

プログラム階層構造の生成, 処理, 文書化能力を有するテキスト・エディタ†

酒井 三四郎^{††} 落水 浩一郎^{†††}

ソフトウェアを資源として有効利用する上で, その機能・構造などを第三者に伝えるための高品質の文書は不可欠である。本論文は与えられたモジュール外部仕様をもとに, モジュール内部論理の設計と文書化を実行する過程における計算機支援の一方式について考察し, それを実現したシステムについて述べている。本システムの機能上の特徴は以下の三つである。(1)階層記述言語とその処理能力を有した会話型エディタにより, 段階的詳細化法による内部論理の設計を進める作業を支援する。(2)その過程で行われた設計上の意思決定すなわち内部論理に関する文書情報を収集し, その情報を含んだモジュールのソースコードを作成する。また, その情報の修正があった場合は対応する部分を自動修正する。(3)保守時など内部論理の把握が必要になった時に, 設計時の意思決定過程を再現するような提示を可能にする。本システムの実現によって, モジュール作成後文書を作成するという労力を大幅に減少させるとともに, 高品質の文書の作成と提示を可能にしたことによってモジュールの論理構造理解時の立ち上がり時間を大幅に短縮することができる。

1. はじめに

ソフトウェアを資源として有効利用する上で, その機能, 構造などを第三者に伝えるための高品質の文書は不可欠である。本論文は与えられたモジュール外部仕様をもとに, モジュール内部論理の設計と文書化を実行する過程における計算機支援の一方式について考察したものである。

一般に, モジュール作成者にとって第一義の関心事は, 迅速かつ正確なモジュールの実現であり, 文書作成等はなかば二次的な作業として, 高品質の文書作成にはらう労力は節約する傾向にある。一方, プログラムの修正, 改良を実施する立場の人間から見ると高品質の文書に対する要求が第一義である。

ところで, モジュール作成者がモジュール作成後, 文書化する情報とはモジュール作成中になされた設計者の意思決定を再現したものであり, 次のような理由で設計活動と文書化を同期させることが可能である。

構造的プログラミングや段階的詳細化の手法⁴⁾をモジュール作成段階に適用すれば, 作成時および作成後の論理構造の把握が容易で論理的正しさの確認しやすいモジュールの作成が可能になると同時に, その段階

における各種の設計上の意思決定は高品質の文書にとって重要な要因となる。

すなわち本システムは, (1)階層記述言語とその処理能力を有した会話型エディタにより, 段階的詳細化法による内部論理の設計を進める作業を支援するとともに, (2)その過程で行われた設計者の意思決定すなわち内部論理に関する文書情報を収集し, その情報を含んだモジュールのソースコードを作成する。また, 設計情報の修正はソースコード中の対応する文書情報の自動修正となる。さらに, (3)保守時など内部論理の把握が必要になった時に, 設計時の意思決定過程を再現するような提示を可能にし, モジュール作成後の文書作成, 保守の労力を大幅に軽減する。

本論文は, 2章でシステムの概要を述べ, 3章, 4章で各機能を詳説し, 5章, 6章で適用例と評価を示す。なお, 本文中で用いる写真は5章で述べる例題によるものである。

2. システムの概要

本システムはマルチバッファを有するテキストエディタであり, モジュールの段階的詳細化法による実現, 内部論理レベルの文書情報の収集, 提示の支援などの機能を有している(図1)。

段階的詳細化における階層を実現する抽象プログラムは一つのバッファの中に置かれる。階層構造は以下のように構成される(図2)。まず, 作成しようとするモジュールと同じ名前をもつバッファを開設し, そこに抽象プログラムを作成する。もし, その中に抽象

† A Text Editor Incorporated with a Function for Creation, Processing and Documentation of a Program Hierarchical Structure by SANSHIRO SAKAI (Graduate School of Electronic Science and Technology, Shizuoka University) and KOICHIRO OCHIMIZU (Department of Information Science, Faculty of Engineering, Shizuoka University).

†† 静岡大学大学院電子科学研究科

††† 静岡大学工学部情報工学科

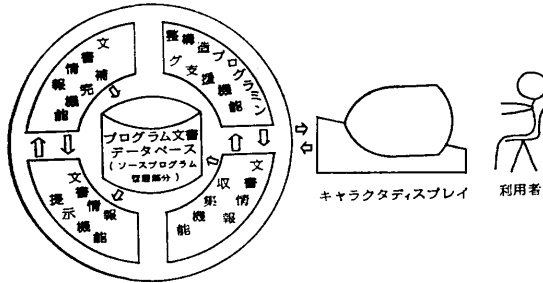


図 1 システムの機能概略

Fig. 1 Schematic diagram of the system.

