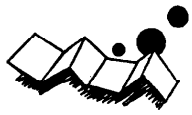
解　説



大規模ソフトウェアの品質管理†

高　村　眞　司††

1. まえがき

システムの品質は，それを構成するハードウェアとソフトウェアの品質にそれぞれ依存していることは言うまでもないが，とりわけシステム全体の利用技術を支配するソフトウェア（特に基本ソフトウェア）はシステムの品質を大きく左右する位置にある．

ところで，ソフトウェアの規模はコンピュータシステムの利用形態の多用化・高度化とともに増大を続けており，代表的なシステムの基本ソフトウェアだけでも1メガ～数メガステップの規模に達している．このような大規模なソフトウェアをいかにして高品質なものとするかという命題については各方面でいろいろな観点からの議論がなされている[1)~4)]．

すなわち，高品質のソフトウェアを実用化するためには，要求定義，設計，製造，試験，保守といったソフトウェアのライフサイクルを通して，一貫した品質管理思想の下に，各段階での適切な品質向上対策の実行が不可欠であるとの基本的な認識は一致してはいるものの，各工程における具体的な技術・手法等については，いまだ模索の域を脱したとは言い難い．

このような背景を踏まえた上で，本稿では DIPS (Dendenkosha Information Processing System) 用 OS の開発において実施した品質管理の考え方と手法を紹介する．本稿で述べる内容は品質管理に関して必ずしも網羅的ではなく，かつ万能でもないが，少なくとも大規模な基本ソフトウェアを開発する場合に遭遇するであろう品質管理上の問題と対策を中心に述べたつもりである．

2. ソフトウェアの品質

2.1 ソフトウェア品質のとらえ方

ソフトウェアを広義に解釈すると，その品質は個々のプログラムの完成度（プログラムとしての品質）とシステム全体として見た総合的な品質（方式としての品質）とに分けて考える必要がある．前者はプログラムの製造・試験等においてバグがどの程度除去されているかにより評価される．これは更に，プログラム個個の（単体レベルの）品質とプログラムを接続したときの品質に分けられる．

後者は，利用目的に合致した機能を具備しているか，所期の性能を満足しているか，異常時の信頼性や過負荷耐力の程度はどうか，使いやすさや保守のしやすさはどうか等々の観点からその充足度が評価される．

プログラムとしての品質が仮に良くても，方式としての品質が悪ければ総合的にはそのソフトウェアの品質は不良である．逆に方式としての品質が良ければ，仮にプログラムとして多少の不良（バグ）が残存していても致命的な問題となる可能性は少ない．要するに高品質のソフトウェアは高品質の方式設定によってのみ得られるものであると言っても過言ではない．

Gunther[16)]によれば，ソフトウェア製品とは，プログラムそのものとは違い，プログラムおよびそれにかかわる計画，ドキュメンテーション，試験，訓練，保守等を含んだより総合的なものであると定義している．筆者も，同様の立場に立つものであり，上で述べた方式品質とは，まさにプログラムをソフトウェア製品にまで高めるための諸々の活動に対する評価の指標であると言い換えることもできる．

さて，品質の評価尺度について考えて見よう．まずプログラム単体としての品質は先にも述べたように，残存（予想）バグ数によって，一応定量的な評価が可能である．

一方，方式としての品質評価尺度はたとえば，サービスへの適用性，操作性，保守性，過負荷耐力，信頼性，性能等が考えられるが，定量的に表わしうるものは少なく，定性的，主観的であるのが実情である．

2.2 大規模化とソフトウェアの品質

システム規模の拡大要素が，ソフトウェアの品質，

† Quality Control in Developing a Large-Scale Software System by Shinji TAKAMURA (Yokosuka Electrical Communication Laboratory).

†† 日本電信電話公社横須賀電気通信研究所

すなわち方式としての品質と製品としての品質それぞれにどのような関わりあいがあるかを例示してみると図-1 のように非常に複雑である．

システム規模の拡大に対して，各種のシステム特性は一般にはその限界値を超えるまでは規模に対して比例的に増加する．この限界値を超える領域でシステムが使われると，システムのサービス特性は正常性を失う可能性が出てくる．すなわち，システムの大規模化に耐えるためには，システムの許容限界値を引上げること，また限界値付近での安定な特性を保証することが必要である．これは大規模化に対する方式としての品質向上対策の基本的な考え方である．具体的な項目としては，性能のボトルネック対策，過負荷対策，システム異常時の対策，ソフトウェア構成の改善等が主なものである．

