社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# ソースコードコメントに着目した 技術負債に対する修正の類似性の調査

岡島　早紀† 　神田　哲也† 　井上　克郎†

† 大阪大学大学院情報科学研究科
〒565-0871 大阪府吹田市山田丘 1-5
E-mail: †{s-okajim,t-kanda,inoue}@ist.osaka-u.ac.jp

**あらまし**　ソフトウェアの開発工程において，開発者はさまざまな理由で最適ではないコードを記述することがある．こういった一時的な解決策が原因となり長期的にみたソースコードの品質が低下することを技術負債という．技術負債の中でも特に開発者が意図的に導入し，自然言語を使用してソースコードコメントにその存在が記述されているものを自認された技術負債 (SATD) という．SATD はそれ自身が欠陥の原因になり得るだけでなく将来的にソフトウェアの保守性の低下を引き起こすため，リファクタリングによって除去されることが望ましい．しかし SATD の多くは除去の優先度が低く，実際に除去される SATD は一部のみであり，SATD は長い間放置される傾向にあることがわかっている．そこで本研究では，SATD に対する修正支援を目指し，SATD コメントが示す内容と SATD へと加えられる修正の類似性を調査した．調査の結果，SATD に加えられる修正の類似度は SATD コメントの量とコメントルールに大きく影響されることが確認された．また調査結果からデータベースを構築し，修正支援ツールの試作を行った．
**キーワード**　SATD，ソースコード，修正支援

# A Study on Similarity of Modifications to Technical Debt Focusing on Source Code Comments

Saki OKAJIMA†, Tetsuya KANDA†, and Katsuro INOUE†

† Software Engineering Laboratory, Department of Computer Science Graduate School of Information Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka, 565-0871
E-mail: †{s-okajim,t-kanda,inoue}@ist.osaka-u.ac.jp

## 1. ま え が き

ソフトウェアの開発工程において，開発者はさまざまな理由で最適ではないコードを記述することがある．こういった一時的な修正が原因となり長期的に見たソースコードの品質が低下することを技術負債という [1]．技術負債は発見や対処が容易でないことからソフトウェアの保守や全体の品質に悪影響を及ぼすため開発プロセスでその存在を特定し管理する必要があることが分かっている [2].

技術負債の中でも特に開発者自身が技術負債の導入時にその存在に気付いていて，かつその存在を自然言語を使用してソースコードコメントに記述しているものを SATD(Self-Admitted Technical Debt) という [3]．Wehabi らの調査 [4] では，SATD を含むファイルは SATD を含まないファイルと比較して変更されるコード行数やファイル構造数が多く，SATD を含むファイルに対する変更はより複雑であるという結果を得た．これは SATD はそれ自身が欠陥の原因になり得るだけでなく将来的にソフトウェアの保守性の低下を引き起こすことを示しており，SATD はリファクタリングによって除去されることが望ましいが，Potdar らの研究によると実際に除去される SATD は 26.3% から 63.5% のみである [3]．また Gabriele らの研究では，SATD が導入されてから除去されるまでに平均して 1,000 以上のコミットがされることから，SATD は長い間放置される傾向にあることがわかっている [5]．SATD が除去されず長い間放置される理由として，他の技術負債と比較した優先度の低さが考えられる．SATD に最も多く見られるのは設計に関するものである [6]．設計に関する SATD とは複雑なメソッドや一時的な回避策を指し，これらはソフトウェアの動作には直接関

係しないため他の SATD と比較して除去の優先度が低くなっていることが予想できる．

前述の通り，SATD はソフトウェアの保守性の低下を引き起こすため取り除かれるのが望ましいが，プロジェクトによっては導入された SATD のうち過半数は除去されずに長い期間放置されているのが現実である．そこで SATD への修正支援ツールとして，ソースコードのファイル構造を街のように可視化し，除去するべき SATD を目立たせて表示することでユーザに除去を促す環境が提案されている [7]．しかし実際に SATD を解消するには，具体的にどのような修正が必要かの情報が求められる．さらに，SATD の迅速な解消を目指すには，SATD コメントがソースコードに追加された時点で修正案を提示する必要がある．

