Chapter 2

柔軟な結合が可能なコンポーネント–Nuts

本章では、前章で述べたオブジェクト指向を基盤とする開発の枠組を

- ホワイトボックス的プログラミング—クラスを対象 (クラスライブラリ、フレームワーク)
- ブラックボックス的プログラミング—インスタンスを対象 (コンポーネント)

に分類した上で、インスタンス結合によるプログラム構築時と、インスタンス自体の拡張時における、それぞれの利用形態の問題点に焦点をあてる。そして、それらの問題点を解消する方策の一つとして、ホワイトボックス的なコンポーネントの利用、およびコンポーネント差分プログラミングと呼ぶ拡張手法を持ったNutsコンポーネントを提案する。

2.1 ホワイトボックス的プログラミングの特徴と問題点

ホワイトボックス的プログラミングとはクラスを対象にしたプログラム形態である。ホワイトボックス的プログラミングでは、既存のクラスに対してサブクラスを定義し、サブクラスで変化する部分 (拡張や制約) のみを記述する。このような拡張方式を差分プログラミングと呼ぶ。差分プログラミングでは、スーパークラスのコードを再利用することにより、最小限のコード量で拡張が可能であると同時に、必要な機能をもったクラスを適宜派生させることにより柔軟な拡張が可能である [25][26]。このようなホワイトボックス的プログラミングにおけるクラスの実装過程を "a-kind-of" 設計側面として 1.2節で説明した。

これに対して、ホワイトボックス的プログラミングにおける "a-part-of" 設計側面では、クラスのインスタンスを作成し、それらを結合させる必要があるが、従来のホワイトボックス的プログラミングの枠組では、インスタンス同士の結合のさせ方に一貫性がなく、以下のような問題があった。

- コンポーネント同士の結合のさせかたが難しい。
- 結合した結果の構造がわかりにくい。

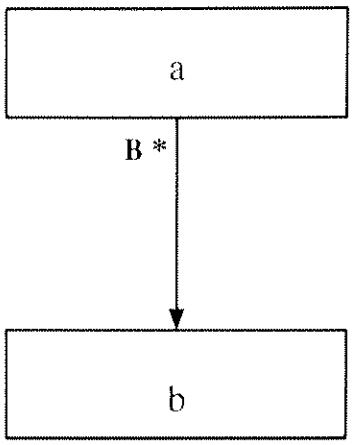


図 2.1: オブジェクト a,b の参照関係

- 通信相手のコンポーネントの場所がわかりにくい。

簡単な例として、インスタンスの初期化時の問題を考える。図 2.1のオブジェクトグラフに示すような、クラス A のインスタンス a がクラス B のインスタンス b を参照する関係を構築したいとする。これには、以下のような様々なバリエーションが考えられる。

1. B を A が生成される前に生成し、A へのコンストラクタ引数として渡す。

```
B *b = new B;
A *a = new A(&b);
```

2. A を生成する前後で B を生成し、A が生成された後で専用のメソッドで渡す。

```
B *b = new B;
A *a = new A;
a -> init(b);
```

3. A のコンストラクタ中で B を生成する。

```
A *a = new A;

A::A()
{
    B *b = new B;
}
```

4. A のコンストラクタ以外の初期化メソッドで B を生成する。

```
A *a = new A;
a -> init();

void A::init()
{
    B *b = new B;
}
```

1. の方法ではインスタンスの生成順序に気を配らなければならない (a は b の後でなければ生成できない)。また、インスタンス毎に異なったコンストラクタ引数があり、単一の方法で生成できない。2. の方法では、a に b を渡すメソッドを忘れないように呼ぶことと、その順番に注意しなければならない。つまり、b が渡されるまで a の中で b を利用する処理は控えなければならない。3. の方法では a の中に b が含まれていることが外から見えないため、b を他のインスタンスと共用したい場合に問題となる。4. の方法では、3 の問題に加えて、初期化メソッドをよぶ順序に注意を払わなければならない。

