特別研究報告

題目

版管理システムにおける履歴情報取得機能の拡張

指導教員

楠本　真二　教授

報告者

佐々木　美和

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 28 年度 特別研究報告

版管理システムにおける履歴情報取得機能の拡張

佐々木　美和

内容梗概

SVN や Git をはじめとする多くの版管理システムが現在多くのソフトウェア開発プロジェクトに導入されている．しかし，従来の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いる．そのため，履歴情報からコメントの変更だけを抽出する，Javadoc が変更されているコミットログだけを表示する，といった操作はサポートされていない．そこで本研究では，版管理システム上でのソースコード内のコンテキスト（構文情報）を利用した情報の絞り込み操作の実現を目的として，版管理クライアントの拡張を行う．これによって使用者は，必要のない部分が削られたより小さく要求に適っ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，プロトタイプを実装し，評価実験を行った．その結果，従来の版管理システムの実行速度と比較した提案手法の実行速度の低下は平均 2.5 倍程度であることが確認できた．従来の版管理システムの実行時間の平均は 0.8 秒程度であったので，提案手法の実行時間の平均はおそよ 2 秒であった．また，実行時間の最大値をとってみても，18 秒であったので，これは現実に提案手法を使用していく際に実用範囲内であると言えた．出力行数についての結果を見ると，従来の出力の 0.8 倍から 0.3 倍程度まで出力行数を減少できていることが確認できた．これにより，提案手法が確かに情報を削減できていることが言えた．

主な用語

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

目次

1 まえがき **1**

2 研究動機 **3**

3 提案手法 **6**

3.1 コンテキスト 6

3.1.1 構文コンテキスト 6

3.1.2 履歴コンテキスト 7

3.2 従来の版管理システムとの違い 8

3.3 拡張対象コマンド 8

3.4 提案手法の動作 8

4 プロトタイプ MJgit の実装 **12**

4.1 MJgit の概要 12

4.2 処理の流れ 13

5 評価実験 **15**

5.1 実験概要 15

5.2 実験の流れ 15

5.3 結果の計測方法 15

5.4 実験対象 17

5.5 実験結果 17

6 考察 **21**

6.1 情報の絞り込みの程度 21

6.2 提案手法による実行速度の低下 21

6.3 コンテキスト指定が活かされる事例 21

7 妥当性への脅威 **22**

8 あとがき **23**

謝辞 **24**

参考文献 **25**

図目次

1 提案手法と従来の版管理システムとの違い 9
2 提案手法の動作案 11
3 提案手法の処理の流れ 13
4 実験内容 16
5 JUnit4 に対する show の実行時間 18
6 JUnit4 に対する diff の実行時間 18
7 JUnit4 に対する show の出力行数 18
8 JUnit4 に対する diff の出力行数 18
9 Log4j に対する show の実行時間 19
10 Log4j に対する diff の実行時間 19
11 Log4j に対する show の出力行数 19
12 Log4j に対する diff の出力行数 19

表目次

1 提案手法の動作案のメリット・デメリット 10
2 MJgit の仕様 12
3 実験対象プロジェクトの概要 17

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][2][3]．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 [4]，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．そのため，そこからソースコードの本質的な意味の変化を読み解くには，大きな労力がかかってしまう [5]．その労力を避け，ソースコードの変更による的確な意味を得るためには，コメントやステートメントに対する変更などを考慮した差分を出力する必要がある [6] [7] [8].

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 [9] 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 [10] や，バグとコメント行数の関係の調査 [11]，コミットログからの開発者の感情の分析 [12] 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．

本研究では，版管理システム上でのコンテキストを利用した情報の絞り込み操作の実現を目的として，版管理クライアントの拡張を行う．ここでのコンテキストは大別して，上記に挙げたようなソースコードの構文情報に基づくものと，版管理システム上の履歴情報に基づくもの，の 2 種類に分類することができる．本稿では前者の構文情報に関するコンテキストに着目し，その実現方法について検討する．さらにプロトタイプとなる拡張 Git クライアント MJgit を開発する．

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．実験題材としては，オープンソースソフトウェアのJUnit4 及び Log4j を採用する [13][14]．実験結果により, 確かに情報量を削減できることと，実行速度の低下が実用範囲内であることが確認できた．また，ステートメント以外のコンテキストの出力行数が大きく減少したことから，提案手法はステートメント以外のコンテキストの絞り込みを行う際に特に役に立つであろうことが言えた．

以降，2 章では研究動機として，従来の版管理システムの問題点について述べる．3 章では提案手法について説明し，4 章ではプロトタイプの実装について述べる．5 章では評価実験の内容と実験結果について述べる．6 章では評価実験の結果について考察を行い，7 章では妥当性への脅威について述べる．最後に 8 章で本研究のまとめと今後の課題について述べる．

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．

Listing 1: Example.java

```
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.doSomething1();
9        break;
10       case 2:
11       this.doSomething2();
12       break;
13       ...
14       // Handle an error case
15       case -1:
16       this.handleError();
17       default:
18       this.doDefault();
19     }
20   }
21   ...
22 }
```

