QAC Clinic

Volume 1.1 （株）東陽テクニカ・ソフトウェア・ソリューション 1998

プロジェクト管理者

静的テスト

静的テストとはなんでしょう。ソフトのプログラマーや開発マネージャーにとって、テストはコーディングの終了後に行うものです。実際にプログラムを動作させ、そのパフォーマンスがテストされます。このようなテストを動的テストと呼びます。これに対し、プログラムを動作させずに行うテストが静的テストです。しかし、この静的テストはあまり知られていません。なぜでしょう。恐らくこの静的テストという言葉の代わりに他の言葉がひんぱんに使われているからでしょう。例えばコードの「インスペクション（検査）」という言葉は聞きなれた言葉です。例えば、「コード・インスペクション」や「設計インスペクション」といった表現はよく耳にします。この他にも、「コード・レビュー」とか「設計レビュー」といった表現がよく使われます。本文ではコード・インスペクションに関係した静的テストに焦点を当てていきます。コード・インスペクションというのは読んで字のごとく、手作業であれ、自動ツールであれ、ソースコードを検査してエラーの可能性のあるものを見つけ出すことを意味します。 コード・インスペクションのレベルはかなり異なります。プログラマ個人が自分のコードを検査することから、元ＩBM社員のM.E.Faganによって提案された非常にフォーマルなレビューグループによるコードの検査方法まで様々です。レビューのための自動ツールもカッコだけをチェックする単純なものから、PRL社のＱＡＣ／Ｃ＋＋のような深いレベルでの静的チェックを行うものまでいろいろあります。

２ページに続く

ＱＡＣが今すぐ必要だ！！

プログラマー

日本のコーディング・エラーTop 5

自分のコーディング能力が業界の他のプログラマーと比較してどの程度なのかを考えたことがありますか。Ｃ言語に関して欧米と比較し日本ではどのようなエラーが多いのでしょうか。

詳しい説明を始める前に、本資料の中で使用するデータについて説明します。日本のコードに関しては（株）東陽テクニカ・ソフトウェア・ソリューションのコード解析センターのデータを使用します。過去２年間に弊社のコード解析センターは、日本企業約７０社のコード解析を行い、その解析総行数は2,000,000行を超えています。欧米のコードに関しては、ＱＡＣの製造元である英国ＰＲＬ社のデータを使用します。

