261.
(10) 松本，ほか："市松模様蛇行CCD撮像素子"，信学技法，SSD 77-3 (1977-04).
(11) Chamberlain, S.G., et al.: "MTF simulation including transmittance effects and experimental results of charge-coupled imagers", IEEE Trans. Electron Devices, ED-25, 2, p. 145 (Feb. 1978).
(12) Kamins, T.I., et al.: "Photosensing arrays with improved spatial resolution", ibid., p. 154.
(13) White, M.H., et al.: "Characterization of surface channel CCD image arrays at low light levels", IEEE J. Solid-State Circuits, SC-9, 1, p. 1 (Feb. 1974).
(14) White, J.M., et al.: "A multiple-gate CCD photodiode sensor element for imaging arrays", IEEE Trans. Electron Devices, ED-25, 2, p. 125 (Feb. 1978).
(15) Anagnostopoulos, C.: "Signal readout in CID image sensors", ibid., p. 85.
(16) Bluzer, N., et al.: "Buffered direct injection of photocurrents into charge-coupled devices", ibid., p. 160.
(17) Kim, J.C.: "InSb charge-injection device imaging array", ibid., p. 232.
(18) Schroder, D.K., et al.: "Free carrier absorption in silicon", ibid., p. 254.
(19) Sei, H., et al.: "A meander channel CCD linear image sensor", ibid., p. 140.
(20) 清，ほか："CCDイメージセンサとシグナルプロセッサ"，信学技報，ED 77-2 (1977-04).
(21) Barbe, D.F., et al.: "Signal processing with charge-coupled devices", IEEE Trans. Electron Devices, ED-25, 2, p. 108 (Feb. 1978).
(22) Blouke, M.M., et al.: "Three-phase, backside illuminated 500×500 CCD imager", ISSCC Digest of Technical Papers, p. 36 (Feb. 1978).
(23) 宮川，ほか："MOS型イメージセンサを用いた単一撮像板方式カラーカメラ"，昭52テレビ学全大，4-11.
(24) 清水，ほか："固体撮像板を用いた固体単板式カラーカメラのシミュレーション"，同上，4-12.
(25) 大井，ほか："2板式カラーカメラの実験"，昭53テレビ学全大，3-21.
(26) 佐藤，ほか："単一撮像板カラーテレビカメラの解像度の検討"，同上，3-19.
(27) 梅本，ほか："単板カラーテレビジョンカメラの試作"，同上，3-20.

UDC 681.32.06 : 519.681/.683

# プログラムのモジュール化技法

（紹介）中所武司

中所武司：正員　日立製作所システム開発研究所
Modularization Techniques for Software Programs. By Takeshi CHUSHO, Regular Member (Systems Development Laboratory, Hitachi Ltd., Kawasaki-shi).

資料番号：昭 54-14 [海外研究動向-2]

## 1. まえがき

ソフトウエアの信頼性，生産性向上のためのプログラミング方法論は，ソフトウエア工学の主要テーマの一つであり，プログラムの構造化，抽象化の研究が盛んである．これらの技法はプログラムのモジュール化を促すことから，ここではモジュール化技法としてまとめるが，その目的は，プログラムを幾つかのモジュールに分割して理解しやすい構造にすることにより，プログラミング，テスト，保守，拡張およびそれらの管理を容易にすることと，情報のカプセル化による信頼性向上である．これまでの研究経過は，大まかには，

70年代初期：方法論の提案
同上　中期：方法論の具体化と言語の提案
同上　後期：言語処理系と関連ツールの開発

と考えられる．

本文では最近のモジュール化技法の実用化に関する論文を中心に紹介する．

## 2. モジュール分割の方法

まず，幾つかの代表的なモジュール分割方法について述べる．

（1） 階層構造

プログラムの階層構造化は，古くはDijkstraが多重プログラミングシステムTHE[1]に適用している．彼は，最もハードに近いところからオペレータに至るまでの間を5段階の機能レベルに階層化した．そのため，個々のレベルのプログラムは並行処理を行うにも係わらず逐次処理の概念の延長で考えることができ，理解しやすい構造になっている．

（2） Information hiding[2]

これはモジュール分割の評価基準として導入された概念で，各モジュールの設計時の詳細情報を他のモジュールから隠すと共に，そのインタフェースをインプリメンテーションに無関係に設定することにより，プログラムの柔軟性と理解容易性を向上させる．

（3） 段階的詳細化[3],[4]

これは，新たなプログラムの開発時に，まず大まかなことを決め，次第に詳細な設計に進む過程をそのままプログラム化する技法である．すなわち，各段階の抽象レベルで閉じたプログラムを作成し，次段階ではその詳細化を行うという過程を繰返し，最後に実際の計算機で処理できるレベルに達してプログラミングを終了する．このようにして開発されたプログラムはモジュールの階層構造になる．

