Chapter 6

コンポーネント差分プログラミング

ホワイトボックス型の再利用は、サブクラス化により柔軟な拡張が可能であり、Nutsにおける主要な拡張手段となっている。しかし、サブクラス拡張では、クラスの実装を継承するため、クラスの脆弱性や多重継承の問題がある (第2.6節参照)。

本章では、サブクラス拡張を補うとともに、その問題点を回避できる拡張手段として、コンポーネント差分プログラミングと呼ぶ手法を提案する。コンポーネント差分プログラミングとは、クラス定義に戻ることなしに、コンポーネント単位で拡張機能との差分を追加／削除／変更が行える拡張手法を言う。

さらに本章では、コンポーネント差分プログラミングをNutsのフレームワーク上で実現する特殊なコンポーネント (ベクターコンポーネント) の実装について説明する。

6.1 コンポーネント差分プログラミングとは

コンポーネント差分プログラミングとは、拡張機能の差分だけを含んだコンポーネントを既存のコンポーネントに結合させることで拡張する手法である。この方法の利点として以下のものが挙げられる。

1. 拡張機能もコンポーネントとして分離されているため、それ自体の再利用が可能である。

2. 拡張機能をコンポーネントとして扱えるため、プログラム中への追加／削除／変更が簡単に行える。

3. 拡張機能と被拡張コンポーネントのクラス階層が別になるため、被拡張コンポーネントのクラス階層を熟知していなくとも、拡張機能の開発が可能である。

4. 多重継承によらずに拡張機能の多重化が可能である。

コンポーネント差分プログラミングが適している一例として、アイコンに他のアイコンをドラグ&ドロップする機能を追加する場合を取り上げる。

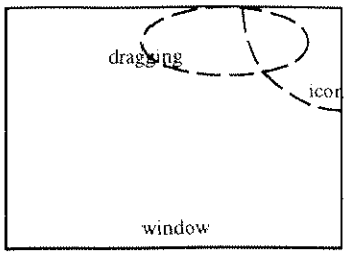

図 6.1: オブジェクト指向によらない拡張

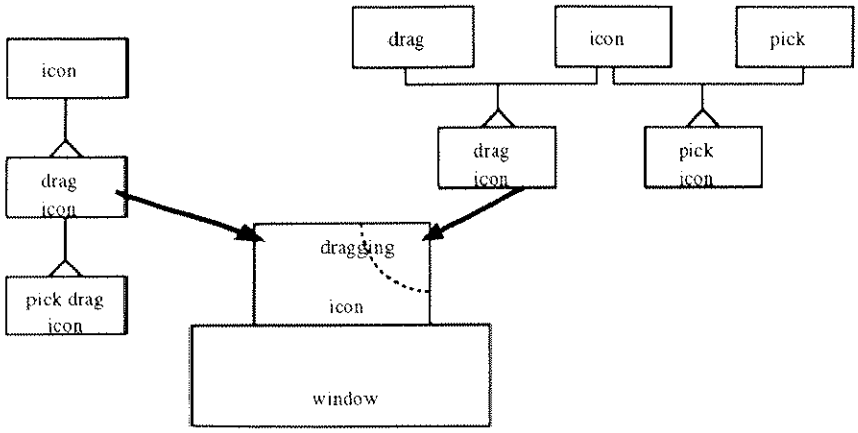

図 6.2: 従来のオブジェクト指向による拡張

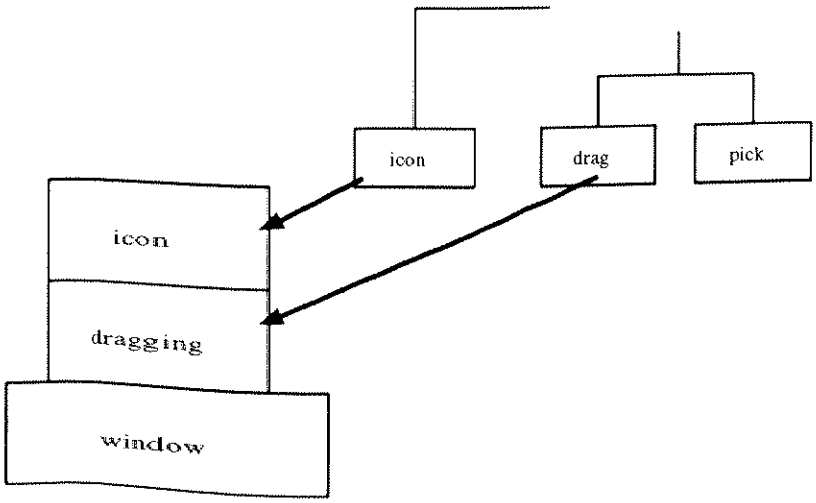

図 6.3: ベクターによる拡張

オブジェクト指向以前のプログラミングでは、ウインドウとアイコンの区別がなく、アイコンやそのドラグ機能は、ウインドウの一機能としてウインドウ中に包含されていた(図6.1)。この方法では、アイコンとドラグ機能をウインドウから分離するのが困難であり、ウインドウ、アイコン、ドラグ機能、それぞれの再利用が困難である。

