Chapter 3

Nutsアーキテクチャ

本章では、柔軟なソフトウェア開発のための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る、Nutsコンポーネントのアーキテクチャについて詳説する (Nutsコンポーネントを用いたコンポーネント差分プログラミングについては、第6章で述べる)。

第2章で、従来からあるコンポーネントをベースとしたプログラミングにおいて、記述能力、理解容易性、拡張性などの観点から、コンポーネント間の相互関係 (結合や通信) の構築手法について論じた。本章では、コンポーネント間の相互関係をコンポーネント間の静的な結合関係の記述 (構造モデル) と、実行時における動的なコンポーネント間の相互作用の記述 (制御モデル) という二側面に分解し、それぞれをモデル化する。以下にNutsの構造モデル／制御モデルについて説明する。

3.1 Nutsの構造モデル

Nutsでは、部品の組み立てに木構造を用いている。第1.7節で示したように、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造をもっている。Nutsでは、木構造のこのような特徴を利用し、部品は単体で木構造中の構成要素になると同時に、部品が集まってできた部分木も、また部分木が集まってできたより大きな部分木も、単体の部品と構造上区別することなく統一的に扱えるようになっている。また、部分木の追加／削除／取り替えが可能で、機能の変更が容易に行える。

Nutsコンポーネントでは、以下のようなコンポーネントモデルにより、コンポーネントの結合が木構造を形作るようにしている。

- 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 (図3.2)。
- 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。

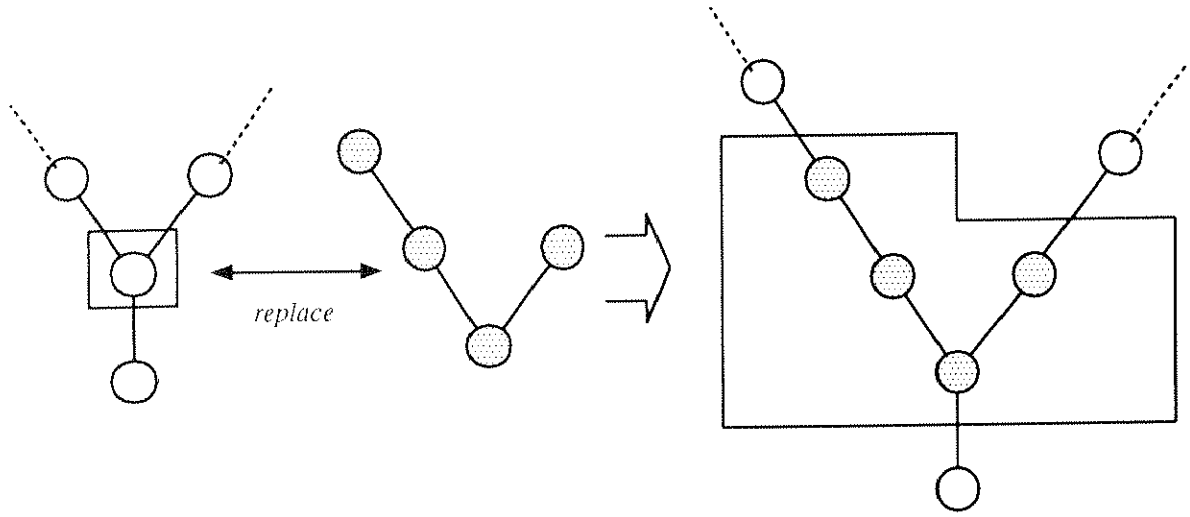


図 3.1: Nuts の木構造

- 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 のアプリケーション構造は、例えば図 3.3上の構造に対して、図 3.3下のように

「new コンポーネントクラス名 (親の固有名, 自身の固有名);」[1]

という記述を列挙することで定義できる (これを列挙したファイルを構造記述ファイルと呼ぶ)。構造記述ファイルを用いれば、より大規模なソフトウェアに対しても、親子関係による局所的な関係を列挙するだけで、全体構造を構築できる。

親子関係の列挙は、図 3.3のようなテキストをエディタを用いて直接記述することもできるし、Nuts/Builder[76](第 7.1節参照) のような GUI ツールを用いて、図 3.3上のようにコンポーネントを積み上げることにより、図 3.3下の構造記述ファイルを生成することも可能である。

