ソースコードの品質向上を目的とした特定のコメントを検出するツールの試作

田上諭
宮崎大学 工学研究科
tanoue@earth.cs.miyazaki-u.ac.jp

甲斐秀一
株式会社スカイコム
h-kai@skycom.jp

片山徹郎
宮崎大学 工学教育研究部
kat@cs.miyazaki-u.ac.jp

要旨

TODO コメントと言ったマーキングの役割をするコメントや不適切なコメントを検出できるようになると，ソースコードの品質向上に役に立つ．本稿では，ソースコードの品質向上を目的として，特定のコメントを検出するツールを試作する．試作するツールでは，正規表現を利用し，検出文字列のパターンを *XML(Extended Markup Language)* ファイルに設定できる．これにより，検出したい特定のコメントを検出できる．新たに検出したい特定のコメントが出てきた場合も，検出文字列のパターンとして追加することにより，柔軟に対応できる．株式会社スカイコムの協力で *LOC(Line of Code)* が *5000* 行以上のプロジェクトを対象に試作したツールを適用し， 適合率及び再現率を調査した．調査した結果，検出文字列のパターン調整することにより再現率は *100* %を，適合率は *80* %を，それぞれ達成した．また，一度設定した *XML* ファイルは，他のプロジェクトでもそのまま適用でき，再利用可能なことが確認できた．試作したツールを用いることで，マーキングの役割をするコメントやソースコードに存在する不適切なコメントを検出できる．マーキングの役割をしているコメントや不適切なコメントを減らしていくことで，ソースコードに存在する課題の削減や理解容易性の向上が図れる．

1. はじめに

コメントは，ソフトウェアの動作に影響を与えずに，自然言語で自由に記述できる．そのため，コメントを用いるとソースコードでは表現できない内容も記述できる．例えば，TODO コメントは，ソースコードに存在する課題をコメントとして追加することができ，適切に記述することで有力な情報となる．例えば，Google C++ Style Guide[1] では，TODO コメントを記述することが推奨されている．このようなマーキングの役割をするコメントは，ソースコードに存在する問題点を見つけるのに有力な情報となるので，コメントとして残しておくことが推奨される．

しかし，TODO コメントは，その存在を忘れてしまう可能性があり，TODO コメントが存在したままの状態になってしまうことがある．TODO コメントが表している課題を修正した時点で，TODO コメントは削除されるべきである．すなわち，TODO コメントなどのマーキングの役割をするコメントは，ソースコードに課題が残っていることを調べるために，必要に応じて検出する必要がある．

また，マーキングの役割をするコメントとは別に，ソースコードに存在すると問題のあるコメントが存在する．例えば，処理そのものを説明しているコメントである．このようなコメントは，ソースコードを読めば理解できるので，存在する意味がない．単に読む量が増え，手間となる． 本研究では，このようなコメントを不適切なコメントと呼ぶ．不適切なコメントの詳細については，2 章で説明する．不適切なコメントは，目視による確認で

リスト 1. 改修履歴コメント

```
1 //平成18年6月12日　田上修正
2 /*2016/08/12　田上　修正*/
```

リスト 2. 意図が曖昧なコメント

```
1 //暫定的な処理
2 //とりあえず0で
```

発見可能であるが，人的資源の不足や開発の遅れから，十分な確認作業が行われない可能性があるので，ツールによる支援が必要となる．

そこで本研究では，ソースコードの品質向上を目的として，特定のコメントを検出するツールを試作する．試作するツールは，正規表現を利用し，検出文字列のパターンを XML(Extended Markup Language) ファイル [2] に設定できる．この XML ファイルの検出文字列のパターンにマッチするコメントを検出する． 一度設定した XML ファイルは，その他のプロジェクトでも再利用できる．新たに検出したい特定のコメントが出てきた場合は，検出文字列のパターンとして追加することにより，柔軟に対応できる．本ツールを利用することにより，特定のコメントを検出できる．マーキングの役割をしているコメントや不適切なコメントを減らしていくことで，ソースコードに存在する課題の削減や理解容易性の向上が図れる．

本稿の構成は，次のとおりである．2 章では，不適切なコメントを定義する．3 章では，試作したツールの外観とその機能を説明する．4 章では，ツールの実装方法について説明する．5 章では，ツールの適用例について説明する．6 章では，ツールの考察を行う．7 章では，本稿についてまとめる．

