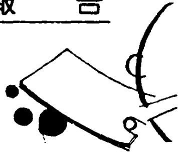
**報　告**



# パネル討論会
# ソフトウェアメトリクスの現状と課題
## 昭和58年前期第26回全国大会† 報告

パネリスト
片岡　雅憲[1)]，花田　収悦[2)]，寺本　雅則[3)]
吉田　　征[4)]，大場　　充[5)]，司会　藤野　喜一[6)]

### はじめに

ソフトウェアメトリクスという用語が最近盛んに使われるようになってきました．今大会でもソフトウェアエンジニアリング分野の発表 61 件のうち，この領域に関係するものが 15 件になり研究が進んでいることを表わしています．

ソフトウェアメトリクスとは，ソフトウェアの生産性と品質向上のためにソフトウェア作業に関する環境条件および活動の状況を科学的に計測・把握することと，その結果えられたデータに基づいて分析し，生産物，あるいはサービスを作り出す作業のプロセスを制御，改善していく基本的活動の一つであるといえます．ソフトウェア作業にはいろいろな資源が必要ですが，主なものには，人＝工数，物＝コンピュータなど，時間などがあります．これらは投入資源であり，また作業の結果としての生産物やサービスサポートの生産性や品質を評価する方法が必要になります．

このためには，まずソフトウェアメトリクスとして生産性や品質を測定する尺度を決めること，その次にどういうようにそれを測定するかということが必要になります．また測定の対象としてどんな作業項目があるのか，プロセスにはどんなものがあるかを明確に定義しておくことも大切です．このようにデータを収集したあと，それを評価分析するためのモデルが必要です．

いままで研究開発されているモデルには生産性ではコストモデル，品質関係での品質や信頼性モデルとよばれるものがあります．

また作成の対象となるプログラムの特性として複雑さの問題がありますが，これには論理的構造からみたものと，物理的な構造からみた複雑さと二つの種類があるのではないかということ，また複雑さにも関連することですが，人間はどうしてミスをするのかというようなことも大変重要な問題であると考えられます．

コストモデルについても，多くの研究が行われていますが，ベームやバシリなどの研究結果が発表されています．コストモデルは対象とするソフトウェアの種類や規模，その特性，環境，方法，ツールなど生産性要因とよぶ要素をデータ化して，モデルに与えることにより，必要な工数とか期間などがでてくるものです．その結果と，モデルにより得られた生産性の基準とをあわせて評価していくことにより，作業環境や方法などにフィードバックして改善をはかることができます．すべてのソフトウェアに共通に利用できるようなコストモデルを作ることはむつかしく，ソフトウェアの種類や規模や部門などによって，異なるモデルを利用することが，自然の方向といわれています．

また品質モデルはデータをとって，実際にバグがどれだけ発見できたか，さらにどのくらい検査する必要があるのか，といった予測を行うものである．この場合，検査に要する時間だけでなく，検査項目数や，検査工数などをパラメータとしてみるとか，研究が行われています．工程からみると後工程ほど，品質面のトラブルの修理時間が大きくなるといわれています．いずれにしてもデータを収集することが基本であり，できれば，必要なデータを自動的に収集する方法の研究も各所で行われているといわれています．

例えば，開発工数も新規作成の場合と流用が多い場合とでは，必要工数も異なることがわかってきており，この辺については詳しい研究について，お話しがあることになっています．

† 日時　昭和58年3月17日(木)，12：30～15：00
場所　東京工業大学講堂
1) 日立，2) 横須賀通研，3) 日電，4) 富士通，5) 日本 IBM，6) 日電

以上開発フェーズに関係することが中心でしたが，セールスサポート，ポストセールスといったいわゆるSEが関連する部分の生産性や品質の問題もあります．またプロジェクトの管理者もプロジェクトの計画立案，実行段階でのデータ収集，分析による適切な対象などを考えていくことが大切ですが，ソフトウェアは人間の知識作業に負うところが多いので，人間的な面からの配慮に基づく環境改善なども重要です．

まだソフトウェアメトリクスは緒についたというところであり，今後一層の努力が必要ですが，その一つには，要求仕様が定まったときに，開発すべきソフトウェア規模の見積りの予測手法の研究をあげることができます．

ソフトウェア作業は，ハードウェアに比較して，目に見えにくいと言われているわけですが，最近のようにいろいろな試みが行われることによって，ずいぶん見えるようになってきていると思われますが，これを一層進歩させるためにも，ソフトウェアメトリクスは一段と重要な役割をはたすようになると予想されています．

今回はソフトウェアメトリクスについて第一線で活躍しておられる方々に特にお願いして，パネル討論会を開くことができました．非常に新しい分野のため，今回の討論会がソフトウェアに関係する方々にとって，何らかのお役に立つことができればコーディネータとして大変嬉しく思います．ご協力いただいた方々に改めてお礼を申し上げます．

発言順にパネラをご紹介いたしますと，

片岡（日立）さんから「ソフトウェアの構造とテスト十分性指標」

花田（横須賀通研）さんの「改造率と生産性に関する計量事例」

寺本（日電）さんの「コストモデルの適用」

吉田（富士通）さんの「ソフトウェアの計量化：事例と実感に違いはないか」

大場（日本IBM）さんの「ソフトウェア品質測定モデルと問題点」

となっています．

質問・討論は最後に載せてありますが，中山さん(富士通)には吉田さんのプレゼンテーションをデータを使って補足していただき，討論の盛り上げに大変ありがとうございました．

当日はあいにく大変寒い日で，パネラの方々もふるえながらの発表・討論でしたが，本当にご苦労さまでした．

（司会　藤野　喜一）

## ソフトウェアの構造とテスト十分性指標

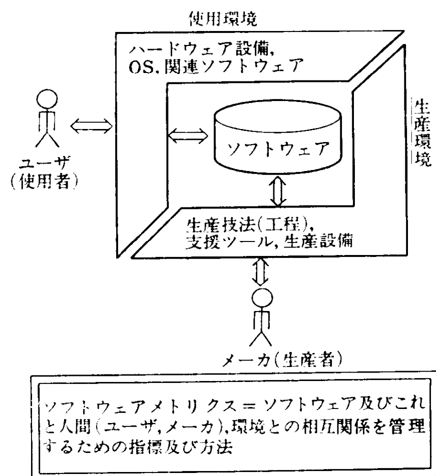
片岡　雅憲

### 1. ソフトウェアの生産技術とメトリクス

ソフトウェアの生産技術の改善方法について，2つの典型的な意見がある．1つは，「生産技術改善の原理・原則はソフトウェアであっても，ハードウェアのそれとまったく変りはない」との信念であり，もう1つは，「ソフトウェアの生産技術は，それ独自のものが構築されるべきである」との主張である．