マシン名が存在すればそれらと同じ名前をもつ新しいバッファを開設する。新しいバッファの中で、我々は抽象マシンの詳細化（抽象プログラムを書くこと）をつづける。もし可能なら PL/I ステートメントを書くこともできる。詳細化はすべての抽象マシンが PL/I ステートメントで置きかえられるまで続けられる。最後に、エディタは階層構造をもつバッファ群から PL/I プログラムを構築し、抽象マシンの説明を適切な位置にコメントとして挿入する。

内部論理を理解するために、本システムはレベル・チャートを与える。レベル・チャートとは抽象のレベルに応じて行さげを行った、抽象マシンの動作の説明文のリストである。我々はそれを任意のレベルから任意の深さまで見ることができし、また、説明文に対応するソースコードを見ることがもできる。

このようなシステム機能の特徴は以下の点にある。

- (1) 階層のあるレベルの抽象プログラムは一つのまとまった機能をもつが、それを一つのバッファ中に実現した。
- (2) 設計作業と文書作成の作業とを同期させて、モジュール作成と文書作成に時間的ずれが生じないようにした。
- (3) 段階的詳細化の作業に伴う設計の変更を容易にし、また、作業効率の点から作成しつつある構造の認識が容易であるようにした。
- (4) モジュールの論理構造の把握にあたっては、ソースコードと文書を会話的に対応させて見ることを可能にし、また、任意のレベルの詳しさを文書を読む

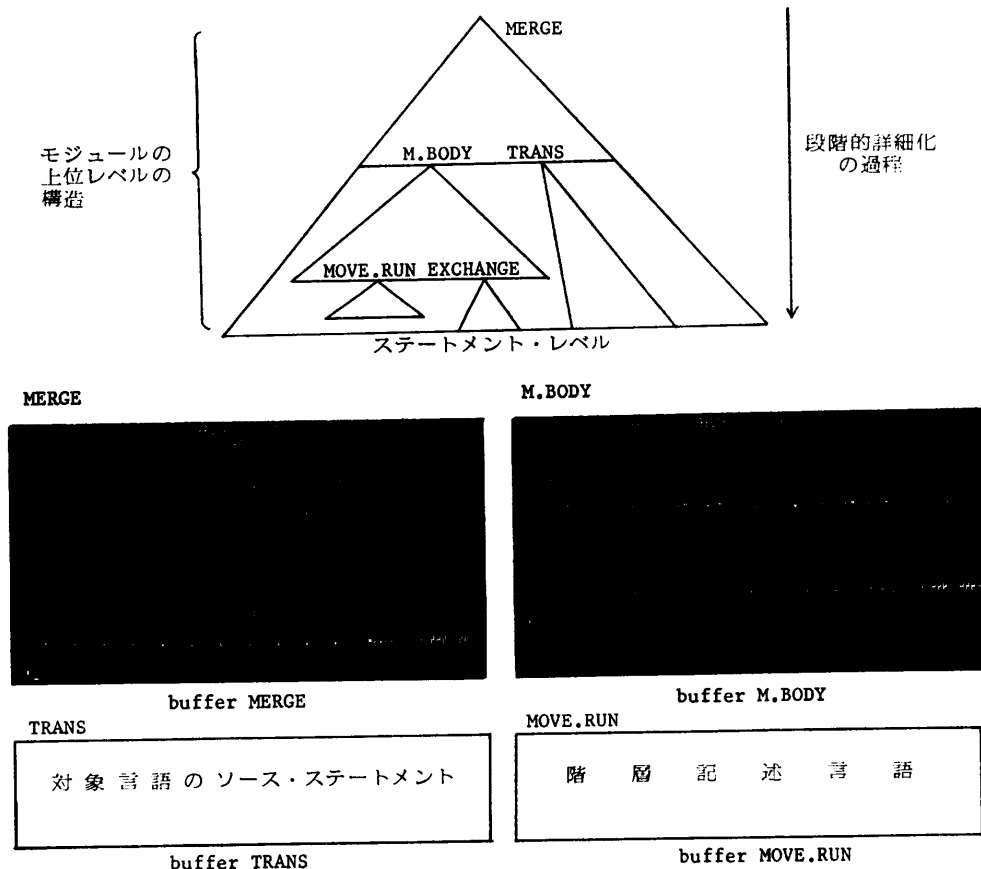


図 2 バッファによる階層構造の表現

Fig. 2 Representation of hierarchical structure by multiple buffers.

