ソフトウエア工学

講義ノート

長井　務

第一章　ソフトウエア工学の基礎

第１課　ソフトウエア概論（２コマ）

１節　ソフトウエアの本質

１　一般的定義

ソフトウエア(software)、ハードウエア(hardware)

「コンピュータシステムの中で、装置や部品類を除外した結果残った部分のシステムがソフトウエアである。」
「目的とされる昨日サービスを、ハードウエアによって実現するための応用技術のすべて」

オペレーティングシステム(operating system)

「コンピュータのハードウエアの処理能力をより効率よく活用するためのプログラムサービス」

広義のソフトウエアの体系

基本ソフト・応用ソフトというプログラム群、設計における技法・方法論、要員のノウハウ・経験、各種ドキュメントを体系化したもの。

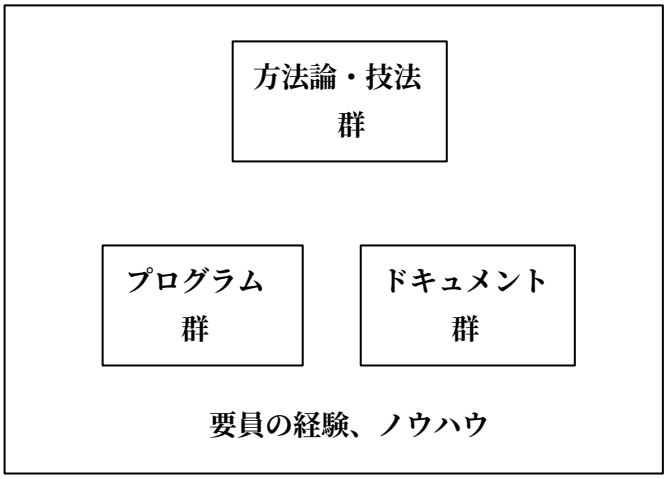


図 1

(1) 技法・方法論

システムの具現化のための概念や手順を示し、問題解決のためのアプローチであり、標準化された設計、開発手順が提供される。
要求定義技法、システム設計技法、プログラム設計技法、テスト技法

(2) プログラム

ハードウエアとのインターフェイス部分であり、データ処理の作業順をコンピュータに指令する命令群によって構成されている。
基本ソフトウエア・・ハードウエアの処理効率を向上させ、ユーザにとってコンピュータ操作がしやすい環境を提供する
オペレーティングシステム・・システムプログラム
応用ソフトウエア・・適用業務処理を実現する
ユーザ・パッケージプログラム・・アプリケーションプログラム

(3) ドキュメント

ソフトウエアの開発および保守作業に必要とされる利用価値の高い、有効性のある資料である。
要求定義書、各種設計仕様所、テスト条件書、運用手引書、オペレーションマニュアルなど

(4) 要員

抽象的なシステム化ニーズを、プログラム言語の命令に置き換えていく具現化の工程を、要員の頭脳と手作業によって行う
要員の技能や経験、アイデアやノウハウ

2 ソフトウエア構成

(1) 階層構造(hierarchical structure)

システム(system)

＊最も上位のレベルで、統括された機能要素群の総称である。

＊単一機能を有する個々の要素が、ある相互関係のもとに集約された集合体である。

サブシステム(subsystem)

① 下位のレベルで、複数のプログラムによって構成されている。

② プログラム同士は、中間ファイルによって関連付けられ、その入出力理によって、プログラム間のデータの受け渡しが行われる。

③ 処理の対民ぎゃ取り扱うデータの種類、あるいは同一処理機能などによって区分けられる。区分けられた単位をジョブ(job)という。

プログラム(program)

① 次の下位レベルで、複数のモジュールによって構成されている。

② モジュール同士は**アーギュメント(argument：引き数)**によって関連付けられ、その設定によって、モジュール間のデータの受け渡しが行われる。

③ プログラムは、データ処理の流れである加工手順によって区分けられる。

④ ある入力データを、要求された出力情報として加工変換する処理単位が、１本のプログラムとして構成される。区分けられた単位を**ジョブステップ(job step)**という。

モジュール(module)、サブルーチン(subroutine)

① 最下位のレベルで、モジュールとサブルーチンから構成されている。

② モジュールは、１つのまとまった分割機能を持つもので、独立にコンパイルできる命令の集合体である。したがって、プログラムや他のすべてのモジュールから呼び出しが可能となる機能単位である。個々のモジュールが持つ機能は独立性が保たれており、１つのまとまった

処理機能を実現することができる。

③　サブルーチンは、いろいろなモジュールやプログラムから呼ばれ、共通の処理を行うモジュール（共有制を持つモジュール）である。共通の処理を行うために必要とされるデータは、アーギュメントによって引き渡される。したがって、呼び出し側はデータを入力アーギュメントとして、サブルーチンに渡すだけでよい。サブルーチン内で行われる共通的な処理は隠蔽化されることになる。

段階的詳細化のアプローチによる下位レベルが設定された階層構造のソフトウエア構成は図＊－＊である。

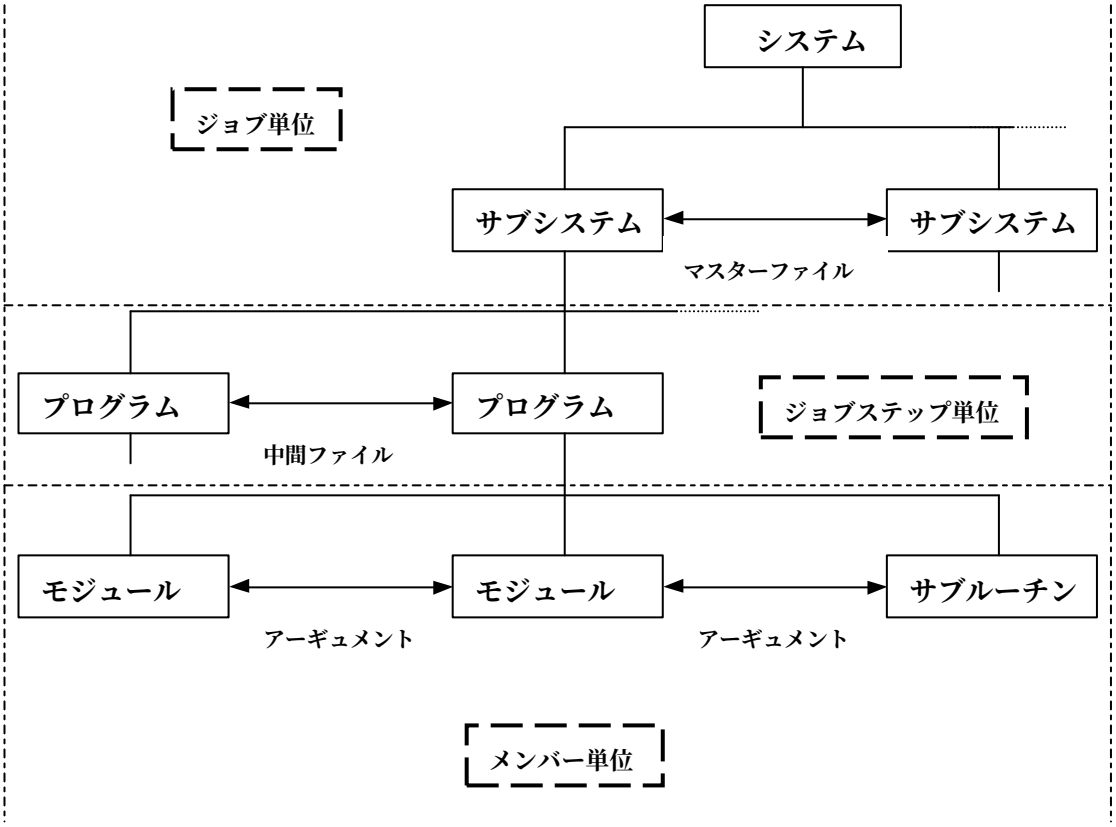


図 2

(2) 構造と手続き

ソフトウエアの構造

ソフトウエアの構成要素

構造的な面
手続き的な面

ソフトウエアの構造

ソフトウエアが持つ各機能の構成を示すものであり、機能の階層条件を表示したものである。

ソフトウエアの機能設計における構造決定の要素

縦の構造・・機能の階層構造における詳細化レベルの度合いを示す

構造が深いほど、機能間の制御関係が入り組み、ソフトウエアが複雑化する。

横の構造・・機能の持つ制御範囲の広さの度合いを示す

構造が広いほど、分割された機能が多いことを示し、ソフトウエアが大規模化する。

ソフトウエアの手続き

ソフトウエアが持つ処理手順の関係であり、繰り返しや判断条件、あるいは処理の流れの制御（戻りや飛び越し）などとったアルゴリズムそのものを示す。これは、ソフトウエアが持つ個々の機能を、どのような手順で実現すべきかといった、ロジックの構造に着目しているといえる。

効果的なソフトウエア設計

不完全なソフトウエア設計を回避するためには、ソフトウエア構造の設計をきちんと終えたあとに、ソフトウエア手続きの設計を行う必要がある。構造設計によりソフトウエアの機能構造が定義づけられることにな

り、その次の手続き設計によって、定義づけられた個々の機能単位の手続きを明確にすればよい。

ソフトウエアの構造は図＊－＊である。

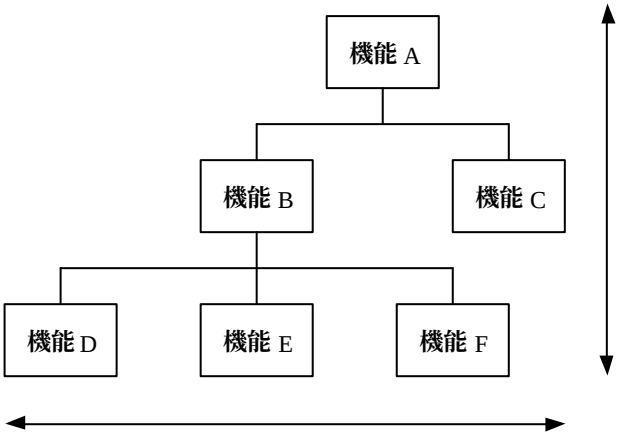


図 3

3　ソフトウエアの概念(software concept)

ソフトウエア開発技術・・方法論の発展がさまざまな技法を生み出した

方法論(methodology)・・ソフトウエアを作成するための基本的な尺度、あるいは、定性的な指針

技法(technique)・・ソフトウエアを作成するための具体的な手順、あるいは、定量的な科学的アプローチ

ソフトウエアの概念

モジュラリティの概念(modularity concept)

ソフトウエアの構造という側面から、部分化という概念

段階的詳細化の概念(stepwise refinement concept)

ソフトウエア開発工程における設計指針に適用

抽象化の概念(abstraction concept)

手続きや制御だけでなく、データに対応する処理まで抽象化する考えで、ソフトウエア開発技法に新しい変革をもたらせた。
これは、情報隠蔽の概念(information hiding concept)、抽象データ型(abstract data type)、オブジェクト指向(object oriented)を生み出した

ソフトウエアのパラダイム(paradigm)

これまでの手続きや制御の抽象化というソフトウエアの機能構造に着目した伝統的な技法に対して、データの抽象化というソフトウエアのデータ構造に着目した新しい技法が提示された。さらに、オブジェクトの抽象化では、機能までも固有のデータ（オブジェクト）そのものに抽象化してしまうという、今までのソフトウエア開発技法・・・データとプログラムの分離・独立を前提・・・には考えられなかった着想が導入された。

ソフトウエアの概念は図＊－＊である

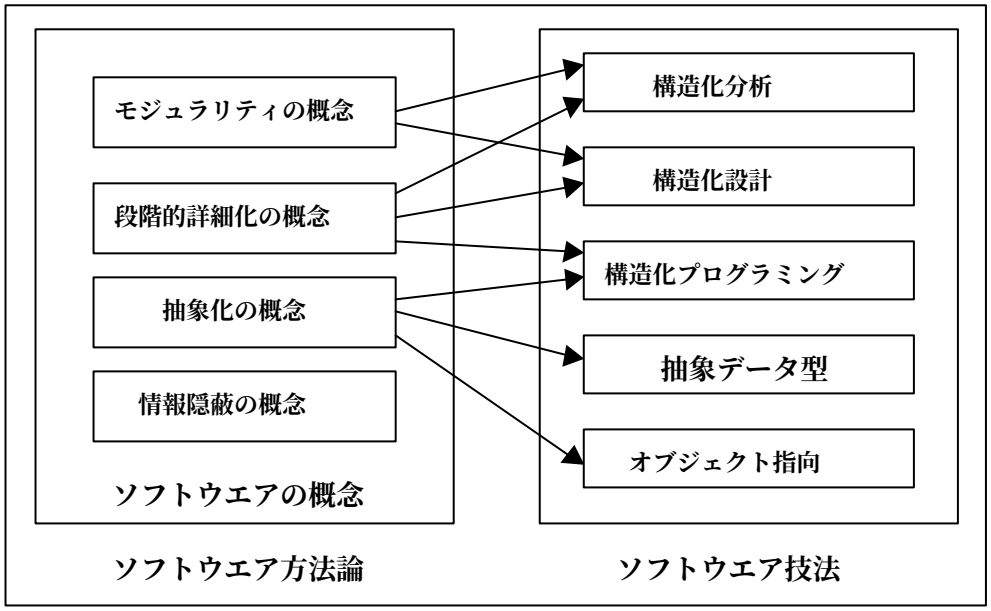


図 4

(1) モジュラリティの概念

モジュラリティとは

ソフトウエアを複数のモジュールに分割し、それぞれのモジュールに機能を与え、ソフトウエア全体を管理すること。

分割・統合(partion and marge)の概念

ソフトウエアは、モジュールを統括（連係編集）することによって実行可能となるので、ソフトウエアは独立性を保ちながら、モジュールごとに機能分割した後に統括する。

＊１　モジュール分割の方法

深さの分割

対象となるモジュール自身を段階的に詳細化する。ソフトウエア構造の縦の構造に対応した分割基準である。

深さの分割手順は図＊－＊である。

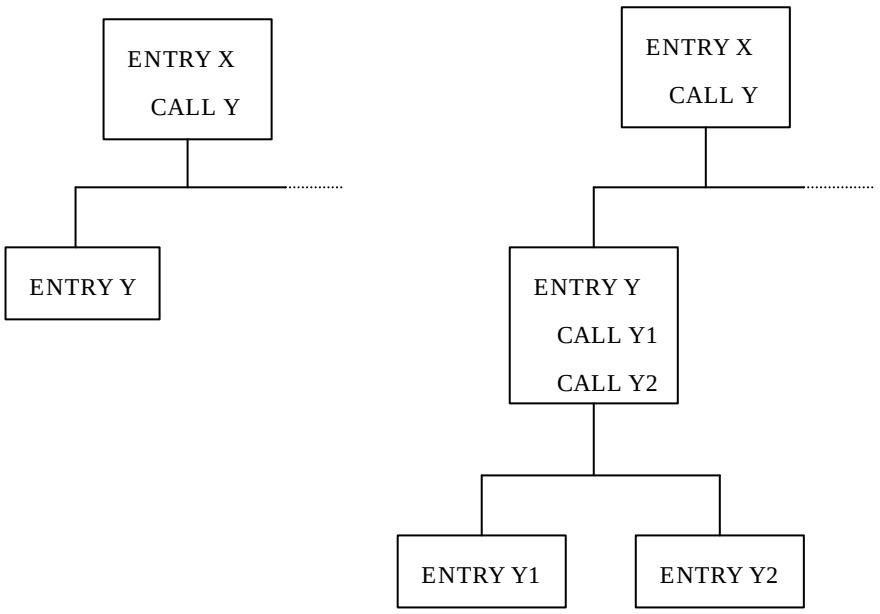


図 5

幅の分割

機能の大きさからの制約に依存することによって行われる。ソフ

トウエアの横の構造に対応した分割基準である。

幅の分割手順は図＊－＊である。

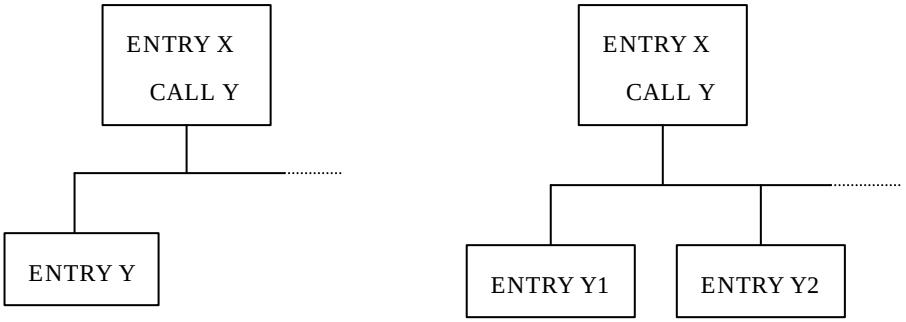


図 6

モジュールの独立性

モジュール分割の基準として、モジュールの独立性を保ちながら、両極端な構造（深すぎたり広すぎたりする）にならないよう中間的構造へと最適化を図る。

モジュール分割の基準である中間的構造は図＊－＊である。

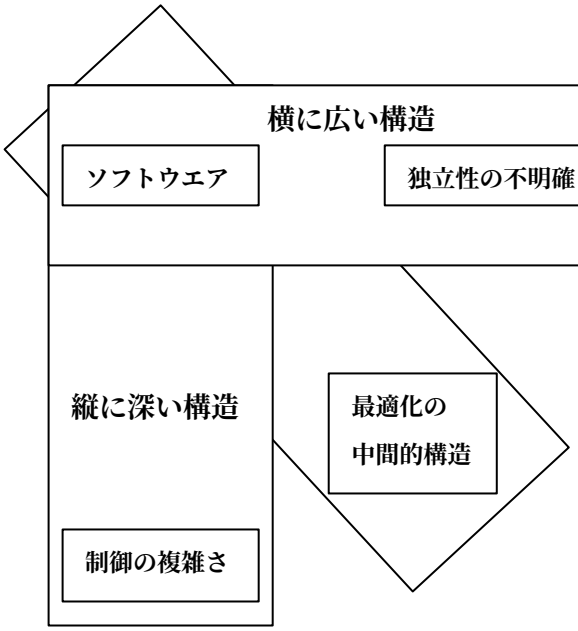


図 7

＊２　モジュール間の関係

モジュールを分割すると、モジュール間のインターフェイスといった関連性の問題が生じてくる。

分割モジュール数の増加

モジュール個々の機能が独立化され、モジュール１つの総ステップ数が減る。したがって、処理手順も簡素化される。全体として、モジュール開発は容易となり、工数も短縮化される。並列的にいくつかのモジュール開発を同時に進められる。が、一方では、アーギュメントに関する制御が複雑となる。アーギュメントに関する設計とモジュール制御に関する設計にかかる工数が増大化する。

分割モジュール数の減少

アーギュメントに関する制御は簡素化される。したがって、インターフェイスに対する設計工数は少なくなるが、モジュール個々の機能の独立性を保つことが難しくなり、モジュール１つの総ステップ数が増え、処理手順も複雑化される。

分割モジュールの関係は図＊－＊である。

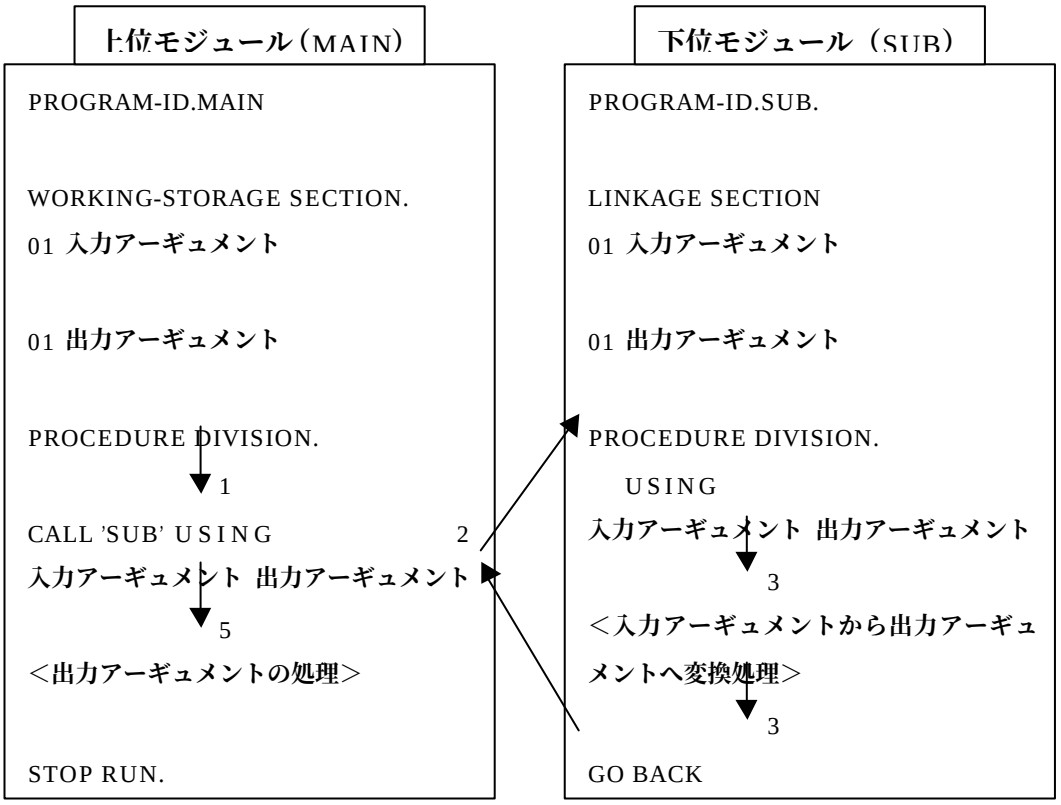


図 8

(2) 段階的詳細化の概念

段階的詳細化とは

解決すべき対象である外界の事象に対して、レベルを分けながら徐々に細分化して解決を進めるという考え方。トップダウンアプローチ(top-down approach)という場合もある。

