Chapter 9

議論

第7、8章で、Nutsによるアプリケーション開発の実施例について述べたが、本章では実施例をもとに、開発効率の観点からNutsの効果を評価する。

次に、デザインパターンカタログ [24] に示されている様々なオブジェクト連係の手法を取り上げ、それらをNutsのコンポーネントアーキテクチャやコンポーネント差分プログラミングと比較する。さらに、既存の開発フレームワークとしてET++[5]、IntelligentPad [88] を取り上げ、それらとの比較に基づいて、Nutsの利点を議論する。また、リフレクション機能を利用した拡張手法 (OpenC++[11]) と比較する。

9.1 Nutsによるソフトウェア開発効率の向上に関する議論

本節では、アイコン投げシェル (第6.5節参照) と配送計画支援システム Navimos(第8章参照) の開発を例に、開発効率の観点からNutsの効果について議論する。

9.1.1 アイコン投げシェルの開発におけるNutsの効果

アイコン投げシェル (図9.1) の基本機能の開発において、Nutsを用いた場合とそうでない場合を比較する。基本機能とは、以下を指す。

- ウインドウは、アイコン投げウインドウとディレクトリ表示ウインドウからなる。
- 各ウインドウはウインドウ上の任意の点のドラグによりスクロールできる。
- アイコン投げウインドウは上下左右の方向に、ディレクトリ表示ウインドウは上下方向にのみスクロールできる。
- アイコン投げウインドウの上にアイコンを配置できる。
- アイコン投げウインドウの上に置かれたアイコンはマウスによって投げることができる。

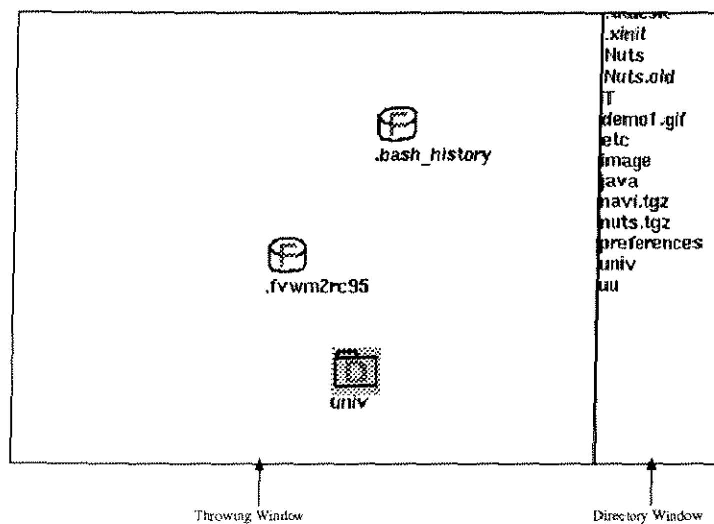

図 9.1: アイコン投げシェルのイメージ

- 投げられたアイコンはアイコン投げウインドウの縁まで来ると止まる、あるいは跳ね返る。

以上の基本機能を実現するプログラムを Nuts を用いる場合と用いない場合で実装した。Nuts を用いる場合の実装は著者が、そうでない場合の実装は、Nuts は知らないがオブジェクト指向を熟知した C++プログラマが行った。

開発データの比較を表 9.1に、具体的なクラス名を表 9.2に示す。表 9.1中、隣接した部品とは、ある部品が直接他の部品への参照を保持している場合を指す。また、表 9.2中の数字は、コンパイル時に知っている必要がある他クラス定義の数である。

Nuts を用いた場合は、Nuts を用いない場合に比べてプログラム行数にしておよそ 48%となっている。また全部品種中で Nuts における再利用可能部品の割合は、58%であったが、Nuts を用いない場合では、20%に留まっている。

まず、Nuts と Nuts を用いない場合のプログラム構造の理解容易性を比較する。Nuts によって実装されたプログラムは、構造記述ファイル (図 9.2) でプログラム構造 (図 9.3) が容易に把握できる。これに対して、Nuts を用いていない場合では、ソースコードの解析により図 9.4のような構造であることが判明したが、全体構造を把握するにはかなりの労力を要した。保守性の観点からプログラム構造の理解が容易に行えることが重要であり、Nuts が保守性に優れたフレームワークであると言える。

次に、上記の結合関係を構築するためのコストについて比較する。Nuts を用いない場合には、20 箇所の部品間結合が構築されている。これらの部品間結合の構築は、第 2.1節で示したようなさまざまな参照関係構築のための手法を用いて行う必要があり、コストがかかる。一方で、Nuts の場合も、11 箇所の部品間結合があるが、実際にはこれらの結合は、Nuts の構造モデルに従って自動的に構築されるため、部品間結合のためのコストが発生し

表 9.1: アイコン投げシェルの開発におけるデータ比較

項目	Nuts	非 Nuts
プログラム行数	648	1384
開発工数 (日数)	2	4
部品種数	10	10
再利用可能部品種数	7(58%)	2(20%)
部品間の結合数	11	20
隣接した部品間のメッセージ通信数	0	20
隣接していない部品間のメッセージ通信数	8	3
機能拡張のための部品数	5	0

表 9.2: アイコン投げシェルのクラス (数字は、コンパイル時に知っている必要がある他クラス定義の数)

Nuts		非 Nuts	
Vapp	0	TshApp	5
Vdwin	0	Dwin	2
Vfwin	0	Twin	3
Vfolder	0	TshWin	4
Vstr	0	Tset	1
Vwind	0	Eelt	2
NutsCatcher	0	Npict	0
NutsPitcher	0	Tmodel	0
NutsXScroller	0	Pool	2
NutsXVerticalScroller	0	PathName	0
NutsXBounceBorder	0		
NutsXRootWindow	0		

```
newVapp(V);
newNutsXRootWindow(V,root);
newVwind(root,wind);

newNutsXHorizontalScroller(wind,fwin_sc);
newNutsXVerticalScroller(fwin_sc,fwin_sc2);
newNutsXBounceBorder(fwin_sc2,fwin_bd);
newVfwin(fwin_bd,fwin);//アイコン投げウインドウ

newNutsXVerticalScroller(wind,dwin_sc);
newVdwin(dwin_sc,dwin);//ディレクトリ表示ウインドウ
```