これに対して、オブジェクト指向プログラミングでは、アイコンオブジェクトを導入し、アイコンがウインドウから分離される。アイコンにドラグ機能を追加したい場合は、アイコンのサブクラス化(ドラグアイコンクラス)を行う(図6.2)。しかしこの方法では、ドラグ機能自体をアイコンクラスから分離して再利用することはできない。このときのドラグ機能はアイコンのためのものであり、ウインドウのドラグには再利用できない。さらに、サブクラス拡張では、他の拡張機能(例えばアイコンを選択可能にする)と組み合わせたい場合に、拡張機能の種類の組合せの数だけサブクラス化を行う必要があり、クラス数の爆発を起こす。

他の方法として、ドラグのような拡張機能を別のクラス階層で定義し、ドラグアイコンをアイコンクラスとドラグクラスの多重継承で行う方法がある(図6.3)。この方法では、第1.1節で挙げたような多重継承の問題がある。この問題を回避するために、実装の多重継承によらずに、インタフェースの多重継承を行う方法では、継承の度に実装を行う必要があり、再利用の効果が損なわれる。

これに対して、コンポーネント差分プログラミングでは、以下のようにアイコンを拡張することができる(図6.3)。

- ドラグ機能をコンポーネントとしてアイコンとは別のクラスで定義する(利点の3.)。
- ドラグ機能自体がコンポーネントとしてアイコンから分離するため、ドラグ機能自体を再利用できる(利点の1.)。
- ドラグ機能がコンポーネント化されているため、コンポーネントの結合で簡単にアイコンを拡張できる(利点の2.)。
- さらに他の拡張機能(例えばアイコンの選択)を追加したい場合は、それらの拡張コンポーネントを順次アイコンに結合することで行える(利点の4.)。

このように、コンポーネント差分プログラミングを用いることで、プログラム部品(ウインドウ、アイコンやドラグ機能)の再利用性と拡張性を高めることができる。しかし、従来のオブジェクト指向をベースとした開発フレームワークでは、第1.1節で挙げたような理由から、コンポーネントベースの差分プログラミングをサポートしていない。デザインパターンカタログ[24]では、部品に修飾を施す手法としてDecoratorパターンを取り上げているが、修飾機能を汎用的なコンポーネントして扱うまでには至っていない(Decoratorパターンとコンポーネント差分プログラミングの差異については第9.2節参照)。

Nutsでは、構造モデル／制御モデルの枠内でコンポーネント差分プログラミングを実現するためのコンポーネントを導入し、ベクターコンポーネントと名付けた。以下でベクターコンポーネントを用いた、Nutsのコンポーネント差分プログラミングについて説明する。

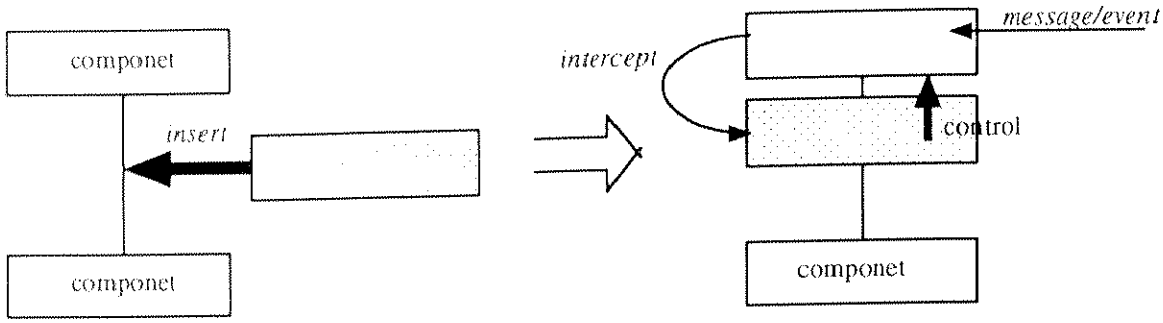

図 6.4: コンポーネント差分プログラミングの拡張手法

6.2 ベクターコンポーネント

Nuts における、コンポーネント差分プログラミングは、基本的に以下のように行う。

1. 透明でどこにでも挿入可能なコンポーネントを用意し、その中に拡張機能の差分をカプセル化する。

2. プログラム中の修飾したいコンポーネントに上記の透明コンポーネントを結合させる。このとき透明コンポーネントは、どのような位置にも挿入可能であり、かつ挿入することによってプログラム構造に影響を及ぼさない。

