プログラミング—言語・基礎・実践— 10−15
（1993. 1. 29）

ユーザー入力のパターンを用いたGUIの記述と実装

北山 文彦
日本アイ・ビー・エム (株) 東京基礎研究所

質の高いGUIを記述・実装するための枠組としてパターン駆動の概念を提唱しその有効性を議論する。ユーザーインターフェースの設計は表示部分と対話部分に分けることができるが、本研究で提唱する概念は対話部分に関するものである。具体的には、パターン駆動は、(1) 記述方法、および (2) 実装のフレームワークを規定するもので、GUIの質を決定する重要な要因である操作の一貫性を考慮したものである。また、GUIの実装の場面においては、このフレームワークによりインタラクションスタイルと呼ばれる操作法の概念も部品化され、GUIの実装がより効率的に行なうことができるようになる。

PATTERN DRIVEN USER INTERFACE: DESCRIPTION AND IMPLEMENTATION FRAMEWORK

Fumihiko Kitayama
IBM Research, Tokyo Research Laboratory
5-19, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan

This paper proposes a new concept for GUI design and implementation, *Pattern Driven User Interface; PDUI.* Design of GUI consists of the presentation part and the dialog part. PDUI focuses especially on the dialog part of GUI. In the concrete, PDUI is regarded as (1) a description language and (2) an implementation framework. The description language is intended to represent consistency among manipulations, consistency which is tightly related to the quality of the GUI. On the other hand, the productivity of application programs with GUI increases using the framework.

1 はじめに

計算機が小型化しユーザーの裾野が広がるにつれて、多くのアプリケーションでユーザーインターフェースとしてGUIを備えるようになった。しかし、その設計や実装に関してはプログラマの経験がまだ蓄積されておらず、GUIの設計および実装はアプリケーション開発の手間のかなりの部分をしめている [5] 。本稿では、従来のイベント駆動の考え方をさらにすすめたパターン駆動の概念を提唱し、このフレームワークが質の高いGUIの設計や効率的なアプリケーション開発に有効であることを述べる。

GUIの設計は、ユーザーへの表示部分とユーザーとの対話部分に分けられる。ユーザーへの表示部分に関しては、多くのUIMSやGUI builderに見られるように直接操作を用いたレイアウターによって開発が簡単に行なえるようになっている。一方、ユーザーとの対話部分の設計・実装に関しては、最近になってイベント駆動の考え方が一般的になってきているが、そのプログラミングはまだまだ複雑であり、よい対話の設計に関する指針を与えているとは言えず、課題を残した部分である

本稿のフレームワークは、この対話部分の設計と実装に関するものであり、パターン駆動ユーザーインターフェース (Pattern Driven User Interface; PDUI) と呼ぶ。PDUIは、具体的には記述言語と実装フレームワークからなる。(1) 記述言語としては質の高いGUIの簡明な記述を行なえること、(2) 実装フレームワークとしてはGUIの効率のよい開発が行なえること、がPDUIの目標である。

GUIの設計を行なう場合、プロトタイピングの手法を中心にして、試行錯誤により設計を行なうのが一般的であり、ユーザーにとって使いやすいインターフェースとは何か、その設計方法論、必要なツール等については、明らかにされているとは言えない。対話部分に関しては、操作の一貫性が重要であることが言われており [6][9] 、PDUIでは、この一貫性に注目して、質の高いユーザーインターフェースが自然にかつ簡潔に表現できる。

また、GUIの構築方法論については、表示部分、対話部分 (あるいはまとめてGUI部分) 、それ以外のアプリケーション部分に分離して開発がすすめられることが望ましい [1] 。なぜなら、各部分は専門の設計者が独立して設計を行なうことができ、開発や保守も独立に行なうことができ、開発効率が上がるためである。さらに、各部分の部品化も容易であり、生産性に寄与する。多くのUIMSではこの原理に従っている (例えば、[1]) 。

しかし、実際には、通常のUIMSの実装方法のようにサブシステムによる分離では実行効率や可搬性の面から問題が多く、実用的アプリケーションの開発には不向きである。このため、最近は、一旦CやC++のコードを生成してからコンパイルを行なう生成方式がとられるようになっているが、この場合、上記に述べたようなユーザーインターフェース部分との分離が不完全であり、開発効率を下がりやすくなっている。UIMSのようなサブシステム型とGUI bulider のような生成型の特長をうまくとり入れた方式を考える必要がある。

一方、オブジェクト指向の考え方が一般になるにつれて、アプリケーションの設計においてもオブジェクト指向をとりいれて、設計の再利用をはかり、ソフトウェアの生産性を上げることが考えられている。アプリケーション・フレームワーク [2] は、クラスそのものだけではなく、それらとその用い方まで含めた再利用可能な設計の枠組であり、そのような試みの一つである。その適用分野ごとに枠組を考える必要があるが、例えば、GUIのフレームワークとしては、MVC[3] 、UniDraw[8] 、MacApp[7] 、などが有名である。