ことができるようにした。

(5) 設計作業時に収集した文書情報のレビューのために(4)と同じ機能を設計者に与え, 不十分な文書情報の補完をレベル・チャートの修正を通じて可能にした。

本システムは, ソフトウェア開発・保守支援システム PDB²⁾のサブシステム³⁾である。PDBの下でシステム設計, プログラム設計を経てモジュールの外部仕様書ができた後利用される。

以下の節で各機能について説明する。

3. 整構造プログラミング支援機能

段階的詳細化法によるプログラミングは, それを手作業で行う時の煩わしさを取り除くことによって, さらに有効な方法となる。本システムは階層記述言語とその処理能力を有した会話型エディタによって, 整構造モジュールの作成を支援する。

3.1 階層記述言語

階層構造を表現する方法には図式による方法⁵⁾と設計用言語による方法⁶⁾がある。本システムでは機械処理の効率と修正の容易さを重視し, 以下に述べるような言語を設計した(表1)。

本言語はある抽象のレベルに対応して, 一つの層を記述する。その層を構成する要素を機能単位^{*}と呼ぶ。機能単位をその特徴から三つに分類する。第一は, その機能単位がさらに詳細化を必要とするもので, 表1の(1)のように表す。DCL, COND, BLOCK というキーワードは機能単位の役割を表し, DCL はデータやデータ構造を確保する抽象マシン, COND は条件

表 1 階層記述言語の概要

Table 1 Summary of hierarchy description language.

(1)	さらに詳細化される機能単位の表現 DCL [機能単位名: 機能単位に関する自然語による説明] COND [機能単位名: 機能単位に関する自然語による説明] BLOCK [機能単位名: 機能単位に関する自然語による説明]
(2)	ソース・コードで直接表現する場合 [想定するプログラム言語のステートメント(列)]
(3)	エントリ文(またはプロセジャ文)とリターン文(エンド文)の表現 ENTRY [想定するプログラム言語のエントリ文: 自然語による説明] RETURN [想定するプログラム言語のリターン文: 自然語による説明]
(4)	機能単位間の関係を表すキーワード IF THEN ELSE DO BEGIN END WHILE

* 2章で抽象マシンと呼んでいたもので, 本言語においては機能単位と呼ぶ。

を判断する抽象マシン, BLOCKはある演算を実行する抽象マシンをそれぞれ表す。第二は, その機能単位を想定するプログラム言語のステートメントまたはステートメント列で直接表現する場合で, 表1の(2)のように表す。第三の機能単位は, そのモジュールの入口と出口を表現し, 表1の(3)のように表す。この場合, 想定するプログラミング言語(たとえば PL/I)にも, 入口と出口を表すステートメントが存在する。しかし, それらは論理構造を構築する上で一番初めに認識するものであり, その入口から入った時の機能を代表するものとみなせる。とくに, 一つのモジュール内に複数の入口を持つ場合, モジュールの第一レベルの設計は, 入口と対応する出口の配置である。よって特別に用意した。

そして, これらの機能単位は表1の(4)に示すようなキーワードを用いて, 接続, 選択, 繰り返しの構造を作る。記述例はすでに図2に示した。また, 構文規則を付録に示す。

3.2 テキスト単位の編集機能

モジュールの段階的作成では, 異なった抽象のレベルに対応してそれぞれ層が構成される。よって, モジュール設計の修正とはそれぞれの層における機能単位を他の機能単位で置きかえるというように, その層を記述しているテキストの編集にほかならない。この作業は通常のテキストエディタの機能で行えるので, 表2に示したような QEDX³⁾のコマンド体系に準ずるものを用意した。

表 2 テキスト編集のコマンド一覧表

Table 2 The table of commands for text editing.

コ マ ン ド	機 能
I, A, C, D, M S J	テキストの挿入, 付加, 交換, 消去, 移動 文字列の交換または消去 ポインタの移動
#XB, #XU #XS, #XF	バッファ, ファイルの定義状態の表示
BB, BD, BC BF	バッファの定義, バッファとファイルの解放 ファイルの定義
R, W	バッファ間またはバッファとファイルの間でのテキストの読み/書き

3.3 段階的詳細化を助ける機能

前述したように段階的詳細化の過程とは抽象のレベルに対応して層を次々と構成していく過程である。さらに, ある層を構成する任意個の機能単位は, それぞれ別々に任意の順序で詳細化されるべきである。

よって, 階層記述言語による層の記述は図2に示し

表 3 階層構造操作のコマンド一覧表
Table 3 The table of commands for manipulating hierarchical structure.

