Chapter 5

透明なコンポーネント

プログラマは、Nutsの構造モデルに従うことで、柔軟なプログラム構造を実現できるが、その際、適切な親子関係の構築などNutsの構造モデルの規約は最低限守らなければならない。これに対して、親子関係の条件などに無関係にどこでも自由に挿入できるコンポーネントがあれば、さらに柔軟なコンポーネント結合によるプログラミングが可能になる。

本章では、柔軟なコンポーネント結合を実現するためにNutsの構造／制御モデルを拡張し、(NutsCoreのサブクラスという意味で) 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「透明コンポーネント」を導入する。透明コンポーネントは、どこにでも挿入可能であるという性質から部分木のローカルルートや、次章で説明するコンポーネント差分プログラミングで必要なベクターコンポーネントのベースとなる機能を提供する。

5.1 透明コンポーネントと親子関係の例外

コンポーネントを結合して親子関係を構築するための条件 (例:Windowの親はWindowでなければならない等) は、構築時に自動的に検査され、条件を満たさない場合は結合できない。例えば、NutsのX-Window部品の基底クラスはNutsXComponentである。NutsXComponetのサブクラスで、X-WindowのSimpleWindow部品であるNutsXSimpleWindowクラスのコンストラクタでは、次のように実現している。

```
NutsXSimpleWindowe::NutsXSimpleWindow(NutsCore *parent,char *name) :
  : super(parent,name) //super はスーパークラスを示すマクロ
{
     assert(_parent->isSubClass("NutsXComponent"));
     //親への参照は引数の parent ではなく、NutsCore クラスのコンストラクタが
     //セットした_parent を用いる
}
```

これにより、親がNutsXComponentであるという条件を満たさない場合はassertに失敗 (強制終了) するようになっている。

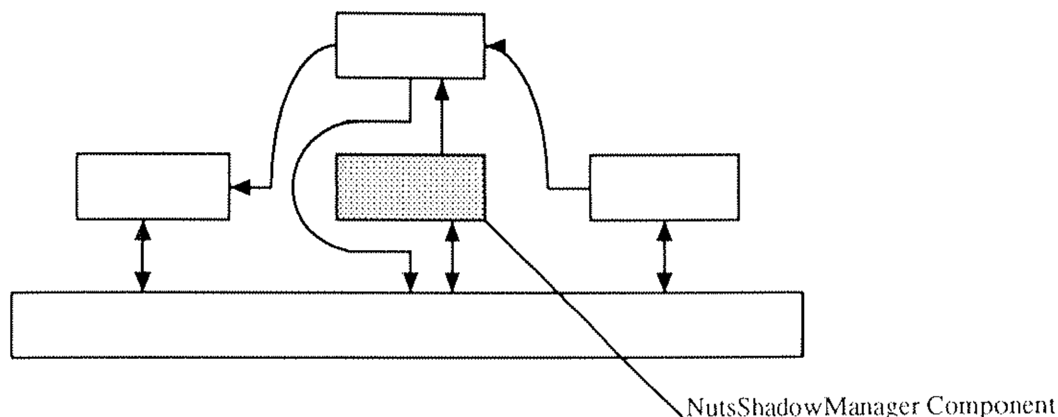

図 5.1: 透明コンポーネントの親子関係

これに対して、透明コンポーネントとは、部品木のどこにでも (Window 部品と Window 部品の間にも) 挿入できるコンポーネントである。

このような、透明コンポーネントの性質を実現するため、透明コンポーネントでは Nut の構造モデルに対して、以下のような拡張を行っている。

- 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、その子コンポーネントからは存在しないように見える。
- 子から親の条件検査は、透明コンポーネントの親コンポーネントに対して行われるため、透明コンポーネントはこの検査を回避できる。
- 透明コンポーネントは、兄弟コンポーネントからも存在しないように見える。

