修士学位論文

題目

開発履歴の理解支援を目的とした **Git** クライアントの拡張

指導教員

楠本　真二　教授

報告者

佐々木　美和

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

平成 30 年度 修士学位論文

開発履歴の理解支援を目的とした Git クライアントの拡張

佐々木　美和

内容梗概

効率的なソフトウェア開発の管理を実現するためには，版管理システムを用いた開発履歴の理解が必須である．しかし，一般的な版管理システムにおける履歴操作は grep や diff などのテキストベースな操作に制限されている．そのため，ソースコードの構文情報や意味に基づいた履歴操作は容易ではない．ソースコード検索に関する様々な研究が行われてきたが，これらの方法では開発履歴を追うことができない．

また，一般的な開発履歴の一覧はメタ情報のみを表示しており，検索対象もメタ情報のみを排他的に検索する手法である．そのため，上で述べたように，ソースコードの中身に基づいた検索は難しく，また，検索にヒットする結果のみを出力するので，履歴の文脈を理解することが難しくなる．その検索方法も，開発者は基本的に版管理システムにあらかじめ用意された方法を用いており，開発者自ら設定した検索手法を用いて情報を絞り込むことは容易ではない．

本研究では，版管理システムとして広く用いられている Git を対象に，リポジトリの履歴操作系コマンドに対してソースコード検索技術を統合する手法と，開発履歴にコミット情報をもとにタグを付ける手法を提案する．

ソースコード検索技術の統合により使用者の関心外の情報を削除することが可能となり，効率的な情報の絞り込みが可能となる．提案手法の実現のために，Git の Java 実装である JGit を拡張した MJgit を設計し実装した．MJgit では git-show や git-diff，git-log といった過去のコミット情報を取得するコマンドに対し，メソッド名や変数名の指定といった検索クエリを使用できる．

また，タグ付与機能の追加により，開発履歴を開発者の設定したタグで分類することが可能となり，使用者が開発履歴を理解する助けとなる．こちらの提案手法の実現のために，同じく JGit を拡張した BJgit を設計し，実装した．BJgit では git-log で出力される開発履歴に，コミット情報に基づいたタグを付けたり，タグに基づいて開発履歴のフィルタをしたりすることで，使用者の理解を支援することが可能である．また，ユーザー自らタグを作成する機能や，タグ作成のためのフレームワークも提供した．

MJgit と BJgit の評価実験として，実際のリポジトリを用いた性能評価実験，及び被験者を用いた有用性の確認実験を行う．

主な用語

版管理システム

開発履歴

履歴理解

ソースコード検索

タグ

Git

MJgit

BJgit

情報の絞り込み

情報の付与

目次

第 I 部　序論　1

1　研究の背景と目的　1

2　用語　3

2.1　版管理システム 3

2.2　JDT 3

2.3　AST(抽象構文木) 3

第 II 部　MJgit　4

3　はじめに　4

4　研究動機　6

4.1　例 1 あるメソッドの開発履歴の理解 6

4.2　例 2 バグ修正と一時的なパッチの除去 6

5　提案ツール：MJgit　9

5.1　概要 9

5.2　処理の流れ 9

5.3　コード検索技術 10

5.4　拡張対象となるコマンド 11

5.5　実装 12

5.6　各コマンドの振る舞いと出力例 14

6　性能評価実験　17

6.1　概要 17

6.2　実験設計 17

6.3　実験結果 18

6.4　考察 20

7　被験者実験　22

7.1　概要 22

7.2 被験者 22
7.3 実験タスク 22
7.4 タスクの実行順序とツール有無の割り当て 23
7.5 実施の流れ 24
7.6 定性評価 24
7.7 実験結果 25
7.8 考察 29

8 妥当性の脅威 30

9 関連研究 31

10 おわりに 32

第 III 部 BJgit 33

11 はじめに 33

12 研究動機 35

13 提案ツール：BJgit 38
13.1 概要 38
13.2 処理の流れ 38
13.3 タグの設定ファイル：tagConfig.json 39
13.4 タグ作成機能 40
13.5 実装 43
13.6 BJgit の出力例 44

14 性能評価実験 45
14.1 概要 45
14.2 実験設計 45
14.3 実験結果 46
14.4 考察 46

15 被験者実験 48
15.1 概要 48

15.2 被験者と実験対象リポジトリ 48
15.3 実験タスク 48
15.4 実験の流れ 49
15.5 アンケート評価 49
15.6 実験結果 50
15.7 追加インタビュー 51
15.8 考察 51

16 妥当性の脅威 53

17 関連研究 54

18 おわりに 55

第 IV 部 結論 56

謝辞 57

参考文献 58

図目次

1 MJgit の研究動機 7
2 MJgit における git-diff コマンドの処理の流れ 10
3 AST ベースのコード検索手法 10
4 MJgit における git-log コマンドの処理の流れ 13
5 git-diff コマンド出力の比較 14
6 git-log コマンド出力の比較 15
7 git-diff に対する実行時間の比較 19
8 実行時間の増加率と Java ファイル数の相関 19
9 git-log に対する実行時間の比較 20
10 git-log で出力されたログ数の比較 20
11 被験者の熟練度 23
12 タスク完了時間の比較 25
13 個々のタスクに対する完了時間の比較 26

14 アンケート結果 28
15 BJgit の研究動機 36
16 BJgit における git-log コマンドの処理の流れ 39
17 タグ [Huge-Add-Change] の実装例 42
18 従来の git-log の出力例 43
19 BJgit の git-log の出力例 43
20 各プロジェクトにおける BJgit の実行時間と速度増加率 46
21 アンケート結果 49

表目次

1 拡張対象となる Git コマンド 11
2 MJgit の検索クエリと拡張コマンド 13
3 実験対象プロジェクトの概要 17
4 比較した 3 つのコマンド 18
5 タスクの実行順序と MJgit 有無の割り当て（一部） 24
6 被験者ごとのタスク完了時間の比較 26
7 BJgit に実装されているプリセットタグ 42
8 比較した 5 つの実験設定 45

第 I 部

序論

1　研究の背景と目的

Git や SVN に代表され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で利用されている [1]．これら版管理システムを操作しリポジトリ内のソースコードの開発履歴を理解することで，効率的な開発の管理が可能である [2]．開発履歴の理解により，なぜコードが変更されたのか [3]，いつバグが混入したのか [4]，誰をデバッグ担当にすべきか [5]，といった開発プロジェクトにおける様々な疑問を解消することが可能となる．

一般的な版管理システムでは，diff や log などの履歴操作機能がサポートされている．例えば，git-diff コマンドを用いれば，最新リビジョンとその直前リビジョンの差分を diff 形式のフォーマットで取得できる．git-log コマンドを用いれば，リポジトリに保存されている開発履歴をコミッターの情報や，コミットされた日付，変更されたファイル名などと共に取得できる．ユーザーはこれらのコミット情報に対して版管理システムでサポートされている検索手法を用いて出力を絞り込むことができ，特定のコミッターによるコミットのみを表示するなどの操作が可能である．また，これらのコマンドはテキストデータに対する汎用ユーティリティとして設計されており，正規表現に基づいた強力なパターンマッチングを利用することも可能である [6].

しかしながら，これらの履歴情報を取得するコマンドには二つの問題点がある．一つ目は，正規表現を適用することができるなどの高い汎用性と引き換えに，ソースコードの構文情報やその意味に基づいた操作を行うことはできないことである．二つ目は，基本的にサポートされているログに対する検索方法では，検索にヒットするログのみが出力され，その他のログが非表示にされることである．これは，開発の全体像を掴みたい時などには不適切な手法であると言える．これらの問題は，既存の版管理システムがユーザーの関心の広さに対応した情報の提供方法をサポートしていないことが原因で引き起こされる．

関心の広さとは，その時ユーザーが関心を持っている情報がどの程度具体的かで決まる．例えば，バグ修正の最中に特定のメソッドや変数がどのように変更されたのかを知りたい時などは関心の対象となっている情報が局所的であり，関心は狭いと言える．このような時は．例えば設定ファイルの追加などの関心の外にある情報を取り去って，ユーザーの欲している情報に特化した出力を提供することで，ユーザーの履歴理解の支援をすることができる．

対して，ある開発者が新規にあるプロジェクトに参加する場合を考える．このような場合に，それまでの開発の様子を，誰が大きな変更を加えたかなどの特徴を把握しつつ全体像を捉えたい時などは，関

心の対象となっている情報がリポジトリ全体に横断しているので，関心は広いと言える．このような時は検索などによって排他的に情報を消去するのではなく，特徴を表現する情報を付加することで，ユーザーは開発の全体像と特徴を同時に掴むことができ，効率的に履歴を理解することができる．

そこで，本研究ではユーザーの関心の広さに応じて履歴理解を支援する手法を提案する．

第一に，関心が狭い時に履歴理解を支援する手法として MJgit を提案した．この手法は，git-diff や git-log などの開発履歴を取得するコマンドに対してソースコード検索を組み合わせた手法である．これによって，ユーザーが求めている情報をさらに詳細に取得し，履歴理解を支援することを目的としている．つまり，情報を絞り込んで取得することに特化した手法となる．詳細は第Ⅱ部で説明する．

第二に，関心が広い時に履歴理解を支援する手法として BJgit を提案した．この手法は，git-log などの開発ログを取得するコマンドに対して，あらかじめ設定してあるタグの条件に基づいてコミットログにタグを付けることができるようにした手法である．設定された特徴を持つログをタグによってハイライトし，特徴を掴みやすくすることで履歴理解を支援することを目的としている．MJgit とは逆に，情報を絞り込まず，特徴を表現した情報を付加することに特化した手法となる．詳細は第Ⅲ部で説明する．

2 用語

2.1 版管理システム

版管理システムとは，ファイルに対して「いつ」「誰が」「何を変更したか」などの情報を記録し，過去のある時点の状態を復元したり変更内容の差分を表示できるようにするシステムのことある．主としてソフトウェア開発におけるソースコードの管理などに使われている．版管理システムはほとんどのソフトウェア開発プロジェクトに導入されている．

2.2 JDT

JDT(Java Development Tools) とは，Eclipse で Java 開発を行うためのプラグインであり，Eclipse Foundation によってオープンソースソフトウェアとして提供されている．[7]

2.3 AST(抽象構文木)

抽象構文木 (Abstract Syntax Tree) とは，構文情報を木構造で表したソースコードの中間表現である．JDT では，Java で書かれたソースコードを AST に変換し保持することができる．この AST を辿ることによって，ソースコード中の必要な情報を得ることが可能である．

第 II 部

MJgit

3 はじめに

多くのプログラミング言語では，単一のソースコードファイルはコードの本質たる実行ステートメントだけではなく，様々な種類の記述（コメントやアノテーション，コピーライトなど）で構成されている．中でもコメントやコピーライトは自然言語で記述されており，これがテキストベースの検索に対してノイズになることがある．例えば，ある開発者がコードクローン [8] によって引き起こされるバグ伝搬 [9] をチェックするために，テキストベースの操作を用いて `if` 文を検索している場合を考える．この場合，実行ステートメントの `if` 文だけでなく，コメント内の “if” という文字列が検索に引っかかってしまい，結果として開発者に対するノイズになる．本提案手法は，ユーザーの関心が狭い状況での適用を想定しているので，ノイズとなり得る関心外の情報を除去してユーザーに提供したい．

提案手法である MJgit のキーアイデアは，開発履歴を操作する Git コマンド（git-diff，git-log，git-show など）に対してソースコード検索機能を統合することである．この 2 つの統合により，Git コマンドによるリビジョンのメタ情報（誰が，いつ，なぜ，など）の検索と，コード検索による構文を考慮した検索，という 2 つの観点を組み合わせた履歴操作が可能となる．これによって，より特定の情報に特化した情報を取得することができる．組み合わせたコマンドの具体例は以下の通りである．

```
$ git log --method=x --author=miwa
$ git log --method=x --since=2018-01-01
$ git log --method=x --grep='fix for'
$ git diff --method=x revA..revB
```

`--method` オプションは我々の提案するツールで拡張されたクエリである（ここでは実際の表記から簡略化している）．Git は標準で`--author` や`--since`，`--grep` などの検索をサポートしている．各クエリはそれぞれ開発履歴の「誰が」，「いつ」，「なぜ」を明らかにする．我々の手法は，このようなメタ情報を指定しながらメソッド `x` に焦点を合わせてコードの変更履歴を確認することができる．

本提案手法の目的は，上記のアイデアを実現し，関心の狭いユーザーに対する効率的な開発履歴の理解を実現することにある．第Ⅱ部では，この目標を達成するための Git クライアントの拡張ツール，MJgit を実装し評価する．この提案ツールは，指定されたメソッド名や変数名などの構文情報に基づいて Java ファイルの開発履歴を絞り込むことができる．拡張クエリが指定されていない場合，MJgit は拡張対象である Git クライアントの Java 実装である JGit と同じように動作するため，MJgit は JGit に対

して完全な後方互換性を持つ．MJgit の評価実験として，実際のソフトウェアリポジトリを用いた性能評価実験，及び有用性を確認するための被験者実験を実施した．実験結果により，実行速度は低下するが，その低下度合いは実用的な範囲内であることに加え，MJgit の有用性を確認できた．

以降第Ⅱ部では，4 節で研究動機として，従来の版管理システムの問題点を述べる．5 節では提案手法について説明し，6 節では性能評価実験を，7 節では被験者実験を行い，それぞれの考察を述べる．また，8 節では妥当性の脅威について述べ，9 節では関連研究について述べる．最後に，10 節で本提案手法 MJgit についてまとめ，今後の課題について述べる．

4 研究動機

4.1 例 1 あるメソッドの開発履歴の理解

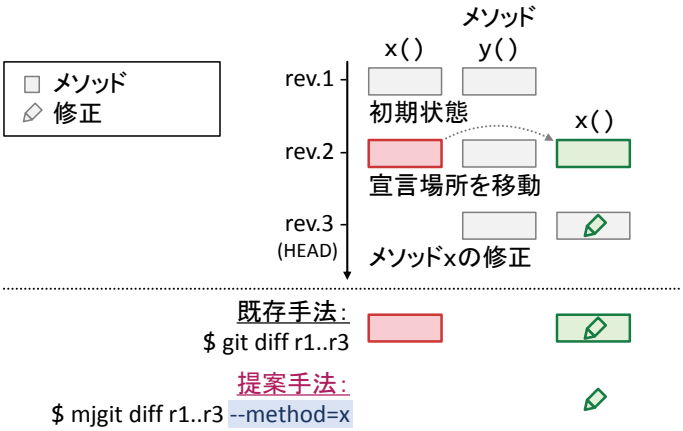
図 1(a) はメソッド x と y の 2 つのメソッドの開発履歴を表している．初期状態として，リビジョン 1 の時点で 2 つのメソッド x と y が図のような順番で宣言されているとする．リビジョン 2 で，メソッド x の宣言がファイルの最後（メソッド y の後ろ）に移動されている．リビジョン 3 でメソッド x の中身が修正された．ここで，ある開発者がメソッド x の開発履歴を理解しようとしている．

