テスト、面白いですか？本当に？

開発プロセスで単体テストと機能テストを利用する

Jeff Canna

2001年 3月 01日

テスト。私はテストがダ～イ嫌いです。私は、いままでズ～とテストが嫌いでした。単体テストと機能テストはどちらも「本来の」仕事の邪魔になります。誰でも、作成したコードは完ぺきだと思っているものです。コードを変更する必要がある場合に、誰が見ても処理内容がわかるように十分なコメントが記述されることはめったにありません。私はもっと大人になる必要があるでしょうか？ (カウンセラーに相談する必要があるかもしれません)

過去数年の間に、単体テストは、私がソフトウェアを作成する場合の主要な方法になりました。Extreme Programming (XP) と呼ばれる取り組みやすいプログラミング方法論 (参考文献を参照) のおかげです。この方法論では、追加したすべての機能について単体テストを実施し、そのテスト結果を保管する必要があります。単体テストに失敗したコードは組み込むことができません。ベースになるコードが多くなるにつれて、開発者は、これらのテストによって、自信を持って変更を組み込むことができます。

私は当初、これらの単体テストの存在によって機能テストが不要になると考えていました。これは間違いでした。機能テストと単体テストはまったく違うものです。これらがどのように異なるのか、またこれらを共に利用して開発プロセスを強化するにはどうすればよいかを理解するまでに長い時間がかかりました。

この記事では、単体テストと機能テストの違いについて説明します。さらに、日常の開発においてそれらを使用するプロセスの概要も説明します。

テストと開発プロセス

開発者にとってテストは非常に重要であり、日常的に実施する必要があります。開発サイクルの特定の段階にテストをまとめて実施することはできません。顧客にシステムを渡す前には、最後に必ずテストを実施する必要があります。テスト以外に仕事が完了したことを知る方法があるでしょうか。テスト以外に小さなバグの修正がシステムの主要な機能を停止させるかどうかを知る方法があるでしょうか。テスト以外に現在の予想よりも優れた状態にシステムを進化させる方法があるでしょうか。単体テストと機能テストはどちらも、開発プロセスの一部として組み込まれる必要があります。

© Copyright IBM Corporation 2001

単体テストはコードの作成方法の中心に置く必要があります。特に、取り組んでいるプロジェクトの時間的な制約がきびしく、プロジェクトを常に管理したい場合はなおさらです。単体テストは、非常に重要なので、コードを記述する前にテストを記述する必要があります。

保守された単体テストのセットには次の特徴があります。

- 可能な限り最も実践的な設計を表します。
- クラスの文書化に最適な形式を提供します。
- クラスがいつ「完成」したかを判断します。
- コードに対する自信を開発者に与えます。
- 迅速な変更の基礎になります。

単体テストによって、システムと共に自然に発展する設計文書が作成されます。このことを忘れないでください。システムと共に自然に 発展する文書は、ソフトウェア開発における聖杯 (特効薬) と言えます。コード化されたユース・ケースのセットを提供するよりもクラスを文書化する方法が効果的です。それがこれらの単体テストの内容です。つまり、クラスの機能を文書化するコード化されたユース・ケースのセットであり、制御された一連の入力が与えられます。常に単体テストを実施して成功する必要があるので、この設計文書は常に最新の状態になります。

テストはコードを記述する前に記述する必要があります。そうすることで、テスト対象のクラスの設計が提供され、コードの小さな塊に集中して取り組むことができます。この実践方法はまた、設計を簡素化します。将来のことを考えて不要な機能をインプリメントすることもなくなります。最初にテストを記述すると、どの時点でクラスが完成したことになるかを知ることができます。すべてのテストに成功したときにこのタスクは完了します。

最後に、単体テストは、大きな自信を提供し、これが開発者の満足に変わります。コードを変更するたびに単体テストを実行すれば、変更によって損なわれるものがあるかどうかが直ちにわかります。

機能テストは、システムがリリースできる状態になったかどうかを確認するものであり、単体テストよりも重要です。機能テストによって、有効な方法で作業中のシステムを定義することができます。保守された機能テストのセットには次の特徴があります。

