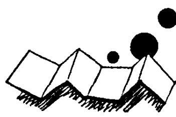
Vol. 23 No. 4 情報処理 Apr. 1982

解説



プログラムの検査[†]

岸田孝一[††]

1. はじめに

「デバッグ（虫取り）」と呼ばれる原始的なプログラム検査技法は，プログラミングの歴史が始まったころから存在した．メモリ・ダンプやトレーサなど，そのための素朴な自動化支援ツールはかなり古くから作られ，現在でもなお利用されている．

しかし，残念ながら，単なるデバッグだけに頼るだけでは，プログラム中の「虫」を完全に取り除くことはむずかしい．この争実を最初に指摘したのは，E. W. Dijkstra であった．第2回 NATO ソフトウェア工学会議（1969年 Rome）で発表した「構造化プログラミング」に関するワーキング・ペーパの中で，かれは次のように述べている．

——テスティングは，プログラム中にバグが存在することを証明する手段としてなら使えるが，しかし決して，バグの不在証明にはなりえない[1]．

このことばは最初，テストあるいはデバッグの無用性だけを示唆するものと単純に解釈された．こうして，1970年代の前半から半ばにかけて，いわゆる「構造化ブームの嵐が吹き荒れ，各種の新しい技法やツールが，デバッグやテストを不要にするという宣伝文句つきで，相次いで提唱された．一部には，Go to 文を使いさえしなければ，だれにでも正しいプログラムが書けるかのような迷信さえ生まれたのである．

しかし，それらの構造化手法が一応の評価を得て世に普及された今日でも，プログラム中のバグの人口密度は，それほど顕著に減少したようには見えない．構造化プログラミングの現実的な効用は，それらのバグの存在を突き止めるために必要なテスティング作業を以前よりも容易にしてくれたというだけにすぎない．

ソフトウェア開発プロセスの中で人間が重要な役割をになっている以上，その成果物としてのプログラムの中に，さまざまな（人為的）誤りが含まれることは避けられず，したがってテスティング（検査）は，開発工程の重要な一部分を感じている．近代的なプログラム検査の方法論は，前述の Dijkstra の発言を逆手にとって，プログラム中に必ず存在するであろうバグを，システマティックなやり方で，可能なかぎり数多く発見することを意図して，組み立てられている．

本稿では，そうしたプログラム検査のための技法やツールを概観し，その効用や限界について簡単に説明する．

2. テストとは何か

2.1 デバッグとテスト

近代的なプログラム検査の方法論は，デバッグとテストとをはっきり区別することから始まる．

デバッグは，プログラムの具体化工程の一部であり，プログラマ自身の手で行われる．その目的は，プログラム中のバグを発見し，それを修正してプログラムを正しいものにすることである．

一方，テスト工程は，プログラマがもう自分のプログラムにはバグが残っていないと確信して，プログラミング作業の完了を宣言したあとを引き継いで行われる．それは一種の制御実験であり，人間の手で完全にコントロールされた環境下でプログラムを実行し，プログラムの持っている機能を明らかにすることを目的としている．

本来要求された機能がそれとちがった形で存在したり，あるいは不必要と思われる機能が含まれていたりした場合，それらはすべてバグであろうと推測される．経験が示すところによれば，ほとんどすべてのプログラムは，プログラマの手を最初に離れた時点では，このようなバグを必ず複数個含んでいる．したがって，この事実にもとづいていいかえれば，テスト（検査）とは，「バグの発見を意図して，プログラムの実行を試みること」だといえよう[2]．

もともとバグのないプログラムを書こうと努力して

---

† Program Testing by Kouichi KISHIDA (Software Research Associates, Inc.).
†† (株)ソフトウェア・リサーチ・アソシエイツ

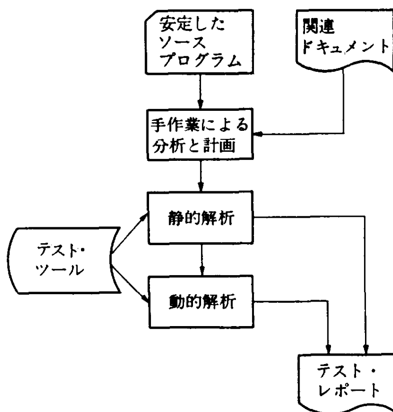


図-1 テスト工程

作業し，しかもデバッグ過程を通じてそのことの確信を固めてきたプログラマ自身にとって，このような客観的な態度でテストを行うのは非常にむずかしい．かれらは，一種の心理的な壁によって，バグの発見からさえぎられているからである．

