情報処理学会論文誌 Vol.0 No.0 1–13 (??? 1959)

効率的な開発履歴理解のための Git に対するソースコード検索機能の統合

佐々木 美和[1,a)] 松本 真佑[1,b)] 楠本 真二[1,c)]

受付日 xxxx年0月xx日, 採録日 xxxx年0月xx日

概要：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．一方，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．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ことを提案する．こ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 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 といった過去のリビジョンの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．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 16 名を用いた有用性の確認実験を行う．

キーワード：版管理，履歴理解，ソースコード検索，MJgit，Git，構文抽象木

Integrating Source Code Search into Git Client for Effective Understanding of Code Change History

Miwa Sasaki[1,a)] Shinsuke Matsumoto[1,b)] Shinji Kusumoto[1,c)]

Received: xx xx, xxxx, Accepted: xx xx, xxxx

Abstract: In order to achieve effective development management, it is important to manipulate and understand the change histories of source code in a repository. Although general version control systems provide change history manipulations, these systems are restricted to line-based and textual operations such as grep and diff. As such, these systems cannot follow the syntax/semantics of the source code. While various studies have examined querying and searching source code, these methods cannot follow historical changes. The key concept of this paper is the integration of a source code search technique into Git commands that manipulate historical data in a repository. With this integration, it becomes possible to mask information that is not interested by the developers, and it is possible to narrow down efficient information. This paper presents MJgit, a prototype tool for achieving the above goal. In MJgit, developers can use search queries such as specifying method names and variable names for time direction operations such as git-show, git-diff, and git-log. As an evaluation, we conducted a performance experiment using actual software repositories. Also, we confirmed effectiveness and efficiency of our tool by user study where 16 subjects participated.

Keywords: Version control system, understanding code change history, source code search, MJgit, Git, abstract syntax tree

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[a)] m-sasaki@ist.osaka-u.ac.jp
[b)] shinsuke@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

1. はじめに

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 [7]．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 [22]．開発履歴の理解により，なぜコードが変更されたのか [19]，いつバグが混入したのか [14]，誰をデバッグ担当にすべきか [2]，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，grep や diff などの Unix 系コマンドに基づいた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．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，正規表現に基づいて強力なパターンマッチングも利用可能である [13]．しかしながら，これらのコマンドはそ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．

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 [20] によって引き起こされるバグ伝搬 [17] 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．

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 [5,6,8,9,15,18,23–25]．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 [4,10]．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 [11] 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．

本研究のキーアイデアは，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．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．

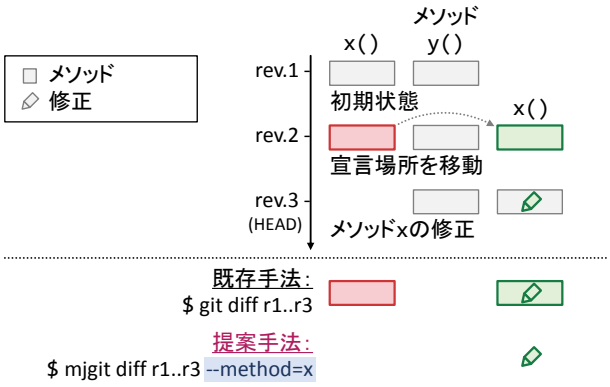
本研究の目的は，上記のアイデアを実現し，開発者に対する効率的な開発履歴の理解を実現することにある．本論文では，この目標を達成するための Git クライアントの拡張ツール，MJgit を実装し評価する．この提案ツールは，指定されたメソッド名や変数名などの構文情報に基づいて Java ファイルの開発履歴を絞り込むことができる．拡張クエリが指定されていない場合，MJgit は拡張対象である Git クライアントの Java 実装（JGit）と同じように動作するため，MJgit は JGit クライアントに対して完全な後方互換性を持つ．MJgit の評価実験として，実際のソフトウェアリポジトリを用いた性能評価実験，及び有用性を確認するための被験者実験を実施した．