課題

「昇順に並んでいる数値群（重複なし）の中から、K（外部から与えられる数値）に等しい値を見つけ出し、その位置を印刷する」

問題解決手順

段階的洗練化を用い、概念的なものから詳細的なものへと各レベルに応じながら具現化する。最終的にはプログラム１ステートメントに対応したプログラム記述言語として表現する

１　課題をいくつかの機能に分解する。

この分割工程が、段階的洗練化の手順になっている。

２　「K の格納」

プログラムのメモリ上に外部から値を読み込み、初期値設定することを示す。

３　「数値群の配列設定」

数値群を配列として設定しメモリ上に格納することを示す。配列を取り扱うことによって、データの構造も決定されることになる。

１次元配列・・A(i) (i=1,2,3…,N)

格納されている数値群・・A(1), A(2),…, A(N)

４　「数値の検索（２分検索法)」

２分検索処理を行う。これだけでは、実際の処理手順が不明確である。

５　「位置の印刷」

数値の検索処理において検索条件を満足したときに得られた位置を印刷することを示す。

６　「数値の検索（２分検索法)」の洗練化と具現化

＊１　「配列の範囲を初期値設定」

検索を行うための配列範囲をメモリ上に初期値設定することを示す。
　最小値(MIN)に１、最大値(MAX)に N を格納する

＊２　「K と等しい値があるか検索範囲があるまで検索処理を繰り返し」

配列を２分検索によって処理することを示す。ただしこれだけではどのような手順で、K と同一の値を検索するかが不明確である。

７　「K と等しい値があるか検索範囲があるまで検索処理を繰り返し」の洗練化と具現化

＊１　「検索範囲として配列を２分し、中間点を設定する」

配列の検索範囲を２分する手順を示す。
　(MIN+MAX)/2 の除算値を切り上げる
　対象となる検索範囲の中央の地点が位置付けられる

＊２　「中間点に格納されている数値と K を比較する」

上記の計算結果で得られた中間点に格納されている数

値と K との比較を行う

等しければその中間点が求める位置である。それ以外は再度検索を行う手順へ進む。

＊３　「検索範囲の再設定を行う」

中間点の値とが等しくない場合、検索範囲を設定しなおす。

A（中間点）>K の場合と A（中間点）<K の場合がある。前者の場合、（中間点-1）を MAX に置き換え、後者の場合、（中間点+1）を MIN に置き換える。

これによって、前回の検索半のさらに半分の幅が新たな検索範囲として設定される。

8　プログラム記述言語を記述する

２分検索処理のプログラム記述言語

```
開始
  K を読む
  A(I)(I=1,2,3,...,N)を１次元配列とする
  MIN=I
  MAX=N
  DO WHILE(K と等しい値があるか、MIN>MAX まで)
    MID={(MIN+MAX)/2}切り上げ
    IF A(MID)>K THEN
       MAX=MID-1
    ELSE
      IF A(MID)<K THEN
         MIN=MIN+1
      ELSE
         MID を印刷する
      END IF
    END IF
  END DO
終了
```

(3) 情報隠蔽の概念

情報隠蔽とは

モジュール間の関連性において対象となるモジュールの必要となる部分（機能もしくはデータ）のみを見せ、それ以外の部分における情報はすべて隠してしまうことである。

これによって、必要とされる情報のみに対象が絞られ、結果としてモジュール間の独立性が高められることになる。

たとえば、モジュール A、B を設定する。この２つのモジュールは、お互いに必要としあう情報がないとする。このようなばあい、モジュール A が有するデータやプロセスは、モジュール B からはまったくアクセスすることができないよう構成されている。また逆に、モジュール A からもモジュール B に対して同様のアクセスができない状態となる。

情報隠蔽の概念

モジュール分割に適用でき、結果として有効な指針を与える。すなわち、効果的なモジュラリティとは、モジュール同士が必要な情報だけで結合しあっているものであり、モジュールの独立性が保障されていることが前提条件となるということである。

この概念を、モジュール仕様における記述方法にまで発展させたのが、D.L.Parnas である。これは、データとそれにかかわるいくつかの機能（ある演算処理を通してデータを操作する手続き関係）を、１つのモジュールにまとめる設計概念である。

STACK のモデルは図＊－＊である。

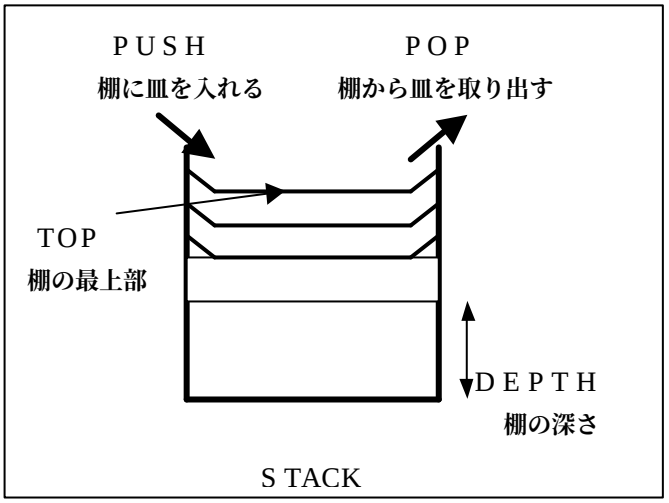


図 9

Parnas モジュールによる自動皿洗い機のモデル仕様例

Function PUSH(a); PUSH モジュールで a をもらう
Possible values:none a をもらってもモジュール自身は値を格納せず
Integer:a a は整数値である
Effect:call ERR1 A が P2 より大きいか、a が 0 より小さいか、スタッ
if a>P2 a<0 `DEPTH`=P1 クの深さが P1 と同じとき ERR1 ルーチンを呼ぶ
Else [VAL=a; `DEPTH`= 以外のとき VAL モジュールに a を渡し、スタック
` DEPTH`+1;] 深さに 1 を加える

Function POP POP というモジュールである
Possible values:none モジュール自身は値を格納せず
Parameters:none パラメータはなし
effect:call ERR2 スタックの深さが 0 のとき ERR2 ルーチンを呼ぶ
if `DEPTH`=0
the sequence
PUSH(a):POP エラーがなければ、PUSH(a)と POP を続けて実行す
has no effect ると、もとの状態に戻る
If no error calls occur

Function VAL VAL というモジュールである
Possible values:integer; VAL は整数値を格納するが、初期設定をしない
initial values
undefined
parameters:none パラメータはなし
effect:error call もし、スタックの深さが 0 となったらエラー
if `DEPTH`=0 処理を呼ぶ

Function DEPTH DEPTH というモジュールである
Possible values:integer; DEPTH は整数値を格納するが、初期値は 0 である
initial values 0
Parameters:none パラメータはなし
Effect:none DEPTH の動作はすべて他の呼んでいるモジュール
によって他律的に決定される

(4) 抽象化の概念

抽象化とは

対象とするものの詳細な構造や諸条件にとらわれることなく、一般化された概念に基づく解を取り出すことである。

この考え方は、段階的詳細化にも適用されている。すなわち、複雑な問題をレベルごとに分けながら洗練していくことによって、解決のための思考範囲を限定するわけである。

そして、ある段階で抽象化した部分は、次のレベルで具体化を図り解決していけばよい。これによって、問題が持つ複雑さを軽減することが可能となる。

実現要素(design entity)

＊１　手続きの抽象化(procedural abstraction)

問題の解決に必要となる、あるまとまった操作や演算などを１つの手続きとして集約してしまうことである。そのまとめた手続きの内部処理については、一切考えずに機能だけを抽象化してしまう考え方である。

既存の手続き型言語では、サブルーチンやモジュール呼び出し、あるいは関数といった形で実施されている。

＊２　制御の抽象化(control abstraction)

手続き型言語で記述するプログラムのアルゴリズムを構造化することである。

プログラム　制御の流れ

テキスト・・プログラム言語で書かれた

制御の流れ

時間の経過とともに変化する計算の状態であり、プログラムのアルゴリズムそのものである。入力データが出力データへと変化される過程を示している。時間とともにすいする要素であり、プログラムの動的構造といえる。

テキスト

ソースプログラムリストを構成するステートメント群である。時間とともに変化することはなく、プログラムの静的構造といえる。

構造化プログラミングの誕生

アルゴリズムを追うのは、時間の経過とともにテキストを追跡していくことである。このとき、時間の経過がテキストに反映されていると、状態の変化がよく理解できることになる。

そのためには、アルゴリズムの制御構造を構造化すればよい。具体的には、３つの基本制御構造（連接、選択、繰り返し）のみを用いてアルゴリズムを設計することである。

＊３　データの抽象化(data abstraction)

プログラムを取り扱うデータに対して、そのデータに固有な操作の集合も合わせて定義することによって、抽象化を図るというものである。

データの定義　内部構造によって定義される
データに対する一連の操作によって定義される

抽象データ　内部構造とその操作を１つにまとめて定義したデータである。この場合、データ構造とそれに伴う一連の操作は内部に覆い隠されており、外部からアクセスすることはできないように隠蔽されている。

抽象データ型　抽象データを、プログラムで使用する変数についてそのデータ型の定義（実数や整数、文字など）と変数の宣言とに分けて取り扱えるようにしたときのデータ型をいう。この場合、変数の型定義をしてからその変数に対して実現すべき操作を記述する必要がある。

オブジェクト（object）

データの抽象化の度合いをもう一歩推し進めたものがオブジェクト指向である。このとき、データとそれにかかわる操作を一体化したものを、物であるオブジェクトとしてとらえる。これは、データとその操作を、オブジェクトというもので抽象化したことになる。

そして、オブジェクト同士がメッセージを交換しながら、一連の処理を実現していく。オブジェクトそのものが、自主的にメッセージの交換を機能付けているといってよい。

4　ソフトウエアの一般的特性

(1)　目に見えない製品

ドキュメントである仕様書からプログラミングされることによって作成される。作成されたプログラムは、磁気ディスクやフロッピーディスクなどに磁化されて記憶される。したがって、プログラム自体はまったく目に見えない。

一方、媒体変換によって出力されたソースプログラムリストや画面上に映し出されたプログラムを、コンピュータに対する知識を持たない人々が見ても、まったく理解できない。

(2)　品質は恒常的に維持され、向上する

ハードウエア機器は、長時間経過すると故障したり性能が劣化したりする。これは部品の物理的特性に依存する。

ソフトウエアでは作りこまれた品質が恒常的に維持される。また、ソフトウエアに信頼性は、偶然的に潜在的バグ(bug：プログラムが内包するエラー)が見つかり、それを正しく取り除くことによって向上していく。

(3)　潜在的バグが潜む

ソフトウエアの作成は人手にたより、又、ソフトウエア自体が定量的に分析することが難しい。現状のソフトウエア生産管理技術で

は、完璧なソフトウエアを作ることは不可能であり、必ず潜在的なバグが内包されている。

(4) バグは他人のほうが発見しやすい

プログラマは、自分がコーディングしたプログラムに対しては完璧さを望んでおり、エラーがないように心がけてきたという心理的安心感とプライドを持つことが多く、エラーの発見に対する消極さを促し、間違いを見逃しやすい。

(5) 機能は社会情勢とともに変化する

ソフトウエアとはEDPS(Electronic Data Processing System)のDP部分であるといえる。これは、データを加工処理し、付加価値の高い情報として提供するための機能の中枢部である。また、ある目標を実現するために構築されるわけで、それらは何らかの形で社会システムと関連をもつ。したがって、日々刻々と移り変わっていく社会情勢に、ソフトウエアの機能も対応していく必要がある。

このことが、長期的な保守作業を実施させることになり、ソフトウエアが不要となるまで維持され、ライフサイクルの長期化にもなる。

(6) 変更時に波及効果が生じる

ソフトウエアのある個所を修正したら、他の部分に新しい潜在的バグが作りこまれるという波及効果を及ぼす。これは、あいまいな仕様書類や構造化されていないプログラム群、あるいは、目に見えない特性などに起因する不明確さに依存することが多い。

(7) 作成者の世界観が含まれる

方法論も概念も作成者によってまちまちであり、ソフトウエアの機能実現のための手続きも、各人各様である。このことにより、他人の作成したソフトウエアの読解を困難にさせ、修正ミスを誘発させる。

(8) 構築にはバイナリ的思考が必要

ソフトウエアのロジック展開には、二つのうちのどちらか一方というバイナリ的思考が必要である。たとえば、ある条件に対して YES か NO か、ループ処理内かループ抜け出しか、ファイル読み込み時に AT END か否かなどといった考え方である。この考え方を２進形式の処理手順で記述表現したものが、ソフトウエアのロジックとなる。ロジックの組み合わせによって、与えられた機能がソフトウエアに実現されることになる。

(9) 簡単に複写できることの功罪

いろいろな記憶媒体上に記憶することができるため、記憶媒体同士で簡単に複写することができる。このことが、ソフトウエアの再利用化を促進させ生産性の向上を図らせる一方、安易なソフトウエアのデッドコピーを行わせてしまうことにもなる。

２節　よいソフトウエアとは

よいソフトウエアという概念には、「基幹となる認識」と、「処理効率」、「理解容易性」という概念が含まれている。その中で、「基幹となる認識」は時代の移り変わりに関係なく、それに反して、「処理効率」、「理解容易性」は、時代の推移によっては、相反する増減効果の傾向を示す。これは、図＊－＊に示す。

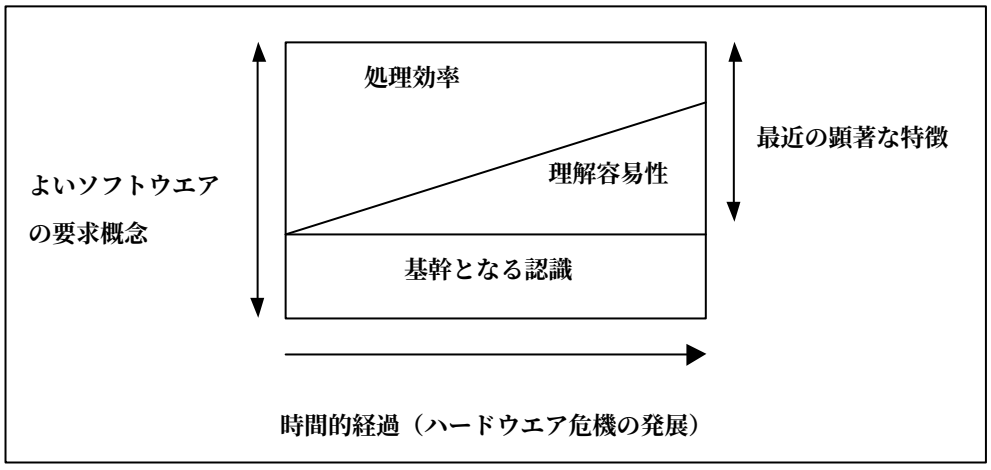


図 10

1　基本的要素

(1)　ユーザの要求仕様が正確に反映されている

ソフトウエアを開発するための目的は、すべてユーザニーズから生じ、どんなに高度な技法を駆使して作成されたソフトウエアでも、ニーズを満足していなければ、まったく価値がない。

(2)　潜在的バグが少ない

現段階では完璧なソフトウエアの構築は不可能に近い。これは、ソフトウエア生産工程ではソフトウエアの品質を定量的に分析できないというテストの不充分性があることと、人間に依存した作業が主体であることから起こる、避けられないヒューマンエラーがあるからである。これらによって発生する潜在的バグを極力少なくする方策が必要である。

(3)　開発コスト以内である

開発コストは人的工数に依存する。開発に参画する人数に開発日数を乗算した総合計がコスト値となる。開発コストには制限があり、ソフトウエアの規模と機能内容の複雑さというレベルに見合わないこともある。想定された見積もり工数以内の開発コストであり、定められた納期内に開発が完了することが要求される。

(4)　運用が容易である

運用が容易であることは、煩雑なオペレーションを必要とせず、システムの稼動操作において人間（オペレータ）の介在があまり起こらず、トラブルリカバリー処理が絶やすことである。

(5)　安全性が高い

ソフトウエアが扱うデータの機密保護が考慮されており、ハードウエアに起因するトラブルに対する適用能力を備えていることである。

2　処理効率

処理効率とは、ソフトウエアが EDPS としてデータを処理するときのシステム稼働率である。これは、ハードウエア機器の発展に大きく影響されている。

以前は、よいソフトウエアを作成するためには、ハードウエアの性能をさいだいげんに発揮できる機能を組む必要があった。

ところが、ハードウエア技術の高度化が、ハードウエアの性能面からの制約を、ソフトウエアの世界からなくすにいたった。そして、ソフトウエアにおける処理効率への考慮をほとんど不要とさせるようになってきた。

(1)　時間効率がよい

時間効率とは、ソフトウエアが実際に稼動しているときの実行処理時間である。

バッチ処理・・ターンアラウンドタイム(TAT:Turn Around Time)

リアルタイム処理・・レスポンスタイム(resuponse time)

ファイルへのアクセス時間の短縮化や処理手順の最適化などを、ソフトウエアで考慮する必要があり、時間効率の向上が重要である。

(2)　資源効率がよい

プログラムの保存記憶領域の圧縮かを図り、プログラム実行時のメモリ領域の最適化を図るようにプログラム構造を工夫した。

3　理解容易性

ソフトウエア構築の工程では、ハードウエアの制約はほとんど考えず、ソフトウエアのわかりやすさのほうが重要視されるようになった。時代の推移とともにソフトウエアに対する需要が高まったために、よいソフトウエアの第一条件として分かりやすいことが要求されてきた。

(1)　構成や設計構造が平易

きちんと設計を行い、構造化されたソフトウエア構成にする必要がある。

(2) 検査がしやすい

見やすくわかり易いソフトウエア記述であり、部品化が行われ機能の独立性が保たれている必要がある。

(3) 保守がしやすい

ソフトウエアに対するユーザニーズは、社会情勢の変化とともに変わっていく。このことが、ソフトウエアのライフサイクルを長期化させ、その間ソフトウエアの保守作業を繰り返させることになる。

そのときに保守しやすいためには、ソフトウエアがわかりやすく、構造化が保たれており、整合性のあるドキュメント管理がされていることが必要である。

(4) 高品質のドキュメントを含む

ドキュメントとして、仕様書やテスト条件などがある。

処理効率と理解容易性の比較については図＊－＊に示す。

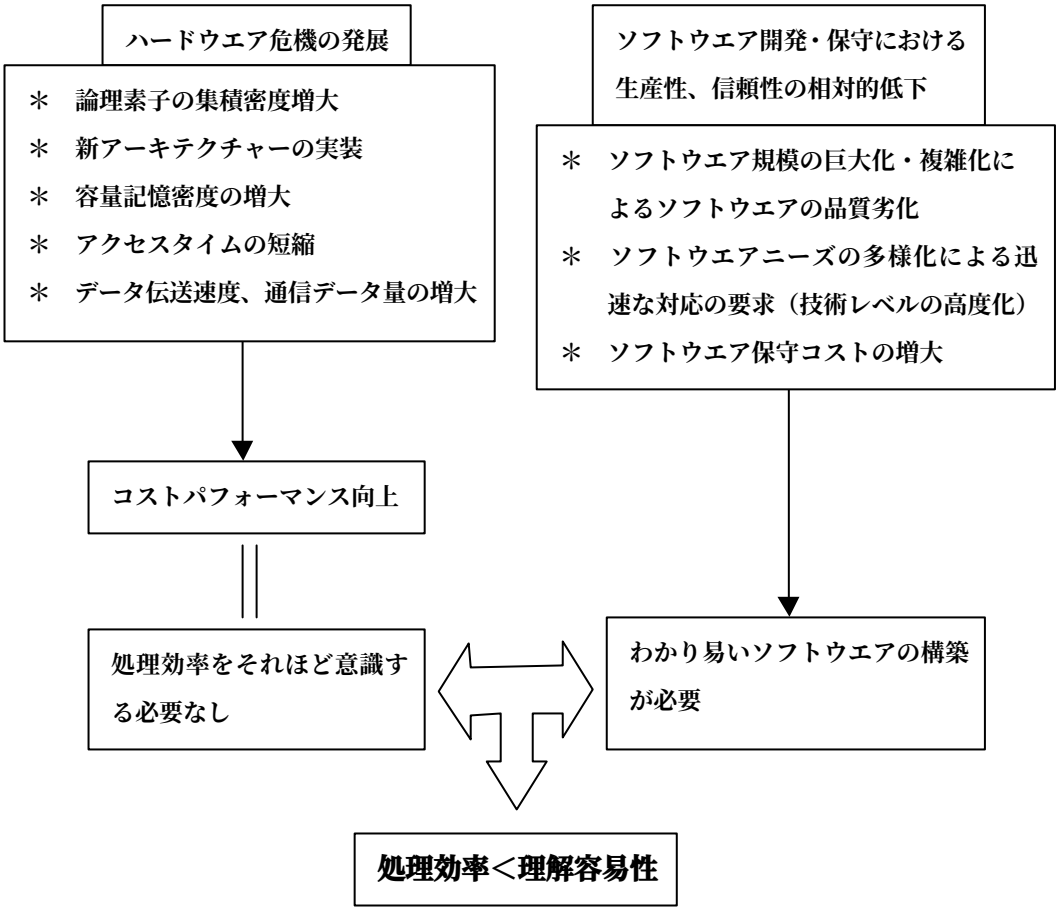


図 11

３節　ソフトウエア産業界の現状と将来

ソフトウエア危機 (software crisis)

