社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2009－SE－163 (37)
2009／3／19

# ソースコードコメントのテキスト分析

中村 大賀 †

† 日本アイ・ビー・エム 東京基礎研究所

ソフトウェアのソースコードのコメント部分には、品質に関連し得る記述がしばしば自然文で記述されている。本稿では、ソースコードからコメントブロックを抽出し、自然言語処理技術を適用してテキスト記述を分析・抽出可能にするためのアプローチを提案する。本アプローチでは、分類されたコードコメントに特徴的な表現を実データから抽出し、表現のパターンを定義することでコードコメントとの自動マッチングを可能にする。特に、オープンソースソフトウェアのソースコードを分析し、コードコメントのうち品質にネガティブに寄与し得るものを分類し、自然言語の構文解析結果と組み合わせた考察を行った。また、コードコメントの自動検出を行うツールのプロトタイプを作成した。

# Analyzing Natural-language Text in Source Code Comments

Taiga NAKAMURA †

† IBM Research, Tokyo Research Laboratory

Code comments in software often contains a narrative text description which can be related to the quality of software. We propose a novel approach to analyzing source code comments by extracting comment blocks and applying the natural language processing (NLP) technology. With this approach, we identify distinctive patterns of syntactic expressions based on the content of various source code comments, and extract comments that match those patterns. We use the data from several open source software projects, manually classify code comments, and evaluate the result of NLP processing. We also describe a prototype tool for automatically detecting matched comments.

## 1 はじめに

多くのプログラミング言語では、ソースコード中にコメント (注釈) を記述することが許されている。コンパイラやインタプリタはコードコメント部分を無視するため、コメントの有無や記述された内容はソフトウェアの動作を直接変えるものではない。しかしながら、ソフトウェアの品質という観点からはコードコメントの記述はしばしば重要視される。その理由として代表的なものは、保守性や移植性への寄与である。[9] では、コードを読むという立場でコードコメントの有用性について述べており、例えば、実装されているアルゴリズムが基にしている文献への参照の記述などを有用なコードコメントの例として挙げられている。また、機能性や信頼性などソフトウェアの動作そのものに関わる情報もコードコメントに含まれることもある。例えば、コードが動作するプラットフォームの制約や前提条件、特定の実装を選択した根拠などの記述がその例である。

このように公式なドキュメンテーションに近い位置づけのコードコメントは、適切に記述することでソフトウェアの品質向上に寄与することが期待できる。これに対して、開発者が自由に記述する内容のものもある。これらには、他のプロジェクト生成物には記述されないようなソフトウェアに関わる重要な問題が記述されている可能性がある。このような種類のコードコメントに対しては二つの立場が考えられる。第一の立場では、これらのコメントがソースコードに残存すること自体が品質に対して悪であるとする。この立場では、このようなコメントは確認の上削除することが望ましい。第二の立場では、これらのコードコメントの存在を品質上の手がかりとして利用する。すなわち、これらが多く残存する部分の周辺では機能的にも何らかの問題を抱えている傾向にあるという仮説に基づき、ソースコード中でのこれらのコメントの出現頻度や分布を品質に関するメトリクスの一つとして利用することになる。

いずれの立場でも、ソースコード中に存在する、公式なドキュメンテーションとは異なる種類のコードコメントを見つけることが必要である。本稿では、ソフトウェアの品質に関する情報を含んでいるコードコメントを抽出することを目標にコードコメントを分析する手法について述べる。ここで対象とするようなコードコメントは、人間が読むことを想定して自然文で書かれていることが多い。従来は、目視によるソースコードのレビューやインスペクションの一環として、チェック対象の一つとしてコメントも調べられていた。人手でのレビューは効果が高いが労力を要するため、ソフトウェアの規模が大きくなると網羅的なチェックが困難になる。

そこで本稿では、コードコメントに含まれるテキストに対して自然言語処理技術を適用することで、自動抽出を可能にすることを目指す。特に、現実のソースコードにおけるコードコメントにどのような記述が出現するかを調べ、それに基づいて注目すべき記述にどのようなパターンがあるかという知見を得る。