そこで，一般に，テスト活動は，プログラミング（デバッグを含む）から完全に切り離して，プログラマとは別の人間の手で実施するようにするのがよいと考えられる．プログラム開発チーム内に専任のテスト要員を置くか，開発チームとは独立したテストチームを作るか，あるいは第三者機関にテストを委託するか，といった方法が，時と場合によって使い分けられるべきであろう．

2.2 テスト工程

図-1 は，テスト工程の標準的な流れを示す．テスト担当者の仕事は，プログラマによるデバッグが完了し，一応の「安定状態」に達したプログラムのソース・リストを，仕様書，設計書，デバッグ報告書などの関連ドキュメントと突き合わせながら，手作業による分析を行うことから始まる．

それは，テスト担当者が対象プログラムの機能や内部構造を理解するための準備段階であり，同時にそれ以降の作業の進め方を考える計画の段階でもある．

テスト計画の全体的な枠組みは，主としてプログラムの構造にもとづいて組み立てられ，一方テスト・データは，プログラムの機能仕様から作り出される．もちろん，プログラマがデバッグ時に用いたデータが残っていれば，それはテストの出発点として利用できる．

手作業による分析と計画が終わると，続いて，自動化されたソフトウェア・ツール群の助けを借りたテストが実施される．この段階は利用されるツールの性質によって，静的テストと動的テストの２段階に分けて考えられる．

前者は，文字通り，プログラムを動かさずに，ただそのソース・コードを解析して，見かけ上の誤りや異常を検査する段階である．素朴に考えると，この種の解析は プログラムのコーディングが完了してデバッグに入る直前に位置づけるべきであるかのようにみえる．しかし，徹底的な静的解析には，かなりのマシン・タイムや労力を要するので，デバッグ完了後に，プログラムの形式上の完成検査という形で行うほうがよい．その結果として得られるプログラム構造に関する情報は，当然，テスト計画に反映される．

一方，動的解析は，実際にテスト・データを投入してプログラムを動かす形で行われる．テスティング作業の中で最も大きなウェイトが行われるのはこの段階であり，そのためのツールとしても，いろいろなものが用意されている．

動的テストにおいては，単に入力データとプログラムから出力を突き合わせて機能上の検査を行うだけでなく，そのデータがプログラム中でどのように処理されたかのチェックも行われる．このように，テスト過程におけるプログラムの内部動作を目に見える形で把握し，それを定量的に評価しようというのが，最近のプログラム・テスト技術の１つの特徴でもある．

テスト作業を通じて発見された誤りや異常は，ふつう，テスト担当者によってすぐその場で修正されることはない．テスト結果は，管理者を通じてプログラマに報告され，プログラマが修正を行う．なぜなら，テストの目的は，デバッグとはちがって，正しいプログラムを作ることでなく，プログラムの正しさの度合いを定量的に判定することだからである．こうした目的意識を区別することによって，テスト担当者の対象プログラムに対する心理的客観性が保たれる．

3. テストの指導原理

3.1 計測性

テストは，一種の制御実験であり，したがってその進歩は，客観的・定量的な指標によって測定されなければならない．デバッグとテストとを区別する第一の原理は，この「計測性」である．

いわゆる「機能テスト」および「構造テスト」という対極的な２通りのテスト方法論は，この計測性の概

念と関連している．

伝統的なデバッグやテストは，もっぱら機能テストを重視する形で行われてきた．すなわち，仕様書にもとづいて作られたデータをプログラムに投入し，得られた出力が予想通りのものかどうかを調べるやり方である．それは，プログラムの内部構造を無視して，外側から見た機能だけを検査する「ブラックボックス・テスト」である．

プログラムは，その最終利用者にとっては一種のブラックボックスであり，その意味で，機能テストが有用であり，また重要であることはいうまでもない．しかし，機能テストの進行過程を客観的かつ定量的に計測することは非常にむずかしい．また，ブラックボックス方式では，われわれにとって望ましい機能がプログラム中に実現されているかどうかの確認はできるが，逆に望ましくない余分な機能やコードが含まれていないかどうかの検査は，ほとんど不可能である．

そこで，この欠陥を補うために，「ホワイトボックス」方式による「構造テスト」が，最近，重要視されるようになってきた．

これは，それぞれのテストの実行にさいして，プログラムの内部動作をモニタリングし，プログラムの各部分がどのように起動されたかを調べようというものである．テスト結果としては，通常の出力データの他に，入力データがプログラム措置全体のうちのどの程度をカバー（通過）したかが報告される．この「テスト・カバレージ」（被覆度または網羅度）は，現在，実務上有効に利用できるほとんど唯一のテスト計量化指標であり，さまざまな種類のものがそれぞれ異なる目的のために提案されている[3]．

