15. ソフトウェアマネージメントシステム：FAST1

青山学院大学　理工学部経営工学科
井田昌之，木村公則

1．まえがき

我々は富士銀行業務管理部より依頼を受け、かつ全面的な協力のもとにＦＡＳＴ１（Fuji-Aoyama Structural Translator 1）を作成した。開発は昭和４７年７月より当学科のＩＢＭ－１８００（1w=16bit, 24 kw）を用いて始められ，現在ＵＮＩＶＡＣ－１１０８（1w=36bit, ２５６kw）で約１００名のプログラマを対象にリリースされている。

ＦＡＳＴ１は「プログラミング負担の軽減」と「ソフトウェアの全体的管理」を主眼としている。機能的には生成フレームをシステムから切り離したジェネレータという形態をとっており，その柔軟性を保ちつつ所要マンパワーの分散を計っている。

2．プログラミング負担の軽減と作成されたプログラムの維持

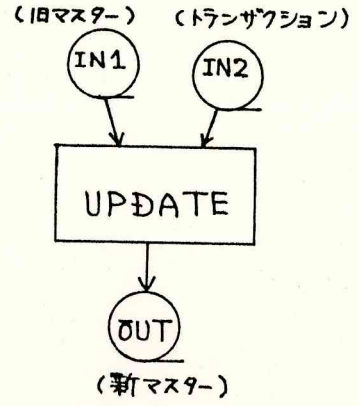
一般にプログラムの構造化の要件としてトップダウン概念とＧＯ ＴＯ ＦＲＥＥ概念があげられるが前者は巨視的構造化，後者は微視的構造化と考えることができる。事務処理プログラムの場合，作成されたプログラムは反復して使用される点に特長があり、プログラマはプログラムの新規作成だけでなく既存のプログラムの修正、保守に大きく力をさかれている。巨視的構造化はprogramming efficiencyに有効であり、微視的構造化はprogram maintainabilityに有効であると考えられるが、ＦＡＳＴ１ではトップダウン概念の実現を目指しており、後者に対する具体的なエイドは持っていない。しかし、トップダウンの結果により、既存の言語と比較してプログラムの見やすさがあがる事が後述される。

概念的な比較を行うために図１を示す。この図は口座番号順に昇順に並べられたマスターファイルＩＮ１に対してトランザクションＩＮ２を用いて更新作業を行なうプログラムＵＰＤＡＴＥを示したものである。単純に書かれたＦＯＲＴＲＡＮプログラム（図１（a））はファイル処理

構造的プログラミングとその経験　シンポジウム
1975年 7月22～24日

```
@FOR,IS UPDATE,UPDATE
      LOGICAL IN1REQ,IN2REQ,OUTREQ
      IN1REQ=.TRUE.
      IN2REQ=.TRUE.
      CALL OPEN(IN1,IN2,OUT)
      .
      .
C     MAIN LOOP
1000  IF(IN1REQ) CALL GET(IN1)
      . . .
      IF(IN2REQ) CALL GET(IN2)
      . . .
      IF(K00ZA1.GT.K00ZA2) GO TO 1
      IF(K00ZA1.LT.K00ZA2) GO TO 100
C     UPDATE
      . . .
      . . .
100   CONTINUE
      . . .
      . . .
      . . .
      IF(OUTREQ) CALL PUT(OUT)
      . . .
      GO TO 1000
C     CLOSING
9999  CALL CLOSE(IN1,IN2,OUT)
      . . .
      STOP
      END
```

(a) Using FORTRAN




```
$GEN  UPDATE
'PATTERN PROG2T01
      KEY=K00ZA
*PROC1 /* IN1 = IN2 */
      . . .
      . . .
*PROC2 /* IN1 < IN2 */
      . . .
      . . .
'END
```

(c) Using FAST1

```
PROCEDURE DIVISION.
MAIN.
      PERFORM INITIALIZE THRU INIT-X.
      PERFORM DRIVER THRU DRIVER-X.
      PERFORM END-JOB THRU END-JOB-X.
      STOP RUN.
      . . .
DRIVER.
      PERFORM EOF-CHK THRU EOF-CHK-X.
      PERFORM ENDRIV THRU ENDRIV-X.
DRIVER-X.   EXIT.
      . . .
      . . .
```

(b) Using COBOL

図1. Comparison of coding steps for the program UPDATE.

