展　望

ソフトウェアの信頼性
Reliability on software

中所　武司

エンドユーザからみれば，システムの不良の原因がソフトウェアの不良である場合も，ソフトウェアそのものへの関心は二義的なものであった．しかし，実際には社会的な影響の大きなシステムの事故原因がソフトウェア不良であるケースは多い．電子商取引や電子政府の時代の到来と共に，ソフトウェアの稼動形態が多様化し，信頼性の概念が広がっている．不良を作りこまない設計・製造技術と不良を検出するテスト技術は，インターネットに代表される高度情報化社会に不可欠の技術となっている．

1．はじめに

システムのユーザからみればソフトウェアそのものへの関心は二義的なものであった．しかし，実際には社会的な影響の大きなシステムの事故の原因がソフトウェア不良であるケースは多い．たとえば，西暦2000年問題は，ソフトウェアの存在を社会的に認知させるのにはおおいに役立ったと思われるが，ソフトウェアの高信頼化技術の進歩につながったか否かは不明である．

ここでは，ソフトウェア固有の特質に関連したソフトウェアの信頼性について述べる．ソフトウェアの高信頼化技術としては，不良を作り込まない設計・製造技術と不良を検出するテスト技術にわけられる[1]．

インターネットに代表される高度情報化社会を迎え，電子商取引や電子政府を支えるソフトウェアの稼動形態も多様化してきた．従来から高信頼化を追求してきたビジネス分野の基幹業務を中心とする大規模ソフトウェアや産業分野の機器制御ソフトウェアに加えて，最近ではwebアプリケーションおよびその連携が重要になってきている．このような新しい分野では，従来のトップダウン開発技法とは異なり，ボトムアップ開発ともいえるコンポーネントベース開発技法が注目されている．

本稿では，最初に信頼性の意味とソフトウェア不良による事故の例について述べ，後半でソフトウェア設計・製造技術とテスト技術について説明する．

2．ソフトウェアの信頼性の意味

ソフトウェアの信頼性は，1991年にISOでソフトウェア品質特性として規格化された6項目（機能性，信頼性，使用性，効率性，保守性，移行性）の1つとして定義されている[2]．すなわち，信頼性とは「明示された条件下で，明示された期間，ソフトウェアの達成のレベルを維持するソフトウェアの能力をもたらす属性の集合」であり，副特性として，成熟性，障害許容性，回復性がある．簡単にいえば，障害はできるだけ発生しないほうがよいし，障害が発生してもその影響が小さいものがよく，障害はできるだけ速やかに回復するのがよい．

ソフトウェアの障害は主に出荷前に検出できずに残った不良（いわゆるソフトウェアのバグ）が原因となることが多い．このような不良を少なくするという意味でのソフトウェアの信頼性は，図1に示すソフトウェア開発技術の3要素との関連が深い．プロジェクト，プロダクト，プロセスは，生産技術の観点では「誰が何をど

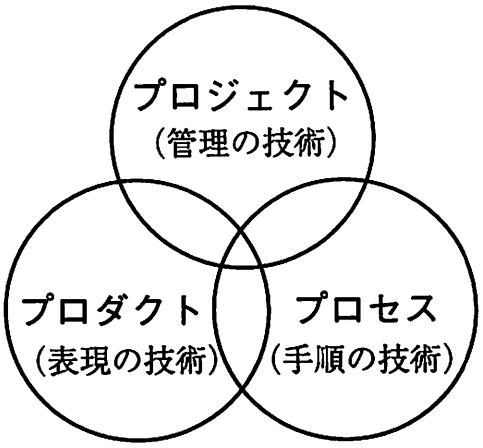


図1　ソフトウェア開発技術の3要素

のように作るか」という視点に対応する．

第1のプロジェクトでは，開発者および彼らが担当する工程や作り出すものに関する管理の技術が重要である．第2のプロダクトでは，設計仕様書やテスト仕様書，プログラムなどの生産対象に関する表現の技術が重要である．第3のプロセスでは，設計，製造，テストなどの工程に関する手順の技術が重要である．

さらに，信頼性のより広い意味においては，単に出荷時の不良のみならず，稼動後の環境変化への対応が重要になっている．従来のようにあらかじめ想定された限定的環境下で稼動するアプリケーションに加えて，最近ではインターネットに接続されたコンピュータ上で稼動するアプリケーションが増加している．このようなソフトウェアの信頼性は，ハッカー対策やウィルス対策を含むセキュリティ対策にも依存することになるが，その多くはソフトウェアの作り方にも関係する．また，ユーザニーズの絶えざる変化に対応するためには，保守性も重要である．インターネットに接続されて稼動するアプリケーションでは，その稼動環境（ハードウェア，オペレーティングシステム）に依存しないで動作可能であるというプラットフォーム非依存性などの観点で移行性も重要である．

3．事故・事件の事例の考察

3.1　西暦2000年問題の本質

西暦2000年問題は，出荷時には不良でなかったものが，環境の変化に伴って不良となった典型的な例といえる[3]．アプリケーションの寿命が開発者の予想を越えていたという誤算よりも，ソフトウェア誕生から半世紀を経てなおまともな保守技術が確立していなかったことのほうが大きな誤算と言えるかもしれない．ソフトウェア工学の観点からは次のような課題を指摘できる．