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 [15, 16]，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 [17, 18]，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
f0984121 (shinsuke 2017-02-01 20:25:36  1) /* Licensed under the ... */
ac5f6dac (m-sasaki 2017-02-01 20:25:36  2) public class Example {
ac5f6dac (m-sasaki 2017-02-01 20:25:36  3)   public void Foo() {
...
ac5f6dac (m-sasaki 2017-02-01 20:25:36  6)     switch (case) {
ac5f6dac (m-sasaki 2017-02-01 20:25:36  7)       case 1:
ac5f6dac (m-sasaki 2017-02-01 20:25:36  8)       this.doSomething1();
ac5f6dac (m-sasaki 2017-02-01 20:25:36  9)       break;
...
b8beb025 (n-ogura  2017-02-04 12:00:30 14)       // Handle an error case
b8beb025 (n-ogura  2017-02-04 12:00:30 15)       case -1:
b8beb025 (n-ogura  2017-02-04 12:00:30 16)       this.handleError();
ac5f6dac (m-sasaki 2017-02-01 20:25:36 17)       default:
ac5f6dac (m-sasaki 2017-02-01 20:25:36 18)       this.doDefault();
...
```

この結果より，バグ箇所を編集した開発者（n-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-statement
ac5f6dac (m-sasaki 2017-02-01 20:25:36  6)     switch (case) {
ac5f6dac (m-sasaki 2017-02-01 20:25:36  7)       case 1:
ac5f6dac (m-sasaki 2017-02-01 20:25:36  8)       this.doSomething1();
ac5f6dac (m-sasaki 2017-02-01 20:25:36  9)       break;
...
```

さらに，バグを埋め込んだコミット ID:b8beb025 の作業の詳細を確認する際は，以下のような show コマンドにより変更内容を取得できる．

```
$ git show Example2.java b8beb025
commit b8beb025798db84c9fccfe44d394dbf1ec404974
Author: n-ogura <n-ogura@sdl.osaka-u.ac.jp>
Date:   Mon Feb 04 12:00:30 2017 +0900

  Add an error handler for Foo

diff --git a/Example.java b/Example.java
index 480d81a..4fb1b29 100755
```

```
--- a/Example.java
+++ b/Example.java
@@ -15,4 +15,6 @@
 ...
+       case -1:
+       this.handleError();
        default:
        this.doDefault();
 ... (abbr.)
@@ -31,3 +31,12 @@
+    public void handleError() {
+       if (status == 0) {
 ...
```

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．以下は handleError 関数に絞り込んだ際の例である．

```
$ git show Example2.java b8beb025 --context=in-method:handleError
commit b8beb025798db84c9fccfe44d394dbf1ec404974
Author: n-ogura <n-ogura@sdl.osaka-u.ac.jp>
Date:   Mon Feb 04 12:00:30 2017 +0900

   Add an error handler for Foo

diff --git a/Example.java b/Example.java
index 480d81a..4fb1b29 100755
--- a/Example.java
+++ b/Example.java
@@ -31,3 +31,12 @@
+    public void handleError() {
+       if (status == 0) {
 ...
```

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する．以下でコンテキストの定義，従来の版管理システムとの違い，拡張対象になるコマンド，さらに提案手法の動作について詳しく説明する．また 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　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，及び開発履歴から取得可能な情報の 2 つに大別することができる．ここでは前者を「構文コンテキスト」，後者を「履歴コンテキスト」と呼ぶこととする．2 つのコンテキストについて以下で説明する．

3.1.1　構文コンテキスト

構文コンテキストとは，ソースコードが持つ構文に関する情報のことを指す．構文コンテキストは様々な分類が可能であるが，本研究では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 3 種類に分類する．各コンテキストを具体的な例とともに説明する．