そこで本研究では，修正支援を目的に SATD コメントが示す内容と SATD へと加えられる修正の類似性を調査し，修正支援ツールの試作を行った．具体的には，調査対象となるプロジェクトの過去のすべての編集内容をトレースし，SATD コメントの削除に伴ってなされたコードの編集内容とコメント内容を記憶した．つぎに SATD コメントをベクトル化したのち，クラスタリング手法のひとつである K-means 法を用いてクラスタリングを行った．最後に SATD コメントの内容をもとに分けられたそれぞれのクラスタにおいてコードの編集内容の類似性の調査を行った．類似性の調査にはコードの編集内容をトークン化したものの組に対して局所アラインメントのスコアを算出し，スコアが閾値 $\alpha$ を超えるものを修正が類似すると定義して，各プロジェクトの SATD に対する修正の類似性を調査した．

## 2. 背　　景

本節では今回の研究の基となる関連研究について説明する．2.1 節では技術負債，2.2 節では自認された技術負債，2.3 節では SATD の検出手法について説明する．

### 2.1 技 術 負 債

ソフトウェアの開発工程において，予算の減少やスケジュールの短縮，顧客の要求等に応えるために開発者は一時的な解決を図ることがある．こういった最適ではない解決策を負債とみなし，負債により長期的に見たソースコードの品質が低下することを技術負債という [1]．技術負債には不十分なテスト，コンパイラの警告などさまざまな種類のものが存在するが，ここでは特にソースコード中にみられる負債について述べる．既存研究では，技術負債はソフトウェアの品質や保守において悪影響を及ぼす可能性があるため，開発プロセスでその存在を特定し管理する必要があることが分かっている [2]．技術負債を特定する手法としては，ソースコードメトリクスやコードスメル等を用いたソースコード分析 [8] [9] が使用される．

### 2.2 自認された技術負債

自認された技術負債 ( Self-Admitted Technical Debt : SATD ) とは，Potdar らによって提案された概念であり，技術負債の中でも特に開発者が意図的に導入し，ソースコードコメントを使用してその存在が文書化されているものを指す [3]．SATD を表すコメントを SATD コメントとする．実際にソースコード中に現れた SATD コメントの例を表 1 に示す．

表 1　SATD コメントの例

| プロジェクト名 | SATD コメントの例 |
|---|---|
| CAMEL | //TODO is this needed |
| GERRIT | //Not exact but we cannot do any better |
| HADOOP | //TODO fix this |
| LOG4J | //This feels like a hack and it does not work |
| TOMCAT | //FIXME Add flags if possible |

Wehabi らの研究 [4] では SATD を含むファイルと SATD を含まないファイルにおける変更の複雑さを比較することで，SATD がプロジェクトの品質に与える影響を調査している．5 つのオープンソースプロジェクトに対して調査を行い，SATD を含まないファイルと比較して SATD を含むファイルの方が変更されたコード行数やファイル構造数が多く，変更が複雑であるという結果を得た．このように SATD はそれ自身が欠陥の原因になり得るだけでなく将来的にソフトウェアの保守性の低下を引き起こすため，リファクタリングによって除去されることが望ましい．