- 技術的課題：ソフトウェアの保守技術
  （作ったのに直せない？）
- 社会的課題：ソフトウェアの品質保証
  （バグはあって当たり前？）
- 歴史的課題：プログラミングパラダイム
  （どんな作り方をしたの？）

まず第1に，既存のプログラムを変更するという観点では典型的な保守の問題であり，日常的業務の1つにすぎないが，西暦2000年問題は幾つかの問題が重なって解決を難しくしてきた．すなわち，膨大なデータベースの変更と関連処理部分の変更，長期間使用されてきた現行システムの機能を正しく反映した仕様書がないことによるプログラム変更個所の特定の困難性およびその変更が副作用を引き起こさないことを確認するテストの困難性などである．

第2に，ソフトウェアの故障によって生じる社会的影響の大きな情報システムが多数存在しているため，ソフトウェアの実態を知らない一般の人たちにはブラックボックス的恐怖からのパニック現象も見られた．ソフトウェアの信頼性が損なわれるという意味では第一義的には質の問題であったが，膨大な数の対象システムに対応できるエンジニアの数が限られているという点では量の問題でもあった．

第3には，上記の保守や品質保証の困難性の

根本的原因としてプログラミングパラダイムの問題があった．西暦2000年問題は半世紀以上続いた手続き型のプログラミングパラダイムが世紀末の断末魔の苦しみにあえいでいる姿であったと言える．

3.2 某宗教団体関連ソフト会社事件の本質

今年の春ころ，かつて反社会的な事件を起こした某宗教団体関連のソフトウェア会社が海上自衛隊の指揮管制支援システムや警視庁の警察車両管理システムを含む官公庁システムおよび多数の民間会社のシステムを受注していたことがメディアに大きく取り上げられた．ソフトウェア工学的観点では，先に述べた3要素に関して発注者の視点で次のような問題があった．

- プロジェクト管理：誰が開発したか不明．
- プロセス管理：どのように開発したか不明．
- プロダクト管理：何を開発したか不明．

発注者から大手の元請けを経て次々と下請けに回される過程で，下請け会社で誰がどのように開発するのかということには無関心であった．唯一の関心が業界相場の約2／3の価格にあったとすると，ＩＴ革命が旧態依然とした労働集約型産業によって支えられていることを露呈したことになる．信頼性確保のためにその社会的重要性に見合ったコストをかけるという初歩的なリスクマネージメントさえいまだ不在といえる．これらの事実の発覚後，関連するソフトウェアを破棄したとするシステムもあるが，これは逆にいえば，プロダクトの検査が厳密には行われていないことを意味する．実装された機能が必要な機能をすべて含んでおり，かつ余分の機能を含んでいないという必要十分性のチェックがおこなわれていれば，あえてシステムを破棄する必要はないはずである．今回の事件の怖さは，通常のシステムテストや受け入れテストにおいて仕様書に記述された機能に関するテストだけ実施したのでは，危険なコードの埋め込みなどによる過剰機能が見逃されやすいことにある．

3.3 多発する設計不良と検査漏れ

社会的に重要な役割をになうコンピュータシステムに関して，ソフトウェアが原因の事故が多いが，その多くは設計やプログラミングのミスに起因している[4]．複雑な機能を実現する過程で，非同期処理に関連して発生する例外的な事象を見落とし，テスト項目からも漏れ，金融機関のオンラインシステムや証券取引所売買システムがダウンしている．

初歩的なミスも多い．ある電話会社の料金過大請求の原因はデータ読み込みエリアの初期化漏れだった．また，最大2048台の端末を処理できるように設計されていた座席予約システムが10年後に2200台に端末を増設してダウンしたり，6年間正常に動作してきた航空管制システムがたまたまパイロットの異常に長いメッセージを受信してダウンしたり，某銀行システムに連休直前の預金の引出しが集中し，他行の口座からの引出し取扱件数の許容量を超えた後の引き出しに関して顧客の口座の残高が減らないという事故などは，いずれも処理可能な最大値を越えたときの例外処理が抜けていた．

コンピュータが大惨事を引き起こすこともある．1994年の名古屋空港での航空機墜落事故では二百数十人が亡くなっているが，その原因の一つは，着陸やり直しの命令を受けたコンピュータがその後のパイロットの手動による着陸操作に反抗して異常な機首上げを引き起こしたためである．なんとコンピュータによる自動操縦とパイロットによる手動操縦の両方の機能が同時に実行可能な設計になっていたのである．この場合，ソフトウェア開発担当者に責任無しと言えるだろうか．

さて，なぜこのような設計ミスやテスト漏れを回避できないのかといえば，それは現在のプログラムの作り方に原因がある．実はコンピュータの実用化から今日までの半世紀の間，コンピュー

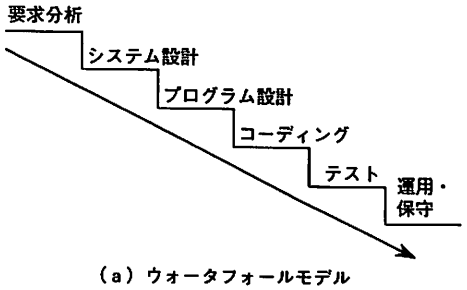
タへの命令を1ステップずつ手続き的に記述するというプログラムの作り方はほとんど変わっていない．たとえば，銀行のオンラインシステムには約1,000万ステップのプログラムからできているものがある．「データを読み込む」とか「残高と引き出し額を比較する」とか「手数料を差し引く」などといった手続き処理を1,000万ステップも記述してミスがないほうが不思議である．

