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.

コード内コンテキストを用いた履歴情報取得のためのGitクライアント拡張

佐々木美和† 小倉 直徒†† 柗本 真佑†† 楠本 真二††

† 大阪大学基礎工学部情報科学科
†† 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: †{m-sasaki,n-ogura,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし 一般的な版管理システムでは，ソースコードはただのテキスト情報として扱われており，ソースコードの構文情報に基づいた情報の取得操作はサポートされていない．そこで本研究では，ソースコードの構文情報を一種の文脈（コンテキスト）とみなし，コンテキストを用いた情報の絞り込みの実現を目的として，Git クライアントを拡張する．これによって開発者は，関心のない部分が削られた情報だけを取得することが可能になる．提案手法の有用性を評価するために，プロトタイプ MJgit を実装し，評価実験を行う．

キーワード 版管理システム，履歴情報の絞り込み，コンテキスト，Git，抽象構文木

1. まえがき

SVN や Git をはじめとする版管理システムは，多くのソフトウェア開発プロジェクトに導入されるている [1]．特に近年のソフトウェア開発は大規模化と遠隔分散化が進んでおり，版管理システムによる成果物，及び履歴情報の管理は必要不可欠となっている．

しかしながら，ほとんどの版管理システムでは，管理対象となるソースコードは単なるテキストファイルとして扱われており [2]，ソースコードの構文情報を利用した情報の取得操作は支援されていない．例えば，自身の編集したソースコードの中から，コメント部分のみの差分を確認するといった処理や，逆にコメントを省いたプログラムの実行ステートメント部分のみを確認することは不可能である．Linux コマンドにおける Diff や Grep 等のコマンドやオプションはサポートされているものの，基本的にはテキストデータに対する行単位での操作のみというのが現状である．

多くのプログラミング言語では，ソースコード中に様々な情報（実行ステートメントやコメント，著作権情報，注釈など）を混在して記述するという文化が浸透している．この文化は，ある単一のソースコードのみに注力する際には問題にはならないが，バグ箇所の履歴追跡といった履歴情報に横断的，かつ特定の情報のみに（実行ステートメントのみなど）横断的な操作を行いたい際には，その他の情報（コメントや著作権情報など）がノイズとなり得る．このコード中の様々な情報を一種の文脈（以降，コンテキスト）と見なし，コンテキストに応じた履歴情報の絞り込み操作が可能となれば，開発者に対する手助けとなると考えられる．

さらに，このソースコード中のコンテキストを考慮した履歴操作は，開発者のプログラミング時のみならず，開発履歴を対象としたマイニング研究 [3] にも広く活用できる可能性がある．この研究分野は MSR（Mining Software Repository）とも呼ばれており，変更履歴を用いたバグの予測 [4] や，バグとコメント行数の関係の調査 [5]，コミットログからの開発者の感情の分析 [6] など多岐に渡って実施されている．これらの研究ではその目的に応じて，実行ステートントが変更されていないコミットを除外する，コメントを含むコミットのみを絞り込む，といったマイニングの前処理が行われることが多い．版管理システムそのものがコンテキストを考慮した操作を実現することで，これら前処理に必要な労力を削減できると考えられる．

本研究の目的は，版管理システム上におけるコンテキストを利用した情報の絞り込み操作の実現である．そのためのアプローチとして，構文に基づくコンテキストの整理および，プロトタイプとなる拡張 Git クライアント，MJgit の開発を行う．提案手法の評価実験として，実際のソフトウェア開発リポジトリを対象に，MJgit 利用時の実行速度，及び履歴情報の削減量について確かめる．

2. 研究動機

本章では，あるソースコードに対するバグの修正というシナリオに基づいて，既存の版管理システムにおける問題点，及び研究の動機について説明する．本研究のアイデアは，特定のプログラミング言語や特定の版管理システムに依存しない一般的なものである．ここでは具体的な例示のために，プログラミング言語を Java，版管理システムを Git として説明を行う．

まず，以下のようなソースコード Example.java が与えられたとする．

```
1  /*  Licensed under the MIT License */
2  public class Example {
3    public void Foo() {
4      ...
5      int case = getCase();
6      switch (case) {
7        case 1:
8        this.doSomething();
9        break;
10       case 2:
11       this.doSomething();
12       break;
13       ...
14       // Handle an error case
15       case -1:
16       this.handleError();
17       default:
18       this.doDefault();
19     }
20     ...
```