両者は一見，対立しているが，現実を良く掘り下げてゆくと，両方の面があることがわかる．QC（品質管理）や，構造的アプローチのソフトウェアへの適用は，まさしく，前者の考え方に基づくものである．一方，「ソフトウェアの生産工程は，人間の運動系（肉体労働）よりも，情報処理系（精神労働）への依存性が極めて高いという点でハードウェアのそれと異なる」との後者の主張も十分に尊重されてしかるべきである．今日，われわれに求められているのは，上に述べた2つの考え方，すなわち，伝統的な知恵と新しい思想との融合であろう．

ソフトウェアの環境を簡単にモデル化すると**図-1**のようになる．ソフトウェアは，利用者であり生産者である人間の活動の成果物であり，また，両者の間には使用環境，生産環境が介在している．このように見たとき，ソフトウェアメトリクスとは，「ソフトウェ




ソフトウェアメトリクス＝ソフトウェア及びこれと人間（ユーザ，メーカ），環境との相互関係を管理するための指標及び方法

**図-1**　ソフトウェアメトリクス

ア，人間，環境の3者，及びその相互関係を科学的に管理するための指標及び方法論」ととらえることができる．そして，ここで特に大切なことは，3者間の相互関係を系統的，構造的にとらえて明確化し，改善してゆくことである．以下，本稿では，このようなとらえ方の一例を述べることとしたい．次章では，人間とソフトウェアとの関係の例として，人間の情報処理系とソフトウェアの構造との関係について述べる．また，次々章では，生産環境の1つであるテスト方法と，ソフトウェアの構造との関係を述べる．

## 2. 人間の情報処理系とソフトウェアの構造

人間の情報処理系は物理信号処理系と論理情報処理系から構成されている[1]．前者では，視覚・聴覚等の感覚器官により，外界の物理信号を検知し，さらには，この中から文字，語，文等のパターンを抽出し，認識する．一方，後者では，これらの意味を解釈し，知識として記憶し，また，この記憶に基づいて推理する．

人間の精神労働の所産であるソフトウェアの構造は，当然，上記の人間の情報処理系の構造と密接な関係を持っている．すなわち，ソフトウェアの構造を，物理的構造と論理的構造からなる二次元的構造としてとらえることができる．表-1 に示すように，前者はソフトウェアの物理的形状の構造であり，管理指標としては，ソースカード枚数，オブジェクトコード長，モジュール数，ドキュメント枚数等がある．一方，後者はソフトウェアの論理的意味の構造であり，管理指標としては，データ要素数，機能要素数，これら相互の関係演算子数，また，これらを総合的に評価するエントロピ[2]等がある．

表-1　ソフトウェアの物理的構造と論理的構造

| 項番 | 項　目 | 物理的構造<br>(物理的形状構造) | 論理的構造<br>(論理的意味構造) |
|---|---|---|---|
| 1 | データの構造 | データ形式，データ長，記憶域上のレイアウト，データ間のポインタ | データ要素の意味，要素間の意味的関係(包含関係，排他関係，直積関係等) |
| 2 | 操作の構造 | モジュールと入力・出力データの関係，モジュールの親子関係，モジュール間の制御関係 | 機能要素（入力・出力データ要素間の関係），要素間の意味的関係 |
| 3 | メトリクス | ソースカード枚数，オブジェクトコード長，モジュール数，ドキュメント枚数 | データ要素数，機能要素数，要素間関係演算子数，エントロピ |
| 4 | 人間の情報処理との対応 | 物理信号処理(五感，パターン認識系等) | 論理情報処理（命題論理，述語論理等） |

このような二次元的な構造のとらえ方は，ソフトウェア生産の全工程に一貫して適用できる．そして，ともすれば物理構造主体となりがちな，従来の一次元的構造把握法に比べ，より効果的な生産技術と管理指標を可能とする．

## 3. ソフトウェアの構造とテスト十分性指標

ソフトウェアの構造と人間の情報処理系が先に述べたように関係づけられるとすると，両者の間に介在する生産環境もこれらと無関係であり得ない．ここでは，特に，ソフトウェアの生産技法のうちテスト技法をとりあげ，ソフトウェアの構造との関係を考えてみることにしよう．

テスト十分性指標は，テストの十分性を客観的に評価するための指標であり，テストの方法と質を定める重要な指標である．この指標をソフトウェアの構造と対応づけると表-2 のようにまとめられる．表に示すように$F$指標と$C$指標によるテストは各々の特徴を持ち相補的関係にある．すなわち，$F$指標＝100%は，$C$指標＝100% を意味せず，逆も成り立たない．優れたテストを行うためには，両者を適切に組み合わせる必要がある．

日立製作所では，上記$F$指標に基づきテスト項目を作成し，テスト実行時に$C$指標を測定するという系統的テスト技術を実用化し，活用してきた．そして，この中には，$F$指標に基づくテスト項目作成支援ツール及び$C$指標に基づくテスト十分性自動測定ツールが含まれている[3]．これら自動化ツールに支援された系統的テスト技術は，不良の早期摘出に大きな効果をあげている．例えば，従来方法では，単体テスト工程での不良摘出率（単体テスト工程での摘出不良数÷全テスト工程での摘出不良数）が約 30%（1979年データ）であったのに対し，$F$指標，$C$指標＝100% とした系統的単体テストでの不良摘出率は約 90% と大幅に改善された．

## 4. おわりに

実用的なソフトウェアメトリクスを開発するに当

表-2　テスト十分性指標と系統的テスト支援ツール

| 分　類 | 十分性指標(例) | 特　　徴 | 支援ツール(例) |
|---|---|---|---|
| 物理構造に基づく尺度 | $C_0$：命令実行比<br>$C_1$：分岐実行比 | 内部構造上の特異点も含めた網羅的テストができる | テスト十分性自動測定ツール |
| 論理構造に基づく尺度 | $F_0$：機能要素実行比<br>$F_1$：機能関係実行比 | ユーザの観点からの厳密なテストができる | テスト項目作成支援ツール |

