日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

オープンソースソフトウェアを対象としたリファクタリングが欠陥混入に与える影響の調査

藤原 賢二　伏田 享平　吉田 則裕　飯田 元

リファクタリングとは，ソフトウェアの外部的な振る舞いを変えることなく内部構造を改善することである．一般に，ソフトウェアの開発が進むと，リファクタリングが必要となる設計品質の低い箇所がソースコード中に増加していく．このような箇所は「コードの不吉な匂い」と呼ばれている．ソフトウェア開発において実施されるリファクタリングは，不吉な匂いを除去することでソフトウェアの欠陥を予防できると言われている．本研究では，リファクタリングが欠陥混入に与える影響についてオープンソースソフトウェアを対象に調査した．そのために，ソフトウェア開発履歴からリファクタリングの実施頻度，欠陥の混入頻度および欠陥の修正頻度を計測した．本稿では，分析に用いた手法と分析結果について報告する．

1　はじめに

ソフトウェアの設計品質を向上させる技術としてリファクタリングがある．リファクタリングとは，ソフトウェアの外部的な振る舞いを変更することなく内部の構造を改善することを言い，典型的なリファクタリングパターンが Fowler によってまとめられている [1] [2].

一般に，ソフトウェアの開発が進むと，設計品質の低い箇所がソースコード中に増加していく．このような箇所は「コードの不吉な匂い」と呼ばれており，リファクタリングは不吉な匂いを除去することを目的として実施される．そして，リファクタリングを実施し，設計品質の低いコードを除去することで，開発時における欠陥の混入を予防できると言われている．しかし，欠陥の混入を予防するためにはどの程度のリファクタリングをどのような頻度で行えば良いのかは明らかになっていないため，ソフトウェア開発において，リファクタリングの実施を効果的に行うことが難しいという問題がある．開発者がより効果的にリファクタリングを実施するためには，欠陥を予防するために最適なリファクタリングの種類や実施頻度が明らかにすることが望ましい．

本研究では欠陥混入を予防する観点から，開発者がリファクタリングを効果的に行うことができるようになることを目的とする．そのために，リファクタリングがソフトウェア開発に与える影響を明らかにするために，オープンソースソフトウェアを対象に分析を行った．

2　提案手法

2.1　概要

本研究では，ソフトウェア開発におけるリファクタリングが欠陥混入に与える影響を分析する．そのために，一定期間にどれだけリファクタリングが実施されたのかを示す，リファクタリング頻度を測定する．また，一定期間にどれだけ欠陥が埋め込まれたのかを示す欠陥の混入頻度と，どれだけ欠陥が修正されたのかを示す欠陥の修正頻度を測定する．

近年，多くのソフトウェア開発においてソースコードの変更履歴を記録するための版管理システムが利用されている．また，ソフトウェア開発中に発見された欠陥は，Bugzilla や Trac などに代表されるバグ管理

An investigation into the effect of refactoring on defect introductions in open source software projects

Kenji Fujiwara, Kyohei Fushida, Norihiro Yoshida, Hajimu Iida, 奈良先端科学技術大学院大学 情報科学研究科, Graduate School of Information Science, Nara Institute of Science and Technology.

システムによって集中管理されている．本手法では，版管理システムおよびバグ管理システムに記録されている情報を用いて分析を行う．

2.2 測定値の定義

版管理システムに記録されたソフトウェアの全リビジョンを V，各リビジョンを v_i とすると，$V = [v_1, v_2, ..., v_n]$ と表せる．次に，リビジョン v_i から v_{i+1} の間にソースコードに対して行われた変更を op_i とし，op_i においてリファクタリングが行われたかを返す $r(op_i)$ を定義する．$r(op_i)$ は，op_i においてリファクタリングが実施された場合は 1，そうでなければ 0 を返す．$r(op_i)$ を用いて，リビジョン v_j から v_k におけるリファクタリング頻度 $f_r(j,k)$ を次のように定義する．

$$r(op_i) = \begin{cases} 1 & (\text{ if } op_i \text{ contains a refactoring }) \\ 0 & (\text{ otherwise }) \end{cases}$$

$$f_r(j,k) = \frac{\sum_{i=j}^{k} r(op_i)}{v_k - v_j} \qquad (j < k, \quad v_j, v_k \in V)$$