（4） データの抽象化[5]

これは，プログラムを各抽象レベルで表現されたモジュールの階層構造とする点で前項と同じであるが，手続きよりもデータの抽象化を重視する．すなわち，データの利用者はそのデータの詳細構造や操作アルゴリズムを知る必要はないので，それらはまとめて定義し，カプセル化する．そして，データの利用者には抽象データ型名とそのオペレーション名の集合だけが与えられるので，その型のデータへのアクセスはこれらのオペレーションを通じてのみ行われる．

（5） 複合設計[6]

これは，個々のモジュールの強度が強く，モジュール相互の結合度が弱くなるようなモジュール分割を良しとする．モジュール強度は高い順に，機能的，情報的，連絡的，手順的，時間的，論理的，暗合的レベルが設定され，結合度は弱い順

に，データ結合，スタンプ結合，制御結合，外部結合，共通結合，内容結合に分類される．従って，個々のモジュールは機能単位に構成し，情報の受渡しは引数で行うのが良い．

（6） uses 関係の木構造化[7]

モジュールAがその仕様書の内容を完遂するためにモジュールBの正しい実行が必要な場合，“A uses B”という．そして，モジュール間の関係は従来のような単なる引用関係でなく，この uses 関係では握する．プログラムは uses 関係が木構造になるようなモジュール構造にするのが良く，そのため，“A uses B”の関係は以下の4条件を満たす場合に限られる．

（i） Bを用いることによりAが簡単になる
（ii） BがAを用いないことでBが複雑にはならない
（iii） Bを含みAを含まぬ有用なサブセットがある
（iv） Aを含みBを含まぬ有用なサブセットはない

## 3. モジュール構造の定量的尺度

### 3.1 複雑度 (Complexity)

個々のモジュールの複雑度の定量的尺度として，cyclomatic number が提案[8]されている．これは，制御グラフ $G$ の節点数 $n$，辺数 $e$，連結要素数 $p$ として，

$$V(G)=e-n+p$$

で定義される．$p$ は通常の出入口が各1個のグラフでは2である．この指標は，構造化プログラムの選択文と繰返文の数に1を加えた数に等しく，$G$ が連結平面グラフのときの領域数にも等しいことから妥当性が与えられる．実際のプログラムの測定結果から $V$ が10以下のものが良いモジュールと考えられる．特に構造化の度合を重視する場合は，出入口が各1個の真のサブグラフの数 $m$ を $V$ から差引いた指標を用いれば，構造化プログラムはすべて1になる．

この指標は分岐の判断に用いられる変数の数を考慮していないので不都合の生じる場合がある．そこで，別に制御変数に注目した指標が提案[9]されている．まず，制御変数 $v$ の複雑度を

$$C(v)=D(v)\cdot I(v)\cdot(1/n)$$

とする．ここで，$D(v)$ は $v$ の値を変更および参照するモジュールと $v$ を宣言するモジュールとの関係の度合を表し，$I(v)$ は $v$ によって生じるモジュール間相互作用の度合，$n$ はモジュール数である．そして，モジュール $p$ の複雑度は，この $C$ を用いて，

$$M(p)=[f_p\cdot X(p)]+[g_p\cdot Y(p)]$$

とする．右辺第1項は $p$ をコールするための制御変数の複雑度 $C$ の和，第2項は $p$ が他のモジュールをコールするために用いる制御変数の複雑度 $C$ の和である．結局，プログラムの複雑度は各モジュールの複雑度 $M$ の和で求まる．この指標はインプリメンテーションの前に計算できるので設計の見直しに使える．

このほか，従来の指標がデータの影響を含んでいない点を改め，制御構造とデータ使用形態を合せた指標が提案[10]されている．これは，まず前章(2)，(3)で述べたような階層構造の各抽象レベルのプログラムを Hierarchical Abstract Computer (HAC) と考え，$(\Delta, I, \Omega, T)$ で表す．$\Delta$ はデータ集合，$I$ は命令 (instruction) の集合，$\Omega$ は命令コード (operation code) の集合，$T$ はデータ型の集合で，各命令は；$\omega\in\Omega$，$d_i\in\Delta$，$x_i$ を各命令のラベルとして，

$$\omega d_1 d_2\cdots d_m x_1 x_2\cdots x_n$$

とする．このとき，HAC の複雑度は，

$$\sum_{j\in I}\{\log_2|\Omega|+m*\log_2|\Delta|+n*\log_2(|I|+1)\}$$

とする．この指標は構造化プログラムに対して良い結果を示す．

### 3.2 モジュールの大きさ

モジュールの大きさは，いちおうの目安として，ソース文で50行以下とかプリンタ用紙1ページ以内が良いといわれている．少し視点は異なるが，全コンパイル費用を最小にするための最適モジュールサイズとして，高級言語は50ステートメント，アセンブリ言語は100ステートメントくらいという解析結果[11]がある．

