ベクターコンポーネント: コンポーネント結合による差分プログラミング

上田哲郎[†,††] 久野靖[†]

本稿では，拡張性と柔軟性において，従来のサブクラス化による方法に優るコンポーネントベースの差分プログラミングについて述べる．コンポーネントベースの開発において，プログラムの再利用のために拡張機能の差分のみをコーディングしてプログラムに追加しようとした場合，従来のやり方では，クラス定義に戻ってサブクラス拡張を行わなければならず，クラス階層を熟知した上級のプログラマでなければ困難であった．本稿で提案するコンポーネント差分プログラミングでは，拡張のための差分のみを持ったコンポーネント（ベクターコンポーネント）を開発し，それをプログラム中に挿入できる．本方式の基盤となるアーキテクチャとして，筆者らは木構造をベースとした汎用的フレームワーク Nuts を開発した．Nuts では，部品組み立て型プログラミングをベースに複数の部品が組み合わさったものも一つの部品として振舞うようなフラクタルな構造を持つ．ベクターコンポーネントは，Nuts 上の他のコンポーネントに対して透明で存在しないかのように振舞い，フレームワーク中のどこにでも挿入できる．挿入されたベクターコンポーネントは，結合したコンポーネントに成りすまし，一部の制御を横取りすることにより拡張機能を追加する．

Vectors: A Component Architecture for Differential Programming

TETSURO UEDA[†,††] and YASUSHI KUNO[†]

This paper proposes a new component architecture which supports differential programming at component composition level. Although the conventional differential Object-Oriented Development techniques are based on subclassing at the source level which required both programming language skills and detailed knowledge of library classes, our approach overperforms the conventional ones in the sense that we can prepare special components: vectors, which can be freely inserted into existing component structures and incrementally modify their target (base) components. This is attained by the following reasons: we use tree-structured generic component architecture: Nuts. In Nuts, a group of components substitutes other components topologically similar. When a vector is inserted at the root of subtree, it is invisible from lower (root-side) components, however freely intercept and modify messages to the subtree in order to extend its behaviors. Vectors are quite effective in adding various functions to GUI-based components. Thus, they are valuable tools to construct functional-rich component-based programs through incremental development.

1. はじめに

現在のソフトウェア開発では，C++[12] や Java[8] などのオブジェクト指向言語が普及し，それらに基づいたクラスライブラリやコンポーネントの再利用による効率的なソフトウェア開発が可能となっている．

しかし，クラスライブラリ[6] のようにクラスを用いる方法（ホワイトボックス再利用）では，サブクラスを作成して変数，メソッドの追加やオーバーライドが行えるため拡張性が高い反面，クラス構造を熟知しなければ使いこなせないという難点がある．この問題を緩和する手段として，デザインパターン[5] のようなプログラム構造に対するヒントや，フレームワーク[14][10] による構造のモデル化が試みられているが，見通しのよいプログラムを短期間で開発するという観点では十分でない．

一方，クラスのインスタンスを部品（コンポーネント）として用いる方法（ブラックボックス再利用）では，GUI ベースの開発ツールを用いて，コンポーネントレベルでのプログラム構造の把握／変更が容易に

† 筑波大学経営政策科学研究科企業科学専攻
Graduate School of Systems Management, University of Tsukuba
†† 日産自動車 (株) 電子情報研究所
Nissan Motor, Co., Ltd. Electronics and Information Systems Research Laboratory

行える[20)15)]．しかし，インスタンスの生成プロセスやインスタンス結合関係の仕様は，部品の種類毎にまちまちであり，生成のための予備知識や手動によるインスタンス間の結線の必要性がある．すなわち，プログラミングを必要としなくなっても，それによって直ちにプログラムの開発が容易になるとは言えない面がある．

そこで筆者らは，クラスを対象としたプログラミングによる拡張性を維持しながら，インスタンスの単純な組合せによりプログラムの構築が可能なコンポーネントアーキテクチャNutsを考案した[18)19)]．Nutsのフレームワークに従ったプログラムは木構造を持ち，コンポーネントの結合関係を容易に見通せる．また，コンポーネント間の通信は，動的な通信相手の探索によるため，プログラマがコンポーネントの通信仕様に合わせて手動で結線する必要がない．Nutsについてのより詳しい解説は19)に記した．

著者らはさらに，Nutsフレームワーク上で，コンポーネントレベルでの差分プログラミングが可能な手法を考案した．本手法では，拡張機能の差分を組み込んだコンポーネント（ベクターコンポーネント）をプログラム構造中に挿入し，既存のコンポーネントに結合させることでコンポーネントの持つ機能を拡張できる．ベクターコンポーネントによる拡張は，委譲をベースとしており，サブクラス化による方法と同程度に強力である．また，ベクターコンポーネントもNutsのコンポーネントアーキテクチャに従っており，他のコンポーネントと同じ形でプログラム中に存在するため，拡張機能も含めてプログラム構造の把握や変更が容易である．