2. 研究動機

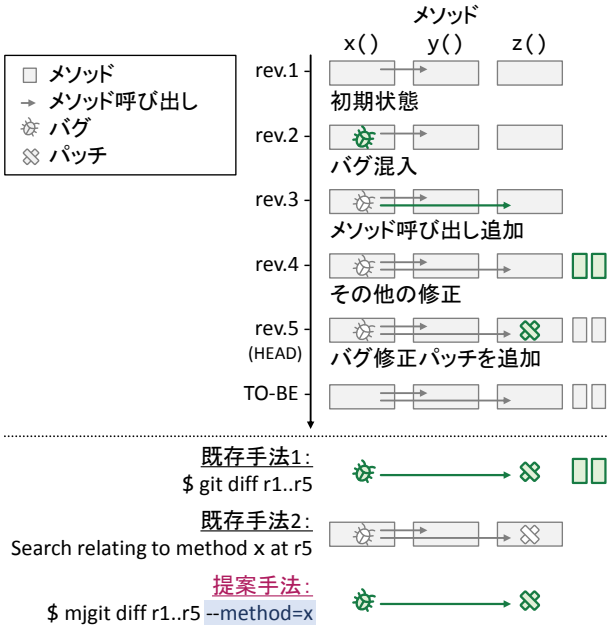
本研究の研究動機として，**図 1** に示す 2 つの履歴理解について，git-diff を用いたシナリオを説明する．各図には 2 個，または 3 個のメソッドの開発履歴が記されている．

2.1 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



(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: 研究動機

Fig. 1 Motivating examples

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可能である．

2.2 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．

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 [16] 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 [12]．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 [6, 8, 9, 15, 18, 23, 24] や変更波及解析 [1, 3, 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

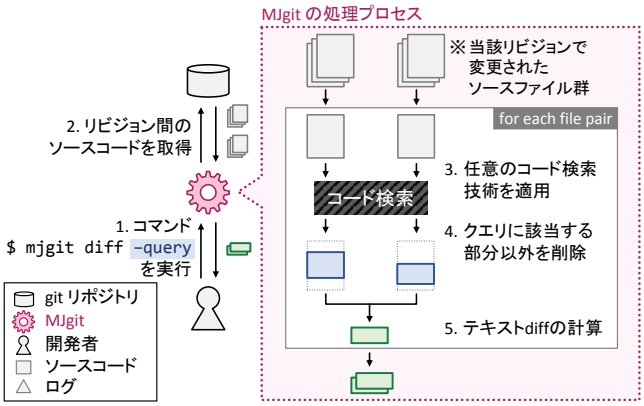


図 2: MJgit における git-diff コマンドの処理の流れ

Fig. 2 Processing flow of MJgit for git-diff command

-4cm -3.5cm

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．

3. 提案ツール：MJgit

3.1 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，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（3.2 節），コード検索技術（3.3 節），及び拡張対象となる Git コマンド（3.4 節）について説明する．

3.2 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 3.3 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 [13] 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．

3.3 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 [1,5,25]．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 [4,10]．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．

