インターフェースビルダの光と影

増井 俊之*

SHARP 株式会社

1 インターフェースビルダとは

計算機のグラフィックインタフェースの作成を支援するいわゆる「インターフェースビルダ」は近年非常に成功したアプリケーション分野のひとつといえるだろう[1]。「インターフェースビルダ」という名前は NeXT 社のワークステーションに登載されている「Interface Builder」のようなシステムという意味で本稿では一般名詞として使うことにするが、大雑把には「ウィンドウシステムのツールキット部品を直接操作によりウィンドウ画面に配置しながらアプリケーションの見栄えを設計し、かつツールを操作したときのアプリケーションの動作記述を支援するシステム」のようなものを想定している。[2] では「ユーザインタフェースの視覚的・対話的設計システム」として Interface Builder をはじめとする各種のシステムが紹介されているが、本稿でいうインターフェースビルダはそのサブセットである。

一般にインターフェースビルダと呼ばれるものは以下のような特徴を持っているのが普通のようである。

- 直接操作インタフェースによるグラフィック画面設計とテキスト編集によるプログラミングを同時進行させながらアプリケーションを作成する。
- ボタンやスライダのようなインタフェース部品それぞれに対し、それらの部品が操作されたとき (e.g. ボタンが押されたとき / スライダが動かされたとき) 呼ばれるようなサブルーチンを対応付ける。
- 配置した部品の情報からサブルーチンの雛型を生成することができる。
- 部品の配置情報は特別の配置情報ファイルに格納して利用する。

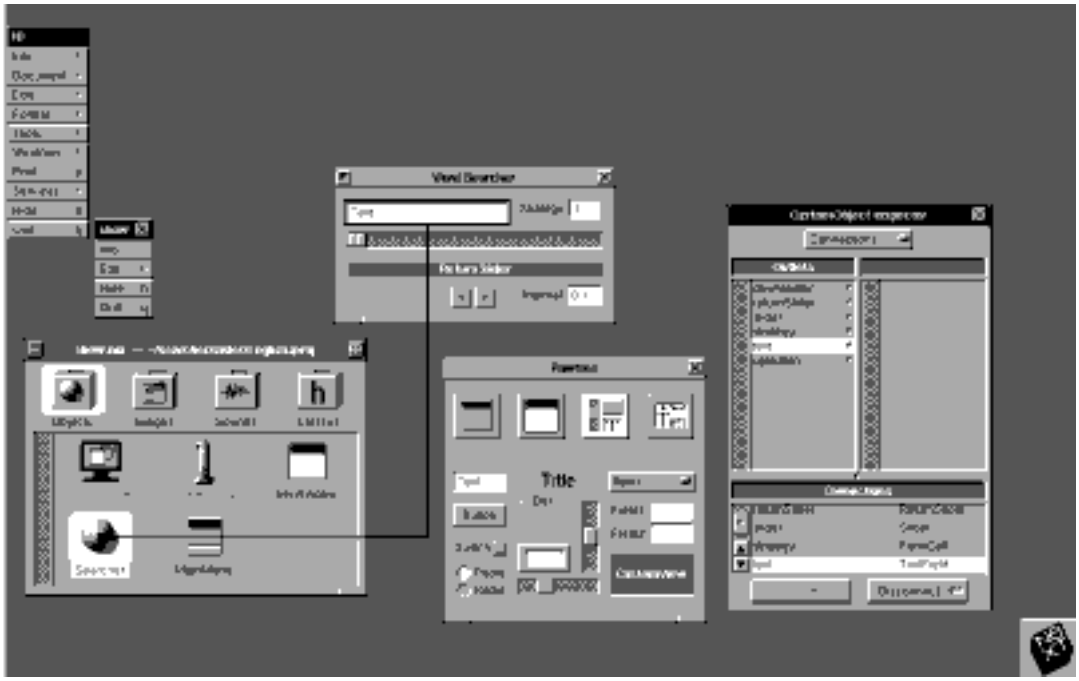

図 1: NeXT の Interface Builder

- サブルーチンをきちんと書く前にアプリケーションの見かけの動作をテストすることができる。
- アプリケーション作成のプロジェクト全体を管理する機能がある。
- インタフェース部品の初期化のためなどのコードは自動的に生成される。