- ユーザーの要件を効果的な方法で入手します。
- システムがそれらの要件を満たすという自信をチーム (ユーザーと開発者) に与えます。

機能テストによって、ユーザーの要件を効果的な方法で入手できます。従来型の開発では、ユース・ケースを使用して要件を取得します。通常は、ユース・ケースに関する議論を行い、その調整に多くの時間を費やします。それらが終わったときには、紙だけが残ります。機能テストは、自己検証するユース・ケースのようなものです。Extreme Programmingの方法論によってこの概念を説明することができます。XPストーリーは、顧客と開発者の間で将来についてかわされる会話に対する約束です。機能テストはこの会話の結果です。機能テストを含まないストーリーは効果的に構築できません。

機能テストによって単体テストで残されたギャップが埋まり、チームはコードに対する自信を深めます。単体テストでは見つからないバグが多数存在します。単体テストは必要なコード をすべてカバーしますが、必要なシステム をすべてカバーしない場合があります。機能テストは、単体

テストで見つからなかった問題を明らかにします。保守され、自動化された機能テストのセットでもすべてをキャッチすることができない場合があります。しかし、最高の単体テストのセットと比べてもさらに多くの問題を見つけることができます。

単体テストと機能テストの比較

単体テストは、コードが正しく処理できること を開発者に示します。機能テストは、コードが処理する内容が正しいこと を開発者に示します。

単体テスト

単体テストはプログラマーの観点から記述されます。単体テストは、クラスの特定のメソッドが一連の具体的なタスクを正しく実行することを保証します。各テストは、既知の入力が与えられたときにメソッドが予想どおりの出力を生成することを確認します。

フレームワークのテストを行わずに保守可能な自動化された単体テストのセットを記述することは不可能です。テストを記述する前に、チームの合意に基づくフレームワークを選択する必要があります。フレームワークは頻繁に使用することになるので、好きなフレームワークを選ぶ方が効果的です。Extreme ProgrammingのWebサイトには利用可能な単体テストのフレームワークがいくつか用意されています (参考文献を参照)。私が最もよく使うのは、Javaコードのテスト用のJunitです。

機能テスト

機能テストは、ユーザーの観点から記述されます。これらのテストでは、ユーザーの期待どおりにシステムが機能することを確認します。

システムの開発は、しばしば家を建てることにたとえられます。このたとえが完全に正しいわけではありませんが、単体テストと機能テストの違いを理解するためにこのたとえを活用することができます。単体テストは、工事の監督が家の建築現場を訪れることに似ています。監督は、家の内部のさまざまなしくみ、基礎、骨組み、電気設備、配管などの検査を中心に行います。彼は、家の各部分が正しく安全に機能すること、つまり構築基準を満たしていることを確認 (テスト) します。このシナリオにおいて機能テストは、家の持ち主が同じ建築現場を訪れることにたとえられます。彼は、内部のしくみが正しく機能すること、つまり工事の監督が彼の職務を果たしていることは当然のことと考えます。家の持ち主は、この家に住むとどうなるかということに重点を置きます。彼は、家の外観がどうなっているか、各部屋の大きさが適当かどうか、家が家族の要望を満たしているかどうか、ちょうど朝日が射しこむ位置に窓があるかなどを確認します。家の持ち主は、家についての機能テストを実行します。彼は、ユーザーの観点から見ます。工事の監督は、家についての機能テストを実施し、建築家の観点から仕事をします。

単体テストと同じように、フレームワークのテストを行わずに保守可能な自動化された機能テストのセットを記述することはほとんど不可能です。Junitは、単体テストに大きな効果を発揮しますが、機能テストを記述しようとするとそれほど効果はありません。機能テストにおいてJunitと同じように使用できるものはありません。この目的で使用できる製品もありますが、これらの製品が実稼動環境で使用されているのを見たことはありません。ニーズを満たすテスト用のフレームワークが見つからない場合は、自分で構築する必要があります。

作業中のプロジェクトの構築についてどれほど熟知していても、またどれほど柔軟なシステムを構築しても、作り出すものが使えなければ、時間を無駄にしたことになります。このため、機能テストは開発における最も重要な部分と言えます。