3.4 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

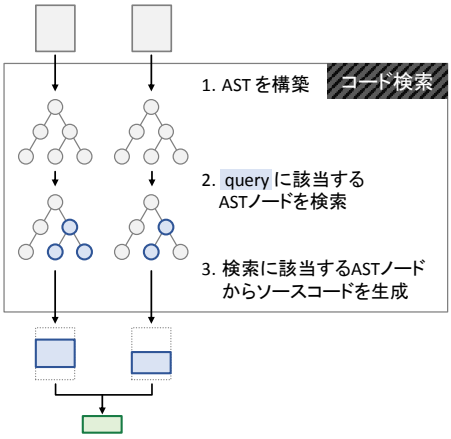


図 3: AST ベースのコード検索手法
Fig. 3 AST-based code search technique

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．

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

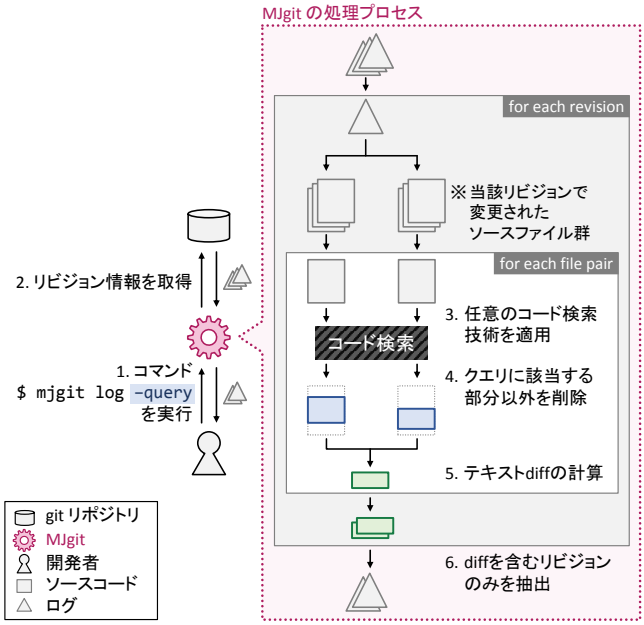


図 4: MJgit における git-log コマンドの処理の流れ
Fig. 4 Processing flow of MJgit for git-log command

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．

3.5 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]．

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

表 1: 拡張対象となる Git コマンド
Table 1 Extended git commands

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	≥1
git-blame	ファイルの最終更新を取得する	≥1

[*1] https://eclipse.org/jgit/
[*2] https://github.com/kusumotolab/m-sasaki_MJgit

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．

3.6　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．

3.6.1　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 2 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．

3.6.2　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，

表 2: MJgit の検索クエリと拡張コマンド
Table 2　Queries and extended git commands of current MJgit

拡張コマンド	git-log, git-show, git-diff
指定可能なクエリ（クエリ指定方法）	実行ステートメント（`exec-statement`）
	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`*）

```
$ jgit diff r1..r3
- public void x(int a, int b){
-     // aとbを入れ替える
-     int tmp = a;
-     a = b;
-     b = tmp;
- }

@@ -27,4 +20,13@@

+ public void x(int a, int b){
+     // aとbを入れ替える
+     int tmp = a;
+     a = b;
+     b = tmp;
+     a ++;
+     b --;
+ }
```

```
$ mjgit diff r1..r3 --method=x
  public void x(int a, int b){

      int tmp = a;
      a = b;
      b = tmp;
+     a ++;
+     b --;
  }
```

図 5: git-diff コマンド出力の比較
Fig. 5　Comparison of git-diff command outputs

```
$ jgit log --name-status
commit 3e383445c7f82
Author: miwa <miwa@osaka.ac.jp>
Date:   Sat Jul 28 08:29:05 2018

    メソッドxにnullチェックを追加した
src/Main.java

commit c5afd3dae1b7e
Author: shin <shin@osaka.ac.jp>
Date:   Fri Jul 27 17:04:06 2018

    コピーライトの修正
src/Main.java
src/Logger.java

commit 058efd8ae793c
Author: miwa <miwa@osaka.ac.jp>
Date:   Fri Jul 27 10:14:58 2018

    変数iの計算に使用されている
    マジックナンバーを定数に変更
src/Logger.java
```

```
$ mjgit log --variable=i
commit 058efd8ae793c
Author: miwa <miwa@osaka.ac.jp>
Date:   Fri Jul 27 10:14:58 2018

    変数iの計算に使用されている
    マジックナンバーを定数に変更
```

