情報処理学会論文誌 Vol.53 No.2 631–643 (Feb. 2012)

dcNavi：デバッグを支援する関心事指向推薦システム

塩塚 大[1,†1] 鵜林 尚靖[2,a)] 亀井 靖高[2]

受付日 2011年6月14日, 採録日 2011年11月7日

概要：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の現象をチェックしたり，様々なコード上の箇所を行き来したり，あるいは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．本論文では，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）を提案する．dcNavi では，リポジトリに含まれるプログラム情報やテスト結果，および修正履歴を活用することで，デバッグの関心事に応じた適切なヒントを提供する．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も評価する．

キーワード：デバッグ，関心事グラフ，推薦システム，バグ修正パターン

dcNavi: A Concern-oriented Recommendation System for Debugging Support

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

Received: June 14, 2011, Accepted: November 7, 2011

Abstract: Programmers tend to spend a lot of time debugging code. They check the erroneous phenomena, navigate the code, 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 propose *dcNavi* (Debug Concern Navigator), a concern-oriented recommendation system for debugging. The *dcNavi* provides appropriate hints to programmers according to their debug concerns by mining a repository containing not only program information but also test results and program modification history. 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.

Keywords: debug, concern graph, recommendation system, bug fix patterns

1. はじめに

デバッグを行う際，経験を積んだプログラマであれば比較的スムーズにバグを修正できる．これは「こういった症状のときにはこうしたらいい」と分かる場合が多いからである．

一方でプログラミング初心者は様々な問題に直面する．たとえば，テスト結果や例外をもとに具体的にどういった修正をすべきか分からない場合がある．他の問題として，使用しているライブラリの使い方はこれで正しいのか，他にテストすべき項目はないのか，あるいはソースコードのどこを特に注意して見直すべきか，などに直面する．このような問題に直面したとき，プログラマの多くは過去のバグ修正情報を利用する [3]．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

[1] 九州工業大学
Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[2] 九州大学
Kyushu University, Nishi, Fukuoka 819–0395, Japan
[†1] 現在，メルコ・パワー・システムズ株式会社
Presently with Melco Power Systems Co., Ltd.
[a)] ubayashi@acm.org

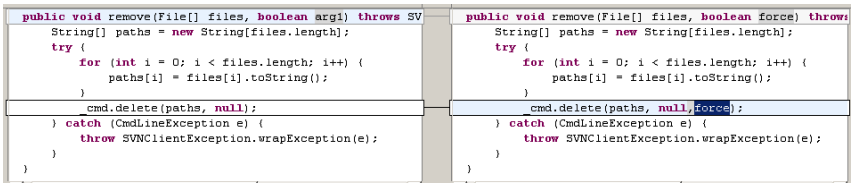

図 1 subclipse プロジェクトのリビジョン 984 での remove メソッドの修正（右側が修正後）
Fig. 1 Modification of the “remove” method [subclipse project, revision 984] (right: after modification).

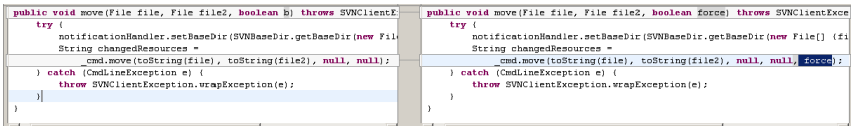

図 2 subclipse プロジェクトのリビジョン 920 での move メソッドの修正（右側が修正後）
Fig. 2 Modification of the “move” method [subclipse project, revision 920] (right: after modification).

その修正方法を問題解決の糸口にする．たとえば，**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 の修正方法を提示できれば，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．

本論文では，デバッグ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）を提案する[*1]．本論文でいう「推薦」とは，開発者にデバッグに有用な情報を具体的に提示することを指す．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 Robillard らの関心事グラフ [13] 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 [12]（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．

さらに，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 [10]（旧 Mylar [6]）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．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．

次章以降の構成は次のとおりである．まず 2 章で研究の動機を述べ，3 章で DCG，4 章で dcNavi について説明する．5 章で dcNavi を用いて行った実験について説明し，6 章で関連研究を紹介する．最後に 7 章でまとめる．

2. バグ修正履歴とデバッグ支援

本章では，まず最初に，デバッグ時にどのような問題に直面しやすいかを簡単な例題シナリオで説明する．その後，解決すべき課題を明らかにする．

2.1 例題シナリオ

ここではファイル操作をするプログラムの実装を例として用いる．ファイル操作ではアプリケーション外部とのやりとりが行われるため例外が発生しやすい．そのため特にプログラミング初心者は間違いを犯しやすい．

図 3 左にプログラミング初心者 A が作った Java 言語で書かれた Property クラスの readFile メソッドを示す．このメソッドは引数で指定したファイルから 1 行を読み込みその値を返し，読み込みに失敗した場合は null を返す．図 3 右に A が作った readFile メソッドに対する 2 つのテストを示す．テストの実行にはテストフレームワークの JUnit [5] を利用している．ファイルが存在する場合のテスト testReadFile は成功した．しかし，ファイルが存在しない場合のテスト testReadFileFileNotExist で例外 FileNotFoundException が発生した．このテストを成功させるために A はデバッグを行った．