たって，本稿に述べたような構造的アプローチが大切である．このアプローチでは，ソフトウェアの構造を二次元的にとらえ，人間の情報処理系と関係づける．したがって，ソフトウェアと人間とのインタフェースである生産工程，支援ツール等のソフトウェア生産環境を総合的に把握するのに極めて有効であると考えている．

### 参考文献

1) Klatzky, R.L. 箱田他訳：記憶のしくみ，サイエンス社 (1982).
2) Halstead, M.H.: Elements of Software Science, Elsevier North-Holland (1977).
3) 片岡他：ソフトウェア開発支援システム (CASD システム)，日立評論，Vol. 62, No. 12 (1980).

## 改造率と生産性に関する計量事例

花田　収悦

### 1. はじめに

最近ソフトウェアの生産性向上等の観点からソフトウェアの再利用指向が強まっており，この傾向はしばらく続くものと思われる．したがって，保守のフェーズはもちろんのこと，ソフトウェアの開発時においても既存のソフトウェアをベースに改造を主体とする作成方法が頻繁に採用される．

このような既存のソフトウェアを利用する場合に，利用する既存のソフトウェア（母体）の規模，修正（改造）個所の局所性，改造母体のモジュール構造などが生産性（作成コード量／工数）に影響する，と言われてきた．

本稿では，このうち改造率（作成コード量／母体のコード量）と単位作成コード量あたりの生産性に関する計量事例を示し，若干の考察を行う．

### 2. 計量環境と計量事例

ソフトウェアの生産性に関するデータについては計量環境を十分に把握することが重要であり，それなくしてはデータの流用や利活用が難しい．生産性のデータから結論（数値）のみを流用することは危険であり，自社の生産環境にマッピングした上で利活用するのが正しいマナーであろう．逆に，そのようなことが可能であるような様式にしてデータを公開するように互いに留意しあうことが今後の発展につながる．

#### 2.1 計量環境

（1）作成コード量の計量

改造時の作成コード量は，①新規に追加したコード量($A$)，②更新したコード量($B$)，の和 ($A+B$) で計量する．改造作業においてはこの他に削除するコード量が存在するが，これまでの経験によれば量的には大きくない，という理由で無視している．

コードの単位はステートメント（文，行）であり，コメント行は計量しない．

コメント行は，特にソースプログラムの理解性の向上には有効なので計量の対象にすべきであるという主張はあるが，有効なコメントと無効なコメントの判定が煩わしい，コーディング等の作業要領でコメント行の挿入を指導しており平均的にはソースプログラムの3～4 割程度のコメント行が存在する，等の理由からまだ計量対象にはしていない．しかし，最近ではソースプログラムから保守用ドキュメントを自動作成するツールなどが出現しており，この場合にコメント行の有無が理解性に大きく影響するという事例もあるのでいずれはコメント行を計量対象にすべきであると考えている．

（2）工　数

改造母体の理解に要する工数や，各種ドキュメントの作成工数，およびその他の製造工数（コーディング，テスト，デバッグなどに要する工数）を含める．

特殊な評価作業やそのプログラムについて他人に教育するための工数などは含めていない．

（3）その他の生産環境

コード量と工数以外の計量上の主要な生産環境は以下のとおりである．

① 改造の局所性，および母体のモジュール構造の良さ，については考慮していない（できない）．

② 担当者のスキル，改造母体についての知識は同等と仮定する．

③ 計量対象のプログラムは言語処理プロセッサ（コンパイラなど）．記述言語は PL/I 類似言語(SYSL : SYStem description Language)．

④ コードの作成量は 1000 行以上．母体規模は 50～350 K 行．

#### 2.2 計量事例

12 の言語処理プロセッサの開発事例のデータを表-1 に示す．

12 のデータのうち性能向上のために種々の評価作業と手直しを実施したケース 7 を特異データとして除外した 11 のデータを統計処理して近似式を求めると次のようになる．

（1次式近似）$Y=-2.8X+3.8$

表-1 改造率と生産性の測定データ

| ケース | 改造率 | 工数比 |
|---|---|---|
| 1 | 0.04 | 4.7 |
| 2 | 0.16 | 3.2 |
| 3 | 0.23 | 3.3 |
| 4 | 0.27 | 1.8 |
| 5 | 0.32 | 3.5 |
| 6 | 0.35 | 2.4 |
| 7 | 0.63 | 4.1 |
| 8 | 0.64 | 1.8 |
| 9 | 0.73 | 1.9 |
| 10 | 0.90 | 1.9 |
| 11 | 1.00 | 1.1 |
| 12 | 1.00 | 0.9 |

注 1) 改造率＝新規作成コード量／新規作成コード量＋流用コード量（総規模）
2) 工数比はケース11と12の平均値を基準（1.0）としている.
3) ケース7は特異データである（理由は本文に述べる）.

（2次式近似） $Y=2.9X^2-6.0X+4.4$

（3次式近似） $Y=-12X^3+22X^2-14X+5.1$

ここに，$X$：改造率，$Y$：$X=1.0$ の時の1K行あたりの工数の平均値（すなわち，ケース11と12の平均値）を1（基準）とした場合に，新規作成コード1K行あたり，所要工数の相対値である．改造率＝(新規作成コード量)÷(新規作成コード量＋流用コード量)で表わす．したがって，$X=1.0$ とは全面新規作成の開発形態を指すので，その平均値を基準工数に採用した相対値（比）によって$Y$を表現する．

3つの近似式のうち，2次式近似（$Y_2$ と表わす）が妥当性がある．1次式近似式では粗すぎる．またグラフ表示してみるとわかるが，3次式近似では $X=0.8$〜$0.9$ 付近で $Y=1.9$〜$2.0$ に漸近して $X=1.0$ で $Y=1.0$ に落ちこむ理由の説明がつきにくい．結局のところ指数関数的に漸減する $Y_2$ を中心に，$\pm\alpha$ (0.5〜1.0) のバラつきの範囲内の値 ($Y$) をとる，と現時点では解釈することとしたい．