3.2 分割可能性

ふつう，大規模なソフトウェアの開発は，システム全体をいくつかのモジュールに分けて行われる．そのようなプログラムのテストは，モジュール・レベルとシステム・レベルの2段階に分けて実施される．そのさい重要な意味を持ってくるテストの指導原理は，構造化プログラミングのそれと同じ「分割可能性」の考え方である．

いくつかのサブモジュールから構成されるプログラムの場合，個々のサブモジュールおよびメイン・モジュールだけの個別の単体テストの総和は，システム全体を組み合わせた統合テストとまったく同じ効果を持つものでなければならない．

プログラムの機能および構造を徹底的に実施するには，テスト対象の大規模なプログラムを，操作しやすいモジュール単位に分割できるなら，それにこしたことはない．もしそうしたモジュール・テストが完全な形で行えれば，統合システム・テストの役割は，単に形式的な確認の1ステップにすぎなくなる．

モジュール・テストにさいして，システムの階層措置のどちらから出発するかで，「トップダウン」および「ボトムアップ」の2つのアプローチが区別される．前者のやり方では，まだ実現されていない下位モジュールの機能をシミュレートする「スタブ」が必要であり，後者の場合には，架空の上位モジュールの役割を果たす「ドライバ」が必要となる．いずれにせよ，単独のモジュールをシステムの他部の分から切り離してテストするには，そのための特殊なテスト環境を用意してやらなければならない．そのような環境は，いわばテスト対象モジュールに対する入力の一部分を構成するものである．

3.3 再現性

発見されたバグの修正や，仕様または設計の変更に伴うプログラムの手直しが行われたときには，その正しさを確認するために，以前に行ったのと同じテストをもう一度くりかえす必要が生じてくる．すなわち，テスティングにおける第3の指導原理は「再現性」の概念である．

バッチ型のテスト環境では，そうしたテストの再現を期するためには，テスト・データおよび出力結果の保存および管理のためのシステムを用意してやらなければならない．一方，端末装置を通じて，人間がプログラムと対話しながらテストを進めて行くオンラインの環境では，キーボードから入力されたテスト・コマンドや入出力データの詳細なログ（記録）を取っておくという機能が，きわめて重要な意味を持ってくる．

マルチ・プログラミング・システムやリアルタイム・システムなどの制御をつかさどる中心的なモジュールのアルゴリズムは，時間に依存しており，同一テストの再現が困難である．この種のプログラムのテストは，むしろ静的な形で，ときには，いわゆるプログラムの正しさの数学的な証明手法を用いて行われる．

4. テスト・ツール

プログラム・テスティングの仕事は，比較的単純な検査手続きをかなり多くの回数をくりかえして実行するということを含んでいる．そうした体系的な反復作業を通じて，対象プログラムの正しさについての確証

が，少しずつ，定量的に積み上げられていくのである．

もし，何らかの機械的支援も得られないとしたら，それはきわめて退屈な作業になるであろう．他人の書いたプログラムを読み，あるいは実行してその正しさを検査するためには，それが開発されてきた工程をもう一度追体験しなければならず，ときには自分で新しくプログラムするよりも多くの労力と時間を必要とするだろう．

しかし，現実には，テストに割り当てられる資源はそれほど十分でなく，きわめて限られた時間内に，少ない労力で，しかも一定水準以上の成果を達成することが義務づけられることが多い．そこで，各種の自動化ツールの利用が必要になる．

プログラム・テキストを読んでその内容を分析する形の自動化ツールの歴史は，ほぼ，コンピュータの創生期にまでさかのぼることができる．アセンブラやコンパイラなどはそうしたツールの一種であり，一部にテストの機能を含んでいる．しかし，テストを主目的とするプログラム分析ツールが真剣に考えられ始めたのは，比較的新しく，1970 年代の半ば以降のことである．

これらのツールの体系は，前述したテストプロセスのどこで利用されるかによって，

（1） 静的テスト・ツール

（2） 動的テスト・ツール

の2種類に分類される．以下，それぞれの分類ごとに，どのようなツールがあるかについて概説する．

4.1 静的テスト・ツール

（1） コード監査

これは最も原始的なツールであり，ソース・プログラムを読んで，その書き方（スタイル）に関するチェックを行うものである．すなわち，あらかじめ定められた規約に照らして，プログラム中の各ステートメントを検査し，もし違反したものがあれば，エラー・レポートを作成する．コーディング規約は，別にプログラムの機能とは無関係ではあるが，プログラムの形式を統一し，読みやすさを向上させることによって，間接的にその信頼性や保守性に寄与する．

