Vol. 30 No. 4　　情　報　処　理　　Apr. 1989

解　説

1. プログラミング言語と環境

1.3 オブジェクト指向言語のプログラミング環境†

——Smalltalk-80 における環境を例に——

横手　靖彦††

1. はじめに

プログラミングとは一種の人間の芸術的知的生産活動である．したがって，それを支援する環境に関する議論は，プログラミング言語やそれを利用するユーザの要求や技量に大きく依存するため，どのような環境がよいか優劣をつけることは難しいし，どのような機能があれば十分かを決めることも難しい．しかし，プログラミング言語を特定する，あるいはプログラミングパラダイムを特定する，などの制約を与えることによってこの困難さは軽減されると同時に議論を行うことの意義も出てくる．

本稿では，プログラミング環境に関する議論のまとをオブジェクト指向言語に絞ることにする．とくに，Smalltalk-80 を中心とするオブジェクト指向言語におけるプログラムの作成，およびそのデバッグを支援する環境について解説する．本特集は，「新しいプログラミング環境」に関するものであり，「新しい」という観点からすると，Smalltalk-80 のプログラミング環境は当てはまらない．しかし，Smalltalk-80 のプログラミング環境がもつ，オーバラッピングウィンドウや，メニューによるコマンド実行，モードレスエディタ，言語とその支援環境の統合化，オープンシステムによるユーザ独自の環境の構築などは，その後のプログラミング環境に少なからず影響を与えている．したがって，Smalltalk-80 のプログラミング環境を知ることは，本特集の他の解説を理解するうえでも，また今後のプログラミング環境について考えていくうえでも必要なことであろう．

本稿では，オブジェクト指向計算モデルに関しては，ある程度の知識があるものとして話を進めていく．オブジェクト指向言語のプログラミング環境に関しては，さまざまなところで解説されている[4),15)]が，ここでは，以下のような内容を述べることとする．**2.** において，まず，オブジェクト指向言語を用いてのプログラムの作成支援環境について述べる．ここでは，どのような機能がプログラミングを支援するために必要か，という点を中心に議論する．さらに，デバッグ支援に関しても述べる．**3.** において，Smalltalk-80 におけるプログラミング支援環境の実際をみていくことにする*．「百聞は一見にしかず」という言葉どおりに，本章では，実際の操作例を多く用いることによって，どのような機能が具備されているかをみていくことにする．さらに，Smalltalk-80 におけるデバッグ環境の実際を概観する．**4.** において，プログラミング環境をリフレクティブ計算の観点より議論する．特に，リフレクティブ計算のデバッグ支援への応用に関して議論する．また，Smalltalk-80 のデバッグ機能をリフレクティブ計算の観点より分析してみる．

2. オブジェクト指向言語のプログラム作成とそのデバッグのための支援環境に必要なもの

プログラミング環境は，プログラミングという一種の芸術的な人間の知的生産活動を，支援するものである．そのためには，システムはユーザの考えていることをなんの変換の必要もなくそのままの形で引き出せるように支援する必要がある．さらに，システムはな

† A Programming Environment in an Object-Oriented Programming Language by Yasuhiko YOKOTE (Sony Computer Science Laboratory Inc.).

†† (株)ソニーコンピュータサイエンス研究所

* ここで取り上げるのは，Parc Place 社よりリリースされている，仮想イメージバージョン 2.2 をもとにして解説する．

にを提供できるかをユーザに示せるとともに，ユーザからのさまざまな要求に柔軟に応えることができなければならない．

実際には，プログラミング環境は用いるプログラミング言語に大いに依存すると同時に，それを用いる人間側の要求や，技量にも大きく依存する．したがって，プログラミング環境を考える際にはプログラミング言語とは切り離して考えることはできない．本章では，オブジェクト指向プログラミング言語における，プログラミング環境に関して，その必要事項を探っていく．

2.1 プログラミング支援

オブジェクト指向言語は，クラスの概念をもつものと，もたないものに2分できる．前者には，Smalltalk-80[3]，Flavors[10]，Eiffel[8]，などがあり，後者には，アクタモデルの流れを汲む，Actors[1]，ABCL/1[21]，などがある．しかし，これら言語の違いは，プログラミング支援環境の必要事項を探る際には重要ではない．重要なことは，これらの言語がどのようなプログラミングスタイルをとるか，ということである．オブジェクト指向言語を用いたプログラミングにおいては，プログラミングに際して，まず問題領域の何をオブジェクトとして表現するかを決めることが重要である．また，クラス階層のあるシステムでは，表現したオブジェクトを，いかにしてクラス階層の中に整理するかが重要になる．この整理の仕方次第では，アプリケーション全体の効率はもとより，プログラムの読みやすさ，ひいては，保守のしやすさ，デバッギングのしやすさにも影響を与えることになる．

よく，オブジェクト指向言語を用いたプログラミングには，探索型のプログラミングスタイルが向いている，といわれる．これは，今までに作成されたクラスの機能を再利用しながらプログラミングを進めていくことが多いとともに，オブジェクト指向言語のもつ動的束縛の機構は，すべてのプログラムを完成させることなく，プログラムの作成およびテストが可能であるからである．

探索型プログラミングを分析してみると，次の5つのステップが互いに密接に関わり合いながら存在しているのがわかる．

- 理解．問題の分析とともに，既存のクラス群の中から，どのクラスが問題解決に利用できるかを理解する段階．
- 設計．問題領域のなにをオブジェクトとして表現するか？ どのようにクラス階層を構築するか？ などの，モデリングの段階．
- 入力．組み立てたモデルをプログラム言語として表現する段階．
- 実行．プログラムを実際に実行させてみて，問題が解決しているかをみる段階．
- デバッグ．望みどおりの動作をしない，プログラムが終了しない，などの誤りを修正する段階．

探索型プログラミングでは，入力，実行，デバッグの段階は，一つのサイクルを成しており，これは状況によっては設計の段階をも含むことがある．また，理解の段階は設計，入力，デバッグの段階と関連をもっている．

したがって，オブジェクト指向言語におけるプログラミング支援環境として，おおむね次の基本的な機能を支援する必要がある．

R1：オープンシステムであること☆．あらゆるユーザの要求をすべて満たすようにシステムをあらかじめ構築しておくことは不可能であるので，ユーザが自分の要求を満足するようにシステムを自由に変更や自己拡張できる機構を提供する必要がある．また同時に，以下に述べるような機能は互いに密に結合化されている必要がある．

