ソフトウェア工学 101－2
（1994．11．18）

オブジェクトのグループ化を支援する視覚的な開発環境の実現

川上豊　加藤康之　満永豊
kawakami@nttlsl.ntt.jp　kato@nttlsl.ntt.jp　mitunaga@nttlsl.ntt.jp

NTTアクセス網研究所

オブジェクト指向言語では互いに作用を及ぼしあう素なものの集まりとしてシステムを記述する。しかし、「ものを組み合わせてできるもの」という枠組を提供しているとは考えられない。そこで、筆者らはオブジェクトをまとめ挙げるための枠組、グループオブジェクトを導入し、その記述、再利用、試験が視覚的な画面操作により動的に行えるような開発環境を作成した。この開発環境により、任意の個数のオブジェクトからなる部品を組み合わせてシステムを簡単に構成できるようになり、複雑で規模が大きいシステムの構築が行い易くなると考えられる。

Realization of Graphical Programming Environment Supporting Group Objects

Yutaka Kawakami　Yasuyuki Kato　Yutaka Mitsunaga

NTT Access Network Systems Laboratories

In object-oriented language, systems are described as consisting of atomic objects which interact with one another. But it does not explicitly provide a framework for describing non-atomic objects which are composed of objects. We introduce Group Objects for that purpose, and developped the programming environment. This environment supports dynamic defining, reusing and testing of Group Objects by graphical instruction. Software engineers can conbine components, which consist of any number of objects, into systems. And one can easily construct complex large scale software systems.

1. はじめに

一般に、プログラミングは、システムの実現に必要な機能を考えるというトップダウン的な過程とソフトウェアの部品を作成し、それらを有機的に結合していくボトムアップ的な構築過程とが交錯する複雑なものである[4]。オブジェクト指向の分野でもこうしたトップダウン、ボトムアップの概念が両立している。トップダウン設計ではシステムをサブシステムに分割、詳細化していき、もっとも素な部分をプログラミング言語上でオブジェクトとして実現する。また、オブジェクト指向言語における継承は、サブクラス化により既存の部品を再利用して新しい部品を作成し、それらを組み上げることによってボトムアップ的にシステムを構成することを支援する機構である。

しかし、トップダウン的な手法とボトムアップ的な手法が融合されているとは考えられない。その原因の1つとして、プログラマが暗黙の内に行っている、局所的なトップダウン設計に基づく(ボトムアップ的な)実装を簡単に試験できないことが挙げられる。動的に「オブジェクトの集まりをオブジェクトとみなす」ことが実現できる機構を導入することにより、設計を簡単にプログラムに反映でき、そうしてできた1つのオブジェクト(とみなせるもの)を試験して正当性が確認された後で部品として使用できる。これにより、トップダウンとボトムアップを適切に交互させながらプログラミングを進めることが可能となる。

著者らは、オブジェクトの集まりを、メッセージ通信などオブジェクトと同様のインターフェースをもつ枠組、グループオブジェクトとして動的に記述し、再利用、試験することが可能な開発環境を実現した。この開発環境を通してグループオブジェクトの記述、再利用、試験が、視覚的な画面操作により簡単に実現できるようになる。このことにより、グループオブジェクトの変更と試験を交互に繰り返し、試行錯誤しながら相互作用の疎密に応じたオブジェクトのまとめ挙げが行えるようになる。

本稿では、グループオブジェクトの概念について述べた後、グループオブジェクトを動的に作成、試験する開発環境の実現、将来への展望について述べる。

2. グループオブジェクト

ここでは、互いに関連の深いオブジェクトの集まりをあるひとつのオブジェクトとみなし、それをグループオブジェクト(以下、GOと略す)と呼ぶ。「オブジェクトとみなす」という表現を用いたのは、メッセージを配送することができるというようなオブジェクトと同様のインターフェースをGOの枠組がもつことを強調するためである。GOは、オブジェクト指向データベースの分野で使われている複合オブジェクトの考えに近いものであるが、これとは異なる。そこで、複合オブジェクトとの違いに注目してグループオブジェクトについて詳しく述べる。