図 6: git-log コマンド出力の比較
Fig. 6　Comparison of git-log command outputs

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．

4. 性能評価実験

4.1 実験の目的

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．

4.2 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．

表 3: 実験対象プロジェクトの概要
Table 3 Summary of subject projects

	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

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．

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．

4.3 実験結果

4.3.1 git-diff の結果

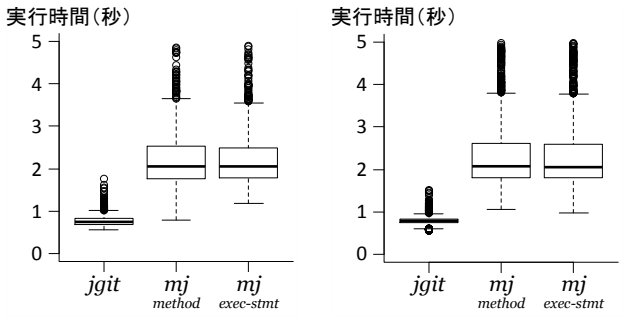
git-diff の実行結果を**図 7** に示す．3 つの箱ひげ図はそれぞれのクエリを指定した時の git-diff の実行時間を表している．ただし，実行時間が 5 秒を超えていた 2 つの外れ値は図から除外している．

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

表 4: 比較した 3 つのコマンド
Table 4 Compared three commands

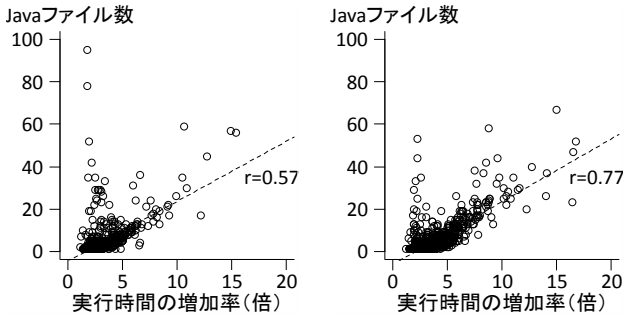
ラベル	実行したコマンド
jgit	`$ jgit diff A..B`
	`$ jgit log`
mj_{method}	`$ mjgit diff A..B --method=main`
	`$ mjgit log --method=main`
$mj_{exec\text{-}stmt}$	`$ mjgit diff A..B --exec-statement`
	`$ mjgit log --exec-statement`



(a) JUnit4 (b) Log4j

図 7: git-diff に対する実行時間の比較

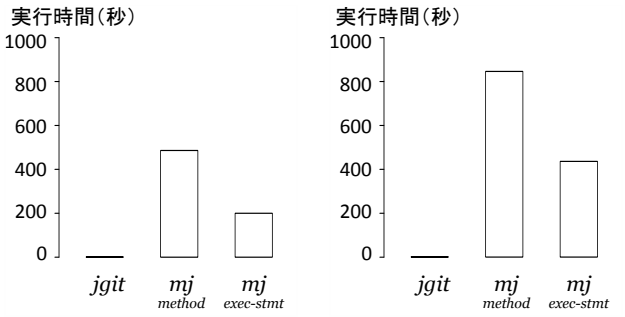
Fig. 7 Comparison of execution times for git-diff commands



(a) JUnit4 (b) Log4j

図 8: 実行時間の増加率と Java ファイル数の相関

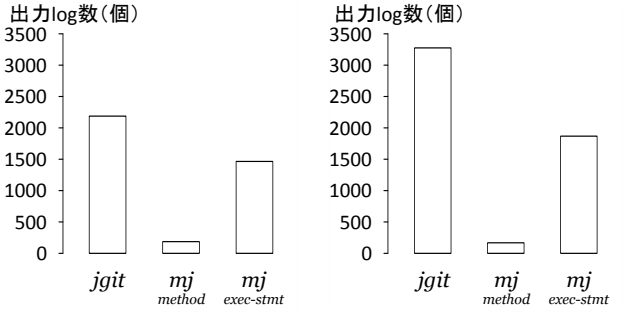
Fig. 8 Relationship between performance reduction rate and number of java files



