編集履歴から算出した開発者の関心度に基づくコード補完

Liao Ziyang　丸山　勝久

近年の統合開発環境は，過去に記述された大量のソースコードから得られる集合知を活用することで，開発者が作成中のソースコード記述において使用する可能性の高いメソッドをより優先的に推薦するコード補完機能を提供している．このような統計データに基づく手法は開発者全体の傾向に合わせたコード補完に適している反面，開発者個人に対して適切な補完候補を提供できるとはかぎらない．本論文では，開発者が過去にソースコードを記述した際の編集履歴を活用することで開発者の関心度を算出し，それに基づき開発者に提示される補完候補の順番を並び替える手法を提案する．この手法を Eclipse プラグインとして実装したシステムを用いることで，それぞれの開発者に適したコード補完が実現でき，プログラム作成における生産性の向上が期待できる．

1　はじめに

ソースコードを記述する際，開発者は効率よくプログラムを作成するために，ライブラリやフレームワークに存在するクラスやメソッドを再利用して開発を行う．しかしながら，開発者がすべてのクラス名やメソッド名をあらかじめ覚えておくことは不可能である．このため，どのようなクラスやメソッドがフレームワークやライブラリに存在するのかを，開発者は再利用時に調べる必要がある．文献[7] では，開発者が Web を利用して検索するクエリーの 34.2%は API（Application Programming Interface）に関するものであり，そのようなクエリーの 64.1%が実際の API 名と違う名前で検索されているという事実を報告している．さらに，再使用したいクラス名やメソッド名が長いと，それを正確に入力するのは面倒である．クラスやメソッドの再利用を促進するためには，クラスやメソッドを探す手間やそれらの名前を入力する手間を削減する必要がある．

これに対して，多くの統合開発環境（IDE: Integrated Development Environment）では，コード補完を主要な機能のひとつとしている[10] [13]．たとえば，Eclipse は，特定のインスタンスを格納した変数を入力するだけで，そのインスタンスが提供するメソッドの一覧を提示するコード補完を提供している．開発者は，利用したいメソッドの呼出しコードを正確に入力する必要はなく，提示された補完候補の中から適切なメソッドを選択するだけでよい．本論文では，このようなインスタンスへの参照をトリガーとしたメソッド呼出しに関するコード補完を対象とする．

ここで，Eclipse が従来から備えているコード補完では，参照されるインスタンスの型（インスタンスが属するクラス）に基づき，単純にメソッドの一覧を提示する．このため，選択される可能性の高いメソッドがより優先的に（メソッド選択ウィンドウにおいて上位に）提示されるとはかぎらない．このため，膨大な数のメソッドを提供しているインスタンスが参照された場合に，開発者が適切なメソッドを選択する手間は大きい[2] [5]．

このような背景を踏まえ，データマイニング技術を用いてメソッド呼出しコードを補完する手法やシステムがいくつも提案されている[1] [4] [8] [9] [11] [14]．これらの手法やシステムでは，大量のソースコードを集合知として扱うことで，一般的な状況下において記

Code Completion Using Developer's Interests based on Edit History.

Ziyang Liao, Katsuhisa Maruyama, 立命館大学情報理工学部, Department of Computer Science, Ritsumeikan University.

述される可能性の高いコードを予測し，その予測に基づきコードを補完する．このような集合知の活用は，開発者全体の傾向に合わせたコード補完の実現に適している．しかしながら，特定の開発チームや開発者個人に対するコード補完という観点からは必ずしも高い精度が得られるとはかぎらない．

本論文では，開発者がソースコードを作成した際の編集履歴を活用することで，開発者の特性や状況をリアルタイムに反映する仕組みを組み込んだコード補完手法と，それを Eclipse プラグインとして実装したシステムを提案する．本手法では，開発者の特性や状況を，開発者が特定のメソッドをどのように使用しているかという視点から算出した関心度で捉える．その上で，それぞれのメソッドに対する個々の開発者の関心度に基づき，コード補完時に選択される可能性の高いメソッドをより優先的に提示するように，メソッド呼出しにおける補完候補の一覧を並び替える．本システムを用いることで，開発者に適したコード補完が実現でき，プログラム作成における生産性の向上が期待できる．