GUIフレームワークの利点としては、設計が容易になると同時に、部品化可能なコンポーネント (クラス) の抽出が容易であり、それらの再利用や組み換えにより、GUIを容易かつ柔軟に構築できることである。また、これらのコンポーネントは、GUI設計上のいくつかの基本概念に対応するものであり、プログラムの理解を容易にする効果もある。従来のフレームワークの概念として、View, Command, Manipulator などがあるが、本稿のフレームワークでは、さらにユーザーとのインタラクションのスタイルをコンポーネント化することにより、高機能なフレームワークを目指している。

以下では、PDUIの概念について説明し、特に記述言語としての側面、実装フレームワークとしての側面について述べる。次にPDUIの応用について述べ、その有効性について議論する。

2 PDUIとは何か

この節では、PDUIの定義に必要な用語をまず定義し、次に、PDUIの概念を定義する。PDUIは、記述言語としての側面と実装フレームワークとしての側面を持つが、PDUIの考えにもとずいた仕様記述言語の記述法、および、それによって記述されたユーザーインターフェースを実装するためのフレームワークの詳細について説明する。

2.1 基本概念

対話処理におけるユーザーインターフェースを考えるにあたって以下の用語を定義する。

アクション ユーザーが機械に対して行なう最小の動作の単位。アプリケーションが実装されているハードウェアに依存する。例えば、マウスのボタンが押された、離された、キーボードがタイプされたなどである。アクションはいくつかの種類に分けられるが、さらに属性をつけることによって個々のアクションの区別をつける。属性としては、マウスのボタン押下なら左ボタンと右ボタンのうちどちらが使われたかというような情報である。

アクション列 いくつかのアクションを組み合わせたもの。アクション単体もアクション列である。組み合わせ方としては、通常、アクションを時間的に並べる逐次列がある。しかし、2つ以上のアクションを同時に行なう並行列や、逐次列と並行列を組み合わせた複雑なアクション列を考えることもできる。例えば、ドラッグはマウスボタンを押すアクションやマウスを動かすアクションを組み合わせたアクション列で表現される。

物理操作 一つのまとまった操作を表わすアクション列。通常は、アプリケーションの意味をもった一つの操作となる。

インタラクションスタイル 物理操作を構成する部分的なアクション列の構造。単なるアクション列ではなく、一般的構造を表現できるようにしたある種の正規表現である。これは、アプリケーション全体あるいはある操作範疇内の操作の統一性を表現するのに用いられる。例えば、グラフィカルエディタの操作で、操作対象を選択してから操作内容(コマンド)を指定するというのは一つのインタラクションスタイルである。

コマンド ある物理操作に対応してアプリケーションに指令される命令。コマンドは物理操作にマッピングされるものであり、これ自体に物理的操作方法の概念は含まない。例えば、データベースを更新するというのは一つのコマンドである。しかし、更新をするためにユーザーはコマンドラインからキーボードでコマンド名をタイプするのか、マウスでアイコンをクリックするのかは問題にしない。

メソッド コマンドが実行されるときに、呼ばれるアプリケーション内のコード(プログラム)の一単位。一つのコマンドに対し、一つのメソッドだけが呼ばれるとは限らず、いくつかのメソッドがあるルールに従って組み合わされて呼ばれる。すなわち、一つのコマンドの実行を分解した一単位である。

2.2 PDUI 定義: パターン駆動によるインタラクション処理

パターン駆動ユーザーインターフェース(PDUI)とは、ユーザーの入力するアクション列のパターンにより、対応するメソッドを呼び出すような、記述および実装のためのフレームワークである。パターンというのはある構造をもったアクション列の集合の表現であり、ユーザーの入力するアクション列がその構造を持つ(一致する)場合に、パターンに束縛されたメソッドを呼び出す。一連のアクション列に対し、複数のパターンが一致してもよく、この場合、一つの物理操作に対し複数のメソッドが呼び出されるが、これらのメソッドは適当な方法で組み合わされる。このパターンは、上述のインタラクションスタイルに他ならない。

例えば、画面上にアイコンとして表示されたオブジェクトを画面上の別の地点へ移動するというコマンドを考える。この物理操作としては、オブジェクトをマウスで指定し、ドラッグする(ボタンを押したままマウスを移動させる)というものである。このとき、インタラクションスタイルとしては、(1) オブジェクトの上にマウスポインタをもってくる、(2) マウスの右ボタンを押す、(3) ドラッグする、という3つが考えられる。それぞれに束縛された意味としては、(1) は対象となるオブジェクトの指定、(2) は移動というコマンドの指定、(3) は移動先の指定、である。

ここで、移動ではなく複写を行なうコマンドを考える。このコマンドは移動のコマンドとほぼ同様なコンテキストであるので、操作の一貫性を考えると、インタラクションスタイル (1) と (3) を用いなければならない。移動コマンドとの違いは、マウスの右ボタンではなく左ボタンを使うということにすると、(2)′マウスの左ボタンを押す、という新たなインタラクションスタイルを追加するだけでよい。これには、複写のメソッドが束縛される。このようにPDUIの記述ではわずかな追加で、複写コマンドの物理操作を一から作り直す必要はなく、しかも一貫性のあるユーザーインターフェースを構築していくことができる。