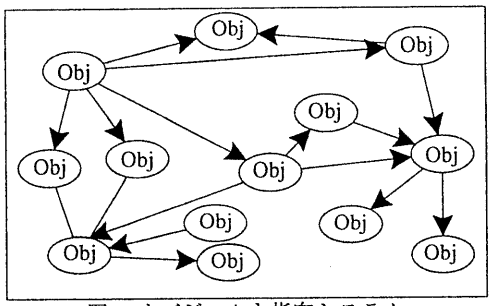

図.1. オブジェクト指向システム

インスタンス変数にオブジェクト識別子をもつことにより、いくつかの部品となるオブジェクトから構成されるオブジェクトを複合オブジェクト[3]と呼び、複合オブジェクトとその構成要素となっているオブジェクトとの関係をIS-PART-OF関係という(図.2 参照)。システム設計者は、IS-PART-OF関係が階層構造になるように、つまり、複合オブジェクトがその構成要素であるオブジェクトの集まりを代表するものとみなせるように設計を試みる。これがうまくいくと、階層の上位のオブジェクトが下位のオブジェクト群を代表するとみなせ、GOが実現できる。しかし、一般には、こうした設計が必ずしも実際のプログラムに反映されるとは言えない。何故なら、プログラミングの過程で、階層の親と子の間だけでなく、叔父、叔母等の他のオブジェクトとも相互作用させねばならなくなり、階層構造が崩れてしまう可能性があるから

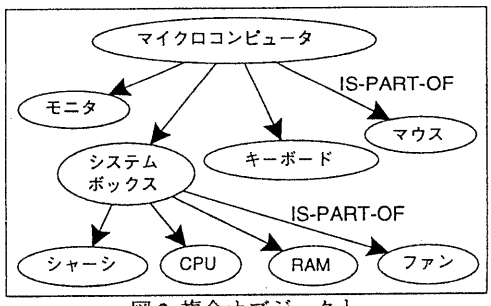

図.2. 複合オブジェクト

である。また、複合オブジェクトのような静的な機構に対して部分的な試験を簡単に行う機構を導入するのは難しい。

よって、GOを実現する手段として複合オブジェクトによる階層構造をつくることは、オブジェクト指向データベースのようにデータの静的な観点に重きをおいたシステムでは有効であるが、プログラミングのような動的な要素の多い場面では不適切であるといえる。

オブジェクト指向言語ではシステムを相互作用する、(GOでないという意味で)素であるオブジェクトの集まり[6]ととらえる。複合オブジェクトのように通常のオブジェクトを他のオブジェクト群を代表させるために用いることは、あるオブジェクトのインスタンス変数にIS-PART-OF関係という特別な意味をもたせ、素であるというオブジェクトの性質を変えてしまう。そこで、GOにおいては、システムを構成する素なオブジェクト群とは独立に、オブジェクトをまとめあげるための枠組を追加する形で導入することにする(図.3参照)。こうしたGOという枠組が満たすべき条件は4つ存在する。

(1) **GO**が互いに作用しあうオブジェクト群を代表することができる。

複合オブジェクトによるまとめ挙げでは、できるだけオブジェクト間の相互作用をIS-PART-OF関係に限定するという難しい設計を必要とする。GOでは、システムを相互作用する素なオブジェクトの集まりとみなすというオブジェクト指向の概念を崩すことなく、オブジェクト間の相互作用の疎密を考慮することによる自然なまとめ挙げを行うことができる。図.3において、閉曲線で囲まれた横線の部分がGOを表す。図.3では、オブジェクト3つを要素としてもつGOと5つを要素としてもつGOとがある。

(2) 通常のオブジェクトと同様に、あるオブジェクトから**GO**にメッセージを配送することが可能である。

GOは要素となる全てのオブジェクトを参照可能で、それらのオブジェクトに属するメソッドのうち登録されたものに対して、通常のオブジェクトと同様に、外からGOにメッセージを配送することができる。図.3において、GOを表す閉曲線に、参照を表す矢印が出入りしている。これは、GOが通常のオブジェクトと同様に参照されたり、メッセージを配送したり、されたりできるということを表す。