以下，2 章で Eclipse に搭載されているコード補完システムを紹介し，それらの問題点を述べる．その後，3 章で開発者の関心度に基づくコード補完手法とそのシステムを提案する．最後に，4 章でまとめと今後の課題を述べる．

2 コード補完システム

本章では，既存のコード補完システムを紹介し，その問題点を示す．

2.1 既存のコード補完システム

ここでは，Eclipse に搭載されている Content Assist と Code Recommenders を紹介する．

2.1.1 Content Assist

Eclipse Content Assist（以降，ECA）[6] は，開発者が記述中のコードに応じて，それに関連するコード補完を支援する．Eclipse の Java エディタにおいて，特定のコード上にカーソルを置き，「Ctrl + Space」を押すことでコード補完が起動する．起動によりポップアップウィンドウ上に補完候補の一覧がアルファベット順に提示されるので，補完したいコードを選択する．選択されたコードが，エディタのカーソルの位置に挿入される．

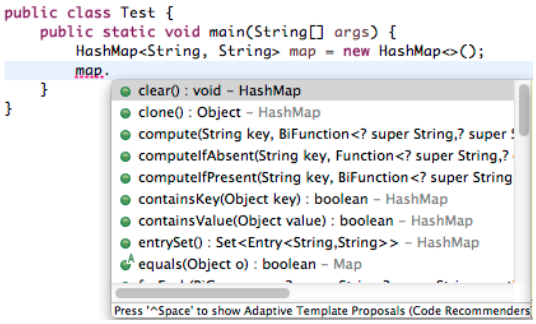


図 1 Content Assist の動作例

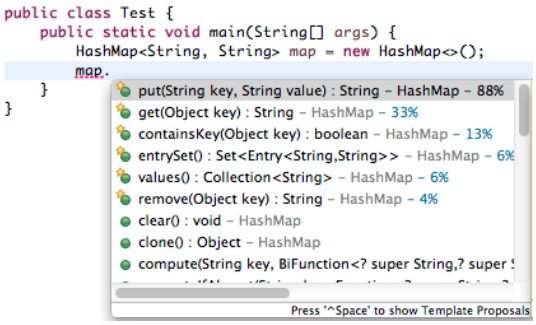


図 2 Code Recommender の動作例

図 1 に，ECA の動作例を示す．変数 `map` に格納されている `HashMap`（`java.util.HashMap`）クラスのインスタンスへの参照に対して，そのクラスが提供するメソッドの一覧が補完候補として提示されている．

2.1.2 Code Recommenders

Eclipse Code Recommenders（以降，ECR）[4] では，既存のソースコードに登場するメソッドの利用パターンの統計データに基づき，補完候補の順位付けを行う．これにより，開発者がより選択する可能性の高いメソッドを補完候補の上位に提示する．

図 2 に，ECR の動作例を示す．図 1 とは異なり，`HashMap` が提供するメソッドの一覧が順位付けされている．

2.2 問題点

ECA のようなアルファベット順に提示するコード補完では，開発者に対する支援として十分とはいえない．たとえば，図 1 と図 2 に示した HashMap には 24 個のメソッドがあり，親クラスのメソッドやフィールドも含めると，提示される補完候補は全部で 33 個となる．同様に，String（java.lang.String）クラスには 67 個のメソッドがあり，提示される補完候補は全部で 76 個となる．このような状況において，ECA では，開発者が利用したいメソッドが下位に提示されることが頻繁に発生し，そのメソッドを選択するために多くの時間を費やすことになる．

これに対して，ECR では，メソッドの利用パターンの統計データに基づき，選択される可能性が高いメソッドをより上位に提示する．これにより，ECA において問題であった，メソッドの選択に費やす手間を削減することが可能となる．たとえば，ECA において 20 番目に提示される HashMap の put(String, String) メソッドは，ECR において 1 番目に提示される（図 2 を参照）．また，String において頻繁に利用される indexOf(int) メソッドは 2 番目に表示される（ECA では 23 番目に提示される）．

一方，ECR にも問題が残されている．コード補完を実現するためには統計的データが必須であり，現時点では Eclipse API と Java 標準 API の一部しかコード補完の対象としていない．つまり，普及していないライブラリや開発者が独自に作成したライブラリに対して，ECR は役に立たない．また，用意されている統計データはすべての ECR に共通であり，そのデータが個々の開発者の特性や状況を代表しているとは限らない．