どちらのテストも必須であり、それらを記述するためのガイドラインが必要になります。

単体テストの記述方法

単体テストの記述を始めるときには、どうすればよいか困ることが、よくあります。まずは、新しい コード用の単体テストの作成から始めるのが最善の方法です (既存のコード用の単体テストの作成から始めることは困難ですが、それも可能です)。新しいコードから始めて、プロセスに慣れたら、既存のコードに取り組みそれらのテストを作成します。

前述のように、単体テストはテスト対象のコードを記述する前に記述する必要があります。まだ存在しないコード用のテストを記述するにはどうすればよいでしょう。これは当然の疑問です。この手法の習得の90 %は精神的であり、10 %が技術的です。つまり、単純にテスト対象のクラスが存在すると想定するという意味です。想定してからテストを記述します。最初は多くの構文エラーが発生しますが、そのまま続けてください。この試行錯誤を通して、クラスがインプリメントするインターフェースを定義します。次の段階として、単体テストを実行し、構文エラーを修正し (つまり、テストによって定義されたインターフェースを持つクラスをインプリメントし)、そしてテストを再実行します。毎回、間違いを十分修正できる量のコードを記述し、このプロセスを繰り返します。コードがテストに合格するまでテストを繰り返します。すべての単体テストに成功したとき、コードは「完成」します。

一般的に、クラスに含まれるすべてのpublicメソッドについて単体テストを実行する必要があります。しかし、取得や設定のメソッドなどの非常に単純な機能を持つメソッドについては、"特殊" な方法で取得や設定を行う場合を除き、単体テストは不要です。これを判断する効果的なガイドラインは、コード内の一部の動作についてコメントする必要があると思われるたびに単体テストを記述することです。多くのプログラマーと同じようにコードにコメントを記述することを好まない場合、単体テストはコードの動作を文書化するための手段になります。

単体テストは、テストされた関連クラスと同じパッケージに入れます。このような編成にすれば、各単体テストで、メソッドを呼び出し、テスト対象のクラス内にある`package` または`protected` のアクセス修飾子を持つ変数を参照することができます。

単体テストではドメイン・オブジェクトを使用しないようにします。ドメイン・オブジェクトとはアプリケーションに固有のオブジェクトです。たとえば、スプレッドシート・アプリケーションに登録オブジェクトが含まれることがありますが、このようなオブジェクトはドメイン・オブジェクトです。ドメイン・オブジェクトについて既にわかっているクラスがある場合は、それらのオブジェクトをテストで使用してもかまいません。しかし、これらのオブジェクトを使用しないクラスがある場合は、テストによってこれらのオブジェクトをクラスに関連付けないようにします。このようなやり方を避ける理由はすべてコードの再利用にあります。多くの場合、あるプロジェクト用に作成されたクラスは他のプロジェクトにも適用されます。これらのクラスの再利用が容易な場合があります。しかし、再利用されるクラスのテストで別のプロジェクトのドメイン・オブジェクトを使用する場合は、テストを開始するまでに非常に時間がかかる可能性があります。通常の場合、テストは失敗するかまたは書き直されます。

このメカニズムは効果的に利用できますが、テストを実行しなければ包括的な単体テストのセットには価値がありません。早い段階で頻繁にテストを実行すると、コードに対する絶対の自信を常に持つことができます。プロジェクトの進行に合わせて、機能を追加していきます。テストを実行すると、インプリメントしたばかりの新機能によって損なわれたものがあるかどうかがわかります。

単体テストの記述のメカニズムを習得したら、再び既存のコードに取り組みます。既存コード用のテストの記述は困難な場合があります。テスト自体のためのテストは行わないようにします。テストは、有効なテストを持たないクラス (またはまったくテストを持たないクラス) を修正する必要が見つかった場合に、「必要に応じて」記述します。そのような状況こそがテストを追加する状況です。通常のように、クラスの単体テストでは、そのクラスの各メソッドの機能を取得する必要があります。最も簡単にテスト対象を見つける方法は、既存のコード内のコメントを参照することです。単体テストではすべてのコメントを取り込む必要があります。メソッドの最初にあるメソッドの機能を説明するコメントのブロックを単体テスト用に変換します。