	コ マ ン ド	機 能
(1)	@S (機能単位名)	()内に示された機能単位を詳細化することを宣言し、そのために必要なバッファを開設する。
(2)	@T (機能単位名)	(1)と同様。ただし、()内に示された機能単位は、対象言語のステートメント列に詳細化されることを指示する。
(3)	@C $\left\{ \begin{smallmatrix} S \\ T \end{smallmatrix} \right\}$	(1), (2)で開設されたバッファに対して、内容はそのままタイプだけを変更する。@CS(@CT)の時、カレントバッファのタイプを@S(@T)で開設されるバッファのタイプに変更する。
(4)	@D (機能単位名)	()内に示された機能単位以下のレベルの詳細化を無効にし、使用されていたバッファを解放する。
(5)	@R [(機能単位名)]	機能単位名のクロスリファレンスを表示する。()内に機能単位が指定されている時は、それに関する部分のみを表示する。
(6)	@P (出力バッファ名)	定義されている階層構造の概観を()内に示されたバッファに出力する。
(7)	@E (出力バッファ名)	定義されている階層構造に従ってモジュールを組み立てて、()内に示されたバッファに出力する。同時に文書情報をコメントとしてソースコード中にうめこむ。
(8)	#X@	階層構造定義用に使用されているバッファに関する情報(バッファ名、タイプなどのリスト)を表示する。
(9)	&S (出力バッファ名)	エディタ内に展開されている階層構造の定義を()内に示されたバッファに退避する。
(10)	&R	カレントバッファの内容を(9)のコマンドで退避したデータとみなして、元の状態を復旧する。

たようにエディタ内の複数のバッファを用いて行うのが自然な方法といえる。この方法を実現する諸機能について以下に述べる。

- (1) 各バッファを階層記述用に確保し、階層記述言語を用いてある抽象レベルの階層の記述を可能にする。また、各バッファには上位レベルの階層に現れた機能単位名と同じ名前をつけて各階層間の関連を保持する機能。
- (2) 段階的詳細化を行っているモジュール作成者に必要な情報を提供する機能。たとえば階層間の関連や詳細化の進行状況など。
- (3) 各バッファの内容を入力として、定義によってモジュールを構築するトランスレータの機能。

以上の機能のために表3に示すようなコマンドを用意した。

本エディタは複数のバッファを用いているが、それらは三つのタイプに分類される。第一は、表3の(1)のコマンドで開設されるもので、階層記述言語を記述する。第二は、表3の(2)のコマンドで開設されるもので、対象言語のステートメントを記述する。これら二つのタイプのバッファによって階層構造を定義する。第三のタイプは、表3の(6), (7), (9)のコマンドで用いられるもので、通常のテキストエディタのバッファと同様のものである。また、これらすべてのバッファのうち一つをカレントバッファと呼び、そのバッファの内容が端末に表示され、編集コマンドの対象となる。

ここで第一、第二のタイプを区別するのは、その処理内容が大幅に違うからである。第一のタイプのバッファの内容はシステムによって解析され、種々の情報が収集されてモジュール構築の骨組みとなる。一方、第二のタイプのバッファの内容は、モジュール構築の時にそのままきあがるモジュールにコピーされるだけである。また、この区別を利用者に指定させるのは、バッファの内容から判断する場合に生ずる効率低下と、対象言語（現在は PL/I）を他の言語にも拡張する場合、新たに作成して追加する言語依存の処理モジュールの量を少なくするためである。ただし、いったんいずれかのバッファを用いて記述を進行させている場合に、バッファのタイプを変更したくなる時がある。この時、表3の(3)のコマンドを用いてバッファのタイプを変更できる。

