オープンソースリポジトリのバグ修正履歴を再利用したデバッグ推薦の評価実験

塩塚 大[†1] 鵜林尚靖[†2] 亀井靖高[†2]

プログラマはデバッグに多くの時間を費やす傾向がある．プログラマはエラー現象をチェックしたり，バグ修正履歴を探索したりして，コードを修正する．このような一連のデバッグ作業を自動化することができれば，他の生産的な作業に時間を割くことができる．我々は、この問題に対処する方法として，デバッグ支援のための関心事指向推薦システム dcNavi（Debug Concern Navigator）を提案してきた．本論文では，Eclipse プラグインプロジェクトに関する 9 つのオープンソースリポジトリを対象に，再利用性の観点からバグ修正履歴に基づいた推薦の有効性について評価する．

Experimental Evaluation of Debug Recommendation Reusing Past Bug Fixes in Open Source Repositories

Masaru Shiozuka,[†1] Naoyasu Ubayashi[†2] and Yasutaka Kamei[†2]

Programmers tend to spend a lot of time debugging code. They check the erroneous phenomena, search the past bug fixes, and modify the code. If a sequence of these debug activities can be automated, programmers can use their time for more creative tasks. To deal with this problem, we previously proposed *dcNavi* (Debug Concern Navigator), a concern-oriented recommendation system for debugging. In this paper, we evaluate the effectiveness of our approach in terms of the reusability of past bug fixes by using nine open source repositories created in the Eclipse plug-in projects.

†1 九州工業大学 (現在，メルコ・パワー・システムズ株式会社)
Kyushu Institute of Technology
†2 九州大学
Kyushu University

1. はじめに

デバッグを行う際，開発者は過去のバグ修正情報を利用することがある．特に，経験の少ないドメインでのソフトウェア開発や，大規模なソフトウェアの開発では原因の特定に時間がかかる場合が多く，類似したバグやその修正方法を修正の取っ掛かりにすることがある．

我々は，デバッグにまつわるこれらの作業を支援することを目的に，既存のリポジトリを活用した推薦システム dcNavi（Debug Concern Navigator）[9] を提案している．dcNavi の特徴は推薦の際に DCG（Debug Concern Graph，デバッグ関心事グラフ）という，デバッグ時に発生した情報とプログラム解析情報から構成される有向グラフを活用する点である．DCG は関心事グラフ[8] を拡張したものである．関心事グラフは作業に関係したメソッドやクラスなどを互いに関連付けた有向グラフである．ここにテスト結果やバグ修正パターン[7]（メソッドのパラメータの変更のような典型的な修正方法）などのデバッグ情報を追加することで，どのメソッドでどういう例外が発生しどのような修正が行われたかという検索が容易となる．

本論文では，バグ修正履歴に基づいた推薦がデバッグにどの程度有効かを定量的に明らかにする．今回，Eclipse プラグインで Mylyn[5]（旧 Mylar[2]）に関連した 9 つのオープンソースプロジェクトを対象として dcNavi の評価実験を行った．実験で評価したのは dcNavi が提供する推薦機能の 1 つであるライブラリの正しい利用法に関するものである．リポジトリの変更履歴をもとにバグを含んでいたメソッド（修正対象）を取出し，ライブラリの誤り易い例とその際の修正方法（修正例）を推薦した．なお，修正対象は変更履歴から実際にその後どのように修正されたかは分かっているものを用いた．修正例が正解かの判定はバグ修正パターンの一致/不一致で行った．例えば修正対象と修正例がいずれも同じメソッドを追加するような修正をしていた場合に一致と判断する．これはバグ修正パターンが一致していれば実際の修正方法と修正例がある程度似ており修正の参考になり得るのではないかと考えたからである．その結果，9 プロジェクトで平均して適合率 22.2%，および再現率 40.2%で修正例を推薦することができた．

次節以降の構成は次の通りである．まず 2 節で研究の動機を述べ，3 節で dcNavi について説明する．4 節で dcNavi を用いて行った実験について説明し，5 節で関連研究を紹介する．最後に 6 節でまとめる．