- **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) {
    ```
  - **変数宣言ステートメント**: 変数宣言の記述箇所
    ```
    int n = 0;
    ```
  - **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
```

3.1.2 履歴コンテキスト

履歴コンテキストとは，版管理システムが保持している履歴情報から得られる情報のことを指す．具体的な履歴コンテキストは様々なものが考えられる．

- ソースコードの編集を行った著者名
- ソースコードの変更回数
- コミット時の総編集行数
- あるファイルと同時にコミットされたかどうか
- ...

これらのコンテキストを整理することで，例えば以下のような場合に提案手法が役に立つ．

- 変更された回数の多いソースコードを探し，そのプロジェクトの中心となっているソースコードを発見する．
- 変更回数の少ないソースコードを発見し，メンテナンスをする．
- あるファイルにバグを見つけた．同時にコミットされたファイルもメンテナンスしたいので同時にコミットされたファイル一覧を見る．

- 多くのコミットをしている著者の行ったコミットを見て，どのような貢献をしているか確認したい．

Git には既に，ある著者によって行われたコミットログだけを表示する機能や，特定のファイルがコミットされたログだけを出力する機能など，履歴コンテキストに関連するコマンドがいくつかサポートされている．しかし，これらのコマンドで履歴コンテキストによる絞り込みを直接の目的としたものは少なく，実際に履歴コンテキストによる絞り込みを行いたい場合には，シェルスクリプトによりカウントやソート等のテキスト処理が必要となる場合が多い．そのため，履歴コンテキストによる絞り込みを行いたい場合には，使用者自身が既存のコマンドやシェルスクリプトを駆使しなければならないというのが現状である．

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(a) と図 1(b) 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 1(b)）．

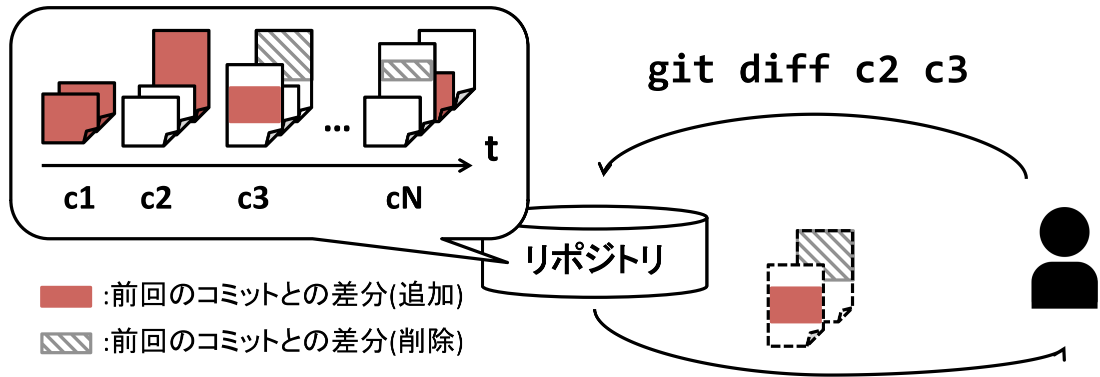
3.3 拡張対象コマンド

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．

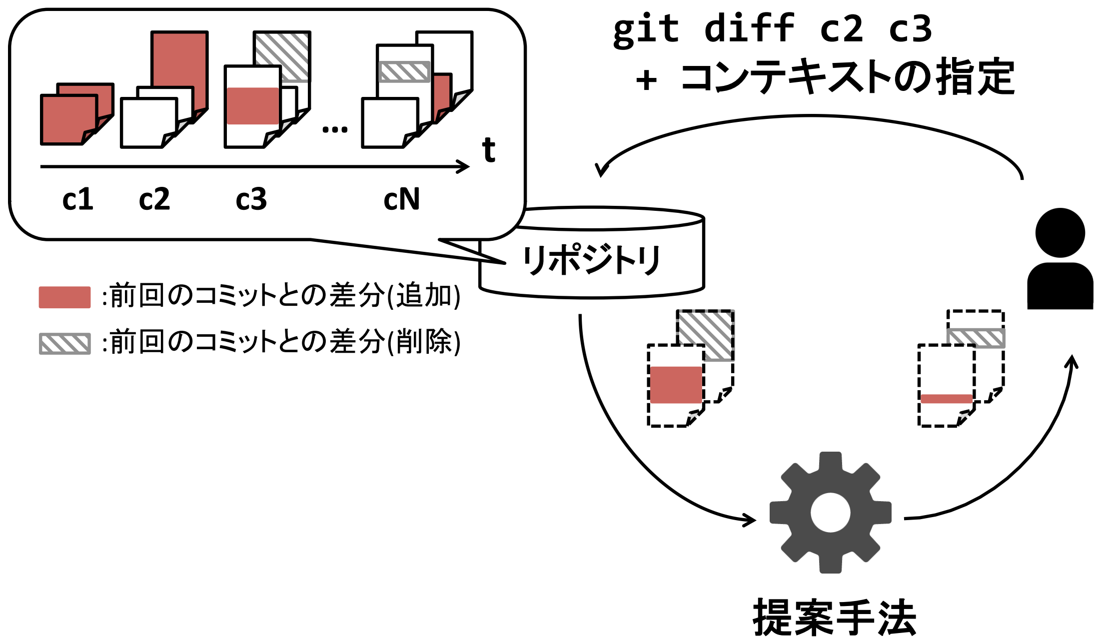
- **diff**: 手元のソースコードと最新コミットとの差分や，2 つのコミット間の差分などを表示する
- **show**: コミットの詳細を見る
- **log**: コミットのログを見る