## 2 アプローチ

本節では、まず本稿におけるコードコメント分析の方針を定義するためにコメントを分類する。その上で、コードコメントを分析する手順を述べる。

### 2.1 ソースコードコメントの分類

表1に本稿で使用するソースコードコメントの分類を示す。前節の記述に従い、まず適切に記述することで品質上ポジティブに寄与し得るもの (ポジティブタイプコメント) と、存在が直接・間接的にネガティブに寄与し得るもの (ネガティブタイプコメント) に分けた。その上で、本稿で中心に扱う後者についてさらに3種類に分類した。

これらの分類は、後に示す実際のソフトウェアの分析において観察されたネガティブタイプコメントを比較的曖昧性が少なく分類できるように定義したものである。一般のソフトウェアにおけるあらゆるコメントがこの分類に当てはまる保証はない。また、ポジティブタイプコメントが適切に書かれていない場合にも品質上の問題が生じ得るが、本稿の分析では扱っていない。

### 2.2 分析手順

図1に、コードコメントの分析し結果を活用するための枠組みを示す。

まず分析フェーズでは、実際のソースコードから

Table 1 コードコメントの分類

| 品質へのポジティブ要素 | |
|---|---|
| ライセンス・著作権情報 | |
| 仕様・API定義 | |
| 変更履歴・変更者・変更日時 | |
| 処理の解説 | |
| 実装の背景・根拠・参照文献等の記述 | |
| **品質へのネガティブ要素** | |
| 課題点 | まだ不十分・未完成であること、暫定的な実装であること、既知の問題があり今後修正・再実装が必要であることについて述べたもの。 |
| 疑問点 | 実装や仕様に対する疑いや不満、言い訳について述べたもの。修正が必要な可能性があるが断定できていないもの。正しく動くかどうか判断できていないもの。 |
| 変更点 | 開発・変更の痕跡を残したもの。ドキュメンテーションとしての変更履歴のように管理されておらず、コードの改変箇所に直接マークをつけたもの。デバッグの過程でコメントアウトされたコード片 |

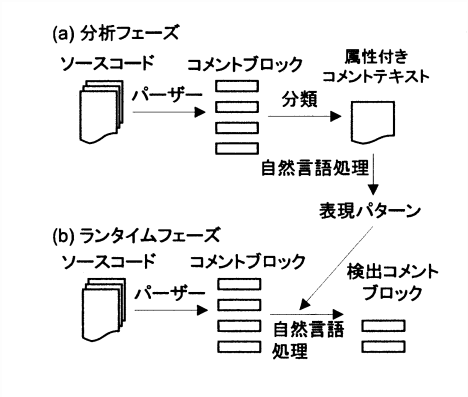


Fig. 1 コードコメント分析のアプローチ

各コメントブロックを取り出す。コメント部分を抜き出すことは各プログラミング言語のための基本的な構文解析器で可能である。本稿では、Java[1]およ

[1] Java およびすべての Java 関連の商標およびロゴは Sun Microsystems, Inc. の米国およびその他の国における商標。

び C/C++ プログラムを対象とし、Eclipse IDE[2] に含まれている構文解析器を呼び出すことでコメントの抽出を行った。なお、Java や C/C++ では /* から */ までの複数行コメントと// から行末までの一行コメントがあるが、後者の場合 Eclipse の構文解析器では各行が別のコメントブロックとして認識される。本稿では複数のコメントブロックが連続して存在する場合にはそれらを連結して一つのブロックとして扱うものとした。

本稿では、特に日本語によるコメント記述を分析の対象とする。ソースコードの日本語文字コードをデータごとに手動であらかじめ設定した上で、非英数字の文字が少なくとも一文字含まれるコメントブロック日本語コメントブロックとして抽出した。

次に、抽出された各コメントブロックの内容を目視で確認し分類する。この結果、属性付きテキストコメントが得られる。これを XML 形式で表示した例を図 2 に示す。なお図 2 では、コメントテキストと分類に加えて、ファイル名、コメントブロックの位置およびフォーマルコメントであるかどうか (Javadoc に限り true となる) も属性として抽出している。

```
<doc id="157">
<title>Foo.cpp</title>
<itemset>
<item name=".fileName" val="Foo.cpp"/>
<item name=".pathName" val="/"/>
<item name=".offset" val="9425"/>
<item name=".isFormal" val="false"/>
<item name=".commentType" val="1"/>
</itemset>
<textlist>
<text name="comment">エラーメッセージを出力できるようにすべき</text>
</textlist>
</doc>
```