図 9.2: Nuts を用いたアイコン投げシェルの構造記述ファイル

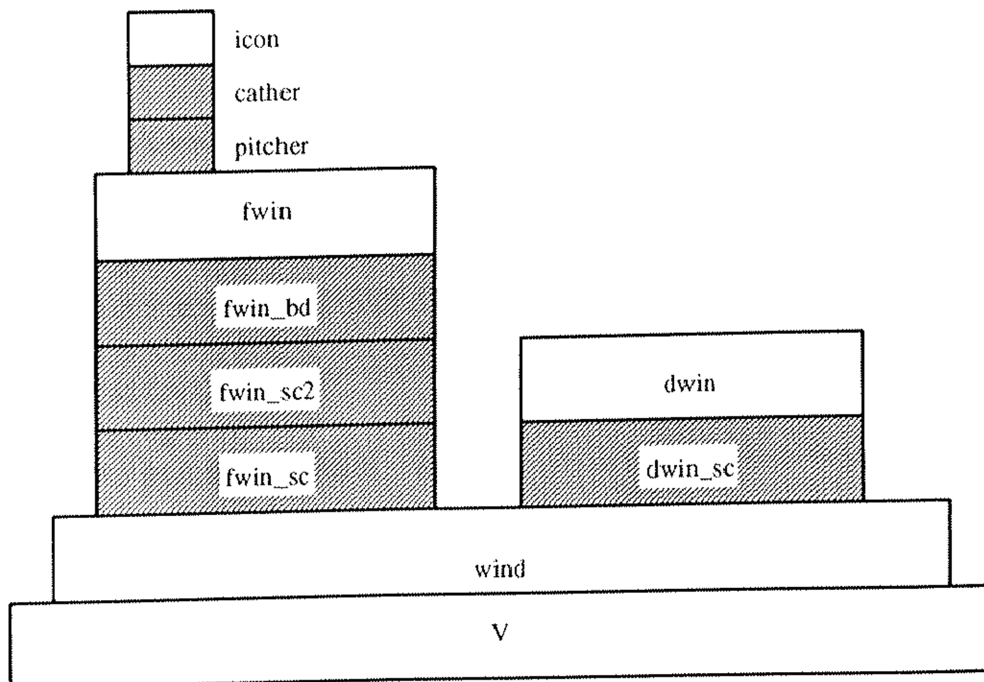

図 9.3: Nuts を用いたアイコン投げシェルの構造

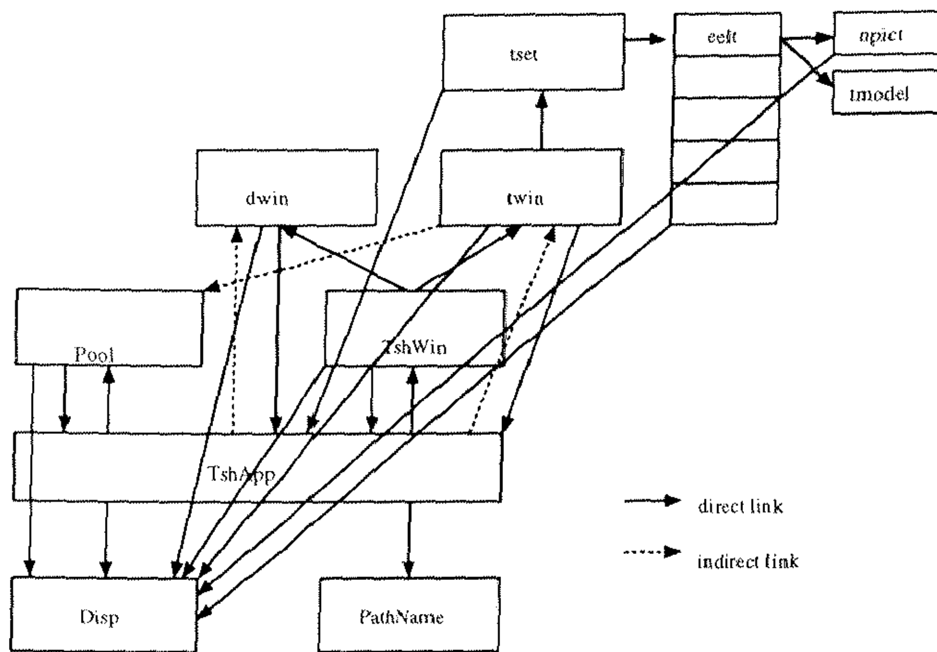

図 9.4: Nuts を用いない場合の構造

ない。

さらに、隣接していない部品間のメッセージ通信のためのコストを比較する。Nutsを用いない場合、3箇所で隣接していない部品間でのメッセージ通信が必要になっている。このようなメッセージ通信は、第2.1節で示したようなさまざまな参照取得のための手法を用いて行う必要があり、複雑さの原因となっている。一方、Nutsの場合も、8箇所で隣接していない場合のメッセージ通信が必要となっているが、これらはNutsの制御モデルを用いた動的探索によって行われるので、通信相手の参照を取得するために個別の参照獲得手法を実装する必要はない。

Nutsを用いない場合のプログラムでは、オブジェクト指向にもとづいて部品単位に分割されてはいるが、部品間の結合関係、およびメッセージ通信において部品同士が密接に関係しており、部品の再利用や、プログラムの一部の仕様変更が困難である。一方、Nutsでは、それぞれの部品の独立性が高く、部品同士が緩やかに結合しているため、部品を抜き出して別のプログラムで再利用したり、プログラムの一部の部品を別の部品に置き換えることなどが容易に行える。