SATD はその定義からあくまでも一時的に導入されたものであり，将来的に取り除かれることを想定しているが，Potdar らの調査によれば SATD は全ソースコードファイルのうち 2.4% から 31.0% に含まれており，そのうち実際に除去される SATD は 26.3% から 63.5% のみであることが分かっている [3]．Gabriele らの 159 のオープンソースソフトウェアに対する調査 [5] では，プロジェクト内に含まれる SATD は平均して 51 個であり，プロジェクトの規模が大きくなるにつれて SATD の数も増加するとされている．また SATD が導入されてから除去されるまでに平均して 1,000 以上のコミットがされることから，SATD は長い間放置される傾向にあることがわかる．SATD が放置される原因として考えられているのが他の技術負債と比較した優先度の低さである．Potdar らの調査では SATD を 5 つの種類に分類したところ，最も割合が高いのは設計に関する負債であり，これは SATD 全体の 42% から 84% を占めていることがわかった [6]．ここでいう設計に関する負債とは，具体的には不十分な抽象化，長すぎるメソッド，一時的な回避策や不十分な実装などを指し，これらはソフトウェアの動作には直接関係しないため他の SATD と比較して除去の優先度が低くなっている．

### 2.3 SATD の検出手法

SATD は負債を導入した開発者自身によってソースコードコメントに文書化される特性から，ソースコードコメントを通じた検出が可能である．Potdar らは Eclipse，Chromium OS，Apache HTTP サーバー，ArgoUML の 4 つのプロジェクトに含まれる 101,762 のコメントを手作業で解読することで SATD の検出を行い， SATD コメントに多用される 63 の単語・フレーズを発見した [3]．ほかにもソースコードコメントを利用した SATD の検出手法としてこれまでに正規表現を用いたもの [10]，自然言語処理 (NLP) を用いたもの [5]，テキストマ

イニングを用いたもの [11] などが存在している．2.1 節で述べたソースコードメトリクスやコードスメルによって検出される技術負債は高い誤検出率の影響を受ける可能性があるのに比べて，ソースコードコメントを通して検出される技術負債は開発者が自ら負債と認めている特性から信頼度が高いとされている [12] [5].

## 3. 調 査 手 法

3. 章では, SATD を指すコメントの内容と, 実際にその SATD に対して加えられた修正の類似性の調査手法を述べる．今回は調査対象のプログラム言語を Java に限定している．調査にあたって，以下の手順で必要なデータを得た．

（1） SATD が削除されたコミットの特定
（2） SATD コメントが指すコード範囲の特定
（3） SATD に対する修正の類似度の算出

### 3.1 SATD が削除されたコミットの特定

3.1 節では，SATD コメントとその SATD に対して加えられた修正の類似性を調査するため，プロジェクトの各コミットからソースコードコメントの削除を抽出し，ソースコードコメントを通じて SATD を特定する手法について述べる．具体的には,

（1） コメントが削除されたコミットの特定
（2） SATD コメントの識別
（3） SATD が削除されたコミットの特定

の 3 つの工程で実現する．

#### 3.1.1 コメントが削除されたコミットの特定

まずはじめにソースコードコメントが削除されたコミットの特定を行う．今回はバージョン管理システムの Git を使用している．調査対象を Java で記述されたコードに限定しているので，プロジェクトのすべてのコミットを対象に `git show` コマンドを実行し， Java ファイルの変更箇所を確認する．コメントの除去があったコミットのハッシュ値とコメントの内容を記憶する．複数行にまたがるコメントは 1 つのコメントとして処理する．また，SATD と無関係なコメントを除くために，以下のコメントは対象としない．

- 自動的に生成されたコメント
- コメントアウトされたソースコードの断片
- ライセンスコメント

#### 3.1.2 SATD コメントの識別

2.2 節で記述した通り，SATD は SATD コメントを通じてその存在箇所を特定することができる．今回は Huang らによるテキストマイニングを用いた SATD コメント識別手法 [11] を使用した．Huang らの検出ツールでは，入力として受け取ったテキストのラベル (w/SATD or w/ SATD) を出力する．

#### 3.1.3 SATD が削除されたコミットの特定

ここまでで SATD コメントが削除されたコミットの特定を行った．しかし，SATD コメントが削除されることと SATD が削除されることは同義ではない [4], [13]．SATD コメントが削除された場合，以下の 3 つのケースに分類することができる．