１　ソフトウエア危機

(1)　ソフトウエアの巨大化、複雑化に伴う開発費用の増大

(2)　ハードウエア対ソフトウエアのコスト費用の逆転

(3)　保守に関する工数の増大とそれに比例したバックログの増加

(4)　ソフトウエアの需要に対する供給能力の低下

(5)　ソフトウエアトラブルの社会問題化

２　情報産業における諸問題

(1)　ユーザの要求を明確に記述する方法がなく、システム稼動後にトラブルを起こしやすい

(2)　大規模なソフトウエアほど長期の仕様凍結が行われ、ユーザに対してタイムリーな対応が取れない

(3)　一貫した設計方法論もなく、場当たり的な設計が、ソフトウエアの品質を劣化させている

(4)　ソフトウエア生成工程におけるドキュメンテーションの基準がなく、不明確な仕様記述がソフトウエアの品質を劣化させている

(5)　ソフトウエアの正しさのチェックは、設計段階ではなく最終フェイズであるシステムテストの段階となるが、この時点でミスが発覚しても納期に間に合わないことが多い

(6)　設計より製造（プログラミング）工程を重視することが多く、ソフトウエアの品質を劣化させている

(7) ソフトウエアの再利用化に対して消極的であり、生産性の低下を招く

(8) 開発工程の多くが人手作業であり、生産性を低下させている

(9) ソフトウエアの正当性を証明できないことが、信頼性の低下を招く

(10) 良質なソフトウエアの基準が定量的に測定できず、正当なシステム評価ができない

(11) 保守作業に膨大な工数が費やされ、要員の工数効率を悪化させる

(12) 保守作業が長引くと、ドキュメントの品質も悪化し、他に多くの悪影響を及ぼす

(13) プロジェクト管理が困難で、スケジュール管理があいまいとなる

(14) 工数見積もりの算定基準がなく、納期遅延や開発経費超過を招く

第２課　ソフトウエア工学（１コマ）

１節　提唱と歴史

1　1960 年代前半

ソフトウエアへの関心が薄く、コストや品質の議論もなし

2　1960 年代半ば

汎用オペレーティングシステムの開発が進み、大規模なソフトウエアシステムが構築された。コストや品質の議論が始まる。

3　1968 年

ソフトウエア工学(software engineering)が提唱された

NATO の科学委員会（西ドイツ）
ソフトウエア生産・保守における実務工程を理論的方法のもとに体系化し、工学的アプローチを図るという意図
言葉だけの先行で実質的な方策は見出されなかった

4　1960 年代後半

アンバンドリング政策（価格分離政策）
1969 年 IBM 社
ハードウエアとソフトウエアの価格を別に設ける規約
ソフトウエアへの問題意識が高まり、品質、生産性に関する議論が活発となり、プログラミングの方法論における基礎研究が始まる

5　1970 年代前半

品質向上のためソフトウエア設計工程における信頼性の保証と、プログラムは理論的な方法論による設計方法に従い、一定の基準によって作成するべきであることが提言された
構造化プログラム、モジュラ・プログラミング、構造化設計

6　1970 年代半ば

ソフトウエア工学の語が定着し始めた
1975 年　ソフトウエア・エンジニアリング国際会議(ICSE:Internatinal Conference on Software Engineering)

7　1970 年代後半

要求定義技術や設計方法など、開発工程の上流部分における重要性が認識され始めた
ソフトウエア・エンジニアリングのライフサイクル論が位置付けられた
ICSE 国際会議：1979 年に第４回目を実施
日本：1976～1981 年　「ソフトウエア生産技術開発計画」プロジェクト

8　1980 年代前半

学問・実務レベルともソフトウエア工学への関心が高まる
業界はソフトウエア・プロダクト(puroduct)やツール(tool)類を製品化し、販売した
ICSE 国際会議：1981 年に第６回目を日本で実施
日本：1981～1985 年　「ソフトウエア保守技術開発計画」プロジェクト

9　1980 年代後半

ソフトウエア工学分野の学問レベルから実務レベルへの移行転換期
ソフトウエアの生産性、信頼性、保守性という品質特性は改善、向上したと言えず、危機が続いている
ソフトウエア生産の自動支援化が始まる
日本：1985 年から５ヵ年計画「ソフトウエア生産工業化システム計画」(SIGMA:Software Industrilized Generator & Maintenance Aids、略称Σ計画)
ソフトウエア開発環境の整備・拡充をワークステーション(work station)上で行い、ソフトウエア生産を工業化し自動化をはかって、生産性を向上させる

2節　設計法の変遷

1　初期　　1970 年代前半

ソフトウエアにおけるモジュラリティの概念が、設計方法論として具現化され、ソフトウエア設計に対する考えとして提出される
段階的抽象化：Wirth,N.,1971、トップダウン設計、モジュラ・プログラミング

2　成長期　1970 年代半ば

モジュール分割方法やモジュール内の設計方法などのソフトウエア設計工程での方法と方法論が提言される

モジュール分割に関する方法

構造化設計：Constantine,L.L./Myers,G.J./Steven,W.P.,1974
複合設計：Myers,G.J.,1975

モジュール内部構造に関する方法

構造化プログラミング：Dijkstra,E.W./Hoare,C.A./Dall,o.,1972
ジャクソン法：Jackson,M.A.,1975
ワーニエ法：Warnier,J.D.,1974

抽象化の概念をソフトウエア設計に適用するような方法が提言された

データ抽象化：Liskov,B.H.,1974

Parnas 法：Parnas,D.L.,1972

3　発展期　1980 年代前半

70 年代に考案された方法・方法論をソフトウエア開発支援ツールに組みこみ実用化をはかる

プログラム・ジェネレータ（プレコンパイラ、グラフィックイメージ入力エディタなど）

会話型簡易言語（第４世代言語、データベース操作言語など）

テスト支援システム

ライブラリ管理支援システム

ソフトウエア再利用支援システム

4　変革期　1980 年代後半

新しいソフトウエア開発環境の設定が方向付けられる

AI を含めた知識工学とソフトウエア工学の合流による新しい展開

体系化された高レベルなソフトウエア開発環境の実現

ソフトウエア生産の自動化

プロトタイピング法によるソフトウエア開発

オブジェクト指向型プログラミングによる開発

エキスパートシステムからのソフトウエア開発支援

ソフトウエア自動生産への方向は図＊－＊である

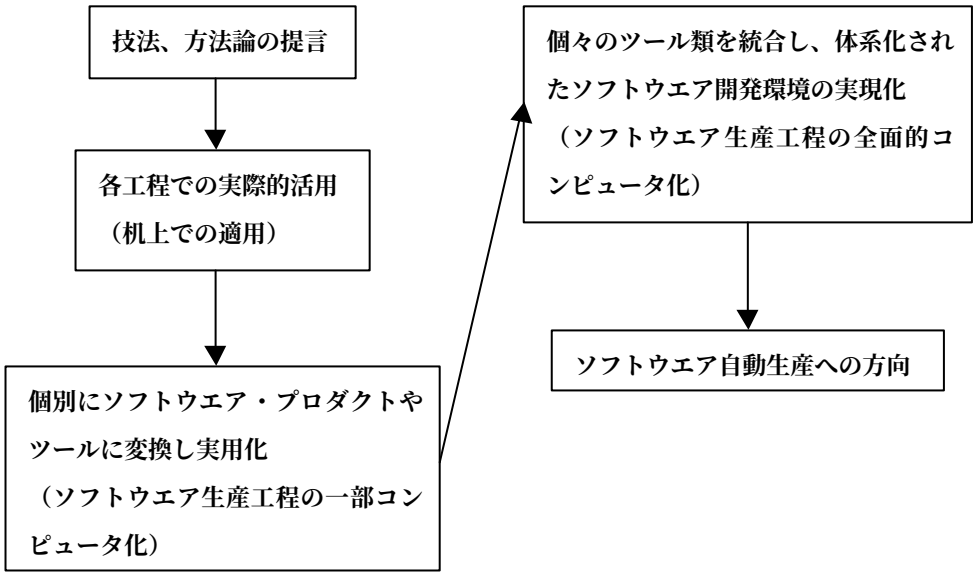


図 12

3節　ソフトウエア工学の定義

「ソフトウエア工学とは、ソフトウエア開発管理工程の全局面において、工学的な方法や方法論、すなわち、ある原理原則のもとに体系化された理論そして技術、というものを導入し適用することによって、ソフトウエア生産を質的および量的に向上させていくための実務的学問分野である

1　普遍的な科学法則の適用が基礎となる

2　ソフトウエア設計、製造、検査、保守のなかで構築される手法を体系化した方法論が、ソフトウエア工学の概念を形成する

3　ソフトウエア開発管理における全工程を、ソフトウエアのライフサイクルと定義し、ライフサイクル・モデルに立脚した方法、方法論がソフトウエア工学の各論テーマとなる

4　ライフサイクルには、製造、設計、運用、保守を含む

5　プログラムだけでなく、ドキュメントも含む

6　工学的アプローチは、ソフトウエアの生産性および信頼性の向上とコスト費用の削減を実現する

4節　ソフトウエアのライフサイクル

ソフトウエアの**ライフサイクル(software life-cycle)**とは、要求に応じてソフトウエア・システムが誕生し、稼動、運用されていくなかで、保守が繰り返され、最終的に廃棄されるまでの期間を示す

ソフトウエアのライフサイクルを、各工程ごとに区分け、フェイズとして構成しモデル化したものを、**ライフサイクル・モデル**という

B.W.Boehm のライフサイクル・モデル（Freeman や Benjamin のモデルもある）

これは、図＊－＊で示す

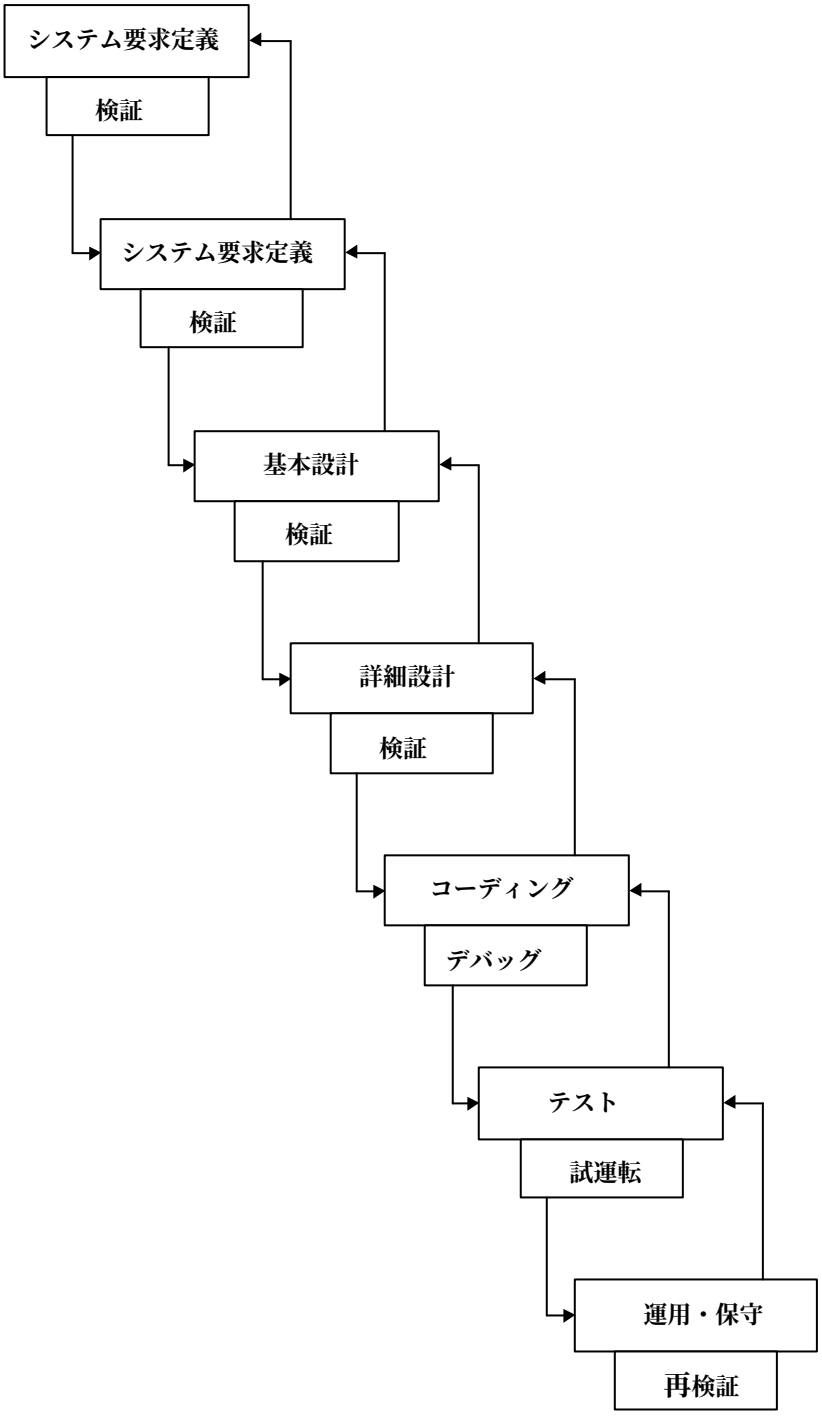

システム要求定義
検証
システム要求定義
検証
基本設計
検証
詳細設計
検証
コーディング
デバッグ
テスト
試運転
運用・保守
再検証

図 13

1　コストを含めた工数負荷は要求特定や設計という上流部分に多く費やす必要がある

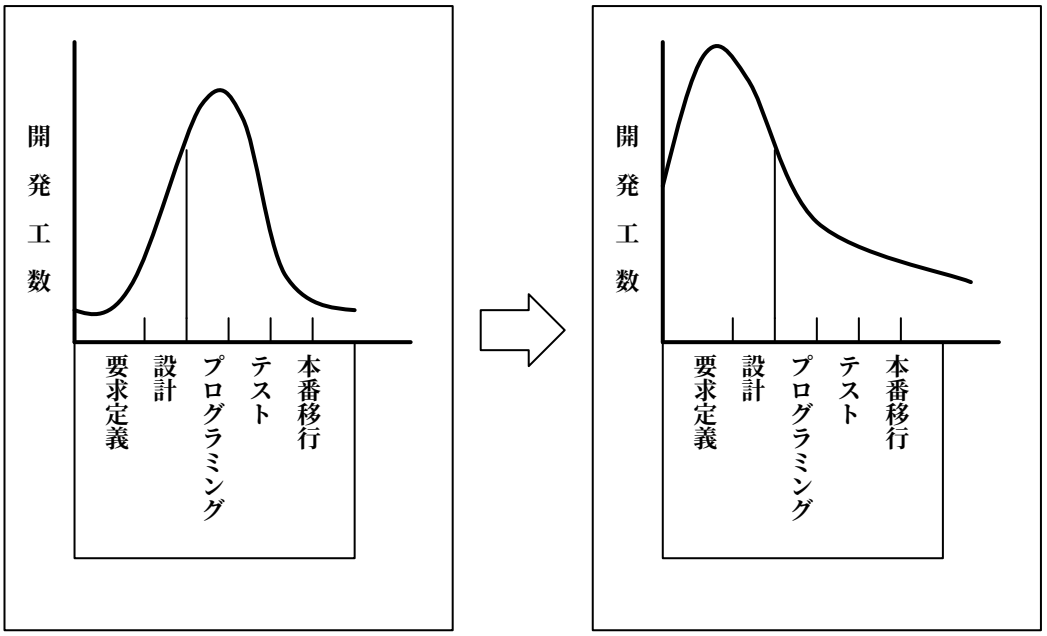


図 14

2　システム構成の具現化は、トップダウン的思考と抽象化の概念、あるいは、段階的詳細化に依存する

3　設計製造工程は、段階的詳細化法によりトップダウン的形式で作業し、テスト工程では、ソフトウエア要素の組み合わせによる集約作業であり、ボトムアップ形式となる

4　ライフサイクルの工程は逐次処理系であり、上流工程から順次作業を進めていくことになる。各工程はすべての事項を明確に定義し、最終的にはプログラム言語１命令ずつに変換される

5　各工程の間に、レビューなどの検査機構をはさみ、品質検査を徹底して潜在的バグを字肯定に持ち込まない

6　各工程ごとに作成されるドキュメント類は、次工程への引継ぎ資料となり、品質検査の資料となる。また、ドキュメントの記述内容が評価の対象となり、量的な面を設計工程における実績の尺度としたり、進行状況の工程管理のチ

ェックポイントとしている。

7　ソフトウエアの品質を安定させるため、各工程で作成するドキュメント類の様式や記述方式を標準化し、規格化する

8　保守作業は変更・修正内容によってライフサイクルの対応する工程に戻るというリサイクル処理となる

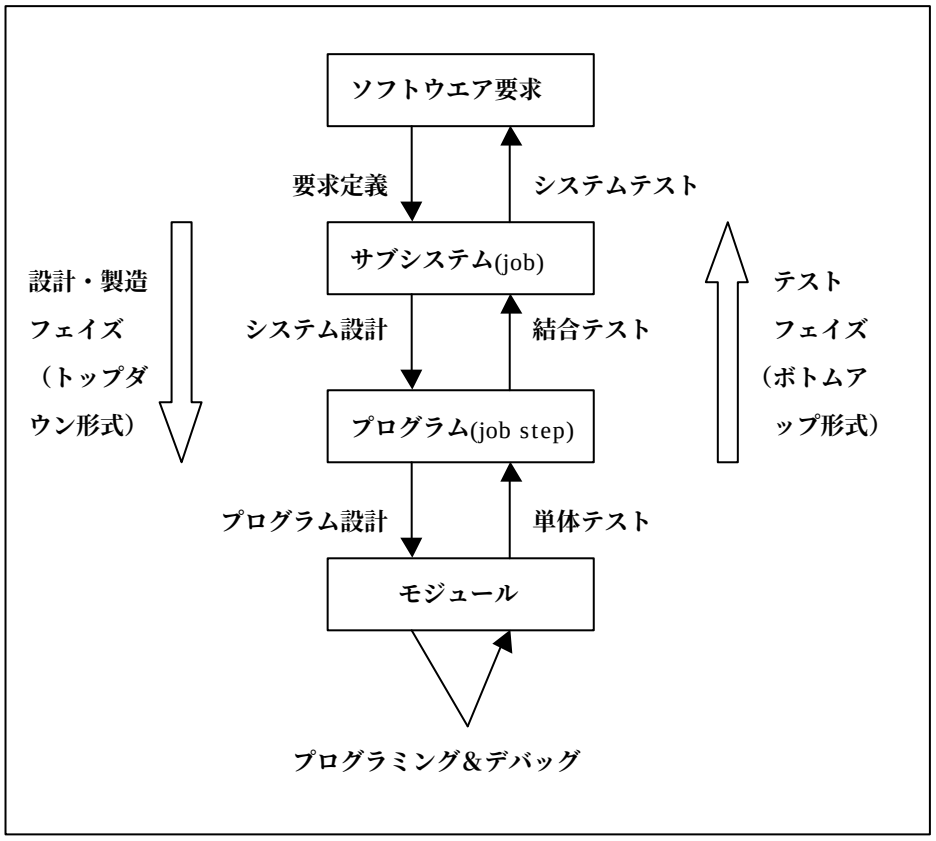


図 15

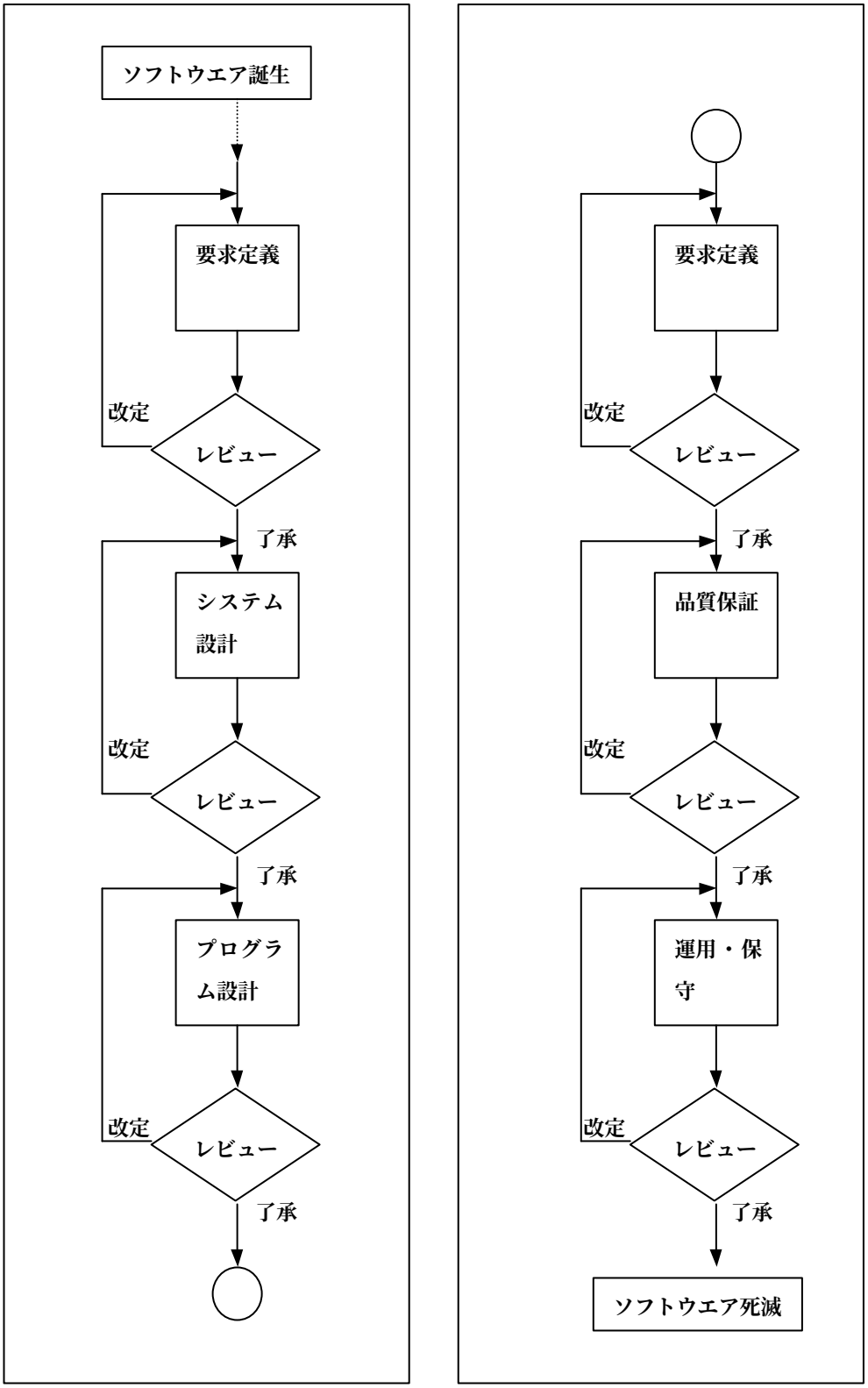

ソフトウエア誕生
要求定義
改定
レビュー
了承
システム
設計
改定
レビュー
了承
プログラ
ム設計
改定
レビュー
了承
要求定義
改定
レビュー
了承
品質保証
改定
レビュー
了承
運用・保
守
改定
レビュー
了承
ソフトウエア死滅