既存手法： 開発履歴を確認する簡単な方法としては，git-diff を用いてリビジョン 1 から 3 までの差分を見る方法であり，これは Git クライアントをインストールしていれば，追加のインストールなしに使用することができる．しかし，このようなテキストベースのコマンドは，ソースコードの構文情報やフローを考慮していない．したがって，これらのコマンドは，プログラムの機能自体に影響がなく広範囲に行われた変更（コードのフォーマット変更やメソッドの並べ替えなど）がある場合，開発者にとって不要な情報を大量に提供してしまう．図の例の場合，git-diff を使用するとメソッド x は削除されて，かつ追加されたように見えてしまうが，実際は移動しただけである．ペンのアイコンで表されている本質的な修正は，メソッド移動のノイズに埋もれてしまう．

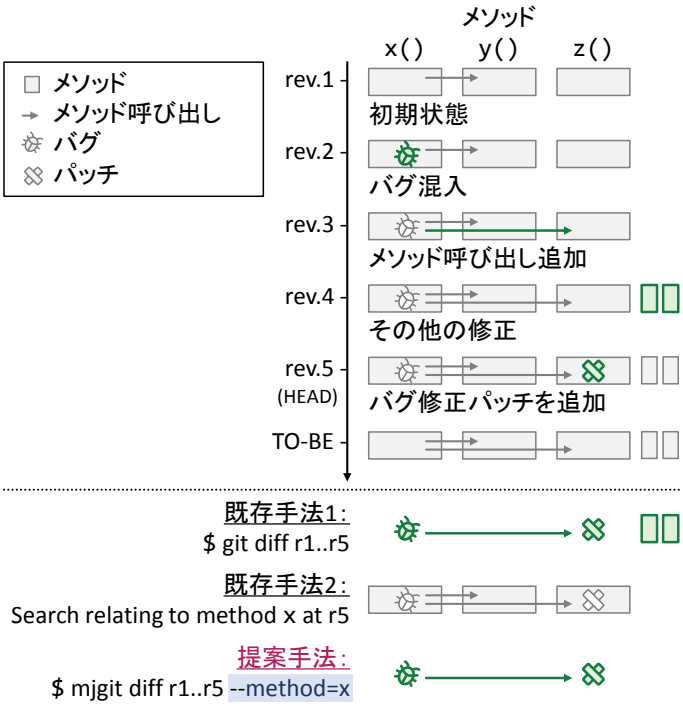
提案手法： 提案手法により，開発者はリポジトリから履歴を取得する際に構文情報を指定できる．この場合，メソッド名 x を git-diff に指定することで，表面的な変更やプログラムの本質に関わらない変更をフィルタリングすることができる．結果として，図のようにメソッドの移動による出力を排除し，本質的な修正部分のみを開発者に提示することが可能である．

4.2 例 2 バグ修正と一時的なパッチの除去

図 1(b) はより実用的かつ複雑な状況を表している．初期状態として，リビジョン 1 の時点で 3 つのメソッド x，y，z が宣言されており，メソッド x はメソッド y を呼び出している．リビジョン 2 でメソッド x にバグが混入された．次にリビジョン 3 でバグを含むメソッド x がメソッド z を呼び出すように変更された．さらにリビジョン 4 でメソッド x，y，z 以外の場所でいくつかの変更が行われた．そしてリビジョン 5 で開発者がバグに気付き，一時的な修正パッチをメソッド z に適用した．このパッチはメソッド x から渡された引数の null チェックなどの一時的な解決方法である．ここで，ある開発者がバグを根本から修正しようとしている．さらに，一時的なパッチも除去する必要がある．



(a) 例 1 あるメソッドの開発履歴の理解



(b) 例 2 バグ修正と一時的なパッチの除去

図 1　MJgit の研究動機

既存手法 **1**：ここに git-diff を実行すると，リビジョン 1 から 5 までのテキスト差分を得ることができる．しかしこのコマンドの場合，最初の例と同様に，リビジョン 4 で行われた，バグ修正とは無関係の修正がノイズとなる．通常，版管理システムを利用するにあたって，コミットを小さく分割することが推奨されている [10] が，多くの修正が単一のコミットにまとめられてしまうという状況がしばしば起こる [11]．このような絡み合ったコミットも同様のノイズを生み出す要因となりうる．

既存手法 **2**： この例のような状況では，ソースコード検索 [12–18] や変更波及解析 [19–21] が有効なアプローチとして知られている．変更波及解析は構文木またはフローグラフに基づいて依存関係を追跡することで，修正が生み出す潜在的な影響範囲を特定する．開発者は，「メソッド x のみに関連するメソッド」という検索をリビジョン 5 の時点のソースコードに適用することで，関心のあるメソッドを見つけることができる．ただし，これらの手法では開発履歴をサポートできないという制限がある．従って出力結果にはリビジョン 1 から 5 で変更されていないメソッド y が含まれてしまう．メソッド y にバグが存在せず，開発者の関心外にある場合，この情報はノイズになる可能性がある．

提案手法： 提案ツール MJgit を使用すると，開発者はリビジョン 1 から 5 での変更点かつメソッド x に関連する差分のみを取得することができる．これによって，図のようにノイズのない開発者の関心部分だけが出力される．この例では，図のようにバグとパッチのみを抽出可能であり，プログラムの変更理解の支援，及びそのメンテナンス作業の手助けとなる．

5 提案ツール：MJgit

5.1 概要

MJgit の目的は，Git コマンドに対してソースコード検索機能を統合することにより，関心の狭いユーザーに対して効率的な開発履歴の理解を支援することである．MJgit の基本的なアイデアは，Git における履歴情報を取得するコマンド（git-diff や git-log 等）に対して，ソースコードの特定の部分を絞り込む拡張クエリの指定を可能とする，という点にある．このアイデア実現にあたっては，どのようにコード検索を実現するか，どの Git コマンドを拡張するか，の 2 点が重要となる．本節では，まず MJgit 利用時の処理の流れについて概説し（5.2 節），コード検索技術（5.3 節），及び拡張対象となる Git コマンド（5.4 節）について説明する．

5.2 処理の流れ

git-diff コマンドを対象とした時の MJgit の処理の流れを図 2 に示す．図の各手順について以下で説明する．

1. まず開発者は 2 つのリビジョン，A と B 間の差分を取得するために git-diff を実行する．この時，MJgit で追加された拡張クエリを指定する．
2. 次に，MJgit は各リビジョンのソースコードを取得する．
3. 任意のコード検索技術により，指定クエリに該当するソースコード箇所を検索する．提案手法はコード検索技術には依存しておらず，任意の検索技術を用いることができる．MJgit で採用した具体的なコード検索技術については 5.3 節で述べる．
4. コード検索で該当した箇所以外の全ての行を，ソースコードファイル中から削除する．このように，テキスト diff 計算の前にコード検索を適用し，該当しない部分を削除したテキストファイルを生成して差分計算することにより，様々な履歴取得コマンドとコード検索の組み合わせが実現できる．なお，テキスト diff の計算には標準 Git による Longest Common Subsequence アルゴリズム [6] を用いる．
5. 最後に，クエリ該当部分のみが記述されたソースコードファイルに対し，diff アルゴリズムを適用することで，必要のない情報を排除した差分を開発者に提示する．

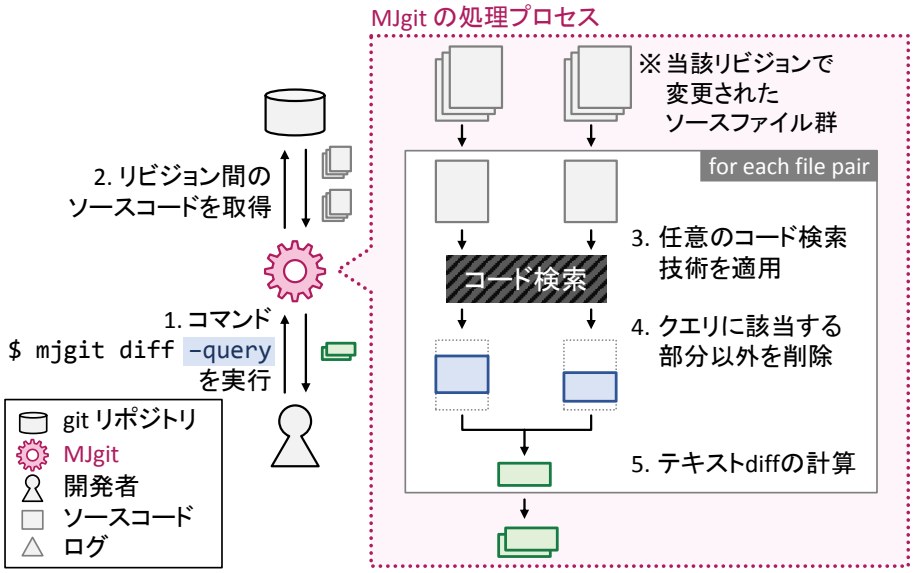


図 2　MJgit における git-diff コマンドの処理の流れ

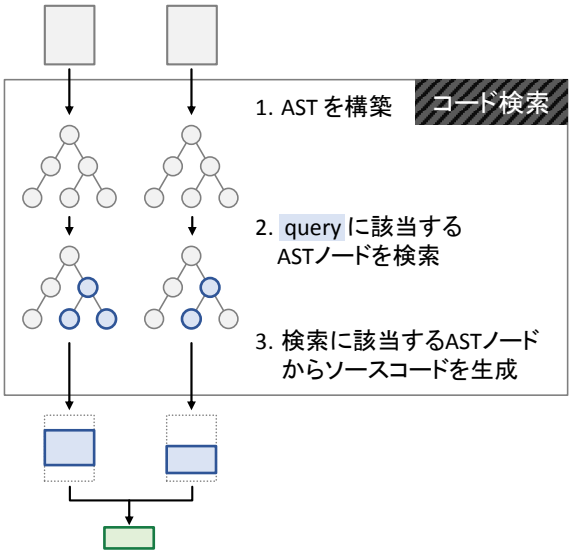


図 3　AST ベースのコード検索手法

5.3　コード検索技術

単一のソースコードから特定のプログラムスニペットやステートメントを検索する方法はいくつか存在する [21–23]．同様に，大量のソースコードファイルを格納するデータベースから特定のソースコードファイルを検索するためにも多くのアプローチが提案されている [24, 25]．いずれの手法も，提案手法におけるコード検索を実現する手法として適用することが可能である．

本論文では，具体的なコード検索技術として Eclipse JDT（Java Development Tools）の AST を用いた静的コードスライシングを採用する．この方法では，生成された AST のルートから探索を始め，クエリに該当する AST ノードを順に発見していくことで，クエリの該当箇所を検索する．例えば `method=x` というクエリの場合，以下 2 つの AST ノードがクエリの該当箇所となる．

- ノード名が `x` である `SimpleName` ノードを持つ `MethodDeclaration` ノード（`x` の宣言）
- ノード名が `x` である `SimpleName` ノードを持つ `MethodInvocation` ノード（`x` の呼び出しを含む文）

また，AST の各ノードには AST 生成の元となったソースコードファイルへの行番号が対応付いている．この行番号を元に，ソースコードファイルのどの行がクエリに該当するかを探し出す．最終的にクエリに該当した行以外の全ての行を削除し，テキスト diff への入力とする．

AST を用いた静的コードスライシングの処理の流れを図 3 に示す．この手法では，1. AST を構築する，2. クエリに該当する AST ノードを検索する，3. クエリに該当する AST ノードのみからテキストを生成する，の 3 手順で構成される．なお，コード検索は対象リビジョンで変更された Java ファイルに対してのみ適用される．

5.4 拡張対象となるコマンド

Git はリポジトリを管理・操作するための様々なコマンドをサポートしている．提案手法は履歴理解の支援が目的であるため，「過去のリビジョンに対して」「履歴を取得する」コマンドへの適用が適当である．この 2 つの性質を満たすコマンドを表 1 に示す．この表のコマンドは，それぞれ 1 つ以上のリビジョンからログやソースコードファイルを取得する．また，これらのコマンドは `since` や `author` などのリビジョン情報を取得するクエリをサポートしている．

表 1　拡張対象となる Git コマンド

コマンド名	概要	対象 リビジョン数
git-diff	リビジョン間の差分を取得する	2
git-show	リビジョンの詳細を取得する	1
git-log	リビジョンログを取得する	$\geq$1
git-blame	ファイルの最終更新を取得する	$\geq$1

なお，表に示す各コマンドは，コマンドごとにその出力の単位が異なる．git-diff ではリビジョン間のテキスト差分情報の集合が，git-log ではリビジョン情報（すなわちログ）の集合が出力の単位となる．このリビジョン情報は差分情報を内包する．git-show は，git リポジトリ内の 3 種類のオブジェクト（commit，blob, tree）の詳細を取得するコマンドであり，git-diff と git-log が要素の集合を取得するのに対し，git-show は単一の要素を取得する．

各コマンドで出力単位が異なるのに対し，MJgit によるソースコードの何を絞り込むかという「クエリの意味」は，どの Git コマンドでも共通である．例えば，`method=x` というクエリを指定した場合，メソッド `x` の宣言とメソッド `x` の呼び出しを含む文，の 2 つに関心を絞り込むという意図となる．そのクエリ指定により得られる出力は，どの Git コマンドと組み合わせるかによって変わるが，クエリの解釈そのものは一貫性を保つ．

このクエリ解釈の一貫性は，MJgit のクエリ検索が差分情報の生成元となるソースコードのみに対して働くことに起因する．git-diff では処理対象が差分情報そのものであり，クエリ検索はその差分情報に直接適用できる．git-log によりリビジョン情報の集合を取得する場合でも，個々のリビジョン情報が内包する差分情報にクエリ検索が適用される．よって，クエリの解釈は git-diff と全く同じ内容となる．

git-diff 以外のコマンドへの拡張の一例として，git-log コマンドの処理の流れを図 4 に示す．git-log ではリビジョン情報の集合に対して処理が行われるが，コード検索自体はリビジョン情報が内包するソースコード群に対して適用される．よって，処理の各手順は先に述べた図 2 と同等である．