Fig. 2　属性付きテキストコメント例

コードコメントテキストの自然言語処理および集計・分析には、IBM Content Analyzer (ICA) [6] を使用した。ICA では入力テキストの構文解析結果に基づき、係り受けのパターンを自動的に集計・統計処理することが可能である。

この出力をもとに、品質にネガティブに寄与し得るコードコメント中に特徴的に出現する表現を調べた。なお、本稿のアプローチでは、意味解析や各ソフトウェアで固有に使われる語彙等を使用したより高度な言語解析の情報は行わない。これにより、現在利用可能な技術の範囲で実用上十分な精度での解析が可能である一方、テキストとして記述した文章そのものだけが分析の対象となり得る点で制約を受ける。

ランタイムフェーズでは、分析フェーズと同様にコメントブロックを抽出した後、上で得られたパターンを自然言語処理結果に対して当てはめ、マッチするコードコメントを出力する。ユーザーは、検出されたコードコメント部分を個々に確認するか、検出結果を集計した情報をメトリクスとして利用する。

## 3　オープンソースソフトウェアの分析

上記の手順に基づいてコードコメントに日本語の記述を含むオープンソースプロジェクトのソースコードを分析した結果を示す。表 2 に使用した 9 プロジェクトの概要を示す。プログラミング言語は全て Java または C++ であり、ソフトウェアの規模は中規模以下である。

Table 2　分析に使用したプロジェクト

| # | プロジェクト | 言語 | ファイル数 |
|---|---|---|---|
| a | BeanBinder-0.2.3 | Java | 71 |
| b | kimera-2.11 | C++ | 45 |
| c | mgoban-r17 | Java | 248 |
| d | Morpho-2.0.10 | Java | 115 |
| e | PostgresForest-4.0.2 | Java | 482 |
| f | qpilot-r158 | C++ | 49 |
| g | S2Dao-1.0.49 | Java | 519 |
| h | S2Remoting-1.1.2 | Java | 18 |
| i | simBio-1.0 | Java | 507 |

これらのソフトウェアに含まれるコメントテキストそれぞれの内容を著者が読んで検討し、ネガティブタイプコメントに該当するものを判別した。見つかった候補は、さらにコードの該当部分を確認して記述の意味を確認した上で分類した。例えば、「ソートすべき?」というコメント内容に対しては、一見ソート処理が未実装であることに対する疑問点の記述であるようにも見えるが、該当コード部分はソートすべきかを判定する条件分岐であるためネガティブタイプコメントには分類しなかった。

なお、Eclipse などの開発環境が生成したスタブコードに自動的に含まれていたと推測できるコード

コメントはネガティブタイプコメントには分類しなかった。このようなコメントは特に PostgresForest (プロジェクト e) では数多く見られた。これらの自動生成コメントを除去していないことを品質上の問題とみなすことも可能である。

表 3 に分類結果を示す。全 7,478 個の日本語コメントブロックのうち、ネガティブタイプコメントに分類されたのは 331 個 (約 4.4%) であり、2 つのプロジェクトでは一つもネガティブタイプコメントが見つからなかった。ネガティブタイプコメントの種類ごとの出現数はプロジェクトによって大きく分布が異なり、特に変更点に関するネガティブタイプコメントはプロジェクト e 以外ほとんど見られなかった。プロジェクト e では、変更箇所をコメントでマークするプラクティスが使用されていることが推察される。

Table 3 手動による分類結果集計

| # | 日本語コメント数 | 課題点 | 問題点 | 変更点 |
|---|---|---|---|---|
| a | 66 | 1 | 0 | 0 |
| b | 71 | 1 | 0 | 1 |
| c | 382 | 13 | 22 | 0 |
| d | 1,064 | 0 | 0 | 0 |
| e | 3,757 | 31 | 19 | 135 |
| f | 218 | 50 | 3 | 0 |
| g | 373 | 3 | 1 | 2 |
| h | 43 | 0 | 0 | 0 |
| i | 1,504 | 35 | 13 | 1 |