3. 上記の透明コンポーネントは、結合したコンポーネントへのメッセージを横取りし、そのコンポーネントになりすます。その上で、修飾に関係のあるメッセージに対して、拡張機能に基づいた処理を行う。

このような、コンポーネント差分プログラミングを実現するには、以下のような拡張機能の運搬を担うためのコンポーネントが必要である。

- 透明であること—どこにでも侵入できるため
- 拡張機能に相当する部分の制御を横取ること—他の部品になりすますため

Nuts では、このようなコンポーネントをベクターコンポーネントと呼ぶ。ベクターコンポーネントは、透明コンポーネントとすべての Nuts コンポーネントに備わっているメッセージ横取り機能を組み合わせることで実現できる。具体的には以下のように行う。

ベクターコンポーネントの作成 透明コンポーネントクラス (NutsShadowManager クラス) から派生したベクターコンポーネントのクラスを定義する。このベクターコンポーネントは、透明コンポーネントの一種であり、被修飾コンポーネントとその親コンポーネントとの間に自在に挿入できる。

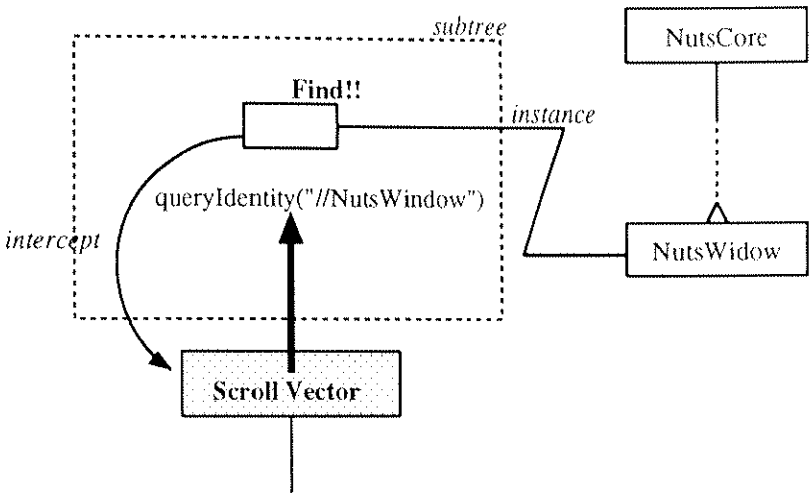

図 6.5: 被修飾コンポーネントの探索

ベクターコンポーネントの結合 ベクターコンポーネントは、それが修飾するコンポーネントの種類 (クラス) が決まっている。そのため、ベクターコンポーネントは、挿入位置を根とする部分木中で修飾対象のクラスに属するコンポーネント (群) を探索し (図 3.2 の child 方向探索による)、そのコンポーネントのメッセージを自動的に横取りする。

例えば、ウインドウにスクロール機能を付加するベクターコンポーネントは、ウインドウコンポーネントに対する拡張機能を持っている。従って、この場合のベクターコンポーネントは、図 6.5に示すように、"//NutsWindow"(表 3.1参照) をキーとした child 方向への queryIdentity() 探索によって、ウインドウクラスのサブクラスからなるコンポーネントを修飾相手として認識し、そのメッセージを横取りするように設定する。

探索により修飾相手の認識を行う理由は、ベクターコンポーネントを多段に積み上げて多重の修飾を行おうとした場合に、被修飾コンポーネントが自身の直接の子コンポーネントになるとは限らないためである。

ベクターコンポーネントによる拡張 ベクターコンポーネントの結合により、被修飾コンポーネントへのメッセージ (イベントを含む) は、すべてベクターコンポーネントに横取りされる。メッセージを横取りしたベクターコンポーネントは、メッセージのオーナー (すなわち本来のメッセージ受信者) を修飾する。先の例では、ウインドウのスクロールに関するマウスのイベントをベクターコンポーネントが横取りし、ウインドウにスクロール機能を付加する。

また、ベクターコンポーネントが多段に積み重ねられている場合でも、透明コンポーネントはメッセージ横取り時にメッセージのオーナーを変更しないので、どの段のベクター

コンポーネントでも、メッセージのオーナーを修飾するという統一的な方法で修飾が可能である。

6.3 NutsVector クラス

ベクターコンポーネントを、NutsVector クラスで定義する。NutsVector クラスは、NutsShadowManager のサブクラスであり、その差分としてベクターコンポーネントに必要な要件を追加している。