一方，プログラムとしての品質を向上させるための考え方・技法等は，各方面で議論されており，設計，製造，試験等のプログラムのライフサイクルの各過程でいろいろな工夫をこらす必要がある．特に試験の過程では，プログラムの規模増大によりすべての場合をつくした試験は工学的に不可能であるため選択的な試験とならざるを得ない．選択的な試験ではどのような環境で，どのような項目を試験するか，すなわち試験自体の信頼性，有効性，完備性等が十分吟味されなければならない．これがプログラムとしての品質を上げる上での基本的姿勢である．

2.3 方式品質と所要技術

方式としての品質を向上させるためには，それを規定する要素ごとにどのような取組みが必要であるかを概観する．

（1） 性　能

性能はある方式の品質を評価する上で最も重要な要素の１つである．したがって，性能の評価技術，特に方式設計時の性能の予測評価技術が重要である．具体的には，方式のモデリングとそのモデルを用いて解析的手法やシミュレーションによる方式の静的・動的特性の予測・評価技術である．この技術で重要な点は予測精度とモデルの忠実度である．

（2） 機　能

ある機能を提供する場合，その機能の適用性を拡大するために，以下のようなアプローチがある．

（a） 機能の抽象化

ある機能を設定する場合，できるだけ抽象的な概念で機能を抽出する．また，その機能の利用者の固有な使い方に依存しないように抽象化する．

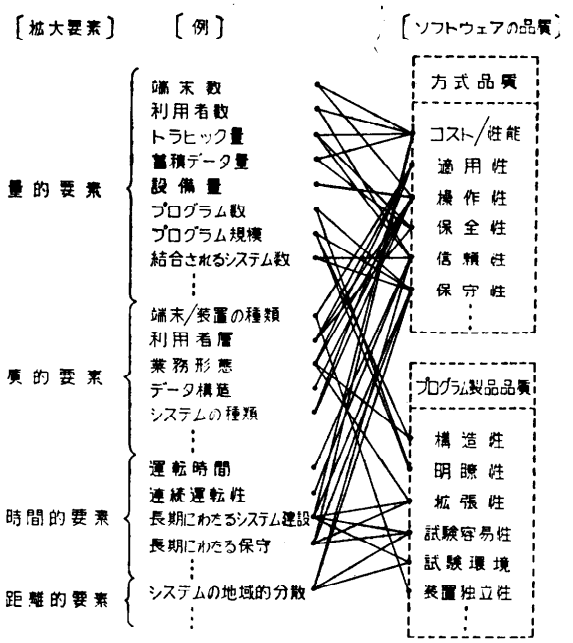


図-1　規模の拡大要素とソフトウェアの品質

（b） 仮 想 化

利用者にはハードウェア固有の物理的特性を意識させず，論理的な機能として見せる方法――ハードウェア機能の仮想化――である．機能抽象化の範ちゅうに入る．

（c） 階 層 化

システムに要求される機能は通常漠然としたものが多い．そこで機能を詳細化する過程でトップダウン的に分解してゆくと機能の階層関係（一般には木構造）を明確にできる．更に，機能階層をいくつかのレベルに分け，同一レベル内の機能を互いに排他的とする．このように機能の階層化やレベル分けをすることは，機能の抜けや追加，更にはプログラム構造を明快にする上で非常に重要である．なお，現実問題としては，すべての適用領域に対して機能の階層を読み切ることは不可能なので，類似の適用領域ごとに階層を定義してゆくことになる．したがって，規模の拡大，適用領域の変化などによって階層化の最適点が変ることは覚悟せざるをえない．その結果，階層の段数が増す傾向は避けられない．いずれにしても，ポリシとメカニズム[5]を明確に分離して機能を定義すること，下位の機能に上位の機能を含まないようにすることが重要な基本姿勢である．

(3) 構　造

プログラム構造は機能の階層に対応して規定するのが原則である．ただし，性能条件の厳しいプログラムにあっては，例外的にこの原則にそわない構造を取らざるをえない場合もある．たとえば，上位階層が持っている機能と同種の機能を下位階層が重複して持ったり，階層間で一般的に定められたインタフェースとは異なった特殊なインタフェースを設けることによって，厳しい性能条件を満足するなどの措置がとられることがある．

(4) 信　頼　性

方式としての信頼性は，それを構成する部品（ハードウェア，ソフトウェア）の平均故障時間間隔 (MTBF) と，故障修理時間 (MTTR) とを用いて，以下のように稼動率 (A) で表わすことができる．

$$A=1-(\mathrm{MTTR}/\mathrm{MTBF})$$

稼動率を高めるためには，部品の MTBF を延長し，一方 MTTR を短縮する必要がある．MTBF を延長するためにはハードウェアの故障，およびソフトウェアのバグ(オペレーションミスを含む) を減らすこと，一方 MTTR を短縮するには，ハードウェア，ソフトウェアおよび人間を含めたシステム回復処理の高速化がそれぞれ必要である．後者の具体的方式例として，ハードウェア構成の冗長化方式（たとえばデュプレックス構成，マルチプロセッサ構成)，OSの高速再開方式，サービスプログラムの高速再開方式，データファイルの高速復元方式等がある．これらは基本ソフトウェアまわりの方式技術として重要な位置を占めている．