一般的に，プログラミングの習慣や好みは空間的な局所性をもち，同じ開発者は特定のクラスに対して同じメソッドを使用する傾向にある．反対に，異なる開発者は，たとえ同じクラスを利用していたとしても，異なるメソッドを使用する可能性がある．たとえば，String のメソッドを呼び出して同じ文字列処理を実現する方法はいくつも存在し，開発者によって利用するメソッドが異なることはめずらしくない．このような場合において，開発者を区別せずに利用パターンを統計的に処理すると，開発者間の特性の違いが打ち消されることがある．これにより，複数の開発者がそれぞれ頻繁に利用しているメソッドが補完候補の上位に提示されないという望ましくない状況が発生する．

さらに，メソッドの利用は時間的な局所性をもつ．たとえば，ある開発者が（Java Swing 対応の）GUI アプリケーションを作成している場合を考える．その際，同じメニュー（JMenu クラスのインスタンス）に対して，複数のメニュー項目（JMenuItem クラスのインスタンス）を連続的に付け加えることは多い．この場合，JMenu の add(JMenuItem) メソッドが繰り返し利用される．また，複数のボタン（JButton クラスから生成されたインスタンス群）に対して，それらの色やアクションをまとめて設定することも多い．この場合，JButton の setForeground(Color) メソッドや setAction(Action) メソッドが繰り返し利用される．メソッドの利用が連続的であるという局所性は，あるメソッドが利用された場合には，そのメソッドが直後にも利用される可能性が高いことを指している．反対に，利用されなくなったメソッドが再度利用される可能性は低い．コード補完の効果を高めるためには，このような時間的な局所性を考慮する方がよい．しかしながら，ECR は，メソッドの利用に関するソースコード上の連続性は考慮しているものの，時間的な連続性は考慮していない．このため，かつては頻繁に利用されていたが，最近あまり利用されないメソッドが補完候補として上位に提示されるという望ましくない状況が発生する．

以上をまとめると，アルファベット順に補完候補を表示する ECA は，プログラム作成における生産性向上という観点で十分な効果をもつとはいえない．また，メソッドの利用パターンに関する統計データに基づく ECR は，開発者全体の傾向に合わせたコード補完という観点では効果が得られるものの，開発者固有の特性や状況を反映させたコード補完には対応できていない．

3 関心度に基づくコード補完

本章では，従来のコード補完手法の問題点に対処した手法を提案し，その手法を実装したシステムについ

て説明する．

3.1 コード補完手法

提案手法は，2.2 節で述べた問題点を解決するために，既に存在するソースコードではなく，開発者がソースコードを記述した際の編集履歴を用いる．編集履歴とは，開発者が過去に実行した編集操作（開発者を表す識別子，編集が行われた時刻，編集内容）を集めたものを指す [12]．編集操作の取得には，Eclipse プラグインの ChangeTracker [3] を活用する．このプラグインは，Eclipse の Java エディタ上のすべての編集操作をバックグラウンドで自動的に捕捉する．本手法では，エディタ上のコードを直接書き換える編集操作（タイピングによる文字列の挿入および削除，文字列のカット/コピー/ペースト，コード補完や簡易修正による文字列の挿入や削除）のみを利用する †1．

本手法では，編集操作が記録されるたびにそれを分析することで，それぞれのメソッドに対する開発者の関心度をリアルタイムに算出する．コード補完では，関心度の高いメソッドを補完候補の上位に提示する．メソッド m に対する開発者 dev の関心度は，次に示す 2 つの視点に基づき算出する．

(1) 使用頻度 $Freq(dev, m)$

過去の編集において dev が m を使用した回数を表す．メソッドの使用とは，dev が m を呼び出すコードを記述する，あるいは，コード補完により m を選択することを指す．

(2) 経過した編集数 $Edit(dev, m)$

過去の編集において dev が m を最後に使用してから，現時点までに実施された編集操作の総数を指す．

開発者の関心度 $DOI(dev, m)$ は，次に示す式を用いて算出する（$c > 0$，$\alpha \geq 0$，$\beta \geq 0$ である）．

$$DOI(dev, m) = c \times \frac{\sqrt{\alpha + Freq(dev, m)}}{\beta + Edit(dev, m)}$$