2. 不適切なコメント

不適切だと考えられるコメントの候補をあげ，コメントか不適切かどうかアンケートを行った．アンケートの対象は，株式会社スカイコムの開発者 16 人である [3]．アンケートにおいて 8 割以上が不適切だと答えたものを，不適切なコメントと定義する．以下に，不適切なコメントの分類とその理由を示す．

- 改修履歴コメント

  改修履歴コメントの例を，リスト 1 に示す．リスト 1 に示した改修履歴コメントは，改修履歴が書かれているコメントである．コメントがメンテナンスされずにソースコードの現状とは異なった内容になっている恐れがある．また，改修履歴コメントは改修がされていく度に，コメントが増えていき，ソースコードが読みにくくなる．以上の理由から， 改修履歴のコメントを残すのではなく，バージョン管理ツールを利用するべきである．

- 意図が曖昧なコメント

  意図が曖昧なコメントの例を，リスト 2 に示す．リスト 2 に示した意図が曖昧なコメントは，どのような意図で書かれているのかよくわからないコメントである．意図が曖昧なコメントが存在すると，読み手によって解釈が異なる場合があり，混乱を招いてしまう．

- 処理そのものを説明しているコメント

  処理そのものを説明しているコメントの例を，リスト 3 に示す．リスト 3 に示したコメントは，処理そのものを別の言葉で言い換えただけのコメントである．このようなコメントは，ソースコードの読み手に追加情報を一斉提供せずに，単に読むソースコードの量を増やすだけである．処理そのものを説明しているコメントは，今回試作するツールで検出するのは難しい．本稿では，このコメントは検出対象とはしない．その理由については，7 章の今後の課題で説明する．

リスト 3. 処理そのものを説明しているコメント

```
1 // ここで10回ループする
2 while (i < 10) {
3   i++; // iをインクリメントする
4 }
```

