U.D.C. 681.32.06.002:[658.52.011.56:681.322]

業務アプリケーション向け構造化プログラミングパッケージ“UDDT”

Structured Programming Package for Business Applications “UDDT”

松本良治* Yoshiharu Matsumoto
山野紘一** Kouichi Yamano
谷　昌之* Masayuki Tani

構造化プログラミングは，ソフトウェアの品質及び生産性を向上する技法として，有効であることが確認されている。しかし，汎用言語COBOLを使って構造化プログラミングを適用しようとすると，プログラム構造及び言語体系上，幾つか難点がある。

そこで日立ソフトウェアエンジニアリング株式会社では，日立製作所の協力を得て，COBOL言語を用いて構造化プログラミングを容易に実現でき，かつプログラムのドキュメンテーションの自動化を図るためのツール，UDDTを開発した。

UDDTを適用した場合の事例評価で，従来に比べ生産性で約20%，品質で約50%向上することを確認した。

1 緒　言

大規模ソフトウェアを開発するとき，プログラムの複雑さをいかに低減させるかということと，作成したプログラムの理解を助けるための記録をいかに残しておくかが重要な問題となってくる。前者に対しては，構造化プログラミング[1]，トップダウン設計[2]，複合設計[3]などの技法が提案され，その成果も報告されている。後者に対しては，HIPO[4]（Hierarchy and Input Process Output）図，構造設計書などがある。しかし，構造化プログラミング，トップダウン設計などの技法は，具体的手順の理解が難しく，実際面での適用が容易でない。また，ドキュメントに対しては，手書きのため，プログラムの修正が必ずしも反映されず，プログラムとドキュメントの不一致を生じている。日立ソフトウェアエンジニアリング株式会社では，これらの問題を解決するツールとして，UDDT[5),6)]（Unified Design and Documentation Tool）を開発した。

UDDTは，COBOL言語に対して構造化プログラミング用の構文を追加するとともに，ドキュメントの自動作成によるプログラムとドキュメントの一体化を図り，COBOLを使用した構造化プログラミングを容易に実現することによって，ソフトウェアの品質と生産性の向上をねらったものである。

なお，UDDTはVOS（Virtual Storage Operating System）2/VOS3のもとで使用することができ，昭和54年5月以降，一般ユーザーに提供している。

2 UDDTによる構造化プログラミング

2.1 UDDTのトップダウン設計

構造化プログラミングは，プログラムの作成を上位レベルから下位レベルへ段階的に機能の詳細化を行ない，その結果を記述していくことである。各段階では，できるだけ機能的に閉じるようにプログラム構造を設定する。その構造は，できるだけ分かりやすく，詳細化された機能間の論理の流れは，明確である必要がある。すなわち，人は一瞬のうちに，多くの細部まで理解し記述することはできないので，このような段階的な詳細化が必要である。そこで，UDDTでは，設計者の能力や経験に依存せず，機能の詳細化を分かりやすく進めることができるように，疑似文を導入した。また，ドキュメントには，直観的な分かりやすさを生かしたHIPO形式の図形表現を採用した。

機能の詳細化を実現するプログラム構造は，機能単位の構成ができることが必要である。COBOLのプログラム構造は，一つのプログラムが複数のモジュールの集合で構成されると