同様に，変更 op_i において欠陥が混入されたかを返す $d(op_i)$ と欠陥の混入頻度 $f_d(j,k)$，変更 op_i において欠陥が修正されたかを返す $f(op_i)$ と欠陥の修正頻度 $f_f(j,k)$ を次のように定義する．

$$d(op_i) = \begin{cases} 1 & (\text{if defects are introduced at } v_{i+1}) \\ 0 & (\text{otherwise}) \end{cases}$$

$$f_d(j,k) = \frac{\sum_{i=j}^{k} d(op_i)}{v_k - v_j} \qquad (j < k, \quad v_j, v_k \in V)$$

$$f(op_i) = \begin{cases} 1 & (\text{if defects are fixed at } v_{i+1}) \\ 0 & (\text{otherwise}) \end{cases}$$

$$f_f(j,k) = \frac{\sum_{i=j}^{k} f(op_i)}{v_k - v_j} \qquad (j < k, \quad v_j, v_k \in V)$$

以降，本節ではソフトウェア開発におけるリファクタリングの実施時期の特定および欠陥の混入時期の特定方法について説明する．

2.3 リファクタリングの実施時期の特定

リファクタリングの実施時期を特定するにあたり，本研究では UMLDiff アルゴリズム [7] を利用する．UMLDiff アルゴリズムは，同一プロジェクトにおける 2 リビジョンのスナップショットを入力として与えることで，それぞれのスナップショットからプログラムの構造情報を復元し，それらの差分を求める．Xing らは，UMLDiff アルゴリズムで得られた差分情報を基に，2 リビジョン間で実施されたリファクタリングを検出する手法を提案している [8]．

本手法では，まず最初と最後のリビジョンを UMLDiff アルゴリズムに入力し，対象プロジェクトの開発全体を通して実施されたリファクタリングを抽出する．次に，検出されたリファクタリングについて，それぞれ対応するコードを手作業で見つけ，そのコードが埋め込まれた時期を版管理システムを用いて特定する．

2.4 欠陥の混入時期の特定

本手法では，ソフトウェアの開発中に発見され，その後修正された欠陥の混入時期を特定する．混入時期の特定には SZZ アルゴリズムを用いる [5]．SZZ アルゴリズムはまず，版管理システムから欠陥の修正を目的とした，ソースコードの変更を特定する．その後，修正時に変更されたコードがいつ埋め込まれたのかを調べることで，欠陥の混入時期を特定する．

修正を目的とした変更の抽出は，版管理システムに記録されたコミットログや，バグ管理システムに記録された情報を利用して行う．

3 適用実験

前節で述べた手法の有効性を確認するために，オープンソースソフトウェアである Columba[†1] を対象に適用実験を行った．Columba に関する情報を表 1 に示す．

実験では，まず UMLDiff を用いてリファクタリングの実施を検出し，それぞれの結果に対してリファクタリングかどうか手作業で確認を行った．UMLDiff による誤検出を除いた結果を表 2 に示す．結果として，15 種類のリファクタリングについて，リファクタリングの実施を計 79 個抽出することができた．

Columba は，コミットログに対するルールが厳格に定められており，バグ修正に対応する変更には **[bug]**

†1 http://sourceforge.net/projects/columba/

表 1 実験対象プロジェクトの概要

種類	開発期間	総リビジョン数	最終 LOC
メールクライアント	2006/7/9-2011/7/11	458	192,941

表 2 検出されたリファクタリング

リファクタリング	個数
Convert inner type to top level	1
Convert top level to inner	2
Die-hard/legacy classes	1
Downcast type parameter	3
Encapsulate field (get)	1
Extract class	2
Extract method	7
Extract subsystem/package	4
Generalize type (method)	8
Generalize type (parameter)	37
Information hiding	6
Inline subsystem/package	1
Pull-up method/field/behaivior	5
Push-down method/field/behaivior	1
合計	79

や [**fix**] という文字列がタグとして記述されている．実験においては，これらのタグを用いることで修正を目的とした変更を 322 個抽出し，それらを用いて欠陥を混入した変更を 243 個抽出した．