3. 試作ツール

試作したツールは，Java 言語で開発した．Java 仮想マシンがインストールできる環境であれば，任意の OS 上で動作する．

ツールの外観と機能,「コメントを解析」機能の詳細，XML ファイルについて，以降それぞれ説明する．

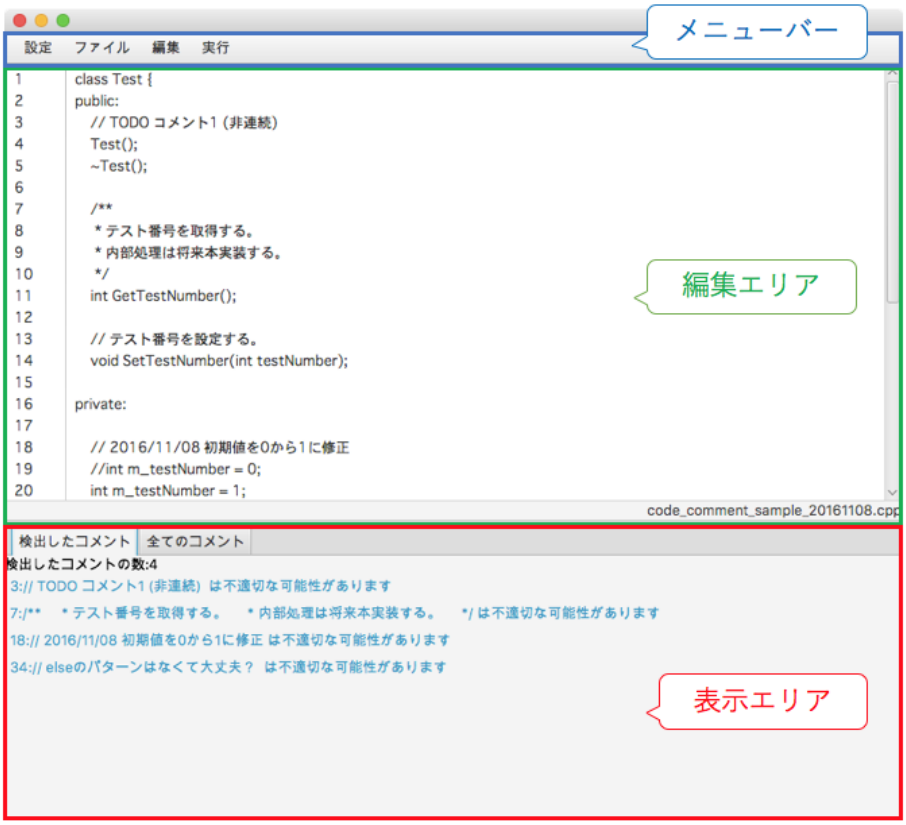

図 1. ツールの外観

3.1. ツールの外観と機能

本研究で試作したツールの外観を，図 1 に示す．試作したツールはメニューバー，編集エリア，表示エリアから構成している．ツールのユーザーは，メニューバーからメニューアイテムを選択することによって，ツールの機能を利用できる．

ユーザーがメニューバーの「設定」をクリックすると，設定メニューを表示する．設定メニューから，下記に示す 1 つの機能を利用できる．

- 文字コード
  ユーザーが「文字コード」にカーソルを合わせると更に 2 つのメニューを表示する.「UTF-8」と「Shift-JIS」である．ユーザーは，どちらかを選択することができ，起動時は「UTF-8」をデフォルトで選択している．ユーザーが文字コードを選択すると，試作したツールは選択した状態の文字コードで編集エリアのコードを開き直し，次回以降は選択した文字コードでファイルを開く．

ユーザーがメニューバーの「ファイル」をクリックすると，ファイルメニューを表示する．ファイルメニューから，下記に示す 3 つの機能を利用できる．

- ファイルを開く
  ユーザーが「ファイルを開く」をクリックすると，ファイルオープンダイアログを表示する．ユーザーが，ファイル―オープンダイアログからファイルを選択すると， ファイルの内容を編集エリアに表示する．

- ファイルを保存
  ユーザーが「ファイルを保存」をクリックすると，ファイルセーブダイアログを表示する．ユーザーがファイルの保存先及びファイル名を入力し，ファイルセーブダイアログの「Save」ボタンをクリックすると，編集エリアの内容を読み込みファイルに保存する．

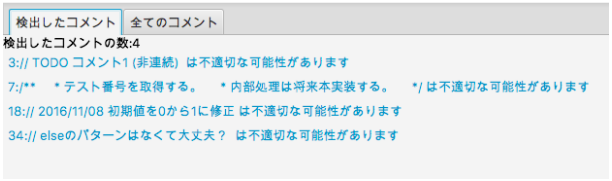

図 2. パターンにマッチしたコメントを表示するタブ

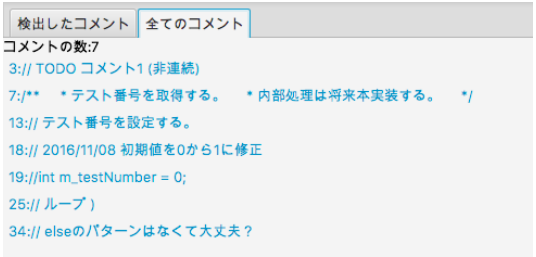

図 3. 全てのコメントを表示するタブ

- 複数ファイルのコメントを解析
  ユーザーが「複数ファイルのコメントを解析」をクリックすると，ファイルオープンダイアログを表示する．ユーザは，ファイルオープンダイアログから複数のファイルを選択できる．ユーザがファイルを選択すると，それぞれのファイルの内容を読み込み，それぞれコメントを解析する．その後，それぞれの解析結果を表示エリアに表示する．

ユーザーがメニューバーの「編集」をクリックすると，編集メニューを表示する．編集メニューから，下記に示す 2 つの機能を利用できる．

- 編集エリアをクリア
  ユーザーが「編集エリアをクリア」をクリックすると編集エリアの内容をクリアする．

- 表示エリアをクリア
  ユーザーが「表示エリアをクリア」をクリックすると表示エリアの内容をクリアする．

ユーザーがメニューバーの「実行」をクリックすると実行メニューを表示する．実行メニューから，下記に示す 1 つの機能を利用できる．

- コメントを解析
  ユーザーが「コメントを解析」をクリックすると編集エリアの内容を読み込み，コメントの解析を行う．試作したツールは，コメントを解析した結果を表示エリアに表示する．詳細については，次節で述べる．

3.2. コメントを解析

コメントは，行コメント (//) とブロックコメント (/* */) に対応している．行コメント及びブロックコメントの表記が (//)，(/* */) であるプログラミング言語であれば，どんな言語でもコメントを解析できる．