データ全体のうちフォーマルコメント (Java プログラムの Javadoc) は 3,340 個であったが、そのうちネガティブタイプコメントに分類されたものは 30 個のみであった。つまりこのデータでは、定義上ドキュメンテーションとしての位置づけが強いフォーマルコメント以外の場所で品質にネガティブに寄与するような内容が書かれている傾向を示している。

以下、それぞれのタイプに分類されたコードコメントの具体例をいくつか示す。

課題点:

- 「未実装」 (プロジェクト b): 関数名を定義しているテーブルの中で、一つの関数が未実装であることを記述している。
- 「TODO スコアモードの undo に未対応」 (プロジェクト c) : undo ボタンが押されたときの処理を実装するメソッドで処理が未実装であることを記述している。
- 「共通で使うようなら、他のファイルに移動させる」 (プロジェクト f): リファクタリングの必要性を記述している。

疑問点:

- 「この文字が何を洗わしているのか不明。」 (プロジェクト c): 理解できていないことの記述 (漢字の表記ミスは原文のまま)。
- 「並列処理時にエラーが発生した場合には、close 時に切り離しを行う。 FIXME: なぜ？」 (プロジェクト e): 理解できていないことの記述。
- 「TODO setValue で更新出来た方がよいでしょうか？」 (プロジェクト i): 現在の実装に対して変更が必要な可能性を提示している。

変更点:

- 「Ver1.1 エラーリトライ 回数」 (プロジェクト e): 特定のバージョンでの改変部分にマーキング
- 「040128MOD パーティションの特定ができないのにパーティション番号を求めエラーとなるを修正 ...」 (プロジェクト e) 修正日時・理由を変更箇所に記述 (この後に続くコードコメントアウト部は省略した)

このような実例を観察した結果から、いくつかの定性的な考察が得られる。まず、「未実装」「未対応」のような語はネガティブタイプコメントに特徴的な語であり、検出すべきパターンと考えられる。すなわち、キーワードとして、これらの語を直接検出、または派生の係り受けパターンとして「実装... ない」「対応... ない」など動詞と否定語の組み合わせなどの派生パターンを定義することが可能である。しかし問題は、個々のパターンの出現頻度が高くないことである。分析対象のデータ中、「未実装」という語が出現するのは全体で 2 件だけである。したがって、数多くのネガティブタイプコメントを検出可能にするには、このような個別の例を多数準備する必要がある。

また、TODO や FIXME 等の良く知られた定型キーワードも、細かい分類の認識はできないものネガティブタイプコメントの認識には役立つことがわかる。ただし、これらの定型キーワードの代わりにプロジェクトや開発者ごとに独自のマークを使用している例も多い。例えばプロジェクト e では、デバッグのための改変箇所のコードコメントに @@ という記号を使用している開発者と、修正箇所に MOD というキーワードを記述している開発者がいることが観察される。これは定型キーワードだけを用いる場合の限界を示している。

次に、以上をふまえて、分類されたコードコメントのテキストを自然言語処理によって分析し、ネガティブタイプコメントに特徴的な表現を集計した。特徴的な表現かどうかを定量的に評価するには、単純にネガティブタイプコメントにおける出現数が多いというだけでなく、ネガティブタイプコメントでないものには現れにくいものであるかが重要であるこのような特性は、以下で定義される表現ごとの相対頻度[8]で評価することができる。

$$\text{相対頻度} = \frac{\text{対象のデータに出現する割合}}{\text{全体のデータに出現する割合}}$$

すなわち、ある表現について、それが出現するネガティブタイプコメントの割合を分子とし、それが出現する全コメントの割合を分母とする。相対頻度が 1 より大幅に大きいほど、相対的にネガティブタイプコメントに特徴的な表現であることになる。