4 考察

特定したリファクタリングの実施時期と，欠陥の混入時期から求めたリファクタリング頻度，欠陥の混入頻度および欠陥の修正頻度をプロットしたグラフを図 1 に示す．図中の頻度はそれぞれ 25 リビジョンを 1 間隔として計算している．リファクタリングが最も実施された 151 リビジョンから 175 リビジョンの期間以降は，欠陥の混入頻度が減少傾向にある．実際，151 リビジョンから 156 リビジョンにおいては，コミットログに大規模なリファクタリングを実施したと記録されており，このリファクタリングにより，ソースコードの品質が向上し，以降の欠陥の混入が減少したのではないかと考える．また，全体を通して，欠陥の混入頻度が上昇した後に，リファクタリングの実施頻度が高くなり，その後欠陥修正の頻度が高くなっていることから，リファクタリングをきっかけに欠陥の修正が行われたと考えることができる．

5 関連研究

リファクタリングと欠陥の関係を分析している研究として，Ratzinger らはコミットログを用いてリファクタリングを検出し，欠陥との関係を分析している [4]．彼らは，ある期間におけるリファクタリングの回数が多かった場合，後続する期間の欠陥の発生量が減少すると報告している．そして，リファクタリングは欠陥の混入を予防する効果があると述べている．しかし，彼らは時間に対する詳細な変化については分析していない．本研究の手法を用いることで，リファクタリングと欠陥の時間的な関係をより詳細に分析することが可能である．Kim らは，3 つのオープンソースプロジェクトを対象に，ソフトウェア開発においてリファクタリングがどのような役割を果たしているのかを調査した [3]．彼女らは，開発履歴からリファクタリングの履歴を復元し，リファクタリングの役割を次の 4 つの観点から評価している．

- リファクタリングの後にバグ修正が行われているか
- リファクタリングは開発者の生産性を向上するか
- リファクタリングはバグ修正を促すか
- ソフトウェアのメジャーリリース前にどの程度リファクタリングが行われているか

彼女らは，分析の結果，Eclipse JDT プロジェクトにおいてリファクタリングが行われたリビジョンの 5 リビジョン以内に 26.1%の確率でバグ修正が行われていると報告している．また，リファクタリングによりバグ修正にかかる時間が減少するとも述べている．