以下，第2章ではオブジェクト指向をベースとしたプログラミングの問題点について整理する．第3章では，Nutsのコンポーネントアーキテクチャの詳細について，第4章では，コンポーネント差分プログラミングについて，第5章では，ベクターコンポーネントの例について，第6章では，関連研究を取り上げ，最後に議論とまとめを行う．

2. オブジェクト指向に基づくプログラム開発

クラスをベースとしたオブジェクト指向プログラミングでは，図1に示すように，クラスの設計を行う“a-kind-of”関係構築とインスタンスの結合を行う“a-part-of”関係構築の2側面があり[3)]，プログラマはそれぞれの側面でプログラムを設計する．以下に各側面でのプログラム開発について整理する．

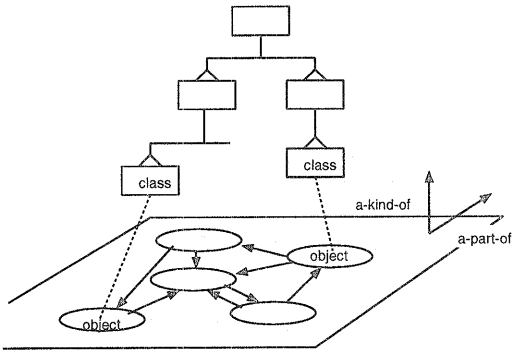

図1 オブジェクトの開発側面
Fig. 1 Two sides of developing objects.

2.1 “a-kind-of”開発側面

開発の“a-kind-of”側面では，クラス階層を構築する．プログラマは，スーパークラスの実装を継承したサブクラスを生成し，サブクラスでスーパークラスとの差分のみを記述することで（差分プログラミング），効率のよいプログラム開発が可能である．Smalltalk-80システム[6)]やJavaのクラスライブラリ[8)]を用いた独自クラスの開発がこれに相当する．“a-kind-of”側面での開発には，以下のような問題がある．

- 将来の拡張に備えたクラス階層の構築は困難
- 複数のスーパークラスの性質を合成するのは簡単でない（多重継承の問題）
- サブクラス化によりクラスのカプセル化が壊れる（クラスの脆弱性の問題）

これに対して，Javaなどで用いられているクラスのインタフェースのみを継承する方法では，多重継承や脆弱性の問題は回避できるが，差分プログラミングのメリットを享受できない．一般に，“a-kind-of”関係の設計はクラス階層を十分に理解し，実装とインタフェースの使い分けが可能な上級のプログラマでなければ行えない．

2.2 “a-part-of”開発側面

開発の“a-part-of”側面では，プログラマは，ある程度汎用的に作られた部品クラスのインスタンス（コンポーネント）を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．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．Active-X[7)]やJavaBeans[4)]などコンポーネントウェアと呼ばれるコンポーネント群を用いたプログラム開発がこれに相当する．しかし，コンポーネントに基づく開発には以下のような問題がある．

- 不特定多数のコンポーネントと結合させるため


```
// app.C
#include <NutsAll.h>
newNutsApp(app);
newNutsShell(app,shell);
newNutsStdout(app,out);
```

図 2 構造の記述例
Fig. 2 An example of coding.

の結線が煩雑であり，かつ容易に結合状態が見通せない

- 一般に大規模ソフトウェアでは，コンポーネント相互の依存関係は複雑に絡み合っており，プログラムの拡張をコンポーネント間の依存関係を維持しながら，a-part-of の側面だけで行うのは困難である．

これに対し，コンポーネントの結合構造に対するフレームワーク（MVC フレームワークなどが代表的）[10)14)] やパターン [5)]，を用意することで，構造の見通しを良くする方法があるが，これらがカバーするのはプログラムの一部であり，プログラム全体を一つのフレームワークに基づいて構築することは難しい．

まとめると，“a-part-of”,“a-kind-of” の分業は確かに望ましいが，現状ではどちらにも問題があり，またプログラマはどちらにも精通しなければならない．

なお，クラスとサブクラスではなく，コピーと委譲に基づくオブジェクト指向言語 [13)11)] もあるが，あるオブジェクトを雛型として新しいオブジェクトを定義する部分は概念的に “a-kind-of” に相当するので，前述の問題点があてはまる．

3. Nuts のアーキテクチャ

前章で説明したオブジェクト指向プログラミングにおける問題点を克服する試みとして，著者らが開発したコンポーネントアーキテクチャNuts について述べる．Nuts では，クラスをベースとしたオブジェクト指向言語によって “a-kind-of” 側面での差分プログラミングによる拡張性を維持しつつ，コンポーネント結合方法の統一，委譲による拡張など “a-part-of” 側面での理解容易性／拡張性を高めている．

Nuts のコンポーネントアーキテクチャについては 19) に詳しいが，議論の都合上，コンポーネントの結合関係および通信手段を，それぞれ Nuts の構造モデル／制御モデルとして以下に説明する．