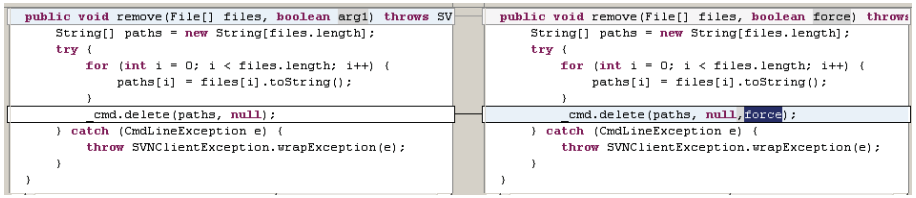

図 1　subclipse プロジェクトのリビジョン 984 での remove メソッドの修正（右側が修正後）

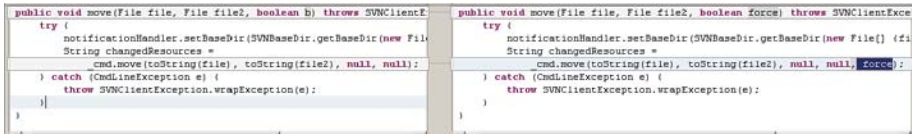

図 2　subclipse プロジェクトのリビジョン 920 での move メソッドの修正（右側が修正後）

2. バグ修正履歴とデバッグ支援

デバッグを行う際，開発者は実行結果やプログラムの挙動から「過去に同じようなバグに遭遇したことがある気がする」と感じ，実際に似たような修正を行うことがある．例えば，図 1 の remove メソッドおよび図 2 の move メソッドでは，いずれもメソッドのパラメータの変更を行っている．ソースコードから察するにいずれもコマンドを実行する際に強制実行する（force 引数）ことを指定していなかった，という同じ誤りであったと推測できる．もし，リビジョン 984 の remove を修正する際にリビジョン 920 の move の修正方法を推薦できれば，互いに類似した誤りであるから修正の参考になると期待できる．

過去の修正情報を得る方法としてバージョン管理システムの利用（ソースコード検索など）が考えられる．ただし，そこからデバッグに関連した情報を得るためには次のような工夫が必要になると考える．まず，一般にバージョン管理システムでは仕様変更などのようにデバッグとは関係ない変更もコミットされる．そういったものは出来るだけ排除する必要がある．また，全てのバグ修正が役立つとは限らない．例えば，import 文の追加などは再利用する必要はなさそうである．最後に，特定のクラスやメソッドを探したい場合，単純にテキストベースで検索しただけでは，クラスやメソッド以外のコメントや変数名の一部が検索にヒットしてしまう場合がある．特定の API を検索したい場合では型まで考慮して検索できる必要がある．

表 1　dcNavi が提供する推薦機能一覧

問題の状況	入力	出力（推薦）
ライブラリの利用方法が分からない	ライブラリ	ライブラリを利用したメソッド
例外やテスト結果を活用できない	例外	関連ソースコード
テスト作成方法が分からない	ライブラリ	関連したテスト
レビューすべき箇所が分からない	対象ソースコード	修正候補箇所

3. デバッグ推薦システム dcNavi

本節では実装したツール dcNavi について説明する．dcNavi は統合開発環境 Eclipse のプラグインとして作成した．

3.1 提供する推薦機能

dcNavi は表 1 に示した推薦機能を提供している．例えばライブラリの正しい利用方法が分からない場合には，あるライブラリの誤り易い例と，そのときの修正方法を推薦する．推薦例を確認することで，コードの誤りに気づける可能性がある．他に，例えばテスト結果や例外情報からどう修正をすべきか分からない場合には，過去にその例外に関連したソースコードとそのときの修正方法を推薦する．同じ例外が発生したソースコードであれば似た操作やライブラリを使っている可能性があり修正の手掛かりとなり得ると期待できる．