一般にプログラマは，創造的であろうとするあまりに，標準化規約の順守についてはルーズであることが多く，コード監査は，製品としてのプログラムの外見上の品質を整える上で，予想外の効果をもたらす．

（2） 静的解析

これは，コンパイラのフロント・エンド部における文法チェックをより徹底的に拡張した形で，ソース・プログラムの分析および検査を行うものである．

静的解析ツールが行う検査の主要なねらいは，プログラム中に存在する構造上の異常や誤りの可能性をしらみつぶしに見つけ出すことである．それは，大きく分けて，制御の流れに関する検査（例：デッドコード・や無限ループは存在しないか）と，データの構造や流れに関する検査（例：すべての変数の値は，それが参照される前に定義されているか）とに分かれる．

静的解析ツールの持つ問題点の1つは，そこでの検査があまりに徹底的でありすぎるために，異常レポートに報告されるものの多くが実際には誤りでないことである（ふつう，報告された異常のうち真の誤りであるものの割合は 10～20% 程度だといわれる）．したがって，その実施には，かなりのマシン・タイムと人間の労力を必要とする[6]．

（3） 記号実行・評価

これは，まだ実験段階にあるツールだが，その正しさが本当にクリティカルであるような小規模なプログラム・モジュールの論理を数学的に検証するには，有用だと考えられている．

その基本的なアイデアは，プログラム中の各変数に実際のデータでなく，記号的な値を与えることで，プログラムに実行過程の論理的なシミュレーションを実施し，途中経過や最終出力を検査しようというものである．これまでに，いくつかの実験的システムが試作され，一定の成果を達成している．

4.2 動的テスト・ツール

（1） テスト・データ生成

プログラム・テストの理論的基礎として評価されている Goodenough および Gerhart の「定理」には，次のことが述べられている[4]．

――もし，あるプログラムについて，テスト・データの選択に関する一貫した完全な判断基準が存在し，それを満足するようなデータを用いて行われたテストの結果が正しかったならば，そのプログラムは正しい．

この定理は，体系的なテストが，数学的なプログラムの正しさの証明と同程度の力を持ちうる可能性を示唆するものではあるが，しかしそこには，実用的なテスト・データ生成の指針については何も述べられていない．

何らかの計画にしたがって，実際にテストを行うに

は，ふつう大量のテスト・データまたはテスト・ケースを用意しなければならず，それにはかなりの労力を要する．この人手を省くために，事務計算プログラムのテストには，よく，テスト・データ・ファイルの自動生成ツールが利用される．

この種のツールは，たしかに量的には十分なデータを提供してくれるが，データの質は，特に構造テストのカバレージについていえば，不十分であることが多い．しかし一方，プログラムのある構造的な特性をテストするためのデータを生成することは，一般にきわめてむずかしく，まだ有用な自動化ツールが開発されるまでにはいたっていない．

テスト・データの完全性をチェックする最新の技法としては，変異分析 (Mutation Analysis) と呼ばれるものがある．これは，プログラムに人為的な変形（誤り）を加えて，テスト・データがそれを検出しうるか否かを調べようというもので，現在はまた実験段階を出ないが，今後の発展が注目される．

（2） テストベッド

プログラムの開発にさいして，モジュール化の技法が一般化するにつれ，テストにおいても単独のモジュールだけを取り出して検査するための人工的な環境が，しだいに整備されはじめた．これが，モジュール・テストベッドと呼ばれるものである．数年前までは，バッチ型のテスト環境が多かったが，最近では，TSS を利用したオンライン対話型のテストベッドが，あちこちで実用化され，効果をあげている．

テストベッドは，単にドライバやスタイブを提供するだけでなく，テスト・データの自動生成や，テスト過程の自動記録，および過去の記録を利用した再テスト機能などを持たなければならない．また，ふつうは，テスト作業の定量化のために，後述のカバレージ分析の機能もそこに含まれることが多い．