図 16

第３課　ユーザ要求定義法（２コマ）

EDPS(Electronic Data Procesing System)

１節　要求定義とは

ユーザニーズ

ハードウエアニーズ
ソフトウエアニーズ

要求定義

ソフトウエアシステムの稼動により、どのようなニーズが満足されるかを定義ずける
原始データの入力方法、データ精査の方法、ニーズを満たす出力情報としてデータを加工変換する手順、付加価値を有する出力情報の編集方法など、要求仕様の形式でドキュメント化する工程

２節　要求定義における諸問題

最も抽象度が高い内容を扱うが、極めて困難を伴う工程である
効率よく、正しく要求定義を行うためには、豊富な経験とノウハウ(know-how)としての技能が必要であり、人的要員に強く影響される工程である
要求定義の対象となる相手はエンドユーザである、問題が多々発生する

１　具体的問題点

(1)　ユーザのニーズが不明確

(2)　ユーザがシステムに対する適用範囲を理解できない

(3)　ユーザニーズを明確に記述できるドキュメント技法がない

(4)　要求定義の不明確さが潜在的バグを増大化させる

(5) 要求定義の正しさを評価する基準がない

(6) 要求定義のミスをテスト段階で検出してもリカバリーできない

(7) ユーザ不在のままの要求定義を行いやすい

(8) 要求定義のための技法がほとんど体系化されていない

(9) ユーザと開発者の専門分野が異なり対応が取りにくい

2 問題解決へのアプローチ

(1) 要求定義の検討期間は可能な限り長く取り、ライフサイクルの上流部の工を多く取っておく

(2) 開発者とユーザ間において、密接なコミュニケーションの実施を行う

(3) 適用用務に関する知識の蓄積とユーザの現場環境を把握する

(4) 一般に普及しているソフトウエア要求仕様技法を活用する

(5) 見やすくわかり易い要求仕様ドキュメントを作成する

(6) 要求定義の内容に対する品質検査には、ユーザと他のプロジェクトの開発者を交え、あいまいな部分がなくなるまで、内容が要求仕様通りになるまで行う

3節 ソフトウエア要求定義技法

ソフトウエア要求定義(software requirement definition)

米国：ソフトウエア要求仕様に関する技法、方法論の実用化が積極的に図られている。支援ツールやプロダクトとして普及しているものも多い
母国語である英語が仕様記述言語として利用しやすいことや、仕様記述言語と日常生活上での言語が同一であり記述表現上におけるギャップが少ない

ソフトウエアツールやプロダクト類をパッケージ化して流用することが習慣化している

日本：今だ適用化していない。現状では次の問題点がある

(1) 感じ機能のマンマシーンインターフェイスが弱く、操作が複雑であり日本語による要求仕様記述のツール化を妨げている

(2) ソフトウエアパッケージの流通機構が脆弱であり、ソフトウエアツールの効果的活用が十分できない環境にある

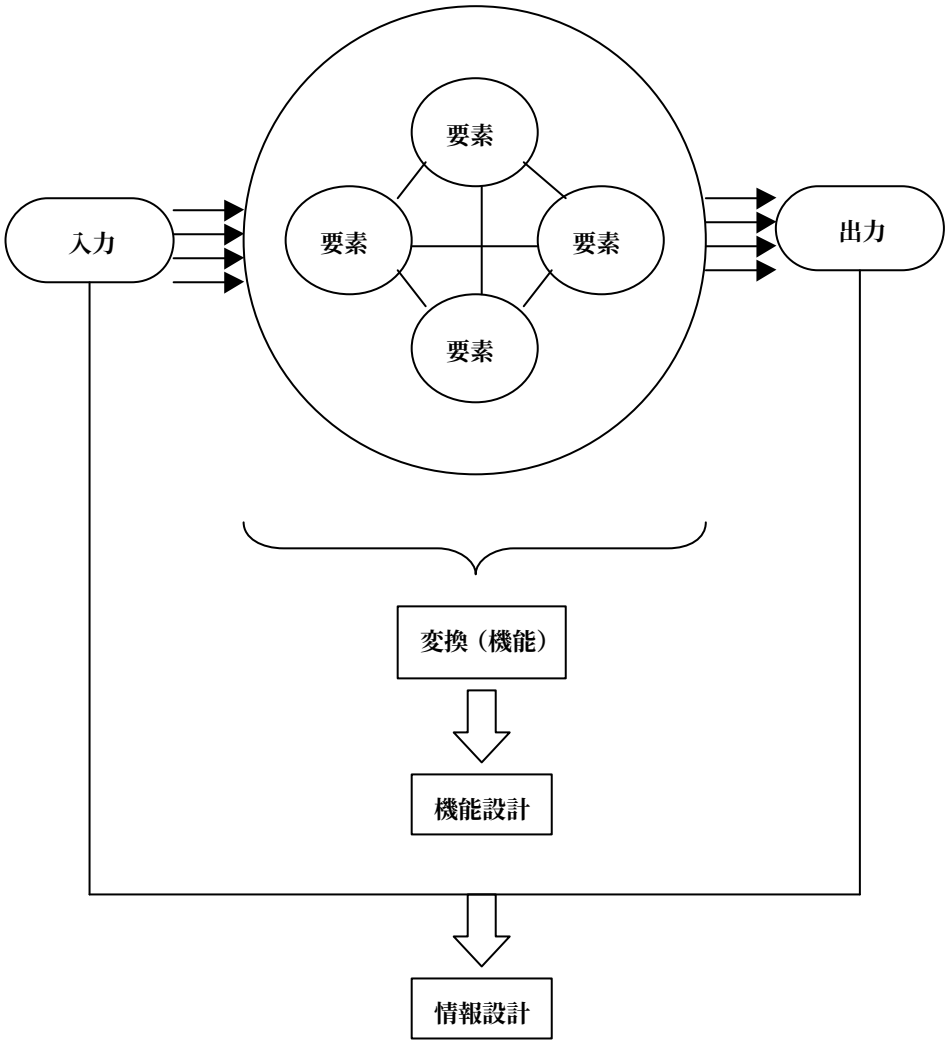


図 17

(3)　１つの要求定義技法がすべてのソフトウエア・アプリケーションタイプに適用できないことを理由に、現場ではツールを使いたがらない面もある

(4)　全般的に要求定義工程に関する研究、開発のレベルが低い。どうしても、より具体性のあるプログラミングやデバッグといったことに関心が集中する。この結果、ソフトウエア・ライフサイクルの下流部分におけるツール化は積極的に行われるが、上流部分である要求定義ではほとんど対応が取れていない

1　一般論

(1)　ソフトウエアシステムのモデル

(2)　ソフトウエアシステムの構成要素

(3)　要求定義の設計アプローチ

(4)　データ志向の設計技法か、プロセス志向の設計技法

2　構造化分析

DeMarco の構造化分析(structured analysis)

情報設計を前提とした要求定義の仕様技法

ユーザの要求定義仕様(requirement specification)

原始データを要求されている出力情報へと付加価値をつけて変換していく過程は、ソフトウエアシステムによるデータ処理の流れそのものである
どういう出力情報を要求しているかをもとに、原始データに対する変換手順を考えれば、それがソフトウエアシステムのための要求定義となる

構造化分析

データの流れを図式化することによって、情報の変換工程を解析しながら要求を分析したうえで要求定義を作成する技法(technique)である

(1) データフロー・ダイヤグラム

データフロー・ダイヤグラム(DFD：Data Flow Diagram)

① データの入力源と出力先

② データの流れ

③ データの処理

④ データの蓄積

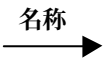
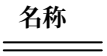
名称	図式
データの入力源 データの出力先	
データの流れ （データフロー）	
データの処理 （バブル）	
データの蓄積 （ファイル）	

図 18

DFD の書き方

① データの入力源とデータの出力先の設定

② コンテキスト・ダイヤグラムの作成

③ バブルの階層化

部分機能分割法(functional decomo-position)

④ DFD 作成の終了

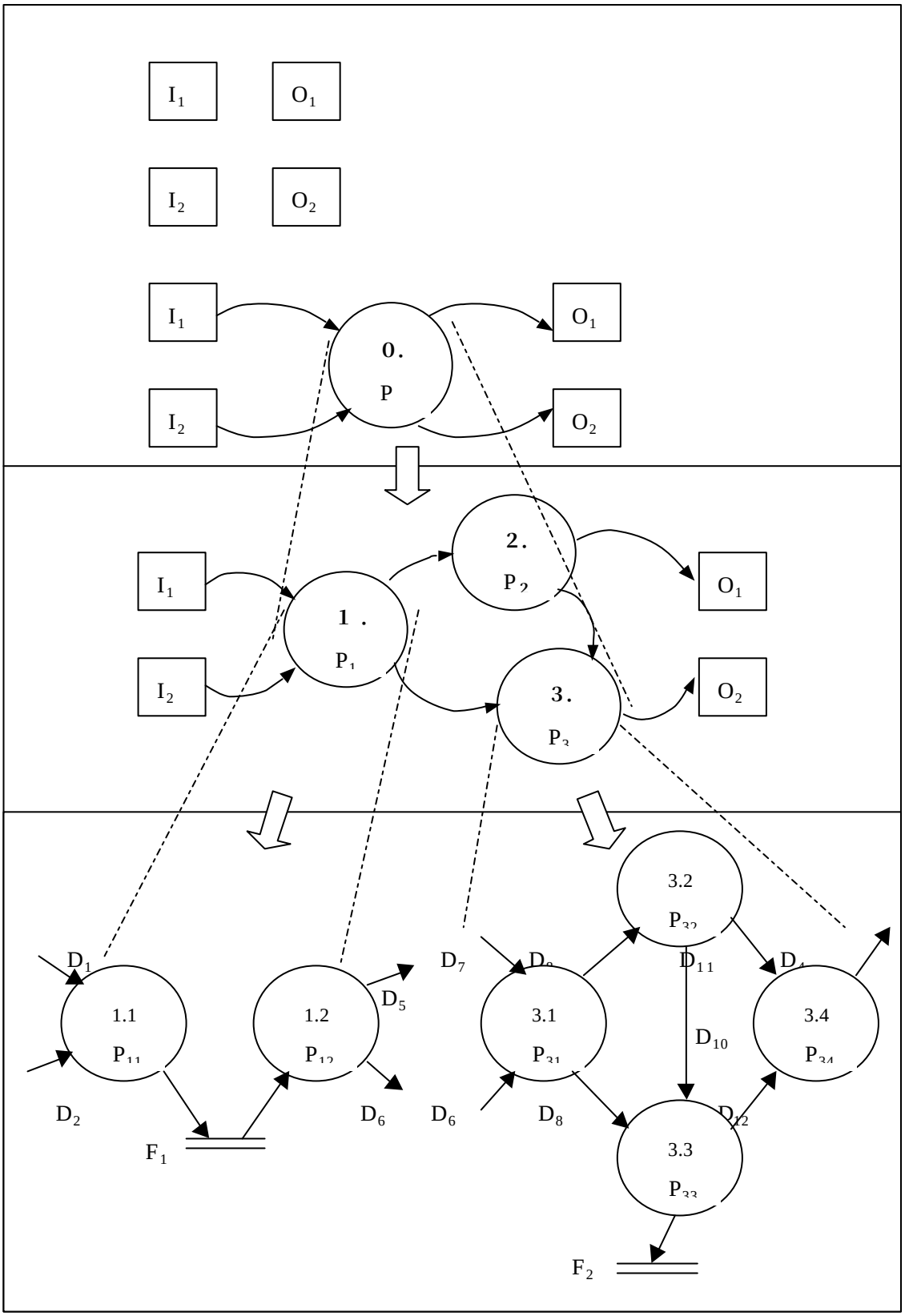

I1
O1
I2
O2
I1
0.
P
O1
I2
O2
2.
P2
I1
1.
P1
O1
I2
3.
P3
O2
3.2
P32
D1
D7
D11
D4
1.1
P11
1.2
P12
D5
3.1
P31
3.4
P34
D10
D2
D6
D6
D8
D12
3.3
P33
F1
F2

図 19

(2) データ・ディクショナリ

データ・ディクショナリ(data dictionary)

構造化定理

(3) ミニ・スペック

ミニ・スペック(mini-spec)

構造化言語(structured language)

デシジョン・テーブル(decision table)

デシジョン・ツリー(decision tree)

(4) 構造化分析の特徴

① ツールを用いて記述をする際の制約がなく、簡単に設計が行える

② 図式の種類が４つしかなく、それらを組みあわせるだけで仕様を作成することができる

③ ユーザにとっても理解しやすく、できあがった要求仕様の内容に対して検査がしやすいので、ユーザニーズがどれだけ正確に要求定義に反映されていることがわかり易い

④ 構造化の考え方がすべてのツールに一貫して適用されており、段階的詳細化によるトップダウンアプローチによる設計が進めやすい

⑤ データの流れとそれに対応した処理工程が、資格的に把握できる。要求定義を文書で書き下ろすよりも、直感的に理解することができ、誤解が少ない

⑥ データの流れは適用業務の処理形態を適切に表しており、実際の業務処理をモデル化しやすい。ただし、適用業務システムにおける物理的

な情報やデータ処理の設計については、対象外となっている。したがって、システムのコストパフォーマンス設計や性能評価分析などの設計には、この技法を適用できない

⑦ データフローに制御情報を用いることはできない。したがって、ここで取り上げた構造化技法は制御システムの要求定義に用いることはできず、事務処理分野の適用業務処理システムが中心となる

⑧ 構造化分析で用いる各ツールを机上で使う場合、作成した要求定義仕様書について、必ずレビューを実施したほうがよい。レビューは設計者同士でなく、ユーザも交えて行うこと

(5) サンプル例

適用業務処理「成績管理処理」

成績評価の登録・訂正処理および検索処理をシステム化することを、構造化分析技法を用いて要求定義仕様を作成する

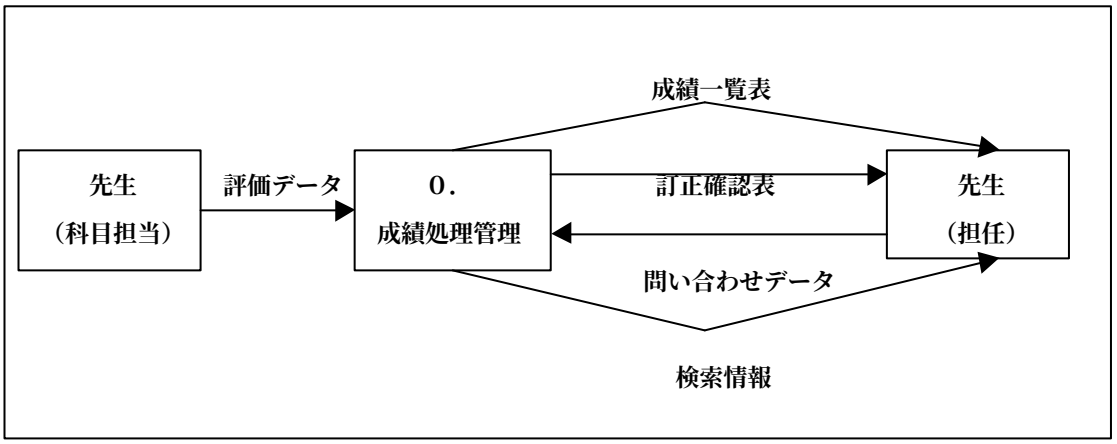


図 20

3 良質な要求定義仕様

(1) ユーザ自身が、記述された要求仕様内容を把握しやすい

(2) あいまいさが少なく、ある程度定型化された記述形式である

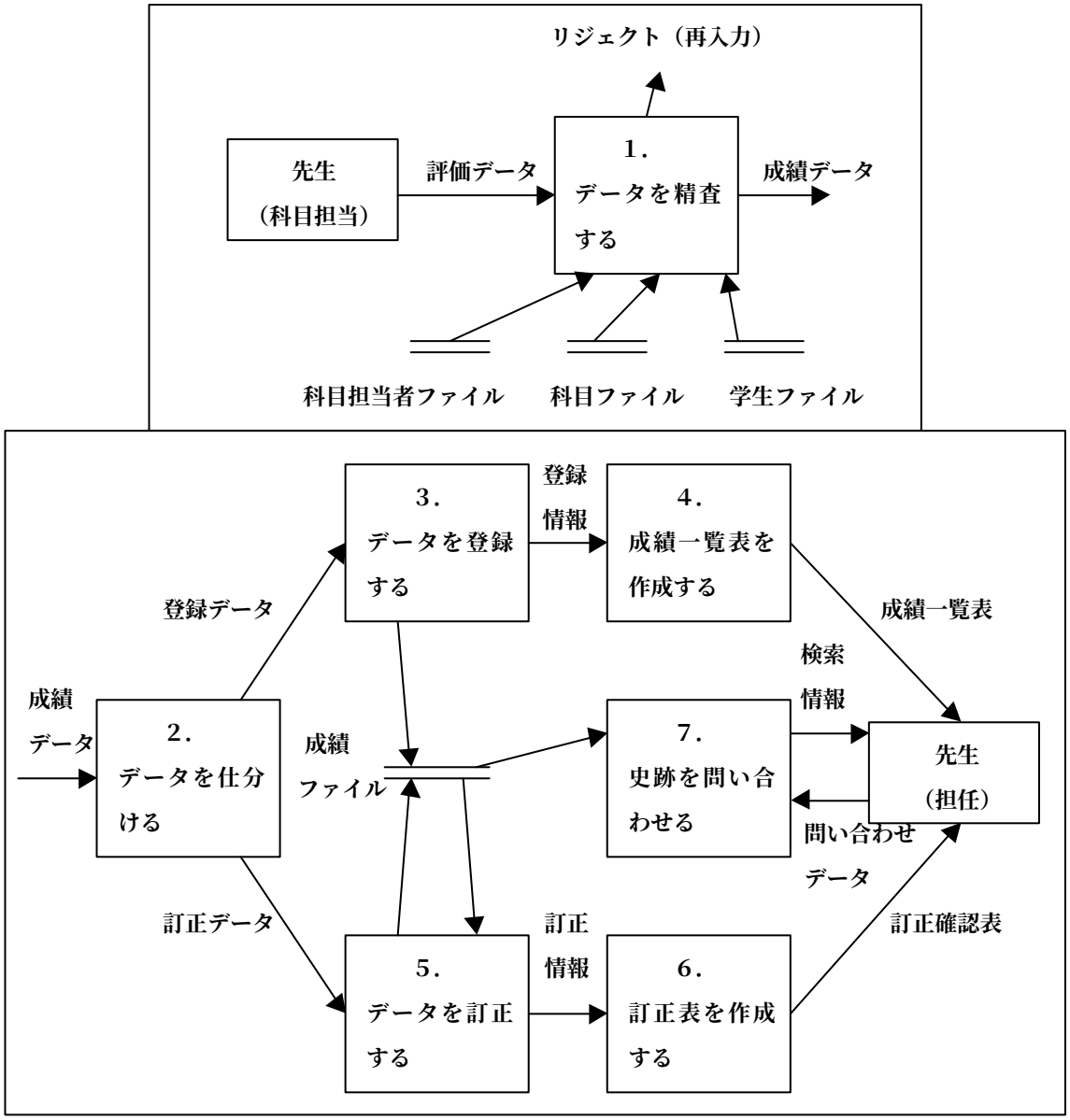


図 21

(3) 記述方式にあまり制約がなく、簡便に作成できる

(4) 段階的に詳細化できる記述形態をとっている

(5) ライフサイクル上の次工程であるシステム設計やプログラム設計とインターフェイスがとりやすく、考え方に一貫性が保てる

(6) 修正がしやすく他への影響が少ない

(7) 検査基準が設定できる構成である

第二章　ソフトウエア作成の方法

第４課　システム設計技法（２コマ）

１節　システム設計とは

要求定義で明らかになった仕様を EDPS としてシステム化するために、ハードウエア構造とソフトウエア構造を設計する

１　システム設計上のポイント

(1) システム全体から着目したデータの流れをつかむ。

(2) システムを構成するハードウエアの最適化をはかり、構成を決定する

(3) システムが持つべきデータ変換の工程、すなわち、ソフトウエアに要求されている機能をどのように構成するかを考慮する

(4) ソフトウエアをどのようなプログラム構成として具体化するかについて節計する

(5) ソフトウエアシステムが持つ機能を、階層的個々の要素単位に分割し、データの流れに着目しながら時系列的に要素単位を決定していく。個々の要素単位がプログラムに対応する。この段階でソフトウエアシステムを構成するプログラムの構造が決定する。

(6) プログラムの構成が大規模すぎたり、複数の機能を包括している場合、さらにいくつかのプログラム（モジュール）へ分割する必要がある。。プログラム分割をどうするかがシステム設計の基準となる

(7) プログラムが取り扱う入出力データ及び共有ファイルの構造をどうするか

考慮する。各プログラム単位がアクセスするファイルの最適化された処理形態が実現されるように設計を考慮する。プログラムが最も効率よくアクセスできるファイル構造にする。

(8) 設計を行うに当たって、どのような技法や方法論を適用すべきか考慮する

2 システム設計のいろいろな側面

システム設計の工程

(1) ハードウエアシステムの設計

(2) ソフトウエアシステムの設計

＊ ファイル設計
ソフトウエアシステムがファイルに対してどのようにアクセスするかで、ファイルの設計内容（編成や形式、アクセス手順など）は決定されてしまう。

＊ システム機能設計
ソフトウエアの機能単位をプログラムとして識別し、プログラムの全体構成を設計するアプローチをとる。

2節 システム設計技法

ソフトウエアのシステム機能を具体的なプログラムへと詳細化していく工程での効果的な設計技法の解説である。基本的には、モジュラ里ティの概念をベースとしたソフトウエアの概念構造における設計である。