3.2 デバッグ関心事グラフ DCG

図 3 は，図 2 に示した subclipse プロジェクトのリビジョン 920 における修正を元に作られた DCG である．DCG ではリビジョン R における修正の情報を，修正前のリビジョン (R-1) の情報と，修正後のリビジョン R の情報の組で表現する．修正前の情報として修正されたメソッドと，そのメソッド内のプログラム要素を保存する．具体的には表 2 の 3 種類のいずれかの関係に該当するプログラム要素を保存する．例えば，図 3 では修正前の CmdLineClientAdapeter クラスの move メソッド（459 と書かれている方．この数字はこのメソッドが定義されている行番号を表す）では，メソッド内の 461 行において SVNNotificationHandler クラスの setBaseDir メソッドを呼び出していることを表す．これらの関係は先行研究の関心事グラフで定義されたもので，全部で 7 種類提案されている内の 2 種類を DCG では使った[⋆1]．修正後の情報として具体的にどういった修正が行われたかを分類したバグ修正パターンを保存する．バグ修正パターンは文献[7] で提案されたもので，プログラムの構文の変更をもとに誤り易いバグの修正を 27 種類に分類している．DCG ではそのうちの表 3 に示した 4 つの修正を支援対象とした．例えば，図 3 では修正後の move メ

⋆1 この他に修正前の情報としてテスト結果等も保存するが，今回は以降でも触れないので割愛する．

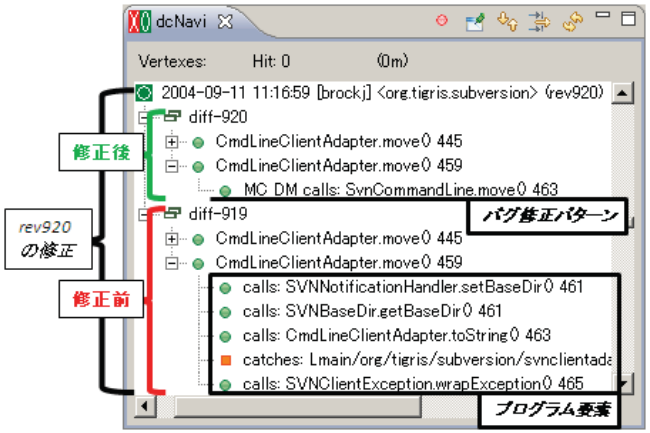

図 3　DCG の例：subclipse プロジェクトのリビジョン 920 の修正

表 2　DCG で利用するメソッド内のプログラム要素間の関係

関係	説明
(calls, m1, m2)	メソッド m1 内部においてメソッド m2 を呼出している
(creates, m, c)	メソッド m 内部においてクラス c のオブジェクトを生成している
(catches, m, e)	メソッド m 内部において例外 e を捕捉している ※ DCG で追加

表 3　DCG が対象とするバグ修正パターン

バグ修正パターン	説明
MC-DAP	パラメータの変更
MC-DM	同一オブジェクトに対する呼び出すメソッドの変更
SQ-AOB	2〜3 文からなる小さなブロック内でのメソッドの追加
SQ-ROB	2〜3 文からなる小さなブロック内でのメソッドの削除

ソッド内で呼び出すメソッドの変更を行ったことを表す（図 2 の修正は実際にはオーバーロードなので呼び出すメソッドの変更であり，パラメータの変更ではないと判断している）．なお，文献[7] では SQ-AOB と SQ-ROB をまとめて SQ-AROB という一つにしているが，本論文では分割している．

3.3　リポジトリからの DCG 生成方法

リポジトリから DCG を生成する上での 3 つの要点を説明する．対象としたリポジトリはバージョン管理システムの 1 つである Subversion（SVN）の形態のリポジトリである．

3.3.1　バグ修正の開始/終了のリビジョンの判別方法

コミットログに着目しキーワード「bug, fix, patch」が含まれるリビジョンをバグ修正の終了のリビジョンと判断し，その直前のリビジョンをバグ修正開始のリビジョンと判断する．この判別方法はバグ修正パターンを提案している文献[7] で紹介された方法である．

3.3.2　修正差分からのバグ修正パターンの識別方法

バグ修正の開始から終了までの間で編集されたファイルについて，最長共通部分列（LCS）を利用し修正に関連した行番号を求める．行番号が連続している間は 1 つの修正の塊（以降では Hunk[7]）と見なし，この Hunk 内に含まれる AST ノード同士の比較によりバグ修正パターンを求める．例えば，同一メソッドが含まれていてパラメータだけが異なれば MC-DAP と判断する．DCG ではバグ修正パターンに分類される修正を行ったリビジョンで，特に表 3 に示した 4 パターンに分類されるバグ修正を収集する．これによりコメントの変更のような再利用上のメリットが低そうな修正情報を排除できる．また，修正行が 7 行未満の Hunk だけを収集する．実験によりこの条件で 41.29%のバグ修正を分類出来た．