A はデバッグする際に以下のような 4 つの問題に直面した．

第 1 に，例外の内容から察するにファイルが存在しない場合の処理が要求されている．しかし，どこにどのような処理を記述すべきか分からない．ある例外に初めて接した

*1 本論文は文献 [14] の拡張版である．本論文では，オープンソースリポジトリを対象とした評価実験に対する考察を追加している．

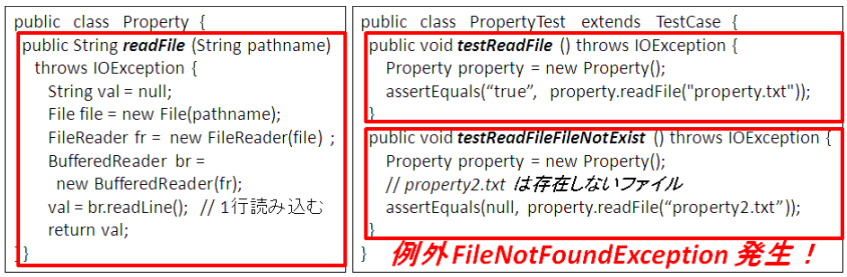

図 3 例題プログラム
Fig. 3 Example program.

際にこういった問題に直面しうる．また，見覚えがあるような例外であっても具体的な修正方法を思い出せない，ということは頻繁に起きうる．

第 2 に，今回使用した File，FileReader，BufferedReader などのライブラリの使い方はこれで正しいのかどうか分からないという問題である．使った経験があまりないライブラリを利用し，バグに遭遇した際にこういった問題に直面しうる．今回 A が利用した File は，ファイルの存在に関係なくコンストラクタが呼ばれればインスタンスを返す．一方他の言語では，たとえば C 言語の fopen 関数などはファイルが存在しない場合に NULL を返す．操作的に似たようなライブラリであっても微妙に仕様が異なる場合に間違いを犯すことがある．

第 3 に，readFile メソッドのテストとして，他にテストすべき項目はないのか分からないという問題である．テストの作成に慣れていない場合にこういった問題に直面しうる．

最後に，ソースコードのどの箇所を特に注意して見直すべきか分からないという問題である．経験を積んだプログラマであれば，どこで間違えやすいかを知っているためレビューする際にそこに絞って間違いを短時間で発見する．一方初心者はそういったポイントを理解していないことが多く修正に手間どることが多い．

2.2 解決すべき課題

前節で述べた問題と本研究の解決方針を以下のように整理する．また，以下に記したようなデバッグの際に頻繁に直面する問題をデバッグにおける関心事と呼ぶことにする．

- **例外やテスト結果をうまく活用できない**：本研究では，これらの情報（例外やテスト結果）の活用手段として，その情報に関連したソースコードの推薦を行う．過去に同様の例外が発生したソースコードであれば，似たような操作やライブラリを使っている可能性がある．その際の修正方法はバグの原因推測の材料となりうると期待できる．
- **ライブラリの正しい利用方法が分からない**：本研究では，あるライブラリの誤りやすい使用例と，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，自分自身のコードの誤りに気づける可能性がある．
- **テスト作成方法が分からない**：本研究では，あるライブラリの利用の際に関連して過去に実施されたテストの推薦を行う．これによりテストの作成に慣れていないプログラマは，具体的にどういったテストを行えばよいかヒントが得られる．
- **レビューすべき箇所が分からない**：本研究では，過去の修正履歴をもとに，特に開発者が間違えやすい箇所の提示を行う．文献 [12] によると if 文条件やメソッドのパラメータなど間違えやすい箇所はバグ修正パターンとして限定できる．あるメソッド，ライブラリに関連してそれらの典型的な間違え方などを提示する方法は効果的である．

上記の関心事に対して推薦を行う際，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．一般的に，ソフトウェア開発では，ソースコードの変更情報はバージョン管理システムに保存されるからである．しかし，バージョン管理システムだけでは必ずしも十分な推薦を行うことはできない．というのは，多くのバージョン管理システムではソースコードの変更情報は保存されているが，テスト結果や例外情報は保存されていないからである．さらに，特定のクラスやメソッドを見つけたい場合，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単なる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．

本論文では，デバッグに関わるこれらの課題を緩和することを目的に，過去のバグ修正履歴に基づいた推薦システム dcNavi を提案する．dcNavi では，デバッグの履歴，テスト結果，プログラム解析情報などを関心事グラフとして表現し，有用なバグ修正方法を開発者に提示する．

3. デバッグ関心事グラフ

本章では，dcNavi のベースとなるデバッグ関心事グラフ（DCG）について述べる．

表 1 関心事グラフで定義されたラベル（DCG で利用するラベルのみ記載）
Table 1 Edge labels in DCG.

辺	説明
(declares, c, f or m)	クラス c においてフィールド f，あるいはメソッド m を宣言している
(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
(calls, m1, m2)	メソッド m1 内部においてメソッド m2 を呼び出している
(catches, m1, e1)	メソッド m1 内部において例外 e1 を捕捉している（※ DCG で追加）

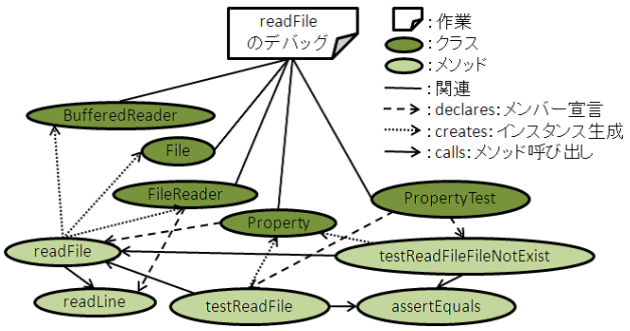

図 4 readFile の関心事グラフ表現
Fig. 4 DCG representation (readFile).