(5) そ の 他

その他の事項として操作性，保全性等があるが，ここでは省略する．保守性については後述する．

広義には方式としての信頼性に関係するが，大規模システムで特に問題となる過負荷対策について若干触れる．

システムの過負荷状態は，限度以上の処理要求が到着し，それが有限なシステム資源を奪い合うことにより資源不足を生じ，そのためシステムがデッドロックに陥ったり，処理要求が雪だるま式に滞留する等の現象として現われる．これを防止するために，過負荷になる傾向の事前検出，不足するであろう資源の動的な供給，処理要求の入力制限などの要素を考慮した方式設定を行う必要がある．

2.4 製品品質と所要技術

要求仕様が固まって以降，これをプログラムとして仕上げるまでの過程における，品質向上のための所要技術を概観する．

ソフトウェアのライフサイクルはたとえば**図-2** に示す各段階があり，ソフトウェア工学的技術は各段階に応じて，**表-1** のようなものがある．これらは，生

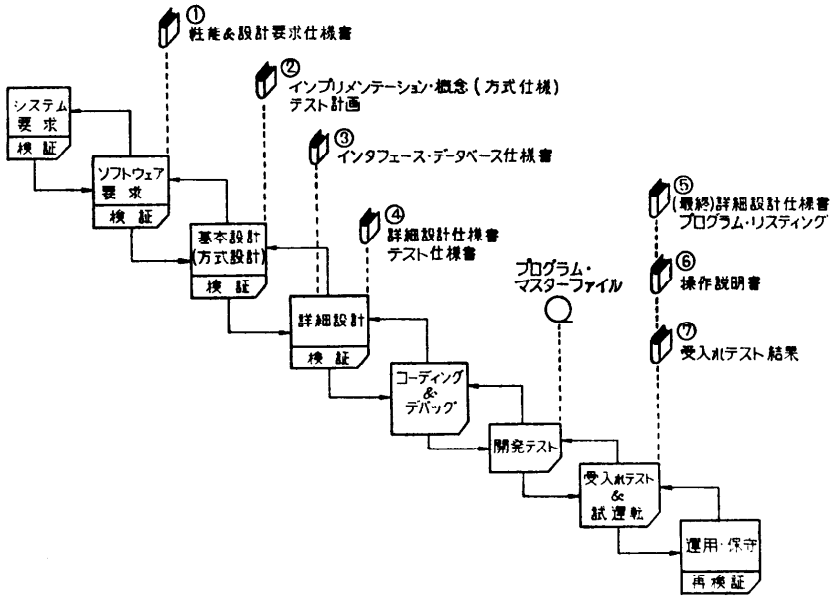


図-2[1] ソフトウェア・ライフサイクル

表-1 ソフトウェア生産技術

技術	主な技法
要求定義技術	要求定義言語による要求の記述
設計技術	階層的設計法（トップダウン設計，ボトムアップ設計） 共通化設計（ジェネレータ） 設計用形式言語による記述 ドキュメンテーション（HIPO など）
製造技術	プログラミング言語 ストラクチャードプログラミング コーディング規約 構造解析
テスト技術	階層的試験 試験ツール（情報取得，環境擬似，データ供給） ディグレードチェック 進捗管理（試験項目，バグ）
保守技術	ファイル保守ツール モジュール管理ツール
管理技術	工程管理（PERT）

産性向上，製品の品質向上という二面性を持っており，製造対象プログラムの特性，製造体制，技術レベル，コストなど多面的な観点から適切な手法や技術を選択すべきである．

3. 品質管理活動の具体例

DIPS-OS の開発において，品質向上のために，どのような取組みをしているか，管理技術面と品質向上手法・支援技術面とに分けて要点を紹介する[6]．

3.1 管理技術

（1） プロジェクト管理

高品質な大規模ソフトウェアを実現するには，多数の人間が長期間にわたって，統一のとれた設計思想の下に製品化活動に従事しなければならない．そのためプロジェクト管理は特に大規模ソフトウェアの実用化において重要である．

プロジェクト管理活動の対象は，要求の探索，製品計画，設計，製造，試験，保守に至るソフトウェアライフサイクルのすべてに及ぶ．管理内容はライフサイクルの段階に応じて異なり，また管理のための組織も一般には段階に応じて異なる．

