呼び出し関係を用いた単体テストおよび静的検査の可視化手法の改善とその評価

Improvement of a Visualization Technique for Unit Testing and Static Checking using Caller-Callee Relationship and Its Evaluation

武藤 祐子* 岡野 浩三† 楠本 真二‡

Summary. Software visualization has attracted lots of attention. We have already proposed a hybrid method which visualizes coincidence between specification and implementation from two aspects: static checking and ordinal testing by test suites. In this paper, we propose Priority Layout to emphasize important classes, and implemented our method into a tool. We have evaluated the time in finding bugs at source code and test cases between using Priority Layout, ISOM Layout and uncomplicated tables instead of graphs. As a result, the time in finding bugs by proposed graph are half as much as by the table.

1 はじめに

近年，ソフトウェア規模の増大によって，ソフトウェアの可視化の必要性が高まっている．可視化手法は次の二つに分類できる．ソフトウェア構造そのものを可視化する手法と，ソフトウェアから得られたメトリクスを可視化する手法である．

可視化対象の粒度として，コード単位，オブジェクト単位，クラスファイル単位等が挙げられる [1]．オブジェクト指向プログラミングにおいて，クラス単位の可視化はよく用いられる方法であり，文献 [2] ではクラス間の関係の可視化を行っている．

本稿では，国際標準化機構の定める 6 つの品質特性のうち，機能性に着目し，ソフトウェア品質の向上を目的としたソフトウェア可視化手法の改善および評価を行った．

機能性を測る手法として，単体テスト，静的検査，モデル検査等が挙げられる．単体テストとは，与えられたモジュールがテストケースを満たすかどうかによって，そのモジュールが仕様を満たしているか検査する手法であり，ソフトウェアテストの初期段階に位置する．問題点として，テスト結果は用いられたテストケースの品質に依存することが挙げられる．テストケースの品質のメトリクスとしてコードカバレッジがあり，これはテストケースが対象ソースコードをどの程度の割合で実行したかを計測する方法である．テストケースのコードカバレッジが低い場合，充分にテストが行われていないと言える．

一方，静的検査やモデル検査はソースコードを実行することなく，プログラムやそのモデルを検査する．よく用いられる静的検査ツールとして，ESC/Java2 がある．このツールは Design by Contract の考え方に基づき，JML (Java Modeling Language) という Java 用の契約記述言語の付加されたソースコードを入力すると，ソースコードが契約を満足するかどうかを判定する．また，検査結果は JML によって書かれた契約の質に依存し，契約をごく一部しか書いていない場合は，検査が行われるソースコードも一部となる．従って契約の質の推定やその向上が望まれているが，現在広く普及している手法は存在しない．

我々の研究チームでは単体テストと静的検査の結果に対する可視化手法を提案している [3]．この手法では，単体テストおよび静的検査の通過率を定義し，その値を円グラフで表し，プログラム全体の構造を重み付き有向グラフで表現する．通過率はプログラム中のクラスに対して算出され，クラスがノードに対応し，クラス間の

*Yuko Muto, 大阪大学大学院情報科学研究科

†Kozo Okano, 大阪大学大学院情報科学研究科

‡Shinji Kusumoto, 大阪大学大学院情報科学研究科

呼び出しをエッジとする．

本稿では，この提案手法を改良し，さらなる評価実験を行った．重要なクラスを強調するために，Priority Layout というグラフレイアウトの提案を行った．評価実験では，学生にツールを用いてプログラムとテストケースのバグを探してもらった．その際，グラフを用いた本手法と，グラフの代わりに表を用いた手法を比較した．表を用いた場合と比較して，グラフを用いる本手法ではバグ発見に要した時間が半分であるという結果が得られた．

以降本稿では，まず 2 節で関連研究を紹介し，3 節で提案手法を説明する．評価実験について 4 節で内容と結果を述べ，5 節で議論する．最後に 6 節でまとめる．

2　関連研究

2.1　ソフトウェアの可視化

オブジェクト指向プログラミング言語に対する動的な振る舞いの理解の補助を目的として，GraphTrace [2] という手法が提案されている．この手法では，実行時のクラスの継承関係やメソッド呼び出し等の関係を木構造で表現しており，ケーススタディとして可視化が有効な例を示している．また，論文 [1]，[4] では，コンポーネントに対して静的および動的の両方の観点から可視化を行っている．

これらの手法ではソースコードの情報と実行時情報のみを対象としており，テストカバレッジのようなその他の情報は可視化対象ではない．このように，現在，ソフトウェアの構造に対する可視化は複数存在するが，ソフトウェア品質について可視化を行う研究は少ない．