3.1 基本コンセプト

DCG の基本コンセプトは，Robillard らの関心事グラフ [13] に基づいている．ここでは，関心事グラフについて簡単に説明する．

関心事グラフはプログラム要素であるクラス，フィールド，メソッドを頂点とし，要素間の関係を**表 1** で定義されたラベルで表現する有向グラフである．表 1 では（label, v1, v2）として v1 から v2 への辺にプログラム構造上の label の依存関係があることを表す．関心事グラフでは例外捕捉の関係（catches）は定義されていなかったが，メソッドを特徴付ける情報であると判断し DCG では追加している．

図 4 に 2 章の例題シナリオで出てきた readFile の関心事グラフを示す．最上位の「readFile のデバッグ」と書かれた頂点が作業名を表し，楕円形がプログラム要素を表している．関心事グラフはある作業をする際のプログラム理解に用いられる．

関心事グラフは，下記の 3 ステップでプログラマ自身が作っていく．なお，この作業は，Mylyn を用いることにより自動化される．Mylyn では，Eclipse 上でのプログラマのファイルの編集，選択，閲覧，およびカーソルの位置などを監視し，どういったプログラム要素が作業に関連しているかを取得することができる．これにより，関心事グラフを自動で作ることができる．

(1) これからどういった作業をするか決定し，ルートノード（作業名ノード）を作る．
(2) 作業に関係しそうだと思ったクラスをルートノードに登録していく．
(3) 登録したクラスから表 1 の関係にある要素を適宜グラフに追加していく．

関心事グラフを基盤とした理由は次の 2 点である．第 1 に，関連情報を推薦するにあたってプログラム要素間の関係を活用したいと考えたからである．この関係を用いることで，たとえば，あるメソッド内でどういったライブラリが使われているか，という情報の取得が容易となる．第 2 に，大量のデバッグ履歴を扱いたいと考えたからである．活用できる履歴が多くなれば，欲しい情報が発見できる可能性は高くなると期待できる．一方で，過度に詳細なプログラム情報を保持すると推薦に時間がかかりすぎるという問題が生じる．関心事グラフではプログラム情報を抽象化することで詳細な情報を減らす工夫をしている．具体的には，ローカル変数に関する情報は含まれない．またローカルな制御フローやデータフローも考慮していない．さらに異なるインスタンスによる同じメソッドの呼び出しや同じフィールドの参照はそれぞれ区別していない．

3.2 DCG 構成要素

DCG はプログラム要素および**表 2** に示した 3 種類をグラフの頂点とする有向グラフである．頂点間の辺には関係があり，プログラム要素間では関心事グラフで定義された関係を利用する．そしてプログラム要素と新規に追加された 3 頂点間では単純な関連を利用する．これにより，たとえば「あのテストで例外が発生した時点のメソッド」や「修正の際によく参考にされているメソッド」などの検索が容易となる．関心事グラフのみでは，テスト結果や異なるバージョン間の情報は持たないため，このような検索を実現することは難しい．

DCG で新規に追加された 3 頂点である，diff-n 頂点，テスト結果頂点，およびバグ修正パターン頂点について順に説明する．それぞれについて具体的に**図 5** の DCG を用いて説明する．図 5 の DCG は readFile のテスト testReadFileFileNotExist に失敗した直後から成功した直後までの DCG を表している．

diff-n 頂点

まず，diff-n 頂点について説明する．DCG ではデバッグに関係した情報のみを収集するために，テストに失敗した時点をデバッグの開始とし，テストに成功した時点をデバッグの終了と定義する．これは，テストに失敗した際に

表 2 DCG で新たに追加された頂点
Table 2 Graph nodes introduced in DCG.

頂点	説明
diff-n	関連したクラスがデバッグ作業の n 回目のテスト実行に関係していたことを表す
テスト結果	関連したテストがこのテスト結果であったことを表す
バグ修正パターン	関連したメソッドがこのバグ修正パターンの修正を行ったことを表す

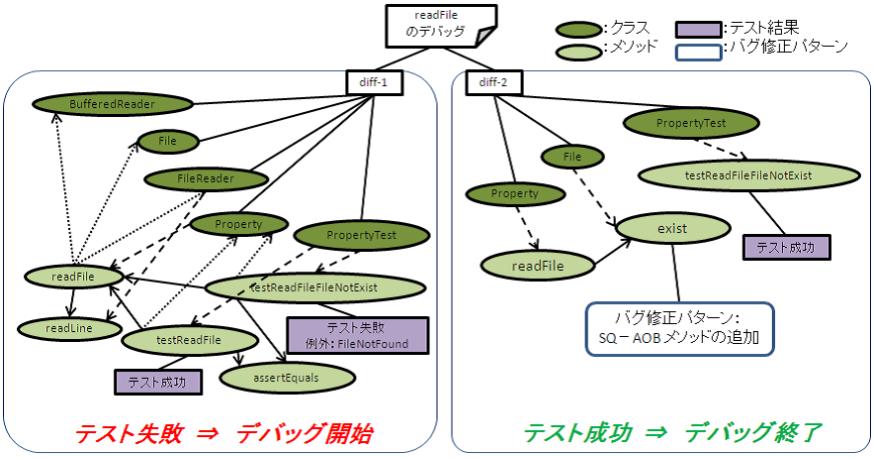

図 5 testReadFileFileNotExist テストの失敗から成功までの DCG
Fig. 5 A DCG recording debugging activities (testReadFileFileNotExist).

表 3 DCG で取り扱うバグ修正パターン
Table 3 Bug fix patterns introduced in DCG.