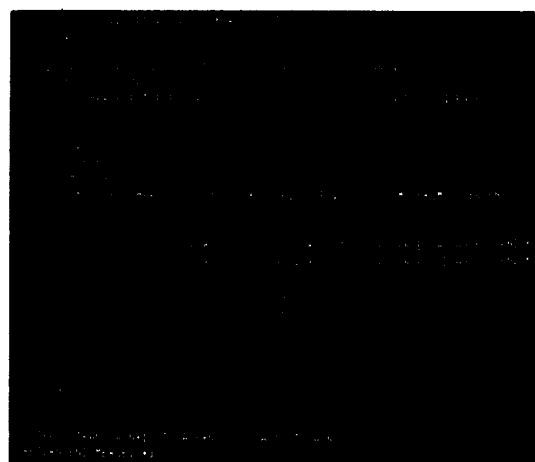


図 3 @P コマンドによる表示例—階層構造の概観—

Fig. 3 An example displayed by @P command.
—Hierarchy summary—.

```

*2 MERGE 2 TUPLES FROM I- AND J-SEQUENCE INTO K- AND L-SEQUENCE *
*3 *
*4 *
*5 *
*6 *
*7 *
*8 *
*9 *
*10 *
*11 *
*12 *
*13 *
*14 *
*15 *
*16 *
*17 *
*18 *
*19 *
*20 *
*21 *
*22 *
*23 *
*24 *
*25 *
*26 *
*27 *
*28 *
*29 *
*30 *
*31 *
*32 *
*33 *
*34 *
*35 *
*36 *
*37 *
*38 *
*39 *
*40 *
*41 *
*42 *
*43 *
*44 *
*45 *
*46 *
*47 *
*48 *
*49 *
*50 *
*51 *
*52 *
*53 *
*54 *
*55 *
*56 *
*57 *
*58 *
*59 *
*60 *
*61 *
*62 *
*63 *
*64 *
*65 *
*66 *
*67 *
*68 *
*69 *
*70 *
*71 *
*72 *
*73 *
*74 *
*75 *
*76 *
*77 *
*78 *
*79 *
*80 *
*81 *
*82 *
*83 *
*84 *
*85 *
*86 *
*87 *
*88 *
*89 *
*90 *
*91 *
*92 *
*93 *
*94 *
*95 *
*96 *
*97 *
*98 *
*99 *
*100 *

```

(中 略)

```

IF /A2 *
  LP = 1
  THEN
    *1 *
    *2 *
    *3 *
    *4 *
    *5 *
    *6 *
    *7 *
    *8 *
    *9 *
    *10 *
    *11 *
    *12 *
    *13 *
    *14 *
    *15 *
    *16 *
    *17 *
    *18 *
    *19 *
    *20 *
    *21 *
    *22 *
    *23 *
    *24 *
    *25 *
    *26 *
    *27 *
    *28 *
    *29 *
    *30 *
    *31 *
    *32 *
    *33 *
    *34 *
    *35 *
    *36 *
    *37 *
    *38 *
    *39 *
    *40 *
    *41 *
    *42 *
    *43 *
    *44 *
    *45 *
    *46 *
    *47 *
    *48 *
    *49 *
    *50 *
    *51 *
    *52 *
    *53 *
    *54 *
    *55 *
    *56 *
    *57 *
    *58 *
    *59 *
    *60 *
    *61 *
    *62 *
    *63 *
    *64 *
    *65 *
    *66 *
    *67 *
    *68 *
    *69 *
    *70 *
    *71 *
    *72 *
    *73 *
    *74 *
    *75 *
    *76 *
    *77 *
    *78 *
    *79 *
    *80 *
    *81 *
    *82 *
    *83 *
    *84 *
    *85 *
    *86 *
    *87 *
    *88 *
    *89 *
    *90 *
    *91 *
    *92 *
    *93 *
    *94 *
    *95 *
    *96 *
    *97 *
    *98 *
    *99 *
    *100 *
  END
END MERGE

```

図 4 @E コマンドによる表示例—モジュールの構築—

Fig. 4 An example displayed by @E command.
—Composition of module—.

また、表 3 の (9), (10) のコマンドは中断したエディティング・セッションを再開するために用いる。表 3 のコマンドのうち (6), (7) による表示例を図 3, 図 4 に示す。ただし、これらの図は複数画面にわたる表示をつないだものである。

4. 文書情報収集提示機能

モジュールの理解にとって効果的な方法はそのモジュールが作成された過程が説明されることである。本システムはモジュール作成過程と同期して文書情報を収集し、作成されるモジュールにコメントとして保存する。提示はそのモジュールのソースコードから文書情報を抽出して行われる。また、提示される文書情報の質を高めるための補完機能を有している。以下にその各機能を説明する。

4.1 文書情報の自動収集機能

図 2 からわかるように、モジュールの上位レベルの構造がエディタ内の複数のバッファによって保持されている。この情報は作成されるモジュールの論理構造に関する情報として重要である。また、各機能単位に関する説明は表 1 に示した言語によって入力済みである。これらの情報を整理してコメントの形式でソースコード中に埋めこむことによって、文書情報の自動収集ならびにソースコードと文書との一体化を実現する。

また、付加されるコメントには詳細化のレベルを表す番号が最上位のレベルを“1”として、詳細化が進むにつれて“2”, “3”, … というように付けられている。これをレベル番号と呼ぶ (図 4)。

4.2 内部論理構造の提示機能

本システムの提示方式をレベル・チャートと呼ぶ。前述した諸機能によって得られたソースコードにはレベル番号を伴ったコメントが埋め込まれている。モジュールを一つのブラックボックスと考えた時の機能を説明したコメントにはレベル番号 1 が付いている。そのブラックボックスが複数のサブブラックボックスに分けられている時、アルゴリズムの詳細化が一段階進んだと考えられ、それぞれの機能を説明したコメントにはレベル番号 2 が付いている。以下同様に 3, 4, … というように付いている。このレベル番号とそのレベルで実現している機能を説明したコメントをレベル番号に応じて行さげを行って表したものがレベル・チャートである。