- **blame**: ファイル内の各行の最終更新の情報を見る
- **grep**: ファイルから文字列を検索する

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 2(a) に，適宜処理型を図 2(b) 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．



(a) 従来の Git の動作



(b) 提案手法の動作

図 1: 提案手法と従来の版管理システムとの違い

図 2(a) 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．

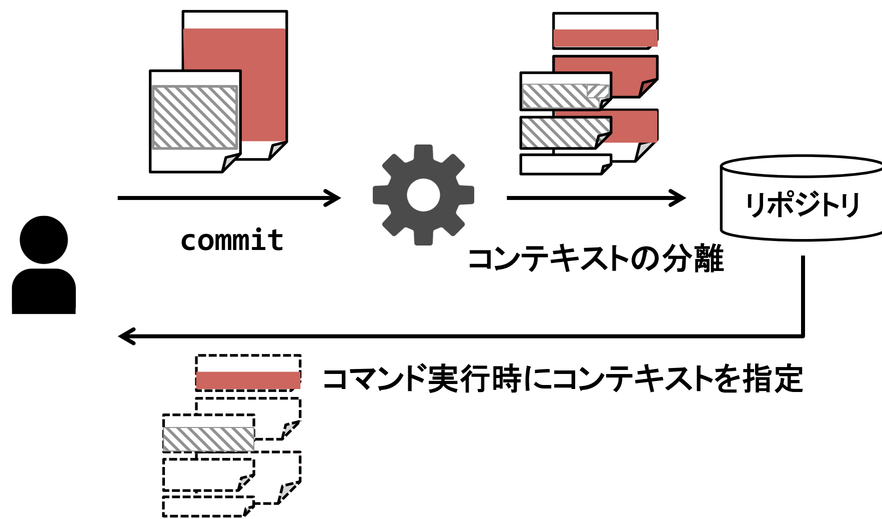
事前処理型のメリットは，コミット時に絞り込みを済ましておくことで，今後一切絞り込み処理を行わなくてよくなり，コンテキスト絞り込み時の速度が速くなることである．デメリットは，コミットの実行速度が遅くなることと，コミット時に処理を行う必要があるので，既にコミットが実行されてしまっている既存のリポジトリに適用できないこと，また，実装時の拡張対象コマンドとして，コンテキストを絞り込めるコマンド（diff や show など）に加え，commit コマンドも拡張する必要があることである．

図 2(b) 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．

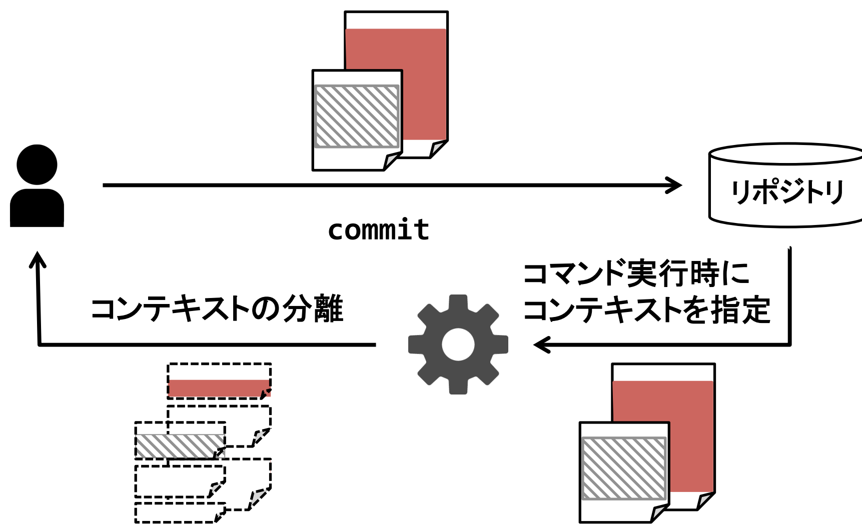
適宜処理型のメリットとしては，コミット時の速度低下が発生しないこと，コミット時に処理を施す必要がなく，既存のリポジトリに適用可能である点などが挙げられる．また，実装時の拡張対象コマンドはコンテキストを絞り込むコマンドだけで良い．デメリットとしては，コンテキスト指定時に毎回絞り込み処理を行うので，実行速度が遅くなるという点が挙げられる．

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 絞りこむコマンド	コンテキストを 絞り込むコマンド



(a) 事前処理型



(b) 適宜処理型

図 2: 提案手法の動作案

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[19] 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，3.1.1 章で述べた 2 種類のコンテキストの内，構文コンテキストのみが対象である．より具体的には，実行ステートメント，コメント，Javadoc，アノテーションの 4 種類が対象となる．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．