さらに、a が内部的に保持する b に対して、外部のインスタンス c がアクセスしたい場合には、以下のようなバリエーションがある。

1. A の外から b のサービスにアクセスするメソッドを設ける。

```
a->foo();

void A::foo()
{
    b -> foo();
}
```

2. A の外に b の参照を公開するメソッドを設ける。

```
B *b = a -> get();
b -> foo();
B *A::get()
{
     return b;
}
```

3. b を大域変数にセットし、どこからでもアクセスできるようにしておく。

1.,2. の両方法とも参照関係の入れ子が深くなると、それぞれで内部にアクセスするためのメソッドを用意せねばならず煩雑である。3. の方法は、大規模なプログラムでは大域変数の管理が不可能となり問題外である。

以上のような、インスタンスの結合、及びインスタンスの参照の入手に関するバリエーションは、デザインパターンカタログ [24] でも取り上げられておらず、実際のプログラミングの際にどの方法を採用するかは、プログラマによりまちまちである。また、いずれのバリエーションも以下のような問題を含んでいる。

- 特定のクラス間の関係が密接過ぎる— インスタンス結合時には相手の種類、結合のさせ方に注意しなければならず、柔軟なインスタンス間の結合を妨げている。

- 全体の構造がクラス定義の中に埋没してわからない— インスタンスの参照関係が簡単に把握できない。特にクラス内部でインスタンスを生成する場合は、現在のインスタンスをサブクラスのインスタンスに変更するために、クラス内部の生成部分を探して書き換えなければならない。

- メソッドを呼ぶ順番に注意する必要があり、バグの温床となる— 参照関係の構築をプログラマが手動で行うため、手順を間違った場合は不可解なバグ (Null にアクセスするなど) に悩まされることになる。また第三者による保守のためには、手順を詳細に記したドキュメントを整備する必要がある。

このように、ホワイトボックス的な利用では、差分プログラミングによる拡張性は確保されているが、インスタンス化したときのインスタンス間の結合関係構築に統一性が確保されておらず、インスタンス間の相互関係 (全体構造) を把握するのが困難になっている。

ホワイトボックス的プログラミングと本章で提案する Nuts コンポーネントにおけるコンポーネント結合の比較については、第 2.4節で述べる。

2.2 ブラックボックス的プログラミングの特徴と問題点

ブラックボックス的プログラミングとは、クラスのインスタンスを対象としたプログラミングの形態である。ブラックボックス的なプログラミングでは、予め属性を調整したインスタンス (コンポーネント) を永続的なストレージ (ディスクなど) に保存しておく (OLE の StructuredStrage [81] や OpenDoc の Bento [71] などが保存のためのファイルシステムに関するコンポーネントの仕様である)。プログラム構築時にコンポーネントを利用する場合は、ストレージから読み出してインスタンスの状態を復元し、プログラム中に組み込む。

コンポーネントは、ある程度汎用的に作られた部品クラスのインスタンスであり、フレームワーク上で相互に結合してプログラムを構築する [81][65]。これにより、必要な機能をもったコンポーネントを収集し結合するだけで、容易にプログラム開発が可能である[1]。

[1] このためにコンポーネントの仕様では、コンポーネントが持つインタフェースの探索手段 (OLE の IUnkown インタフェースや CORBA のオブジェクトリファランス獲得メソッド) を規定している

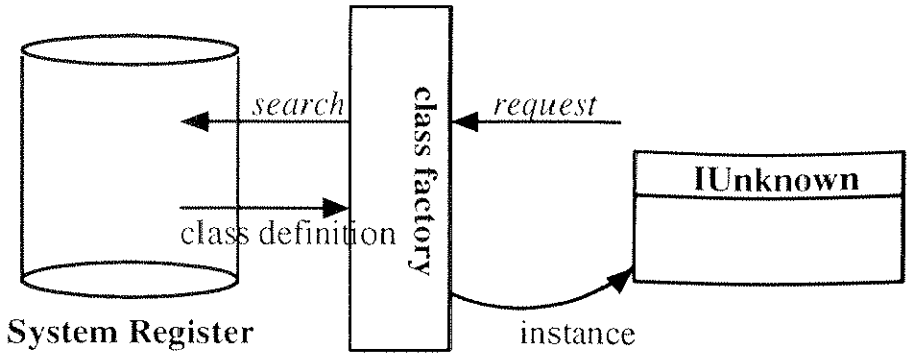


図 2.2: ActiveX コンポーネントの取得

フレームワークは、コンポーネント間の結合の仕方を規定しており、ほとんどの場合はコンポーネントを GUI 上で結線したり重ね合わせるだけで、プログラムの構築が可能となっている。