このような履歴に対して情報を取得するコマンドでは，履歴情報に内包されるソースコードにコード検索技術を適用することができる．よって，提案手法により拡張されるソースコードの構造に基づいた検索クエリは，標準 Git に実装されている日付や著者名に対するクエリと同様に，クエリ解釈の一貫性を保つ．表 1 に示した履歴に対して情報を取得するコマンドでは，履歴情報に内包されるソースコードにコード検索技術を適用できるため，MJgit の拡張対象は同表に示す 4 つのコマンドとした．

5.5 実装

MJgit の実装は，Java ベースのオープンソースの Git クライアント，JGit [*1] を拡張して行われた．MJgit のソースコードは GitHub 上で公開されている[*2]．

[*1] `https://eclipse.org/jgit/`

[*2] `https://github.com/kusumotolab/m-sasaki_MJgit`

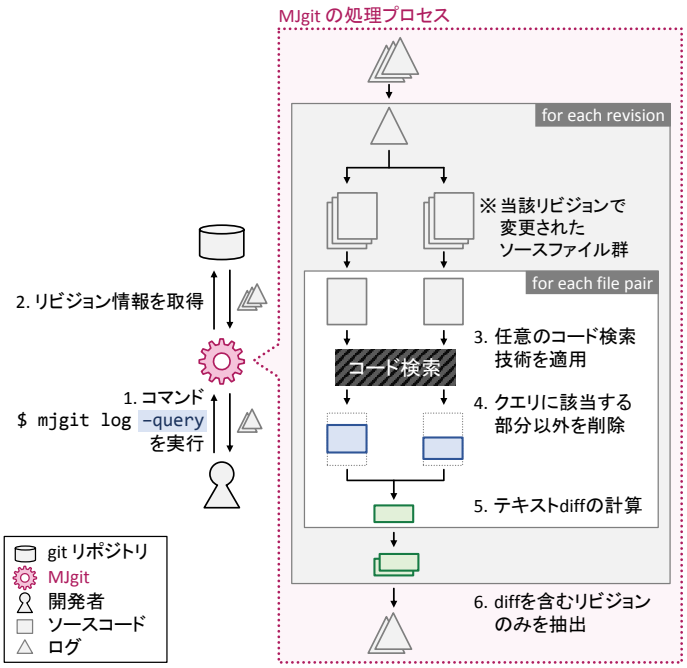


図 4　MJgit における git-log コマンドの処理の流れ

現在の MJgit がサポートする検索クエリ，及び拡張コマンドの一覧は表 2 の通りである．なお，前節で示した，表 1 の拡張対象コマンドのうち，git-blame のみ現在未実装となっている．MJgit は現在，メソッド名と変数名を指定するクエリをサポートしている．さらに，`exec-statement`，`comment`，`javadoc`, `annotation` というクエリもサポートしており，指定されたタイプのプログラム文だけを指定することができる．例えば，実行ステートメントだけの差分やリビジョンログは `exec-statement` クエリで取得することができる．コード検索は，これらの構文情報についてのパラメータを指定することで実行される．また，現在の MJgit では，クエリは git-diff と git-show，git-log で使用できる．検索クエリが指定されていない場合，ツールはオリジナルの JGit クライアントと同じ動作をする．

表 2　MJgit の検索クエリと拡張コマンド

拡張コマンド	git-log, git-show, git-diff
指定可能なクエリ	実行ステートメント（`exec-statement`）
（クエリ指定方法）	コメント（`comment`）
	Javadoc（`javadoc`）
	アノテーション（`annotation`）
	メソッド名の指定（`method=`*`method_name`*）
	変数名の指定（`variable=`*`var_name`*）

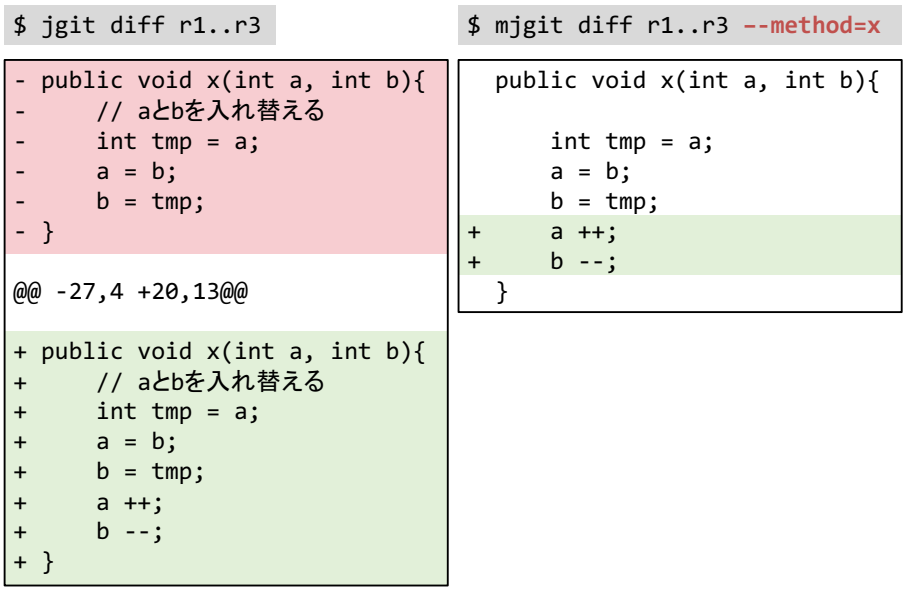


図 5　git-diff コマンド出力の比較

5.6　各コマンドの振る舞いと出力例

ここでは，2 つの特定リビジョンを対象とする git-diff と git-show，及び複数のリビジョンを対象とする git-log のそれぞれの出力例を説明する．

5.6.1　git-diff と git-show

git-diff と git-show は 2 つの特定リビジョン間のソースコードの差分を確認するコマンドである．このコマンドにソースコード検索を組み合わせることで，出力される差分を使用者の指定した構文情報だけに絞り込むことができる．

例えば，`method=x` を指定すると，`x` の宣言とその呼び出しを含む文が検索され，それ以外の差分は省略される．変更されたファイルにコンパイルエラーが存在し，AST が構築できない場合，検索クエリは無視され，通常のテキストベースな diff による差分が出力される．

具体的な出力結果を図 5 に示す．左の出力がクエリを指定しないオリジナルの出力結果であり，右の出力は，左の出力に `method=x` クエリを指定することで情報を絞り込んだ際の結果である．この図は 4 節で述べた一つ目の例（図 1(a)）に対応しており，メソッド `x` の移動と修正が混在しているケースを表す．`method=x` クエリを指定することで移動がフィルタリングされ，かつコメントが除外され本質的な修正のみが提示されている．

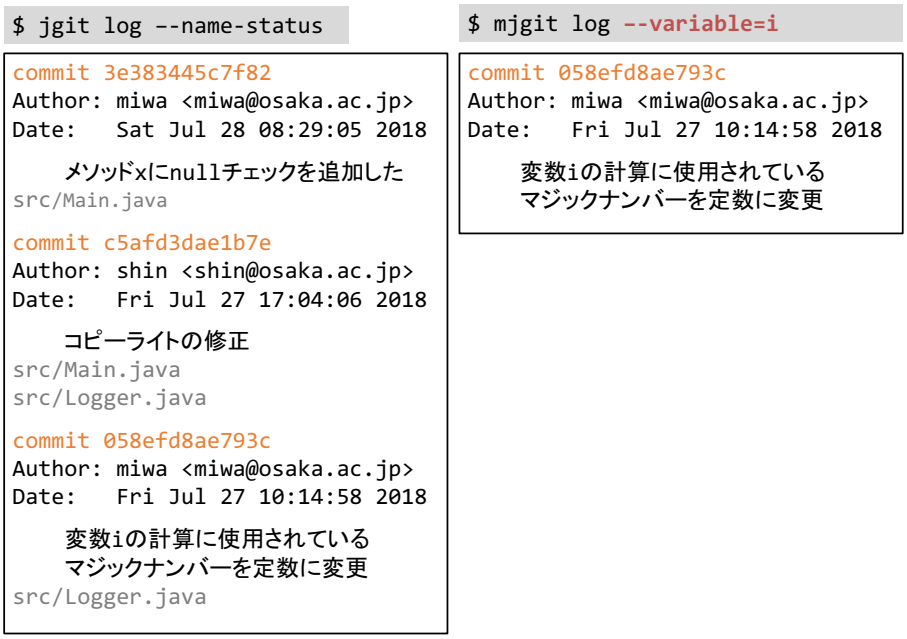


図 6　git-log コマンド出力の比較

5.6.2　git-log

git-log はリビジョンのログを取得するコマンドである．このコマンドにソースコード検索を組み合わせることで，複数のログの中から指定した構文情報を変更したリビジョンのみを絞り込むことができる．

例えば，`variable=i` を指定すると，変数 `i` の宣言やそれが用いられている文が変更されたリビジョンが検索され，`i` に関連しないログは全て省略される．なお，現在の実装ではクエリ `variable` に用いることが可能な変数の種類は局所変数のみであり，フィールドは対象外である．リビジョンに構文エラーを含むファイルが存在している場合，AST が構築できず，指定クエリが変更されたかどうかを判断できない．よって，そのログは構文エラーを含むという警告文とともに出力される．

具体的な出力結果を図 6 に示す．左の出力が提案クエリを指定しないオリジナルの出力結果であり，右の出力は，`variable=i` クエリを指定することで情報を絞り込んだ出力である．なお，左のコマンドでは各リビジョンで編集したファイル名が出力される，Git の標準オプション`--name-status` を指定して実行している．

この例では，3 つのログが存在するリポジトリに対して git-log を実行している．`variable=i` クエリを指定することで，`i` が変更されたログだけが出力されている．

オリジナルの git-log には，ファイル名や編集者を指定してログをフィルタする機能が設けられている．ここで，ファイル `src/Logger.java` の変数 `i` の履歴を確認したい開発者が，標準 Git コマンドを用いてファイル名 `src/Logger.java` を指定したとする．しかし，このファイルは 2 つ目のリビ

ジョン `c5af3d` でコピーライトの変更が行われており，ノイズとなりうる．ここで MJgit で変数 `i` というクエリを与えることで，この開発者の関心にマッチした検索が可能となる．

6 性能評価実験

6.1 概要

本実験の目的は，MJgit による拡張クエリの実行性能が実用的かどうかを確認することである．MJgit では指定されたクエリを処理するために，抽象構文木の構築と該当する部分木の検索という処理が必要である．この追加処理が，Git 利用時における実行性能に対してどの程度影響を与えるかを確認する．

6.2 実験設計

2 つの Java のオープンソースリポジトリに対して MJgit を実行し，その実行速度や出力されるログの数の減少率についてオリジナルの JGit と比較する．より具体的には，検索クエリを指定した git-diff と git-log の実行時間をオリジナルの JGit の実行時間と比較する．git-diff は，オープンソースリポジトリである JUnit4 と Log4j の全ての隣接リビジョンのペアに対して実行した．git-log はリポジトリ内の全リビジョンに対して実行した．いずれのコマンドも 10 回ずつ実行し，その平均を計測値とする．

対象プロジェクトの概要を表 3 に示す．両プロジェクト共に 1,000 以上のリビジョンを持っており，15 年以上開発され続けている．git-diff では，検索クエリは Java ファイルが変更されたリビジョンのみに作用するため，Java ファイルが変更されていないリビジョンは実験対象外とした．同様に，リビジョンにコンパイルエラーが含まれている場合のリビジョンも実験対象から除外した．

MJgit では様々な検索クエリを組み合わせることができる．この実験では，表 4 に示す 3 つのコマンドを比較した．コマンドはそれぞれ *jgit*, mj_{method}, $mj_{exec\text{-}stmt}$ とラベル付けする．*jgit* はオリジナルの JGit のコマンドを表す．git-diff では，mj_{method} はメソッド main のみの差分を出力し，$mj_{exec\text{-}stmt}$ は実行ステートメントの差分のみを出力し，コメントや Javadoc，アノテーションなどの実行分以外の変更をフィルタリングする．git-log では，mj_{method} はメソッド main が変更されたログのみを出力し，$mj_{exec\text{-}stmt}$ は実行ステートメントが変更されたログのみを出力する．

表 3　実験対象プロジェクトの概要

	JUnit4	Log4j
プロジェクト開始月	2000/12	2000/11
全リビジョン数	2,187	3,275
git-diff 対象のリビジョンペア数	900	1,891
最新リビジョンでの Java ファイル数	443	309

jgit コマンドは，性能比較におけるベースライン計測のために設定した実験タスクである．本実験では，このベースラインに対して mj_{method} と $mj_{exec\text{-}stmt}$ の性能低下の程度を確認する．さらに，mj_{method} と $mj_{exec\text{-}stmt}$ はクエリ検索の該当のしやすさが性能に与える影響を調べるために設定したタスクであり，mj_{method} はクエリ検索に該当する部分が比較的少なく，$mj_{exec\text{-}stmt}$ は逆に該当部分が多くなるクエリである．

6.3 実験結果

6.3.1 git-diff の結果

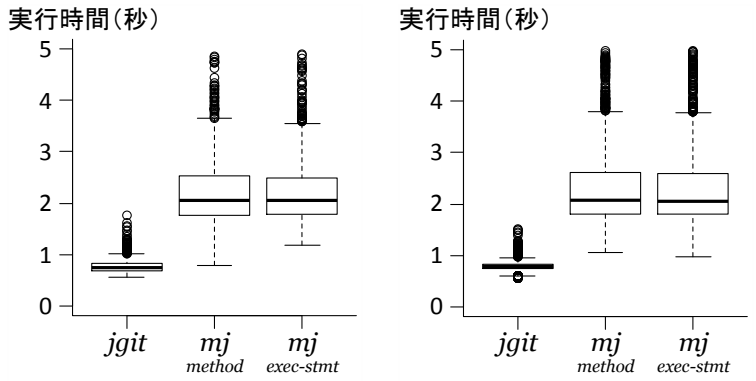
git-diff の実行結果を図 7 に示す．3 つの箱ひげ図はそれぞれのクエリを指定した時の git-diff の実行時間を表している．ただし，実行時間が 5 秒を超えていた 2 つの外れ値は図から除外している．