4．設計・製造技術

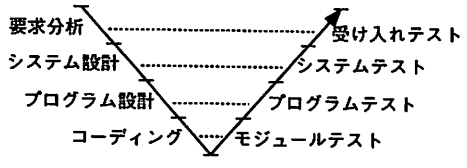
4.1 プロセスモデル

ソフトウェアの信頼性向上のためのソフトウェア開発方法論が論じられるようになったのは1970年ころである．当時，コンピュータの大型化に伴いソフトウェアも急速に大規模化していたが，早晩ソフトウェアがハードウェアの大型化に対応できなくなるのではないかという心配があり，ソフトウェア危機といわれた．

そのため70年代にはソフトウェア開発工程，すなわち要求分析，設計，製造，検査，保守運用を対象とした方法論，技法，ツールが開発され，要求分析から運用・保守にいたるライフサイクルモデルが注目されるようになった．その初期の代表的なものは図2(a)に示すウォータフォールモデル（滝モデル）である．要求分析，システム設計，プログラム設計，製造（コーディング），検査（テスト），運用・保守の順にソフトウェアをトップダウンに開発していくものである．各工程の終了時には，その工程の作業の結果をまとめた要求仕様書，設計仕様書，テスト仕様書などのドキュメントやプログラムが成果物として作成される．



(a) ウォータフォールモデル



(b) 設計とテストの対応

図2　従来のソフトウェア開発プロセス

このモデルでは，図2(b)に示すように，仕様はシステム全体の大まかなことから順にトップダウンに決定されていくが，プログラムのテストは逆に部分から全体へとボトムアップに実施される．そのため，最も重要なシステム全体のテストが最後に実施され，ソフトウェア開発の最初の段階で作り込んだ誤りが最後の段階にならないと検出できないという欠点がある．一般に誤りの検出・修正のコストはその誤りを作りこんだ時点から検出までの期間が長くなるほど高くなる．

そこでこのようなトップダウン開発モデルの改良方式として，まず全体の核となる部分を開発し，段階的にそのまわりの部分を開発していくスパイラルモデルがある．設計上の重要な部分を早く作って，開発の初期段階で仕様や性能の確認を行うことにより，最終段階での設計変更などによる大幅な手戻りを防ぐことができる．特に機能をサブ機能に分割し，サブ機能単位でプログラムを順次完成させるという方法で機能拡張しながら全体を完成させる方式をインクリメンタル開発と呼ぶ．このインクリメンタル開発を含むクリーンルーム手法については，本特集の佐藤の論文で詳述されている．またこのサブ機能単位の開発を要求を満たすレベルに達するまで何度も繰り返し開発する方式を反復開発（iterative development）と呼ぶ．

4.2 プロセス評価モデル

製品としてのソフトウェアの品質を高めるためには，それを作り出すプロセスそのものを改善する必要がある．そこで，成熟したプロセスから高品質のソフトウェアが生産されるという考えに基づいて，プロセスの成熟度を評価するモデルが作られている．表1は米国で開発された5段階のプロセス成熟度モデルCMM (Capability Maturity Model)である．

表1　プロセス成熟度モデル

成熟段階	内容	代表的技術
1：初期レベル	管理されていない状態	
2：反復可能レベル	反復実行可能な作業と統計的情報に基づく管理の実施	・プロジェクト計画・実行監視 ・ソフトウェア構成管理 ・品質保証
3：定義されたレベル	定義されたプロセスに基づくプロジェクト管理の実施	・組織的プロセス定義 ・協調作業 ・統合的管理
4：管理されたレベル	測定データに基づく品質とプロセスの改善の実施	・品質管理 ・定量的なプロセス管理
5：最適化レベル	継続的なプロセス改善の実施	・プロセス変更管理 ・欠陥予防

国際標準化機構ISOでも，一般の製品を生み出す品質システムを規定したISO9001をソフトウェアの開発，供給，据付および保守に適用するための指針がISO9000-3に規定されている[5]．国内でもISO9000シリーズ規格に基づき，審査登録機関がソフトウェア供給者の品質システムを審査し，適格と判断された供給者を登録，公開する品質システム審査登録制度がある．

4.3 コンポーネントベース開発

従来の手続き型パラダイムをベースにしたプロセス重視の開発方法は大規模ソフトウェアをゼロから作り上げることを前提にしていた．しかし，従来の方法では，最近のインターネットに代表される高度情報化社会に新たに必要とされる電子商取引などのアプリケーションソフトウェアをタイムリーに供給していくことはできない．そこで，プロダクト重視の視点から，ボトムアップ開発をベースとしたコンポーネントウェア[6]が注目されるようになった．

