Chapter 4

Nutsのクラス階層

Nutsのコンポーネントは構造／制御モデルに従ったアーキテクチャを持つ。Nutsでは、このモデルの機能をNutsの基底クラス群で実現している。すべてのコンポーネントクラスは基底クラス群からの派生クラスとなるため、必然的に構造／制御モデルを継承する。またすべてのコンポーネントは、基底クラスのオブジェクトとして統一的に扱える。

Nutsの基底クラス群はNutsCore、NutsManager、NutsApp(図4.1)各クラスからなる。本章では、これらのクラスの実装について詳説する。

4.1 NutsCoreクラス

NutsCoreクラスはNutsの基底クラスであり、別のコンポーネントの子コンポーネントになるための性質を実現する。Nutsのすべてのコンポーネントは、NutsCoreクラスのサブクラスであり、木構造のリーフ部に接続するための最低限の機能を備えている。また、すべてのコンポーネントはNutsCore型として統一的に扱える。

4.1.1 Nutsコンポーネントの規約

NutsCoreクラスは、表4.1に示すNutsコンポーネントの規約をNutsコンポーネントの共通部分として実現している。新規コンポーネントクラスを作成する場合は、これらのコンポーネント規約が保たれるように、各メソッドをサブクラスで独自定義(オーバーライド)しなければならない。しかし、これらの作業をクラス派生毎に手動で行うのは煩雑であり、定義の抜けなどで、予期せぬコンポーネントの動作を起こしかねない。これらの作業は新規クラス名とそのスーパークラス名をキーに機械的に作成可能であり、新規クラスの作成をツールに委ねることができる。

nutsCreateNewClassツールを用いれば、規約に沿ったクラスの定義が自動的に生成される。このツールは新規クラス名とスーパークラス名をキーにして新規クラスのヘッダファイル、ソースファイルを生成する。NutsCoreクラスから派生したNutsTestClassを新規に生成するために、

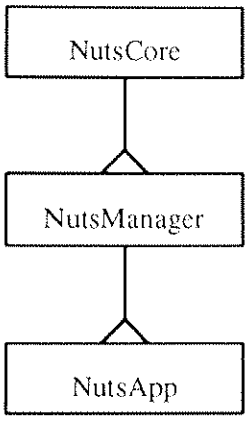

図 4.1: Nuts の基底クラス群

"nutsCreateNewClass NutsTestClass NutsCore"

を実行した結果、生成されたヘッダファイル、ソースファイルのリストを図 4.2、図 4.3、に添付する。

4.1.2 NutsCore クラスのメンバ変数

NutsCore クラスは、メンバ変数として表 4.2に示す 6 つの変数を持つ。

この中で、_root メンバ変数は、すべてのオブジェクトに共通の変数であり (static 宣言されている)、常にアプリケーションの土台部品を参照している。これと似たものに_leader メンバ変数がある。_leader の値は個々の部品毎に異なり、プログラムの初期段階では、_leader と_root の値は等しいが、後述するグループ化を行うと_leader は、自身が属するグループのローカルな土台部品を参照するように変更される (第 5.4節参照)。メッセージ探索の際に、部品木の全体探索を行うつもりで、探索開始点を_root としたものは、終始全体探索になるが、_leader を探索開始点とした場合は、グループ化によってグループ内のローカルな部分木の探索に置き換わる。

