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.

コード内に出現する識別子情報に基づくコミット分類

山内　健二†　楊　嘉晨†　堀田　圭佑†　肥後　芳樹†　楠本　真二†

† 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: †{y-kenji,jc-yang,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェアの開発工程において，これまでに記述されたソースコードの内容や，それにより実現されたソフトウェアの挙動について整理や検討が必要な場面がある．このような整理や検討は，バージョン管理システムやタスク管理システムを利用し，各コミットにおけるソースコードへの変更内容を，タスクという開発作業の単位で把握することで実現できる．このときコミットをタスク単位で分類するためには，タスクとコミットの対応関係の特定が必要となる．この特定を行うための手段として，コミットコメントやコミットを行った開発者の名前を利用することが考えられる．しかし，コミットコメントの内容の不備や開発者とコミットとに多対多の関係などがあることで，対応関係の特定が困難な場合がある．そこで本論文では，各コミットにおけるソースコードの差分に出現する識別子情報を利用してタスク単位でのコミット分類を行い，タスク単位での変更内容の把握を支援する手法を提案する．また提案手法の評価として，いくつかのオープンソース・ソフトウェアに対して，提案手法によるコミットの分類を行い，本手法がどのようなタスクを分類するのに有効であるかを考察した．
キーワード　バージョン管理システム，タスク管理システム，クラスタリング

Classification of Commits by Analyzing Identifiers in Source Code

Kenji YAMAUCHI†, Jiachen YANG†, Keisuke HOTTA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita-Shi, Osaka, 565–0871 Japan
E-mail: †{y-kenji,jc-yang,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　In software development, there is a need to understand changes of source code and implemented software behavior. To grasp those information, we can use a link between commits and tasks, with version control system and task management system. To detemine commit-to-task links, some studies have proposed approaches using commit comments and developers' names. However, it is difficult for those approaches to detect such links when there are obscure commmit comments or many-to-many relations between commits and developers' names. This paper proposes a technique to group commits into related tasks by using idetifier names in changes of souce code on each commit. To validate our approach, we applied it on several open-source software projects and found commits that were ideally grouped into a task. Furthermore, we discuss what type of tasks can be detected by our approach.
Key words　version control system, task control system, clustering

1. ま え が き

ソフトウェアの開発工程において，これまでに記述されたソースコードの内容や，それにより実現されたソフトウェアの挙動 (以下 **実装内容** と呼ぶ) について整理や検討が必要な場面がある．このような整理や検証は，機能の追加や変更など，開発工程で設定される目標ごとに実装内容を把握することで，効率的に行うことができる．本論文においては，このような開発工程での目標について，それを達成する作業群を，それぞれタスクと呼ぶ．タスク単位での実装内容の把握が有用な場面として，コードレビュー，リリースノートの作成，タスク管理システムとの同期などが挙げられる．