3.4　推薦アルゴリズム

今回実験評価したライブラリの正しい利用方法の推薦アルゴリズムを説明する．この推薦では，修正対象メソッド（入力）を dcNavi 上で選ぶと（例えば図 3 では move メソッド），そのメソッド内で利用しているライブラリの過去の修正例を得ることができる．

推薦候補は次のようにして求める．

STEP1. DCG 全体からメソッドの修正後コードを全て取得する．ここで求める修正後コードはバグ修正パターンが 1 個以上含まれていなければならない．そして，バグ修正パターンに関連したメソッドが修正対象メソッドに含まれていなければならない．これにより，修正対象メソッド内で呼ばれていたメソッドに関連した修正例のみが取得できる．

STEP2. 得られたメソッドの修正後コードに対応するその修正前コードを全て取得する．ここで得られたメソッドが推薦候補となる．

STEP3. 修正対象メソッドと推薦候補を比較し以下で定義する類似度を計算する．類似度は 2 つのメソッド M1, M2 の**共通要素数の割合**で定義する．推薦候補の比較にいずれも修正前コードを用いる．なぜなら修正対象メソッドはこれから修正されるため推薦時点では修正後コードを持たないためである．

`sim(M1, M2) = M1 と M2 の共通要素数 / ((M1 の要素数 + M2 の要素数)/2)`

STEP4. 類似度の高い順にランキングし，修正候補を推薦していく．DCG では修正前と修正後を組で保存する．したがって，メソッドの修正前である修正候補から対応する修正後を取得することは容易である．実際に推薦されるのは修正前であるが，利用者は修正後との差分を確認することができる．

表 4　対象オープンソース

プロジェクト	開発期間	NOR（NOB）	LOC
google code	2009-12-16 ～ 2010/5/14	18(2)	2743
industrial mylyn	2010-05-20 ～ 2010-07-09	50(2)	10953
Mylyn Mntis Connector	2007-02-22 ～ 2010-07-14	537(107)	18536
Origo	2006-12-22 ～ 2010-10-08	3761(867)	5769
qcMylyn	2009-08-06 ～ 2010-09-26	223(80)	13117
Redmine Mylyn Connector	2008-05-26 ～ 2010-05-19	423(134)	14729
Remember The Milk	2008-01-26 ～ 2009-11-24	70(13)	7259
Scrum Vision	2008-05-24 ～ 2010-07-12	431(14)	43263
subclipse	2003-06-20 ～ 2010-10-14	4745(923)	167100

NOR: Number of revisions　NOB: Number of bug fix revisions　LOC: Line of code

表 5　マシンの環境

項目	内容
Operating System	Windows XP Home Edition (5.1, Build 2600) Service Pack 3
Processor	Intel(R) Atom(TM) CPU N280 @ 1.66GHz (2 CPUs)
Memory	2038MB RAM
Eclipse	Version: 3.5.2
dcNavi	Version: 0.1.0

4. 評 価 実 験

本節ではオープンソースを対象としておこなった dcNavi の評価実験について説明する．リポジトリから生成した DCG をもとにライブラリの正しい利用法に関する推薦を行い，4.2 節で定義する正解（バグ修正パターンの一致度）に基づき推薦の質を確認した．

4.1 対象プロジェクトと実験環境

実験対象としたオープンソースを表 4 に示す．対象としたのは Eclipse プラグインで Mylyn に関連した 9 つのオープンソースプロジェクトである．異なるプロジェクトを混ぜて使うことで、一般的な誤りやドメインに特化した情報が集まるのではないかと考えこのような選定をおこなった．推薦元として使うデータは，自プロジェクトのリビジョン全体の 80%から作られた DCG と、他の 8 プロジェクトのリビジョン全部から作られた DCG を用いた．これに対して，自プロジェクトの残り 20%から作られた DCG に対し推薦を行った．実験を実施したマシンの環境を表 5 に示す．