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

はプログラムにバグが含まれている可能性が高く，またテストに成功した際にはそれが修正されている可能性が高いからである．図 5 の左側上の「diff-1」と書かれた頂点は，下位の頂点がデバッグの開始時，つまりテストに失敗した時点で関係していたことを表す．同図の「diff-2」頂点は，下位に頂点がデバッグの終了時に関係していたことを表す．ここでは説明のため，diff-1 と diff-2 の間でメソッド呼び出しの追加という修正を行ったものとしている．diff-n 頂点の上位には関心事グラフ同様に対象とする作業が，下位にはクラスが関連づけられる．

テスト結果頂点

次に，テスト結果頂点について説明する．図 5 の「テスト成功」,「テスト失敗 例外：FileNotFound」と書かれた頂点は，テストに関連づけられる頂点で，ある時点におけるそのテストのテスト結果を表している．テスト結果頂点はただ 1 つのテストに対応づけられる．図では testReadFile に対し「テスト成功」という頂点が関連づけられ，testReadFileFileNotExist に対し「テスト失敗 例外：FileNotFound」が関連づけられている．

バグ修正パターン頂点

最後に，バグ修正パターン頂点について説明する．図 5 の「バグ修正パターン：SQ-AOB メソッドの追加」と書かれた頂点は，メソッドに関連づけられる頂点で，diff-1 から diff-2 の間でそのバグ修正パターンに分類される修正を行ったことを表している．図では diff-1 から diff-2 の間で File クラスの exist メソッドを readFile メソッドに追加したことを表している．

バグ修正パターンは文献 [12] 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やすい修正を 27 種類に分類している．文献 [12] ではバグ修正全体の約 4 割〜6 割が分類可能であったと述べられている．DCG ではその中でも特に**表 3** に示した 4 パターンを取り扱う．この 4 パターンが他のパターンに比べ頻出するパターンであり，特にプログラマが間違えやすいというデータがあるからである [12]．他のパターンとして，インタフェースの変更であったり，クラスに新たなフィールドを追加したりするなどのパターンもあるが，特定のクラスに強く依存した変更は異なる他のクラスやメソッドでの再利用は難しいと考える．また，if 文に関係したパターンも提案されているが，

このパターンは出現頻度が非常に多く，かつ特徴に乏しいという欠点がある．先の 4 パターンは，いずれもメソッド内という限られた範囲内のパターンであり，しかも 1 つのパターンに対し必ず 1 つのメソッドが対応するという特徴があり取扱いが容易である．

本章では，DCG の基本コンセプトおよびその構築過程について説明した．DCG はデバッグの過程を蓄積するため，これを再利用することにより開発者が同じようなバグに遭遇した際に有用な推薦を行うことが可能となる．一方，バグ修正には自プロジェクトの過去の修正だけでなく，オープンソースリポジトリなどで公開されている他プロジェクトの修正からの推薦も有効である．次章で紹介する dcNavi では，DCG の構築とそれに基づいた推薦だけでなく，既存のプロジェクトから DCG を生成する機能も提供している．

4. 関心事指向推薦システム dcNavi

本章では，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ており，DCG 構築，推薦，およびリポジトリからの DCG 自動生成の 3 機能を持つ．以下順にそれぞれの機能と実現方法について述べる．

4.1 DCG の構築

図 6 にツールによって生成された先ほどの図 5 に対応する DCG を示す．JUnit の実行を監視し，あるテストに失敗したときから成功するまでの区間で発生した情報を蓄積していく．したがって，テスト駆動開発のような頻繁にテスト実行をともなう開発を支援対象とする．なお，利用性を考慮し，ツリーを用いてグラフを表現している．

ツール上の最上位の頂点が図 5 の「readFile のデバッグ」のルートノードに対応する．readFile のデバッグを行っている間は，この頂点をルートにし下位に情報が追加されていく．ルートにはデバッグを開始した日時，プログラマ名，タスク名，プロジェクト名およびリビジョンを表示している．ここでのリビジョンとはテスト実行ごとにカウントされる番号で，77 という数字はこの頂点が作られた際の JUnit の総実行回数を表している．

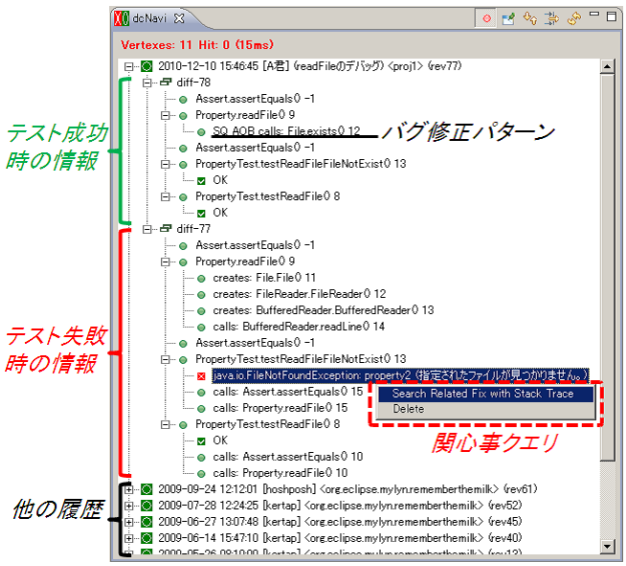

図 6 dcNavi よって生成された DCG
Fig. 6 A DCG generated by dcNavi.