## 4. プログラミング言語

2. で述べたようなモジュール分割法を適用するには，それに適したプログラミング言語が必要である．ここでは代表的なものを幾つか紹介する．

（1） CLU[5]

これは，2.(4)のデータ抽象化機能を有する．Simula 67 の class に類似した cluster を用いて新しいデータ型とその型に固有のオペレーションが定義され，その型のデータへのアクセスはこれらのオペレーションを通じてのみ行われる．

図1の例では，抽象データ wordbag のデータ構造と3個のオペレーションが定義されており，そのとき，他の抽象データ wordtree が用いられている．図中，rep は抽象データの詳細構造を示し，cvt は引数やもどされるデータの型が rep で定義された型であることを示す．オペレーションの引用は，その名前の前に cluster 名を $ 記号で結びつけて行う．

```
wordbag=cluster is
  create,  % create an empty bag
  insert,  % insert an element
  print;   % print contents of bag
  rep=record[contents : wordtree, total : int];
create=proc( )returns (cvt);
        return (rpe $ {contents : wordtree $ create ( ), total : 0});
        end create;
insert=proc (x : cvt, v : string);
        x. contents : =wordtree $ insert (x. contents, v);
        x. total : x. total+1;
        end insert;
print=proc (x : cvt, o : outstream);
       wordtree $ print (x. contents, x. total, o);
       end print;
end wordbag;
```

図1 CLU による抽象データ定義例

CLU の処理系は，データ型などモジュール間で必要な情報をライブラリに保存することにより，型チェックを徹底すると共に，モジュール単位の分離コンパイルを可能にしている．

（2） Alphard[12]

これは，抽象化機能と検証機能に重点が置かれた言語である．抽象化に用いられる form は，図2の例のように，specification, representation, implementation の3部から成る．

specification はオペレーション名とその引数および結果の型あるいは前置条件，後置条件などを規定する．representation はデータ構造などのオブジェクト生成に必要な情報を記述する．implementation では各オペレーションの本体が定義される．図の例の istack は，

**local** $x$ : istack

のように用いられる．

```
form istack (n : integer) =
beginform
specifications
  requires n > 0;
  let istack = <...x_i...>where x_i is integer;
  invariant 0 < length(istack) < n;
  initially istack=nullseq;
  function
    push (s : istack, x : integer)
      pre 0 < length(s) < n post s=s'~x,
    pop (s : istack) pre 0 < length (s) < n post s=leader(s'),
    top (s : istack) returns (x : integer)
      pre 0 < length(s) < n post x=last(s'),
    empty (s : istack) returns (b : boolean)
      post b=(s=nullseq);
representation…;
implementation…;
endform;
```

図 2　Alphard の抽象化機能 form の例

（3） Mesa[18]

本言語は図3の例のように2種類のモジュールを設けており，そのうちの定義モジュールはデータ型の宣言，手続き名およびその引数と結果の型の規定によるインタフェースの定義などを行う．一方，implementer と呼ばれるプログラムモジュールは変数宣言と手続き本体の定義を行う．このオブジェクトは，Simula の class と同様に複数個生成される．

モジュール間での識別子へのアクセスを規制するために，PRIVATE, PUBLIC, READ-ONLY などの属性を付加できる．

```
Abstraction : DEFINITIONS=
  BEGIN
  ......
  it : TYPE=...; rt : TYPE=...;
  ......
  p : PROCEDURE;
  p1 : PROCEDURE [INTEGER];
  ......
  pi : PROCEDURE [it] RETURNS [rt];
  ......
  END
Implementer : PROGRAM IMPLEMENTING Abstraction=
BEGIN
OPEN Abstraction;
x : INTEGER;
......
p : PUBLIC PROCEDURE=<code for p>;
p1 : PUBLIC PROCEDURE [i : INTEGER]=<code for p1>;
......
pi : PUBLIC PROCEDURE [x : it] RETURNS [y : rt]=
        <code for pi>;
......
END
```

図 3　Mesa の2種類のモジュールの例

（4） そ の 他

以上の (1)～(3) の言語はいずれも Simula 67 の影響を受けており，プログラム構造が似ている．その他では，教育用に設計された言語 Pascal を拡張し，手続きに external, entry 属性，変数に extval 属性を付加したり，手続きを引数にしたときにその型を完全に宣言させることにより，モジュール化機能を強化し，分離コンパイルを可能にしたシステム[14]がある．

又，Pascal の設計者自身が，モジュール間で用いられる手続き，型，変数を use-list と define-list で相互に指定し合うようなモジュール化機能と多重処理機能を付加して，Modula[15] 言語を設計している．