一般的に，使用頻度が高いメソッドに対して開発者は強い関心を持つといえる．よって，$Freq$ の値が大きいほど DOI の値が高くなるようにしている．単純な比例関係ではなく平方根を採用したのは，$Freq$ の値が大きくなるにつれ，DOI の値の増加率を抑えるためである．α は $Freq$ による DOI の増加に対する影響を調整する係数である．$\alpha = 0$ のときその影響は最大となり，α の値を大きくするとその影響は相対的に小さくなる．

また，2.2 節で述べたメソッドの利用は局所的であるという前提に基づき，$Edit$ の値が増加するにつれ，DOI の値を減少させることにしている．β は $Freq$ による DOI の減少に対する影響を調整する係数である．$\beta = 0$ のときその影響は最大となり，β の値を大きくするとその影響は相対的に小さくなる．

本システムの実装では，使用頻度にできるだけ敏感となるように $\alpha = 0$ とした．また，予備実験の結果に基づき，経過した編集数が少ないときに急激に関心度が減少しないように $\beta = 100$ とした．c は DOI の値が極めて小さくなる（数値が切り捨てられる）ことを避けるために導入した定数である．本システムの実装では $c = 10,000$ とした．

編集操作から取得した開発者の識別子に基づき，それぞれのメソッドに対する関心度を開発者ごとに独立に算出することで，開発者間の特性の違いが打ち消されることを避けることができる．また，経過した編集数に基づき関心度を算出することで，同じ開発者であっても，ソースコード記述時の状況の違いを考慮したコード補完が可能となる．

3.2 コード補完システム

提案手法を実現するコード補完システムの概要を図 3 に示す．以下に，それぞれのモジュールの詳細を説明する．

3.2.1 メソッド特定モジュール

このモジュールは，ChangeTracker の出力する編集操作を監視し，それをリアルタイムに分析することで，開発者が使用（記述あるいは選択）したメソッドを特定する．同時に，現時点での編集操作の総数を数え，特定したメソッドが最後に利用された時点を表すシーケンス番号を記録する．

ChangeTracker には，個々の編集操作をイベントと

†1 ファイルのオープンやクローズ，メニュー項目の選択などの編集操作は無視する．

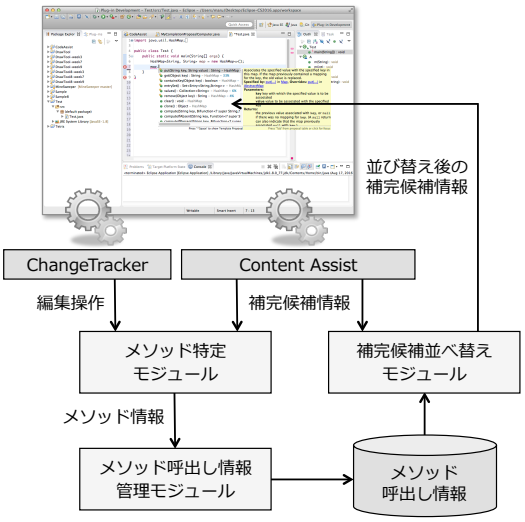


図 3　システムの概要

してリアルタイムに通知するモードと，編集操作を履歴としてまとめてファイルに出力するモードがある．本システムでは通知モードを利用して，編集操作を受け取る．この編集操作には，開発者を表す識別子，編集が行われた時刻，編集が行われた Java ソースファイルのパス，ソースコードにおける編集箇所を指すオフセット値，挿入文字列，削除文字列の情報が記録されている．

さらに，本システムでは，ChangeTracker から受け取った（コードを直接書き換える）すべての編集操作にシーケンス番号を付与する．これは，現時点までに受け取った編集操作の総数を指す．シーケンス番号は，本システム（が組み込まれた Eclipse）が初めて起動される際に 0 に初期化される．システム終了時には現時点での変更操作の総数（最新の変更操作のシーケンス番号の値）をファイルに保存し，以降の起動時にはファイルに保存した前回の値で初期化する．