テスト結果頂点は，その結果を出力したテストの下位に表示している．図では diff-77 の下位の PropertyTest.testReadFileFileNotExist() の下位に例外 FileNotFound が追加されている．

プログラム要素は diff-n 頂点直下に表示している．これらの要素の取得は，Mylyn の API を利用している．具体的には，作業に関連したクラスと，そのクラスに対して declares の関係にあるメソッド，およびフィールドが取得できる．残りの calls/creates/catches 関係にあるメソッド内部の要素は，Mylyn で取得したメソッドを解析することで取得している．なおツール上では取得されたメソッドが宣言されているクラスはメソッド名の左に表示するようにしている．

バグ修正パターンは，そのパターンに該当する修正を行ったメソッド名の左に表示するようにしている．たとえば，図では diff-78 の下位の Property.readFile メソッド内で File.exist() の追加（SQ-AOB）を行ったことを示している．

利便性の向上を図る機能として，ツール上でプログラム要素をダブルクリックすると，最新の時点での対応するソースコードにジャンプできる．たとえば，Property.readFile() をダブルクリックすると Property クラスの readFile メソッドが宣言されている箇所にジャンプできる．また，収集された時点での古いリビジョンのソースコードを参照することもできる．今回のデバッグ以前に生成された履歴はツールの下の方に表示している．

4.2 推薦

4.2.1 提供する推薦機能

dcNavi は**表 4** に示した推薦機能を提供している．たと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がかりとなりうると期待できる．他に，たと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やすい例と，そのときの修正方法を推薦する．

4.2.2 ツール機能

dcNavi における推薦手順をツールの機能面から紹介する．ここでは，表 4 の「例外やテスト結果を活用できない」

表 4 dcNavi が提供する推薦機能一覧
Table 4 Recommendation facilities supported by dcNavi.

問題の状況	入力	出力（推薦）
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
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
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

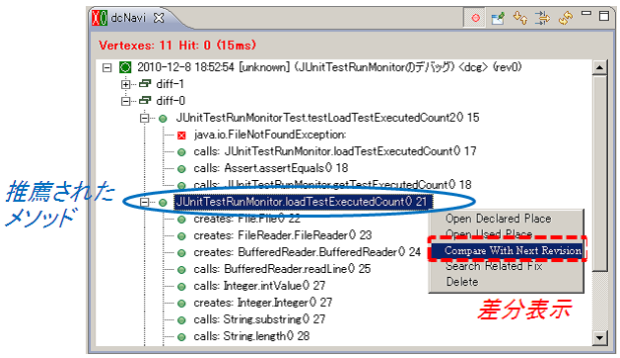

図 7 推薦結果
Fig. 7 Recommendation result.

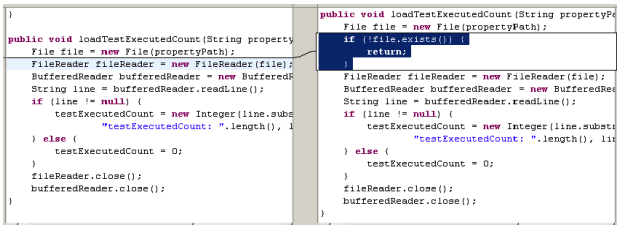

図 8 推薦された関連ソースコード修正前（左）・修正後（右）
Fig. 8 Recommended source code (left: before modification, right: after modification).

という関心事に対する推薦を例に取り上げる．

図 6 では例外を入力として選択している．関心事に応じた検索クエリなので関心事クエリと呼ぶことにする．これを実行すると，図 7 に示すように，例外に関連したプログラム要素が表示される．そして，推薦されたプログラム要素を選択し，前後の差分表示を実行するとソースコードの修正前後（図 8）が閲覧できる．推薦されたコードでは readFile メソッドと似たファイル操作をしている．このソースコードでは同様の例外が発生したため対策として修正後ではメソッド呼び出しが追加（SQ-AOB）されていることが分かる．またバグの原因は file に問題があったことも分かる．このように推薦結果は，誤りの原因を推測する材料として，そして具体的な修正方法を考える際に役立つ．

4.2.3 推薦アルゴリズム

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すべて取得する．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すべて取得する．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（プログラム要素数）の割合で定義する．対象となるプログラム要素は，クラス，フィールド，メソッドである．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．

4.2.4 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 [12]）と見なし，この Hunk 内に含まれる抽象構文木（AST）ノードどうしの比較によりバグ修正パターンを求める．たとえば，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%のバグ修正を分類できた．

表 5 対象オープンソース
Table 5 Open source projects (evaluation data set).

プロジェクト	開発期間	NOR（NOB）	LOC
google code	2009-12-16～2010/5/14	18(2)	2,743
industrial mylyn	2010-05-20～2010-07-09	50(2)	10,953
Mylyn Mntis Connector	2007-02-22～2010-07-14	537(107)	18,536
Origo	2006-12-22～2010-10-08	3,761(867)	5,769
qcMylyn	2009-08-06～2010-09-26	223(80)	13,117
Redmine Mylyn Connector	2008-05-26～2010-05-19	423(134)	14,729
Remember The Milk	2008-01-26～2009-11-24	70(13)	7,259
Scrum Vision	2008-05-24～2010-07-12	431(14)	43,263
subclipse	2003-06-20～2010-10-14	4,745(923)	167,100

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

4.3 リポジトリからの DCG 自動生成

dcNavi では，過去の修正履歴がないと推薦を行うことができない．これは，dcNavi を導入しても，履歴情報が蓄積されるまでは，推薦機能の恩恵にあずかれないことを意味する．また，ゼロから DCG を構築するのはコスト面からも現実的ではない．