また、アプリケーションの骨組みを構築するのに必要な部品を自動的に収集し、その構造記述ファイルを自動合成するユーティリティの実現も可能である。例えば nutsCreateMotifApp ユーティリティは、OSF/Motif ウジェットを用いたウインドウアプリケーションに必要な部品を収集し、図 3.4のような構造記述ファイルを出力する。これをコンパイル／実行すれば、OSF/Motif ウジェットを使った標準的なウインドウ (メニューバーやファイルメニュー

[1] この記述は、C++標準のプリプロセッサによりマクロ展開され、変数宣言とインスタンスの生成手続きに置き換えられる。
例えば `NewComponent(parent,child);` は、`NutsCore *child = new Component(parent,"child");` となる。

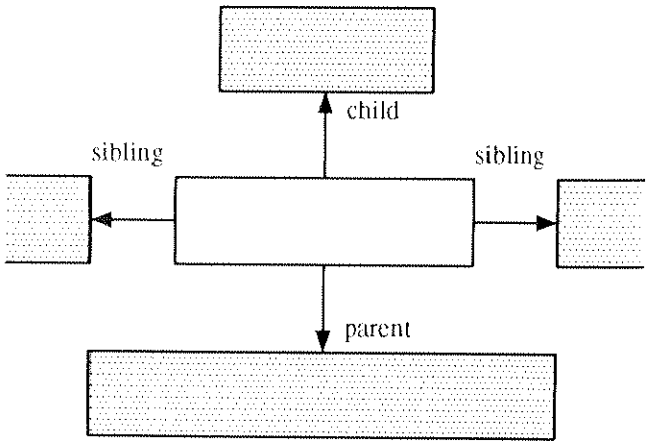
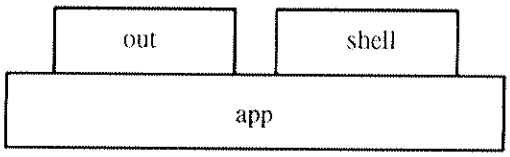


図 3.2: コンポーネントの親子関係




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

図 3.3: 構造の記述例

を持つ) が表示される (図 3.5)。実際のアプリケーションの開発では、この骨組みに対して、アプリケーション固有の部品を構造記述ファイルに追加する。

このような、プログラムの初期構造の自動生成機能は、Nuts 以外の開発フレームワーク (MFC の AppWizard 等) でも見られるが、Nuts の場合、その結果が構造モデルに従った構造記述ファイルの内容として、直観的に把握できる点が特徴である。

このように、構造をモデル化し、それを構造記述ファイルで管理することにより、アプリケーションの静的な構造の把握が容易に行える。さらに、構造記述ファイルの編集によりコンポーネントの追加、変更、削除などのプログラム拡張が簡単に行える。

```
#include <NutsAll.h>

newNutsXApp(motif);
newNutsMotifToplevel(motif,toplevel);
newNutsApplicationShell(toplevel,shell);
newNutsFormWindow(shell,fm);
newNutsMenuBar(fm,menu_bar);
newNutsPulldownMenu(menu_bar,File);
newNutsPushButton(File,About_this);
newNutsAlertCommand(About_this,alert_cmd);
newNutsSeparator(File,sp1);
newNutsPushButton(File,New);
newNutsPushButton(File,Open);
newNutsOpenCommand(Open,open_cmd);
newNutsPushButton(File,Save);
newNutsPushButton(File,Save_as);
newNutsSeparator(File,sp2);
newNutsPushButton(File,Print);
newNutsDoubleLineSeparator(File,sp3);
newNutsPushButton(File,Quit);
newNutsExitCommand(Quit,ext_cmd);
newNutsPulldownMenu(menu_bar,Edit);
newNutsUnmappedNoResizeApplicationShell(toplevel,alert_shell);
newNutsAlertWindow(alert_shell,alert_wind);
```