最後に、ウインドウを「スクロールする」、アイコンを「投げる」といった拡張機能の再利用性について比較する。Nutsを用いない場合では、それぞれの機能はウインドウとアイコン部品の中に含まれている。これらの機能は、ソフトウェア設計当初から想定されたものであり、ウインドウやアイコンの実装時に最初から反映されている。一方、Nutsを用いる場合では、これらの機能がベクターコンポーネントとして実装されており、機能の追加が容易に行える。Nutsを用いた場合のアイコンは、投げられることを前提としていないア

表 9.3: アイコン投げシェルの開発における比較

項目	Nuts	非 Nuts
理解容易性	構造記述ファイルにより容易	ソースコードの解析必要
結合コスト	構造記述ファイルの記述により自動的に木構造が構築される	必要な結合関係をその都度構築する必要がある
通信コスト	制御モデルにより隣接していない部品間での通信も容易	通信が必要な部分に結合関係を作るか、参照が入手できる方法を用意する必要がある
再利用性	部品同士は緩やかに結合しており、単独で抜き出して再利用可能	他の部品と密接に関係しており再利用困難
拡張性	ベクターコンポーネントを用いた差分プログラミングにより拡張可能	ソースコードを解析しクラス階層を理解した上で、サブクラス化により拡張する必要がある

イコンに対し、後から投げる機能を追加できた。このように、Nuts を用いた場合は実装のより後の段階まで設計の自由度を維持できる。結果的に Nuts では、設計当初には予想し得なかった拡張機能を追加する場合も、低いコストで実現できる。

簡単な例として、ディレクトリ表示ウインドウを水平方向にもスクロールできるに拡張する場合を考える。Nuts を用いない場合は、ディレクトリ表示ウインドウの垂直スクロールに関する部分を検索し、そのコードを解析した上で水平方向スクロールも可能なように改良しなければならない。これには熟練した C++のプログラマでも数分かかる。一方、Nuts を用いた場合は、ディレクトリ表示ウインドウとは無関係に (ディレクトリ表示ウインドウ自体はスクロールされることを前提に作られていない)、その親部品として水平方向スクロールベクターを挿入するだけで拡張が可能であり、ベクターを用いない場合より短時間で実現できる。(図 9.2の下から 2 行目に`newNutsXHorizontalScroller(dwind_sc,dwin_sc2);`を挿入する)。ベクターを用いる方法は、Nuts を用いない場合と比較して、拡張のためのコストが低いと言える。

さらに重要な点は、拡張のための機能が再利用できることである。例えば、上記の例で、Nuts を用いない場合の水平方向スクロール機能の拡張は、ディレクトリ表示ウインドウのためだけのものである。一方で、Nuts を用いる場合は、水平スクロールベクターが存在しなかったとしても、水平スクロールベクターを一度開発しておけば、別の用途 (ディレクト

リ表示ウインドウ以外のスクロール) にも再利用可能となる。

以上の比較結果を表 9.3にまとめる。この表からも、Nuts を用いない場合に比べて、Nuts を用いた場合の開発効率は高いと言える。

9.1.2 Navimos の開発における Nuts の効果

第 8章で述べたように、巡回配送のための配送計画支援システム Navimos は、Win32/MFC 版と Unix/Nuts 版が存在する。開発はともに著者が行った。本節では、開発効率の観点で両者を比較する。開発データから両者を比較した結果を表 9.4に示す。この表は、両者の性能がほぼ同じになるようにデータを調整した結果である (Win32/MFC 版から Unix/Nuts 版には含まれない機能の開発に関わる部分を削除した)。Nuts 版の方が、プログラム行数や開発工数の点でかなり低コストで開発されていることがわかる。

比較結果を表 9.5にまとめる。全体構造の把握という点では、Win32/MFC 版は、MFC が提供するドキュメントビューアーキテクチャ以外の構造はプログラマに任されており、統一性がない。したがって、全体構造の把握はソースコードの解析によらねばならず、手間がかかる。一方、Unix/Nuts 版は、構造記述ファイルによって全体構造が容易に把握できるため、保守や将来的な拡張にも迅速に対処できる。

次に、地図上の配送ノードのアイコンをドラグする拡張機能を追加する場合を比較する。Win32/MFC 版では、ブラックボックスである地図表示部品 (map.ocx) を使用した。この部品は地図描画に関するメソッド (地図のスクロールやアイコン表示) を利用すれば、容易に地図を含んだアプリケーションを開発できる。しかし、地図表示部品は完全なブラックボックスであり、用意されたメソッド以外の拡張は不可能である。例えば、地図上のアイコンをドラグするという拡張はできない。

これに対して、Nuts の地図表示部品は第 7.2節で述べた Nuts 部品を用いるため、ホワイトボックスコンポーネントとして地図表示部品の拡張が可能である。さらに、地図上のアイコンをドラグするという拡張は、ドラグ機能を持ったベクターコンポーネントの挿入で容易に実現できる。

実際に、Win32/MFC 版 Navimos は、実業務への適用で第三者に拡張を依頼した。その際、MFC の修得、および Navimos 自体の構造の把握のために、実際の作業開始まで 2 週間を要した。Nuts 版では、フレームワークの理解容易性が高く、またそれぞれの部品の独立性が高いため、全体構造を把握せずとも拡張部品の開発が行える。したがって、第三者による実業務適用のための拡張も短時間で可能だと考えている。

9.2 デザインパターンとの比較

Nuts 独自の構造／制御モデルに類似した拡張性を提供し得るものとして、デザインパターンカタログに含まれるパターンが挙げられる。本節では、デザインパターンを上記の視点から検討し、デザインパターンのみでは実際のプログラミングに問題があることを示す。その上で、Nuts では具体的な開発フレームワークとして、デザインパターンをどのよ

表 9.4: Navimos の開発におけるデータ比較

項目	Unix/Nuts	Win32/MFC
プログラム行数	2913	約 17200
開発工数 (人・月)	0.4	6
部品種数	32	46
Navimos 固有種数	8	40