それでは、特定の項目に関して日本と欧米の比較を行っていきましょう。最初に関数のプロトタイプ使用率から見ていきます。

3ページに続く

目次

3	ツールを使おう *Dr. Les Hatton*
5	コードの複雑度
7	信頼できるコンパイラー
8	ミススペルのバグ
9	コード解析センター

静的テスト

1ページから続く

確実なコードインスペクションが、バグのないソフトウェアを開発するための最も重要かつ効果的なツール[2]であることを強調しておきます。

図１はインスペクションとテストに関する対投資効果（ＲＯＩ）を示したものです。データはＨＰからのものです。

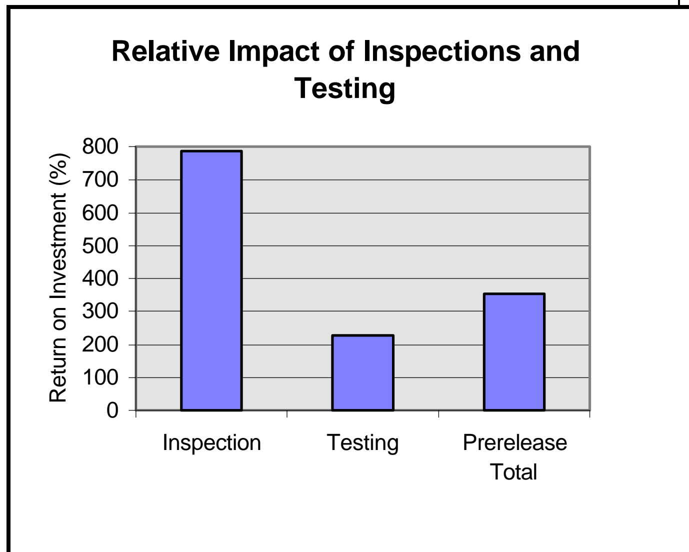


Figure 1

テストと比較しインスペクションの方が3倍投資効果あることを示しています。

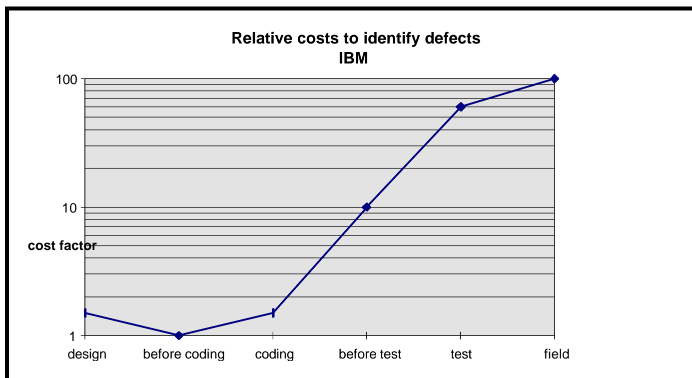


Figure 2

図２はＩＢＭのデータです。ソフトウェア開発サイクルにおけるバグ発見の相対的コストを示しています。テストのプロセスでコストが急激に上昇しているのが分かります。このデータは、コスト的に見てバグを発見する最も効果的な場所はコーディング中かそれ以前であることを示すＨＰのデータを裏付けているようです。

これは何も特別な考えではありません。バグを早く見つければ見つけるほど、その修正にはコストも時間もかかりません。前述したように、コードインスペクションはバグのないソフトウェアを開発するために最も効果的なツールであることが広く知られています。しかし、実際にはこのコードインスペクションはあまり行われていません。これには次の3つの理由があります。

1. 期待するほどにバグが検出できない。
2. 人件費がかかる。
3. マネージャーがコードインスペクションの重要性を一度もまじめに勉強したことがない。

プロジェクトのスケジュールが大幅に遅れているような場合、マネージャーはコードインスペクションを簡単に削ってしまいます！

コードインスペクションでは何を見ることが重要なのでしょうか。

1. 言語の文法エラー
2. 社内コーディング規格の違反
3. ソースコードのレイアウトに関する規格の違反
4. コードの複雑度（テスト性）
5. データ構造のチェック
6. アルゴリズムのチェック
7. 設計基準のチェック

１と２の項目は膨大な情報量が関係するため、手作業で行った場合は全体の問題の１％を見つけるだけでも困難です。プログラマーやチェッカーにコードインスペクションのための知識がないのではなく、人手で行う作業には適していないわけです。

コードインスペクションを正確かつ統一的に行うためだけでなく、限られた時間の中で作業を完了するためにはこのプロセスを自動化すべきです。このプロセスが自動化されないとコストやバグの問題がかならず発生します。

ＱＡＣのような深いレベルでのチェックのできる静的分析ツールを使用すると、チェッカーは人間の思考が重要になる５から７の項目に集中することができます。

静的テストの続き

ここで重要な点をいくつかあげてみましょう。

- 動的テストに対して、静的テストは、プログラムを走らせずに実行される。
- コードインスペクションは静的テストの一つである。
- コードインスペクションはソフトウェアのバグを減らす最も有効な手段として広く知られている。
- 自動化されたコード分析は広範囲なコードインスペクションの一部である。

みなさんの中には過去に静的テストツールを使われたことがあり、もう二度と使いたくないと思われている方もいらっしゃるかもしれません。それはなぜでしょう。おそらく最も良く知られているツールがＵＮＩＸの一部として提供されてきたＬｉｎｔでしょう。Ｌｉｎｔは基本的には有効なツールですが、利用性に限度があり、難しいワーニングがたくさん出力される典型的なツールです。

静的ツールは初期のＬｉｎｔから改善されてきました。深いレベルでの静的テストツールであるＱＡＣは、厳密にはＬｉｎｔとは異なるツールですが、Ｌｉｎｔのテスト項目を上回る項目をテストします。（ワーニング数 800以上に対し500）

またＱＡＣはソフトウェア・メトリックスも提供します。ワーニングとメトリックス・データを解析することでソースコードをより客観的に検証することが可能です。例えば、関数内の行数、分岐文の数、取り得るパスの数、ネストの深さなどの計測が可能です。ソースコードがテスト可能か否か、また保守性にすぐれているか否かの分析の助けになります。

ＱＡＣのワーニング・メッセージの内容は旧Ｌｉｎｔに比べて分かりやすく、必要なメッセージのみ出力させるための設定が可能です。この機能により、ユーザーはメッセージの数を減らしたり、特定の項目に注目して解析することができます。会社のプログラミング規格をＱＡＣに組み込んで違反を自動的にチェックさせることも可能です。規格が守られているか否かのチェックを手作業で行うと、その作業はいつまでたっても終了しないという考えには多くのマネージャーが同意されるのではないでしょうか。