例えば ActiveX コンポーネントの場合は、図 2.2のように、システムのレジストリに登録されているコンポーネントクラスを探索し、要求するクラス定義を入手する。次にクラスに対してインスタンス生成メソッド (`CreateInstance()` スタティックメソッド) を呼ぶと共通インターフェース (ActiveX コンポーネントに共通のインタフェース`IUnknown`) を持ったインスタンスが生成される。このコンポーネントを ActiveX サーバー (コンポーネントを載せるコンテナ) に登録するとコンポーネントの利用が可能となる。

ActiveX の例のように、ブラックボックス的なプログラミングを行うコンポーネントでは、入手できるクラスの情報は、必要なコンポーネントに関する断片的なものであり (単なる雛型といえる)、クラス階層など新規コンポーネントの開発に必要な情報は含まれない。その結果、プログラマが利用できるのは、クラスの実体であるブラックボックス的なインスタンスである。

ブラックボックスコンポーネントでは、スクリプティングと呼ばれる方法でイベントハンドリング等を記述する以外にコンポーネントの機能を拡張する手段はまったくない。例えば ActiveX コンポーネントでは、オートメーションと呼ばれるスクリプト言語 (主に Visual Basic) を用いてイベントハンドリング (ボタンが押された時の応答など) を記述できる。その他のカスタマイズは、属性値 (プロパティと呼ばれる) により、背景色やフォントを変更するなどの小規模なものに限られる。

ブラックボックスのまま汎用的に利用できる部品の範囲は限られており、現在流布しているコンポーネントを見ると、GUI 部品や通信部品など、比較的プリミティブな処理をカプセル化したものに留まっている。これらは定型的な処理をサポートするだけであり、柔軟な拡張を必要とする非定型分野においてコンポーネントは十分に役立っていない。

ブラックボックス的プログラミングと Nuts コンポーネントにおける拡張性の比較については、第 2.4節で述べる。

2.3 ホワイトボックス的プログラミングが可能なコンポーネント

ホワイトボックス的プログラミングは、オブジェクト指向プログラミングの要素であるオブジェクトのカプセル化、クラス継承、多相性を持つ。クラス継承は実装の継承を行い、差分プログラミングを可能にする。しかし、それらを実際のプログラミングにおいて利用するには障壁が高く、ホワイトボックス的な利用は普及していない。

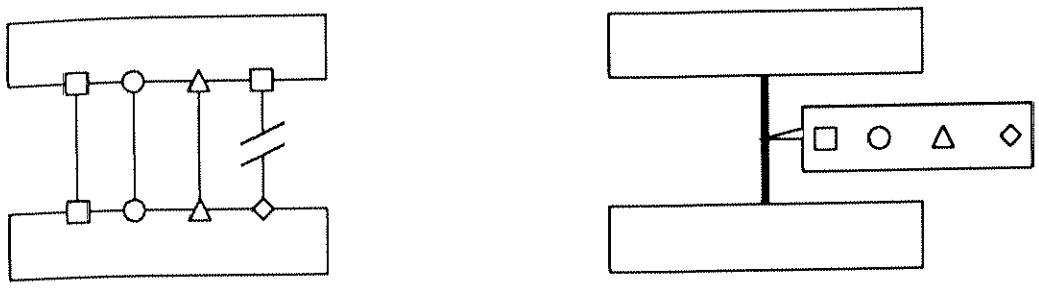
一方、ブラックボックス的プログラミングは、クラスの実装を継承せず、強固なカプセル化の性質のみを持つ。そのため、エンドユーザでも、インタフェースに一致する様々なオブジェクトを収集し結合することで、容易にプログラム部品を利用できる。今後のオブジェクト指向をベースとしたプログラミング形態は、難解なホワイトボックス的プログラミングから、扱いやすいブラックボックス的プログラミングへと移行し、より大粒で高機能なコンポーネントをベースとしたプログラミングが定着すると考えられる。しかしながら、コンポーネントベースプログラミングは、本来のオブジェクト指向プログラミングが持っていたコンポーネント自体の拡張性の高さを欠いている。

これまでの問題点を整理すると、実際のプログラミングで役立つソフトウェア部品として、ホワイトボックス／ブラックボックス的プログラミング両者の利点を兼ね備えた、以下のような性質をもつ、コンポーネントアーキテクチャが必要である。