PDUI は、従来のイベント駆動と比べると: (1) 処理ルーチンの呼び出しがイベントという簡単なものではなく、アクション列という構造をもったものである、(2) 一つの物理操作に対し、複数のメソッドを自動的に組み合わせて呼ぶ、という違いがある。これにより、高度な呼び出し処理が簡単に記述できる。すなわち、従来のイベント駆動の方式は、一個一個のイベントに対して単独の処理ルーチンが起動されるだけであったので、複数のイベントが連係するような場合はプログラマの責任で分散したそれらの処理ルーチンを実装する必要があった。PDUIでは、複数イベントの連係処理は自動的に行なわれるため、記述する必要はなく、それらの処理も一箇所に実装すればよい。また、メソッドを組み合せてコマンド処理を行なうので、一つ一つのコマンドを別々に実装するよりも、コードの記述量も少なく共有や再利用が促進されると考えられる。

パターン駆動という点では、UNIX 上の sed や awk などに代表されるようなストリームを対象にした言語と似ている。しかし、これらのプログラムは、文字列が処理対象であるのに対し、PDUI ではユーザーの動作であるアクション列が対象となる。また、アクション列には、単純な逐次的な列だけではなく、同時に動作をするといった並行列の構造もあり、単なるストリームよりは複雑である。

```
application:
<アプリケーションのクラスインターフェースの集まり>
interaction:
<インタラクションスタイルの記述の集まり>
window:
<ウィンドウの記述の集まり>
cast:
<変数の宣言の集まり>
```

図 1: 記述の概観

2.3 仕様記述言語としてのPDUI

ここでは、PDUIの概念をとりいれたユーザーインターフェース仕様記述言語について説明する。実際に仕様の記述が行なわれるには、PDUIの概念を用いた対話部分の仕様だけでは不十分で、ユーザーへの表示部分の仕様も記述できる必要がある。そこで、この対話部分と表示部分の記述単位としてウィンドウという概念を導入し、表示部分の記述はこのウィンドウ記述の中で行なわれる。

図1に示すように、記述は4つの記述セクションの集まりからなる。それらは、アプリケーションコードとのインターフェースの記述セクション(application:)、対話の仕様を記述するインタラクションスタイルの記述セクション(interaction:)、GUI記述の中心となるウィンドウに関する記述セクション(window:)、さらに、対話処理を実装するための変数を定義するセクション(cast:)からなり、それぞれタグ名に続いて記述が行なわれる。なお、これらの記述セクションの順番は任意である。

以下では、それぞれの記述セクションについて、そこで用いられる概念と記述法について説明する。最後に簡単な記述例を紹介する。

2.3.1 アプリケーションコードインターフェース・セクション

PDUIを用いた記述言語では、アプリケーションの実装はC++のようなオブジェクト指向言語で行なわれることを想定している。従って、ユーザーからの入力アクションによるコマンドの処理は、あるクラスのオブジェクトに対しメッセージを送ることによって行なわれる。このようなクラスのプロトコルを記述するのがこのセクションである。実際の記述法は、通常のオブジェクト指向言語のクラスインターフェースの記述法と同じである。例えば、アプリケーションコードがC++で記述されていれば、C++のクラスインターフェースの記述の集まりとなる。

2.3.2 アクション列の記述法

ここでは、記述の基本になるアクション列の表記法について述べる。アクション列は、アクションの組み合わせとして表現されるが、一つのアクションは、ActionName(Args) で表現される。ActionName はそのアクション固有の名前である。Args はアクションに応じた引数で、例えば、マウスの座標がわたされる。カンマで区切って2つ以上の引数が指定されてもよい。アクションの組み合わせ方には、逐次列と並行列の2つがあり、a1,a2 をアクションまたはアクション列としたときに、それぞれ、a1 -> a2, a1 + a2 で表わされる。

例えば、マウスカーソルをある場所(Place)に移動してマウスの左ボタンをクリックするという操作は、

```
MouseMoveTo(Place) -> PushButton(Left)
-> ReleaseButton(Left)
```

と表わされる。さらに、キーボードのシフトキーを押しながら、マウスの右ボタンで Place1 から Place2 へドラッグするという操作は、

```
MouseMoveTo(Place1)
-> (  PushButton(Right)
      -> MouseMoveTo(Place2)
      -> ReleaseButton(Right)
   ) + Press(ShiftKey)
```