R2：検索機能．システムにはすでにさまざまのクラスが定義されているが，ユーザは自分の望む機能がこれらのクラス群の中にあるかどうか，またある場合にはそれが自分の望む機能とどう違うかを，効率よく知る必要がある．検索機能が充実していればいるほど，理解の段階を効率よく行うことができ，設計や入力の段階にも影響を与える．

R3：差分コンパイル．プログラムの全体ができてない状態でもテスト実行できる必要がある．また一部の修正が全体に影響を及ぼすようなことがあってはならないし，そのためにすべてのモジュールを再コンパイルするようなことがあってもならない．

R4：デバッガ．次節で詳しく述べる．

2.2 デバッグ支援

プログラミングの大部分の時間はデバッグに費やされる．したがって，デバッガのできの優劣はプログラムの生産性に大きく影響を与える．計算機上での表現と，人間の思考上での表現の間に隔たりがある以上，両者の間のマッピングはどうしても必要なことである．誤りは人間の思考そのものにある場合もあるし，

☆ ここでの「オープンシステム」は単にさまざまのハードウェアが接続できる，という意味ではない．

マッピングの際に生じるものもある．いずれにせよ，デバッガの仕事の本質は，「誤りの原因を適切に指示すること」である．誤りの原因がシステム側で分かってしまえば，システムはそれを修正して実行を続行すればいいわけである．誤りには次の二つが考えられる．

- 文法的なもの．マッピングの際に生じる誤りであり，プログラミング言語の文法誤り，単語のスペル誤り，などの誤り．
- 意味的なもの．問題領域のモデル化の誤り，アルゴリズムの誤り，などの誤り．

文法的な誤りは，コンパイラによって検出可能である．しかし，意味的な誤りの検出は簡単にはいかないので，誤りの原因を突き止めるための支援環境が必要になる．

3. Smalltalk-80 におけるプログラムの作成とデバッグのための支援環境

2. での議論が実際のシステムにおいてどのように実現されているかを，Smalltalk-80 を例に述べる．Smalltalk-80 のプログラミング言語としての特徴に関しては，いまさら詳しく述べる必要はないと思われるが，本章で述べる議論の助けになる程度にその特徴を簡単にまとめておく．

オブジェクトとメッセージ：Smalltalk-80 における計算は，オブジェクト同士がメッセージを交換し合いながら進んでいく．したがって，Smalltalk-80 のプログラムは，基本的にあるオブジェクトにメッセージを送るというメッセージ式のみである．

メッセージの受信によりオブジェクトに定義されたメソッドが起動され，オブジェクトの状態変化はそのメソッドにより成される．すなわち，メッセージはオブジェクトに受信されてはじめて意味をもつ．そのため，同じ名前をもつメッセージでも，違った意味をもたせることができる (polymorphism)．

クラス：クラスは共通の性質をもつオブジェクトの集合である．砕いた言い方をすれば，オブジェクトはクラスから生成され，その際に必要な情報はクラスにある．また，クラスには，そのオブジェクトが受信可能なメッセージに対応したメソッドや，インスタンス変数のシンボリックな名前，クラス変数のための辞書，などが格納されている．Smalltalk-80 では，クラスもまたオブジェクトである*．

クラスは階層構造をなしてシステム中に整理される．すなわち，あるクラスは，複数のクラスのサブクラスとして定義されている**．この際に同名のメソッドは，サブクラスのものが優先される．擬変数 self と super は，このクラス階層内のメソッドの検索順を制御する．self はメッセージの受信オブジェクトを動的に指す．すなわち，メッセージを受信したオブジェクトのスーパクラスのメソッドが起動され，その中でさらに self にメッセージが送られる場合には，self は最初にメッセージを受信したオブジェクトを指す．メソッドの検索はそのスーパクラスからではない．これに対して super は，super という文字が現れるメソッドを定義しているクラスのスーパクラスからメソッドの検索をはじめる．この self と super の機構は，アブストラクトクラスの定義を可能にし，トップダウンなプログラミングが可能になる．

以下本章では，Smalltalk-80 のユーザインタフェースについて述べた後，プログラムの作成環境，クラス検索やメソッド検索を中心とする，プログラム理解の手助けになる機能，およびデバッグ支援機能に関して述べることにする．

3.1 ユーザインタフェース

Smalltalk-80 の代表的なスクリーンイメージを図-1 に示す．階層構造をもつポップアップメニュー*** によるオーバラッピングウィンドウシステムをユーザに提供している．ここでは，マウスは必要不可欠なものとして設計されている．キーボードがなくても Smalltalk-80 の環境を使うことが可能であり，あまつさえプログラムの作成をも可能にしている．

これらのユーザインタフェースはすべて Smalltalk-80 で記述され，ユーザに公開されている．したがって，ユーザはこれらのクラスを利用することにより，既存の環境と同じインタフェースをもつアプリケーションを容易に構築することができる．

Smalltalk-80 のユーザインタフェースは Model, View, Controller という3種類のオブジェクトの三位一体モデル (MVC モデル) を基本に構築されている．すなわち，一つのウィンドウはこれら3種類のオブジェクトの協調動作により機能する．ここで，モデ

---

* したがって，クラスを生成するクラスが必要になるが，Smalltalk-80 ではこのためにメタクラスを導入している．この議論に関しては，ブルーブック[1]を参照されたい．