機能テストの記述方法

機能テストは非常に重要ですが、開発では、のけ者と評価されることがあります。ほとんどのプロジェクトには、機能テストを実施する独立したグループがあります。通常は、常にシステムと対話して、システムの機能が正しいかどうかを判断する大勢の人々がいます。このようなグループを作るのはばかげています。

機能テストも単体テストと同様のアプローチで実行する必要があります。ユーザーとの対話処理 (ダイアログなど) を生成するコードを記述する必要があるときに直ちに、ただし実際にそのコードを記述する前に、テストを記述します。ユーザーと協力して、ユーザーの要件を取り入れた機能テストを記述します。新しいタスクを始めるときには、必ず機能テストのフレームワーク内でそのタスクを記述します。その後開発作業を進め、新しいコードを追加したら単体テストを行います。すべての単体テストが成功したときに、元の機能テストを実行し、テストが成功するかまたは修正が必要かを調べます。

理論的には、機能テスト・グループの概念はなくなるはずです。開発者は、ユーザーと共に機能テストを記述する必要があります。システム用の機能テストのセットを作成した後には、機能テストを担当する開発チーム・メンバーが、最初のテストのバリエーションを使用してシステムを酷使する必要があります。

単体テストと機能テストの境界

多くの場合、単体テストと機能テストの境界は不明確です。正直なところ、私にも明確な境界がわからないことがあります。私は、単体テストを記述するときに、次のようなガイドラインに従って、記述する単体テストが実際には機能テストであるかどうかを判断します。

- 単体テストがクラスの境界を超えている場合、それは機能テストである可能性があります。
- 単体テストが非常に複雑になる場合、それは機能テストである可能性があります。
- 単体テストが変更されやすい場合 (つまり、有効なテストであっても、さまざまなユーザーの設定を処理するために絶えず変更する必要がある場合)、それは機能テストである可能性があります。

- 単体テストの記述がテスト対象のコードの記述よりも困難な場合、それは機能テストである可能性があります。

「機能テストである可能性 がある」という言葉に注意してください。ここで使用できる確かな定着したルールはありません。単体テストと機能テストには境界があります。しかしその境界は開発者自身が決める必要があります。単体テストに慣れるほど、特定のテストが単体テストから機能テストへの境界を超える状況がわかるようになります。

まとめ

単体テストは、開発者の観点から記述され、テスト対処のクラスの特定のメソッドに焦点を当てます。単体テストを記述するときには次のガイドラインを使用してください。

- 単体テストはテスト対象のクラスのコードを記述する前に記述します。
- コードのコメントを単体テストに取り込みます。
- 「特殊」な機能を実行するすべてのpublicメソッド (つまり取得や設定ではないメソッド。ただし特別な方法で取得や設定を行う場合は対象となります) をテストします。
- 各テスト・ケースをテスト対象のクラスと同じパッケージに入れ、パッケージと保護されたメンバーにアクセスできるようにします。
- 単体テストではドメイン固有のオブジェクトを使用しないようにします。

機能テストは、ユーザーの観点から記述され、ユーザーが関心を持つシステムの動作に焦点を当てます。有効な機能テストのフレームワークを見つけるかまたは自分で開発します。機能テストを使用して、ユーザーに本当に必要なものを識別します。このようにして、機能テストの担当者は自動化されたツールと、ツールを使用するための出発点を得ることになります。

単体テストと機能テストは、開発プロセスの中心に置きます。そうすることで、システムの機能と成長に自信を持つことができます。それらを中心に置かない場合は、自信を持つことができません。テストは楽しい仕事ではないかもしれませんが、機能する単体テストと機能テストがあれば、開発の楽しさが増えます。

関連トピック

- AntとJunitを用いた漸進的開発 (developerWorks, 2000年11月) では、特にAntとJunitを用いた場合の単体テストの利点について説明されています。
- Extreme Programming の方法論について理解してください。
- Extreme ProgrammingのWebサイトから各種の単体テストのフレームワークをダウンロードすることができます。

© Copyright IBM Corporation 2001
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)