6.3.1 NutsVector クラスの初期化フェーズ

NutsVector クラスではその初期化フェーズのmanage() 中で、特定の被修飾クラスのコンポーネントを探し、そのコンポーネントへのメッセージを自身が横取りするように設定する。実際には、以下のように記述している。

```
void NutsVector::manage()
{
  super::manage();

  NutsCore *obj;
  char      buffer[255];

  sprintf(buffer,"//%s",_decorateClassName());
  //char *_decorateClassName():
  //被修飾クラスのクラス名を返す pure virtual 関数

  if(obj = this->queryIdentity(this,buffer))
       obj->receiveMessage("pass to",this);
}
```

すなわち、ベクターコンポーネントは、自身がルートである部分木中から被修飾クラスのサブクラスに一致するコンポーネントを自身を探索開始点として、queryIdentity() メソッドで探索する。次に、見つかったコンポーネントに対して、横取り開始メッセージを送り、自身がメッセージを横取りするように設定する。被修飾コンポーネントへの横取り開始メッセージの入力ポートとしてreceiveMessage() を使っており、横取り開始メッセージ自体が横取り対象となっている。従って、すでに被修飾コンポーネントでメッセージの横取りが設定されている場合は、多重の横取りが設定されるため (第 4.1.6節参照)、ベクターコンポーネントによる多重修飾が可能になる。

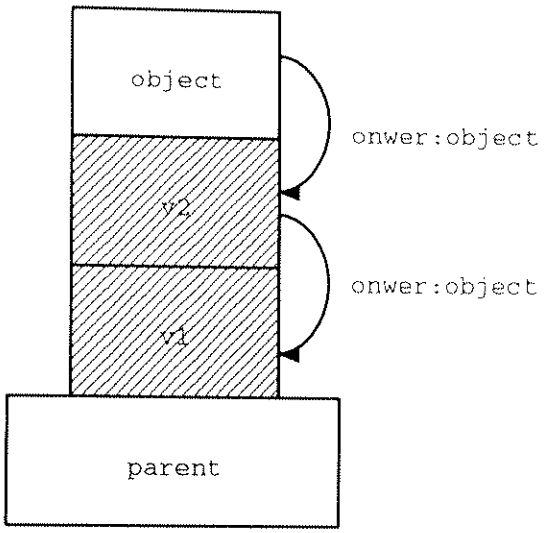

図 6.6: ベクターによる多重修飾

6.3.2 NutsVector クラスでのメッセージ処理

横取りしたメッセージは、ベクターコンポーネントのrecognizeMessage() で受け取り、ここに横取りしたいメッセージのハンドラを記述する。このときrecognizeMessage() の引数として、本来メッセージを受け取るはずのコンポーネント (被修飾コンポーネントへの参照) がメッセージのオーナーとして渡る。よって、ベクターコンポーネントは常にメッセージのオーナーに対して修飾を行えばよい。ベクターコンポーネントを多段に積み重ねた場合も、透明コンポーネントの一種であるベクターコンポーネントは、メッセージ横取りの際にメッセージのオーナーを変更しないので、いずれの段でもメッセージのオーナーが被修飾コンポーネントに一致する。

ベクターコンポーネントを多段に積み上げた場合のメッセージの流れは以下のようになる (透明コンポーネントの仕様 第5.3.2節参照)。

- メッセージは最後にメッセージを横取りしたベクター (通常最も下の階層のベクター、図 6.6のベクター v1) が最初にメッセージを受け取る。透明コンポーネントの仕様に従い、ベクターによるメッセージの横取り過程では、メッセージのオーナーは変更されていないので、オーナーは本来のメッセージの受信者のままである。

- 図 6.6のベクター v1 での処理が終ると、次に同図ベクター v2 がメッセージを受理する。透明コンポーネントの仕様に従い、ベクターは横取り先でのメッセージ解釈の可否に関わらず、メッセージを受理する。