Example.java には switch 文の中で break 命令を忘れるという典型的なバグが含まれている．具体的には，変数 case が値-1 を取る際に default まで処理されてしまう．これを修正するには 16 行目と 17 行目の間に break 命令を加える必要がある．

このバグを修正することを考える．まずバグ修正者はバグの箇所（16 行目）を特定し，break 命令を加えて当該バグを修正する．加え，このバグ箇所の処理はその関数名（handleError()）からエラーの対処のための追加処理であることが読み取れる．よって，この switch 処理全体を記述した開発者と，バグ箇所の処理を追加した開発者が異なる可能性がある．バグに関する一般的な事実として，バグの箇所はパレートの法則に従っており 2 割のソースコードに 8 割のバグが含まれる [7], [8]，編集に関わる開発者の数が増えるほどバグが発生しやすい [9], [10]，などが知られている．よってバグ修正作業だけでなく，当該バグ付近の変更を追跡してその内容をレビューするべきであるといえる．

Git ではソースコード全ての行の編集者を調べる blame コマンドが用意されており，以下のような結果を得ることができる．

```
$ git blame Example.java
f098 (shinsuke  1) /* Licensed under ... */
ac5f (sasaki    2) public class Example {
ac5f (sasaki    3)   public void Foo() {
...
ac5f (sasaki    6)     switch (case) {
ac5f (sasaki    7)       case 1:
ac5f (sasaki    8)       this.doSomething();
ac5f (sasaki    9)       break;
...
b8b5 (ogura    14)       // Handle an error
b8b5 (ogura    15)       case -1:
b8b5 (ogura    16)       this.handleError();
ac5f (sasaki   17)       default:
ac5f (sasaki   18)       this.doDefault();
...
```

この結果より，バグ箇所を編集した開発者（ogura）が特定できる．この例ではソースコード全体を単純化しており，全ての出力結果を確認することは容易である．しかしながら，実際のソースコードにはコメントや著作権情報など様々な情報が含まれており，これらの情報がノイズとなり得る．ここでこの blame コマンドに対して，「実行ステートメントに絞り込む」や，より具体的な「switch 文に絞り込む」というコンテキストを指定した操作が可能であれば，目的とする情報へのアクセスが容易になる．switch 文というコンテキストで絞り込む際の振る舞いの一案を以下に示す．

```
$ git blame Example.java --context=switch-sta
    tement
ac5f (sasaki  6)     switch (case) {
ac5f (sasaki  7)       case 1:
ac5f (sasaki  8)       this.doSomething();
ac5f (sasaki  9)       break;
...
```

さらに，バグを埋め込んだコミットの詳細を確認する際は，git show コマンドにより変更内容を取得できる．blame の際と同様に，当該コミットでのあらゆる変更内容が取得される．ここで興味がある情報は handleError 関数の処理内容である．よって，先の例と同様に「実行ステートメントに絞り込む」や「handleError 関数に絞り込む」といったコンテキスト指定が有効である．

Git コマンドには，行数を指定するというオプションも存在しており，現状でもこれらの例と同等の操作は可能である．また grep オプションにより，特定のキーワードに絞り込むことも可能である．さらには，Git の出力結果に対して Linux のコマンド群を組み合わせれば，より複雑な情報取得は可能である．しかしながら，これらの行数指定や文字列検索はテキストデータに対する汎用的な処理である一方で，提案するコンテキスト操作はソースコードに特化した意味単位での処理と見なすことができる．単なるテキストに対する処理に加え，ソースコード特化の意味単位の処理を支援することで，プログラミングの手助けになると考えられる．

3. 提案手法

本章では，2 章と同様に，プログラミング言語を Java，版管理システムを Git と限定して検討を行う．

3.1 構文情報を用いたコンテキスト

本研究では，コンテキストという言葉を，「ある対象の周辺から得られる文脈に関する情報」の意味で用いる．版管理システムにおけるコンテキストとしては，ソースコードの構文情報から取得可能な情報や，開発履歴から取得可能な情報な

どが考えられる．本章では，前者の構文情報に基づくコンテキストを「構文コンテキスト」と呼び，これをプログラムの振る舞いに影響を与えるかという観点から，大きく以下 3 種類に分類する．