この10年間でアプリケーション・ソフトの利用や製品に占める組み込みソフトの量は劇的に増加しています。これらは私たちの生活に非常に密接に関連しています。ソフトウェアに問題が発生すると、人命が失われたり、多額の損害が生じる可能性があります。ソフトウェアのバグを取り除くための全ての手段を用いて静的テストを確実に行うことがますます重要になっています。

ツールを使おう

Dr. Les Hatton, 「*Safer C*」*の著者*

ソフトウェア工学が「繰り返し発生するエラー」に対して甘いことは長年言われ続けていることである。製造業では20年から30年前に静的プロセス管理を導入し、この問題はシステマティックに解決されている。しかし、我々は「繰り返し発生するエラー」の多くを既に確立した技法を用いて検出できることが分析によって明らかになっている。ツールのサポートが無いとこのようなエラーが検出できない。ＱＡＣのような自動化された静的エラー検出ツールは、繰り返し発生するソフトウェアのエラーを効果的に検出し、既に成熟期を迎えたハードウェア開発との品質ギャップを埋めてくれる力強い存在である。

日本のコーディング・エラー Top 5

ページ1からの続き

1990年にＩＳＯのＣ規格が制定されて以来、関数プロトタイプが規格に盛り込まれ、その使用が奨励されてきました。プロトタイプを使用しないとコンパイラーは関数を呼んでいる時の引数の型がチェックできません。この規格が制定されて既に7年が経過していますが、プロトタイプの使用率は欧米の平均で約60%、日本の平均は20%に過ぎません 。

暗黙宣言が挿入されている比率も見てみましょう。暗黙宣言の挿入とは、型の宣言を何も行っていない関数のことを意味します。Ｃ言語では、関数の型に関して宣言が行われていないと暗黙的にデフォルトで整数型にされてしまいます。このような暗黙宣言の挿入率ゼロが理想的な形です。換言すれば全ての関数の型宣言が行われているのが理想といえます。欧米の平均は9%ですが、日本の平均は15%です。

ＱＡＣのワーニングはユーザーが管理し易いようにグループに分類されています。例えば、移植性に関するワーニング、保守性に関するワーニングといった具合です。このグループの中に「重要なワーニング」というグループがあります。ソースコード中に深刻な問題を発生させる可能性のある所にこのワーニングが出力されます。欧米の平均がソースコード１０００行当たり１２個の警告に対して、日本の平均は１０００行当たり６５個という結果が出ています。

ISOの規格もＣ言語も万国共通なわけですから、なぜ日本の平均が欧米に比べてこんなに高いのかは読者の判断におまかせします。

それでは東陽テクニカのコード解析センターの統計から見た日本のトップ５の問題点を見ていきましょう。発生頻度と問題の深刻度からみて最も大きな問題は、大きな型から小さな型へのデータの暗黙的変換です。例えば、整数型からキャラクター型への変換、整数型からショートへの代入や、ロングから整数型への代入などがこれにあたります。

```
int foo(int a)
{
   char c;
   c = a;
   c++;
   return c;
}
```

上記の例では、整数"a"がキャラクター"c"に代入されています。もし"a"のデータが大きい場合は、データが失われてしまう可能性があります。これまでのコード解析の結果を見ると、日本では30行に１つの割合で(1：30)暗黙的な変換が発生しています。ただし、静的テストのワーニングは暗黙的な変換が発生している場所を指示するだけで、実際にバグかどうかの判断は実データを使用する動的テストで行わなければならないことを忘れないでください。しかし、もし暗黙的変換が発生していなければ、動的テストは不要になることは確かです。

この問題は欧米と比較し、おそらく日本の方がより多く発生していると思います。これは、日本の方がより多くの組込み型アプリケーションが存在していることと、プログラマーがメモリーをセーブするために小さなデータの型を使用する傾向があることに由来しています。

次の重要な問題が未設定変数です。日本の未設定変数の平均は250行に1つ検出に対して、欧米の平均は840行に1つです。この問題は開発アプリケーションの種類には依存しません。単にプログラマーの不注意から発生する問題です。ソースコード中の関数のネストが深いほど、また行数が多ければ多いほど、この問題が発生する頻度が増加します。プログラマーが、分岐文の「True」の状態をコーディングしていると「False」の状態を忘れてしまうのがこの問題の原因です。

```
int foo(int a)
{
   int b;
   if (a) {       b = a;
   }

   return (b) ? 1 : 0;
}
```