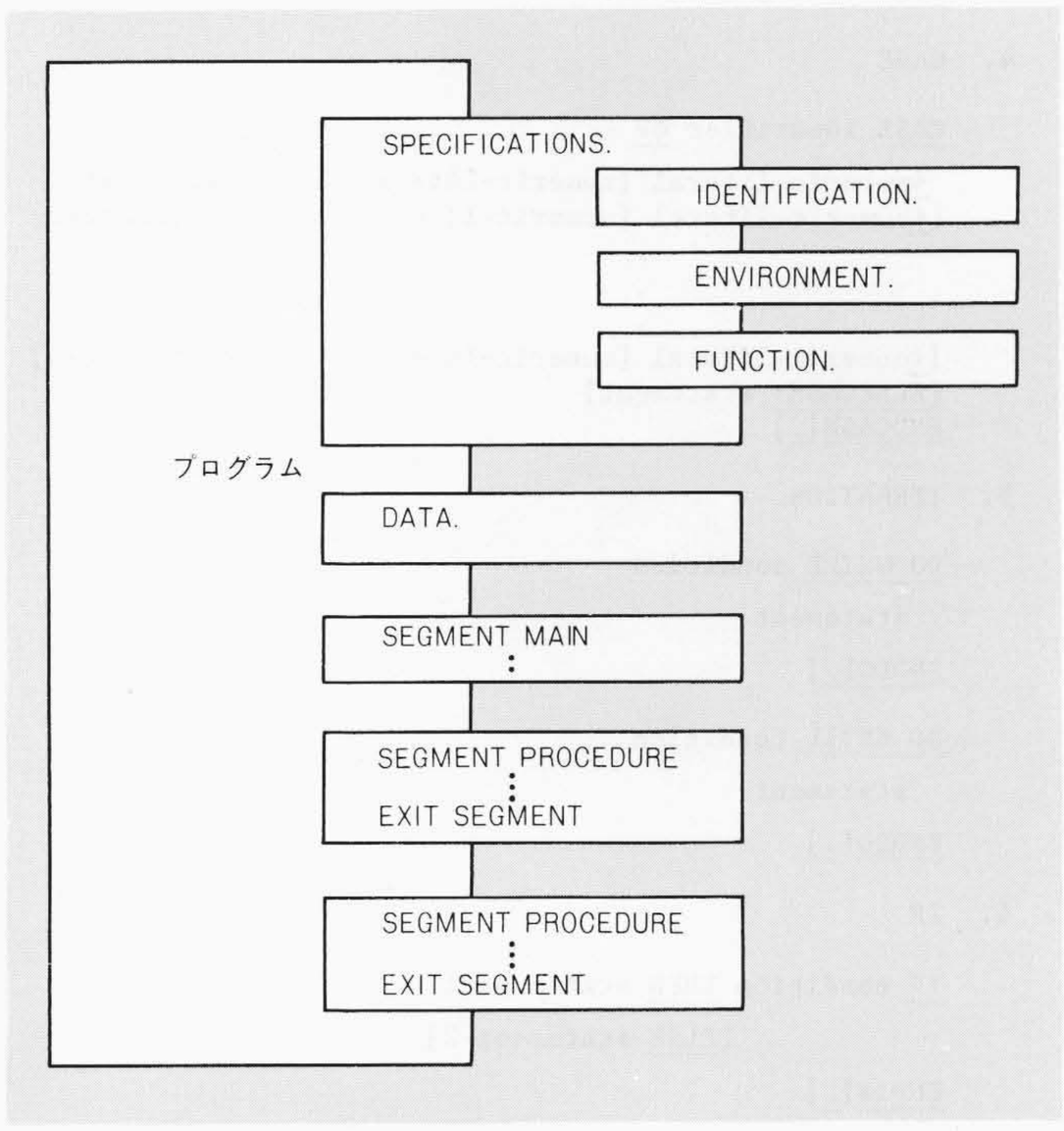


図1　UDDTによるプログラム構成　UDDT(Unified Design and Documentation Tool)を用いて作られるプログラムの構成を示している。

* 日立ソフトウェアエンジニアリング株式会社　** 日立製作所システム開発研究所

いう概念はなく，機能の詳細化を難しくしている。

UDDTでは，COBOLの構造に対して，論理的構成を実現するために，COBOLの最も基本的な単位であるセクションに着目し，セクションの概念に代わるものとして，セグメントという概念を導入した。セグメントは，UDDTのプログラム構成の単位であり，COBOL言語のセグメンテーション機能をも吸収している。UDDTでのプログラム構成を**図1**に示す。

2.2 言語機能

構造化プログラミングに適した言語は，ブロック構造的記述ができる言語である。そのような言語で記述するプログラムは，基本制御構造といわれる連続，繰返し，選択の制御構造をもち，制御の流れが連続したプログラムとなるため読みやすく，かつ品質の高いプログラムとなる。COBOLには言語体系上，このような構造を作成できる言語機能が少ないうえ，繰返し，条件及び分岐命令による論理構造の複雑さなどにより，制御の流れが不明確なプログラムになりやすいという問題がある。UDDTは，COBOLによる構造化プログラミングを実現するために，以下に述べる言語機能をCOBOLに付加した。その機能一覧を**図2**に示す。

```
1.  SEGMENT

    SEGMENT { MAIN
              SUB
              PROCEDURE [(DEBUG)]
              DECLARATIVE }  segment-name
            [USING parameter...].

2.  DENOTE

    statement-number. character-string.

3.  PERFORM

    PERFORM segment-name

4.  CASE

    CASE identifier OF
     <numeric-literal [numeric-literal]...>: statement-1
    [<numeric-literal [numeric-literal]...>: statement-2]
                        .
                        .
                        .
    [<numeric-literal [numeric-literal]...>: statement-n]
    [ELSECASE: statement]
    ENDCASE[.]

5.  ITERATION

    DO WHILE condition
      statement
    ENDDO[.]

    DO UNTIL condition
      statement
    ENDDO[.]

6.  IF

    IF condition THEN statement-1
                [ELSE statement-2]
    ENDIF[.]

7.  EXIT-SEGMENT

    EXIT SEGMENT [.]
```

