CodeGlass: GitHubのプルリクエストを活用したコード断片のインタラクティブな調査支援システム

柴藤大介[1,a)]　有薗拓也[1,b)]　宮崎章太[1,c)]　矢谷浩司[1,d)]

概要：ソフトウェア開発においてソースコードの理解は必要不可欠である．特に，実際の開発においてはコードの一部分（コード断片）の理解が必要となることが多い．我々が行ったインタビュー調査では，GitHub のプルリクエストの説明文とコメントに，コード理解において有用な情報が含まれていることが明らかとなった．しかし既存のシステムでは，コード断片の理解支援のためにプルリクエストが十分に活用されていない．そこで我々は，ユーザが選択したコード断片に関連する過去のプルリクエストを抽出し，ユーザに提供する CodeGlass を開発した．CodeGlass では，プルリクエストの説明文を解析し，実装内容や開発背景に関する文章をインターフェース上で強調して表示することが可能となっている．CodeGlass のアルゴリズムにより，選択されたコード断片が過去のバージョンにおいて分裂していた場合にも，関連する過去のプルリクエストをユーザに提供することができる．我々が行った CodeGlass の定量的および定性的評価の結果，コード断片の理解や専門的用途における CodeGlass の有用性が確認された．

1. はじめに

ソフトウェア開発においてソースコードの理解は必要不可欠である．ソフトウェア開発者はコードを書くだけでなく，同じチームの他の開発者によるコード変更をレビューしたり，開発プロジェクトのスケジュールや進捗を管理したりする必要がある．その全てのタスクにおいて，正確なコード理解が常に必要となる．このため，ソフトウェア開発におけるコード理解の重要性は広く認知されている．

実際のソフトウェア開発では，ソースコード全体ではなく，コードの一部分（コード断片）の理解が必要となることが多い．例えば，開発者はバグを修正するために，関連するコード断片（関数やクラスなど）の現状を把握してからコードの修正を行う．コードレビューにおいても同様に，主に変更のあったコード断片に対して問題がないことを確認する．このようにコード断片の理解はソフトウェア開発において重要であり，コード断片の実装内容や開発背景といった情報をユーザに提供することができれば，ソフトウェア開発を支援できると考えられる．

[1] IIS-Lab
東京大学
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
a) shibato@iis-lab.org
b) arizono@iis-lab.org
c) miyazaki@iis-lab.org
d) koji@iis-lab.org

近年のソフトウェア開発では，GitHub[*1]という多人数開発を支援するコードホスティングサービスが広く利用されている．GitHub では，コード変更の管理とコードレビューを支援する機能であるプルリクエストを用いたソフトウェア開発が推奨されている．他の開発者にコード変更の意図や内容を正しく伝えるためには，プルリクエストにコード変更の説明文を記すことが重要とされている．この説明文

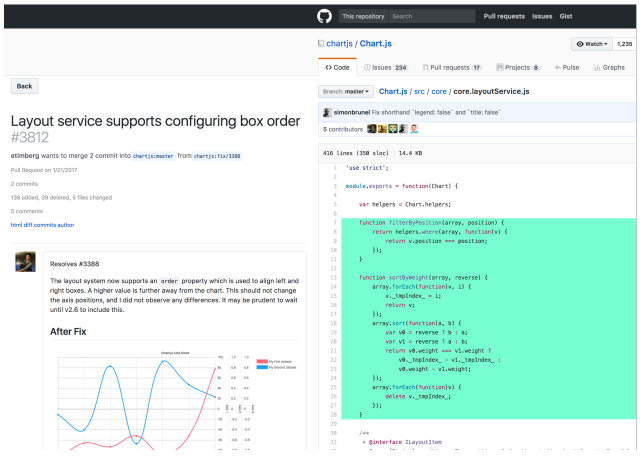


図 1: CodeGlass 上でユーザは選択したコード断片に関連する過去のプルリクエストを参照することができる．プルリクエストの説明文に含まれる実装内容や開発背景といった情報は，ユーザのコード理解を支援することができる．