表 4 から 6 に、ネガティブタイプコメント 3 種類それぞれに含まれる単語ごとの出現頻度および相対頻度を、相対頻度が上位のものについて示した。(表 6 では単語数が非常に多かったため値がより大きいもののみを示した。) これらの表から、TODO や FIXME などの定型キーワード以外にも実際にネガティブタイプコメントに偏って出現している表現の存在を確認できる。一方、プロジェクトに固有と思われる表現や、一部開発者だけが使っていると思われるくだけた表現も上位に出現してしまっている。これは、今回分析に使用したデータ数がプロジェクト特有の性質を自然に打ち消すことができるほど大きくないためと思われる。この表をもとに人手で適切な表現を選んでパターン化することは可能であるが、プロジェクト固有の表現を効率良く打ち消すことができればより望ましい。これは今後の課題である。

リストに出現している英単語の多くは、コメントアウトされているコード片に由来している。テキスト分析を行う際、コメントアウトされたコード部分を前もって判定し扱いを分けることも今後の課題である。

Table 4　コメント中の単語毎出現頻度: 課題点タイプ (相対頻度 2.0 以上)

| 単語 | 出現数 | 相対頻度 |
|---|---|---|
| イクナイ | 6 | 38.7 |
| 下記 | 6 | 38.7 |
| テーブルチェック | 6 | 38.7 |
| ハードだ | 6 | 38.7 |
| FIXME | 14 | 36.8 |
| アン | 5 | 33.1 |
| TableTreeViewer | 4 | 26.5 |
| viewer | 4 | 26.5 |
| A | 6 | 24.2 |
| ツリー | 6 | 21.6 |
| TODOMASU | 13 | 15.6 |
| 不正 | 3 | 13.3 |
| getTable | 3 | 13.3 |
| viewer. | 4 | 10.8 |
| setContentProvider | 2 | 9.4 |
| TableTree | 2 | 9.4 |
| FULL | 2 | 9.4 |
| H | 2 | 9.4 |
| MULTI | 2 | 9.4 |
| SELECTION | 2 | 9.4 |
| SCROLL | 2 | 9.4 |
| table | 4 | 7.4 |
| Table | 4 | 7.4 |
| new | 5 | 7.1 |
| V | 2 | 6 |
| BORDER | 2 | 6 |
| tree | 2 | 6 |
| SWT. | 2 | 4.5 |
| コード | 6 | 4.3 |
| 多重化 | 6 | 3.8 |
| 追加する | 7 | 3.6 |
| ビュー | 5 | 3.6 |
| parent | 2 | 3.5 |
| TODO | 10 | 3.4 |
| score | 3 | 3.3 |
| private | 2 | 2.9 |
| 引数 | 3 | 2.6 |
| コメント | 6 | 2.3 |
| status | 2 | 2.3 |
| その場しのぎ | 2 | 2.3 |
| にくい | 2 | 2.3 |
| type | 2 | 2.3 |
| list | 2 | 2.3 |

## 4　分析ツール

図 1 のランタイムフェーズに対応する部分として、コードコメントを分析し結果を出力するツールのプロトタイプを試作した。ツールは、Eclipse

Table 5　コメント中の単語毎出現頻度: 疑問点タイプ (相対頻度 2.0 以上)

| 単語 | 出現数 | 相対頻度 |
|---|---|---|
| CellEditor | 6 | 27 |
| 要 | 3 | 17.1 |
| 返る | 2 | 15.7 |
| 不要だ | 2 | 15.7 |
| なぜ | 2 | 15.3 |
| 無い | 7 | 12.3 |
| レコード | 2 | 10.1 |
| FIXME | 5 | 10.1 |
| よい | 2 | 9.9 |
| 呼ぶ | 7 | 9.3 |
| CellEditorListener | 2 | 9 |
| TODOMASU | 7 | 8.7 |
| swing. | 7 | 7.2 |
| javax. | 7 | 6.7 |
| TODO | 17 | 6.6 |
| Grid | 2 | 5.9 |
| long | 2 | 5.9 |
| close | 2 | 4.8 |
| 複数 | 3 | 3.9 |
| 確認 | 3 | 3.3 |
| 定義する | 2 | 2.8 |
| 違う | 3 | 2.7 |
| 適当だ | 3 | 2.7 |
| 並列処理 | 2 | 2.6 |
| null | 2 | 2.4 |
| event. | 2 | 2.4 |
| で | 2 | 2.2 |
| getLabelProvider | 1 | 2.1 |
| 本当に | 1 | 2.1 |
| やる | 1 | 2.1 |
| せる | 1 | 2.1 |
| 強制的だ | 1 | 2.1 |
| 蛙 | 1 | 2.1 |
| waitExecute | 1 | 2.1 |
| 紺 | 1 | 2.1 |

