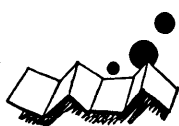


解説

オブジェクト指向言語の プログラミング環境†



福永光一†† 沼尾雅之††

"To support program development, help takes the form of methods for finding information about existing functionality, methods for accessing that functionality, methods for describing new programs, and methods for discovering and fixing any faults in those programs.

.....In particular, the system should be able to help the user find out what went right, find out what went wrong, find out what can be done next, and find out something about any system component."

—Adele Goldberg¹⁾—

1. はじめに

一般にプログラミング環境の良し悪しを統一的な枠組の中で論ずることは難しい。それは、プログラミング環境には、さまざまな要因が、複雑な相関をなしながら影響を与えるからである。とくに、オブジェクト指向のような新しい言語のプログラミング環境では、種々のアイデアが模索段階にあり、それらを見通しよく整理する座標軸が見出しにくいというのが実情である。そこで、本解説では、著者らの体験をもとに、オブジェクト指向の言語上の特徴とプログラミング環境との関係に対するいくつかの視点を与え、それに関連し、実在する先進的プログラミング環境の特徴的な機能の紹介を行う。それらを結び付け個別の応用の場での意味付けをする作業は、読者に委ねたい。

2. プログラミング・スタイルの変遷と その要因

まず、この章では、筆者の経験をもとにプログラミング・スタイルとプログラミング環境が過去どのように影響を与えてきたか、そのあらましを検討してみよう。次章以降で、それをもとに、オブジェクト指向のプログラミング環境を議論することにする。

著者の一人が初めてプログラミングを習ったのは確

か1970年のことである。習ったプログラミング言語はもちろん FORTRAN で、最初はコーディング・シートにプログラムを書き込むと、だれかがパンチしてくれて、1週間後に実行結果のリストとともにカードの束が返ってきた。以来学生、ソフトウェア開発職、研究職と過ごしてきた間に、環境の変化とともに種々のプログラミング・スタイルをその都度選択するという経験をしてきた。使用言語は、FORTRAN, PL/I, LISP, PROLOG, 自作のオブジェクト指向言語²⁾というふうに変わり、プログラムの処理もクローズド・バッチ、オープン・バッチ、大勢が端末を共有する TSS (最初はテレタイプ、次にディスプレイ端末)、自分専用のディスプレイ端末を用いての TSS と変遷した。最近では、筆者のもう1人は LISP や Smalltalk-80 専用の個人用ワークステーションで仕事をしている³⁾。

筆者らのこれらの経験の中で、プログラミング・スタイルの選択に影響を与える要因として強く記憶に残っているものに、次のようなものがある。

● PL/I でソフトウェア開発をしているときに、テレタイプを端末とした TSS サービスが使えるようになった。しかし、プログラム・リストの打出しに時間がかかるのでデバッグには不向きで、ちょっと試しただけでもとのバッチ方式に戻ってしまった。それ以後、ディスプレイ端末上での応答の早い画面エディタが使えるようになるまで TSS にはなんら魅力を感じなかった。

● 研究職として実験的に小さなプログラムを数多く書くようになると、これまでのような綿密な設計を経ないので、プログラムの挙動の予測が十分にできなくなった。そのため、デバッグ用のツールにお世話になったり、プログラムをより小さな構成要素に分けて、それらごとに実行を繰り返す、というようなことを頻繁にするようになり、文字どおり端末に向かってプログラミングをするようになった。

● LISP や Smalltalk-80 専用のワークステーションは、高度なプログラミング環境が用意されているた

† Object Oriented Language Programming Environment by Koichi FUKUNAGA and Masayuki NUMAO (Tokyo Research Laboratory, IBM Japan, Ltd.).

†† 日本アイ・ビー・エム(株)東京基礎研究所

め、プログラムの生産性が著しく向上する。その結果、プログラミング環境に対する依存性が高まり、簡単なメモでも紙ではなく画面上に書き込むので、それこそ一日中計算機の前に座って作業をするようになる。悪くいえば、そこから抜け出せなくなるわけで、逆にプログラミング環境のちょっとしたできの良さ悪しがプログラムの生産性やプログラマの精神衛生に大きな影響を与えることになる。

これらの個人的体験の流れには、二つの大きな軸がある。一つは、計算機をコーディング用の道具として使っているか、設計用に使っているかである。もう一つは時代の差による計算機環境の差である。これら二つの要因が互いに影響を及ぼし合っており、その中で、プログラマはプログラミング中の思考が助けられる、あるいは中断が一番少なくて済む行動様式を選んでいる。このような見地から、上の体験を一般化すると、プログラミング・スタイルについて次のような結論が引き出せそうである。

(1) 対象プログラムの性質

仕様の定まったプログラムを設計手順を踏んで作成している場合には、紙の上の方がむしろ考えやすい。デバッグの手順もあらかじめ考えつくので、エディタ以外のツールの必要性はあまり感じない。これとは反対に、アイデアを固めるための実験をしつつプログラムを作っていく探索型(exploratory)プログラミング⁴⁾の場合には、プログラムの挙動の予測が困難なため、対話型の充実したプログラム環境がほしい。

(2) 使用できる計算機設備

計算機資源へのアクセス機会(端末の空き、応答時間など)が不十分な場合には、どんなツールにも魅力を感じない。むしろ、紙の上でじっくり設計を考える方がよい。(探索型プログラミングは、ほぼ不可能である。)計算機環境がよくなつてはじめて、ツールは意義をもつ。また計算機環境がよくなつて、ツールに依存するようになると、そのできに対する要求も増大する。