- **実行ステートメント**: プログラムの振る舞いが記述された記述箇所のこと．さらに以下に示すような詳細な分類が可能である．
  - **if ステートメント**: if 文の記述箇所

```
if (n > 0) {
```

  - **switch ステートメント**: switch 文の記述箇所

```
switch (case) {
```

  - **関数呼び出しステートメント**: 関数呼び出しの記述箇所

```
public void handleError() {
```

- **コメント**: プログラムの振る舞いに寄与しないコメントの記述箇所のこと．
  - **ラインコメント**: 1 行コメントの記述箇所

```
// Handling an error case
```

  - **ブロックコメント**: 複数行に渡るコメントの記述箇所

```
/* Licensed under the MIT License
```

  - **Javadoc**: メソッドの API 仕様の記述箇所

```
/**
  * Error handler
  * @param
  ...
```

- **アノテーション**: 注釈に関する記述箇所．Java ではメソッドやクラスのメタ情報の記載に用いられる．プログラムの振る舞いに直接影響を与えないが，コンパイラがこの情報を読み取り開発者に警告を出す等の形で用いられる．

```
@Override
```

3.2 従来の版管理システムとの違い

提案手法と従来の版管理システムとの比較を図 1 と図 2 に示す．これらの図では，あるリポジトリに c1 から cN までコミットされた履歴情報が保存してある状態を表している．ここで，ユーザーがコミット c2 とコミット c3 の差分をコマンド「git diff c2 c3」によって要求した時，従来の Git では，コミット c2 とコミット c3 の差分が，追加分も削除分も全てそのまま渡される．しかし，提案手法では，コマンド「git diff c2 c3」にコンテキストを指定する命令を追加することで，情報が絞り込まれた差分が使用者に渡される（図 2）．

3.3 拡張対象コマンド

版管理システムには多くのコマンドが用意されているが，本研究の提案手法では，履歴情報の取得における絞り込みを目的としている．よって，拡張するコマンドは全てのコマンド

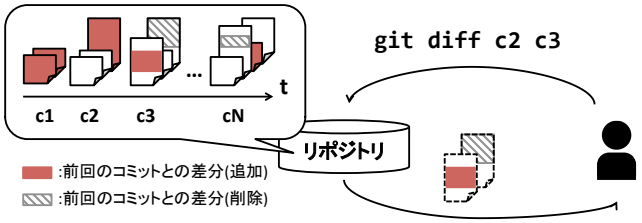

図 1　従来の Git の動作

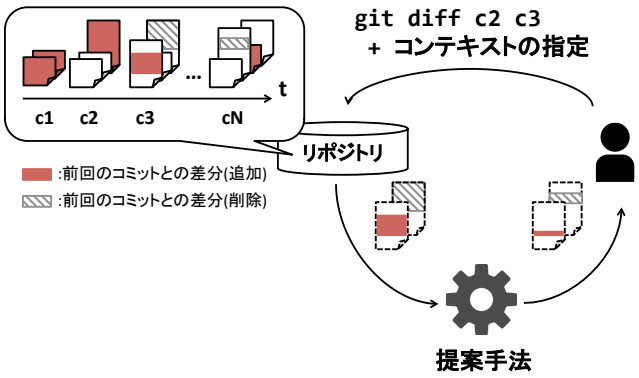

図 2　提案手法の動作

ではなく，履歴情報取得に関わるコマンド，つまりソースコードの情報を取得するコマンドのみを対象とする．Git の場合では，以下のようなコマンドが拡張対象となる．

- **diff**: 手元のソースコードと最新コミットとの差分や，2 つのコミット間の差分などを表示する
- **show**: コミットの詳細を見る
- **log**: コミットのログを見る
- **blame**: ファイル内の各行の最終更新の情報を見る
- **grep**: ファイルから文字列を検索する

3.4 提案手法の動作

提案手法の動作について説明する．以上で述べた提案手法について，どのような動作で実現するか，二通りの方法を提案する．事前処理型と適宜処理型である，事前処理型を図 3 に，適宜処理型を図 4 に，それぞれのメリット・デメリットをまとめた表を表 1 に示す．