この例に示すように、if文の中でbはデータを受け取りますが、ａがFalseの場合にはｂは未設定データとともに使用されてしまう可能性があります。

良く知られている3番目の問題が誤ったキャストです。キャストは適切な型変換が行われれば問題無いのですが、キャストによって本質的な問題がマスクされてしまう可能性があります。マスクとはここでは問題が隠されてしまうことを意味します。コンパイラーや静的テストツールは、キャストが行われていると誤った型変換が行われている場所でワーニングを出力しません。コンパイラーも静的テストツールもプログラマーが意図してキャストを行っているのだと解釈してしまいます。

```
char c;
int a = 10;

(char)c = a;
```

上記の例は誤ったキャストだけでなくシンタックス・エラー（構文エラー）も含んでいます。左辺値をキャストすることはＣ言語の規格に違反します。なぜプログラマーはこのようなミスを犯してしまうのでしょうか。１つの理由はプログラマーがキャストについて学習をしていないこと。そして、２つ目は、たいていのコンパイラーはキャストに対して何のワーニングも出力しないためです。事実、あるコンパイラーでは、上記の例はＣ言語の拡張機能として扱っています。

型の異なるポインター間でのキャストも良く見受けられます。このような場合、コンパイラーは型の異なるポインター間で代入が行われているというワーニングを出力してきますが、このワーニングを止めるためにプログラマーはキャストをしてしまうことが多いようです。もちろんこれはとても危険な方法で、後になってバグが発生すると非常に見つけ難いバグになります。

4つ目の問題は、不適切なマクロの生成とその誤った使用です。マクロは前述した問題と同様に、後になって見つけるのが非常に困難なバグを発生させてしまう可能性があります。ここでの問題は、実際にマクロが拡張された後の拡張コードがプログラマーには見えないことです。結果として、目視検査の実行やデバッガーの利用が非常に制限されてしまいます。それでは、面白い例を見てみましょう。

```
#define AAA    2
#define BBB    4;
#define CCC    8

int foo(void)
{
   int i;
   int idx = BBB;
   for (i = 0; i < BBB; i++) {
     printf("the number is %d", idx);
     idx++;
   }
}
```

idxの代入において実際に拡張された行は次のようになります。

```
int idx = 4;;
```

２つ目のセミコロンに注目してください。次の行の拡張は下記のようになります。

```
for (i = 0; i < 4;; i++) {
```

セミコロンが１つ余分に追加されてしまったため、ｉ＋＋が実行されなくなってしまい、ここにエンドレス・ループが出来あがってしまいました。プログラマーがマクロとその引数に対して完全にカッコを使用しないことがこの問題の大きな原因です。

悪い書き方

```
#define cube(x)  x * x * x
```

良い書き方

```
#define cube(x)  ((x) * (x) * (x))
```

トップ５の問題の最後は「評価の順序」と「優先順位」です。これは、日本だけの問題ではなく、欧米でも問題になっています。プログラマーはこの２つの考え方を混同しているようです。次のコードの一部を見てみましょう。

```
a = foo() * boo() + coo();
```

大抵のプログラマーはこの式では掛け算を最初に行い、その後で足し算を行うことに同意するでしょう。それでは、どの関数が最初に呼ばれますかという質問にあなたは何と答えますか。式は常に左から右に評価されると考えているプログラマーがたくさんいます。従って、この質問に対して`foo()`が最初に呼ばれると答えます。それでは次のようにカッコを追加した場合はどうでしょう。どの関数が最初に呼ばれますか。

```
a = foo() * ( boo() + coo());
```

答えは「決まっていない」です。この場合評価の順序はコンパイラーによって決定され、内容によって評価の順序が異なる可能性があります。コードにカッコを追加することで優先順位が変わり、足し算が最初に行われます。もし３つの関数が完全に独立した内容なら、どの関数が先に呼ばれたとしても変わりはないかもしれませんが、３つの関数がグローバル変数にアクセスし変更しているような場合はどうなるでしょうか。後になって見つけるのが非常に困難なバグの原因となります。「優先順位」はオペレーター(演算子)に関連する問題であり、ISOのC言語の規格に明確に定義されています。これに対して「評価の順序」はオペランド(非演算子)に関連する問題です。評価の順序によって結果が異なるようなコーディングを行っているプログラマーは、自分自身の頭痛の種を作っているだけでなく、他のプログラマーや、プロジェクトのマネージャーにとっても頭痛の種になります。