** ただし，現在のインプリメンテーションでは，歴史的理由により，システム関係のクラスには，マルチプルインヘリタンスの機能は使われていない．

*** 標準システムのメニューは階層構造をもっていないが，システムとしては階層構造メニューのためのクラスを用意しているので，簡単に既存のメニューを階層構造メニューに変更できる．

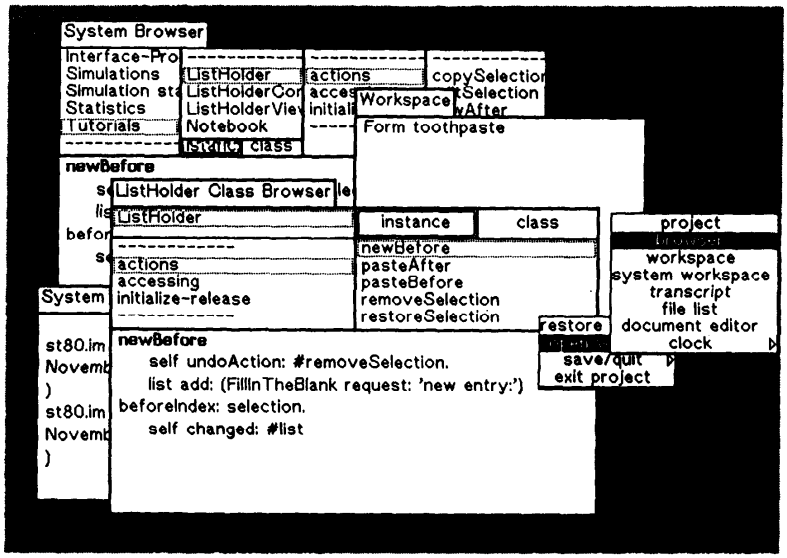

図-1 Smalltalk-80 の代表的なスクリーンイメージ

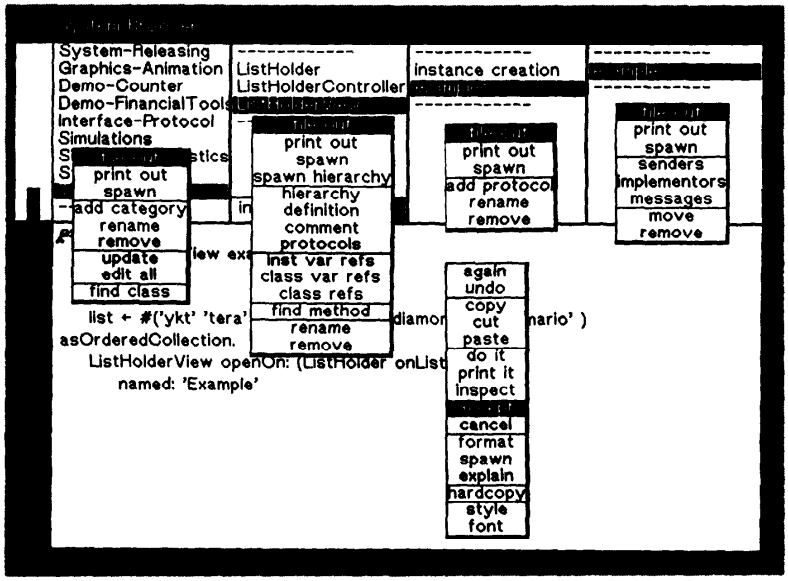

図-2 ブラウザのスクリーンイメージ

ルはアプリケーションプログラムであり，ビューはアプリケーションの状態を表示しているウィンドウであり，コントローラはイベント処理を行う．この機能分担は，アプリケーションプログラムをユーザインタフェース独立にし，ユーザの望む形でのインタフェースを容易に実現することを可能にする．

3.2 プログラムの作成

プログラムの作成はすべてブラウザを介して行われる（SystemBrowser のスクリーンイメージを図-2に示す）．ブラウザには，システム内に存在するすべてのクラスが，カテゴリごとに分類されて表示されている．ユーザはそれらの項目を，マウスでクリックし

ていくことにより，クラスの定義をみることができる．Smalltalk-80 ブラウザのもつ機能を列挙すると以下のようになる．

- 新しいクラスの定義と既存クラスの再定義，およびメソッド単位での編集，
- プログラムの外部システムファイルへの出力，
- カテゴリやプロトコルの管理，
- クラス名によるクラスの検索，
- ユーザが指示したクラスを利用しているメソッド（そのクラスにメッセージを送る式を含むメソッド）の検索，
- インスタンス変数やクラス変数を使っているクラス内のメソッド（サブクラスを含む）の検索，
- ユーザが指示したメソッドを利用しているメソッドの検索や，そのメソッド（同名のメソッドすべてを含む）定義の検索，
- メソッド内で利用している他のオブジェクトのメソッド定義の検索，
- ユーザが指示したクラスとそのスーパクラスからなるブラウザの生成，等々．

これらの検索機能によって得られる情報は，マウスの操作によって新たなブラウザが開かれて編集が可能である．

ブラウザでの編集の単位はメソッドである．メソッドの定義においては変数名の定義を特に最初にする必要がない．メソッドをコンパイルする段階で，必要に応じて未定義の変数名をどこに定義するかをシステムが聞いてくるからである．

エディタはモードレスエディタになっており，ユーザは今自分がどのモードにいるかを気にする必要がなく，プログラミングに専念できる．このエディタは，通常の文字列操作の機能のほかに，差分コンパイルの機能や，使用単語の説明機能，プリティプリントの機能，フォント変更の機能などをもっている．

3.3 プログラム理解の手助けになる機能

Smalltalk-80 のブラウザは，豊富にある既存のクラスの中から自分が必要とする機能をもつクラスを効率よく検索することを支援するための機能として，以下のものを備えている．

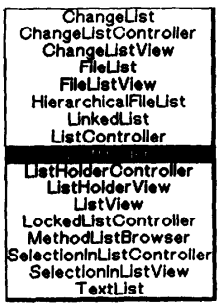

図-3 クラス検索機能

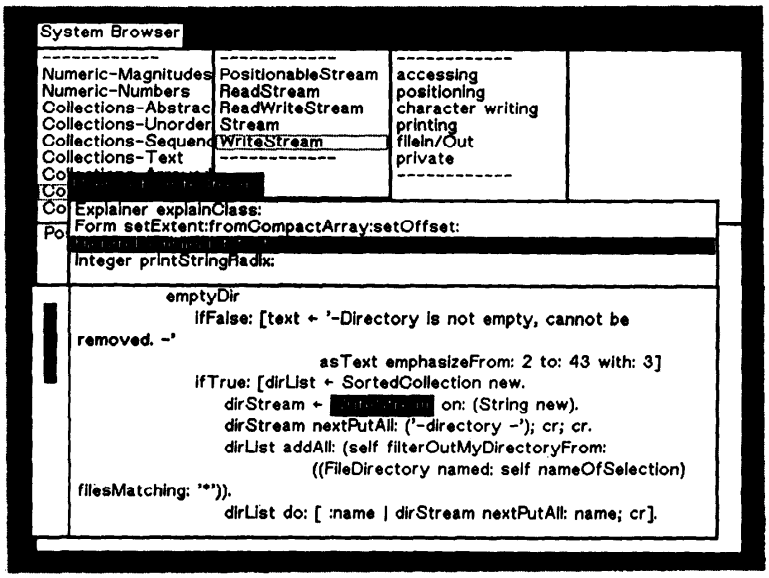

図-4 クラス利用者の表示

クラス検索機能：クラスの検索には，基本的にクラス名の入力を必要とする．しかし，完全名を入力する必要はなく，ワイルドカードの使用を許している．たとえば，「*List*」の入力により，図-3 のようなメニューが現れるので，望むクラス名をマウスでクリックすることにより，そのクラス定義がみられる．