3.1 構造モデル

Nuts ではコンポーネント間の静的な関係を次のようにモデル化する．

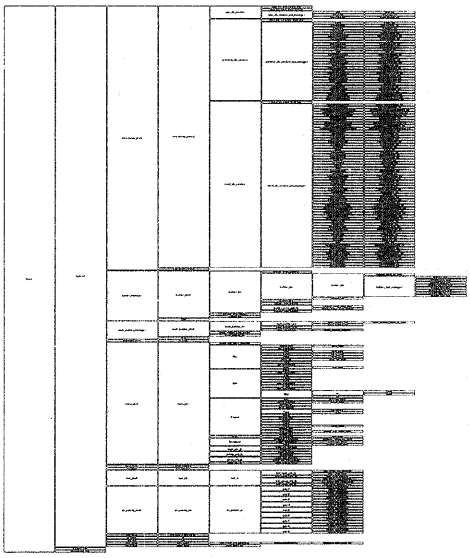

図 3 Nuts/Builder の構造 (左がルート)
Fig. 3 The structure of Nuts/Builder.

- コンポーネント間の参照関係を親子兄弟関係のみの局所的な参照に限定する．
- 各コンポーネントは単一の親と，ゼロ個以上の子を持つ（全体構造は木構造となる）．
- 親子関係の接続の可否はコンポーネントのクラスによって定まり，構築時に自動的に検査される．
- すべてのコンポーネントはルート（根元にあるコンポーネント）を参照できる．
- すべてのコンポーネントは固有名を持つ．

Nuts のプログラム構造は，例えば図 2 上の構造に対して，図 2 下のように「部品クラス名 (親部品固有名, 部品の固有名);」（実際には C++のマクロ呼び出し）という記述を列挙することで定義できる．大規模なプログラムもすべてこの部品の親子関係の列挙だけで構築できるため，プログラムの全体構造（“a-part-of” 関係）を容易に把握できる．例えば，Nuts/Builder は，GUI 上でアイコンを積み上げることにより，図 2 と同等の記述を生成できるツールである．Nuts/Builder 自体は，Nuts コンポーネント 381 個から構成されており，その様子は図 3 に示すようにコンポーネントの木構造として視覚的に把握できる．

3.2 制御モデル

Nuts では，前述の木構造に沿ってメッセージを伝搬させることで，隣接していないコンポーネント間の相互作用（メッセージ通信）を可能にしている．具体的には，親子兄弟の 4 方向（図 4）への探索により通信相手を決定できる．探索の方法（宛先）としては，

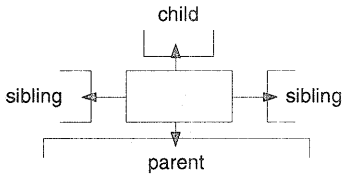

図4 メッセージ探索方向
Fig. 4 The direction of messages.

以下のものが使用できる．

- 指定した名前を持つ部品
- 指定した名前のクラスに属する部品
- 指定した名前のクラスまたはそのサブクラスに属する部品
- すべての部品
- 以上の論理式による組合せ

探索の結果，通信相手が見つからない場合，メッセージは破棄される．メッセージには，宛先，内容，引数，オーナー（発信元コンポーネントへの参照），返信の宛先（後述するメッセージ横取りの際にはオーナーとは異なる）が含まれる．また，メッセージを受理したコンポーネントは，返り値として自身への参照を返すようになっている．探索によるメッセージ通信を用いることで次の利点が得られる．

- 通信相手のコンポーネントの位置や存在の有無を意識する必要がない．
- 制御への副作用を気にすること無しに構造が変更できる．

一方 Nuts の制御モデルは，探索ための時間コストを必要とするが，以下の方法によって時間コストを軽減できる．

- 受信者がいる方向（親子兄弟のいずれか）がわかっている場合，探索方向（図4）を指定し探索範囲を絞る．
- 探索によって受信者への参照を一度だけ取得し，以後その参照を使い，探索を行わない（ただし参照先が移動／消滅すると致命的エラーとなる）．
- 通信の頻繁な受信者への参照をキャッシュする．

これまでのアプリケーション開発事例では，以上の方法の組合わせよって，時間コスト的な問題は発生していない．

4. コンポーネント差分プログラミング

第2章で示したように，Nuts においても，コンポーネント自体の拡張は，“a-kind-of” の側面において，サブクラス化によって行わなければならない．すなわち，Nuts フレームワークは，“a-part-of” 関係の見通しを良くするが，それとは別次元の（図1）“a-kind-of” 側面における拡張性の向上には直接役立っていない．

そこで筆者らは，拡張機能の追加や変更も “a-part-of” 関係で記述可能な手法である「コンポーネント差分プログラミング」を考案し，Nuts フレームワークに組み込んだ．これにより，“a-kind-of” 関係を理解せずとも拡張機能の追加や変更が可能であり，同時に拡張機能も含めてプログラムの構造を容易に把握／編集できるようになる．