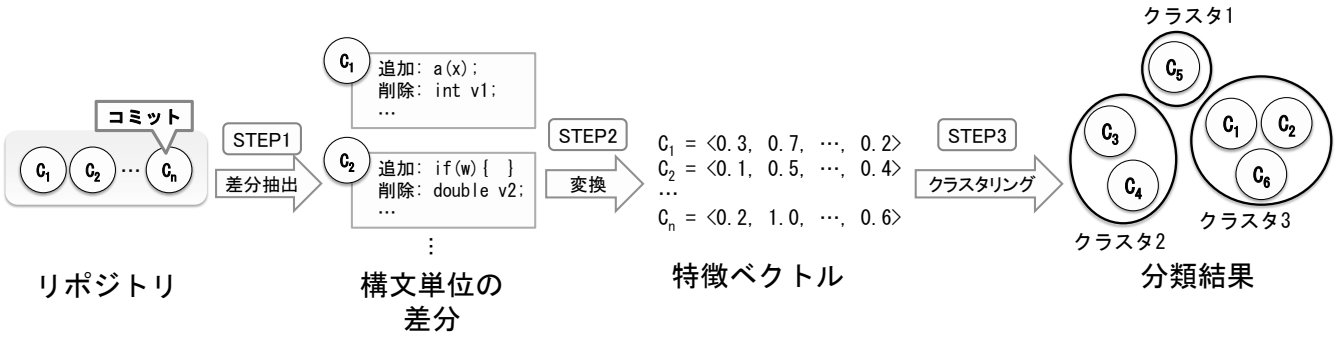

図 1: コミット分類の流れ

このようなタスク単位での実装内容の把握は，バージョン管理システム (Version Control System，VCS) を利用することで，より効率的に行うことができる．VCS からは，ソースファイルに対して行われた変更の内容を，コミットという単位で取得できる．それぞれのコミットがどのタスクに関係しているのかを特定し，コミットをタスクごとに分類することで，タスク単位での実装内容の把握が容易となる．これまでに，コミットコメントやコミットを行った開発者の名前を用いて，コミットとタスクの対応関係を特定する手法が提案されている [1,2]．しかし，これらの手法には，適切なコミットコメントが記述されていない場合や，開発者とタスクが多対多で対応している場合に，コミットとタスクを適切に関連付けることができないという課題がある．

コミットコメントや開発者の名前を用いる手法以外のコミットの分類手法には，ソースコードの差分情報を利用する方法がある．ソースコードの差分を用いることで，コミットコメントの記述が不十分な場合や，開発者とタスクが多対多で対応している場合でも，コミットの分類を行うことができる．これまでにソースコードの差分を用いたコミットの分類手法が提案されている [3] が，それらはタスク単位でのコミット分類を対象としていない．また，ソースコード中の構文の構造のみを利用しているため，分類の結果にそのソースコードが書かれた目的などを反映できないという課題がある．

そこで，本論文では，ソースコードの差分に含まれる識別子情報に基づいてコミットの分類を行う手法を提案する．識別子情報を用いることで，抽象構文木などの構文情報からは得ることのできない，そのソースコードが書かれた目的などの情報を得ることができる．それぞれのソースコードが書かれた目的は実装内容と強く関連するため，識別子情報を用いてコミットの分類を行うことで，タスク単位でのコミット分類を実現することができると期待できる．オープンソース・ソフトウェア (Open Source Software，OSS) に対して提案手法を適用し，提案手法がタスク単位でのコミット分類を行うことができているかを評価した．その結果，コミットコメントや開発者名を用いた場合は適切にコミットの分類ができない事例に対し，適切にコミットの分類を行うことができたことを確認した．

2. 研究動機

2.1 タスク単位でのコミット分類

タスク単位での実装内容の把握を支援する手段の 1 つとして，タスク単位でのコミット分類が挙げられる．コミットをタスク単位で分類することで，それぞれのタスクを実現するためにソースコードに対してどのような修正が加えられたのかを把握することができる．

タスク単位でコミットを分類する場合，分類の基準の候補としてコミットコメント，コミットを行った人物の名前，ソースコードの差分情報という 3 つの情報が挙げられる．タスク管理システムを用いて開発されているプロジェクトの場合，各タスクには一意なタスク ID が割り当てられている．従って，コミットを行う際に，コミットコメントに該当するタスク ID を記述することで，タスクとコミットの関連を把握することが可能になる [1]．そのため，全てのコミットについて，コミットコメントでタスク ID を記述していれば，そこからコミットをタスク単位で分類できる．しかしながら，コミットコメントに，タスク ID の記述が行われていない等の不備がある場合，コミットとタスクの関連の把握が困難になる．

また，1 つのタスクに対して，1 人の開発者のみが割り当てられているならば，コミットを行った人物の名前を分類基準として利用できる．しかし，開発者とタスクが多対多で割り当てられている場合には，開発者名を用いてタスク単位でコミットを分類することは難しい．

ソースコードの差分情報を用いる方法は，各コミットでソースコードに対してどのような修正が加えられたのか，という情報に基づき，コミットの分類を行う．ソースコードの差分情報は各コミットに必ず含まれる情報であるため，コミットコメントや開発者名を用いる手法を適用できない状況でも，コミットの分類を実現することができる．従って本論文では，ソースコードの差分情報を用いてコミットの分類を行う手法を提案する．

2.2 関連研究

これまでに，コミットの分類を行う手法がいくつか提案されている．Hindle ら，及び Hattori と Lanza はそれぞれコミットコメントに対して自然言語処理を適用してコミットの分類を行うことで，各コミットがソフトウェアの挙動に与える影響を