4.1.3 NutsCore クラスの初期化フェーズ

NutsCore クラスの初期化フェーズを図 4.4に示す。

1. コンストラクタで親子／兄弟の参照関係を構築するため、以下のように記述している。

```
NutsCore::NutsCore(NutsCore *parent,char *name)
{
    //.....
    // 親子関係の構築
```

```
#ifndef NutsTextClass_INC
#define NutsTextClass_INC

#include<NutsCore.h>

#undef  super
#define super NutsCore

#define CLASS_NAME "NutsTextClass"

class NutsTextClass : public super{

private:
protected:
public:

    NutsTextClass(NutsCore *obj,char *name) : super(obj,name){ }

    //virtual ~NutsTextClass(){ }
    //virtual void initialize();
    //virtual void realize();
    //virtual NutsCore *recognize(int,void *,NutsCore *,void *);

    virtual NutsCore *createObj(NutsCore *obj,char *name)
        {return NutsTextClass::CreateObj(obj,name);}
    static NutsCore *CreateObj(NutsCore *obj,char *name)
        {return new NutsTextClass(obj,name);}
    virtual const char *const className(){return NutsTextClass::ClassName();}
    static const char *const ClassName(){return CLASS_NAME;}
    virtual Bool isSubClass(char *class_name){
        if(strcmp(class_name,NutsTextClass::className()) == 0) return True;
        else return super::isSubClass(class_name);
    }
};

#define newNutsTextClass(app,name) NutsTextClass *name = \
new NutsTextClass(app,#name)

#endif
```

図 4.2: nutsCreateNewClass によって自動生成されたヘッダファイル

```
#include <NutsTextClass.h>
/*
void NutsTextClass::initialize()
{
    super::initialize();
}
*/
/*
void NutsTextClass::realize()
{
    super::realize();
}
*/
/*
NutsCore *NutsTextClass::recognize(int       opc
                                   ,void     *opr
                                   ,NutsCore *source
                                   ,void     *call_data)
{
    NutsCore *rtn;

    switch(opc){
    case :
        rtn = this;
        break;
    default:
       rtn = super::recognize(opc,opr,source,call_data);
    }
    return rtn;
}
*/
```

図 4.3: nutsCreateNewClass によって自動生成されたソースファイル

表 4.1: Nuts コンポーネントの規約

規約内容	NutsCore クラスでの実装名
クラス名を保持し、それを返すメソッドを持つ。	`#define CLASS_NAME char *className();`
そのクラスがあるクラスのサブクラスかどうか調べるメソッドを持つ。	`Bool isSubClass(char *);`
メッセージの解釈をおこなうメソッドを持つ。	`recognizeMessage();`
メッセージがそのクラスで解釈されない場合、スーパークラスで解釈する。	`if(!recognizeMessage())` `super::recognizeMessage();`
newNuts クラス名 (親部品名, 部品名) によってインスタンスを生成できる	`NutsCore *create(NutsCore *,char *);` に変換するマクロ

```
        _parent = parent -> identity();
        _parent -> registerObj(this);

        // 兄弟関係の構築
        NutsCore *youngest = _parent -> youngestIdentity();
        _sibling = youngest -> siblingIdentitiy();
        _parent->registerYoungestObj(this);
    }
    NutsCore *identity(){return this;}
    NutsCore *siblingIdentity(){return this;}
```

引数parent には、親コンポーネントへの参照が入る。parent->identity() によって、コンポーネントの参照を確認し、メンバ変数_parent にセットする。その後、親に対して自身を登録する。

兄弟の場合は、親に対して末っ子 (最後に生成されたコンポーネント) への参照を youngestIdentity() によって取得し、younget->siblingIdentity() によって、コンポーネントへの参照を確認した後、メンバ変数_sibling にセットする。その後、親に対して自身を末っ子として登録する。

identity()、siblingIdentity() は、共にthis を返すだけの参照の確認になっているが、第5章で説明する透明コンポーネントでは異なった動作をするようにオーバーライドする。

2. 親コンポーネントからinit() が呼ばれる。

3. init() は、initialize()、manage() の各初期化メソッドを呼んで自身を初期化す

表 4.2: NutsCore のメンバ変数

変数	内容
`static NutsCore *_root;`	ルートコンポーネントを指す。NutsApp クラスがセットする。
`NutsCore *_parent;`	親コンポーネントへの参照。
`NutsCore *_sibling;`	兄弟コンポーネントへの参照。
`NutsCore *_leader;`	通常は_root と同じ。後述するグループ化では異なる。
`char *_name;`	コンポーネント固有名を保持。
`NutsCore *_redirectObj;`	横取りコンポーネントへの参照。デフォルトはthis
`char *_redirectName;`	横取りコンポーネントの名前。デフォルトはNULL

る。クラス固有の初期化はinitalize()、manage() をオーバーライドすることにより行う (initalize()、manage() の相違については NutsManager クラスの節で説明する)。