レベル・チャートにおいては表示するレベルの最大値を指定することができる。このことによってモジュール作成者の段階的詳細化の過程をたどることができる。つまり、まず第 1 レベルまで表示すればモジュール全体としての機能が得られる。次に第 2 レベルまで表示すれば、1 段階詳細化されたレベルでのモジュールの論理構造を得ることができる。以下同様に順次見ていけば、そのモジュールがどのような詳細化の過程を経て作られたかを知ることができる。

一方、それぞれのレベルにおけるモジュールの論理構造に関する情報は、以下のようにして表す。レベル・チャート作成にあたってシステムは自動的に、if 文、while 文の条件に関する説明、then 節、else 節に関する説明に、それぞれ「IF」, 「WHILE」, 「THEN」, 「ELSE」を、表示の際に補う。その際、挿入したキーワードは区別しやすいように他の説明文とは色を変え

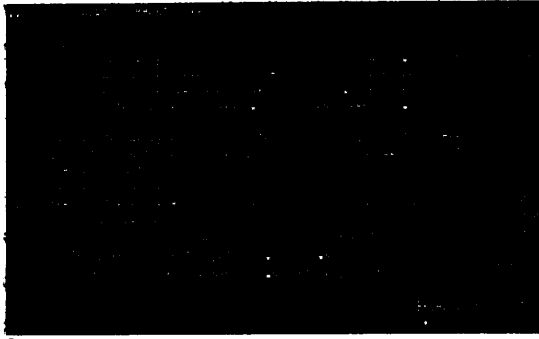


図 5 レベル・チャート (レベル 2)
Fig. 5 Level chart (level 2).

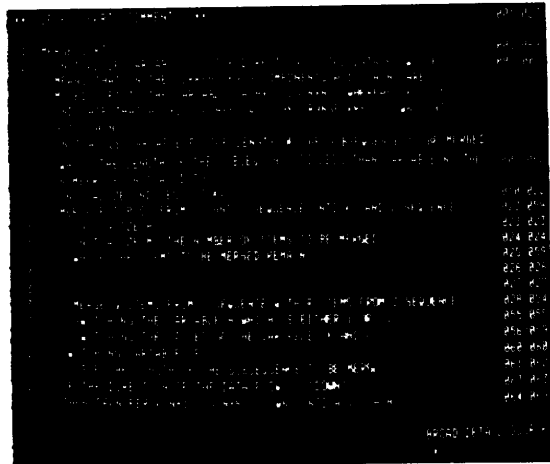


図 6 レベル・チャート (レベル 3)
Fig. 6 Level chart (level 3).

て表示する。図 5 に 2 レベルまで、図 6 に 3 レベルまで表示したレベル・チャートを示す*。

また、ソースコードと文書が別々の媒体上にある場合、その対応をとることがむずかしいが、レベル・チャートにはレベル番号やコメントとともに、そのコメントに対応するソースコードのステートメント番号の上限と下限が付加されている (図 5、図 6 の右端の 3 桁の数字の組)。これによってコメントとソースコードの対応をつけ、自動的に対応するステートメント列を切り出して表示することが可能である (図 7)。ただし、while 文の条件に関するコメントに付加されている数字の意味は上述したものとは違って、while 文の繰り返しの範囲を示している。以上述べた一連の操作を行うコマンドを表 4 に示した。

一般に、レベル・チャートのような文章風の記述で内部論理を説明する方法は、とくに詳細なレベルまで

* ただし、これらのレベル・チャートは 4.3 節で述べる機能によって、説明文が補完、修正されている。

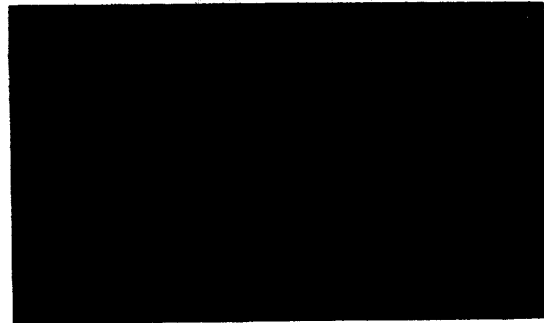


図 7 説明文に対応するソースコードの切り出し表示
Fig. 7 A display of a part of the source code corresponding with the comment.

表 4 内部論理構造提示用のコマンド一覧表
Table 4 The table of commands for displaying structure of internal logic.