調査している [4,5]．これらの手法はコミットコメントを用いる手法であるため，コミットコメントに不備がある場合，コミットの分類を行うことができない．

また，Dragan らは，ステレオタイプと呼ばれる典型的な設計種別を利用してコミットの分類を行っている [3]．しかしこの手法では，設計上の影響があった修正にしか対処できず，バグ修正，リファクタリングなどのタスクに対する分類に有効でない．

そこで本論文では，ソースコードの差分情報に含まれる識別子名を用いてコミットの分類を行う手法を提案する．識別子名はそのソースコードが行う処理の内容と強く関連しており，かつ構文情報からは得ることのできない情報が含まれている．従って，識別子情報を用いることで，既存手法では対応できない事例に対しても，タスク単位でのコミットの分類が実現できると期待できる．

3. 提案手法

提案手法では，VCS のリポジトリを入力とし，そこに蓄積されたコミットをタスク単位で分類した結果を出力する．この時，分類はコミット前後の，すなわちリビジョン間におけるソースコードの差分に含まれる識別子の出現回数に基づいて行われる．

提案手法の処理の流れを図 1 に示す．この図に示すように，提案手法によるコミットの分類は以下の 3 ステップからなる．

STEP1　リポジトリから抽出した各コミットでのソースコードに対する変更内容を，構文として意味のある単位 (以後 **構文単位の差分** と呼ぶ) で解釈した上で取得する．

STEP2　STEP1 で得られた構文単位の差分それぞれについて，含まれる識別子を全て取得し，それぞれ英単語として意味のある単位に分割した上で，各単語の出現回数を元にした **特徴ベクトル** というベクトルの生成を行う．

STEP3　生成した特徴ベクトルをコミットの特徴量とみなしてクラスタリングのアルゴリズムを適用し，コミットの分類を行う．

以降の小節で，各手順についての詳細を述べる．

3.1 (STEP1) 構文単位の差分の抽出

本手法ではまず，各コミットについて，構文単位の差分を抽出する．

各コミットにおいて，そのコミットで変更されたソースファイルを特定し，それぞれのソースファイルについて，Change Distilling [6] と呼ばれる手法を用いることで，リビジョン間での構文単位の差分を求める．これらは，文や式としてまとまりのある形式で抜き出される．

図 3 に示す例の場合，下線部の部分について字句的な変更があったとして，その変更を含む，“`private int logCounts`”，“`!IS_LOGGED`”，“`log(messages)`” という 3 つの構文単位の差分が得られる．

3.2 (STEP2) 識別子名の抽出と特徴ベクトルの生成

次に，それら構文単位の差分に含まれる識別子を抽出する．この時，その差分に含まれる識別子と合わせて，その差分を含むメソッド名およびクラス名の抽出も行う．変更の対象がフィールドかメソッドの宣言ならば，それらが含まれるクラスの識別子のみ抽出する．

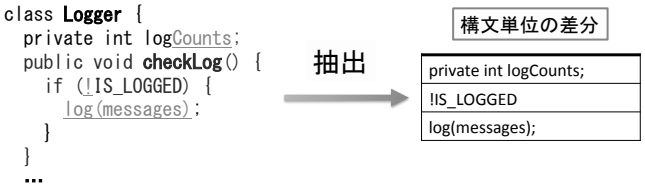

図 3: ソースコード変更と構文単位の差分の例

図 3 に示す例であれば，3.1 節で説明した 3 つの構文単位の差分 “`private int logCounts`”，“`!IS_LOGGED`”，“`log(messages)`” について，それぞれ識別子が抽出される．例えば，“`private int logCounts`” について考えると，差分自体に含まれる識別子 “logCounts” と，差分が含まれるクラスの名前である “Logger” が抽出される．同様に “`!IS_LOGGED`”，“`log(messages)`” についても，それぞれ考え，最終的に “logCounts”，“Logger”，“IS_LOGGED”，“Logger”，“checkLog”，“log”，“messages”，“Logger”，“checkLog” という識別子群が得られる．

