Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Making Sense of Software of Unknown Pedigree

SOUP 開発経路が未知のソフトウェアの活用

Legacy code turns the ideal process on its head

レガシーコード に対する理想的なプロセス

By Mark Pitchford, Dr. Dobb's Journal

9 10, 2009

URL:http://www.ddj.com/development-tools/219700483

Mark Pitchford specializes in software test as a Field Applications Engineer with LDRA.

従来ソフトウエアテストツールは、ベストプラクティスとされる理想的なプロセスに従ってデザインされて実装されるコードに対して利用することを前提に設計されてきた。明確に定義された要件、コーディングスタンダードへの準拠、より良く管理された開発プロセス、首尾一貫したテスト体制など。

レガシーコードの利用は一見理想的に思われる。確かにそれは実製品として年月をかけて証明されている有用な資産であるが、アプリケーションに精通する熟練者の経験則に従い場当たり的に開発されていることが多い。そして近代的な開発プロセスに従うとは限らず、また完全なドキュメントも残されないことになる。

このようなレガシーコードを SOUP（software of unknown pedigree 開発経路が未知のソフトウェア）と呼ぶが、これを基にして開発される製品が顧客あるいは組織の改善戦略として、スタンダード準拠することが求められることが増えている。SOUP を活用して最新のスタンダードに準拠し、追加の機能を開発することは、大きな課題である。

SOUP に潜む危険

SOUP を用いた多くのプロジェクトでは、一般に十分なカバレッジが得られていない。システムの機能テストに頼っていて多くのコード領域が未実行のままである。そのようなコードが運用されると、異なったデータ・環境により未実行であったパスが実行されて予期せぬ結果をもたらすことになる（Figure 1)。

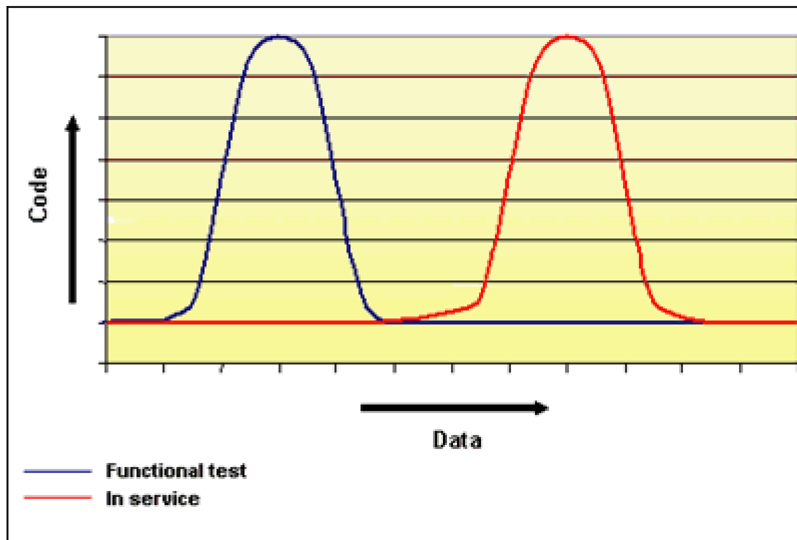

Figure 1: 従来式の機能テストでは多くのコード領域が証明されない。青線は従来式のテストで実行されるコード領域を示し、赤線は製品が現場で運用されて実行される箇所を示す。

欧州 European System and Software Initiative の PET レポート(Performance of Errors through experience-driven Test)による調査では、出荷されるコードの多くがたくさんのエラーを抱えていることを言及している。 調査結果は静的解析や動的解析などの新しい手法により検出されるバグは、機能テストされ継続的にリリースされてきたような製品でも非常に多いことを証明している。

多くの場合、新しい製品開発のベースとなるレガシーコード(コード資産)は現場で運用されてきたことで十分にテストされていると主張される。しかしながら現場で運用しても、特定箇所のコードが実行される状況が揃わないことなど珍しいことではない。要するにそのようなアプリケーションへの機能テストや運用実績だけでは多くの未実行パスが存在することになり、システムの機能テストの想定を上回るような現場状況に遭遇した場合持ちこたえることができない。