コンポーネントの木構造は、次節で説明する NutsManager クラスとの組合せによって決定されるので、最終的なコンポーネントの結合関係については第 4.2.2節で説明する。

4.1.4 NutsCore クラスでのメッセージ処理の流れ

NutsCore でのメッセージ処理の流れを以下に示す (図 4.5)。

1. メッセージの送信者からsendMessage() メソッドが呼ばれる。
2. receiveMessage() メソッドを呼び、自身が探索キーの条件に一致しているかを検査する。
3. 一致している場合は、メッセージ横取りが設定されていないか検査する。
4. メッセージの横取りが設定されている場合は、横取り先へメッセージを転送する。
5. メッセージの横取りが設定されていない場合は、recognizeMessage() メソッドを呼び実際の処理に移る。
6. メッセージを受理したコンポーネントが属するクラスのrecongizeMessage() で解釈できない場合は、スーパークラスのrecognizeMessage() を呼び出す。
7. メッセージが自身で解釈できると、receiveMessage() はthis を、そうでない場合は、NULL を返す。

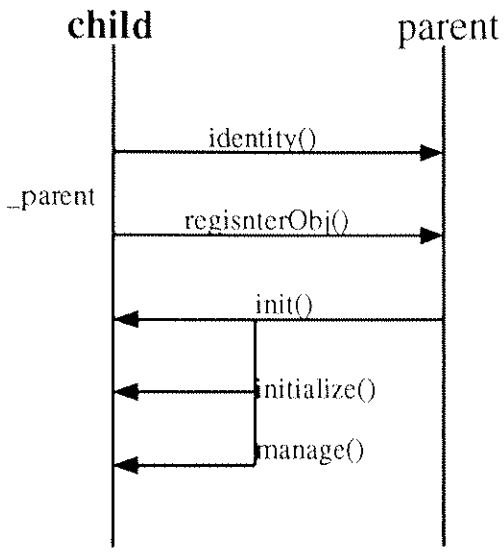

図 4.4: NutsCore クラスの初期化フェーズ

イベントモデルの場合は、sendMessage()、receiveMessage() が、それぞれsendEvent()、receiveEvent() に置き換わるだけで、実際の処理はrecongizeMessage() で扱われる。従ってすべてのメッセージは、recognizeMessage() で処理されることになり、メッセージによるコンポーネントの挙動は、recognizeMessage() の解析により把握できる。

sendMessage()、receiveMessage()、recognizeMessage() は、どれも公開メソッドであり、コンポーネントへのメッセージ入力ポートと見ることができる。sendMessage() のポートから入力する場合が、Nuts のメッセージモデルに従った制御に相当する。それ以外に、すでに別の方法でコンポーネントへの参照が入手されており、受信者探索の不必要な場合には、receiveMessage()、recognizeMessage() の各ポートから直接入力することもできる (図 4.5の点線)。receiveMessage() への入力ではメッセージの横取りが考慮されるが、recognizeMessage() への入力では、メッセージの横取りとは無関係に必ずそのコンポーネントでメッセージが解釈される。

4.1.5 NutsCore クラスで認識可能なメッセージ

NutsCore クラスが認識するメッセージとして、表 4.3に示すものがある。NutsCore クラスはすべてのコンポーネントの基底クラスなので、これらのメッセージは、すべてのコンポーネントで認識可能である。

"What is your name?" など、結果を文字列で表示するコンポーネントは、"STDOUT" という名のコンポーネントに対して結果文字列を引数とした Write メッセージを送る。このメッセージも探索により、STDOUT コンポーネントに伝達されるので、STDOUT コンポーネントが見つかれば表示されるが、見つからない場合は表示されない。STDOUT という名のコンポーネントが、標準出力を扱うコンポーネントの場合には標準出力に結果が