(3) プログラムの規模

これは(1)とかなり相関が強いが、一気に大きなプログラムを書く場合(従来型プログラミング)は、全体を見回したいので、ディスプレイ上の工夫がないと紙の方がよくなる。逆に、小さなプログラムから積み上げていく場合(探索型プログラミング)には、結果を迅速に確認しながら作業を進められる対話型環境のほうが絶対によい。

(4) ツールとプログラム言語の相関

高級な機能をもつ言語には、それなりのツールが必要である。逆に、ツールが貧弱なら、簡単な言語のほうがよいことが多い。特に、探索型プログラミングに向いているといわれる言語は、その特徴(遅延束縛、バックトラッキングなど)に見合ったツールが用意されていないと、その効果を発揮しにくい。

3. オブジェクト指向のプログラミング・スタイルと支援機能

前章でプログラミングには手順を踏んで設計を進めていくやり方と、探索型のものとの2とおりがあることを示し、それぞれでプログラミング環境に対する要求の度合いが異なることを述べた。では、オブジェクト指向言語は、どちらのプログラミング・スタイルにより適しているであろうか。この議論を明確にするために、まずオブジェクト指向言語の二大特徴であるメッセージ通信と、クラスとその継承機能が、プログラミング作業中の思考をいかに助けるかを検討してみると、次のようになる。

(1) メッセージ通信

- 通信し合うオブジェクトというメタフォアは受け入れやすい。しかも、メッセージは通信手段としてはバンド幅が狭いので、自然に小さなオブジェクトを数多く集めたプログラムを作るようになり、モジュール性を高める。

- メッセージの解釈が受信側でなされるため、送信側は相手を気にする必要がない。送信先を実行時に決められるので、不特定の相手に(相手の種類が異なっても)共通のメッセージを送るような抽象度の高いプログラムが心理的抵抗なしに容易に書ける(polymorphism, late binding)。しかも、プログラムのテストの際は、当面のシナリオに必要な受信オブジェクトのみを用意すれば、全体のプログラムが完成していなくても、送信オブジェクトが意図どおりの働きをするかどうか、大略調べられる。

(2) クラスとその継承機能

- いくつかのクラスに共通な性質を、上位クラスにまとめて書き、それを継承することにより、プログラミングの手間が大幅に軽減できる。それのみならず、抽象性の高い概念を上位クラスとして明示的に表現できるから、それをいろいろな文脈の中で再利用できることになる。再利用に際して文脈ごとの特殊化(サブクラスの導入あるいはクラスのさらなる一般化)が必

要となることもあるが、これらの作業の手間も、再利用できるクラスがない場合に比べて、はるかに少なく済む。これを差分プログラミングという。

●通常のオブジェクト指向言語では、上位クラス内で参照されたメソッドの探索は、特に指定しないかぎり、再び最下位のクラスから始められる。この機能を使うと、上位クラスで抽象的な操作手順を記述し、具体的操作方法は個々の下位クラスに任せるといったトップ・ダウンなプログラミングが行える。しかも、この際下位クラスとして当面必要なものだけを用意してテスト・ランをしてもならん支障はない。(84年の第5世代コンピュータ国際会議の招待講演で、D. G. Bobrowはこのようにして定義される上位クラスのメソッドを abstraction methods と呼び、“many times its importance is overlooked.”と言った。)

以上の議論から、オブジェクト指向言語は抽象的レベルでのプログラミングを助け、しかも受信オブジェクトや下位クラスを部分的に用意するだけで、一応のテストが行えることが分かる。また既存のクラスを借りてきて、その一部を修正することにより、簡単にプログラムが作れることも多い。この意味でオブジェクト指向言語は探索型プログラミングに向いているといえよう⁴⁾。

逆に、オブジェクト指向言語を使って一気に大きなプログラムを書くのは、あまり得策のようにとは思えない。というのは、オブジェクト指向で書かれたプログラムの分かりやすさは、オブジェクトの分割やクラス階層の整理にかかっており、少なくとも筆者らの経験^{3), 5)}では、これらの最終的なあるべき姿を事前に予測するのは、ひどく難しいからである。一度設計を済ませたととしても、実際にはプログラムを書きながら、何度もそれを練り直すことになる。したがって、設計を済ませてから一気にプログラムを書くという態度を捨て、最初から実験を繰り返しつつプログラムを徐々に組み立てていく方法をとったほうが、近道のように思える。

このようなオブジェクト指向言語の特徴を生かそうとすれば、当然それなりのプログラミング環境が必要となる。まず前章で述べたように、探索型プログラミングのためには、次章で示されるような充実した計算機設備の下での対話的環境が必要である。(もちろん、計算機環境がよくない状況下での従来型プログラミングの設計時にも、オブジェクト指向的考え方をを用いてモジュール性を高めることは可能である。しかし、ク

ラス階層の練り直しなどを何度も繰り返すうちに突如問題の構造がみえてくる、というような有難さはとても味わえないであろう。)

さらに、その環境はオブジェクト指向に特有な概念である、メッセージ通信やクラス階層の情報の管理に工夫のあるものでなければならない。このためには、次のような支援機能が必要となろう。

