第22章

デベロッパーテスト

c2e.com/2261

本章の内容

- 22.1 ソフトウェア品質におけるデベロッパーテストの役割
- 22.2 デベロッパーテストへの推奨アプローチ
- 22.3 テストの知恵袋
- 22.4 典型的なエラー
- 22.5 テストサポートツール
- 22.6 テストの改善
- 22.7 テスト記録の保管
- 22.8 参考資料
- 22.9 まとめ

関連トピックス

- ソフトウェアの品質：第20章
- コラボレーティブコンストラクション：第21章
- デバッグ：第23章
- 統合：第29章
- コンストラクションの準備：上巻第3章

「テスト」は品質向上のための最も一般的な作業である。多くの業界、学術研究機関、企業でテストが実践されている。テストの方法はさまざまで、一般的に開発者によって実行されるものもあれば、専門の検査機関によって実行されるものもある。

■ **単体テスト**

プログラマやプログラマのグループによって作成された単体のクラス、ルーチン、または小さなプログラムをテストする。全体的なシステムとは切り離した状態でテストを行う。

■ **コンポーネントテスト**

複数のプログラマまたはプログラミングチームが作業に関与したクラス、パッケージ、小さなプログラム、またはその他のプログラム要素をテストする。全体的なシステムとは切り離した状態でテストを行う。

■ **統合テスト(または結合テスト)**

複数のプログラマまたはプログラミングチームによって作成された2つ以上のクラス、パッケージ、コンポーネント、またはサブシステムを組み合わせてテストする。この種のテストは一般にテストするクラスが2つ以上になった段階で開始され、システム全体が完成するまで続けられる。

■ **回帰テスト**

既にテストにパスしているソフトウェアの欠陥を検出するために、以前に実行したテストケースを繰り返し実行する。

■ **システムテスト**

他のソフトウェアシステムやハードウェアシステムとの統合を含め、最終構成のソフトウェアをテストする。下位レベルの統合ではテストできないセキュリティ、パフォーマンス、リソースの無駄使い、タイミングの問題などをテストする。

本章でいう「テスト」は、開発者によるテスト(デベロッパーテスト)を指す。このテストは一般に、単体テスト、コンポーネントテスト、統合テストで構成されるが、回帰テストやシステムテストが含まれる場合もある。専門の検査機関では、ベータテスト、顧客による受け入れテスト、パフォーマンステスト、構成テスト、プラットフォームテスト、負荷テスト、ユーザビリティテストなど、その他にもさまざまな種類のテストが実施されるが、それらを開発者が実行することはほとんどない。本章では、この種のテストは取り上げない。

通常、テストはブラックボックステストとホワイトボックス(またはガラス張り)テストの2つに大きく分かれる。「ブラックボックステスト」は、テストする項目の内部のしくみをテスト担当者が確認できないテストである。当然ながら、自分で書いたコードのテストには使わない。「ホワイトボックステスト」は、テストする項目の内部のしくみをテスト担当者が知っているテストである。つまり、開発者が自分のコードをテストするときに使うものだ。ブラックボックステストとホワイトボックステストには、それぞれ長所と短所がある。

本章では、開発者が実施するテストであるホワイトボックステストを中心に見ていく。

プログラマによっては「テスト」と「デバッグ」を同じ意味で使用しているが、慎重なプログラマはそれらを区別している。テストはエラーを検出するための手段であり、デバッグは既に検出されているエラーの原因を特定し、修正するための手段である。本章では、エラーの検出のみを扱う。エラーの修正については、「第23章　デバッグ」で説明する。

一般的にいうテストは、コンストラクション時のテストよりもはるかに広いテーマである。システムテスト、負荷テスト、ブラックボックステストなどのテーマについては、章末の「22.8　参考資料」で取り上げる。

22.1 ソフトウェア品質におけるデベロッパーテストの役割

テストは、ソフトウェア品質保証プログラムにおいて重要な位置を占める。それどころか、多くの場合、ソフトウェア品質保証プログラムの実体はテストだけであると言ってもよい。しかし、さまざまなコラボレーティブ開発プラクティスの方がテストよりも高い割合でエラーを検出でき、しかも、エラー1件あたりの検出コストはテストの約半分であることを考えると、これは由々しき事態である(Card 1987)。個々のテスト(単体テスト、コンポーネントテスト、統合テスト)では、一般に存在するエラーの高々50%しか検出されない。これらのテストを組み合わせても、検出されるエラーは60%に満たない(Jones 1998)。

参照　レビューの詳細については、「第21章　コラボレーティブコンストラクション」を参照。

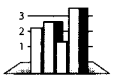
人は細菌に感染するが、プログラムをバグのあるプログラムの近くに置いても、プログラムがバグに感染することはない。プログラマがバグを持ち込んだに違いない。
— Harlan Mills

「セサミストリート」でソフトウェア開発のアクティビティをリストアップし、「この中で他と似ていないものはどれ？」とたずねたら、「テスト」という答えが返ってくるだろう。次のような理由から、テストはプログラマにとって、なかなか飲み込めないアクティビティである。

- テストの目標は、他の開発作業の目標とはまったく逆となる。目標はエラーを見つけることである。効果的なテストとは、ソフトウェアを壊すことだ。他の開発作業の目標はどれもエラーを防いでソフトウェアが壊れないようにすることだ。
- テストでは、エラーがないことは決して証明できない。徹底的なテストの結果、数千個のエラーが検出されたとしたら、それはすべてのエラーが検

出されたことを意味するのだろうか。それとも、見つけなければならないエラーがあと数千個残っているのだろうか。また、エラーがまったく見つからないということは、ひょっとしたらソフトウェアが完璧なのかもしれないが、テストケースが役に立たないか、あるいは不完全なのかもしれない。

- テスト自体はソフトウェアの品質を改善しない。テストの結果は品質の指標となるが、それだけのことであり、品質を改善するわけではない。テストの量を増やしてソフトウェアの品質を改善しようとするのは、痩せたくて体重計に乗る回数を増やすようなものだ。体重計に乗る前に何を食べたかによって体重が決まるように、どのソフトウェア開発テクニックを使ったかによってテストで検出されるエラーの数も決まる。痩せないのなら、新しい体重計を買うのではなく、ダイエット方法を変えなければならない。ソフトウェアを改善したいのなら、テストを増やすのではなく、開発作業を改善しなければならない。



HARD DATA

- テストでは、自分のコードからエラーが発見されることを前提にしなければならない。そのような前提に立たないと言うのであれば、自分でそう言うくらいだから、エラーは発見されないだろう。プログラムにエラーがないものとして実行すると、見つけていたはずのエラーを簡単に見逃してしまう。今では規範とされているGlenford Myersの研究では、何人かの経験豊富なプログラマに15個の欠陥があることがわかっているプログラムをテストさせたところ、平均的なプログラマが検出したのは、そのうちのわずか5個だった。最も成績の良かったプログラマでも9個という結果だった。エラーが検出されなかった主な原因は、エラーのある出力を入念に調べていないことだった。エラーは見えていたのに、プログラマはそれらに気付かなかったのだ(Myers 1978)。
  自分のコードからエラーが見つかりますようにと願うようでなければならない。そう願うのは不自然な行為に思えるかもしれないが、他のだれかにエラーを見つけられるくらいなら、自分で見つける方がよいだろう。

最大の問題は、一般的なプロジェクトでデベロッパーテストにどれくらい時間をかけるかである。基準とされている数字は、すべてのテストでプロジェクト全体の50%だが、これは誤解されやすい。まず、この数字はテストとデバッグを合わせたものである。テストだけであれば、これほど時間はかからない。次に、この数字は一般的な所要時間を表すもので、理想的な時間を表すものではない。そして、この数字には、デベロッパーテストだけでなく独立テストも含まれている。

図22-1に示すように、プロジェクトの規模や複雑さによって、デベロッパーテストにはプロジェクト全体の8～25%の時間が割かれるべきだろう。これは報告されているデータの多くと一致するものだ。

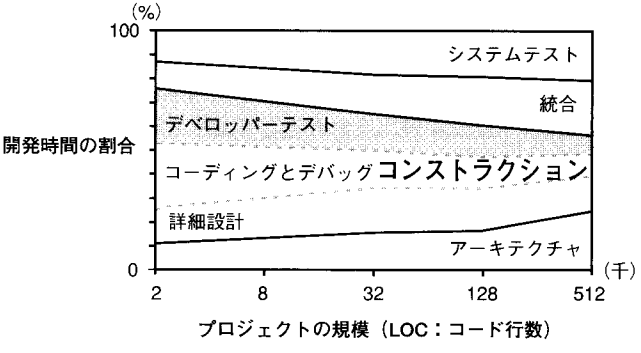


図22-1 プロジェクトの規模が大きくなればなるほど、デベロッパーテストが開発時間全体に占める割合は少なくなる。プログラムのサイズによる影響については、「第27章 プログラムサイズが及ぼす影響」を参照

もう1つの問題は、デベロッパーテストの結果をどうすればよいかである。デベロッパーテストの結果からは、即座に開発中の製品の信頼性を判断できる。テストで検出された欠陥を修正しないとしても、テストはソフトウェアの品質を物語る。デベロッパーテストの結果は、ソフトウェアを修正するためのガイドとして利用することもできる。そして、通常はそのような使い方をする。最後に、テストで検出された欠陥を記録していくと、頻度の高いエラーの種類が次第に明らかになる。この情報を基に、適切なトレーニングコースを選択したり、今後のテクニカルレビューの作業項目を決めたり、テストケースを設計したりすることができる。

22.1.1 コンストラクション時のテスト

広いテストの世界では、本章で取り上げるような「ホワイトボックス」、つまり「ガラス張り」テストは無視されることがある。一般に、クラスはブラックボックスとして設計されるから、そのクラスがどのようなものか知るために、クラスのユーザーがインターフェイスごしに覗き込んだりしない。ただし、クラスをテストする際には、クラスをガラス張りであると見なし、クラス内部のソースコードや入出力を調べることができると都合がよい。クラスの内部がどのようなものかわかれば、クラスをより徹底的にテストすることができる。もちろん、クラスを書いているときに見えていない部分は、テストでも見えない。このため、ブラックボックステストにも利点がある。

コンストラクションの際には、一般にルーチンやクラスを作成し、それを頭の中でチェックしてから、レビューやテストにかける。統合テストやシステムテストをどのような方法で行うとしても、クラスやルーチンを他のコンポーネントと組み合わせる前に、単体のままで徹底的にテストすべきである。複数のルーチンを書いている場合は、それらを1つずつテストする。ルーチンを単体でテストするのは容易なことではないが、デバッグするのは単体の方がずっと簡単である。テストが済んでいない複数のルーチンを組み合わせてテストした場合、エラーが検出されたとしても、どのルーチンが犯人なのかわからなくなってしまう。テスト済みのルーチンに新しいルーチンを1つずつ追加すれば、新たに検出されたエラーの原因が新しいルーチンにあるのか、または新しいルーチンとのやり取りにあるのかがわかる。これならデバッグは簡単だ。