- コンポーネントをブラックボックスとして扱い、機械的にコンポーネントを結合することでプログラムを構築できる。
- 柔軟な結合を可能にするため、隣接していないコンポーネント間でも通信可能な仕組みを備えている。これにより通信手段が構造による不当な制約を受けない (構造と通信手段がある程度独立である)。
- 結合の際に、様々な粒度のコンポーネントがすべて類似した形をしており、単一な方法で互いに結合、連携できる (結合のためのインタフェースを単一にする)。
- 複数のコンポーネントを組み立てたもの (オブジェクトコンポジション後のコンポーネント) も、また1つのコンポーネントとして統一的に扱える。
- コンポーネントはホワイトボックスとして扱い (実装を継承する)、新たなコンポーネントの開発を既製クラスからのサブクラス化によって差分プログラミングにより行える。

以上のような特徴をもつコンポーネントは、ホワイトボックス的プログラミングが可能なコンポーネントであり、コンポーネントベースのプログラミングに、サブクラス化による拡張性を加味したコンポーネントである。著者は、このようなホワイトボックス的プログラミングを可能とするコンポーネントアーキテクチャを開発し、Nutsと名付けた。Nutsの特徴は以下の通りである。

- Nutsコンポーネントは、実装の継承を行うホワイトボックス的なプログラミングにより、コンポーネント自体を差分プログラミングによって柔軟に拡張できる。



conventional component composition

(a)

Nuts component composition

(b)

図 2.3: オブジェクトの結合

- Nuts コンポーネントはオブジェクト結合のためのインタフェースを統一し、オブジェクトコンポジションによる拡張が容易である。

以上のような特徴により、Nuts ではソフトウェアの開発を、プログラム部品の開発 (サブクラス化によるホワイトボックス的側面での作業) とそれらの結合によるプログラムの構築、既存プログラムの保守 (コンポーネント結合によるブラックボックス的側面での作業) とに効率的に分割できる。

2.4 Nuts コンポーネントの結合

一般にソフトウェア開発では、システム全体をいくつかのサブシステムに分割し、さらにそれぞれのサブシステムを機能単位に細かく分割する。そして、オブジェクト指向プログラミングでは、各機能単位を抽象レベルの観点からクラス階層に分割し、それぞれをオブジェクトとして定義する。プログラムは、クラスを実体化させたオブジェクトを生成し、それらの結合による相互作用に基づいて動作する。オブジェクト指向プログラミングでは、オブジェクト間に多くの複雑な相互関係が生まれるため、プログラムの一部を変更する際には、プログラム中のすべてのオブジェクトのことを知っていなければならないことが多い。つまり、オブジェクトの参照関係とその参照先のオブジェクトの種類をすべて把握した上でなければ、プログラムの部分的変更が困難である。

この問題に対して、プログラムの小規模変更のために、プログラム全体を作り直すのではなく、一部の部品の取り替えで対応できることが望ましい。言い替えれば、オブジェクト指向プログラミングでは、ソフトウェア部品の再利用性／拡張性だけでなく、それらを組み合わせたプログラムも柔軟性が高い必要がある。しかし前述のように、オブジェクトの関係が強すぎる場合、オブジェクト自体の再利用性や拡張性は高くとも、プログラム全体としては柔軟性の低いプログラムになるという結果に陥る。

オブジェクト同士の強い関係を 2 つのコンポーネント間の局所的な関係で見てみる。コ

ンポーネント同士の結合を、コンポーネント間のスロットの接続ととらえた場合、従来のコンポーネントは、複数のスロットを有し、それぞれのスロットには特定の型が指定されている (図 2.3a)。このようなタイプのコンポーネントでは、コンポーネント結合の際に以下の点に注意を払わなければならない。

- スロットの接続相手、およびスロットの型を一致させなければならない。
- コンポーネントは隣接したもの同士で直接メッセージの受渡しを行うため、任意のコンポーネント間でメッセージ通信を行うためには、コンポーネント同士が隣接するようにしなければならない。

一方、Nuts のコンポーネントは、どのようなデータ型も伝送可能なただ一つのスロット (親子関係) しか持たない (図 2.3b)。このため、以下のような利点がある。

- スロットの接続に関する煩雑さが解消する。
- メッセージの受渡しは、メッセージ送信が必要になったときにはじめて受信者を探索した上で行うため、コンポーネントの接続位置についても注意を払う必要が無い。