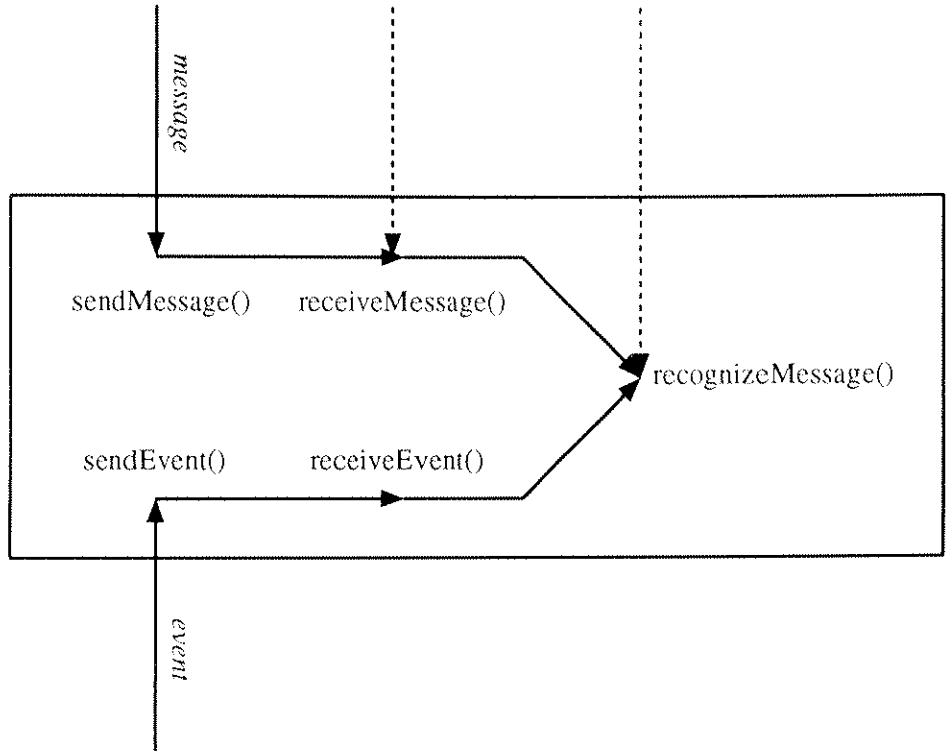

図 4.5: メッセージ授受の流れ

表示される。あるいは別のケースとして、STDOUT という名のコンポーネントが、独自のメッセージウインドウの場合には、そのウインドウに結果が表示される。STDOUT というコンポーネントが見つからない場合はどこにも出力されない。第 3.2.4節で説明したように、STDOUT という名のコンポーネントをマルチプレクサとして定義し、複数の結果出力先を切替えることができる。

4.1.6 メッセージの多重の横取り

前述のように、"pass to..." メッセージによりメッセージの横取りが開始されるが、このメッセージ自体も横取りの対象である。従って、すでに横取りが開始されているコンポーネントに対してメッセージを横取りしようとした場合、すでにメッセージを横取りしているコンポーネントからさらにメッセージを横取りすることになる。

例えば、図 4.6に示すように、a → b → c の順で横取りが設定されているとする。このとき、a に対して a → d の横取りを設定しようとして "pass to d" のメッセージを送ると、それは→ b → c と横取りされ、最終的に c に対して作用する。結果として、a → b → c → d の多段の横取りが実現する。

以上のように、"pass to..." 自体が横取りの対象であるので、メッセージの横取り先を変更したい場合には "pass to..." は使用できない (そのつもりで "pass to..." を使用すると多

表 4.3: NutsCore クラスで認識するメッセージ

What is your name?	コンポーネント固有名を返す。
What is your parent name?	親コンポーネント固有名を返す。
What is your class name?	クラス名を返す。
pass to ...	... コンポーネントへの参照によるメッセージの横取りを開始する。
redirect to ...	... コンポーネントへの名前によるメッセージの横取りを開始する。
cancel redirect	メッセージの横取りを中止する (このメッセージは横取りされない)。
is redirected?	メッセージ横取りが設定されているかどうかを返す (このメッセージは横取りされない)。
add event	イベント登録を開始する。
remove event	イベント登録を削除する。
clone	自身をルートとする部品木を複製して、そのルートを返す。
listup	子のノードの表示。
delete	自身をルートとする部品木を削除する。