これまで述べてきた問題の内、未設定変数の問題以外に対してはコンパイラーは何もワーニングを出力しません。それでは、どのようにしてこれらの重要な問題を解決したら良いのでしょうか。プログラマーやプロジェクト・リーダーにとって、コードを静的にテストし、適切なワーニングを出力するQACは非常に力強い味方になるはずです。

コードの複雑度

コードの複雑度とは何でしょう。なぜ複雑度を気にしなければならないのでしょう。複雑度が私たちのソフトウェア・プロジェクトにどのような影響を与えるのでしょうか。辞書には「複雑度」が次のように説明されています。

「複雑度」

たくさんの異なる部分をもち、それらの部分の互いが理解し難い、または処理し難い方法で結びついている状態の度合い。

この解釈からすると、複雑度が増加すればするほど理解が困難になるという推測ができます。しかし、辞書には複雑度がどのレベルに達した時に人間の理解を超えてしまうのかの説明はされていません。

なぜコードの複雑度の心配をしなければならないのでしょうか。理由は、コードがテスト可能かどうか、また保守が可能かどうかの指標になるからです。

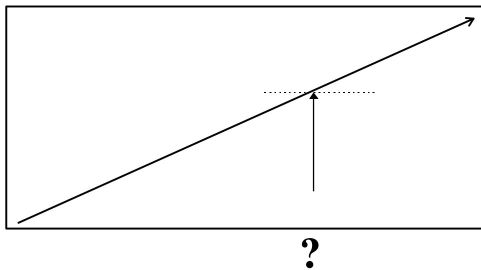
「経路複雑度」の提唱者であるThomas McCabeは次のように述べています。

ソフトウェア工学には深刻な問題が存在している。いかにしてテストと保守が可能なソフトウェアのモジュールを作成するかという問題である。

非常に興味深いことに、McCabeは1976年にこのように述べています。22年たった現在でもソフトウェア・マネージャーやプログラマーは依然としてこの深刻な問題を抱えています。

ロジックは簡単です。プログラムのモジュールの複雑度が高すぎてそのモジュールの理解が不可能な場合、誰がそのモジュールを保守することができますかという話です。さらに、テストケースの数も大きな問題です。もし、この複雑なモジュールを完全にテストするために何千、いや何百万ものテストケースが必要だとしたらどうしますか。テストが行われずに多くのバグが残されたまま使用されているソフトウェアはたくさんあると推測されます。

プログラミングが開始された時点ではコードの複雑度はゼロです。しかし、何ヶ月にも渡りプログラミングが進むに従って人間の理解を超えるモジュールやプロジェクトの予算から考えてテスト不可能な数のテストケースを必要とするモジュールが増えてきます。



それでは、テストや保守が不可能になるポイントをどのように知ることができるでしょうか。これは非常に重要な事です。なぜなら、コーディングの終わりに近づいた時点ではコードがテスト不可能で理解もできないことが分かっても遅すぎるからです。

ソフトウェアの複雑度の計測は長い間の課題でした。そして、何年にもわたり多くの手法が開発されてきました。ほとんどの手法は、構文解析をベースとしているもので、ソースコード中の単一または複数のプロパティ（特性）を計測するものです。これらのプロパティをグループ分けして計測することを「ソフトウェア・メトリックス」と呼んでいます。全ての異なるソフトウェア・メトリックスについての技術的説明は本文ではしませんが、複雑度の概念を説明するいくつかのメトリックスにつぃては触れていきます。1976年にThomas McCabeは論文を発表しました。この中で、McCabeは、プログラムの複雑度を制御フローによって定義することができ、ルーチンの中の「decision point （決定点）」をカウントすることで計測できると主張しました。C言語の場合、if, while, for, caseを決定点として数え、その合計数に１を加えて計測します。

McCabeのメトリックスは普通サイクロマティック・コンプレキシティ（経路複雑度）と呼ばれています。 McCabeは著書の中で、この経路複雑度が10を超えると、そのルーチンは人間の理解の範囲を超え、結果的に欠陥率が急速に増加すると述べています。しかし、メトリックスと複雑度の関係は1対1の関係には無い点に注意してください。

例えば、if文がネストした経路複雑度10の方が、case文の経路複雑度10に比べ実際の複雑度が高いのは明らかです。また、 McCabeのメトリックスでは、下記の例に示すようなif文中のロジカル・オペレーター（論理演算子）に依存する複雑度は考慮されていません。

```
if (a) {
}

if (( a && b && c) || d) {
}
```