2 実例

現在いろいろなインターフェースビルダが販売 / 配付されているが、ここでは実例として NeXT 社の Interface Builder, MicroSoft 社の Visual Basic, Silicon Graphics 社のワークステーションで動作する Forms Designer を紹介する。

2.1 NeXT の Interface Builder

NeXT 社は 1988 年より自社製の UNIX ワークステーションのウィンドウシステム “NEXTSTEP” [2]のための GUI 構築ツールとして “Interface Builder” をバンドルしており、これが現在の各種インターフェースビルダの元祖となっている。その後 NeXT ワークステーションのバージョンアップに対応して各種の改良が行われ現在第 3.0 版がリリースされているが、基本的な機能は変わっていない。

NEXTSTEP のツールキットは Appkit と呼ばれ、あらゆる部品が Objective-C のクラスで実現されている。例えばボタンを示すクラスは Object(クラス階層の根) → Responder(イベントに反応するもの) → View(描画可能な領域) → Control(一般のインタフェース部品) → Button というクラス階層で構成されている。 Appkit 自体は X などで使われているような一般的な特徴をもっている。ツールキットのメインループは常にマウスなどのイベントを待っており、各種のイベントが来るたびに対応するクラスの対応するメソッドが呼び出される。例えばボタンをマウスでクリックしたときは Button クラスのインスタンスに `mouseDown` というメッセージが送られる。部品が継承可能なクラスで実現されているので、既存の部品のサブクラスを作りメソッドを追加したりすることにより再利用することが可能になっている。このため例えば既存のスライダを再利用して少しだけ動作の異なるスライダを作ったりすることが可能である[3]。

Interface Builder は以下のような機能をサポートしている。

- Drag & Drop による Appkit 部品のウィンドウへの配置
- Appkit のクラス階層の管理
- メソッド、インスタンス変数の管理
- オブジェクト間の接続関係をグラフィックに操作
- 各種リソースの管理

インタフェース部品や一般のオブジェクトのインスタンスは画面上のアイコンやウィンドウなどで表現され、これらの間を線で結んで関係付けることによりインスタンス変数を初期化したりメッセージの送り先を指定したりすることができる。例えばオブジェクト `A` のインスタンス変数 `i` の初期値をオブジェクト `B` とする、というような設定が `i` と `B` を線で結ぶだけで指定できる。

対話的に設計されたクラス定義は “Unparse” 機能により Objective-C のプログラムテキストの雛型に変換される。ユーザはその後これを編集するが、その際にインスタンス変数の追加などにより前のクラス定義と整合がとれなくなることがあるので、テキストの修正を Interface Builder に知らせるための “Parse” 機能も用意されている。最新のバージョンでは Project Builder というシステムにより Makefile などの資源が管理され、デバッグやテストの機能もサポートする総合的プログラミング環境となっている。

Interface Builder はインターフェースビルダの元祖だけあって完成度は高い。 Interface Builder では視覚的以外のオブジェクトの名前や関係もある程度ビジュアルに操作することができ、また Project Builder がプロジェクト管理機能も持っているため、 Objective-C のプログラミング環境と思って使うことも可能である。

---

*本資料は 1994 年 2 月 14 日 (財) 日本情報処理開発協会における講演において使用したものである。

[1]今のところユーザがプログラマに限られているのでインターフェースビルダがたくさん売れたという話は聞かないが、独自のアプリケーション分野を形成したという点で特筆すべきと考える。