重の横取りとなる)。代わりに "cancel redirect" メッセージを送って一度メッセージの横取りを解除した後、再度新たな横取り先を設定する。このように、横取り対象とはならないメッセージとして以下の 2 つがある。

- cancel redirect : メッセージ横取りの中止
- is redirected : 横取りされているかどうかの表示

4.2 NutsManager クラス

NutsManager クラスは、NutsCore クラスのサブクラスであり、その差分として親コンポーネントになる性質を追加している。従って、NutsManager クラスのインスタンスは木構造の中間ノードになることができる。

4.2.1 NutsManager クラスのメンバ変数

NutsManager クラスは、メンバ変数として表 4.4に示す 2 つの変数を持つ。NutsObjList クラスは、NutsCore コンポーネントの参照リストを管理するクラスである。

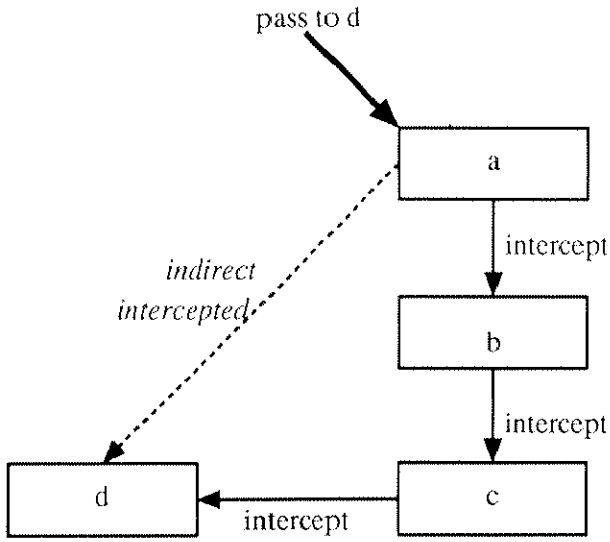

図 4.6: メッセージの多重の横取り

表 4.4: NutsManager のメンバ変数

変数	内容
`NutsObjList _children;`	子コンポーネントのリストを保持する。
`NutsCore *_youngest;`	最後に登録されたコンポーネントへの参照を保持する。

4.2.2 NutsManager クラスの初期化フェーズ

NutsManager の初期化フェーズを以下に示す (図 4.7)。

1. 子コンポーネントからの登録要求によって、親子関係を構築すため、以下のように実現している。

```
void NutsManager::registerObj(NutsCore *obj)
{
    _children.add(obj);
}
void NutsManager::registerYoungestObj(NutsCore *obj)
{
    _youngest = obj;
}
```

`registerObj()` メソッドは NutsCore クラスのコンストラクタから呼ばれ、子コン

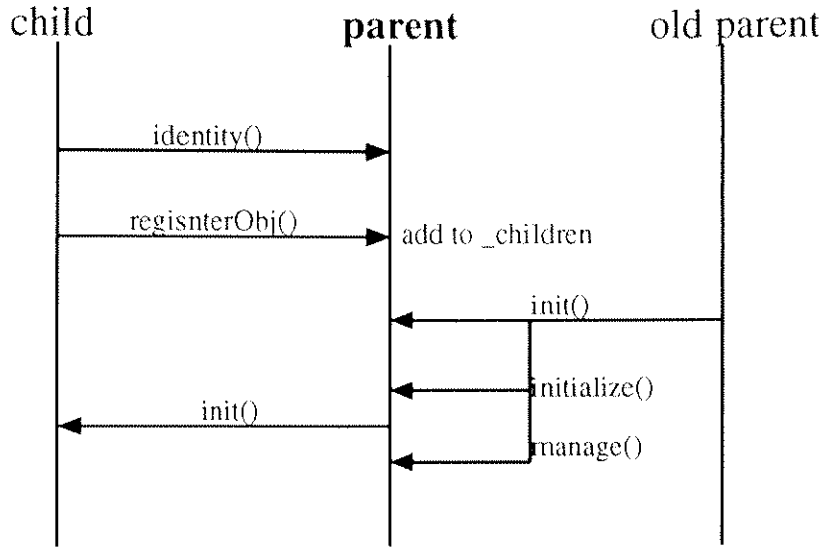

図 4.7: NutsManager クラスの初期化フェーズ

ポーネントへの参照をメンバ変数である_children リストに追加する。同時に、メンバ変数_youngest に最後に作られたコンポーネントへの参照を保持する。この値は、新しい子コンポーネントが追加される度に上書きされる。コンポーネント間の参照関係は、NutsCore クラスのコンストラクタの初期化フェーズによって図 4.8のように構築される。