開発進行中のレガシーコードに対して後の改訂や新しいアプリケーションの追加が求められると、今まで未実行であったコードパスが全く未知なデータと合わさって実行されないとも限らない(Figure 2)。

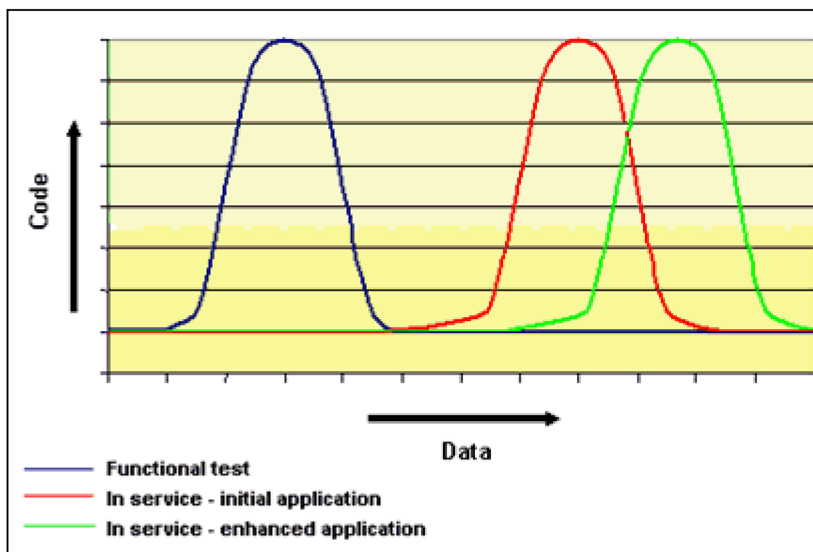

Figure 2: 機能テストされた上に出荷先で実行されていたとしても未確認の実行パスは存在し得る。コードに対する拡張は未知のデータと処理の組合せを生じることになり、結果未だかって実行されてこなかったパスに出くわすことになる。

このような状況は開発チームをずっと前に離れた個人により開発されていたものである場合厄介である。とりわけドキュメント化が今日求められるような高い標準でなかった時分に開発されていたレガシーコードなど。

市況のプレッシャーにより一から書き直すことは問題外とされる場合、どのような選択肢が可能でしょうか？

SOUP の動的な振舞いを静的に解析

レガシーコードの全てのルートを実行することは困難な作業であり、abstract interpretation（抽象解釈、抽象実行） のような数学的手法を用いてプログラムの全ての実行可能性を評価する多くの静的解析ツールが存在する。それらはランタイムエラー（バッファオーバフロー、ゼロ割、ポインタエラーなど）の無いことを証明し、ランタイムエラーとなり得る箇所を特定する。

そのような主張は魅力的に聞こえる。特にコードへの変更や理解すら必要無く問題を分離できるという都合のよい事実だけを信じるなら。

なるほどシンプルなコード領域に対してコードへの理解無く多少の問題を検出して修正できることは事実ではあり、ツールの評価段階などで大きな励みとなる。そして、そのような方法でソフトウエアが適切に堅牢になると心引かれることであろう。

しかしながらコードが複雑になると、そのようなツールは推測に頼る部分が多くなり、誤検知が多くなる。そしてこの本質的にアプリケーションの最も複雑な箇所に相当し、ソースの最もわかりにくい部分のまさしくコード内を確認しエラーを除去しなければならない

このような事実をさしおいたとしても、これらツールではコードが機能を満たしていることの証明は出来ない。コードが堅牢であることが証明できても、機能上の欠陥証明には適さない。