これらの性質を実現するため、透明コンポーネントが存在した場合、親子関係の参照を図 5.1のように構築する。

5.2 メッセージの横取りと透明コンポーネント

メッセージの横取りにおいても、透明コンポーネントは例外的に振舞う。

メッセージの横取りは通常は以下のように行われる。

- メッセージのオーナーは横取りされたコンポーネントに変更される。
- 横取り先でそのメッセージが認識できなければ、改めて本来の受信者がメッセージを受け取る。

これに対して、透明コンポーネントでは、次のように仕様を変更する。

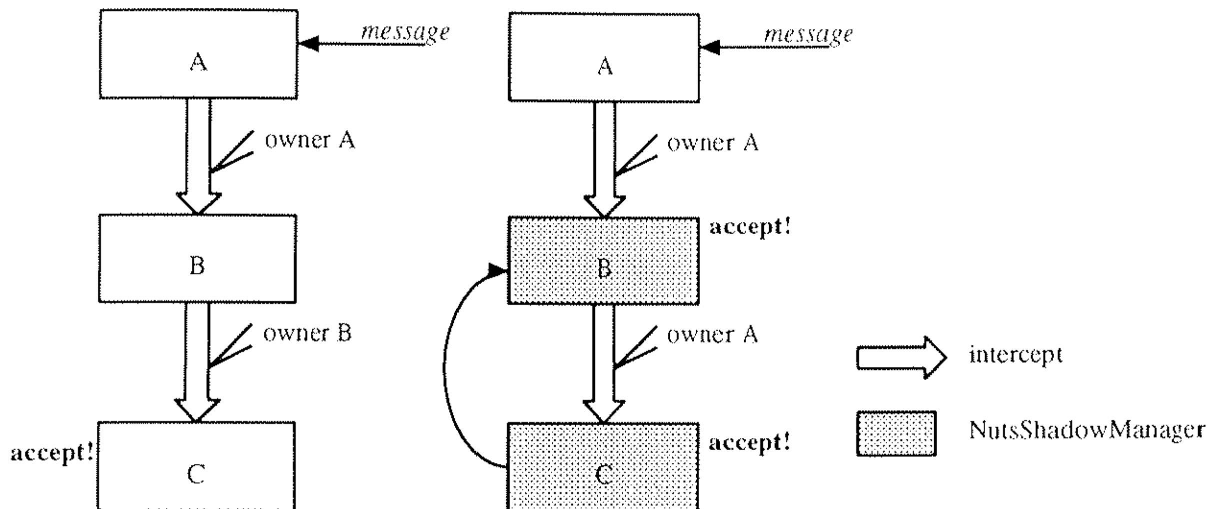

図 5.2: 透明コンポーネントからの横取り

- 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。

図 5.2(a) のように、すべてのコンポーネントが通常のコンポーネントである場合は、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消費される。一方、図 5.2(b) のように B,C が透明コンポーネントの場合は、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。これにより、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対等に扱われる。

5.3 NutsShadowManager クラス

NutsShadowManager クラスは透明コンポーネントを実装している。NutsShadowManager は NutsManager のサブクラスであり、その差分として透明コンポーネントに必要な構造／制御の例外を実現している。

5.3.1 構造モデルの例外

第 4.1.3節で示したように、NutsCore クラスのコンストラクタでは、

```
NutsCore::NutsCore(NutsCore *parent,char *name)
{
    .....
    _parent = parent->identity();
}
NutsCore *NutsCore::identity(){return this;}
```

となっている。つまり NutsCore クラスでは、親コンポーネントに対して、identity() メソッドによってコンポーネントへの参照を確認し、(identity() の返り値を) メンバ変数 _parent にセットしている。また、通常のコンポーネントはidentity() 確認に対して、自身の参照を返すだけである (第 4.1.3節参照)。これに対して、NutsShadowManager クラスではidentity() をオーバーライドし、以下のようにしている。

```
NutsCore *NutsShadowCore::identity(){return _parent;}
```

つまり、透明コンポーネントはidentity() の確認に対して、その親コンポーネントへの参照を返すので、子コンポーネントは、透明コンポーネントの親への参照を親として保持することになる。この場合、子から親の条件検査は、透明コンポーネントの親コンポーネントに対して行われる (すなわち_parent が親の親を示している)。これにより、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、その子コンポーネントからは存在しないように見える。