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次章で述べる．

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[2]

[2]抽象構文木が生成できないため

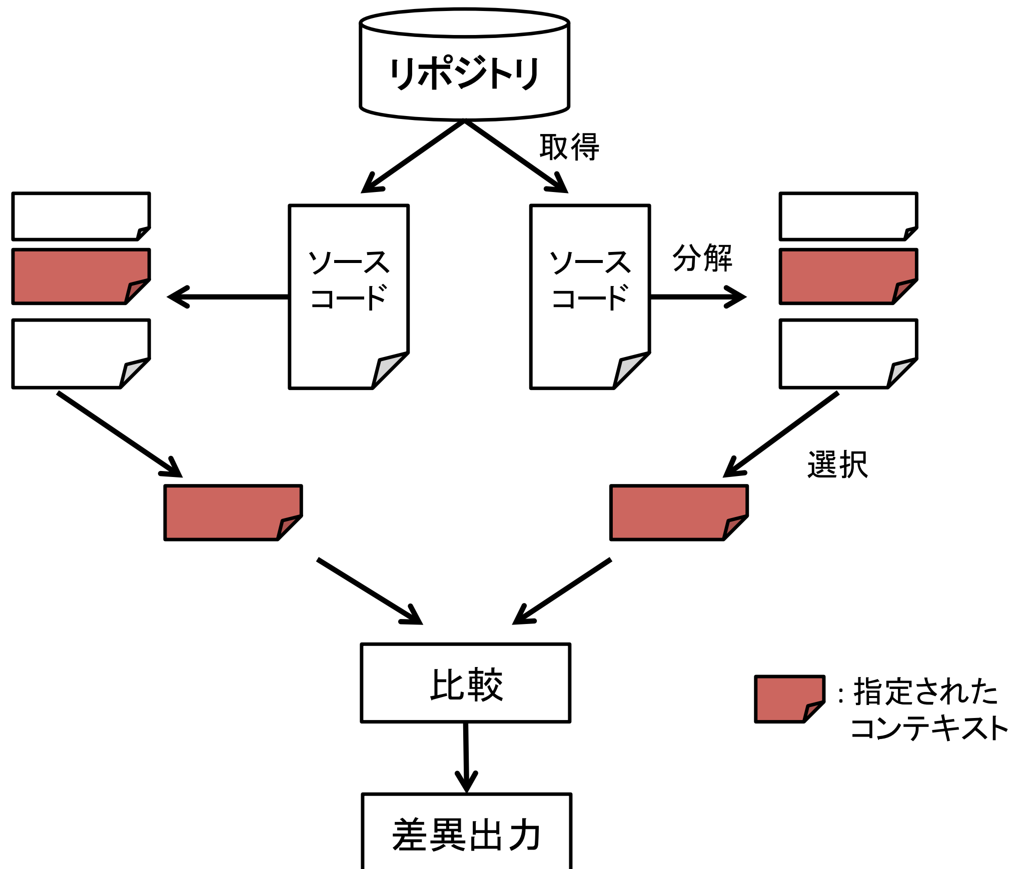


図 3: 提案手法の処理の流れ

4.2　処理の流れ

提案手法の処理の流れを図 3 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．

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)[1] 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する．そして，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．

[1] https://projects.eclipse.org/projects/eclipse.jdt

5 評価実験

提案手法を実装し，有用性を評価するために評価実験を行った．実験内容と結果について述べる．

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

実験内容を表した図を図 4(a)，図 4(b) に示す．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．まずこ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，各図に示されたコマンド群を実行する．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，各コマンド実行時の実行時間と出力行数を測定した．

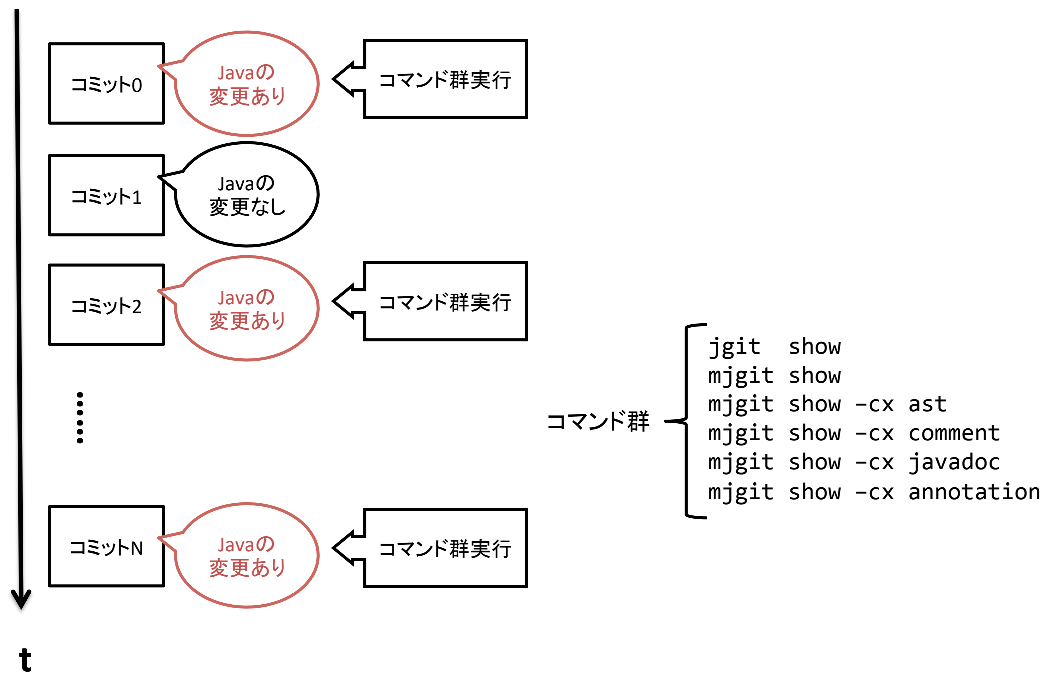
MJgit は Java ファイルにだけ動作するため，テキストや他のリソースのみを変更したコミットには一切寄与しない．よって，Java ファイルの変更が行われたコミットだけを対象として実験を行う．各図のコマンド群は，MJgit で拡張したコマンド diff と show の，コンテキストを指定しない場合と，各種コンテキストを指定した場合の実行時間と出力行数を拡張前の JGit と比較するために実行するものである．

5.3 結果の計測方法

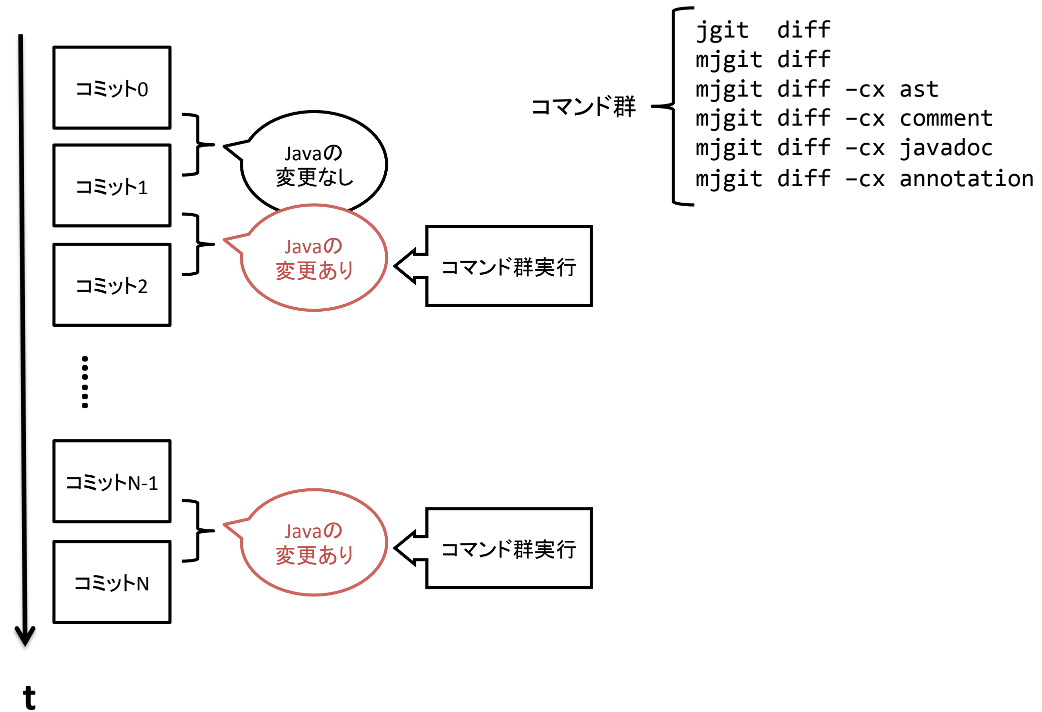
計測対象のメトリクスは時間と行数である．そのため，Linux の標準コマンドを用いて比較的容易に収集することが可能である．実行時間の計測としては，コマンドの実行直後からその処理結果が得られるまでの時間の長さを用いる．最新リビジョン（HEAD）を対象とした際の計測方法を以下に示す．