以上のように、Nuts コンポーネントを用いれば、機械的なコンポーネントの結合作業で容易にアプリケーションプログラムを構築できる。

2.5 モジュール性に関する議論

本節では、前節で見た Nuts コンポーネントの結合をコンポーネントのモジュール性と言う観点から考察する。

ソフトウェアの拡張性、再利用性、互換性を高めるためには「ソフトウェアのモジュール性」が重要であると考えられている。モジュール性の基準として下の 5 つの基準に照らし合わせて考える [40]。

1. モジュールの分解しやすさ
2. モジュールの組み合わせやすさ
3. モジュールのわかりやすさ
4. モジュールの連続性
5. モジュールの保護性

Nuts コンポーネントアーキテクチャは、1.,2. について、クラスをベースとしたオブジェクト指向言語のサポートによる "a-kind-of" 側面での分解しやすさ、および木構造の特質にもとづく "a-part-of" 側面での組み合わせ易さを併せ持っている。木構造はトポロジー的に自己相似であり、部分木を木構造に結合したものがまた部分木として振舞うフラクタル構

表 2.1: 各コンポーネントの結合度

コンポーネント	結合度
Nuts	1
Applet class	4
Label View	5
View Control	20

造をもっているため、一つのコンポーネントも複合コンポーネントも単一な方法で組み合わせることができる。3.,4. に関して、Nuts コンポーネントは、他のコンポーネントと相互に関係してはいるが、動的メッセージ探索によりそれらは緩やかに結合している。このため、コンポーネントの部分的な抜き出しや変更に対して頑強な構成となっている。

上記基準を満たすための原則として「より少ないインタフェースの実現 (コンポーネントはできるだけ少ないコンポーネントと通信すべき)」がある [40]。Nuts では、各コンポーネントは親子／兄弟の隣接コンポーネントとのみ直接通信する。これにより、不特定多数のコンポーネントとの通信や通信を集中管理するコンポーネントが存在しないため、柔軟かつ頑丈な構造を構築できる。

少ないインタフェースを表す指標としてクラス連結数がある [39]。クラス連結数は他のクラスとの協調の総数を示す。例えば、Nuts コンポーネントは、前節で示したように、一種類のコンポーネントと単一な方法で通信を行うため、結合度は 1 である。これに対して、MFC[49] の View コントロールでは、MFC フレームワークとの結合度が 20、Java の Applet クラスや Java Beans[22] の Label View の結合度は、それぞれ 4,5 になる (表 2.1)。以上から、Nuts コンポーネントの結合度は低くモジュール性は高いと言える。

2.6 サブクラス拡張の問題点とコンポーネント差分プログラミング

前節で説明した、Nuts のコンポーネント結合モデルは、ホワイトボックス的プログラミングにおけるインスタンスの結合規則の不統一に起因する以下のような諸問題を解決する。

- コンポーネント同士の結合のさせかたが難しい。
- 結合した結果の構造がわかりにくい。
- 通信相手のコンポーネントの場所がわかりにくい。

一方で、コンポーネントの拡張は従来通りサブクラス化によって行う。差分プログラミングによるコンポーネントの拡張は、効率的で柔軟であるが、以下のような問題がある。

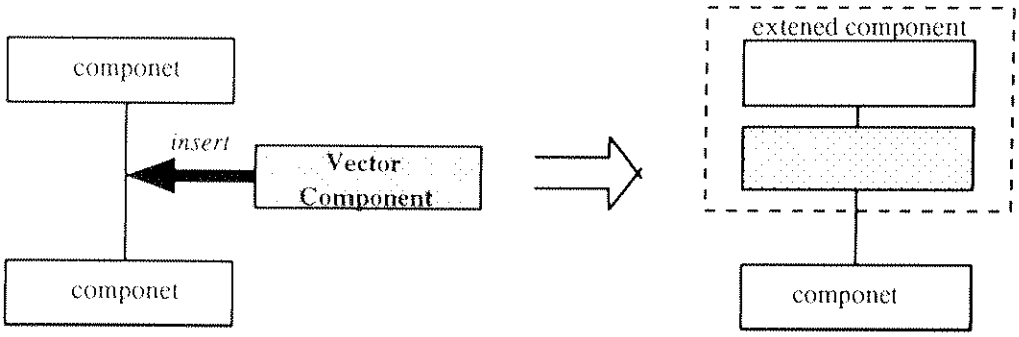


図 2.4: ベクターコンポーネントによる拡張

- 将来の拡張に備えた汎用的なクラス階層を構築することは難しい。
- 拡張のためにスーパークラスに手を加えなければならない場合もある (一つのサブクラスのためにスーパークラスを修正することは、他のサブクラスに影響を及ぼすこともあり危険である)。

