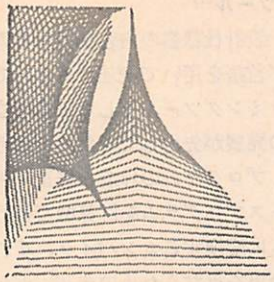
UDC 681.34.06.001.6

小特集



# II. 汎用計算機のソフトウェア開発環境

中所武司（ちゅうしょたけし）
(株)日立製作所システム開発研究所

## 1. まえがき

1970年ごろからのソフトウェアの大規模化とともに，その信頼性と生産性向上のための技法の確立が急務とされ，二つのアプローチがとられた。その第一は「構造化プログラミング」という言葉に代表されるプログラミング方法論と言語に関する研究である(1)。第二のアプローチは「ソフトウェア工学」という言葉に代表される分野の研究であり，ソフトウェア開発工程のあらゆる局面，すなわち，要求定義，設計，プログラミング，テスト，保守運用の各々を対象とした方法論，技法，ツールの開発が行なわれてきた(2)。

本論文では，このような背景を踏まえて，大規模ソフトウェアの生産技術の現状およびその課題に対する展望について述べる。

## 2. 大規模ソフトウェア開発上の課題

ソフトウェアライフサイクルの各々の段階での生産性阻害要因として次のようなものがある。

まず要求定義段階ではユーザの要求を完全な仕様にまとめる必要があるが，実際には，ユーザの要求を明確に文書化し，検証する技術がないため，あいまいさを残したままソフトウェアの開発を行なっている。そのため，開発をほぼ終了したシステムテスト段階で，ユーザ要求との不一致が検出され，大幅なプログラム変更が生じる。このように開発工程の最初に作り込んだ誤りを後の工程で検出し，除去する費用は膨大なものになる。ある統計(3)では，誤りの検出，除去費用は図1に示すように誤りの検出が遅れるほど指数関数的に増大する。

次の設計段階では，要求定義を満たすためにどのようなソフトウェアを作るかを決定しなければならないが，一般的手法はなく，経験と勘に頼った設計が行なわれている。そして，この段階で作成された設計仕様が要求定義を満たしていること，および所定の計算機

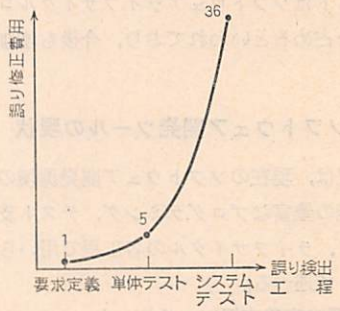



図1 誤り検出時期と修正費用の関係

システム環境下で機能的にも性能的にも正常に動作することを検証する手段がないことから，後のテスト工程で検出される重大な誤りの多くがこの設計段階で組み込まれている。

プログラミング段階では，最近開発すべきソフトウェアが量的に増大しているにもかかわらず，プログラマの増加には限界があることから，新たな危機的状況が生じつつある。この問題は，基本的には従来のソフトウェア工学の延長上での生産性向上の努力を行なっていく必要があるが，プログラマの増加とともに「ソフトウェアの生産性はプログラマの個人差による影響が大きい」という問題がより深刻になっている。すなわち，ある調査(4)ではツールの良し悪しによる生産性の差が 1.49 倍であるのに対し，人間的要因による差は 4.18 倍であったと報告されている。

テスト段階では，現在ソフトウェア開発工数全体の約半分を費している(5)が，これはほかの分野ではみられない現象である。本来，ソフトウェアのテストはプログラムが仕様書どおりりに作られていることを確かめるのが目的なので，仕様書とプログラムが等価であることを計算機を用いて自動証明する方法が理想的であるが，未だ実用技術はない。実際には，テストデータを用いてプログラムを実行し，その結果を確認するという動的テスト法が用いられている。この方法で

は，人手作業に頼ること，および，もし誤りがあれば必ずそれを検出できるような効果的なテストデータの作成方法がないことのためにテスト費用がかさむ原因となっている。

最後の保守段階では，開発担当者とは別の人がプログラムの修正や機能拡張を行なうことが多い。ところが，保守仕様書と呼ばれるドキュメントと実際のプログラムの内容が一致していないことが多く，保守担当者はソースリストを頼りにプログラムを解読し，変更するため効率が悪いうえに，以前に正常動作をしていたところまでがおかしくなることがよく生じる。この保守コストはソフトウェアライフサイクルコスト全体の7割を占めるといわれており，今後も増加傾向にある。