コラボレーティブコンストラクションのプラクティスには、テストにはない長所がいろいろある。テストの問題の1つは、テストが正しく実行されないために、その実力をうまく引き出せない場合が多いことだ。開発者は数百個ものテストを実行するが、それでもコードを部分的にしかテストできない。テストカバレッジが十分であると「実感」しても、それは、実際のテストカバレッジが十分であることは意味しない。テストの基本概念を理解すれば、適切なテストを実施し、テストの効果を引き出すのに役立つだろう。

22.2 デベロッパーテストへの推奨アプローチ

デベロッパーテストに体系的に取り組めば、わずかな作業であらゆる種類のエラーを検出することができる。テストには、次の基本項目を必ず盛り込むこと。

- 関連する要求をそれぞれテストして、それらの要求が実装されていることを確認する。この段階のテストケースは、要求の策定時に計画するか、あるいはテストする項目を作成し始める前のなるべく早い段階に計画する。要求で見落とされがちなものをテストすることも検討する。セキュリティレベル、記憶域、インストール手順、システムの信頼性はすべて、テストの対象にふさわしいが、要求の策定時には見落とされがちだ。
- 関連する設計項目をそれぞれテストして、設計が実装されていることを確認する。この段階のテストケースは、設計段階で計画するか、あるいはテストするルーチンやクラスの詳細なコーディングを始める前のなるべく早い段階に計画する。
- 「基礎テスト」を使って、要求や設計をテストするテストケースに、詳細なテストケースを追加する。データフローのテストを追加した後、コード

を徹底的に検査するのに必要な残りのテストケースを追加する。少なくとも、コードのすべての行をテストすること。基礎テストとデータフローテストについては、この後で説明する。

- プロジェクトで既に判明しているエラーと、過去のプロジェクトで検出されたエラーのチェックリストを使用する。

製品に沿ったテストケースを設計すれば、コーディングのエラーよりも高くつきやすい要求や設計のエラーを防ぐのに役立つ。欠陥は早く見つければ見つけるほど安価なので、欠陥をなるべく早くテストして検出する計画を立てよう。

22.2.1 テストは先か、それとも後か

開発者は、コードを書いた後にテストケースを作成した方がよいのか、それともコードを書く前に作成した方がよいのかで悩むことがある(Beck 2003)。欠陥のコストの上昇を示すグラフ(上巻第3章の図3-1)は、テストケースを先に書くと、エラーがコードに紛れ込んでから検出および削除されるまでの時間が短くなることを示している。これは、テストケースを先に書いた方がよい理由の1つである。

- コードを書く前にテストケースを書いた方が、コードを書いた後にテストケースを書くよりも手間がかからない。コードを書く作業は、テストケースを書く作業をそのまま繰り返すようなものだからだ。
- テストケースを先に書いた方が、エラーが早く検出されるので、それらを修正するのが簡単である。
- テストケースを先に書けば、少なくとも、コードを書く前に要求と設計について少し考えることになる。これにより、コードが良くなることが多い。
- テストケースを先に書くと、コードを書く前に、要求の問題点が明らかになる。これは、要求が明確でないとテストケースを書くのが難しいためだ。
- テストケースを保存しておけば(ぜひ、そうしよう)、テストを後から行うこともできる。

本書では、「テストファーストプログラミング」を、過去10年間に登場したソフトウェアプラクティスの中で最も効果的なものの1つであると同時に、一般的なアプローチとしても優れたものであると考えている。しかし、デベロッパーテストの一般的な制限を受けるため、テストの万能薬ではない。次は、これについて見ていこう。

22.2.2 デベロッパーテストの限界

デベロッパーテストには、次のような限界がある。

デベロッパーテストは「クリーンテスト」になりがちである

開発者は、あらゆる角度からコードの不備をテストする「ダーティテスト」ではなく、コードが動くかどうかをテストする「クリーンテスト」を実施する傾向にある。テストに慣れていない組織では、クリーンテストとダーティテストの割合がだいたい5対1になる。テストに慣れている組織では、クリーンテストとダーティテストの割合がだいたい1対5になる。この割合は、クリーンテストの数を減らしても逆転しない。つまり、ダーティテストの数に25倍の差が出る(Johnson 1994)。

デベロッパーテストはテストカバレッジを楽観的に見る傾向にある

平均的なプログラマは、自分はテストカバレッジ95%を達成したと考えるが、実際に達成しているのは、せいぜい80%、最悪の場合は30%、平均的なケースで50～60%である(Johnson 1994)。

デベロッパーテストは高度なテストカバレッジを避ける傾向にある

ほとんどの開発者は、「全ステートメントカバレッジ」と呼ばれるテストカバレッジで十分であると考えている。出発点としては悪くないが、とても十分であるとは言えない。カバレッジの標準としては、「全分岐カバレッジ」と呼ばれるものを満たしている方がよい。つまり、すべての分岐を少なくとも1つのtrue値と1つのfalse値で評価するのである。これを達成する方法については、「22.3　テストの知恵袋」で説明する。

これらの傾向は、デベロッパーテストの価値を下げるものではなく、デベロッパーテストを正しく捉えるのに役立つポイントである。デベロッパーテストは価値のあるものであると同時に、それだけでは適正な品質保証を実現するのに十分でない。このため、独立テストやコラボレーティブコンストラクションなど、他のプラクティスで補う必要がある。

22.3 テストの知恵袋

プログラムをテストしてもプログラムが正しいことを証明できないのはなぜだろうか。テストを使ってプログラムが動作することを証明するには、考えられる限りの入力値と、考えられる限りの入力値の組み合わせによって、プログラムをテストしなければならない。たとえ簡単なプログラムであっても、その作業は実現不可能な規模になるだろう。たとえば、名前、住所、電話番号を受け取って、それらをファイルに保存するプログラムがあるとしよう。これは

どこから見ても単純なプログラムである。いつもあなたが正確さに頭を悩ませているプログラムに比べたら、はるかに単純だ。名前と住所は最大で20文字、使用できる文字は26種類であるとした場合、入力は何通り考えられるだろうか。

名前	26^{20}（20文字、文字は26種類）
住所	26^{20}（20文字、文字は26種類）
電話番号	10^{10}（10桁、数字は10種類）
入力候補の総数	$= 26^{20} \times 26^{20} \times 10^{10} = 10^{66}$

このように入力の数が比較的少ない場合でも、テストケースは1の後に0が66個もつくような数になる。これから計算すると、ノアを箱舟から引っ張り出して、このプログラムを1秒につきテストケース1兆個の速さでテストさせたとしても、今日までにテストできるのは全体の1%にも満たないことになる。現実的な量のデータを用意したとしても、すべての可能性を徹底的にテストすることがとうてい不可能であることは、明白である。

22.3.1 不完全なテスト

完全なテストは不可能なので、率直に言えば、テストのコツはエラーを検出しやすいテストケースを選ぶことである。10^{66}通りのテストケースのうち、エラーを明らかにする可能性があるものはほんのわずかである。同じ結果を繰り返すようなテストケースではなく、何か違うことを教えてくれるテストケースに的を絞って、選択する必要がある。

参照 すべてのコードをカバーしたかどうかを確認する方法の1つは、カバレッジモニタを使用することである。詳細については、「22.5.4 カバレッジモニタ」を参照。

テストの計画を立てる際には、新しい発見が得られないものは排除する。なぜなら、同様のデータでエラーが検出されることがなければ、新しいデータでテストしても、おそらくエラーは検出されないからだ。基本的な項目を効率よくカバーするためのさまざま方法が提案されているが、ここではそのうちのいくつかを見てみよう。

22.3.2 構造化された基礎テスト

構造化された基礎テストとは堅苦しい名前だが、いたって単純な概念である。これは、プログラムのすべてのステートメントを最低でも1回はテストしなければならない、というものだ。ステートメントが論理文(if、whileなど)である場合、それらを完全にテストするためには、if文やwhile文に含まれている式の複雑さに応じてテストを変更する必要がある。基本項目をすべてカバーする最も簡単な方法は、プログラムのパスの数を計算して、プログラムのすべてのパスを通る必要最小限のテストケースを作成することだ。

「コードカバレッジテスト」や「ロジックカバレッジテスト」という言葉を聞いたことがあるだろうか。これらはプログラムのすべてのパスをテストするための手法である。これらはすべてのパスをカバーするので、構造化された基礎テストに似ているが、**必要最小限の**テストケースですべてのパスをカバーするという発想はない。コードカバレッジテストやロジックカバレッジテストを使用する場合は、構造化された基礎テストで同じロジックをカバーする場合よりも、多くのテストケースを作成することになるだろう。

基礎テストに最低限必要なテストケースの数を計算する方法は、次のように簡単だ。

1. ルーチンの直線パスを1とする。
2. 次のキーワードまたはそれに相当するものを検出するたびに1を加える。

   if, while, repeat, for, and, or
3. case文のcaseごとに1を加える。case文にdefault句が含まれていない場合は、さらに1を加える。

この手順は、上巻第19章19.6.2の「複雑さを定量化する方法」の項で、複雑さを測定したときのものと同様である。

例を見てみよう。

リスト22-1 Javaプログラムのパスの数を計算する

ルーチン自体を「1」と数える

ifで「2」と数える

```
Statement1;
Statement2;
if ( x < 10 ) {
  Statement3;
}
Statement4;
```

この場合は、最初に1と数えて、ifにぶつかったところで2と数える。したがって、プログラムのすべてのパスをカバーするには、少なくともテストケースが2つ必要である。この例では、次のテストケースを作成する必要があるだろう。

- if文の内側(Statement3)を実行する(x < 10)。
- if文の内側を実行しない(x >= 10)。

このようなテストのしくみをきちんと理解できるように、このコードをもう少し現実的なものにしてみよう。現実的という意味には、エラーが紛れ込んだコードも含まれる。

図22-2 バグ付きのコードに注意

次に示すのは、もう少し複雑な例である。リスト22-2のコードはこれ以降の説明に使用するもので、エラーがいくつか含まれている可能性がある。

リスト22-2 Javaプログラムの基礎テストに必要なテストケースの数を計算する

ルーチン自体を「1」と数える

forで「2」と数える

ifで「3」と数える

```
1  // 手取り給与の計算
2  totalWithholdings = 0;
3
4  for ( id = 0; id < numEmployees; id++ ) {
5
6    // 社会保険料の源泉徴収額の計算（最高額を下回った場合）
7    if ( m_employee[ id ].governmentRetirementWithheld <
         MAX_GOVT_RETIREMENT ) {
8      governmentRetirement =
         ComputeGovernmentRetirement( m_employee[ id ] );
9    }
10
11   // 既定を老齢年金負担額なしに設定
12   companyRetirement = 0;
```