[2]最初は “NeXTstep” という名前だったがその後 “NeXTSTEP” に変わり、現在は “NEXTSTEP” と呼んでいるようである。

[3]と言うのは簡単であるが、 4節で述べるように実はこれは簡単ではなかったりする。

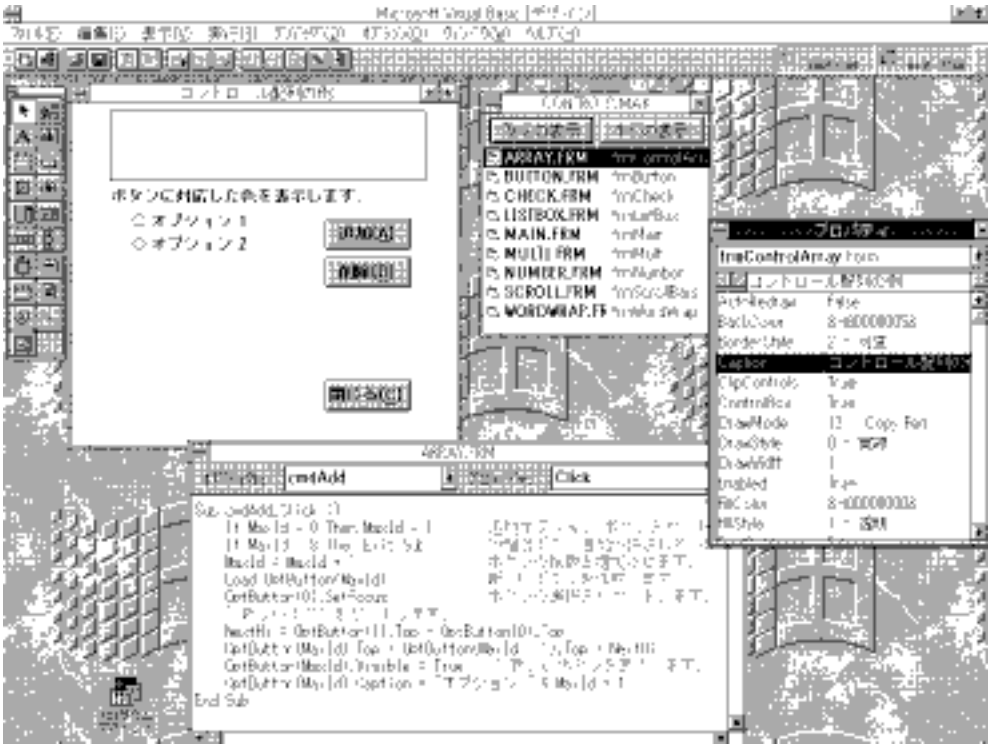

図 2: Visual Basic

2.2 Visual Basic

Visual Basic は Microsoft 社の Windows 用のプログラミング言語 / アプリケーション開発環境である。 Basic と名うってはいるが、いわゆるパソコン BASIC とは似ても似つかぬもので、単純な構造的インタプリタ言語である[4]。

Visual Basic は特定のツールキットに対応して後から開発されたインターフェースビルダではなく、最初から統合的プログラミング環境として意図されたシステムである。ボタンなどの部品はウィンドウの上で Click & Drag による直接操作により作成 / 配置し、属性をメニューで設定することができる。部品が操作されるときは、その部品名 (e.g. “`button1`”) とその上の操作名 (e.g. “`click`”) を組みあわせた名前 (e.g. “`button1_click`”) をもつサブルーチンが呼ばれるので、配置の後でこれらのサブルーチンを定義することによりアプリケーションが完成する。

Visual Basic はプログラミング言語としては洗練されたものではないが、インタプリタ言語であることからインタフェースの作成 / テストのサイクルを迅速に行なうことができるし、 OLE, DDE のような機能を使って Excel や Word のような既存のアプリケーションと組みあわせて使うこともできるし、統合的プログラミング環境としてよくできているため今後着実にユーザが増えると予想される[5]。

[4]普通の BASIC、 Visual Basic、 Perl を比較すると以下のようになるが、 Visual Basic はむしろ Perl に近い。どこが “Basic” なのだろう??

	BASIC	Visual Basic	Perl
行番号	必要	無	無
再帰呼び出し	不可能	可能	可能
変数	大域変数のみ	局所変数 / 大域変数	局所変数 / 大域変数
構造体	無	有	有 (Ver.5 以降)
制御構造	if/then/else, for 文	if/then/else, for, while, ...	if/then/else, for, while, ...
モジュール化	困難	可能	可能
文の区切り	改行またはセミコロン	改行またはコロン	セミコロン
変数の型	なし	指定可能	なし
描画命令	含む	含む	含まない
...			

