修士学位論文

題目

ソースコードコメントに着目した
技術的負債に対する修正の類似性の調査と支援ツールの試作

指導教員
井上 克郎 教授

報告者
岡島 早紀

平成 31 年 2 月 6 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 30 年度 修士学位論文

ソースコードコメントに着目した技術的負債に対する修正の類似性の調査と支援ツールの試作

岡島 早紀

内容梗概

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的負債という [2]．技術的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品質や保守において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．

近年，技術的負債に関する新しい概念として自認された技術的負債（Self-Admitted Technical Debt：SATD）が提唱された [14]．SATD とは技術的負債の中でも特に開発者が意図的に導入したもので，自然言語を使用してソースコードコメントにその存在を記述しているものを指す．技術的負債がソースコードコメントを通じて文書化されるという特徴から，既存研究では自然言語処理や正規表現などを用いた SATD の検出手法が提案されており，開発者が自ら負債と認めていることからソースコードコメントを通じて検出される技術負債は信頼度が高いとされている [4] [1]．SATD を含むファイルへの修正は複雑になる [18] ことから，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 の多くを占めるのは設計に関するもの [1] で，そ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，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 [1]．

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性の予測の信頼度は 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．信頼度が高いプロジェクトにおいて試作ツールを実行したところ，修正支援ツールが有用であることを確認できた．

主な用語

ソースコードコメント

Self-Admitted Technical Debt

修正支援

目 次

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解決策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的負債という [2]．開発者は技術的負債をのちに解消するつもりでも，技術的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 [8]．技術的負債の検出手法にはソースコードメトリクスやコードスメル等を用いた静的コード分析などがある [10] [20]．

近年，自認された技術的負債（SATD）という概念が提唱された．これは開発者自身が技術的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指す [14]．開発者自身が自認しているという特徴からソースコードコメントを使用した SATD 検出は前述の静的コード分析を使用した技術的負債の検出と比較して信頼度が高いことが分かっている [4] [1]．Shihab らの研究により SATD を表現するのに多用される単語・フレーズが特定されており，既存研究における SATD 検出手法ではそれらを利用した自然言語処理や正規表現，テキストマイニングなどを用いた手法が提案されている [1] [9] [7]．

Wehabi らの調査 [18] 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れる．しかし，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 [14]．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 [1]．SATD が除去されず長い間放置される理由として，他の技術的負債と比較した優先度の低さが考えられる．SATD を 5 つの種類に分類したところ，最も多く見られたのは設計に関する SATD であった [3]．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 [21]．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 へと加えられる編集の類似性を調査し，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各クラスタにおいてコードの修正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修正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 α を超えるものを修正が類似すると定義した．また特定のクラスタ内で一番多く見られた修正内容をクラスタを代表する修正として，クラスタの ID と編集内容をキーとするデータベースを構築し，入力として受け取ったコメントに対してクラスタリングを行い該当するクラスタを代表する修正を出力する修正支援ツールを試作した．

以降，2 章では本研究のベースとなる関連研究について説明する．3 章では，技術的負債を指すソースコードコメントの内容と技術的負債に加えられた修正との間にみられる類似性を調査する手法について説明する．4 章では，実際に 5 つのオープンソースプロジェクトに対して類似性の調査を行った結果を述べる．5 章では，ソースコードコメントと修正にみられる類似性を利用してデータベースを構築する方法とその実行例を記し，6 章では本研究のまとめを述べる．

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

2 背景

本章では本研究の基となる関連研究について説明する．2.1 節では技術的負債，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，2.4 節ではバージョン管理システムについて説明する．

2.1 技術的負債

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的負債という [2]．技術的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の負債について述べる．既存研究では，技術的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 [8]．技術的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 [10] [20] が使用される．

2.2 自認された技術的負債

自認された技術的負債 (Self-Admitted Technical Debt : SATD) とは，Potdar と Shihab らによって提案された概念であり，技術的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 [14]．このようなソースコードコメントに見られる技術的負債を自然言語で表現したものを 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．

Wehabi らの研究 [18] 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，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れる．

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 [14]．Gabriele らの 159 のオープンソースソフトウェアに対する調査 [1] 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的負債と比較した優先度の低さである．Potdar らの調査では SATD を設計，欠陥，ドキュメント，要求，テストの 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 SATD 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 [3]．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ことが想定できる．

2.3 SATD への対処

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．このような特性から，SATD はソースコードコメントを通し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 [14]．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 [9]，自然言語処理 (NLP) を用いたもの [1]，テキストマイニングを用いたもの [7] 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的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的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 [4] [1]．

Zamptti らは 5 つの Java オープンソースプロジェクトを対象に SATD コメントの除去に際して行われるソースコードの編集について調査を行った [19]．SATD コメントの除