図 3 の事前処理型では，コミット時にリポジトリにファイルを保存する前にコンテキストによる絞り込みを済ましておき，その状態でリポジトリに保存する．そして，コマンド実行時にコンテキストが指定されれば，すでに絞り込まれている情報から該当するものを選択して使用者に渡す．

事前処理型のメリットは，コミット時に絞り込みを済ましておくことで，今後一切絞り込み処理を行わなくてよくなり，コンテキスト絞り込み時の速度が速くなることである．デメリットは，コミットの実行速度が遅くなることと，コミット時に処理を行う必要があるので，既にコミットが実行されてしまっている既存のリポジトリに適用できないこと，また，実装時の拡張対象コマンドとして，コンテキストを絞り込めるコマンド（diff や show など）に加え，commit コマンドも拡張す

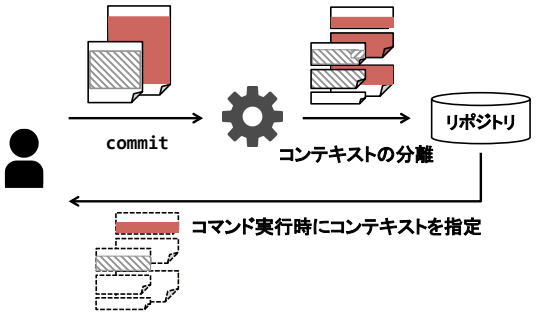

図 3　事前処理型

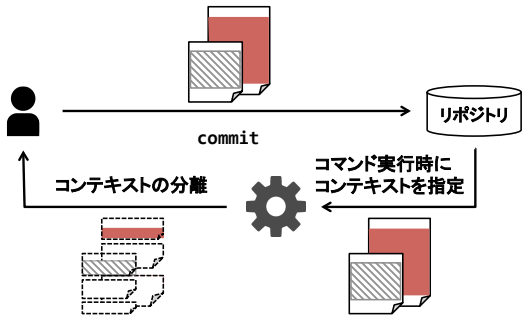

図 4　適宜処理型

る必要があることである．

図 4 の適宜処理型では，コミット実行時には従来のシステム通りにリポジトリにファイルを保存し，コマンド実行時にコンテキストが指定されればコンテキストの絞り込みを行う．

適宜処理型のメリットとしては，コミット時の速度低下が発生しないこと，コミット時に処理を施す必要がなく，既存のリポジトリに適用可能である点などが挙げられる．また，実装時の拡張対象コマンドはコンテキストを絞り込むコマンドだけで良い．デメリットとしては，コンテキスト指定時に毎回絞り込み処理を行うので，実行速度が遅くなるという点が挙げられる．

4. プロトタイプ MJgit の実装

4.1 MJgit の概要

提案手法のプロトタイプとして，オープンソースの Git クライアント JGit(注1) を拡張した MJgit を開発した．MJgit の仕様を表 2 に示す．コンテキストの絞り込み操作は Java のソースコードのみが対象となる．利用可能なコンテキストの種類は，実行ステートメント，コメント，Javadoc，アノテーションの 4 種類である．拡張した Git コマンドは show と diff の 2 つであり，動作方法は適宜処理型を採用した．これは，既存のリポジトリにも柔軟に適用できるという適宜処理型の利点を優先したためである．

表 1　提案手法の動作案のメリット・デメリット

	事前処理型	適宜処理型
コミット処理の速度	遅い	速い
コンテキスト絞り込み処理の速度	速い	遅い
既存のリポジトリへの適用	不可	可
実装時の拡張対象コマンド	commit と コンテキストを 絞り込むコマンド	コンテキストを 絞り込むコマンド