この考え方は、大きくて複雑な機能を持つシステムを分割しながら、より小さく独立性の高い機能体（プログラムもしくはモジュール）に解きほぐしてしまうという発想にもとづいている。

実例

Constantine の構造化設計(Structured Design)
Mayers の複合設計(Composite Design)

構造化設計

要求定義仕様をコンピュータシステムとして実現するために、プログラムあるいはモジュール群の構成に具体化していくための設計手順を示したものである。

構造化設計という通りに、段階的詳細化にもとづくアプローチにより、ある１つの要求しよう機能を階層的にプログラムを分割することが出来るようになっている。

プログラムを分割したモノが**モジュール**であり、モジュール同士は**引き数**によって関連づけられている。

構造化設計は構造化分析との親和性が高いことが特徴である。

構造化分析により作成した DFD の各バブルを要求定義仕様の１つと見なすと、バブルからプログラムの構造を導き出すことが出来る。

構造化設計には、設計を支援するためのツールとして、***バブルチャート***と**階層構造図**が用意されている。

設計技法としては、データ処理手順の違いから **STS 分割**と **TR 分割**の２種類がある。

１　モジュールと引き数

モジュール(module)の定義

(1)　１つのまとまった機能(function)を実現する命令(statement)群の集まりである

(2)　独立に翻訳(compile)できる

(3)　翻訳されたモジュールは、他のモジュールによって呼び出されることがある

(4)　モジュール間のインターフェイスは引き数(argument)を用いる

モジュールの構成

データ部

呼び出したモジュールから受け取るデータである引き数（入力用）と、呼び出したモジュールへ戻すデータである引き数（出力用）と、自分自

身が使う作業用データが定義される。

プロセス部

入力された引き数を、出力する引き数へ変換するための、処理手順が命令として記述される。

モジュール間の関連

通常は、CALL 命令によって定義される。その際、モジュール間のデータ受け渡しを引き数として記述するようになっている。

引き数には、単純なデータのものと、データに関する情報を標識として持つフラグがある。

フラグには、データの妥当性評価や処理結果の状態などといったものも含まれる。

モジュール間の実行制御関係

(1) 上位モジュール内で、他のモジュールへの呼び出し命令(CALL 文)が実行されると、プログラムの実行は下位モジュールへ移る

(2) 下位モジュールは、上位モジュールから処理すべきデータ（入力用引き数）を受け取り実行を始める。この間上位モジュールは停止状態のままとなる

(3) 下位モジュールは実行を続けながら上位モジュールへ戻すためのデータ（出力用引き数）を用意する。全ての処理が終わり境界識別子(END 文)に達すると、プログラムの制御が上位モジュールへ移る

(4) 上位モジュールは、CALL 文の次の命令からふたたび実行を始める。このときに、下位モジュールから戻された引き数を参照して処理を実行することが出来る。

2　バブルチャートと階層構造図

(1) バブルチャート(bubble chart)

バブルチャートはデータ処理の流れを示した図表である。

バブルチャートの表記法は図＊－＊である。

図 22

(2) 階層構造図(hierarchical syructured chart)

モジュール同士の従属関係を階層的に表現したものである。

階層構造図の表記法は図＊－＊である。

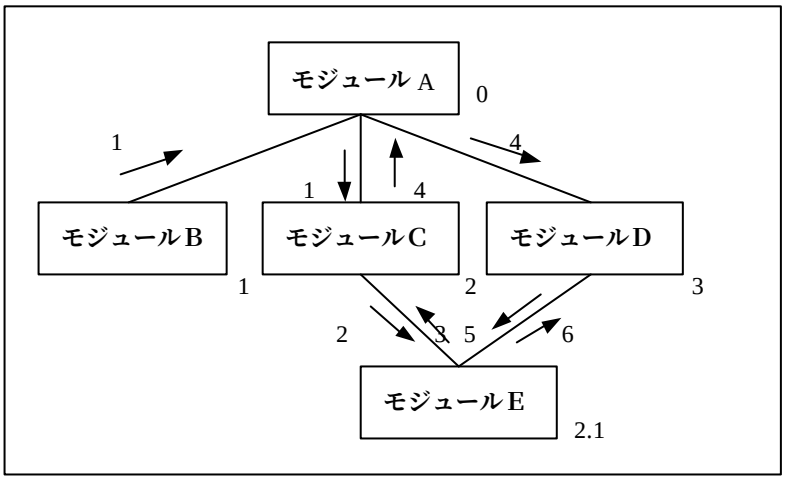


図 23

階層構造図の規約

① モジュールの呼び出し順序は関係ない。慣例として左から右へ想定する

② 同じモジュールが呼び出される場合は、それらを一カ所にまとめる

③ モジュールの上下関係について、呼び出す回数などは表記する必要はない

④ モジュールの名称は機能を表すような表現「～を・・・する」であること

⑤ モジュール間のインターフェイスを示す引き数を設定するとき、モジュール同士の制御についても考慮しなければならない

⑥ 引き数の名前は、呼び出す側と呼び出される側で同一でも異なってよい

⑦ 引き数の矢印記号について、白抜きの丸印はデータそのものを表し、黒塗りの丸印は標識としてのフラグを表す

⑧ 引き数の矢印は、引き渡す方向を表している

3　STS 分割と TR 分割

(1)　STS 分割技法

データの流れに着目し、各データ要素を変換していく機能を
S(souce：源泉)　/　T(transform：変換)　/　S(sink：吸収)
という区分によりモジュール分割していく技法である

源泉の部分は入力モジュールに相当し、**変換**の部分は処理モジュールに相当する。**吸収**の部分は、出力モジュールに相当する。そして、これらのモジュール全体を制御するための上位モジュールによって、それぞれのモジュール群が呼び出される関係となる。

モジュール分割の手順

① 対象となる事象の問題構造をいくつかの部分機能の集まりと見なして、個々の機能を識別する。

図＊－＊

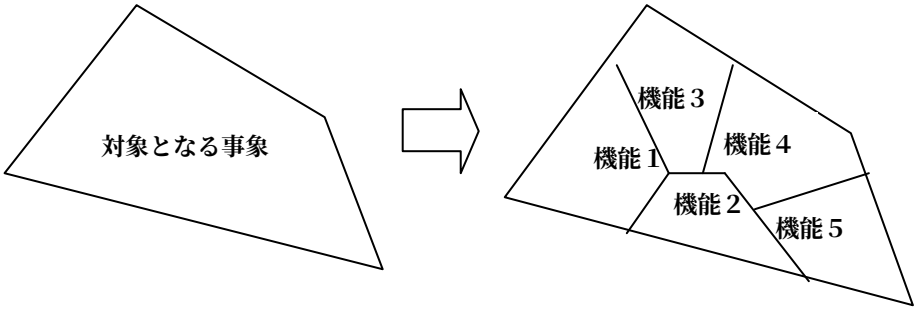


図 24

② 個々の機能間に置いて、入力源から出力先へ結びつく主要なデータの流れを見つけだす。

図＊－＊

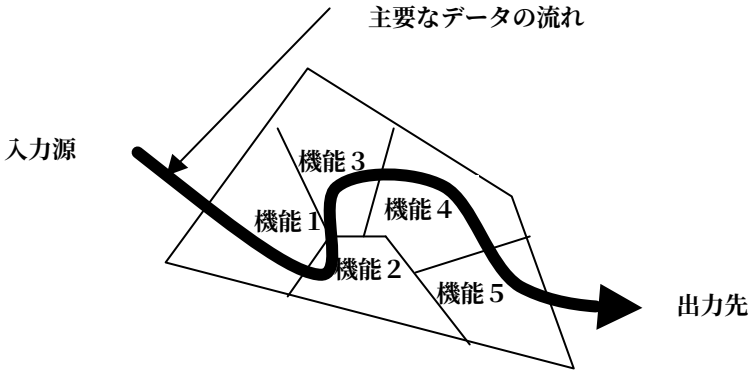


図 25

③ 主要なデータの流れを各機能をバブルに置き換え、バブル間のデータを明確にする。図＊－＊

④ 主要なデータの流れに着目し、最大抽象入力点と最大抽象出力点の位置を決定する。図＊－＊

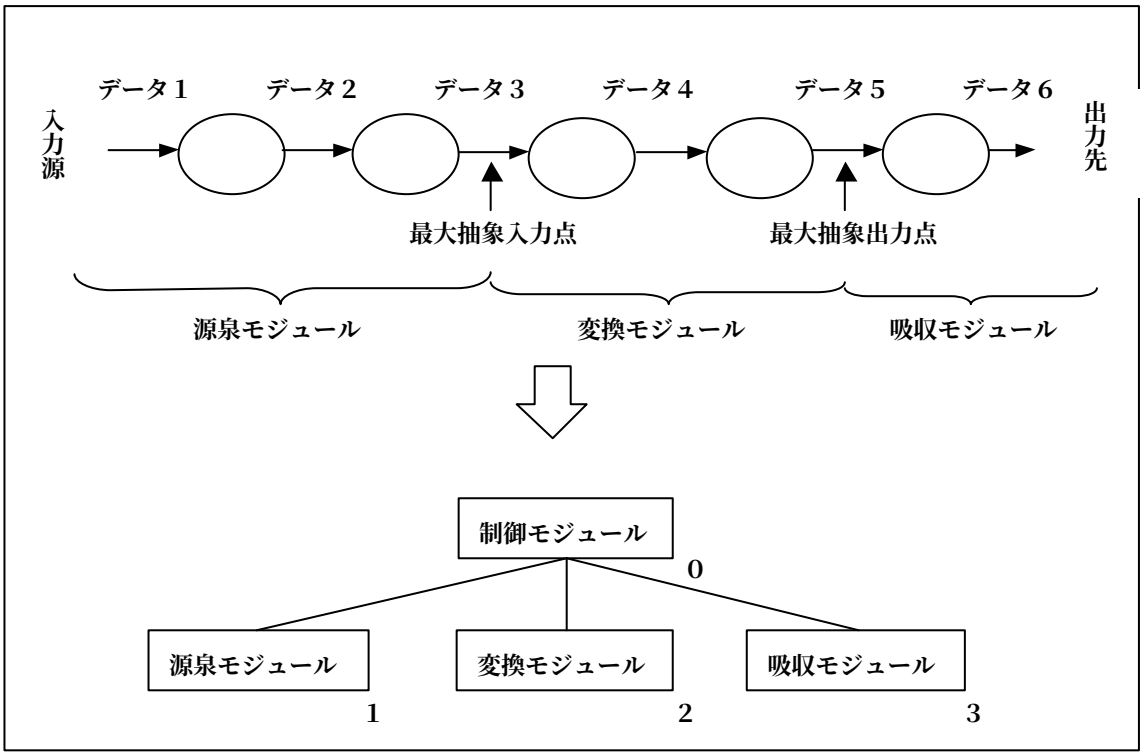


図 26

⑤ 最大抽象入力点と最大抽象出力点を基準に、主要なデータの流れを３つの部分に分解する。

３つの部分とは、入力源から最大抽象入力点までの源泉部（S)、最大抽象入力点から最大抽象出力点までの変換部（T)、最大抽象出力点から出力先までの吸収部（S）であり、２つの抽象点により３分割するので、STS 分割と呼ばれている。

分割されたそれぞれの部分が各モジュールに相当する。

全体を制御するためのモジュールを上位に関連づけて、モジュールの階層構造図を作成する。

⑥ モジュール同士のすべての従属関係について、引き数を決定してそれらを階層構造図に追加する。このとき、バブルチャートの各データの

流れを参照する。それらのデータがモジュール間の引き数に相当する。

図＊－＊

⑦　源泉モジュール、変換モジュール、吸収モジュールそれぞれについて、上記の処理を繰り返す。

⑧　モジュール分割の終了条件としては、最下位レベルにおけるモジュールの論理構造であるアルゴリズムが直感的に把握できるまでとする。こうして、全モジュールの階層構造図と引き数の関係が完成する。

図＊－＊

(2) TR 分割技法

対象となる事象の問題構造を部分機能に分割する際に、主要なデータの流れが存在せずに、いくつかの分岐が発生するような場合に用いるモジュールの分割技法である。入力源から発生する入力データの持つ意味属性によって、変換の処理内容が異なる場合である。それぞれの入力データ（**トランザクション**）ごとに、モジュールを分割してしまう方法をとる。

物流管理システムにおける TR 分割の例

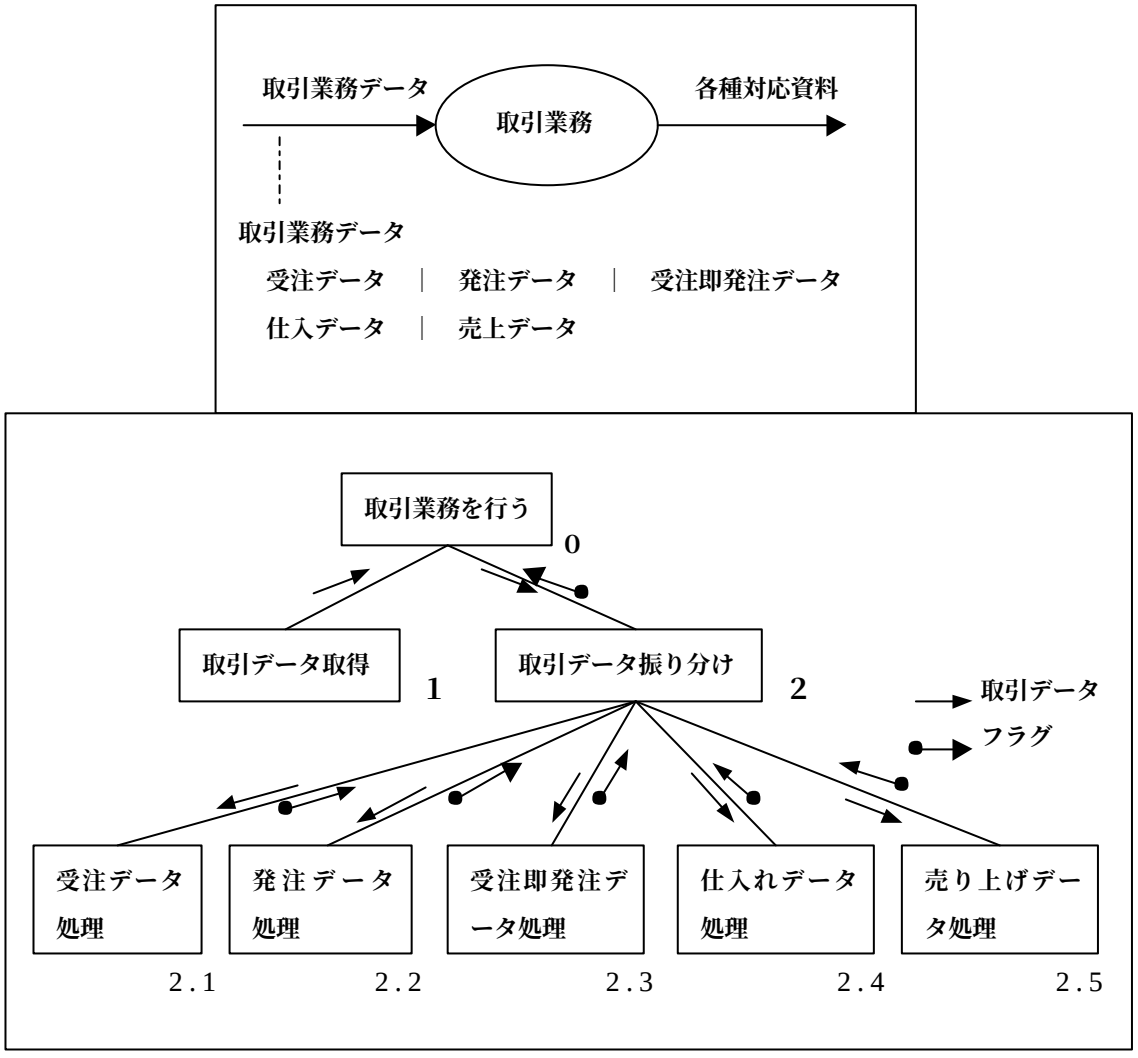


図 27

業者から依頼された取引業務データにいくつかの種別が存在し、その種別によって処理が異なるとする。こういった場合は、取引業務データの種別を識別して、対応するモジュールごとに振り分けるといった構造にすればよい。

こうして、分割したそれぞれのモジュールについて、さらに STS 分割を用いて設計を進めればよい。

TR 分割によるモジュール構造図は＊－＊である。

分割を行う場合の留意点

① 設計する人によりモジュール分割の結果が異なり、同一のモジュール構造にならない場合がある。これは、個々人の設計に関する技量やセンス、経験に依存する面が多いからである。このため、数名の人々により設計結果をレビューする必要がある

② モジュール分割を行った場合に、１モジュールの規模（ステップ数）はすくない方がプログラミングやデバッグが容易となり、モジュールの機能の独立性も保証されることになる。ただし、あまりに分割しすぎるとモジュール間のインターフェイスが煩雑となりやすい

③ １モジュールの大きさをステップ数で制限してもよい。

④ 個々のモジュールが参照したりアクセスするデータは、極力特定のモジュールに押さえなければならない。モジュールごとの情報の隠蔽化をはかり、不必要な情報（データやプロセス）は他のモジュールからアクセスしないように、情報の局所化をはかった方がよい

4　構造化分析との対応

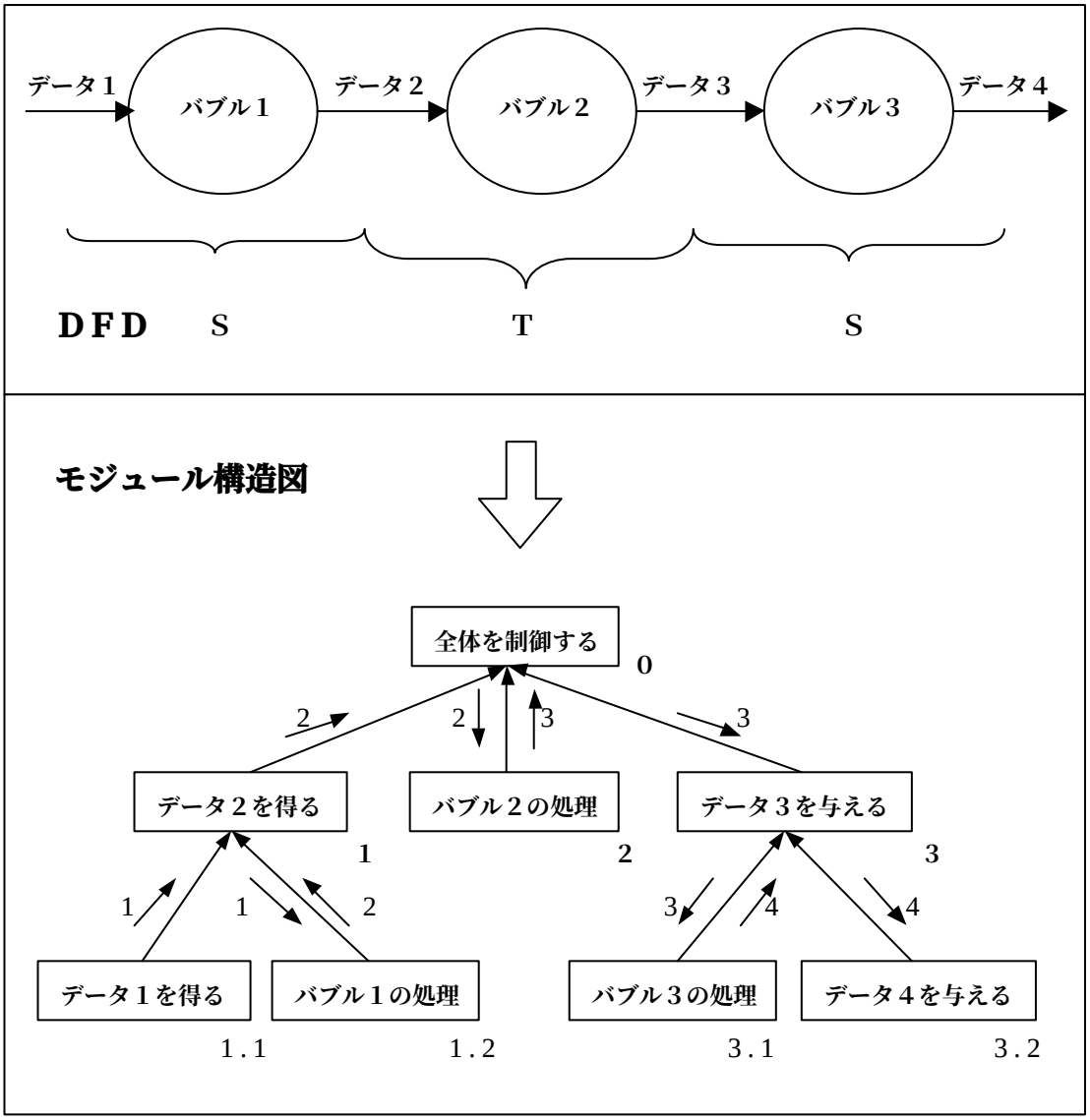
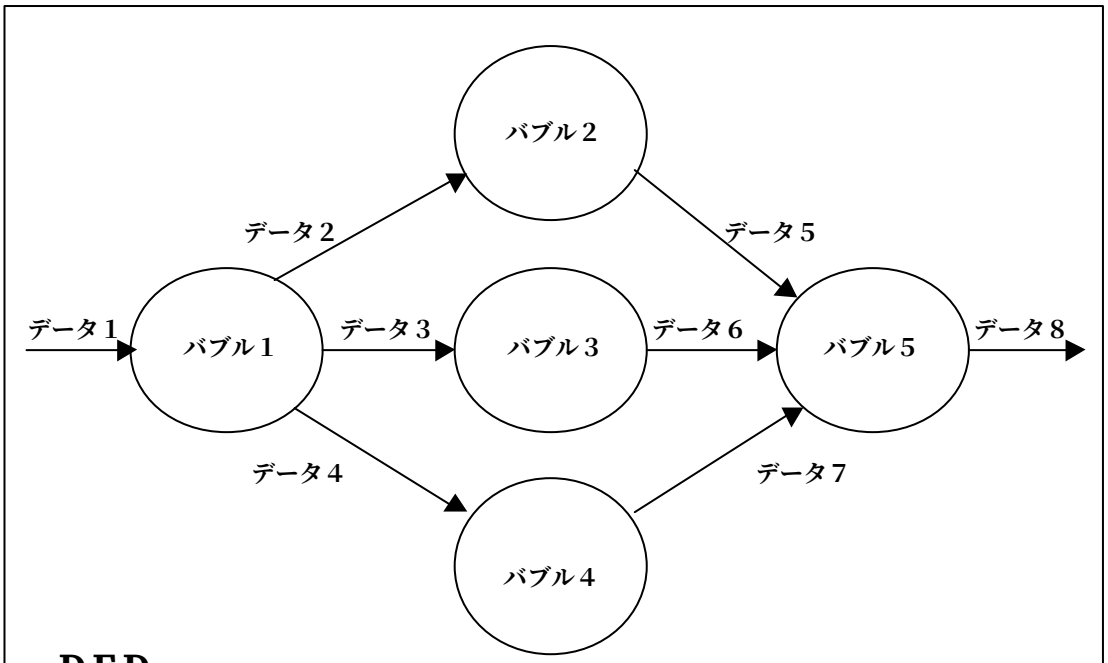