表 9.5: Navimos 開発における比較

項目	Unix/Nuts	Win32/MFC
理解容易性	構造記述ファイルにより容易	MFC の Document-View 構造以外の部分は、ソースコードの解析必要
結合コスト	構造記述ファイルの記述により自動的に木構造が構築される	MFC の Document-View 構造以外の部分は、必要な結合関係をその都度構築する必要がある
通信コスト	制御モデルにより隣接していない部品間での通信も容易	通信が必要な部分に結合関係を作るか、参照が入手できる方法を用意する必要あり
再利用性	部品同士は緩やかに結合しており、単独で抜き出して再利用可能	ActiveX コンポーネントはそのまま再利用可能だが、それ以外の固有部品は、他の部品と密接に関係しており再利用困難
拡張性	ベクターコンポーネントを用いた差分プログラミングにより拡張可能	ActiveX コンポーネントの拡張は不可。それ以外の部分は、ソースコードを解析しクラス階層を理解した上で、サブクラス化により拡張する必要がある

表 9.6: Navimos の開発に用いた部品リスト

部品種	Unix/Nuts	Win32/MFC
汎用部品	NutsXtApp	CAboutDlg
	NutsXApplicationShell	CMainFrame
	NutsXTransiationShell	CMapocx
	NutsFormWindow	CButton
	NutsShellWindow	CStatusBar
	NutsDrawingAreaWindow	CProgressCtrl
	NutsAboutThisWindow	
	NutsFrame	
	NutsMenuBar	
	NutsPulldownMenu	
	NutsSeparator	
	NutsScrolledText	
	NutsPushButton	
	NutsDial	
	NutsFileSelectionBox	
	NutsInterpreter	
	NutsStdOut	
	NutsCore	
	NutsBatch	
	NutsListupObjectToPs	
	NutsDragger	
	NutsHorizontalScroller	
	NutsVerticalScroller	
固有部品	mapLoader	CConditionDialog
	navimosSaving	CTruckDialog
	navimosExecSavingCmd	CMapView
	navimosManager	CNavimosApp
	navimosSpot	CNavimosDoc
	mapvScopeWindow	CNavimosToolBar
	nodeRec	nodeCore
	routeRec	nodeRec
	truckRec	routeRec
		truckRec
		nodeManager
		routeManager

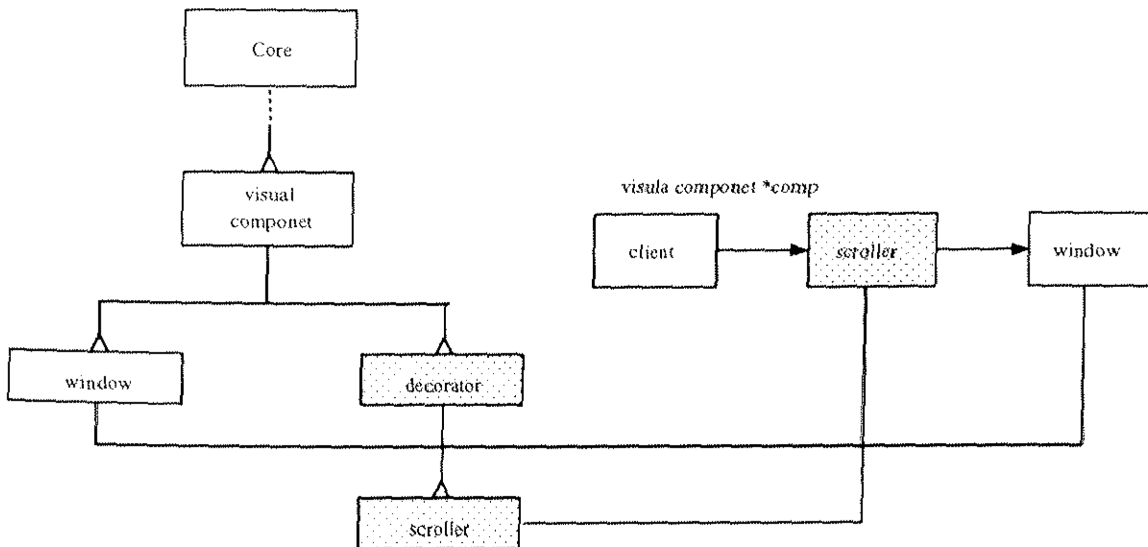

図 9.5: Decorator パターンの使用

うに具現化し、再利用性を高めているかを述べる。

9.2.1 修飾に関するパターンとの比較

Decorator パターンとは、「オブジェクトに責任を動的に追加する。Decorator パターンは、サブクラス化よりも柔軟な機能拡張方法を提供する」パターンである [24]。このパターンは、Nuts のベクターコンポーネントによる修飾に近いと考えられる。

Decorator パターンでは、既存のコンポーネントと同じインタフェースを持つ (共通の祖先クラスから派生した)Decorator コンポーネントによって、既存のコンポーネントを包み込む (図 9.5)。コンポーネントの利用者から見ると、インタフェースが同じなので本来のコンポーネントにアクセスしているのか、その Decorator にアクセスしているのか区別できない。すなわち、クライアント側のコードを変更することに無しに、機能の変更が可能になる。

Decorator コンポーネントは、Nuts のベクターコンポーネントに相当するものだと言える。しかし、Decorator パターンを実装する場合、デザインパターンカタログ [24] には示されていない問題点が 2 つある。以下に、Decorator パターン適用の問題点とベクターコンポーネントでの解決方法について説明する。

1. 図 9.5の例では、Decorator コンポーネントの挿入時において、クライアントから Window コンポーネントの参照を得る方法は、第 2.1節で示したように、さまざまなバリエーションがある。Decorator コンポーネントから間接的に Window コンポーネントをアクセスするように変更する場合は、それぞれの参照獲得手法に沿った方法で変更しなければならない。一方 Nuts のベクターコンポーネントでは、Decorator コン