兄弟コンポーネントから透明コンポーネントを隠す仕組みも同様の手法による。透明コンポーネントも NutsCore クラスを継承しているので、透明コンポーネントが作られた時点では、_sibling は兄弟を指し、親コンポーネントの_youngest が透明コンポーネントを指すように設定される。その上で、NutsShadowManager クラスではsiblingIdentity() をオーバーライドし、以下のようにしている。

```
NutsShadowCore *NutsCore::siblingIdentity(){return _sibling;}
```

これにより透明コンポーネントは、それ自体が兄弟コンポーネントとして設定されないため、兄弟コンポーネントからも隠れた存在となる。透明コンポーネントを含めた、コンポーネントの参照関係は図 5.3のようになる。

5.3.2 制御モデルの例外

透明コンポーネントでは、receiveMessage(),receiveEvent() をオーバーライドし、透明コンポーネントへのメッセージを横取りした場合に例外的な動作を行うようにしている。

receiveMessage() を例にとれば、NutsCore クラスでは、

```
NutsCore *NutsCore::receiveMessage(NutsCore *onwer,...)
                        //owner: メッセージのオーナーへの参照
{
```

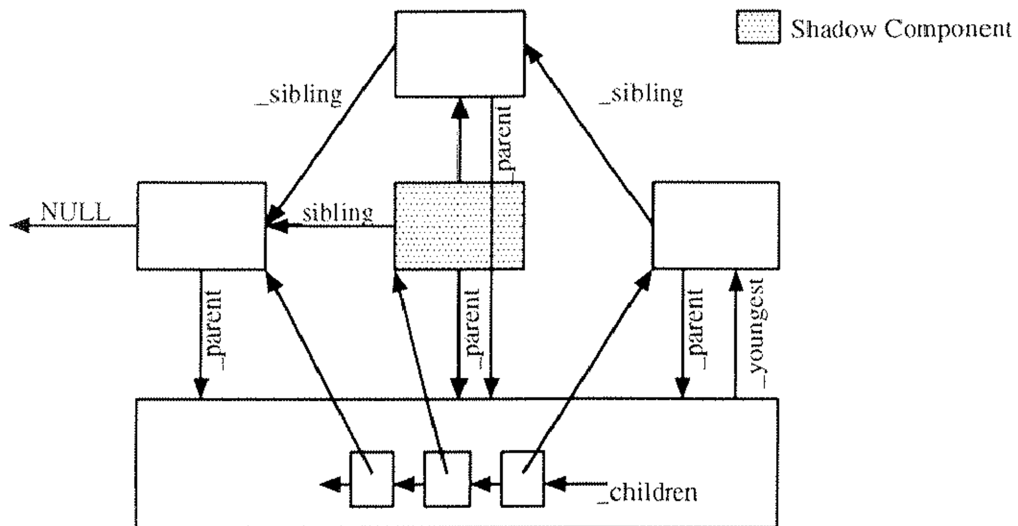

図 5.3: 透明コンポーネントの参照関係

```
    .....
    NutsCore *rtn;
    if(_redirectObject){
       if(!(rtn = _redirectObj->receiveMessage(this,...))
           rtn = this->receiveMessage(this,...));
    }
    return rtn;
}
```

としている。これは、横取りがある場合にオーナーを自分自身に変更し (receiveMessage() の第一引数がオーナーへの参照)、横取り先で解釈できなかった場合 (receiveMessage() がNULL を返す場合) に限り、自身での解釈を試みるようになっている。これに対して、NutsShadowManager クラスでは、

```
NutsCore *NutsShadowManager::receiveMessage(NutsCore *onwer,...)
{
    .....
    if(_redirectObject){
          _redirectObj->receiveMessage(owner,...));
          return  this->receiveMessage(owner,...));
    }
}
```

としている。これは、透明コンポーネントが、オーナーを変更せず (receiveMessage() の第一引数がowner のまま)、また横取り先での解釈の可否に関わらず (receiveMessage()

の返り値によらずに)、常にメッセージを解釈することを示している。これにより、透明コンポーネントへのメッセージを横取りした場合、メッセージのオーナーは変更されず、また横取りが設定されていても、横取りされた後で、常にメッセージの解釈を試みる (制御の例外を利用するケースについては第 6.3.2節を参照)。

