COB における仮想関数はインタフェースのみの継承と考えることができよう．また C++ で

☆ 狭い意味では継承 (Inheritance) は delegation と区別される場合もあるが，ここではこれを含むと解釈する．

はスーパクラスを private にして，インタフェースの継承なしにインプリメンテーションを継承することができる．COB ではこの機能はないが前述のクラスビューを作ることでビューを用いるユーザに対しては二つの継承を分離することができる．

4.2 単純継承と多重継承

単純継承の利点は言語仕様，インプリメンテーションが簡単になることである．Smalltalk をはじめ，Objective-C, Class C, Andrew Class, Object C, Object Pascal などは単純継承を採用している．

多重継承では，継承の機能がより一般的で強力になる．クラスの複数の側面を抽象化してそれぞれスーパクラスを作ることでより強力で自然なプログラムの構成が可能になり，継承による記述力，組織化の機能が向上する．また複数のクラスが共有できる性質をそれらのスーパクラスとすることで情報の共有化がさらに促進できる．（これらのクラスにすでに別のスーパクラスがある場合には単純継承では扱えない．）LISP 系の言語では “mixin” というクラスを用いてこうした利点を積極的に生かすプログラミングスタイルが使われている．

多重継承を採用している言語は LISP 系の言語に限らず，COB, Trellis/Owl, Eiffel, Traits なども含まれ，最近の C++ でも多重継承が実現されている[22]．Smalltalk でも多重継承を導入する試みがなされ，クラスによる多重継承[2]のほかにオブジェクトに perspective という概念を用いてオブジェクトレベルでの多重継承を行う試みも考えられた[8]．

多重継承ではどうしても継承の仕組みが複雑になり，言語仕様の Semantics，インプリメンテーションに十分な注意を要する．言語仕様での中心的な問題はメソッドの決定法及びスーパクラスのデータの扱いなどであり，これらは以降の節で考えることにする．

インプリメンテーションの複雑さも重要な問題で，特に実行効率を重視する場合には十分に考える必要がある．Smalltalk[2]では単純継承のインプリメンテーションを基本にして，複数のスーパクラスのメソッドはそれぞれのサブクラスであらかじめコンパイルしておく方法をとっている．C++[22]ではオブジェクト内の各データのオフセットがコンパイル時に決定できるような言語仕様になっており，多重継承によりオブジェクトの参照の効率が落ちないよう工夫されている．

4.3 スーパクラスのデータ

サブクラスからみたスーパクラスのデータの扱いにはいくつかの形がある．Smalltalk では前に述べたようにスーパクラスの内部はサブクラスより参照でき，スーパクラスのデータはサブクラスのデータの一部とみなされる．したがって，スーパクラスでのインスタンス変数の名前がサブクラスで使われてはならず，もし使われた場合には同一の変数を指すことになる．Common Loops でも同様である．

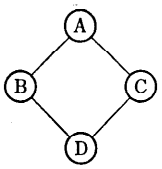

図-1 クラスの階層

しかしながらこの方法はスーパクラスの private な部分がサブクラスに公開されない場合には情報隠蔽に反することになり問題となる．そのため多くの言語ではスーパクラスのデータはサブクラスとは独立した別のものであると考えられる．

多重継承の場合にはさらに次のような問題がある．**図-1** のクラスのハイアラーキを考えよう．ここで各ノードはクラスを表し，実線は継承を示している．上のクラスがスーパクラスで下がサブクラスである．さて，ここでクラスAはクラスDのスーパクラスであるためDのオブジェクトはAでのデータを含んでいる．このとき，Dのオブジェクトが Aのデータを一種類持つ場合を共有化，二種類持つ場合を多重化と呼ぶとする．

共有化が自然な場合はたとえば**図-2** のような場合である．window はスクリーン上に表示するデータをもっており，named-window, captioned-window は