図 3.4: nutsCreateMotifApp の出力例

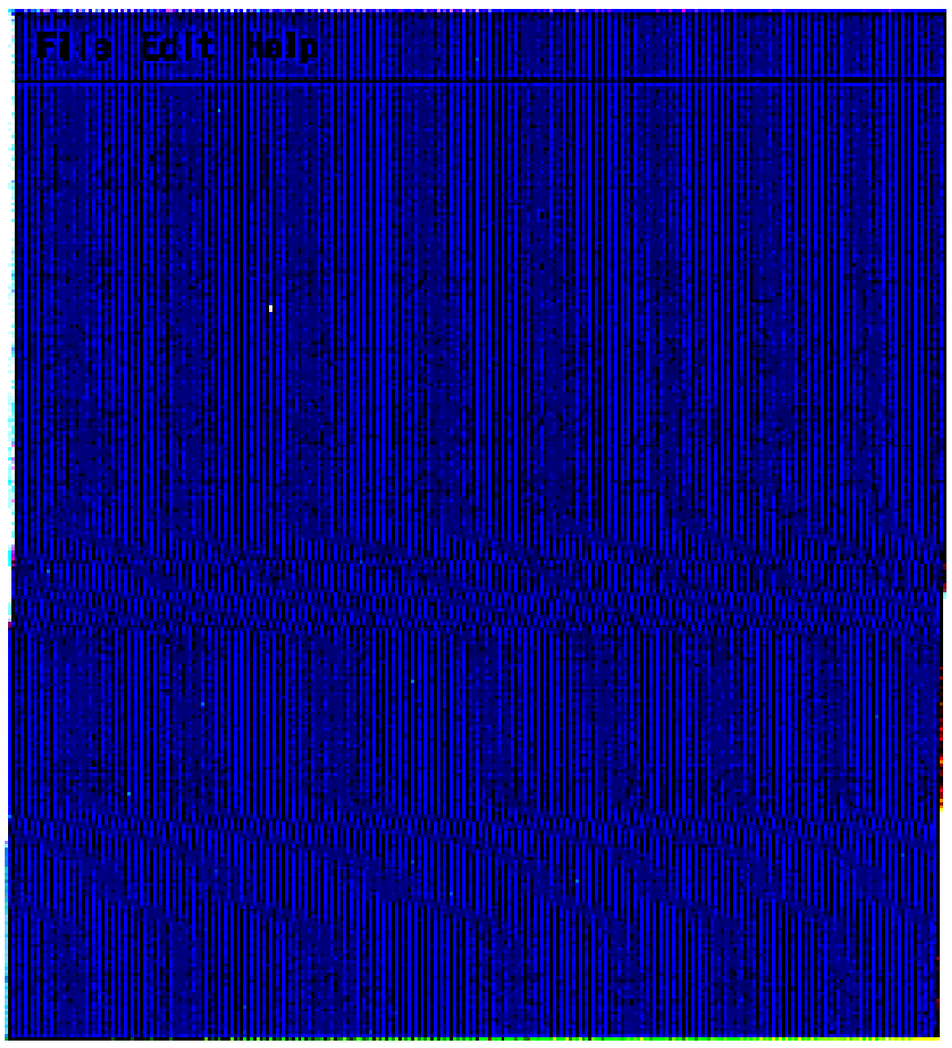


図 3.5: 自動生成されたウインドウ

3.2 Nuts の制御モデル

Nuts では、実行時における動的なコンポーネント間の相互作用の記述を制御モデルと呼んでいる。

プログラム実行中は、複数のコンポーネントが互いに通信する。コンポーネント間の通信を行うためには、通信相手への参照が入手できなければならない。しかし、それが Nuts の構造モデルで規定される親子兄弟関係だけに限定されたものならば、隣接していないコンポーネントとの通信が困難になる。

Nuts では、このような隣接していないコンポーネント間の通信のために、その都度プログラマがコンポーネントへの参照取得のための様々な仕組み (第 2.1 節参照) を用意する必要が無いよう、汎用的な通信手段として Nuts の制御モデルを用意している。Nuts の制御モデルでは、隣接していないコンポーネント間の通信は、通信相手の動的探索によって行う。