「コメントを解析」は，まずソースコードに存在する全てのコメントを抽出する．次に，XML ファイルに記述されている検出文字列のパターンに基づきマッチングする．検出文字列のパターンにコメントの記述内容がマッチした場合，マッチしたコメントとして検出する．検出したコメントが検出対象のコメントかどうかは，最終的には人により判断する必要がある．

コメントの解析結果は，表示エリアで確認できる．表示エリアは，検出文字列のパターンにマッチしたコメントを表示するタブと，全てのコメントを表示するタブの 2 つに分けている．「検出したコメント」タブを，図 2 に示す．このタブでは，パターンにマッチしたコメントを確認できる．解析結果をクリックすると，編集エリアにおいて，そのコメントのある行に移動する．「全てのコメント」タブを，図 3 に示す．このタブでは，ソースコードにある全てのコメントを確認できる．解析結果をクリックすると，編集エリアにおいて，そのコメントのある行に移動する．

3.3. XML ファイル

XML ファイルの例を，リスト 4 に示す．XML ファイルでは，Java 言語が利用できる正規表現 [4] で検出文字列のパターンを記述することができる．

リスト 4 に記述した検出文字列のパターンで，検出可能なコメントは「//TODO メソッドを分かりやすくする」，「//将来は，データをファイルから読み込めるようにする.」といった，「TODO」という単語が含まれているコメント及び「将来」という単語が含まれているコメントである．また，「//2017/3/03 田上 修正」といった「(?=.*\d{2,4}[.\-/年] \d{1,2}[.\-/月]\d{1,2}日

リスト 4. XML ファイルの例

```
<?xml version="1.0" encoding="UTF-8" ?>
<comments>
    <comment type="TODOコメント">
        <regularExpression>TODO</regularExpression>
        <regularExpression>将来</regularExpression>
    </comment>
    <comment type="改修履歴コメント">
         <!-- 検出する日付フォーマットは以下
             yyyy/mm/dd、yyyy.mm.dd、yyyy-mm-dd、yyyy年mm月dd日 (yyyyは2桁以上4桁以下)-->
        <regularExpression>(?=.*\d{2,4}[.\-/年]\d{1,2}[.\-/月]\d{1,2}日{0,1})</regularExpression>
    </comment>
</comments>
```

{0,1})」の正規表現がマッチするコメントである．

「//Todo:インデントを綺麗にする．」というコメントを検出したい場合は，TODO という検索ワードを大文字と小文字を区別付せずに判断するように XML ファイルを変更する必要がある．正規表現により，リスト 4 の 4 行目を，「<regularExpression>(?i)TODO</regularExpression>」に変更する．なお，「?」などの正規表現に使われる特殊文字を検出文字列のパターンとしたい場合は，「?」の前に「\」を追加し「<regularExpression>\?</regularExpression>」とする必要がある．

4. 実装

試作するツールの「コメントを解析」機能は，コメントの抽出及びパターンマッチにより構成する．以降，それぞれについて述べる．

4.1. コメントの抽出

コメントの抽出には，JavaCC(Java Compiler Compiler) を用いる．JavaCC は，Java 言語が利用できるパーサジェネレータとしてよく使われる [5]. EBNF(Extended Backus Naur Form)[6] に似た文法で文法ファイルを記述しコンパイルすると，Java 言語で記述されたパーサを自動生成する．

コメントの抽出に利用している文法ファイルを，リスト 5 に示す．リスト 5 の 23 行目の定義で，行コメント (//) に一致する字句か判断でき，24 行目の定義で，ブロックコメント (/* */) に一致する字句か判断する [7]．それ以外のものは，コメントの抽出には必要ないので，25 行目で<OTHERS>の字句と判断し，34 行目から 41 行目で構文解析を行う．コメントを見つけるだけならば，「((<OTHERS>*)(<LINE_COMMENT>|<BLOCK_COMMENT>)(<OTHERS>*))*」の構文規則の記述ですむ．今回試作したツールでは，コメントを取得して，そのコメントと検出文字列のパターンが一致しているかを調べたい．そこで，構文に一致した際にそのコメントを取得するコードを追加している．