また，そのクラスがシステム中のどのクラスのどのメソッドで参照されているかを検索する機能も備わっている．図-4 は，WriteStream クラスがシステム中のどのクラスのどのメソッドで参照されているかを検索した結果を表示している．

メソッド検索機能：メソッド検索機能には，現在参照中のメソッドを利用しているメソッドや，そのメソッドと同名のメソッドを検索する機能，そのメソッド内で利用しているメソッドの他のオブジェクトでの定義を検索する機能がある．図-5 は WriteStream クラスの print: メソッドと同名のメソッドの InstructionPrinter クラスでの定義を参照している例である．

変数の検索機能：インスタンス変数とクラス変数に関しては，その利用者を検索する機能を備える．インスタンス変数の検索は，そのクラスとそのクラスのスーパクラス中のメソッドが検索対象になる．クラス変数に関しては，システム内の全メソッドが検索対象になる．図-6 は WaitingSimulationObject クラスのスーパクラスである DelayedEvent クラスに定義されている resumptionCondition 変数を利用しているメソッドを検索している例である．

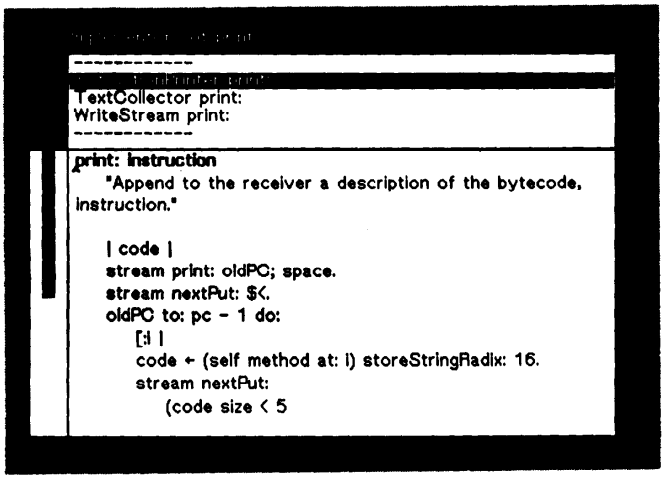

図-5 同名メソッドの検索

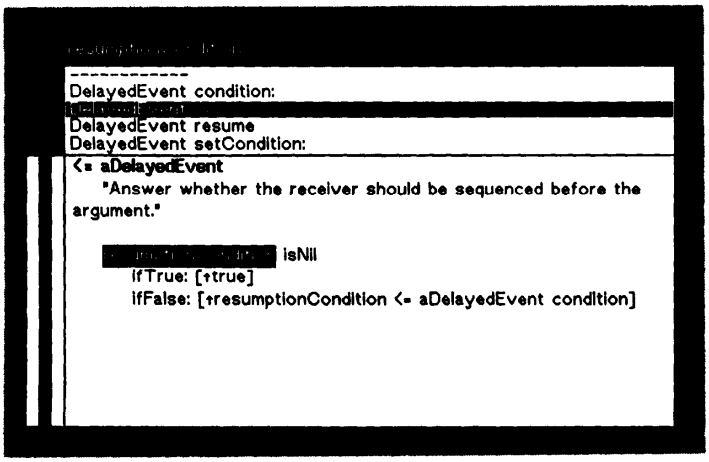

図-6 インスタンス変数の検索

説明機能：説明機能には，クラス定義時に付加されたコメントを表示する機能のほかに，セレクトした単語に対する説明機能を備える．図-7 は，values に対する説明を求めている例である．

編集機能：エディタはモードレスエディタである．機能としては，文字列のカットアンドペーストの機能，操作の取り消し機能，同じ操作の再試行機能，などを備える．

3.4 デバッグ支援

Smalltalk-80 では実行中に誤りが生じた時点で，ノーティファイアが呼び出され（図-8），ユーザに対して，処理の続行か，デバッガの呼び出しか，を聞いてくる．ノーティファイアは，呼び出しのためのメッセージ式（たとえば，「self halt.」）をプログラム中に挿入することにより，明示的に呼び出すことも可能である．

デバッガ：Smalltalk-80 では，メッセージの受信によりメソッドが呼び出されるごとにコンテキス

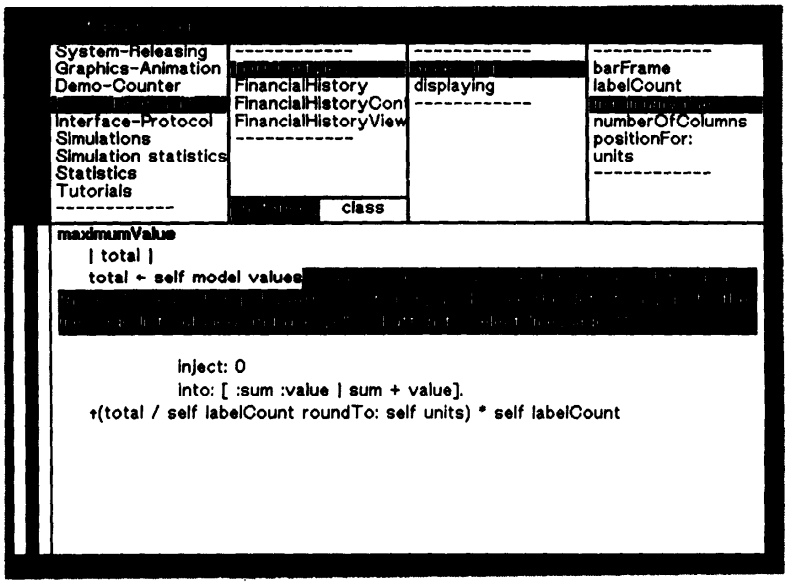

図-7 単語の説明機能の例

トが生成されて，メソッドの実行に必要な情報（メッセージの送信者，命令ポインタ，評価用スタック，など）が格納される．デバッガは，このメッセージパッシングによるコンテキストのチェインを操作することにより，デバッグ作業を進める．

Smalltalk-80 のデバッガが備える機能をまとめると以下のようになる．

- コンテキストチェインの表示，
- 実行の続行と，再試行，
- メッセージ式の評価を一つ進めること，
- 実行を次のメソッドの呼び出し直後まで進めること，
- 各変数の値の操作，等々．