[5]Visual Basic 以前は Windows のプログラミングは普通のユーザには不可能であったが Visual Basic を使えば誰でも Windows のアプリケーションが書けてしまうので、パソコン Basic の黎明期のように誰もがこぞって変なプログラムを書きはじめそうな気がする。

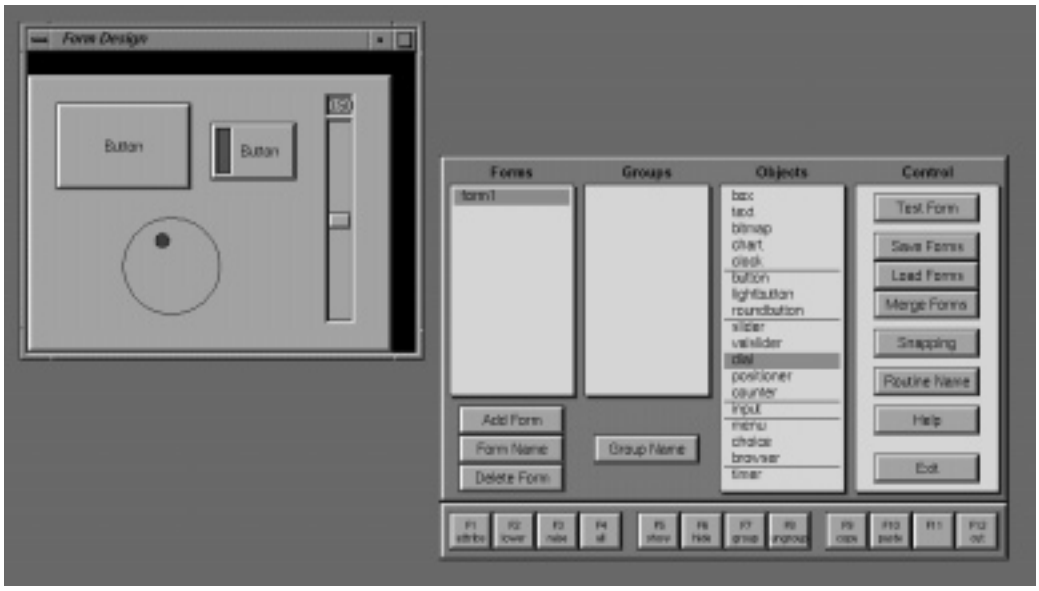

図 3: Forms Designer

Visual Basic はまた、描画のための命令を内蔵していたり[6] Drag & Drop のインタフェースを標準でサポートしていたりという特徴も持っている。

2.3 Forms Designer

Forms は Utrecht University の Mark H. Overmars により開発されたシリコングラフィクス (SGI) 社のワークステーションで動作するツールキットであり、 Forms Designer というインターフェースビルダを装備している。

Forms は比較的シンプルなツールキットである[7]。 Xlib などと異なり描画などの処理は SGI の GL ライブラリを使用するので Forms ライブラリに含まれる関数のほとんどはツール部品を作成 / 表示したり属性を変更したりするためのものとなっている。

Forms の実行の中心は `fl_do_forms()` という関数である。これが呼ばれるとライブラリはユーザの入力待ち状態となり、部品に対しなんらかの操作があったときその部品を返り値としてリターンする。 Forms Designer は他のインターフェースビルダと同様に、操作ウィンドウ (Form と呼ぶ) の上への部品の配置および属性の設定を支援し、`fl_do_forms()` を含むメインプログラムの雛型を生成する。 Forms 及び Forms Designer はシンプルではあるが、パブリックドメインのソフトウェアでソースが全て公開されており全体のサイズも小さいため見通しがよいという特徴もある。

2.4 その他

以上に紹介したものの他、 Sun の “DevGuide” や X ウィンドウ上の簡易インタプリタ言語 “Tcl” とそのツールキット “Tk” で使用する “xf” というインターフェースビルダもよく使用されているようである。