図2 UDDTの言語機能一覧 COBOL言語で構造化プログラミングを実現可能とするために，UDDTが追加した言語機能を示している。

2.2.1 段階的記述機能

プログラムの作成で，その記述を段階的に詳細化する機能として，DENOTE及びSEGMENTを設けている。

(1) DENOTE

機能の詳細化の過程で，一時期詳細化ができない機能に対して，段階的に詳細化することを明示する文で，自然語で記述する。DENOTEの記述例を**図3**(a)に示す。番号と自然語で機能を記述したのが，DENOTEである。

(2) SEGMENT

SEGMENTは，機能の一つの単位である。DENOTEで記述した機能を詳細化する。SEGMENTの記述例を**図3**(b)に示す。**図3**(a)のDENOTEを詳細化している。

2.2.2 制御構造記述機能

プログラムの制御をより明確化する言語機能として，IF，DO，CASEの各文を設けた。

(1) IF文

COBOL言語のIF文とは異なり，入れ子構造が記述できるようにした。

(2) DO文

COBOL言語の繰返しは，対象となる文を別のセクション又は段落に記述していたが，制御の対象を列内に記述し，プログラムが分かりやすくなるようにした。

(3) CASE文

UDDTでは，多重分岐の制御の流れを明確にするためにCASE文を設けた。COBOL言語とCASE文による多重分岐の記述の比較例を**図4**に示す。

3 UDDTによるソフトウェア開発

UDDTによるプログラムの設計例を，**図3**(a)及び(b)に示す。

設計の開始は，DENOTEを用いて自然語で機能を記述する〔**図3**(a)〕。機能は，番号を付けて記述する。流れはプログラムの流れに沿っている。次に，DENOTEを別のSEGMENTとして記述する〔**図3**(b)〕。各DENOTEは，一つのSEGMENTを構成する。DENOTEの番号と，SEGMENTの番号を対応させる。**図3**(b)のSEGMENTは，同図(a)のDENOTEの1及び2に対応している。SEGMENTの記述は，DENOTEあるいはCOBOLコーディング，及びその混在で行なうことができる。機能の記述を詳細化しながら繰り返すと，構造化プログラミングによる設計は終了し，同時にコーディングも終了する。