（続く）

(続き)

```
13
14  // 裁量可能老齢年金負担額の決定
15  if ( m_employee[ id ].WantsRetirement &&
16    EligibleForRetirement( m_employee[ id ] ) ) {
17    companyRetirement = GetRetirement( m_employee[ id ] );
18  }
19
20  grossPay = ComputeGrossPay ( m_employee[ id ] );
21
22  // 個人退職年金負担額の決定
23  personalRetirement = 0;
24  if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
25    personalRetirement =
        PersonalRetirementContribution( m_employee[ id ],
26      companyRetirement, grossPay );
27  }
28
29  // 週給の計算
30  withholding = ComputeWithholding( m_employee[ id ] );
31  netPay = grossPay - withholding - companyRetirement -
32    governmentRetirement - personalRetirement;
33  PayEmployee( m_employee[ id ], netPay );
34
35  // この社員の週給を会計に加算
36  totalWithholdings = totalWithholdings + withholding;
37  totalGovernmentRetirement =
      totalGovernmentRetirement + governmentRetirement;
38  totalRetirement = totalRetirement + companyRetirement;
39 }
40
41 SavePayRecords( totalWithholdings, totalGovernmentRetirement,
     totalRetirement );
```

15行目: ifで「4」と数え、&&で「5」と数える

24行目: ifで「6」と数える

この例では、最初に1つ、5つのキーワードにつき1つずつ、合計6つのテストケースが必要である。だが、6つのテストケースですべての基本項目がカバーされると考えてはならない。これは、最低限6つのテストケースが必要であるという意味である。テストケースを慎重に作成しなければ、すべての基本項目をカバーすることはほとんど不可能である。ポイントは、必要なテストケースの数を数えるときに、同じキーワードに注意することだ。コードの各キーワードは、trueまたはfalseと評価される何かを表す。trueと評価されるものにつき少なくとも1つのテストケースと、falseと評価されるものにつき少なくとも1つのテストケースを作成することに注意しよう。

この例においてすべての基本項目をカバーするテストケースは、表22-1のようになる。

表22-1 リスト22-2の6つのテストケース

テストケース	テストの内容	テストデータ
1	基本ケース	すべての論理判定がtrue
2	forの最初の条件がfalse	numEmployees < 1
3	1つ目のifがfalse	m_employee[id].governmentRetirementWithheld >=MAX_GOVT_RETIREMENT
4	2つ目のifがfalse（&&の前がfalse）	not m_employee[id].WantsRetirement
5	2つ目のifがfalse（&&の後がfalse）	not EligibleForRetirement(m_employee[id])
6	3つ目のifがfalse	not EligibleForPersonalRetirement(m_employee[id])

注：この表は、これ以降のテストケースを追加して拡張していく。

ルーチンがこれよりも複雑であったとしたら、すべてのパスをカバーするためのテストケースの数は、あっという間に増えてしまうだろう。ルーチンが短ければ短いほど、テストするパスの数は少なくなる。ANDやORの数が少ない論理式は、テストしなければならないバリエーションが少ない。テストが簡単になることも、ルーチンを短く保ち、論理式を単純に保つ動機の1つとなる。

さて、このルーチンのテストケースが6つ作成され、構造化された基礎テストの要件が満たされたが、これでルーチンが完全にテストされると確信できるだろうか。きっとそうではないだろう。この種のテストは、すべてのコードが実行されることを保証するだけで、データのバリエーションまでは計算に入れていない。

22.3.3 データフローテスト

前節と本節は、コンピュータプログラミングにおいて、制御フローとデータフローが同じく重要であることを示す格好の例である。

データフローテストは、データの使用にも、少なくとも制御フローと同じくらいミスがあるという考えに基づいている。Boris Beizerによれば、コードの半分以上は、データの宣言と初期化で占められている(Beizer 1990)。

データは次のいずれかの状態でプログラム中に存在する。

■ 定義された

データは初期化されているが、まだ使われていない。

■ 使用された
データは、ルーチンへの引数などとして、計算に使われている。

■ 破棄された
データは一度定義されたが、何らかの理由で未定義になっている。たとえば、データがポインタである場合は、おそらくポインタが解放されている。データがforループのインデックスである場合は、おそらくプログラムの制御がループを抜け、そのプログラミング言語ではループを抜けるとインデックスの値が定義されない。ファイルのレコードへのポインタである場合は、おそらくファイルが閉じられ、レコードポインタが無効になっている。

そうすると、「定義された」、「使用された」、「破棄された」という用語に加えて、変数に何かをする直前または直後に、ルーチンに入る、またはルーチンから抜けることを意味する用語があると便利である。

■ 入った
制御フローが変数に作用するルーチンに入っている。たとえば、ルーチンの先頭でその変数が初期化されるなど。

■ 抜けた
制御フローが変数に作用するルーチンから抜けている。たとえば、ルーチンの最後で状態変数に戻り値が代入されたなど。

データの状態の組み合わせ

通常のデータの状態は、変数が「定義された」、「1回以上使用された」、「破棄された」という順番で変化する。次のような順番の組み合わせパターンはあやしいので注意しよう。

■ 定義された — 定義された
変数に値を代入する前に変数を2回定義しなければならないとしたら、プログラムが悪いのではなくて、コンピュータに問題がある。実害がないとしても、無駄だし、エラーのもとである。

■ 定義された — 抜けた
変数がローカル変数であるとしたら、定義しただけで使わずにルーチンを抜けるのは意味がない。それがルーチンの引数やグローバル変数であるとしたら、問題はないだろう。

■ **定義された — 破棄された**
変数を定義した後で使わずに破棄したら、その変数は無駄なものであるか、その変数を使うはずのコードが抜けていることを意味する。

■ **入った — 破棄された**
変数がローカル変数であるとしたら問題である。定義されても使われてもいない変数を破棄する必要はないはずだ。一方、それがルーチンの引数やグローバル変数であるとしたら、変数が破棄される前にどこかで定義されている限り、このパターンに問題はない。

■ **入った — 使用された**
この場合も、変数がローカル変数であるとしたら問題である。変数を使用するには、まず定義する必要がある。一方、それがルーチンの引数やグローバル変数であるとしたら、変数が使われる前にどこかで定義されている限り、このパターンに問題はない。

■ **破棄された — 破棄された**
変数を2回破棄する必要はないはずだ。変数は生き返ったりしない。変数が生き返ったとしたら、プログラミングに問題がある証拠だ。また、2回の破棄はポインタにとって致命的である。マシンをハングアップさせる手っ取り早い方法は、ポインタを2回破棄する(解放する)ことだ。

■ **破棄された — 使用された**
破棄された変数を使用するのは、ロジックのエラーである。コードが動いているように見えても(たとえば、ポインタが既に解放されたメモリを依然としてポイントしているなど)、それは偶然である。マーフィーの法則によれば、最も大きな被害をもたらす場面でプログラムが動作しなくなるだろう。

■ **使用された — 定義された**
変数を使用してから定義しているとしたら、問題かもしれないし、そうでないかもしれない。これは、その変数が使われる前に定義されているかどうかによる。いずれにしても、このパターンを目にしたら、事前に定義されているかどうかを確認した方がよい。

テストを始める前に、データの状態がこのようなあやしい順番になっていないかどうか確認しよう。順番のあやしいものを確認したら、データフローテストケースを作成するポイントは、すべての「定義された — 使用された」パスを調べることだ。これは次のように何段階かのレベルで行うことができる。

■ すべての「定義された」

すべての変数のすべての定義をテストする。つまり、変数に値が代入されるすべての場所をテストする。すべてのコード行を調べるとしたら、これは既定で行うはずなので、戦略としては弱い。

■ すべての「定義された — 使用された」

「変数がある場所で定義され、別の場所で使われる」という組み合わせをすべてテストする。すべてのコード行をテストしただけでは、すべての「定義された — 使用された」の組み合わせがテストされる保証はないので、これはすべての「定義された」をテストするよりも強力な戦略である。

例を見てみよう。

リスト22-3 Javaプログラムのデータフローをテストする

```
if ( Condition 1 ) {
  x = a;
}
else {
  x = b;
}
if ( Condition 2 ) {
  y = x + 1;
}
else {
  y = x - 1;
}
```

このプログラムのすべてのパスをカバーするには、Condition 1がtrueの場合のテストケースと、Condition 1がfalseの場合のテストケースが必要である。また、Condition 2がtrueの場合のテストケースと、Condition 2がfalseの場合のテストケースも必要である。これは、テストケース1(Condition 1=true、Condition 2=true)とテストケース2(Condition 1=false、Condition 2=false)の2つのテストケースで処理することができる。この2つのテストケースで、構造化された基礎テストに必要なものはすべて揃う。この2つで、変数を定義するすべてのコード行をテストすることもできる。つまり、弱いデータフローテストを自動的に実行することになる。

ただし、「定義された — 使用された」の組み合わせをすべてカバーするには、テストケースをもう少し追加する必要がある。目下のところ、「Condition 1=true、Condition 2=true」のテストケースと、「Condition 1=false、Condition 2=false」のテストケースの2つがある。つまり、

```
x = a
...
y = x + 1
```

および

```
x = b
...
y = x - 1
```

の場合である。しかし、「定義された — 使用された」の組み合わせをすべてテストするには、テストケースがあと2つ必要だ。すなわち、「x = aとy = x - 1」および「x = bとy = x + 1」である。この例では、テストケース3(Condition 1 =true、Condition 2=false)とテストケース4(Condition 1=false、Condition 2 =true)の2つを追加すればよい。

テストケースを開発する良い方法は、必ずしもすべての「定義された — 使用された」データフローがカバーされるとは限らないが、構造化された基礎テストから始めることである。そして、「定義された — 使用された」データフローをチェックするのに必要なテストケースを追加して、テストケースを完成させればよい。

前述のように、リスト22-2のルーチンの構造化された基礎テストは、6つのテストケースから成る。「定義された — 使用された」データフローをすべてテストするには、テストケースがさらにいくつか必要である。これには、既存のテストケースでカバーされるものと、そうでないものがある。表22-2に、構造化された基礎テストによって生成されるもの以外に追加されるテストケースをまとめた。

表22-2 リスト22-2で追加されるテストケース

テストケース	テストの内容
7	12行目でcompanyRetirementが定義され、26行目で初めて使われるケース。これは既存のテストケースでカバーされるとは限らない
8	15行目でcompanyRetirementが定義され、31行目で初めて使われるケース。これは既存のテストケースでカバーされるとは限らない
9	17行目でcompanyRetirementが定義され、31行目で初めて使われるケース。これは既存のテストケースでカバーされるとは限らない

データフローのテストケースを何度かリストアップしていると、どのテストケースが新しいもので、どのテストケースが既にカバーされているものか、ピンとくるようになる。行き詰まったら、「定義された — 使用された」の組み合わせをすべてリストアップしてみよう。たいへんな作業に思えるかもしれない

が、基礎テストではテストされずに残っていたテストケースが必ず見つかるだろう。

22.3.4 同値分割

良いテストケースは、入力データの大部分をカバーする。2つのテストケースがまったく同じエラーを発生させるとしたら、どちらか一方でよいことになる。「同値分割」という概念は、この考えを定式化したもので、必要なテストケースの数を減らすのに役立つ。

参照 同値分割については、章末の「22.8 参考資料」で紹介する書籍で詳しく解説されている。

リスト22-2の7行目は、同値分割を適用するのにうってつけだ。テストする条件は、

```
m_employee[ ID ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT
```