通信相手の動的な探索には、以下のような利点がある。

- 構造モデルに従って組み立てられたコンポーネント同士が直接隣接していなくても、相互にメッセージ通信を行える。
- プログラマは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くなり、プログラマの負担が軽減される。
- コンポーネント間の静的な構造と通信手段がある程度独立しているため、通信への副作用を気にすることなく構造の変更を行える。

Nuts の制御モデルには、メッセージモデルとイベントモデルがある。メッセージモデルは、キーによる動的な受信者の探索と呼び出しを、イベントモデルは、イベント受信者の登録によるイベント発生時の直接的な呼び出しを、それぞれサポートしている。以下にそれぞれのモデルを説明する。

3.2.1 メッセージによる制御

メッセージモデルは、キーをもとにオブジェクトを探索し、キーに当てはまったオブジェクトにメッセージを送る方式である。キーには、表 3.1のものが使用できる。

探索の方向として、探索開始点を中心に図 3.2の各方向に探索を行うメソッドがそれぞれ用意されている。探索開始点としては、自分が参照を保持している任意のオブジェクトが指定できる。典型的にはルートコンポーネントからの child 方向探索による部品木の全体探索を行う。また、自分自身を開始点とすれば、自分を根とする部分木の全体探索が可能である。

ルートコンポーネントを探索開始点にして、child 方向の探索の場合のメッセージの流れを説明する (図 3.6)。

1. 探索開始点から、部分木中に含まれるすべてのコンポーネントにメッセージが伝達される。

表 3.1: メッセージ探索キー

キー	記述例
指定した名前を持つコンポーネント	"shell"
指定した名前のクラスに属するコンポーネント	"/NutsShell"
指定した名前のクラスまたはそのサブクラスに属するコンポーネント	"//NutsShell"
特に指定なくすべてのコンポーネント	"*"
以上の論理式による組合せ	"shell&/NutsShell"

2. リーフに到達したメッセージはそれ以上伝達されず、リーフで消費されない場合は消滅する。

3. それぞれのコンポーネントに到着したメッセージは、コンポーネントの探索キーに一致している場合、受理される。

4. 受理されたメッセージは、(後述するメッセージの横取りが設定されていなければ) 受理したコンポーネントのクラスで解釈を試みられる。

5. 解釈された場合、メッセージは消費される (それ以上クラス階層の探索を行わない)。解釈できない (メッセージに対応した処理ルーチンが定義されていない) 場合は、スーパークラスに送られる。

6. メッセージは消費されるまでクラス階層をさかのぼり、基底クラス (NutsCore クラス) に至っても解釈されなかったメッセージは消滅する。

このように、メッセージモデルでは直交した2つの木構造 (部品木とクラス木) を探索することで、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決定する。また相手が見つからない場合はそのメッセージが消滅するだけで、致命的なエラーとはならない。

3.2.2 イベントによる制御

動的なコンポーネント探索を行うメッセージ通信モデルでは、探索のオーバーヘッドが根本的に避けられず、スケーラビリティが欠如している。これに対処するために、Nuts ではメッセージモデルの他に、イベントを用いた制御モデルを用意している。

イベントモデルでは、図3.7のように、イベントリスナオブジェクトをルートコンポーネント (木構造の土台コンポーネント) に登録しておく。イベントリスナオブジェクトは図3.8に示すメンバを含んでおり、`event_type` に一致したイベントが発生したときに、`receiver` コンポーネントへ直接メッセージを送信する。