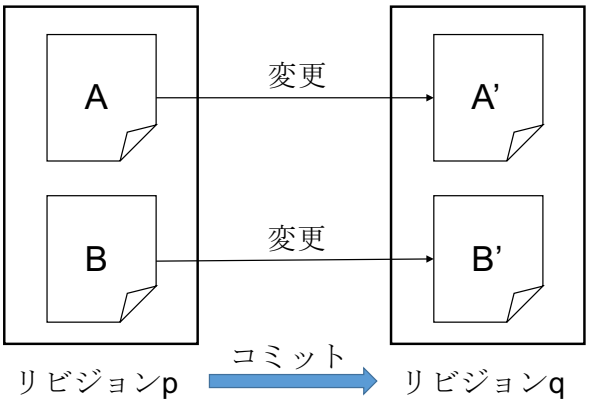


図 1: コミットとリビジョンの例

去と同時に SATD のコード部に加えられる編集を 3 種類 (メソッド・クラスの削除に伴う SATD 除去，メソッドの編集に伴う本質的な SATD 除去，メソッドの編集を伴わない偶発的な SATD 除去) に分類したところ，一般的に SATD の除去はソースコードの編集に伴って行われるが，クラス・メソッドの削除に伴う SATD 除去も多く見られることがわかった．

2.4 バージョン管理システム

バージョン管理システムとは，ファイルやディレクトリに対して施される追加・編集・削除などの変更履歴を管理するシステムのことである．このような変更履歴のデータを管理する場所をリポジトリ（repository）という．ファイルやディレクトリに対して行った編集をリポジトリに反映させる操作のことをコミット（commit）といい，コミットによって得られる状態のことをリビジョン（revision）という．コミットとリビジョンの例を図 1 に示す．コミットを実行すると，コミット日時，コミットを実行した開発者，コミットメッセージ，前回のコミットからの差分が自動的に生成されるコミットのハッシュ値とともに記録される．コミットのハッシュ値はリビジョン番号，SHA-1 ともいい，リビジョン番号を指定することで特定のリビジョンにおけるファイルの内容や別のリビジョンとの差分を得ることができる．

バージョン管理システムには集中型のものと分散型のものが存在する．集中型バージョン管理システムでは 1 つのリポジトリを開発者全員で共有するのに対し，分散型バージョン管理システムでは共有リポジトリ（リモートリポジトリ）のほかにローカル PC の上にローカルリポジトリというリポジトリを作成する．分散型バージョン管理システムではコミットはローカルリポジトリに対して行い，ローカルリポジトリの内容をリポートリポジトリに反映させる操作のことをプッシュ（push）という．代表的なものとして，集中型リポジトリには SVN，分散型リポジトリには Git が挙げられる．

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性の算出

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. ソースコード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．SATD の特定には，Huang らによるテキストマイニングを用いた SATD 識別ツール [7] を使用している．

3.1.1　ソースコード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 Java ファイルの変更箇所を確認する．ここで変更箇所の確認には `git show` コマンドを使用する．`git show` コマンドとは，“`git show` コミットのハッシュ値” とコミットを指定することでコミットの詳細を得ることができるコマンドである．また，“`git show` コミットのハッシュ値 – ファイルパス” とコミットのハッシュ値とファイルパスを指定することで特定ファイルの特定コミットにおける変更の詳細を得ることができる．実際に `git show` コマンドを実行した例を以下に示す．

Listing 1: `git show` を実行した例

```
1 commit 01b411892727b2ca1d49511bce2c6aef4ab64636
2 Author: Antonin Stefanutti <antonin@stefanutti.fr>
```

```
3  Date: Wed Apr 20 17:36:09 2016 +0200
4
5      Use lambda expression to produce Metrics ratio gauge
6
7  diff --git a/examples/Application.java b/examples/Application.java
8  index 77c606ccaec..2202d45a368 100644
9  --- a/examples/Application.java
10 +++ b/examples/Application.java
11 @@ -92,15 +92,8 @@ class Application {
12    @Produces
13    @Metric(name = "success-ratio")
14    // Register a custom gauge that's the ratio of the 'success' meter on the 'generated' meter
15 -  // TODO: use a lambda expression and parameter names when Java 8 is a pre-requisite
16 -  Gauge<Double> successRatio(@Metric(name = "success") final Meter success,
17 -  @Metric(name = "generated") final Meter generated) {
18 -  return new RatioGauge() {
19 -  @Override
20 -  protected Ratio getRatio() {
21 -  return Ratio.of(success.getOneMinuteRate(), generated.getOneMinuteRate());
22 -  }
23 -  };
24 +  Gauge<Double> successRatio(Meter success, Meter generated) {
25 +  return () -> Ratio.of(success.getOneMinuteRate(), generated.getOneMinuteRate()).getValue();
26 +  }
```