のプラグインとして実装されており、Eclipse 上のファイルエディタから呼び出すか、あるいはフォルダ構造のツリービューアー上で一つ以上のソースファイル・ディレクトリを選択した上で呼び出すことができる。後者の場合には、選択された複数ソースファイルに対して逐次的に分析が実行される。

分析対象のソースファイルからコメントブロックを抽出し構文解析を行った上で、ツールでは出力にマッチさせるパターンを指定できるようになっている。パターンの記述方法は IBM Content Analyzer (ICA) で定義されている形式を使用する。3 に表現パターンの例を示した。

Table 6　コメント中の単語毎出現頻度: 変更点タイプ (相対頻度 8.0 以上)

| 単語 | 出現数 | 相対頻度 |
|---|---|---|
| createNormalArgs | 3 | 21.2 |
| DEPARTMENT | 3 | 21.2 |
| finally | 3 | 21.2 |
| FROM | 8 | 21.2 |
| getContext | 3 | 21.2 |
| makeBaseSql | 3 | 21.2 |
| native | 3 | 21.2 |
| PagerContext. | 3 | 21.2 |
| popArgs | 3 | 21.2 |
| testMakeBaseSql | 3 | 21.2 |
| try | 8 | 21.2 |
| wrapper. | 3 | 21.2 |
| ネイティブ | 3 | 21.2 |
| 変化 | 3 | 21.2 |
| 変化ない | 3 | 21.2 |
| Cannot | 1 | 15.8 |
| component. | 1 | 15.8 |
| ERROR | 1 | 15.8 |
| find | 1 | 15.8 |
| getParent | 1 | 15.8 |
| getValueString | 1 | 15.8 |
| JOptionPane. | 1 | 15.8 |
| MESSAGE | 1 | 15.8 |
| parent. | 1 | 15.8 |
| showMessageDialog | 1 | 15.8 |
| targetName | 1 | 15.8 |
| 起点 | 1 | 15.8 |
| 統一性 | 1 | 15.8 |
| assertEquals | 3 | 15.3 |
| Exception | 3 | 15.3 |
| pushArgs | 3 | 15.3 |
| void | 3 | 15.3 |
| ない | 3 | 15.3 |
| if | 24 | 15.1 |
| public | 3 | 12 |
| sql | 3 | 12 |
| tokenizer. | 11 | 11.6 |
| Ver | 89 | 11.6 |
| 対応 | 28 | 11.1 |
| 修正 | 11 | 10.6 |
| m | 21 | 9.9 |
| SELECT | 3 | 9.9 |
| Error | 1 | 8 |
| message | 1 | 8 |
| 自身 | 1 | 8 |

パターンにマッチしたコメント部分はリストビューに検出場所とともに表示される。リスト項目をダブルクリックすることで、該当のソースファイルを開き出現部分にジャンプすることも可能である。

```
<mi category=".incomp" value="${1.str}
${0.str}">
 <w id="0" lex="実装">
   <w id="1" lex="未" />
 </w>
</mi>
<mi category=".incomp" value="${1.str}
${0.str}">
 <w id="0" lex="ない">
  <w id="1" pos="verb" lex="実装する" />
 </w>
</mi>
```

Fig. 3　表現パターンの記述例

## 5　関連研究

ソースコードコメントの系統的な研究としてもっとも多く研究されているのは、コメント部分に何らかの形式化を持ち込む手法である。コードコメントを公式なドキュメンテーションとして扱うにあたり、ドキュメンテーションとしての価値を高めるための記法を導入する手法には、Java の Javadoc[7] のように、言語の仕様自体に特別な種類のコメントを組み込むことでその中で formalism を実現するもの、Doxygen[3] のようにツールレベルで記法を定義することで同様のことを実現するものがある。