以上のようにベクターは、いずれの段も同じ条件 (メッセージ受理の機会とオーナーにおいて) であり、ベクターを多段に積み上げた場合も一段の場合と同様に機能する。

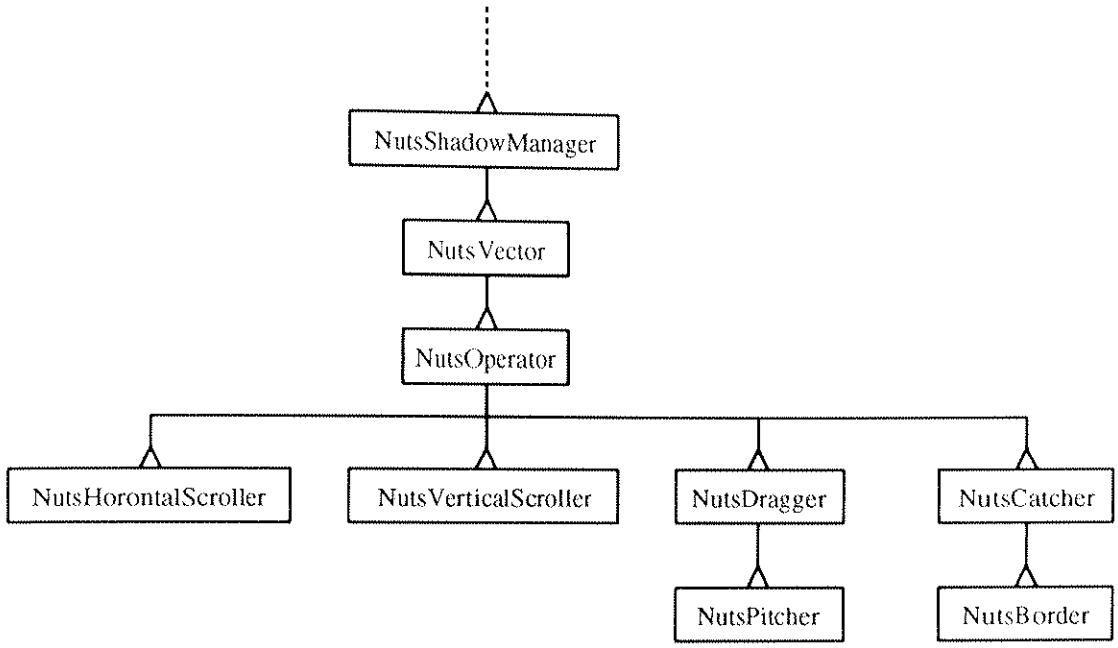

図 6.7: 透明コンポーネントのクラス階層

6.4 ベクターコンポーネントの例

例として、ウインドウコンポーネント (NutsWindow クラス) に通知されるマウスの press, motion, release の各イベント (Nuts のイベントとして実装) を横取りし、GUI の修飾を行うベクターコンポーネント群を取り上げる。

まず、そのようなベクターコンポーネントのベースクラスとして NutsOperator クラスを定義する (図 6.7)。NutsOperator クラスは NutsVector の派生クラスであり、部品木中のどこにでも挿入可能で、それが結合したウインドウコンポーネントへのマウスイベントを横取りする。また NutsOperator は、GUI の修飾に必要な種々の情報 (ウインドウの表示位置やマウスポインタの位置など) を保持する変数を持つ。

このように、NutsOperator はベクターコンポーネントなので、GUI 部品のクラス階層とは独立したクラスによって、GUI 部品の修飾に必要な機能を実現し、それを GUI 部品に付加できる。

6.4.1 ウインドウをスクロールするベクター

NutsWindow クラスはマウスの press, motion, release の各イベントを受け取るウインドウクラスである。一方、NutsHorizontalScroller は、NutsOperator のサブクラスであり、NutsWindow コンポーネントに水平方向のスクロール機能を追加するベクターコンポーネントである。NutsHorizontalScroller ベクターを NutsWindow コンポーネントとその親ウインドウコンポーネントの間に挿入することで (図 6.8左)、NutsWindow コンポーネント