この問題を解決するため，dcNavi では，オープンソースリポジトリなどの既存データから自動的に DCG を生成する機能を提供している．5 章の実験評価も，この機能を用いて実施した．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．

ただし，DCG を自動生成するにあたって，2 つの問題を解決する必要がある．1 つは，デバッグの開始と終了をどう判定するかという問題である．もう 1 つは，関心事グラフに追加する要素をどう判定するかという問題である．通常は Mylyn により IDE から自動追加されるが，自動生成の場合は何らかの方法を考える必要がある．1 つ目の問題に対しては，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デバッグ終了のリビジョンと判断し，その直前のリビジョンをデバッグ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 [12] で紹介された方法である．2 つ目の問題に対しては，デバッグの対象となった要素のみを DCG に追加することで対応した．

5. 評価実験

本章ではオープンソースを対象として行った dcNavi の評価実験について説明する．今回の実験の目的は，dcNavi による推薦の有効性と課題について初期的な評価を行うことである．課題については，今後どのような改善を行えばよいのか，考察した．

5.1 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 5** 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行った．

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 6** に示す．実験は表 6 の 1 台のマシンですべて行った．

5.2 推薦の質と正解

5.2.1 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．

5.2.2 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：**たとえば両者がともに MC-DAP のパターンに分類される修正を行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：**たとえば両者がともに 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．ここではメソッドどうしの一致を，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

表 6 マシンの環境
Table 6 Evaluation environment.

項目	内容
Operating System	Windows XP Home Edition (5.1, Build 2600) Service Pack 3
Processor	Intel(R) Atom(TM) CPU N280 @ 1.66 GHz (2 CPUs)
Memory	2,038 MB RAM
Eclipse	Version: 3.5.2
dcNavi	Version: 0.1.0

表 7 実験結果
Table 7 Experiment result.

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 8 プロジェクトごとの実験結果
Table 8 Experiment result per project.

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

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，より修正の参考になりう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．さらにそのうちの 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 つの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した．1 回のコード変更で，複数のバグを修正した場合，このような状況が発生する．本来であれば，個々のバグ修正に対し，正解か否かを判定すべきであるが，dcNavi における「リポジトリからの DCG 自動生成」機能の制約上，複数のバグ修正を区別することは難しい．また，一般的に考えても，バグ修正の粒度をどのようにとらえるかは難しい問題である．バグ修正の粒度によりバグ数が変化するため，それが実験評価結果に影響を与える可能性はある．それがどの程度なのかは，今後の研究課題である．

5.3 結果

9 プロジェクトを平均した実験結果を**表 7** 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 7 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．同じ Mylyn 関係のオープンソースでも，あまりバグの発生がないもの，サイズが小さいものも存在することを示すため，表には掲載している．なお，N/A の部分は，9 つのプロジェクトの平均の評価値には影響を与えない．

表 8 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．

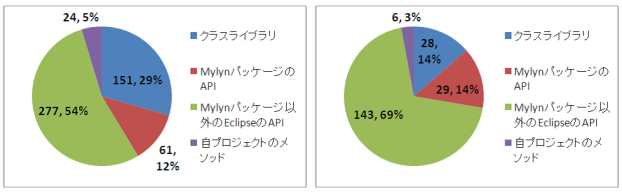

図 9 正解集合（左）と適合集合（右）の分布
Fig. 9 Answer set (left) and correct recommendation set (right).

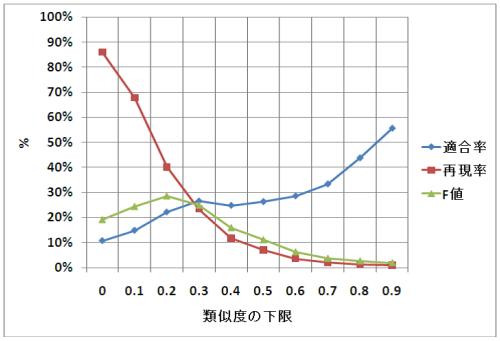

図 10 類似度の下限を変化させた場合の推薦の質の変化
Fig. 10 Relation between recommendation quality and similarity (lower limit).

5.4 考察

ここでは，推薦の質について，正解集合および適合集合のドメイン上の分布，正解と類似度の関係，ランキング表示の課題，の各視点から考察する．

5.4.1 正解集合および適合集合のドメイン上の分布

正解集合および適合集合のドメイン上の分布の確認を行った．正解集合とは実験時においてあらかじめ分かっている正解からなる集合で，適合集合とは推薦結果のうちの正解からなる集合である．

正解集合および適合集合のドメイン上の分布を**図 9** 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と全体に占めるパーセンテージを表す．

ドメインは以下の 4 つに分類している．

(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.*）

正解集合では 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．さらに，図 9 を見ると，Eclipse 関係の API が占める割合が大きいことが分かる．正解集合の 29%，適合集合の 69%を占める．今回の実験対象となったのは Mylyn 関係の Eclipse プラグインであるが，Mylyn そのものよりは開発プラットフォームである Eclipse 関係の方がバグ修正に関するノウハウを必要としていることがうかがえる．一方，今回の実験対象では，自プロジェクトの誤りは全体的に多くなかった．なお，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うることが分かった．

上記のドメインの分類では，(1) のクラスライブラリが汎用，残りの (2) から (4) が「Mylyn + Eclipse プラグイン」に特化したものに大きく分けることができる．正解集合の場合，前者が 29%，後者が 71%の割合となる．一方，適合集合の場合は，前者が 14%，後者が 86%となる．これは，汎用的なものよりは，特定のドメインに特化したリポジトリからの推薦の方が有効であることを裏付けている．