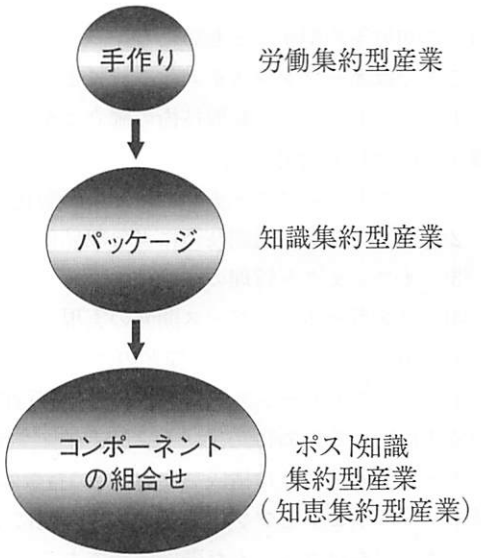


図3　ソフトウェア産業の変遷

この開発形態の変化は図3に示すソフトウェア産業の発展過程ととらえることができる．ソフトウェア産業は，生産コスト（プログラマの人件費）あたりの生産量（プログラムのステップ数）を生産性の基準とする労働集約型産業から，業務の知識をパッケージ化した知識集約型産業へと移行してきた．しかし，変化の激しい時代には業務の知識も常に変化していく．使い慣れたパッケージに新たな機能を付加していく対処方法ではベンダー依存性が強く，迅速な機能変更ができないなどの保守の問題が残る．ポスト知識集約型産業では，業務の専門家が知恵をしぼって効果的なアプリケーションの開発・保守に関与していく必要がある．その1つの解として，分野ごとの業務の知識をコンポーネントに対応させ，その柔軟な組み合わせでアプリケーションを作成する方法が現実的である．ポスト知識集約型産業は，業務の専門家の知恵をいかにコンピュータ化していくかという意味に

おいて知恵集約型産業ともいえる.

ここではオープンシステム化を背景としたコンポーネントウェアの基盤技術の確立に至る経緯を次の流れで説明する.

(1) ソフトウェアアーキテクチャの階層化
(2) オブジェクト指向技術による部品化
(3) オブジェクト管理の標準化
(4) コンポーネントベース開発の実現

まず第1にインターネットなどのネットワーク上でのアプリケーション間の接続性や移行性の要求に応えるために，ソフトウェアアーキテクチャは図4のような構成をとる．その特徴は，基本ソフトウェアとアプリケーションの間にミドルウェアをはさみこんだ階層構造である．ミドルウェアが基本ソフトウェアの違いを解決することにより，アプリケーションの構築やネットワーク上の他の場所にあるアプリケーションとの連携が容易になる.

第2にこのようなアーキテクチャの各階層はオブジェクト指向技術におけるクラス部品（オブジェクト）の集合として容易に実現できる．その結果，部品の利用やカスタマイズが容易になり，上流工程からこのようなオブジェクト指向プログラミングへのシームレスな流れを形成するためのオブジェクト指向分析・設計技法が実用化された.

第3にこのようなオブジェクトの管理技術が標準化されることにより，分散環境下でのアプリケーション共通オブジェクトの共有方式やオブジェクト間の連携方式の標準化が実現した.

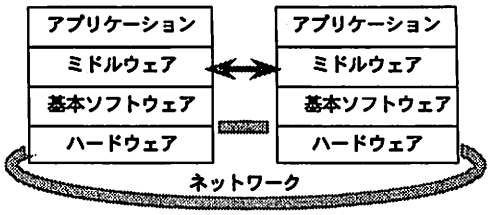


図4　ソフトウェア構成の階層化

具体例として，標準化団体OMG(Object Management Group)が普及をはかっているCORBA(Common Object Request Broker Architecture)やマイクロソフト社のDCOM(Distributed Component Object Model)がある.

第4のコンポーネントベース開発の実現はこのような背景のもとになされた．個々のアプリケーションが作りやすくなるほか，ネットワーク透過性の実現によりネットワークを経由したアプリケーション間の連携も容易になる.

4.4 コンポーネントの粒度的課題の解決

オブジェクト指向部品の活用はまずグラフィカルユーザインタフェース（GUI）の分野で効果をあげた．その理由はボタンやスクロールバーなどのGUI部品はその機能を理解しやすいためである．しかしながら一般には抽象データ型レベルのクラス部品は粒度が小さく，プログラミング技術が要求されるので，部品の有効活用が難しいという問題があった．そこで，アプリケーションと部品の間の粒度的なギャップを埋めるために，最近では従来の部品の概念を発展させたコンポーネント，デザインパターン，アプリケーションフレームワークなどの技術が開発された.

たとえば，従来のオブジェクト指向プログラミングにおいては，プログラム部品を型（クラス）として定義しておき、実際のアプリケーションの中では必要に応じてその型から実体（インスタンス）を生成して利用する方法がとられていた．それに対して，狭い意味でのコンポーネントは，このようなクラスからインスタンスを生成するという概念を必要とせず，はじめからインスタンスオブジェクトとして扱われる．そのためプログラムの定義内容を知る必要はなく，ブラックボックスとして扱えばよいので利用が簡単である．インターネット経由で取り込んできてすぐに利用できるコンポーネントも増えて