4.2 推薦の正解の定義

4.2.1 推薦の質の表現方法

推薦の質を計測するために，適合率（Precision），再現率（Recall）および F 値（F-measure）を用いた．適合率とは正確性の指標で，推薦結果内の正解の割合である．再現率とは正解の網羅性の指標で，推薦候補内（自プロジェクトの 80%および他の 8 プロジェクト）に含まれる全正解に対する推薦結果内の正解の割合である．F 値とは適合率と再現率の調和平均で *2/F-measure = 1/Precision + 1/Recall* である．

表 6　実験結果

条件	適合率 (%)	再現率 (%)	F 値 (%)
1 を満たす	68.06	0.57	1.13
2 を満たす	28.60	27.33	27.95
1 と 2 を満たす	22.15	40.16	28.55

類似度下限 0.2 で実施

表 7　プロジェクトごとの実験結果

Project	適合率 (%)	再現率 (%)	F 値 (%)
projecthosting-connector-for-mylyn	N/A	N/A	N/A
industrial-mylyn	N/A	N/A	N/A
Mylyn-Mntis Connector	23.44	32.61	27.27
Origo	N/A	0.00	N/A
qcMylyn	34.42	50.24	40.85
Redmine-Mylyn Connector	22.29	33.94	26.91
Remember The Milk	3.85	20.00	6.45
Scrum Vision	0.00	0.00	N/A
subclipse	14.20	40.52	21.03
Average	22.15	40.16	28.55

条件 1 および条件 2 を満たす場合

4.2.2 正解の定義

以下の条件を満たす場合に正解と定義する．

- **条件 1．バグ修正パターンのパターンの種類が一致している：**例えば両者がともに MC-DAP のパターンに分類される修正をおこなっていれば一致していると判断する．
- **条件 2．バグ修正パターンの対象となったメソッドが一致している：**例えば両者がともに move メソッドを追加するような修正を行っていれば一致していると判断する．ここではメソッド同士の一致を，そのメソッドが宣言されているパッケージ，クラス，メソッド名，およびインタフェース（パラメータの型や数，戻り値の型）を文字列として見た際に完全に一致していることを意味する．図 1 では delete メソッドが変更され，図 2 では move メソッドが変更されているのでこの場合は不一致と判断されるが，もし一致していれば（どちらも delete メソッドを修正，あるいはどちらも move メソッドを修正）同じメソッドの使い方を間違っていたことになり，より修正の参考に成り得ると考えこのような条件を作った．

今回の正解判定についての制約事項を述べる．実際の修正では 1 つの修正対象メソッド内に，複数の箇所（Hunk）の修正が行われることがある．さらにその内の 1 つの箇所に複数のバグ修正パターンが含まれることがある．しかし，今回は 1 メソッド内の 1 つバグ修正パターンについて，推薦されたメソッドの 1 つのバグ修正パターンが上述の条件を満たせば正解とする．

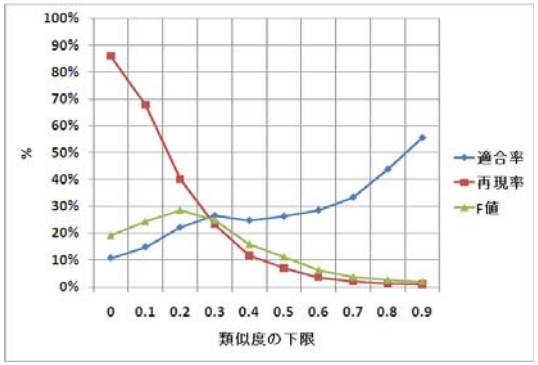

図 4　類似度の下限を変化させた場合の推薦の質の変化

4.3 結　　果

9 プロジェクトを平均した実験結果を表 6 に示す（類似度下限 0.2 で実施）．適合率は 22.15%，再現率は 40.16%，F 値は 28.55%であった（参考までに，条件 1 あるいは条件 2 のみの場合も表 6 に示している）．なお，表中，N/A になっている箇所はバグを含むメソッド数が 0 またはごく少数との理由で推薦の評価に適さない場合である．表 7 にプロジェクトごとの実験結果を示す．qcMylyn プロジェクトでは適合率は 34.42%，再現率は 50.24%，F 値は 40.85%であった．