である。この条件文には、同値のケースが2つある。m_employee[ID].governmentRetirementWithheldがMAX_GOVT_RETIREMENT未満のケースと、MAX_GOVT_RETIREMENT以上のケースだ。プログラムの他の部分にも、このような同値のケースがあるかもしれない。その場合、m_employee[ID].governmentRetirementWithheldでテストしなければならない値が2つであるとは限らないが、プログラムのこの部分だけに関して言えば、この2つをテストすればよい。

同値分割について考えてみても、プログラムを既に基礎テストとデータフローテストでカバーしていたとしたら、新しいテストケースはそれほど思い付かないだろう。ただし、プログラムを外側から(ソースコードではなく仕様から)調べている場合や、データが複雑で、その複雑さが必ずしもプログラムのロジックに反映されていない場合には、同値分割について考えてみるとよいだろう。

22.3.5 エラーの推測

優秀なプログラマは、公式のテスト手法だけでなく、それほど公式でないヒューリスティックな手法を使って、コードのエラーを暴き出す。ヒューリスティックな手法の1つは、エラーを推測することである。「エラーを推測する」という表現は、この繊細な概念に似つかわしくない呼び名だ。これは、プログラムにエラーが紛れ込んでいる場所を推測し、それを基にテストケースを作成することを意味するのだが、それにはそれなりの素養が必要であるからだ。

参照 ヒューリスティックについては、上巻第2章の「2.2 ソフトウェアメタファの使用法」を参照。

推測の基になるのは、直感や過去の経験である。「第21章 コラボレーティブコンストラクション」では、インスペクションの利点の1つは一般的なエラーのリストを生成し、維持することであると指摘した。このリストを使って、新しいコードをチェックする。過去に発生したエラーの種類を記録しておけば、「エラーの推測」によってエラーを発見できる可能性が高くなる。

ここでは、それ自体がエラーの推測に役立つという特殊なエラーについて説明しよう。

22.3.6 境界分析

境界条件、いわゆる「1つ違い(off-by-one)エラー」は、テストがその威力を発揮する領域の1つである。たとえば、numとすべきところをnum-1にしていたり、>を使うべきところに>=を使っていたりするのは、よくある誤りである。

境界分析とは、境界条件を調べるテストケースを作成することだ。図22-3を見てみよう。maxよりも小さい値の範囲をテストするとしたら、3つの条件が考えられる。

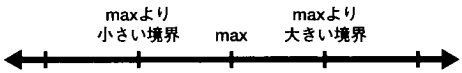


図22-3 3つの境界条件

この図が示すように、「maxより小さい」、「maxそのもの」、「maxより大きい」という3つの境界条件がある。テストケースを3つ用意すれば、一般的なミスをすべて防ぐことができる。

リスト22-2のコードには、次の条件文が含まれている。

```
m_employee[ ID ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT
```

境界分析の原則に従えば、表22-3の3つのケースを調べればよいことになる。

表22-3 リスト22-2での境界分析

テストケース	テストの内容
1	テストケース1は、上記の条件文がtrueの場合が、境界において最初のtrueとなる場合である。したがって、m_employee[ID].governmentRetirementWithheldをMAX_GOVT_RETIREMENT-1に設定する。このテストケースは既に生成されている

表22-3 リスト22-2での境界分析(続き)

テストケース	テストの内容
3	テストケース3は、上記の条件文がfalseとなる場合なので、境界のfalseの側である。したがって、m_employee[ID].governmentRetirementWithheldをMAX_GOVT_RETIREMENT+1に設定する。このテストケースは既に生成されている
10	m_employee[ID].governmentRetirementWithheld = MAX_GOVT_RETIREMENTのケースをテストする新しいテストケースを追加する

複合境界

境界分析は、値の範囲に最小値と最大値がある場合にも適用される。この例では、grossPay、companyRetirement、またはPersonalRetirementContributionにも最小値または最大値があるが、これらの値の計算は他のルーチンの範囲内なので、テストケースの説明は割愛する。

境界に変数の組み合わせが絡んでくると、境界条件は少しややこしくなる。たとえば、2つの変数の値を掛け算した場合、両方の値が大きな正の整数だったらどうなるだろうか。大きな負の値だったり、0だったりした場合はどうなるだろうか。ルーチンに渡されるすべての文字列が極端に長かったらどうなるだろうか。

リスト22-2の例では、多数の従業員が高い給与(それぞれ25万ドルとか。夢のようだね)を得ていた場合に、変数totalWithholdings、totalGovernmentRetirement、totalRetirementの値がどうなるか確認した方がよいだろう。これにより、テストケースがもう1つ必要になる。

表22-4 リスト22-2の複合境界のテストケース

テストケース	テストの内容
11	大勢の社員が高い給与を得ている例として(「大勢」や「高い」がどれくらいかは、開発するシステムによる)、1,000人の従業員にそれぞれ25万ドルの給与が支払われていて、だれも社会保険料の源泉徴収を受けておらず、全員が老齢年金の源泉徴収を望んでいるとする

考え方は同じだが、鏡の反対側にあるテストケースは、給与が0ドルの少数の従業員である。

表22-5 リスト22-2の複合境界のテストケース

テストケース	テストの内容
12	それぞれ給与が0ドルの10名の従業員

22.3.7 悪いデータ

境界条件付近で発生するエラーを推測するほかに、データが不正なケースをいくつか推測することができる。悪いデータのテストケースとしては、次が挙げられる。

- データが小さすぎる(またはデータがない)
- データが大きすぎる
- データの種類が正しくない
- データのサイズが正しくない
- データが初期化されていない

これらに基づいて推測されるテストケースの中には、既にカバーされているものがある。たとえば、「データが小さすぎる」はテストケース2とテストケース12でカバーされているし、「データのサイズが正しくない」のテストケースはなかなか思い浮かばないだろう。いずれにしても、「悪いデータ」からテストケースがさらにいくつか追加される。

表22-6 リスト22-2の悪いデータのテストケース

テストケース	テストの内容
13	社員1億人分の配列。大きすぎるデータをテストする。もちろん、「大きすぎる」とはどれくらいなのかはシステムによって異なるが、ここでは例として、この値がきわめて大きいと仮定する
14	給与が負数。データの種類が正しくないことをテストする
15	社員の数が負数。データの種類が正しくないことをテストする

22.3.8 良いデータ

プログラムのエラーを見つけ出そうとしていると、代表的なケースにもエラーが潜んでいる可能性があることを見逃しがちだ。基礎テストで取り上げられる代表的なケースは、通常、1種類の良いデータを表す。良いデータは他にもあるので、チェックした方がよいだろう。テストする項目によっては、次の種類のデータをチェックすると、エラーが見つかることがある。

- 代表的なケース — 期待される値の中間あたりの値
- 正常構成の最小値

■ 正常構成の最大値

■ 古いデータとの互換性

正常構成の最小値は、1つの項目をテストするだけでなく、一連の項目をテストするのにも役立つ。これは境界条件の最小値に似ているようだが、期待される正常値の集合から最小値の集合を作成するという点で異なる。1つの例は、スプレッドシートをテストするときに、空のスプレッドシートを保存することだ。ワードプロセッサをテストする場合は、空のドキュメントを保存する。この例では、正常構成の最小値をテストするとしたら、次のテストケースが追加される。

表22-7 リスト22-2の最小値のテストケース

テストケース	テストの内容
16	従業員1人のグループ。正常構成の最小値をテストする

正常構成の最大値は、正常構成の最小値の逆である。境界条件のテストという点では同じだが、この場合も、期待される正常値の集合から最大値の集合を作成する。たとえば、製品のパッケージに書かれている「スプレッドシートの最大サイズ」と同じ大きさのスプレッドシートを保存したり、ワードプロセッサで推奨される最大サイズのドキュメントを保存したりすることが挙げられる。この例の場合、正常構成の最大値は正規の従業員数の最大値によって決まる。それが500人であるとすれば、次のテストケースが追加されることになる。

表22-8 リスト22-2の最大値のテストケース

テストケース	テストの内容
17	従業員500人のグループ。正常構成の最大値をテストする

正常データの最後のテストは、古いデータとの互換性をテストすることである。このテストは、プログラムやルーチンが古いプログラムやルーチンの置き換えである場合に必要となる。新しいルーチンは、古いルーチンに欠陥がないとすれば、古いルーチンに古いデータを与えたときと同じ結果を生成する必要がある。こうしたバージョン間の連続性は、回帰テストの基本である。回帰テストの目的は、修正や拡張によって品質が以前の水準を下回っていないかどうかを確認することだ。この例では、互換性という尺度から追加されるテストケースはない。

22.3.9 手元でチェックしやすいテストケースの使用

基本給のテストケースを作成しているとしよう。基本給のデータが必要だが、それは自分で入力する。

```
1239078382346
```

おっと、1兆ドルを超えてしまった。金額を下げて、もう少し現実的な数字にしよう。

```
$90,783.82
```

さて、テストケースを実行したところ、エラーが発見されたとしよう。エラーが発見されたことはどのようにして知るのだろうか。それはきっと、自分で正しい金額を計算していて、答えを知っているからわかるのである。ただし、$90,783.82といった変則的な数値を手で計算しようとすると、プログラムで発見されるエラーと同じようなミスを犯しやすいものだ。$20,000といったすっきりした数値なら計算しやすい。0を電卓に打ち込むのは簡単だし、2倍の計算ならほとんどのプログラマが暗算できるだろう。

$90,783.82といった変則的な値の方がエラーを発見しやすいように見えるかもしれないが、その点に関して、同値分割で同じケースに含まれている数字に大差はない。

22.4 典型的なエラー

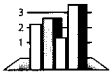
ここでは、プログラマの敵であるエラーをできるだけよく知ることが、テストを最も実り多いものにする、という命題について見ていこう。

22.4.1 最もエラーを含んでいるクラスとは

エラーはソースコードにまんべんなくばらまかれていると考えるのが自然である。1,000行のコードにつき平均で10個の欠陥があるとすれば、100行のコードから成るクラスに1個の割合でエラーが含まれていると考えるだろう。そう考えるのは自然だが、間違っている。

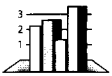
Capers Jonesの報告では、IBMの品質改善プログラムによって、IMSシステムの425個のクラスのうち31個が、エラーが起きやすいクラスであることが判明した。この31個のクラスは、修正されたか、または初めから書き直された。それから1年もたたないうちに、顧客から報告されるIMSシステムのエラーは10分の1に減った。システム保守の総コストは45%削減され、顧客の満足度は、「容認できない」から「良い」へと改善された(Jones 2000)。

ほとんどのエラーは、エラーが起きやすいいくつかのルーチンに集中する。エラーとコードの間には、一般に次のような関係がある。


HARD DATA

- プロジェクトのエラーの80%は、プロジェクトのクラスまたはルーチンの20%で見つかる(Endres 1975; Gremillion 1984; Boehm 1987b; Shull et al. 2002)。
- プロジェクトのエラーの50%は、プロジェクトのクラスの5%で見つかる(Jones 2000)。