（a） 管理組織の階層

図-3 に組織の階層と主な役割りを示す．大雑把に言って，方式としての品質は製品対応技術会議以上の組織で，またプログラムとしての品質は，同会議以下の組織でそれぞれ中心となって管理する．このような基本的管理組織のほかに，目的に応じて必要な組織を作るなど柔軟な組織運営により対処している．

（b） 作業標準の制定

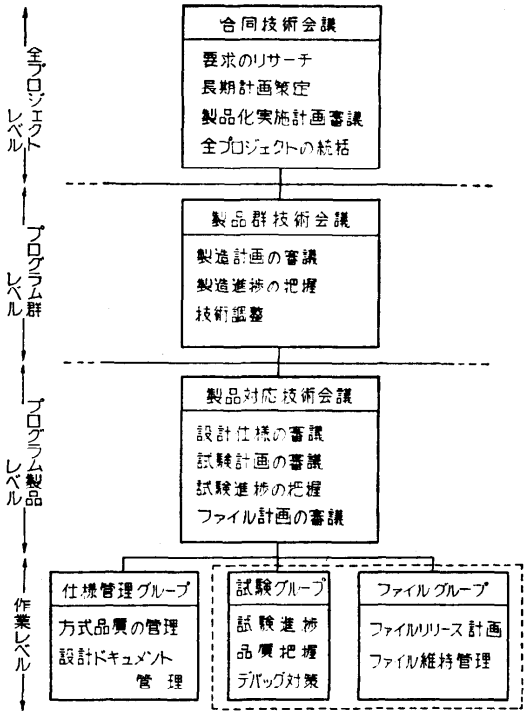


図-3 プロジェクト管理組織の階層

多数の開発従事者からのアウトプットを極力均質化することが製品品質の管理上重要である．このため，各工程の作業内容，作業方法，生産物，その他の標準を定め，管理のよりどころとしてこれを担当者に遵守させている．規定内容の要点を表-2 に示す．

この中で，「工程の定義と生産物」および「設計ドキュメント作成標準」は本標準の中核である．各工程の開始条件は工程計画書で，また終了結果は工程終了報

表-2 作業標準の規定内容

項目	内容
工程の定義と生産物	工程，標準ドキュメント ドキュメント変更ルール，計画書／報告書
組織と運営	会議名，議事録，連絡票 資料の管理
ソフトウェア命名法	ソフトウェアシステム名 システムファイル名，改版番号
設計ドキュメント作成標準	仕様書の形式，目次 仕様書の種類
メッセージ形式，入力形式	メッセージ標識，コンソール入力形式
コーディング規定	
その他	

告書でそれぞれ詳細にチェックし，品質の維持と納期の遵守に役立っている．

また，設計ドキュメントは形式と記述内容のひな型が規定されており，これによりある程度均質なドキュメントが得られている．しかし，文体や記述の詳しさの程度まで規定していないなど，まだ改良の余地を残している．

（2） 製品品質の管理指標

プログラムの品質は従来から検出バグの累積値や消化した試験項目数の累積値から，その完成度を推定する方法がとられている[3),7)]．しかし，試験工程の終期付近では，バグ累積曲線が飽和傾向を示すため，この曲線からバグの枯れ具合，更にはシステム全体としての安定性を判定することが難しくなる．

そこで新たな試みとして，システム全体をユーザの立場から見た総合的な品質指標を定義する[9)]．すなわち，1つのトラブル（プログラムのバグ，ハードウェアの故障，オペレーションミスを含む）がシステム全体に与える影響度に重みをつけ，その発生頻度との総合値（Q）でシステム全体の品質を把握する．

$$Q=(\sum_{i} K_i \cdot F_i)/t$$

ここで，t：期間

K_i：トラブルがシステムに与える影響度

F_i：トラブルの発生回数

発生したトラブルは本指標を用いて，重要度に応じて個別に管理し，迅速な対処を行う．本指標は先に述べた，方式としての品質の一端を表わしているともいえる．

検出バグ累積曲線は，プログラムの完成度を確認するため試験工程の初期から終期に至る間の品質把握に用いる．一方，上で定義した総合品質指標は試験工程の終期におけるシステム全体の安定性推定に用いる．(表-3) 両指標を用いた品質管理の例を図-4 に示す．

なお，総合品質指標は，いわばシステムの信頼度の許容限度を表わすとの見方もできるので，今後更にデータを積み重ねてゆき，「サービス基準値」として一般

表-3　品質管理指標

管理方式	プログラム完成度管理	システム総合品質管理
適用工程	設計，製造，試験の各工程	試験工程の後期および運用期間
管理項目	統計的管理 [バグ数，バグ比率 [試験項目，試験ネットワーク	個別管理 [システム安定度 [バグの影響度
対処姿勢	確実性	迅速性