オリジナルの JGit では，全ての git-diff が 2 秒以内に実行され，実行時間の中央値は 0.8 秒であった．この値を性能比較のベースラインとする．MJgit で拡張クエリと共に git-diff（mj_{method} と $mj_{exec\text{-}stmt}$）を実行した場合，実行時間の中央値は 0.8 秒から 2 秒へと増加した．なお，mj_{method} と $mj_{exec\text{-}stmt}$ の間にはほとんど違いはなかった．さらに，2 つのプロジェクトでの結果を比較しても，両プロジェクト間には大きな違いはなく，結果にはほぼ同じ傾向が見られた．

性能低下の理由を明らかにするために，実行時間の増加率と Java ファイル数の相関関係を調べた．ファイル数との相関を調べた理由は，図 2 に示す通り，MJgit による追加処理が複数ファイル全てに適用されるためであり，この部分が性能低下要因になっているとの仮説に基づく．調査の結果を図 8 に示す．グラフの各プロットは 1 つのリビジョンを示している．x 軸は各リビジョンでコミットされた Java ファイルの数を表し，y 軸は実行時間の増加率を表している．実行時間の増加率は，「$mj_{exec\text{-}stmt}$ の実行時間/*jgit* の実行時間 」で計算されている．相関係数は JUnit4 と Log4j について 0.57 と 0.77 であり，有意であった．

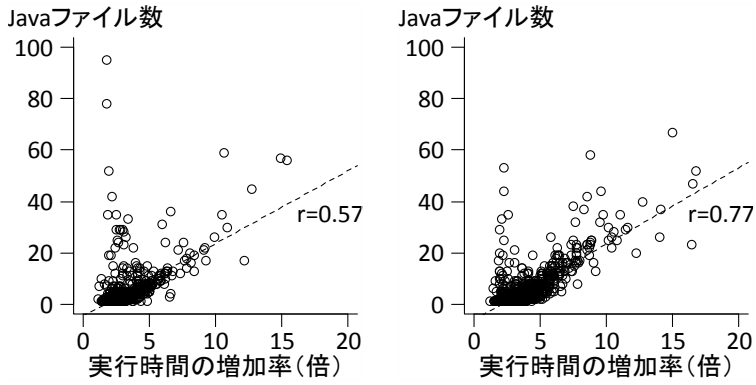
表 4　比較した 3 つのコマンド

ラベル	実行したコマンド
jgit	`$ jgit diff A..B`
	`$ jgit log`
mj_{method}	`$ mjgit diff A..B --method=main`
	`$ mjgit log --method=main`
$mj_{exec\text{-}stmt}$	`$ mjgit diff A..B --exec-statement`
	`$ mjgit log --exec-statement`



(a) JUnit4　　(b) Log4j

図 7　git-diff に対する実行時間の比較



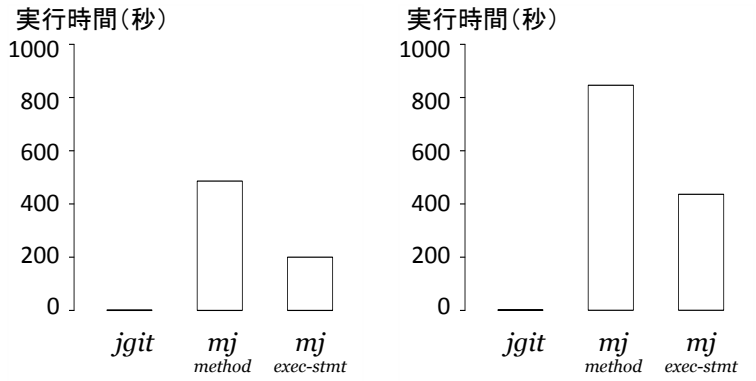
(a) JUnit4　　(b) Log4j

図 8　実行時間の増加率と Java ファイル数の相関

6.3.2　git-log の結果

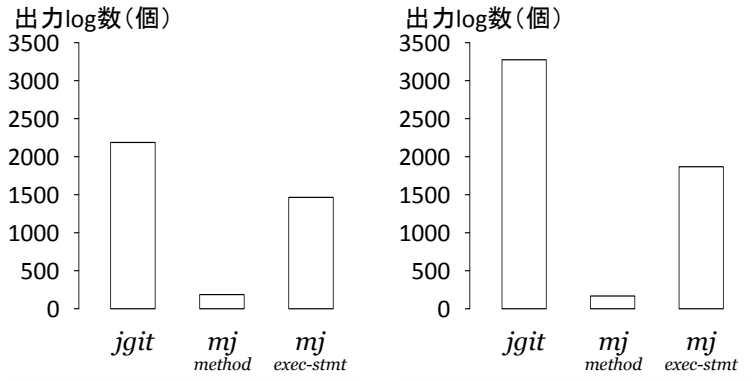
git-log の実行時間の比較を図 9 に示す．*jgit* での実行時間は両プロジェクト共に 1.2 秒と極めて短い時間で完了できた一方，提案手法では最低でも 200 秒であり少なくとも数百倍の時間増加が確認できた．特に時間増加が大きいケースは，Log4j に対して mj_{method} を実行した際の 844 秒である．

拡張クエリによる出力ログ数の削減状況についての結果を図 10 に示す．mj_{method} のような特定メソッドのみをフィルタリングする絞り込み効果の大きいクエリでは，大幅なログ数の削減効果が確認できる．さらに実行時間の結果を表す図 9 と比較すると，このようなクエリほど実行時間が増加する傾向が読み取れる．



(a) JUnit4 (b) Log4j

図 9　git-log に対する実行時間の比較



(a) JUnit4 (b) Log4j

図 10　git-log で出力されたログ数の比較

6.4　考察

ほぼ全てのリビジョンに対して，git-diff は 5 秒以内に完了することができた．git-diff に対する MJgit の拡張クエリの利用は実用的な範囲であると考えられる．

また，実行時間の増加とファイル数の相関から，Java のファイル数が多いと実行パフォーマンスが低下することが確認できた．そして mj_{method} と $mj_{exec\text{-}stmt}$ には大きな差はないことから，実行時間増加の主たる要因はコード検索ではなく，AST の構築だと考えられる．これを改善する方法としては，一度構築した AST をキャッシュのように保存する方法が考えられる．ソースコードの AST を初めて構築した際に，その AST をキャッシュとして保存しておけば，git-log などの何度も AST を構築しなければならないコマンドの速度改善に繋がると考えられる．

その一方で，git-log に対しては無視できない大幅な性能低下が確認できた．特に，絞り込み効果の大

きいクエリほど著しく性能が低下する傾向にあった．この理由については，git-log はファイルとリビジョンの全ての組み合わせに対して，コード検索が適用される点にあると考えられる．この性質により，git-diff での性能低下割合にリビジョン数を乗算する形で性能悪化が現れたと推測できる．上記で述べた AST 構築の工夫や，キャッシュ等の組み合わせによる性能改善が必須であるといえる．

ただし，この実験ではリポジトリ内の全て（2100 個以上）のログを得る，という状況における性能を比較した．しかし，実際の開発過程において過去のログを確認する場合，一部の直近のリビジョンのみに関心があることが多い．この場合の性能低下は全ログの場合と比べて極めて小さく，実際の開発者がこの性能低下をどのように感じるかは被験者実験による調査が必要である．

7 被験者実験

7.1 概要

この実験の目的は，Git を用いた開発履歴の理解というタスクに対して，MJgit がどの程度有用であるかを確認することである．このために，被験者 16 名に対して複数のコードリーディングタスクを与え，MJgit あり/なしの状況での履歴理解に要する時間を比較する．さらに定性評価として，インタビューにより MJgit を使用した印象やコメントを収集し分析する．履歴理解のコードリーディングタスクは，性能評価実験でも用いた Log4j のリポジトリから著者らが作成した．

7.2 被験者

被験者は大阪大学大学院情報科学研究科に所属する教員 1 名，同研究科所属の修士の学生 12 名，大阪大学基礎工学部の学部生 3 人の計 16 名を対象とした．さらに，被験者は後述するタスク難易度の差を除外するために，X と Y の 2 つのグループに分割した．

被験者の Git と Java に対する熟練度を図 11 に示す．全ての被験者はコードリーディングの対象となる Java 言語の文法を熟知している．一部の Git に習熟していない被験者には，練習タスクの最中に実験に用いる最低限の Git コマンドの利用方法を習得してもらった．

7.3 実験タスク

実験タスクは，Git リポジトリに対する開発履歴の理解である．実施した 2 種類のタスクは以下の通りである．

$Task_A$: ある期間中のリビジョンから，特定のメソッド/変数がどのリビジョンで変更されたかを見つける．

$Task_B$: $Task_A$ で発見したリビジョンで，特定のメソッド/変数がどのように変更されたかを理解する．

現実的には，開発履歴の理解という行為は「なぜ」「誰が」「どのファイル/メソッド/変数を」「どのように」といった様々な観点を内包しており，git-log や git-diff，さらには less や Web ブラウザといった様々なツールを組み合わせて行われる．本被験者実験では，「あるメソッド/変数」に着目しているという状況を想定し，そのメソッド/変数がどのリビジョンで変更されたか（$Task_A$）と，そのリビジョンでどのように変更されたか（$Task_B$）の 2 種類にタスクを分類した．

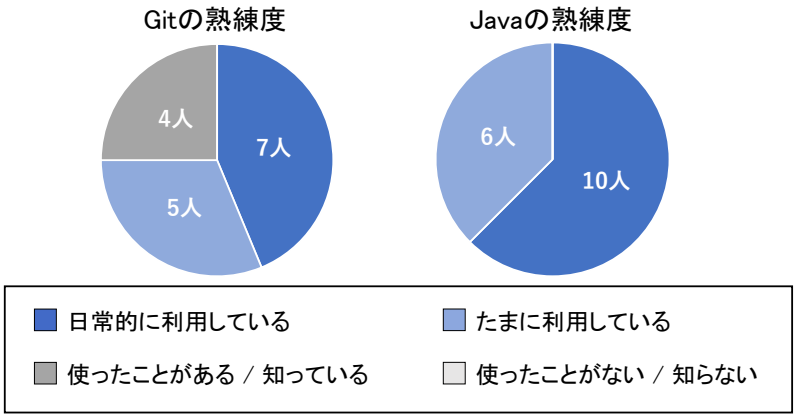


図 11　被験者の熟練度

$Task_A$ は履歴の中から対象のリビジョンを探し出すタスクであり，以下のコマンドの利用を想定している．

```
$  jgit log                 # 既存手法
$ mjgit log --method=x      # 提案手法
```

$Task_B$ は，$Task_A$ で発見したリビジョンでの具体的な変更内容をコードリーディングし，特定のメソッド/変数がどのように変更されたかを理解するタスクであり，以下のコマンドの利用を想定している．

```
$  jgit show                # 既存手法
$ mjgit show --method=x     # 提案手法
```

$Task_A$ は 2 ペア（4 タスク），$Task_B$ は 8 ペア（16 タスク）の計 10 ペアのタスクを用意した．各ペアはタスクの難易度や，コードの変更内容が似たもの同士になるように設定した．そのペアの片方で MJgit を利用可能（ツールあり）とし，もう片方で利用不可（ツールなし）と設定し，ペアごとの結果を比較する．なお，全てのタスクは Log4j のリポジトリに記録された実際の変更履歴から，著者らが作成した．

7.4　タスクの実行順序とツール有無の割り当て

この実験では，MJgit への慣れやタスクへの慣れによる実験への影響が考えられる．この影響を排除するために，先ほど設定した各ペアの実行順序が隣合わないように，かつツールありタスクとツールなしタスクが交互になるようにタスクをシャッフルした．

表 5 に，シャッフルされたタスクの実行順とタスクの答えとなる変更内容の例を示す．タスク ID の数字はタスクの難易度を表しており，数字が同じタスクはペアであることを意味する．タスク ID の末

尾の a と b はペア間の識別子であり，被験者グループ X に対しては a のタスクで提案ツールあり，b のタスクで提案ツールなしとした．一方グループ Y に対しては，b でツールなし，a でツールありとした．

7.5 実施の流れ

被験者実験は以下の流れで実施した．

1. MJgit の機能の解説：MJgit の使用方法や振る舞いについて被験者に説明する．
2. タスクの練習：簡単な練習用タスクを実行してもらい，ツールの具体的な使い方と回答の方法を習得してもらう．
3. タスク実行：7.4 節で決定したタスクの実行順序通りに被験者にタスクを実行してもらう．グループ X/Y の被験者のいずれもツールあり/なしのタスクを交互に 20 回実施する．各タスク実施後は，回答用フォームにその作業時間と回答を記入する．
4. アンケート回答：全タスクを完了した後，オリジナルの JGit と比較した MJgit の印象を回答してもらう．

7.6 定性評価

全タスクの実施後，被験者に以下のアンケートを答えてもらい，ツールの有用性や課題を確認する．

- $Task_A$ で MJgit は使い易かったか（5 段階評価）
- $Task_B$ で MJgit は使い易かったか（5 段階評価）

表 5　タスクの実行順序と MJgit 有無の割り当て（一部）

ID	変更内容	実行順序	MJgit の有無 X	MJgit の有無 Y
1a	引数を追加	1	あり	なし
1b	引数を追加	4	なし	あり
2a	null チェック追加	5	あり	なし
2b	null チェック追加	2	なし	あり
3a	変更なし[*3]	3	あり	なし
3b	変更なし	6	なし	あり
...	...			