となる。ここで、括弧はアクション列の部分列をグループ化するのに使われる。

2.3.3 インタラクションスタイル・セクション

アクション列は、ユーザーインターフェースの操作の構造を表現するものである。この操作の構造を決定する部分構造がインタラクションスタイルであり、インタラクションスタイルの集合を指定すると、そのシステムあるいはアプリケーションのユーザーインターフェースの操作体系が決定される。一つ一つのインタラクションスタイルは種々の操作の意味と結びつけられており、一連の物理操作を構成するインタラクションスタイルの意味から、その物理操作にマッピングされたコマンドの意味が導出される。インタラクションスタイルに対応する意味は、具体的には、呼び出されるメソッドによって指定される。従って、一つのインタラクションスタイルの記述は、アクション列の部分構造とそれに対応するメソッドからなる。

アクション列の部分構造は、アクション列と同等な表現がされる。さらに、部分構造をより一般的に表現するために、正規表現のような記法を用いる。すなわち、"*"、あるいは、"*" ではじまるアクションやアクションの引数は、任意のアクション列や引数を表わすとする。例えば、*->Action1 は Action1 で終了する任意のアクション列を、Action2->*->Action3 は Action2 で始まり Action3 で終了する任意のアクション列を表わす。

メソッドの呼び出しは、そのインタラクションスタイルが活性化されたウィンドウ自身(これもオブジェクト)か、アプリケーションとのインターフェースで記述されたクラスのオブジェクトへのメッセージ送信である。この記述は、ウィンドウへのメッセージ送信の場合は MethodName(Args) で、他の特定のオブジェクトへの場合は ObjectName->MethodName(Args) で表現される。従って、基本的なインタラクションスタイルは、

```
define InteractionName
  Action-Seq-Pattern => MethodName(Args);
```

で表わされる。ここで、InteractionName は、そのインタラクションスタイルにつけられる名前である。Action-Seq-Pattern は、アクション列の表現に "*" を導入して、部分構造を一般的に表現したものである。このパターンにマッチするアクション列がユーザーから入力されたとき、=>の右側のメソッドが呼ばれる。なお、インタラクションスタイル名の後に () でくくって引数を与えることができる。引数をつけたインタラクションスタイル名が指定されたとき、定義中のアクション列やメソッド呼び出しの対応する部分は実引数の文字列で置き換えられたものが使われる。

さらに、インタラクションスタイルは、他のインタラクションスタイルによって規定されることもある。このとき、アクション列の部分構造は指定されたすべてのインタラクションスタイルの部分構造のどれかであり、呼ばれるメソッドは、パターンにマッチしたインタラクションスタイルのメソッドが適当な方法で組み合わされてすべて呼ばれる。このときの記述は、

```
define InteractionName
       I1, I2, ...;
```

となる。I1, I2, ... は、他のインタラクションスタイルの名前である。

2.3.4 ウィンドウ・セクション

ウィンドウは通常の GUI におけるウィンドウやウィジェットと呼ばれるものと同等の概念である。実際にディスプレイ上で目に見えるような矩形領域を表わし、単独あるいは他のウィンドウの組み合わせによって構成されている。この他のウィンドウのことを子ウィンドウ、逆に自分自身が別のウィンドウの構成要素の一つになっているときは、このウィンドウを親ウィンドウと呼ぶ。従って、そのウィンドウの見た目を規定する記述がウィンドウにはあり、子ウィンドウのレイアウトやウィンドウのスタイルなどの情報を記述する。

ウィンドウには、このような見た目の情報だけではなく、ユーザーがそのウィンドウに対してどのようなインタラクションができるかの対話処理の仕様も記述される。すなわち、ユーザーはそのウィンドウに対して、どのようなコマンドや情報を入力できるか、また、そのときの物理操作はどのようにするかを、ウィンドウごとに規定する。実際には、前述のインタラクションスタイルを列挙することによって記述する。

すなわち、図2に示すように、ウィンドウの記述要素として、見た目を規定するスタイル (style:) とウィンドウに対して可能な対話処理を規定するふるまい (behavior:) のサブセクションがある。

スタイル・サブセクションは、そのウィンドウを構成する子ウィンドウの名前およびその属性のペアの一覧からなる。ここで、属性はその子ウィンドウのスタイル情報 (カラーや枠など) や子ウィンドウ同士のレイアウト情報である。特にレイアウト情報は、ウィンドウ名あるいは構造をもったウィンドウ名の集まりで記述される。例えば、

```
horizontal {
  Child1 : Window1;
  Child2 : Window2;
           Window3;
}
```

は、ウィンドウ Window1, Window2, Window3 が水平に配置され、そのうち、はじめの2つには Child1, Child2 という名前がついていることを表わしている。この例のように、子ウィンドウ名とその属性は : で区切り、他のペアとは、; で区切る。3番目の子ウィンドウのように子ウィンドウ名は省略することができる。また、レイアウトを表現する構造は、この例のようにキーワードのあと、{} でくくって記述する。レイアウトの属性は、horizontal の他に vertical, frame by などがある。

```
define WindowName < SuperWindowName ... {
  style:
     ChildWindow: ....
     ....
  behavior:
     InteractionStyle....
     ....
}
```