次に，特徴ベクトルを生成する．このステップは以下の 4 つのサブステップからなる．

STEP-2A　識別子を英単語として意味のある単位で分割する．

STEP-2B　STEP-3A で分割されたあとの単語をそれぞれ見出し語化する．見出し語化した後の単語を **特徴語** と呼ぶ．

STEP-2C　STEP-3C で得られた特徴語それぞれについて，出現回数を集計する．

STEP-2D　STEP-3D での集計結果を，特徴ベクトルへ変換する．

以降，各サブステップごとに説明を行う．

3.2.1 (STEP-2A) 単語分割

STEP1 で抽出した識別子を，英単語として意味のある単位に分割する．この時，識別子それぞれがキャメルケースあるいはスネークケースで記述されていることを前提とする．ここで，キャメルケースとは “getYourName” のような，単語の区切りとなる英字を大文字に，他を小文字とする記述法で，スネークケースは “get_your_name” のような，単語の区切りにアンダースコア (_) を用いる記述法である．また，各分割後の単語については，後述する特徴語の出現回数の集計処理を簡便にするため小文字化する．例えば，“convertedName” と “MAX_COUNTS” という識別子があった場合，それぞれ，“converted name” と “max counts” という形で分割および小文字化される．

3.2.2 (STEP-2B) 単語の見出し語化

次に，分割後のそれぞれの単語について，見出し語化を行う．見出し語化とは，動詞の活用や名詞の格変化で語尾の変化がある単語について，基本形に変形することである．先ほどの例の場合，“converted” と “counts” がそれぞれ “convert” と “count” になる．以降，分割並びに見出し語化までの処理が行われた単語を特徴語と呼ぶ．

図 2 に，ここまでの流れの具体例を示す．この例では，

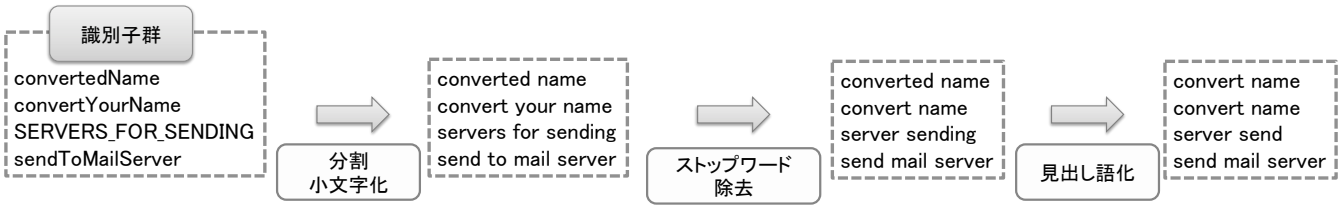

図 2: 識別子の英単語の見出し語化までの過程

“convertedName”，“convertYourName”，“SERVERS_FOR_SENDING”，“sendToMailServer” という 4 つの識別子群が取得できたとして，特徴語群を抽出するまでを示している．得られる特徴語群は，“convert”，“name”，“your”，“server”，“for”，“send”，“to”，“mail” である．

3.2.3 (STEP-2C) 特徴語の出現回数集計

STEP-3B で特定した特徴語群について，それぞれの出現回数を集計する．この時，ストップワードについては集計を行わない．ストップワードとは，自然言語処理における用語で，前置詞や助動詞など，どのような文書でも現れやすいために文書の特徴を示しにくい語群を意味する．これらの単語は，文書分類において，分類の精度へ悪影響を与えるおそれがある．よって，本提案手法では，ストップワードを特徴語群から排除する．

図 2 の例であれば，特徴語群に関して，“convert” が 2 回，“name” が 2 回，“server” が 2 回，“send” が 2 回，“mail” が 1 回という集計結果を得られることになる．

3.2.4 (STEP-2D) 集計結果の特徴ベクトルへの変換

最後に，集計結果を，特徴ベクトルへ変換する．

それぞれのコミットを，各特徴語の出現回数を各要素とするベクトル (以下 **Bag-Of-Words(BOW) ベクトル** と呼ぶ) へと変換し，これを正規化して特徴ベクトルとする．この時，BOW ベクトルの次元数は，全てのコミットで出現した特徴語の数と等しくなる．そのコミットで出現しない単語に対応する要素の値は 0 となる．例えば，コミット A で特徴語 “a”，“b” が 1 回ずつ，コミット B で特徴語 “b”，“c” が 2 回ずつ登場したとして，それぞれのコミットの BOW ベクトルを考える．この時，各要素がそれぞれ “a”，“b”，“c” の出現回数である BOW ベクトルが，コミット A，B でそれぞれ (1,1,0)，(0,2,2) として得られる．