## 3. ソフトウェア開発ツールの現状

本章では，現在のソフトウェア開発環境の中で特に支援機能の豊富なプログラミング，テスト支援ツールを中心に，ライフサイクルの各工程で用いられるツールについて述べる。

### 3・1 要求定義支援ツール[2]

要求定義段階では開発すべきシステムの概念的モデルを構築し，その稼働環境とのインタフェースを明確にする必要がある。1970年代にはPSL/PSA, SREM, SADT などの先駆的な支援システムが開発されたが，定義誤りがシステムテストの段階まで検出されないとか，開発未経験の新システムの要求定義が難しいなどの欠点があった。そこで1980年代には要求定義段階で実行可能なモデルを構築し，仕様の正当性や完全性を早期に確認する技法が研究され，プロトタイピングツールとして試用され始めている。

### 3・2 設計支援ツール[2]

ソフトウェアの設計では要求定義の概念的モデルを計算機で実行可能なモデルに変換し，その詳細仕様を決める。代表的技法としては，データフローに注目してソフトウェアの構成を決めていく複合設計法や，データ構造の設計を主体としたジャクソン法のほか，設計後半でのモジュール化技法として，段階的詳細化技法や構造化プログラミング技法がある。この設計は，ソフトウェア開発における各工程の中でも最も人間の知恵が要求されるところであり，自動化（ツール化）は難しい。現在は設計仕様書の計算機入力化を基本にして，その編集系（エディタ）によってモジュール相互の無矛盾性や完備性をチェックしたり，必要な情報を見やすい形式でドキュメント出力したり，更には段階的詳細化技法に基づいて設計仕様書作成を誘導するような設計支援ツールが用いられ始めている。

### 3・3 プログラミングツール[1]

プログラミングとは，設計仕様書の内容を計算機入力可能なプログラミング言語を用いて記述することである。従って，プログラミングツールとしてはこれまでプログラミング言語の発展が先行してきた。その他の基本的なものとして，プログラムの作成と修正に用いるエディタ，そのソースプログラムを計算機で実行可能な機械語プログラムに変換するコンパイラ，それを実行しながら誤りの検出を行なうためのデバッガなどがある。以下，これらの詳細について述べる。

（1） 言語とコンパイラ　　ソフトウェアの大規模化が進み，保守，拡張費用が増大するにつれて，プログラムの書きやすさよりも読みやすさの方が重要になってきたが，この点ではFORTRAN, COBOL などの従来言語は不十分であった。そこで最近の言語設計では，これまでのような機械語への展開率を向上させる「量的高級化」よりも構造化プログラミングに代表されるようなプラグラミング方法論を反映した「質的高級化」を実現している。すなわち，制御構造に関しては GOTO 文のような無条件分岐を排するとともに，プログラムの段階的詳細化によるトップダウン開発を支援するモジュール構造の導入により，またデータ構造に関しては豊富な基本データ形や新しいデータ形の定義機能，あるいはデータ形とその操作手続きをまとめて定義するデータ抽象化機能の導入などにより，理解容易なプログラムを作成できるようにしている。その代表的言語として Ada があり，筆者らもデータ抽象化と段階的詳細化技法を支援する構造化言語 SPL[6] を開発した経験がある。

一方，コンパイラについては，今まで処理性を重視してアセンブラで記述していたようなプログラムを生産性，移行性の良い高級言語で記述するようになるとともに，実行速度の速い機械語プログラムを生成するための最適化処理の強力なコンパイラが開発されている。しかしながら，プログラム開発時には頻繁にプログラムの修正と再コンパイルを繰返すため，最適化機能よりも，むしろ豊富なデバッグ機能や，わかりやすいエラーメッセージ出力，あるいはプログラム解析による誤りの自動検出などの機能が有効であり，最近のコンパイラにはこのような開発支援機能が付加されている。

（2） エディタ　　エディタは最も使用頻度の高いツールであり，その使い勝手の良し悪しがプログラミング効率に大きく影響するが，従来の行エディタに代り，最近は画面エディタが普及し，操作性は大きく改

善された。しかしながら，これらはいずれもプログラムを単なる文字列とみなして編集するテキスト操作機能が主体となっているため，端末操作数が多いことや文法的に誤ったプログラムも入力可能であるなどの欠点がある。そこで，最近ではこれらのテキストエディタに代り，プログラミング言語の文法規則を内蔵し，構文要素単位の編集機能を有する構造エディタが開発され，普及し始めている。