ソフトウェア開発過程でのUDDTの位置を表わしたのが**図5**である。UDDTは，詳細設計以降，デバッグ完了まで適用できる。

4 出力ドキュメント

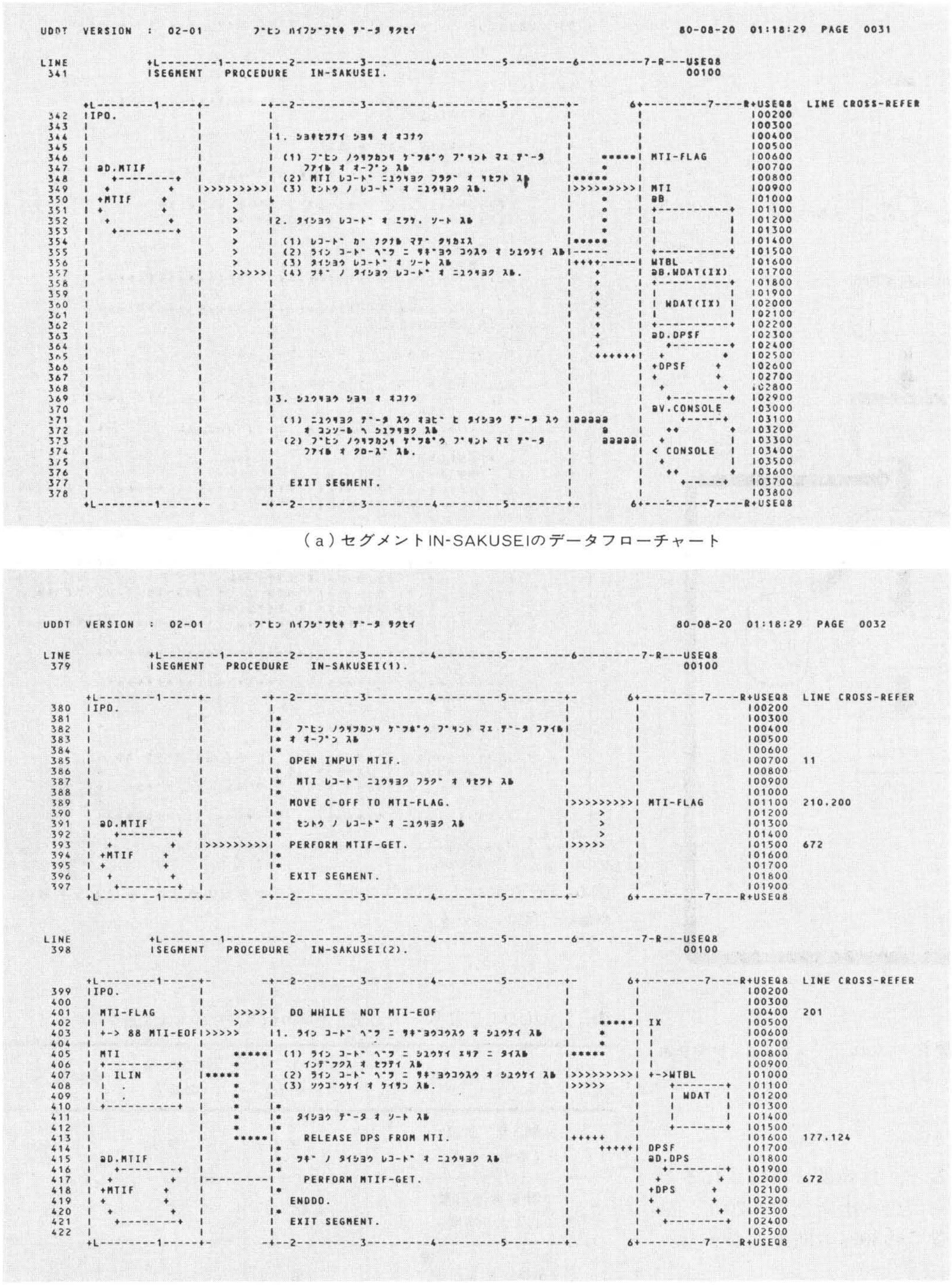
出力ドキュメントは，基本的に詳細設計で作成するドキュメントに沿っている。このドキュメントは，プログラムと対応しており，プログラムの修正などに有効である。UDDTの出力ドキュメントは11種類ある。代表的なドキュメントを以下に紹介する。