```
time mjgit show HEAD
time mjgit show HEAD -cx ast
time mjgit show HEAD -cx comment
time mjgit show HEAD -cx javadoc
...
```

ここでは time コマンドを用いて時間を計測している．これを全てのコンテキスト操作，及び全てのリビジョンに対して実行し，その出力結果を整理することで結果を収集する．



(a) show の実験内容



(b) diff の実験内容

図 4: 実験内容

また行数の計測方法を以下に示す.

```
mjgit show HEAD | wc
mjgit show HEAD -cx ast | wc
mjgit show HEAD -cx comment | wc
mjgit show HEAD -cx javadoc | wc
...
```

時間の計測と同様に，wc コマンドにより計測が可能である.

5.4 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[13] と，Log4j[14] 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，それぞれ図 4(a)，4(b) で説明されたコマンド群を実行すべき対象と判断されたコミット数，つまり，Java ファイルの変更があったコミット数のことである.

5.5 実験結果

JUnit4 に対する実験結果を箱ひげ図にして図 5 図 6 図 7 図 8 に, Log4j に対する実験結果を同様に箱ひげ図にして図 9 図 10 図 11 図 12 に示す.

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

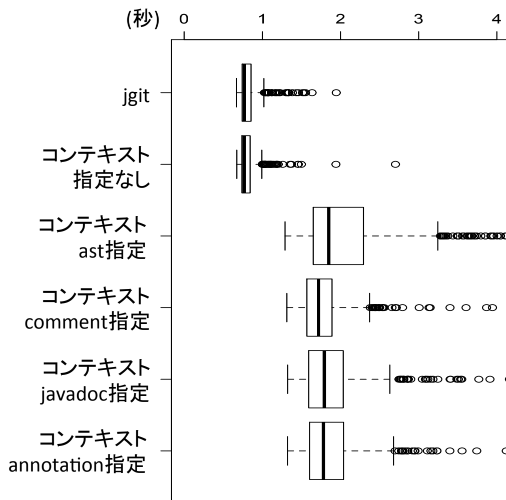


図 5: JUnit4 に対する show の実行時間

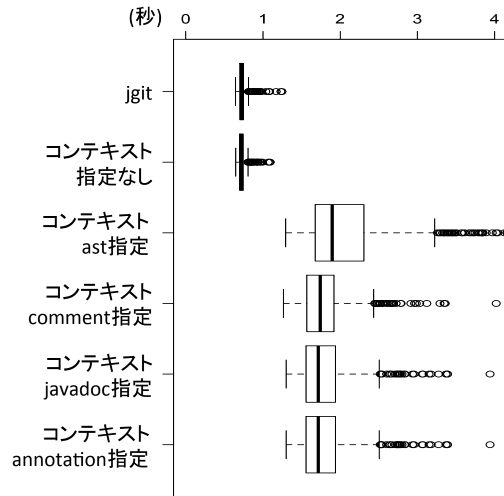


図 6: JUnit4 に対する diff の実行時間

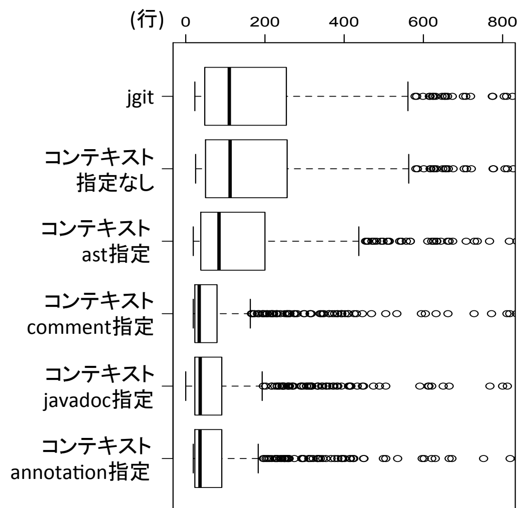


図 7: JUnit4 に対する show の出力行数

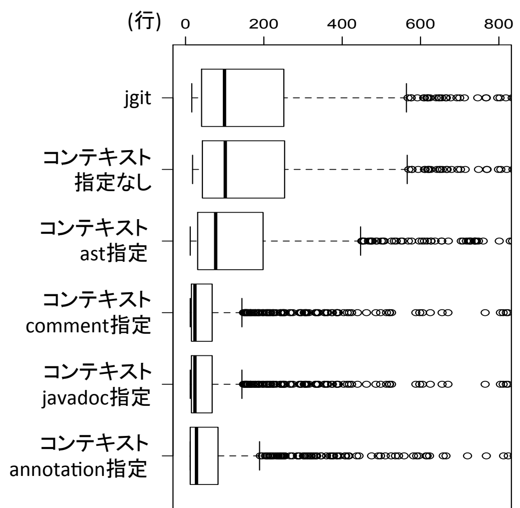


図 8: JUnit4 に対する diff の出力行数

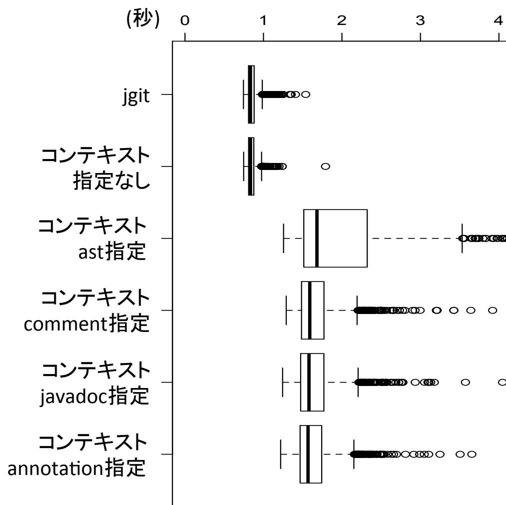


図 9: Log4j に対する show の実行時間

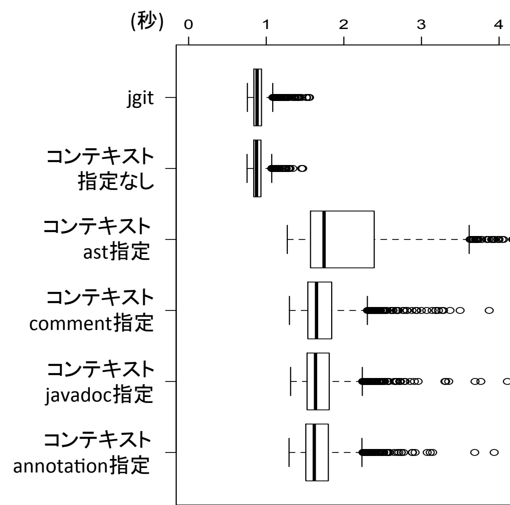


図 10: Log4j に対する diff の実行時間

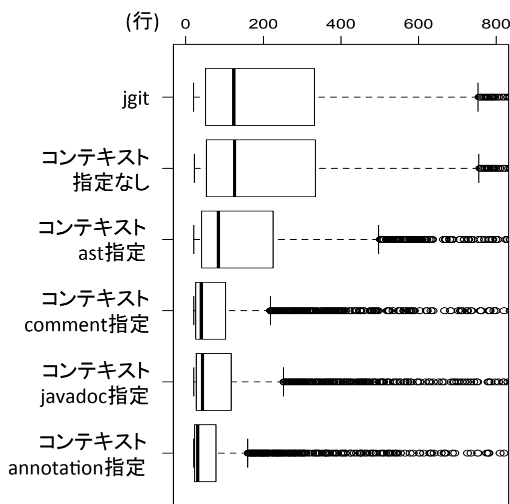


図 11: Log4j に対する show の出力行数

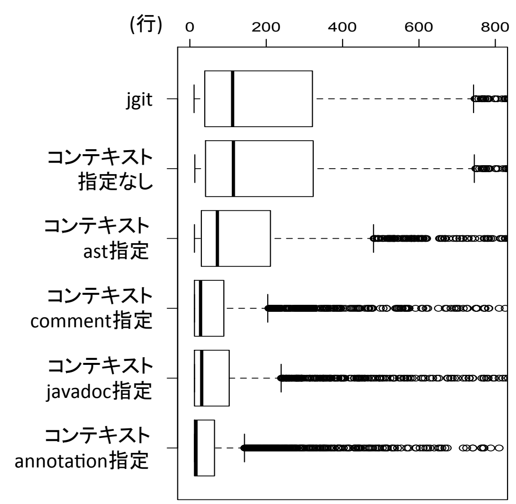


図 12: Log4j に対する diff の出力行数

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．

どの結果を確認しても上二つの箱ひげ図，すなわち「jgit show(diff)」と「mjgit show(diff)」の箱ひげ図は同等である．つまり，MJgit の実行速度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に最も行数の削減率が大きく，平均で 0.4 倍程度であった．

6　考察

本章では，実験で得られた結果について考察する．

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コンテキスト指定で 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．