3.3 (STEP3) クラスタリングアルゴリズムの適用

最後に，3.2.4 節 で生成した特徴ベクトルを用いて，クラスタリングを行う．アルゴリズムには，以下に述べる 2 つの条件を考慮して，Repeated Bisection [7] を選択した．

- 分類後のクラスタ数の指定が不要である．本手法では事前にタスク数，すなわち分類後のクラスタ数を特定できない．そのため，事前に分類後のクラスタ数を指定する必要があるクラスタリングは利用できない．
- スケーラビリティが高い．開発が長期にわたって行われているプロジェクトは膨大なコミット数をリポジトリに蓄積している場合がある．したがって，本手法では，アルゴリズムにスケーラビリティが要求される．

分類後のクラスタ数を事前に指定する必要のないクラスタリングのアルゴリズムには，Repeated Bisection 以外にもいくつか存在する．しかし，それらは計算量が大きく，膨大なデータを処理するには大きな計算時間を要する．Repeated Bisection は他のクラスタリング数の指定が不要なクラスタリングアルゴリズムに比べて高速である [7] ことから 1 つ目の条件と 2 つ目の条件を同時に満たす．

4. 評価実験

本節では 3. 節 で述べた手法に対する評価実験について記述する．いくつかのリポジトリに対して提案手法を適用してコミット分類を行い，タスクに対応したクラスタが存在することを確認する．

4.1 提案手法による分類結果の確認

この実験では，4 つの OSS のリポジトリに対して提案手法を適用し，生成されるクラスタ数を確認する．

Java で記述され，Git で管理されている OSS のうち，これまでに行われたコミット数が多い (5,000 以上) ものから 4 つを選択して実験対象とし，提案手法によるコミットの分類を行った．

表 1 に実験対象のプロジェクトのそれぞれについて，分類結果におけるクラスタ数とともに分類結果を示す．また，図 4 に，分類結果における各クラスタに属しているコミット数の分布を箱ひげ図で示す．図 4 では，各プロジェクトについて，X 軸上方に各クラスタに属しているコミット数の中央値を示す．どのプロジェクトでも 6 か 7 となっている．

a) 望ましい分類結果の例

まず，分類結果がタスク単位でのコミットの分類を適切に行うことができたと考えられる例を示す．表 2 に，Lucene/Solr についてコミット群を分類した結果におけるあるクラスタについて，そのクラスタを構成する 4 つのコミットの情報を示す．

ここで，コミット ID が e4a64f5，5adc910，c5a985e のコミッ

表 1: 分類結果

プロジェクト名	コミット数	分類対象となったコミット数	クラスタ数
Lucene/Solr(注1)	11,159	7,341	919
Jenkins CI(注2)	17,640	8,452	1,102
WildFly(注3)	14,280	10,247	1,204
JRuby(注4)	20,531	13,142	1,487

(注1)：https://lucene.apache.org/solr/
(注2)：http://jenkins-ci.org/
(注3)：http://www.wildfly.org/
(注4)：http://jruby.org/

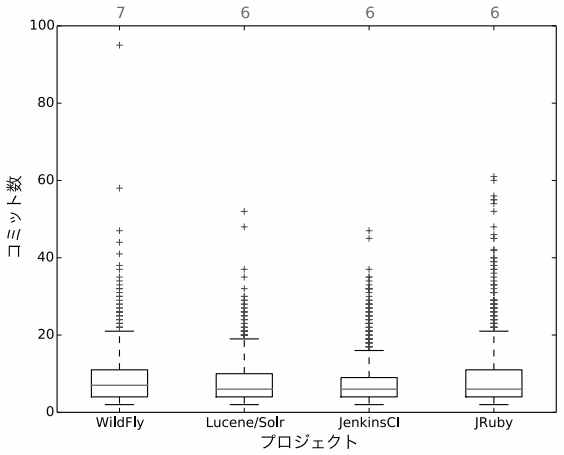

図 4: 各クラスタのコミット数分布

トについては，それぞれのコミットコメントに記述されたタスク ID から，Lucene/Solr の開発において用いられているタスク管理システムで，SOLR-4275 (注5) というタスク ID を割り当てられたタスクと関連しているコミットであるとわかる．