なお，Smalltalk-80 のステップ実行は，前進方向のみであり，後退はできない．

図-9 にデバッガの表示例を示す．copySelections というメソッドが受信側オブジェクト内で未定義であることを示している．サブウィンドウの上段は，メッセージパッシングの過去の履歴を示しており，図では誤りの起こったメソッド中での位置が反転表示されている．中段のサブウィンドウは，上段で選んだメソッドのソースコードが表示されている．下段のサブウィンドウは，インスペクタ（後述）になっており，左側

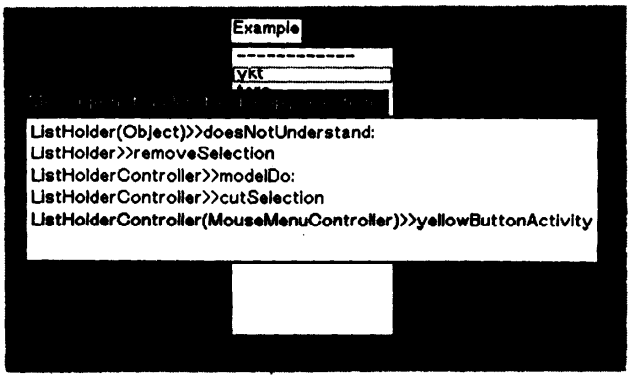

図-8 ノーティファイアの例

の二つのサブウィンドウで，インスタンス変数名とその値が，右側の二つのサブウィンドウで，一次変数とその値が表示されている．

デバッガで見つかった誤りを修正するのはデバッガで行うことができる．わざわざブラウザで行う必要はない．この場合には，中段のウィンドウで，copySelections を copySelection に修正してコンパイルするだけでよい．

インスペクタ：インスペクタはオブジェクトの内部状態を覗きみるための機能であり，inspect メッセージをオブジェクトに送ることにより起動される（**図-10**）．左側のサブウィンドウ内に，オブジェクトのシ

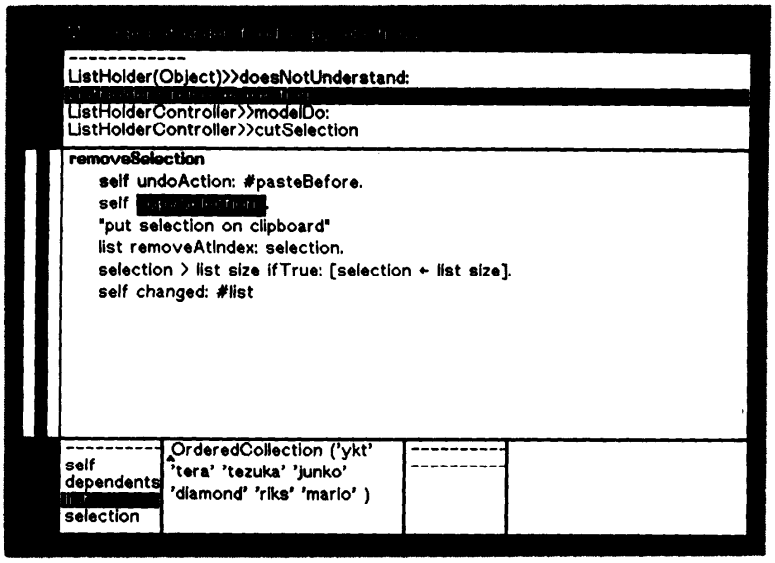

図-9 デバッガの例

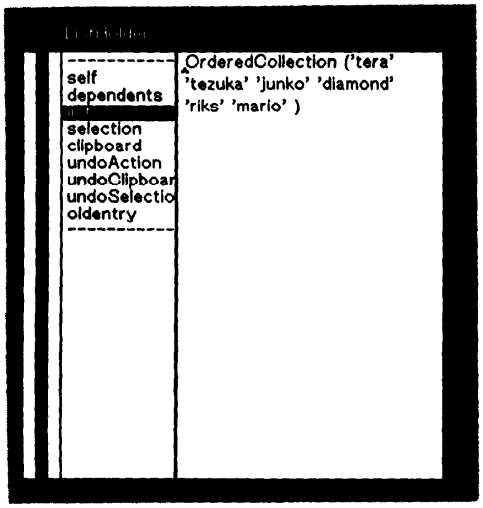

図-10 インスペクタの例

ンボリックな変数名が表示され，右側のサブウィンドウ内に，その値が表示される．右側のサブウィンドウはエディタになっており，値を書き換えて accept することにより，オブジェクトの内部状態を変更できる．

インスペクタの機能は，すべてのオブジェクトが at: や at:put: のような，自らの内部状態を見せたり，変更したりできるようなメソッドをもっていることに依存している*．したがって，オブジェクト側で，これらのメソッドを再定義することによって，インスペクタの機能を無効にすることも可能である．

ミススペルの修正機能：コンパイル時に単語に関する誤りは，新しい変数名としての登録か，ミススペルの修正を行うべきかをシステムは問い合わせてくる(図-11)．変数名の登録のときには，該当するメニュー項目をマウスでクリックすることにより，未定義の変数名を登録してくれる．ミススペルの修正の場合には，「correct it」をクリックすることにより，正しいスペルに修正してくれる．

パフォーマンスモニタ：どのメソッドの実行に全体のどのくらいの時間を費やしているかを調べる機能を備えている．スパイと呼ばれるこの機能は，たとえば，「MessageTally spyOn: [Pen penSampler]」なる式を評価すると，図-12 のような結果が得られる．

3.5 まとめ

Smalltalk-80 におけるプログラミング環境が，2. での必要事項をどの程度満たしているかをまとめてみる．

R1：システム全体を単一の言語 Smalltalk-80 で記述し，しかもユーザインタフェース部を MVC モデルに基づいて構築し，ユーザに公開している．した

* 正確には，Array のようなオブジェクトの場合には，basicAt: や basicAt:put: であり，インスタンス変数をもつオブジェクトの場合には，instVarAt: や instVarAt:put: である．