2. 親コンポーネントからinit() が呼ばれると、初期化ルーチンinitialize()、manage() の間でその子コンポーネントのinit() を呼ぶ。

```
NutsCore *NutsCore::init()
   this->initialize();//子コンポーネント初期化前
   listHandler &h = _child.openList();
   while(h)_child.getNextObj(h)->init();
   _child.closeList(h);
   this->manage();//子コンポーネント初期化後
}
```

すなわち、NutsManager クラスでは、初期化を木構造全体に伝搬させる。この際、initialize() は、その子コンポーネントが初期化される前に、manage() は、すべての子コンポーネント (再帰的にすべての子孫コンポーネント) が初期化された後に呼ばれるという違いがある。

NutsManager クラスのコンポーネントでは、初期化フェーズでinitialize() とmanage() を使い分けることができる。例えば、ウインドウの上にボタンなどの複数の GUI 部品を

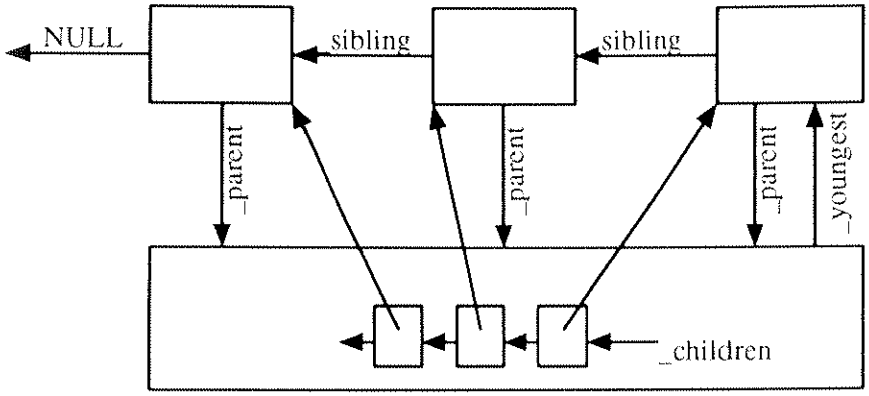

図 4.8: 親子の参照関係

配置したい場合、子であるボタンコンポーネントは親であるウインドウコンポーネントが作成された後でなければ作成できない。したがって、ウインドウを作成する API 関数は initialize() で呼ぶ必要がある。一方、親ウインドウのサイズをボタンがすべて収まるように最適化したいのであれば、それはすべてのボタンコンポーネントが配置されてからでなければできない。したがって、ウインドウサイズの最適化はmanage() 中に記述する。

4.2.3 NutsManager クラスでのメッセージ処理の流れ

NutsManager クラスのコンポーネントは木構造の中間ノードであり、メッセージ探索において、受信者探索を木構造全体に伝搬させるように探索メソッドをオーバライドしている。例えば、child 方向探索であるsendMessage() では、以下のようにオーバーライドしている。NutsCore クラスでの定義は、

```
NutsCore *NutsCore::sendMessage(){
    this->receiveMessage();
}
```

である。これは自身に配信されたメッセージをreceiveMessage() ポートに転送しているだけである。これに対して、NutsManager クラスでは、

```
NutsCore *NutsCore::sendMessage(){
   this->receiveMessage();
   listHandler &h = _child.openList();
   while(h)_child.getNextObj(h)->sendMessage();
   _child.closeList(h);
}
```