静的・動的解析を組み合わせて活用する

そのような静的解析が万能であることが妄想ならば、従来からの静的解析と動的解析の組合せを SOUP 特有の課題に活用できるだろうか？

従来からのフォーマルなテストツールで求められる手順は下図（Figure 3）の通り。

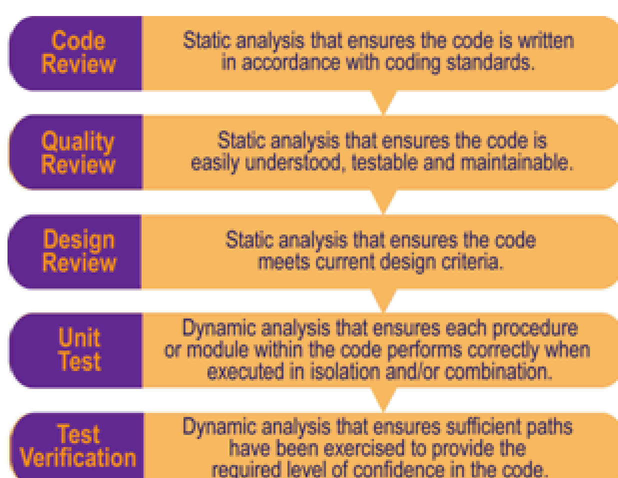

Figure 3: 従来からのテストツール適用手順。各段階で、コードにエラーが無く、実行され、仕様に従っていることを証明するためにテストされる。（上から順に、コードレビュー：コーディングスタンダードに準拠していることを静的解析、クオリティレビュー：コードを可読性、テスト容易性、メンテナンス性などの尺度で静的解析で評価、デザインレビュー：コードがデザインに基準に準拠していることを静的解析、ユニットテスト：各プロシジャーやモジュール単位で正しく動作することを分離・コンビネーションなどで動的に解析、テストベリフィケーション：コードの一定の評価を得るために十分なパスが実行されることを動的に解析/カバレッジ解析）

SOUP を基に開発をすることになった場合、コードが既に存在することと、そしてそれが唯一（少なくとも現状の）の詳細なデザインドキュメントとなることから、より実際的な手法を取ることが必要となる。言い換えると、既存コードは事実上その他のドキュメントよりもシステムの機能を明示している。既存コードを拡張するに当たって、追加機能の結果やコーディングスタンダードを満たすための書き換えによって、既存機能がうっかり変更されないことが肝要である。

それ故に課題は、異なるテスト手順で SOUP の効果的な拡張開発を支援するテストツールのビルディングブロック（機能）の確保と活用である。

最新のテストツールスイートを活用して SOUP を強化

より実際的なアプローチを決定するに当たり、いくつかの根本的な要件が問われる：

- 既存コードに対してどの程度の理解が得られるのか？
- 過去採用されていない新しいスタンダードへの準拠に対して既存コードを改修する必要があるか？
- どのようにして変更の影響を受けるコード領域をテストすることができるか？コードカバレッジのレベルはコードを十分にテストしたことの証明となるか？
- モジュール化が必要なコードはどうするか？新しいコード領域をどうやって証明するか？
- どのようにして改訂されたコードが、基となる SOUP の機能と同等であるかの証明するか？

Improving the Level of Understanding　理解を高める方法

システムの視覚化機能は近代のテストツールでは非常にパワフルであり、特定コード領域、関数間（クラス間）、アプリケーション、システムワイドなどあらゆる範囲で活用できる。静的コールグラフでは、アプリケーションやシステム構成の階層図を、静的フローグラフではプログラムブロック間のコントロールフロー得ることが出来る（下図 Figure 4）　このような色分けされる図は、SOUP の解析・理解に大いに役立つ。