イベント伝達の流れは以下のようになっている。

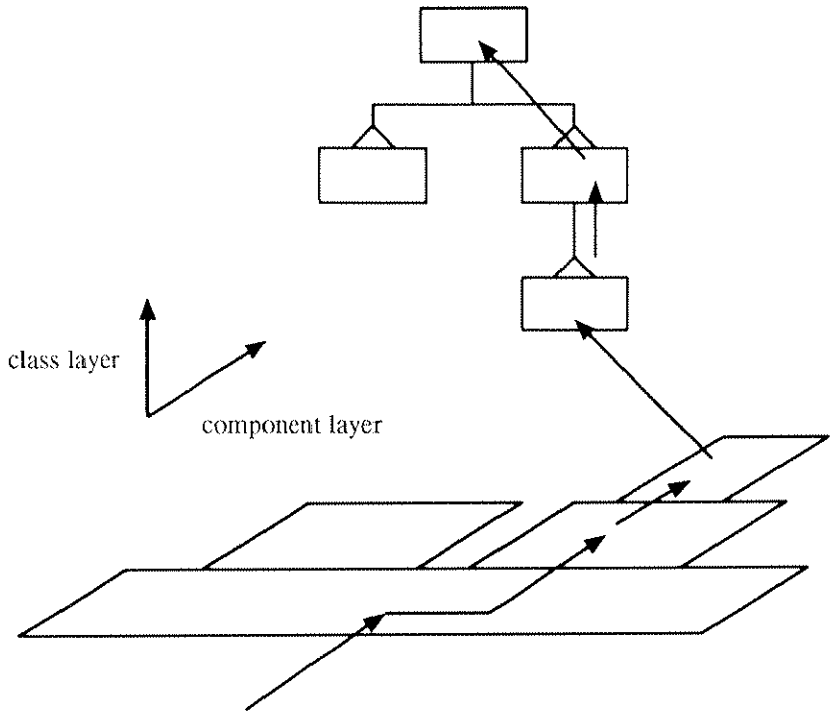


図 3.6: メッセージ探索

1. イベントを受け取るコンポーネントは、コンポーネントの生成時 (コンストラクタ実行中) にルートコンポーネントに対して、イベントリスナを登録する。

2. イベントを発生するコンポーネントは、ルートコンポーネントに対してイベントの発生を通知する。

3. ルートコンポーネント中のイベントリスナは、発生したイベントの種類を調べて、待ち受けていたイベントと一致した場合、コンポーネントにメッセージを送る。

4. コンポーネントの破棄時には、Nuts コンポーネントの基底クラス (NutsCore クラス) のデストラクタが、ルートコンポーネント中のイベントリスナを検査し、自身を登録したイベントリスナがあれば取り除く。これにより、既に破棄されたコンポーネントにイベントが送られる心配はない。

メッセージもイベントも最終的にはコンポーネント内の同じルーチンで処理される。つまり、メッセージとイベントはコンポーネントにたどり着くまでの道筋が異なるだけで、コンポーネントで受理された後は、両者は区別されない (以後はメッセージとイベントを区別せず単にメッセージと記す)。これにより、コンポーネント間の相互作用は、メッセージ／イベントいずれの場合も、コンポーネント内のメッセージ処理ルーチン (`recognizeMessage()` メソッド) により把握できる。