コマンドの形式	コ マ ン ド	機 能
キーボード入力	LC $\left\{ \begin{matrix} 1 \\ \vdots \\ n \end{matrix} \right\}$	レベル n までのレベル・チャートを表示する。
ファンクション・キー入力		提示画面のロールアップ/ダウン
ライトペン入力	レベル・チャート中のある行を押える	説明文に対応するソースコードを参照する場合に対象となる説明文を指定する。
	画面右下の 'SOURCE' を押える	指定されている説明文に対応するソースコードを表示する。
	画面右下の 'COMMENT' を押える	提示画面をレベル・チャートの表示にもどす。
	画面右下の 'BROAD' を押える	現在表示中のレベル・チャートで、表示されるレベルを 1 段下げる (おおまかにする)。
	画面右下の 'DETAIL' を押える	現在表示中のレベル・チャートで、表示されるレベルを 1 段上げる (詳細にする)。

表示した場合に、レベル・チャートだけからモジュールの構造に関する情報を得ることは、他の図式的な方法に比べて、視覚的に訴えるという点において不十分であることは否めない。レベル・チャートは、モジュールの設計時の詳細化の過程を理解できる点、テキストエディタ風の操作でその修正、補完が容易にできる点、端末の画面で利用するための所用スペースの少なさなどに特長がある。

4.3 文書情報の補完機能

提示機能においてモジュール利用者が必要十分な情報を得ることを保証するためには、それだけの情報が収集されてソースコード中に埋めこまれていなくては

表 5 文書情報補完用のコマンド一覧表

Table 5 The table of commands for supplying document information.

コマンドの形式	コ マ ン ド	機 能
キーボード入力	P S	補完の省略
	C	文書情報の変更
	A	文書情報の追加
	D	文書情報の消去
	OK	補完・修正の完了
ライトペン入力	レベル・チャート 中のある行を押え る	補完, 修正の対象となる部 分を指定する

ならない。これはモジュール作成者が階層記述言語でモジュールを設計する際に、機能単位に関する説明をいかに適切に行うかに大きく依存する。しかし、モジュール作成者がその作成作業中にこれらのことに大きな注意を払わなければならないとしたら、それは新たな阻害要因となる。よって、本システムはモジュール作成後、作成者に対して補完機能を提供する。

モジュール作成者はモジュール完成後のレビュー時に、4.2 節で述べた提示機能を利用して自分の行った段階的詳細化の過程が正しく再現されているかどうかを確認する。必要ならば表 5 に示したコマンドを用いてレベル・チャートを修正する。修正は適切なレベルまで表示したレベル・チャートの修正すべき行をライトペンで指示してポインタを移し、テキストエディタ風の操作で行う。その際、そのコメントに対応するソースコードを切り出して表示する機能を利用できる。

たとえば、階層記述言語の〈target-language-description〉(付表 1) で記述されたステートメント(列)に対しては、図 4 に見られるように「?」を内容としたコメント文が付けられる。これに対応するレベル・チャートは「……」が表示される(図 6)。この表示に対して説明を付加した方がよいと判断したならば表 5 のコマンドで補完し、必要なしと判断したならば、同じく表 5 のコマンドの「PS」で補完を省略すればよい。もし、補完されたならば、その情報はソースコード中の「?」と置き換えられる。このように本機能による修正結果は元のソースコードに反映され、対応するコメントの修正、追加が自動的に行われる。図 5、図 6 のレベル・チャートは、このような補完を受けた結果である。

5. 例題と評価

N 個(簡単化のために N は 2 のべき乗)の整数要素 $A(1), A(2), \dots, A(N)$ のシーケンスをマージソ-

ートの手法を用いてソートするモジュールを例に挙げる。このモジュールを PL/I で作ることにし、階層記述言語による定義例は図 2、各種コマンドによる提示は図 3～図 7 に、すでに示した。

本システムは中型電子計算機システム (FACOM-230-45 S) 上で PL/I (14K ステップ) とアセンブリ言語 (2K ステップ) で実現されている。端末はカラーキャラクタディスプレイ装置 (赤白緑の 3 色, 80 字×24 行) を使用し、レベル・チャートは利用者のコマンドでラインプリンタにも出力される。

会話型システムでは応答時間の短さが使いやすさに大きく影響する。本システムのほとんどのコマンドに対する応答時間は 1 秒前後である。ただし表 3 の (6), (7) のコマンドは本節の例題程度のモジュールの場合で 3～5 秒程度かかる。これらの応答時間は実用上十分だといえる。

本システムが扱えるモジュールの大きさは、システム内のバッファの数および収容可能なテキスト行数に依存する。これらのパラメータは本システムの生成時に与えることが可能で、現在バッファ数 26、テキスト行数 1,000 行に設定してある。

我々の経験では、本システムで生成されるモジュールは 100 ステップ前後で、その際必要なバッファは 10～15 個、テキスト行数は 300 行前後である。

また、その 100 ステップ前後のモジュールのソースコード中でコメント行の占める割合は 30～40% であり、適度な量といえる。さらに、これらのコメントは本システムの提示機能を用いなくて直接ソースコードをながめても理解しやすいように配置されている。