[*3] コードフォーマッタ等の変更により，一見変更されているように見えるがプログラム的に一切変更なしというケース．

- MJgit による速度低下をどう感じたか（5 段階評価）
- 今後 MJgit を使いたいか（5 段階評価）
- その他 MJgit についての自由記述

7.7 実験結果

7.7.1 タスク完了時間の結果

$Task_A$ と $Task_B$ における MJgit ありとなしのタスク完了時間の比較を図 12 に示す．縦軸はタスク完了に要した時間であり，横軸が $Task_A$（あるいは $Task_B$）に対するツールなしとありの結果を表す．なお，実験中に被験者からツールの動作について質問を受けたことで作業時間が大きく伸びた 1 ケースは，結果から除外することとした．

$Task_A$ と $Task_B$ いずれにおいても，完了に要する時間が削減されている．特に $Task_A$ では大幅な時間削減が確認でき，その中央値は 140 秒から 72 秒に減少（約 50% 削減）している．ウィルコクソンの順位和検定を用いて有意差を検定したところ，$Task_A$ で $p = 4.11 * 10^{-7}$，$Task_B$ で $p = 2.62 * 10^{-3}$ であり，共に優位な差が確認できた．

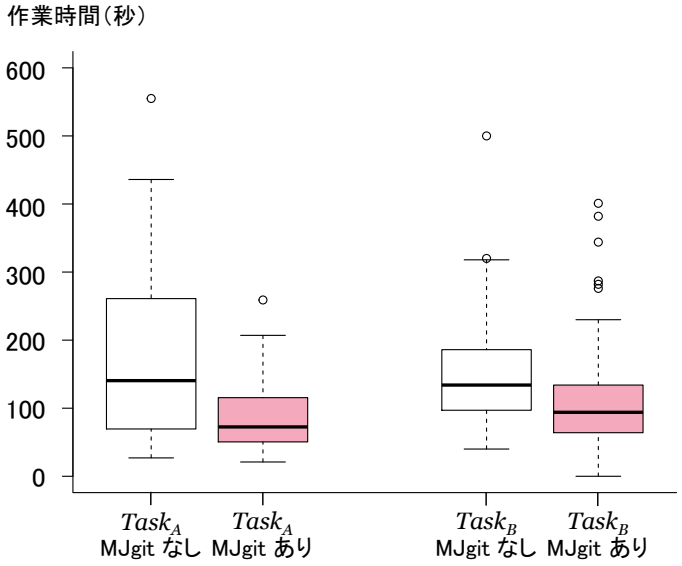


図 12　タスク完了時間の比較

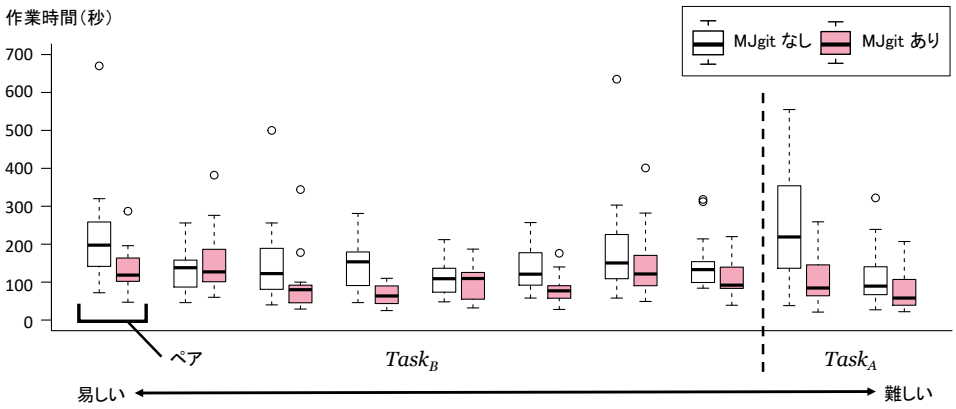


図 13　個々のタスクに対する完了時間の比較

表 6　被験者ごとのタスク完了時間の比較

被験者	グループ	MJgit なし 平均完了時間	MJgit あり 平均完了時間
1	X	143 秒	75 秒
2	X	192 秒	160 秒
3	X	107 秒	106 秒
4	X	160 秒	61 秒
5	X	213 秒	128 秒
6	X	99 秒	44 秒
7	X	152 秒	112 秒
8	X	104 秒	96 秒
9	Y	256 秒	149 秒
10	Y	131 秒	110 秒
11	Y	80 秒	49 秒
12	Y	144 秒	105 秒
13	Y	213 秒	175 秒
14	Y	229 秒	98 秒
15	Y	198 秒	156 秒
16	Y	141 秒	76 秒
平均		160 秒	106 秒

7.7.2 個々のタスクに対するタスク完了時間の結果

より詳細な分析として，個々のタスクに対する作業時間の結果を図 13 に示す．各箱ひげは個々のタスクを表しており，隣り合う 2 つの箱ひげが同難易度のタスクペアを意味している．白色の箱ひげが MJgit なし，ピンク色が MJgit ありを意味する．なお，個々のタスクペアは難易度で並び替えられており，右ほど難しいタスクである．$Task_A$ は複数のログから目的のログを探す内容であり，複数のコマンドの組み合わせが必要なため難易度が高い．

まず，ツールによる効果が大きかったケースとして，左から 3 番目のタスクがある．このタスクは，メソッド宣言順序の移動などによりメソッド全体が大きく変化したように見えるが，本質的な変更は一切ないというタスクである．このようなノイズが多いリビジョンの理解に対して，MJgit は効果を大きく発揮すると考えられる．

一方で，左から 2 番目のペアや 6 番目のペアでは大きな減少は見られなかった．これらのタスクでは，対象メソッドに対して大幅な変更がなく，変更箇所も 1,2 行の if 文の追加などの比較的発見しやすく，理解しやすいタスクであった．このようなノイズが少ないリビジョンに対しては MJgit はあまり大きな効果を発揮できなかった．

7.7.3 被験者ごとのタスク完了時間の結果

被験者ごとのタスク完了時間の比較を表 6 に示す．この表は，各被験者の全 20 個のタスクにおける平均作業時間を，ツールの有無で分類して一覧にした表である．

全ての被験者が，MJgit を使用することで作業時間の平均時間を削減できていた．その中でも被験者 6 と 11 は，出力結果に対して文字列検索するといった外部ツールを併用しており，ツールなしでもタスク完了時間が短い．そのようなツール等に熟練した被験者であっても，ツールを使うことでその作業時間をさらに短縮できており，提案ツールの有用性が伺える．

7.7.4 アンケートの結果

アンケートの結果得られた MJgit に対する意見を図 14 に示す．この結果から，この実験で与えたタスクにおいて MJgit はオリジナルの JGit 以上の使いやすさがあり，実行時間の低下はそれほど気にならないということがわかる．また，git-log を使用する $Task_A$ において MJgit が特に使いやすいことがわかる．そして，今後 MJgit を使用したいと思った被験者は 12 名であった．

MJgit に対する意見の自由記述の回答をいくつか抜粋する．好意的な意見としては以下のような意見があった．

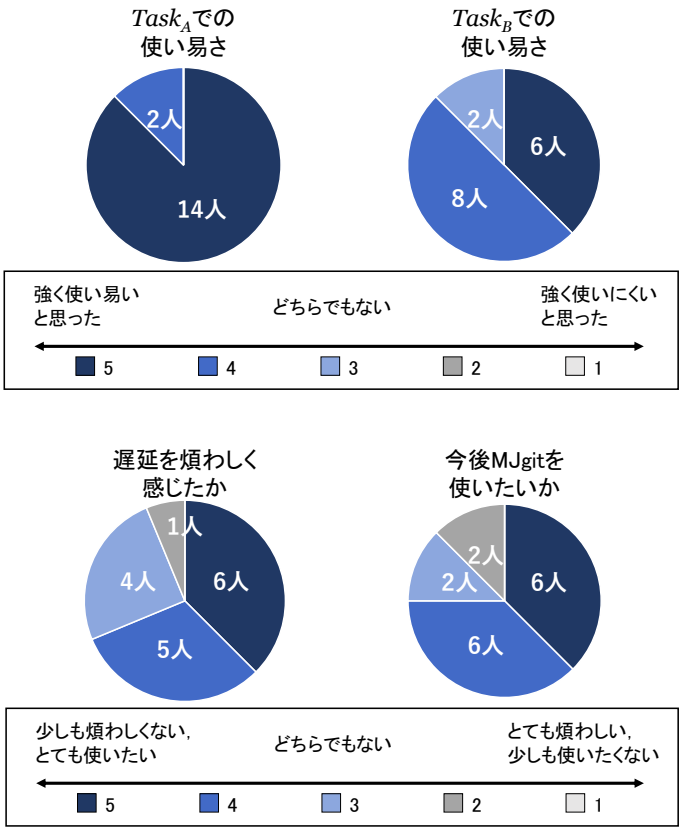


図 14　アンケート結果

> 通常の log コマンドではコミットメッセージから変更内容を予測するしかなく，実際に見たいメソッドが変更されたのかどうかは show を組み合わせてみるしかない．しかし，MJgit では検索したいメソッドの文脈に絞って log を見ることができるため，ここで表示されるコミットが見たいメソッドを変更したものであることが保証されており，レビューの手間が大幅に削減できると思う．

> とても使い易く便利なツールだと思う．特に log コマンドでクエリを指定できるのは便利だと感じた．

> 同様の Java 構文に特化した検索機能が GitHub に実装されていると便利だと思う．

一方で，MJgit の短所や改善点に関する意見もいくつか見受けられた．

> 空行のみの変更がしばしば出てきたので，そのような場合はログ出力を抑制してほしい．また，コードフォーマットの変更のようなプログラムの動作に影響のない diff は出力しないオプションがあると良い．

> 変数名だけでなく，どのメソッドの変数かというスコープも同時に指定できるとより絞り込みがうまく働くと思う．

> Exception 周りのみの差分を得るようなオプションもあると特定の状況で役に立つ．

7.8 考察

被験者実験の結果により，MJgit を使用することで開発者のタスク完了時間を大幅に削減できており，MJgit は開発履歴の理解というタスクに有用であると結論付けることができる．特に，複数のログから関心対象となるログを抜き出すタスク（$Task_A$）で完了時間を半分に削減できていた．複数の開発履歴の中から，履歴に関するメタ情報（日付や開発者等）と Java 構文情報の組み合わせにより，効率的な対象の絞り込みを支援できているといえる．

被験者の中には grep などの文字列検索を併用していた被験者も存在していたが，このような条件下でも MJgit は有効であった．これは，単純な文字列検索ではコードの本質とは関係のないコメントなどに検索がヒットしてしまい，ノイズになるためだと考えられる．加え，コードフォーマッタの適用やメソッドの移動といったテキスト部分が大幅に変更するような状況では，提案手法による構文情報を用いたクエリの効果が大きかった．

また，実験で提示した 20 個のタスクは，1 個当たり数分以内に完了できるような容易な内容であった．実際の開発現場ではより複雑かつノイズの多い状況下での履歴理解が必要であり，このような場合に MJgit はより有効に働くと考えられる．

実行速度については，性能評価実験で大幅な性能低下が確認できたが，本被験者実験のアンケートからはほとんど気にならないという意見が多かった．これは，直近の数リビジョンを確認するような状況では性能低下の影響が小さかったからだと考えられる．

8 妥当性の脅威

内部妥当性に関する議論として，性能評価実験では全てのコミットに対して MJgit を実行しているが，1 コミットに対して計測対象コマンドを一度ずつしか実行しておらず，繰り返しのないデータになっている．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．また，被験者実験におけるタスクの実行順序によって実験に慣れが発生し，結果に影響を与えている可能性も考えられる．

次に外部妥当性に関して考える．性能評価実験では 2 つの Java プロジェクトのみを実験対象とした．他のプロジェクトに対して行なった場合には異なる結果が得られる可能性がある．また，被験者実験では実験で使用した Git についてある程度の知識を持った開発者のみを被験者としている．被験者が Git に不慣れな場合，異なる結果が得られる場合がある．

また構成概念妥当性に関する脅威としては，被験者実験での実施タスクは履歴の理解という複雑な行為の一部を抜き出した内容である点が挙げられる．実際の開発作業で履歴を理解するという状況では，それ以外の様々な手順を踏む必要があり，MJgit による作業効率の向上は実験結果と異なる可能性がある．

9 関連研究

ソースコードファイルに含まれているコメントやコピーライトは自然言語で記述されており，これがテキストベースの検索に対してノイズになることがある．3 節で述べたように，テキストベースの操作を用いて `if` 文を検索すると，実行ステートメントの `if` 文だけでなく，コメント内の “if” という文字列が検索に引っかかってしまい，ユーザーに対するノイズになる．

このような問題を解決するために，ソースコードの検索に関する様々な研究が行われている [12–18, 22, 23]．これらの手法は，抽象構文木や制御フローグラフといったソースコードの構造に基づいた検索を実現する．よって，テキスト表現ではなく構文情報を利用してソースコードの一部（メソッドやステートメントなど）を特定することが可能である．

しかし，これらの手法は特定リビジョンのソースコードに対してのみ有効であり，ソースコードの変化の履歴を追跡することはできない．一部，ソフトウェアリポジトリに対するソースコード検索を取り入れた研究は行われている [24, 25]．しかしながらこれらの研究の焦点は，数万を超える大量のソフトウェアリポジトリをいかにスケールする形で分析させるか，という点にあり，開発者が特定のリポジトリの理解のために直接用いるような仕組みではない．また，Historage [26] はソースコードの変更をメソッド単位という細粒度で分析可能であるが，既存のリポジトリを大幅に再構築する必要がある．

10 おわりに

本提案手法では関心の狭いユーザーに対する効率的な開発履歴の支援を目的として，Git コマンドに対してソースコード検索を統合するツール，MJgit を提案した．性能評価実験では，git-diff に対する性能低下は概ね実用の範囲内であることを確認した．また，被験者実験では，履歴理解というタスクに対してその完了時間を削減できること，及び複数の被験者から好意的な意見を得ることができた．性能評価実験で確認できた git-log に対する大幅な性能低下は，気にならないという意見が多く，直近の数リビジョンを確認するような状況では無視できる範囲であるといえる．

現在，MJgit はメソッド名と変数名の指定，及びプログラム構文の指定をサポートしているが，スコープの指定といったより高度な構文情報の指定や，拡張対象コマンドでありながら MJgit でサポートされていない git-blame の実装は今後の課題である．

第 III 部

BJgit

付ける

11　はじめに

版管理システムでは，開発ログを出力するコマンドが用意されており，そのログの出力に対して検索で情報を絞り込む機能もサポートされている．例えば，git-log コマンドにはコミッターを指定してその人物のコミットログのみを出力したり，ファイル名を指定してそのファイルが変更されているコミットログのみを出力したりすることができる．しかし，この検索方法は排他的であり，検索した条件をもたないログを全て非表示にしてしまう．開発履歴は連続している情報であり，その文脈を理解することはリポジトリの全体像を掴むためにはとても重要である．よって情報を除去してしまう手法は関心の広いユーザーの需要を満たさない．また，その検索方法も簡単なものしか用意されておらず，例えば，自分”以外”のコミットを探す等の複雑な検索方法は基本的にサポートされていない．

本提案手法は，ユーザーの関心が広い状況での適用を想定している．よって情報を除去するのではなく，特定の条件を満たしていることを表すタグという情報をログに付与することで，開発履歴の理解を支援する．