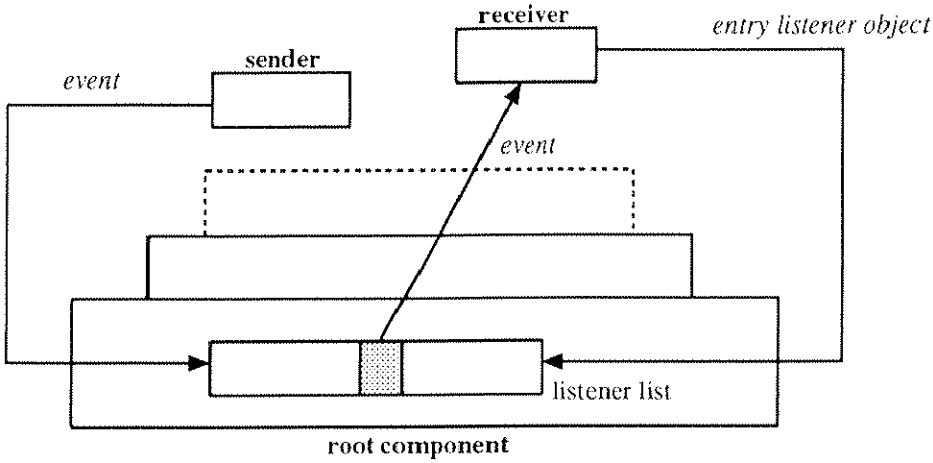


図 3.7: イベントモデル

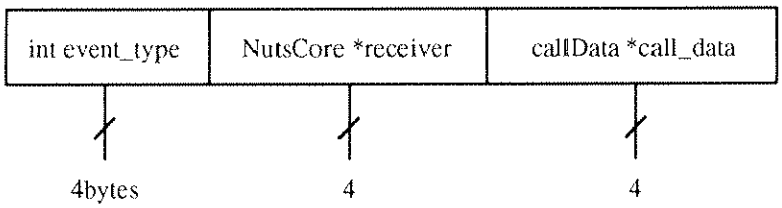


図 3.8: リスナオブジェクトのメンバ

3.2.3 メッセージオブジェクト

メッセージは、図 3.9に示すメッセージオブジェクトを用いて伝送される。メッセージオブジェクトには、宛先、メッセージの種類、引数, オーナー (メッセージの送り手への参照)、返信の宛先 (後述するメッセージの横取りの際にはオーナーとは異なる) が含まれる。通常、メッセージオブジェクトは送り手から受け手へ一方通行で送られるが、受け手では送り手への参照をメッセージのオーナーという形で入手可能であり、これを用いて、受け手から送り手のメソッドを呼ぶことも可能である (図 3.10の`a->get()`)。

また、メッセージを受理したコンポーネントは自身への参照を返す。この返り値を用いれば、メッセージ送信完了後も送り手側から受け手側にメッセージを直接送ることができる (図 3.10の`b->get()`)。

`queryIdentity()` メソッドは、この用途に特化したメソッドで、メッセージ送信と同じ方法でコンポーネントを探索し、見つかったコンポーネントへの参照を返す。`queryIdentity()` を用いて取得したコンポーネントの参照は、以下のような場合に有効である。

- コンポーネントの存在の有無だけが知りたい場合。

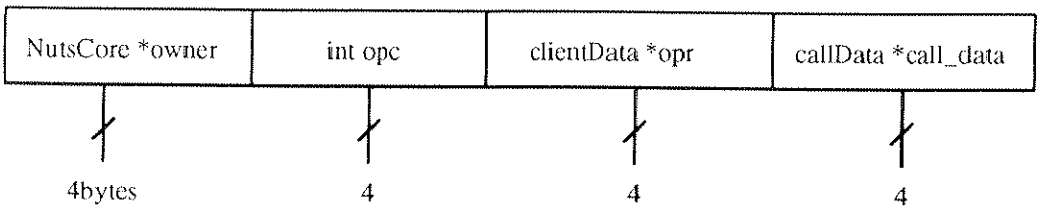


図 3.9: メッセージオブジェクトのメンバ

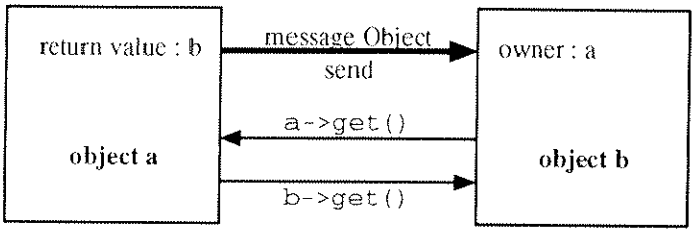


図 3.10: メッセージ送受信者間でのやりとり

- 引数の数が多く (メッセージには一つの引数しか含められない)、それらの引数が渡せるメソッドを直接呼びたい場合。
- 複数のメッセージ送信が連続して必要なことがわかっており、その都度探索するのではなく、最初に 1 回だけ探索して、その後連続してメッセージを直送した方が有効な場合。

ただしこの方法は、最初の 1 回以外はコンポーネントの探索を行なわないため、コンポーネントの位置が移動した場合や、コンポーネントが破棄された場合に、古い参照を使ったメソッド呼び出しを行うと致命的なエラーとなる。

3.2.4 メッセージの横取り

Nuts のコンポーネントは、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。メッセージの横取りには、

1. あるコンポーネント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コンポーネントに成りすます。
2. 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。
3. 受信の可否を外部から制御する (横取り先を Null にするとメッセージは受信されない)。