(1) セグメント関連リスト

プログラムの構造を表わしたドキュメントである(**図6**)。

(2) データフローチャート

機能の詳細化した結果を編集出力したドキュメントである〔**図3**(a)，(b)〕。



（a）セグメントIN-SAKUSEIのデータフローチャート

（b）セグメントIN-SAKUSEIで定義したDENOTE文 1， 2 を詳細化した例

図3　UDDTによるプログラムの設計例　　設計の開始は，自然語で機能の記述を行ない，次のステップで順次UDDT言語で記述する。

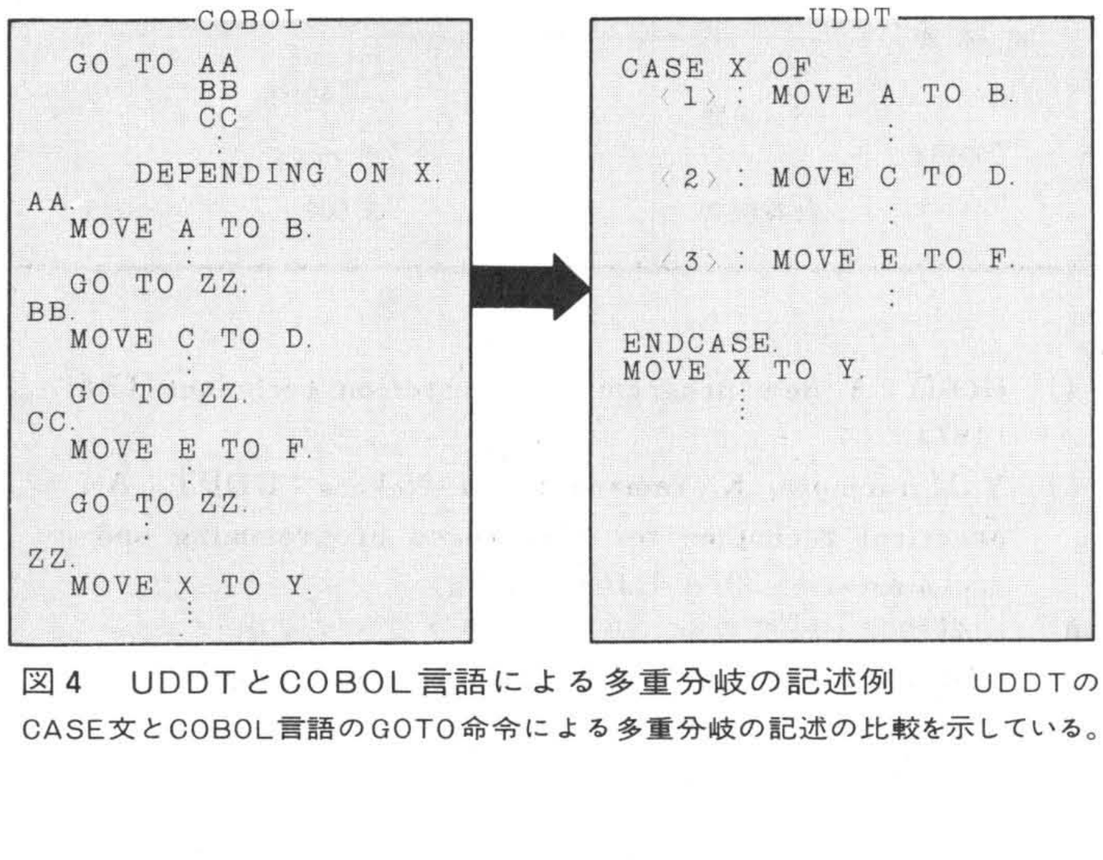


図4　UDDTとCOBOL言語による多重分岐の記述例　　UDDTのCASE文とCOBOL言語のGOTO命令による多重分岐の記述の比較を示している。

5 UDDTの効果

UDDTの効果として，以下のことが期待できる。

(1) COBOL言語での構造化プログラミングの実現を容易に行なえ，プログラム構造が明解な品質の良いプログラムが作成できる。

(2) 手書きで作成していたドキュメントが計算機で自動的に作成できる。ドキュメントの保守が不必要となり，生産性が向上できる。

実際に適用した事例の結果について述べる。**表1**は，約7kステップのプログラムを，UDDTとCOBOLで作成した結果の比較である。開発工数，計算機使用量及び品質が従来に比べ向上しており，構造化プログラミングの効果が表われているといえる。費用はカードパンチ費が増加しているが，詳細