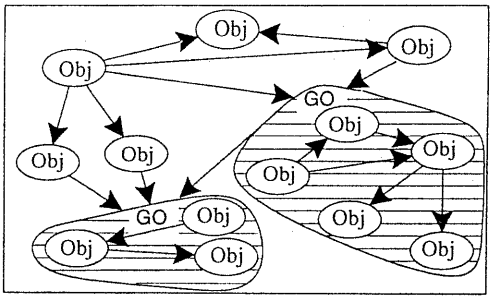

図.3. グループオブジェクト

(3) **GO**を含むオブジェクト群を新たに**GO**として再帰的にまとめ挙げることが可能である。

GOは別のGOの要素となり得る。よって、GOを再帰的に多段に構成することにより、まとめ挙げの階層構造を自然に表現することができる。図.3にはGOを要素として持つGOは無く、一段の階層しか存在していない。例えば、図.3の全てを含むGOを考えてみる。そのGOは前に述べた2つのGOと5つのオブジェクトを要素として持つことになる。さらに、上述の2つのGOはそれぞれ、3つ、5つのオブジェクトを要素として持つから、全体で3段の階層構造になる。このときのGOがつくる階層構造を図.4に示す。図.4の矢印は上のものが下のものを要素として持つことを示す。

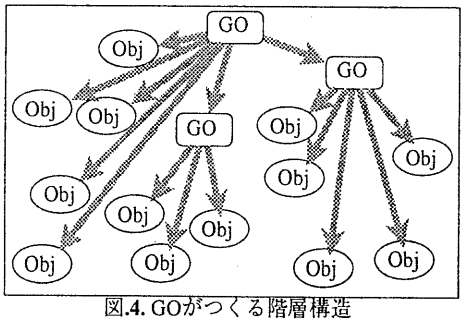

図.4. GOがつくる階層構造

(4) 同じクラスのインスタンスをもつ異なる**2**つの**GO**がシステムの中に共存しても問題が生じない。

GOを開発環境の中で再利用・試験する際に、ダイナミックにGOをロードする機能が必要となる(後述)。このときに、ロードする単位をクラスにすることにより、あるクラスが別のGOで既にロードされているため、GO全体のロードに失敗するという問題が生じなくなる。

3. 開発環境の実現

本節では、2節で述べたGOの具体的な実現法、開発環境について、詳細に述べる。この開発環境はNeXTコンピュータ(NeXT Step 3.2J)上のObjective-C言語と、GUI構築ツールであるInterfaceBuilder[5]を用いて、それらを拡張する形で作成した。この開発環境の下でGOの記述、再利用、試験がコーディングせずに、視覚的な操作のみによって実現できるように実装した。

3.1. グループルートオブジェクト

2節で述べたGOを実現するために、各々のGOは以下に述べるようなグループルートオブジェクト(以下、GROと略す。)をもつ。

(1) **GRO**は**GO**に属する全ての静的なオブジェクト(もしくは、**GO**)への参照を保持する。
(2) **GRO**は**GO**に属するオブジェクトのメソッドのうち、登録されたもののみ**GO**の外に公開する。
(3) **GRO**は**GO**に属するオブジェクトのクラスや公開されたメソッドを実行時にロードする。

まず、InterfaceBuilderについて簡単に述べた後、これらの3つの条件について概念やInterfaceBuilderに関連した実装を詳しく述べる。

3.1.1. InterfaceBuilder

InterfaceBuilder(以下、IBと略す)はユーザインターフェース(UIと略す)を視覚的に定義することにより、APの設計や構築を支援するためのツールで、あるひとつのオブジェクト(オーナーオブジェクト)から繋がっている一連のUI用オブジェクト群をファイル(nibファイル)にアーカイブして管理する。このファイルは大まかに以下の3つの情報を含む。

(1) オーナーオブジェクトへの参照
nibファイルとAPとの媒介の役割をするオブジェクトで、nibファイルではインターフェース情報(ヘッダファイルの情報)のみを持つ。