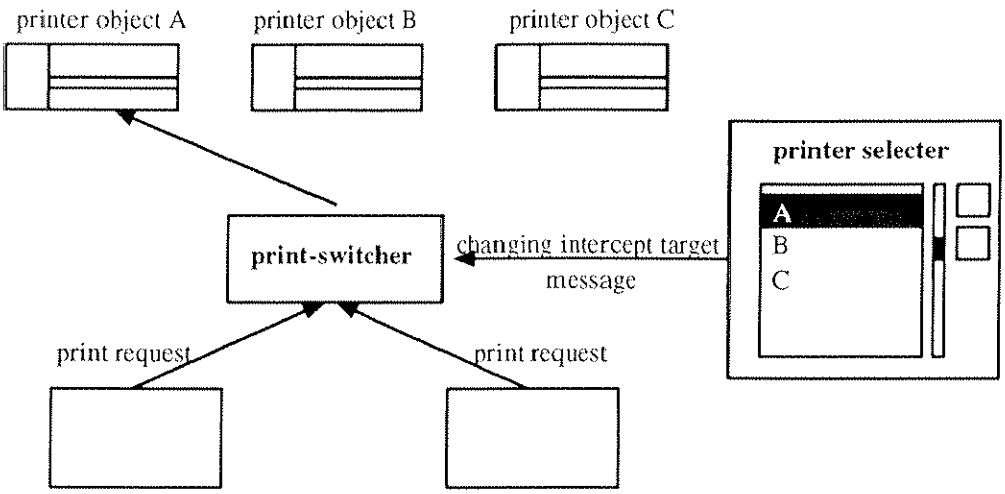


図 3.11: メッセージの横取りによるプリンタの切替え

などの使い方があり、横取りによってメッセージの流れを動的に変化させられる。

1. の使い方は、後述するベクターコンポーネントで必要となる。2. の使い方は、プリンタの切替えのような場面で有効である。あるプログラムが複数のプリンタを切替えて使いたい場合に、プリント要求メッセージの送り先をその都度、使用するプリンタに変更するのではなく、図 3.11のようにプリント要求メッセージの送り先は常に print-switcher に固定とする。プリンタの切替え時には print-switcher にメッセージ横取り先の変更メッセージを送り、使用するプリンタのコンポーネントにメッセージを横取りするように変更する。こうすることで、プリンタを利用する側のコンポーネントは常に、固定されたコンポーネントにメッセージを送れば良く、プリンタ毎にメッセージ送信先を変更する必要が無くなる。

メッセージ横取り機構は、具体的に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 (横取り中止メッセージは横取りされない)。
- 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。

横取りには、以下の 2 つの方法がある。

名前による横取り 横取りされるコンポーネントに、横取りするコンポーネントの名前を設定する。名前による横取りが設定されていると、横取り名によって再度受信者探索を行う (図 3.12a)。イベントモデルでは対応していない。