図 2: ウィンドウの記述の概観

ふるまいを規定するサブセクションは、前述のインタラクションスタイル・セクションで定義したインタラクションスタイルの名前を列挙する。実際にそのウィンドウにアクション列が入力された場合、指定されたインタラクションスタイルのパターンとマッチすれば、そのインタラクションスタイルのメソッドが呼び出され、アプリケーションが実行される。

ウィンドウは一種のクラスであり、あるウィンドウの記述は別のウィンドウの記述を継承することができる。継承される方をスーパーウィンドウと呼ぶ。継承するウィンドウは、すべてのスーパーウィンドウに定義されたスタイル、レイアウトの情報やインタラクションスタイルのセットが有効である。図2に示すように、一行目の<に続けてスーパーウィンドウの並びを記述する。

2.3.5 変数・セクション

PDUI によるユーザーインターフェースの記述の最後のセクションは、そのアプリケーションプログラムに実際に現われるインスタンスの指定である。ここまでのセクションでは、言わゆるクラスの定義のみで、実際に動作するオブジェクトとして何があるかは指定されていなかった。このセクションでは、アプリケーションが起動されたときに実際に生成されるオブジェクトの名前とそのクラス (ウィンドウの場合はウィンドウ名) のペアが列挙される。アプリケーションが起動されたら、PDUI のシステムはこの記述に従ってオブジェクトを生成し、アプリケーションコードやウィンドウ・セクションの記述に従ってアプリケーションを動作させる。

2.3.6 記述例

```
application:
   class Calculator {
      void close( void );
      ...
      void add( void );
      void mult( void );
      ...
   };
cast:
   Calculator: class Calculator();
interaction:
   define specialOperation
            shiftKey(shift);
window:
   define spbutton < button {
      style:
         Self: framed by Frame;
         Frame: color green;
      behavior:
         specialOperation;
   }
   define close_button < spbutton {
      style:
         Frame: color red;
      behavior:
         clickIn(DisplayArea, Calculator,
                 close);
         typeIn('q', Calculator, close);
   }
   ...
   define data_button < button{
      style:
         Number: interger;
      behavior:
         { initialize(N) => Number = N; };
         clickIn(DisplayArea, Calculator, Number);
         typeIn(Number, Calculator,
                inputdigit(Number));
   }
   define func_button < button {
      style:
         Key: string;
         Message: message;
      behavior:
 { initialize(D,M, K=D) => Message=M, Key=K };
         clickIn(DisplayArea, Calculator, Message);
         typeIn(Number, Calculator, Message);
   }
   define calc_window < title_window {
      style:
         ClientArea: vertical {
            horizontal {
               close_button("Close");
               ...
            }
            DigitWindow: char_window("0"),
                         size (11 char, 1 char),
                         color blue;
            horizontal {
               data_button(7);
               data_button(8);
               data_button(9);
               func_button("+", add);}
            horizontal {
               data_button(4);
               ... }
            horizontal {
               ...
            }
            ...
      }
   }

cast:
   MAIN: calc_window;
```

図 3: PDUI 記述例 - 電卓の例

図3に、簡単な電卓の記述例をあげる。紙面の都合上、一部を... で省略をした。また、多くのアプリケーションで共通に使うようなウィンドウ定義やインタラクションスタイルはすでに定義済みとしている。この例では、一般的なボタンウィンドウである button を定義済みとして継承するのに用いており、また、マウスのクリック (clickIn(...)) やキータイプ (typeIn(...)) などのインタラクションスタイル名も定義なしで用いられている。

この電卓は、画面上に操作ボタンと結果表示窓を表示し、主にマウス操作で計算を行なう。また、キーボードによる操作も可能になっている。

まず、アプリケーションコードインターフェース・セクションで、GUI 以外のコードである Calculator クラスのインターフェースが記述される。このクラスは計算を行なったり数値を入力するメソッドが用意されている。次の変数セクションで、このクラスのインスタンス Calculator を作ることを指定している。このオブジェクトは、MVC モデルのモデルオブジェクトにあたるものである。

次のインタラクションスタイル・セクションでは、操作時にシフトキーを併用しなければならないというスタイル一個のみを定義している。他のインタラクションスタイルは既に定義済みのものを使うので、この電卓の例ではこれだけしか新たなものを定義していない。

ウィンドウ・セクションでは、電卓本体のウィンドウとそこに使われる種々のボタンウィンドウの定義が列挙されている。これらのボタンは定義済みウィンドウである button を継承して定義されている。例えば、close_button は、先に定義したインタラクションスタイル specialOperation を含むので、このボタンはシフトキーと併用しないと働かない (このボタンは終了操作であるので誤操作を防ぐためにこうしたのである) 。また、枠の色も赤に変更している。インタラクションとしては、組み込みのインタラクションスタイル clickIn や typeIn を用いて、シフトキーを押しながらボタンがクリックされるかキーボードから'q' がタイプされれば、Calculator オブジェクトに close というメッセージを送る。これは終了処理に他ならない。data_button や func_button も同様に、数値キーや演算キーが押されたときの処理を記述しているが、結局は Calculator オブジェクトに対応するメーセージを送っている。