よってデータから$X$と$Y$との関係は母体に対する改造比率が小さいほど，1K行あたりの相対工数は嵩むことになり，例えば $X=0.1$ 付近では $Y=4$〜$5$ となる．

これは全体を新規作成する場合の1K行の開発に要する工数に比較し，改造率10%程度の改造による開発では同じ1K行規模の新規作成に要する工数が4〜5倍も大きくなることを意味する．

その最大の原因は母体ソフトウェアのモジュール構造が複雑で，かつモジュール間のインタフェースなどが整理されていないために，改造個所に限定した局所的なテスト・デバッグが行えず流用部分を含む全体規模を対象としたテストなどを実施するためと解釈される．また，テスト工程の終了近くでインタフェースのもれやミスが検出され再度のテスト・デバッグを実施するなどの事例も多い．モジュール構造の良さを定量化する技術が切望される（図-1）．

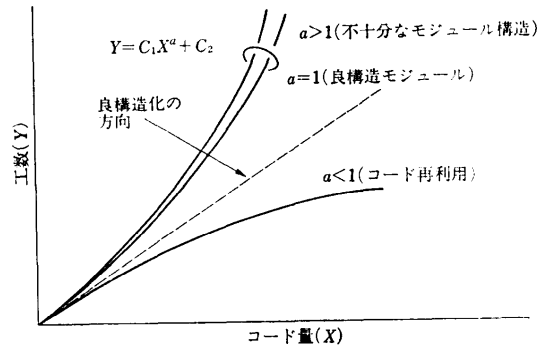



図-1 良構造化に着目したコード量と工数との関係（概念図）

## 3. まとめと今後の課題

本稿では，言語処理プロセッサ（コンパイラなど）の11の開発事例に基づき，改造率が小さい場合（1〜2割）には，全体の新規開発の場合に比較し，同量のコード量を作成するのに4〜5倍の工数が嵩むことを示した．これらのデータを開発計画に反映して工数の過少見積りにならぬようにすることが肝要である．

今後の課題としては

（1） 他種プログラム（制御プログラム，応用プログラムなど）における改造率と工数との関係を表わすデータの収集と分析．

（2） モジュール化の良さを評価できる尺度の確立，およびその計量技術．

（3） （改造）規模の見積り技術．

（4） 計量データの統一的様式による公開・交流の促進．

などがあげられよう．いずれも難問であり地道な取組みが必要である．

最後に，現時点におけるわが国のソフトウェアメトリクスを概括すると未だ揺籃期にあり，ようやく基礎データの収集・分析に着手した程度と認識されること，したがって現状ではソフトウェアメトリクスを軽視しても過信してもいけない，という私見を述べて終りとしたい．

## コストモデルの適用

寺本　雅則

コストモデルとは観測もしくは推定可能な管理データからソフトウェア開発のコストすなわち工数を導き出すために数式，表，プログラム等によって実現された関数であり，実際の値を得ることができるものをいう．近年，ソフトウェア開発プロジェクトの巨大化や多様化，費用の高騰，納期の厳守要求，あるいは見積りや契約内容の具体化などの管理上の理由のほかに，ソフトウェア生産の計量的把握を行って生産性向上のための対策を集中化し効果を高めようとする動きがある[1]．以下，コストモデルをこのような視点から適用する場合のモデルの特殊化の考え方と試行結果について述べる．

### 1. コストモデルの形式と意味

ここで扱うコストモデルの形式は以下のようなものである．

$$Y=\alpha_1\cdot\alpha_2\cdots\cdots\alpha_n\cdot C\cdot X^{\beta} \quad (1)$$

ここで

$Y$: 開発工数

$X$: 開発規模（ソース行数など）

$C$: 定数項

$\alpha_i$: コスト因子の重み（中心値1）

$\beta$: 開発規模に依存する係数

であるが，コスト因子としてはたとえば**図-1** のようなものが考えられる．(1)式によればソフトウェアの開発コストは開発規模とそれ以外の種々の因子の重みの積から得られ，因子の重みの変化によって現実によく合致するものとなり得ると考えられる．ここでこの因子の重みをどのように決めたらよいかという問題が生じる．このため従来多くのデータにもとづき数々のモデルが提案されているが，これらを適用面から見ると以下のふたつの形態に分けることができる．ひとつは統一的モデルとでも言ったらよいようなアプローチで，多くの条件をできるだけ考慮し汎用的に使えるモデルを構成しておき，使用者は対象となるソフトウェアとその開発時の種々の状況すなわち各因子の重みに対応する指標を想定できればすぐにモデルによる開発コストの計算ができるようなものである．この代表的なものが Boehm の COCOMO[2]である．

もうひとつはメタモデルというべきアプローチであり，モデルの基本型のみを設定し，利用者自らのデータによってモデルを精密化し現実（といっても自分のところに限られるが）との適合性を向上させていくものである[3]．

統一的モデルの使用にあたってはソース行数や工数などの計測基準がモデルの想定しているものと一致しなかったり，モデルに盛り込まれている因子以外の要因がコストを左右している（あるいは適用部門の人々がそう感じている）場合には説得性を失うことになる．事実過去に発表されたこのような多くのモデルのあいだには，対象としたデータの母集団の性格の相違にもとづくとおもわれる差が認められている．よって多少データ収集の手間がかかっても，自らのデータと計測基準によってメタモデルを特殊化して使うことにより，以上の問題を克服できると考えられる．

### 2. メタモデルの適用

このような考えから(1)式のようなメタモデルを設定し，自部門のデータによってその部門固有のモデルを導出する試みが社内で行われている．このモデル導出の手順は①収集した過去のデータを適当な変換ののち重回帰分析をはじめとするいくつかの統計的評価を行い，②採用された因子のうちコストへの影響が大きくかつお互いに独立性の高い因子を抽出する．③新しく抽出した因子により原データを再び重回帰分析

- 新規性・難易度
- 仕様変更度
- 対 OS，対ハードのインタフェース
- 責任発注形態
- チームのスキル，能力
- 開発技法，手法
- 品質要求
- 製品（ソフト）の汎用性
- ユーザ特性
- メンバの経験
- 開発ツール