●polymorphic なメッセージ通信に関しては、実行時に行き先が決まることを考慮した、メッセージの流れがよく分かる強力なデバッガ(トレース)や、テスト時に受信オブジェクトがなくても、それを端末からの入力で代替できるようなテスト・ベッド。また、プログラム全体のメッセージ・トラフィックを効率的に表示することにより、メッセージの分割・統合にヒントを与える手段⁶⁾。

●クラスの再利用・差分プログラミングに関しては、既存クラスの中で自分の目的に合ったものを見つけやすくするための、クラス階層全体の構造が利用者の頭に入りやすくする工夫、“類似の”クラスを迅速に検索する機能(強力なブラウザ)。さらに、事前にクラスの分割・統合をとおして、各クラスが表現する概念をすっきりとした分かりやすいものにする工夫^{7), 8)}。

●また abstraction methods を利用したトップ・ダウン・プログラミングを行おうとする場合には、言語上は上位クラスからは下にどのようなサブクラスがあるかが分からないので、abstraction methods が下位クラスのどこで実現されているかが常時分かるようにし、そのインタフェースが合っているか否かを自動的にチェックする機構。

4. 各種オブジェクト指向言語のプログラミング環境

本章では、筆者らが実際に使用した経験に基づいて各種オブジェクト指向言語のプログラミング環境の機能について比較検討を行う。ここで取り上げる言語システムは Smalltalk-80, Flavors および KEE である。いずれも専用ワークステーション上で実現されていて、ビットマップ・ディスプレイやマウスを駆使した統合的プログラミング環境をもつシステムとしては代表的なものである。

本章ではまずこの三つのオブジェクト指向言語システムの特徴を述べ、次にこれらのシステムが、3. で述べたようなオブジェクト指向型言語用のプログラミン

グ支援機能を、いかに実現しているかについて解説する。

4.1 代表的なオブジェクト指向言語システムとその環境の特徴

(1) Smalltalk-80^{9),10)}

この言語は Xerox のパロアルト研究所の Alan Kay が提唱した究極のパーソナル・コンピュータ“Dynabook”の思想をもとに研究が始められたものであり、子供にでも簡単に使える言語を目指して開発が進められた。開発の当初からビットマップ・ディスプレイを用いた高度な対話的インタフェースを最大限に活用することが目標であった。したがって、この言語にとってはマルチ・ウィンドウやマウスによるプログラミング環境は、単にプログラミングを助ける道具という位置付けではなく、言語の一部になっていると考えられる。実際、マウスの操作だけでかなりのプログラミングができるのに対して、キーボードだけでは、プログラムの実行すらできないのである。

また、他の言語システムと異なり Smalltalk-80 はシステムすべてがオブジェクト指向プログラミングで統一されており、しかもそれがすべてユーザに開放されている。したがって、初心者にとってはオブジェクト指向の学習用に使えるし、専門家にとってはシステムの変更・拡張も含めた高度なプログラミングが可能である。

(2) Flavors¹¹⁾

Flavors は、Zeta Lisp 上のオブジェクト指向型言語であり、もともとは、ユーザ定義可能な構造体タイプに端を発している。この構造体タイプに対しては、その変数作成用の関数および、その構造体の各要素変数に対するアクセス用の関数が生成されるようなマクロが定義されている。Flavor はここからさらに、いくつかのタイプの共通性質をまとめたアブストラクト・タイプの定義とオブジェクトのタイプによらない操作が記述できるように拡張されたものである。この辺の記述はマニュアル¹¹⁾に詳しく書かれていておもしろい。

Smalltalk-80 がオブジェクト指向という概念から、その言語仕様が決まったのに対して、Flavors は Lisp Machine のシステムを記述するのに不可欠なものとして、いわば必要性に迫られて作られたものである。したがって、そのプログラミング環境は初心者ではなく専門家による使用を意識したものと考えられ、Smalltalk-80 に比べると使い勝手が全く異なる。つ

まり、初心者が端末の前に座ってすぐに動かせるようなシステムではないのである。かといってマニュアルを読もうにも、500 ページもあるようなものが 10 冊以上もある。要するに機能が多過ぎて、すべてを理解しようというのは不可能に近い。ライブラリは非常に豊富に揃っているが、借しむらくは、そのライブラリがうまく整理されていないので、何がすでに用意されていて、何は新たにプログラムする必要があるかがよく分からない。

プログラミング自体は、ごく普通の言語のプログラムと全く変わりが無い。つまり、テキスト・エディタでコーディングして、それをコンパイルして実行するというパターンである。したがって、マウスは全く使わなくてもプログラミングができる。実際、ウィンドウの選択や、カーソルの移動もキーボード操作だけでできるようになっている。ポップアップメニューなどマウスによる操作もあるが、中途半端な感じで、後から付けたという感じがする。したがって、対話的環境としての完成度は Smalltalk-80 には、はるかに及ばない。

(3) KEE¹²⁾

上記の二つが汎用のプログラミング言語であったのに対して、KEE は知識システム作成用の環境である。したがって、オブジェクトで表現できる対象を知識表現に限定した代わりに、そのプログラミングはできるだけ分かりやすく、対話的にできるように工夫されている。そのために、イメージやグラフィックスを多用した視覚的なプログラミング・スタイルを確立している。