4.1 ベクターコンポーネント

本手法では，ベクターコンポーネント（以下ベクター）と呼ぶ，拡張機能の運搬を担う特別なコンポーネントを用いる．ベクターは，Nuts のコンポーネントアーキテクチャに従っており，透明である点を除いて “a-part-of” の側面で他のコンポーネントと区別されない．ベクターは，機能的には委譲による修飾と類似しているが，他のコンポーネント同様，木構造に挿入することで，ベクターをルートとする部分木全体に対する修飾が可能であり，また Nuts コンポーネント用のツールで取り扱うことができる．

ベクターの要件として以下の2つが挙げられる．

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

これらの要件を実現するために，以下に述べるメッセージの横取り機構，および透明なコンポーネントを導入した．

4.2 メッセージの横取りによる制御の例外

Nuts コンポーネントの仕様を，本来の宛先コンポーネントから，メッセージを横取りできるように拡張した．メッセージの横取りには，

- ある部品からメッセージを横取りすることで別のコンポーネントがその部品に成りすます．
- あるコンポーネントがマルチプレクサとして振舞い，状態に応じて複数のコンポーネントにメッセージを振り分ける．
- 受信の可否を外部から制御する．

などの使い方があり，メッセージの流れを動的に変化させられる．メッセージ横取り機構は，具体的には以下のような仕様を持つ．

- 横取りは，コンポーネントに「横取り開始」メッセージを送ることで設定する．
- 横取りは，コンポーネントに「横取り中止」メッセージを送ることで解除する．
- メッセージが横取りされると，メッセージのオーナーは横取りされたコンポーネント（本来の受信

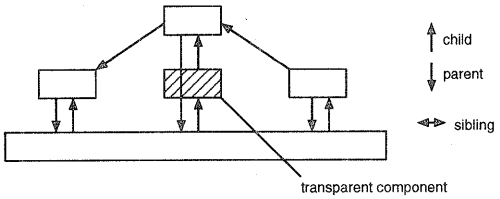

図 5 透明コンポーネントの親子関係
Fig. 5 The relationship of a transparent component.

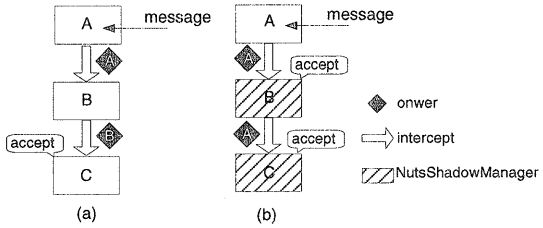

図 6 透明コンポーネントからの横取り
Fig. 6 Message interception from a transparent component.

者）に変更される．

- 横取り先で認識できない（対応するメソッドが存在しない）メッセージは，横取りされたコンポーネント（本来の受信者）で改めて受信される．

4.3 透明コンポーネントと構造の例外

通常の Nuts コンポーネントは，結合時に親の条件（Window の親は Window でなければならないなど）が自動的に検査され，条件を満たさない場合は結合できない．これを拡張して，前記の構造／制御モデルに若干の例外を設けたコンポーネント（透明コンポーネント）を用意し，親の条件とは無関係にどこにでも挿入可能にした．透明コンポーネントは，一般のコンポーネントと同じ形をしているが，構造上も制御上も存在しないように見えるコンポーネントである．

透明コンポーネントは，それと結合する他のコンポーネントに対して，図 5 のような参照関係を構築する．この図のように，透明コンポーネントはその親コンポーネントから見れば通常通り存在しているが，その子／兄弟コンポーネントからは存在しないように見える．これにより，透明コンポーネントは親子関係を気にすることなくプログラム中に組み込める．

4.4 メッセージの横取りと透明コンポーネント

透明コンポーネントでは，メッセージ横取り機構の仕様を次のように変更する．

- メッセージを横取りしても，そのメッセージのオーナーを変更しない．
- 横取り先での認識の可否にかかわらず，常にメッセージを受取り解釈する．

これにより，透明コンポーネント宛のメッセージを横取りすると，その後のメッセージの探索順序は通常と異なったものになる．

通常は，図 6(a) のように A で受け取られたメッセージは，C で最初に解釈され，そのときのオーナーは B である．また，メッセージが C で受理されるとそこで破棄される．一方透明コンポーネントの場合は，図 6(b) のように C が解釈するときのオーナーは A のままである．また，C でメッセージが受理されても B にもメッセージ解釈の機会がある．

すなわち，透明コンポーネントはメッセージの横取り過程においても存在しないかのように振舞うため，透明コンポーネントを多段に積み重ねても，メッセージの横取り（受理の機会とメッセージのオーナー）に関してすべての段が同等に扱われる．これにより，一つのコンポーネントを複数のベクターで多重修飾することが可能になる．