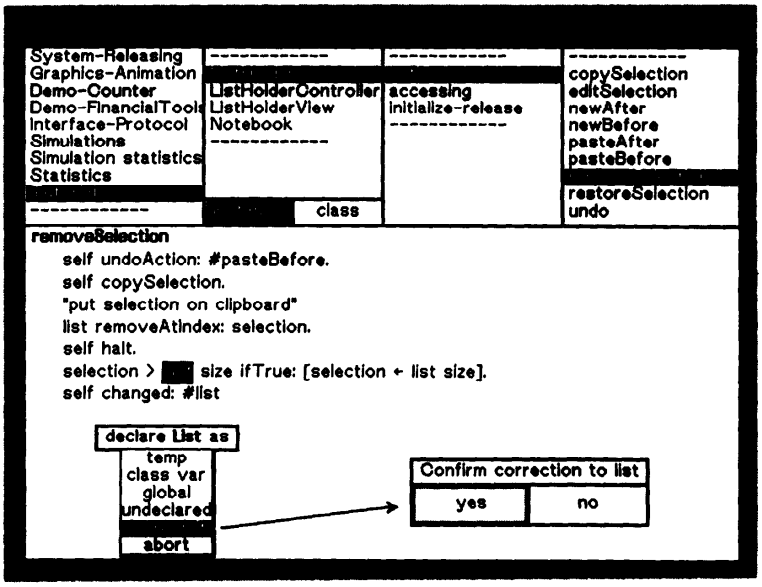

図-11 スペルチェックの例

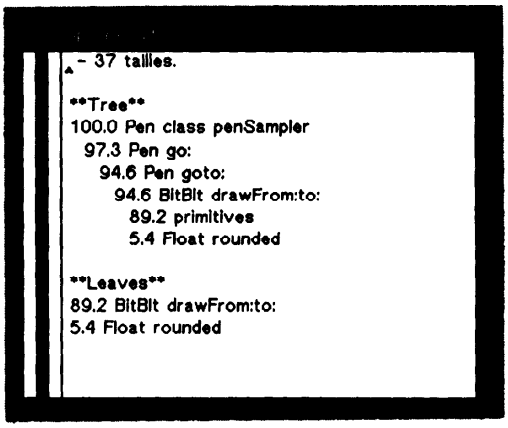

図-12 パフォーマンスモニタの例

がって，ユーザはシステムを自分の要求に適合するように自由に変更や自己拡張できる．また，デバッガで発見した誤りをデバッガ内で修正できるなど，プログラミングを支援するさまざまな機能が密に統合化されている．

R2：名前によるクラスやメソッドの検索機能を備える．しかし，Smalltalk-80 の検索機能はより抽象的な形では提供されていない．その点の改良に関しては文献 14) にみられる．

R3：メソッド単位のコンパイル機能を備える．デバッガで誤りが発見された場合にはその場で修正して，実行を続行できる．

R4：メッセージパッシングの履歴に基づくソースコードレベルでのデバッグを支援している．

4. リフレクティブプログラミング環境

リフレクティブ計算に関する研究は最近日本でも盛んになってきている[13),17),20)]．オブジェクト指向言語へのリフレクティブ計算の導入も行われている[2),7),17)]．本章では，リフレクティブ計算のデバッガへの応用に関して議論する．とくに，Smalltalk-80 のデバッガをリフレクティブ計算の観点より分析してみる．

4.1 リフレクティブ計算

リフレクティブ計算とは，自分自身の状態を自分自身で把握しながら，すなわち自分自身の計算を自分自身で解釈実行しながら，計算が進んでいくスタイルの計算モデルである．一般の計算が行われる領域をオブジェクト領域，オブジェクト領域の振舞いを定義する領域をメタオブジェクト領域と呼んでいる．Maes[7)] によれば，リフレクティブ計算には次の三つの要件が必要である，としている．

（1） システムの自己記述が可能なこと．すなわち，システム自身がシステム自身の状態を定義することができ，しかも変化させることができる．

（2） リフレクティブ計算をプログラムできること．すなわち，オブジェクト領域の計算を定義するメタオブジェクト領域をユーザがプログラムできる．

（3） オブジェクトレベルとメタレベルとの間の因果的結合．オブジェクト領域の計算がメタオブジェクト領域の計算に影響を与えると同時に，その逆も成り立つ．

たとえば，3-KRS[7] や ObjVlisp[2]，ABCL/R[17] の場合では，オブジェクト領域のオブジェクトの状態やそれを操作するメソッドは，メタオブジェクト領域のオブジェクト（メタオブジェクト）の状態（データ）として表現されている．3-KRS や ABCL/R ではメタオブジェクトは各オブジェクトごとに定義されている．また 3-KRS や ObjVlisp ではメタ階層とクラス階層の定義を独立して定義できる．メタオブジェクトがオブジェクトの内部表現を定義している点やメソッドを保持している点は，Smalltalk-80 のクラスがオブジェクトのそれらの情報を保持している点と対応するようにみえる．しかし，前者らのメタオブジェクトとオブジェクトとの間には因果的結合関係があり，オブジェクトの計算をメタオブジェクトが解釈実行していると定義している点は，Smalltalk-80 における場合とは大いに異なる．

4.2 デバッグ支援とリフレクティブ計算

4.2.1 リフレクティブ計算のデバッガへの応用

オブジェクトの定義によれば，オブジェクトの内部の状態を変えることができるのは，オブジェクト内に定義されたメソッドのみである．しかし，プログラミング支援環境に要求される事項のうちの一つは，「オープンシステム」であるということである．すなわち，デバッガもオブジェクト領域のアプリケーションの一つでなければならないということである．ここで，オブジェクトの定義との矛盾が生じる．デバッガがオブジェクト領域のアプリケーションであると，同じ領域のオブジェクトの内部状態を見たり，変更したりすることができなくなる．

リフレクティブ計算はこの矛盾を解決してくれる．先に述べたように，オブジェクトの定義を与えるのは，メタオブジェクトである．したがって，メタオブジェクトとしてデバッグの機能が用意されていれば，オブジェクトの振舞いの変更や内部状態の変更は問題なく行える．しかも，このデバッグの機能はアプリケーションのプログラムとは独立して作成することが可能であるため，完成後のアプリケーションプログラムの保守のしやすさも向上する．

4.2.2 Smalltalk-80 デバッガのリフレクティブ計算による分析

Smalltalk-80 においてはリフレクティブ計算におけるオブジェクトとメタオブジェクトの関係は，仮想イメージと仮想機械の関係に相当する．メソッドの実行は仮想機械に定義された命令に従い，オブジェクトの内部表現は仮想機械によって定義され操作される．すなわち，メタオブジェクト領域にある仮想機械によって，オブジェクト領域の仮想イメージが定義されているわけである．しかし，この両者の間には因果的結合関係はなく，仮想機械をユーザがプログラムできるわけでもない．