KEE では知識は、ユニットといわれるオブジェクトの階層構成で表現される。ユニットには、クラスとメンバの 2 種類があり、これがオブジェクト指向におけるクラスとインスタンスに対応している。そして、ユニット間の関係としてはクラス-サブクラス間の関係とクラス-メンバ間の関係の 2 種類がある。また、クラスには、メンバ・スロットとオウン・スロットがあり、これはインスタンス変数とクラス変数に対応している。したがって一応オブジェクト指向の枠組には従っているが、フレーム表現の拡張として捉えたほうがよいところも多い。たとえば、メソッド自体もスロット中に一つの値として記述されるようになっているし、継承のさせかたも何種類もあり、スロットごとに選べるようになっている。また、各ユニットは独立にそれぞれのスロットをもっていて、その値は関係のリ

シクを張ったときに決定されるようになっている。

ユニットの定義や生成、サブクラスやメンバユニットの関係付けなどは、すべて対話的にできる。また、既存のユニットの編集もマウスによるメニュー選択で簡単にできる。初心者でもシステムの質問に答えながら入力していくことによって、簡単に知識ベースを構築できるように作られている。

4.2 既存クラス検索のためのツール

(1) システム・ブラウザ (Smalltalk-80)

Smalltalk-80 において最も特徴的なのがブラウザである。プログラミング開発の大半はこのブラウザ上で行われるといっても過言ではない。ブラウザは、システムを含むすべてのクラスを管理しているデータベースである。そして、ここでクラスの検索はもとより、クラスの追加・削除といったクラス階層の変更から、メソッドの追加・削除までのすべてができる。

ブラウザは、図-1 に示されているように、5つのサブ・ウィンドウを組み合わせで作られている。上部に位置する4つのサブ・ウィンドウは、システムのデータベースを構成するカテゴリ/クラス/メソッド・カテゴリ/メソッドの4つのレベルからなる階層構造に対応していて、クラス名および、メソッド名を表示・選択するためのものである。そして、ここで選択したクラスの階層や、メソッドのコードが下部に位置するサブ・ウィンドウに表示される。下部のウィンドウは、エディタにもなっているので、表示されたものは、変更可能であり、その結果はすぐにデータベースに反映される。また、同じウィンドウでプログラムの実行もできる。

このように、プログラミングという一連の作業で必要なことを一つのウィンドウにまとめているのが Smalltalk-80 の特徴である。ユーザにとって助かるのは、各サブ・ウィンドウで何ができるかは、マウスの中央のボタンを押せばすべてメニューで表示されるということである。しかもメニューの項目は良く整理されていて、選択に迷うというようなこともない。初心者でも簡単に使いこなせる理由がここにある。

(2) 例によるクラス検索 (Smalltalk-80)

クラスの再利用によるプログラミングのためには、既存のクラスにどのようなものがあり、それがどのように動くのかを知る必要がある。ブラウザは、既存クラスの検索に便利であるが、カテゴリ名や、クラス名だけでは、そのクラスの実際の動作を推定することは難しい。たとえば、あるクラスのオブジェクトが、どのようなメッセージを受けて、どんな反応をするかは、名前を見ただけでは分からない。これを簡単に示してくれるのがクラス・メソッドとして用意されている example というメソッドである。このメソッドはたいのクラスに定義されているが、これを盲目的に実行してみることによって、そのクラスのインスタンス生成から、典型的なインスタンス・メソッドによる例題の実行まですべてを行ってくれる(図-1 では do it を選択することによって、渦巻きが描かれる)。これは『百聞は一見にしかず』という言葉どおり、どんな記述よりも効果的である。これによって、適当なクラスが見つかったならば、そのコードを眺めて、どんなメソッドがどんなパラメータで呼ばれているかが分かる。あとは、既存のコードの切り張りでプログラミングが済んでしまうことが多い。

(3) 多重継承のための検索機構

(Flavors)

Flavors は多重継承なので、クラス間の関係は木構造ではなく複雑なネットワークになる。しかもシステムを構成している Flavor の数が非常に多いので、ネットワークは巨大になり、これを Smalltalk-80 のようにルート・クラスからの木構造というぐあいに単純に表示することは不可能である。また、全体構造が Smalltalk-80 のように秩序だてて設計されているわけではなく、ライブラリの寄せ集めという状態であるから、そもそも全体構造を表示するというのは意味がない。このような特徴をもつ Flavors システムの検索機

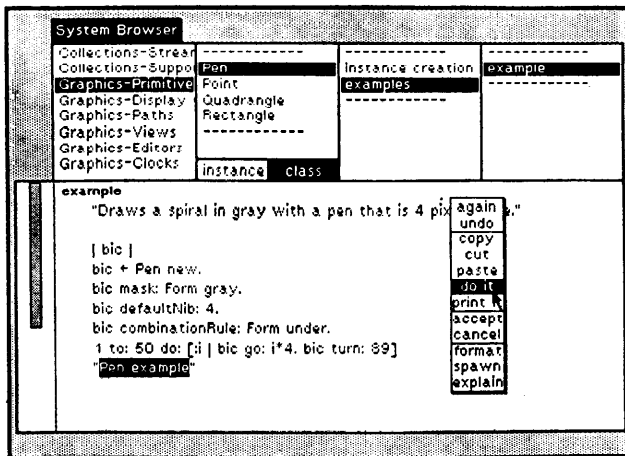


図-1 ブラウザのウィンドウ表示例⁽¹⁾

構が図-2 に示した Flavor Examiner である。これはユーザが指定した Flavor について、そのスーパークラス、サブクラス、メソッドなどをメニューに応じて表示する。これらは、直接の親や子のものだけではなく、先祖すべてとか子孫すべてとかの単位でも表示可能である。このツールでは全体像は把握できないが、特定の Flavor の周りだけが分かれば十分であるという発想であろう。