1 行目から 6 行目ではコミットの詳細が得られる．具体的には，コミットのハッシュ値，コミッター，コミット日時，コミットメッセージを得ることができる．7 行目から 26 行目ではファイルの詳しい変更箇所が得られる．“—” から始まる 9 行目が変更前のファイル名，“+++” から始まる 10 行目が変更後のファイル名を表しており，11 行目の “@@” で囲われた数字はファイル内でどの箇所に変更があったかを表している．この例では変更前の 92 行目から 15 行分，変更後の 92 行目から 8 行分の差分であることがわかる．最後に 11 行目 “@@” の後から 26 行目は実際のファイル内での変更を表している．ここで “ + ” から始まる行が挿入された部分， ” - ” から始まる行が削除された部分であり，この例では 15 行目から 23 行目が削除されており， 24 行目から 25 行目が挿入された部分であることがわかる．

プロジェクトのすべてのコミットに対して `git show` コマンドを実行し，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．ここで， Java には単一行コメントの記述法と複数行コメントの記述法が用意されているが，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，無関係なコメントを調査対象から除くために，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

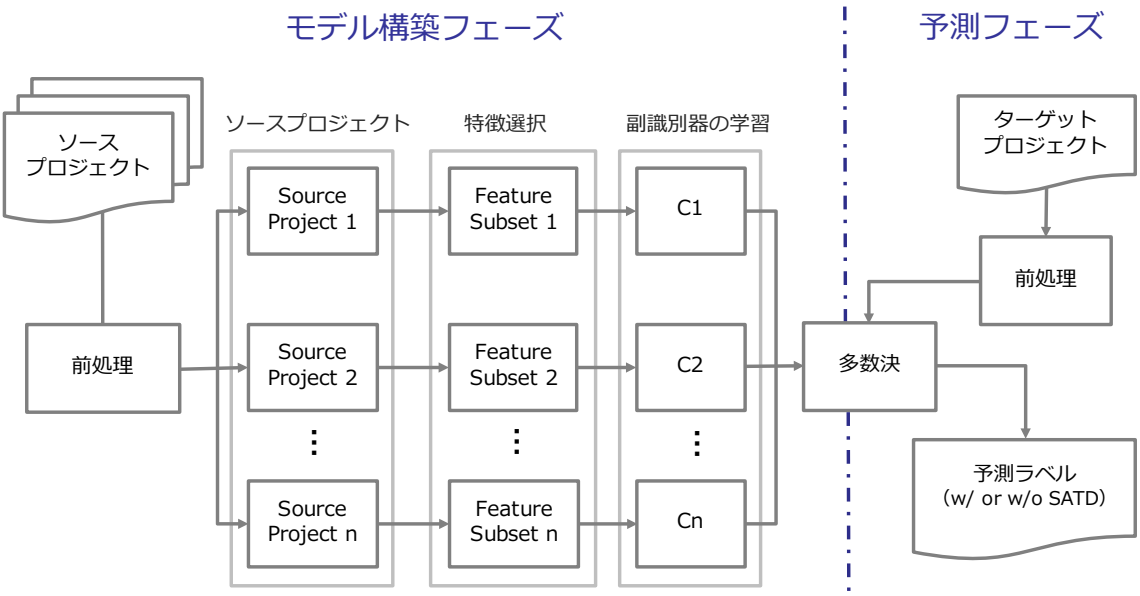


図 2: Huang らによる SATD コメント識別手法

- 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 コメント識別手法を使用した [7]．図 2 に Huang らの SATD コメント識別手法の全体のフレームワークを示す．Huang らの SATD コメント識別手法はモデル構築フェーズと予測フェーズに分かれており，モデル構築フェーズでは複数のプロジェクトのテキストデータに前処理を行ったものをトレーニングデータとして使用し特徴選択を行ったもので副識別器のを学習し，予測フェーズではモデル構築フェーズで得られた副識別器の多数決を SATD コメントであるかそうでないかの予測ラベルとして出力する．予測フェーズとモデル構築フェーズそれぞれの詳細を以下に記す．

テキスト前処理　Huang らによる SATD コメント識別手法ではソースプロジェクトのコメント，ラベルをもとに学習を行う．ソースプロジェクトは Shihab らによって提供されたデータセットを使用している [3]．Shihab らのデータセットは ArgoUML，Columba，Hibernate，JEdit，JFreeChart，JMeter，JRuby，SQuirrel SQL のコメントとそのコメントが SATD コメントに該当するかどうかのラベルが付与されている．コメントは自然言語で記述されているが，トークン化，ストップワードの除去，ステミングの 3 つの過程で前処理を行うこと

で特徴となる単語を抽出する．

Step1 トークン化　トークンとは，自然言語処理を行う際に文章の最小単位として扱われる文字や文字列を指す．トークン化の過程では，コメント中に現れる空白や句読点を区切り文字に使用して文章を単語に分割する．またこの識別器ではコメントに含まれる句読点や数字，記号をすべて除き，最後に，大文字と小文字を区別しないために大文字をすべて小文字に変換する．