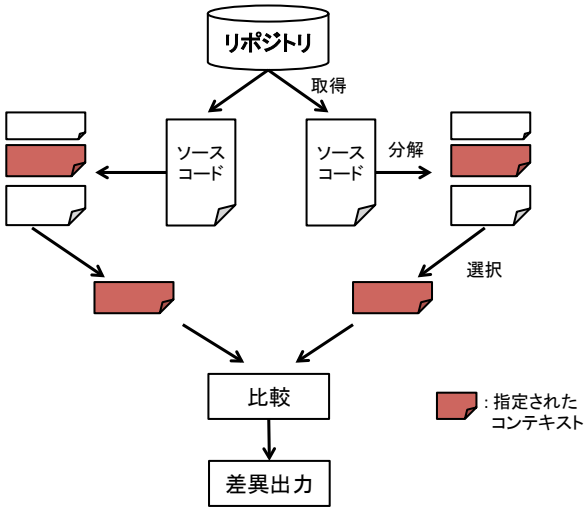

図 5　提案手法の処理の流れ

基本的な操作，及び振る舞いは一般的な Git クライアントと同じである．コンテキスト指定オプション `-cx` により，コンテキスト指定が行われた場合に限り，ソースコードの構文解析，及び情報の絞り込み処理が行われる．具体的な処理の流れは以下で述べる．

4.2 処理の流れ

提案手法の処理の流れを図 5 に示す．この図は，以下で説明する 3 ステップを表している．処理の流れを 3 ステップに分けて以下で説明する．

ステップ 1: 比較されるソースコードの抽出

diff と show は両者とも指定された条件に適った二つのソースコードの差分を表示するコマンドである．JGit のソースコードを見ると，diff と show の処理は，ソースコードをリポジトリから取得してきて以降，比較するまでの処理が全く同じであるとわかった．よって diff と show の拡張のために実装

表 2　MJgit の仕様

動作方法	適宜処理型
拡張対象の Git コマンド	show, diff
コンテキスト指定可能な言語	Java 1.8 以上
コンテキストの指定オプション	`-cx`
指定可能なコンテキストと その指定方法	実行ステートメント（`ast`） コメント（`comment`） Javadoc（`javadoc`） アノテーション（`annotation`）
コンテキストの複数選択	不可
構文エラーが含まれる場合	コンテキスト指定は無効(注2)

(注1)：https://eclipse.org/jgit/

(注2)：抽象構文木が生成できないため

した方法は全く同じである．まず，リポジトリから比較するソースコードを取得している部分を見つけ，そのソースコードをコンテキストによる絞り込みをするために退避させる．

ステップ 2: コンテキストによるソースコードの分解

次に，退避させたソースコードにコンテキストによる絞り込みを行う．ソースコードの構文解析には JDT (Java development tools)(注2)を用いた．JDT の生成する AST（抽象構文木）に基づいて，実行ステートメントやアノテーション等の情報を取得し，ソースコードを分解する．

ステップ 3: 比較されるソースコードと入れ替え

最後に，分割したソースコードからコンテキストで指定された部分を選択し，ステップ 1 で述べたソースコードを抽出した場所に戻す，以降の処理は従来の JGit の処理にもどる．

以上により，指定されたコンテキスト部分のみが比較された差分が出力される．

5. 評価実験

5.1 実験概要

本実験の目的は適宜処理型のデメリットとして述べたコンテキストの処理の拡張による実行速度の低下について調査することおよび，履歴情報の削減が行われているかを調査することである．そのために，既存のプロジェクトに対してプロトタイプを実行し，その実行時間と出力行数を測定した．

5.2 実験の流れ

実験ではまず，プロジェクトにコミット 0 からコミット N までの履歴情報が保存されているとする．MJgit は Java ファイルにだけ動作するため，テキストや他のリソースのみを変更したコミットには一切寄与しない．よって，Java ファイルの変更が行われたコミットだけを対象として実験を行う．従って，プロジェクトの各コミットに対して，Java ファイルの変更がされているかをチェックする．そして，Java ファイルの変更がなされているコミットに対してだけ，拡張したコマンドを実行する．つまり，Java ファイルの変更が行われた全コミットに対して show とその拡張機能全てを実行し，同様に Java ファイルの変更が行われた全コミットの前後に対して diff とその拡張機能全てを実行するということである．そして，各コマンドの実行時間と出力行数を測定した．

5.3 実験対象

実験対象としては，Java で開発されたオープンソースプロジェクト JUnit4 と，Log4j を採用する．それぞれのプロジェクトの概要を表 3 に示す．いずれのプロジェクトも 10 年以上継続，かつ 1,000 コミットを超えている．表の下二行は，Java ファイルの変更があったコミット数のことである．

5.4 実験結果

JUnit4 に対する実験結果を，図 6 図 7 図 8 図 9 に示す．なお，Log4j の結果は JUnit4 の結果とほぼ同様の傾向が確認されたため，ここでは省略する．