メソッドの特定は大きく，メソッドの名前候補を検出する処理と，メソッドの名前候補からメソッド情報を取得する処理に分けられる．以降，それぞれの処理を説明する．

(a) メソッドの名前候補を検出する処理

開発者が記述中のソースコードにおいて，メソッド呼出しコードを見つけ，メソッドの名前候補を検出する．Java の文法においては，自分以外のインスタンスのメソッドを呼び出す際，必ずドット演算子（「.」）をメソッド呼出しの直前に記述しなければならない．本システムでは，この記法を利用して，メソッドの名前を検出する．

具体的には，受け取った編集操作の挿入文字列を監視し，それがドット文字の場合，それ以降に挿入された文字列の収集を開始する．メソッド呼出しの引数は必ず右括弧（「(」）で始まるので（引数がない場合でも括弧は存在するので），ドット演算子から右括弧までの間に収集された挿入文字列を結合したものがメソッドの名前候補となる．たとえば，開発者が `str.chars();` と記述した場合を考える．4 文字目のドット文字に関する編集操作を受け取り，挿入文字列の収集を開始する．10 文字目に到達した時点で，右括弧に関する編集操作を受け取るため，その間に収集した文字列 `chars` がメソッドの名前候補となる．

一方，メソッド呼出しコードが補完された場合，そのコードがまとめて編集操作の挿入文字列に格納される．この場合でも，挿入文字列に含まれる右括弧を探すことで，メソッドの名前を取り出すことが可能である．

(b) メソッド情報を取得する処理

検出したメソッドの名前候補から，実際に呼び出されるメソッドを特定し，その情報（メソッド情報）を取得する．本システムは，編集操作の監視においてドット演算子を検出した時点で，ECA が提供する補完候補情報を取得する．これは，`org.eclipse.jdt.ui` パッケージに存在する `JavaAllCompletionProposalComputer` クラスの `computeCompletionProposals()` メソッドの返り値から取得可能である．取得した補完候補情報 $Prop$ に含まれるメソッド名の一覧とメソッドの名前候補 $name$ を比較し，一致するものが見つかった場合，$name$ を名前とするメソッド m の特定が完了する．一致するものが見つからなかった場合には，$name$ を破棄し，編集操作の監視を継続する．

ここで，メソッドの名前だけでなく，すべて

の引数の型まで決定できた場合には，m を一意に特定することが可能である [†2]．しかしながら，記述中の（不完全な）ソースコードにおいて引数の型を決定することは困難である．また，開発者が補完候補からメソッドを選択した場合，メソッドは一意に特定されているはずである．しかしながら，編集操作の挿入文字列だけで，これを判断することはできない．これら 2 つの理由から，本システムではメソッドの名前だけを用いて m を特定している．

このような設計判断により，メソッドがオーバーロード関係を有する場合，$name$ を名前とする複数のメソッドが $Prop$ において一致する可能性がある．複数のメソッドが一致した場合には，それらのメソッドが所属するクラスの情報をそれぞれ $Prop$ から抜きだし，同じクラスに属するメソッドはひとつに統合する．異なるクラスに所属するメソッドは，それらを統合せずに，複数のメソッドとして扱うことにする．

いま，$name$ を名前とする m の所属するクラスの完全限定名を fqn とする．また，m の特定に成功した際の編集操作 op（m を呼び出すコード記述の最初の編集操作，あるいは，補完候補 m が選択された際に文字列を挿入する編集操作）のシーケンス番号を seq とする．このモジュールは，m の特定に成功するたびに，m に関するメソッド情報 $I(name, fqn, op, seq)$ をメソッド呼出し情報管理モジュールに引き渡す．

3.2.2 メソッド呼出し情報管理モジュール

このモジュールは，メソッド特定モジュールから渡されたメソッド情報に基づき，メソッド呼出し情報を収集し，それをデータベース（実際には，マップとファイルで実装）に格納する．メソッド呼出し情報におけるデータ項目の構造を以下に示す．

- 開発者が使用したメソッド M の名前 $Name$
- M が所属するクラスの完全限定名 Fqn
- M を使用した開発者を表す識別子 Dev
- 現時点までに M が使用された回数 $Times$
- M が最後に使用された時点を表す番号 $Latest$
- 現時点での M に対する関心度 Doi
- M の使用履歴（編集操作の集合）$History$