Step2 ストップワードの除去　ストップワードとは，自然言語処理にあたって，文中に頻繁に使用されるためにカテゴリの分類を行う際にほとんど意味をなさないなどの理由で処理対象外とする単語のことを指す（e.g,“this”，“should”，“to”，etc）．しかし，一般的にはストップワードと呼ばれる単語でも SATD の識別目的では特別な意味を持つことがあるため，今識別器ではストップワードを長さが 2 以下もしくは 20 以上の単語，または前置詞 (e.g.,“the”，“of”，“is”，etc) のみとしたストップワードリストを作成している．

Step3 ステミング　ステミングとは，単語を語幹に落とし込む作業のことをさす．ステミングを行うことで，単語の複数形や名詞，副詞などの語形の変化の影響を受けなくなる．ステミング手法としては Poter Stemmer を用いている．

特徴選択　識別の精度を向上させるために，特徴選択を行う [6]．特徴選択の手法には，広く使用されている手法である情報ゲイン [16] を使用する．情報ゲインでは，以下の手順で情報ゲインスコアを得る．

コメントのデータセットを $C = \{(C_1, L_1), (C_2, L_2), \ldots, (C_N, L_N)\}$ とする．ここで C_i は i 個目のコメントを表し，L_i はそのコメントが SATD である (t) か SATD でない $(\bar{t})$ かのラベルを表している．C_i は単語ベクトルであり，C_i 中に現れる単語の数を n とすると C_i は $C_i = \{w_1, w_2, \ldots, w_n\}$ と表すことができる．ここで w_i は C_i 中に見られる i 番目の単語を指す．このとき特徴 w とコメント C_i の間には以下の 4 つの関係が考えられる．

- (w, t) : コメント C_i が特徴 w を含み，SATD である
- $(w, \bar{t})$: コメント C_i が特徴 w を含むが，SATD でない
- $(\overline{w}, t)$: コメント C_i が特徴 w を含まないが，SATD である
- $(\overline{w}, \bar{t})$: コメント C_i が特徴 w を含まず，SATD でない

これら 4 つの関係の確率から，特徴 w とラベル t に関する情報ゲインスコアは (1) 式で求められる．

$$IG(w,t) = \sum_{t' \in \{t,\bar{t}\}} \sum_{w' \in \{w,\overline{w}\}} p\left(w',t'\right) * \log \frac{p\left(w',t'\right)}{p\left(w'\right) * p\left(t'\right)} \tag{1}$$

すべてのコメントに対して (1) 式の計算を行い，IG スコアが最も高くなった上位 10%のコメントを特徴として使用する．

副識別器の学習　副識別器には，多項単純ベイズ [11] を使用している．多項単純ベイズとは，単純ベイズに似た分類器で,．単純ベイズを使用する利点としては，学習時間が短いことが挙げられる．コメント $C_i = (w_1, w_2, \ldots, w_n)$, とラベル L_i が与えられるとき，

$$p\left(C_i|L_i\right) = \prod_{j=1}^{|n|} p\left(w_j|L_i\right) \tag{2}$$

が成り立つ．ここで，ベイズの定理を適用すると以下が得られる．

$$p\left(L_i = t|C_i\right) = \frac{p\left(L_i = t\right) \times \prod_{j=1}^{|n|} p\left(w_j|L_i = t\right)}{p\left(C_i\right)} \tag{3}$$

$$p\left(C_i\right) = \sum_{t' \in [t,\bar{t})} p\left(L_i = t'\right) \times \prod_{j=1}^{|n|} p\left(w_j|L_i = t'\right) \tag{4}$$

今副識別器では，コメントのラベルの特定に式 (3) を使用する．

多数決によるラベルの決定　Huang らの識別器はこれまでの作業を ArgoUML，Columba，Hibernate，JEdit，JFreeChart，JMeter，JRuby，SQuirrel SQL のすべてのプロジェクトに対して行い，それぞれ副識別器を得る．予測フェーズではそれぞれの副識別器の出力するラベルの多数決をとったものを識別器が識別したラベルとして出力する．

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 [19] [18]．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 のケースは，不要になったメソッド/クラスの削除に伴って付随していた SATD コメントも削除されただけなので，SATD の本質的な削除とはいえない．2 のケースでは，ソースコード部分に変更がなく SATD コメントのみが削除された理由として，すでに SATD は削除されていたが手違いで該当 SATD コメントは削除されず，後に残されていた SATD コメントが削除された場合や SATD コメントが指す内容が間違っていたため SATD コメントのみが削除された場合などが考えられるが，いずれも当該コミットで SATD に対する修正が行われたわけではないので，SATD の削除とはいえない．3 のケースでは，SATD コメントが指す SATD に修正が加わり，SATD が削除された結果として SATD コメントが削除されたことが考えられる．よって，SATD の削除を特定するためには，3 のケースのコミットを特定する必要がある．

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．3.2 節では，SATD コメントが指すコード範囲について述べる．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．実際にソースコード中に現れた SATD コメントとそれが指すコード範囲の例を図 2 に示す．