5.4 NutsGroupManager クラス

NutsGroupManager クラスは、NutsShadowManager のサブクラスで、その差分として部分木をグループ化し、部分木のローカルなルートコンポーネントになるための性質を追加している。部分木のルートは、部分木内に含まれるコンポーネントのメッセージターミナルとして機能する。言い替えれば部分木の外から見たとき、部分木が一つのサブシステムとして (一つのコンポーネントと同じように) 扱える。NutsGroupManager の機能を以下で説明する。

5.4.1 NutsGroupManager クラスの初期化フェーズ

すべての Nuts コンポーネントは、メンバ変数`_leader` (NutsCore クラスで定義されている) に、自分が属するグループマネジャへの参照を保持している。初期状態では、`_leader==_root` であり、すべてのコンポーネントがルートコンポーネントのグループに属している。

NutsGroupManager クラスのインスタンス (以下グループマネジャ) は、透明部品であり、部品木中のどこにでも挿入できる。このときグループマネジャは、自分を根とする部分木に含まれるコンポーネントに対して、`_leader` メンバに`this` をセットすることでグループを作る。同じ`_leader` を保持した部品群を一つのグループと見なすことができる。

具体的には、NutsGroupManager クラスの初期化フェーズは、自身の`setGroup()` メソッドを呼んでいる。

```
NutsGroupManager::initialize()
{
    super::initialize();
    super::setGroup(this);
}
```

`setGroup()` メソッドは、NutsCore クラス、および NutsManager クラスで以下のように実現している。

```
void NutsCore::setGroup(NutsCore *leader)
{
   _root = leader;
}
void NutsManager::setGroup(NutsCore *leader)
```

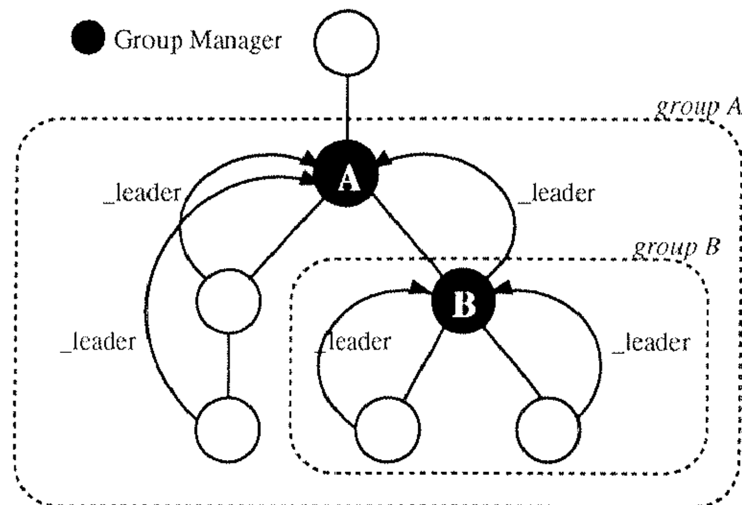

図 5.4: グループ構成

```
{
    super::setGroup(leader);
    listHandler &h = _child.openList();
    while(h)_child.getNextObj(h)->setGroup(leader);
    _child.closeList(h);
}
```

これにより、グループマネジャを根とする部品木の`_leader` はすべてグループマネジャの参照に設定され、グループ化されたことになる。ただし、NutsGroupManager クラスでは、`setGroup()` を以下のようにオーバーライドしている。

```
NutsGroupManager::setGroup(NutsCore *leader)
{
    NutsCore::setGroup(leader);
}
```

`setGroup()` によるグループ化は、すでにグループ化されている部分木の内部にまでは伝搬しない。これにより図 5.4のようにグループの入れ子状態ができる。

5.4.2 NutsGroupManager クラスを用いたメッセージ通信

グループマネジャを木構造中に挿入することにより、グループマネジャをルートとする部分木はグループ化されたことになる (厳密にはルートコンポーネントをグループマネジャとするグループから外れて、NutsGroupManager クラスのコンポーネントをグループマネジャとするグループに属したことになる)。木構造の全体検索を行うメッセージ送信は、