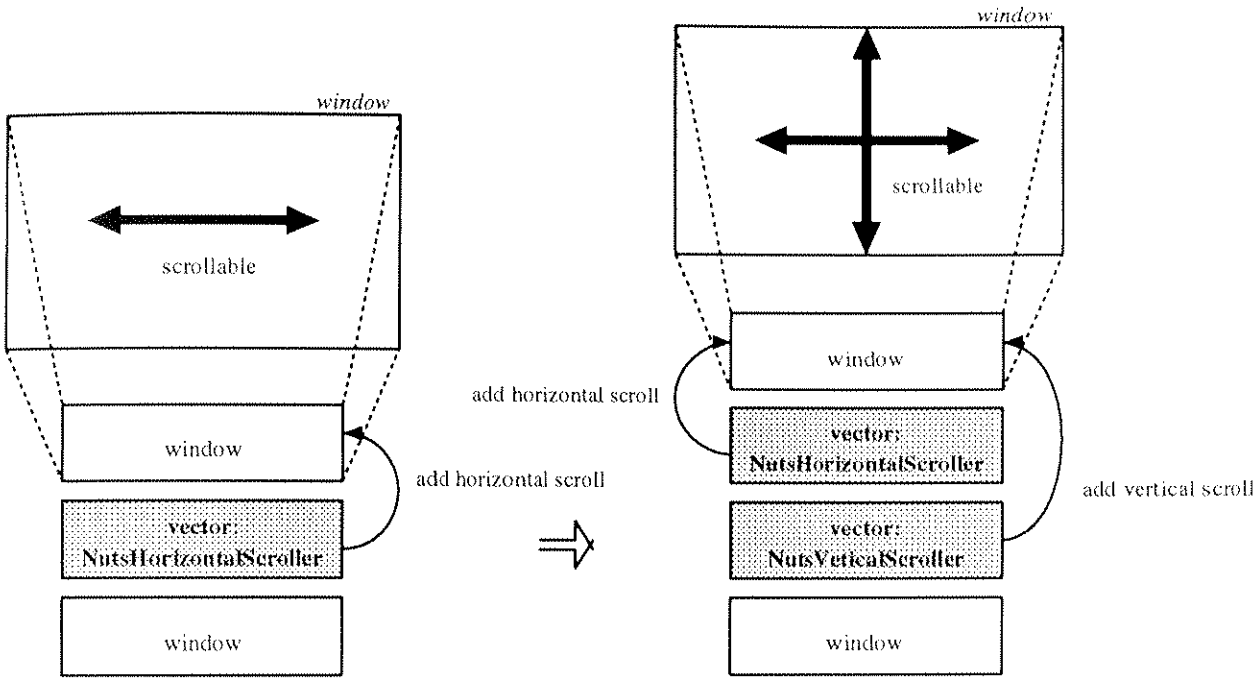

図 6.8: スクロール機能の追加と多重化

にスクロール機能が付加される。

この場合、NutsHorizontalScroller ベクターは、NutsOperator クラスの機能として、NutsWindow コンポーネントへのメッセージをすべて横取りする。その中で、マウスの移動を示す motion イベントのみのハンドラを独自定義 (オーバーライド) しており、マウスの水平方向の移動を検知して、横取りしたメッセージのオーナーである NutsWindow コンポーネントの内容を水平スクロールさせる。

NutsHorizontalScroller のソースコードを図 6.9に示す。実際にマウスがドラグ中かどうかは、NutsOperator クラスがマウスの press,release のイベントを横取りして判断するので、そのサブクラスである NutsHorizontalScroller では、図 6.9のようにドラグ中の motion イベントだけを記述すれば良い。

同様の手法で、垂直方向にスクロールさせるベクター NutsVerticalScroller を開発できる。さらに、NutsHorizontalScroller と NutsVerticalScroller を積み重ねて多重に修飾することで、水平垂直方向に加えて、斜め方向へのスクロールが可能になる (図 6.8右)。

6.4.2 アイコンをドラグする／投げるベクター

NutsIcon クラスは NutsWindow のサブクラスであり、フォントやビットマップなどを用いてウインドウ上のアイコンを表現するクラスである。

NutsDragger は NutsOperator のサブクラスで、NutsIcon をドラグするベクターコンポーネントである。NutsDragger ベクターをアイコンとその親である NutsWindow コンポーネ

```
#include <NutsHorizontalScroller.h>

void NutsHorizontalScroller::
motion(NutsWindow *child,XEvent *e)
{
  if(!_dragged(child))return;

  int dx,x,width,w,h;  Window wind;

  dx     = e->xbutton.x - _xpos(child);
  w      = _width(child);
  h      = _height(child);
  wind   = _wind(child);
  width  = w - abs(dx);

  if(dx > 0)x = 0;
  else      x = -dx;

  XCopyArea(display(),wind,wind,_gc,x,0,width,h,x+dx,0);
  if(dx > 0)
   XClearArea(display(),wind,0,0,dx,h,False);
  else
   XClearArea(display(),wind,w+dx,0,-dx,h,False);
  _xbase(child) -= dx;
  _refresh(child);
  _xpos(child)   = e->xbutton.x;
}
```

図 6.9: NutsHorizontalScroller

ントの間に挿入するとアイコンをドラグできるようになる (この場合もアイコンが選択されたことは NutsOperator クラスで検知するので、NutsDragger ではドラグ動作だけを記述している)。

さらにアイコンをドラグするだけでなく、途中で離して「投げる」ようにもできる [69]。NutsDragger のサブクラスである NutsPitcher ベクターに修飾されたアイコンは、それが選択されてから離されるまでの時間と距離に応じた初速度で離された方向に投げ出される。NutsPitcher ベクターは、アイコンが飛んでいる間、その兄弟／親コンポーネントに対して定期的に移動メッセージを送る。