Myersという人が後に分岐文の中の論理演算子の数を数えることで経路複雑度をさらに拡張しました。このメトリックスは現在「マイヤーズ・インターバル」や「拡張経路複雑度」と呼ばれています。 複雑な論理決定はバグを発生しやすい事を考えると、McCabeとMyersのメトリックスを組み合わせることで、条件分岐文のより現実的な複雑度を見ることができます。 [Moller 1993][1]

もう１つの有効なメトリックスが「静的パスカウント」です。これはプログラムのテスト性を測定するもので、プログラムの中の可能性のある全てのパス（経路）数の上限値です。基本的に、パラレルなパスは足し算され、シリアルなパスは掛け算されます。研究によって適切とされるパスカウント数は様々ですが、一般的に言って２００から１０００の間が適切であると言われています。ちなみにQACの場合このパスカウントの計測上限は500,000,000ですが、お客様のコードにこのようなパスカウントを持つ関数がよく見受けられます。100%のカバレッジテストが必要なプロジェクトなどでは、このメトリックスが、必要とされるテスト数の良い指標となります。

最後が「コード行数」です。コード行数の数え方がたくさんあるため、このメトリックスはかなりあいまいです。QACはコード行数のカウントに関して4つのメトリックスを提供しますが、その中で最も役に立つものの1つが関数内の「実行コード数」です。このメトリックスを見ることによってユーザーは長い関数がどこにあるのかをモニターすることができます。また、COCOMOモデルのベースにもなります。

ここまで説明してくると読者の皆さんは一体１つの関数を何行に抑えれば良いのかといった疑問をもたれると思います。参考文献やリサーチペーパーによって適切な行数はかなり異なります。一般的に知られているのが、Thomas Plum[5]著の「C Programming Guidelines」で述べられている１つの

関数を「50行」に抑えるという考えです。数年前まではこの50行がエディターのスクリーンに収まる限度だというのがその理由でした。その後、Les Hatton博士が、言語に関係なく、モジュールのサイズが約200行の時にバグの発生密度が最も低いという研究結果を発表しています。McCabeの制御複雑度と Hatton博士の研究成果の間をとって、150行程度から始めたら良いでしょう。

前述したように、完成してしまったソフトウェアの複雑度は高く、もしテストができない場合は最悪です。従って、毎日複雑度をモニターし、モジュールの複雑度がプリセットした値に近づいてきた時点でコーディングをやめ、その理由を分析していくアプローチが良いでしょう。関数が大きくなってしまう理由が設計に起因している場合が往々にしてあります。

毎日のモニターを可能にするのがQACなどの自動化されたツールです。モニターをするだけでなく、CやC++のエラーを検出し、全てのソフトウェア・メトリックスを提供します。またQACからは、CSVファイルが出力されるため、スプレッドシートに変換してデータを活用することができます。

関数	経路複雑度	マイヤーズ	ネストの深さ	コード行数	静的パスカウント
Wnd_message	103	35	11	589	500000000
Wnd_create	46	13	7	270	264241200
Wnd_move	24	5	4	96	165888
Wnd_destroy	27	6	4	146	93696
Msg_handle	23	15	3	61	86016
Msg_receive	21	6	3	93	77760
Msg_transmit	37	1	3	36	20480
Wnd_zoom	20	4	3	45	13056
Msg_key	10	0	2	18	96
Msg_mouse	13	6	6	65	80
Msg_queue	9	3	2	51	72

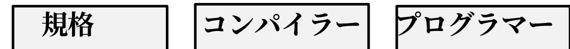
上記はQACからの典型的なメトリックス・データをスプレッドシートに変換した例です。情報は「静的パスカウント」でソートしてあります。プログラマーやマネージャーは、この情報から二つの関数「Wnd_message」と「Wnd_create」の少なくとも4つのカテゴリーで高い数値が示されていることがわかります。実際これらの２つの関数は恐らくテストが不可能でしょうし、コードを書いた本人さえ理解するのが難しいでしょう。結果的に、これらの2つの関数をどうにかしない限りプロジェクト全体に影響が出てしまいます。

このようにプログラマーや特にマネージャーがコーディングの初期の段階で問題のある箇所をピンポイントでモニターすることによって、バグをなくし、プロジェクトを予定どうりに終了することが可能です。