実際にあるメソッドを編集したくなったときには、それをマウスで選択すると、ファイル中のそのメソッドが定義されている部分が先頭に表示された状態で

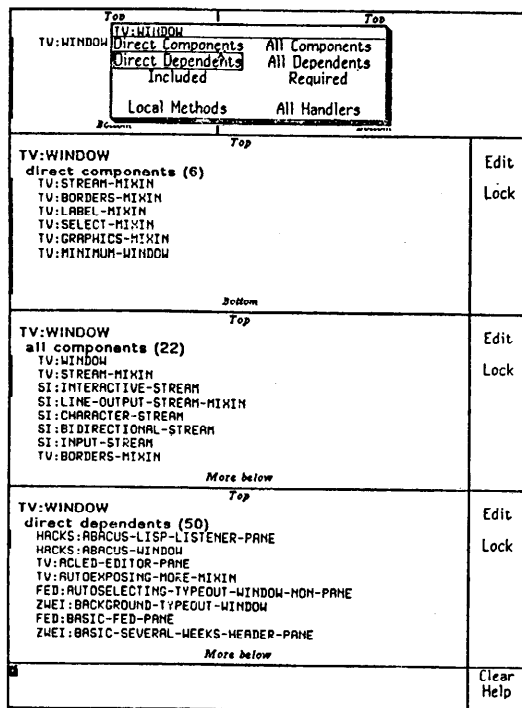


図-2 Flavor Examiner のウィンドウ表示例

ZMACS エディタのウィンドウが開かれる。このように Flavors では、ウィンドウは機能別に用意されているので、ユーザは何種類かのウィンドウの間をいったりきたりしながら作業を進めることになる。

(4) クラス階層のグラフィック表示 (KEE)

KEE のプログラミングに際してもっとも重要なウィンドウが図-3 に示した KEE ウィンドウというものである。このウィンドウ上で、知識ベースの表示、ユニットの定義や、スロットのリストの表示および、その編集ができる。表示されたものはすべてマウス・センシティブであり、そこでマウスをクリックしたときに出てくるメニューを選択することによって、対話的にプログラミングができる。

他の二つのシステムに比べて特徴的なのは知識ベースのグラフィックス表示であり、ユニット間の階層関係がユニット名とそれを結ぶ実線および破線を用いて、そのままの形で表示される。これは、もっとも直接的な表示方法であるが、本などに出てくる図と同じで視覚的にうったえるので、非常に分かりやすい。ユニット数が多くなってグラフが大きくなって、上下左右のスクロール機能と、全体図を縮小・簡素化して表示し、そのうちのどの部分がウィンドウに表示されているかを示すナビゲータなどが用意されているので、簡単に目標としているユニットが見つけれようになっている。また、そのグラフィック上で新たなユニットの生成や、関係のつなぎ換えなどの編集操作が可能である。

一方、メソッドやルールの定義は普通の Lisp で記述するために、その作成・編集には KEE の親言語である Lisp システムのエディタが必要になる。したがって一連のプログラミング作業においては、ユーザはグラフィックス表示のウィンドウ上でのマウスによるメニュー選択と、エディタ上でのタイピングを繰り返

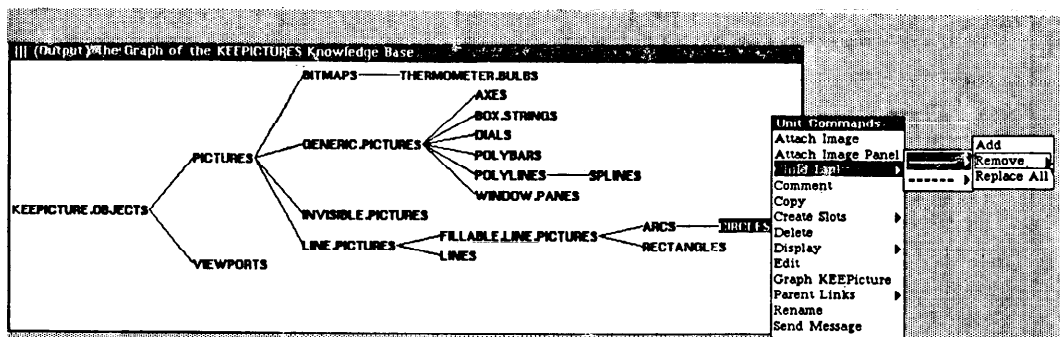


図-3 KEE ウィンドウの表示例

えさなければならず、レベルの統一性に今一つ不満が残る。

4.3 メッセージ・トレースのためのツール

(1) デバッガ (Smalltalk-80)

Lisp におけるデバッグが関数呼出しのスタックをみることによって行われるのと同様に、Smalltalk-80 においてもメッセージ送信のスタックをみるのがデバッグの基本方針である。Smalltalk-80 のデバッガは、図-4 に示されたように大きく三つの部分から構成されている。上段には、どのクラスのどのメソッドが呼び出されたかのスタックが表示され、中段には、上段のスタックで選ばれたメソッドのコードが表示され、また下段には、その実行環境が表示されるようになっている。ここでも、デバッグ作業に必要な機能を一つのウィンドウに集めるという Smalltalk-80 の特徴が現れている。