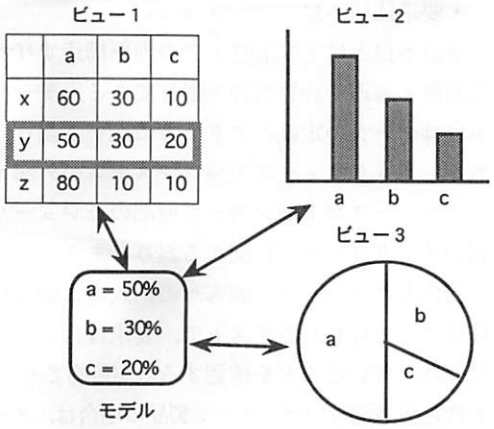


図5　デザインパターンの応用例
（文献(7)から引用）

いる．

デザインパターンは，一般性のある設計問題を解くために複数のクラス間の典型的な協調作業をパターン化したものであり，見かけ上の部品の粒度を大きくする役割を有する．文献(7)のObserverと名付けられたパターンを用いた例を図5に示す．これはアプリケーション側でデータを管理し，ユーザの要求に応じて見やすい形式で表示するという機能の実現方式を示している．データ管理をモデル，表示をビューと呼ぶ．ここでビュー1で変更された値をモデルと他の2つのビューに反映させる手順を考えてみる．このパターンでは，ビューの増減や表示内容に影響されない手順として，まず値を変更したビューがモデルに変更内容を通知し，次にモデルがすべてのビューに変更された旨の連絡をし，必要に応じてビューはモデルに変更内容を要求するという方式が用意されている．最近のアプリケーションに多いこのような処理は，デザインパターンを利用すれば，モデルの元になるクラスとビューの元になるクラスを用いて簡単に実現できる．

アプリケーションフレームワークは，特定のアプリケーション分野に対して，そのソフトウエアアーキテクチャの基本的な枠組みとそれを構成するクラスライブラリをあらかじめ用意しておくものである．個々のアプリケーションはそのフレームワークをカスタマイズして簡単に構築できる．

今後，このようなコンポーネントウェアをベースに，情報システムの構築，利用，保守の形態が激変するとともに，ソフトウェアの信頼性に関しても開発工程での不良の作りこみよりも，どのように信頼性の高いコンポーネントを組み合わせて使用するかが重要になる．

5．テスト・検査技術

5.1　プログラム検証の方法

プログラムの検証は，プログラムが仕様どおりに作られていることを確かめるのが目的であり，一般には次のような方法が考えられる．

(1)　仕様書からプログラムを自動生成
(2)　仕様書とプログラムの等価性証明
(3)　テストデータによる動的テスト

最初の自動プログラミング方式はプログラム開発方式の理想形態であり，常に正しいプログラムが生成されるので検証する必要はない．この実現には，仕様を形式的に記述する仕様記述言語と仕様からプログラムへの変換技術あるいは仕様を満たすプログラム部品の検索技術などが必要であり，まだ研究段階にある．

次に仕様書とプログラムの等価性を自動証明する方式はプログラム検証方式の理想形態であるが，自動プログラミング方式の課題のほかにプログラムの意味理解が必要となり，実現は難しい．

そこで現実には，第3の方法として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現在最も一般的に用いられている．

5.2 テスト工程

プログラムの作成を人手に頼る現在の開発方式では最終的にはプログラム検証のためのテスト工程が不可欠である．テスト工程は幾つかの小工程にわけられるが，図2(b)に示したモジュールテスト，プログラムテスト，システムテストは開発者によるテストである．その後，開発部門とは独立した検査部門によるテストが実施され，納入時に顧客側の受け入れテストがある．

モジュールテストは単体テストとも呼ばれ，プログラムモジュールが仕様書どおりに作られているか否かを検査するのが目的である．次にプログラムテストは結合テストや連動テストとも呼ばれ，いくつかの関連するモジュールを結合して実行し，モジュールを呼び出す側と呼び出される側のインタフェースが一致していることを確かめるのが目的である．システムテストでは，プログラム全体がシステムの仕様を満たしていることが確かめられる．これらのモジュールテストからシステムテストまではプログラムの内部構造をよく知っている開発者自身がテストを行うため，細部にわたってきめ細かいテストが行われる反面，テストの内容が偏ったり，見落としが発生しやすい．

そこで，その欠点を補うために，開発にたずさわってこなかった検査部門が第3者の立場あるいはユーザの立場で改めて行なうシステムテストが検査である．網羅的なテストや様々の異常なケースのテストが徹底して実施される．

文献[8]によると，開発者とは独立したテストによって検出された約16,000件の不良を分析した結果，検出頻度は1,000ステップ当り2〜3件で，主な不良原因は以下の通りであった．括弧内は全体に占める割合である．

- 構造不良　(25%)
- データ不良　(22%)
- 機能設計不良　(16%)
- コーディング不良　(10%)
- インタフェース不良　(9%)
- 要求仕様不良　(8%)

構造不良とはソフトウェアの内部構造すなわち制御・論理や計算処理の誤りである．データ不良はデータの定義とアクセスに関する誤りである．インタフェース不良はシステムの外部インタフェースおよびシステム内部のモジュール間のインタフェースに関する誤りである．

受け入れテストは，顧客への納入時に顧客自身によって行われるテストで，要求仕様どおりに作られていることを確認するのが目的である．不特定多数向けのパッケージ製品の場合は，メーカ側の検査部門の行なう検査が受け入れテストを兼ねる．

5.3 動的テスト法

このような動的テスト法には実用面で2つの問題がある．その第1は効果的なテストデータの作成の困難性という本質的な問題である．テストは誤りの検出を目的に行われるものであり，誤りのないことを保証するものではない．そのため，プログラムの信頼性は，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有無に依存する．