この文法ファイルを JavaCC でコンパイルすると，CommentParser.java などのファイルが生成される．CommentParser.java には，CommentParser というクラスが記述されている．生成されたパーサの使用法を，リスト 6 に示す．リスト 6 の 2 行目で，CommentParser のインスタンスを作成している．コンストラクタに，引数として StringReader 型の文字列を渡す．このパーサに渡された文字列が，構文解析される．構文解析は 5 行目の paser.comment() が実行されるタイミングで実行される．コメントの解析結果は，String を格納する ArrayList インスタンスである comments に保持される．この方法で全ての行コメントと，ブロックコメントを抽出できる．

4.2. パターンマッチ

コメントの解析は，XML ファイルを読み込み解析する．XML ファイルを，リスト 4 に示した．

このファイルから，<comments>の子要素の子要素である<regularExpression>の内容を取得する．取得には，DocumentBuilderFactory クラス [8] を利用する．取得したコメントと取得した内容それぞれのパターンがマッチしているか判断し，パターンにマッチしたものを，「検出したコメント」タブに表示する．

リスト 5. コメントの抽出に使っている文法ファイル CommentParser.jj

```
options {
  STATIC=false;
  UNICODE_INPUT =true;
}

PARSER_BEGIN(CommentParser)
  package CommentParser;

  import java.util.ArrayList;
  import Dictionaly.*;
  import ResultData.*;

public class CommentParser{
}
PARSER_END(CommentParser)

SKIP :{
    <  SPACE:([" ","\t","\r","\f"])+ >
}

TOKEN :
{
      <LINE_COMMENT:"//"(~["\n","\r"])* ("\n"|"\r\n" | "\r")?>
      |<BLOCK_COMMENT: "/*" (~["*"])*("*")+ (~["/","*"](~["*"])*("*")+)* "/">
      |<OTHERS:~[]>
}
ArrayList<String> comment():
{
    Token lineComment,blockComment;
    ArrayList<String> result = new ArrayList<String>();

}
{
    ( (<OTHERS>*)
          (lineComment = <LINE_COMMENT>
          { result.add(lineComment.image); }
          | blockComment =<BLOCK_COMMENT>
          { result.add(blockComment.image); }
          )
         (<OTHERS>*))*
      { return result; }
}
```

5. 適用例

株式会社スカイコムの協力で LOC(Line of Code) が 5000 行以上のプロジェクトを対象に試作したツールを適用し，適合率及び再現率について調べた [3]． 本稿での適合率は，検出結果の中にどれだけ検出すべきコメントが含まれたかを表し，計算式は以下となる．

$$\text{適合率} = \frac{\text{検出できた検出すべきコメント数}}{\text{検出した全コメント数}} \quad (1)$$

また，再現率は検出すべき検出対象のコメントのうち，どれだけ検出できたかを表し，計算式は以下となる．

$$\text{再現率} = \frac{\text{検出できた検出すべきコメント数}}{\text{検出すべきコメント数}} \quad (2)$$

今回の検出対象となるコメントは，TODO コメントと不適切なコメントである．ソースコード分析の結果，理解不足であることを示すコメントが見つかった．理解不足であることを示すコメントの例を，リスト 7 に示す．このような理解不足であることを示すコメントも，不適切なコメントかどうかのアンケート調査は行っていないが，今回の検出対象とした．

計測の方法は，検出すべきコメント及びコメントの総数を目視で確認する．その結果とツールを用いて解析した結果により，適合率及び再現率を計算する．

まず，設定ファイルの調整のためにあるプロジェクトに適用する．このプロジェクトの詳細を， 表 1 に示す．また， 適用した結果を， 表 2 に示す．このプロジェクトの 1 回目は，XML ファイルに記述した検出文字列のパターンに原因があり，再現率，適合率がともに満足な数字ではなかった．その後，検出文字列のパターン調整を行い，3 回目には，再現率が 100 %となり，適合率が 85.2 %と 8 割以上の適合率になった．

リスト 6. 生成されたパーサの使用法

```
public ArrayList<String> comments = new ArrayList<>();
CommentParser paser = new CommentParser(new StringReader(new String("//hello"));
try{
    comments = parser.comment();
}catch(CommentParser.ParseExcepiton e){
    e.printStackTrace();
}
```