Listing 2: SATD と SATD コメントの例

```
1  @@ −183,17 +173,7 @@ public ApplicationAttemptReport run() throws Exception {
2          return;
3        }
4
5  − // TODO need to render applicationHeadRoom value from
6  − // ApplicationAttemptMetrics after YARN−3284
7  − if (webUiType.equals(YarnWebParams.RM_WEB_UI)) {
8  − if (!isApplicationInFinalState(appAttempt.getAppAttemptState())) {
9  − DIV<Hamlet> pdiv = html._(InfoBlock.class).div(_INFO_WRAP);
10 − info("Application␣Attempt␣Overview").clear();
11 − info("Application␣Attempt␣Metrics")._(
12 − "Application␣Attempt␣Headroom␣:␣", 0);
13 − pdiv._();
14 − }
15 − }
16 + createAttemptHeadRoomTable(html);
17       html._(InfoBlock.class);
18
19       // Container Table
```

上の例では，複数行のコメントは単一コメントとして扱うため 5 行目から 6 行目が SATD コメント，7 行目から 15 行目が SATD コメントの指すコード範囲， 7 行目から 15 行目が SATD に対する修正範囲となる．

3.3 SATD に対する修正の類似性の調査

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，削除行と挿入行の両方において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 α 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 [15] 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 [13] 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 [12] ライブラリによって実装している．このとき，“TODO” や “FIXME” など 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

3.3.1 節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，修正箇所を削除行と挿入行に分割し，それぞれトークンを文字列に変換する．クラスタ内での任意の修正の組において，削除行同士，挿入行同士のトークン間での局所アラインメントスコアを求め，それぞれが閾値 α を超えるような修正を類似する修正と定義した．

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．

(a) コメントを削除した修正箇所のコード片に対して字句解析を行う

(b) 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

(c) 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

(d) 空白・改行を削除して文字列を得る

修正箇所のコードのトークン化の例を図 3 に示す．

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

(a) 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]);
}
```

(b) 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]);
}
```