6.3　コンテキスト指定が活かされる事例

前章で説明した，Log4j におけるライセンス文の変更コミットについて考える．このリビジョンでは，Java の 302 個のファイルに対する全ての変更内容はコピーライトに関するものに限定されていた．版管理システムの利用に対する調査として，単一のコミットは単一の意図に限定するべきであるものの，複数の意図が混在したコミットが多数存在することが指摘されている [20]．ここで挙げた Log4j のコミットに対して，コンテキストを指定した際の結果を分析したところ，実行ステートメントが一切変更されておらず，意図の混在しない「良い」コミットであった．提案手法は，このような意図が混在したコミットを洗い出す目的でも利用できる可能性がある．

7 妥当性への脅威

本章で，評価実験に含まれる妥当性への脅威について述べる．本研究で行った実験は，2 つのプロジェクトのみを題材とした．他のプロジェクトに対して行った場合には，異なる結果が得られる可能性がある．

また，実験では Java プログラムが変更してある全コミットに対して MJgit を実行した結果を収集したが，それぞれのコミットに対し，測定したい項目を一度ずつしか実行していおらず，繰り返しのないデータとなる．実行速度のようなノイズが影響する要因に対しては，実験の繰り返しによりそれらの変動要因を除外する必要がある．

実験結果として行数の減少は確認できたが，それが必ずしも有用性に繋がるとは限らない．これは，開発者に対する被験者実験を行うことで有用性に関する評価が得られると考えられる．本実験では，コンテキストを制限したり，ソースコードを Java に限定するなど，制限のあるプロトタイプでの実験であったが，コンテキストの種類を増やす，複数選択を可能にする，Java 以外のソースコードにも対応させるなどして実験を行えば，提案手法についてのより詳細な評価を得ることができると考えられる．

8 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

本研究を行うにあたって，理解あるご指導をしていただき，励まして頂きました楠本真二教授に心より感謝申し上げます．

本研究に関して，的確なご助言と，プロトタイプ MJgit の名前のアイディアを頂きました，肥後芳樹准教授に深く感謝申し上げます．

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂き，何度も相談に乗っていただきました，柗本真佑助教に深く感謝申し上げます．

本研究を進めるにあたり，実験項目について有意義な提案を頂きました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博士前期課程 2 年の小倉直徒氏に心より感謝致します．

本研究を進めるにあたり，MJgit の実装環境を整える際に多大なるご助力を頂きました大阪大学基礎工学部情報科学科 4 年の有馬諒君に心より感謝申し上げます．

本論文を書き進めるにあたり，tex の使い方を詳しく教えて頂きました大阪大学基礎工学部情報科学科 4 年の松尾裕幸君に心より感謝申し上げます．

本研究を進めるにあたり，励まし，ご助言を頂きました楠本研究室の皆様のご協力に心より感謝致します．

最後に，本研究に至るまでに，授業でお世話になりました大阪大学基礎工学部情報科学科の先生方に，この場を借りて心から御礼申し上げます．

参考文献

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

[2] 福安直樹, 吉田敦. プログラムの構文要素に基づく版管理システムのための差分取得手法. 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, Vol. 105, No. 490, pp. 43–48, 2005.

[3] 大森隆行, 丸山勝久. 開発者による編集操作に基づくソースコード変更抽出. 情報処理学会論文誌, Vol. 49, No. 7, pp. 2349–2359, 2008.