3 インターフェースビルダの利点

インターフェースビルダが流行しているのは以下のような強みを持っているためと考えられる。

- マウスでツール部品を画面に配置することはテキストで指定するのに比べ圧倒的に楽である
- 画面の設計、見栄えのテストが簡単にできる (ラピッドプロトタイピング)
- ビジュアルな部分とテキスト編集部分を相補的に使用している

[6] この点はパソコン BASIC に似ている。

[7] 1) ツールキット関数の総数が少ない。 2) 引数の数が少ない。 3) ツール部品以外に構造体をあまり使わない。という点でシンプルである。

画面の配置などはビジュアルに行なう方が速くプログラム本体はテキストエディタを使って書く方が速いという仮定があってこのようになっているわけであろうが、従来のプログラマでもとっつきやすいだろうし実際大抵のプログラマにとって上の仮定はあてはまると思われる。

- インタプリタ言語と組み合わせて使う場合特に高速にプロトタイピングができる
- 自分自身の作成に自分を使っているものが多く評価できる[8]
- プログラミング環境として統合化されている
- 結果的に素人でもウィンドウプログラミングができる

このように、インターフェースビルダは実際に役にたつ (世間に広まった) ビジュアルプログラミングシステムの代表例ということができるだろう。

4 問題点

前節で述べたようにインターフェースビルダは数多くの特長を持っているが問題点もたくさんある。そもそもインターフェースビルダがもてはやされた最大の理由は、それ無しにツールキットのプログラミングを行なうことが非常にやりにくいことが原因であり、あまり積極的なものではないように思われる。問題と思われる点を以下に列挙してみることにする。

- 用意されている仕様と異なるインタフェースを作るのが大変

  インターフェースビルダに用意されている部品がそのまま利用できれば話は簡単だが、仕様が違うインタフェースを作ろうとすると非常に手間がかかることがある。このためインターフェースビルダではユーザの様々な好みに応じるために各部品に対して沢山のオプションを用意しているのが普通であるが、オプションの数が多くなって設定が繁雑になるにもかかわらず、一般的でないインタフェースを作ろうとするとオプションでは対応しきれず苦労することになる[9]。ずっと同じインターフェースビルダを使い続けていれば段々ノウハウが蓄積されるので似たような次の仕事はなんとかできるようになるが、このようなノウハウが非常に沢山必要なので、他のインターフェースビルダ / ツールキットに乗り換えるのが困難になってしまう。

  やり方を思いつけば (知っていれば) 簡単に解けるが、それがわからないとどうしようもないという点においてインターフェースビルダやツールキットは**パズル**であるということができるであろう。おまけに簡単に解けるかどうかわからないのだからたちが悪い。同じことを実現するやり方はたぶんいくつもあるのだろうが、一番良いやりかたを見つけるのは実にむずかしい。

  ほんの少し仕様が違うものを作るのすら大変なのだから、全く違う部品をつくるのは本当に骨が折れるか全く不可能である。だからいろいろなツールやオプションが最初から用意されているインターフェースビルダやツールキットをどうしても使うことになってしまう。

- 使用言語やツールキットなどが制約をうける

  2節で紹介したインターフェースビルダはいずれも特定の言語やツールキット、描画システムと密接に結び付いており、他のものを使うことができない。例えば NeXT の Interface Builder を採用するということはすなわち