**図-1**　コスト因子の例

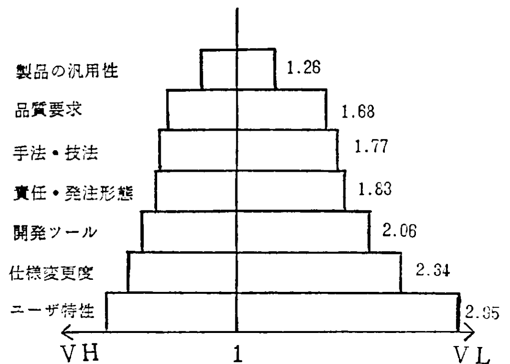



**図-2**　生産性レンジの分布図

にかけ部門専用のモデルをつくる. ②③はモデルが現実をよく説明しているとおもわれるまで繰り返される. 以上の手順は通常多大な計算量と人間の介入を必要とするため①から③までを対話的に行えるシステムをパソコン上でインプリメントして使用している. このモデル導出の結果としてコストモデル式のほかに各因子の重みを変動させたときのコストが変化する割合すなわち生産性レンジの分布図 (図-2) なども得られる. この図では, たとえば手法・技法という因子に関して最悪値 (*VL*) と最良値 (*VH*) とのあいだでは他の条件が同一だとすると 1.77 倍の生産性のひらきがあることを示している. モデルの導出を行う人々は自部門のソフトウェア開発工数の見積り式を得るだけではなく, 上記のような因子ごとの生産性の変動分を知ることにより, どの要因分野に改善手段を集中して講じたらどの程度の生産性向上が期待できるかを判断できるようになる. これがコストモデルが単に見積りの道具だけではあり得ない理由である.

いっぽう, 以上の試行をつうじていくつかの問題点も明らかになった. 一例として負の回帰係数の出現は単にデータが不正確というだけでは済まされない場合がある. たとえばある因子が負の係数を持ったということはスキルの高い人が投入されたプロジェクトは生産性が他よりも低くなると説明できるとしよう. 常識から言えばこれはおかしいが, よく事情を調べるとその部門では非常に難しいプロジェクトに特にスキルの高い人を投入する傾向があり, それにもかかわらずその場合のコストは結果として他のものより割高になってしまうのが通例だとすれば負の係数が正しいといえる. あるいは全く考慮していなかった因子によってコストが大きく左右されていた場合も解析結果だけからはデータの適否は判断がつかなく, 綿密な実情調査が必要になる. またこの試行に参加した人々からは特にデータ収集・蓄積方式は今後とも改善が必要であり, かつ因子を計測する際の基準の明確化が重要という意見が寄せられ, コストモデルの浸透により計量化も進歩することがうかがえる.

### 3. 結 論

以上からコストモデルの分野について体験からいえることは,

1. コストモデルは自らのデータで作成し, これを改善していくことにより精度の良いものができる. このためには通常から一貫性のあるデータ収集が必須.

2. モデルに都合の悪いデータは無視せずその原因を追求することが大切であり, その結果新たな知見を得ることが多い.

3. 統計学の正しい理解 (必ずしも広範または専門的でなくてもよい) と活用方法を普及するための教育が重要.

などである.

### 参考文献


1) Boehm, B. W.: Improving Software Productivity, Proc. COMPCON FALL 81, pp. 2-16 (Sep. 1981).
2) Boehm, B. W.: Software Engineering Economics, Prentice-Hall, Inc. (1981).
3) Baily, J. W. and Basili, V. R.: A Meta-Model for Software Development Resource Expenditures, Proc. ICSE, Vol. 5, pp. 107-116 (Mar. 1981).


## ソフトウェアの計量化: 事例と実感に違いはないか (統計的な検証)

吉田 征

ソフトウェアの開発状態を計量化しようという事例は, 毎年数多く発表・報告されている. 我々ソフトウェアの検査部門という開発現場に在る者にとっては, それらの事例がいかに理論的であるかというよりも, 開発の実体と食い違いが無いかという方がより重要である. 本報告では, 検査部門というソフトウェア開発現場に近い立場から, 我々が開発に携っている百数十

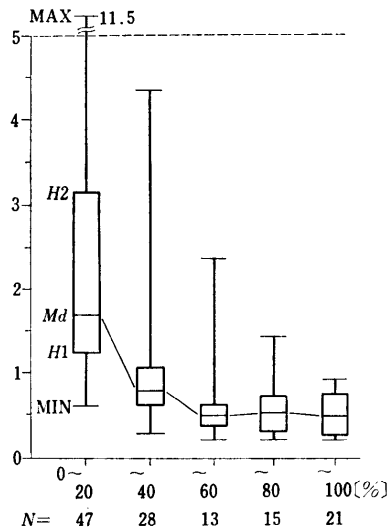



図-1 開発量の新規率と生産性 (含保守)

のプロジェクトを調査し，ソフトウェアメトリクスが現場に適用し得るかどうかについて統計的に検証してみた．

<統計的に扱った意味>

—我々の対象としているベーシックソフトウェア・プロダクツは，多少の難易度の差はあっても，

- 開発環境，
- ツール，
- プログラミング手法，
- 教育状況，
- 信頼性目標

が同様であり，統計的に扱っても問題ない．

—多くのプロジェクトを長期にわたって取り扱ったデータであるため，

- 経験や個人の能力差を均一化して評価できる．
- また，特殊条件のプロジェクトは浮き彫りになると同時に，場合によっては除外できる．

### 1. 開発の新規率と生産性

「既存プログラム（モジュール）の再利用とか，部品化されたモジュールを利用することにより，生産性が向上する.」と言われている．それに対し，経験あるプログラマからは「下手に流用・改造するより新しく作り直した方が良い.」という主張が強く，開発方針の判断に苦慮するところである．

生産性をステップ数／人月で定義し，新規に開発した部分がプロダクト全体に占める割合である新規率との関係を見ると**図-1**のグラフが得られた．新規率を20%ごとに区分して，生産性については5分点表示をとっている．最大値を結んでみると分るように，何らかの方法で効率の良い開発を行っているプロジェクトは，既存のプログラムを利用したことに比例して生産性が高くなっていることが分る．しかし，中央値を結んだ線を見ると，平均的に開発が行われているプロジェクトでは，既存の部品とかプログラムを徹底して（70〜80%以上）利用しないと生産性は向上しないという結果が出ている．