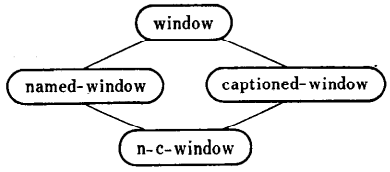

図-2 クラスの階層——例1

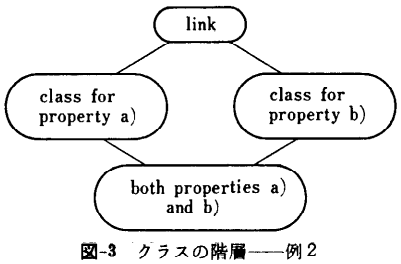

図-3 クラスの階層——例2

それぞれ window に名前，caption をつけたものである．ここで名前と caption の両方をつけた n-c-window を作ったとする．この場合明らかに window のデータは共有化するのが自然である．

これに対して**図-3** の構造を考えよう．ここで link はサブクラスのオブジェクトをリンクする目的で作られたクラスである．ここである性質 a)についてオブジェクトをリンクするクラスと性質 b)についてリンクするクラスを別々に作ったとする．そして a)と b)の両方の性質をもったオブジェクトをそれぞれの性質ごとに異なった二つのリンクを作りたいとする．この場合はクラス link が多重化されてなければならない．

Trellis/Owl, Traits, Common Loops などでは共有化を採用し，Common Objects では多重化を行っている．どちらが言語としてより有効であるかは議論のあるところであるが，モジュール化をくずさない点では多重化の方が理解しやすいといえる．共有化の場合には，複数のクラスの共通のサブクラスを作る際には必ずそれぞれのクラスのすべてのスーパクラスを検討してどのスーパクラスが共有されるかを常に理解する必要がある．

しかし，一方のみの機能では不十分な場合も考えられ，C++ と COB ではディフォルトを多重化の形で実現し，C++ では仮想クラス (virtual class), COB では各継承における共有化の明示によって共有化が可能となっている．

4.4 メソッドの決定

メソッド呼び出しにおいて実行されるメソッドを決定する規則は多くの議論のあるテーマである．単純継承の場合はほとんど Smalltalk のモデルに従っているが，多重継承ではいくつかの方法が考えられている．大きく分けて，一次元化の方法と，グラフ構造を用いる方法がある．

一次元化の方法とは，クラスのすべてのスーパクラスをある方法によって全順序化し，その順序に従ってメソッドをもつクラスで順序上で最初のクラス中のものを用いる方法である．この順序の決め方は Flavors ではスーパクラスの階層のグラフを左から右へ depth-first などでたどる方法がとられている．Common Loops では depth-first up to join という方法で，左から右へ depth-first でたどるがそれを必ずサブクラスはスーパクラスより前に出るように修正した方法を用いている．たとえば**図-4** のクラスの階層では数字による順番でクラスが並べられメソッドが探索される．

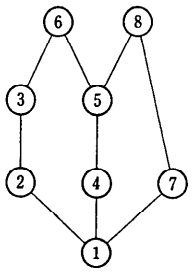

図-4 Depth-first up to join

この方法の興味深いところは，順序上の次のクラスが一定していず，たとえば super というキーワードで実行時によって決まる全順序上の次のクラスを用いるプログラムが書ける点である．図-2 でたとえば named-window と captioned-window はともに display というメソッドをもち，その中で super の display を呼んでいるとする．このとき named-window のオブジェクトに対しては named-window 中の super は window を指し，n-c-window のオブジェクトでは named-window 中の super はクラスの順序を決める規則に従って captioned-window を指すことになる．この方法はこのように非常に柔軟性に富む機能を与えてくれるが，動的に変わる部分が増え，ユーザによる注意深い管理が求められる．

グラフ構造を用いる方法は，継承の階層を示すグラフ構造をそのまま用いてメソッドを決定する方法である．この方法で一般的なのは，スーパクラスのメソッドを継承する場合は，異なったスーパクラスからの同じ名前のメソッドの継承はそれらがさらに上のスーパクラスで定義された同一のものである場合に限る方法である．Smalltalk, Traits, Eiffel, Trellis/Owl, C++ などは基本的にこの方法を用いている．しかしこの方法では，たまたま同じ名前の異なったメソッドは継承できなくなる．この場合は一方の名前を変えるか，同じ名前でサブクラスで新しくメソッドを作るかしなければならない．またこれらの言語ではこうした名前のあいまいさが生じた場合，必要なメソッドを定義しているクラス名をメソッド呼び出しで指定することができる．