まずスタックを調べることによって、エラーが起こるまでのおおよその道筋が分かる。次に実際のコードをみることによって、バグを捜すわけであるが、コード上では現在実行中のメソッドが白黒反転で表示される。ユーザは step コマンドによって、そのメソッドの呼出しの中に入ることもできるし、send コマンドによって、実行点を次のメソッド呼び出しまで進めることもできる。(ただし、後戻りはできない。)しかも、同時にそのメソッドを実行しているオブジェクトのインスタンス変数や、メソッド内の一時変数の値といった実行環境の変化もすべて表示される。そして、このコード・ウィンドウはエディタにもなっているので、ここで直接バグを直すことができ、そのまま再実行

することができる。

(2) 実行時のメッセージ・トレース (KEE)

KEE ではメソッドごとに、それが呼ばれたときにトレース出力やブレイクをする機能がある。この機能は、メソッド・スロットについているメニューを選ぶことによって対話的に使うことができる。もし、そのユニットがクラス・オブジェクトのときには、そのメンバ・ユニット全体についてトレース出力やブレイクを指定することもできる。KEE は多重継承ではあるがメソッド・スロットに関しては、第一番目の親のメソッドを継承することになっているので、メソッドの動的探索は行われていない。したがって、メソッドのトレースは実に簡単であり、複雑な表示機能は必要ない。しかし、機能としては、単に実行時に一行一行、どのユニットのどのメソッドが呼ばれたかを出力するだけであり、Smalltalk-80 のような統合的なデバッグ環境にはなっていない。

KEE ではルールの実行結果に対しては非常にすばらしいグラフィック・トレーサを用意しているが、メソッドのトレースに関しては、Lisp のトレーサとほとんど同じレベルにとどまっている。

5. プログラミング環境の再利用

大きなアプリケーション・プログラムを作ろうとしたときに、一番、頭を悩ませるのがユーザ・インタフェースの設計である。一般にアプリケーション・プログラムでは、そのコーディングの大半が、インタフェース部分に費やされているという。したがって、前節で解説したような優れたユーザ・インタフェースを

もつプログラミング環境を、自分のアプリケーション用のインタフェースとして流用できると非常に便利である。

本章では、オブジェクト指向言語用のプログラミング環境では、このような再利用が比較的容易に行えることを説明し、その実例として、Flavors と Smalltalk-80 についてそのプログラミング環境の再利用の方法を説明する。実際の再利用にあたっては、それぞれのウィンドウ・システムがどのように作られているかを理解することが必要であり、この二つはそれを比較する意味でも興味ぶかい。

5.1 オブジェクト指向の利点

一般にマルチ・ウィンドウ・システムを実

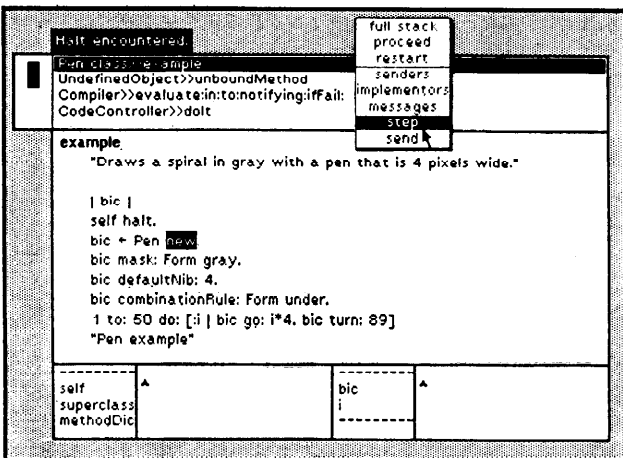


図-4 デバッガのウィンドウ表示例⁽¹⁾

現しようとしたとき、おのおののウィンドウは、基本的な操作、たとえばウィンドウの開閉や移動に対しては、同じように動作するが、一方、それぞれが異なる場所に位置し、異なる大きさを持ち、異なる内容を表示しているという点では、異なっていなければならない。このような機構を従来型言語で実現しようとする、かなり面倒なプログラムになるが、これは、オブジェクト指向のクラス-インスタンスの考え方とまさに一致するものであり、オブジェクト指向では非常に素直に記述することができる。

さらに、既存のインタフェース用クラスのカスタマイズに際しても、オブジェクト指向の特徴、つまりクラスの再利用と差分プログラミングという特徴が非常に役にたつのである。すなわち既存のウィンドウ・クラスの下にサブクラスとして自分用のウィンドウ・クラスを定義すれば、ウィンドウの開閉や移動といった基本的な操作に関しては、特に余分なプログラミングすることなくその機能を利用することができる。

もちろんこのような、アプリケーション向けユーザ・インタフェースのカスタマイズを行うためには、言語仕様上の特徴に加えて、システム自体がユーザに対して開かれたものでなければならない。その程度や方法は、ソースコードを含めてすべてをユーザに公開し、しかもそれらの変更可能である Smalltalk-80 から、ウィンドウ生成やそのアクセスのための関数だけを公開している Interlisp まで、各システムによってさまざまであるが、開かれ方が大きいほど、カスタマイズの柔軟さは大きくなるのは当然である。

5.2 多重継承によるインタフェースの再利用 (Flavors)