(c) 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]);}
```

(d) 空白・改行を削除して文字列を得る

図 3: コードのトークン化の例

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 [5] 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．アラインメントの例を以下に示す．

$S1 =$ ‘$HEAWGEH'$

$S2 =$ ‘$GAWED'$

のとき，この文字列の組から得られるアラインメントの例のひとつとして，

$S1' =$ ‘$AWGEH'$

$S2' =$ ‘$AW - ED'$

がある．

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．先程の例

$S1' =' AWGEH'$

$S2' =' AW-ED'$

では，文字の一致 $match = 3$，文字の不一致 $mismatch = 1$，挿入した “-” の数 $gap = 1$ であることから，$score = 1$ となる．

Smith-Waterman アルゴリズム　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 [17] を用いる．Smith-Waterman アルゴリズムは 4 つの工程から局所アラインメントのスコア最大となるアラインメントを得ることができる．2 つの文字列 $S1 = \langle a_1, a_2, \ldots, a_M \rangle, S2 = \langle b_1, b_2, \ldots, b_N \rangle$ とするとき，Smith-Waterman アルゴリズムでは以下の手順で局所アラインメントのスコア最大となるアラインメントを得る．

(a) $(M+1) \times (N+1)$ の表 $F(M+1, N+1)$ を用意する

(b) 1 番目の行と列を 0 で初期化する

(c) 下の数式に従って表を埋める

$$\mathrm{F}(i,\mathrm{j}) = \max \begin{cases} F(i-1, j-1) + s\left(x_i, y_j\right) \\ F(i-1, j) - 1 \\ F(i, j-1) - 1 \\ 0 \end{cases}$$

(d) 表中の最大のスコアを持つセルからトレースバック

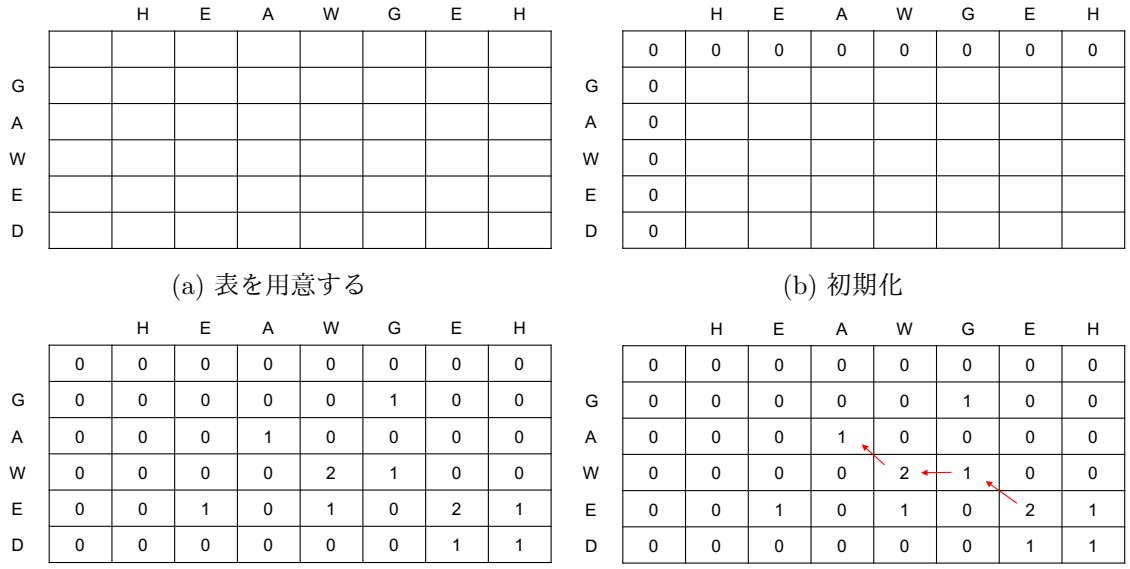
先程の文字列の組 $S1$，$S2$ に対して Smith-Waterman アルゴリズムを実行した例を図 4 に示す．図 4 では，表中のスコアが最大となる $F(4,6)$ からトレースバックを行い，得られるスコア最大のアラインメントは

$S1' =' AWGE'$

$S2' =' AW-E'$

となり，そのスコアは 2 である．

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の組 $m(i)$，$m(j)$ 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



(a) 表を用意する　(b) 初期化

(c) 数式に従って表を埋める　(d) トレースバック

図 4: Smith-Waterman アルゴリズムの例

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．

全修正 $\{m(i) \mid i \in \{0,1,...,N-1\}\}$ 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，

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

から求まる $similarity(C)$ をクラスタ C の**修正の類似度**とする．

修正支援への応用の可不可を判断するため，$similarity(C)$ が閾値 β を超えるクラスタを**有効なクラスタ**とし，全クラスタに占める有効なクラスタの割合を求める．有効なクラスタの割合が低いと，SATD コメント と修正の間に類似性が見られないクラスタが多数存在することを意味する．さらに，有効なクラスタに分類されるコメントの割合を評価の指標として使用する．

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 [9]．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，LOC，コントリビュータ数を表 2 に示す．表中の LOC（Lines of Code）は総コード行数を表す．

表 2: 各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	LOC	コントリビュータ数
CAMEL	17,463	46,005	1,829,500	647
GERRIT	2,793	36,093	380,286	368
HADOOP	11,314	57,242	2,560,319	273
LOG4J	1,894	11,007	243,438	86
TOMCAT	2,404	20,420	570,392	39

今回の調査では合計 35,868 の Java ファイルに対して調査を行った．また総リビジョン数は 170,767 となった．

4.1　SATD が削除されたコミットの特定

4.1.1　ソースコードコメントが削除されたコミットの特定

各プロジェクトのすべてのコミットをトレースし，削除されたコメント数を調べた．結果を表 3 に示す．ここで，節で述べたように，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はすべてカウントせず，複数行にまたがるコメントは単一コメントとしてカウントしている．

表 3: 削除された総コメント数

プロジェクト名	総コメント行数	削除されたコメント行数	削除されたコメント数
CAMEL	445,820	92,389	30,202
GERRIT	60,774	19,879	6,901
HADOOP	554,744	205,626	55,467
LOG4J	73,373	30,397	5,970
TOMCAT	162,267	173,295	42,850

4.1.2 SATD が削除されたコミットの特定

Huang らによるテキストマイニングを用いた SATD コメントの識別ツールに 4.1.1 節で得たコメントを入力として与える．実際に SATD コメントとして識別されたコメントの数は表 4 のようになった．

表 4: 削除された SATD コメント数

プロジェクト名	削除された コメント数	削除された SATD コメント数	削除されたコメント に占める SATD コメントの割合
CAMEL	30,202	263	0.9 %
GERRIT	6,901	65	0.9 %
HADOOP	55,467	941	1.7 %
LOG4J	5,970	37	0.6 %
TOMCAT	42,850	915	2.1 %

削除された総コメント数に対して， SATD コメントは全体の 0.6 - 2.1 %含まれることがわかった．

4.2 SATD コメントが指すコード範囲の特定

3.2 節で述べたように，SATD コメントの除去に伴って削除されたコード部を SATD コメントが指すコード範囲，SATD コメントの除去に伴って追加・削除されたコード部を SATD に対する修正とする．

表 5: SATD に対する修正の類似性

プロジェクト名	SATD コメント数	コントリビュータ数	修正の類似度の平均
CAMEL	263	647	0.53
GERRIT	65	368	0.40
HADOOP	941	273	0.89
LOG4J	37	86	0.77
TOMCAT	915	39	0.72

4.3 SATD に対する修正の類似度の算出

4.1 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 α 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**クラスタの修正の類似度**として求めた．

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 α 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，コントリビュータ数の関係を表 5 に示す．

全体として，SATD コメントの数が多いほど修正の類似度は高くなる．また，修正支援への有用性をはかるために，修正の類似度が 0.8 を超えるようなクラスタを有効なクラスタとして，全クラスタ数に対する有効なクラスタの割合を求めた．結果を表 6 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

表 6: 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

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．よって今回のプロジェクトでは Hadoop のデータが修正支援に活用できる．

Hadoop の同一クラスタ内で見られた SATD コメントと修正の類似性の例を図 5 に記す．同じ SATD コメントに対して類似する修正が施されていることが分かる．