[8] 自分のコンパイラを記述できないプログラミング言語は記述力が低いのと同様、インターフェースビルダも自分自身を記述できるべきであろう。

[9] 先日筆者は NeXT の Interface Builder を使って “ マウスボタンを放すと中心に戻るスライダ ” を作ろうとした。同時に、マウスボタンを押し下げている間はずっとメッセージが出続けるようにしたいと考えた。 Interface Builder で用意されているスライダには、マウスを放したとき定位置に戻るようなオプションは設定されていないが、既存のスライダのサブクラスを作ってメソッドに少し変更を加えるだけでなんとかしようと考えた。最初に問題になったのは、デフォルトの設定ではスライダが動いたときメッセージが送られないということであった。スライダを作成した後 `setContinuous` というメッセージをスライダに送ればそのように設定できることが判明したが、実はこれは Interface Builder の属性パネルで「Continuous」を設定するだけでよいことがわかった (20 分浪費)。次に問題になったのは、マウスを放したときスライダを中心に戻す方法である。マウスボタンを放したイベントを取得すればよいと考えて四苦八苦して結局うまくいかなかったが、実は全てのスライダの移動は `mouseDown` というメッセージにより処理されることがわかったので、この処理の最後の部分でスライダの位置を中心に戻せばよいことがわかった (1 時間浪費)。最後に、スライダを操作している間じゅうスライダの現在位置を示すメッセージを出し続けるための方法が問題になった。このようなオプションも用意されていないため、マニュアルのあちこちを調べた結果、タイマイベントというものを設定して一定時間毎にタイマイベントが発生するようにして自力でイベント処理ループを書けばよいことがわかった (1 時間浪費)。このような試行錯誤を繰り返した結果、最終的に目的のインタフェースを作るのに 3 時間もかかってしまった。

UNIX, Objective-C, AppKit, Display PostScript の採用も意味してしまう。ところが現実にはいろいろな言語を混ぜて使えた方が便利なことが多い。 Interface Builder を Lisp や Perl のようなインタプリタ言語で使えると便利だろうし、 PostScript を使いたいと思う Visual Basic ユーザも多いであろうが、現状ではこういうことは不可能である。このように既存のインターフェースビルダではいろいろなツールや言語を組み合わせて使うための機能は一般に弱いため、システムを総合的に判断して有利なツールを選択し、うまくいかないところは目をつぶらざるを得ず、良いインターフェースビルダを使いたいために嫌でも指定された言語を使うというようなことになってしまっている。

- 一枚岩の統合的環境になってしまう

  インターフェースビルダを中心としたプログラム開発環境は一枚岩の統合的環境になっているものが多い。実際インターフェースビルダの多くはプロジェクト管理機能をもっており、プログラム編集、コンパイル、デバッグなどプログラム開発における各側面をサポートするようになっている。 MESA や Smalltalk に代表されるように、昔はこのような統合的プログラミング環境というものが流行ったものである。近年は UNIX に代表されるような小さなツールの集合体で構成されるプログラミング環境が主流になっているようであるが、インターフェースビルダにより昔のような巨大システムが復活するのかもしれない。

- ビジュアル部とテキスト部の相互変換が面倒

  大抵のインターフェースビルダはビジュアルに編集した部分をプログラムなどのテキストに落とす機能をもっている。これらの部分の役割の区別が明確ならばよいが、どちらを使っても同じことができることも多いのでしばしば混乱が起こる。またインターフェースビルダがプログラムを生成するときは通常雛型の生成だけを行ない、ユーザがこれを編集して最終的なプログラムを作成することになるが、この方式は以下のような不都合がある。

  – 以前修正したテキストを新しい雛型で上書きしてしまう危険がある。
  – テキストの変更がビジュアルな部分に反映されない。ビジュアル部とテキスト部の対応を常に保つように注意しなければならない。

    Interface Builder では修正したテキストを構文解析してビジュアル部に反映させる “Parse” 機能があるが、テキストを編集するたびにこれを起動するのは面倒である。

    また例えば NEXTSTEP の場合、クラス名 / メソッド名 / インスタンス変数名などを Interface Builder で指定することができるが、一旦名前を決めてしまうとその影響が多方面にわたるためスペリングミスなどあったとき後から変更するだけでもかなり大変である[10]。またオブジェクト名、呼び出しメソッド名等いろいろな名前をあらかじめ決めなければならないことも面倒である[11]。オブジェクト指向プログラミングでは一般にクラス階層は後で変更しにくいので熟慮して決めなければならないが、このことに加えてビジュアルな指定とテキスト編集による指定が混在しているややこしさや相互参照のややこしさが加わるためさらに注意が必要で、ラピッドプロトタイピングの敵となる。 Visual Basic では部品名とイベント名を並べたものを関数名として使用するなどそのあたりを簡略化しているが、そのためプログラミング言語としては変な感じになっている。