ポーネントと一般のコンポーネントを区別しない。そのため、一般のコンポーネントを挿入するのと同等の容易さでベクターコンポーネントを挿入できる。

2. Decorator コンポーネントはそれが修飾するコンポーネントと同じインタフェースを持っていなければならない。そうしなければ、クライアントから見た時、Decorator コンポーネントと被修飾コンポーネントを同じ方法でアクセスすることができない。このような、Decorator コンポーネントを設計するには、被修飾コンポーネントのクラス階層を熟知し、適切な位置で Decorator コンポーネントのクラスを派生しなければならない。そして、その Decorator クラスをできるだけ汎用的に用いるために、できるだけ抽象レベルの高いクラスから派生させたい。しかし、かと言って Core クラスのような基底クラスに`draw()` のようなウインドウに特化したメソッドを用意するのは得策ではない。

   結果的に、コンポーネントのクラス階層を充分に検討し、ある分野 (例えばウインドウに関する分野) のできるだけ抽象レベルの高いクラスから Decorator クラスを派生させることになる。Decorator クラスは、図 9.6(a) に示すように、分野毎に存在することになり、クライアントは今どの抽象レベルのコンポーネントにアクセスしているのか (ポインターの型) を意識して、使用する Decorator クラスを選別しなければならない。場合によっては、インタフェースに適合する Decorator が見つからずに新たな Decorator クラスを定義しなければならない場合もある。

   これに対して、Nuts の Decorator パターンであるベクターコンポーネントの実装では図 9.6(b) に示すように、ベクターコンポーネントクラスを Nuts の基底クラス群から直接派生させることで (NutsShadowManger クラスのサブクラス) ベクターコンポーネント自体に高い汎用性を持たしている。具体的には、ベクターコンポーネントの親コンポーネントから見た場合、ベクターコンポーネントも NutsCore 型の抽象的なインタフェースで扱える (図 9.7)。このため、修飾のために、インタフェースの一致した Decorator コンポーネントしか挿入できないといった問題が発生しない。ベクターコンポーネントと一般のコンポーネントは基底クラスに近い抽象レベルで分岐しており、修飾/被修飾コンポーネントが独立したクラス階層に属するものとして、まったく別個に開発可能である。

このような、Decorator コンポーネントとベクターコンポーネントの相違は、被修飾コンポーネントに対して修飾を被せることによって行うか、横取ることによって行うかの違いに起因する。Decorator コンポーネントの被せる方式による修飾は、被修飾コンポーネントと同じインタフェースを持った別のコンポーネント (Decorator) で、被修飾コンポーネントを包み込むことにより、外部からのアクセスに対して、Decorator の存在を意識させない方式である。

この方式では、図 9.8(a) のように、Decorator コンポーネントは被修飾コンポーネントと同じインタフェースを持つ必要があり、被修飾コンポーネントが持たないインタフェースを後から追加できない。また、Decorator コンポーネントは、被修飾コンポーネントの

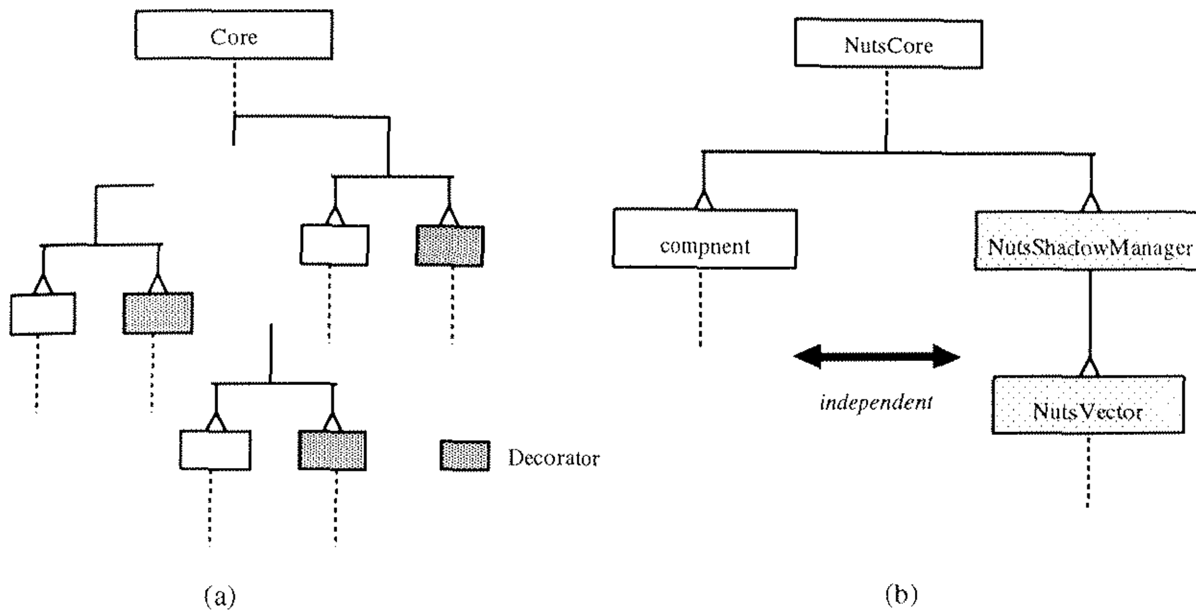

図 9.6: Decorator パターンの使用

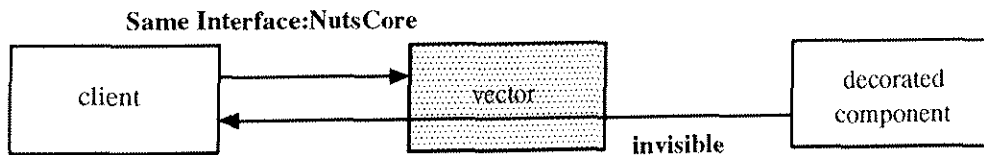

図 9.7: Vector コンポーネントのインタフェース

外部から作用するため、被修飾コンポーネント中の内部的な呼び出しを外部から修飾できない。

これに対して、ベクターコンポーネントは、図 9.8(b) のように、一度被修飾コンポーネントに送られたメッセージを横取りする方式をとっているため、本来被修飾コンポーネントが無視していたメッセージに対しても、ベクターコンポーネントで横取りすれば、被修飾コンポーネントに新たなインタフェースを加えるのと同じ効果が得られる。例えば、コンポーネント設計時には存在しなかったメッセージを受け取るようにベクターによって拡張できる。