## 5. その他の実用化技法

言語のほかにもモジュール化サポートツールが開発されている．

プログラムのグローバル構造の表示用ツール MTR[16] (The Modular Tree Representation) はモジュール間の引用関係を解析し，その有向グラフの支配 (dominance) 関係に基づいて段付け (indentation) をした表を出力する．このシステムは設計段階でグローバルなプログラム構造を改良したり，モジュールのオーバレイ化するのにも有用である．

又，ある計算機センターで良く使われる Fortran ライブラリプログラム 85 個のモジュール設計度を解析した結果，平均 13 モジュールから成り，全体の 47% はモジュール間引用関係が網構造をなし，モジュールサイズは平均 148 行であったという報告[17]があるが，この解析プログラムはデバッグ，オーバレイ化，プログラマ教育に役立っている．

一方，モジュール化の欠点としてオブジェクト効率低下の問題がある．そのため，前述の Mesa, Modula では効率の悪くなる機能は除くことが言語設計の基本方針になっている．この問題の対策の一つは，引用された下位モジュールをインライン展開することであるが，この効果[16]は前述の CLU で確かめられている．

## 6. む　す　び

プログラムのモジュール化技法の有効性の確かさにも係わらず，その実用化が遅れている理由として，

（i） モジュール分割方式が具体化されていない
（ii） 適当なサポート言語がない
（iii） オブジェクト効率に不安がある

などが考えられる．本文で紹介してきたように，これらの問題も徐々に解決されてきているが，現場の設計者，プログラマにとっては特に (ii) の問題が大きい．この解決には，かつてストラクチャードコーディング機能が Fortran, Cobol などの従来言語に導入されたのと同じような方法がとれれば良いが，モジュール化技法はプログラム構造全体に関係するため難しい．従来言語に代る強力なモジュール化サポート言語の出現が望まれる．

これからは，こうした実用化のための研究のほかに，プログラムの自動検証のためのモジュール仕様記述言語の研究，あるいは，プログラム開発の初期段階の計算機処理化のための要求仕様記述システムの研究なども盛んになりそうであ

る．

## 文　献


(1) Dijkstra, E.W. : "The structure of the "THE"-multiprogramming system", Commun. ACM, **11**, 5, p. 341 (May 1968).

(2) Parnas, D.L. : "On the criteria to be used in decomposing systems into modules", Commun. ACM, **15**, 12, p. 1053 (Dec. 1972).

(3) Dijkstra, E.W. : "Notes on structured programming", Structured Programming, Academic Press, London, p. 1 (1972).

(4) Wirth, N. : "Program development by stepwise refinement", Commun. ACM, **14**, 4, p. 221 (April 1971).

(5) Liskov, B., et al. : "Abstraction mechanisms in CLU", Commun. ACM, **20**, 8, p. 564 (Aug. 1977).

(6) Myers, G.J. : "Reliable software through composite design", p. 159, Petrocelli/Charter, New York (1975).

(7) Parnas, D.L. : "Designing software for ease of extention and contraction", Proc. the 3rd Int. Conf. on Software Eng., p. 264 (May 1978).

(8) McCabe, T.J. : "A complexity measure", IEEE Trans. Software Eng., **SE-2**, 4, p. 308 (Dec. 1976).

(9) McClure, C.L. : "A model for program complexity analysis", Proc. the 3rd Int. Conf. on Software Eng., p. 149 (May 1978).

(10) Bail, W.G., et al. : "Program complexity using hierarchical abstract computers", Proc. NCC, p. 605 (June 1978).

(11) Elshoff, J.L. : "On optimal module size with respect to compilation cost", Proc. COMPSAC '77, p. 547 (Nov. 1977).

(12) Wulf, W.A., et al. : "An introduction to the construction and verification of Alphard programs", IEEE Trans. Software Eng., **SE-2**, p. 253 (Dec. 1976).

(13) Geschke, C.M., et al. : "Early experience with Mesa", Commun. ACM, **20**, 8, p. 540 (Aug. 1977).

(14) Kieburtz, R.B., et al. : "A type-checking program linkage system for Pascal", Proc. the 3rd Int. Conf. on Software Eng., p. 23 (May 1978).

(15) Wirth, N. : "Modula : a language for modular multiprogramming", Software-Pract. & Exper. **7**, 1, p. 3 (1977).

(16) Balbine, D. : "MTR- a tool for displaying the global structure of software systems", Proc. NCC, p. 571 (June 1978).

(17) Self, L.E., et al. : "Analyzing the modular design of Fortran programs", Proc. COMPSAC '77, p. 554 (Nov. 1977).

(18) Scheifler, R.W. : "An analysis of inline substitution for a structured language", Commun. ACM, **20**, 9, p. 647 (Sept. 1977).