その一例として，ここでは筆者らの開発した構造エディタ PARSE[7](Production And Reduction Structure Editor) について簡単に説明しておく。その主な機能は次のようなものである。

（i） 自然語で記述された処理概要の設計仕様からプログラムへの段階的詳細化の誘導

（ii） メニュー選択方式および構文テンプレートの自動表示機能による構造化プログラムの誘導

（iii） 構文要素単位のプログラム修正

（iv） 入力プログラムの即時文法チェック

従来の机上作業やコンパイラを用いた文法誤り修正作業が構造エディタで集中的に実行可能となっている。例えば，次のような擬似文と呼ばれる自然語で記述された処理概要を段階的に詳細化する場合を考える。

“add data”

まず refine キーを入力することにより，この擬似文がコメントに変更され，文の入力要求状態になる。対象言語が Pascal の場合は次のようになる。

{add data}
〈statement〉

ここで処理内容を直接入力してもよいし，手続き引用形式で入力してもよい。各々の例を図2の①，②に示す。手続き引用の場合は更に refine キーを入力すれば，図2③のように手続き本体のテンプレートが自動表示され，手続き定義可能状態となる。PARSE はこのような処理概要の詳細化のほかに文法規則に沿った詳細化も誘導する。図2④は，文の入力要求状態で可能な詳細化形式を表示したメニューの中から if 文を選択し，そのテンプレートを自動表示した例であり，構造化プログラムが誘導されている。プログラム修正についても生成の場合と同様に構文要素単位に行なえる。

このように構造エディタは，ソフトウェアの生産性がプログラマの能力差に大きく依存する問題を解決するために，熟練者はもちろん，初心者でも信頼性の高いプログラムを効率良く作成できるような機能を備えている。

### 3・4 テストツール[8]

プログラムのテスト工程では，誤りの検出を目的としたテスト作業とその誤りの原因追求および除去を目的としたデバッグ作業の双方の効率化が重要である。

デバックを支援するツールはデバッガと呼ばれ，アセンブラプログラムを主対象として，機械語レベルでのデータ値の表示と設定，実行の中断や命令トレースなどの機能を備えたものが長く用いられてきた。最近では高級言語への移行とともにソースプログラム内の変数名や名札を用いてコマンド指示のできるシンボリックデバッガが一般化している。

一方，テスト作業についても，ソフトウェア工学の発展とともにその重要性が認識され，テストを系統的に行なうための支援システムが実用化されている。その代表的機能として次のようなものがある。

（1） 単体（モジュール）テストのための環境模擬機能

（2） 入力データと予想結果およびテスト環境など

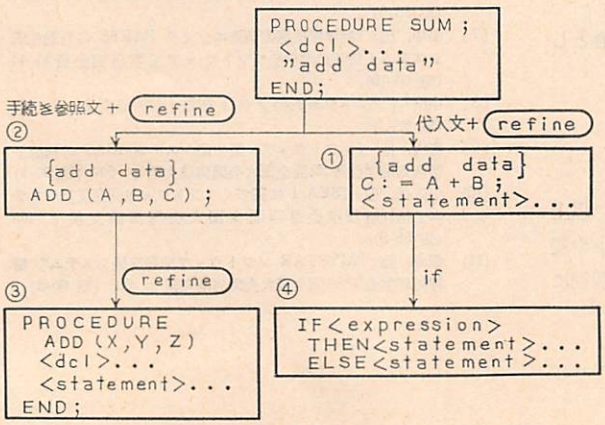



図 2 構造エディタ PARSE の段階的詳細化機能

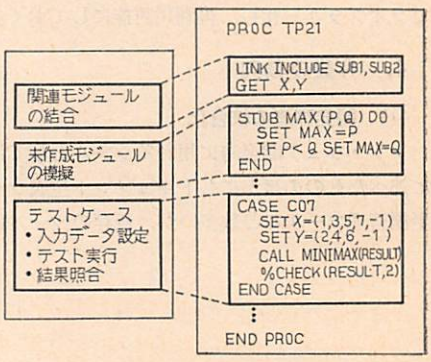



図 3 テスト手続きの例

を記述するテスト手続き言語

（3） 構造テスト支援機能

単体テストによるプログラム誤りの検出，修正費用は，既に図1で示したようにシステムテスト時に比べ1けた少なくて済む。その単体テストの実施のためには，未作成の上位モジュールの代りとなるドライバや下位モジュールの代りのスタブなどのテスト環境模擬機能が必須となる。そして，このようなテスト環境および入力データや予想結果などはテスト手続きとしてまとめて記述しておくことにより，ライブラリー化して再利用したり，テスト作業を自動化することができる。図3に筆者らの開発したテスト手続きの記述例を示しておく。