ここで、ウィンドウのふるまいの記述の中で、アクション initialize に対する処理を定めたインタラクションスタイルが存在するが、このアクションはユーザーの行なうアクションではなく、そのウィンドウのオブジェクトが実際に生成されるときに呼ばれる特別なものである。ここでは、ウィンドウとして生成されたときに、異なったメソッドが呼べるようにオブジェクトのインスタンス変数にメソッド名などを束縛している。

電卓の本体 (calc_window) は定義済みのウィンドウ title_window(タイトルのついた標準的ウィンドウ) を継承して作っており、先に定義したボタンなどを配置し

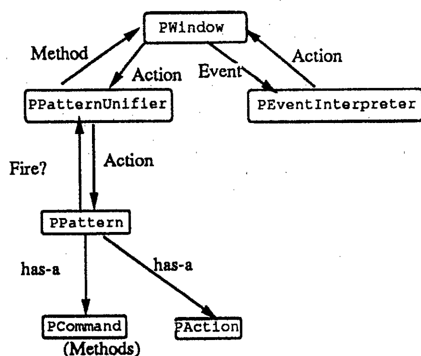

図 4: PDUI 実装フレームワーク - オブジェクトダイヤグラム

ている。このウィンドウ自体にインタラクションの定義はしていない。

2.4 実装フレームワークとしてのPDUI

この小節では、PDUIのもう一つの側面であるユーザーインターフェース部分の実装のためのフレームワークについて述べる。ここでフレームワークとは、プログラムが動作しているときにどのようなオブジェクトが存在し、それらの関係がどうなっており、メッセージをだれに送っているかを抽象レベルで規定するものであり、オブジェクト指向言語でアプリケーションを構築する際の再利用可能な設計の枠組である。

ここでは、PDUIの考えに沿って記述されたユーザーインターフェースを実装するためのフレームワークを説明する。このフレームワークの目的は、(1)PDUIの考え方を仕様の記述から自然な形で実装すること、(2) アプリケーション部分とUI部分の分離が行なわれそれぞれ独立に開発が行なえること、(3) インタラクションスタイルなどのPDUIの概念が実装の上でも再利用可能な部品となっていること、である。

処理の概略は、(1) ユーザーによって発生したイベントをPDUIにおけるアクションとして記号化する、(2) 各ウィンドウに対してこのアクションを分配する、(3) 各ウィンドウにはインタラクションスタイルのオブジェクトの集合があり、現在のアクションをブロードキャストする、(4) 各インタラクションスタイルオブジェクトは、アクションの入力に対し有限オートマトンになっており、自分のパターンとそれまでのアクション列がマッチしたとき発火する、(5) 各ウィンドウはインタラクションスタイルオブジェクトの中から発火したもののメソッドを呼び出す。

図4にオブジェクトダイヤグラムを示す。PWindowはウィンドウに対応するオブジェクトで、図には省略されているが、複数のウィンドウが親子関係のツリーを構成している。各ウィンドウにはPDUIの処理をするためのいくつかのオブジェクトが付随する。ウィンドウへシステムのイベントディスパッチャーからシステムのイベントが入力されてくる。ウィンドウは、このイベントをPEventInterpreterに送りアクション(これもオブジェクトになっている) に変換してもらう。このイベントインタプリタはシステムの生のイベントをPDUIで処理できるように記号化する役割を持つ。

アクションオブジェクトは、ウィンドウからPPatternUnifierに送られる。パターンユニファイヤはそのウィンドウが持つインタラクションスタイルに対応したオブジェクトPPatternの集合を持っており、送られてきたアクションオブジェクトをすべてのパターン処理オブジェクトにブロードキャストする。パターン処理オブジェクトは、先に述べたように有限オートマトンになっており、自分自身のパターンとそれまでのアクション列がマッチするかを判別し、マッチすればパターンユニファイヤに知らせる。

パターンユニファイヤは、マッチしたパターンとして何があるかを調べ、それらのメソッドを呼び出す。メソッドが複数あれば、それらを適当に組み合わせて適当な順番で呼び出す。順番の決め方は、メソッド(PCommandオブジェクトのサブクラス) を範疇化してそれぞれに優先順位をつけることによって行なう。

このようなフレームワークにすることにより、付随するオブジェクトまで含めたウィンドウオブジェクトや、パターン処理オブジェクトは再利用可能な部品として使うことができる。すなわち、ウィンドウの外見だけではなく備わっているインタラクションまで含めてウィンドウは独立した部品として使うことができるし、インタラクションスタイルはPPatternオブジェクトの集合に要素に入れるだけで新しいインタラクションスタイルを別のウィンドウに付加することができる。

3 PDUIの応用

3.1 操作の一貫性

PDUIを記述言語としてみた場合、その大きな特徴は操作の一貫性を表現できることである。一貫性はユーザーにとってそのユーザーインターフェースの使いやすさを決める要因の一つである。従って、一貫性の表現ができるということは、その記述を用いてユーザーインターフェースの評価が行なえることを意味し、ひいては、ユーザーインターフェースの質の向上に役立てることができる。ここでは、この一貫性についてインフォーマルな定義を与え、それがどのように記述されるかを述べる。