図 28

5　モジュール分割基準



ＤＦＤ

モジュール構造図

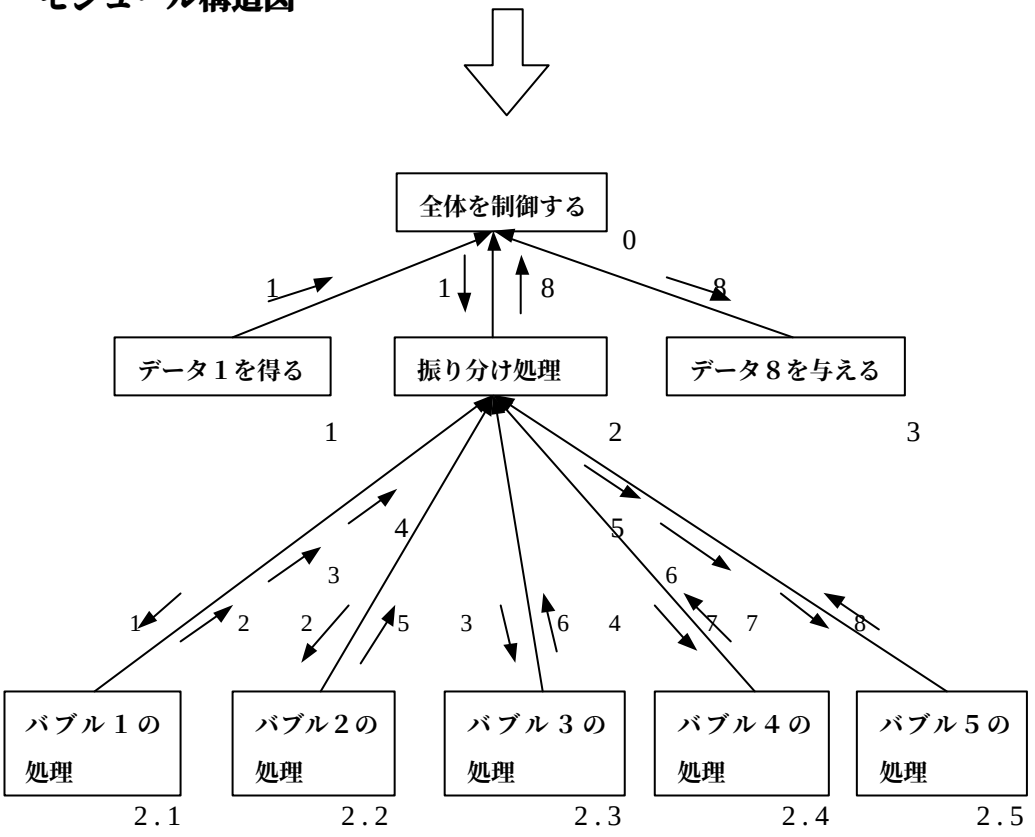


図 29

(1) モジュールの独立性

① モジュール結合度

② モジュール強度

(2) モジュールの制御領域と影響領域

① 最適化

② 制御領域

③ 影響領域

6 特徴

(1) 構造化分析との親和性が高く、技法同士の整合性が容易。段階的詳細化にもとづくウオーターフォール型ライフサイクルモデルに適用しやすい

(2) ファイルのアクセスが中心となるバッチ系のプログラム設計には適していない。バッチ系プログラムはモジュールを分割せずに、複数のプログラムをジョブ制御言語(JCL：job control language)で連結してストリームするかたちとなる。そして、プログラム同士はファイルによってデータを受け渡しする方式をとる。このため、プロセスチャートの作成により設計を進める

(3) システムの複雑さを解決するための方策として、分割・統合の概念のうち、部分に分割するという考え方が導入されている。

(4) モジュール分割の工程には、トップダウンアプローチにもとづく考え方も導入されている。あるレベルのモジュールに対して、段階的に次のレベル

の下位モジュールへと分割をはかる。これにより、階層構造図が順次作成されていくことになる。設計を進めていくことによって、仕様書も同時に作成されることになる

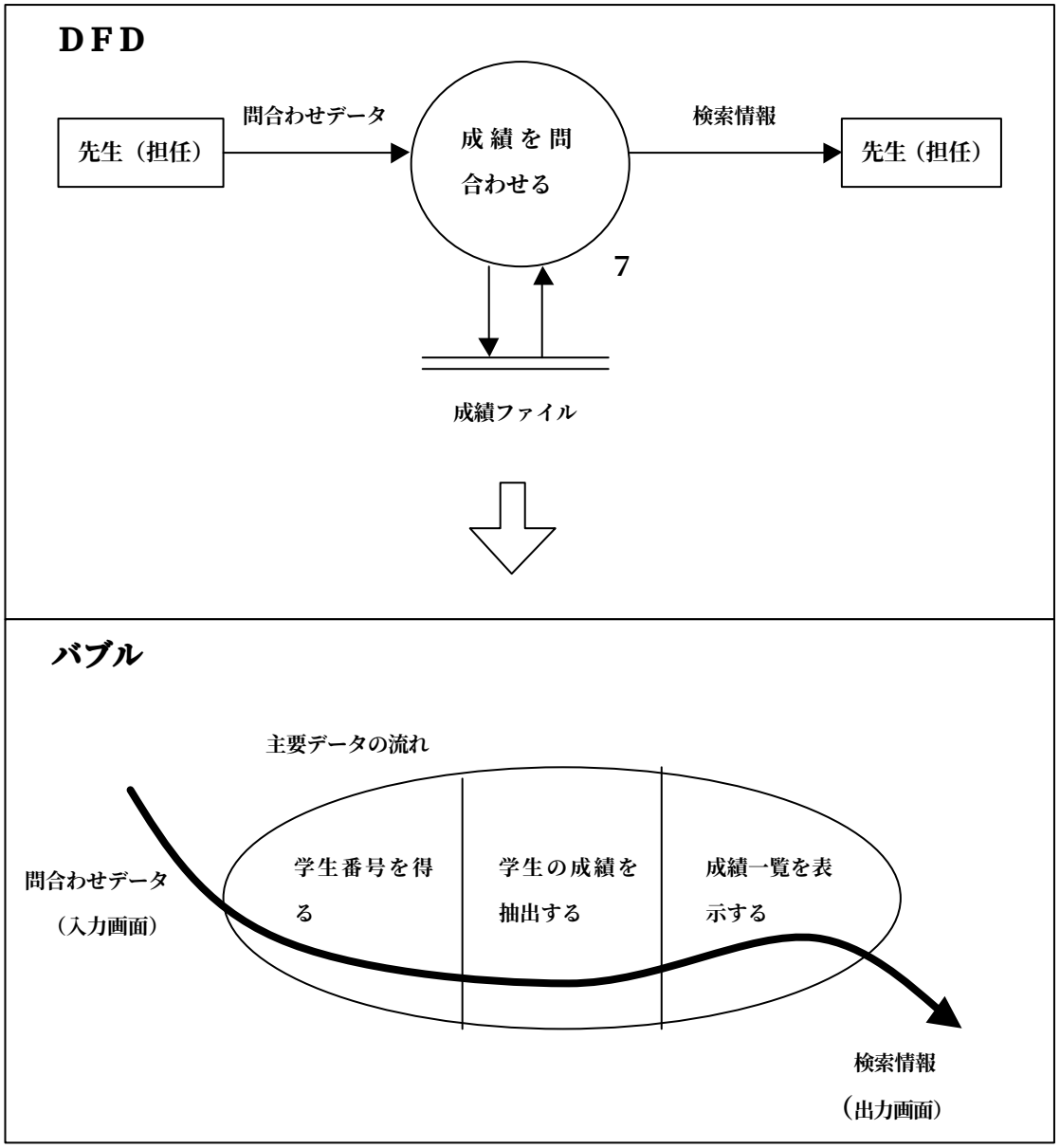


図 30

(5) 効果的プログラミング技法（IPT：improved programming technologies）の各技法と併用すると、より効果が上がる

ドキュメント　　HIPO(hierarchy plus input output)

NS チャート(Nassi-Shneiderman chart)

プログラミング開発

トップダウンプログラミング(top down programming)

プログラム管理　チーフプログラマチーム(chif programmer team)

検査機構　ウオークスルー(walk through)

インスペクション(inspection)

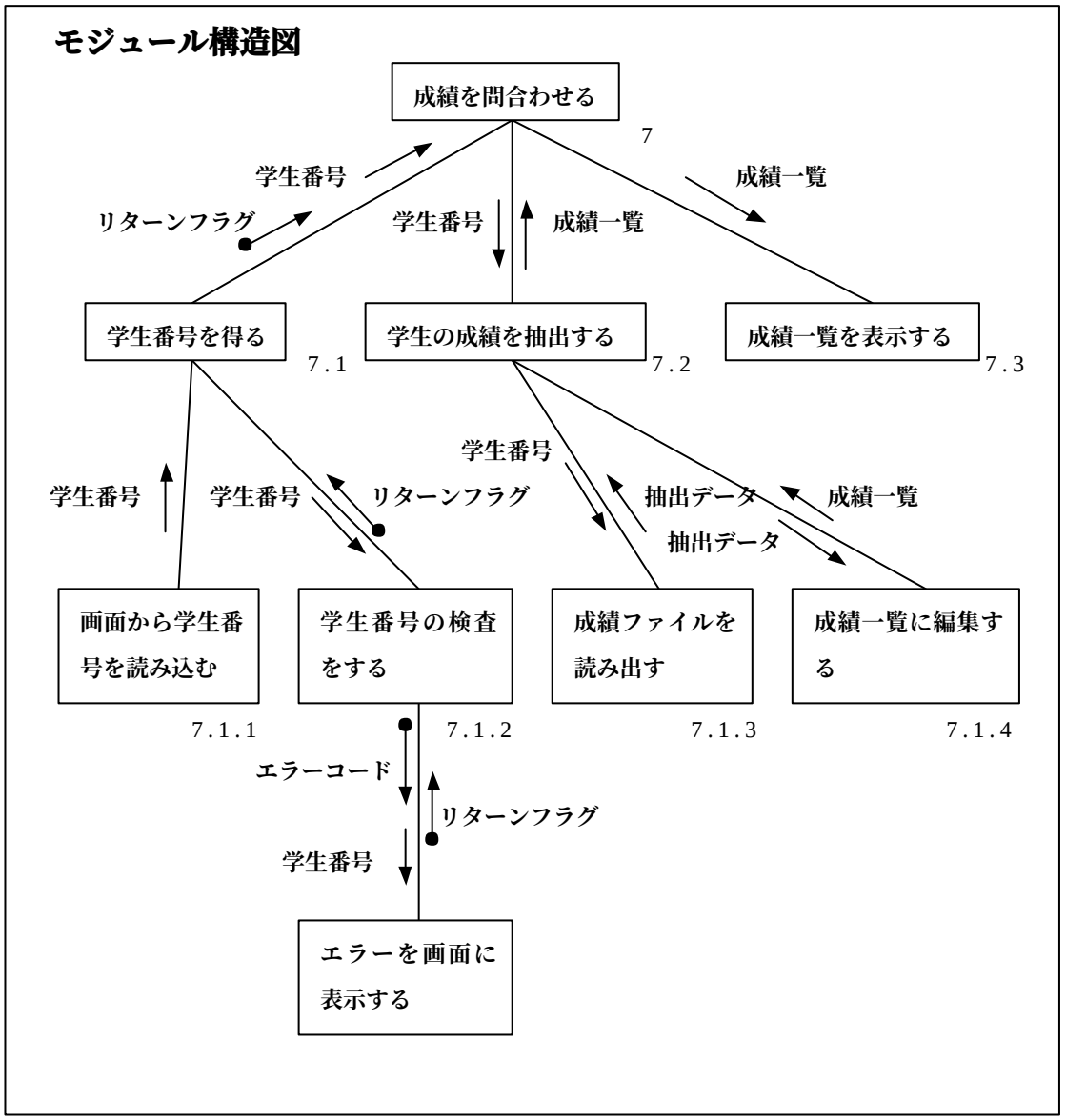


図 31

プログラム設計　構造化設計

7　サンプル例

(1)　バブル構成図

(2)　モジュール構造図

第５課　プログラム設計技法（３コマ）

１節　プログラム設計とは

システム設計の工程で明らかになったソフトウエアの外部構造をもとに、ソフトウエアの内部構造を詳細に設計すること

プログラムもしくはモジュール単位の機能とインターフェイス部分（プログラムの場合は入出力ファイル、モジュールの場合は入出力アーギュメント）をもとに、個々のプログラムもしくはモジュールの内部処理手順を設計すること

内部処理手順とは、アルゴリズム(algorithm)とかロジック(logic)と呼ばれるもので、プログラムやモジュールに与えられている機能仕様をどのような処理手順で実現すべきかを、処理するデータの流れに沿って記述したものである。記述された１文書をプログラム言語の数ステートメントに対応させる場合と、１ステートメントにまでに対応させる場合がある

プログラム設計の重要性は、ソフトウエアシステムの品質を設計段階から作りこむところにある。システム設計工程でプログラムもしくはモジュールに対する機能が明確に定義づけられたとするならば、すぐプログラミング作業に入ったほうがプログラムの製造工数が少なくてすむが、これは、プログラム設計に費やされる工数を、完全に削減できることを示す

しかし、システム設計の内容をそのままプログラミングしてしまった場合、プログラムの信頼性が低下する。

設計仕様書もないままプログラミングを進めていくと、プログラムのアルゴリズムが非手順的となり、制御の見にくいものとなりやすく、結果として煩雑なプログラムロジックが構築され、デバッグにも時間がかかり全体的なプログラム作成工数は増大化する。また、潜在的なバグも発見されないまま、

信頼性の低い製品として開発者の手から離れていく

2節　プログラム設計技法

プログラム設計技法の考慮点

(1)　プログラミングするための前提条件となる詳細設計の工程として位置付けられるもので、あいまいさ(fuzzy)がない状態でプログラムもしくはモジュールの内部構造を正しく設計できること

(2)　プログラム言語と対応がとりやすい設計記述となっていて、指定されたプログラム言語の特徴と整合性のとりやすい記述表現ができること

(3)　プログラムもしくはモジュールに要求されている機能仕様を、正しく具現化できる設計上の方法論を包括していること

(4)　ある一つの手順のもと、標準化しやすい設計工程となっていること

プログラム設計技法のアプローチ

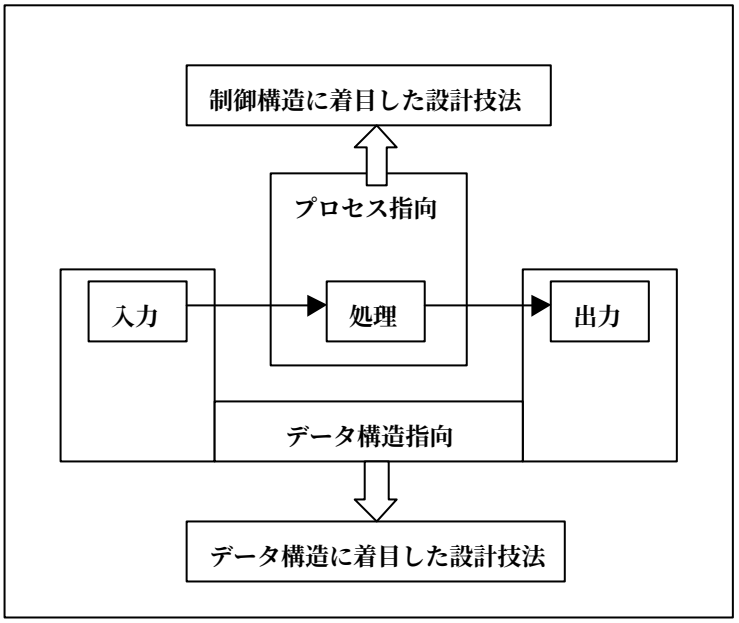


図 32

制御構造に着目した設計技法

入力、処理、出力という３つの部分からなるシステムモデルのうち、中央の処理、すなわちプロセスの制御からプログラムの設計を志向するもの
構造化プログラミング
構造化チャート

データ構造に着目した設計技法

両端の入出力データの定義からプログラムの設計を志向するもの
ジャクソン法
ワーニエ法

構造化プログラミング(Strucured Programming：by Dijkstra)

ジャクソン法(Jackson Structured Programming：by Jackson)

ワーニエ法(Warnier Method：byWarnier)

１　構造化プログラミング

(1)　構造化プログラミングの基幹概念

(2)　アルゴリズム設計上の利点

① プログラムの制御構造が、１つの入口点と出口点で構成される閉じた処理系となる。したがって、モジュールを構成する機能要素単位の独立性が高まる

② プログラムの制御構造が上から下へ逐次流れることになり、わかり易いプログラムとなる。ただし、繰り返しの制御構造では下から上への矢印の戻りがあるが、繰り返しの制御パターンと考える。プログラム言語の GO TO 文には相当しない。

③ プログラムのアルゴリズム構造（机上での静的手順による処理系）がプログラムの実行動作（マシンでの動的環境）と対応がつきやすい

④ 複雑な処理手順は、制御構造の階層化を図ることによって容易に解決できる

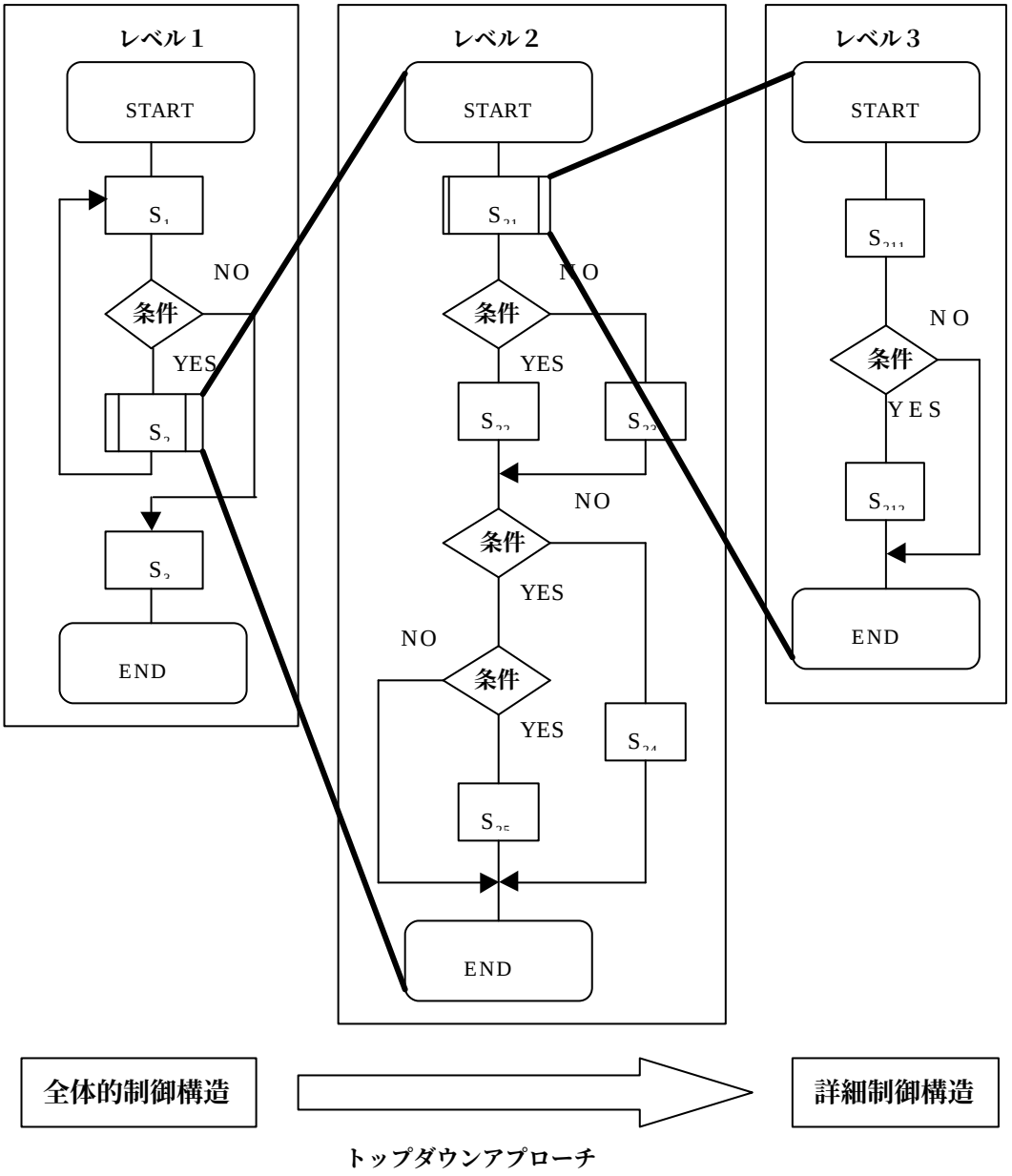


図 33

(3) GO TO 論争

論争の経過

今日の状況

制限つきの GO TO 文を一部認めた上で、構造化のための制御構造を用いながら、プログラムを構造的に設計し、見やすくわかり易いアルゴリズムを構築していくという考え方が広く普及しつつある