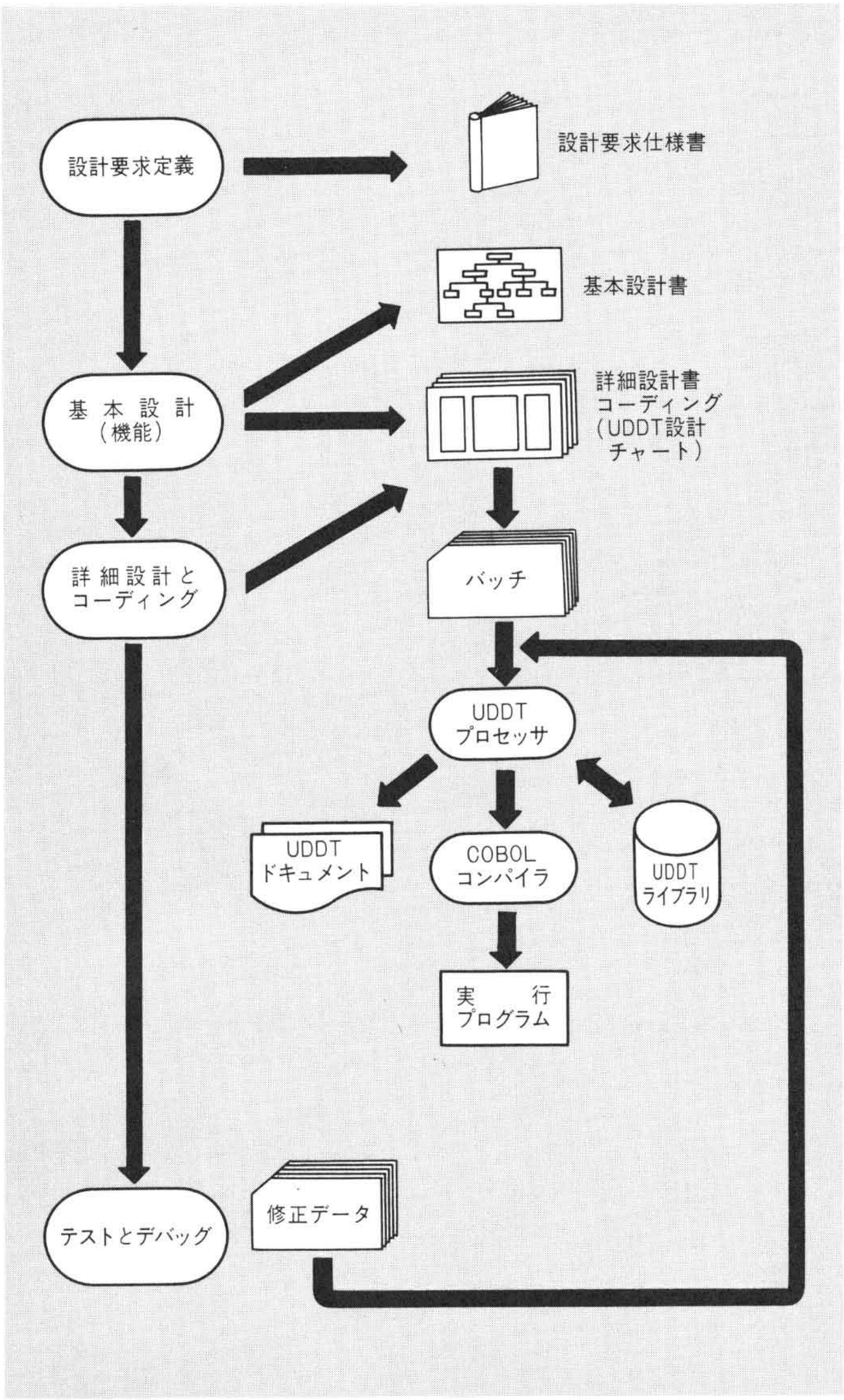


図5 UDDTによるソフトウェア開発の流れ ソフトウェア開発過程でのUDDTの位置付けと手順を示している。

設計書が不要となることを考えると，負効果は吸収されるといえる。本事例での効果は，従来に比べ生産性で約20%，品質で約50%の向上となり，開発費で約30%の向上となった。

6 結 言

UDDTは，COBOL言語の構造化向きでない面に対して，論理構造の概念の導入及び言語機能の導入を行ない，COBOLによる構造化プログラミングを実現可能とした。また同時に，ドキュメントの自動化により，生産性の向上を実現した。現在，UDDTは業務用アプリケーションパッケージとして，一般ユーザーに提供している。

最後に，UDDTの開発に際し，御指導，御援助をいただいた関係各位に対し，厚く謝意を表わす次第である。

参考文献

1) Dahl, O. J, Dijkstra, E. Wand Hoare C. A. R：Structured Programming, Academic Press, London (1972)
2) Wirth, N：Program development by stepwise Refinement, Comm. of ACM144 (1971.4)
3) 国友義久：効果的プログラム開発技法，近代科学社 (1978)
4) HIPO：A new program documentation technique, IBM (1974)
5) Y. Matsumoto, K. Yamano and I. Nakata：UDDT：A practical technique for structured programming and documentation, 3rd UJCC (1978)
6) 山野紘一，松本良治：構造化プログラミング用パッケージUDDTについて—そのドキュメント手法—，情報処理学会，ソフトウェア工学研究会 (1979)