(a) JUnit4 (b) Log4j

図 9: git-log に対する実行時間の比較

Fig. 9 Comparison of execution times for git-log commands



(a) JUnit4 (b) Log4j

図 10: git-log で出力されたログ数の比較

Fig. 10 Comparison of number of logs for git-log commands

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．

4.3.2 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．

4.4 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．

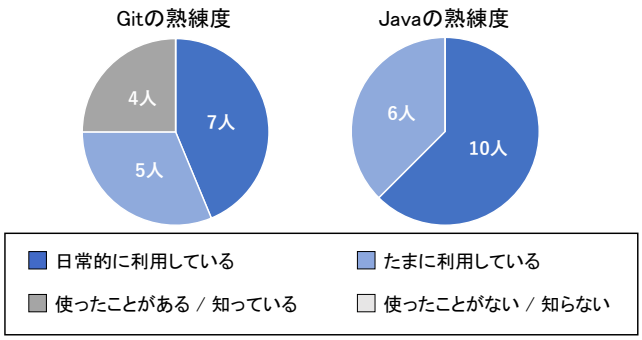


図 11: 被験者の熟練度
Fig. 11 Skill of subjects

5. 被験者実験

5.1 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．

5.2 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．

5.3 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．

$Task_A$ 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．

5.4 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

表 5: タスクの実行順序と MJgit 有無の割り当て（一部）
Table 5 Execution order and allowance for MJgit usage for each task

ID	変更内容	実行順序	MJgit の有無	
			X	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．

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．

5.5 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：5.4 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．

5.6 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）
- 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