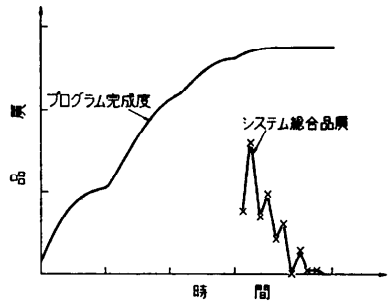


図-4　品質指標適用例

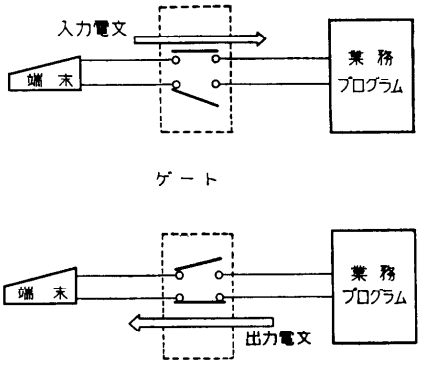


図-5　ゲート制御の概念図

化してゆく予定である．

3.2　方式品質改善方法の具体例

（1） 機能の抽象化

銀行システムなどの実時間システムにおける電文の処理機能について，その抽象化の過程を紹介する．

電文の処理方法はサービスシステムによって多種多様であるが，おおむね ① 電文の流し方と ② 電文が途中で損傷を受けた場合どういう方法で救うかによって特徴づけられる．このうち，① は電文のスケジュール機能であり，端末に対し電文を送信する，受信する，いずれも行わないのどれかを決定することおよび送信する場合には，どの電文を送信するかを決定することにある．この機能は次のように抽象化できる．すなわち，端末と業務プログラムとの間に論理的な電文の通路を考え，その途中に開閉器（ゲート）を設ける (図-5)．ゲートに設定する錠と電文の持つ鍵が一致すれば電文は通過しそれ以外では，電文は通過できない．このような抽象化によって，サービスごとに多様な電文スケジュール機能を，単一の機構で実現可能となった[10)]．

②については，電文がどのような処理過程で罹障したかを判別できることが基本となる．そのため電文の

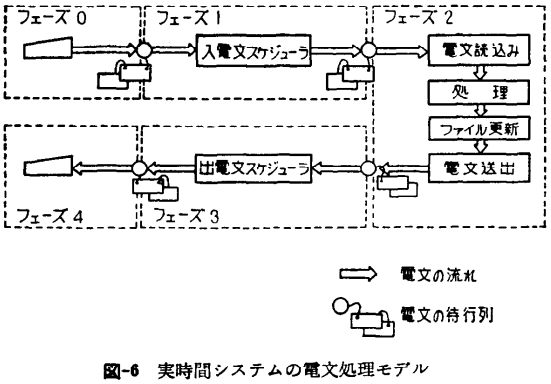


図-6　実時間システムの電文処理モデル

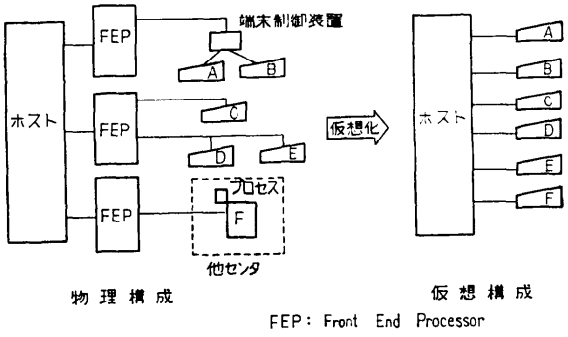


図-7　通信網の仮想化概念図

処理過程を，待ち行列の位置に注目して5つのフェーズに分割し(図-6)，各フェーズごとに電文の救済条件を明確にする方法を採った[11](表-4).