各図における横軸は実行時間 (秒)，または出力行数 (行) であり，縦軸にはどのコンテキストを指定するかというコマンドの種類が並べられている．なお，全ての結果において，実行時間の 4 秒以上の外れ値および，出力行数の 800 行以上の外れ値は表示していない．

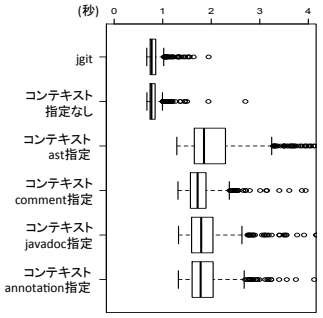

図 6 JUnit4 に対する show の実行時間

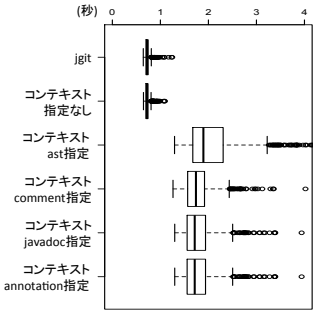

図 7 JUnit4 に対する diff の実行時間

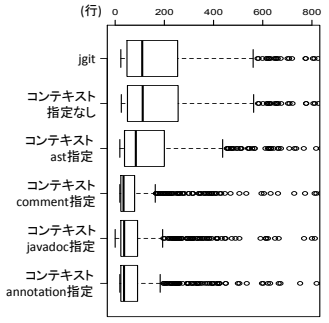

図 8 JUnit4 に対する show の出力行数

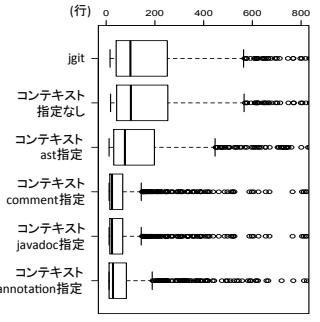

図 9 JUnit4 に対する diff の出力行数

どの結果を確認しても上二つの箱ひげ図は同等である．すなわちオリジナルである JGit の結果と MJgit のコンテキスト指定をしていない場合の結果が同等であると言える．つまり，MJgit の動作はオリジナルである JGit と同等であることが確認できる．実行時間について見てみると，どのプロジェクト，どのコンテキスト，どちらのコマンドを実行した場合でも，コンテキスト指定なしと比較した実行時間の倍率は 2.5 倍程度であった．コンテキスト指定なしの実行速度は平均 0.8 秒程度であるため，2.5 倍はすなわち 2 秒程度ということになる．また，各プロジェクトの実行時間の最大値は，JUnit4 でコンテキストに実行ステートメントを指定した場合の 14 秒，Log4j でコンテキストに実行ステートメントを指定した場合の 18 秒

表 3 実験対象プロジェクトの概要

	JUnit4	Log4j
開始日	2000/12/3	2000/11/16
コミット数	1,165	2,932
全ファイル数	636	438
Java ファイル数	438	309
show の実行対象となるコミット数	630	1,890
diff の実行対象となるコミット数	870	1,891

(注2)：https://projects.eclipse.org/projects/eclipse.jdt

であった．

出力行数については，どの結果においても，実行ステートメントの減少率が一番小さく，コンテキス指定なしと比較して平均 0.8 倍の出力行数を示した．また，アノテーションを指定した場合が最も行数の減少率が大きく，平均で 0.4 倍程度であった．

6. 考　　察

6.1 情報の絞り込みの程度

出力行の削減割合は，すなわちコンテキスト絞り込みにより，どの程度関心以外の情報を省けるかを表す一つの指標となる．当然ながら，ソースコード中の記述内容のほとんどは実行ステートメントに関するものであり，実行ステートメントというコンテキストで削減できる割合は 2 割程度にとどまった．より情報を絞り込むためには，実行ステートメントをさらに詳細に分類し，指定できるように拡張する必要があるといえる．抽象構文木の生成は JDT を用いており，さらなるコンテキストの詳細化は比較的容易に実現可能である．また，実行ステートメント以外のコメントや Javadoc，アノテーションといった記述割合が低いコンテキストでは，概ね半分程度に行数が削減されていた．これらのコンテキストに関心がある際には，不要な情報を省略できていると見なすことができる．

