7A-3

オープンソースツールを活用した単体試験技法とその事例報告

～メトリクスとカバレッジによる計測駆動型実装の提案～

川崎　将人†　　上野　浩一郎‡

三菱電機株式会社 情報技術総合研究所†‡

1. はじめに

ソースコードのメトリクス計測ツールやコードカバレッジ計測ツールが、オープンソースツールとして利用できるようになり、大規模な開発プロジェクトでも全ての実装者の計算機で計測ツールを利用できるようになった。本論文では、実装および単体試験時にメトリクス計測およびカバレッジ計測を随時行う新たな開発技法を提案し、その評価を行い、開発効率が改善されることを示した。

2. 背景

これまでに、ソースコードのメトリクス(例えばソースコードのステップ数や複雑度など)を計測し、リファクタリング([1])を行うべき箇所を発見するという手法([2])や、コードカバレッジを計測し、単体試験コードを充実させることによってソフトウェアの品質を向上させる手法は考えられてきた。これらの手法を適用する際、メトリクス計測ツールやカバレッジ計測ツールは従来は商用製品が中心であったため、ライセンスやコストの制約により、大規模な開発プロジェクトでは全ての実装者の計算機に計測ツールをインストールできなかった。このため、メトリクス計測およびカバレッジ計測は、実装後および単体試験コード作成後にシステム全体で一括して行われていることが多かった。しかしそれでは、改善箇所は広範かつ大量になり、計測結果を有効に活かすことができなかった。

最近ではこれらの計測ツールはオープンソースのものでも十分に利用できるようになり([3],[4])、全実装者の計算機にインストールできるようになり、実装や単体試験時にも計測ツールを随時利用できるようになった。

なお、テスト駆動開発([5])では実装中に随時単体試験を行うが、メトリクス計測やカバレッジ計測を随時行う手法については言及されていなかった。

3. 技法の提案

本論文では、メトリクスとカバレッジによる計測駆動型の実装方法を提案する。

本技法の流れ図を図 1に示す。本技法では、実装中にメトリクスを随時計測し、必要であれば計測終了直後にリファクタリングを行う。これにより、一度リファクタリングを行って改善されたクラスに基づいて新たなクラスの作成を行うため、新たなクラスは比較的メトリクス値が閾値内に収まりやすくなり、システム全体として所定のメトリクス値に収めるためのリファクタリング工数の低減が期待される。また単体試験では、試験対象クラスごとに単体試験を実施してカバレッジを計測し、バグが集中しているならばカバレッジ率を上げるように単体試験コードを追加して試験充実化を図る。これにより、より早い段階でバグを発見できることが期待される。

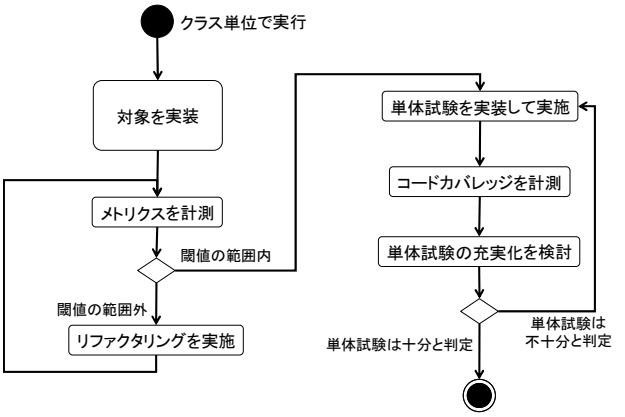


図 1 提案技法の流れ図

従来技法と本技法の違いを表 1に示す。

表 1 従来技法と本技法の違い

計測項目	従来技法	本技法
メトリクス	全クラスの実装後	作成できたクラスやメソッドの実装ごと
カバレッジ	全クラスの単体試験コード作成後	試験対象クラスの単体試験コード作成ごと

4. 本技法の評価

本技法の有効性を示すために、実際の 2 つの開発プロジェクトを比較して確認した。開発 A は本技法で行い、開発 B は従来技法で行うことにした。表 2から分かるように、開発 A および開発 B は使用言語・開発対象・開発規模(クラス数)・開発工期がほぼ同じであるため、比較対象として妥当である。

Unit-Testing Method Using Open Source Tool

† Masato Kawasaki　‡ Kohichiro Ueno

Information Technology R&D Center, Mitsubishi Electric Corporation

表 2 対象開発プロジェクト

	開発 A	開発 B
使用言語	Java	
開発対象	ウェブアプリケーション	
開発規模	25 クラス	22 クラス
全体工期	約 1 ヶ月	約 1 ヶ月
メトリクス計測	実装中随時	実装完了後
カバレッジ計測	行う	行わない