を特長づけている入出力ループに大きなウェイトがおかれてしまう。モジュール化を意識したCOBOLプログラム(図1(b))では、その構造性を保つために余分なコーディングが必要となっている。

FAST1では事務処理プログラムを入出力ファイルの制御を主とするドライバと各レコードに対する個有の処理とにより構成されると考え前者を通常のプログラマから解放しこれを『ロジックユニット』と名づけ独立して管理する。(富士銀行では約80%のプログラムが6つのロジックユニットに帰結できる。)FAST1をインプリメントしている組

織では，その組織個有の処理形態および一般的なループ制御構造をまとめロジックユニットを作成し，その仕様書を作成する。ロジックユニットの保守はユーザの責任であり，その修正はＦＡＳＴ１プロセッサには何の影響も与えない。更にデータ構造は『レイアウト』と呼ばれ，一括管理されており簡単なシート記入により登録修正を行なう。各プログラマは必要とするロジックユニットとレイアウトの引用，更にそのプログラムの"性質"を形成する各レコードに対する個有処理の記述をロジックユニット仕様書に基づいて行なうだけでコーディングが完成する。

図１(c)の例ではロジックユニット ＰＲＯＣ２ＴＯ１の選定．処理キーの設定．そしてトランザクションが存在した時の処理（＊ＰＲＯＣ１）とそれ以外の場合（＊ＰＲＯＣ２）だけをコーディングする。

３．トップダウンプログラミングとマンパワーの分散

ＦＡＳＴ１によるプログラム開発の過程は図２(b)のようになる。これはトップダウンの手順（図２(a)）を直接行なうことに対応する。この考え方は全過程を１人で行なうことのない実際の組織体に合致する。プログラミング部門の構成員はロジックユニット及び全体の管理を行なうＦＡＳＴ１スタッフ，プランナ，レイアウトの管理を行なうファイルマネジャ，プログラマの４種にわけて考える事ができるが図２に示すようにプランナはロジックユニットの選択を通して明示的にプログラム作成の一機能を分担する。この後プログラマはロジックユニットの持つ機能と与えられた仕様をつきあわせ必要なパラメータを記述する。