また、被修飾コンポーネント内部でのメッセージ送信も、ベクターコンポーネントによる横取り対象となる。これにより、コンポーネント内で将来拡張が予想される部分にメッセージ呼び出しを用いておけば、ベクターコンポーネントでメッセージを横取りしてコンポーネントの一部の機能を別の機能で置き換えられる。このように、ベクターコンポーネントでは被修飾コンポーネント内部の拡張を行うこともできる。

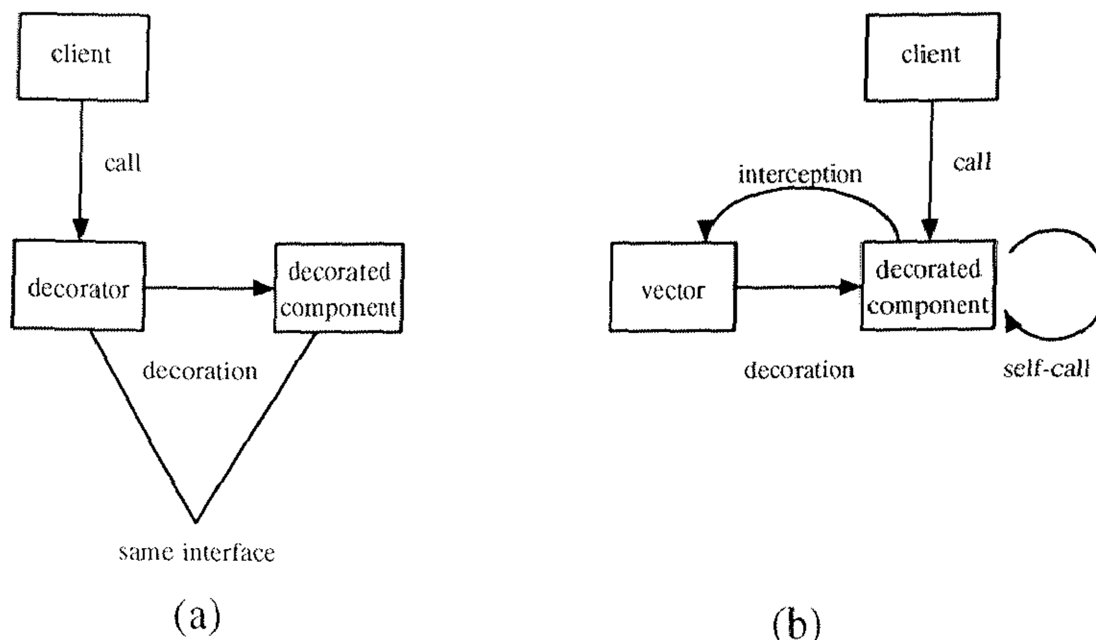

図 9.8: 被せると横取るの相違

ベクターコンポーネントの機構を Decorator として見た場合、図 9.7のように、ベクターコンポーネントは子からの参照を透過することで、親の条件に関する検査も回避している。つまり、子からは見えないことにより、ベクターコンポーネントは完全な透明コンポーネントしてプログラム中のどこにでも挿入可能なコンポーネントとなっている。また、ベクターコンポーネントは、積み上げ可能な単なる NutsCore クラスのオブジェクトであり、ベクターコンポーネントに対して具象的なメソッド呼び出しは行わず、代わりに Nuts の制御モデルを用いて間接的に行う。Nuts 制御モデルは受信者の動的な探索を行うが、探索の段階でベクターコンポーネントがメッセージを横取りしてしまうのではなく、最初、本来の被修飾コンポーネントに配信した後、そのメッセージを横取りするという方法を採っている。この方法により、ベクターコンポーネントは常に被修飾コンポーネントだけを監視していれば良い。また、ベクターコンポーネントで選択的に修飾機能を働かせ、無関係なメッセージに対しては横取り元 (本来の受信者) に解釈の機会を与えたり、横取り元が無視していたメッセージを受け取ることもできる。

9.2.2 機能の変更に関するパターンとの比較

Strategy パターンとは、「アルゴリズムの集合を定義し、各アルゴリズムをカプセル化して、それらを交換可能にする。Strategy パターンを利用することで、アルゴリズムを、それを利用するクライアントから独立に変更できる」パターンである [24]。このパターンは、ベクターコンポーネントによるコンポーネント差分プログラミングにより、動的にアルゴ