リスト 7. 理解不足であることを示すコメント

```
//なぜか動く
//挿入する必要があるかもしれない
```

表 1. 初めに適用するプロジェクトの詳細

LOC	コメント総数	検出すべきコメント数
6854	1990	23

他のプロジェクトでも，設定ファイルが利用できることを確認するために，このプロジェクトで利用した設定ファイルを別のプロジェクトにも適用した．調整した設定ファイルを適用するプロジェクトの詳細を，表 3 に示す．また，別のプロジェクトに適用した結果を，表 4 に示す．実行回数 1 回目で，再現率が 100 %で適合率が 92.3 %と高い結果を得ることができた．良い結果を得られるように調整した設定ファイルが，他のプロジェクトにもそのまま利用できることが確認できた．

6. 考察

不適切なコメントを検出するツールは，他にも開発されている．中村氏 [9] は，Eclipse プラグインとして不適切なコメントを検出するツールを開発している．このツールでは，本論文と同様に XML ファイルを利用し検出文字列のパターンを定義することができる．しかし，パターンマッチによる正規表現は使えない．試作したツールでは，正規表現を利用し検出文字列のパターンの記述ができるので，柔軟に定義できる．

阿萬氏 [10] は，コメントが多くなっているソースコードには，コメントがなければ理解できないようなソースコードが多く存在し，そのようなソースコードがフォールトの温床になり，フォールトが増えているのではとの懸念から, コメントとフォールト潜在との関係に関する定量分析を行った．結果として，コメントの多いソースコードは，フォールト潜在率が高いという傾向が示された．本稿で試作したツールでは，特定のコメントを検出できる．不適切なコメントを検出することによって，フォールトの温床となっているソースコードを見つけることができる．見つけたソースコードを修正することにより，フォールトを減らしていくことができる．

開発者がソフトウェアプロジェクトを短期間でリリースするために，最適でない解決手段を選択することを表す造語として，技術的負債がある [11]．技術的負債には，不注意によって混入するものと，意図的に導入するものがある [12]．意図的に導入された技術的負債は，Self-Admitted Techical Debt と呼ばれる．これは，ソースコードコメントから見つけることができる．従来の研究では，Self-Admitted Techical Debt の検出は，目視に大きく依存していた [13]．そこで，Maldonado 氏は自然言語処理技術を利用し，自動的に Self-Admitted Tchical Debt を検出する方法を提案した [14]．しかし，[14] では再現率を 100 %にできていない．試作したツールでは，文字列検出パターンを用意することで再現率を 100 %にできる．

grep コマンド [15] やテキストエディタのテキスト検索機能を用いても，試作したツールと同様のことができる．しかし，これらは，ソースコードすべてを検索してしまうために，print 文にある自然言語及びメソッド名や変数名として利用されているものまで検出してしまい，適合率が低くなる．本ツールでは 一度全てのコメントを抽出し，抽出したコメントに対して検出文字列のパターンとマッチしているかどうかを判断する．このことにより grep より適合率を高くできる．

本稿の正規表現によるパターンマッチでは，適合率を 100 %にすることは難しい．今回，適用したプロジェクトにはパーサのコードがあり，それを説明するための EBNF(Extended Backus Naur Form) がコメントに記述されていた．このコメントに，なくてもよいという意

表 2. 初めに適用したプロジェクトの適用結果

実行回数	検出すべきコメント数	検出した全コメント数	検出できた検出すべきコメント数	適合率	再現率
1	23	31	20	64.5	87.5
2	23	29	23	79.3	100
3	23	27	23	85.2	100

表 3. 調整した設定ファイルを適用するプロジェクトの詳細

LOC	コメント総数	検出すべきコメント数
9394	1584	24

味を持つ正規表現の「?」があり，曖昧なコメントを検出するための検出文字列のパターンである「?」にマッチしていため，不適切なコメントして検出していた．適合率を 100 %にできるような検出方法の考案が，今後の課題である．

7. おわりに

本稿では，ソースコードの品質向上を目的として，特定のコメントを検出するツールを試作した．試作したツールを用いることで，ソースコードに存在する不適切なコメントやマーキングの役割をするコメントを検出できる．なお， 検出したコメントには，検出対象以外のコメントも含まれてしまうことがあるので，目視による確認が必要となる．また，新たな検出対象のコメントが出てきた場合も，検出文字列のパターンとして追加できる．試作するツールは，正規表現を利用しているので様々なコメントに対して柔軟に対応できる．

