Eclipse統合開発環境とC++testで行う効率的なシステム開発
～コーディングからデバッグ、単体テストをシームレスに実行～

テクマトリックス株式会社 システムエンジニアリング事業部
ソフトウェアエンジニアリング技術部 チームリーダー
橘田　和仁

C++testは、C/C++プログラムの単体テストに必要なテストドライバ、テストスタブ、テストケースを生成し、単体テストを自動実行することで実行時例外やエラーを検出するC/C++対応自動単体テストツールです。「900種類のコーディングルール」「コードレビュー機能」「バグ探偵」「単体テストケースの自動生成」「カバレッジ計測」などの多くの機能をEclipseやVisual Studioに統合することより、効率的な開発サイクルを支援します。

単体テストの重要性

単体テストとは、システムのもっとも単純な機能ポイントにおけるソフトウェアコードテストのことです。（機能ポイントとはC言語であれば1つの関数、C++言語であれば1つのクラスです。）

一般的に単体テストは、QAフェーズではなく開発サイクル中の開発者によって実行されます。単体テストを体系的に実行することによって、エラーのない安定したコードだけをアプリケーションに統合できるため、システム全体の品質が確実に向上します。

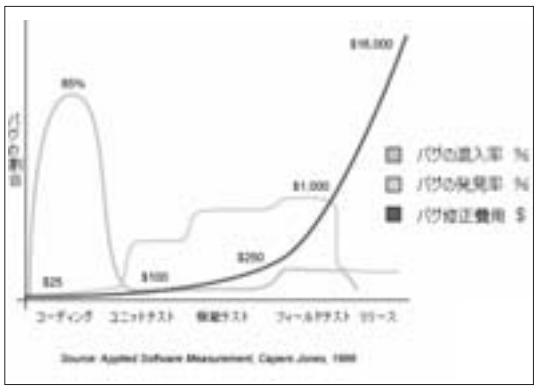

図1 後工程で見つかった欠陥ほど修正コストが増大

一方、図1のグラフのように、プログラム原因のバグが結合テストやシステムテストなどの後工程で見つかった場合の修正コストは、単体テスト時で発見した場合の修正コストと比べると、数十倍になると言われています。ましては、リリース後の障害となると、顧客ビジネスへのインパクトなども含め、その影響は計り知れません。

＊

プログラム原因のバグを開発の後工程に持ち越さないためには、開発サイクルのもっとも早期の時点、つまり、単体テストの段階でコードの欠陥を特定し修正を行うことが重要なプロセスだと言えます。結合テストやシステムテストなどの後工程の段階で、テストケースを考えるのは遅すぎます。なぜならば、この段階ではすでに膨大なパターンの検証が必要な設計になっている場合があるからです。そのような状況になってしまった場合、入力データの組み合わせテストや、論理パスの組み合わせテスト、異常ケースのテストなどを厳密に行うことは実質不可能です。たとえ、これらをすべてテストしようと思っても、テストするリソースが足らず、結果的に検証が不十分になりバグを見逃してしまいます。また見つけられたとしても、前述したように後工程ではバグの修正コストが大きくなってしまいます。

＊

開発者が自分で単体テストを作成する場合、テストドライバ/テストハーネスの準備、入力データの指定、実行できない関数を代替するスタブの用意など、多くの作業が必要になり、さらに開発リソースの負担が増加します。C++testでは、図2のような作業の自動化や作成を支援し、効率的で一貫した単体テストプロセスを可能にします。

テストケースの自動生成と実行
アサーションの自動生成
関数スタブの自動生成/カスタマイズ
実行時カバレッジの計測（6 種類）
関数メトリクスの計測
既存の CppUnit テストケースのインポートと実行
回帰テストの作成/実行
スタブ化する箇所の選択
外部ファイルからのテストケースのインポート
シナリオテストの作成/実行

図 2　C++testが支援する単体テスト関連の機能

また、C++test7.1では独自GUIは存在せず、EclipseやVisual StudioなどのIDEに統合して使用します。Eclipse CDTやVisual Studioで作成されたプロジェクト設定を使用して単体テストを行えば、複雑なセットアップは不要です。Makefileを利用しプロジェクトを作成することもできます。EclipseやVisual Studioに統合することで、テスト駆動開発や反復開発プロセスとの親和性も向上しました。