```
@@ -82,9 +82,8 @@ public class CxfRsInvoker extends JAXRSInvoker {
            cxfRsConsumer.createUoW(camelExchange);
            // Now we don't set up the timeout value
            LOG.trace("Suspending continuation of exchangeId: {}", camelExchange.getExchangeId());
-           // TODO Support to set the timeout in case the Camel can't send the response back on time.
            // The continuation could be called before the suspend is called
-           continuation.suspend(0);
+            continuation.suspend(endpoint.getContinuationTimeout());
            cxfExchange.put(SUSPENED, Boolean.TRUE);
            cxfRsConsumer.getAsyncProcessor().process(camelExchange, new AsyncCallback() {
                public void done(boolean doneSync) {
```

CxfRsInvoker.java

```
@@ -86,9 +87,9 @@ public class CxfConsumer extends DefaultConsumer {

                // Now we don't set up the timeout value
                LOG.trace("Suspending continuation of exchangeId: {}", camelExchange.getExchangeId());
-               // TODO Support to set the timeout in case the Camel can't send the response back on time.
+
                // The continuation could be called before the suspend is called
-               continuation.suspend(0);
+                continuation.suspend(cxfEndpoint.getContinuationTimeout());

                // use the asynchronous API to process the exchange
                getAsyncProcessor().process(camelExchange, new AsyncCallback() {
```

CxfConsumer.java

図 5: 同一クラスタ内の SATD コメントに見られる修正の類似性の例

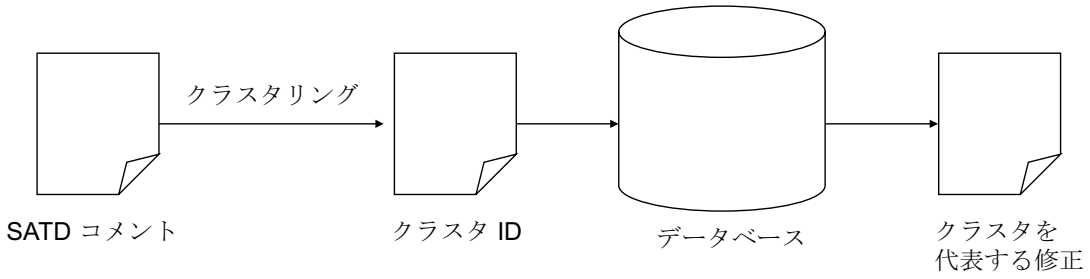


図 6: 修正支援の処理概要

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 と修正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とする．修正支援の処理概要を図 6 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．またデータベースはプロジェクトごとに構築している．

Hadoop のある SATD を入力として修正支援ツールを実行した例を図 7 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 7 から，類似した修正が施されていることがわかる．

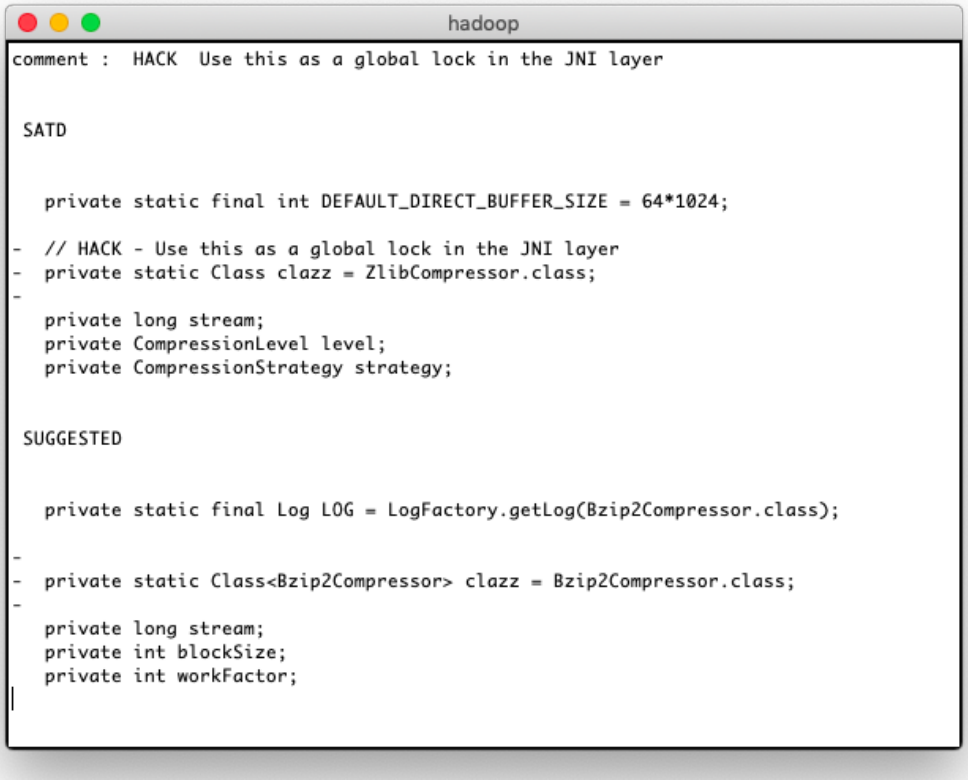


図 7: 修正支援ツールの実行例

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，SATD コメント数の少ないプロジェクトに関しては十分なクラスタリングの信頼度を得られなかったが，SATD コメント数の多いプロジェクトに関しては 0.40 - 0.87 のクラスタリングの信頼度を得ることができた．また，クラスタリングの信頼度にはコメントルールが大きく影響していることがわかった．

修正支援ツールの試作では，調査実験で得たクラスタを用いてデータベースを構築し，実際に Hadoop プロジェクトのある SATD コメントを入力として渡して動作を確認した．

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と考えられる．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．

謝辞

大阪大学情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授には，終始適切な助言と丁寧な指導を賜りました．多くの御助言を頂いた 井上 教授に心より深く感謝いたします．

大阪大学情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授には，研究に関する的確な助言を頂いただけでなく，学業に関する相談に対しても親身なアドバイスを頂きました．松下 誠 准教授に深く感謝いたします．

大阪大学情報科学研究科コンピュータサイエンス専攻 春名修介 特任教授には，研究および研究生活のみでなく，就職活動に関しても多くの助言を頂きました．春名修介 特任教授に深く感謝いたします．

大阪大学情報科学研究科コンピュータサイエンス専攻 神田 哲也 助教には研究について様々なご意見を賜りました．また，論文執筆においてもご援助とご指摘を賜りました．神田 助教の御指導及び御助言のおかげで本論文を完成させることができました．深く感謝申し上げます．

大阪大学情報科学研究科コンピュータサイエンス専攻事務職員である 軽部 瑞穂 氏には，いつも気さくに話しかけていただき，研究生活において多大なご気遣いをいただきました．私が研究室生活を楽しむことができたのは偏に軽部氏のおかげです．心より感謝いたします．