以上により，実時間システムにおける電文制御機構を色々なサービスシステムの多くの業務に対して共用できるようにした．また，フェーズの概念の導入により，サービスシステムごとの電文救済プログラムの規模を大幅に減らすことができた．これは，機能抽象化による適用性の向上が，生産性改善にも寄与していることを示している．

（2） 端末制御の仮想化

表-4　各フェーズにおける救済処理

フェーズ	救済処理	特記事項
フェーズ0 フェーズ1	端末からの電文を再投入する	・電文は伝送路上にあり完全な救済はできない．
フェーズ2	ファイル更新情報の控えと電文の控えを照会する	・電文とファイルとの間の無矛盾性保証が基本である． ・更新前のファイル，更新後のファイルのいずれを基準とするかにより2方式がある．
フェーズ3 フェーズ4	出力電文の控えをフェーズ3または4から再投入する	・電文送達の確認をどの過程で取るかにより，電文の2重送出を許す方式（冗送方式），電文の欠落を許す方式（欠送方式）とがある．

端末はその種類が多く，属性（伝送制御手順，コード体系など）がさまざまなこと，回線構成も多様(半二重/全二重，直通/分岐，交換/専用など)なこと等から，従来の端末制御プログラムや業務プログラムは，これらの相異の影響を直接被っていた．これは，新しい端末や通信網の導入時に大きな支障となるばかりか，バグの潜在を許す可能性が大きい．これを解決するために，通信処理と情報処理を明確に分離し，情報処理（業務プログラム）側から端末や回線の物理的相異が直接見えないようにする方式を導入した[10](図-7)．これにより，業務プログラムは，同一の標準化された手法で端末やネットワークへのアクセスが可能となるほか，端末種別対応のプログラム規模も従来に比べて小さくすることができた．

（3） OS 構成の階層化[15]

表-5 に DIPS-OS における階層化の進展を示す．ポリシとメカニズムの分離の原則に則って，サービスシステム固有のポリシを表現する部分——サービス管理プログラム——と，たとえば実時間処理のメカニズムを提供する実時間パッケージとの間の機能分担を改善している．

これにより，従来サービスシステムごとに作られていた実時間パッケージを1種類のみですませることができるようになった．さらに複数のサービスシステムによって繰り返し使われたため，短期間に高い品質を確保することができた．

表-5　DIPS-OS における階層構成

OS種別	階層構造概念図	説明
A	制御プログラム AP	・2階層構成 ・サービス固有機能がOS基本機能と一本化
B	制御プログラム パッケージ サービス対応共通プログラム AP	・3階層構成 ・処理形態（リアルタイム処理，TSS 処理）に固有な機能をパッケージとして分離
C	制御プログラム パッケージ サービス管理プログラム AP	・4階層構成 ・サービス固有機能に関する機能分担を整理し，階層関係を明確化

（4） 処理能力の評価手法

大規模システムでは処理能力面での品質の安定性も重要であり，システム性能のボトルネックを設計時点で把握し，方式設定に反映する必要がある．

従来からの実測を中心とした処理能力の評価に加えて，待ち行列理論を用いた処理のモデル化とその特性解析，ならびに対象処理システムを正確に擬似した計算機シミュレータによる評価を併用し，処理能力上の方式的問題を事前に解決した．一例を挙げると，実時間システムではハードウェア資源のほかに論理的な資源であるタスクがシステムの処理能力を左右する．そこで，ハードウェア構成とともにタスク構成もあわせて表現するため，従来計算機システムの評価に用いられてきた待ち行列ネットワークに，系内呼数制限を導入した待ち行列モデルを作り，タスク構成の妥当性をあらかじめ評価するようにした．また，計算機シミュレータについては，対象モデルの記述が容易であり，連続的に処理能力の追跡が可能なものを開発し，利便をはかっている．

（5） システム稼動率の改善

密結合マルチプロセッサ構成のシステムでは，系内のあるプロセッサが罹障しても残りのプロセッサが正常である可能性がある．そこで，プロセッサ等の本体系装置の障害に対して，“活きている”プロセッサを用いて障害情報の分析，罹障装置の識別，再構成，システムの再開始処理を自動的に行う．この方式の採用によりシステムの停止からOSの回復までの時間を大幅に短縮でき，MTTRの短縮に有効であった[13]．

（6） 過負荷対策

実時間システムでは，メモリに対する過負荷対策が重要である．従来から不急のプログラムが占有する領域を強制的に取り上げ，空き領域を供給する方式がある．中でも，メモリにロードされていないプログラムへの接近をハードウェアで検出し，それを契機にページ単位でプログラムを入替えるいわゆる要求ページ方式（On Demand Paging）は，ページ単位のきめ細かな管理ができる点で柔軟性があり，広く採用されている．しかし，実時間システムでは，応答時間に対する条件が厳しく，この要求ページ方式は用いられないのが一般的であった．これに対し，通常処理時にはページングが起こらないように制御し，メモリ過負荷時のみページング機構が働く方式を採用した[12]．このように，ページングをメモリ過負荷時の緩和機構として用いることにより，システムの過負荷耐力を向上することができた．