第2の問題は図6に示すようにテストの準備や結果確認の作業に多くの人手作業を必要とすることである．主な作業を以下に示す．

(1) 仕様書からのテスト項目の抽出
(2) 対応するテストデータと予想結果の決定
(3) テストデータを用いたプログラムの実行
(4) 実行結果と予想結果の一致の確認
(5) 誤り検出時の原因究明とプログラム修正

このように動的テストでは，限られた時間と費用の中で高い信頼性を保証できるテストデータを作成する必要がある．その代表的技法として，プログラムの機能仕様に着目する機能テスト法とプログラムの内部構造に着目する構造テスト法がある．

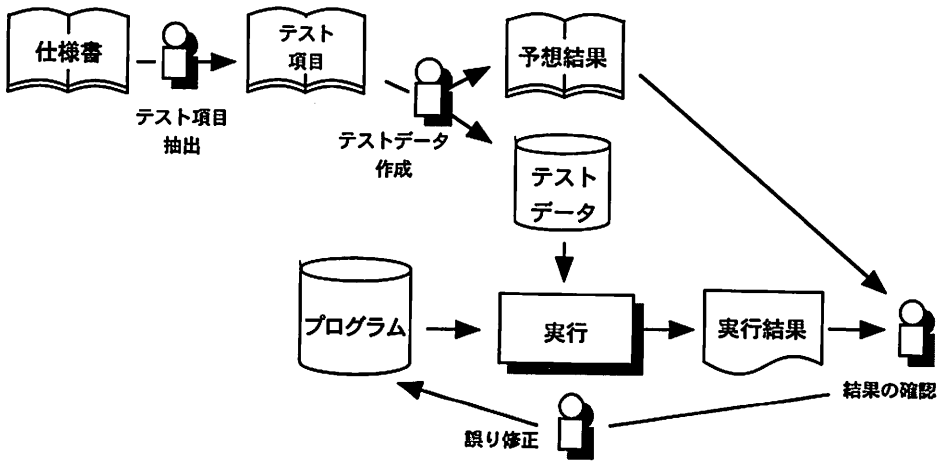


図6　動的テストの実行手順

5.4　機能テスト法

機能テストはプログラムの機能仕様からテスト項目を選ぶものである．プログラムのソースコードは見ないので，ブラックボックステストとか仕様テストとも呼ばれる．なお，テスト項目（テストケース）に対応する具体的な入力値をテストデータと呼ぶことにする．

機能テストの具体的技法は少ないが，一般的方法として機能仕様に記述された入力条件や出力条件および無効な入力条件を表に記入していく方法がある．表の各項目ごとに1個のテストデータを試すだけで他のデータも同じテスト結果になることが期待できるように外部条件を分割するので同値分割法と呼ばれる．この方法は，入力領域を幾つかのドメインに分割し，各ドメインから1つずつテスト項目を選択するドメインテストと呼ばれる方法の1つである．なお，同じ入力でも内部状態などの違いで出力が異なるような場合は出力状態も考慮した表にする必要がある．

表2　同値クラス表の例

入力条件	有効同値クラス	無効同値クラス
文字数	4～8	3以下，9以上
文字の種類	英字と数字の組合せ	英字のみ，数字のみ
（以下略）		

この方法をパスワード入力処理プログラムのテスト項目選択に適用した例を表2に示す．この表は同値クラス表と呼ばれる．この同値クラス表を用いてテストデータを作成する場合，有効同値クラスのテスト項目については1つのテストデータが多くのテスト項目を含むように選ぶことができ，効率よくテストできる．一方，無効同値クラスのテスト項目については個々に対応するテストデータを作成してテスト漏れが生じないようにする．また，具体的な値の選択にあたっては，その条件の限界値やそれから最小単位分だけずれたものを選ぶ．これはプログラム内の条件判定のところで発生しやすい限界値の判定誤りを見つけやすくするためである．

機能テストの自動化ツールとしては，機能仕様を組合せ論理で表現してテスト項目を作成す

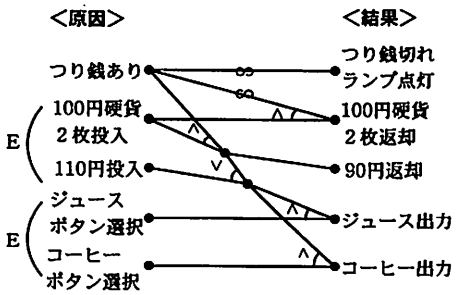


図7　原因結果グラフの例

る原因結果グラフ法に基づくものがある．その手順を次に示す．

(1)　機能仕様を適当な大きさに分割
(2)　各々の入力（原因）と出力（結果）を識別
(3)　原因結果グラフを作成
(4)　このグラフを決定表に変換
(5)　この決定表の各列をテスト項目に変換

自動販売機の制御プログラムを例をとると，組合せ論理に制約条件を付加した原因結果グラフは図7のようになる．図中の∧，∨，∽は各々，論理積，論理和，否定を示す．原因側のEは2つの原因の成立条件が排他的であることを示す．