(2) UI用オブジェクトのリスト
パレットと呼ばれるオブジェクト部品庫から持ってきたnibファイル中に(インスタンス変数の値などの)実体が存在するオブジェクトと、ユーザが定義したインターフェース情報のみをもつオブジェクトとがある。

(3) オブジェクト同志の結合関係のリスト
インスタンス変数に別のオブジェクトをつなぐアウトレット結合と、ボタンなど(ターゲット)が押されたときに別のオブジェクトのどのメソッド(アクション)が実行されるかを管理するターゲット・アクション結合の2つがある。

UIの設定は(2)のnibファイル中に実体があるオブジェクトをカスタマイズしてnibファイルに保存することで実現される。nibファイルで設定されたUIオブジェクトは以下のような手順でAPに取り込まれる。

(a) nibファイルの外部でオーナーオブジェクトをnewなどにより生成する。
(b) オーナーをそのオブジェクトに指定してnibファイルをロードする。

(a)の手順を必要とするのはnibファイルがオーナーオブジェクトの実体を持たず、インターフェース情報のみを持つからである。IBでnibファイルを開いたときの全体像を図.5に示す。図.5には、nibファイルの内容を表すファイルウィンドウ、UIオブジェクトの部品庫であるパレット、ファイルウィンドウなどで選択されたオブジェクトの設定やカスタマイズを行うインスペクタという3つのツールと、編集対象となるウィンドウ、メニューなどのUIオブジェクト群がある。

IBでは、パレットに自作のオブジェクトを追加するためのアプリケーション間インターフェース(以下、APIと略す)等、限定された部分についてだけヘッダファイルやドキュメントが公開されている。実装はこのような公開されたAPIを使って行われた。

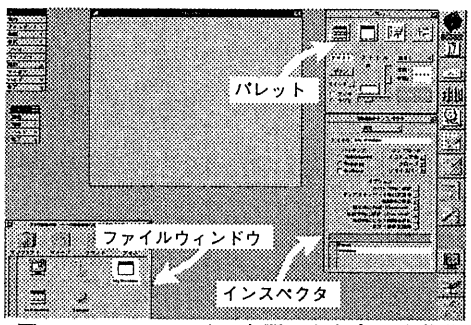

図.5. IBでnibファイルを開いたときの全体像

3.1.2. オブジェクトへの参照の保持

全ての静的なオブジェクトへの参照は、キーが文字列で値がオブジェクト識別子のハッシュテーブルである、ObjectTable(HashTableのサブクラス)を使って実現される。図.6に示したヘッダファイルにおいて、GROのインスタンス変数objectsがObjectTableのインスタンスを持つ。参照

情報の設定はnibファイルに行わせることにした。つまり、ObjectTableをパレットとして作成し、nibファイルに実体が存在することが可能となるようにし、さらにそのインスペクタ(ObjectTableインスペクタ。IBのAPIに準拠。)も作成することによって、そこで参照の設定を行えるようにした。GROで参照情報を設定した後のnibファイルの内容(ファイルウィンドウ)を図.7に示す。この図において、四角で囲まれた6つのアイコンがこのnibファイルで管理しているオブジェクト群である。左上のアイコンがオーナーオブジェクトであり、GROの参照情報を設定するnibファイルではオーナーは常にGROとなる。オーナーオブジェクトから右上のObjectTableのアイコンへと矢印が記してある。この矢印はGROのインスタンス変数objectsでのアウトレット結合を表す。ObjectTableアイコンから下段の3つのアイコンへの矢印がObjectTableを使った、GOの要素となるオブジェクトへの参照[1]を表す。こうした参照はObjectTableインスペクタのボタンを押すことで容易に一括して設定でき、その際にはハッシュテーブル作成のキーとしてアイコンの名前[2]が用いられる。GO記述の際には、オーナーがGROで、ObjectTableアイコンを含み、オーナーアイコンとObjectTableアイコンがアウトレット結合された状態のnibファイルがテンプレートとして開かれる。実際の編集は、ユーザが新たにGOに加えたオブジェクトへの参照など、テンプレートファイルからの差分のみを記述すればよい。