4.4 考　　察

正解と類似度の関係を調べるために，図 4 に類似度の下限を 0 から 0.1 刻みで変化させていった場合の推薦の質を示す．類似度の下限を上げると適合率は上がり，一方で再現率は下がる．類似度が 0.2 付近で F 値が最も高くなる．そのため，本実験では，表 6 および表 7 に示したように類似度下限 0.2 で評価した．

正解集合および適合集合のドメイン上の分布の確認をおこなった．正解集合とは実験時において予め分かっている正解から成る集合で，適合集合とは推薦結果の内の正解から成る集合である．正解集合および適合集合のドメイン上の分布を図 5 に示す．数字はそのドメインに分類できたメソッドの合計数を表す．ドメインは 4 つに分類しており，1. クラスライブラリ（cf. java.lang.*，java.util.*，java.io.*），2.Mylyn パッケージの API（cf. org.eclipse.mylyn.*），3.Mylyn パッケージ以外の Eclipse の API（cf.org.eclipse.core.*，org.eclipse.swt.*），4. 自プロジェクトのメソッド（cf. ch.ethz.origo.*，com.itsolut.mantis.*）である．正解集合

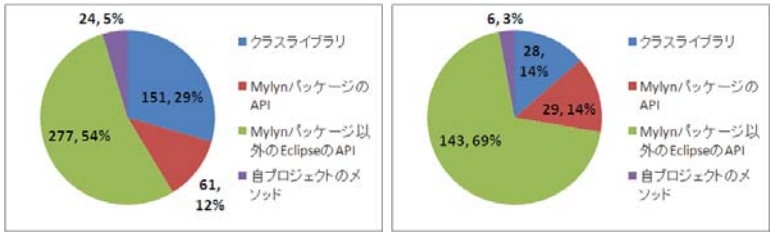

図 5　正解集合（左）と適合集合（右）の分布

では Mylyn に関するものが 12%で，そして適合集合では Mylyn に関するものが 14%であった．自プロジェクトの誤りは，全体的に多くないことが分かった．正解集合のメソッドの種類は全部で 62 種類，適合集合のメソッドの種類は全部で 34 種類であった．約半数の修正がカバーされ得ることが分かった．

5. 関 連 研 究

本節では，デバッグ支援のための推薦システムをいくつか紹介すると共に、dcNavi との関連について述べる．

5.1 BugMem

BugMem[3] はリポジトリ内から類似したバグ修正のパターンを推薦することで，デバッグ支援をおこなう．すべてのプロジェクトで共通するような一般的なバグ（Horizontal Bug）に対して，プロジェクトで固有なバグ（Vertical Bug）を見つけ出そうとしている．パターンを見つけるために修正前後の Hunk 同士の比較の際に，様々なレベル標準化をおこなう方法を提案している．パターン検索の際にはオプションの指定により，どこまで Hunk を抽象化するか決めることができる．dcNavi では Hunk 内のバグ修正パターンに加えて，その Hunk が含まれるメソッド同士の類似度を考慮し推薦をおこなった．

5.2 DebugAdvisor

DebugAdvisor[1] では類似したバグを検索するために，バージョン管理/バグデータベース/デバッガのログをまとめて検索できるクエリ（fat query）を提案している．検索を実現するために，それぞれ形式の異なる情報から特徴を抽出し型付き文書（typed document）と呼ばれる構造に変換する．型付き文書は文法が定義されており，たとえばスタックトレースを表現する際には，呼出し順序が保たれるように表現される．DebugAdvisor は基本的にプログラムの出力同士（バグの記述も含める）を比較し類似したバグを検索する．一方，dcNavi

ではプログラム解析情報とテスト結果を連携させた推薦を行っている．また，DebugAdvisor が特に自プロジェクトの履歴を使って推薦を行うのに対し，dcNavi では他プロジェクトの履歴も活用している．