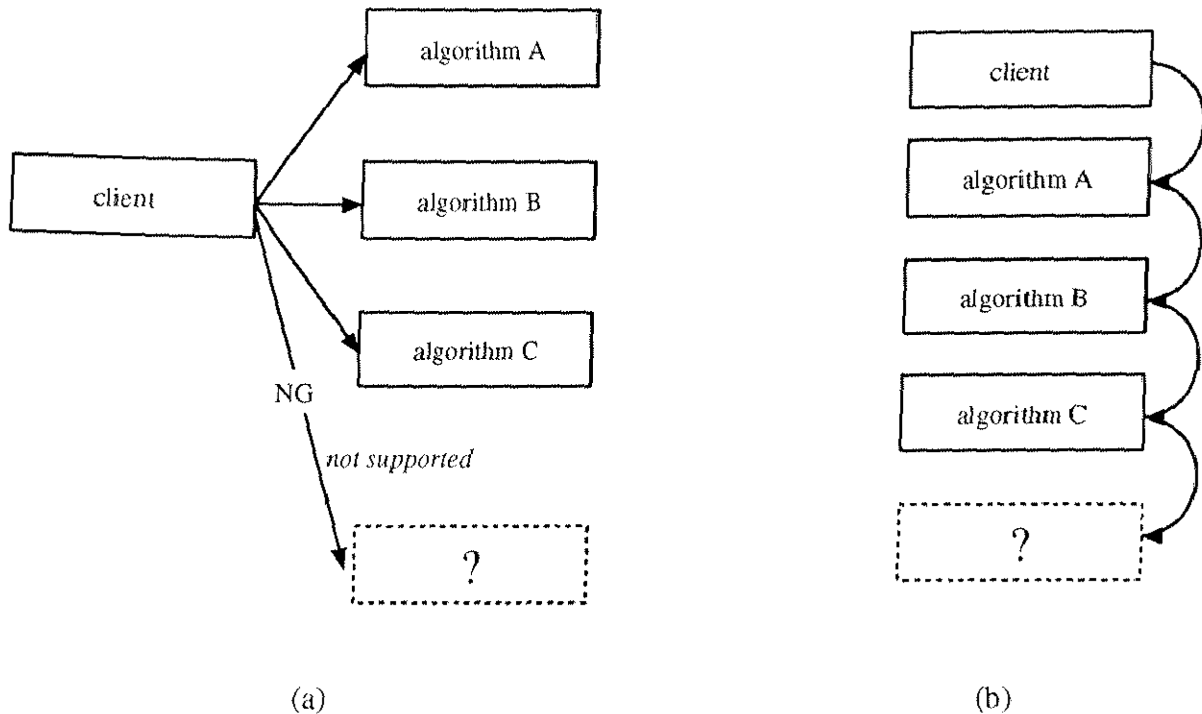

図 9.9: Strategy パターンの実装

リズムを変更する手法に相当する。

前節の Decorator パターンと比べると、Decorator コンポーネントがクライアントと被修飾コンポーネントの間に介在する透明な殻のようなものであるのに対して、Strategy パターンでは、Strategy への参照と言う形でクライアント中に拡張機能を内蔵している。

デザインパターンカタログ [24] に示されていない、Strategy パターンの実装上の問題を 2 つ挙げることができる。

1. Decorator パターンと同様、クライアントから Strategy への参照を得る方法は、第 2.1節で示したように、さまざまなバリエーションがあり、プログラマは各フレームワークの様式にあった方法を採らなければならない。

2. Strategy パターンを用いて拡張する場合は、設計時に将来どのような拡張がなされるかをある程度予想していなければならない。例えば、あるアルゴリズムをカプセル化して、Stategy パターンによって切替え可能にするということは、そのアルゴリズムが将来拡張されることを予想し、クライアント中にアルゴリズムへの参照と言う形で実装しておかなければならない。それ以外の拡張に対しては、スーパークラスに手を加えて改めて Strategy パターンを適用しなければならない (図 9.9a)。

   これに対して Nuts では、コンポーネント間の相互作用を起こす唯一の手段であるメッセージ通信をベクターコンポーネントにより横取りすることで修飾できる。これによ

り、ほとんどあらゆる動作の変更/拡張がコンポーネントレベルで可能であり、被修飾コンポーネント作成時には予想し得なかった拡張にも対応できる (図 9.9b)。

9.2.3 コンポーネントをグループ化するパターンとの比較

Mediator パターンは、「オブジェクト群の相互作用をカプセル化するオブジェクトを定義する。Mediator パターンは、オブジェクト同士がお互い明示的に参照し合うことがないようにして、結合度を低めることを促進する。それにより、オブジェクトの相互作用を独立に変えることができるようになる」パターンである [24]。このパターンは、Nuts におけるグループマネジャに対応する。

Mediator パターンは、Nuts の GroupManager クラスが実現している。NutsGroupManager クラスのコンポーネント (グループリーダ) は、コンポーネントの部分木をグループ化し、グループ内のメッセージの通信ターミナルとなる。グループ内のコンポーネントは、グループリーダを介して間接的に通信することによって、互いの結合度を低くすると共に、グループ内の相対的なメッセージ通信の記述だけに専念できる。これにより、部分木をグループ単位で再配置可能な高機能コンポーネントとして扱える。

Mediator パターンの実装上の問題として以下の 2 点を挙げることができる。

1. Mediator コンポーネントは、それが管理するコンポーネントへの参照を保持し、あるコンポーネントからの通知に対して、適切なコンポーネントのメソッドを呼ぶように設計する。これは、Mediator の負担を大きくすると共に、Mediator がコンポーネントへの参照を陽に管理することで、Mediator 自体の再利用が困難となる。

   Nuts のグループマネジャは、内蔵するコンポーネントへの参照を陽に保持しておらず、グループ化された部分木は、全体木と同じ構造／制御モデルに従っている。よって、グループ内への部品の追加なども従来通り簡単に行える。また、グループマネジャ自体を別の場面で使うこともできる。

2. Mediator パターンによるグループ化が必要なことが後から判明した場合、従来の方法では、Mediator コンポーネントを新たに設計し、コンポーネントの結合関係をそれに合わせて大幅に変更しなければならない。一方、Nuts のグループマネジャは透明コンポーネントの一種であり、グループ化するためにグループマネジャーを挿入したとしても、結合関係の変更の必要がない。また、グループの入れ子状態も、簡単に実現することができる。さらに、グループ内での通信はグループマネジャを介したメッセージ通信に依っているため、グループ内の部品同士は、陽にメンバーへの参照を保持していない。これにより、メンバーをグループに容易に追加できる。

9.2.4 コンポーネント群を単一のコンポーネントとして扱うパターンとの比較

デザインパターンの Facade パターンは、「サブシステムに内在する複数のインタフェースに一つの統一的なインタフェースを与える。Facade パターンは、サブシステムの利用を

容易にするための高レベルインタフェースを定義する」パターンである [24]。このパターンは、Nuts のグループマネジャが持つ部分木を一つのコンポーネントに見せる機能に近い。