実際，これらの分析結果は，既存の類似実験結果 [8] とも符合する．文献 [8] では，汎用性の高い/低いコンポーネントに対する協調フィルタリングを用いた推薦の評価実験を行っている．協調フィルタリングによる推薦は，従来のランダム推薦法や平均値に基づく推薦法よりも，汎用性の低いコンポーネント（特定のドメインに特化したコンポーネント）に対して，特に効果があることが確かめられている．dcNavi による推薦方針は「バグ修正パターンが一致する類似コードどうしはバグ修正内容も似ている」という考え方に基づいており，協調フィルタリングの考え方に近い．

今回の評価実験で得られた結果は，今後，推薦の質を向上させていくうえで有効である．たとえば，クラスライブラリなど利用方法が十分に分かっているメソッドは類似度計算の際に排除したり，あるいは推薦候補から外したりする方法が考えられる．推薦の質向上が期待されるだけでなく，推薦のための DCG 探索の時間を削減できる．

5.4.2 正解と類似度の関係

正解と類似度の関係を調べるために，**図 10** 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 7 および表 8 に示したように類似度下限 0.2 で評価した．

類似度 0.2 という値は一般的なプログラマの感覚からするとかなり低い値である．これは，現状の dcNavi における類似度の計算方式に起因する．dcNavi では，メソッド単位で類似度を計算するため，たとえば，ある API 呼び出しの行数が 1 行で，そのバグの修正に過去の事例を適用する場合，必ずしも類似度は高くならない．なぜなら，API 呼び出し以外のメソッドの内容が異なるからである．汎用的なクラスライブラリ利用に関するバグ修正ではこのような状況が発生しやすい．図 9 を見ると，正解集合におけるクラスライブラリの比率は 29%でこの値は無視できない．

推薦の質を向上させていくには，汎用性の高いクラスライブラリとそれ以外に分け，前者については類似度という

指標を使用しないという方法が考えられる．これは，前出の文献 [8] の見解に沿う考え方であり，前項で述べた今後の改善案とも共通する．今後の課題としたい．

5.4.3 ランキング表示の課題

現在，dcNavi では，類似度の高い順に推薦している．多くの正解は上位に表示されるので，プログラマは，基本的にランキングの上位から修正案の候補を確認してゆけばよい．表示数は変更可能で，デフォルトで最大 30 個まで表示している．ただ，その一方で，現状の方式には，1) 類似度の値のみでランキング表示するため，同じような推薦結果が連続して表示される場合がある，2) 正解がランキングの下位に存在する場合がある，3) 現状のデフォルト表示数は 30 であるが，どの程度の個数が適切なのか明確ではない，という課題がある．

現状の方式は，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，類似した事例は代表例と頻度のみを提示してもらった方がプログラマも修正候補として利用しやすくなる．

グループ推薦によるランキング表示が可能になると，上記の課題は緩和されると思われる．どのように事例をグループ化して提示すべきか，すなわち，各推薦候補をどのようにポートフォリオとして与えるか，は今後の重要な研究テーマの 1 つである．類似度，バグ修正パターン，ドメインの種別などによりポートフォリオ化することが考えられる．

6. 関連研究

本章では，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ともに，dcNavi との関連について述べる．

6.1 BugMem

BugMem [7] 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，デバッグ支援を行う．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 どうしの比較の際に，様々なレベルで標準化を行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どうしの類似度を考慮し推薦を行った．

6.2 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，呼び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どうし（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．

6.3 Whyline

Whyline [9] 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行っている．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．これによりプログラマ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．

6.4 CAR-Miner

CAR-Miner [15] ではあるメソッドが利用された場合の，そのメソッドに関係した例外処理パターンの発見方法を提案している．このパターンを用いることで，あるメソッド列が出現した場合にはどういう例外処理が必要になるかを知ることができる．CAR-Miner ではアプリケーション全体からパターンを探し出す．バージョン管理システムなどの履歴情報は活用していない．そのためまったく履歴がない状態でも利用できる．また学習データは Google code search [4] のような CSE（code search engine）から検索キーワードを入力しそこから取得するため，プロジェクトの初期段階でデータが十分にない場合でも利用できる．dcNavi では，再利用可能なパターンを見つける際に，ソースコード中のプログラム要素の出現順序については特に考慮しなかった．DCG はデバッグ作業のプロセスで発生した情報を順々に蓄積していく．より類似したメソッドを検

索したい場合，あるいは修正されたコード周辺の関係を見たい場合に出現順序に関する情報は有効となると考える．今後の課題である．

6.5 Codebook

Codebook [2] ではソフトウェア開発における，開発物や開発者などの様々な情報を関連付けた有向グラフを提案している．グラフはバージョン管理システムや開発者のディレクトリ，あるいは E-mail などに対し，それぞれクローラを走らせて情報を収集し構築していく．ソフトウェア開発チーム内でのコミュニケーションを支援することを目的に提案された．似たツールである Mylyn では統合開発環境をベースとし作業に関連した情報の収集を行うが，Codebook では開発環境など特に意識せずに情報を集め専用のデータベースに保存する．我々は参照したドキュメントなどの導入も当初検討していたが，よりプログラム解析に近いグラフ構造の方を選択した．今後 Mylyn で可能な範囲で，たとえば Eclipse 内のブラウザで参照したページの収集など，ソフトウェア開発の周辺の情報も取り扱ってみたいと考えている．