（1） メソッド/クラスの削除に伴って SATD コメントも削除された
（2） ソースコード部分に変更はなく，SATD コメントが削除された
（3） ソースコード部分に変更があり，SATD コメントが削除された

(1) のケースと (2) のケースはいずれも偶発的に起こった SATD コメントの除去であり，SATD の本質的な除去とはいえない．SATD の本質的な除去を特定するためには，(3) のケースのコミットのみを調査対象とする必要がある．

### 3.2 SATD コメントが指すコード範囲の特定

SATD に対する修正の類似性を求めるためには， SATD コメントが指す SATD の範囲を特定する必要がある．今回の調査では SATD コメントの削除と同時に削除された箇所を SATD コメントが指すコード範囲，SATD コメントの削除と同時に編集された箇所を SATD に対する修正範囲とする．

### 3.3 SATD に対する修正の類似度の算出

最後に， SATD コメントの内容と SATD に対して行われる修正の類似性を調査する．手順としては， SATD コメントをコメント内容から分類したのち，各クラスタ内におけるコードの修正内容の類似度を求める．コードの修正内容の類似度の算出には局所アラインメントのスコアを使用しており，トークン列に変換したソースコードのアラインメントスコアが閾値 $\alpha$ を超える場合に修正が類似とする．

#### 3.3.1 SATD コメントのクラスタリング

SATD コメントの内容をベクトル化し，クラスタリングを行う．コメントのベクトル化手法には Doc2Vec を用いている．Doc2Vec は gensim [14] ライブラリを使用して Python で実装した．クラスタリングには X-means 法 [15] を用いた．X-means 法とは教師なし学習である K-means 法の拡張アルゴリズムで，クラスタ数 K を自動決定するものである．X-means 法によるクラスタリングは PyClustering [16] ライブラリによって実装している．このとき， SATD を表現するコメントに頻出する単語はクラスタリングにおいて重要な役割を果たさないため，無視している．

#### 3.3.2 分類した SATD に対する修正の類似性の調査

上で得たクラスタ内での修正の類似性を調査する．具体的には，修正箇所のトークンを文字列に変換し，クラスタ内での任意の 2 トークン間での局所アラインメントスコアを求める．

a） 修正箇所のコードのトークン化

修正内容の比較を行うために，修正箇所のコードのトークン化，文字列への変換を行う．この作業は修正によって削除されたコード部と挿入されたコード部に分けて行う．修正箇所のコードの文字列への変換は以下の手順で行う．

（1） コメントを削除したコード片に対して字句解析を行う
（2） ユーザ定義の変数や関数等はすべて同一トークンとみなす
（3） 字句解析によって得たトークンをそれぞれ一文字に変換する
（4） 空白・改行を削除して文字列を得る

コードのトークン化の例を図 1，図 2，図 3，図 4 に示す．

```
StringBuffer buf = new StringBuffer();
for(int i = 0; i < token.length; i++){
  buf.append(token[i]);
}
```

図 1　コメントを削除し字句解析を行う

```
      0       0  = new       0     ();
for(int 0 = 0; 0 <   0  .  0   ; 0++){
   0 .  0   (  0  [0]);
}
```

図 2　識別子や定数をすべて同一トークンとみなす

```
      0       0  =  1       0     ();
 2 ( 3  0 = 0; 0 <   0  .  0   ; 0++){
   0 .  0   (  0  [0]);
}
```