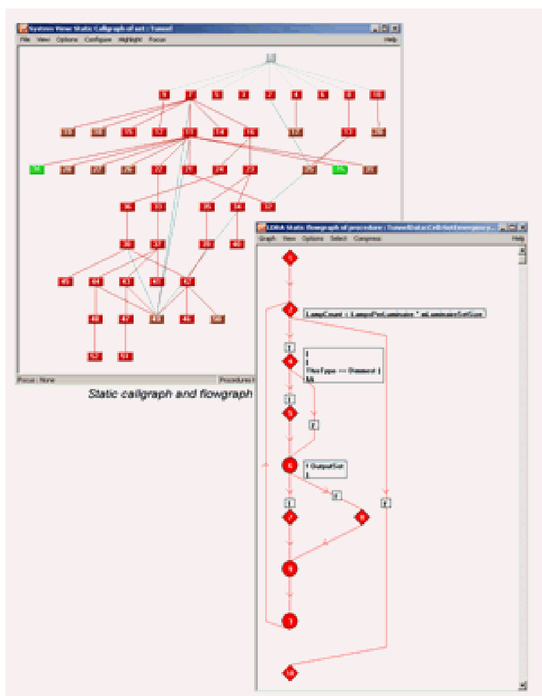

Figure 4: 静的コールグラフとフローグラフ。コード構成、ロジックをグラフィカルに展開。

このようなコールグラフとフローグラフは、コード内で採用される全てのパラメータ、データオブジェクトの包括的な解析の一環にもなる。そして自動ヘッダコメント生成機能や、ソフトウエア部品とそれらの関係に相関する問題に関するデータフローレポートにより補足される。これら情報により、機能拡張に伴い影響を受けるプロシジャーやデータの構造を分離して理解できるようになることは不可欠である。

Enforcing New Standards　新しいスタンダードの施行について

既存の SOUP をベースにした新しい開発では、何らかのスタンダード（社内・産業界・国際的な）への準拠が求められるようになることも考えられる。コードレビュー解析は修正が求められるルールの逸脱をハイライトすることが出来る。

SOUP に対して昨今のコーディングルールのフルセットを施行することは、あまりに厄介であり部分的な妥協が望ましい。そしてユーザ定義のルール集を適応しながらも、それらサブセットが国際的なスタンダードにクロスリファレンスできると良い（Figure 5）