提案手法である BJgit のキーアイディアは，開発ログに対して，カスタマイズ可能なタグを付加することである．このタグという情報の付加により，情報を除去して文脈を断ち切ってしまうことなく履歴を特徴付けることができ，関心の広い需要に応じることができる．また，タグのカスタマイズ性から，簡易的な検索では表現が難しかった，ユーザーが求めている情報に基づいた特徴付けが可能となる．このように，BJgit によってユーザーの広い関心に適した情報を提供することができる．

付加できるタグの具体例は以下の通りである．

- [Me] タグ：使用者がコミットしたログに付く
- [Friday] タグ：金曜日にコミットされたログに付く
- [No-Change-Statement] タグ：Java の実行文が変更されていないログに付く

[Me] は使用者自身でコミットしたタグに対して付く．ユーザーが切り替わった場合でも Git の設定ファイルからユーザーネームを取得し，自動的にユーザーの名前を判断するタグである．このような検索は標準 Git クライアントではサポートされていない．

また，[Friday] タグは金曜日にコミットされたログに対して付く．出力されるログ情報には曜日も含まれているので，出力結果をテキスト処理することで従来手法でも判定可能ではあるが，煩雑な処理が

必要で，ユーザーにとって難易度が高いと言える．BJgit のタグを使用することで，このような複雑な情報の取得にも対応することができる．

[No-Change-Statement] は，そのコミットで Java の実行文が変更されていないログに対して付く．例えば Java ファイルが変更された場合でも，コメントや改行の変更しか行われていなければこのタグが付くことになる．このようなソースコードの内容まで分析して情報を取得することは従来手法ではサポートされていない．

本提案手法の目的は，上記のアイデアを実現し，関心の広いユーザーに対する効率的な開発履歴の理解を実現することにある．第Ⅲ部では，この目標を達成するための Git クライアントの拡張ツール，BJgit を実装し評価する．この提案ツールは，各ログに対してコミット情報に基づいた様々なタグを付けることができる．また，タグをユーザー自身で作成する機能も実装している．タグが用意されていない場合，BJgit は拡張対象である JGit と同じように動作するため，BJgit は JGit に対して完全な後方互換性を持つ．BJgit の評価実験として，実際のソフトウェアリポジトリを用いた性能評価実験，及び有用性を確認するための被験者実験を実施した．実験結果により，BJgit を用いることで開発履歴の理解は支援されるが，重大な速度低下が見られ，それによって BJgit の利便性が悪くなっているなどの課題を確認できた．

以降第Ⅲ部では，12 節で研究動機として，従来の版管理システムの問題点を述べる．13 節では提案手法について説明し，14 節では性能評価実験を，15 節では被験者実験を行い，それぞれの考察を述べる．また，16 節では妥当性の脅威について述べ，17 節では関連研究について述べる．最後に，18 節で本提案手法 BJgit についてまとめ，今後の課題について述べる．

12 研究動機

図 15 はあるプロジェクトにおける開発履歴を簡易的に表している．このプロジェクトは主に Java で開発されており，複数人が参加しているプロジェクトである．1 つのログには多くの情報が含まれており，例えば図 15(a) のようにコミット ID やコミッターの情報，コミットメッセージ，変更されたファイル名や変更内容など様々な情報を含んでいる．また，主に Java で開発されているプロジェクトであっても，Java ファイル以外に様々なファイルを管理しており，コミットによっては設定ファイルや README を変更しているものも存在する．

ここで新しくプロジェクトに参加する開発者がこのリポジトリの開発の特徴を掴もうと履歴を眺めているとする．

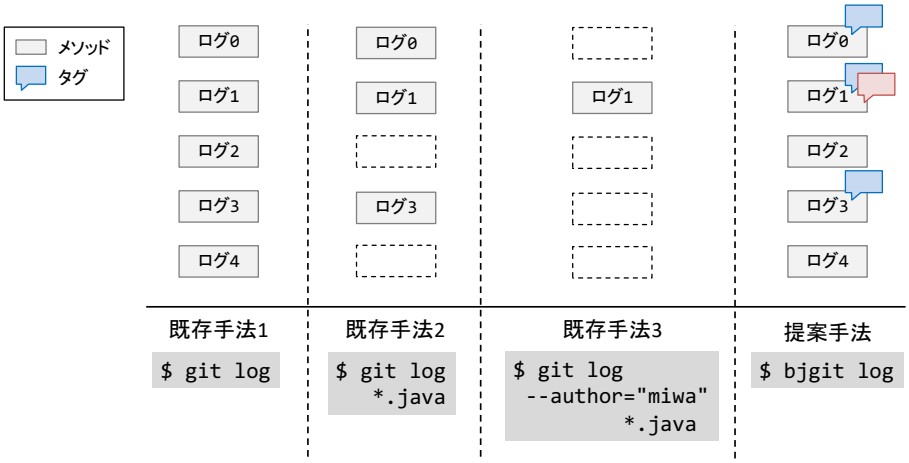
既存手法 1： 開発履歴を確認する簡単な方法としては git-log で開発履歴全てを見る方法である．検索オプションなしで git-log を実行することで，履歴全てを出力することができる．しかし，この手法では全ての履歴が平坦に出力されるので，全体を見ることはできるが，どの履歴に注目していいかがわかりにくい．

例えば`--name-only` オプションを追加することで変更されたファイル名を出力したり，`-p` オプションを追加することで diff を確認しながら履歴を見ることはできるが，これらのオプションでは変更された情報を羅列するだけなので特徴を強調することはできない．また，そのような多すぎる情報は特徴を掴みたいだけの開発者には不要になる可能性が大きい [27].

既存手法 2： 次に，特徴を掴む方法として Java ファイルが変更されているコミットに注目しながら履歴を眺めたいと開発者が考えたとする．この時，既存手法では図 15(b) のようにオプションで正規表現`*.java` を使用することで，Java ファイルが変更されているログのみを出力することができる．しかし，この手法ではオプションの条件に当てはまらないログは非表示になり，開発履歴の繋がりが見えなくなってしまう．Git はこのようにメタ情報に基づいた検索機能をいくつかサポートしており，それらを組み合わせることで様々な情報を絞り込むことが可能である．しかし，これらの検索方法は全て情報を切り出すもので，開発履歴の前後の流れを断ち切ってしまう．開発履歴とは連続した情報であり，文脈が理解しづらくなる出力は理想的ではない．

既存手法３： さらに条件を追加して miwa というコミッターが Java を変更していないかに関心を持ちながら履歴を理解したい時，既存手法ではオプションに条件`--author="miwa"`を追加することで検索条件を併用することができる．しかしこの手法では，既存手法 2 の場合よりも多くの情報が消えてしまう．このように開発者が関心を持っている特徴が増えるほど履歴の文脈が見えなくなるという問題が既存手法には存在する．

(a) あるログが持つ情報の一例



(b) 提案手法と既存手法の比較

図 15　BJgit の研究動機

提案手法　提案ツール BJgit を使用する場合を考える．この手法は情報を除去するのではなく，タグという形でログに付与する手法である．Java ファイルを変更したことを示すタグ (図中の青い吹き出し) と，miwa のコミットを示すタグ (図中の赤い吹き出し) を作成し，履歴に適用した．この手法によって，開発の流れを消すことなく情報を可視化してユーザーに提供することができ，開発者は付加されたタグを頼りに関心の優先順位を切り替えながら，開発全体の特徴を掴むことが可能になる．このようにして BJgit はユーザーの履歴理解を支援することができる．

その他の提案手法の利点　図 15 では簡単のために Git でサポートされている検索手法で可能な検索を

例に挙げているが，”変更されたファイル数が 10 個以上”や，”Java の実行文が変更された”などの Git で標準的にサポートされていない条件の場合，既存手法では検索できない．よって，ユーザー自身でタグを作成でき，このような複雑な特徴に対してタグ付けできるという機能は BJgit の利点である．

13　提案ツール：BJgit

13.1　概要

BJgit の目的は，git-log に対してタグ付与機能を追加することにより，関心の広いユーザーに対して効率的な開発履歴の理解を支援することである．BJgit の基本的なアイディアは，git-log に対して，コミット情報を元にしたタグという情報を付与して出力する，という点である．それによって，ユーザーの関心の優先順位を意識させつつ，開発の流れを絶つことのない情報の提供が可能になる．

このアイディアの実現と共に，タグを元にしたフィルタ機能，ユーザー自らタグを作成して設定できる機能も実装している．

13.2　処理の流れ

BJgit の処理の流れを図 16 に示す．図の各手順を以下で説明する．

1. まず開発者は開発履歴を取得するために BJgit の git-log コマンドを実行する．
2. 次に，BJgit はリポジトリからログの集合を取得する．その後，取得した各ログに対してコミット情報を元にしてタグ付け処理をしていく．このループは”for each revision”で示している．
3. タグの設定が複数ある場合，一つのリビジョンに対して全てのタグ付け判定を行うループに入る．このループは”for each tag”で示している．タグ付け判定はそれぞれのタグに用意されたクラスファイルを実行して行われる．
4. そのログがタグけ条件を満たしていた場合，ログにタグを付与する．
5. タグ付け判定で条件を満たしたタグ全てをログに付ける．またこの時，タグに基づいたログのフィルタ設定をすることもできる．この図の場合，tagB がつくログをフィルタする設定になっているとする．
6. 全てのログに対して全てのタグ付け判定をしたのち，タグ付けと場合によってはフィルタによって非表示にしたログの集合をユーザーに提供する．この例では tagB がつくログをフィルタする設定となっているので，図のように tagB が付いていたタグが非表示にされた状態でユーザーに渡される．

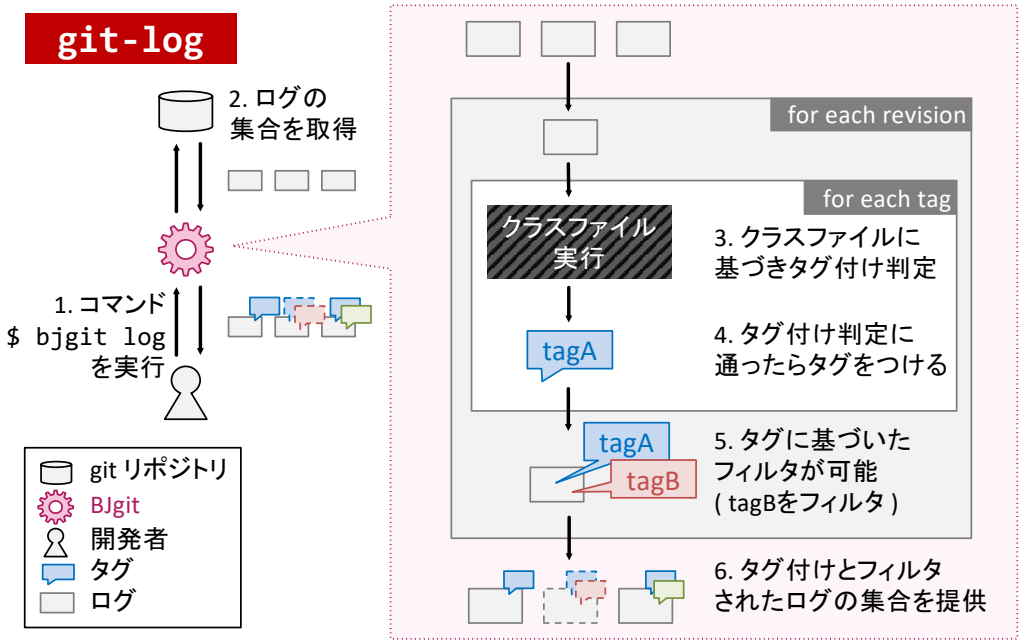


図 16　BJgit における git-log コマンドの処理の流れ

13.3　タグの設定ファイル：tagConfig.json

BJgit はタグ付け機能の細かい設定を記述する tagConfig.json という JSON 形式のファイルを読み込み，各ログにタグをつけている．この節では tagConfig.json の説明をする．tagConfig.json の具体例は以下の通りである．

```
{
  "tag": [
    {
      "tag-name": "Friday",
      "rule": "Friday.class",
      "color": "RED",
     "ignore": "off"
    },
    {
      "tag-name": "Huge-Add-Change",
      "rule": "HugeAddChange.class",
```

```
      "color": "AAA",
      "ignore": "on"
    }
  ]
}
```

この例では，タグ [Friday] と [Huge-Add-Change] の設定を記述している．それぞれの設定値について上記の例を用いて説明する．

- tag-name: ログに出力するタグの名前．[ ] で囲って出力される．任意に設定可能．
- rule: クラスファイル名．タグ判定を実装したクラスファイルの名前を指定する．タグ判定時にこのクラスファイルを実行し，条件を満たすログに対して tag-name に応じたタグが付与される．
- color: タグの色．ここで設定された色でコンソールに出力される．BLACK, RED, GREEN, YELLOW, BLUE, PURPLE, CYAN, WHITE の 8 種類の色を指定することができる．これ以外の文字列を設定すると赤色で出力される．この例の場合両タグ共に赤色で出力されることになる．
- igore: タグのフィルタ設定．これを on にするとそのタグ条件を満たすログを非表示にする．この例の場合，タグ判定を行い，[Huge-Add-Change] の条件を満たすと判断されたログは出力されない，という設定になる．

タグを新たに追加する時は，これらの設定値を全て定義し，tagConfig.json に追記することでタグを増やすことができる．クラスファイルの実装方法などの詳細は 13.4 節で説明する．

13.4 タグ作成機能

BJgit はタグをユーザー自身で作成できる機能を持つ．具体的には，タグ付け判定を実装したクラスファイルを用意し，tagConfig.json に新規タグの設定を書き込むことでタグを追加することができる．タグの作成を容易に行うために，タグのインタフェースを定義し，コミット情報を容易に取得するために各情報に対して getter も定義した．

13.4.1 getter の提供

タグ作成時に使用されるであろう情報を容易に取得できるように BJgit 上で getter を定義した．これを使用することでユーザーは簡単なコミット情報ならすぐに取得でき，タグ作成における実装を簡単にすることができる，

現在 BJgit に実装されている getter によって取得できる情報は以下の通りである．

- コミット ID
- コミットメッセージ
- コミッターのユーザーネーム
- コミッターのメールアドレス
- 日付データ
- 親コミットの数
- コミットされたファイルの数
- 変更されたファイルの名前
- 変更されたファイルのステータス[*4]
- 変更されたファイルの変更前 / 後の生のソースコード
- 変更された Java ファイルの変更前 / 後の AST

13.4.2 タグのインタフェース

タグの実装方法を統一することでどのユーザーが作成したタグでも BJgit がタグ付け判定可能になるように，タグのインタフェースとして TagProcess を定義した．全てのタグは以下の TagProcess を継承して作成される．