信頼できるコンパイラー

プログラマーがコーディングをしている時、少なくとも1日1回はコンパイルをします。しかし、問題は何のためにこのコンパイルをするのかです。実行プログラムを作成するためですか。自分のコードが正しく書かれているかどうかのテストをするためにコードをコンパイラーにかけているプログラマーがたくさんいると思います。おそらく簡単な構文エラーやタイプミスを見つけたり、文法上自分が自信のない所などをチェックしているのではないでしょうか。例えばポインターからポインターへの参照や多次元の配列内の１つのエレメントへのアクセスなどです。

この場合プログラマーは「コンパイラーの持つC言語の知識」に依存しています。しかし、コンパイラーからの情報は実際のC言語情報にかなりフィルターがかかったものです。



フィルターがかかった規格書の情報

みなさんの中で「ＡＮＳＩに完全に合致」という広告をしているCコンパイラーを見たことがありますか。おそらくないと思います。理由は、コンパイラーがエクステンションを持っていると、もはやそのコンパイラーは「ＡＮＳＩに非合致」というカテゴリーに分類されてしまうからです。もちろん「ＡＮＳＩに非合致」などと言ったら誰もそのコンパイラーを買いませんから、コンパイラーメーカーは「ＡＮＳＩ準拠」といった表現をしています。ＡＮＳＩの規格には「準拠」という言葉はどこにも定義されていません。辞書で「準拠」をひくと「合致」とほとんど同じ意味も存在します。「ＡＮＳＩ準拠」と書いてあればコンパイラーは売れるわけです。しかし、これは、実際には何を意味するのでしょうか。問題は、そのコンパイラーがどの程度ＡＮＳＩに従っているかです。

これはプログラマーにとっては驚きかもしれませんが、コンパイラーは静的テスターではなく、またその様な目的に設計されたものでもありません。このような状況をよりはっきりと理解するためにＩＳＯのC言語のプログラミング規格について少し話をします。この規格書はみなさんも入手可能です。Cのプログラミング規格には2つの基本的な目的があります。1つがC言語の定義、そして2つ目がプログラマーとコンパイラーの責任の定義です。

プログラムの信頼性、移植性、保守性などの問題がコンパイラーによって強調されれば問題無いのですが、認証されたＡＮＳＩCコンパラーにさえ問題点があり、たくさんの驚

くような怪しいコードのコンパイルが許されてしまうため、ランタイムになって予期しなかった問題が発生することになります。C言語の場合、問題は次の項目から発生します。

a) 構文エラー

b) 制限事項の違反

c) 正式に特定されていない項目（22項目）

d) 正式に定義されていない項目（97項目）

e) 正式に処理系に依存する動作（76項目）

f) 地域使用の動作（6項目）

g) 偶然定義されていない項目（ISO委員会が定義し忘れた項目）

h) 経験上決定された誤り（定義は良くされているが、プログラマーが依然として間違えるもの）

認証されたコンパイラーには項上記項目のa) とb)を検出することだけが要求されています。言い換えれば、この部分だけがコンパイラーの責任範囲になります。

診断メッセージ

ANSIでは「診断メッセージ」という表現を使っています。コンパイラーが出力しなければならないメッセージのことです。ANSIでは次のように定義されています。

「ANSIに合致した実装（コンパイラー）は文法上のエラーまたはＡＮＳＩの制限事項に関する違反を含む翻訳ユニット（ファイルの意味）に対して少なくとも1つの診断メッセージ（実装側で定める方法によって）を出力しなければならない。その他の場合には診断メッセージを出力させる必要はない。」

このようにコンパイラーメーカーは、コンパイラーからどのように、いつ、どこに、幾つメッセージを出力させるかについて非常に広い自由度を持っています。メッセージ数やメッセージの用語などは当然のことながらコンパイラーメーカーによってまちまちです。

認証されたコンパイラー

C言語の規格が存在するからといって実装（コンパイラー）がそれに従っているとは限りません。また、どの程度規格に合致しているのかは分かりません。コンパイラーが実際に規格に合致しているかどうかを保証するために認証スイートが存在します。米国とヨーロッパにある２つの認証スイートが良く知られています。米国の National Institute of Standards and Technology (NIST) はPerennial Validation Suiteを採用し、英国のBritish Standards Institute (BSI)はPlum-Hall Validation Suiteを採用しています。認証スイートは困難で認証を受けているコンパイラーの数は限られています。Perennial Suitesで認証されたコンパイラーはNISTから発行されている「Validated Products List （認証製品リスト）」に掲載されています。