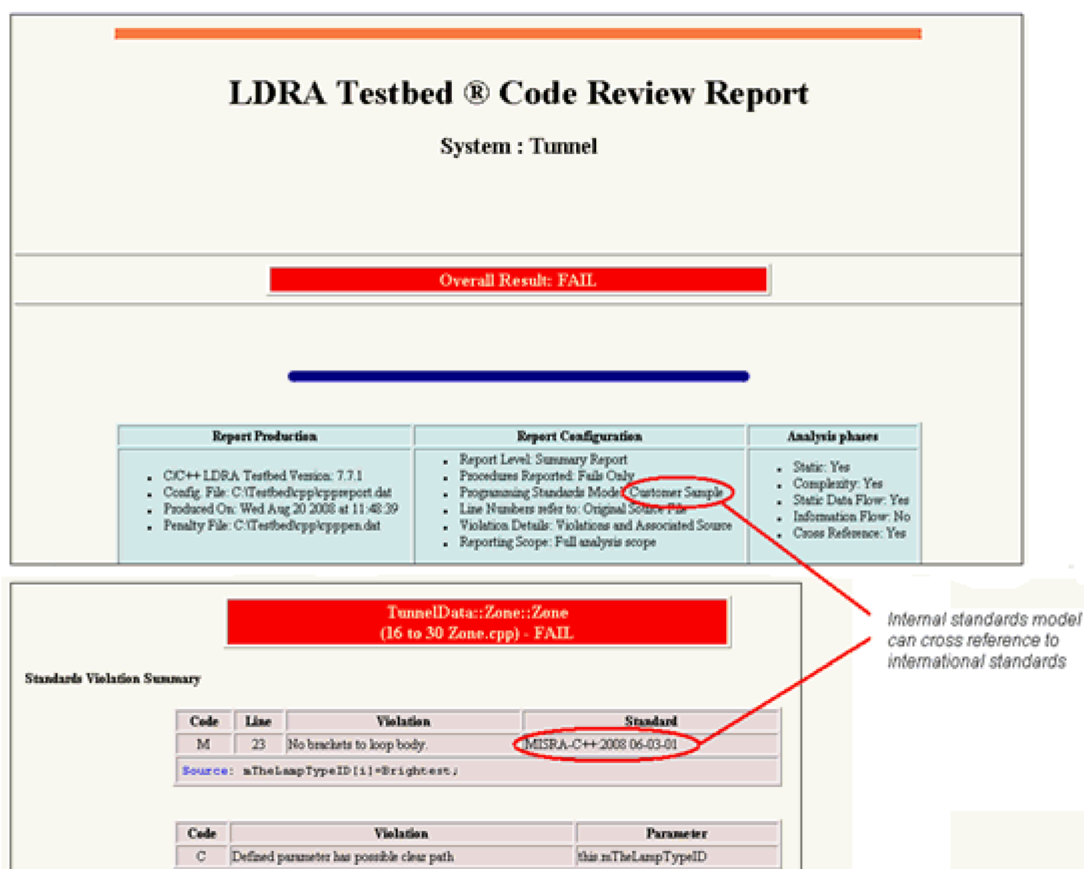

Figure 5: 国際的なコーディングルール（この図では MISRA-C++ 2008）が社内スタンダードとクロスリファレンスされる

レガシーコードが継続して対象となる場合には、長期的には厳密なスタンダードに準拠することが、概して望まれるようになる。実用的なのは相対的に緩やかにコーディングルールを採用すること。避ける必要の高い違反のみを排除する。そして新しいリリースごとで更なるルールを追加して継続的に理想的な状態にする。そうすることで、追加される機能改善によるインパクトを最小の範囲にとどめることが出来る。 あらゆるコーディングスタンダードをサポートする最新のツールでは、下図のように GUI を介してルールを取捨選択できるようになっている。

LDRA Report File Editor - C:\LDRA_Toolsuite\c\creport.dat*

File Edit View Help

Static | Complexity | Data Flow | Cross Ref | Info Flow | Qual Report | Qualsys | LCSAJ | User Def | Section 3 | Section 4 | PDTMCSFVA

Rule Number	Default Strength	Description	Standard	Customer Sample	TBrun Requires	CWE	CERT	JPL	MISRA	SEC-C	DERA	VSOS	MISRA-C 2004	EADS	GJB	CAST	CMSE	HIS	LMTCP	UML	USER-C
186	O	Space missing before or after binary operator.	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐
187	O	Tab character in source.	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☑	☐	☐	☐	☐	☑	☐	☐
188	C	{ or } not on line by itself.	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☑	☐	☐	☐	☐	☑	☐	☐
189	C	Input line exceeds limit.	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☑	☑	☐
190	C	{ ... } contents not indented by *** spaces.	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☑	☐	☑
191	O	Space between function name and parenthesis.	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐
192	O	Static not on separate line in function defn.	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
195	O	Function return type needs a new line.	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
200	C	Define used for numeric constant.	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☑	☑	☐
201	C	Use of numeric literal in expression.	☐	☐	☐	☐	☑	☐	☐	☑	☐	☑	☐	☑	☐	☐	☐	☐	☑	☑	☐

Section 1 read (18 Models) (11 Modifiers)... Sections 2 and 3 read... Section 4 read... Penalty File read.

テストツールはコードを修正することは出来ない。しかしながら、ルールに準拠するためのコード修正をより効率的に支援できる。ドリルダウンでコードレビューレポート内のルール逸脱結果から対象コード行を、エディタを開いて掘り下げていくことができる。

適切なコードカバレッジを得ること

繰返しになるが、プログラムに想定される入力の統計的なサンプルをベースにしたテストでは、一定の値やレンジに依存したテストとなってしまう。運用実績によるコードの証明も結果的には、SOUP への拡張に対する機能テスト実行と同等程度。