3.1.3. メソッドの公開

メソッドの公開は非常に重要な機能である。GOによってオブジェクト群をまとめることができても、GOに属する全てのオブジェクトの中の全てのメソッドを制限無く実行できるということでは、筆者らは情報隠蔽が不完全であると考えるからである。全てのメソッドのうち、登録されたもののみが外部からのメッセージ通信に使えるという制限を導入することによって、プログラムが非常に読みやすくなることが期待される。この機能を実現するには、既に定義され、生成されたオブジェクトに対してどのメソッドを公開するかを指定し、実際にそのメソッドを実行できる必要がある。つまり、メッセージ配送の際のメソッド探索の経路を動的に変更できなければならない。

[1] この参照はアウトレット結合でも、ターゲット・アクション結合でもない、筆者らが定義した別の結合である。

[2] 右上のアイコンのObjectTable、左下のアイコンのWindow等、アイコンの直下のテキスト。IBにより自動的に付与される。

```
@interface GroupRootObject:Object
{
    id      objects;
        // ObjectTableのインスタンス。
        // 全てのオブジェクトへの参照を管理。
    char   *methods[256];
        // 公開するメソッドを管理。
    id      goPath;
        // GO初期化用ファイルの在りかを管理。
}
/* 初期化用メソッド */
- loadGroup;
- initFromGroup:(char*)fileName;
/* メッセージ配送用メソッド */
- forward:(SEL)sel :(marg_list)args;
/* 様々な情報を取得するためのメソッド */
- getObject:(char*)name;
- projectPath;
....
@end
```

図.6. GROのヘッダファイル

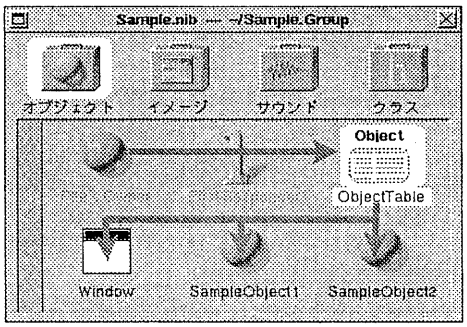

図.7. 参照設定後のファイルウィンドウ

実装に用いたObjective-C言語[5]では、ルートクラスObjectであるという制約のみをもつ、idという動的な型を導入することによって、起動されるメソッドをレシーバのオブジェクトに依存して実行時に決める機構をとる。こうしたメソッドの動的結合の機構を用いて上述の機能を実現する。図.8にGROによるメッセージ配送の部分の大まかなプログラムを示す。「forward::」メソッドはルートクラスObjectで定義されたメソッドで、動的結合による通常のメソッド探索に失敗したとき、エラー処理などのデフォルト応答を行うために呼び出される。そこで、GROでは、このメソッドを継承し、再定義した。図.6に示されたインスタンス変数methodsに登録されたメソッドかどうか調べ、もしそうであれば、そのメソッドをもつGOの要素へとメッセージを配送できるようにした。この結果、GROの生成後に公開されるメソッド

の登録を行うことが可能となるため、IB上でメソッドの登録を行えるようになる。IB上でGROにメソッドを登録するのに用いるパネル[3]を図.9に示す。図.9の「公開メソッド:」というラベルのついた欄に現在編集中のメソッドが表示される。この例では、「[Custom View0 redisplay:]」という文字列が表示されている。空白で区切られた前者が作成中のGOの要素であるオブジェクト名で、後者がそのオブジェクトのもつメソッド名を表す。この欄に上記の書式でメソッドを書き入れて追加や削除のボタンを押すと、公開メソッドを管理するGROのインスタンス変数objectsに結果が反映される。2つのボタンの下のビューは現在、公開されているメソッドのリストを表示している。以上の結果は下の「GOをセーブ」というボタンを押したとき、「METHODS」というファイルに格納される。

3.1.4. GOのロード

全てのGOはルートとしてGROという同じクラスのオブジェクトをもつ構成となっている。つまり、GROが生成された直後はどのGOでも違いは無く、GROの初期化処理において個々のGO間の違いがGROに反映されることになる。GROは初期化の際に以下の情報を順番に設定する。