これらの関係は、いくつかの必然的な結果を知らなければ、それほど重要には思えないかもしれない。1つ目は、開発費の80%がプロジェクトのルーチンの20%で消費されることだ(Boehm 1987b)。最もコストの高い20%のルーチンが最もエラーの多い20%のルーチンと一致するとは限らないが、実に暗示的である。


HARD DATA

2つ目は、エラーの多いルーチンがコストのどれくらいの割合を占めるかにかかわらず、エラーの多いルーチンがきわめて高価であることだ。1960年代のIBMによるOS/360オペレーティングシステムの分析結果では、エラーはすべてのルーチンにまんべんなくばらまかれていたのではなく、一部のルーチンに集中していたことがわかった。そして、このようなエラーの起きやすいルーチンは、「プログラムにおいて最も高価な部分」であることがわかった(Jones 1986a)。これらのルーチンには、1,000行のコードに50個の割合でエラーが含まれており、それらの修正コストがシステム全体の開発コストの10倍に達することもあった(このコストには、カスタマーサポートと保守作業が含まれている)。

3つ目は、高価なルーチンが開発に及ぼす影響が明らかであることだ。「時は金なり」という古いことわざがあるが、この場合は必然的に「金は時なり」という結果になる。問題の多いルーチンをなくせば、コストの80%近くを削減でき、工期も大幅に短縮できる。このことは、品質を改善すると開発スケジュールが改善され、開発コストが削減されるという、ソフトウェア品質の原則を具体的に示している。

参照 過度に複雑なルーチンも、エラーが起きやすいルーチンに分類される。ルーチンの複雑さを見極め、単純にする方法については、上巻第19章の「19.6.2 複雑さを軽減するための一般的なガイドライン」を参照。

4つ目は、問題の多いルーチンが保守に及ぼす影響も同様に明らかであることだ。保守作業は、エラーが発生しやすいと指摘されたルーチンを調べ、設計を見直し、初めから書き直すことに焦点を当てなければならない。前述のIMS

プロジェクトでは、エラーが発生しやすいクラスを置き換えたところ、IMSの生産性が約15%も改善された(Jones 2000)。

22.4.2 エラーの分類

一部の研究者が、エラーを種類別に分類して、種類別に発生頻度を割り出そうとした。どのプログラマにも、境界値の1つ違いエラーやループ変数の再初期化の見落としなど、これはやっかいだと感じているエラーがある。これらの詳細については、本書のチェックリストで取り上げていく。

Boris Beizerは、いくつかの研究データを組み合わせて、エラーをきわめて詳細に分類している(Beizer 1990)。Beizerのデータをまとめると、次のようになる。

25.18%	構造
22.44%	データ
16.19%	実装された機能
9.88%	コンストラクション
8.98%	統合
8.12%	機能要求
2.76%	テストの定義または実行
1.74%	システムアーキテクチャ、ソフトウェアアーキテクチャ
4.71%	その他

Beizerは、結果を小数点以下2桁の精度で示しているが、エラーの種類に関する調査は一般に確定的ではない。調査によって、エラーの種類は大きく異なる。同じような種類のエラーを報告している調査でも、結果は大きく異なっており、0.01%どころか50%も違うものもある。

調査結果にばらつきが見られることから、Beizerのように複数の調査結果を組み合わせたところで、おそらく意味のあるデータは得られないだろう。しかし、確定的でないにしても、何かを示唆するデータもある。次に、これらのデータが示唆する見解をまとめてみよう。

ほとんどのエラーの範囲はごく限られている

ある調査では、エラーの85%がたった1つのルーチンの変更で修正できることがわかった(Endres 1975)。

■多くのエラーはコンストラクション以外に原因がある

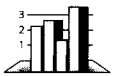
97件の聞き取り調査の結果から、エラーの三大原因が、そのアプリケーション分野の知識不足、要求の変動と矛盾、コミュニケーション不足や協力体制の不備であることがわかった(Curtis, Krasner and Iscoe 1988)。

ひづめの跡を見つけたら、シマウマではなく馬と考えよう。OSは多分壊れていない。そしてデータベースも多分正常だ。

— Andy Hunt、Dave Thomas

■コンストラクション時のほとんどのエラーはプログラマのミスである

数年前に実施された2つの調査の結果から、報告されたエラーのうち約95%がプログラマに原因があり、2%がシステムソフトウェア(コンパイラ、オペレーティングシステム)に原因があり、2%がその他のソフトウェアに原因があり、1%がハードウェアに原因があることがわかった(Brown and Sampson 1973; Ostrand and Weyuker 1984)。1970年代や1980年代に比べると、今日ではシステムソフトウェアや開発ツールがより普及しているので、筆者の予想では、今日のプログラマのミスの割合はこれよりも高いだろう。



HARD DATA

■入力ミス(タイポ)が問題の原因であることが意外に多い

ある調査結果から、コンストラクションエラーの36%が入力ミスであることがわかった(Weiss 1975)。1987年に実施された300万行に及ぶ航空力学ソフトウェアの調査では、エラー全体の18%が入力ミスであることがわかった(Card 1987)。別の調査では、エラー全体の4%がメッセージのスペルミスであることがわかった(Endres 1975)。筆者の同僚は、実行可能ファイルのすべての文字列をスペルチェッカーにかけただけで、筆者のプログラムからスペルミスをいくつか発見した。細かいことに注意しよう。信じられないのなら、ソフトウェアのエラーのうち最も高価なものを3つ挙げるので、それについて考えてみればよい。16億ドル、9億ドル、2.45億ドル。これらはそれまで動作していたプログラムを**たった1文字**変更した結果なのだ(Weinberg 1983)。

■設計への誤解はプログラマのミスの研究における永遠のテーマである

Beizerがまとめた調査は、エラーの16%が設計を正しく理解していないことに原因があることを明らかにした点で価値がある(Beizer 1990)。別の調査では、エラーの19%が設計を正しく理解していないことに原因があることがわかった(Weiss 1975)。時間を惜しまずに、設計を隅々まで理解する必要がある。それに割いた時間がすぐに報われるわけではないが(それどころか、仕事をさぼっているように見えるかもしれない)、プロジェクトが進行する過程で着実に実を結ぶだろう。

■ほとんどのエラーは簡単に修正できる

エラーの約85%は数時間以内に修正できる。約15%は数時間から数日以内に修正できる。そして、約1%はそれよりも時間がかかる(Weiss 1975; Ostrand and Weyuker 1984; Grady 1992)。この結果は、エラーの20%の修正にリソースの約80%が割かれたというBoehmの所見によっても裏付けられ

ている(Boehm 1987b)。要求と設計のレビューを上流で行って、面倒なエラーをできるだけ多く排除しよう。数え切れない小さなエラーをできるだけ効率よく処理しよう。

■組織内のエラーの経験がどれくらいかを調べるとよい

ここで紹介した結果に大きなばらつきがあることは、組織によって経験に大きな差があることを示している。したがって、他の組織の経験を自分の組織に当てはめるのは難しい。一般の予想を裏切るような結果もある。自分の直感を他のツールで補う必要がある。出発点として、開発プロセスを測定して、問題がどこにあるか把握することから始めよう。

22.4.3 | 不完全なコンストラクションによるエラーの割合

エラーを分類するデータが確定的でないとしたら、さまざまな開発作業で発生するエラーのデータの大半もそういうことになる。1つだけはっきりしているのは、コンストラクションには膨大な数のエラーがつきものであることだ。コンストラクションで発生するエラーは、要求や設計のエラーよりも修正コストが低いという意見がある。コンストラクションの個々のエラーを修正するコストはそうかもしれないが、全体のコストで考えた場合、実際のデータはそれを裏付けていない。

筆者の見解は次のとおりである。

- 小規模なプロジェクトでは、コンストラクションのエラーがエラー全体の大部分を占める。小規模なプロジェクト(1,000行のコード)のコーディングエラーの調査では、要求に原因があるエラーが10%、設計に原因があるエラーが15%であるのに対し、コーディングに原因があるエラーが75%にのぼった(Jones 1986a)。この割合は、小規模なプロジェクトの多くを代表するものであると考えられる。

- プロジェクトの規模にかかわらず、コンストラクションのエラーはエラー全体の35%以上にのぼる。大規模なプロジェクトでは、コンストラクションのエラーの割合は低くなるが、それでもエラー全体の35%以上を占める(Beizer 1990; Jones 2000)。かなり大規模なプロジェクトで、75%台に達するという調査結果もある(Grady 1987)。一般に、そのアプリケーション分野をよく理解していればいるほど、全体的なアーキテクチャも良くなる。それにより、エラーは詳細設計やコーディングに集中するようになる(Basili and Perricone 1984)。

- コンストラクションのエラーは、要求や設計のエラーよりも修正コストが低いとはいえ、やはり高価である。Hewlett-Packardが2つのかなり大規

模なプロジェクトを調査した結果、コンストラクションのエラーの平均コストは、設計のエラーの平均コストの20～50%であることがわかった(Grady 1987)。コンストラクションのエラーの数が増えた場合、コンストラクションのエラーを修正するための総コストは、設計のエラーを修正するコストに匹敵するか、その2倍に達するだろう。

図22-4は、プロジェクトの規模とエラーの原因との関係を表している。

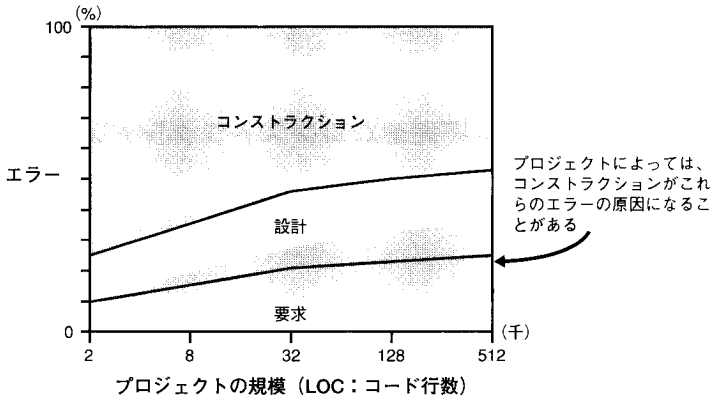
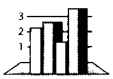


図22-4 プロジェクトの規模が大きくなると、コンストラクション時に発生するエラーの割合は少なくなる。いずれにせよ、大規模なプロジェクトでも、コンストラクションのエラーはエラー全体の45～75%を占める

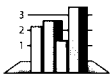
22.4.4 検出が期待されるエラーの数

検出が期待されるエラーの数は、開発プロセスの品質次第である。考えられる範囲は次のとおりである。



- 業界平均では、完成したソフトウェアの1,000行に1～25個の割合でエラーが検出される。ソフトウェアはさまざまな手法を組み合わせて開発されることが多い(Boehm 1981; Gremillion 1984; Yourdon 1989a; Jones 1998; Jones 2000; Morales 2003)。手法によってエラーが10分の1になることはまれだし、10倍になったという話もあまり聞かない(そうだとしたら、そもそも完成していない)。
- Microsoftのアプリケーション部門では、社内テストでコード1,000行あたり10～20個のエラーが検出され、製品のリリース後はコード1,000行あたり0.5個のエラーが検出されている(Moore 1992)。この水準は、第21章の「21.4 その他のコラボレーティブ開発プラクティス」で説明したコードリーディングと独立テストを組み合わせることで達成された。