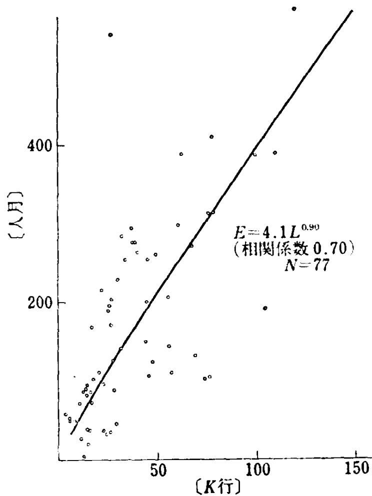



**図-2**　ソフトウェア開発量と投入工数

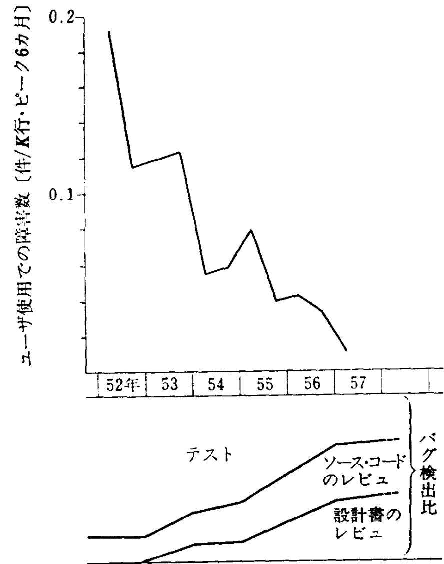



**図-3**　信頼性向上のためにはレビュが効果的

### 2. 開発量と投入工数の関係

一般にソフトウェア開発のための工数は，開発規模に対し指数関数的に増大すると言われてきている．

$E=aL^b$（$E$：投入工数，$L$：規模）

としたとき，$b=1.5$ という値の報告は多く，一般に $b$ の値は 0.8〜1.8 であるという報告が見られる．**図-2** に当部門の開発における散布図を示す．

$E$：プログラム開発のための総投入工数

$L$：新規に開発された分のプログラムステップ数

この結果は，$b=0.9$ であり，ほぼ比例関係（$b=1$）にある．これは，開発規模に比例した工数投入というのが現実であり，$b=1.5$ のような工数投入は行えないのが実体であることを物語っているといえる．

### 3. 信頼性向上のための効果的な方法

「プログラムを開発すれば，人間が作るものである限り，ある確率でエラー（バグ）が必ず入り込む．そのバグをいかに早く，いかに効率良く取り除くかが，ソフトウェアの信頼性を効率良く高めることである.」ことは常識となりつつある．

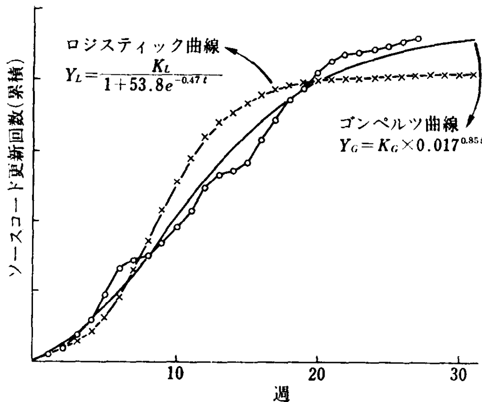



図-4 対話型図形処理プログラム開発の例

これを検証するために，ソフトウェアの信頼性を表わす一つの指標として，ユーザ使用での障害件数で計ってみた．図-3 に示すように，54 年以降の出荷製品の障害件数が激減してきている．これは，組織的に取り組まれてきた設計書レビュ，ソース・コード・レビュと結びつけて評価すると説明がつく．

また，開発工程をプログラミング工程以前とテスティング工程以降とに分けて考えたとき，前工程でのバグ検出比が高いほど，信頼性の高い製品が出荷できていることが分ってきた．

**4. 自動記録の開発履歴情報から信頼性を予測**

時間軸を横軸としてバグ検出の累積値をプロットして成長曲線を求め，信頼性を予測する方法はよく行われている．しかし，“バグ検出数”を開発者からの申告に頼らず，自動的に収集できることが望ましい．

我々は，GEM によりプログラム開発管理を行っており，更新回数，投入行数，削除行数などの開発履歴情報を自動的に収集できる．そこで，GEM データの中で，バグ件数の代りにソース更新回数に着目し，累積更新回数を成長曲線に当てはめてみた．図-4 にその一例を示す．ゴンペルツ曲線によく合致する．製品検査時の信頼性予測の一つの判定尺度として使用すべく，さらに検証を進めている．

**5. GEM の開発履歴情報のグラフ化による諸分析**

さらに，GEM で収集した詳細な履歴情報を種々の角度からグラフ化して分析する方法を開発した（今大会 1 J-8 参照）．図-5 はその一例である．一つのソフトウェアシステムからランダム抽出した 100 モジュールについて，その総投入行数・有効行数・編集回数の歴史的な変化を示している（有効行数＝総投入行数－総削除行数）．段階的開発（5 段階）が行われたことが明白に読みとれる．修正行数の変化の様子や，歩留り（＝有効行数／総投入行数）の値（76%）から品質に関する情報が得られた．

**6. ま と め**

一ソフトウェアの計量化は，日常の生産活動に密着す

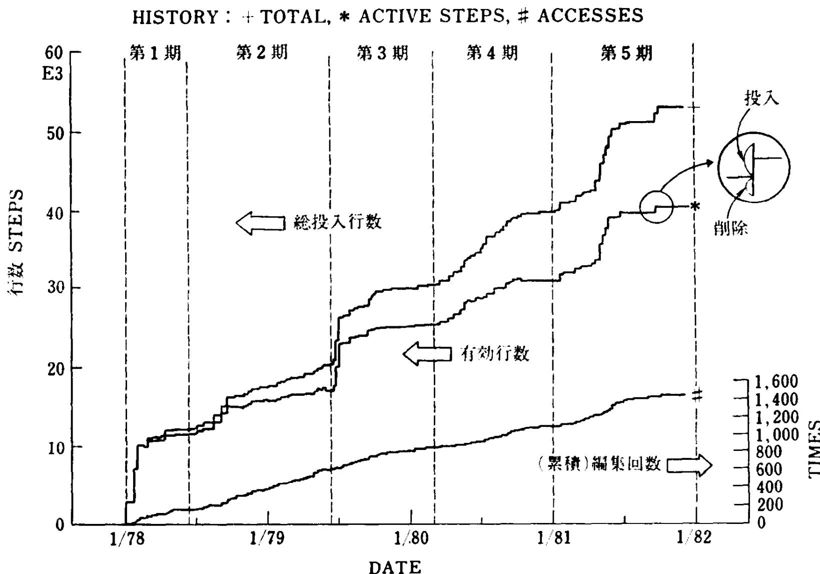



図-5 プログラム開発の履歴

べきものである.(アカデミックなものと泥くささの融合.)