TagProcess では apply メソッド内でタグ付け条件を判定するコードを処理する．apply メソッドの返り値は boolean 型であり，タグが付くべきと判定された場合は true を返す．また，引数の Loginfo は 13.4.1 で用意した全ての getter にアクセスするためのクラスである．

```
public interface TagProcess{
  public boolean apply(Loginfo Loginfo);
}
```

13.4.3 タグ作成の具体例

実際にタグを作成した例を図 17 に示す．この図では 13.4.1 節で用意した getter で取得できる情報を赤くハイライトしている．

この例は [Huge-Add-Change] タグの実装例である．13.4.2 節で説明した TagProcess を継承している．このタグは，各コミットに 100 行以上の追記のあるファイルが 1 つでも存在した場合にタグを付与するという条件を持つ．

[*4] ステータスとはそのファイルが該当ログで追加，編集，消去，リネーム，コピーのどの変更を加えられたのかを表す

```java
public class HugeAddChange implements TagProcess {
  @Override
  public boolean apply(LogInfo info) {
    // 条件: 100行以上の追記があるファイルがひとつでもある
    List<CommitFile> commitFiles = info.getCommitFiles();

    // コミットされたファイルたちの情報取得
    for (File file : commitFiles) {
      int oldLineNum = countLineNum(file.getRowOld());
      int newLineNum = countLineNum(file.getRowNew());

      if (newLineNum - oldLineNum >= 100) return true;
    }
  return false;
  }

  int countLineNum(String rowCode){....
```

図 17　タグ [Huge-Add-Change] の実装例

この例では，コミットされたファイルの生のソースコードを取得し，その行数を変更前と後で比較することで条件を判定することができる．赤のハイライト部分はユーザーが getter を使用するだけで取得できるので，ユーザーはソースコードの行数をカウントする countLineNum メソッドを実装するだけで [Huge-Add-Change] を実装することができる．このように getter とインタフェースを使用することでタグの実装を統一的にかつ容易に行うことができる．

表 7　BJgit に実装されているプリセットタグ

タグ名	タグ付け条件
Me	使用者のコミットしたログ
W/i-1month	1 ヶ月以内にコミットされたログ
Friday	金曜日にコミットされたログ
Holiday	土日にコミットされたログ
Midnight	22:00 - 5:59 にコミットされたログ
>10	変更されたファイル数が 11 以上のログ
Huge-Add-Change	100 行以上の追記があるファイルがコミットされたログ
Merge	マージコミットのログ
No-Change-Statement	Java の実行文が変更されていないログ

```
commit 7cbcc701b985ed44bcde557e3b51e24cb4563cad
Author: m-sasaki <m-sasaki@ist.osaka-u.ac.jp>
Date:   Mon Jun 18 04:27:43 2018 +0900

    git-logで使用するdiff部分の実装

commit a6af1e525b98b3ce605b4667077c2fae8ebbbcb1
Author: shinsuke <shinsuke@ist.osaka-u.ac.jp>
Date:   Thu Nov 09 13:56:04 2017 +0900

    var.nameを指定した際のsimplenameの挙動を修正
```

図 18　従来の git-log の出力例

```
commit 7cbcc701b985ed44bcde557e3b51e24cb4563cad
Author: m-sasaki <m-sasaki@ist.osaka-u.ac.jp>
Date:   Mon Jun 18 04:27:43 2018 +0900
tag: [Me][Midnight][Huge-Add-Change]

    git-logで使用するdiff部分の実装

commit a6af1e525b98b3ce605b4667077c2fae8ebbbcb1
Author: shinsuke <shinsuke@ist.osaka-u.ac.jp>
Date:   Thu Nov 09 13:56:04 2017 +0900
tag: [>10]

    var.nameを指定した際のsimplenameの挙動を修正
```

図 19　BJgit の git-log の出力例

13.5　実装

BJgit の実装は，MJgit と同じく Java ベースのオープンソースの Git クライアント JGit を拡張して行われた．BJgit のソースコードは GitHub 上で公開されている[*5]．

BJgit は使用者自らタグを作成できるツールだが，タグのプリセットとしていくつかのタグを実装してある．現在提供しているプリセットタグの一覧を表 7 に示す．

いくつかのタグは Codoban らの調査 [27] で得られた版管理システムユーザーの要望に答えることができるタグとして実装している．例えば，マージコミット以外を出力したいという要望があると述べているが，それは [Merge] によって解決することができる．また，Śliwerski ら [28] により，金曜日に行われたコミットにはバグ混入することが多いと言われているので，[Friday] も作成した．

[*5] https://github.com/kusumotolab/m-sasaki_BJgit

13.6 BJgit の出力例

図 18 は従来の git-log の出力例である．この例では二つのコミットログがコミット ID, コミットメッセージ，コミッターの情報などと共に出力されている．

対して図 19 が BJgit の git-log の出力例である．図のように，BJgit では出力される情報の中に tag 用の列を追加している．タグを設定してある場合，この列にそれぞれの条件を満たすタグを付与して出力する．

14 性能評価実験

14.1 概要

本実験の目的は，BJgit によるタグ付け機能の実行性能が実用的かどうかを確認することである．BJgit では各ログにタグを付けるかを判断する際に，リポジトリに保存されているコミット情報から必要な情報を取得し，抽象構文木の構築や，生のソースコードの解析などの処理が必要である．この追加処理が，Git 利用時における実行性能に対してどの程度影響を与えるかを確認する．

14.2 実験設計

2 つの Java のオープンソースリポジトリに対して BJgit を実行し，その実行速度をオリジナルの JGit と比較する．より具体的には，タグ付けするタグを，そのタグ付け判定に必要な情報の種類を元に分類し，各カテゴリーごとの実行時間を測定し，JGit の実行時間と比較する．タグ付けはリポジトリ内の全コミットに対して実行した．いずれのカテゴリーのタグも 10 回ずつ実行し，その平均を計測値とする．

対象プロジェクトは 6.2 節で MJgit の性能評価実験対象とした JUnit4 と Log4j である．概要は同じく表 3 に示す．

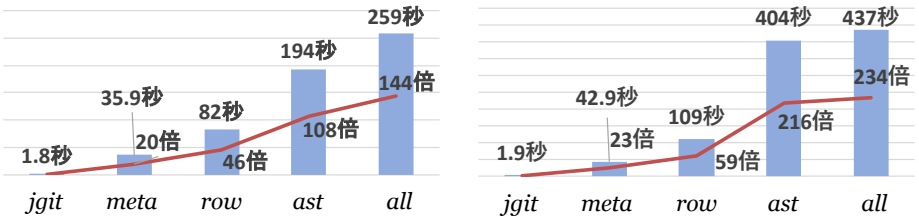
BJgit では様々な条件のタグを付けることができる．この実験では，タグの種類をタグ付けのために必要な情報の特性ごとに 3 種類に分類し，その種類ごとに実行時間を測定した．その 3 種類とは，

- メタ情報を必要とするタグ
- ソースコードの生データを必要とするタグ
- ソースコードの AST を必要とするタグ

の 3 種類である．

表 8　比較した 5 つの実験設定

ラベル	判定するタグ	タグ付に必要な情報
jgit	なし	-
meta	[Holiday]	メタ情報
row	[Huge-Add-Change]	ソースコードの生データ
ast	[No-Change-Statement]	ソースコードの AST
all	*meta*, *row*, *ast* の 3 つのタグ	メタ情報, 生データ, AST



(a) JUnit4　　(b) Log4j

図 20　各プロジェクトにおける BJgit の実行時間と速度増加率

この分類を元に 5 つの実験設定で実行速度を測定した．実験設定の一覧を表 8 に示す．本実験で実行速度を測定した実験設定は，分類された 3 種類にそれぞれ対応するタグである [Holiday],[Huge-Add-Change],[No-Change-Statement] をそれぞれひとつずつ付与したタスク（*meta*, *row*, *ast*）と，それら全てをタグ付けるように設定したタスク (*all*) である．また，オリジナルの JGit のコマンドは *jgit* とラベル付けした．これは，性能比較におけるベースライン計測のために設定した実験タスクである．

本実験では，このベースラインに対して *meta*, *row*, *ast* の各種類のタグに対する性能低下の程度を確認し，さらに *all* によって，タグを複数設定した場合の性能低下の様子を確認する．

14.3　実験結果

実験結果を図 20 に示す．5 つの棒グラフはそれぞれのタグを付けた場合の git-log の実行時間を表している．また，折れ線グラフはそれぞれの実行時間が *jgit* の何倍にまで増加したかを示している．

jgit の実行時間は，両プロジェクト共に 2 秒以内という極めて短い時間で完了できている．その一方，タグをつけた場合の実行時間は，最小の *meta* の場合でも約 36 秒と，20 倍以上の時間増加が確認できた．3 種類のタグのうち，特に時間増加が大きかった種類は *ast* であり，タグ 3 つを実行した *all* が一番遅いという結果になった．

14.4　考察

実験結果から，どのタグをつけても *jgit* より実行時間が増加することが確認できた．ここではそれぞれの増加割合に違いが出た理由を考察していく．

まず一番増加割合が小さかった *meta* と次に増加割合が小さかった *row* を比較する．*meta* で実行されたタグ [Holiday] は各コミットに対して”コミットされた曜日”という一つの情報のみで判断できるタグであった．それに対して，*row* で実行されたタグ [Huge-Add-Change] は各コミットに対して”変更されたファイル全てに対して”100 行以上の追記がないかどうかを確認することでタグ付け判定をするタグである．よって，この分析対象となる情報量の差が実行時間の増加量の差として現れたと考えられる．

次に *row* と，一番増加割合が大きかった *ast* を比較する．この二種類間には，非常に大きな差が確認

できた．*ast* で実行されたタグ [No-Change-Statement] は [Huge-Add-Change] と同様に変更されたファイル全てに対して処理を行わなければならず，加えて”AST を構築して”タグ付け判定をする必要があった．よって，この AST の構築という処理の差が実行時間の増加量の差として現れたと考えられる．

最後に，本実験では，全てのタグを設定した *all* が一番遅くなったという結果が得られた．このことから，タグを多くつけるほど実行時間が増加していくことが予想される．実際の BJgit では 3 つや 4 つのタグではなく，十数種類のタグがつくと考えられ，実際の性能低下はこの結果より更に悪くなると予想される．よって実際のユーザーがこの速度低下を実用上どのように感じるかは，更に多くのタグを設定した被験者実験による調査が必要である．

15 被験者実験

15.1 概要

この実験の目的は，Git のリポジトリの特徴を掴むというタスクに対して，BJgit がどの程度有用であるかを確認することである．このために，被験者 10 名に対してプリセットタグを与え，開発履歴の特徴についていくつかの質問に答えるというタスクを与えた．タスクの最後にアンケートにより BJgit を使用した印象やコメントを収集し分析する．

15.2 被験者と実験対象リポジトリ

被験者は大阪大学大学院情報科学研究科に所属する教員 1 名，同研究科所属の修士の学生 7 名，大阪大学基礎工学部の学部生 2 人の計 10 名を対象とした．

実験対象のリポジトリは，被験者自身が参加している Java で開発されているプロジェクトとした．被験者らは全員普段から Java と Git を使用しており Git の基本的な操作は熟知している．

15.3 実験タスク

実験タスクは，Git リポジトリの特徴理解である．具体的には以下のタスクを被験者に与えた．なお，どのタスクも実験対象リポジトリにおける最新 50 個以内のコミットを対象として答えてもらった．ただし，これらのタスクは作業時間や回答の正確さを求めているものではない．なぜなら，本実験の目的は BJgit を実際に使った場合の使い心地を調査することであり，これらのタスクは具体的に BJgit を使用する状況を被験者に与えるために用意しただけに過ぎないからである．

- あなたのコミットの割合は何割ほどですか？
- 1 ファイルに 100 行以上の追記があったコミットは何個ありますか？
- 深夜（22:00 から 5:59 まで）にコミットしているのが多い人は誰ですか？
- Java の実行文以外のみを変更している人が多いのは誰ですか？
- 変更されたファイル数が 10 以下のコミットは何個ありますか？
- 1 ヶ月以内に行われた，変更されたファイル数が 11 以上のコミットは何個ありますか？
- 土日のコミットが多い人は誰ですか？
- あなたの行ったマージコミットは何個ありますか？

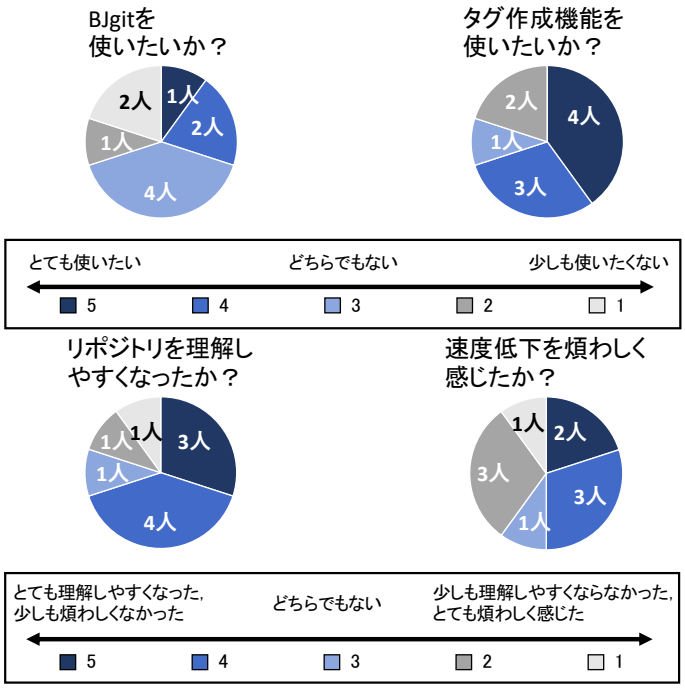


図 21　アンケート結果

15.4　実験の流れ

被験者実験は以下の流れで実施した．

1. BJgit の機能の解説：BJgit の使用方法や振る舞いについて被験者に説明する．
2. タスク実行：15.3 節のタスクを被験者に実行してもらう．ただし，どのタスクも正解不正解を求めているタスクではないので気楽に答えていただくよう被験者に伝えた．
3. アンケート回答：全タスクを完了した後，BJgit の印象を回答してもらう．