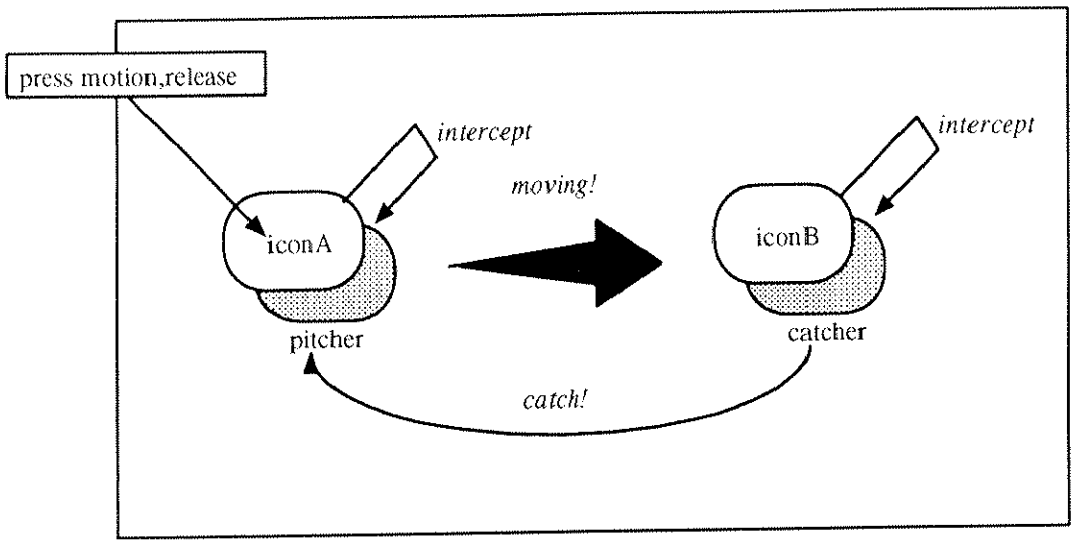

図 6.10: アイコンを投げる場合の制御

6.4.3 アイコンを捕獲するベクター

NutsPitcher ベクターの移動メッセージに反応してアイコンを「捕獲」するのが NutsCatcher ベクターである。

NutsCather ベクターは自身が修飾しているアイコンの位置と、NutsPitcher ベクターが送信する移動メッセージに含まれる移動アイコンの位置が一致したと判断すると、移動中のアイコンに対して捕獲メッセージを送る。捕獲メッセージは、NutsPitcher ベクターに横取りされ、停止処理が行われる (図 6.10)。捕獲メッセージに質量、弾性係数などを含めると (跳ね返り等) アイコン同士の多彩な衝突を模擬できる。

NutsBorder は、NutsCatcher のサブクラスであり、NutsCatcher がアイコンとの衝突を監視するのに対して、NutsBorder はその上に乗っているウインドウの縁とアイコンの衝突を監視する。NutsBounder の捕獲メッセージを質量無限大の完全弾性衝突にすると、投げられたアイコンがウインドウの縁で「跳ね返る」ようになる。

6.4.4 ベクターの挿入

ウインドウ (NutsWindow) とアイコン (NutsIcon) の間に、構造記述ファイルを用いて、前記のアイコンを投げるベクター (NutsPitcher)、アイコンを捕獲するベクター (NutsCatcher) を挿入する方法を示す。挿入前の構造記述ファイルは以下のようになっている。

```
newNutsWindow(parent,window);
newNutsIcon(window,icon);
```

これに対して、ベクターの挿入は以下のように行う。

```
newNutsWindow(parent,window);
newNutsPitcher(window,pitcher);
newNutsCatcher(pitcher,catcher);
newNutsIcon(catcher,icon);
```

このように、ベクターの挿入は構造記述ファイルで容易に指定できる。

6.5 ベクターを用いたアプリケーション開発

ベクターコンポーネントを用いて開発した、アイコン投げシェル [68] を模したGUIシェル (図6.11) について説明する。アイコン投げシェルとは、従来のアイコンのダブルクリックなどを主体とするデスクトップ環境の代わりに、アイコンのドラグ&ドロップによるアクションの起動を可能にするGUIシェルである。さらに、アイコン投げシェルでは、アイコンを投げて他のアイコンにぶつけることにより、ドラグ&ドロップより迅速なアクションの起動を実現する。具体的にこのGUIシェルは以下の機能を持つ。

- 画面上の任意の点をドラグすることで画面がスクロールする。
- アイコンが投げられて別のアイコンに捕獲された時、両方のアイコンの種類に応じた動作を起動する (例えば、コマンドアイコンを実行アイコンに投げつけることでコマンドを起動する)。