4.5 ベクターと被修飾コンポーネント

ベクターは，それが修飾するコンポーネントの種類（あるクラス以下のサブクラス）が決まっている．このことを利用して，必要なら被修飾コンポーネントの内部に直接アクセスすることも可能である（が乱用は望ましくない）．ベクターは，挿入位置を根とするサブツリー中で修飾対象のクラスに属するコンポーネントを探索し（図 4 の child 方向探索による），探索の結果見つかったコンポーネントに結合する．

ベクターは，結合したコンポーネント（被修飾コンポーネント）のメッセージを自動的に横取りするが，その際，被修飾コンポーネントのクラスに共通の振舞いを前提に拡張する．すなわち，被修飾コンポーネントをサブクラス拡張したものにも同じベクターが使える．

4.6 サブクラス拡張との比較

本節では，ウインドウへのスクロール機能の付加と，ウインドウ上でのマウスクリックのハンドリングを例に，サブクラス拡張とコンポーネント差分プログラミングを比較する．

4.6.1 サブクラス拡張の場合

サブクラス拡張では，ウインドウを表す Window クラスから ScrollWindow サブクラスを派生する．ScrollWindow クラスでは，描画原点を表す変数 `_xbase`，`_ybase` を保持しスクロール機能を実現する（図 7）．

ScrollWindow クラスのサブクラス MyWindow ク

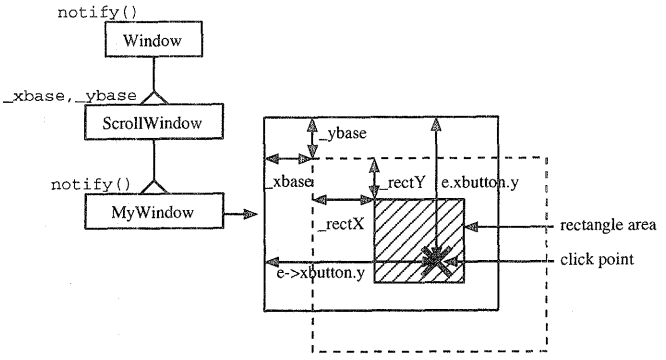

図 7 サブクラス拡張の場合
Fig. 7 Subclassing.

```
MyWindow::notify(XEvent *e){
   int x = e->xbutton.x+_xbase;
   int y = e->xbutton.y+_ybase;//原点のずれを考慮
   if(_rectX < x && x < _rectX+_rectWidth
      && _rectY < y && y < _rectY+_rectHeight){
         fprintf(stdout,"clicked.\n");
   }
}
```

図 8 サブクラス拡張の場合のソースコード
Fig. 8 The source with subclassing.

ラスは，ウインドウ上に四角形領域を描画し，その位置情報（_rectX，_rectY，_rectWidth，_rectHeight に保持）により，マウスが四角形領域内でクリックされたときに"clicked"という文字列を出力する．

MyWindow クラスのプログラマは，マウスクリック時に Window クラスのnotify() が呼ばれることを知った上で，図 8 のようにnotify() をオーバーライドする．

MyWindow クラスのプログラマは，クリック判断時に，スーパークラスである ScrollWindow クラスの _xbase，_ybase による原点のずれを考慮する必要があり，スーパークラスの仕様に立ち入らなければならない．

同様に，MyWindow からスクロール機能を除きたい場合は，MyWindow クラスのスーパークラスを ScrollWindow から Window に変更するが，同時に notify() 中の，_xbase，_ybase も除く必要がある．

このように，サブクラス拡張では，スクロール機能の追加や削除に合わせて，スーパークラスの仕様を考慮した上で，被修飾側のコードを変更しなければならない．

4.6.2 コンポーネント差分プログラミングの場合

コンポーネント差分プログラミングでは，MyWindow を Window クラスのサブクラスとして定義し，スクロール機能は，ウインドウへの修飾コンポーネントとして Scroller ベクターに定義する（図 9）．

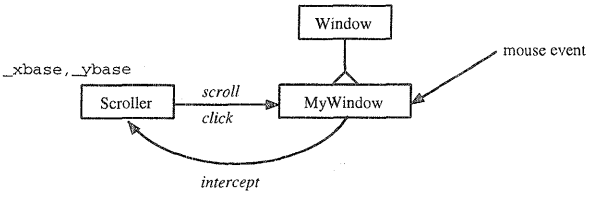

図 9 ベクターとサブクラスの関係
Fig. 9 A vector attached to a subclass instance.

```
MyWindow::notify(XEvent *e){
   int x = e->xbutton.x;
   int y = e->xbutton.y;
   if(_rectX < x && x < _rectX+_rectWidth
      && _rectY < y && y < _rectY+_rectHeight){
         fprintf(stdout,"clicked.\n");
   }
}
```

図 10 コンポーネント差分プログラミングの場合のソースコード
Fig. 10 The source with a vector component.