■ Harlan Millsは「クリーンルーム開発」のパイオニアである。クリーンルーム開発とは、社内テストで検出されるエラーをコード1,000行あたり最大で3個、リリース後の製品で検出されるエラーをコード1,000行あたり最大で0.1個に減らすことができる手法だ(Cobb and Mills 1990)。たとえば、スペースシャトルのソフトウェアを開発するプロジェクトでは、公式の開発手法、ピアレビュー、統計テストから成るシステムを使って、コード50万行あたりのエラーが0個という水準を達成した(Fishman 1996)。

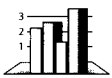


■ Watts Humphreyは、Team Software Process(TSP)を採用したチームが、コード1,000行あたりのエラーが0.06個という水準に達したと報告している。TSPは、最初からエラーを持ち込まないように開発者をトレーニングすることに重点を置いている(Morales 2003)。

TSPプロジェクトとクリーンルームプロジェクトの成果は、「品質の低いソフトウェアを開発して修正するよりも、品質の高いソフトウェアを開発する方が安価である」というソフトウェア品質の一般原則を裏付けるものとなっている。完全にチェックされた8万行のクリーンルームプロジェクトの生産性は、1人月あたり740行だった。コーディング以外の作業も含む完全にチェックされたコードの業界平均は、1人月あたり250～300行あまりである(Cusumano et al. 2003)。このようなコストの削減と生産性の高さは、TSPプロジェクトやクリーンルームプロジェクトでは実質的にデバッグが不要であることによるものだ。デバッグが要らない？ 目標はこうでなくちゃ。

22.4.5 テスト自体のエラー

このような経験をしたことがないだろうか。ソフトウェアにエラーがあるようだ。コードのどこがおかしいのかすぐにわかるだろうと思っていたのに、どこもおかしくないようだ。さらにテストケースをいくつか実行してエラーを洗い出そうとしたが、新しいテストケースではどれも正しい結果が得られる。何時間もかけてコードを何度も読み返し、結果を手作業で計算してみる。すべて正しい。それから数時間後、ふと思い付いてテストデータを調べ直してみる。やられた。エラーはテストデータにあったのだ。コードじゃなくてテストデータのエラーを何時間も探していたとは、何てばかばかしい。



このような経験はだれにでもある。テストケースはテストするコードよりもエラーが紛れ込みやすいくらいである(Weiland 1983; Jones 1986a; Johnson 1994)。その理由は簡単だ —— 特に、開発者がテストケースを作成する場合には。テストケースは、入念な設計プロセスやコンストラクションプロセスを経ずに、その場で作成されることが多い。一時的なテストと見なされることが多く、まるで使い捨てのように気軽に作成される。

テストケースのエラーの数は、次の方法で減らすことができる。

■作業をチェックする

テストケースはコードを開発するときと同じくらい慎重に作成する。これには、テストケースを二重にチェックすることも含まれる。製品コードと同様に、テストケースのコードをデバッガでチェックする。テストデータのウォークスルーやインスペクションを実施してもよいだろう。

■ソフトウェアの開発と同様にテストケースを準備する

テストの準備を始めるなら、要求を策定する段階で、あるいはプログラムを担当することになった時点で着手するのが効果的である。そうすれば、誤った仮定に基づくテストケースを防ぐのに役立つ。

■テストケースを保管する

テストケースを少し熟成させる。回帰テストやバージョン2の作業のために保存しておく。テストケースを捨てずに残しておくことがわかれば、問題を解決しておこうという意欲も湧くだろう。

■単体テストをテストフレームワークに統合する

最初に単体テストのコードを書いたうえで、それぞれのテストが完了したら、それらをシステム全体のテストフレームワーク(JUnitなど)に統合する。統合されたテストフレームワークがあれば、テストケースの使い捨てという傾向はなくなるだろう。

22.5 テストサポートツール

ここでは、市販のテストツールや独自に作成するテストツールについて見てみよう。本書を手に取るころには消えている可能性もあるので、特定の製品は紹介しない。最新の製品については、定期購読しているプログラマ向けの雑誌を参照してほしい。

22.5.1 個々のクラスをテストするための足場固め

「足場」という言葉は建築用語である。足場が組まれるのは、それがないと作業員が建物の中を移動できないからだ。ソフトウェアの足場を組むのは、ひとえにコードを調査しやすくするためである。

足場の1つは、テスト対象のクラスが使用するダミーのクラスである。このようなオブジェクトは「モック(擬似)オブジェクト」または「スタブオブジェクト」と呼ばれる(Mackinnon, Freemand and Craig 2000; Thomas and Hunt 2002)。オブジェクトより低レベルのルーチンでも同じような手法を用

いることができ、こちらは「スタブルーチン」と呼ばれる。モックオブジェクトやスタブルーチンをどれくらいリアルにするかは、どれくらいの正確さが必要かによる。それにより、足場は次のようなものになる。

参照 Jon Bentley著『Programming Pearls, 2nd edition』の「A Small Matter of Programming」に、参考になる足場の例がいくつか紹介されている(Bentley 2000)。

- 何もせずに、すぐに制御を戻す。
- 与えられたデータをテストする。
- 入力パラメータをエコーする、メッセージをファイルに記録するなど、診断メッセージを出力する。
- 対話型の入力から戻り値を取得する。
- 入力とは無関係に標準的な戻り値を返す。
- 実際のオブジェクトやルーチンに割り当てられるのと同じクロックサイクルを消費する。
- 実際のオブジェクトやルーチンの遅いバージョン、重いバージョン、軽いバージョン、あるいは精度の低いバージョンとして機能する。

もう1つの足場は、テスト対象のルーチンを呼び出す偽のルーチンである。これは「ドライバ」や「テストハーネス」と呼ばれる。この場合の足場は次のようになる。

- オブジェクトを固定の入力で呼び出す。
- 入力を対話型で受け取り、その値でオブジェクトを呼び出す。
- コマンドラインから引数を受け取り(オペレーティングシステムがこれをサポートしている場合)、その値でオブジェクトを呼び出す。
- ファイルから引数を読み取り、その値でオブジェクトを呼び出す。
- 既定の入力データを順番に取り出して、オブジェクトを何度か呼び出す。

最後の足場は、ダミーファイルである。ダミーファイルとは、本物のファイルを小さくしたもので、本来の大きさのファイルに含まれるものと同じ種類のデータを持つ。小さなダミーファイルには、いくつか利点がある。ダミーファイルは小さいので、その内容をすべて確認することができ、ファイル自体にエラーがないことを確認しやすい。そして、ダミーファイルはテスト専用なので、明白なエラーを発生させるような内容にすることもできる。

参照 テストツールとデバッグツールの境界線はあいまいだ。デバッグツールの詳細については、第23章の「23.5 デバッグツール」を参照。

cc2e.com/2268

当然、足場を組むにはそのための作業が必要だが、あるクラスでエラーが発見されていれば、その足場を再利用することができる。そして、モックオブジェクトなどの足場を簡単に作成できるツールも登場している。足場を組めば、他のクラスの影響を受けない状態で、クラスをテストできるようになる。こうした足場は、難解なアルゴリズムを使っている場合に特に効果的である。テストするコードが他のコードに埋め込まれているために、テストケースを1つ実行するのに数分もかかっていたのでは、行き詰まりやすい。足場を組めば、コードを直接実行できるようになる。深く埋め込まれたコードを実行するための足場を組むほんのわずかな時間により、デバッグ時間を大幅に短縮できる。

プログラムの足場を組むためのテストフレームワークも続々登場している(JUnit、CppUnit、NUnitなど)。現在の環境が既存のテストフレームワークをサポートしていない場合は、クラスにいくつかルーチンを書き、クラスをテストするための足場となるmain()ルーチンをファイルに追加すればよい。テストするルーチン自体が独立している必要はない。main()ルーチンは、コマンドラインから引数を読み取り、それらをテストするルーチンに渡して、プログラムに統合する前のルーチンを単体で調査できるようにする。コードを統合する際には、ルーチンとそれらをテストするための足場コードをファイルに残したままプリプロセッサコマンドで無効にするか、足場のコードをコメントにすればよい。プリプロセッサで無効にすれば実行可能コードに影響を及ぼすことはないし、ファイルの底に埋もれているのでそれとはわからない。そのまま残しておいても害はない。再び必要になったらそこに残っているし、それを削除したり保存したりする手間も省ける。

22.5.2 差分ツール

回帰テスト(再テスト)は、期待される出力と実際の出力とを自動的に比較するツールがあると、かなり楽になる。出力をチェックする簡単な方法の1つは、出力をファイルにリダイレクトして、diffなどのファイル比較ツールを使用することである。そして、あらかじめファイルに保存しておいた期待される出力と新しい出力との差分をとればよい。出力が食い違っていれば、回帰エラーが検出されたことになる。

参照 回帰テストについては、「22.6.2 再テスト(回帰テスト)」を参照。

22.5.3 テストデータジェネレータ

cc2e.com/2275

プログラムの一部を体系的に調査するようなコードを書くこともできる。数年前、私は独自の暗号化アルゴリズムを開発し、それを使用してファイル暗号化プログラムを書いた。このプログラムの目的は、ファイルをエンコードして、パスワードが正しく入力された場合にのみ、ファイルをデコードすることだった。この暗号化はファイルの外観を変えるだけでなく、ファイルの内容全体を変えるものだった。肝心なことは、プログラムがファイルを正しくデコードできるかどうかだった。さもなければ、ファイルは台無しになってしまう。

そこで、私はテストデータジェネレータを用意して、プログラムの暗号化部分と復号化部分をすべて調査した。テストデータジェネレータは、任意の文字から成る0～500KBのファイルと、1～255文字の任意の文字から成るパスワードを生成した。テストデータジェネレータはそれぞれのケースで、ランダムなファイルを2つ生成する。一方のファイルを暗号化して、自身をもう一度初期化した後、暗号化したファイルを復号して、復号したファイルとオリジナルのファイルを1バイトずつ照合する。1バイトでも違ったら、テストデータジェネレータはエラーを再現するのに必要なすべての情報を出力する。

私は、テストケースの照準を自分のファイルの平均的な長さである30KBに合わせた。500KBという最大サイズからすれば、かなり小さいファイルだ。テストケースの照準を小さいファイルに合わせなければ、ファイルサイズは0～500KBに均一に分散され、テストファイルの平均サイズは250KBになっていただろう。ファイルの平均サイズが小さいことは、均等なサイズにした場合よりも、ファイル、パスワード、EOF条件、異常なサイズのファイルなど、エラーの原因として考えられる状況に対して、テストの数を増やせることを意味する。