（3） 実行モニタリング

きわめて素朴なデバッグ・ツールの一種に，トレーサやスナップショットがある．これらはいずれも，対象プログラムの実行プロセスを詳細に追跡するためのものである．

それをより洗練して，ソース・プログラム中の各ステートメントごとに，実行時の行動をモニタリングして統計的なレポートを作成するツールを，自己計測型インストルメンタと呼ぶ．この種のツールは，1970 年代の半ばにいくつか試作されたが，その利用結果はきわめて衝撃的であった．すなわち，プログラマが「100%デバッグ完了！」を宣言した時点で計測を行うと，全ステートメントのうち，わずかに 20〜50% しか，それまでのテスト・データでは動かされていない，というのである[7]．

この事実に立脚し，かつインストルメンタの性能をより向上させ実用化するために，プログラム中の分岐点だけに計測命令を挿入（インストルメント）して，分岐の何%をテスト・データが実際に通過したかを計測・分析するツールが実用化された．このパーセンテージを一般に C_1 カバレージといい，それを計測するツールをカバレージ・アナライザと呼ぶ．

何ら理論的根拠があるわけではないが，経験の示すところによると，$C_1 \geq 85$ 程度にまでモジュール・テストを実施すれば，ほとんどすべてのバグは発見されるといわれる．もちろん，単に分岐が実行されただけではなく，その結果が機能的な観点から正しいことを人間の目で確認する必要があることは，いうまでもない[8]．

現実のプログラムでは，安全性（プログラムの堅固さ）を高めるための論理的デッド・コードが意図的に挿入されていたり，テスト・データの作成が困難だったりするので，C_1 が 100% に達することはまれである．

複数のモジュールを結合したシステム・テスト時には，C_1 ではなく，システム内に含まれるモジュール呼出し命令の何%が実行されたかを示す S_1 と呼ばれるカバレージ指標が，同様に用いられる．S_1 の目標値はただし 100% である．

（4） 動的表明検査

プログラムの正しさに関する表明 (Assertion) の概念は，構造化プログラミングの方法論とともに，C. A. R. Hoare らによって提唱された．動的表明検査と呼ばれるツールは，プログラマまたはテスト担当者の手によって，プログラム中の任意の場所に挿入された表明が本当に成り立つか否かを，プログラム実行時にチェックしようとするものである．

表明は，一般に論理的命題の形をしており，特殊なコメントを用いて記述される．それを適当なプレ・コンパイラで処理して，チェック用のステートメントをプログラム途中に挿入するわけである．

動的表明検査の考え方は，プログラムの正しさの証明と実務的テスティングとの，ちょうど中間に位置している．それは，たとえばカバレージ分析と組み合わせて適切に使われるなら，プログラムの品質を向上さ

せる上できわめて有効であろう．しかし，現実には，プログラムのどこに，どんな形の表明を挿入すべきかが，平均的プログラマにはなかなか理解できないという，悲しい現実がある．実際には，ある程度の論理的トレーニングを受けたシステム設計者またはテスト担当者が，プログラムを第三者的見地から眺めて，表明の挿入・検査を行う以外にはないだろう．

5. おわりに

以上，本稿では，プログラム検査技法およびツールの現状について，概略的な見取図を描くことを試みた．ページ数の関係で具体的なツールの名前や，応用例を紹介することができなかったのは残念である．興味ある読者は，適当な参考文献（たとえば，W. E. Howden および E. F. Miller の共編になる IEEE の Tutorial Text: SoftwareTesting & Varidation Techniques には，代表的な関連論文が数多く集められている）を読まれることをおすすめする．

参考文献

1) Dijkstra, E. W.: Structured Programming, in Software Engineering Techniques (ed. J. N. Buxton and B. Randell), NATO Science Committee (1970).
2) Myers, G. J.: The Art of Software Testing, John Wiley & Sons, 1979 (松尾正信訳：ソフトウェア・テストの技法，近代科学社).
3) Miller, E. F.: Program Testing—Art meets Theory, Computer (July 1977).
4) Ramamoorthy, C. V. et al.: A Systematic Approach to the Development and Validation of Critical Software for Nuclear Plants, in Proceedings of 4th ICSE (1979).
5) Goodenough, J. and Gerhart, S. H.: Toward a Theory of Test Date Selection, IEEE Transactions on Software Engineering (June 1975).
6) Osterweil, L. J. and Fosdick, L. D.: DAVE-A Validation Error Detection and Documentation System for Fortran Programs, Software-Practice and Experience (6, 1976).
7) Stucki, L.: Automatic Generation of Self-Metric Software, in Proc. 1973 Symposium on Computer Software Reliability (April 1973).
8) Miller, E. F.: Program Testing Technology in the 1980', in Proc. Oregon Conference on Problems in Computing in the 1980'., IEEE Computer Society (April 1978).

(昭和57年2月26日受付)