Smalltalk-80 では仮想機械の仕様を Smalltalk-80 で記述している．Smalltalk-80 による仮想機械のシミュレータの部分は，メッセージ受信ごとに生成されるコンテキストをオブジェクトとすることによって，実現している☆．InstructionStream クラスは仮想機械シミュレータのメインルーチンに該当するメソッドが定義されている．そのサブクラスの ContextPart クラスには，バイトコード☆☆をシミュレートするためのメソッドが定義されている．これらのサブクラスとして MethodContext クラスや BlockContext クラスが定義され，それらのインスタンスがメソッドの実行状態を保持している．デバッガにおけるステップ実行の機能は，このコンテキストオブジェクトに定義された仮想機械シミュレータを用いて実現されている．

シミュレーションされるべきアプリケーションがオブジェクト領域のオブジェクトによって定義されているとすれば，仮想機械シミュレータはメタオブジェクト領域のオブジェクトによって実現されている．これらはすべて Smalltalk-80 を用いて記述されているので，自らの状態を自らが(制限されてはいるが)定義することができ変更することもできるし(条件1)，ユーザはリフレクティブ計算をプログラムできる（条件2）．さらに，仮想シミュレータにおける状態の変化は，その上で実行されているオブジェクトの実行に影響を与えることができるし，その逆も可能である（条件3）．Smalltalk-80 の場合にはオブジェクト領域の

☆ 最近の Smalltalk-80 実装では，コンテキストを仮想機械が実装されているハードウェアに適した形で保存しているが，シミュレートされるオブジェクトのメソッドの実行により生成されるコンテキストはオブジェクトとして表現されている．

☆☆ Smalltalk-80 のメソッドはバイトコードと呼ばれる1バイトの仮想機械の命令列にコンパイルされる．

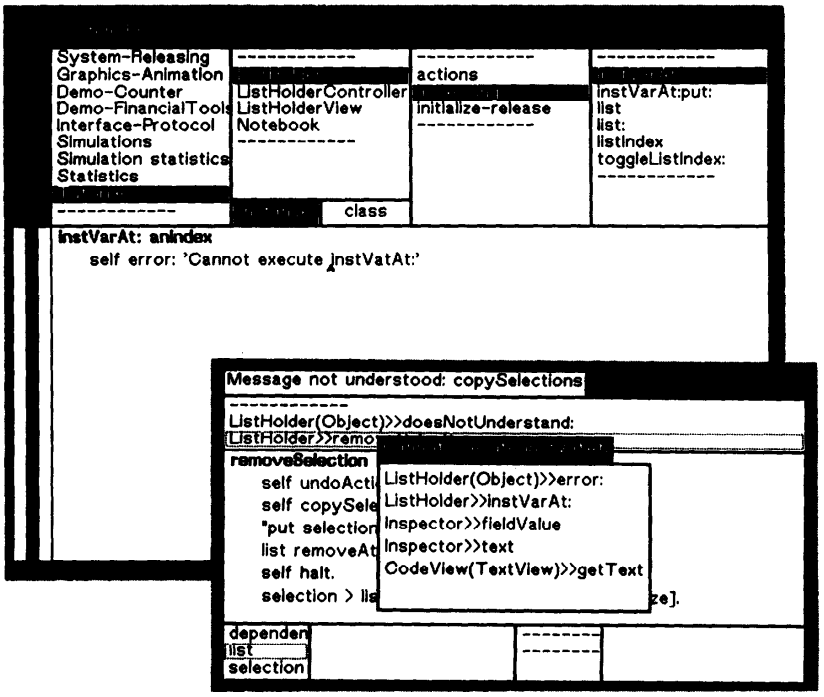

図-13 instVarAt: の再定義によるデバッガの機能の無効化

アプリケーションもメタオブジェクト領域の仮想機械シミュレータも，実装上は一つのフラットな同一の空間上に Smalltalk-80 のオブジェクトとして実現されている．そのため，メタオブジェクトがオブジェクトの内部状態に変更を加えたい場合でも，必ずそのオブジェクトに対してメッセージを送る必要がある．Smalltalk-80 では，すべてのオブジェクトに自分の内部状態をみせるメソッド (at:)，および内部状態を変更するメソッド (at:put:) をもたせることにより，リフレクティブ計算を実現している．

ここで，Smalltalk-80 におけるデバッガについてみてみよう．Smalltalk-80 ではデバッガも Smalltalk-80 で記述されている．ステップ実行の機能は，先に述べた仮想機械シミュレータによって実現されている．また，インスペクタの機能は各オブジェクトに at: や at:put: のメソッドをもたせることによって実現している．したがって，これらのメソッドをサブクラスで再定義して無効にすることによって，デバッガは機能しなくなる．図-13 ではサブクラスで instVarAt: メソッドを再定義して無効にしているが，そのためにデバッガはそのオブジェクトのインスペクトや，ステップ実行が不可能になっている．

Smalltalk-80 におけるデバッガはリフレクティブ計算の一つの応用として実現されているとみることができる．しかし，その実装は内部状態の問合せに対して返答するメソッドを各オブジェクトに定義☆することによって実現している．これらのメソッドは見方によってはリフレクティブ計算を実現するものであるので，構文上はオブジェクトのメソッド記述とは分離されるべきである．さらに，単一ユーザ環境の Smalltalk-80 であるから許されるこの機能は，複数ユーザ環境，あるいは分散環境においては，問題を残している．

5. ま と め

オブジェクト指向言語のプログラミング環境としては Smalltalk-80 のそれが最初である．その後，Lisp マシン上のオブジェクト指向システム (たとえば Flavors など) や Cedar[12]，HyperCard[5]，Eiffel などのオブジェクト指向システムが生まれてきている．以下に Cedar と Eiffel[8),9)] の概要を述べる．

☆ 実際には最上位のスーパクラスの Object に定義されている．

Cedar：プログラム開発のための環境である Cedar は，ガーベジコレクションや遅延バインディング，データタイプの拡張を行った Mesa の拡張言語 Mesa/Cedar で記述されている．Cedar はオープンシステムとして設計されており，アプリケーションはオペレーティングシステムがもつ良く設計されたモジュールを自由に利用できる．プログラミングを支援するツールには，Imager（デバイス独立なグラフィックパッケージ），Viewers（ウィンドウマネジャ），Tioga（ドキュメント作成のためのテキストエディタ），Compiler/Binder，ソースレベルデバッガ，バージョン管理ツール，などが用意されている．