2.2　単体テストと静的検査の組み合わせ

単体テストと静的検査を組み合わせた手法として，Check'n'Crash [5] が提案されている．この手法はバグを自動で発見することを目的としており，まず ESC/Java2 で静的検査を行い，その反例を用いて単体テストのテストケースを生成する．そうすることで問題のありそうな箇所を効率的に見つけることができる．しかし，バグの発見が目的であるため，バグが見つかった箇所以外に関しては保証がない．ESC/Java2 は健全でも完全でもないため，静的検査で問題の検出されなかった箇所に関してはテストが行われず，重要な欠陥があっても見過ごすこととなる．

上記の研究では，静的検査を行った後に単体テストを行っている．逆にテストの後に静的検査を行う研究として [6] がある．静的検査は対象プログラムが大きすぎると完全な検査ができない．一方，テストではすべてのバグを見つける，ということはできない．そこで [6] では次のような手法を提案した．まずテストを行いカバレッジデータを得る．それからテスト済みパスが分かり，テストされなかったコードに対して静的検査を行う．これにより静的検査の対象となるコードが減るため，大きなプログラムにも適用することができ，適用可能範囲が広くなる．この手法は，テストは通過するが静的検査では見つかる欠陥を検知することができない問題がある．前述の通りテストで全てのバグを発見できるわけではない．テストで見過ごされたバグに関してはこの手法では検知できず，課題が残る．

3　提案手法

提案する可視化手法を述べる．ここで Priority Layout が新規のものである．

3.1　可視化手法の概要

可視化対象プログラムを重み付き有向グラフで表現する．このグラフにおいて，ノードとエッジはそれぞれクラスと呼び出し関係を表している．ノードは円グラフになっており，単体テストおよび静的検査の通過率を示す．エッジの太さはエッジの重みを表し，太いエッジほど呼び出し回数が多い．さらに，重要なクラスを強調

するために，新たに Priority Layout を定義する．

3.2 通過率

以下の 4 つの観点から質を観測する．(1) テストケースの品質，(2)JML で書かれた契約の品質，(3) 単体テストの結果，(4) 静的検査の結果である．今回提案する通過率では (1)，(3) および (4) を対象とした．

単体テストにおける通過率として命令網羅率を用いる．この値を算出するために djUnit を用いた．djUnit の出力はソースコードのコード網羅率のみを表し，JUnit のようにテストメソッドが期待通りの出力をしたかどうかは考慮しない．しかし djUnit と JUnit を併用することで (1) と (3) を共に評価できる．

静的検査における通過率として ESC/Java2 による検査結果を用いる．クラス A の持つコンストラクタおよびメソッド数を $M(A)$ とし，そのうち ESC/Java2 で妥当であると判定された数を $M_{passed}(A)$ とする．このときクラス A の静的検査の通過率 $C_s(A)$ を

$$C_s(A) = M_{passed}(A)/M(A) \tag{1}$$

と定義する．

3.3 Caller-Callee Relationship

クラス間の呼び出しについて次のように定義する．メソッド m_1 がメソッド m_2 の記述中で呼び出しとして出現しているときに，メソッド m_1 がメソッド m_2 を呼び出すと定義する．クラス A の持つメソッドがクラス B の持つメソッドを n 回呼び出す場合，クラス A はクラス B を n 回呼び出すと定義する．なおコンストラクタについてもメソッドと同様に扱う．この際の呼び出し回数 n はソースコード中に登場する回数である．

3.4 Priority Layout

重要なクラスを強調することを目的とした Priority Layout を提案する（図 1 参照）．一般的なグラフレイアウトとして円環レイアウトや ISOM Layout [7] などが存在するが，これらはノードやエッジが重ならないことを主目的にしている．本手法では，単体テスト結果および静的検査結果のグラフを見比べるために，クラスやクラス呼び出しの性質を反映するレイアウトが必要である．一般的に，プログラム全体に与える影響が大きいクラスは，それ以外のクラスよりも注意を向けられるべきである．従ってソフトウェア中の多くのクラスから利用されているクラスは重要であると考えられ，呼び出し関係から重要度という評価値を設定した．重要度および Caller-callee relationship を用いて，より重要なクラスを示唆する Priority Layout を定義する．

クラスに対して以下のように重要度を定義する．グラフ $G = (V, E)$ の頂点に対して重みを計算する．頂点 v_i から v_j への辺を e_{ij} と定義する．頂点 v_i から v_j への呼び出し関係が存在する場合は $w(e_{ij}) = 1$ とし，呼び出し関係が存在しない場合は $w(e_{ij}) = 0$ とする．$IN(v_i)$ を v_i を終点とする有向辺の集合とする．頂点 v_i の重みを v_i が終点となる有向辺 e_{ki} の重みの総和として，以下のように定義する．