5.7 実験結果

5.7.1 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．

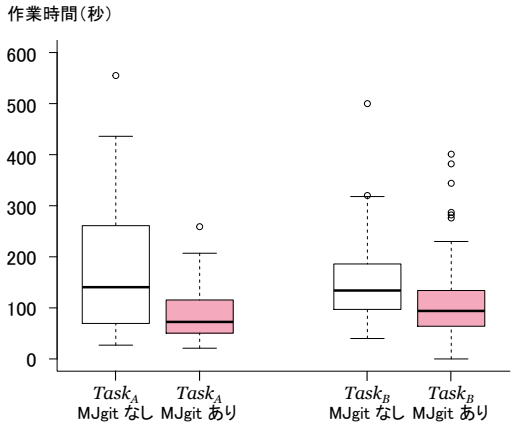


図 12: タスク完了時間の比較
Fig. 12 Comparison of completion time

5.7.2 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，

表 6: 被験者ごとのタスク完了時間の比較
Table 6 Comparison of completion time for each subject

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

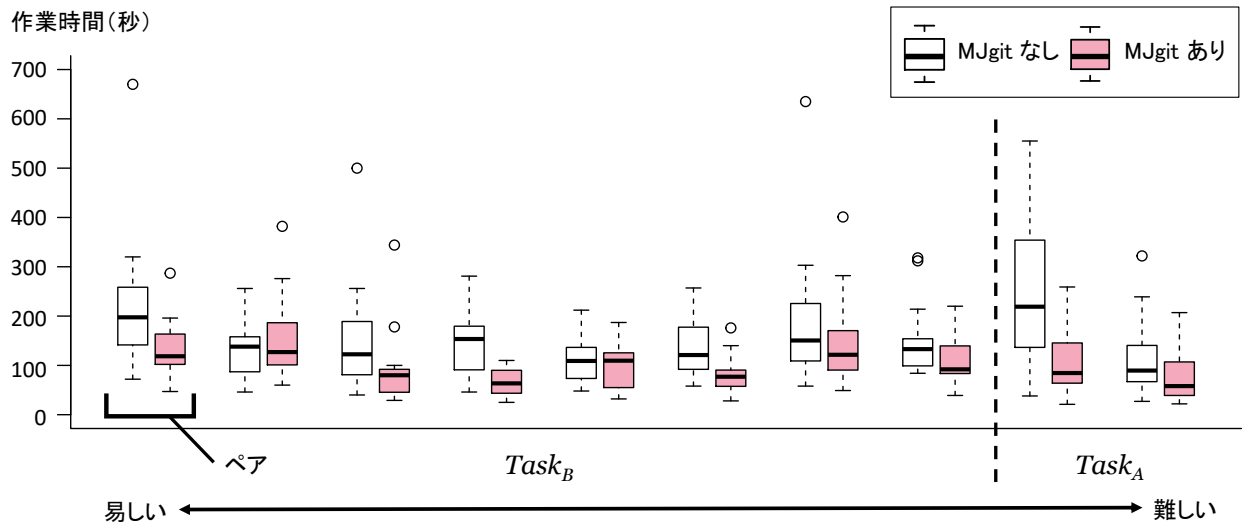


図 13: 個々のタスクに対する完了時間の比較
Fig. 13 Comparison of completion time for each task

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．

5.7.3 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．

5.7.4 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．

> 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．

5.8 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

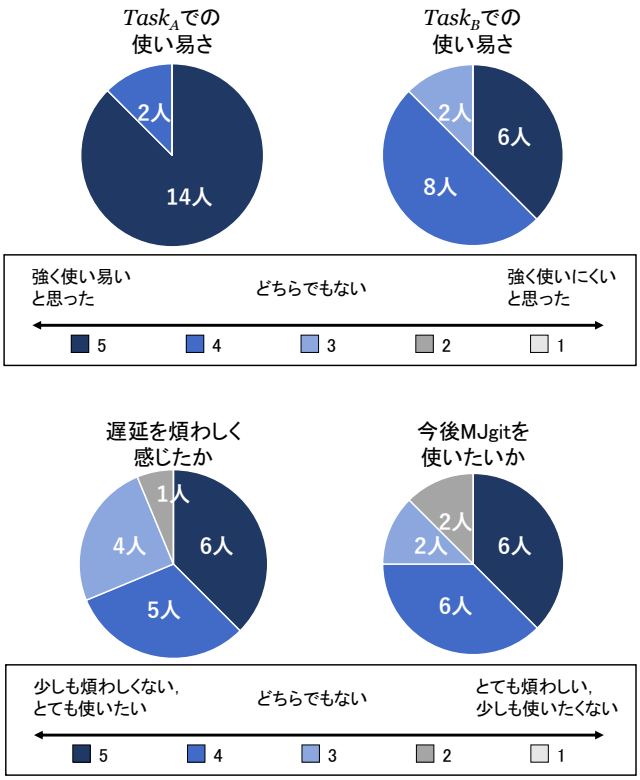


図 14: アンケート結果
Fig. 14　Interview results

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．

6. 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．

7. おわりに

本論文では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．

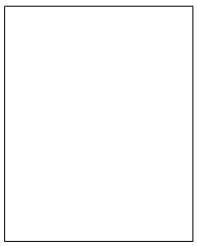
現在，MJgit はメソッド名と変数名の指定，及びプログラム構文の指定をサポートしているが，スコープの指定といったより高度な構文情報の指定や，拡張対象コマンドでありながら MJgit でサポートされていない git-blame の実装は今後の課題である．

謝辞　本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：16H02908，18H03222）の助成を得て行われた．

参考文献

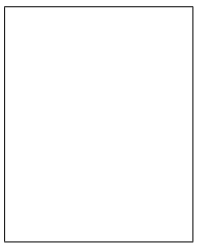
[1] Acharya, M. and Robinson, B.: Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems, *Proc. International Conference on Software Engineering*, pp. 746–755 (2011).
[2] Anvik, J., Hiew, L. and Murphy, G. C.: Who Should Fix This Bug?, *Proc. International Conference on Software Engineering*, pp. 361–370 (2006).
[3] Arnold, R. S.: *Software Change Impact Analysis*, IEEE Computer Society Press (1996).
[4] Bajracharya, S., Ossher, J. and Lopes, C.: Sourcerer: An infrastructure for large-scale collection and analysis of