また，コミット ID が a3e95d0 のコミットは差分の内容を確認すると，コミットコメントでも言及されているように TokenTokenizer クラスへ，オフセットの利用を目的として処理を記述したことがわかる．さらに，その記述で生じたバグに対して，コミット ID が c5a985e のコミットで修正を行ったことが分かる．

よって，TokenTokenizer クラスのオフセット機能の実装というタスクを考えた時，a3e95d0 のコミットはコミットコメントでタスク ID の記述がないが，他 3 つのコミット e4a64f5, 5adc910，c5a985e と同じタスクの作業であるといえる．

まとめると，このクラスタは，1 つのタスクに関連したコミットのみで構成されていると考えられる．

b） 望ましくない分類がなされた例

次に，コミットの分類が適切に行われなかった例を示す．表 3 に，Jenkins CI に提案手法を適用して得られたあるクラスタで，そのクラスタを構成する 3 つのコミットの情報を示す．

まず，IOUtils クラスに対する変更を行っている ID が b3553d6 と 7d0bac1 のコミットについて述べる．IOUtils クラスはソフトウェア内での入出力処理に関するユーティリティクラスであるため，機能が独立した静的なメソッドのみで構成されている．そのため，それぞれのメソッドの実装は異なるタスクといえる．コミット ID が b3553d6 と 7d0bac1 のコミットで行われた変更内容を見ると，それぞれのコミットは異なるメソッドについての実装である．そのため，これらは異なるタスクについての実装であるといえる．

さらに，ID が a545a39 のコミットで行われた変更内容を確認すると，Launcher クラスの内部クラスについて，アクセスレベルの変更を行っている．これは，コミットコメントからも分かるように，該当の内部クラスについてシリアライズを行うためのものであるため，入出力関係の処理を実装している同じクラスタに分類された他の 2 つのコミットとは関連がない．

このように，表 3 に挙げたクラスタに含まれているコミットは，それぞれ異なる実装内容を実現している．そのため，このクラスタは，1 つのタスクのみに関連している，すなわちタスク単位として望ましく分類されているクラスタとはいえない．

表 2: 望ましい分類結果の例 (Lucene/Solr)

コミット ID	修正されたソースファイル	コミットコメントに記述されたタスク ID
e4a64f5	TestTrie.java	SOLR-4275
5adc910	TestTrie.java	SOLR-4275
c5a985e	TrieTokenizerFactory.java	SOLR-4275
a3e95d0	TrieTokenizerFactory.java	記述なし

(注5)：http://issues.apache.org/jira/browse/SOLR-4275

5. 考　　察

変更が多岐にわたるタスクというものを考え， 4. 節 で得られた結果から，このようなタスクの分類に対して手法が有効であるかを考察する．

あるコミット群について，各コミットでの差分を個別に把握するだけでは，それぞれのコミットで実現された実装内容の関連を把握しにくいが，実際には 1 つのタスクを実現しているものがある．このようなタスクのうちで，変更が多岐にわたるタスクというものを考え，提案手法でこれらのタスクがどの程度検出可能かを評価する．

まず，変更が多岐にわたるタスクの定義を述べる．実装上の問題から，単一のファイルのみに対する編集で完結しないタスクが存在する場合がある．このようなタスクが複数コミットで実現され，しかも各コミットで変更されるソースファイルが異なっている場合，主として変更されているファイルを一意に決定しにくいために，実装内容の把握が困難であると考えられる．

そこで，クラスタ内のコミットで編集対象となったファイルすべてについて，編集対象となったコミット数を考える．この

表 3: 望ましくない分類の例 (Jenkins CI)

コミット ID	修正されたソースコード	コミットコメント
b3553d6	IOUtils.java	added a convenience method
7d0bac1	IOUtils.java	doh
a545a39	Launcher.java	for serialization work these interfaces need to be public