5.5　構造テスト法

構造テストはプログラムの内部構造に基づいてテスト項目を選ぶので，ホワイトボックステスト，プログラムテストなどとも呼ばれる．その基本的な方法として，プログラムの実行の流れを示す制御構造を制御フローグラフと呼ばれる有向グラフで表わし，そのパス解析に基づいて入口から出口に至る幾つかのパスをテスト項目として選ぶことから，パステストとも呼ばれる．

この場合，すべてのパスをテストするのが基本であるが，通常のプログラムでは繰り返し処理を含むためパスの数が膨大となり，すべてのパスの実行は不可能な場合が多い．そこで，実際にはいくつかの簡易化されたテスト網羅基準が用いられる．

一般によく用いられるのは，全アークを1回以上実行するようなパスセットを選ぶ全アーク網羅基準である．これはすべての分岐を1回以上実行するので全分岐網羅（分岐テスト）とも呼ばれる．しかし，この基準では繰り返し処理に関しては繰り返し条件の成立回数が1回の場合だけテストすればよいことになる．そこで，繰り返し数に依存した誤り検出のために，0回の場合や複数回(最大数または2回)の場合もテストした方がよい．また，パスの数が膨大となる繰り返し処理を対象外とすればパスの数は有限となるので，全パス実行が可能になる．

以上に述べた基準はいずれもプログラムの実行の流れを示す制御フローだけに基づいていたが，変数の値の定義参照関係を示すデータフローに着目した網羅基準もある．たとえば，変数値を定義しているノードからそれを参照しているノードまでの部分パスに着目してテストする方法である．

しかし，このようなパス解析に基づく構造テストでは誤りの発生しやすい分岐条件自身のテストが不十分になりがちである．そこで，分岐条件が論理和や論理積を用いた複数の条件からなる場合は論理式を分割し，各々が真と偽の場合をテストする．さらに，各条件が値の大小を比較する式の場合はその比較演算にかかわらず，<，=，>の3種類のテストを行なう．

さらにプログラムテストやシステムテストでは，モジュール呼び出しに関連して，全モジュールを1回以上実行する基準や全モジュール呼び出しを1回以上実行する基準などがある．

構造テストの運用に際しては，テストデータ生成のためのパスを機械的に選んだ場合は実行不可能なものが混じることがある．そこで，実際の構造テスト支援ツールでは，人手で作成したテストデータを用いてテスト実行したときに，プログラムの中で実行された部分を表示するものが多い．この情報をもとに未実行の部分を実行するテストデータを追加することにより，テスト網羅率が向上する．

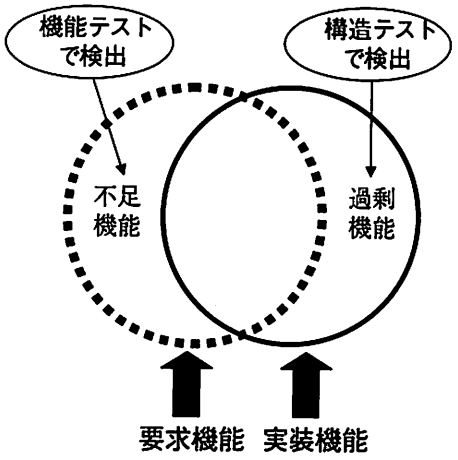


図8　機能の過不足の検出方法

ただし，大規模ソフトウェアのシステム全体のテストでは網羅基準の100%達成は難しいことが多い．特にすべての例外処理を実行して検査するのは技術的にもコスト的にも難しい面がある．この場合は全体をいくつかのサブシステムに分割し，各々のテストで100%の網羅基準を達成するのが実際的である．

なお，必要な機能がプログラム化されていない場合はテスト対象となるべきパスが存在しないので，構造テストではその検出が原理的に不可能であり，機能テストで検出する必要がある．したがって，図8に示すように，要求機能と実装機能のずれのうち，不足機能は機能テスト，過剰機能は構造テストによって検出する必要がある．実際には特定の不足機能と過剰機能が同一のミスによる場合も多いが，セキュリティ的観点では，過剰機能がないことを確認することが重要である．

以上に述べた技術を利用した実用的なテストの手順の一例を以下に示す．

(1) 機能テスト法によるテスト項目の選択
(2) 対応するテストデータの作成と実行
(3) 未実行の分岐を検出し，テストデータ追加
(4) テスト網羅率が基準に達するとテスト終了

6．おわりに

以上，ソフトウェアの信頼性に関して，対応する技術とその限界について述べた．やや悲観的な結言になるが，最初に事例で述べたように，現在の技術では，人間はプログラムの開発時に設計ミスをするし，検査ではテスト漏れによりそのミスを見逃してしまうというヒューマンファクタの問題が残されている．

トーマス・クーンの「従来のパラダイムが行きづまり，危機的状況になったとき，新しいパラダイムが生まれ，科学革命が起こる」という科学史観[9]を借りれば，現在はまさに高度情報化社会への激しい変化のなかで，従来の手続き型のパラダイムに代わる新しいプログラミングパラダイムが求められている．

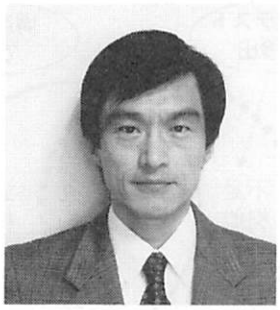
残念ながら現時点では，人間は自らが作りだしたコンピュータを完全にはコントロールできていないし，コンピュータも自らをつくりだした人間を超えられてはいない．頻発する例外処理の漏れに見られるように「人間は予期せぬ出来事を予期できるか？」という逆説的命題を乗り越えて，新しいパラダイムが確立する日まで，最善の努力はするにせよ，コンピュータは人間が教えたように間違えるということを忘れてはならない．