COB では，クラスの再利用の度合いを高めるためにこの規則を変更して，メソッドを継承する場合は，メソッドを定義しているクラスのなかで，グラフの構造上最も現在のクラスに近いスーパクラス☆ がある場

☆ つまり，そのメソッドと同じ名前のメソッドをもつ任意のスーパクラスは必ずこのスーパクラスのスーパクラスになっている場合である．

合にはそのクラスのメソッドを用いるようにしている．さらに，メソッドを用いる場合にオブジェクトを指定する式の型に関連のないスーパクラスはメソッド決定の際には考慮しないために，同一の名前で異なるメソッドの継承も可能である．

5. 型について

型については本誌の 2.4 で説明されているので，一般的な議論は省略し，それぞれの言語での扱いについてのみ触れる．

手続き型言語でも Objective-C や Class C などはオブジェクトはすべて同一の型をもち，オブジェクト指向の部分については型のないプログラミングスタイルをとる．

これに対して，C++，COB，Trellis/Owl，Object Pascal などではクラスによる型づけがなされる．

型づけを行う場合にまず問題となるのは，サブタイプの概念である．サブタイプとはある型（サブタイプ）を別の型（スーパタイプ）の一部とみなす機能である．クラスを型とする立場ではクラスの継承によってサブタイプが導入されると考えられる．型づけを行う言語では，クラスのインタフェースの継承がサブタイプとスーパタイプの関係を導入するとみなされる．

前に述べたように，C++ と COB では型づけを利用してコンパイル時にできるだけ解析を行い実行効率を高める方法をとっている．このためオブジェクト指向の動的な部分を扱うには特別の機能が必要になり，前述の仮想関数がこの役割をもっている．C++ と COB では型の強さにやや違いがあり，C++ では実際のオブジェクトに関係がなくその型を cast によって変換できる場合があるが，COB ではそのような変換は許されない．

強い型づけを行うと継承によってスーパクラスを共有する場合に問題が生じる．たとえば，link というクラスを作りそのデータとして link のオブジェクトへのポインタを作り，next というメソッドによってこのポインタを取り出すとする．ここで link のサブクラスを作るとこのサブクラスのオブジェクトをリンクできるが，このクラス中で next を呼び出すと link へのポインタが取り出され，それをサブクラスへのポインタへと型変換を行う必要がある．C++ 及び COB では型変換は実際に実行文を含んだ操作であり，この部分はオーバヘッドとなるし，C++ でスーパタイプからサブタイプへの変換は必ずしも安全ではない．link がほかのサブクラスをもつ場合もあり得るために必要な型変換とも考えられるが，もし link 中のポインタの型がパラメータ化されていると，安全な型づけでしかもオーバヘッドなくエレガントに扱うことができる．パラメータをとるクラスは Trellis/Owl，Eiffel そして COB において実現されている．CLU などの一般的な言語では，パラメータは型のみならずコンパイル時に決まる定数を用いることができ，実パラメータを指定するごとにソースコード上で代入を行いコンパイルする方法を用いているのに対し，COB や Trellis/Owl ではパラメータは型に限られ，クラスのコンパイルは仮パラメータをもつものを一回コンパイルするのみで，実パラメータは個々の場合の型情報を区別し必要な変換を行うために用いられる．

5. おわりに

手続き型言語にオブジェクト指向を導入する上で問題となる基本的な概念について概説した．各言語はそれぞれの背景，設計思想，及び適用分野をもっておりそれらによって導入される個々の概念が異なり非常に多様な形になっている．

オブジェクト指向と手続き型言語の結びつきは単に言語にとどまらず，プログラミングの方法論や環境においても当然考えられるべきである．Smalltalk のような総合環境がオブジェクト指向の手続き型言語においても十分想定されるし，こうした言語によるプログラミングの方法論の研究も重要なテーマであろう．