テストケースの生成

C++testが作成するテストケースは、C/C++のソースコードで生成されます。C++testのテストケースの形式は、一般によく使用されているCppUnitの形式に似ていますが、より幅広い機能をユーザに提供しています。Cコードのテストが可能であるほか、テストフレームワーク中のprivateデータ、protectedデータ、メンバ関数へのプログラム的なアクセスが可能です。

```
01:    void TestSuite_test_EX4_checkAge_10()
02:    {
03:        /* Pre-condition initialization */
04:        /* Initializing argument 1 (age) */
05:        int _age   = 99;
06:        /* Initializing global variable gstatus */
07:            gstatus   = 0 ;
08:        /* Initializing global variable gval */
09:            gval   = cpptestLimitsGetMaxInt();
10:        {
11:            /* Tested function call */
12:            int _return   = EX4_checkAge(_age);
13:            /* Post-condition check */
14:            CPPTEST_POST_CONDITION_INTEGER("int
               _return", ( _return ))
15:            CPPTEST_POST_CONDITION_MEM_BUFFER("int
               gstatus ", ( gstatus ), 8)
16:        }
17:    }
```

図 3　C++testが自動生成したテストケース

図 3 が、C++testが自動生成するテストケースです。ソースコードの中から、分岐条件に関連する値を取得してテストケースの入力値に使用しています。C++test7.1からは、分岐条件の前後の値もテストケースの入力値として設定するようになり、自動生成テストケースも使いやすくなっています。

そして、CppUnit形式のテストケースは非常に自由度が高く、テストケース内でのメモリの確保、データ型のキャスト、ポインタに関連する操作など、ソースコードで表現できることをそのまま実行できます。もちろん一連の処理をまとめて実施するような、シナリオテストのようなことも行えます。

C++testは既存のCppUnitのテストケースを、手を加えることなく実行できます。インポートしたCppUnitのテストケースに対してもカバレッジ計測が行われるので、既存のテスト資産もより有効に活用できます。

回帰テストの自動化

回帰テストとは、既存のエラーを修正したときや、新しい機能を追加したときに発生するコードの変更によって既存の機能に影響が出ていないか、予想外の変化が発生していないかを検証することです。

回帰テストはその重要性が認識されているにもかかわらず、実際のテスト計画に含まれない場合がほとんどです。回帰テストが実行されない理由にはさまざまな原因がありますが、大きな理由として変更のたびにテストを実行することは効率が悪く、大量のリソースを使用するからではないでしょうか。単体テスト時における回帰テストにも同様のことが言えます。

*

プログラマがどのような作業に時間を掛けているかを調べると、実際にコードを書いている時間は少なく、デバッグに多くの時間を使っていることに気づくでしょう。またプログラマであれば、原因を特定するまでに丸一日費やすようなバグに遭遇したことがあるのではないでしょうか。

図 4 の例を見てください。例えば、ある関数に対してテストケースが20個あるとします。プログラマはデバッガを使用して順番にテストを

実行していきます。そして10番目のテストケースを実行したときに、プログラム上で異常が発生したとします。ここでプログラマはソースコードを修正しますが、その場合、プログラマはもう一度1番目のテストケースからテストを実行するでしょうか？実際は10番目のテストケースが正常に実行されるようにプログラムを修正し、次の11番目のテストケースを実行してしまうことが多いようです。

つまりこの場合、プログラムの修正前に実施したテストケース1～9に影響がでていないことは保障されていませんので、新たなバグが混入している可能性があります。

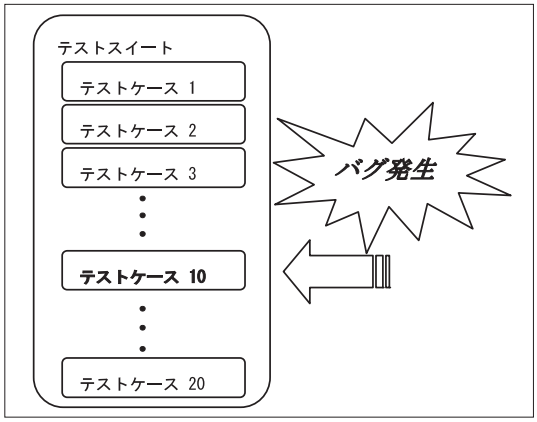

図4　テストの実施途中でバグを発見

この問題は、体系的に単体テストケースを作成することにより解決できます。単体テストケースを作成しておけば、テストを繰り返し実行できるようになり、いつでも同じようにテストを実行できます。テストの自動化にもなりますので、回帰テストの実行がコンパイル並みに簡単になります。テストの途中でバグが見つかったとしても、迷わず1番目のテストケースから実行できる環境になります。

一見するとテストを実行するまでに準備工数がかからない「デバッガを使用したテスト」のほうが効率よく思えますが、体系的に作られた単体テストケースにより、生産性が向上していることに気づくはずです。