参考文献

(1) 中所武司：ソフトウェア工学，朝倉書店（1997.5）
(2) 保田勝通：ソフトウェア品質保証の考え方と実際，日科技連出版社（1995.11）
(3) 中村英夫：ソフトウェアの信頼性と2000年問題，日本信頼性学会誌，Vol.21，No.7，pp.378-381（1999.9）
(4) 中所武司：知の現在：なぜコンピュータは間違えるか？，思索の樹海，明治大学，pp.41-46，(1998.4)

(5) 飯塚悦功（監修）：品質システム要求事項の解説，日科技連(1998.11)
(6) 中所武司：今なぜコンポーネントウェアか，コンポーネントウェア，共立出版，pp.1-12 (1998.8)
(7) E.Gamma, R.Helm, R.Johnson and J.Vlissides :Design Patterns, Addison-Wesley (1995)
(8) ボーリス・バイザー（小野間 訳）：ソフトウェアテスト技法，日経BP出版センター (1994.2)
(9) T. Kuhn（中山茂 訳）：科学革命の構造，みすず書房(1971.3)

（ちゅうしょ　たけし／明治大学）



中所　武司

1969年東京大学工学部電子工学科卒業．1971年同大学院修士課程修了．同年(株)日立製作所入社．同社システム開発研究所主任研究員を経て，1993年から明治大学理工学部情報科学科教授．ソフトウェア工学の研究に従事．1982年度情報処理学会論文賞，1986年度大河内記念技術賞受賞．著書「ソフトウェア工学」(朝倉書店)など．電子情報通信学会，日本ソフトウェア科学会，人工知能学会，IEEE Computer Society，ACM各会員．工博．

日本信頼性学会誌「信頼性」
Vol.22, No. 5　2000. 7月号
(通巻105号)

目　　次

お知らせ …… i
目　次 …… V
巻頭言 …… 堀籠　教夫 …… 365
展　望　「ソフトウェアの信頼性」
ソフトウェアの信頼性 …… 中所　武司 …… 367
大規模ソフトウェアの高信頼化技術 …… 佐藤　和夫 …… 379
ＦＡ組み込みソフトウェアの高信頼化技術 …… 南角　茂樹 …… 387
情報制御システム統合のための高信頼化技術 ……
…… 新井　利明、関口　知紀、中橋　晃文 …… 395
解　説　構造物のライフサイクルエンジニアリングにおける信頼性理論の役割
…… 鈴木　誠 …… 404
サロン　バタフライ物語 …… 藤川　忠重 …… 412
「心」と「技術」の関係を研究する会に参加を！… 濱野　恒雄 …… 413
報　告　ブリヂストン見学会報告 …… 木村　紳 …… 416
勝関橋見学会続報 …… 濱野　恒雄 …… 419
第８回研究発表会報告
…… 野見山敦子、中村　國臣、鈴木　和幸、石田　勉 …… 420
学会情報　平成１１年度第６回理事会議事録 …… 総務委員会 …… 429
第22回年次総会プログラム …… 432
第22回年次総会議事録 …… 450
平成11年度奨励賞講評…… 表彰委員会 …… 454
会員状況 ……456
編集後記 …… 絹川　博之 …… 457

論　文　三つの確率論的手法を用いた順序依存形故障論理の定量化
…… 龍　偉、佐藤　吉信、堀籠　教夫 …… 461
ワイブル分布の統計的推測とその周辺
３パラメータワイブル分布を中心に …… 廣瀬　英雄 …… 473

The Journal of Reliability Engineering Association of Japan
Vol.22 No.5 July 2000

Contents

Preface Michio Horigome

Special Survey

—Reliability on Software—

Reliability on Software Takeshi Chusho

Highly- Reliable Technology for the Huge Functions' Software
Kazuo Sato

The High Trust Technology of the Software for Factory Automation
Shigeki Nankaku

Dependable Computing Technology for Combining Information and Control systems
Toshiaki Arai, Tomoki Sekiguchi & Teruyasu Nakahashi

Review

Role of Reliability Theory in Life-Cycle Engineering of Civil Structures
Makoto Suzuki

Salon

Story of Butterfly Tadashige Fujikawa

Invitation to a Group for Studying the Relationship between Spirit and Technique
Tsuneo Hamano

Report

Visiting Report of Bridgestone Co. Shin Kimura

Visiting Report of the Bridge "Kachidoki" (sequel) Tsuneo Hamano

Eighth Annual Symposium on Reliability
Atsuko Nomiyama, Kuniomi Nakamura,
Kazuyuki Suzuki & Tsutomu Ishida

From Editor Hiroshi Kinukawa

Paper

Quantification of Sequential Failure Logic Using Three Probabilistic Approaches
Wei Long, Yoshinobu Sato & Michio Horigome

Statistical Inference on the Weibull Distribution and Its Related Topics
Three-parameter Case Hideo Hirose

Published by Reliability Engineering Association of Japan

REAJ誌 2000 Vol.22, No. 5 (通巻105号)