MyWindow のnotify() は図 10 のようになる．

Scroller クラスは，描画原点を表す変数_xbase，_ybase を保持し，Window 部品に通知されるマウスのドラグメッセージを横取して，結合した Window 部品をスクロールする．

マウスのクリック時には，Window 部品に Nuts メッセージが通知されるが，このメッセージも Scroller に横取りされ，クリック座標に_xbase，_ybase の値が加算される．その上で，Scroller は上記メッセージを本来の受信者である Window 部品に送る．従って，図 10 のコードは変更の必要がない．

このように，スクロール機能の追加／削除は，Scroller の結合で制御できるため，被修飾部品である Window クラスに手を加える必要はない．

また，サブクラス拡張の例では，プログラマがスー

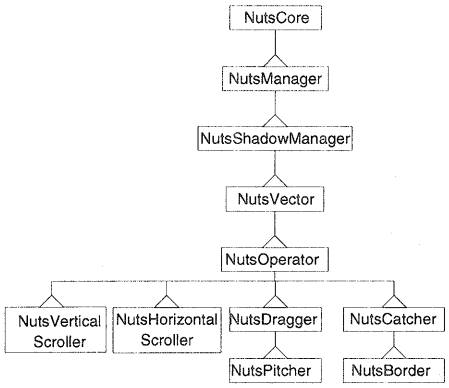

図 11 透明コンポーネントのクラス階層
Fig. 11 The class layer.

パークラスで定義されている`notify()` の存在を知っている必要があったが，Nuts では，マウスのクリックメッセージの受け口を（Window クラスの受け口とは無関係に）MyWindow クラスに設ければよく，クラス階層を熟知している必要がない．

以上のように Nuts では，部品の相互作用をメッセージレベルで記述することにより，クラスの階層構造や被修飾コンポーネントクラスの仕様に立ち入らずに拡張が可能である．

4.7 ベクターによる多重修飾

実装を継承するサブクラス拡張では，複数のクラスの性質を合成するために多重継承を用いるが，それには以下のような問題がある．

(1) 名前の衝突や，変数スペースの共有などの解決がプログラマに任されるため，記述が煩雑
(2) 複数のクラスを合成する場合，クラス数の爆発が起きる

これに対して，ベクターを多段に重ねることによる多重修飾では，各々のベクターは，それぞれ固有の変数やメソッドを持つため，(1) のような問題は発生しない．

なお，インタフェースのみを継承する多重継承でも前記の問題は発生しないが，共通部分の共有もできない．ベクターでは，ベクターのクラスをサブクラス拡張することにより，複数のベクターで性質の共有（次章の NutsOperator がこれに相当）が可能である．

また，(2) の問題に対しても，ベクターを用いた多重修飾では，合成したい性質を持ったベクターを多段に重ねればよく，スケーラビリティを確保できる．

5. ベクターの例

本章では，ベクターの例としてウインドウコンポーネントに通知されるマウスの press，motion，release の各イベント（Nuts のメッセージとして実装）を横取りして GUI 機能の拡張を行うベクター群を取り上げる．

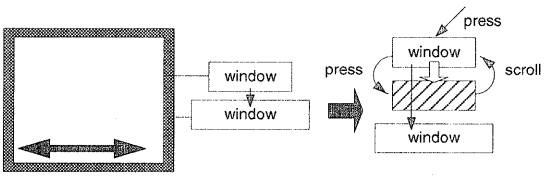

図 12 Scroller による修飾
Fig. 12 Decoration by Scroller.

まず，そのようなベクターのベースクラスとして NutsOperator クラスを定義する（図 11）．NutsOperator クラスは NutsVector クラスのサブクラスであり，部品木中のどこにでも侵入し，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できる．

5.1 ウインドウをスクロールするベクター

NutsWindow はウインドウ領域を提供し，マウスの press，motion，release の各イベントを受け取るクラスである．一方，NutsHorizontalScroller は，NutsOperator のサブクラスであり，NutsWindow に水平方向スクロール機能を追加するベクターである．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侵入させることで（図 12），NutsWindow コンポーネントに水平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，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 13 に示す．NutsOperator クラスが，マウスの press，release のイベントを横取りして実際にマウスがドラグ中かどうかを判断するので，そのサブクラスである NutsHorizontalScroller では，ドラグ中の motion イベントだけを記述すれば良い．同様に，スクロールの原点情報（図 13 中`_xbase()` の返す値）も NutsOperator が保持しており，NutsHorizontalScroller など NutsOperator のサブクラスではそれを共有できる．