分散処理や高度なユーザインタフェースなどの普及にともなってプログラム開発が大規模化，複雑化しており，それに対処するパラダイムとしてオブジェクト指向が期待されて久しい．これを背景に，Sun Microsystems による C++ を用いた UNIX System V の書き換えに象徴されるように，新しい言語によるオブジェクト指向の本格的な普及は着実に進んでいる．

参 考 文 献

1) Bobrow, D. G. et al.: Common Loops: Merging Lisp and Object-Oriented Programming, Proceedings of ACM OOPSLA '87 (Object-Oriented Programming Systems, Languages and Applications), pp. 17-29 (1986).
2) Borning, A. H. and Ingalls, D. M.: Multiple Inheritance in Smalltalk 80, Proc. of National Conf. on AI (AAAI 82), pp. 234-237 (Aug. 1982).
3) Cox, B. J.: Message/Object Programming—An

Evolutionary Change in Programming Technology, IEEE Software, pp. 50-61 (Jan. 1984).
4) Cox, B. J.: Object Oriented Programming—An Evolutionary Approach, Addition-Wesley (1986).
5) Curry, G. et al.: Traits—An Approach to Multiple Inheritance Subclassing, ACM Conference on Office Automation Systems (June 1982).
6) DeNatale, R. J.: ClassC-Extensions to Object-Oriented Programming, IBM T. J. Watson Research Center (Feb. 1984).
7) Goldberg, A. and Robinson, D.: Smalltalk 80: The Language and Its Implementation, Addition-Wesley (1983).
8) Goldberg, I. P. and Bobrow, D. G.: Extending Object-Oriented Programming in Smalltalk, Proc. of LISP conference (Aug. 1980).
9) Kamimura, T. et al.: COB Reference Manual, IBM Tokyo Research Laboratory (Mar. 1988).
10) Lieberman, H.: Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, ACM OOPSLA '86 Proceedings, pp. 214-223 (1986).
11) Meyer, B.: Reusability: The Case for Object-Oriented Design, IEEE Software, pp. 50-62 (Mar. 1987).
12) Meyer, B.: Eiffel: Programming for Reusability and Extendability, SIGPLAN Notices, Vol. 22, No. 2, pp. 85-94 (1987).
13) Meyrowitz, N.: Object C Report, IRIS (Institute for Research in Information and Scholarship), Brown University (June 1985).
14) Moon, D. A.: Object-Oriented Programming with Flavors, ACM OOPSLA Proceedings, pp. 1-8 (1986).
15) Palay, A. J.: The Andrew Class System, manuscript.
16) Schaffert, C., Cooper, T. and Wilpot, C.: Trellis—Object-Based Environment, Language Reference Manual, Technical report DEC-TR-372, Degital Equipment Corporation (Nov. 1985).
17) Schaffert, C. et al.: An Introduction to Trellis-Owl, ACM OOPSLA '86 Proceedings, pp. 9-16 (1986).
18) Schmucker, K.: Object-Oriented Languages for the Macintosh, Byte, August, pp. 177-185 (1986).
19) Snyder, A.: Encapsulation and Inheritance in Object Oriented Programming Languages, ACM OOPSLA '86 Proceedings, pp. 38-45 (1986).
20) Snyder, A.: CommonObjects: An Overview, SIGPLAN Notices, Vol. 21, No. 10 (1986).
21) Stroustrup, B.: The C++ Programming Language, Addition-Wesley (1986).
22) Stroustrup, B.: Multiple Inheritance for C++, Proc. of EUUG (European Unix-User Group) Spring Conference (May 1987).
23) Stroustrup, B.: What is "Object-Oriented Programming?", Proc. of ECOOP (European Conference on Object-Oriented Programming) (June 1987).
24) Stroustrup, B.: Possible Directions for C++, Proc. of USENIX C++ Workshop (Nov. 1987).
25) Tesler, L.: Object Pascal Report, Structured Language World (1985).
26) Unger, D. and Smith, R. B.: Self: The Power of Simplicity, ACM OOPSLA '87 Proceedings, pp. 227-242 (1987).
27) Wegner, P: Dimensions of Object-Oriented Language Design, ACM OOPLSA '87 Proceedings, pp. 168-182 (1987).

(昭和62年10月1日受付)