Nuts のグループマネジャは、Facade コンポーネントとしての機能も果たしている。すなわち、グループの外からグループを一つのコンポーネントとしてアクセスできる。そして、Mediator コンポーネントと同様、グループ内部はメッセージ通信による緩やかな部品間結合が実現されているため、Facade コンポーネントによるグループ化が、グループマネジャの挿入により容易に可能である。

9.3 他のフレームワークとの比較

本節では、第 1.5節で述べたアプリケーションフレームワークと Nuts フレームワークを比較する。アプリケーションフレームワークは、開発しようとしているアプリケーションのドメインに適合する場合は利用効率が高い。そのような既存のアプリケーションフレームワークとして、ET++ [5] と IntelligentPad [88] を取り上げ、Nuts と比較する。ET++を取り上げる理由は、Nuts と同様に C++によるソースコードが流通しており、ホワイトボックス的な拡張を行えることによる。また、デザインパターンによりフレームワーク構造が説明されており、フレームワーク修得のための労力が比較的小さいと考えられる。また、IntelligentPad を取り上げる理由は、パッドの貼り合わせが Nuts の木構造に近く、さらにパッド自体の開発も C++で可能であるとされていることによる。

9.3.1 フレームワーク ET++との比較

本節では、C++を用いた開発フレームワーク ET++ [5] と Nuts を比較する。

ET++と Nuts で、簡単な"Hello world" というラベルのついたウインドウプログラムを記述する場合を比較する。ET++では図 9.10のように記述する [70]。一方、Nuts を用いた場合は、図 9.11のようになる。これを見ると、Nuts は構造が木構造に統一されているため、プログラムの構造が理解し易い。一方、ET++では、特定の仮想関数をオーバーライドして予め定まった手順でインスタンスを確保しなければならず、プログラマは、インスタンス確保の場所、順番、構築に必要な他部品への参照に常に注意をはらわなければならない。

また、ET++ではサブクラスとして作った新たなクラスのインスタンスを使うために、既存クラスでそのクラスのインスタンスを使用していた部分を探して書き換えなければならない。Nuts では、すべての部品を NutsCore クラスの部品として統一的に扱っているため、新たなクラスの部品が加わったとしても、それを利用する既存クラスのソースコードには手を加える必要がない。プログラマはコンポーネント単位で部品の追加、交換、削除が行える。

```
//include files...
class HelloDialog : public Dialog
{
MetaDef(HelloDialog);
HelloDialog() : Dialog(){ }
VObject *DoMakeContent();
};

VObject *HelloDialog::DoMakeConent()
{
return new Matte(newNuts
TextItem("Hello_World"));
}

class Hello : public Application
{
public:
MetaDef(Hello);
Hello(int argc,char *argv[])
: Appclication(argc,argv){ }
Manager *DoMakeManager(Symbol){
return newNuts HelloDialog();}
};

main(int argc,char *argv[])
{
Hello app(argc,argv);
reurn app.Run();
}
```

図 9.10: ET++を用いた Hello world

```
//include files...
newNutsMotifApp(hello);
newNutsMotifToplevel(hello,top);
newNutsApplicationShell(top,shell);
newNutsDialog(shell,dialog);
newNutsLabel(dialog,Hello_World);
```

図 9.11: Nuts を用いた Hello world

9.3.2 IntelligentPad との比較

本節では、コンポーネントウェア製品のうちで、構造的に Nuts の木構造に近いものとして、IntellignetPad[88][82] を取り上げ、Nuts と比較する。IntelligentPad では、プログラム部品をパッドのメタファとして取扱っている。パッドにはスロットがあり、同じ型のスロット同士を接続し、パットを貼り合わせることでソフトウェアが構築できる。パッド自体は、手続きパッドと呼ばれるスクリプト言語やC++を用いて開発でき、ホワイトボックス的なコンポーネントとしての側面も持っている。

ただし、IntelligentPad はエンドユーザ向けのブラックボックス的なコンポーネントの傾向が強く、C++でのコンポーネントの開発、プログラムの構築は熟練したプログラマーでなければ難しい。また、IntelligentPad ではスロットに様々な型があり、第 2.4節で指摘したように、プログラマは接続するスロットの型に注意しなければならない。さらに、パッド間の制御は接続されたスロットで伝達されるが、直接貼り合わされていないパッド間の伝達には、貼り合わせ関係以外の例外的な接続を作らざるを得ず、パッドの貼り合わせというプログラム構造の一貫性を壊している。

Nuts では、一種類のスロット (親子関係) で様々な種類のメッセージを伝達することによりプログラマは接続関係に注意する必要がない。また、直接貼り合わされていない部品間も動的な受信者探索という方法で、構造に影響を与えることなく通信が可能になっている。

9.4 コンパイル時 MOP を用いた拡張との比較

既存のクラスやコンポーネントの機能をより広範囲に拡張する方法として、リフレクション機能を用いる方法がある。C 言語にリフレクション機能を採り入れたものの代表として、OpenC++[11][12] がある。OpenC++のコンパイル時 MOP を用いれば、コンパイルを前提とした言語でも一種のプリプロセスによって、関数呼び出しや値の設定／読み出しを横取りし、拡張機能を追加できる。

しかし、コンパイル時 MOP は極めて自由度が高いため、本稿で検討しているようなコンポーネントレベルでの機能拡張には強力過ぎ、他と整合性の取れない拡張を行ってしまう危険がある。また、拡張ごとにメタクラスの処理系を用意することは煩雑であるうえ、複数の拡張を組み合わせた際に正しく動作することを保証するのは簡単ではない。

これに対し、本論文で提案しているベクターコンポーネントはコンポーネントという同一のレベルでの設計が行えるため簡潔でわかりやすく、通常のコンポーネントを操作するツールをそのまま利用できる。また、コンポーネント間のメッセージ通信の横取りに機能が絞られているため、安全で扱いやすい。言語処理系も特別なものである必要はなく、通常の C++コンパイラがそのまま利用できる。