これらの動作を、画面スクロール、アイコン投げ、アイコン衝突などの拡張機能をベクターコンポーネントに組み込み、図6.12(a) の構造に対して図6.12(b) のように挿入することにより実現できた。すなわち、アイコン投げシェルの機能を持たない図6.12(a) のプログラムに対して、図6.12(a) の部品群とは別個にアイコン投げシェルの拡張機能だけを持った部品群を開発し、挿入するだけでプログラム全体としての拡張が実現している。

アイコン投げシェルの開発において、Nutsを用いた場合と用いない場合の開発効率の比較については第9.1.1節で述べる。

6.6 考察

本章では、コンポーネント差分プログラミングの仕組みと、それをNutsで実現するベクターコンポーネントの実装について説明した。また、ベクターコンポーネントを用いた実際のコンポーネント差分プログラミングの例を示した。

サブクラス拡張による柔軟なコンポーネントの拡張が可能なホワイトボックス型コンポーネントNutsと、コンポーネント差分プログラミングを組み合わせることにより、柔軟性と開発効率の高さを両立したプログラミングスタイルが提供できる。

アイコン投げシェルの開発において、コンポーネント差分プログラミングを用いた場合とそうでない場合の開発効率の比較について、第9.1.1節で述べる。また、コンポーネント

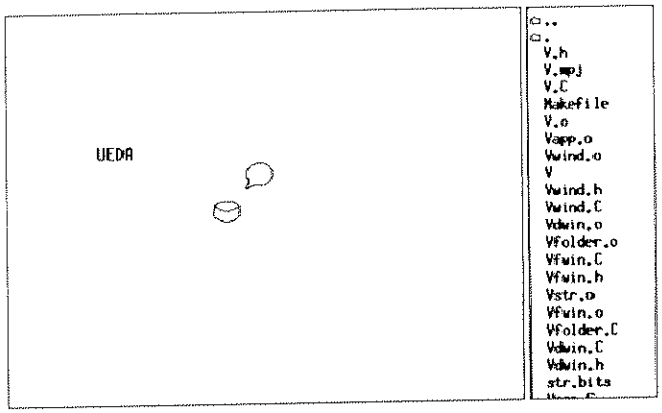

図 6.11: GUI シェル

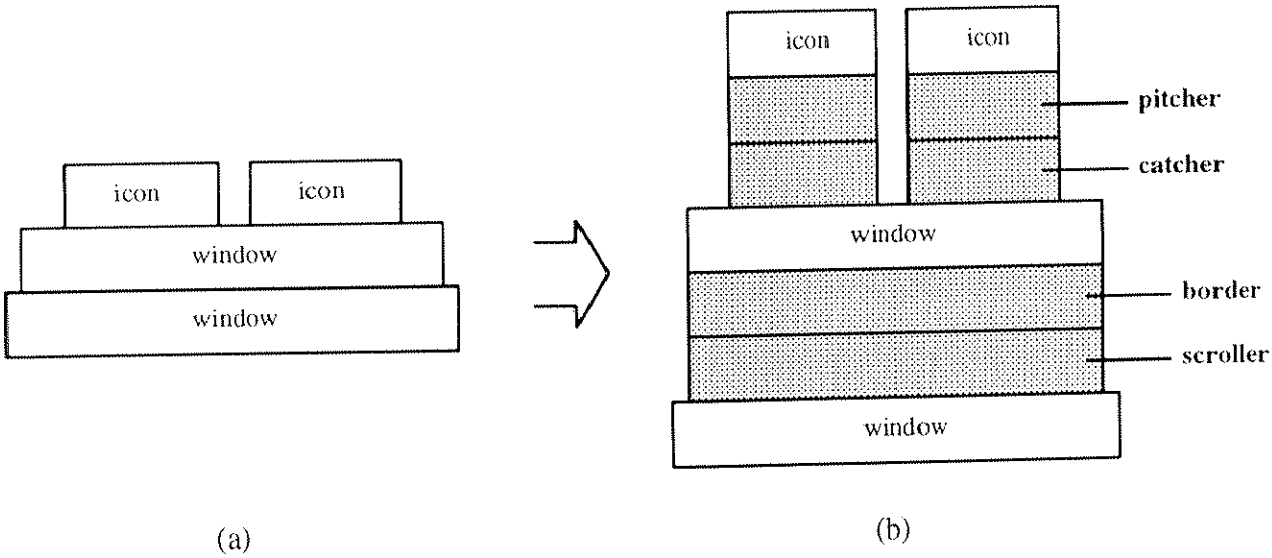

図 6.12: ベクターコンポーネントによる GUI シェルの実装

差分プログラミングと似た手法として、デザインパターンの Decorator パターンが挙げられるが、それとの差異については第 9.2.1節で述べる。