単体テストでのテストケース作成は、コードの品質を保証するための重要なタスクですが、体系的に作られた単体テストは機能の問題とバグを発見するだけでなく、定期的な回帰テストとしても効果を発揮します。そして、人間の思考を必要とするタスクにより多くの開発リソースを割り当てることができるようになるのです。

カバレッジ分析

テストについての知識のない担当者が気ままに行った単体テストと、体系的に実施された単体テストでは、その効果には大きな差があります。その差が大きく現れるのがカバレッジ分析を利用したテストの実施です。

ツールを使用してカバレッジ分析を行っていない場合、プログラマは全パスの55%～60%ぐらいしかテストを実行していないようです。そして、プログラマがカバレッジを意識していないとソフトウェアは複雑になります。またプログラマ自身が複雑になっていることに気づかないことにもなります。

複雑なソフトウェアには膨大な入力パターンの組み合わせや、網羅すべき論理パスが存在し、すべてをテストすることは不可能です。

```
1:    int checkViolation(char b) {
2:        int* y = 0;
3:    if(b > 200) { b = 0;}
4:        if(b != 0) {
5:        if (gstatus != 0)
6:            y = &gval;
7:        if (y != 0)
8:            return *gstatus + *y;
9:        else
10:           y = 0;
11:       }
12:       else {
13:       if (gstatus != 0)
14:           y = &gval;
15:       if (y != 0)
16:           return *gstatus + *y;
17:       else
18:           y = &gval;
19:           *y = *gstatus;
20:       }
21:       return *y;
22:   }
```

図5　テストできない複雑なコード

一般的にカバレッジを計測すると、プログラマが行うテストで、全パスの80%～90%がカバーされるようです。カバレッジ計測が定着すると、テストの実施が容易な設計、プログラミングを心がけるようになりますので、さらにカバレッジが向上します。

C++testでは、単体テスト実行時に、行、ブロック、パス、判断、条件、MC/DC の6種類のカバレッジ情報をレポートします。これらのカ

```
1:          int checkViolationRef(char b) {
2:              if(b == 0) {
3:                  return falseViolation(b);
4:              }
5:              else {
6:                  return trueViolation(b);
7:              }
8:          }
```

図6　テストを意識したシンプルなコード

バレッジ情報から、単体テストの妥当性を確認することができます。

また、C++testに組み込まれているコーディング規約の中にも、関数の複雑度をレポートするルールが含まれていますので、カバレッジ計測と併用することにより、ソフトウェアが複雑になることを抑止できます。

コードレビュー

C++test7.1では、ソースコードのレビュー作業を支援する「コードレビュー」機能が強化されました。新しくなった「コードレビュー」ではソース構成管理システムに格納する前に、変更されたコードや新たに追加されたコードを特定し、レビュー担当者に通知できるようになりました。これにより、ソース管理システムへ格納する前に、設計の問題やロジックのエラーなどの発見に有効なピアレビューやペアプログラミングが容易に実践できるようになりました。

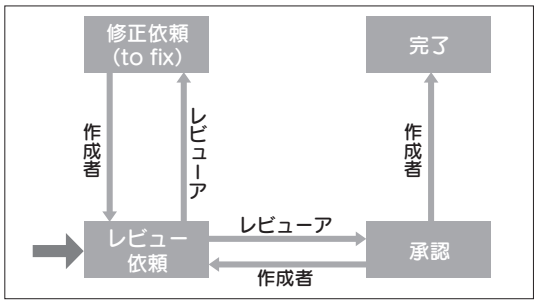

図7　コードレビューのワークフロー

Eclipse統合開発環境との連携

C++test7.1はEclipse統合開発環境上で動作するので、Eclipseに含まれる多くの機能を利用することができます。さらにEclipseには、多くのプラグインが存在します。そのなかには、ドキュメント生成プラグインやCVS、Subversionなどの構成管理クライアントなどもあります。これらのプラグインとC++testとを組み合わせて利用することにより、コーディング、単体テスト、フロー解析、構成管理へのチェックイン/チェックアウト、修正差分の確認、テストケースの共有などをシームレスに実行できます。

また、図8のようにC++testで作成した単体テストのテストケースは、EclipseのIDE上でデバッグ実行することができます。

図8　Eclipse上で単体テストケースをデバッグ

単体テスト、静的解析、フロー解析で高品質なソフトウェアの開発を支援する、C++testとEclipse統合開発環境を使って、効率的なシステム開発を実践してみませんか？