[*1] https://github.com

には，コード変更の実装内容や開発背景といった，コード変更を理解するための情報を記載することが求められる．我々が行ったプログラマを対象としたインタビュー調査では，プルリクエストに含まれる情報がコード断片の理解において有用であることが示唆された．しかし，プルリクエストは主にソフトウェア開発の効率化を目的として利用されており，コード断片の理解を支援するためには十分に活用されていない．

そこで本研究では，コード断片に関連する情報を提供する CodeGlass というシステムを開発した（図 1）．CodeGlass はユーザが選択したコード断片に関連する過去のプルリクエストを抽出し，ユーザに提供する．さらに，プルリクエストの説明文を解析し，実装内容や開発背景に関する文章をインターフェース上で強調して表示することが可能となっている．LHDiff [1] を改良したアルゴリズムにより，選択されたコード断片が過去のバージョンにおいて分裂していた場合にも，CodeGlass は関連する過去のプルリクエストを提供することができる．学生による CodeGlass の定量的評価により，コード断片の理解のための情報収集において，CodeGlass が有用であることが分かった．さらにプログラマを対象とした定性的評価を行った結果，専門的用途における CodeGlass の有用性が明らかとなった．本研究がソフトウェア工学とヒューマン・コンピュータ・インタラクションの分野にもたらす貢献は以下の通りである．

- ユーザが選択したコード断片に関連する過去のプルリクエストを表示し，実装内容や開発背景に関する文章を強調する CodeGlass インターフェースの構築
- LHDiff [1] の改良による，過去のバージョンにおけるコード断片の追跡精度の向上
- コード断片の理解における CodeGlass の有用性の定量的・定性的評価

2. 関連研究

2.1 コード変更の理解支援

GitHub をはじめとする多くのソフトウェア開発管理システムでは，コード差分を記録することでコード変更の追跡が可能となっている．加えて，開発管理システムの多くがバージョン管理の機能を提供しており，コード変更の管理やコードレビューのために広く活用されている．先行研究では，ソフトウェア開発におけるコード変更を理解することを支援するための様々な手法が調査されてきた．

ソースコードの多くの部分に影響を与えるコード変更がバグを含んでいた場合，システムに重大な問題が生じてしまう恐れがある．Buckner ら [3] が開発した JRipples では，構文解析を行うことにより，コード変更にバグが含まれていた際にソースコードの中で影響を受ける可能性があるクラスをユーザに提示することができる．Zhang ら [22] が開発した CRITICS は，与えられたコード変更と類似した過去のコード変更を提示することにより，コード変更にバグが含まれることを防ぐために着目すべき点を開発者がより正確かつ効率的に発見することが可能となった．JRipples や CRITICS がコードレビュアーのコード理解の支援を目的としている一方で，正確なコードレビューを行うためにはコード変更の説明が詳細に記述されていることも重要である．そこで Buse ら [4] は，コード変更からその説明文を自動生成する手法を実装した．ユーザ評価の結果，既存のコミットメッセージの 89 %は Buse らの手法を用いて生成できうる内容であることが分かった．同様に，Linares-Vasquez ら [13] が開発した ChangeScribe もまた，コミットメッセージとして使用可能な説明文をコード変更から自動生成するシステムである．ChangeScribe には説明文にコード変更の内容だけでなくその背景も含めるために，複数の情報抽出アルゴリズムが実装されている．

本研究の目的は，前述の先行研究とは異なり，コード断片の理解支援である．コード理解においては，コード変更に付随する説明文が有益であることが既に明らかとなっている [5], [19]．本研究はコード断片に関連する過去のコード変更を抽出し，それらに含まれる説明文をユーザに提供することで，コード断片の理解支援を目指す．

2.2 ドキュメントの作成と活用の支援

ドキュメントはコード理解における有用な情報源であり，その重要性は広く認知されている．しかし実際のソフトウェア開発プロジェクトではドキュメントを作成する時間が不足していることが多い [6]．そこでドキュメントの作成支援や，既存のドキュメントを活用したコード理解の支援に関する研究が広く行われてきた．