このため、クラスの構造を十分に理解したプログラマでなければ、ホワイトボックス的プログラミングを十分に使いこなすのは難しい。

これに対して一般のプログラマは、機能拡張のための差分のみをコーディングしたコンポーネントを別に開発し、既存コンポーネントと結合させることで機能を拡張できる、コンポーネントベースの拡張が扱い易いと考えられるが、以下のような理由から、この方法はほとんど行われていない。

1. 拡張機能だけを持ったコンポーネントとの結合は、フレームワークが前提とするプログラム構造に抵触する場合がある。

2. 一般にコンポーネントは未知のコンポーネントとの結合を前提として作られていない。

著者は、上記のような問題点を回避し、コンポーネントベースプログラミングの利便性 (コンポーネントの結合によって機能の追加／変更／削除が容易にできる) の上でプログラムの拡張が行える手法を考案した。

このような、コンポーネントベースでの拡張が可能な新しい手法として、コンポーネント差分プログラミングと呼ぶ方法を提案する。コンポーネント差分プログラミングでは、上記の問題点を回避できる特殊なコンポーネントに、拡張機能の差分をカプセル化し、フレームワーク中に組み込まれているコンポーネントと結合させ、拡張機能を追加する (図 2.4)。コンポーネント差分プログラミングには以下のような利点がある。

- 拡張機能もコンポーネント化されているため、それ自体の再利用が可能である。
- 拡張機能をコンポーネントとして扱えるため、プログラムに対する追加／削除／変更が簡単にできる。

- 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。

Nutsでは、コンポーネント差分プログラミングを実現するための特殊なコンポーネントを、ベクターコンポーネントと呼んでいる。ベクターコンポーネントは、Nutsコンポーネントの一種であり、他のコンポーネントと同じ形をしているが、コンポーネント差分プログラミングを行うための以下のような特徴を持つ。

- ベクターコンポーネントは、他のコンポーネントにとって存在しないかのように振舞うことができる「透明な」コンポーネントである。これによって前述のようなフレームワークや被拡張コンポーネント側の制約に拘束されることなく、どこにでも拡張機能を組み込むことができる (前記問題1. の回避)。

- Nutsコンポーネントは、未知のコンポーネントによって修飾されるための特別な仕組みは持たない。しかし、Nutsコンポーネントは、基本的にメッセージの受信により動作するため、このメッセージをベクターコンポーネントが横取ることによって、コンポーネントの動作を拡張／変更できる (前記問題2. の回避)。

以上のように、Nutsコンポーネントとベクターコンポーネントを組み合わせることで、コンポーネントレベルのままで (クラスに戻ることなく) 差分プログラミングが可能になり、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。コンポーネント差分プログラミングおよびベクターコンポーネントについては、第6章で詳説する。

2.7 ブラックボックス的なプログラミングと比較したNutsの拡張性

本節では、ボタンコンポーネントの拡張性を例に、ブラックボックスコンポーネントの代表であるActiveXコンポーネントとNutsコンポーネントを比較する。

ActiveXコンポーネントとして開発されたCButtonコンポーネントを利用する場合を考える。プログラマは、システムレジストリから、CButtonCtrlコンポーネント (ボタンの定義そのもの) のクラスを検索し、そのインスタンスをCWndコンポーネント (ウインドウを表すクラス) に組み込んだ形でCButtonクラスのインスタンスとしてプログラム中に取り込む (図2.5)。プログラマから見た場合のCButtonコンポーネントは、CWndコンポーネントの一種にしか見えず、それ自体をサブクラス化してもCButtonCtrlを拡張することにはならない。したがって、ボタンを押す毎にボタン上のラベルの数字が増加していくようなCMyButtonへの拡張はできない。