このような課題に対しては、構造化カバレッジを活用して入力データで偏ることの無いテストを行えるようになる。

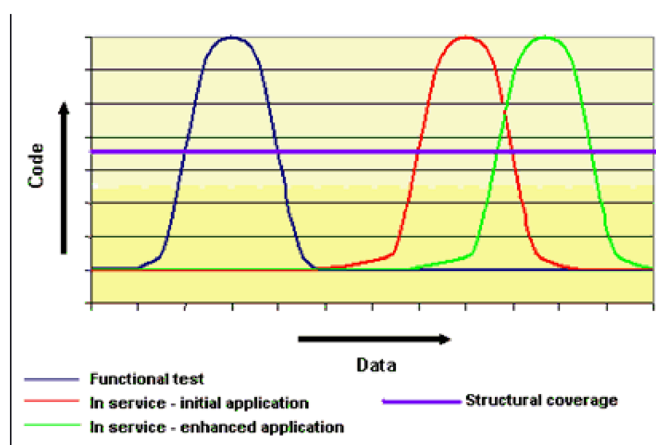

Figure 6: データや機能の変更によってテストされないコードパスを生み出さないことを適切な構造化カバレッジで保証する

システムワイドなテストについて

システムワイドな機能テストでは全パスはテストされないが、多くの箇所をテストすることは明らか。一定機能のソフトウエアには実行を開始するためのロジカルな開始点がある。そしてカバレッジテストツールはソフトウエアのどの箇所が実行済みであるかを特定する手段を提供し、注意を引くためにハイライトする。

ツールはテスト対象コードの写しを用いて、追加の関数コールを埋め込んで（インスツルメンテーション）実行されるパスを特定する。そしてホワイトボックステストを介して、どれだけのコードが実行されたかの統計を取る。カバレッジ結果によって色分けされるコールグラフとフローグラフはテストされるコードへの理解を深め、追加のカバレッジを得るために必要なデータを明確にすることができる。

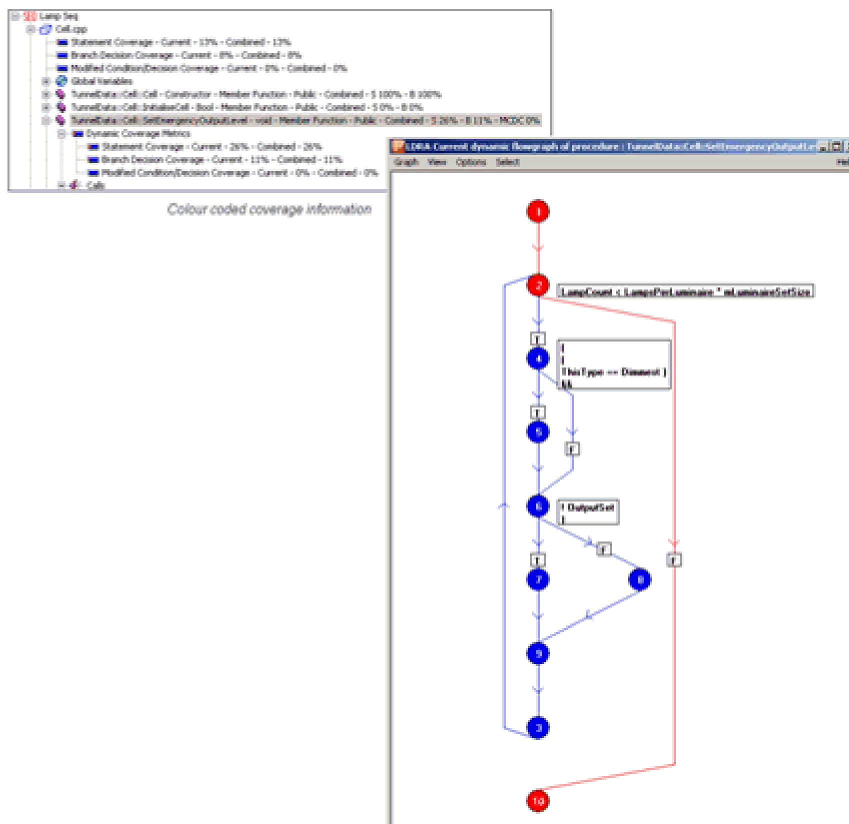

Figure 7: 色分けされたコードのグラフ表示から未実行コードパスを特定

Unit Testing　ユニットテスト

必然的にシステムワイドなテストでは証明されないパスが存在する。しかしながらコードに対するインスツルメンテーションはシステムワイドでもユニットテストでも採用できるので、システムテストで証明されないコード領域をユニットテストで試みることができる。これは例外処理や防御的なコードなどの一般的な状況下では実行できないようなコードに対しても活用できる。