研究生活にて大いにお世話になりました，大阪大学情報科学研究科コンピュータサイエンス専攻 井上研究室の皆様に感謝いたします．

最後に学生生活及び研究生活を支えてくださった家族に感謝いたします．

参考文献

[1] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 315–326. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901742

[2] 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," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 47–52. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882373

[3] E. d. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, vol. 00, Oct. 2015, pp. 9–15. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MTD.2015.7332619

[4] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 609–613.

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

[6] M. A. Hall, "Correlation-based feature selection for machine learning," Tech. Rep., 1999.

[7] 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. [Online]. Available: https://doi.org/10.1007/s10664-017-9522-4

[8] 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, Mar. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2014.12.027

[9] E. D. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, vol. 00, Sept. 2018, pp. 238–248. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ICSME.2017.8

[10] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 350–359, 2004.

[11] A. McCallum and K. Nigam, "A comparison of event models for naive bayes text classification," in *IN AAAI-98 WORKSHOP ON LEARNING FOR TEXT CATEGORIZATION*. AAAI Press, 1998, pp. 41–48.

[12] A. Novikov, "annoviko/pyclustering: pyclustering 0.8.2 release," Nov. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1491324

[13] D. Pelleg and A. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *In Proceedings of the 17th International Conf. on Machine Learning*. Morgan Kaufmann, 2000, pp. 727–734.

[14] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 91–100. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.31

[15] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[16] F. Sebastiani, "Machine learning in automated text categorization," *ACM Comput. Surv.*, vol. 34, no. 1, pp. 1–47, Mar. 2002. [Online]. Available: http://doi.acm.org/10.1145/505282.505283

[17] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, Mar. 1981. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/7265238

[18] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 179–188.

[19] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal?: An in-depth perspective," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 526–536. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196423

[20] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '13. New York, NY, USA: ACM, 2013, pp. 42–47. [Online]. Available: http://doi.acm.org/10.1145/2460999.2461005

[21] 智. 一ノ瀬, 秀. 畑, and 健. 松本, "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," **情報処理学会関西支部支部大会講演論文集**, p. 4p, 2016. [Online]. Available: https://ci.nii.ac.jp/naid/40020965289/