このような拡張を実現しようとした場合、CButtonではなく、CButtonCtrlクラスを拡張しなければならない。しかし、一般にブラックボックス的なプログラミングではソースコードは流通していない。例えばCButtonCtrlの以下の部分をオーバーライドした、CMyButtonCtrlクラスを派生すれば、ボタンの選択に連動してラベルを変更するように拡張できるが、これは一般のユーザには不可能である。

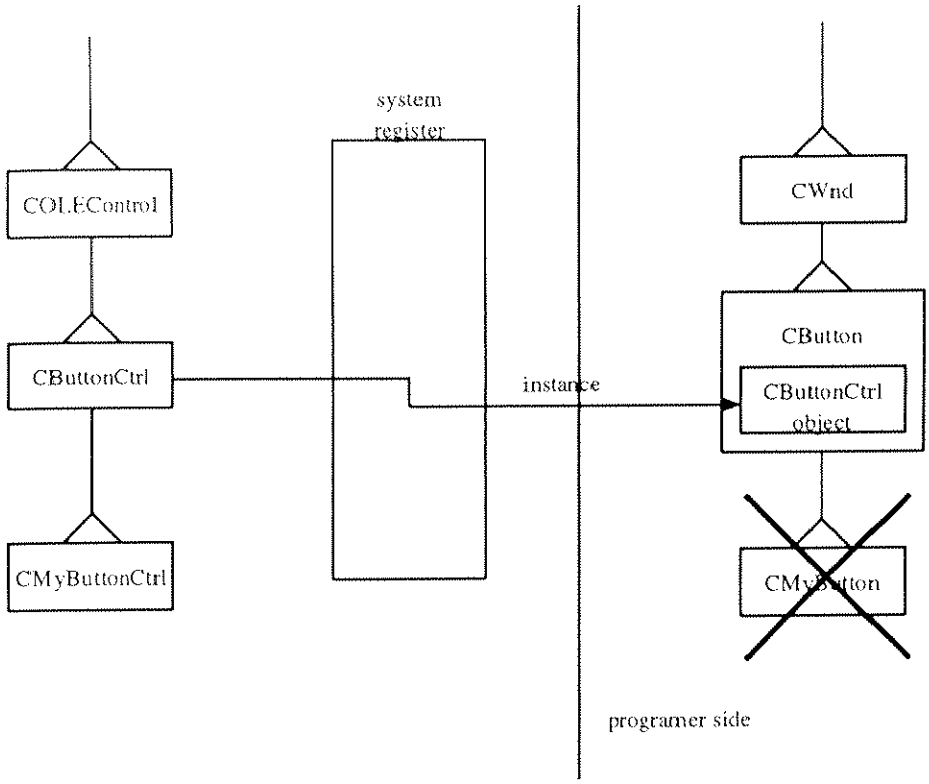


図 2.5: ActiveX コンポーネントの使用と拡張

```
void CMyButtonCtrl::OnDraw(CDC *,cnst CRect &,const CRect &)
{
  //カウントに合わせたラベルの変更に関する記述
}
void CMyButtonCtrl::OnClick(USHORT)
{
  //ボタンが押された時のカウントアップの記述
}
```

Nuts コンポーネントでは、部品のインタフェースはすべて同じであり、ブラックボックス的な部品として自在にプログラム中に組み込める。さらに、Nuts コンポーネントのプログラミング形態はクラスを対象としており、ソースコードとコンポーネントをセットで配布する。したがって、プログラマは NutsButton クラスから NutsMyButton クラスを派生させ、ボタンの選択に連動するような拡張を行うことができる。具体的には、ボタンが押されたときに届くメッセージのハンドラ中で、ラベルの値を変更する。

```
void NutsMyButton::recognizeMessage(NutsCore *,int opc,void *,void *)
{
```

```
  switch(opc){//opc はメッセージの種類
  case NUTS_CLICKED:
   //ボタンが押された時のカウントアップの記述
   //カウントに合わせたラベルの変更に関する記述
  break;
  }
}
```

以上のように、Nuts ではクラスを対象としていることにより、プログラマが柔軟にコンポーネントを拡張できる。

あるいは、第 2.6節で説明したようにコンポーネント差分プログラミングによる拡張も可能である。コンポーネント差分プログラミングでは、ボタンが押されたときに届くメッセージを横取りして、ボタンのラベルを変更するようなコンポーネントを別に開発し、ボタンに結合させることで拡張する。