(4) プログラミング設計上の留意点

① 机上の方法論であるためプログラム設計者の経験に依存する

② プログラム設計工程の標準化を推進する必要がある。具体的には、プログラムの設計仕様書（詳細設計仕様書）として、ドキュメントの記述における標準化を実施する

③ アルゴリズムの制御構造を設計するときに、構造化という規約が生じることで、プログラム設計者個々人の技量を発揮することが少なくなる。技巧的に凝ったロジックを作成することが許されないがゆえに、パターン化したルーチンワークとなり、モチベーションの低下を招くこともある

2 構造化チャート

プログラム設計工程で用いられる図式（chart）

構造化プログラミングとフローチャート

構造化チャートの例

IBM の IPT における NS チャート(Nassi-Shneideerman chart)

英国の標準である **DSD**(Design Structured Diagram)

日立製作所の **PAD**(Problem Analysis Diagram)

NTT の **HCP**(Hierarchical abd ComPact description chart)

日本電気の **SPD**(Structured Programming Diagram)

富士通の **YAC**(Yet Another Control chart)

プログラム言語の開発と周辺機器

(1) NS チャート

NS とは Nassi-Shneideerman chart の略である
lPT の文書化の一技法である。プログラム設計における論理構造の表現に有効であり、NS チャート用いることによりプログラムの制御構造は階層的に構成される。

NS チャートの図式記法

サンプル例

特徴

① 完全なプログラムの構造化表現が可能となる図式表現の記述には GO TO 文に対応するものがまったくなく、GO TO 文のないアルゴリズム構造が構築される

② 図式表現であるため、他のプログラム記述言語のような文章表現に依存したドキュメントより、見やすくわかり易いプログラム仕様書となる

③ BOX の大きさを適切にして、全体構造がわかりやすくなるようにする

④ 条件文や制御構造の階層が深くなりすぎると書きづらくなる

⑤ チャートの大きさが固定化されてしまうため、一部の変更、修正が難しい。図の配置を変更しなければならず、大幅な書き換えが必要となる。保守工程にはほとんど使用できない。

(2) PAD チャート

PAD とは Problem Analysis Diagram の略である
問題分析図と呼ばれ、木構造チャートの一つである

PAD チャートの図式記法

サンプル例

特徴

① ストラクチャード。プログラミングを図式表現で記述できるため、見やすくわかり易いプログラム仕様書となる

② プログラムの制御構造が、基本線を元に上から下、左から右へと展開されるため見やすくなる

③ プログラムのアルゴリズム構造を設計するだけでなく、プログラミングテストにも一つの方法論を提示している。
TREE WALK
プログラムの制御構造だけでなく、データ構造に対しても PAD チャートを用いて記述することができる。データ構造とプログラム構造との対応が取れることになるため、構造を主体とした設計に適用しやすい

④ 基本線をもとに図が展開されており、配置を自由に設定することができる。すなわち、図の表現に融通性があることから、変更、修正がしやすい。したがって、プログラムの製造工程から、検査、保守工程にも使用できる

(3) HCP チャート

HCP とは Hierarchical abd ComPact description chart の略である

木構造チャートの一種である

HCP チャートの図式記法

サンプル例

特徴

① ストラクチャード・プログラミングを図式表現で記述できるため、見やすくわかり易いプログラム仕様書となる

② プログラムの機能要素に対応させながら処理手順を記述することができ、プログラムの WHAT と HOW をともに把握することができる

③ 入出力データと、個々の機能処理との関連が、矢印によって明確となる。ただし、データの流れを表す矢印とプログラムの制御構造を表す矢印が重複して見えることがあり、図全体が見にくくなることがある

④ データの表し方については次のような図を用いることでわかりやすくなる

3 ジャクソン法

(1) 設計手順

① データ構造設計

② プログラム構造設計

③ 手続き設計

④ プログラム仕様設計

(2) 構造一致の場合の設計手順

① データ構造設計

② プログラム構造設計

③ 手続き設計

④ プログラム仕様設計

(3) 構造不一致の場合の設計手順

ジャクソン法の特徴

(1) データ構造の設計を間違えると、適切なプログラム構造が見出せなくなる場合がある。したがって、プログラム構造設計時にデータ構造の正しさをフィードバックする必要がある

(2) データ構造がきちんと設計されていれば、以降の設計工程は誰が行っても同一のものとなる。プログラム構造はデータ構造に依存するという原理が伺える

(3) ジャクソン法にもとづいてプログラムを設計すれば、プログラムの制御構造は自然と構造化プログラミングの構成となる

(4) ジャクソン法は、あくまでもプログラム単位の論理的内部構造の設計技法であり、モジュール分割の手法としてとらえるべきでない。プログラム構造における基本構成要素は、モジュール単位としてではなく、同一プログラム内のステートメント群の集合体としたほうがよい

(5) 事務処理分野におけるプログラム設計技法として、最も効果的に適用できる方法論である。通常、事務処理系プログラムは、処理形態が決まっているものが多い。したがって、プログラム構造を適用処理にあわせてパターン化しておくと、標準化がとりやすくなる

4 ワーニエ法

(1) ワーニエ法の設計手順

(2)　設計手順

①　出力データ構造の設計

②　入力データ構造の設計

③　プログラム構造図の設計

④　構造的表現にもとづくフローチャートの設計

⑤　手続き命令の設計

⑥　プログラム詳細仕様の設計

(3)　設計手順の具体例

①　出力データ構造の設計

②　入力データ構造の設計

③　プログラム構造図の設計

④　構造的表現にもとづくフローチャートの設計

⑤　手続き命令の設計

⑥　プログラム詳細仕様の設計

第６課　プログラミング技法（１コマ）

１節　プログラミング言語の推移

機械語

低水準言語（アセンブリ）

高水準言語（COBOL,FORTRAN,PL/I,ALGOL,PASCAL）

オブジェクト指向言語(object oriented language)

自然言語(natural language)

２節　構造

１　わかりやすいデータ構造

２　わかりやすいアルゴリズム構造

アルゴリズム(algorithm)

対象となる問題をどのような手順に従って解決していけばよいかを系統的に記述したものである

ストラクチャード・コーディング(Structured coding)

構造化プログラミングで提唱された構造化の概念を適用したコーディング技法である。

ストラクチャード・コーディングの技法的なアプローチ方法

① 構造的に設計されたプログラムの詳細仕様書をもとに、ストラクチャード・コーディングの規約を当てはめプログラミングを行う

② 適正化されたプログラム構造（１つの入り口と１つの出口をもつ）をそのまま反映させたプログラミングを行う

③ GO TO 文は制限つきのものだけを使用する。制限つき GO TO 文は EXIT への迂回やループの途中脱出などである

④ 有効コメントを多用し、空白行や段付けなどにより見易さとわかりやすさを考えてプログラミングを行う

⑤　変数名や段落名などはできる限り意味をもたせる名前付けをする

⑥　字下げ構造をきちんととる。たとえば、データ構造の表現やループ文、IF 文などのネストの場合に適用する

⑦　不要なスイッチエリアを多用しない

⑧　ソースプログラムリストの中での１ページに、１つの機能ステートメントの集合体のみを記述し、ページがえをはかる

⑨　マルチステートメント（１行に複数命令を記述したステートメント）は行わない

COBOL 言語によるストラクチャード・コーディング記述

(1)　選択（IF THEN ELSE 文）

(2)　複合選択（CASE 文）

(3)　繰り返し（DO WHILE 文）

(4)　繰り返し（REPEAT UNTIL 文）

(5)　コメントの基準設定

(6)　GO TO 文の使用

(7)　階層的構造の活用

第三章　ソフトウエアのテスト・保守・管理

第７課　テストと技法（２コマ）

プログラムのテストとは、プログラムの中に存在しているであろうエラー（潜在

的バグ）を見つけだすためにプログラムを実行させ調べる工程である

１節　テストの概念

１　テストの定義と問題点

(1)　ソフトウエアの品質は設計工程で決定されており、テストの段階で品質を向上させることには限界がある

(2)　プログラムの中のエラーを見つけ出す過程に限界がある

(3)　心理的側面に左右されやすい

(4)　テストの充分性が不明確である

２　実務上の問題点

(1)　ソフトウエアの品質はテストによって決定されるのではなく、その前段階である設計に依存するという認識をもたなければならない。

(2) テストのしやすさはプログラムの構造に左右される。階層的に構造化され、独立した機能を実現させるプログラムはテストしやすい

(3)　プログラムを作成する者とプログラムをテストする者とは異なったほうがよい。

(4) ノーマルに結果を出力するテストデータは無意味である。エラーを発見することがテストの目的であり、プログラムにエラーを起こさせるようなテストデータが必要となる。異常データや例外データをどこまで考慮し、それらによってテストできるかが、プログラムの品質を決定できる

(5)　テストケースの設計においては、テストデータ値を決定するだけでなく、各テストデータによってプログラムを実行させた結果、出力されるであろう値もあらかじめ設定しておく。期待すべきテスト結果を事前に充分検討しておく必要がある。机上デバッグによってテストを効率よく実施することができる。

(6) テストケースに追加が生じたときは、波及効果を防ぐためにも今までテストしていたテストデータに追加として包括させたほうがよい。これは、プログラムのある部分を変更したとき、その影響が他の個所に悪い影響を与え、潜在的バグを作りこんでしまうことがある

2節　テストの方法

1　机上テスト

2　マシンテスト

マシンテストを行う具体的手順

(1) 机上テストによりテストケースを設計する

(2) テストケースには期待される出力結果も含めて考慮する

(3) ソースプログラムをコンパイルし実行形式のロードモジュールを作成する

(4) テストケースがファイルの入出力を取り扱う場合、ディスク上にファイルの領域を確保する

(5) ディスク上にテストケースに対応するデータを格納する

(6) ロードモジュールの実行環境を整える（アクセスするファイルをプログラムに組み込む手続きをする）

(7) ロードモジュールを実行させ出力結果を得る

(8) 出力結果を確かめ期待された出力通りかどうかチェックする

(9) (5)～(8)の作業を繰り返す

3節　テストケースの設計技法

テストケースとは、プログラムのある動作状態を調べるためにいろいろなテストデータを場合分けによって設定したものであり、期待される出力結果を含める。

ブラックボックステスト(black box test)

プログラムの外部仕様にもとづいたテストケース設計技法

ホワイトボックステスト(white box test)

プログラムの内部仕様にもとづいたテストケース設計技法

1　ブラックボックステスト

(1)　同値分割法

入力条件を２つ以上の同値クラス（有効同値クラスと無効同値クラス）に分割してテストケースを設計させるという指針にもとづくテスト手法

有効同値クラス
　プログラムにとって有効となる入力をもつテストケース

無効同値クラス
　プログラムにとって無効となる入力をもつテストケース

(2)　原因結果グラフ

原因と結果の関係をグラフによって関連付け､テストケースを導き出すプログラムテスト技法
　原因
　　入力データをプログラムの動作状態を決定付ける原因とみなす
　結果
　　出力データはプログラムが実行実現する結果とみなす

具体的手順

①　プログラムの外部仕様を機能的に分割しとらえ直す

② プログラムにとっての入力条件である原因と、出力条件である結果とに分け、それらを列挙し連番を振る

③ 原因結果グラフを記述する。グラフの関係を示す記号は図＊－＊に示すものを用いる。
中間接点の取り扱い

④ 原因、結果の関係の中では、外部的条件からの制約が生じることがある。たとえば、原因の中でどれか１つのみが対象となったり同時に起こらないといった実行動作環境に影響されることがある。こららの条件を明記しておかないと無用なテストを行いテスト実行の効率が低下する。そこで、制約条件を原因結果グラフに追加して記述する。制約条件記号は図＊－＊に示すものを用いる。

⑤ 上記まで完成した原因結果グラフをもとに、テストケーステーブルを作成する。ただし、テストケーステーブルの中で原因に対応するテストケースは膨大な数となる。たとえば、原因の個数が５個合った場合は２の５乗通りのテストケースが成立する。実際には、制約条件があったり、重複的な条件があるので、それらを除外していき最適化をはかる。そして、残ったテストケースがテストデータとして設計されることになる。テストケーステーブルは図＊－＊に示す構成をとる。

サンプル例

「３辺（A,B,C）を入力し、正三角形が２等辺三角形か、不等辺三角形か、三角形でないかを判定するプログラムがある。このプログラムに対してテストケースを原因結果グラフによって設計する」

① A,B,C の入力に対して、入力条件（原因）と出力結果（結果）を設定する

② 原因結果グラフを作成する。グラフには中間節点や制約条件も記述しておく。

③ 原因結果グラフをもとに、最適化したテストケーステーブルを作成する。

2　ホワイトボックステスト

構造テストとも呼ばれ、プログラムのアルゴリズムである内部構造を見ながら、詳細ロジック仕様書やソースプログラムリストをもとにテストケースを設計し、テストを実行する

(1)　命令網羅

プログラムのすべての命令を、少なくとも１回以上実行させることを基準としたテスト方法である

(2)　分岐網羅

プログラムのすべての判定条件命令（IF 文やループ脱出条件文など）で、真と偽を少なくとも１回以上実行させることを基準としたテスト方法である

(3)　条件網羅

プログラムのすべての判定条件命令で、条件の真および偽の組み合わせを満たすことを基準としたテスト方法である

(4)　条件・分岐網羅

分岐網羅と条件網羅をあわせたものである。プログラムのすべての判定条件命令で、条件の真および偽の組み合わせを満たすとともに、少なくとも１回以上実行させることを基準としたテスト方法である

3　設計手順

(1)　プログラムの外部仕様書にもとづき入出力の境界条件、交換条件および、これらの条件を満たさない誤りのあるテストケースを作成する。設計の手順は、ブラックボックステストの各論的技法を用いることにする。あくまで、プログラムの機能（WHAT）の面だけに着目し、テストケースを設計する。

(2) プログラムのソースリスト、あるいは、詳細仕様書（プログラム記述言語などにより、１ステートメント単位まで詳細に記述されているアルゴリズムの仕様書）を見ながら、分岐場所を調べる。そして、ステートメントが分岐する点、すなわち、判定条件命令が、真、偽とも実行するかどうか確認する（条件網羅、分岐網羅)。もし、不足しているならば、テストケースを追加する。

(3) (1)と(2)のテストケースが､プログラムのすべての経路を通るか同じかも確認する（命令網羅)。またこのとき、ループ処理においては、ループ０回（ループを通らずに迂回してしまう場合)、ループ１回、ループ最大回（ループ脱出条件に対応）についても調べる。もし、不足しているならば追加する。

(4) テストケースを設計したら、それぞれのテストケースに対応した出力結果を机上で想定する。そして、これらの内容をテスト条件書としてドキュメント化する。

テスト条件書のサンプルは図＊－＊である

４節　モジュールのテスト技法

モジュール集積テストともいう。モジュールをいくつか集積させて１つのプログラムとして完成せせる過程でのテストである。

１　ボトムアップテスト(bottom up test)

下位モジュールから個々のユニットテストを開始し、順次上位モジュールを結合してテストを進めていく技法である。下位モジュールと上位モジュールを結合せせるとき、上位モジュールはまだ作成されていないため、仮のモジュールを設定する必要がある。この仮のモジュールをテストドライバと呼ぶ。

これは、モジュールの呼び出し制御と、下位モジュールへのアーギュメント引渡し、下位モジュールからアーギュメント受け取りと表示といった機能を有するモジュールである。

モジュール関連図とモジュール構造図は図＊－＊、図＊－＊である。

特徴

(1)　下位モジュール同士の開発テストを平行して進めることが可能

(2)　テストドライバというシミュレーション用モジュールを作成しなければならない。また、ユニットテスト用に特別の環境設定が必要となるため、テストドライバの作成が煩雑になる場合がある。

(3)　モジュール間のインターフェイス上でのトラブルが発生しやすい。ボトムアップテストでテストが進められているため、最後にならないと全モジュールを連結した稼動状態にならない。このときに、インターフェイスに関する障害が発生することが多い。

2　トップダウンテスト(top down test)

上位モジュールからユニットテストを開始し、順次下位モジュールを結合してテストを進めていく技法である。上位モジュールと下位モジュールを結合せせるとき、下位モジュールはまだ作成されていないため、仮のモジュールを設定する必要がある。この仮のモジュールをスタブ（株分け）と呼ぶ。

これは、上位モジュールが必要な情報を戻し、制御のやり取りを満足させる機能を持つ。モジュール制御が渡った時の状態を示すメッセージを表示し、上位モジュールへアーギュメントを戻す機能を有するモジュールである。

モジュール関連図は図＊－＊である

テストの手順

特徴

(1)　プログラムがトップダウンアプローチにより、モジュールが分割されていることが前提となる

(2)　モジュール間インターフェイスに対する仮説設定がいらない。プログラムを構成しているモジュールの構造をそのまま継承しながら、上から下へとテストすることができる。最後の下位モジュールのユニットテストが完了した段階で、すべてのモジュール結合が実施されたことになり、プログラムテストが完了したことになる

(3)　上位モジュールは、次のレベルの下位モジュールテスト時に何度も実行

が行われていることになる。この結果、上位モジュールに内包されている潜在的バグが発見しやすくなり、品質が向上していく

(4) テストドライバよりスタブのほうがモジュールとしての機能がシンプルとなり、作成しやすい場合が多い

(5) プログラム開発の最初の工程では、いくつかのモジュールを同時にユニットテストすることが難しく、並行的にテストすることができない。このため、テスト作業の効率が多少低下することになる

3 その他のテスト

(1) サンドイッチテスト(sandwich test)

ボトムアップテストとトップダウンテストのそれぞれの利点を取り出し、いっしょにしたテストである。
上位モジュールと下位モジュールのユニットテストを同時にスタートさせる。いずれ中間のモジュールで合致し、そこでプログラムの全モジュールテストが完了したことになる

手順

トップダウン方式によりスタブをプロトタイプとしてすべて作成してしまう。プログラムの全体構造を先に形づくってから、個々のモジュールを下位からボトムアップ方式によってテストする。

(2) ビッグバンテスト(big bung test)

すべてのモジュールを個別にユニットテストし、お互いのインターフェイスについて考慮せず、すべてのモジュールのユニットテストが完了した時点で、一度に完了してしまう方法である

プログラムもモジュール分割した場合、もっともトラブルを起こしやすい個所は、モジュール間のインターフェイス部分である。このことから、インターフェイスをプログラムテストの最終段階で実施することは、あまり得策ではない。

一度に結合してプログラムをテストし、おかしな結果が出た場合、どの

モジュールが原因なのかも把握しづらい。１つずつモジュールを集積していくアプローチでは、エラー原因は今回結合したモジュールに起因していることが多い。が、ビッグバンテストでは、そうはいかない。すべてのモジュールを調べなければならなくなる。

　小規模なプログラムで、モジュール構造がきちんと設計され、インターフェイスが煩雑でないものならば、効果がある。

第８課　保守技法（１コマ）

１節　保守とは

１　修正のための保守

ソフトウエアが保有している欠陥に対して修正すること

(1) SE とユーザ相互の誤解や連絡漏れ、ユーザ不在のままの独断的ソフトウエア開発経過などにより、ユーザの要求仕様に対してくいちがいが発生しているとき

(2) プログラマや SE によるケアレスミス、テスト不足によるソフトウエアの品質不良、あるいは、本番稼動時の例外データを取り扱時にプログラムが異常終了を起こしたとき

(3) ソフトウエア稼動時の性能上の問題。ファイル構造の不適合によりシステムのレスポンスタイムが要求とおりでない場合や、見積もりミスや設計ミスあるいは連携編集ミスなどにより、CPU 上のメモリ負荷の過剰が発生した場合など

(4)　開発上の規約違反。プログラム構造の設計内容やプログラムのコーディングにおける標準化に対しての不一致や、修正漏れあるいは標準化規約無視による仕様書などを含めたドキュメントの未整備がおこったとき

２　適合のための保守

ソフトウエアのライフサイクルにより、ソフトウエアを維持、管理していく

上での外部環境の変化に伴う修正をすること

(1) ハードウエア機器構成（周辺装置や端末装置など）の変更や、ホストコンピュータを他メーカー機種へリプレースしたとき

(2) ソフトウエア構成（OS や開発支援ツール、プロダクト類）の切り替えやバージョン、リリースアップに伴う変更など

(3) ファイル構造（データベースの物理構造や論理構造、ファイル編成方式）の変更や拡張が生じたとき

3　改善のための保守

ソフトウエアシステムをよりよく使用したいという要求にもとづき修正をする

(1) ソフトウエアの稼動効率の向上をめざし、ファイルのアクセス方法をアルゴリズムの面から改善するとき

(2) ユーザからの追加仕様や要求変更などに伴い、ソフトウエアシステムの機能拡張を行い改善をめざすとき

(3) センタ側でのシステム運用効率をめざし、運用マニュアルの整備や JOB ストリームの自動生成などをはかるとき

(4) ユーザ、あるいは、オペレータなどといったシステム運用担当者の操作容易性を考慮したソフトウエア面での変更など

4　予防のための保守

あらかじめ将来の変更に対する拡張や信頼性の向上を考慮してソフトウエアを修正することである

2節　保守作業法

保守工程

保守工程とは、ソフトウエアのライフサイクル上で、分析、設計、開発、検査が終了し、システムの本番稼動がスタートした時点からソフトウエアが破棄されるまでの期間を示す。この工程上の保守作業には既存修正と新規追加がある

既存修正

すでに開発済みのソフトウエア構成要素における一部を修正する作業。この際、ソフトウエアシステムの全体構成と機能要素をすべた把握しておかなければならず、新規開発より困難な作業となることが多い。ある個所を修正すると、他の個所に悪影響を及ぼすことがあり、このことを波及効果というが、これを最小限に抑えることが求められる。