Flavors では多重継承によって、インタフェースの必要機能を揃えていく。これは再利用のもっとも素直な方法である。つまり、インタフェースとして必要な機能は Flavor のライブラリという形で揃えられていて、それらを多重継承するサブ・クラスを定義することによって、望むべきインタフェースのクラスを作ろうとするものである。これを Flavor の混合 (mixin) と呼び、『調味料』というこの言語の名前の由来となっている。このための Flavor はたいてい名前の後に mixin という接尾語がついていて『混合用調味料』を表現している。たとえば、一般的なウィンドウである tv: window は、図-5 に示されているように、ウィンドウとしての必要最小限の機能をもつ tv: minimum-window に、グラフィックスを出力できる

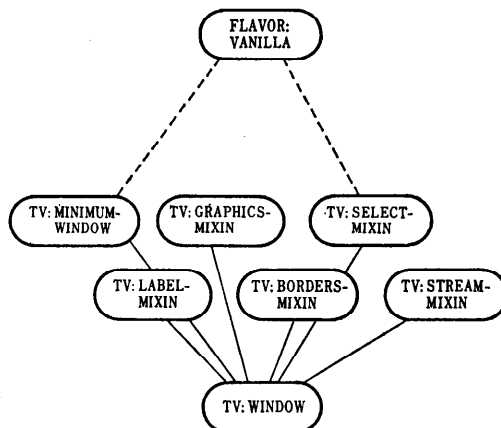


図-5 多重継承によるウィンドウ・クラスの構成

ようにするための tv: graphics-mixin、マウスによる選択が可能になるための tv: select-mixin、ウィンドウの左上にラベルを表示するための tv: label-mixin、枠を書くための tv: borders-mixin、ウィンドウからの入出力をサポートするための tv: stream-mixin を多重継承して作られている。

さて、どうやって自分のアプリケーション用のインタフェースのための Flavor を見つけだすかであるが、ユーザ・インタフェース管理システムというものによって、対話的に自分の好きなウィンドウが作れるようになっている。まず、Frame-Up Layout Designer というものによって、実際に作りたいウィンドウのアウトラインをそのままマウスを使ってデザインしていくことができ、その後、手順にしたがって、コマンドのトップレベル・ループや、入力構文チェック、マウス入力などを定義していくことによって、簡単にユーザ・インタフェースを構築していくことができる。

ただし、特殊用途のウィンドウを作るときには、自分で Flavor を捜さなくてはならず、そのときはマニュアル¹³⁾を調べて、上述したような『混合用調味料』の Flavor を組み合わせなければならないが、多重継承のメソッド探索の都合で、親 Flavor を書く順番までを考えなければならないことがあり、プログラミングは複雑になる。

5.3 MVC コンセプトによるインタフェースの再利用 (Smalltalk-80)

Smalltalk-80 では多重継承ではなく、複合オブジェクトによってインタフェースの必要機能を実現しようとしている。これを再利用するためには、Flavors のように単にライブラリを継承すればよいのと異なり、複合オブジェクトの構造を理解する必要がある。

典型的な例として、Smalltalk-80における各ウィンドウは、モデル、ビュー、コントローラの三つのオブジェクトから構成される。これを MVC とよぶ。MVC については、いろいろなところで解説がされているので¹⁴⁾、ここでは詳しくは説明しない。ここでは、実際のウィンドウ・システムがいかに種々のオブジェクトの組み合わせによって作られているかについて述べる。Smalltalk のマウスは3ボタンであり、ウィンドウ上でマウスをクリックすると、それぞれのボタンに応じて異なるメニューが現れる。たとえば、ブラウザのコードビューの上では、赤ボタンはテキストの選択であり、黄ボタンは選んだテキストをエディットしたり実行したりするメニュー選択であり、青ボタンはブラウザのウィンドウ自体を動かしたりするメニュー選択である。これをどう実現しているかという、実際に異なるボタンごとに反応するオブジェクトを重ね合わせているのである。これを図-6に示す。この例では、すべてのウィンドウの下地である Standard System View の上に、テキストのエディティングをするためのビューである Text View のサブクラスでメソッドのコードに対するメニューを用意した Code View を乗せて、さらにその上に実際のテキストである Paragraph が乗っていることが分かる。

MVC のコンセプトは、アプリケーション・プログラムとインタフェース部分の明確な分離にある。その第一の利点は、アプリケーション・プログラムは、自分のインタフェースであるウィンドウの状態を気にすることなく記述できることである。そして第二の利点は、アプリケーション・プログラムはモデルとしておのおのが開発して、それに対するインタフェースは、既存のビューとコントローラを再利用してカスタマイズできるということである。したがって、開発者は、MVC の概念をよく理解して、しかも既存のビューとコントローラにも通じていなければならない。しかし、この基本さえ理解すれば、その後できる応用範囲が非常に広いことは画期的である。特殊なウィンドウが必要になっても、それを簡単に実現することが可能になる。これは概念自体が非常に汎用的であり、しかもビュー、コントローラのクラス階層がよく整理されているためである。

6. ま と め

オブジェクト指向プログラミング環境を理解する場合に重要なことは次のことである。オブジェクト指向言語の場合、通常のプログラミング言語と異なり、プログラムの管理・再利用に係わる機能を言語内に含