データベースでは，$(Name, Fqn, Dev)$ を検索キーとしてメソッド呼出し情報（このデータ項目を $M[Name, Fqn, Dev]$ とおく）を管理する．$Name$ と Fqn はメソッド特定モジュールから渡されるメソッド情報 $I(name, fqn, op, seq)$ から抜き出したメソッドの名前 $name$ とそのメソッドが所属するクラスの完全限定名 fqn である．Dev は，編集操作 op に保存されている開発者を表す識別子を指す．検索により，データベースに $M[Name, Fqn, Dev]$ が見つからない場合には，$(Name, Fqn, Dev)$ を検索キーとするデータ項目を新規に作成し，その $Times$ と $Latest$ の両方に初期値 0 を格納する．これにより，データベースには必ず $M[Name, Fqn, Dev]$ が存在することが保証される．

いま，開発者 d が利用したメソッド m（名前を n，クラスの完全限定名を f とする）のメソッド呼出し情報を格納したデータ項目を $M_q = M[n, f, d]$ とおく．さらに，M_q の $Times$ と $Latest$ の値をそれぞれ T_q と L_q とする．3.1 節で述べた使用頻度と経過した編集数は，次の式を用いてそれぞれ算出する．

使用頻度 $Freq(d, m) = T_q + 1$

経過した編集数 $Edit(d, m) = Seq - L_q$

Seq は $I(name, fqn, op, seq)$ から抜き出したシーケンス番号 seq を指す．

このようにして求めた $Freq(d, m)$ と $Edit(d, m)$ から関心度 $DOI(d, m)$ を算出し，その値を M_q の Doi に格納する．同時に，$Freq(d, m)$ の値を M_q の $Times$，Seq の値を M_q の $Latest$ に上書きする．さらに，$I(name, fqn, op, seq)$ の編集操作 op を M_q の $History$ に追加する．これにより，メソッド呼出し情報の更新（あるいは作成）が完了する．

図 5 に示すソースコードを開発者が上部から順番に記述し，コード補完を起動した際にデータベースに格納されているメソッド呼出し情報を XML 形式で出力した結果を図 4 に示す．`<proposal>`タグに囲まれている内容がひとつのデータ項目に対応する．たと

†2 メソッドがオーバライド関係を有する場合，実際に呼び出されるメソッドを静的に一意に決定することはできない．ただし，インスタンス参照の宣言型に基づき，$Prop$ にはオーバライドメソッドのうちのひとつだけが提示される．よって，m は一意に決定できる．

```
<?xml version="1.0" encoding="GB2312" standalone="no"?>
<applyoperation>
<eventcounter>97</eventcounter>
<proposal>
<string>chars() : IntStream - CharSequence</string>
<value>101</value>
<eventnumber>54</eventnumber>
<count>2</count>
<history author="liaoziyang" eventnumber="47" file="/Test/src/Test.java" offset="137" time="1453367201384"/>
<history author="liaoziyang" eventnumber="54" file="/Test/src/Test.java" offset="152" time="1453367230597"/>
</proposal>
<proposal>
<string>codePointAt(int index) : int - String</string>
<value>129</value>
<eventnumber>84</eventnumber>
<count>2</count>
<history author="liaoziyang" eventnumber="61" file="/Test/src/Test.java" offset="167" time="1453367240078"/>
<history author="liaoziyang" eventnumber="84" file="/Test/src/Test.java" offset="232" time="1453367503297"/>
</proposal>
<proposal>
<string>codePoints() : IntStream - CharSequence</string>
<value>80</value>
<eventnumber>68</eventnumber>
<count>1</count>
<history author="liaoziyang" eventnumber="68" file="/Test/src/Test.java" offset="193" time="1453367249805"/>
</proposal>
<proposal>
<string>concat(String str) : String - String</string>
<value>84</value>
<eventnumber>75</eventnumber>
<count>1</count>
<history author="liaoziyang" eventnumber="75" file="/Test/src/Test.java" offset="213" time="1453367480185"/>
</proposal>
</applyoperation>
```

図 4　メソッド呼出し情報を XML 形式で出力した結果

```
public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str = "Hello World";
        str.chars();
        str.chars();
        str.codePointAt(index);
        str.codePoints();
        str.concat(str);
        str.codePointAt(index);
        str.
    }
```