open-source code, *Science of Computer Programming*, Vol. 79, pp. 241–259 (2014).
[5] Binkley, D. and Harman, M.: A large-scale empirical study of forward and backward static slice size and context sensitivity, *Proc. International Conference on Software Maintenance*, pp. 44–53 (2003).
[6] Cohen, T., Gil, J. and Maman, I.: JTL: The Java Tools Language, *Proc. International Conference on Object-oriented Programming Systems, Languages, and Applications*, pp. 89–108 (2006).
[7] D. Alwis, B. and Sillito, J.: Why are software projects moving from centralized to decentralized version control systems?, *Proc. Workshop on Cooperative and Human Aspects on Software Engineering*, pp. 36–39 (2009).
[8] D. Roover, C., Noguera, C., Kellens, A. and Jonckers, V.: The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse, *Proc. International Conference on Principles and Practice of Programming in Java*, pp. 71–80 (2011).
[9] de Moor, O., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Sereni, D. and Tibble, J.: Keynote Address: .QL for Source Code Analysis, *Proc. International Working Conference on Source Code Analysis and Manipulation*, pp. 3–16 (2007).
[10] Dyer, R., Nguyen, H. A., Rajan, H. and Nguyen, T. N.: Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories, *Proc. International Conference on Software Engineering*, pp. 422–431 (2013).
[11] Hata, H., Mizuno, O. and Kikuno, T.: Historage: Fine-grained Version Control System for Java, *Proc. International Workshop on Principles of Software Evolution and the annual ERCIM Workshop on Software Evolution*, pp. 96–100 (2011).
[12] Herzig, K. and Zeller, A.: The Impact of Tangled Code Changes, *Proc. Working Conference on Mining Software Repositories*, pp. 121–130 (2013).
[13] Hunt, J. W. and Szymanski, T. G.: A Fast Algorithm for Computing Longest Common Subsequences, *Communicaitons of the ACM*, Vol. 20, No. 5, pp. 350–353 (1977).
[14] Kim, S., Zimmermann, T., Pan, K. and Whitehead, E. J. J.: Automatic Identification of Bug-Introducing Changes, *Proc. International Conference on Automated Software Engineering*, pp. 81–90 (2006).
[15] Kimmig, M., Monperrus, M. and Mezini, M.: Querying source code with natural language, *Proc. International Conference on Automated Software Engineering*, pp. 376–379 (2011).
[16] Meyer, M.: Continuous Integration and Its Tools, *IEEE Software*, Vol. 31, No. 3, pp. 14–16 (2014).
[17] Mondal, M., Roy, C. K. and Schneider, K. A.: Bug Propagation through Code Cloning: An Empirical Study, *Proc. International Conference on Software Maintenance and Evolution*, pp. 227–237 (2017).
[18] Paul, S. and Prakash, A.: A framework for source code search using program patterns, *Transactions on Software Engineering*, Vol. 20, No. 6, pp. 463–475 (1994).
[19] Rastkar, S. and Murphy, G. C.: Why Did This Code Change?, *Proc. International Conference on Software Engineering*, pp. 1193–1196 (2013).
[20] Rattan, D., Bhatia, R. and Singh, M.: Software clone detection: A systematic review, *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199 (2013).
[21] Ren, X., Shah, F., Tip, F., Ryder, B. G. and Chesley, O.: Chianti: A Tool for Change Impact Analysis of Java Programs, *Proc. International Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 432–448 (2004).
[22] Sun, X., Li, B., Leung, H., Li, B. and Li, Y.: MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks, *Information and Software Technology*, Vol. 66, pp. 1–12 (2015).
[23] Urma, R. G. and Mycroft, A.: 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).
[24] Volder, K. D.: Jquery: A generic code browser with a declarative configuration language, *Proc. International Symposium on Practical Aspects of Declarative Languages*, pp. 88–102 (2006).
[25] Weiser, M.: Program Slicing, *Proc. International Conference on Software Engineering*, pp. 439–449 (1981).



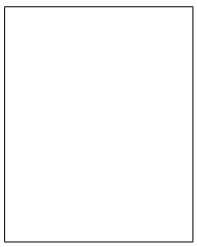
佐々木 美和

平成 29 年大阪大学基礎工学部情報科学科卒業．同年より同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在学中．



柗本 真佑 （正会員）

平成 22 年奈良先端科学技術大学院大学博士後期課程修了．同年神戸大学大学院システム情報学研究科特命助教．平成 28 年大阪大学大学院情報科学研究科助教．博士（工学）．エンピリカルソフトウェア工学の研究に従事．



楠本 真二 （正会員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部助手．平成 8 年同講師．平成 11 年同助教授．平成 14 年同大学大学院情報科学研究科助教授．平成 17 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．IPSJ，IEICE，JSSST，IEEE，JFPUG，PM 学会，SEA，各会員．