定義 3.1 (重要度)

$$w(v_i) = \sum_{e_{ki} \in IN(v_i)} w(e_{ki}) \tag{2}$$

この頂点の重みを重要度と設定する．またこの評価値はあるクラスの他クラスから呼び出されている数である．

重要度の高いクラスを上部に配置する．重要度が同一のクラスが複数存在する場合は，水平位置は一定間隔をあけてランダムとする．

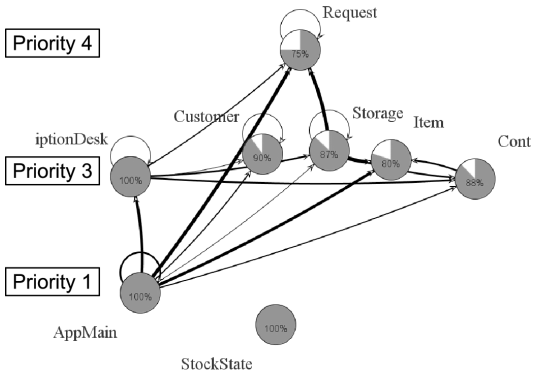

図 1　Priority Layout の適用例

4　評価実験

被験者に提案手法を実装したツールを使用してもらう実験を行った．

4.1　評価方法

評価方法の選択に際して GQM [8] を用いた．その結果，測定するメトリクスとして以下の 2 つを選択した．

- ソースコードにおけるバグ発見に要する時間
- テストケースにおける欠陥発見に要する時間

また，アンケートとして以下の 3 問に答えてもらった．

Q1　二つのグラフを比較する方法はバグ発見に役立ったと思いますか
Q2　Priority Layout と ISOM Layout を比較してどちらが良かったですか
Q3　バグや欠陥の発見に時間が最もかかった要因は何だと思いますか
さらに，ツールを使ってみて感じたこと等を自由に回答してもらった．

4.2　実験の手順

はじめに，被験者に対して提案手法を実装したツールについての説明を行い，練習問題に取り組んでもらった．その後，2 問の問題に解答してもらい，アンケートに記入してもらった．

被験者の解答の手順は以下の通りである．

1. 単体テスト結果のグラフをビューに表示
2. 静的検査結果のグラフをビューに表示
3. 単体テストおよび静的検査の結果を見比べ，バグの潜んでいるクラスを判断
4. メソッドビューを使用し，バグが含まれているメソッドを特定
5. バグが含まれているメソッドについて，その原因を仕様書を元に調べて解答
6. 単体テストのテストケースの欠陥を解答

被験者としてソフトウェア工学を学ぶ学生 10 人に協力してもらった．被験者は全員，今回用いた言語である Java の文法や単体テストおよび静的検査について理解しており，Eclipse の使用経験がある．被験者への配布物は，充分な JML が付加された Java ソースコード，テストケースおよび日本語で書かれた仕様書である．提案手法のグラフと単純な手法を比較するため，単純な手法として Multiple Table View を用いた．このビューはクラス名，静的検査の通過率および単体テストの通過率を持ち，呼び出し関係の情報は持たない．表 1 に示すように，被験者をグループに割り当てた．グループ A および B は ISOM Layout と Priority Layout の比較を行っており，これは本手法において Priority Layout が従来手法をレイアウト面で改善

表 1　グループ割り当て

Group	Method used in Problem 1	Method used in Problem 2
A	ISOM Layout	Priority Layout
B	Priority Layout	ISOM Layout
C	Multiple Table	Priority Layout
D	Priority Layout	Multiple Table

表 2　ソースコードにおけるバグ発見に要した時間

Method	Avg. of Time (min)
ISOM Layout	12.90
Priority Layout	10.79
Multiple Table	21.48

表 3　テストケースにおける欠陥発見に要した時間

Method	Avg. of Time (min)
ISOM Layout	13.66
Priority Layout	14.90
Multiple Table	26.39

できたかどうかを調査することを目的としている．グループ C および D は Priority Layout と Multiple Table View の比較を行っており，呼び出し関係を用いた可視化手法が他の手法より有用であるかどうかの調査を目的としている．

4.3　実験に用いた問題

実験対象として，在庫管理プログラム [9] を基に，バグを加えたものを用いた．このプログラムは 7 クラスで構成されている．

1 問目には，仕様書に書かれた「Item の数量は 0 以上である」という要件を不満足とするバグを加えた．在庫管理される品物を示す Item クラスにおいて，数量を表す amount フィールドが存在し，コンストラクタに amount が 0 以上になるように事前条件が記述されている．しかし，品物を倉庫に追加する addItem メソッドにおいて追加する品物の数量をチェックする記述がないため，Item コンストラクタの事前条件に違反する可能性がある．実際のプログラムでは，AppMain クラスの main メソッドに，Item コンストラクタの事前条件に違反するような addItem メソッドを呼び出す記述がある．このように，仕様書上の 1 つの要件に対して複数のクラスが関与する状況を想定している．静的検査を行うと AppMain クラスの通過率が他のクラスより低い結果となるが，不満足なのは AppMain クラスが呼び出している Item クラスに対する要件であり，このような不具合はクラス間の呼び出し関係を可視化することで発見しやすくなると考える．