[4] 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, 2010.

[5] 尾崎憲幸, 吉田敦, 山本晋一郎, 阿草清滋. プログラムの正規化に基づいた差分抽出法の提案. 情報処理学会研究報告ソフトウェア工学（SE）, Vol. 1999, No. 37, pp. 25–32, 1999.

[6] 吉田敦, 山本晋一郎, 阿草清滋. 依存関係に基づく差分抽出ツール. 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, Vol. 95, No. 53, pp. 53–60, 1995.

[7] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pp. 31–34, 2008.

[8] 大森隆行, 丸山勝久. 追跡性を考慮したソースコード変更の抽出. 情報処理学会研究報告ソフトウェア工学（SE）, Vol. 2007, No. 33, pp. 159–166, 2007.

[9] Huzefa Kagdi, Michael L. Collard, and Jonathan 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.

[10] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. Int'l Conf. Softw. Eng.*, ICSE, pp. 181–190, 2008.

[11] Hirohisa Aman, Sousuke Amasaki, Takashi Sasaki, and Minoru 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.

[12] Vinayak Sinha, Alina Lazar, and Bonita Sharif. Analyzing developer sentiment in commit logs. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR, pp. 520–523, 2016.

[13] JUnit4. http://junit.org/junit4/.

[14] Log4j. http://jakarta.apache.org/.

[15] Giulio Concas, Michele Marchesi, Alessandro Murgia, Roberto Tonelli, and Ivana Turnu. On the distribution of bugs in the eclipse system. *IEEE Trans. Softw. Eng.*, Vol. 37, No. 6, pp. 872–877, 2011.

[16] Carina Andersson and Per Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Trans. Softw. Eng.*, Vol. 33, No. 5, pp. 273–286, 2007.

[17] Elaine J. Weyuker, Thomas J. Ostrand, and Robert 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.

[18] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE, pp. 18:1–18:9, 2010.

[19] JGit. https://eclipse.org/jgit/.

[20] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC, pp. 262–265, 2014.