これをさらに推し進めた立場では、コメント部分の記述を既存のプログラミング言語文法の拡張として用いることで、コード静的解析の新しい技術を適用可能にするものがある。null dereference 解析など。より形式レベルの高い記法を仮定してコメント部分の情報を役立てる研究も多く見られる。本稿では、コードコメントを記述する際に特定の記法や制限言語の使用を積極的に仮定しないアプローチを取った。

メトリクスの一つとしてコメントの分量を使用する手法もよく知られている。代表的なものは全体の行数に対するコメント行数の割合、あるいはその派生形に基づくものである。[4] プログラミング言語ごとの適正なコメント率は異なり、その事実に対する調査が [1] などに見られる。

よいコメントの書き方についての書き方は慣習による部分も多く、必ずしも統一した見解が定着しているわけではないが実践的なガイドは利用可能である。[5] 本稿で品質へのポジティブ要素として挙げたタイプのコメントも、書き方が不適切であれば問題を生じる可能性がある。

## 6　おわりに

本稿では、自然言語での記述を含むソースコードコメントを分析するアプローチについて論じた。実際のソースコードに出現するコメントを集め、各コメントブロックを分類した上で、潜在的な品質上の問題につながるコメントブロックに出現する記述の傾向を分析し、自然言語処理の精度が期待できる範囲の、比較的単純な形式の表現パターンで、従来捉えられなかったような注目すべき自然文テキストのコメントがある程度検出可能であることを示した。さらに、指定した表現パターンを含むコードコメントを検出するプロトタイプツールを構築した。

本稿で述べた手法を改善するとともにより多くのデータを分析することで、従来簡単に調べることができなかったコードコメントの記述を効果的に分析することが可能になると期待できる。

本稿で述べたアプローチは、特定の表現が「書かれている」ことを問題にするものであるが、コードコメントをなるべく書かないようにするという圧力を与えるのは目的ではない。理想的には、このような手法を使って積極的にコードコメントを見直すことでコード全体の品質を向上させることが期待される。そのためには、コードコメントから得られる情報はなるべく建設的に使用すべきであると考える。特に品質問題をもとに開発者自身を咎めたり、彼らの生産性や能力の評価に使用したりすることは避けるべきである。

本稿におけるコードコメントの分類は、著者の主観で行われた。定量的な分析結果はこの分類に依存するため、複数の評価者によって分類を検証することが今後必要である。

また、本稿で分析に使用したデータの制約から、知見の適用範囲にも不確定要素を含んでいる。

- 本稿における分析データは全てオープンソースソフトウェアである。コードコメントの性質は開発体制や文化に依存する可能性があり、異なる環境で開発されるソフトウェアに対して表現のパターンがどの程度汎用に利用可能であるかは不明である。
- 本稿では、英語によるコードコメントは分析の対象外とした。日本語と英語の言語的な特徴の違いがどう影響するかは今後の課題であ

る。ソフトウェア開発の国際化に伴い、英語が母国語でない開発者がコードコメントを英語で記述することも増えているため、言語の習熟度も記述に影響を与える可能性がある。

- 本稿では、Java および C++のソースコードのみを使用しており、プログラミング言語による差異は分析していない。典型的なコメントの分量はプログラミング言語によって大きく異なることが知られており[1)]、コードコメントの書き方・性質の違いについても差異がある可能性がある。

## 参考文献


1) *How Open Source Comments (by Programming Language)* http://www.riehle.org/2008/11/10/how-open-source-comments-by-programming-language/.

2) *Eclipse.org*, http://www.eclipse.org/.

3) *Doxygen*, http://www.doxygen.org/.

4) Pete Goodliffe *Code Craft: The Practice of Writing Excellent Code*, No Starch Press, 2006.

5) *Google C++ Style Guide*, http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

6) *IBM Content Analyzer*, http://www-06.ibm.com/jp/software/data/products/ica/.

7) Douglas Kramer, *API Documentation from Source Code Comments: A Case Study of Javadoc*, Proceedings of the 17th annual international conference on Computer documentation, 1999.

8) 那須川 哲也, **テキストマイニングを使う技術/作る技術**, 東京電気大学出版局, 2006.

9) Diomidis Spinellis, *Code Reading*, Addison-Wesley, 2003.