15.5　アンケート評価

全タスクの実施後，被験者に以下のアンケートに答えてもらい，ツールの有用性や課題を確認する．

- BJgit によってリポジトリの理解はし易くなったか（5 段階評価）
- BJgit を実際に使いたいか（5 段階評価）
- BJgit のタグ作成機能を使いたいか（5 段階評価）
- BJgit の速度低下は煩わしかったか（5 段階評価）

15.6 実験結果

15.3 節で述べたように，本実験のタスクは時間や正答率を調査するものではない．よって実験結果はアンケート結果のみを掲載する．

アンケートの結果得られた BJgit に対する意見を図 21 に示す．この結果から，BJgit を使用することでリポジトリの理解はし易くなるということは確認できたが，BJgit を実際に使用したいかという質問をすると，どちらでもない人が一番多いという結果も確認できた．また，タグ作成機能を使用してみたいと考えている被験者が多いこと，性能低下を煩わしく感じている被験者が多いことも確認できる．

BJgit に対する自由記述による意見をいくつか抜粋する．好意的な意見やタグや BJgit の機能へのアイディアとしては以下のような意見があった．

> 遅さはそれほど気にならなかったです．

> コミュニティメンバのコミットを表すタグが欲しい．
> OSS では自分以外にも複数人で開発していることが一般的．メンバーよるコミットと，メンバではない外の人がプルリクしたコミットなどを区別したい場面はあるはず．

> CUI 以外からタグが見れると面白そうだと思いました．例えば github 上でなど．

> タグがついたコミットを検索する機能がほしいです．

> 既に実装されているタグに対する逆のタグもプリセットで欲しいなと思いました．
> （Me に対して Other, Holiday に対して Weekday のように）

一方で，BJgit の短所や改善点に関する意見もいくつか見受けられた．

> log が全部一度に吐き出されるのは使いづらい．less コマンドにパイプすること前提になるので，もうちょっと工夫がほしい．

> タグのカスタマイズが面倒臭そう．
> tagConfig.json でカスタムできるのはタグ名，色，ignore 程度しかなく，タグの処理そのものを変更するには新たに class ファイルを追加する必要がある．class ファイルはコンパイル済みのバイナリファイルなのでそのままでは編集できない．新しいタグの作成時には 変更する → コンパイルする → class ファイルを配置し直す……という手順が必要で煩雑．
> class ファイルではなく，java ファイルを直接読み込むようにすべきだと思う．こうすればその場でどんどん自由にカスタムできる．

> スピードが遅い．

15.7 追加インタビュー

BJgit を使いたいという被験者が少ない点について原因を明らかにするために，追加でインタビュー調査を実施した．追加インタビューは 15.6 節のアンケートで BJgit を全く使いたくないと回答した 2 人に対して実施した．

彼らに BJgit を使いたくないと思う原因について挙げてもらったところ，

- git-log の出力結果を less コマンドのように 1 画面ずつ表示できない
- プリセットタグに実際の開発上で使えそうなタグがなかった
- プリセットタグのほとんどがコマンドを組み合わせれば可能な気がする
- 動作が遅い

のような意見が上がった．このうち出力結果を less 表示できないというのは JGit の仕様上の問題である．

15.8 考察

被験者実験の結果より，BJgit を使用することでリポジトリの理解をし易くなったと結論付けることができる．しかし，実際に使用したい被験者が少なかったことと，性能低下を煩わしく感じている被験者が多かったことから，BJgit の実用性を高めるためには解決しなければならない課題があると言える．

まず第一に BJgit の性能低下に対する不満が多く見られた．この不満を解消するために，BJgit にキャッシュ機能を追加すべきと考えた．本提案手法のタグという情報は，一度条件判定をしてしまえばコミット情報が変わることはないので保存して次に再利用可能な情報である．なので，キャッシュ機能を実装することができれば最初のタグ付け判定のみ時間がかかるが，その後の出力の速度低下は大幅に軽減させることができると考えられる．

また，次に多かった意見が git-log を Git のように less 表示できないことに対する不満であった．これは BJgit のオリジナルである JGit における git-log の仕様として less 表示ができないことに起因する問題である．この仕様を変更することは，提案手法のアプローチとは乖離していたので less 表示が不可能なまま BJgit を開発したが，実際に使用してもらう場合，このようなツールの使いやすさに繋がる問題は避けられない．よって BJgit の改善点として less 表示を可能にするべきだと考える．

また，タグに基づいてコミットを非表示にするフィルタ機能だけではなく，タグに基づいてコミットを表示する検索機能が欲しいという意見も確認できた．本提案手法の基本的なアイディアはログに対してタグという情報を付加することだが，ユーザーの関心を考えると，広い関心から狭い関心へと関心が狭まっていくことはあり得る状況なので，タグに基づいた検索機能を実装することで，より実際に使っ

てもらいやすいツールになると考えられる．

現在実装されているプリセットタグに開発に役に立ちそうなものが少ないという不満も確認できた．このことから，プリセットであっても便利なタグを実装してユーザーの関心を引くことが必要であると言える．よってアンケートで集まったタグ案の中に，テストや ISSUE に関するタグ案が寄せられていたので，それらを参考にタグを作成すべきである．

最後に，被験者はタグの作成機能に興味と期待を持っているということがわかった．ただ，アンケートの自由記述にあったように，現在のタグ作成機能は煩雑な操作をしなければならないので，ユーザーにとって使いづらい可能性がある．よってコンパイルやクラスファイルの配置，tagConfig.json への追記等も容易にできるような開発環境の提供ができるように改良をするべきだと考えられる．

16　妥当性の脅威

内部妥当性に関する議論として，性能評価実験では全てのコミットに対して MJgit を実行しているが，1 コミットに対して計測対象コマンドを 10 回ずつしか実行しておらず，繰り返しの少ないデータになっている．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．また，被験者実験におけるタスクの実行順序によって実験に慣れが発生し，結果に影響を与えている可能性も考えられる．

次に，外部妥当性に関して考える．性能評価実験では 2 つの Java プロジェクトのみを実験対象とした．他のプロジェクトに対して行なった場合には異なる結果が得られる可能性がある．また，被験者実験では実験で使用した Git についてある程度の知識を持った開発者のみを被験者とし，対象リポジトリ自身の参加しているプロジェクトとしている．被験者が Git に不慣れな場合や，途中でプロジェクトに参入する場合，異なる結果が得られる場合がある．

また，構成概念妥当性に関する脅威としては，被験者実験でのタスクはリポジトリの特徴の理解という複雑な行為の一部を抜き出した内容である点が挙げられる．実際の開発作業でリポジトリの特徴を理解するという状況では，それ以外の様々な手順を踏む必要があり，実際の BJgit による作業効率の向上は実験結果と異なる可能性がある．

17 関連研究

開発履歴を効率的に理解するために，様々な手法が提案されている [29–31].

Francisco ら [30] は History Slicing という手法を提案した．これは，任意の数のソースコード片に対して，任意の範囲のログに渡りコード変化を自動的に識別する手法である．これにより，開発者が追跡する情報量を大幅に減少させることができる．

また，ソフトウェア開発における情報の可視化という観点でも様々な可視化ツールが提案されている [32–35].

Tu [33] らはソフトウェアがどのように進化したかを理解するために．ソフトウェアを視覚化し，構造やアーキテクチャの変化を推論するための手法を提案した．この手法によって大規模システムを保守するユーザーの理解を支援することができる．

また，Yoon [34] らは，Azurite というコミットされていない情報を可視化するツールを提案している．この手法はコードの変更履歴を視覚化し，特定のコード片の履歴の表示や，選択的な UNDO 操作などの操作をサポートとしている．

これらの提案手法は，ソースコードの進化を自動識別したり，可視化したりすることでユーザーの履歴理解や開発を支援している．しかし，これらの手法の対象は特定のソースコードの情報であり，ユーザーの関心が狭い時に理解を支援する手法と言える．よって，本提案手法とは関心の広いユーザーに適した情報を提供できるという点で異なる．

18 おわりに

本提案手法では関心の広いユーザーに対する効率的な開発履歴の支援を目的として，git-log に対して，タグ付与機能を追加したツール，BJgit を提案した．性能評価実験では，AST の構築が必要なタグを実装していると性能が大きく下がることと，タグを追加した分性能が下がることが確認できた．被験者実験では，性能低下や機能面に対する不満が多く見られた．ただ，BJgit を利用することでリポジトリの理解を支援可能だということは確認できたので，BJgit の有用性はあると言える．今後はアンケートで得られた問題点を解消していくことで BJgit の利便性を向上させていく必要がある．

今後の課題としては，

- キャッシュ機能を実装し，性能低下を抑える
- プリセットタグとして実際の開発に使用できるものを準備する
- タグの検索機能を実装する
- タグ作成機能をより使いやすく改良する
- 出力結果を 1 画面ずつ表示できるように改良する

があげられる．

第 IV 部

結論

本論文では，版管理の開発履歴に対する理解を支援することを目的として，ユーザーの関心の広さに応じた 2 つの手法を提案した．関心が狭い時の提案手法として，Git コマンドにソースコード検索を組み合わせた MJgit を提案した．対して，関心がリポジトリ全体に広く横断している時の提案手法として，開発履歴にタグを付与する機能を追加した BJgit を提案した．

MJgit については，被験者実験より，オリジナルの JGit と比較して有用性を確認した．また，今後の課題としては未実装の git-blame コマンドの拡張や，さらに詳細なクエリの開発を考えている．

BJgit については，被験者実験より，リポジトリの理解を支援することはできるが，利便性に問題を抱えていることを確認した．その問題を解決するために，キャッシュによる速度低下の緩和，タグ作成を容易にする開発環境の提供などが今後の課題として挙げられる．

また，BJgit のアンケート調査より，BJgit にも検索によって情報を絞り込みたいという要望が寄せられた．このことから，ユーザーの関心の範囲が変化していくことに対応できるような手法が求められていると言える．よって，これらの手法を統合させ，ユーザーの関心の変化に対応できる手法として提案することが本研究の今後の課題である．

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．

本研究の評価をするにあたり，MJgit と BJgit の被験者実験に参加してくださり，励まし，ご助言を頂きました楠本研究室の皆様に感謝申し上げます．

最後に，本研究に至るまでに，講義，演習等様々な場面でお世話になりました，大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心より御礼申し上げます．

参考文献

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

[2] Xiaobing Sun, Bixin Li, Hareton Leung, Bin Li, and Yun Li. MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks. *Information and Software Technology*, Vol. 66, pp. 1–12, 2015.

[3] Sarah Rastkar and Gail C. Murphy. Why did this code change? In *Proc. International Conference on Software Engineering*, pp. 1193–1196, 2013.

[4] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proc. International Conference on Automated Software Engineering*, pp. 81–90, 2006.

[5] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proc. International Conference on Software Engineering*, pp. 361–370, 2006.

[6] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communicaitons of the ACM*, Vol. 20, No. 5, pp. 350–353, 1977.

[7] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/.

[8] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199, 2013.

[9] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. Bug propagation through code cloning: An empirical study. In *Proc. International Conference on Software Maintenance and Evolution*, pp. 227–237, 2017.

[10] Mathias Meyer. Continuous integration and its tools. *IEEE Software*, Vol. 31, No. 3, pp. 14–16, 2014.

[11] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proc. Working Conference on Mining Software Repositories*, pp. 121–130, 2013.

[12] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *Transactions on Software Engineering*, Vol. 20, No. 6, pp. 463–475, 1994.

[13] Raoul Gabriel Urma and Alan Mycroft. Source-code queries with graph databases – with application to programming language usage and evolution. *Science of Computer Programming*, Vol. 97, No. Part 1, pp. 127–134, 2015.

[14] Coen D. Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. The SOUL tool suite for

querying programs in symbiosis with eclipse. In *Proc. International Conference on Principles and Practice of Programming in Java*, pp. 71–80, 2011.

[15] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .QL for source code analysis. In *Proc. International Working Conference on Source Code Analysis and Manipulation*, pp. 3–16, 2007.

[16] Tal Cohen, Joseph Gil, and Itay Maman. JTL: The java tools language. In *Proc. International Conference on Object-oriented Programming Systems, Languages, and Applications*, pp. 89–108, 2006.

[17] Kris D. Volder. Jquery: A generic code browser with a declarative configuration language. In *Proc. International Symposium on Practical Aspects of Declarative Languages*, pp. 88–102, 2006.

[18] Markus Kimmig, Martin Monperrus, and Mira Mezini. Querying source code with natural language. In *Proc. International Conference on Automated Software Engineering*, pp. 376–379, 2011.

[19] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[20] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *Proc. International Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 432–448, 2004.

[21] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proc. International Conference on Software Engineering*, pp. 746–755, 2011.

[22] Mark Weiser. Program slicing. In *Proc. International Conference on Software Engineering*, pp. 439–449, 1981.

[23] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proc. International Conference on Software Maintenance*, pp. 44–53, 2003.

[24] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, Vol. 79, pp. 241–259, 2014.

[25] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proc. International Conference on Software Engineering*, pp. 422–431, 2013.

[26] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Historage: Fine-grained version control system for java. In *Proc. International Workshop on Principles of Software Evolution and the annual ERCIM Workshop on Software Evolution*, pp. 96–100, 2011.

[27] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian P Bailey. Software history under the lens: A study on why and how developers examine it. In *Proc. International Conference on Software Maintenance and Evolution*, pp. 1–10, 2015.

[28] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proc. Working Conference on Mining Software Repositories*, pp. 1–5, 2005.

[29] Reid Holmes and Andrew Begel. Deep intellisense: A tool for rehydrating evaporated information. In *Proc. Working Conference on Mining Software Repositories*, pp. 23–26, 2008.

[30] Francisco Servant and James A. Jones. History slicing: Assisting code-evolution tasks. In *Proc. International Symposium on the Foundations of Software Engineering*, pp. 43:1–43:11, 2012.

[31] Alexander W.J. Bradley and Gail C. Murphy. Supporting software history exploration. In *Proc. Working Conference on Mining Software Repositories*, pp. 193–202, 2011.

[32] P. Weissgerber, M. Pohl, and M. Burch. Visual data mining in software archives to detect how developers work together. In *Proc. Working Conference on Mining Software Repositories*, pp. 9–9, 2007.

[33] Qiang Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *Proc. International Workshop on Program Comprehension*, pp. 127–136, 2002.

[34] Y. Yoon, B. A. Myers, and S. Koo. Visualization of fine-grained code change history. In *Proc. IEEE Symposium on Visual Languages and Human Centric Computing*, pp. 119–126, 2013.

[35] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proc. International Conference on Software Engineering*, pp. 309–319, 2009.