(1) GOに属する全てのオブジェクトのクラス情報(CLASSESファイル)
(2) GOに属する全ての静的なオブジェクトへの参照の情報(nibファイル)
(3) GOで公開されたメソッドの情報(METHODSファイル)

先ず、CLASSESファイルで指定されたクラスのオブジェクトファイルをロードする。この際、ロードされるGOがシステム中の別のGOと同じクラスをもっていると、クラス名が衝突し、リンクしてある全体のオブジェクトファイルのロードが失敗する。これを避けるため、リンクせずに1つのクラスのオブジェクトファイルを単位として動的ロードを行う。その後、nibファイルをロードすることによりGOに属する静的なオブジェクトを生成し、GROからの参照情報を設定する。但し、動的なオブジェクトはここでは生成されない。それらの生成はGOの枠組によって管理されないからである。これは静的なオブジェクト中のあるメソッドなど個々のオブジェクトによって管理される。最後に、METHODSファイルから、公開するメソッドを設定する。このような初期化処理を行うことによりGROがGOの枠組を満足するものになる。

[3] このパネルはIBに元々ついていたものではない。筆者らが作成して、動的ロードによりIBに取り込んだものである。

```
- forward:(SEL)sel :(marg_list)args
  // sel は呼び出されたメッセージのセレクタ。
  // args はそのメッセージの引数リスト。
{
  char  *calledMethName
    = (char*)sel_getName(sel);
   // 呼び出されたメソッド名。sel から取得。
  char  *objName;
   // 登録されたメソッドをもつオブジェクト名。
  char  *methName;  // そのメソッドの名前。
  id      object;  // そのメソッドをもつオブジェクト。
  int      i = 0;
  while( methods[i] ) {
    objName = getObjName(methods[i]);
    methName = getMethName(methods[i]);
    object = [self getObject:objName];
    if( !strcmp(methName, calledMethName) )
     // 呼び出されたメソッド名と同じ名前の
     // メソッドが登録されている場合。
    {
      if( [object respondsTo:sel] )
       // object が sel に応答できる場合。
      {
        return [object performv:sel :args];
         // そのオブジェクトに sel を実行させる。
      } else {
        return [object forward:sel :args];
         // そのオブジェクトのエラー処理へ。
      }
    }
    i++;
  }
  [self doesNotRecognize:sel];
   // 呼び出されたメソッドが登録されていない
   // とき、エラー処理を行う。
  return self;
}
```

図.8. GROのメッセージ配送機構

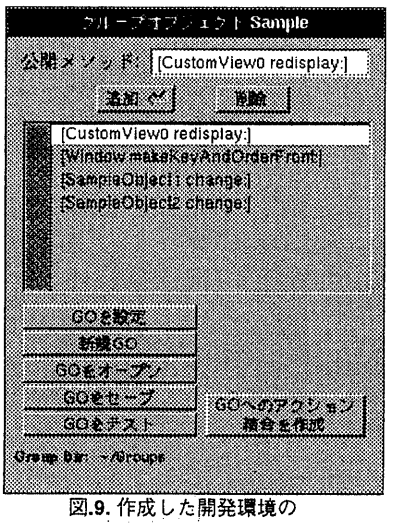

図.9. 作成した開発環境のメソッド登録用パネル

3.2. GOの再利用

一般に、オブジェクト指向言語を使用することにより、以下の2つの点で再利用性が向上するといわれている。

(1) インターフェースと実装を分離することにより、既存のオブジェクトの振る舞いを容易に理解でき、そのままシステムに組み込める。

(2) サブクラスを作り、差分をプログラミングすることにより、既存のオブジェクトを変更した新たなオブジェクトを簡単に作成できる。

このGOを支援する開発環境では、主に(1)の観点での再利用、つまり、「そのままシステムに組み込む」形式の再利用を支援している。本節では既存のGOを簡単に再利用できることを示す。