—そのためには,長期にわたってデータを蓄積・分析・フォロすることが大切である.(真理は容易には見つからない.)

—管理のためのデータ収集には限界があり,開発の機械化によるデータの自動収集が必要になってきている.(GEMの活用は有効.)

これらを着実に実行していくことにより,ソフトウェアの計量化が定着し,発展するものと考える.

## 参考文献


1) 宮本 勲:ソフトウェア・エンジニアリング:現状と展望,TBS 出版会.
2) Cook, M. L.: Software Metrics: An Introduction and Annotated Bibliography, ACM SIGSOFT SOFTWARE ENGINEERING NOTES, Vol. 7, No. 2 (Apr. 1982).
3) 伊土誠一他:「Capture & Recapture 法」による潜在バグの推定法とその応用,情報処理学会ソフトウェア工学研究会資料 19-1 (1981).
4) 大場 充:プログラムテストの妥当性評価に関する実験報告,処理装置学会ソフトウェア工学研究会資料 21-4 (1981).
5) Kudoh, S. and Hohya H.: Software Developmemt Management with GEM, Spring COMPOM IEEE 1983.
6) 中川 徹他:ソフトウェア開発履歴分析—GEM記録の活用,情報処理学会第26回全国大会講演論文集 (1983).


## ソフトウェア品質測定モデルと問題点

大場 充

1972年に Jelinski-Moranda のソフトウェア信頼度成長モデルが発表されて以来,ソフトウェア品質の測定や事前評価のための道具としてソフトウェア信頼度成長モデルが研究され,数多くのモデルや推定法が提案され検討されてきた.わが国でも,ソフトウェア品質管理の道具として検討されてきた.わが国では,ソフトウェア品質管理の道具として,ロジスチック曲線やゴンペルツ曲線を応用したモデルの適用例が多い.ここでは,品質測定・評価モデルとしてのソフトウェア信頼度成長モデルについて考える.

### 1. 古典的なモデルの問題点

Jelinski-Moranda, Musa, Littlewood, そして Goel-Okumoto 等のモデルは,信頼度成長曲線としては同型な平均値函数に従う指数型信頼度成長モデルで

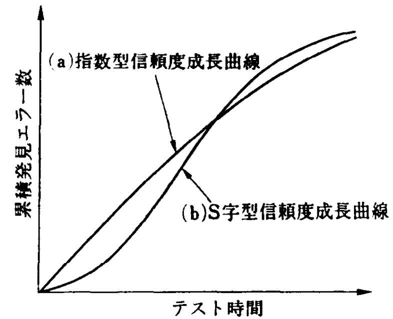



図-1 ソフトウェアの信頼度成長曲線

ある.これは,ソフトウェア・エラー(バグ)の発見が相互に独立であり,かつ各観測時点での発見エラー数が,ソフトウェア中に残存するエラー数に比例するという仮説に基づいている.しかし,現実のソフトウェア・テストにおけるエラー発見過程を観察すると,その信頼度成長は図-1 の曲線 a のような指数型である例よりも,曲線 b のようなS字型である例が多い.そのようなS字型信頼度成長曲線を示すデータに対して,古典的な指数型信頼度成長モデルを適用した場合,そのパラメータ推定によって得られる総エラー数や MTTF の期待値は現実を説明しないことになる.

### 2. ソフトウェアのS字型信頼度成長

ソフトウェア・エラーの発見過程を観測して得られる信頼度成長曲線が,S字型となる例が多い理由として,以下の3つが考えられる:

1) 何をもってエラー発見とするかの解釈の差,
2) テストへの投入労力の不均一性,
3) エラーの相互依存性(非独立性).

上記 1)の問題では,『エラー現象の発見』をもって「エラー発見」とする解釈と,『エラー原因または発生条件の解明』をもって「エラー発見」とする解釈がある.テストへの投入労力がテスト期間を通じてほぼ一定で,かつエラーの相互依存性が無視できる程度であれば,エラー現象発見時刻データの信頼度成長は指数型となる.しかし,エラー原因または発生条件解明時刻データの場合は,発生条件の確認やエラーの原因究明のための時間遅れによって,S字型成長を示す.したがって,時間遅れの影響の大きい初期のデータを除くと,通常の指数型成長で近似できる.一般に,時間遅れを含む信頼度成長に対しては,後述の遅れS字型モデルがよく適合する.

上記 2)の問題は,ソフトウェアの信頼度成長をカレンダ時間で追跡するか,より詳細な実テスト時間

(例えば，労働時間，CPU 時間，テスト・ケース数等）で追跡するかによって，観測される信頼度成長が変化するという問題である．実使用（テスト）時間に対する信頼度成長は，後述のエラーの独立性が保証される場合，指数型となる例が多い．他方，カレンダ時間に対する信頼度成長は，テストに投入される労力の時間分布が一般にテスト期間の後半にピークをもつ『ずれた山型』となっていることから，後述の加速S字型成長となることが多い．

上記 3) の問題は，プログラムの1つの計算経路上に2個以上のエラーが存在するとき，その経路上で先行するエラーを発見することが同一経路上の他のエラーを発見する必要条件になって，指数型成長モデルで仮定する『エラーの独立性』が成立しない例があるという問題である．一般に計算経路は木構造をもつ束を形成しているため，1つのエラーの発見は，そのエラーに従属する一般に2個以上のエラーを発見可能状態にする．システム制御系のソフトウェアでは，テストの最終段階以外，エラーの独立性はほとんど成立しない．このような場合，観測される信頼度成長はS字型となり，後述の加速S字型成長モデルでよく近似される．