結果は満足のいくものだった。100個のテストケースを走らせただけで、プログラムに2つのエラーが見つかった。いずれも、通常の使用法では見つからないような特殊なものだったが、エラーであることに違いはなく、見つかってよかった。それらを修正した後、プログラムを数週間にわたって走らせ、のべ10万個以上のファイルの暗号化と復号を繰り返したが、エラーは見つからなかった。テストしたファイルの内容、サイズ、パスワードの範囲から見て、プログラムが正しいことが確信できた。

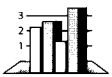
この話の教訓をまとめてみよう。

- ランダムデータジェネレータを正しく設計すれば、プログラマが思い付かないような変わった組み合わせのテストデータを生成することができる。
- ランダムデータジェネレータを使用すれば、プログラマではとうてい不可能なほど、プログラムを徹底的に調査することができる。

- ランダムに生成したテストケースは、現実的な入力を重視するものに少しずつ改良していくことができる。これにより、ユーザーが使用する可能性が高い範囲にテストの照準が合わせられ、その範囲での信頼性が高まる。
- モジュール型の設計はテストで活きてくる。この場合は、暗号化と復号化のコードをユーザーインターフェイスコードから切り離して使用することができたので、テストドライバを簡単に作成することができた。
- テストドライバでテストするコードを変更しなければならない場合でも、テストドライバを再利用することができる。前述の2つのエラーを修正したとき、私はすぐにテストを再開することができた。

22.5.4 | カバレッジモニタ

cc2e.com/2282



HARD DATA

Karl Wiegersは、コードカバレッジを測定せずに行ったテストでは、一般にコードの50～60%しか調査されないと報告している(Wiegers 2002)。カバレッジモニタは、調査されたコードと調査されなかったコードを追跡するツールである。一連のテストケースによってコードが完全に調査されたかどうかを確認できるので、特に体系的なテストに効果の高いツールだ。テストケースをすべて実行した後、カバレッジモニタが実行されていないコードを示した場合は、テストを増やす必要があることがわかる。

22.5.5 | データレコーダとログの記録

プログラムを監視して、エラーが発生したときにプログラムの状態に関する情報を収集するツールがある。飛行機が墜落したときに調査する「ブラックボックス」のようなものだ。ログがきちんと記録されていれば、エラーを診断するのに役立つだけでなく、ソフトウェアがリリースされた後に効果的なサービスを提供することができる。

データレコーダを独自に開発する場合は、重大なイベントをファイルに記録すればよい。エラーに先立つシステムの状態と、エラー状態に陥ったときの詳細を正確に記録する。この機能を開発中のコードに埋め込み、リリースバージョンのコードで無効にすればよい。あるいは、記憶域のサイズを自動的に調整し、エラーメッセージの配置と内容に配慮するログ機能を実装して、それをリリースバージョンに埋め込んでもよいだろう。

22.5.6 シンボリックデバッガ

シンボリックデバッガとは、コードのウォークスルーとインスペクションを行うプログラムである。デバッガは、コードを1行ずつ実行し、変数の値を追跡し、常にコンピュータと同じようにコードを解釈する機能を持つ。デバッガでコードを1行ずつ実行し、その動きを監視するプロセスは、非常に重要である。

参照 利用できるデバッガの種類は、テクノロジ環境の成熟度によって異なる。この現象については、上巻第4章の「4.3 テクノロジの波に乗って」を参照。

デバッガによるコードのウォークスルーは、いろいろな意味で、レビューで他のプログラマのコードを1行ずつ追いかけるのと同じである。同僚が見落とすものも、デバッガが見落とすものも、あなたが見落とすものとは異なる。デバッガには、チームレビューよりも労働集約型ではないという利点がある。さまざまな入力データでコードの実行を監視すれば、思いどおりのコードが実装されているかどうかを確認するのに役立つ。

良くできたデバッガでは、コードが実行されるようすをそのまま確認できるので、言語の習得にも役立つ。高級言語のコードとアセンブラコードの表示を切り替えながら、高級言語のコードがアセンブラにどのように変換されるのかを確認することができる。また、レジスタやスタックを監視して、引数が渡されるようすを確認することができる。さらに、コンパイラによって最適化されたコードを調べて、どのような最適化が実行されるのかも確認できる。これらの利点は、デバッガの本来の用途である検出済みのエラーの診断とはあまり関係がないが、斬新な発想でデバッガを使用すれば、その本来の目的をはるかに上回る効果が得られるだろう。

22.5.7 システム混乱ツール

cc2e.com/2289

テストサポートツールには、システムをわざと混乱させるものがある。プログラムは99回目まで動作していたのに、同じデータの100回目の実行では動作しなかったという話をよく耳にする。原因はほぼ同じで、どこかで変数の初期化を忘れてしまったのだ。100回のうちの99回は、初期化されていない変数の値がたまたま0になるので、このエラーを再現させるのはそう簡単ではない。

この種のテストサポートツールの機能はさまざまだ。

■ **メモリの設定**

初期化されていない変数がないことを確認したい場合、初期化されていない変数がたまたま0になることがないように、プログラムを実行する前にメモリに任意の値を設定するツールがある。場合によっては、メモリに特定の値を設定する。たとえば、x86プロセッサでは、0xCCという値はブレークポイント割り込みのマシン語コードである。メモリを0xCCで埋めておけば、実行すべきではないものを実行するようなエラーが発生したときに、デバッガのブレークポイントにヒットしてエラーを検出できる。

■ **メモリのシェイキング**

マルチタスクシステムでは、相対アドレスではなく絶対アドレスのデータに依存するコードが書かれていないことを確認するために、プログラムの実行時にメモリの内容を再配置するツールがある。

■ **選択的なメモリ不足**

メモリドライバによっては、プログラムがメモリを使い果たす、メモリの割り当てに失敗する、メモリの割り当てを何度か繰り返したら失敗する、逆にメモリの割り当てに何度か失敗したら成功するといった、メモリ不足の状態をシミュレートできる。これは特に、動的にメモリを割り当てながら動作する複雑なプログラムのテストに効果的である。

■ **メモリアクセスチェック(境界検査)**

境界チェッカーは、ポインタが正しく処理されることを確認するために、ポインタの処理を監視する。このようなツールは、初期化されていないポインタや未解決のポインタを検出するのに役立つ。

22.5.8 | エラーデータベース

cc2e.com/2296

強力なテストツールの1つは、報告されたエラーのデータベースである。このデータベースは、管理ツールであると同時に、テクニカルツールでもある。このデータベースを利用すれば、再発するエラーを調べたり、新しいエラーが検出され、修正される割合を調べたり、公開されているエラーと公開されていないエラーの状態とそれらの重大度を調べたりできる。エラーデータベースに登録すべき情報については、「22.7　テスト記録の保管」で説明する。

22.6 | テストの改善

テストを改善するための作業は、他のプロセスを改善するための作業と同じである。このプロセスがどのようなものか正確に理解すれば、それを少し変化

させて、その影響を観察することができる。プラスに働いた変更があったら、さらに良くなるようにプロセスを修正する。ここでは、これをテストで行うための方法について見ていこう。

22.6.1 | テスト計画

効果的なテストの1つの鍵は、プロジェクトの開始当初からテスト計画を立てることである。テストを設計やコーディングと同じように重要であると位置付ければ、そのための時間が確保され、重要であると見なされ、結果として高品質なプロセスになるだろう。テスト計画は、テストプロセスを**反復可能な**ものにするための重要な要素でもある。テストを繰り返すことができなければ、改善することはできない。

参照 テスト計画には、それを文書にまとめることも含まれる。テスト計画書の詳細については、第32章の「32.7 参考資料」を参照。

22.6.2 | 再テスト（回帰テスト）

製品を徹底的にテストして、エラーが見つからなかったとしよう。その後、その製品のある部分が変更されたので、変更前と同じようにすべてのテストにパスするかどうか、つまり、変更によって新しいエラーが持ち込まれていないかどうかを確認したい。ソフトウェアが後戻りしていないこと、すなわち「回帰」していないことを確認するためのテストを「回帰テスト」という。

ソフトウェア製品に変更が加えられた後に体系的に再テストを実施することができなければ、高品質なソフトウェア製品を生産することはほぼ不可能である。変更が加えられるたびに以前とは別のテストを実行したのでは、新しいエラーが紛れ込んでいないことを確認するすべはない。したがって、回帰テストでは毎回同じテストを実行しなければならない。製品が成熟するにつれて新しいテストが追加されることもあるが、それまでのテストも引き続き実施する。

22.6.3 | 自動テスト

回帰テストを管理する方法として考えられるのは、それを自動化することだけである。同じテストをいく度となく繰り返し、同じ結果がいく度となく繰り返されるのを見ているうちに、人は麻痺してしまい、エラーを簡単に見逃してしまうようになる。これでは、回帰テストの目的は台無しだ。テストの第一人者であるBoriz Beizerは、手動によるテストでミスをする割合は、テストし

ているコードのバグの割合に匹敵すると報告している。Beizerは、手動によるテストの効果は、すべてのテストを正しく実行した場合の約半分であると推測している(Johnson 1994)。

テストの自動化には、次のような利点がある。

- 自動テストでは、手動によるテストよりもミスが発生する可能性が低い。
- テストを自動化すれば、わずかな作業でプロジェクトの他の部分にも利用できるようになる。
- テストを自動化すれば、頻繁に実行できるようになるので、チェックインされたコードによってコードが動かなくなっていないかどうかを確認できる。テストの自動化は、デイリービルド、スモークテスト、エクストリームプログラミング(XP)などのテスト主導のプラクティスの土台の一部である。
- テストを自動化すれば、問題を早期に発見できる可能性が高まる。多くの場合、これにより問題を診断して修正する作業は少なくなる。
- 自動テストは、コードの変更時に持ち込まれたエラーをすばやく検出する機会を増やすので、広い範囲のコードを変更する場合のセーフティネットとなる。
- 自動テストは、テクノロジ環境の変化に敏感に反応するので、変化の激しいテクノロジ環境では特に効果的である。

参照 テクノロジの成熟度と開発プラクティスの関係については、上巻第4章の「4.3 テクノロジの波に乗って」を参照。

自動テストをサポートするための主なツールは、テストの足場を提供し、入力を生成し、出力を捕捉し、実際の出力と期待される出力を比較するものだ。ここまで紹介してきたさまざまなツールの中には、これらの機能の一部を実行するものもあれば、すべてを実行するものもある。

22.7 テスト記録の保管

プロジェクトへの変更がプロジェクトを改善したかどうかを確認するためには、テストプロセスを反復可能にすることに加えて、プロジェクトを測定できなければならない。プロジェクトを測定するために収集できるデータをいくつか紹介しよう。

- 欠陥の管理上の記載事項(報告日、報告者、タイトルまたは説明、ビルド番号、修正日)
- 問題の詳細な説明
- 問題を再現するための手順
- 問題に対して推奨される回避策
- 関連する欠陥
- 問題の重大度(致命的、中度、軽度など)
- 欠陥の分類(要件、設計、コーディング、テスト)
- コーディングエラーの細分類(1つ違い、不正な代入、無効な配列インデックス、不正なルーチン呼び出しなど)
- 修正によって変更されたクラスおよびルーチン
- 欠陥の影響を受けたコード行数
- 欠陥の検出にかかった時間
- 欠陥の修正にかかった時間