一方，このようなテスト実行と並行して，プログラム内のどこが既にテストされたかというテスト網羅情報を収集，分析して，未実行部分のテストを促す構造テスト支援機能もプログラムの品質向上に不可欠である。すなわち，被テストプログラムの品質は，もし誤りがあれば必ずそれを検出できる効果的なテストデータを用いたか否かに依存するため，網羅的テストを行なう必要がある。

### 3・5 保守ツール

保守とは，システムの運用開始後に検出された誤りの修正や機能拡張のためのプログラム変更作業である。大規模ソフトウェアでは保守費用が開発費用を上回り，更に増大している。しかしながら，保守作業の多様性のために技術の進歩は遅れている。現状では，開発時に保守容易性を考慮して，次のような対策をとっておくことが重要である。

（1） 設計時に仕様書をきちんと作成するとともに常にプログラムとの一致を図っておく。

（2） プログラミング時に構造化を心がけ，ソースプログラム自身のドキュメント性を高めておく。

（3） テスト時にはテスト作業をテスト手続きとしてライブラリー化し，再利用可能にしておく。

## 4. 今後の動向

### 4・1 開発環境の統合化

ソフトウェア開発時に用いるツールとしては，以上に述べたもののほかにもドキュメント生成ツールや変更歴管理ツールなど数多いが，既存のツールは個別に開発されたものが多く，複数のツールの組み合せ使用が必ずしも容易でない。最近は，データベースの共有化やユーザインタフェースの一様化に基づき，要求定義段階からシステムテストまで一環して支援する統合ソフトウェア開発環境が普及し始めている。その例としては，ICAS[9]（日立），SEA/I[10]（日本電気），MYSTER[11]（東芝）などがある。

### 4・2 知識工学応用

これまでのソフトウェア工学的アプローチに基づいて構築されてきたソフトウェア開発環境は，ソフトウェアライフサイクルの各段階で有効と思われる方法論や技法を定式化し，ツール化することによって作業効率とプログラムの品質を改善するものであった。しかしながら，このような方法だけでは，ソフトウェアが今後ますます複雑化，大規模化するにもかかわらず，その開発を担当できる経験者の数が限られているという問題を十分解決することはできない。今後は，現在研究の盛んな知識工学的アプローチを導入し，ソフトウェア開発の専門家のノウハウを計算機に組み込むことにより，ソフトウェア生産自動化率およびソフトウェア部品再利用率などを向上させていく必要がある。

（昭和60年11月20日受付）

## 文　献


（1） 中所：「プログラミング言語とその会話型支援環境」情報処理 **24**, 6, 715（昭 58-6）

（2） 中所：「ソフトウェア工学」コンピュータソフトウェア（日本ソフトウェア科学会）**1**, 2, 92（昭 59-7）

（3） A. R. Sorkowitz: "Certification Testing; a Procedure to Improve the Quality of Software Testing" *Computer* **12**, 8, 20 (1979)

（4） B. W. Boehm: "Software Engineering Economics" (1981) Prentice-Hall, Englewood Cliffs

（5） M. V. Zelkowitz: "Perspectives on Software Engineering" *Computing Survey* **10**, 2, 197 (1978)

（6） 中所，他：「段階的詳細化，データ抽象化を支援する言語 SPL のコンパイル技法」情報処理学会論文誌 **21**, 3, 223（昭 55-5）

（7） 田中，他：「計算機誘導型構造エディタ PARSE の自動生成システム」情報処理学会ソフトウェア工学研究会資料 44（昭 60-10）

（8） 中所：「ソフトウェアのテスト技法」情報処理 **24**, 7, 842（昭 58-7）

（9） 青山，他：「ソフトウェア一貫生産システム ICAS の構成」情報処理学会第30回全国大会講演論文集 p. 597（昭 60-3）

（10） 小久保，他：「SEA/I に基づくソフトウェア開発支援システム」情報処理学会第29回全国大会講演論文集 p. 469（昭 59-9）

（11） 飯塚，他："MYSTAR ソフトウェア開発支援システム" 情報処理学会第30回全国大会講演論文集 p. 583（昭 60-3）