なお 2 問目には別のクラスについて同程度のバグを混入した．

4.4　実験結果

表 2，表 3 にソースコードにおけるバグの発見に要した時間およびテストケースにおける欠陥発見に要した時間を示す．なお不正解および時間内に解答が得られなかった場合は除外した．両者ともグラフを用いた場合の回答時間は表を用いた場合の半分程度であった．従って，今回のような例題に関して，単体テストと静的検査の結果を呼び出し関係を表現するグラフを用いて可視化する手法が有用だと言える．

一方，Priority Layout と ISOM Layout の比較に関しては，ソースコードにおけるバグ発見およびテストケースにおける欠陥発見時間において，差は見られなかった．しかしながら，アンケートの Q2 において，Priority Layout の方が ISOM Layout より良かったと回答した人が多かった．被験者の主観としては Priority Layout の方が良いという結果であるにもかかわらず，2 つのレイアウトにおける回答時間に差が出なかった理由として以下が考えられる．アンケートの Q3 においてバグや欠陥の発見に時間がかかった要因は「プログラム理解」であるという回答が多く，プログラム理解に要する時間がグラフを読み取る時間よりも大きかったためだと考えられる．従ってプログラムを用いずにグラフのみを比較する実験が必要である．

被験者から得られた意見として,「クラス間の依存関係などが視覚的に理解できる」

「注目すべきクラスが視覚的に理解できる」等の肯定的な意見があった．改善点として「メソッドの呼び出し関係も可視化してほしい」等が挙がっており，プログラム理解のためにメソッドに対するより詳細な情報が必要であることが分かる．

5 議論

結果の内的妥当性について述べる．被験者はソフトウェア工学を学ぶ学生であり開発の基礎を習得している．また被験者は本ツールをこれまでに使ったことがなく在庫管理プログラムも初見である．従って被験者およびグループ間の能力差はない．

結果の一般性について述べる．今回の実験では問題として小規模なプログラムを用いており問題に関して一般性は高いとはいえない．但し対象プログラムは典型的な在庫管理問題を扱っているため同種の実システムに関して有用であると考えられる．よって今後より詳しく実験をすることが必要である．例えば，大規模な問題に関してグラフを使う場合と使わない場合を比較することが挙げられる．

6 おわりに

本稿では，単体テストと静的検査を用いたソフトウェア品質可視化手法に対して改善と評価を行った．可視化の改善のために Priority Layout を提案し，被験者を用いて評価実験を行った．実験結果から本可視化手法は他の手法に比べてバグ発見に要する時間が短いという結果が得られ，Priority Layout は他のレイアウトと比べて回答時間は差がなかったものの，被験者からは好意的な意見を得た．

今後の課題として，より詳しくバグ箇所の特定を行うことが挙げられる．さらに JML の品質に関してさらなる調査と評価を行う必要がある．

謝辞 本研究の一部は科学研究費補助金基盤 C(21500036) と文部科学省「次世代 IT 基盤構築のための研究開発」(研究開発領域名：ソフトウェア構築状況の可視化技術の普及) の助成による．

参考文献

[1] Welf Lowe, Morgan Ericsson, Jonas Lundberg, and Thomas Panas. Software comprehension - integrating program analysis and software visualization. 2002.

[2] Michael F. Kleyn and Paul C. Gingrich. Graphtrace—understanding object-oriented systems using concurrently animated views. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 191–205, New York, NY, USA, 1988. ACM.

[3] Yuko Muto, Kozo Okano, and Shinji Kusumoto. A visualization technique for software quality using unit testing and static checking with caller-callee relationship. In *Proceedings of International Conference on Advanced Software Engineering*, 2011. (to appear).

[4] Antonio Gonzalez, Roberto Theron, Alexandru Telea, and Francisco J. Garcia. Combined visualization of structural and metric information for software evolution analysis. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pp. 25–30, New York, NY, USA, 2009. ACM.

[5] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 422–431, New York, NY, USA, 2005. ACM.

[6] Vangala Vipindeep and Pankaj Jalote. Efficient static analysis with path pruning using coverage data. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pp. 1–6, New York, NY, USA, 2005. ACM.

[7] Kohonen T. The self-organizing map. *Proceedings of the IEEE*, pp. 1464–1480, 2002.

[8] Victor R. Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, College Park, MD, USA, 1992.

[9] 尾鷲方志, 岡野浩三, 楠本真二. 在庫管理プログラムの設計に対する jml 記述と esc/java2 を用いた検証の事例報告. 電子情報通信学会論文誌 D, Vol. 91, No. 11, pp. 2719–2720, 2008.