- グラフィック操作による作業がテキスト編集による作業より効率が劣ることがある

  例えばウィンドウが大量に必要な巨大なアプリケーションを作るのにインターフェースビルダを使用するとき、同じような操作を何度も何度もマウスで実行しなければならないことがある。また既存のデータや別のプログラムからインタフェース画面を自動生成することもできないのでインターフェースビルダでは手工業的作業しかできない。

- WIMP インタフェースしか考慮されていない

  これもツールキットの責任であるが、インターフェースビルダで設計できるのはグラフィックディスプレイを持つ卓上計算機上のマウスとキーボードを使うインタフェースだけであり、音声を使うインタフェースやペンを使う携帯端末などのインタフェース作成には役にたたない。

[10] 例えば Interface Builder で間違ったクラス名を設定すると、間違った名前のファイルがいくつも出来てしまうし、間違った名前が Makefile の中に書きこまれるし、他のファイルから間違った名前で参照されるし、直すのに相当骨が折れる。

[11] もちろんこれは普通のプログラミングでも同様ではあるが。

- 動的な動作を記述できない

  インターフェースビルダで設計できることは実はインタフェースのほんの一部である。インターフェースビルダを使っても図形エディタを作れるわけではないし、アニメーションを記述できるわけでもない。結局インターフェースビルダは静的なツール部品を配置することしかできないわけで、どちらかというと“レイアウトビルダ”と言った方が正解かもしれない。ユーザインタフェースの作成にでは静的なものよりも動的な反応の作成が難しいが、インターフェースビルダはそういうものをサポートしてくれるわけではない。昔の UIMS でよく使用されていたような、インタフェースの状態遷移の設計に使うこともできない。

- 制御構造が固定的

  本稿で紹介したインターフェースビルダで使用されるツールキットは基本的に複数プロセスを扱うことができない。一方擬似的に複数プロセスを扱わなければならないことは非常に多い。例えば時計を表示するウィンドウのような簡単なアプリケーションでも、時刻を更新してはそれを通知する時計プロセスとユーザ操作を処理するプロセスのふたつのプロセスが必要になるが、ツールキットは複数プロセスを扱えないのでタイマによってイベントや割り込みを発生させるといった細工が必要になってしまう。またツールキットは基本的に常にユーザの入力待ち状態にいる必要があるため、別の関数やプロセスからきたメッセージに応答できるようにするには制御構造をひと工夫する必要がある。

- 巨大になりがち

  一枚岩のインターフェースビルダは巨大になりがちなので中で何が起こっているのかわかりにくいことがある。内部動作を推測するしかないので、何かを変更する場合、とにかくやってみてから考えようという態度になってしまう。

- テスト機能が不充分

  多くのインターフェースビルダにはテストモードが用意されているが、アプリケーションは全く動かさずにテストするわけであるから殆ど意味がないし、アプリケーションからの情報を必要とするインタフェースには全く役にたたない。

5 問題点の解決方法の摸索

前節で述べたような問題点を解決するのはかなり大変のようではあるが、それぞれについて解決法を考えてみようと思う。

- グラフィック部とテキスト部の整合について

  現在一般的な OS の普通のファイルシステムを使っている限り、グラフィック操作とテキスト編集の内容を常に整合させることはかなり難しいと思われる。構文エディタを使って統合システムとすれば少しは良くなるかもしれないが、 VisualBasic のような統合的システムでもそれは実現されていない。まぁテキストのみのプログラミングでも同様の問題は起こるのだから名前ぐらい最初からちゃんと設計すれば良い話なのかもしれない。

- 制御構造について

  タイマがしばしば必要になったり制御構造をうまく書けなかったりするのはインターフェースビルダのせいではなくツールキットの責任である。並行処理をちゃんとサポートするツールキットが最低限必要である。

- 統合的プログラミング環境について

  All in one のシステムが良いか悪いかは知らないが、好きか嫌いかかと問われれば私は嫌いと答える。小さなツールを自分で作って既存のツールと混ぜて使うことができないからである。統合的インターフェースビルダでは「ウインドウはこう初期化しろ」「ボタンオブジェクトへはこうメッセージを送れ」といったことをシステムに言われる通りに書かなければならない。御丁寧にプログラムの雛型まで作ってくれるが、変なインデントのプログラムを与えられると不愉快になることすらある。