ユニットテストでは各ユニット（セクション）コード機能の正しさを分離して証明できる。ただ単一関数/プロシジャーであれ、アプリケーション内の単一サブシステムでもテストにはマニュアルでテストハーネスを構築するための工数は考慮すべき課題であり、またテスト担当者に相当レベルの経験と知識が要求される。

近年のユニットテストツールなら容易な GUI ツールを介して、自動的にテストハーネスコードを生成し、入出力へのデータ設定を支援することでそれら課題を最小限にできる。そのようなツールによって従来から頭痛の種であった C++のプライベートメンバー変数へのアクセスなども解消される。そして実行はホスト環境でも実ターゲット環境でも行える。それら一連のテストケースは保存され、定期的に自動実行させて（夜間を通すなど）、継続中の開発によって、実績のある機能が悪影響を受けることの無いことを確認できる。

モジュール化に対処すること

SOUP のアプリケーションによっては ユニットテストやサブシステムといった用語は、にわかにイメージしにくい。構造化やモジュール化によって各々の重みは下がるものの、機能テストやサブシステムなどコード構造へのテストは課題となる。しかしながらユニットテストツールは大変柔軟に活用できる。ユーザによりテストが必要とされるソースコードだけを包括して、テストケースを実行するためのハーネスコードを生成できる。システム内の特定機能の振舞いをテストするために、もし大きな領域のコードを含める必要がある場合は、それもできる。

それは ”寝た子を起こさない（letting sleeping dogs lie）” といった目的に適うが、しかし長期的なゴールとして事態を改善したい場合には、インスツルメンテーションされたコードを活用して、異なる入力パラメータにより関数内のどのパスが実行されたかを理解することができる。分離した状態でも、広範に渡るコールツリー状況に於いても。

正しい機能動作を保証する

実用的な観点で SOUP ベースの開発に最も重要な一面は、ソフトウエア機能の全ての側面が、扱われるコードやデータの変更によっても期待通りであることを確かにすること。スタンダードへの準拠であれ追加機能であれ、コードへの変更が生じる場合に、最も重要なのは機能が不用意に変更されないこと。

確かにユニットテストを徹底活用して、テストツールの支援を介してそのような確認は出来るが、それは追加の工数・予算を伴うことになる。しかしながら課題は、特定範囲内で動作している各機能を検査しないこと。

ここで自動テストケース生成は解決の鍵となる。静的にコードを解析して、テストツールは高いパーセントのパス領域を実行するテストケースを自動生成する。関数を実行するために入出力データが自動生成され、それらは後の活用に備えて保存できる。

そしてそれらはバッチ化されたリグレッションテストに継続的に活用され（夜間、週単位など）、それら同じテストによって開発中のコードが想定外の動作をしないことを確認できる。これらリグレッションテストはオリジナルソースコードの機能に対するクロスリファレンスを自動提供することとなる。

これにより拡張されたソフトウエアのリリースを保証し、クライアントは新機能の恩恵と改善された堅牢性を不安無しに得ることが出来る。

まとめ

テストツールはフォーマルな開発プロセスに沿って手順どおりに活用されるのが理想的であるが、商用製品ではレガシーな SOUP（Software of Unknown Pedigree）コードを基にした派生・差分開発が現実となっている。そのような状況でもテストツールを実践的に用いれば、レガシーな SOUP コードを目的に適った信頼できるソースコードとして効果的に活用できる。

富士設備工業株式会社 電子機器事業部 http://www.fuji-setsu.co.jp
〒591-8025 大阪府堺市北区長曽根町1928-1 Tel: 072-252-2128