操作の一貫性とは、アプリケーション内部において、物理的な操作方法とそのふるまいが構造的および意味的に規則的であり統一されていることをいう [6][9] 。一貫性のあるアプリケーションにおいて、ユーザーは、この規則を理解すれば、そのアプケーションのすべての操作方法を覚えていなくても、正しい操作をエラーなく行な

うことができる。さらに、このようなGUIの開発では、一貫性はユーザーのコマンドの解釈処理の実装を大幅に簡単化する可能性を持つ。なぜなら、規則性を用いて効率的かつ簡便な処理を用いることができ、統一されていることから、同じ処理ルーチンを再利用する機会が多くなるためである。

例えば、あるGUIエディターにおいて、画面上に表示されているいくつかのオブジェクトに対し、コマンドAまたはコマンドBを実行する操作を考える。コマンドAとコマンドBはそのアプリケーションのふるまいにおいてほぼ同等であるとする。このとき、オブジェクトを指定する操作とコマンドAまたはBを指定する操作を行なうが、コマンドAの場合のオブジェクト指定法(例えば、オブジェクトのアイコンをクリックするなど)とBの場合の指定法が同じ方法でなければ、一貫性があるとは言えない。同様に、コマンドの指定法も同じ規則でなければならない。コマンドAはコマンドのアイコンをマウスでクリックしなければ指定できないのに対し、Bの方は、メニューの選択でなければいけないというようなGUIの仕様はよくないと考えられる。その他にも、オブジェクトの指定とコマンドの指定の順や、そのときのフィードバック、各コマンドの実行のタイミングなど、一貫性をもたせるべき要因はたくさんある。

ここで、PDUIの用語を用いて操作の一貫性を定義すると、アプリケーション全体で、同じコンテキストでは物理操作は同じインタラクションスタイルを用いていること、である。特にコンテキストは、その物理操作にマッピングされたコマンドを構成するメソッドによって決定される。言い換えれば、アプリケーション全体でインタラクションスタイルが統一的に使われ、インタラクションスタイルとメソッドが一意的に束縛されていること、である。

このような一貫性がどのように記述されるかを、再び前述の画面上のオブジェクトを移動するというコマンドの操作を例に述べる。このときの3つのインタラクションスタイル: (1) オブジェクトの上にマウスポインタをもってくる、(2) マウスの右ボタンを押す、(3) ドラッグする、は、それぞれ次のように記述される。

```
(1)MoveMouseto(*Object)->*
(2)*->PressMouseButton(right)->*
(3)*->PressMouseButton(*button)
    ->MoveMouseto(*Target)
    ->ReleaseMouseButton(*button)
```

実際にFileというオブジェクトをPrinterオブジェクトへ移動するには、

```
MoveMouseto(File)
  ->PressMouseButton(right)
  ->MoveMouseto(Printer)
  ->ReleaseMouseButton(right)
```

という物理操作になるが、これは、インタラクションスタイル(1),(2),(3)に対し、

```
*Object=File, *button=right, *Target=Printer
```

とすれば一致する。

ここで、移動ではなく複写をするコマンドを考えると、対象オブジェクトの指定やそれに対する操作の方法は、移動でも複写でも同じであり、コンテキストは同じである。よって、一貫性を考えるとインタラクションスタイル(1)と(3)は複写コマンドの物理操作でも用いられるべきである。そこで、複写コマンドを意味するインタラクションスタイル((2)′マウスの左ボタンを押す)は、

```
(2)' *->PressMouseButton(left)->*
```

となり、これを加えるだけで、移動コマンドと同じような操作方法を持つ複写コマンドの物理操作が定義されたことになる。

すなわち、移動コマンドと複写コマンドの物理操作間で、マウスのボタンが右と左で異なるだけで他は一貫性のあるユーザーインターフェースが記述されたことになる。このようにして、あるアプリケーションのすべてのコマンドが記述されれば、操作の一貫性がある質の高いユーザーインターフェースが表現されたことになる。

3.2 部品化

ソフトウェアを効率よく開発するには、ソフトウェアの部品化をすすめ再利用を促進する必要がある。GUIの開発においてもよく使われるようなモジュールは、共通に使えるようにしておくと開発の手間を低減することができる。例えば、ウィンドウ部品やウィジェットと呼ばれるものは、その画面上の形態やユーザーからの操作に対するふるまいを部品化したものであり、押しボタンやテキスト入力の部品などが代表例である。

しかし、従来からあるこれらの部品は、表示に関する部分と簡単なイベント処理程度のインタラクションだけであり、より複雑なインタラクションの部品化とは言えない。さらに、操作の一貫性はまったく考慮されてないと言える。