図 5　開発者が記述したソースコードの例

えば，CharSequence クラスの chars() メソッドは，現時点までに 2 回使用されており，それが最後に使用された時点を表すシーケンス番号は 54 である．また，現時点での関心度の値は 101 となっている．

3.2.3　補完候補並べ替えモジュール

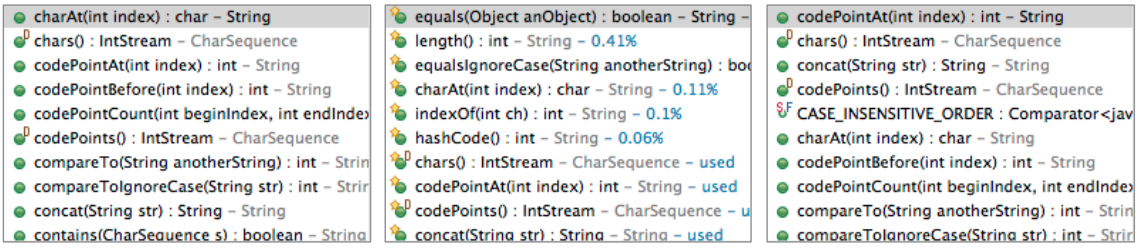
このモジュールでは，開発者 dev がコード補完を起動した（「Ctrl + Space」を押した）際，ECA の提供する補完候補情報 $Popr$ を取得し，補完候補として提示されるメソッドの一覧 $List$ を作成する．$List$ 内の各メソッド m に対して，メソッドの名前 $name$ とそのメソッドが所属するクラスの完全限定名 fqn を取り出し，検索キー $(name, fqn, dev)$ を作成する †3．

†3　ChangeTracker と同様に，dev は Eclipse の現在の使用者情報から取得する．

その後，メソッド呼出し情報を格納するデータベースにおいて，この検索キーに一致するデータ項目 $M_q[name, fqn, dev]$ を検索する．データ項目が見つかった場合，$List$ 内の m に M_q の関心度 Doi の値を割り付ける．見つからなかった場合，m には関心度の最小値 0 を割り付ける．

すべてのメソッドに関心度が割り付けられた $List$ に対して，関心度の大きい順に並び替えを適用する．並び替えた結果を $Prop$ に反映させることで，並び替え後の補完候補を開発者に提示する．ECA を拡張する際には，org.eclipse.jdt.ui.text.java パッケージに存在する IJavaCompletionProposalComputer インタフェースを実装し，それを拡張ポイント org.eclipse.jdt.ui.javaCompletionProposalComputer に登録すればよい．

開発者が図 5 に示すソースコードを記述中に，変数 str に対してコード補完を起動した際の動作例を図 6 に示す．関心度に基づく補完候補の提示結果 (c) を見ると，従来の (a) や (b) の提示結果と異なり，図 5 のソースコードにおいて，より直前により頻繁に使用されたメソッドがより上位に提示されていることが分かる．



(a) Content Assist　(b) Code Recommenders　(c) 関心度(DOI)

図 6　java.lang.String のインスタンス参照に対するコード補完の動作例

3.3　議論

図 6 に示すように，提案手法は開発者の編集履歴を反映したコード補完を実現している．また，本手法では，従来のデータマイニング技術に基づくコード補完手法のように，あらかじめ既存のソースコードを大量に収集し，利用パターンを学習させておく必要がない．このため，ECA が起動可能な状況であれば，開発者が独自に記述した，あるいは，記述中のソースコードに対しても，開発者の特性や状況に対応したコード補完が可能である．つまり，本システムの適用場面は広い．

さらに，本手法は，開発者が記述中のソースコードの編集操作をリアルタイムに分析して，メソッドの使用を補完候補の提示結果に反映させている．このため，開発者の特性や状況に対して即座に追従するコード補完が実現できているといえる．

一方，本手法では，個々の開発者が記述したソースコードのみに基づき関心度を算出している．このため，開発の初期段階などにおいて開発者が記述したソースコードが少ない場合には，コード補完の効果がそれほど期待できない．これは，本システムにおいて，一度も使用されていないメソッドの関心度を 0 としていることに起因する．