2.8 C++開発フレームワークとしての Nuts

現在、Nuts フレームワークはオブジェクト指向言語 C++で実装しており、C++言語の開発フレームワークとしても位置付けられる。例えば、Nuts の利用は、これまで C++言語を用いながらも以下のような点で、効果が現れていないと考えているプログラマに適している。

1. 新しいソフトウェア開発のたびに基底クラスから再構築している：オブジェクト指向プログラミングでは、既存のクラスを再利用することによって、ソフトウェアの生産性を向上させるが、開発プロジェクトのたびに基底クラスから設計しなおしていたのでは生産性の向上につながらない。

2. クラス構造の再設計の必要性が頻繁にある：クラス階層が的確に設計されていない場合は、クラスを再利用しようとしてもサブクラス拡張に適したクラスが見つからずに、クラス階層そのものを見直す必要が生じる。

3. オブジェクト間の参照関係が複雑になり、保守や第三者への説明が困難 (第 2.1参照)：クラス階層を的確に設計できたとしても、オブジェクトの結合関係が複雑な場合は、第三者がそれを理解するのが難しく、保守や拡張を妨げる要因となる。

4. C++言語を通常の C 言語と同じように用い、オブジェクト指向をベースとしたソフトウェア開発の枠組を採り入れていない。

1. の問題については、Nuts では、Nuts のコンポーネントアーキテクチャに従う限り、基底クラスの再利用が可能である。Nuts の基底クラスは、Nuts コンポーネントとしての共通の性質をまとめたものであり、基底クラスからサブクラス派生することで Nuts コンポーネントの性質を持ったコンポーネントを作成可能であると同時に、差分プログラミングに

よって個別の性質を容易に付加できる。2. の問題については、Nuts のコンポーネント差分プログラミングを用いれば、拡張したい部品のクラス構造を変更することなく、拡張機能を付加できる。被拡張部品と拡張機能を定義したクラスは基底クラスで分岐した別のクラス階層を持ち、拡張機能の追加が被拡張部品のクラス構造に影響を及ぼすことがない。3. の問題に対しては、Nuts では、コンポーネントの結合関係を統一しており (第 3.1節参照)、構造の理解や第三者への説明が容易である。また、隣接していないコンポーネント間の汎用的な通信手段 (第 3.2節参照) を用意することで、通信のためにコンポーネント結合関係が制約を受けることがない。4. の問題に対しては、プログラマがクラスやクラス階層の設計を適切に行えない場合でも、Nuts では、Nuts コンポーネント群のクラス階層が提供されるため、それらを利用することによって、必然的にオブジェクト指向プログラミングの枠組を採り入れられる。

以上のように、Nuts では、Nuts コンポーネントのブラックボックス的な性質により、C++でコンポーネントベースのプログラミングが可能になる。さらに、C++プログラマであれば、差分プログラミングによって新規コンポーネントの開発、導入が容易に行える。このように Nuts は、C++の開発フレームワークとして有効に利用できる。

2.9 考察

本章では、現在のオブジェクト指向プログラミングに基づくソフトウェア開発の問題点をホワイトボックス的／ブラックボックス的プログラミングの観点から整理した。それに基づいて、コンポーネントベースのプログラミングでありながら、拡張性の高い Nuts コンポーネントアーキテクチャを提案した。Nuts コンポーネントアーキテクチャは、ホワイトボックス的プログラミングとして、差分プログラミングに基づく柔軟な拡張性を保証するとともに、インスタンス化したときのオブジェクトを単一な種類のコンポーネントとして柔軟に結合できるため、オブジェクトコンポジションによる拡張性も併せ持っている。

さらに、ホワイトボックス的な側面でのクラス派生による拡張の問題点に対して、コンポーネント差分プログラミングと呼ぶ手法を導入することで、拡張まで含めたコンポーネントベースのプログラミングを可能としている。

Nuts コンポーネントは、オブジェクト指向プログラミングの 4 つの特徴 (オブジェクト、カプセル化、継承、多相性) を備えており、大規模で柔軟なソフトウェアを迅速に開発する必要のあるプログラマ、およびそれを保守するプログラマにとって有効である。

以下の章では、ホワイトボックス的なプログラミングが可能なコンポーネント、およびコンポーネント差分プログラミングを実現する Nuts のアーキテクチャについて述べる。