PDUIでは、操作をインタラクションスタイルの集まりとして表現するので、複雑な操作も簡単なパターンの組み合わせとして表現され、同時に一貫性も考慮される。すなわち、このインタラクションスタイルを部品として用意しておけば、複雑なインタラクションが操作の一貫性を保ったまま再利用可能となる。フレームワークのところでも述べたようにインタラクションスタイルは有限状態オートマトンを内部に持つオブジェクトとして実装される。また、このオブジェクトはフレームワークの構成要素の一つであるので、部品としての独立性も高い。このため、PDUIを用いることによって、種々のインタラクションスタイルを表わすようなオブジェクト(クラス)を作っておけば、GUIの部品化が行なわれ再利用性が促進される。

4 まとめ

本稿では、GUIの対話処理を記述・実装するための枠組として、イベント駆動をさらにすすめたパターン駆動の概念を提唱し、その考え方にもとずいたユーザーインターフェース仕様記述言語と、GUIの実装フレームワークについて説明した。さらに、この枠組を用いることによって、操作の一貫性があるような質の高いユーザーインターフェースが表現でき評価・改良の役に立つことや、GUIを実装する上でインタラクションスタイルが部品化され、複雑な対話処理を行なうようなソフトウェア部品の再利用が促進されることを示した。

現在、PDUIの記述言語や実装フレームワークはライブラリとして実装を計画している段階であるが、その細部についてはまだ未完成な部分や問題点がある。

記述言語に関しては、インタラクションスタイルを複数指定して物理操作を規定する場合、必らずしも設計者の意図した操作になるとは限らないという問題点がある。すなわち、個々のインタラクションスタイルでは操作のパターンは明確であるが、これらがいくつか組み合わさったとき、本来不法な操作が可能になってしまったり、逆にパターンの間に矛盾が生じ本来可能な操作が不能になってしまうことがある。例えば、A->*とB->*いうパターンが同時に活性化される必要があるようにGUIの仕様を決めてしまった場合、明らかにこの2つのパターンは同時にマッチしないので、仕様に矛盾がおきる。また、A->*と*->BというパターンがまったくⅠ無関係のコマンドを起動し、かつ、この2つのコマンドは同時に実行されるべきでないとき、A->...->Bというアクション列がおきたときに動作が不正になる。

また、複数のメソッドが呼び出されるとき、それらのメソッドの呼び出し順が指定できないため設計者が予期できないようなバグを発生させるおそれがある。また、設計者自身でメソッドの呼び出しをコントロールしたい場合もあるが、現在はそのようなことを許していない。そのような記述やメカニズムを用意すると、記述法が繁雑になったり仕様に矛盾が生じるもとになる。

実装フレームワークに関しては、多くのインタラクションスタイルが存在する場合の実行効率が問題となることが予想される。これは、記述されたものをより効率的なコードにコンパイルすることを考えれば、ある程度は解決できると思われる。しかし、GUIアプリケーションの開発では、プログラムの変更に対して再コンパイルのターンアラウンドが短いことが要求される。従って、コンパイル方式よりも、再コンパイルなしにインタラクションスタイルなどの部品の構成が自由に変更できるようなフレームワークの方が望ましく、この実装フレームワークのままで実行効率を高めることが課題である。

PDUIは本稿で述べたようなユーザーインターフェースの仕様記述言語や実装フレームワークに用いられるのみではなく、最近注目されつつあるDemonstrational Interface[4]の実装の枠組として使える可能性もある。PDUIは操作の部分構造をパターンとして分解して表現するものであるので、ユーザーの行なった物理操作のアクション列から言語の構文解析のようにインタラクションスタイルを自動抽出することを行なえば、ユーザーの操作とその意図を推論することに用いることができる。これは、次回のユーザーの操作が行なわれるときにこの推論を用いて、より「知的」なユーザーインターフェースを実現することができる。

本研究の今後は、記述言語の詳細をより明確化させ実際のGUIが記述できるようにし、さらに、実装フレームワークはGUIライブラリとして実装を行ないその有効性を検証することである。

参考文献

[1] M. Green. The university of alberta user interface management system. In *SIGGRAPH'85*, pages 205–213. ACM, 1985.

[2] R. E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[3] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug./Sep. 1988.

[4] B. A. Myers. Demonstrational interafaces: A step beyond direct manipulation. *IEEE Computer*, 25(8):61–73, 1992.

[5] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *SIGCHI'92*, pages 195–202. ACM, 1992.

[6] S. J. Payne and T. R. G. Green. Task-action grammars: A model of the mental representation of task languages. *HUMAN-COMPUTER INTERACTION*, 2:93–133, 1986.

[7] APPLE PROGRAMMER'S and DEVELOPER'S ASSOCIATION. *MacApp: The Expandable Macintosh Application.* Apple, 1987.

[8] J. M. Vlissides and M. A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transaction on Information Systems*, 8(3):237–268, July 1990.

[9] 北山. ユーザー・インターフェース設計・評価のための形式的記述の枠組について. In 第*5*回ヒューマンインターフェースシンポジウム, pages 117–126. 自動計測制御学会, 1989.