のように、自身のreceiveMessage() ポートに転送した後、その子コンポーネントにsendMessage() の探索を伝搬している。

表 4.5: NutsManager クラスで認識するメッセージ

How many children do you have?	保持している子コンポーネントの数。
Which is the youngest child?	最後に構築された子コンポーネントの名前。
listup	子のノードの階層表示。
dump ps	自身をルートとする木構造のポストスクリプトダンプ。

表 4.6: NutsApp のメンバ変数

変数	内容
`NutsListenerList _listener;`	イベント待ちコンポーネントのリストを保持する。

4.2.4 NutsManager クラスで認識可能なメッセージ

NutsManager クラスが認識するメッセージとして、表4.5に示すものがある。NutsManager クラスは、NutsCore クラスのサブクラスであるので、NutsCore クラスで解釈可能なメッセージをすべて受理すると同時に、それらをすべての子コンポーネントに配信する機能がある。例えば、メッセージ "listup" は、NutsCore クラスでも定義されているが、NutsManager クラスでは、子のノードの名前を階層的に表示するようにオーバーライドしている。

4.3 NutsApp クラス

NutsApp クラスは、NutsManager クラスのサブクラスであり、その差分として部品木のルートコンポーネントになる性質を追加している。NutsApp クラスのインスタンスは、プログラム中ただ一つだけ存在し、部品木のルートコンポーネントとなる。また、メッセージ通信におけるイベントモデルのリスナリストを保持し、コンポーネントからの`sendEvent()`呼び出しを処理する。さらに、`main()` に渡された引数 (`int argc,char **argv`) を保持する。

4.3.1 NutsApp クラスのメンバ変数

NutsApp クラスは、メンバ変数として表4.6に示す 1 つの変数を持つ。

NutsApp クラスのコンポーネント (ルートコンポーネント) は、制御モデルのうちのイベントモデルを扱う機構を持つ。`_listener` は、イベントリスナ (第 3.2.2節参照) を保持するためのリストである。

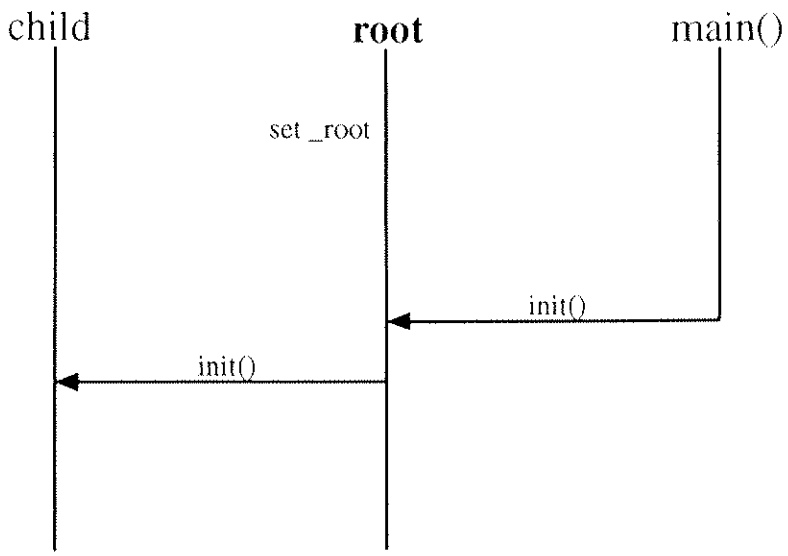

図 4.9: NutsApp クラスの初期化フェーズ

4.3.2 NutsApp クラスの初期化フェーズ

NutsCore クラスのコンストラクタはNutsCore::_root にルートコンポーネントへの参照をセットするために以下のように実現している。

```
NutsCore::NutsCore(NutsCore *parent,char *name)
{
    assert(parent == NULL && _root != NULL);
    //parent:親コンポーネントへの参照。NutsApp では NULL。
    if(!parent)_root = this;
}
```

ルートコンポーネントには親コンポーネントは存在せず、コンストラクタの親コンポーネントへの参照を表すparent 引数はNULL である。この場合に、すべてのコンポーネントに共通のスタティック変数_root に既に値がセットされていればassert[1]に失敗するようになっているので、ルートコンポーネントはプログラム中ただ一つしか存在しないことが保証される。