新規追加

すでに開発済みのソフトウエア構成要素に新しく機能を追加する作業。既存の部分と新規追加の部分におけるインターフェイスについて留意する必要がある。

保守作業の手順

1　既存ソフトウエアの理解

設計仕様所の中で、システムの概要を記述したドキュメントをもとに、システムの機能面を捉え全体像を把握する。その後に、詳細仕様設計書やテスト条件書、あるいは、ソースプログラムリストを読み、システムの詳細な処理手順を理解する。これらは、すべて、机上で行われる

2　既存ソフトウエアの修正

既存のソフトウエアに対して修正をかける。実際には、ソースプログラムのメンテナンス、及びリコンパイルによって最新の実行形式モジュールを作成する。そして、ユニットテストを行うとともに、仕様書類の関連事項の修正も済ませる。ソースプログラムの修正時においては、特に前述した波及効果に注意して作業を進めなければならない

3　新規ソフトウエアの開発

新規機能追加のときは、要求仕様に合わせたプログラムを別途開発する。プログラムの設計、コーディング、デバッグ、ユニットテストまで行う。このときも。既存のソフトウエアとのインターフェイスを意識しながら作用を進めていく必要がある。

4　結合テストによる一貫性検証

ユニットテスト終えた単体のプログラムを、既存のソフトウエアシステムに組みこみ、一貫性、すなわち、プログラム間の整合性を検証する。また、システム全体として、修正や追加変更による波及効果のテストを行う。このときは、開発時に用いたシステムテスト用データにより再スタートしたほうがよい。これは、既存のソフトウエア部分に対する波及効果の度合いが、効果的にはかれるからである。

5　保守完了時の検査実施

保守作業がすべて完了した時点で、修正したソフトウエアの個所に対応した仕様書類の記述内容を検査する。最新のソフトウエア環境と整合性の取れたドキュメント記述かどうかが、検査のポイントとなる。というのも、ソフトウエア自体は無形で目に見えないものであるがゆえに、ドキュメントが唯一のソフトウエア記述媒体となるからである。

6　保守管理状況表の作成・管理

ソフトウエアは、長期的なライフサイクルを持っており、そのほとんどは保守工程である。したがって、保守の状況を管理しておくことは重要なことである。そこで、保守状況管理表などを作成し、変更発生日付、発生原因、修正概要、修正詳細、修正者、修正工数、波及効果度などを記録しておく。

3節　保守に関する現状の問題点

(1)

(2)

(3)

(4)

(5)

(6)

(7)

(8)

(9)

(10)

(11)

(12)

(13)

(14)

(15)

(16)

(17)

(18)

(19)

(20)

(21)

(22)

４節　解決へのアプローチ

１　ソフトウエア開発工程上の工夫

現状の技術レベルと管理手法の範囲内でのアプローチ

(1)　開発工程における標準化の基準を設定し、その検査機構を組織し運営する。標準化を設定することによって、ソフトウエア全体の構成が統一化され、誰が見ても理解しやすくなる。このことが、保守作業の効率を高めることになる。標準化規約としては、ソフトウエアシステムのシステム構造やプログラム構造のパターン化、仕様書の書き方の標準化、プログラムのモジュールの再利用化、コーディング上の規約、検査方法などがあげられる。

(2)　ソフトウエアの分析、設計段階に保守要員（担当者が異なる場合）を参加させる。ソフトウエアライフサイクルの上流部工程で、保守要員からの要望やアイデアを取り入れ、設計を構築していく。

(3)　ソフトウエアの品質基準を「保守の容易さ」としてとらえて、設計を行う。保守の容易さとは、構造化されたモジュール分割されていること、拡張性があること、高品質のドキュメントが整備されていること、標準化が守られていること、わかり易いプログラムコードであること、テストが容易であることなどがあげられる。

２　ソフトウエア保守工程上の工夫

ライフサイクルの全局面に着目したアプローチ

(1) ソフトウエア開発時に使用した支援ツールを、有効に活用する。
開発支援ツール

ライブラリ管理ツール、データディクショナリ／ディレクトリ、テストデータジェネレータ、スクリーンエディタ、デバッグツールなど

(2)　保守環境の整備と保守作業の標準化を推進する。開発済みソフトウエアの引継ぎ時における受入検査を実施し、引継ぎ資料などの品質チェックを行う。開発工程での標準化と同様に、保守工程においても作業内容の基準を設定する。また、保守作業上における監査機構も設置し運用する。

(3)　保守作業の経過を履歴情報として蓄積する。そして、ある時期にシステム評価を実施し、保守を通してソフトウエア全体を評価する。この結果を時期システムの開発時時にフィードバックし、設計や開発の改善を目指す。

(4) プロジェクト開発要員の一人が、保守要員として保守作業を継続する。ソフトウエアのライフサイクル全局面にわたって、一人の要員が管理していくので、最も効率のよい対応が取れる。また開発と保守の引継ぎ上のトラブルもなく、要員のローテーションも円滑に進み、理想的な体制といえる。が、現実的には、要員不足のため難しい。

3　保守用新技術の開発

ソフトウエア開発部門の生産性向上が期待できる唯一のアプローチである。保守作業から属人性を廃し、保守支援環境をハードウエアおよびソフトウエアによって構築し自動化を図る。

(1) 保守支援用ソフトウエアツールの実用化を図る。専用ミニコン上に構築し、保守用データベースの管理、運用を図る。

保守用データベース

ドキュメント管理、ファイルデータ管理、ソースプログラム管理、テストデータ情報管理、JCL(Job Control Language)管理、保守履歴管理

(2)　高性能保守用ワークステーションの実用化を図る。リアルタイム処理が可能で、ホストコンピュータに負荷をかけないようなインテリジェントタイプが併設されている。保守用データベースと、会話型で容易なインターフェイスをもつ環境が設定されているとともに、日本語版画面エディタ機能を持つシステム構成である。

第９課　ソフトウエア工学の展望（２コマ）

１節　ソフトウエア工学の動向

２節　ソフトウエア工学の今後

１　CASE

(1)　CASE とは

CASE(computer aided software engineering)　とは「コンピュータ支援によるソフトウエア工学」と訳されるが、ソフトウエアのライフサイクルの各局面を自動化するための仕組みを持たなければならない。また、何らかのパラダイムに立脚した方法論や技法が導入されている。この場合の、ライフサイクルには、分析や設計だけでなく、製造や運用そして保守まで含まれている。すなわち、新規にソフトウエアを作成するという工程を自動化するだけでなく、運用や保守工程まで自動化しなければならないことを明らかにしており、これによって、初めてソフトウエア開発・管理における生産性と信頼性を向上させることができる

構造化パラダイムの導入という意味は、ソフトウエア工学で生み出された方法論や技法の中で最も広く普及し定着しているものを用いようということである。方法論はソフトウエアの概念、技法は構造化分析や構造化設計が相当し、プログラミング工程以降は自動化を目指すことから、内部的にはプログラムの自動ジェネレイト(automatic generate)が行われることを前提としている。

CASE を用いる利点として、ソフトウエア生産工程における自動化が進み、仕様書やプログラムの自動生成や設計・製造過程での自動検証などが実現できる。また、属人性に依存することなく、標準的な生産工程を導入することが可能となる。さらに、保守工程における自動化も実現できる。

(2)　CASE の構成要素

＊１　ワークステーション

CASE を操作するためのインターフェイス部分である。

ハードウエアは高性能のマイクロプロセッサ(RlSC：reducd instruction set computer)を搭載し、高密度のビットマップディスプレイ、ポインティングデバイスを装備している。

また、数十メガバイトのハードディスクやネットワークインターフェイスも備えている。

ユーザインターフェイスには、マルチメディアウィンドウシステム(multi media window system)によるグラフィックユーザインターフェイスが用いられている。

これらによって、DFD や ER チャートやモジュール構造図あるいはデータのスキーマ構造などが、図イメージのまま操作することができる。

CASE の構成要素図は図＊－＊である。

＊２　構造化技法の適用

CASE を用いてソフトウエア生産を行うときに、すべて構造化技法のもとにガイドされながら要求定義や設計を進めることになる。

要求定義工程　　構造化分析法
システム設計工程　構造化設計技法
製造工程　　構造化チャートによるプログラムジェネレイト

＊３　ソフトウエアデータベース

リポジトリ(repository：知識の宝庫)

エンサイクロペディア(encyclopedia：百科事典)

ソフトウエアを生産する際に必要となるいろいろな**仕様書**や**設計書（画面、帳票、ファイル、データベース、プログラム)**、あるいは、内部的に自動生成されたソースプログラムテキストや翻訳されたオブジェクトプログラムそして実行形式プログラム、テスト用データなどを格納するためのデータベースである。

格納された情報そのものの妥当性チェックを動的に行うだけでなく、これらの情報で関連しあうもの同士の整合性のチェックを同時に行ってから有機的なリンクを張るといったことも実現する。

保守時における情報変更をすべて動的に実行するとともに、変更管理機能としての**バージョン管理(version control)**も提供する。

複数のワークステーションからアクセスされることもあるので同時実行制御や、アクセス範囲を限定するための機密保護管理なども有する。

(3) CASE の動向

第一世代

CASE という言葉がはじめて定着するとともに、単体ベースの CASE が中心であった。

第二世代

CASE の統合化が進み、上流工程から下流工程へのソフトウエア自動生成として**フォワード・エンジニアリング**(forward engineering)の考え方が実現された。

第三世代

開発工程の支援だけでなく、保守工程の支援というアプローチも導入されるようになってきた。具体的には、既存ソフトウエアの再利用やプロトタイピング支援のために、**リバース・エンジニアリング**(reverse engineering)や**リ・エンジニアリング**(re- engineering)という考え方が適用されている。

リバース・エンジニアリング

ソーステキストから逆変換しながら下流工程から上流工程へと戻りながら設計書や仕様書などを再作成するといった機能を工学的に実現するための技法を持つ。

リ・エンジニアリング

リバースによって得た情報をもとに部分的に修正を加えながら、上流工程から下流工程へと向かって既存システムに類似した新しいシステムを再構築するといった機能を工学的に実現するための技法を持つ。

CASE の機能的分類

コンポーネント(component)CASE

ソフトウエアのライフサイクルのうちある一部分のフェイズだけをサポートしているものであり、**上流工程（分析・設計）**だけを中心としているものを **upper-CASE**、**下流工程（製造・検査）**だけを中心としているものを **lower-CASE** と呼ぶ。

この方式は IBM 社が提唱しているもので、各ソフトウエアベンダーの中でもっとも得意としているフェイズについての CASE を作成すればよいという考え方をベースとしている。ただし、前提条件として、ソフトウエアデータベースであるリポジトリの情報構造について、その内部構造の使用はパブリック・ドメインとして明らかにする必要がある。なお、実際にはこのリポジトリの規格化が、**ISO（国際標準化機構）**と **ANSI（国際規格化協会）**を中心に、**情報資源管理辞書(IRDS)**という形で進められている。

インテグレイテッド(integrated)CASE

ソフトウエアのライフサイクルの全フェイズをサポートしており、ソフトウエアデータベースを介して、核フェイズごとに必要となる情報を取り出したり格納することができるようになっている。

CASE のアプローチは図＊－＊である

2　オブジェクト指向パラダイム

構造化パラダイムをベースとしたソフトウエア工学は、CASE の普及により実用段階に入っている。最近では新しいパラダイムであるオブジェクト指向パラダイムが台頭しつつある。

CASE を構築するため、技術革新を進める過程における台頭の起因

＊１　構造化パラダイムによるソフトウエア開発プロセスに問題があることが起因

構造化分析から構造化設計につながる部分は親和性があり、フェイズ同士の関連がスムーズに進む。しかし、プログラミングフェイズで用いる構造化プログラミングや構造化チャートといった技法への連動がうまく取れなくて、そのフェイズ間にギャップが存在するという問題である。

これは、プログラミング工程ではシステム動作における動的状態をアルゴリズムなどにも組み込まなければならないのに、構造化設計技法により作成された仕様は主にプログラムの静的状態しか設計できていないために生ずる問題といえる。

これに対して、オブジェクト志向パラダイムでは、オブジェクト志向分析からオブジェクト指向設計そしてオブジェクト指向言語への一貫した関連があり、ギャップはまったく存在しない。

＊２　データベースとプログラムそのものの親和性が弱く、インピーダンス・ミスマッチ（impedance mismatch）が生じることが起因

データベース言語とプログラミング言語との間に整合性が取れず、どちらかに合わせて強制的に組み込まなければならない。また、ソフトウエア開発プロセスでも重要なテーマであるデータ設計において、たとえば、データベースのスキーマ設計に構造化パラダイムを適用することはできない。

もし、データベースに関係モデルを用いるならば、ER モデルや正規化技法を用いて別途スキーマを設計しなければならない。これに対して、プロジェクト志向では、こういったインピーダンス・ミスマッチがまったく存在しない。というのも、オブジェクト志向データベースは、オブジェクト指向プログラミング言語で記述するオブジェクトそのものを２次元記憶領域に記憶させておきたいという機構を実現するために生み出されてきたからである。

＊３　ソフトウエアデータベースをどのように実現するかが問題となったことによる起因

さまざまなソフトウエアに関する情報を格納し管理しなければならない。そのソフトウエアデータベースを、たとえば関係データベースモデルによって実現しようとしても無理があることがわかってきた。これに対して、オブジェクト指向データベースモデルの適用が、より実現性のあるアプローチであることが指摘され始めた。というのも、いろいろな

ソフトウエア情報それぞれを個々のオブジェクト単位として識別した上で管理すればよいからである。

構造化とオブジェクト指向との比較は図＊－＊である

(1) オブジェクト指向の概念

プロセス重視のプログラム開発

外界の対象となる事象を抽象化するための階層や表、リストなどいろいろなデータ構造に写像しながらモデリングを進め、それらのデータ構造を処理するためのプロセスを具体化するという手順を踏むのが一般的であった。

また、プロセスをどのように手続きとして抽象化すべきか（手続きの抽象化）といったことに対して関心が高まった。

さらに、プロセスを実現するための手段として、数多くの手続き型プログラミング言語が開発された。

プロセスを設計する上で、データとプロセスは極力別々にしておいたほうがソフトウエア（プログラム）における独立性を高める上でも重要なことであるという考え方も生まれた。

データ志向(data oriented)のプログラム開発

データ抽象化(data abstraction)のソフトウエア開発が注目され始めた。これは、データの内部構造について隠蔽してしまい、ある定義された操作を外部からアクセスすることによって処理を実行するという方法である。これによって、データの部分が情報隠蔽されて抽象化されることになる。

その考え方をプログラムのモジュール化における具体的な指針として適用したものが**抽象データ型**(abstract datatype)である。

これは、プログラムが取り扱うデータとその操作を対応付け、データ型と操作をひとまとめにして抽象化したものである。すなわち、ある処理機能をモジュールとして定義する場合に、内部の複雑な動作を隠蔽するために、それを**型**(type)を持つ変数として定義する。これによって、そのモジュールが持つべき操作が抽象化され、ある型を持つデータとして取り扱うことが可能となる。

この抽象データ型をクラスという概念で発展させたものが、オブジェクト指向である。

オブジェクト指向(object oriented)の概念

＊1　オブジェクトとクラス

物(object)・・外界の事象に存在するすべての問題

クラス(class)・・オブジェクトの、共通する属性

インスタンス(instance：実体化)・・クラスをオブジェクトに変換

クラス名(class name)・・クラスを識別するための名称

インスタンス変数(instance variable)・・インスタンスのための変数

メソッド(method)・・クラスの振る舞いを記述したもの

カプセル化(encapsulation)

クラスの定義においてその**振る舞い(behavior)**を示す**メソッド（プロセスに相当）**と、**クラスの状態(atate)**を示すインスタンスされたオブジェクト（データに相当）をいっしょに統合化する考え方。

これにより、オブジェクトの内容が隠蔽化されることになる。オブジェクトそのものが、抽象データ型によって定義された**実態(entity)**に相当するわけである。

このとき、ある定められたインターフェイスによってのみオブジェクトをアクセスするという制約を受けることになる。具体的には、オブジェクトの振る舞いはインスタンス変数とメソッドによって記述するが、その振る舞いの参照については**メッセージセレクタ(message selector)**によってのみ許される。さらに、参照そのものをあるオブジェクトに制限することもできる。

このように、構造化パラダイムではデータとプロセスは分離するという考え方であったが、オブジェクト指向パラダイムではデータとプロセスをカプセル化するという考え方に変わっている。

また、クラスからインスタンスされたオブジェクトは、他のオブジェクトからメッセージを送られることによって起動する。このことを、**メッセージ・パッシング(message passing)**と呼ぶ。

クラスの定義は図＊－＊である

＊２　クラスの構成

すべてのオブジェクトはあるクラスに属することになるが、複数のオブジェクト同士の重複を排除し統合する必要がある。そのためのグループ化に、クラス階層による汎化(generalization)関係という考え方を導入する。

汎化とは、共通する部分のみに着目して一般化された分類を行うことである。これをクラス同士の階層関係に用い、その汎化関係のことを is-a 関係と呼ぶ。

is-a 関係は図＊－＊である

汎化の反対を特化(specialization)という。これは、あるレベルのクラスに対してより特化した性質を付加してクラスを分類しながら定義することができる。

is-a 関係において、最上位のクラスをスーパークラスと呼ぶ。そして、スーパークラスに属するすべての下位のクラスのことをサブクラスと呼ぶ。

スーパークラスの属性はサブクラスに引き継がれる。このことをインヘリタンス(inheritance：継承)と呼ぶ。サブクラスでは、スーパークラスで定義したインスタンス変数やメソッドを繰り返し定義する必要がなくなり、クラス間の一貫性が保たれることになる。また、サブクラスで必要となった部分だけを追加するという差分プログラミングが可能となる。

＊３　オブジェクトの構成

１つのオブジェクトがいくつかのオブジェクトから構成されている場合、集約化(aggregation)という概念を用いて定義する。

集約化とは、いくつかの部分をまとめて１つのものとしてとらえる

という考え方である。これをオブジェクト同志の階層関係に用い、その集約化関係のことを part-of 関係と呼ぶ。

この集約化を用いることによって、オブジェクト同士が入れ子構造になっている複合オブジェクトについても定義することが可能となる。

なお、集約化の反対を分解(decomposition)という。

part-of 関係は図＊－＊である。

(2) オブジェクトと志向パラダイムの動向

オブジェクト指向は、当初言語の世界から発展してきた。もともとはシミュレーション言語である Simula67 にさかのぼることができるが､やはり 1983 年に米ゼロック社からリリースされた Smalltalk-80 が大きな原動力となったといえる。

オブジェクト指向パラダイムが構造化パラダイムに比べて有望視されている理由には、図＊－＊のような背景が挙げられる。たんに、プログラミングのための規範を提供するだけでなく、オペレーティングシステムやアーキテクチャ、データベース、そしてユーザインタフェイスまで幅広い分野に適用できることが特徴となっている。

現在、製品化されているもの

① オブジェクト指向プログラミング言語(object oriented programming language)

Smalltalk-80 を始めとして、1983 年に ICOT(institute for new generation computer technology：新世代コンピュータ技術開発機構)から Prolog をベースとして開発された ESP や米ゼロックス社から LISP をベースに開発された LOOPS がある。1984 年には米 PPI 社が Objectiv-c、そして 1985 年には米 AT&T 社が C++(Version 1.0)を、それぞれ発表している。

② オブジェクト指向オペレーティングシステム(object oriented operating system)

1988 頃から出荷され始めた米キー・ロジック社の KeyOS シリーズがある。

③ オブジェクト指向アーキテクチャー(object oriented architecture)

1988 年に IBM 社から発表された AS/400 のアーキテクチャがある。

④ オブジェクト指向データベース(object oriented database)

1987 年に米セルビオ・ロジック・デベロプメントから出荷された GemStone や 1988 年に米グラファエル社から販売された G-BASE そして米オントロジック社から販売された ONTOS(Vbase)などがある。

⑤ オブジェクト指向ユーザインターフェイス(object oriented user interface)

エンドユーザ向けとして米ヒューレット・パッカード社の NewWave などが、支援ツール向けとして米ネクスト社の NextStep などがある。また、最近では、UNIX 上の GUI(graphic user interface)である OPEN LOOK や OS/2 プレゼンテーションマネジャ、そして OSF/Motif などもオブジェクト指向の考え方を取り入れつつある。

⑥ オブジェクト指向分析(object oriented analysis) とオブジェクト指向設計 (object oriented design)

オブジェクト指向プログラム言語を用いてシステムを完成する際に、どのようにしたらうまくプログラムへ具体化できるかという指針を示すものである。コードとヨードンの提唱するものがある。

オブジェクト指向分析の手順

① オブジェクトを抽出し確定する
② 解決すべき問題に対してオブジェクトとなりうるものを列挙し、統廃合しながらその名前を決める
③ クラスとオブジェクトの構造を決定する
汎化と特化の関係である is-a と集約化と分解の関係である part-of を用いて、性的な構造関係を設定する。必要ならば、それらのオブジェクト群をより大きな単位にグループ化し、サブジェクトとしてまとめる。サブジェクトという、より抽象化されたまとまりによって、参照することができる。
④ オブジェクトごとの属性（インスタンス変数）とサービス（メ

ソッド）を定義する。また、このときオブジェクト感の動的な関係を、インスタンス結合として結びつける。

オブジェクト指向設計の手順

① ハードウエアとソフトウエアのアーキテクチャの設計を行う
とくに、プログラミングへの以降や開発環境についても考慮しなければならない
② クラス構造の見直しと最適化を図る
③ 各オブジェクトが持つべきサービスについて詳細に記述する
この際には、すでによく用いられている各種の図式（ペトリネット図や ER 図や状態遷移図など）を利用してもよい。これは、オブジェクト同士状態とその間のメッセージ遷移などを設計しなければならないからである。
④ 計仕様をもとに強力なオブジェクト指向プログラミング支援環境によって、ソースプログラムのテンプレートを生成する