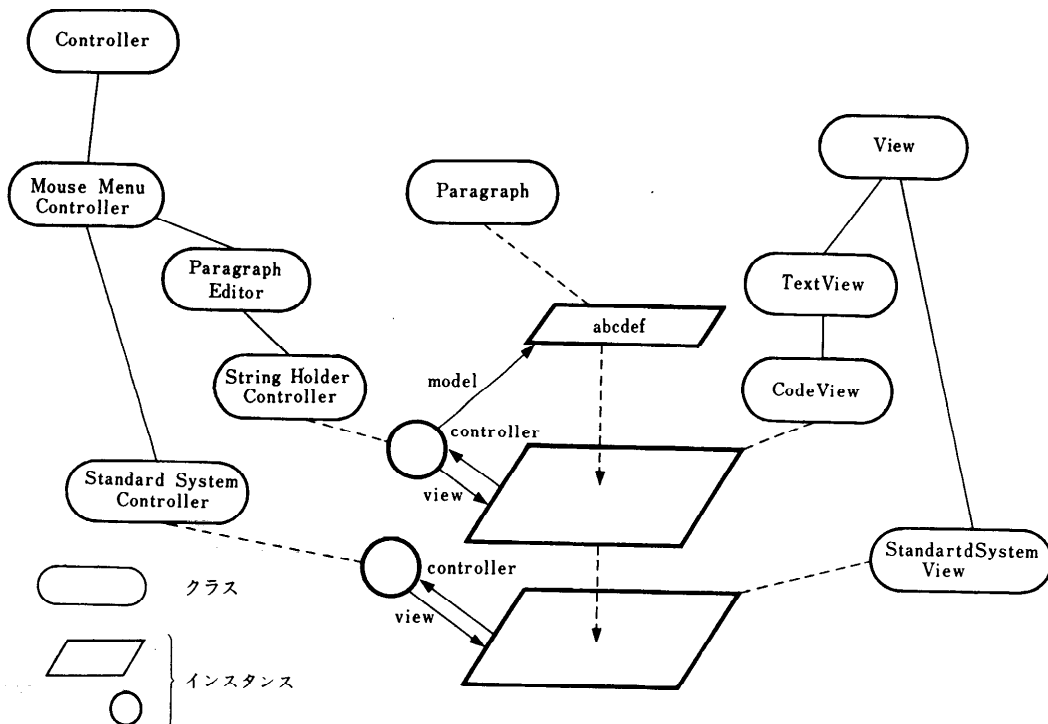


図-6 複合オブジェクトによるウィンドウの構成

み、しかもプログラミング環境用の記述言語としてそれ自身が優れているという事情がある。それゆえ、言語と環境を分離して論ずることはむしろ不自然であり、両者を統合的なプログラミング・システムとしてみる視点が要求される。この種のプログラミング・システムでは、これまで存在したプログラミングや既存のプログラムの管理などの作業モードの差が消失し、プログラマはその差を意識することなく行動することができる。その利点の一つとして、プログラムの再利用が自然に促進されることになる。このような思想を徹底して実現したものが Smalltalk-80 である。この徹底の度合は、それ自身一つの文化をなすと呼べるほどのものであり、今後のプログラミング・システム的设计に大きな影響を与え続けるであろう。本解説がこれらの文化を理解する一助となり、読者の今後のプログラミングに資するところが少しでもあれば幸いである。

参 考 文 献

- 1) Goldberg, A.: The Influence of an Object-Oriented Language on the Programming Environment, Interactive Programming Environment, Barstow, D.R. et al. ed., McGraw-Hill, pp. 141-174 (1984).
- 2) Fukunaga, K., Hirose, S.: An Experience with a Prolog-based Object Oriented Language, Proc. of the ACM Conf. on Object Oriented Programming Systems, Language and Applications, pp. 224-231 (1986).
- 3) 沼尾雅之, 藤崎哲之助: Prolog の視覚的プログラミング環境, オブジェクト指向, 鈴木則久編, 共立出版, pp. 225-243 (1985).
- 4) Sheil, B. A.: Power Tool for Programmers, Interactive Programming Environment, Barstow, D.R. et al. ed., McGraw-Hill, pp. 16-37 (1984).
- 5) Fukunaga, K.: A Knowledge Based Support Tool for Code Understanding, Proc. 8th International Conf. on Software Engineering, pp. 358-363 (1985).
- 6) 片山佳則: オブジェクト表現構築のためのクラス構成支援, WOOC '87 (1987).
- 7) 横井伸司, 福永光一: 対象指向言語用知的プログラミング環境, 情報処理学会第31回全国大会 1P-5 (1986).
- 8) 垂水浩幸他: 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).
- 9) Goldberg, A. and Robinson, D.: Smalltalk-80 The Language and Its Implementation, Addison-Wesley (1983).
- 10) Goldberg, A.: Smalltalk-80 The Interactive Programming Environment, Addison-Wesley (1984).
- 11) Overview of Flavors, Symbolics Common Lisp: Language Concepts, pp. 47-54, Symbolics Manual 2 A (1986).
- 12) Software Development System User's Manual, KEE Version 3.0, IntelliCorp (1986).
- 13) Programming the User Interface, Symbolics Manual 7 B (1986).
- 14) Ward Cunningham: Smalltalk-80 によるアプリケーション・プログラムの作り方 Model-View-Controller, bit, Vol. 18, No. 4, pp. 379-396 (1986).

(昭和63年1月5日受付)