6. おわりに

本システムの特長は、(1) 段階的詳細化法による整構造モジュールの作成過程の阻害要因を除去できた。(2) その過程で行われたモジュール作成者の設計上の意思決定情報を収集し、それを再現するような表示方法を採用したことによって、そのモジュールの論理構造理解時の立ち上がり時間を大幅に短縮することができた。(3) レベル・チャートはモジュールのソースコードから自動作成され、逆にレベル・チャートの修正をモジュール中のコメントに反映させることにより、文書とソースコードを一体化して管理できる、点にある。

本システムによる文書化はソースコードだけ得ればよいというモジュール作成者にとっては負担の増加に

なる。しかし、ソフトウェアを資源として有効利用していく上で、その機能、構造などを第三者に伝えるための高品質の文書は不可欠であり、この種の負担は必ず必要なものである。本システムはモジュール作成後文書類を作成するという労力の大幅な減少と、高品質の文書の作成のために、モジュール作成時に若干の負担増を課したものである。

しかし、文書情報の質をあるレベル以上に保つことに関しては客観的な尺度、方法は存在せず、モジュール作成者の能力に依存している。この問題の解決法の一つとして、使用経験を積むことによって表1に示した言語で階層構造を定着する時、どのような説明を与えたらよいかについて、一定の規準を設ける方法が考えられる。

現在、本論文で述べた方法を、モジュール作成局面だけでなくソフトウェアライフサイクルのさらに先立つ局面にも応用したシステムを実現しており、これについては別の機会に述べたい。

謝辞 本システムの開発を手伝っていただき、有益な討論をしていただいた岡本勝男氏（現在（株）ソニー）、八木文治氏（現在（株）日本電気）に謝意を表します。

参 考 文 献

- 1) 落水, 酒井, 八木, 岡本: アルゴリズムレベルのプログラム文書自動化ツール, 情報処理学会ソフトウェア工学研究会資料 11-1(1979).
- 2) 落水: ソフトウェア開発・保守支援システムPDB, 情報処理学会ソフトウェア工学研究会資料 17-5(1981).
- 3) MULTICS PROGRAMMERS' MANUAL QEDX.
- 4) Dahl, O.J., Dijkstra, E.W. and Hoare, C. A.R.: *Structured Programming*, Academic Press, London (1972).
- 5) 二村, 川合, 堀越, 堤: PAD(Problem Analysis Diagram) によるプログラムの設計および作成, 情報処理学会論文誌, Vol. 21, No. 4, pp. 259-267 (1980).

- 6) Van Leer, P.: Top-down Development Using a Program Design Language, *IBM Syst. J.*, Vol. 15, No. 2, pp. 155-170 (1976).

(昭和 56 年 6 月 25 日受付)

(昭和 57 年 3 月 18 日採録)

付表 1 階層記述言語の構文規則

A. Table 1 Syntax of the hierarchy description language.

```

<module> ::= <entry-st> <dcl-st>* <statement>* <return-st> |
            <entry-st> <dcl-st>* <module>* <return-st>

<statement> ::= <simple-st> | <complex-st>

<simple-st> ::= <block-st> | <if-st> | <while-st> |
              <target-language-description>

<complex-st> ::= DO [<repeat-control>]; <statement>* END;

<dcl-st> ::= DCL (<l-bracket> <functional-unit-name> : <explanation>
              <r-bracket>);

<block-st> ::= BLOCK (<l-bracket> <functional-unit-name> :
                     <explanation> <r-bracket>);

<entry-st> ::= ENTRY (<l-bracket> <entry-statement> :
                     <explanation> <r-bracket>);

<return-st> ::= RETURN (<l-bracket> <return-statement> :
                       <explanation> <r-bracket>);

<while-st> ::= DO WHILE <condition>; <statement>* END;
<if-st> ::= IF <condition> THEN <statement>
           [ ELSE <statement> ]

<repeat-control> ::= <cond-st> | <target-language-description>

<condition> ::= <cond-st> | <target-language-description>

<cond-st> ::= COND (<l-bracket> <functional-unit-name> :
                   <explanation> <r-bracket>)

<target-language-description> ::= (<l-bracket> <target-language>
                                   <r-bracket>)

<l-bracket> ::= [
<r-bracket> ::= ]

```

注 1) 以下は終端記号である。

<functional-unit-name> 8文字以内の英数字列（.（ドット）と _（アンダバー）を含む）。

<target-language> 対象言語（たとえば PL/I）のステートメント列。

<entry-statement> 対象言語のプロセッサ文、または、エントリ文。

<return-statement> 対象言語のエンド文、または、リターン文。

<explanation> 自然語による説明文。

注 2) [] は選択的な構文。

< >* は 0 回以上の繰り返し。

< >+ は 1 回以上の繰り返し。