図 3　字句解析によって得たトークンを文字に変換

```
00=10();2(30=0;0<0.0;0++){0.0(0[0]);}
```

図 4　空白・改行を削除して文字列を得る

b）　局所アラインメント

修正の類似性の調査には局所アラインメント [17] を用いる．局所アラインメントとは，2 つの文字列間の類似する部分を求めるアルゴリズムである．2 つの文字列を比較し，文字の挿入のあった部分に - （ギャップ）を入れることで文字列の対応する位置を合わせる．この結果得られる文字列のことをアラインメントという．一般に，2 つの文字列から構成可能なアラインメントは複数存在する．そこで得られたアラインメントを評価するために局所アラインメントのスコアを導入する．局所アラインメントのスコア $score$ は以下の式から導出される．

$$score = match - mismatch - gap$$

上記では，$match$ は位置が一致する文字の数，$mismatch$ は一致しない文字の数，$gap$ は挿入した “-” の数である．アラインメント間での文字の一致が多いほどスコアは高くなり，文字の削除・挿入が多いほどスコアは低くなる．局所アラインメントのスコアの算出には Smith-Waterman アルゴリズム [18] を用いた．

c）　修正の類似度の算出

修正の類似度はクラスタごとに算出する．あるクラスタ $C = \langle m_0, m_1, \ldots, m_{N-1} \rangle$ における任意の修正 $m(i)$，$m(j)$ をそれぞれ削除行と挿入行に分割し，削除行ごと，挿入行ごとに局所アラインメントのスコアを算出する．ここで $m(i)$，$m(j)$ 間の削除行の局所アラインメントのスコアを $score\_del(i,j)$，挿入行の局所アラインメントのスコアを $score\_ins(i,j)$ とする．

$M \times M$ の表 $H\_del(N,N)$，$H\_ins(N,N)$ を用意し，以下に従って表を埋める．

$$H\_del(i,j) = \begin{cases} \texttt{if } score\_del(i,j) > \alpha \texttt{ then } 1 \\ \texttt{else } 0 \end{cases}$$

$$H\_ins(i,j) = \begin{cases} \texttt{if } score\_ins(i,j) > \alpha \texttt{ then } 1 \\ \texttt{else } 0 \end{cases}$$

つぎに，$\mathrm{H}(i,j) = H\_del(i,j)$ and $H\_ins(i,j)$ となる表 $H$ を用意する．ここで $H(i,j) = 1$ のとき，$m(i)$ と $m(j)$ は削除行と挿入行のそれぞれが類似することを意味している．$H(i,j) = 1$ となる $m(i)$ と $m(j)$ を**類似する修正**とする．

最後に，$i \in (0, 1, ..., N-1)$ に対して以下の計算を行う．

$$common(i) = \sum_{j=0}^{N-1} H(i,j)$$

これは同一クラスタ内に $m(i)$ と類似する修正が $common(i)$ 個含まれることを意味する．$common(i)$ が最大となるような $i$ を $i\_max$ とするとき，$m(i\_max)$ を**クラスタを代表する修正**とし，

$$similarity(C) = \frac{common(i\_max)}{N}$$

をクラスタ C における**修正の類似度**とする．

## 4. 調 査 実 験

SATD コメントを通じた修正支援を実現するためには，SATD コメントの内容と SATD への修正内容に類似性を見つけ出す必要がある．4. 章では，3. 章で述べた手法を実際にオープンソースプロジェクトに適用し，SATD コメントの内容と SATD への修正内容にどの程度類似性がみられるかを調査する．

**実 験 対 象**

今回の調査は Camel，Gerrit，Hadoop，Log4j，Tomcat の 5 つの Java オープンソースプロジェクトを対象に行っている．それぞれアプリケーションドメインや規模，コントリビュータの数が異なるもので，開発が活発に行われていること，十分な量のコメントが含まれていることを条件に，Maldonad らによって選定されたものである [10]．実験対象のプロジェクト名，Java ファイル数，総リビジョン数，総コード行数，コントリビュータ数を表 2 に示す．

表 2　調査対象プロジェクトの規模

| プロジェクト名 | Java ファイル数 | 総リビジョン数 | KLOC | コントリビュータ数 |
|---|---|---|---|---|
| CAMEL | 17,463 | 46,005 | 1,830 | 647 |
| GERRIT | 2,793 | 36,093 | 380 | 368 |
| HADOOP | 11,314 | 57,242 | 2,560 | 273 |
| LOG4J | 1,894 | 11,007 | 243 | 86 |
| TOMCAT | 2,404 | 20,420 | 570 | 39 |