### 3. S字型信頼度成長モデル

Jelinski-Moranda によって提案されたモデルを Goel-Okumoto の提案した非定常ポアソン過程（NH-PP）モデルの平均値函数に置換えて表現すれば，時刻 $t$ までに発見されるエラー(累積)数の期待値，$m(t)$，は：

$$m(t)=N(1-e^{-\phi t}), \qquad (1)$$

で与えられる．ここで，$N$はテスト開始時にソフトウェアに潜在した総エラー数であり，$\phi$ はテスト中の単位時間当りのエラー発見率である．

これに対し，エラーの原因究明等による時間遅れを考慮した，遅れS字型成長モデルの平均値函数は次式で与えられる：

$$M(t)=N[1-(1+\phi t)e^{-\phi t}]. \qquad (2)$$

ただし，$\phi$ はテストにおける単位時間当りのエラー発見・原因究明完了率である．(1)式と(2)式の関係は，以下の微分方程式で説明される：

$$\frac{d}{dt}m(t)=\phi[N-m(t)], \qquad (3)$$

$$\frac{d}{dt}M(t)=\phi[m(t)-M(t)]. \qquad (4)$$

次に，加速S字型成長モデルは，エラーの現象発見がそれまでに発見されたエラーの総数に依存する場合のモデルである．その平均値函数は，テスト開始時点で既に発見可能なエラーの総エラー数に対する比率をパラメータ $r$ として，次式で与えられる：

$$\mu(t)=\frac{N(1-e^{-\phi t})}{1+\left(\frac{1-r}{r}\right)e^{-\phi t}} \qquad (5)$$

(5)式はエラー発見率が，発見済みエラー数 $\mu(t)$ の函数，$\log\mu(t)/\log N$，に定数比例する場合の近似モデルとなっている．

### 4. 今後の課題

以上のように，ソフトウェアの信頼度成長曲線は，何をどのように観測するかによって大きく変わる．このため，Jelinski-Moranda が最初に行ったように，われわれはソフトウェア・エラー発見過程の物理的モデルの構築から始めることが重要である．物理的意味の不明確なモデルは，見かけの信頼度成長の傾向を示すだけで，現実を説明することはない．したがって，測定が擬似測定とならないよう，モデルに含意された仮定を検討することが必要となる．

そのような意味で，モジュール構造のソフトウェア，既存のモジュールを利用したソフトウェアの信頼度成長の解明や，エラーの性質を考慮したモデルの構築等が今後の課題である．また特にわが国では，ソフトウェアの信頼度成長モデルとしてのゴンペルツ曲線の物理的意味を解明，または説明することが緊急の課題である．

**質問**（三菱電機　篠原）品質モデルにおいて投入努力一定は前提であり，問題は，いかにそのような状況でテストを行うか，ではないか．

（大場）たしかにそうだ．Musa のモデルはテスト最終フェーズの場合でバグの減少により独立性が高くなって指数型モデルによく合う．しかしプログラムの種類による差もあり，たとえば OS では投入努力をノーマライズしてもS字型になりやすい．

（片岡）変更歴のデータから，変更の多いモジュールと安定したモジュールとのあいだに，機能的・構造的な差があるか．

（吉田）中核となるテーブルや制御部に変更が多いようだ．これらの部分は仕様変更に柔軟でないためかも．

（花田）作ったコードのうちどれくらいが捨てられているか，総投入量と有効量の差が捨てられているの

では.

(富士通　中山) そのとおり. その比が歩留まりである.

GEM に関するデータを少し示す.

- モジュール作成時期／変更.

例えば第3期モジュールは大修正を何回かうけ, 設計変更を示している.

- 歩留まり／作成時期

古いものほど歩留まりが悪いが, それだけでなく機能と歩留まりの関係もある.

- 歩留まり／有効行数

大規模なものほど歩留りが悪いか?→はっきりしない.

(大場) 歩留まりと品質の関係はどうか.

(富士通　中山) 歩留まりはたくさんのうちのひとつのメジャ, それも間接的品質の絶対値をはかるよりも歩留まりの変化から品質変化のモニタリングに利用する.

(日本 IBM 遠山) ソフトの工程管理にハードのようなQC の考え方とソフト独自のファクタが必要な点は同感である. しかしハードは大量生産可能であるのに対し, ソフトは支援システムがあるにしても手工業的要素が大きい. そのギャップをどうするか.

(片岡) ソフト独自のものとハードの考え方は融合すべきである. マスプロ化が可能であるかについてはソフトの物理構造・論理構造に分けて

- 物理…物理的標準化をすすめ, 部品としての供給をすすめることでマスプロ化.
- 論理…創造性の要求される, ソフトのソフトたるゆえんであるから, それぞれが工夫するしかないのでは.

(藤野) 今の質問に関連するがハード側では標準化をきちんと行う「くせ」「しつけ」がついているのでソフトもこれを参考にすべきではないか.

(会場より　高田) コストモデルにおいて過去のソフトは行数がわかっているが新しいもののコスト予測には行数見積りが問題になるのではないか.

また, プロジェクトが中止となった場合などソース行数をメジャとすることの意味はどのくらいあるのか.

(寺本) 行数見積りはさまざまの方法があるが, まだ完全に確立していない. しかし, コストモデルのねらいがコスト予測よりも生産性要因の分析にうつりつつあり, その意味でコストモデルが有用である.

(日立　黒崎) プログラムを流用すべきかどうかの判定には改造率だけでなく, 既存のもののわかりやすさ, いじりやすさ, ドキュメントの程度, 改造者の質など多くの要因があろうが, これらをどうしているのか.

(花田) そのような要因はあまり考えず担当者がつみあげでやっている. 実際には複数のメーカが仕事をしているため, ドキュメントなどは判定要因にはしていない.

前の質問に関連して, 規模見積りは類似ソフトとの比較や経験などによっているが, 経験が意外に正しいことが多い. 見積り変動の原因は, 要求仕様の変更によることが多く, 要求仕様を決めてから見積ることにしている. バラつきは 0.8〜1.4 程度である.