Eiffel：強い型付けをもつオブジェクト指向言語であり，Eiffel プログラムは中間言語としてのCのプログラムに変換されてから，ネイティブコードにコンパイルされる．また，オプションの指定により，独立したCプログラムを生成することもできる．現在，Eiffel は UNIX*上に構築されており，一つのクラスは一つのファイルに対応する．差分コンパイルは，クラス同士の相互依存関係に基づいて必要なクラスを選びだしコンパイルする．既存のクラスの検索機能や表示機能には，クラスアブストラクタ（クラスのインタフェースに関するドキュメントを表示），クラスフラッタ（クラス階層をコラプスした形で表示），good（システムのクラス階層をグラフィカルに検索するツール）などが用意されている．また，プログラムのデバッグは言語のもつアサーションの機能を用いた対話型のデバッガを用意している．

Smalltalk-80 を実際のプログラム開発に使おうとしたときに，その実行にすべての環境が必要である，ということが問題点として指摘される．どんなに小さなアプリケーションを実行するにも，実際に必要でない機能をも含めて最低1メガバイトにも及ぶ仮想イメージが必要になる．そのために Smalltalk-80 はプロトタイピングに用い，完成したソフトウェアを別の言語を用いて記述し直す場合も多い．Modular Smalltalk[18] の手法はこの問題点を改善するものである．Modular Smalltalk では Program と Module と呼ばれるプログラム単位を導入し，名前の import, export をユーザが記述することによって，イメージと独立なネイティブコードにコンパイルされたアプリケーションを生成することができる．

本稿ではプロジェクトを組んでソフト開発をする場合に考慮しなければならないバージョン管理に関する問題については紙面の都合上述べなかった．機会を改めて述べたいと思う．Smalltalk-80 の開発者たちは開発当初からこの問題に対しては議論が成されており，いくつかのツールも作られている[11]．

本稿で述べたデバッグ支援機能は，逐次オブジェクト指向言語におけるものであるが，並行オブジェクト指向言語におけるデバッグ支援の問題も今後は重要になってくる．並行実行環境においては，非決定性の要因が多く入ってくるため，そのデバッグ支援も複雑になる．近年並行オブジェクト指向計算に関する研究も盛んになり，それに基づくプログラミング言語も開発されているが[6),19),21]，そのデバッグ環境についての研究は，まだ緒についたばかりである[16]．

本稿では，オブジェクト指向言語における，とくに Smalltalk-80 におけるプログラミング環境に関して，プログラムの支援環境と，デバッグ支援を中心に述べた．Smalltalk-80 においては，言語そのものがプログラム作成のためのツール群と融合し，一つの統合化されたプログラミングシステムを構成している．探索型プログラミングの名前が示すとおり，ユーザはオブジェクトという世界の中をゲームをやるような感覚で，ソフトウェアの開発をしていくことができる．プログラミング環境を考える際には，実にさまざまの要素について考慮しなければならない．本稿では触れなかったが，人間工学的なアプローチも必要であろう．たとえば，ビットマップディスプレイやマウスの是非なども考慮しなければならない．本稿が，今後のより使いよいプログラミング環境開発の一助になれば幸いである．

★ UNIX は AT&T ベル研究所が開発し，AT&T がライセンスしているオペレーティングシステムである．

参考文献

1) Agha, G. A.: Actors: A Model of Concurrent Computation In Distributed Systems, Technical Report 844, MIT (1985).
2) Cointe, P.: Metaclasses are First Class: the ObjVlisp Model, Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1987, ACM, pp. 156-167 (1987).
3) Goldberg, A. and Robson, D.: Smalltalk-80: The Language and Its Implementation, Addison-Wesley (1983).
4) Goldberg, A.: The Influence of an Object-Oriented Language on the Programming Environment, Interactive Programming Environments, McGraw-Hill, pp. 141-174 (1986).
5) Goodman, D.: The Complete HyperCard

Handbook Bantam Books (1987).
6) 石川 裕，所真理雄：並行オブジェクト指向言語 Orient 84/K コンピュータソフトウェア，Vol. 3, No. 3, pp. 24-42 (1986).
7) Maes, P.: Computational Reflection, Technical Report TR-87-2, VUB AI-LAB (1987).
8) Meyer, B.: Object-oriented Software Construction, Prentice-Hall (1988).
9) Meyer, B.: The Eiffel Environment, UNIX Review, August (1988).
10) Moon, D. A.: Object-Oriented Programming with Flavors, Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1986, ACM, pp. 1-8 (1986).
11) Putz, S.: Managing the Evolution of Smalltalk-80 Systems, Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, pp. 273-286 (1983).
12) Swinehart, D. C., Zellweger, P. T., Beach, R. J. and Hagmann, R. B.: A Structural View of the Cedar Programming Environment, ACM Trans. Prog. Lang. Syst., Vol. 8, No. 4, pp. 419-490 (1986).
13) 田中二郎，太田祐紀子，的野文夫，神田陽治：GHC による仮想ハードウェアの構築とリフレクト機構，日本ソフトウェア科学会第4回大会論文集，pp. 455-458 (1987).
14) 垂水浩幸，岡村和男，阿草清滋，大野 豊：クラス再利用支援のためのオブジェクトモデル，コンピュータソフトウェア，Vol. 3, No. 3, pp. 61-70 (1986).
15) 福永光一，沼尾雅之：オブジェクト指向言語のプログラミング環境，情報処理，Vol. 29, No. 4, pp. 334-343 (1988).
16) 本田康晃，柴山悦哉，米澤明憲：並列オブジェクト指向言語のデバッグ方式に関する考察．日本ソフトウェア科学会第3回大会論文集，pp. 165-168 (1986).
17) Watanabe, T. and Yonezawa, A.: Reflection in an Object-Oriented Concurrent Language, Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1988, ACM, pp. 306-315 (1988).
18) Wirfs-Brock, A. and Wilkerson, B.: An Overview of Modular Smalltalk, Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1988, ACM, pp. 123-134 (1988).
19) Yokote, Y. and Tokoro, M.: Experience and Evolution of Concurrent Smalltalk, Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1987, ACM, pp. 406-415 (1987).
20) 横手靖彦，寺岡文男，所真理雄：オブジェクト指向分散オペレーティングシステム Muse におけるオブジェクト管理機構，日本ソフトウェア科学会第5回大会論文集，pp. 193-196 (1988).
21) 米澤明憲，柴山悦哉，Briot, J.-P.，本田康晃，高田敏弘：オブジェクト指向に基づく並列情報処理モデル ABCM/1 とその記述言語 ABCL/1，コンピュータソフトウェア，Vol. 3, No. 3, pp. 9-23 (1986).

(昭和63年12月1日受付)