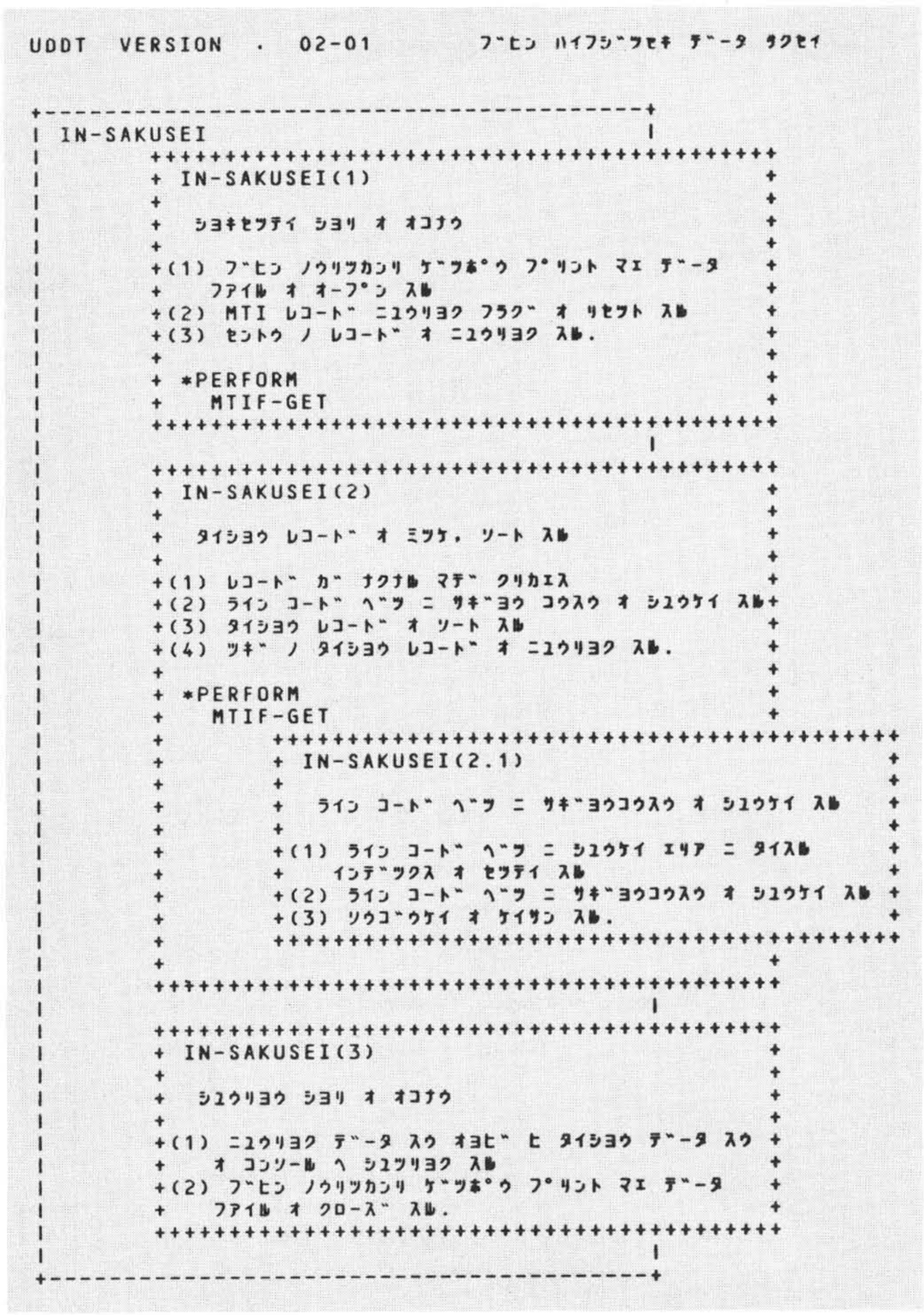


図6 セグメント関連リスト プログラムの構造を，セグメント単位に階層的に表現している。

表1 UDDTの効果 UDDT適用の結果得られる低減量を示している。

比較項目		COBOLの場合	UDDTの場合	低減量(%)
開発工数（単位：人月）		9.3	7.3	21
計算機使用量（単位：時間）		40	35	13
品質（単位：不良件数/1,000ステップ）		6	2.7	55
開発費（単位：1,000円）	作業量	4,100	2,900	29
	カードパンチ費	120	140	−17
	計算機費	4,000	3,500	13