3.3 プログラム品質の改善方法の具体例

（1） 設計の詳細化手法とドキュメント

トップダウン的手法により，機能の階層をそのままモジュールに対応させるとモジュールがネスト構造を持つようになり，プログラム実行時のオーバヘッドが大きくなる．これを解決するためにモジュールの分割過程で，新たに必要となる機能をそれぞれのモジュールに追加することによりネストを起こさない工夫を行う．また，機能の階層化，モジュールの分割過程およびモジュール間インタフェースを表わすドキュメントとして，それぞれ「機能ツリー図」，「機能票」，「モジュール票」を規定している．これらのドキュメントは詳細設計からコーディングに至る過程のドキュメントとして，フローチャートに代るものである．

（2） 試験手法

（a） 段階的試験

大規模ソフトウェアは何段階にも階層化されたプログラムで構成されるのが一般である．このようなソフトウェアの試験は，その階層を考慮して進める必要がある．具体的な試験手順を以下に示す．

① 単体試験

階層単位の品質を上げることがまず基本であり，そのため階層ごとに閉じた試験を徹底的に行う．机上デバッグが中心である．

② 接続試験

隣接する階層との間を1つの単位として接続し，階層間のインタフェース整合性を中心に試験する．マシンデバッグと机上デバッグを併用する．

③ 総合試験

実際の運転環境に近い環境でシステム全体を通した試験を行う．単に機能試験だけでなく，処理能力，過負荷耐力，連続運転性など，多面的な試験を行うマシンデバッグが中心である．

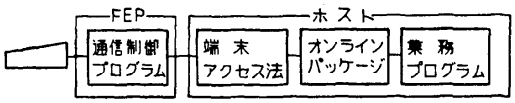
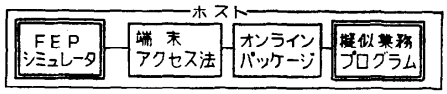
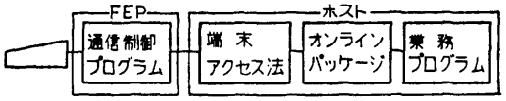
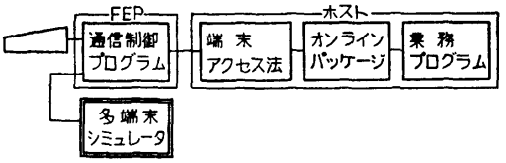
以上のような段階的試験により，プログラムの誤りを早い機会に検出でき，また並行して複数の階層の試験ができるため工期の短縮と高品質のソフトウェアを製造する手段として有効である．

表-6 に DIPS-OS のオンライン系ソフトウェアの試験体系を例示する．

（b） スルー試験

接続試験や総合試験における階層間のインタフェース試験では，階層ごとの単体品質がある程度揃っていることが前提である．これを事前にチェックするため

表-6 DIPS-OS における試験形態の例

分類		試験形態	説明
実構成			
接続試験	FEP 内接続試験		接続相手のシミュレータを用い，互いに並行して試験を行う．
	ホスト内接続試験		
総合試験	機能試験		実構成と同一環境で総合的な確認を行う．
	性能試験		多数端末からの高い負荷状態を作り出す多端末シミュレータを並用し，過負荷試験を行う．

 は 試験環境設定用ツール

に，対象となる階層間で最も重要な試験パスを選び，正常に動作するかを確認する（スルー試験）．この試験により，重要パスを中心にして細部の試験が進められること，重要パスが優先して試験されるため重大なインタフェースバグが早期に発見できること等の利点がある．

（3） 試験ツール

ソフトウェア工学の立場から開発ツールの体系化が試みられるようになってきている[8]．

試験ツールは試験環境の設定など試験を支援するツールと，デバッグ作業を支援するツールに分類できる．DIPS-OS の開発においてはオンライン系ソフトウェアの試験手法体系化にあわせて，試験支援ツールの充実に力点を置いている[14]．（前者の例を，既出の表-5に示してある．）

また，大規模システムのデバッグでは，試験データの作成・供給ツールやシステムの動作タイミングに関わるデバッグ情報の取得ツールなどデバッグ支援ツールも品質の早期安定化に不可欠である．なお，タイミングに関わる異常はきわめて稀な確率で起こるためデバッグ情報を常時取れるように，ツールを OS 機能の一部として組込むなどの工夫を行っている．

（4） プログラムの保守

プログラムの保守性を良くするには機能分担やモジュール構造など方式としての品質を良くすることが基本であり，そのための手法は前述した．ここでは実際の保守作業を中心に保守性の改善について触れる．

（a） ファイルの管理とシステム生成

プログラムの保守作業をソフトウェアシステムのマスタファイルの維持管理とシステム生成処理との一貫した流れとしてとらえ，この作業の流れをプログラムにより自動化することにより，保守作業の省力化と保守品質の向上をはかっている．