もう１つの問題がフローティング・ポイント（浮動小数点）の処理です。Ｃ言語の規格では、現在Numerical Extensions Groupというところでフローティング・ポイントの処理の仕方について検討中のためまだ定義されていません。どのようなコンパイラーやマシンの組み合わせでもフローティング・ポイントの計算の仕方を評価できる練習プログラムがいくつかあります。最も良く知られているのがパラノイアです。paranoia send paranoia.c from paranoiaというメッセージをインターネットのnetlib@research.att.comに送ることによって、Cプログラムとして入手可能です。作成したソースコードのコンパイルランが必要に応じて可能です。これによって次のような事項がはっきりします。

- 乗算、除算、減算に対するガード・ディジットが十分かどうか
- 乗算、除算、減算、加算、平方根のために算術が切られて正しく丸められているかどうか
- 丸めに対し、ビットが正しく使われているかどうか
- アンダーフローがぶっきらぼうか、徐々におこなわれているか、あいまいかどうか
- 無限が存在するかどうか
- 比較と減算とに矛盾がないかどうか

最後になりますが、コンパイラーはソース・コードをコンパイルするためのものであり、エラーを犯しているプログラマーであるみなさんに対して警告を発するものではありません。みなさんの道具箱の中に、QACのような品質を測定するためツールを備えてください。

ミススペルのバグ

ソースコードを書いている時にプログラマーが何かをミスタイプすると大体はコンパイラーがこれを捕捉します。例えば、キーワードのスペルが間違っていたり、セミコロンが抜けていたりすると、コンパイラーは何かしらの文法エラーを知らせてきます。しかし、コンパイラーが見逃してしまうミスタイプがあります。次のコードを見てください。

```
void foo(int n)
{
  int m;
  switch (n)
  {
  case 1:  m = 2; break;
  case 2:  m = 4; break;
  case 3:  m = 8; break;
  defualt: m = 0: break;
  }
  return m;
}
```

エラーがどこにあるかわかりますか。残念ながらコンパイラーはこの場合何の助けにもなりません。上記のコードを問題無く通してしまいます。このコードをQACでスキャンすると次のようなメッセージが出力されます

"警告(3202) 'defualt' というラベルはこの関数内では使用されていないため取り除くことができます。"

良く見ると分かるように"default" のスペルが間違っています。では、なぜこれがコンパイラーによって検出されないのでしょう。理由は、"case" と "default"はswitch文ではあたかもgotoのようにラベルと考えられてしまうからです。 gotoラベルをswitch文の中に組み込むのは合法なので、スペルの間違っている「defualt」は、コンパイラーによって単に使用されていないラベルとして認識され無視されます。これは、非常に危険なバグで、目視検査によってこのバグを見つけ出すのは非常に困難です。

次のミスタイプも古典的な例です。どちらもコンパイラーから見ると文法的には問題がありません。

```
if (a == b) {
}

if (a = b) {
}
```

どちらのif文が間違っているのでしょうか。

次の例はお客様のコードの中によく出てくるケース

`i=j`と書こうとしたが、`i==j` と書いてしまった

これらに対してコンパイラーは何も警告を発してくれません。QACは上記の全てのケースに対して警告メッセージを出力します。

コード解析センター

お客様がより高品質のソフトウェアを開発して頂けるよう、１９９５年に東陽テクニカにコード解析センターを設立しさまざまなサービスを提供させて頂いております。 次の３つがコード解析センターの主業務です。

- お客様のコードのサンプル解析
- 解析レポートの作成
- コーディングの問題に関するコンサルティング

QACを使用し、お客様からお預かりした5 ~10Kのソースコードを解析します。さまざまな観点からコードを分析しワーニング・メッセージとメトリックス結果をベースにコードの品質を報告をします。解析レポートには次のような内容が含まれます。

- ソース・コードの統計データ
- リスクの高いワーニングの出力
- バグの可能性のある場所の特定
- メトリックス・データ

また、コンサルティングを通じて、お客様が抱える現在のソフトウェア開発環境の問題点の指摘や、グループ・ディスカッションを通じて「どのようにデバッグを行っていったら良いのか」、「社内コーディング基準をどうするか」などについて検討します。さらに、QACをより効果的にお使いいただくことを目的にQACワークショップが弊社にて定期的に開催されております。お客様を訪問してのワークショップの開催も可能ですので、ぜひとも御相談下さい。

連絡先
株式会社　東陽テクニカ
ソフトウェア・ソリューション
〒113-8514
東京都文京区湯島３－２６－９
Tel 03-5688-6800 Fax 03-5688-6900
E-mail : ss_sales@toyo.co.jp