### 4.1　SATD に対する修正の類似度の算出

3.1 節で得た SATD コメントに対してクラスタリングを行い，クラスタごとに修正の類似度を求める．クラスタに含まれ

表 3　SATD に対する修正の類似性

| プロジェクト名 | SATD コメント数 | クラスタの修正の類似度の平均 |
|---|---|---|
| CAMEL | 263 | 0.53 |
| GERRIT | 65 | 0.40 |
| HADOOP | 941 | 0.89 |
| LOG4J | 37 | 0.77 |
| TOMCAT | 915 | 0.72 |

表 4　有効なクラスタの割合

| プロジェクト名 | SATD コメント数 | 有効なクラスタの割合 | 有効なクラスタに分類されるコメントの割合 |
|---|---|---|---|
| CAMEL | 263 | 0.16 | 0.12 |
| GERRIT | 65 | 0.00 | 0.00 |
| HADOOP | 941 | 0.78 | 0.40 |
| LOG4J | 37 | 0.75 | 0.10 |
| TOMCAT | 915 | 0.48 | 0.16 |

る全修正を削除行と挿入行に分割し，任意の 2 修正間で削除行ごと，挿入行ごとに局所アラインメントのスコアを求める．削除行の局所アラインメントスコアと挿入行の局所アラインメントスコアがともに閾値 $\alpha$ を超えるような修正の組を**類似する修正**とする．類似する修正が最も多くなるような修正を**クラスタを代表する修正**とし，クラスタを代表する修正に類似する修正の数をクラスタ内の全修正の数で割ったものを**クラスタ内の修正の類似度**として求めた．また，修正支援への応用の可不可を判断するために，修正の類似度 $> \beta$ となるクラスタを**有効なクラスタ**とし，全クラスタ数に占める有効なクラスタの割合と有効なクラスタに分類されるコメントの割合を求めた．

比較する 2 つのトークン列のうち長い方の長さを $L$ とするとき，閾値 $\alpha$ を以下のように設定した．

$$\alpha = \begin{cases} L \times 0.6 \ (L < 30 \text{ のとき}) \\ 18 \ (L \geq 30 \text{ のとき}) \end{cases}$$

調査対象のすべてのプロジェクトに対して各クラスタの修正の類似度を求めた．得られた修正の類似度の平均と SATD コメント数の関係を表 3 に示す．表 3 から，一般に SATD コメントが多いほど各クラスタ内における修正の類似度は高くなることがわかる．

修正支援への応用の可不可を判断するために，各プロジェクトにおいて全クラスタ数に占める有効なクラスタの割合と有効なクラスタに分類されるコメントの割合を求めた．ここで，有効なクラスタの閾値 $\beta = 0.8$ としている．得られた結果を表 4 に示す．

Camel や Gerrit のように有効なクラスタの割合が低い場合，複数の種類の修正が含まれるクラスタが多いことを意味している．Hadoop は SATD コメント数が多いことが要因となって有効なクラスタの割合と有効なクラスタに分類されるコメントの割合が高くなっているが，同様に SATD コメントが多い Tomcat の値はあまり高くない．これは Tomcat に記述されている SATD コメントが “TODO” や “FIXME” の単語のみのものが多く，コメントのクラスタリングがうまく行われなかったことが原因である．Log4j は有効なクラスタの割合に対して有効なクラスタに分類されるコメントの割合が極端に低くなっている．これはそもそもの SATD コメント数が少ないことが原因で，コメントに類似性が見られなかった多数の修正がひとつのクラスタに分類されているためである．

コメント内容によってクラスタリングされた修正内容を修正支援に使用するには，有効なクラスタの割合と有効なクラスタに分類されるコメントの割合が高いものが有用である．よって今回の調査対象のプロジェクトの中では Hadoop のデータが修正支援に最も有用である