株式会社スカイコムの協力で LOC が 5000 行以上のプロジェクトを対象に試作したツールを適用し， 適合率及び再現率について調査した．調査した結果，検出文字列のパターン調整することによって，再現率は 100 %を，適合率は 80 %以上を，それぞれ達成した．また，一度設定した XML ファイルは，再利用可能なことが確認できた．本ツールを利用することにより，特定のコメントを検出できる．マーキングの役割をしているコメントや不適切なコメントを減らしていくことで，ソースコードに存在する課題の削減や理解容易性の向上が図れる．

以下に，今後の課題を示す．

- 適合率の向上
  本稿の方法でも，正規表現の記述方法の工夫や検出対象のコメントによっては，適合率を 100 %にできる．しかし，正規表現によるパターンマッチのみでは限界がある．例えば，学習させたデータに基づいて，適切なコメントか判断する検出手法の提案などが必要となる．

- 処理そのものを説明しているコメントのみの検出
  処理そのものを説明しているコメントに使われる単語は，他のコメントにも頻繁に現れる．そのため，処理そのものを説明しているコメントをパターンマッチで検出しようとすると，多くの適切であるコメントを拾ってしまう．コメントの周辺の情報を取得し，前後の繋がりをもとに，コメントの内容が処理の説明かどうか判断する検出手法の提案などが必要となる．

- メンテナンスされていないコメントの検出
  適切に書かれたコメントでも，メンテナンスされていない場合は，不適切なコメントになりううる．バージョン管理ツールである git などの情報を利用し，コメントの周辺が変更されているにも関わらず，コメントが修正されていないところを検出するなどの方法で，メンテナンスされていないコメントを検出できると考えている．

参考文献

[1] StyleGuide: Google C++ Style Guide. `https://google.github.io/styleguide/cppguide.html`

[2] XML:“Extensible Markup Language (XML) 1.0 (Fifth Edition)”, W3C Recommendation, 2008

[3] 甲斐秀一, 田上諭, 片山徹郎, “ソースコードに存在する不適切なコメントを検出手法の適用事例”, ソフトウェア・シンポジウム *SS2017* 発表予定, 2017.

表 4. 調整した設定ファイルを適用したプロジェクトの適用結果

実行回数	検出すべき検出対象のコメント数	検出した全コメント数	検出できた検出すべきコメント数	適合率	再現率
1	24	26	24	92.3	100

[4] クラス Pattern: Pattern (Java Platform SE8). https://docs.oracle.com/javase/jp/8/docs/api/java/util/regex/Pattern.html.

[5] JavaCC: JavaCC - The Java Parser Generator. https://javacc.org/.

[6] ISO JTC1/SC22: “ISO 14977:1966”, International Organization for Standardization, 1996.

[7] 青木峰郎: “普通のコンパイラをつくろう”, ソフトバンククリエィティブ株式会社, 2009.

[8] クラス DocumentBuilderFactory: DocumentBuilderFactory (Java Platform SE8). https://docs.oracle.com/javase/jp/8/docs/api/javax/xml/parsers/DocumentBuilderFactory.html.

[9] 中村大賀, “ソースコードコメントのテキスト分析”, 社団法人 情報処理学会 研究報告 *SE-163*, pp.287–294, 2009.

[10] 阿萬祐久, “オープンソース・ソフトウェアにおけるコメント記述及びコメントアウトとフォールト潜在との関係に関する定量分析”, 情報処理学会論文誌 *Vol.53 No.2*, pp.612-621, 2012.

[11] W.Counnigham, “The wycash portfolio management system”, *in Addedum to the Proceedings on Object-oriented Programing System, Languages, and Application*, pp.29-30, 1992.

[12] M.Fowler: Techical debt quadrant. https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[13] A.Potdar, and E. Shihab, “An exploratory study on self-admitted techical debt”, *IEEE International Conference on Software Maintenance and Evolution*, pp.91-100, 2014.

[14] E. d. S. Maldonado, E. Shihab, and Nikolaos Tsantali, “Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt”, *IEEE Transactions on Software Engineering*, DOI:10.1109/TSE.2017.2654244, 2017.

[15] Grep: GNU Grep 3.0. https://www.gnu.org/software/grep/manual/grep.html