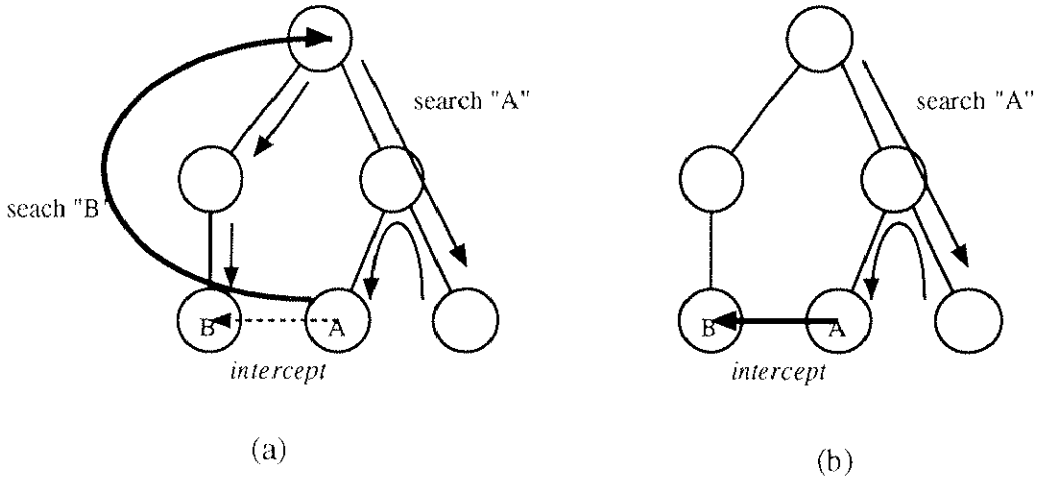


図 3.12: メッセージ横取りの方法

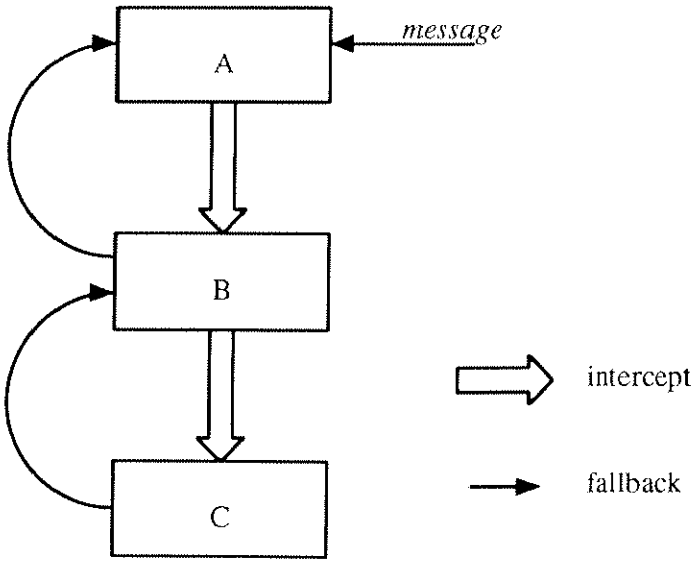


図 3.13: メッセージの横取り

参照による横取り 横取りされるコンポーネントに、横取りするコンポーネントの参照を設定する。参照による横取りでは、設定された横取りコンポーネントへの参照を基に、メッセージを転送する (図 3.12(b))。

横取り者からさらにメッセージを横取りすることもできる。多重の横取りでは、メッセージが認識できない場合、もっとも最後に横取りしたコンポーネントから逆順に、メッセージが解釈されるまで横取り順序をさかのぼる。図3.13の例では、A で受け取られたメッセージは最初 C のコンポーネントで受理される。このときのメッセージのオーナーは B である。C で解釈されない場合、メッセージ受理の機会は B,A と横取り順序をさかのぼる。

メッセージ横取り機構をこのような仕様にしたのは、すべてのメッセージを一旦横取りコンポーネントに横取りさせた後、横取り先で認識できない場合 (横取りしたいメッセージではなかった場合) は、通常通り本来の受信コンポーネントでも解釈できる機会を保証するためである (多重に横取りが行われていた場合も含む)。

3.3 考察

ホワイトボックス的なコンポーネントを実現する Nuts のコンポーネントアーキテクチャを構造モデルと制御モデルに分けて説明した。

Nuts では、プログラム構築時に、すべてのコンポーネントを統一的に扱い、単一の方法で生成、結合が可能となっている。構造モデルには、木構造を採用することにより、部品の入れ子状態が簡単に構築できると同時に、構造記述ファイルによるプログラム全体構造の理解が容易になっている。

また制御モデルには、メッセージ受信者の動的な探索を用いることで制御と構造を独立させており、構造の柔軟な変更、未完成状態でのテストなどが可能になっている。

制御のメッセージモデルは、探索のオーバーヘッドが問題となる場合は利用できない。これに対して、Nuts ではイベントモデルによる探索のオーバーヘッドを回避する方法を用意している。現在の Nuts の仕様では、メッセージモデルとイベントモデルの使い分けによって、実行速度上の問題は発生していないが、メッセージモデルで探索オーバーヘッドを回避する手段としては、キャッシュを用いる方法がある。この方法では、最近通信した相手への参照をキャッシュに保存し、探索を開始する前にまずキャッシュに探索相手がいないか検査する。キャッシュでのヒット率が高ければ、頻繁に交信が行われる部品間で一時的に参照関係が構築されたの同じ効果を得ることができる。

本章では、Nuts アーキテクチャをホワイトボックス的なコンポーネントの側面から説明した。第 6 章では、これに加えてコンポーネント差分プログラミングを可能にするための Nuts アーキテクチャの拡張について説明する。次章では、Nuts のアーキテクチャを定義する Nuts のクラス階層について述べる。