初期化は、main() 中で、NutsCore::_root->init(); [2]を呼び出すことにより開始する。NutsApp クラスは NutsManager クラスのサブクラスであるので、init() 呼び出しは、その子コンポーネントの初期化を呼び出し、結果的に木構造全体が初期化される。

[1]条件が満たされない場合エラーメッセージを出力して強制終了する手続き
[2]NutsApp クラス中で、friend void main(int,char **) と定義している

表 4.7: ルートコンポーネントのレパートリ

クラス名	内容
NutsApp	初期化後、標準入力からのコマンドプロンプト状態になり、Nuts のメッセージをコマンドとして入力できる。例えば、"Btn" という名のコンポーネントにそのクラス名を尋ねたい場合は、"Btn:What is your class name?" と入力する。これは、メッセージオブジェクトに変換され、探索によって "Btn" で受理され結果が表示される (結果表示の方法については第 4.1.5節参照)。
NutsXApp	X-Toolkit のレベルのウインドウアプリケーションを作成する場合のルートコンポーネント。初期化後`XtMainLoop()` に入る。
NutsMotifApp	OFS/Motif のウジェットを用いたウインドウアプリケーションを作成する場合のルートコンポーネント。

表 4.8: NutsApp クラスで認識するメッセージ

What is program name?	argv[0] すなわちプログラム名を返す。

ルートコンポーネントは、作成するアプリケーション毎に適したものを選択して用いる。例えば、ウインドウアプリケーションを作ろうとした場合、ルートコンポーネントは NutsApp のサブクラスである NutsXApp クラスのコンポーネントを用いる。NutsXApp クラスでは、`manage()` で (NutsApp クラスの manage() は初期化の最後に呼ばれる) マウス等のイベントディスパッチループに入るなど、ウインドアプリケーションに特有の動作を行う。ルートコンポーネントの種類としては、表 4.7に示すものがある。

4.3.3 NutsApp クラスで認識可能なメッセージ

NutsApp クラスが認識するメッセージとして、表 4.8 に示すものがある。ルートコンポーネントに対して、NutsManager クラスのメッセージ "listup" を送ることで、すべての部品名が木構造の階層に従って "STDOUT" という名の部品に表示される。また、"dump ps" を送ることでポストスクリプト形式のファイルがダンプされる。このファイルをプリンタに出力することによって、紙上で視覚的に木構造を把握することもできる。

表 4.9: Nutsライブラリの内訳

カテゴリ	部品数	総行数
基底部品	32	7500
X Window 部品	24	5500
OSF/Motif 部品	83	12000
ユーティリティ部品	20	4600

4.4 考察

Nutsコンポーネントの構造／制御モデルを実装したNutsの基底クラス群の、メンバ変数、初期化フェーズ、メッセージフロー、認識可能メッセージ、について説明した。Nutsコンポーネントに求められる規約は、Nutsの基底クラス群にすべて含まれているので、これらのクラスのサブクラス化を行うことによって、Nutsコンポーネントとしての資格を持った新規コンポーネントクラスを作成できる。

Nutsのライブラリはこれら基底クラス群から派生した200種類の汎用Nutsコンポーネント群 (付録A参照) からなり、そのコード量はC++でおよそ30000行となっている (内訳表4.9)。現在Nutsライブラリは、HP-UX、Linux、FreeBSD各OS上で稼働している。なお、グラフィックス部品はX Window、OSF/Motif Widgetをカプセル化したものである。

ユーティリティ部品には以下のようなものがある。

- プロセス間通信 (ソケット) を担う部品、
- 携帯電話を用いたデータ通信を行う部品、
- ポストスクリプトを出力する部品、
- ニューラルネットのノードを構築するための部品、
- デジタル地図協会発行の地図データベース (DRMA) を表示する部品、

今後は、さまざまアプリケーションドメインに特化したコンポーネントを開発していく予定である。

Nutsの基本的な構造／制御モデルは、本章で説明したNutsの基底クラス群で説明されるが、次章では、コンポーネント差分プログラミングのために拡張した、構造／制御モデルについて解説する。