5.3 Whyline

Whyline[4] では，問題コードの代わりに実行履歴などに対して質問を重ねていくことで問題箇所を絞り込む方法を提案している．これはデバッグの多くが，まずプログラムの振る舞いについて疑問を持ち，それをツールを使って確かめる，という一連の作業だからである．提案しているツールではプログラム解析と実行履歴を活用したデバッグ支援をおこなっている．質問は利用者が変数やイベントなどの入力をもとに Whyline が生成する．その質問内容に合致するプログラム実行上の箇所へ移動することができる．実行結果をもとにプログラム実行を移動するので，通常のデバッガに加え指定した時点へ（任意の時点ではなく質問可能な位置）の逆戻りが可能なデバッガといえる．dcNavi では，直接的に問題箇所を特定するのではなく過去の修正情報を推薦することで，現状から想定される修正例を示した．これにより開発者は問題の特定に加え，具体的な修正方法の例を知ることができると考える．

5.4 FixWizard

FixWizard[6] では，過去の修正情報をもとに，修正対象のコードに関連するような修正コードの推薦方法を提案している．dcNavi とかなり近いことをおこなっている．大規模なオブジェクト指向プログラムでは，同じような役割を持ち，同じような機能を提供し，あるいは同じようなやり取りを他のオブジェクトと行うオブジェクトが存在する．これらは似たようなコードあるいは似たような状況で現れる．似たものの一方に修正が行われた場合に，他の似たもの（code peers と呼んでいる）も修正され得るとしている．FixWizard では仮説として「似たような修正は code peers において頻繁に発生し得る」を掲げ実験をもとに実証している．また，同じような機能をもつメソッドやクラスを実装しようとした際，開発者はコピー&ペーストすることが多いので，同じようなコードを作る傾向にある．したがって，「code peers は同じような実装や名前，あるいはインタフェースを持つ」という仮説を立てこれも実証している．今後 DCG の推薦精度を改善していく上で参考にし，ツール同士の比較実験などをしていきたいと考えている．

6. おわりに

本論文では，過去のデバッグ情報に基づいた推薦の有効性について，オープンソースを用いた評価実験を行った．メソッドなどのプログラム要素の実行順序，あるいはデータフローなどを無視した場合でも，抽象化されたプログラム同士の比較である程度の質の推薦が可能であった．現在、多くの推薦システムはランキング方式を採っている．ランキングの上位にあるほど推薦として確からしいという考え方に基づいているが，その一方で 1 つの推薦よりは関連する複数の事例を推薦する方式（グループ推薦）の方が有効な場合も多いと考えられる．バグの要因は 1 つだけでなく複合している場合が多いからである．また，類似した複数の事例を提示して貰った方がプログラマも修正の確信が持てやすくなる．どのように事例をグループ化して提示すべきは，今後重要な研究テーマと考えられる．

参考文献

1) Ashok, B., Joy, J., Liang, H., Rajamani, S. K., Srinivasa, G., and Vangala, V.: DebugAdvisor: A Recommender System for Debugging, In *Proceedings of the 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pp.373-382, 2009.
2) Kersten, M. and Murphy, G. C.: Mylar: A Degree-of-interest Model for IDEs, In *Proceedings of the 4th International Conference on Aspect-oriented Software Development (AOSD2005)*, pp.159-168, 2005.
3) Kim, S. et. al.: Memories of Bug Fixes, In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering(FSE 2006)*, pp.35-45, 2006.
4) Ko, A. J. and Myers, B. A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, In *Proc. of the 30th International Conference on Software Engineering (ICSE 2008)*, pp.301-310, 2008.
5) Mylyn, http://www.eclipse.org/mylyn/.
6) Nquyen, T. T., Nquyen, H. A., Pham, N. H., Al-kofahi, J. and Nquyen, T. N.: Recurring Bug Fixes in Object-oriented Programs, In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering(ICSE 2010)*, pp.315-324, 2010.
7) Pan, K., Kim, S. and Whitehead, Jr. E. J.: Toward an Understanding of Bug Fix Patterns, *Empirical Software Engineering*, Vol.14, No.3, pp.286-315, 2009.
8) Robillard, M. P. and Murphy, G. C.: Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering(ICSE 2002)*, pp.406-416, 2002.
9) Shiozuka, M., Ubayashi, N., and Kamei, Y.: Debug Concern Navigator, In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, pp.197-202, 2011.