6.2 提案手法による実行速度の低下

拡張クライアントでは，Git の標準的な処理に加え，抽象構文木の生成処理とソースコードの分割処理が追加されており，実行速度は低下する．実験の結果より，拡張による実行時間の増加は平均 0.8 秒で終わる処理に対して，平均 2.5 倍の増加にとどまっていた．数秒で終えることができるという観点から，実用的な範囲に収まっていると考える．また，もっとも実行速度が悪化するケースは，Log4j に対するあるコミット（コミット ID: cb47e2e8）に対して show コマンドを実行した場合であり，コンテキスト指定なしで 1.1 秒，実行ステートメントの指定で 18 秒であった．このコミットでは，全ての Java ファイルに対してライセンス文が一括置換されており，302 個のファイルが変更されていた．このような巨大なコミットは希であり，一般的な開発過程においては速度の低下よりも情報の絞り込みという利点が生きると考えられる．

7. あ と が き

本研究では，版管理システムにおいて履歴情報を取得するにあたり，その情報をコンテキストによって絞り込む拡張機能を提案した．これにより，ユーザーは見る必要がない情報を絞り込め，本当に欲しい情報を取得することができるようになり，作業効率の向上が期待できる．提案手法の有用性を確認するために，JGit を拡張したプロトタイプを実装した．また，評価実験を行い，プロトタイプによって低下する実行時間が実用範囲内であることと，出力行数が減少している，つまり，情報が絞り込めていることを確認できた．

今後の課題として，プロトタイプの改変と，被験者実験を行うことを考えている．本研究で実装したプロトタイプでは，指定できるコンテキストが限定されている，コンテキストの複数選択ができない，などの制限が多い．そこで，プロトタイプを改変し，提案手法の様々な面での有用性を示していきたいと考えている．また，本研究では定量的な実験のみを行ったので，確認できた有用性はあくまで数値的なものでしかない．例えば，出力行数の減少が確認できたが，その減少がそのまま有用性に繋がっているとは断言できない．そこで，実際に開発者に使用してもらうことで現実に即した評価を行いたいと考えている．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（S）（課題番号：JP25220003），および文部科学省研究費補助金若手研究（B）（課題番号：JP26730155）の助成を得て行われた．

文　　献

[1] B. De Alwis and J. Sillito, “Why are software projects moving from centralized to decentralized version control systems?,” Proc. Works. Cooperative and Human Aspects on Softw. Eng., pp.36–39, 2009.

[2] 谷宗一郎，上野秀剛，“コンテキストの推定による開発支援システム間の情報統合，” Technical Report 9，奈良工業高等専門学校，奈良工業高等専門学校，jul 2010.

[3] H. Kagdi, M.L. Collard, and J.I. Maletic, “A survey and taxonomy of approaches for mining software repositories in the context of software evolution,” J. Softw. Maint. Evol., vol.19, no.2, pp.77–131, 2007.

[4] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” Proc. Int’l Conf. Softw. Eng., pp.181–190, ICSE, 2008.

[5] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, “Lines of comments as a noteworthy metric for analyzing fault-proneness in methods,” IEICE Transactions on Information and Systems, vol.E98.D, no.12, pp.2218–2228, 2015.

[6] V. Sinha, A. Lazar, and B. Sharif, “Analyzing developer sentiment in commit logs,” Proceedings of the 13th International Conference on Mining Software Repositories, pp.520–523, MSR, 2016.

[7] G. Concas, M. Marchesi, A. Murgia, R. Tonelli, and I. Turnu, “On the distribution of bugs in the eclipse system,” IEEE Trans. Softw. Eng., vol.37, no.6, pp.872–877, 2011.

[8] C. Andersson and P. Runeson, “A replicated quantitative analysis of fault distributions in complex software systems,” IEEE Trans. Softw. Eng., vol.33, no.5, pp.273–286, 2007.

[9] E.J. Weyuker, T.J. Ostrand, and R.M. Bell, “Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models,” Empirical Softw. Engg., vol.13, no.5, pp.539–559, 2008.

[10] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, “An analysis of developer metrics for fault prediction,” Proceedings of the 6th International Conference on Predictive Models in Software Engineering, pp.18:1–18:9, PROMISE, 2010.