ファイルの維持管理はソースマスタファイル，オブジェクトマスタファイルにそれぞれ分けて管理する．ここでの主要技術は，ファイル更新ジョブのジョブ制御文を自動生成すること，および目録によるファイルの登録管理である．

システム生成処理は，使用目的に合致するモジュールとデータをマスタファイルから選択して組合わせ，所要のソフトウェアシステムを生成するものである．この一連の処理をプログラムにより自動化している．

4. むすび

大規模ソフトウェアの開発における品質管理の課題と対策について，基本ソフトウェアの開発例をひきながら述べた．

現在，ソフトウェア工学の分野で品質といえば，その多くは本稿でいう“プログラム品質”を意味し，その向上技法に興味が注がれている．この面で着実に成果を上げていることは疑う余地はない．

しかし，大規模ソフトウェア——たとえば基本ソフトウェア (OS) ——を高品質化するためには，“方式品質”の改善により一層の力を注ぐべきである．たとえば，近年の LSI 技術の急速な進歩に伴うハードウェアとソフトウェアのトレードオフの問題，分散システム化の問題，ネットワーク化の問題等いずれを見てもコンピュータシステムの方式としてどう対処すればよいかが問われている．

ところで，方式品質を上げるための何らかの手ごろな手法があるであろうか．

残念ながら今のところ否と言わざるをえない．それは，方式品質が客観的・定量的に表現あるいは評価しにくいこと，したがって方式は開発者（設計者）のアイディアと妥協の産物の域を脱していないことによると考えられる．

したがって，方式品質を定量化・客観化するための試みに取り組むことがまず必要である．そのためには，過去における方式設定を反省するという地道な作業から取りかかる必要がある．

更に，これにも増して重要なことは，優秀な方式技術者を育成することである．これも一朝一夕にはできないが，長期的な展望に立脚した教育プランと実践が必要である．

最後に，本文では触れなかったが，大規模化の側面としてソフトウェア資産増大の問題がある．処理対象の多様化，処理量の増大とともにソフトウェア資産も増大し，これらの維持管理をどうするか，特に新旧何世代ものファイルをどのようにして効率よく現行維持し，継承してゆくかなどの問題に対して，方式品質の管理，方式の継承・移行性といった広い立場からの取り組みが大切である．

参考文献

1) 上條史彦：ソフトウェアリライアビリティに関する最近の話題，電子通信学会，Vol. 59, No. 4, pp. 369-377 (1976).
2) 宮本 勲他：ソフトウェアエンジニアリングとは，Bit, Vol. 9, No. 4, pp. 297-303.
3) 杉浦宜紀：ソフトウェアの信頼性，沖電気研究開発，Vol. 41, No. 2, pp. 63-72 (1975).
4) 辻ケ堂信他：大規模オペレーティングシステムの開発管理，電気学会，Vol. 98, No. 1, pp. 20-23 (1978).
5) Wulf, W. et al.: Policy/Mechanism Separation in HYDRA, ACM Operating System Review, Vol. 9, No. 5, p. 132 (1976).
6) 吉田庄司他：DIPS-104 OS の実用化，通研実報，Vol. 26, No. 8, pp. 1-25 (1977).
7) 安部城一他：総合デバグ時の PB 曲線の性質について，情報処理学会論文誌，Vol. 20, No. 4, pp. 306-313 (1979).
8) 河田 汎他：基本ソフトウェア開発ツールの実情，情報処理，Vol. 20, No. 8, pp. 694-702 (1979).
9) 高村眞司他：大規模実時間システム用 OS の設計，通研実報，Vol. 28, No. 12, pp. 2615-2633 (1979).
10) 松田晃一他：リアルタイムシステムの電文制御方式，通研実報，Vol. 26, No. 8, pp. 2221-2239 (1977).
11) 大橋楷一郎他：リアルタイムシステムの障害処理方式，通研実報，Vol. 26, No. 8, pp. 2261-2272 (1977).
12) 伊東洋一他：DIPS-104 OS の制御プログラム，通研実報，Vol. 26, No. 8, pp. 2203-2220 (1977).
13) 大橋楷一郎他：DIPS-104 OS の信頼性向上技術，通研実報，Vol. 28, No. 12, pp. 2711-2724 (1979).
14) 拝原正人他：多端末試験システム，通研実報，Vol. 28, No. 12, pp. 2725-2743 (1979).
15) 大島 裕他：DIPS オペレーティングシステムの標準化，通研実報，Vol. 28, No. 12, pp. 2635-2655 (1979).
16) Gunther, R. C.: Management wethodology for software product engineering.

(昭和55年6月6日受付)