メトリクスとして、クラスの行数とメソッドの行数を使用した。リファクタリングは、閾値の範囲外のクラスやメソッドをメトリクス計測ツールによって検出し、そのクラスやメソッドに対して行った。それぞれの開発において、メトリクス計測に要した時間とリファクタリングに要した時間を計測し、開発終了後に比較した。それぞれの作業の進め方と計測箇所のイメージを図 2および図 3に示す。比較には、メトリクス計測を行った時間とリファクタリングを行った時間の総合計(以下リファクタリング時間と呼ぶ)、つまり、図 2の $\sum_{i=1}^{n} ti$ と図 3の t を使用した。また、それぞれの開発において、単体試験工数と発見バグ数を記録し、その推移を比較した。

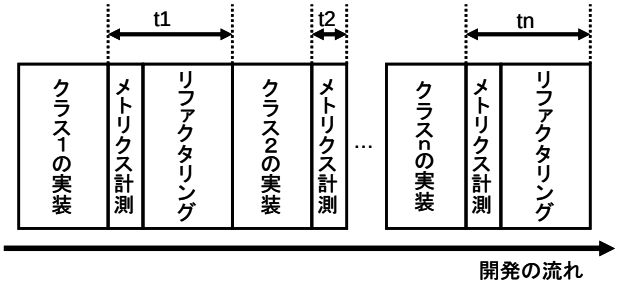


図 2 提案技法の計測箇所のイメージ

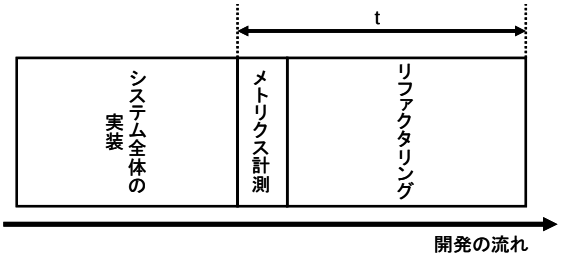


図 3 従来技法の計測箇所のイメージ

また、カバレッジ計測については、開発 B では行っていない。

5. 評価結果

開発終了後にまとめた、リファクタリング時間および、単体試験工数と発見バグ数の関連を表 3および図 4に示す。

表 3から、リファクタリング時間について、提案技法を適用した開発 A の方が短くなっている。つまり、期待されたリファクタリング時間の縮小が示された。

表 3 リファクタリング時間の比較

	開発 A	開発 B
リファクタリング時間(分)	200	480

次に、図 4について考察する。工数 500 分くらいまでは、開発 B の方が発見バグ数が多くなっているものの、開発 A の方が線形であるため、工数の見積もりや管理が容易であると言える。工数 1000 分以降は、開発 A の方が発見バグ数が多くなっていることから、カバレッジを計測することによって効率的にバグを発見できていると分かる。

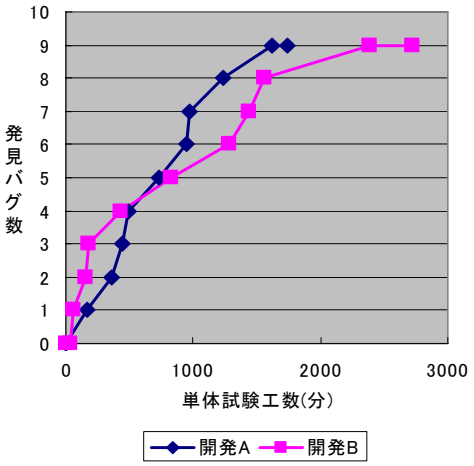


図 4 単体試験工数と発見バグ数の関連

6. まとめと今後の課題

本論文では、実装中にはメトリクスを随時計測し必要に応じてリファクタリングを行い、単体試験時にはカバレッジ計測を利用して効率的にバグを発見する技法を提案した。そして、実際の開発に適用することによって、その有用性を示した。今後は、大規模な開発に適用するとともにサンプル数を増やすことによって、より詳細な検証を行っていく。

7. 参考文献

[1] マーチン・ファウラー,「リファクタリング プログラミングの体質改善テクニック」, ピアソン・エデュケーション, 2000

[2] 藤原巧ら,「ソフトウェアメトリクスを用いた Java プログラムに対する『不吉な匂い』の検出」, 情報処理学会第 66 回全国大会, 2005

[3] 「Eclipse Metrics plugin」, http://sourceforge.net/projects/metrics/

[4] 「djUnit-dJWiki」, http://works.dgic.co.jp/djwiki/Viewpage.do?pid=@646A556E6974

[5] KENT BECK,「テスト駆動開発入門 TDD:TEST-DRIVEN DEVELOPMENT BY EXAMPLE」, ピアソン・エデュケーション, 2003