GOを支援する開発環境ではGOをパレットの中に組み込んでいる。このことにより、IBの中で、パレットに存在する他のUI部品と同様に実体が存在する形式でGOをnibファイルに組み込むことができる。図.10にGOを含むパレットを示す。このパレットから「Group Object」という名前のアイコンをつかみ、ファイルウィンドウに投げこむとファイルシステムを表示するブラウザが開く。このブラウザにおいて、再利用したいGOの初期化用ファイルが存在するディレクトリを選択すると、GOのロードが行われ、指定したGOがファイルウィンドウに現れる。以上の視覚的な画面操作により、以前に作成したGOを階層の上位のGOの中などで簡単に再利用することができる。

また、GOに属するオブジェクト群のメソッドの中で、外部に選択的に公開する機能(3.1.3参照)を採用することよって、GOの振る舞いを容易に理解することが可能になる。このことも、再利用性の向上に貢献すると考えられる。

3.3. GOの試験実行

この開発環境では記述されたGOを簡単に、しかも様々な状況に対処できるように柔軟に試験することができる。これを実現するために、試験実行は2つの段階をもつ。

(1) nibファイル編集によるオブジェクトや結合の変更という静的な配置の決定。

(2) それぞれのオブジェクトやGOの相互作用を記述し、試験を実行。

図.9のパネルの「GOをテスト」というボタンを押すと、開発環境によって、図.11に示したようなテンプレートのnibファイルが自動生成される。ここで生成されるnibファイル中の全てのオブジェクトは実体がファイルの中に存在するため、nibファイルをロードするだけでAPの実行が可能である。図.11の右上にあるGO Testerアイコンはテスト用のボタン1つからなるAPを簡易に作成できるオブジェクトを表し、もう1度「GOをテスト」のボタンを押すと、図.11のnibファイルを実行して図.12のウィンドウが現れる。GO Testerはこのウィンドウに加えてほぼ自分自身と同じインターフェースをもつオブジェクト(CustomClass)を管理する。図.12のテキスト編集用のビューに試験したいGOの機能を表すプログラムを書いて、「式の評価」ボタンを押す。すると実行時にコンパイルが行われ、それをロードしてCustomClassオブジェクトを生成し、nibファイルで設定されたGO Testerと他のオブジェクトとの結合をCustomClassオブジェクトに再現する。つまり、CustomClassにGO Testerの振りをさせるのである。「実行」ボタンが押されると先程、テキスト編集用ビュー

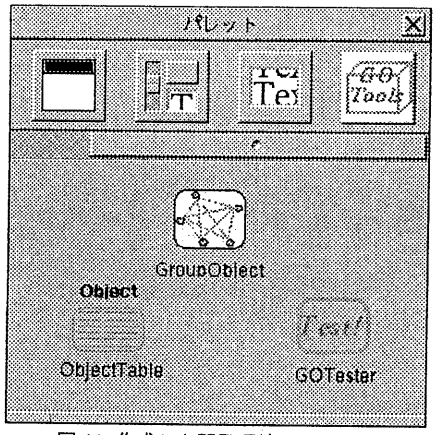

図.10. 作成した開発環境のパレット

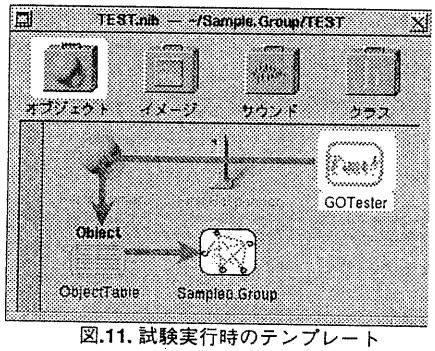

図.11. 試験実行時のテンプレートnibファイル

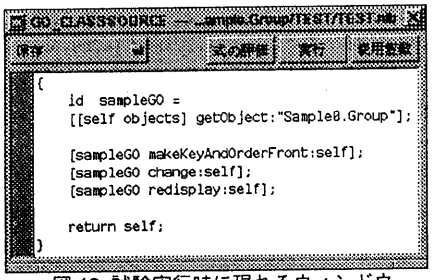

図.12. 試験実行時に現れるウィンドウ

に記したCusotmClassのメソッドを呼び出し、試験実行を行う。

4. 関連研究と将来への展望

本節では、contracts[2]、IntelligentPad[1]という2つの関連した研究と本開発環境との比較、考察を行う。contracts、IntelligentPadはそれぞれオブジェクト間の相互作用を隠蔽している点、既存の部品を組み合わせることによりAPを作成することができる点で本開発環境と関連が深い。以上の2つの観点から本開発環境の将来の展望についても触れる。