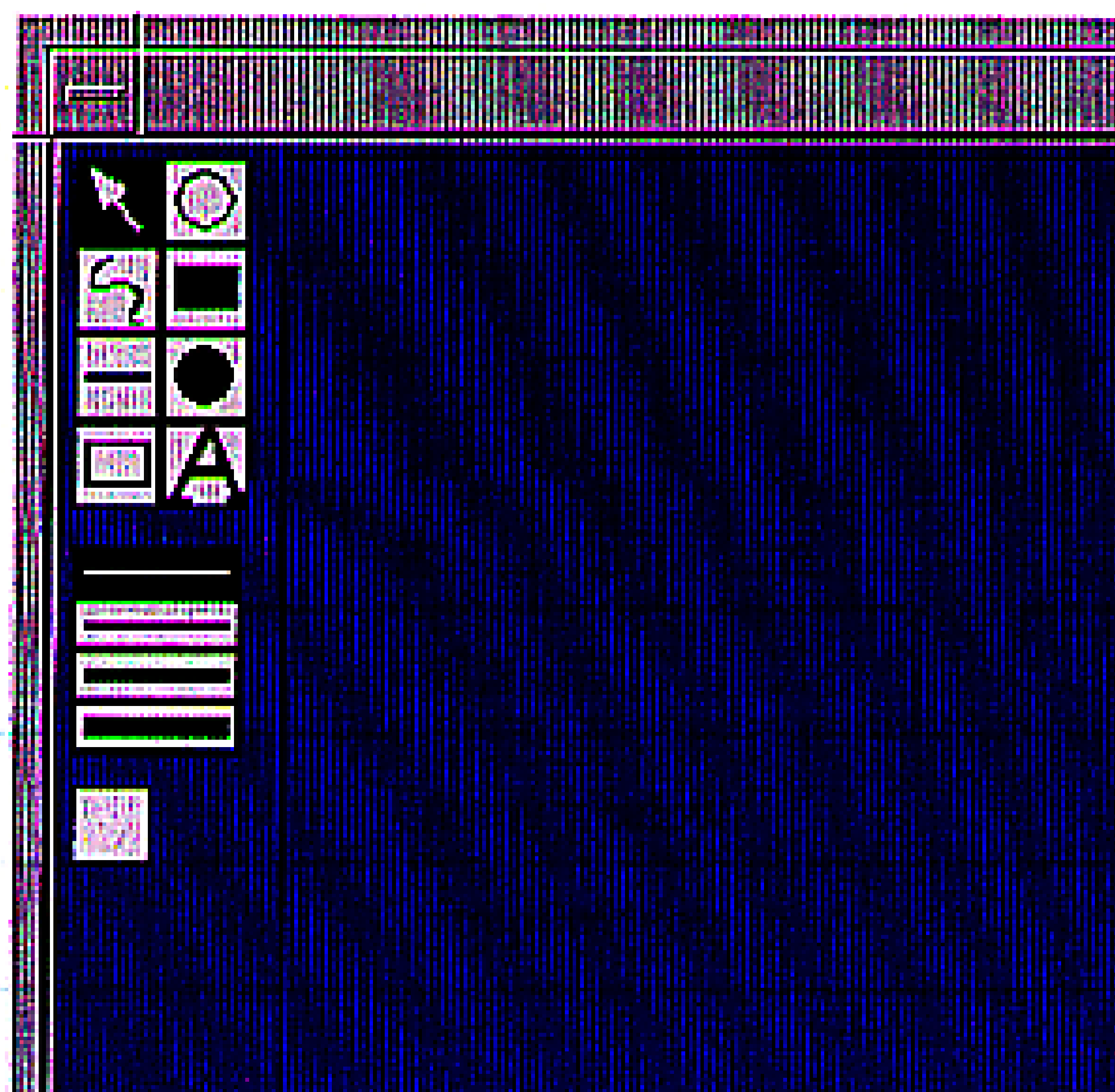

図 5.5: キャンバスウインドウ

_leader->sendMessage() であった。これは、グループ化が行われる前は_leader == _root であるため、ルートコンポーネントのグループ、すなわち木構造全体への探索を意味した。グループ化が行われた後は、_leader は、ローカルなグループマネジャを示しているため、受信者の探索対象が同一グループに属する部分木に限られる。グループ化によって、部分木内で閉じたメッセージ通信を行うことにより、Nuts の部品群は、部分木単位でプログラム中のどこにでも再配置可能となる。これにより、部分木を一つの部品として扱える。