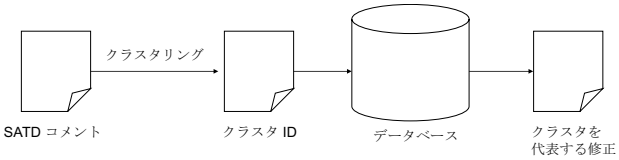


図 5　修正支援の処理概要

## 5. 修正支援ツールの試作

．最後に SATD コメントの内容と SATD への修正の類似性を利用した修正支援ツールの試作を行った．本章ではデータベースの構築方法とケーススタディを記す．

### 5.1 データベースの構築

4. 章で得たクラスタをデータベースの構築に使用する．具体的には，クラスタ内で最も多く見られたコードの修正内容をクラスタを代表する修正とし，クラスタの ID とクラスタを代表する修正の内容をキーとしてデータベースの構築を行った．データベースの構築には GNU Database Manager を用いた．また言語は Python を使用している．実際の修正支援では，追加された SATD コメントを入力として受け取り，クラスタリングを行い，該当クラスタを代表する修正を出力することを想定している．修正支援の処理概要を図 5 に示す．

### 5.2 ケーススタディ

ここでは試作した修正支援ツールの実行例を記す．データベースの構築には全体の SATD コメントのうち 90% を使用し，残り 10% をケーススタディに用いた．

Hadoop のある SATD コメントを入力として修正支援ツールを実行した例を図 6 に示す．修正支援ツールを実行すると，該当プロジェクト名のウィンドウが立ち上がり，入力として渡した SATD コメントと SATD コメントの周辺コード，データベースから得られたクラスタを代表する修正内容が表示される．図 6 から，入力 として与えた SATD コメントの周辺に実際に加えられた修正と出力として得られた該当クラスタを代表する修正が類似していることがわかる．

## 6. ま と め

5 つの Java オープンソースプロジェクトに対して， SATD コメントの内容と実際に SATD に加えられた修正の類似性を調査した．

調査実験では，一般的に SATD コメントが多いほどクラス

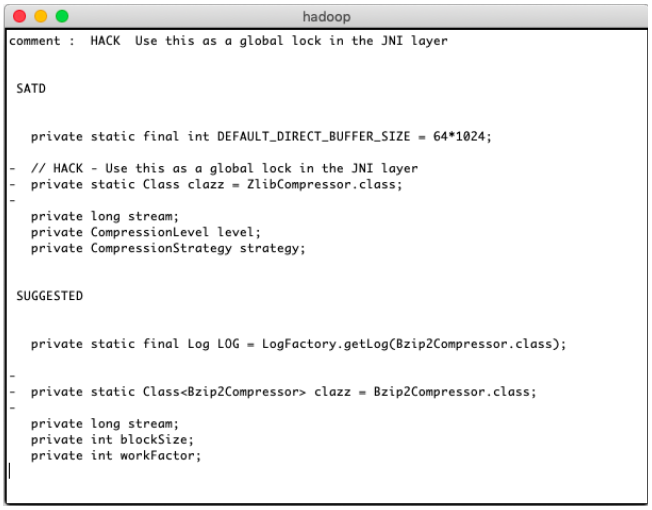


図 6　修正支援ツールの実行例

タ内の修正の類似性が高くなることがわかった.SATD コメント数が多くても有効なクラスタに分類されるコメントの割合が低いプロジェクトも存在する. これは SATD コメントが詳細に記述されていないことが原因であり，有効なクラスタに分類されるコメントの割合は，SATD コメントの数だけでなくコメントルールが大きく影響することがわかった. 有効なクラスタの 割合に対して有効なクラスタに分類されるコメントの割合が極端に低くなっている. これは そもそもの SATD コメント数が少ないことが原因で，コメントに類似性が見られなかった多数の修正がひとつのクラスタに分類されているためである.