表 4: 各プロジェクトでの，多岐にわたるクラスタの割合

プロジェクト名	クラスタ数	変更が多岐にわたるクラスタ数	割合 (パーセント)
Lucene/Solr	919	321	34.9%
Jenkins CI	1,102	234	21.2%
WildFly	1,204	457	38.0%
JRuby	1,487	264	24.0%

コミット数が占めるクラスタ内の全コミット数に対する割合 (**編集割合** と呼ぶ) が，0.5 を超えるファイルが存在しないクラスタを，変更が多岐にわたるタスクに対応するクラスタとみなす．

変更が多岐にわたるタスクの例として，4. 節 で望ましい分類結果の例として挙げたクラスタが対応するタスクを考える．このタスクでは，4 つのコミットにより，TrieTokenizerFactory.java と TestTrie.java にまたがって編集が行われている．これらソースファイルの編集割合はそれぞれ 0.5 であるから，このタスクは変更が多岐にわたるタスクである．

最後に，編集割合を利用し，変更が多岐にわたるクラスタを特定し，これらが各実験対象のプロジェクトについてどの程度存在しているかを測定する．結果は表 4 である．表から，実験対象となったプロジェクトについては，どれも 20%以上変更の多岐にわたるクラスタ，そしてそれらに対応するタスクが存在することが分かる．

以上から，変更が多岐にわたるタスクが，一定の割合でプロジェクト内で存在することがいえる．そして，提案手法はこれらのタスクを自動で検出できるという点で有用性があるといえる．

6. 結果の妥当性

本論文における手法や考察の妥当性について，以下で説明する点に留意する必要がある．

6.1 提案手法の妥当性

まず，提案手法はソースコード中に出現する識別子が実装内容を反映していることを仮定している．このため，識別子に対する命名に不備がある場合，望ましい分類が得られない．

また，1 つのコミットでは 1 つのタスクの内容しか実装していないことも仮定している．そのため，複数の独立したタスクが 1 つのコミットに混在しているコミットが多い場合は，本手法で正しく分類できない場合がある．

6.2 考察の妥当性

今回，分散の変更の分散が激しいタスクというものを考え，これがプロジェクト内で行われたタスク全体に占める割合を考えて手法の有効性を考察した．しかし，この考察において，変更の分散が激しいタスクとして検出されたクラスタそれぞれが，実際にどの程度 1 つのタスクとしてみなせるかを評価できていない．

そのため，検出された変更の分散が激しいクラスタそれぞれが，どの程度タスク単位の分類として正しいかを定量的に評価する必要がある．

7. あ と が き

本論文ではソフトウェア開発において，タスク単位での実装内容の把握が有用であることを踏まえ，これを支援するために，バージョン管理システムのリポジトリに蓄積されたコミットをタスク単位で分類する手法を提案し，実際にいくつかのオープンソース・ソフトウェアを対象として分類を行った．

その結果，タスク単位での分類として望ましい結果が存在することを確認した．また，提案手法がどのようなタスクの分類に有効であるかを，変更が多岐にわたるタスクというものを定義して考察した．

今後の課題は，以下の通りである．

- 分類精度向上のための，識別子からのより正確な特徴語抽出
- 複数のタスクに関連したコミットに対応した分類

謝辞 本論文は，日本学術振興会科学研究費補助金基盤研究 (S)(課題番号：25220003)，挑戦的萌芽研究 (課題番号：24650011)，及び文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の支援を受けて行われた．

文　献

[1] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli. A machine learning approach for text categorization of fixing-issue commits on cvs. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 6:1–6:10, 2010.

[2] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 97–106.

[3] N. Dragan, M.L. Collard, M. Hammad, and J.I. Maletic. Using stereotypes to help characterize commits. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, pp. 520–523, 2011.

[4] A. Hindle, D.M. German, M.W. Godfrey, and R.C. Holt. Automatic classication of large changes into maintenance categories. In *Proceedings of the 2009 IEEE 17th International Conference on Program Comprehension*, pp. 30–39, 2009.

[5] L.P. Hattori and M. Lanza. On the nature of commits. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering 2008.*, pp. 63–71, 2008.

[6] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, pp. 725–743, 2007.

[7] G. Karypis. CLUTO - a clustering toolkit. Technical report, Digitical Technology Center, 2003.