一例として、マルチドキュメントインタフェースのウインドウを考える。これは、図 5.5 のような描画のためのツールパレット (ペンやブラシなど) と絵を描くための描画領域を持ったウインドウである。このウインドウのコンポーネント構成は図 5.6のようになっている。実際のプログラムでは、このキャンバスウインドウが複数同時に開けるマルチドキュメントのインタフェースとしたい。このとき、図 5.6の木を全体として一つのキャンバス部品として扱えると便利である。

従来のオブジェクト指向プログラミングでは、以下の方法が考えられる。

1. ベースとなるフレームウインドウ自体をサブクラス化し、ツールボックスとキャンバスの間でのメッセージ受渡しを行うようにする。

2. 別にメッセージターミナルの役目を担うクラスを作り、フレームウインドウをそれとの多重継承によるウインドウに置き換える。

1. の方法では、例えばウインドウをスクロールウインドウに置き換える際に、グループ化機能を持ったスクロールウインドウのためのサブクラス化を行わなければならない。一般には、ウインドウの種類と拡張機能の組み合わせの数だけサブクラス化を行う必要があり、クラス数の爆発の原因となる。2. の方法では、プログラムの理解容易性の観点から好ましくない。

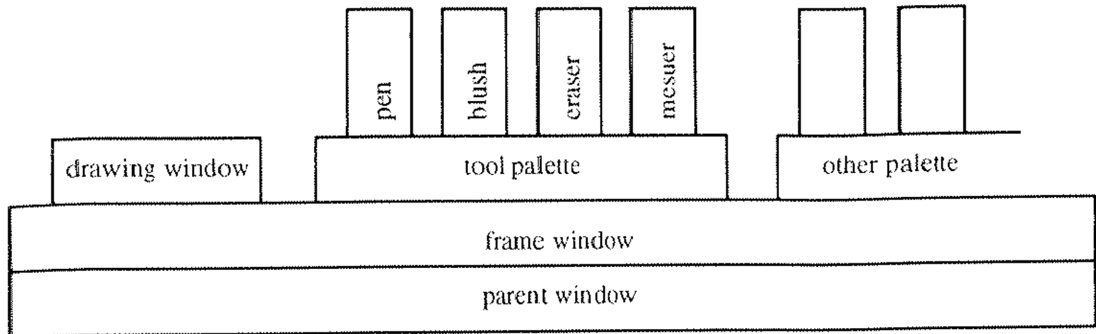

図 5.6: キャンバスウインドウの構成

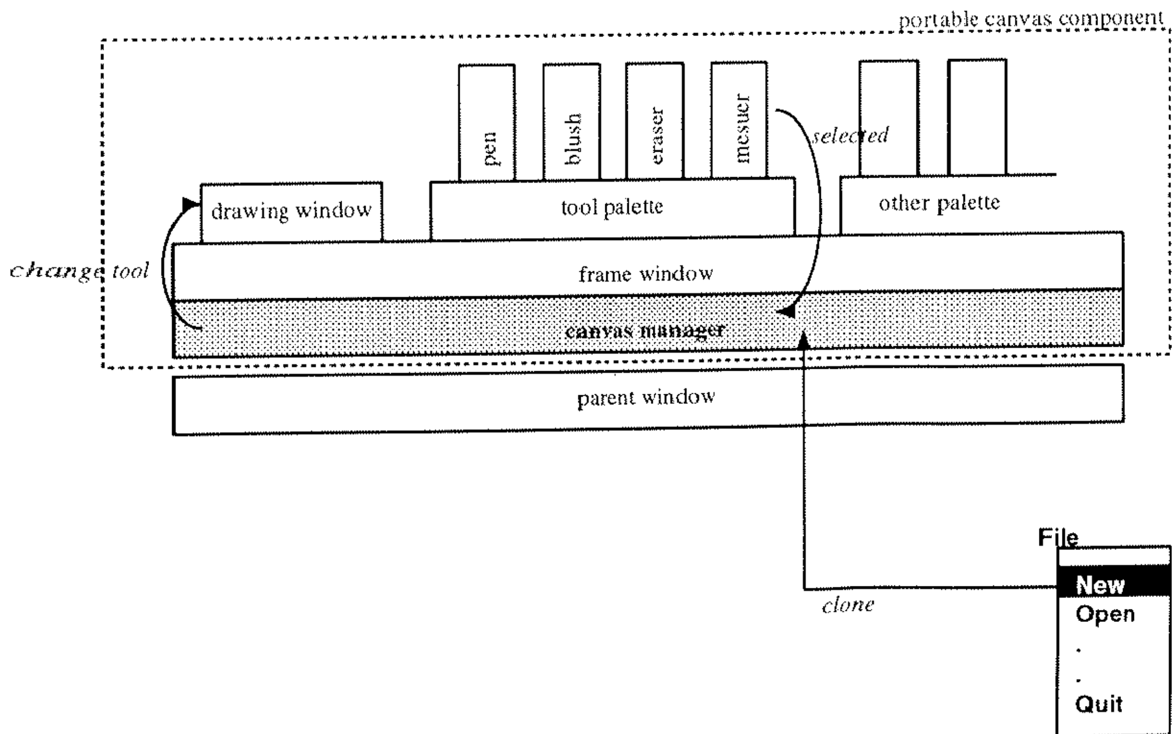

図 5.7: 部品化したキャンバスウインドウ

これに対して、グループマネジャコンポーネントを用いる方法では、図 5.7のようにフレームウインドウの親コンポーネントとして、グループマネジャから派生したキャンバス部品マネジャを挿入するだけですむ。この方法では、グループ化のためにサブクラス拡張を用いておらず、ウインドウとグループ化機能が部品単位で分離されたままになっている。このため、それぞれを再利用できると同時に、多重継承の問題も発生しない。