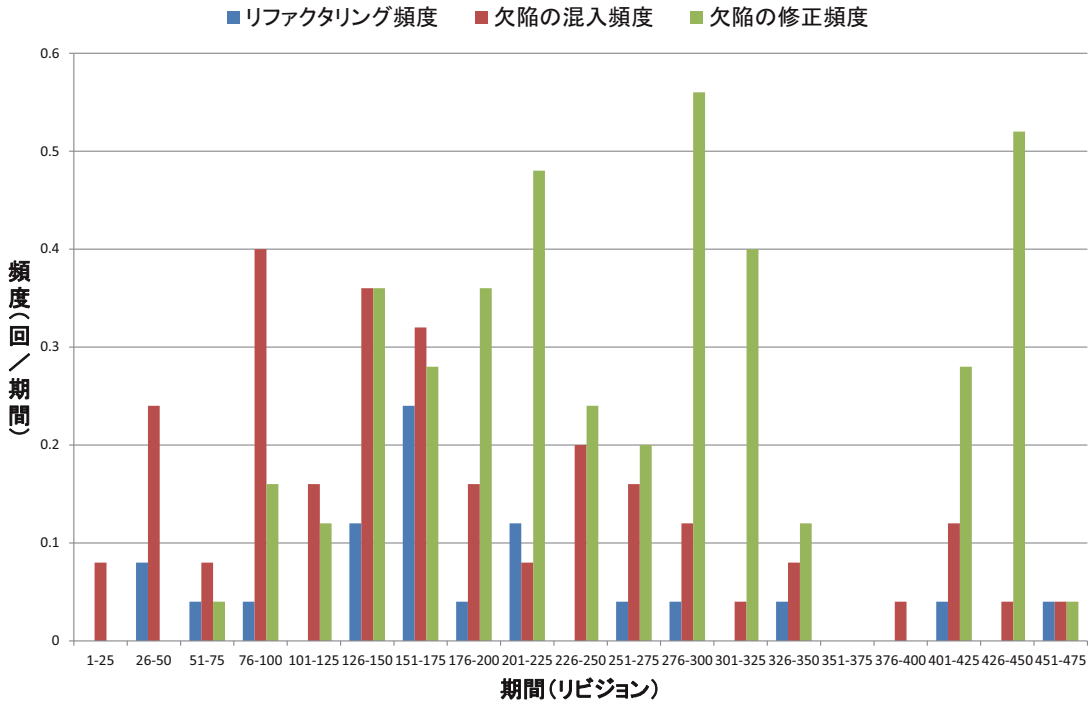

図 1　Columba におけるリファクタリング頻度と欠陥の混入／修正頻度

そして，リファクタリングと共にバグ修正が行われることが多いことと，メジャーリリース直前に多くのリファクタリングが行われていたと報告している．彼女らは，バグ修正の観点から，リファクタリングがソフトウェア開発の品質にどのような影響を与えるかを評価している．本稿では，欠陥の混入という観点から，リファクタリングが与える影響を分析する手法を提案し，分析を行った．

リファクタリングの検出手法として，Weißgerber と Diehl はソースコードの構文及び識別子情報を用いたリファクタリングの検出手法を提案している [6]．彼らの手法は，リファクタリングと思われるコードの変更箇所を，構文情報と識別子情報を用いて抽出する．そして，それらの変更箇所を，コードクローン検出手法を用いてランク付けし，リファクタリングを検出する．この手法で検出可能なリファクタリングは 10 種類で，UMLDiff による手法が検出できる 33 種類と比べて少ない．

6　まとめと今後の課題

本稿では，ソフトウェアの開発履歴から，リファクタリングの実施頻度および，欠陥の混入頻度を測定することで，リファクタリングが欠陥混入に与える影響を詳細に分析する手法を提案した．そして，オープンソースソフトウェアを対象にリファクタリングの実施頻度，欠陥の混入／修正頻度を測定し，それらの関係を調査した．

今後は，より多くのオープンソースソフトウェアに対して手法を適用するとともに，今回適用したプロジェクトよりも規模の大きなプロジェクトを対象に分析を行う予定である．今回，UMLDiff アルゴリズムを，最初と最後のリビジョンに対して適用し，その結果を用いてリファクタリングの実施時期を特定した．しかしこの手法では原理上，最初のコミット以降に追加されたクラスやメソッドに対して実施されたリファクタリングについては検出することができないため改善の必要がある．

謝辞 本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」の委託に基づいて行われた．また，日本学術振興会 科学研究費補助金 基盤研究（C）(課題番号:22500027)，および研究活動スタート支援（課題番号:22800040,22800043）の助成を得た．

参 考 文 献

[1] Fowler, M.: Refactoring Home Page, `http://refactoring.com/`.

[2] Fowler, M.: *Refactoring: improving the design of exsiting code.*, Addison Wesley, 1999.

[3] Kim, M., Cai, D., and Kim, S.: An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution, *In Proc. the 33rd International Conference on Software Engineering(ICSE 2011)*, 2011, pp. 151–160.

[4] Ratzinger, J., Sigmund, T., and Gall, H. C.: On the relation of refactorings and software defect prediction, *In Proc. the 5th Working Conference on Mining Software Repositories(MSR 2008)*, 2008, pp. 35–38.

[5] Śliwerski, J., Zimmermann, T., and Zeller, A.: When do changes induce fixes?, *In Proc. the 2nd International Workshop on Mining Software Repositories(MSR 2005)*, 2005, pp. 1–5.

[6] Weißgerber, P. and Diehl, S.: Identifying Refactorings from Source-Code Changes, *In Proc. the 21st IEEE/ACM International Conference on Automated Software Engineering(ASE 2006)*, 2006, pp. 231–240.

[7] Xing, Z. and Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing, *In Proc. the 20th IEEE/ACM International Conference on Automated Software Engineering(ASE 2005)*, 2005, pp. 54–65.

[8] Xing, Z. and Stroulia, E.: Refactoring Detection based on UMLDiff Change-Facts Queries, *In Proc. the 13th Working Conference on Reverse Engineering(WCRE 2006)*, 2006, pp. 263–274.