4.1. 相互作用の隠蔽

Helm, R.らは、システムを構成する各々の素なオブジェクトの定義(クラス定義)とは独立に相互作用の1つの側面を記述するための枠組としてcontractsを導入した。例えば、様々なViewの中の1つが変更されたら他の全てのViewに告知し、変更を反映するといったオブジェクト群の相互作用をSubjectViewと名付け、クラス定義とは別にメッセージの流れを記す形式、contractとして記述する。彼らはこうしたcontractsに継承を導入するなどして、これを再利用の単位にすることを意図している。

contractsは設計の詳細をクラス定義と別に記している点、継承をもつ点などがGOより優れるが、大規模なシステムのプログラミングでは、相互作用全てをcontractsとして記述するのは煩雑であり、その点、GOではオブジェクト群を相互作用も含めてそのままの形で隠蔽しているので非常に簡単にすむ。

GOでは隠蔽されたオブジェクト群の相互作用を分かりやすい形で抽出していないため、変更して再利用を行うことは充分に支援されているとはいえない。これを改善するために、GOの外部に公開するメソッドについて、内部のオブジェクト群とのメッセージのやり取りをcontractsと同様の形式でまとめたものを自動的に作成する技術が必要であると筆者らは考えている。

4.2. 部品の組み合わせによるAP作成

田中譲らが提唱したIntelligentPadは、紙(パッド)として様々な機能を実現し、「貼る」というメタファによりパッドを合成し、複雑な機構をもつパッドを作成することにより、APの作成を行うシステムである。これは、以前に作成したGOをIBを通して簡単に再利用できることに似ている。しかし、IntelligentPadは紙に貼るというメタファから生じる制約のため、汎用の開発環境として使えるか検討の余地があるのに対し、GOはそれ自身もオブジェクトのように使えるため、汎用の環境として使用できる。

IntelligenntPadではパッドの部分構造を条件として検索を行う方法[1]が検討されている。GOにおいても「そのまま組み込む」形式の再利用をさらに進めるため、既存のGO群から目的に適したGOを検索する方法を検討することが必要であると筆者らは考えている。

5. まとめ

本稿ではオブジェクトをまとめ挙げるための枠組、グループオブジェクト(GO)について述べ、GOの記述、再利用、試験が視覚的な画面操作により簡単に行える開発環境を紹介した。

こうした開発環境を導入することにより、プログラマがGOの変更と試験を繰り返し、試行錯誤をしながら、オブジェクト間相互作用の疎密に応じた情報隠蔽・再利用の単位を作ることが可能となる。よって、トップダウンとボトムアップを交互させながらプログラミングを行い易くなり、複雑で規模が大きいシステムの構築を行い易くなると考えられる。

参考文献

[1] 赤石, 田中:"IntelligentPadにおける部分情報検索", 情処論, Vol.35, No.2, pp.232-242, 1994

[2] Helm,R. et al: "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", ECOOP /OOPSLA '90 Conference Proceedings Special Issue of SIGPLAN Notices Vol.25, pp.169-180, 1990

[3] Kim, W. et al: "An Object Modeling Technique For Conceptual Design", ECOOP'87, Lecture Notes in Computer Science, Springer-Verlag, New York, pp.192-202, 1987

[4] メイヤー, B.: "オブジェクト指向入門", 酒匂寛, 酒匂順子訳, 二木厚吉監訳, アスキー, 1990

[5] NeXT: "NeXT Step Concept", NeXT Computer Inc., 1992

[6] Watari, S. et al: Extending Object-Oriented Systems to Support Dialectic Worldviews", Proceedings of Advanced Database System Symposium '89, pp.161-168, 1989