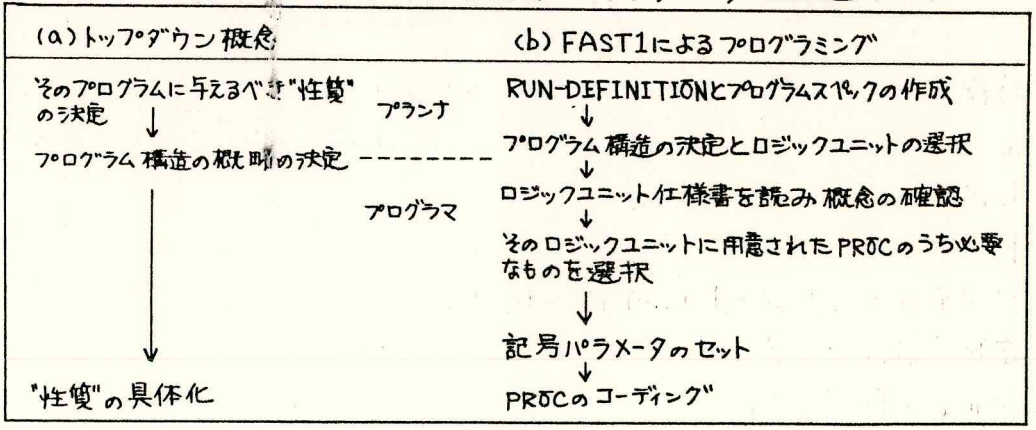


図2. Topdown Programming

構造的プログラミングとその経験 シンポジウム
1975年 7月22～24日

4. システム構成

ＦＡＳＴ１は図３に示すように３種類の独立した機能を持つルーチンと全体をコントロールするスーパーバイザより構成される。

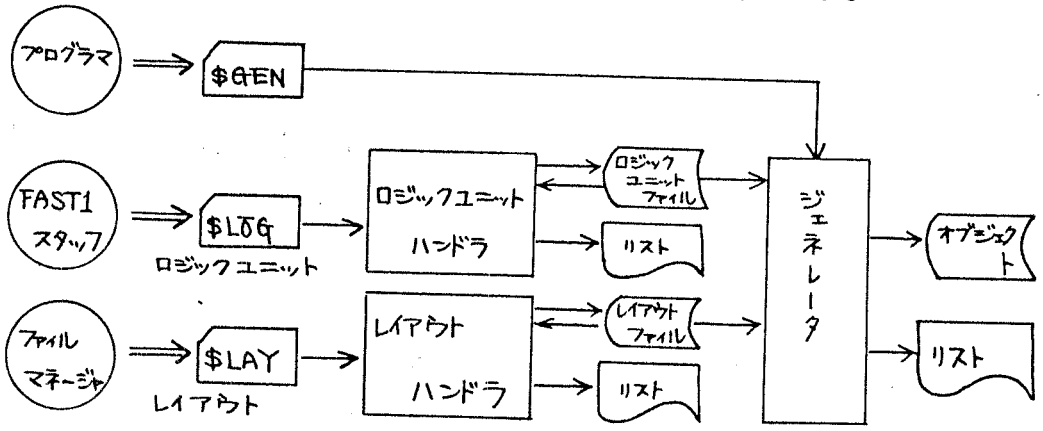


図３. Configulation of FAST1 system

5. ロジックユニット

図４に例示したようにロジックユニットはいわゆるマクロ定義に近い形式をとっている。生成にあたっては可変シンボルと９つの生成用カウンタの修飾と、カラム７に＊を持つ生成コントロールステートメントの制御を受ける。この生成コントロールステートメント中の最大の特徴は＊ＰＲＯＣステートメントであり、このステートメントによりロジックユニット中に未定義なコーディングを残すことができる。ジェネレータ入力中にＰＲＯＣ－パラメータが与えられると対応する＊ＰＲＯＣステートメントを置きかえてユニット中

```
'NAME=PROG1T01
'PARM &INPUT
'INPUT &10(&IN)
'OUTPUT &20(&OUT)
'ATEND &999(&IN)
'INITIAL
        *PREC 1
#1      *SETA 1
#2      *SETA 2
'MAIN
        *IF(&INPUT EQ 'CARD').CARD
&10     READ(#1,END=&999) &IN
        *GO.NEXT
.CARD   *NOP
        *IF(&INR LE 14).OK
        *ERR ILLEGAL REC.SIZE
.OK     *NOP
&10     READ(5,&100,END=&999) &IN
&100    FORMAT(13A6,A2)
.NEXT   *NOP
        *PROC1,1=&10,2=&20,3=&999;
&20     WRITE(#2) &OUT
        GO TO &10
'FINAL
&999    CONTINUE
        *PROC2;
        *END
'END
```

図４. ロジックユニットの例

に組み込まれる。この＊ＰＲＯＣステートメントを設けることにより前もって与えられた処理パターンの選択だけでなく，任意の処理をロジックユニット中に組み入れることができるようになる。

6．レイアウト

レイアウトの登録項目としては、階層レベル．項目名．長さ．属性．配列の大きさがある。またフィールドの再定義も可能である。

7．ジェネレータ入力

ジェネレータ入力は全体で3つのセクションより成り立っている。すなわち使用するロジックユニット等を定義するパターンセクション。プリント出力のためのプリントフォマットセクション（オプショナル）。ロジックユニットに対するキーワードパラメータの記述と＊ＰＲＯＣステートメントに対するＰＲＯＣ－パラメータの記述を行なう展開パラメータセクションである。各ＰＲＯＣ－パラメータの記述のためには１６種のステートメントが用意されており、データの詳細な扱いを可能としている。これらは入出力文・宣言文を含んでいないので教育が容易である点に特徴がある。図５にジェネレータ入力の概念的な例と．その結果生成されるＦＯＲＴＲＡＮプログラムの例を図６に示す。

```
$GEN CARDTP
'PATTERN PROG1TO1
        IN=(CARDF)
        OUT=OUTFIL(TAPEF)
'PARM
        INPUT=CARD
*PROC1
        IF KEY:IN NOT = 1,
                RETURN 1 ;
        MOVE ALL:IN TO ALL:OUT;
        RETURN 2 ;
        END PROC1;
'END
```

図５ サンプル入力

```
      DIMENSION LY0001(14),LY0002(14)
10    READ(5,100,END=999) LY0001
100   FORMAT(13A6,A2)
C *PROC1
      IF(LY0001(1)-'1') 1,2,1
1     GO TO 10
2     . . .
      . . .
20    WRITE(2) LY0002
      GO TO 10
999   CONTINUE
      STOP
      END
```

図６ 生成結果

8．生産性および実行時間の検討

ＦＡＳＴ１のプログラムの作成はジェネレータ入力のコーディングを

意味する。従って表1に示すようにコーディングステップ数はFORTRANで書かれたプログラムの約半分で済んでいる。また作成日数もFAST1で作成した方が少なくなっており，銀行側では所要工数は今までの30％と設定している。これは2章．3章で述べた2つの基本概念の実施効果と考えられる。