修正支援ツールの試作では，調査実験での結果から得た修正支援に有用と判断された Hadoop のクラスタを用いてデータベースを構築した．全 SATD コメントのうち 90% をデータベースの構築に使用し，残りの 10% をケーススタディに使用した．実際に Hadoop プロジェクトのある SATD コメントを入力として与えたところ，入力 として与えた SATD コメントの周辺に実際に加えられた修正と出力として得られた該当クラスタを代表する修正が類似していること確認できた.

今回は調査対象を Java で記述されたものに限っているが，技術的負債の特定にソースコードコメントを使用しているため，自然言語でコメントを記述できる言語には今回の手法を応用できる.

また今回はオープンソースプロジェクトのみを調査対象として扱っている．そのため，コメントルールが厳格に定められている企業が開発したプロジェクトなどに関しては，今回の調査結果と異なる結果になる可能性がある.

## 文　　献


[1] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, “Managing technical debt in software-reliant systems,” Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER ’10), pp.47–52, 2010.

[2] Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” J. Syst. Softw., vol.101, no.C, pp.193–220, March 2015.

[3] A. Potdar and E. Shihab, “An exploratory study on self-admitted technical debt,” Proceedings of the 30th International Conference on Software Maintenance and Evolution(ICSME2014), pp.91–100, 2014.

[4] S. Wehaibi, E. Shihab, and L. Guerrouj, “Examining the impact of self-admitted technical debt on software quality,” Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER2016), pp.179–188, March 2016.

[5] G. Bavota and B. Russo, “A large-scale empirical study on self-admitted technical debt,” Proceedings of the 13th International Conference on Mining Software Repositories (MSR2016), pp.315–326, ACM, New York, NY, USA, 2016.

[6] E.d. Maldonado and E. Shihab, “Detecting and quantifying different types of self-admitted technical debt,” Proceedings of the 7th International Workshop on Managing Technical Debt (MTD2015), pp.9–15, Oct. 2015.

[7] 一ノ瀬智浩，畑　秀明，松本健一，“ソースコード上の技術的負債除去を活性化させるゲーミフィケーション環境の開発，” 2016 年度 情報処理学会関西支部 支部大会 講演論文集，pp.1–4，2016.

[8] R. Marinescu, “Detection strategies: metrics-based rules for detecting design flaws,” Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM2004), pp.350–359, 2004.

[9] N. Zazworka, R.O. Spínola, A. Vetro’, F. Shull, and C. Seaman, “A case study on effectively identifying technical debt,” Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE2013), pp.42–47, 2013.

[10] E.D. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, “An empirical study on the removal of self-admitted technical debt,” Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME 2017), pp.238–248, Sept. 2017.

[11] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, “Identifying self-admitted technical debt in open source projects using text mining,” Empirical Softw. Engg., vol.23, no.1, pp.418–451, Feb. 2018.

[12] F.A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, “Antipattern and code smell false positives: Preliminary conceptualization and classification,” Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016), pp.609–613, March 2016.

[13] F. Zampetti, A. Serebrenik, and M. Di Penta, “Was self-admitted technical debt removal a real removal?: An in-depth perspective,” Proceedings of the 15th International Conference on Mining Software Repositories (MSR2018), pp.526–536, 2018.

[14] R. Řehůřek and P. Sojka, “Software Framework for Topic Modelling with Large Corpora,” Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, pp.45–50, May 2010.

[15] D. Pelleg and A. Moore, “X-means: Extending k-means with efficient estimation of the number of clusters,” In Proceedings of the 17th International Conference on Machine Learning, pp.727–734, June 2000.

[16] A. Novikov, “annoviko/pyclustering: pyclustering 0.8.2 release,” Nov. 2018. https://doi.org/10.5281/zenodo.1491324

[17] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, NY, USA, 1997.

[18] T.F. Smith and M.S. Waterman, “Identification of common molecular subsequences.,” Journal of molecular biology, vol.147, no.1, pp.195–197, March 1981.