グループマネジャは、透明なのでウインドウとウインドウの間にも挿入できる。また、グループマネジャ上にどのように部品木が構成されていたとしても (ウインドウがスクロールウインドウに置き換えられたとしても)、ツールの選択に関するメッセージを受け取って、描画領域に反映させることが可能である。さらに、マルチドキュメントのアプリケーションで、新しいキャンバスを作成したいときは、キャンバスのグループマネジャに対して、`clone()` メッセージを送ることでキャンバスウインドウ全体を複製できる。

このようにグループマネジャは、それを挿入することで部分木を可搬にし、部分木単位でひとつの部品として振舞うようにできる。

5.5 考察

本章では、Nuts の構造／制御モデルを拡張して、透明なコンポーネントを実現する方法を説明した。

Nuts のコンポーネントを用いたプログラミングでは、木構造をベースとした構造モデルを用いることで、容易にコンポーネント同士の結合が可能である。しかし、木構造自体が、柔軟なコンポーネント結合を阻害する要因になることも考えられる。特に、親子関係構築のための条件は厳守する必要があるため、このことがコンポーネント間のメッセージの受け渡しや修飾などコンポーネントの機能を影で補助するようなコンポーネントの挿入を妨げていた。

そこで、Nuts の構造モデルを拡張し、通常のコンポーネントの影で機能する透明なコンポーネントの仕様を追加した。これにより、Nuts コンポーネント同士がより柔軟に結合できるようになった。

また、制御モデルにおけるメッセージ横取り機能を透明コンポーネントに拡張した。透明コンポーネントへのメッセージの横取り仕様では、透明コンポーネントを複数段組み合わせた場合、透明コンポーネント間でメッセージの横取りがあっても、各段の透明コンポーネントがメッセージを解釈する機会を保証した。これにより、一つの通常コンポーネントに対して、透明コンポーネントの機能 (通常コンポーネントの修飾等) を多重に付加することが可能になった。

本章で説明した透明コンポーネントは、次章で説明するコンポーネント差分プログラミングのベースとなるものである。