データを収集したら、それを基に次の値を計算すると、プロジェクトが順調かどうかを判断することができる。

- 各クラスの欠陥の数(悪いクラスから良いクラスの順にソートし、可能であればクラスのサイズで正規化する)
- 各ルーチンの欠陥の数(悪いルーチンから良いルーチンの順にソートし、可能であればルーチンのサイズで正規化する)
- 検出された欠陥1件あたりのテストの平均所要時間
- テストケース1件あたりの欠陥の平均検出数
- 修正された欠陥1件あたりのプログラミングの平均所要時間
- テストケースがカバーするコードの割合
- 各重大度の分類において突出している欠陥の数

22.7.1 個人的なテスト記録

プロジェクトレベルのテスト記録の他に、個人的なテスト記録を保管しておくと効果的かもしれない。これらの記録には、よくあるエラーのチェックリストと、コーディング、コードのテスト、エラー修正の所要時間の記録を盛り込むとよい。

22.8 参考資料

cc2e.com/2203

「真実を知らせよ」という連邦制定法がある以上、本章よりもテストを詳しく取り上げている書籍が何冊かあることを白状しないわけにはいかない。これらの書籍は、本章では取り上げなかったシステムやブラックボックステストについて解説している。また、デベロッパーテストについてもさらに詳しく解説し、原因結果グラフといった公式のアプローチや、検査機関を設置することの一部始終を解説している。

テスト

- 『Testing Computer Software, 2nd edition』(Cem Kaner、Jack Falk、Hung Q. Nguyen著、John Wiley & Sons、1999年)
  『基礎から学ぶソフトウェアテスト』(日経BP社、2001年、テスト技術者交流会訳)
  本書の執筆時点では、ソフトウェアテストに関する最高の1冊。アクセス数の多いWebサイトや市販のアプリケーションなどの一般市場に配布されるアプリケーションのテストに最適だが、一般書としても役立つ。

- 『Lessons Learned in Software Testing』(Cem Kaner、James Bach、Bret Pettichord著、John Wiley & Sons、2002年)
  『ソフトウェアテスト293の鉄則』(日経BP社、2003年、テスト技術者交流会訳)
  『Testing Computer Software, 2nd edition』の次に読むのにお勧めである。著者が学んだ300近くの教訓を11の章に分けて解説している。

- 『Introducing Software Testing』(Louise Tamre著、Addison-Wesley、2002年)
  テストを理解しなければならない開発者を対象とした入門書。タイトルに反して、経験豊富なテスト担当者にも役立つテストの詳細にも踏み込んでいる。

■『How to Break Software: A Practical Guide to Testing』(James A. Whittaker著、Addison-Wesley、2002年)
ソフトウェアを失敗させる23種類の攻撃と、人気の高いソフトウェアパッケージを使った例を紹介する。その独特のアプローチから、テストの主な情報源としても、テスト本の補足情報としても利用できる。

■「What Is Software Testing? And Why Is It So Hard?」(James A. Whittaker著、『IEEE Software』(January 2000) pp.70-79)
ソフトウェアテストの問題点を紹介し、ソフトウェアを効果的にテストするための課題について説明するすばらしい記事。

■『The Art of Software Testing』(Glenford J. Myers著、John Wiley & Sons、1979年)
『ソフトウェア・テストの技法』(近代科学社、1980年、長尾真監訳)
ずいぶん前に出版されたソフトウェアテストの本だが、まだ出版されている[※1]。本書はわかりやすく書かれており、自己診断テスト、プログラムテストの心理学と経済学、プログラムのインスペクション、ウォークスルー、レビュー、テストケースの設計、クラスのテスト、高次元のテスト、デバッグ、テストツールとその他のテクニックを取り上げている。約200ページと短く、読みやすい。冒頭のクイズはテスト担当者ならどう考えるかを体験させるものとなっており、コードを分解する方法がいくつあるかを具体的に示している。

※1 訳注
原書は2004年6月に第2版が発行されている。

テストの足場

■『Programming Pearls, 2nd edition』のコラム5「A Small Matter of Programming」(Jon Bentley著、Addison-Wesley、2000年)
『珠玉のプログラミング ― 本質を見抜いたアルゴリズムとデータ構造』(ピアソン・エデュケーション、2000年、小林健一郎訳)
テストの足場を組むための参考になる例がいくつか紹介されている。

■「Endo-Testing: Unit Testing with Mock Objects」(Tim Mackinnon、Steve Freeman、Philip Craig著、eXtreme Programming and Flexible Processes Software Engineering - XP2000 Conference、2000年)
デベロッパーテストをサポートするモックオブジェクトの使用法を解説した初めての論文。

■「Mock Objects」(Dave Thomas、Andy Hunt著、『IEEE Software』(May/June 2002))

デベロッパーテストをサポートするモックオブジェクトの使用法を紹介するとても読みやすい記事。

cc2e.com/2217

■ www.junit.org
JUnitを使用する開発者をサポートするサイト。同様のリソースがcppunit.sourceforge.netとnunit.sourceforge.netでも提供されている。

テストファースト開発

■ 『Test-Driven Development: By Example』(Kent Beck著、Addison-Wesley、2003年)
『テスト駆動開発入門』(ピアソン・エデュケーション、2003年、長瀬嘉秀監訳、永田渉、武田知子訳)
テストケースを先に書き、そのテストケースを成功させるコードを書くという特徴を持つ、「テスト駆動開発」を余すところなく解説する。エバンジェリスト調な語り口ではあるが、アドバイスは的確で、ページ数も少なく、ポイントを押さえている。実際のコードを使った例を満載している。

IEEE標準

■ IEEE Std 1008-1987(R1993) 「Standard for Software Unit Testing」

■ IEEE Std 829-1998 「Standard for Software Test Documentation」

■ IEEE Std 730-2002
「Standard for Software Quality Assurance Plans」

cc2e.com/2210

チェックリスト22-1 テストケース

- ❑ クラスまたはルーチンに適用される要求ごとに、専用のテストケースを作成したか。
- ❑ クラスまたはルーチンに適用される設計要素ごとに、専用のテストケースを作成したか。
- ❑ コードの各行を少なくとも1つのテストケースでテストしたか。これを、すべてのコード行を調査するのに必要なテストの最小数を割り出すことによって確認したか。
- ❑「定義された ― 使用された」データフローのすべてのパスを少なくとも1つのテストケースでテストしたか。
- ❑「定義された ― 定義された」、「定義された ― 抜けた」、「定義された ― 破棄された」など、正しくないと思われるデータフローパターンについてコードをチェックしたか。
- ❑ 過去に頻繁に発生したエラーを検出するために、よくあるエラーのリストを使ってテストケースを作成したか。
- ❑ 単純な境界条件(最大値、最小値、1つ違いの境界値)をすべてテストしたか。
- ❑ 複合境界条件をテストしたか。つまり、変数に小さすぎる値や大きすぎる値が代入されるような入力データの組み合わせをテストしたか。
- ❑ データの種類が正しいかどうかをチェックするテストケースを作成したか。たとえば、給与計算プログラムの従業員数が負数の場合をテストしたか。
- ❑ 代表的な値(中間あたりの値)をテストしたか。
- ❑ 正常構成の最小値をテストしたか。
- ❑ 正常構成の最大値をテストしたか。
- ❑ 古いデータとの互換性をテストしたか。古いハードウェア、オペレーティングシステムの古いバージョン、ソフトウェアの古いバージョンとのインターフェイスをテストしたか。
- ❑ テストケースは手作業でも調べやすい内容か。

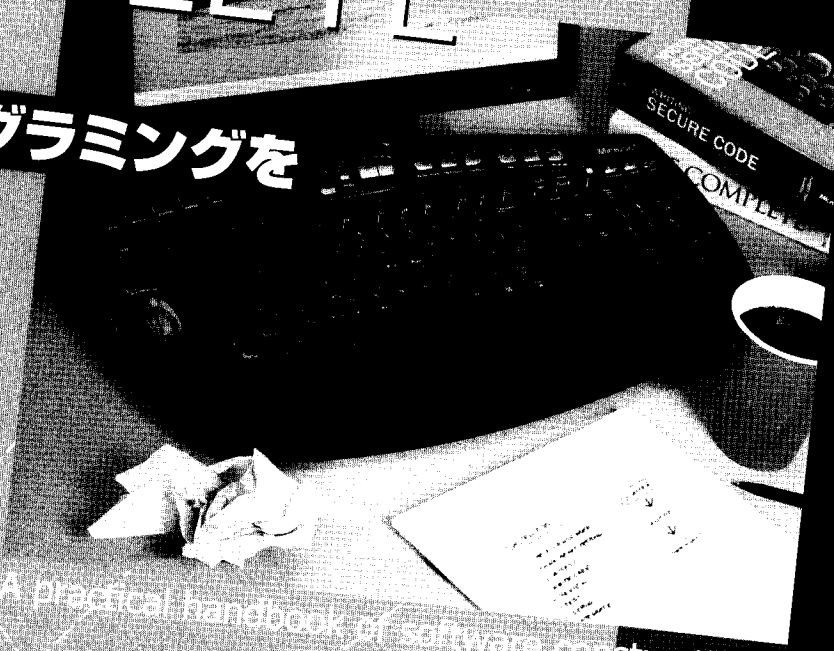
22.9 まとめ

- 開発者によるテストは、本格的なテスト戦略において重要な位置を占める(独立テストも重要だが、本書では取り上げない)。
- コードよりも先にテストケースを書くと、テストケースよりも先にコードを書く場合と時間や労力は変わらないが、欠陥検出/修正サイクルは短くなる。
- 利用できるテストはさまざまだが、テストは優れたソフトウェア品質改善プログラムの一部にすぎない。要求や設計の欠陥を最小限に抑えるといった高品質な開発手法も同じくらい重要である。コラボレーティブ開発プラクティスも、エラーの検出という点ではテストと同じくらい効果的である。そして、これらのプラクティスはテストとは異なる種類のエラーを検出する。
- 基礎テスト、データフロー分析、境界分析、悪いデータ、良いデータを基に、多くのテストケースを確実に生成することができる。エラーの推測を基に、さらにテストケースを生成することができる。
- エラーはエラーを発生しやすい一部のクラスやルーチンに集中する傾向にある。エラーを発生しやすいコードを突き止め、設計を見直し、書き直すこと。
- テストデータにはテスト対象のコードよりもエラーが紛れ込みやすい。このようなエラーを追いかけていると、コードが改善されないまま時間がどんどん過ぎていくので、テストデータのエラーはプログラミングエラーよりもやっかいである。テストをコードと同じくらい慎重に作成して、こうしたエラーを防ぐこと。
- 自動テストは一般に便利であり、回帰テストに欠かせない。
- 長い目で見れば、テストプロセスを改善する最も良い方法は、テストを日常化し、評価し、それを改善するための手法を用いることである。

コードコンプリート

CODE COMPLETE

完全なプログラミングを目指して



Steve McConnell 著
(株)クイープ 訳

A practical handbook of software construction

日経BPソフトプレス