この問題に対する解決策として，ECR における統計データを，それぞれのメソッドの関心度の初期値に反映させることを考えている．また，特定のソースコード集合を用意し，ソースコードの編集時刻順に並べ，さらにソースコードを上部から記述したと仮定して，編集履歴を擬似的に作成することが考えられる．擬似的な編集履歴を用いることで，開発者が実際に開発を始める前に，それぞれのメソッドに対する関心度を擬似的に設定しておくことができる．

4　おわりに

本論文では，開発者がソースコードを作成した際の編集履歴に着目し，開発者の関心度に基づくコード補完手法とそれを実装したシステムを提案した．それぞれのメソッドに対する関心度は，(1) 開発者がそのメソッドを使用した回数と，(2) 開発者がそのメソッドを最後に使用してから現時点までに経過した編集数で算出する．関心度をリアルタイムに算出することで，開発者の特性や状況をより反映した補完候補の提示が可能である．

今後の課題として，実際の開発者を用いた被験者実験の実施があげられる．我々は，いくつかのサンプルコードを用いて，本システムの有用性を確認している．しかしながら，本システムの効果を明確に裏付ける実験データは存在しない．また，現在の関心度の算出式は著者らの直感と経験に基づいて定義されており，その妥当性に関する評価も行っていない．被験者実験を通して，本システムの有用性と関心度の算出式のチューニングを実施していく予定である．

さらに，本システムの実装には，一部不完全な部分が存在する．たとえば，ペーストにメソッド呼出しコードが挿入された場合には ECA が起動しないため，このコード記述に含まれるメソッドの特定に失敗する．また，メソッド特定の精度を向上させるためには，メソッドの引数の型の決定が必須である．今後，本システムの実装の改良も実施する予定である．

謝辞 本研究に関してご討論頂きました，立命館大

学の紙名哲生氏に感謝いたします．本研究の一部は，科研費（15H02685）の助成を受けたものである．

参 考 文 献

[1] Akbar, R. J., Omori, T., and Maruyama, K.: Mining API Usage Patterns by Applying Method Categorization to Improve Code Completion, *IEICE Transactions on Information and Systems*, Vol. 97, No. 5(2014), pp. 1069–1083.

[2] Bruch, M., Monperrus, M., and Mezini, M.: Learning from Examples to Improve Code Completion Systems, *Proc. Int'l Symp. Foundations of Software Engineering (FSE'09)*, 2009, pp. 213–222.

[3] ChangeTracker: https://github.com/katsuhisamaruyama/changetracker.

[4] Code Recommenders: http://www.eclipse.org/recommenders/.

[5] Duala-Ekoko, E. and Robillard, M. P.: Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study, *Proc. Int'l Conf. Software Engineering (ICSE'12)*, 2012, pp. 266–276.

[6] Eclipse: http://www.eclipse.org/.

[7] Hoffmann, R., Fogarty, J., and Weld, D. S.: Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers, *Proc. Symp. User Interface Software and Technology (UIST'07)*, 2007, pp. 13–22.

[8] Hou, D. and Pletcher, D. M.: An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion, *Proc. Int'l Conf. Software Maintenance (ICSM'11)*, 2011, pp. 233–242.

[9] Hsu, S.-K. and Lin, S.-J.: MACs: Mining API Code Snippets for Code Reuse, *Expert Systems with Applications*, Vol. 38(2010), pp. 7291–7301.

[10] Murphy, G. C., Kersten, M., and Findlater, L.: How Are Java Software Developers Using the Eclipse IDE?, *IEEE Software*, Vol. 23, No. 4(2006), pp. 76–83.

[11] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Tamrawi, A., Nguyen, H. V., Al-Kofahi, J., and Nguyen, T. N.: Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion, *Proc. Int'l Conf. Software Engineering (ICSE'12)*, 2012, pp. 69–79.

[12] 大森隆行, 丸山勝久: 開発者による編集操作に基づくソースコード変更抽出, 情報処理学会論文誌, Vol. 49, No. 7(2008), pp. 2349–2359.

[13] Robbes, R. and Lanza, M.: How Program History Can Improve Code Completion, *Proc. Int'l Conf. Automated Software Engineering (ASE'08)*, 2008, pp. 317–326.

[14] Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H.: MAPO: Mining and Recommending API Usage Patterns, *Proc. Euro. Conf. Object-Oriented Programming (ECOOP'09)*, 2009, pp. 318–343.