  ではインターフェースビルダはどのように分解すればよいのであろうか。単純に考えて、

  1. ボタン、スライダなどの各部品
  2. 部品を配置するための図形エディタ

3. 使用する部品の名前、リンク管理
4. プロジェクト管理
5. グラフィック描画システム

のように分解することはできる。これらを別々のモジュールとしておいて、その上に全体を管理するシステムがあれば、現存のインターフェースビルダと同じように使うこともできるし、高級なプロジェクト管理機構を使うように拡張したり、異なる言語やグラフィックシステムを使うことができるようになるだろう[12]。

独立した部品を組みあわせる方法としては例えばいろいろな部品間でひとつの共有空間を使用する方式が考えられる [1]。現在、 NEXTSTEP の部品をコルーチンによるスレッドで並行動作させ、 Linda サーバを介して他の部品やシステムと交信するという方式を実験している。このような実装では Objective-C による統合的環境という性質は失われるが、他のシステムと一緒に動かしたりすることは圧倒的に易しくなるので総合的には有利であろう。このようにすれば例えば制御画面の設計は NeXT の Interface Builder で行ない、グラフィック描画は SGI のワークステーションに任せるといった部品化が可能である。またインターフェースビルダが WIMP インタフェースにしか対応していないというのは問題でなくなるし、小さなプロセスの集まりとしてシステムを構成することにより見通しがよくなるという効果もある。並列動作が苦手というツールキットの問題ももちろん解決するだろう。

- インターフェースビルダのブートストラップ

  4節で述べたように、通常インターフェースビルダに用意されているインタフェース方式は限られておりユーザが簡単に拡張することはできない。しかしインターフェースビルダもひとつのアプリケーションプログラムであり、インターフェースビルダの作成に自分自身を使っているものも多い。ということはコンパイラ作成の場合と同じように、自分にどんどん新しい機能を追加していって次の版の開発には拡張版を使用するといったことが簡単にできてもいいはずである。これができれば、奇妙なインタフェース方式を新開発して既存のインターフェースビルダに追加し、そのインターフェースビルダでは以後その奇妙なインタフェースを標準で使うといったことも可能になる。現存のインターフェースビルダは一枚岩システムなのでこういうことはむずかしいが、小さな部品の集合として構築すればなんとかなるかもしれない。最近流行の研究結果を取り込んで、「制約解決機構付きインターフェースビルダ」とか「予測実行インターフェースビルダ」といったものを簡単に作れるかもしれない。

  また UI 部品の作成にインターフェースビルダを使うこともできるかもしれない。現存のインターフェースビルダはアプリケーションを作成するためのものであるが、部品そのものをさらに小さな部品の組みあわせとしてインターフェースビルダで設計できれば素晴しいことである。ひとつのボタンもアプリケーションと同じように開発するわけである。

  再利用は Interface Builder ではサブクラス化である程度実現されているが、充分とはいえない。たとえばスライダもダイアルも見かけが違うだけで動作は似たようなものと考えられるが、見かけと動作を分離して動作の部分を再利用できるようにはなっていない。見栄えと動作も分離、再利用できるようになればいいであろう。もちろんあんまり粒度を小さくするとそれはそれで使いにくくなってしまうかもしれないが。

6 結論

本稿で述べたような様々の問題点は早急に解決するとは思えないが、なんといってもインターフェースビルダは急成長株のアプリケーションであり、 Visual Language のひとつの方向として動向に注目しておくことが大事であろう。

参考文献

[1] Toshiyuki Masui. User interface construction based on parallel and sequential executiln specification. *IEICE Transactions*, Vol. E 74, No. 10, pp. 3168–3179, October 1991.

[2] 暦本純一, 垂水浩幸. ユーザインタフェースとオブジェクト指向, 「オブジェクト指向コンピューティング」第 6 章, pp. 213–244. 岩波書店, 1993.

[12] Interface Builder や Forms はグラフィックシステムは含んでおらずユーザが自力でプログラムすることになっているので 5. は分離されているといえる。 DevGuide では使用するグラフィックシステムを PostScript, Xlib などから選択することができるようになっている。