5.2 修飾の多重化

同様の手法で垂直方向にスクロールさせる NutsVer-

```
#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 13 NutsHorizontalScroller
Fig. 13 NutsHorizontalScroller.

ticalScroller ベクターを開発できる．さらに，NutsHorizontalScroller と NutsVerticalScroller を積み重ねて多重に修飾すると，水平／垂直両方向へのスクロールが可能になる．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と，“a-kind-of” 側面における多重継承と同様の機能を，コンポーネントの結合で実現できる．

5.3 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 コンポーネントの間に侵入させ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．NutsDragger のサブクラスである NutsPitcher ベクターに侵入されたアイコンは，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，その親／兄弟コンポーネントに対して（衝突相手が親兄弟中にいるので探索範囲を絞っている）定期的に移動メッセージを送る．

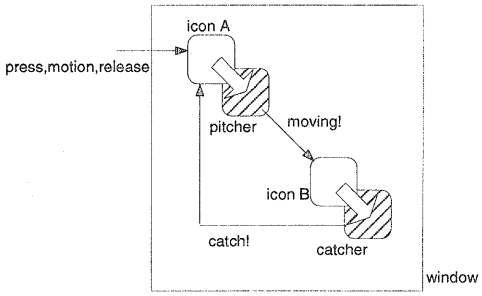

図 14 アイコン投げの制御
Fig. 14 Controlling icon throwing.

5.4 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．

NutsCatcher ベクターは自身が修飾しているアイコンの位置と移動中の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 14）．捕獲メッセージに質量，弾性係数などを含めると（跳ね返りなどの）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．NutsBorder 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．

5.5 ベクターを用いたアプリケーション開発事例

ベクターを用いて，アイコン投げシェル [16)17)] を模した GUI シェル（図 15）を開発した．この GUI シェルは次の機能を持つ．

- 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）．

これらの動作は，前章で説明した画面スクロール，アイコン投げ，アイコン衝突などの拡張機能を持った

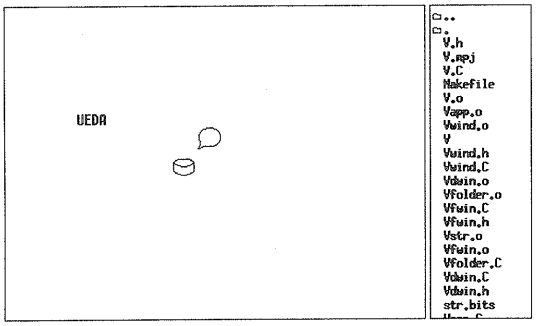

図 15 GUI シェル
Fig. 15 The icon throwing shell.

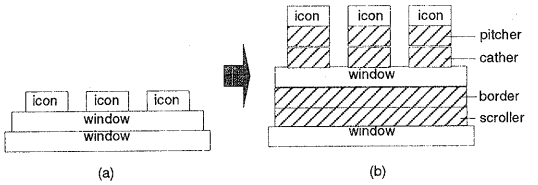

図 16 ベクターの侵入
Fig. 16 The interception of vectors.

ベクターを図 16(a) の構造に対して図 16(b) のように侵入させることにより実現できる．

6. 関連研究

“a-part-of” の側面でコンポーネントを拡張する方法として，パターンカタログ [5] に示されている Decorator パターンが挙げられる．Decorator パターンは，Decorator コンポーネントで，被修飾コンポーネントを包み込むことにより，外部（クライアント）からのアクセスに対して，Decorator コンポーネントの存在を意識させない修飾方式である．

この方式では，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修飾できない．

これに対して，コンポーネント差分プログラミングでは，ベクターが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．このため本来，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，ベクターによる横取り対象となる．これにより，自分自身にメッセージを送るメソッド（抽象メソッド）では，ベクターによってメッセージを横取りし解釈することで，部分的に機能の差し替えが行える．

Garnet システム [9] で用いられている interactor は，interactor を用いて GUI の機能を差し替えるという点でベクターに類似している．interactor は，（MVC フレームワークのコントローラに相当し）GUI とアプリケーションの実装を分離するためのものであり，すべてのコンポーネントをメッセージ通信とその横取りによって拡張できるベクターの方がより適応範囲が広い．また，interactor 自体の機能は，GUI 部品へのイベント配信であり，イベントに対する反応は GUI 部品側で個別に行われる．この点で interactor は，部品の拡張を目的としてるベクターとは性格が異なる．

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．OpenC++[1)2)] に代表されるコンパイル時 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，本稿で提案しているベクター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．

7. まとめ

“a-part-of” 側面でのプログラム拡張が可能な手法として，コンポーネント差分プログラミングについて述べた．コンポーネント差分プログラミングは，コンポーネントレベルでプログラムを拡張するのに十分な機構であり，実際のアプリケーション開発に役立っている．

現在のコンポーネント差分プログラミングは，Nuts

のメッセージ呼び出しに相当する部分のみ拡張可能であり，コンポーネント内部でのメソッド呼び出しなど，Nutsのメッセージ機構を用いていない部分は拡張対象となっていない．この問題について，コンパイル時MOPを用いて，通常のメソッド呼び出しをベクターで横取り可能なNutsのメッセージ呼び出しへ変換する方法が考えられ，今後の検討課題である．

謝辞