6.6 FixWizard

FixWizard [11] では，過去の修正情報をもとに，修正対象コードに関連する推薦方法を提案している．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と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）も修正されうるとしている．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うる」を掲げ実験をもとに実証している．また，同じような機能を持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．

FixWizard も dcNavi も「似たような修正」をリポジトリから探索し，それをデバッグ時のヒントとして提示している．しかしながら，dcNavi では，コードの類似性のほかにバグ修正パターンも考慮している点が異なる．バグ修正パターンはバグの原因を示唆しており，これはコードの類似判定のみからは推定できない．ただ，dcNavi では FixWizard のように類似性の判定にオブジェクトの機能まで考慮しておらず，この部分については，今後 dcNavi を改良してゆく際の参考にしてゆきたい．

7. おわりに

本論文では，デバッグ支援のための関心事指向推薦システム dcNavi を提案した．dcNavi を利用することにより，現在直面しているバグに応じた情報推薦を得ることが可能となった．現状では API の利用の誤りなどのデバッグを中心に本提案手法は効果を発揮できると考えている．今後は，テスト実行などの実行履歴などを取り入れてゆくことで，論理エラーのような細部のバグにも対応してゆきたいと考えている．

また，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どうしの比較である程度の質の推薦が可能であった．

今後より幅広いドメインに対する実験を考えている．また，今回の実験対象となったオープンソースは基本的にソフトウェア開発の相当な経験がある開発者が中心となっていると思われる．初心者が多いような開発では，異なった結果になることは十分に考えられる．

謝辞 本研究の一部は，文部科学省科学研究補助費（基盤 B：課題番号 23300010）による助成を受けた．

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, *Proc. 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] Beqel, A., Khoo, Y.P. and Zimmermann, T.: Codebook: Discovering and Exploiting Relationships in Software Repositories, *Proc. 32nd ACM/IEEE International Conference on Software Engineering* (*ICSE 2010*), pp.125–134 (2010).

[3] Budge, S., Nagappan, N., Ramalingam, G. and Rajamani, S.: Global Software Servicing: Observational Experiences at Microsoft, *Proc. IEEE International Conference on Global Software Engineering* (*ICGSE 2008*), pp.182–191 (2008).

[4] Google Code Search Engine (2006), available from ⟨http://www.google.com/codesearch⟩.

[5] JUnit, available from ⟨http://www.junit.org⟩.

[6] Kersten, M. and Murphy, G.C.: Mylar: A Degree-of-interest Model for IDEs, *Proc. 4th International Conference on Aspect-oriented Software Development* (*AOSD2005*), pp.159–168 (2005).

[7] Kim, S. et al.: Memories of Bug Fixes, *Proc. 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE 2006*), pp.35–45 (2006).

[8] 亀井靖高，角田雅照，柿元 健，大杉直樹，門田暁人，松本健一：ソフトウェアコンポーネント推薦における協調フィルタリングの効果，情報処理学会論文誌，Vol.50, No.3, pp.1139–1143 (2009).

[9] Ko, A.J. and Myers, B.A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, *Proc. 30th International Conference on Software Engineering* (*ICSE 2008*), pp.301–310 (2008).
[10] Mylyn, available from ⟨http://www.eclipse.org/mylyn/⟩.
[11] Nquyen, T.T., Nquyen, H.A., Pham, N.H., Al-kofahi, J. and Nquyen, T.N.: Recurring Bug Fixes in Object-oriented Programs, *Proc. 32nd ACM/IEEE International Conference on Software Engineering* (*ICSE 2010*), pp.315–324 (2010).
[12] 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).
[13] Robillard, M.P. and Murphy, G.C.: Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, *Proc. 24th International Conference on Software Engineering* (*ICSE 2002*), pp.406–416 (2002).
[14] Shiozuka, M., Ubayashi, N. and Kamei, Y.: Debug Concern Navigator, *Proc. 23rd International Conference on Software Engineering and Knowledge Engineering* (*SEKE 2011*), pp.197–202 (2011).
[15] Thummalapenta, S. and Xie, T.: Mining Exception-handling Rules as Sequence Association Rules, *Proc. 31st International Conference on Software Engineering* (*ICSE 2009*), pp.496–506 (2009).

塩塚 大

2009 年九州工業大学情報工学部知能情報工学科卒業．2011 年同大学院情報工学府情報科学専攻修士課程修了．現在，メルコ・パワー・システムズ株式会社に勤務．ソフトウェアテストやデバッグ手法に興味を持つ．

鵜林 尚靖 （正会員）

1982 年広島大学理学部数学科卒業．1999 年東京大学大学院総合文化研究科広域科学専攻広域システム科学系博士課程修了．博士（学術）．1982〜2003 年（株）東芝に勤務．2003 年九州工業大学情報工学部助教授，2010 年九州大学大学院システム情報科学研究院教授，現在に至る．2003 年度情報処理学会山下記念研究賞受賞．ソフトウェア工学，プログラミング言語モデルに興味を持つ．ACM，IEEE-CS，日本ソフトウェア科学会，電子情報通信学会各会員．

亀井 靖高 （正会員）

2005 年関西大学総合情報学部卒業．2007 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．2009 年同大学院博士後期課程修了．同年日本学術振興会・特別研究員（PD）．2010 年カナダ Queen's 大学博士研究員．2011 年九州大学大学院システム情報科学研究院助教．博士（工学）．ソフトウェアメトリクス，マイニングソフトウェアリポジトリ等の研究に従事．IEEE，IEICE 各会員．