これらの概念を具体的に支援するものとして，生成されるプログラムには次の機能が付加されている。

i）データタイプの変換はFAST1実行時ルーチンが行なうが，それらは配列添字チェック機能等をもつのでロジックユニットが完璧ならば無関係なエリアをこわすことがない。

ii）実行時のエラーダイアグノーズは統一したメッセージが出力されエラータイプを容易に知ることができる。

iii）生成時オプションを"デバッグモード"にセットすることによりエラー時にジェネレータ入力中のどのステートメントに起因するエラーであるのかを印刷することができる。

これらの3点は逆に実行速度の低下や実行時のプログラムサイズを増加させる欠点を持っているがプログラム作成および保守上の長所を損じる程ではない。また今後の改良は充分可能である。

プログラム	言語	ステップ数	論理ステートメントの割合	フローチャート作成	コーディング	デバッグ	実行時間(秒)	メモリー数(Kwd)
A	FAST1	86	9.8%	5時	8時	5日	93	54
	FORTRAN	150	18.5	——	——	——	90	49
B	FAST1	89	8.9	1時	7時	20分	59.5	23
	FORTRAN	319	12.6	——	——	——	31.6	16
C	FAST1	105	13.9	4時	11時	20日	58.7	26
	FORTRAN	496	15.0	1日	6日	30日	37.4	17
D	FAST1	161	5.6	5時	10時	10日	115.0	25
	FORTRAN	444	11.1	3日	5日	20日	61.0	18
E	FAST1	——	——	5時	6時	10日(推)	——	21
	FORTRAN	——	——	1日	7日	14日	113.4	16

表1．プログラム作成日数の比較

管理面においては、生産性の向上による人員の削減が可能である他、ロジックユニットを統一管理しているため実行時のファイルアサインの手順等オペレーション形式の統一がされつつある。またレイアウトも同様に統一管理しているのでフィールド名の統一が行なわれ、いわゆるテクニカルタームとしての概念が明確化しつつある。

9．まとめ

前述の結果からFAST1の目的は一応達成されたと考えられる。しかし今後の使用状況などにより決定される未解決の点も残されており、次システムの検討項目として特に以下の点があげられている。

全体の流れを意識せずにプログラムを組めるようにジェネレータ入力の自己充足性を高める。生成の最適化フェイズを分離することにより、portability を高める。ロジックユニット定義に対するマクロライブラリを作成し独立して維持させる。レイアウトの機能を高め、アクセス方法の定義やテーブルの扱いを可能にする。トップダウンプログラミングに対する独自の教育体制を確立する必要がある。

10．謝辞

本システム作成の機会を与えて下さった富士銀行業務管理部部長代理白鳥秋彦氏に感謝いたします。また森建二、田又義朗両調査役、並びにFAST1スタッフの皆様、初期故障に対して不平も言わず使用して下さった開発二係の皆様、約300回のデバッグを我慢強く流して下さった電子計算課の皆様、さまざまな御指導をいただいた経営工学科、間野浩太郎教授、横山巽子講師にあわせて深謝いたします。約1万ステップにわたるプログラムの作成にあたっては井田、木村両名の他に、岡田一、井田いく子、森銅真喜子諸氏の協力をおおいだ。

［討　論］

佐藤　SP とは直接関係ない質問だが、特に RPG を利用しなかった理由はどこにあるのか？（編註　RPG：Report Program Generator）

井田　(1) シート記入は、いわゆる普通のプログラマにとってあまり馴染みがないこと、および (2) 各カラムに意味を持たせたりするのは非常にわかりにくいことからみて、ファイル処理全体をカバーするのには、シート記入などの方法ではうまくいかないだろうという感じがした。

佐藤　すると感じだけで決めたわけか？　いまの話からすると、“生産性向上のために RPG が有効であろう”ということは十分に検討に値すると思えるがどうか？　複雑なチェックなどは RPG では、やりにくいのかもしれないが、少なくともマスターファイルの更新ないしはファイルの作成や update といったたぐいについては非常に有効ではないかという気がするのだが…。

井田　実際問題として富士銀行の場合、店の数はたくさんあるし、ファイルの数も山ほどになって、その間のチェックを個々別々にしなければならない。　するとどうも個別の処理ということを記述していくときに、もう少し言語といったものを導入して、自分であとから保守をしやすいようにしておいた方が良いのではないか、と考えた。

目　　次

°印は講演者

構造的プログラミングとその経験

シンポジウム報告集

1975. 7. 22～24

情　報　処　理　学　会
プログラミングシンポジウム委員会