研究に際して適切なアドバイスをくださった，筑波大学経営政策科学科の松本正雄教授，同寺野隆雄教授，日産自動車(株)基礎研究所前所長高尾 洋氏（故人），同電子情報研究所岸則政シニアリサーチャに感謝します．

参 考 文 献

1) Chiba, S.: A Metaobject Protocol for C++, *in Proc. of the ACM Conference of Object-Oriented Programming Systems, Languages, and Applications*, pp. 285–299 (1995).
2) Chiba, S.: Open C++2.5 Reference Manual, Technical report, Institute of Information Science, University of Tsukuba (1997).
3) Coad, P. and Yourdon, E.: *Object-Oriented Analysis*, Youdon Press, Englewood Cliffs, NJ (1990). (邦訳: 羽生田栄一監訳,「オブジェクト指向分析 (OOA)」, トッパン, 1993).
4) Feghhi, J.: *Web Developer's Guide to Java Beans*, International Thomson Publishing (1997). (邦訳: 豊福 剛訳,「JavaBeans 入門」, オーム社, 1997).
5) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Micro-Architectures for Reusable Object-Oriented Software*, Addison-Wesley (1994). (邦訳: 本位田真一, 吉田和樹監訳,「オブジェクト指向における再利用のためのデザインパターン」, ソフトバンク, 1995).
6) Goldberg, A.: *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA (1984). (邦訳: 相磯秀夫監訳,「Smalltalk-80 対話型プログラミング 環境」, オーム社, 1986).
7) Kruglinski, D. J.: *Inside Visual C++*, Microsoft Press (1996).
8) Lemay, L. and Perkins, C. L.: *Teach Yourself Java in 21 days*, Sams.net (1996). (邦訳: 武舎広幸, 久野禎子, 久野靖訳,「Java 言語入門」, プレンティスホール, 1996).
9) Myers, B. A., Giuse, D., Dnneyberg, R. B., Zanden, B. V., Kosbie, D., Pervin, E., Mickish, A. and Marchal, P.: Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment, *IEEE Computer*, Vol. 23, No. 11, pp. 71–85 (1990).
10) Prosise, J.: *Programming Windows95 with MFC*, Microsoft Press (1997). (邦訳: 榊正憲監訳,「MFC による Window95 プログラミング」, アスキー, 1997).
11) Smith, R. B. and Ungar, D.: Programming as a Experience: The Inspiration for Self, *ECOOP'95 Conference Proceedings* (1995).
12) Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley (1993).
13) Ungar, D. and Smith, R. B.: Self: The Power of Simplicity, *OOPSLA'87 Conference Proceedings*, pp. 227–241 (1987).
14) Wilson, D. A., Rosenstein, L. S. and Shafer, D.: *Programing with MacApp*, Addison-Wesley, Reading, MA (1990).
15) 柿山薫, 小山清美: コンポーネントウエアを適用した開発事例と開発方法論, 情報処理学会 OO シンポジウム'96 オブジェクト指向最前線, pp. 135–138 (1996).
16) 久野靖, 角田博保, 大木敦雄, 粕川正充: アイコン投げシェル, 第 38 回プログラミングシンポジウム, pp. 65–74 (1997).
17) 久野靖, 大木敦雄, 角田博保, 粕川正充:「アイコン投げ」ユーザインタフェース, コンピュータソフトウエア, Vol. 13, No. 3, pp. 38–48 (1996).
18) 上田哲郎, 久野靖: Nuts—柔軟な部品間結合をサポートするコンポーネントアーキテクチャ, 情報処理学会情報システム研究会 (1997).
19) 上田哲郎, 久野靖: Nuts—ホワイトボックスコンポーネントアーキテクチャ, 電子情報通信学会論文誌, 印刷中 (1999).
20) 青山幹雄: コンポーネントウエア：部品組み立て型ソフトウエア開発技術, 情報処理学会誌, Vol. 37, No. 1, pp. 71–79 (1995).

(平成 11 年 3 月 10 日受付)
(平成 11 年 4 月 28 日採録)

上田 哲郎（正会員）

1964 年生．1990 年九州大学大学院総合理工学研究科情報システム学専攻修了．同年日産自動車(株)入社．現在同電子情報研究所勤務．1996 年筑波大学経営政策科学研究科企業科学専攻博士後期課程修了，博士（システムズ・マネジメント）．情報処理学会会員．

久野　靖（正会員）

1956年生．1984年東京工業大学理工学研究科情報科学専攻博士後期課程単位取得退学．同年東京工業大学理学部情報科学科助手．筑波大学経営システム科学専攻講師を経て，現在同助教授．理学博士．プログラミング言語，プログラミング環境，オペレーティングシステム，ユーザインタフェース等に興味を持つ．著書に「言語プロセッサ」(丸善),「UNIXの基礎概念」(アスキー) 等がある．情報処理学会，日本ソフトウェア科学会，ACM，IEEE Computer Society 各会員．