Sridhara ら [16] は Java の関数の要約を自動生成する SWUM というシステムを実装した．SWUM はソースコード全体と関数の構造的関係性を解析することで，ドキュメントとして使用可能な関数の説明文を自動生成することができる．McBurney と McMillan [14] は SWUM をさらに改良し，与えられた関数のコード内での使用例も追加できるようにした．これらの手法により，自動生成される説明文に文脈を考慮した情報を追加することが可能となった．また Stylos ら [17] は，Java の API のドキュメントが簡単に検索できるようになる Jadeite を開発した．ユーザは Jadeite 上でドキュメントにエイリアスとなるクラス名や関数名を追加することができる．このエイリアスは実際の Java の API と紐づいており，ユーザは API の名前だけでなくそのエイリアスでも API のドキュメントを検索することができる．

既存のドキュメントをコード理解のために活用する先行研究も多く存在する．Stack Overflow の投稿には API の使用方法が記述されていることが多いため，ユーザは独自にドキュメントを作成・管理することなく API の使用方法を

参照できる可能性がある．そこで，Subramanian ら [18] は API と Stack Overflow 上の投稿のコード例を紐付けるアルゴリズムを実装した．また Dekel と Herbsleb [7] は，膨大なドキュメントからコード理解において重要な情報を抽出することでコード変更のデバッグの成功率を改善できることを示した．さらに Treude ら [20] は，教師あり学習を用いて Stack Overflow から現在のドキュメントには含まれていない情報を抽出する手法を実装し，教師なし学習を用いた手法と比較してより多くの有用な情報を Stack Overflow から抽出できることが明らかとなった．

このように既存の情報源を活用したコード理解の支援に関する研究が広く行われてきたが，情報源としての GitHub のプルリクエストの有用性はまだ明らかとなっていない．プルリクエストは GitHub を用いた開発において一般的に使用されており，プルリクエストの説明文はドキュメントとして活用できる可能性がある．コード理解を支援するための情報源としての GitHub のプルリクエストの可能性を示すことも本研究の目的の 1 つである．

3. インタビュー調査

3.1 プログラマの要望調査

実際のソフトウェア開発でのコード理解における障壁を調査するために，我々は IT 企業に所属する 5 人のプログラマ（PA1–5，全員男性）にインタビューを行った．実験参加者は全員，ソフトウェア開発プロジェクトにおいて GitHub を日常的に利用していた．インタビューでは実験参加者がコード理解をどのように行い，その際にどのような障壁が存在するのかを重点的に尋ねた．

開発背景を理解する難しさ

チーム開発においては，実装やコードレビューのために他の開発者が実装したコードを理解する必要があるが，他人が書いたコードの理解には大変な労力を要する．また，実際のソフトウェア開発におけるコード理解では，コードの実装内容だけでなく，なぜ実装されたのかという開発背景も把握することが重要である．PA1 は，コードレビューにおいて背景知識の不足がコード理解における問題の一つであることを以下のように指摘した．

> プロダクトの背景がわからないことが課題となっていて，長く見ている人が初めてそのチーム内でレビューできるようになっていて，まだやっぱり新入りの人がコードレビューできるような状況までドキュメントが整備されていないし，やっぱり背景理解が難しい．[PA1]

コード理解のための情報源としてのプルリクエスト

実験参加者らはコード変更をレビュアーに説明するために，GitHub の機能の一つあるプルリクエストを活用していた．プルリクエストでは，説明文に目的や内容を記述することで，コード変更をより分かりやすく他の開発者に伝えることが可能となっている．PA3 と PA5 は，コードが実装された理由を後から確認するために，開発背景といった情報を含むプルリクエストを参照することがあると指摘した．

> それは，実装を見ててこれってどういう意図でこうなってんのとかが，気になった時は，結構プルリクエスト上で議論がなされていることが多いので，それを探しに行くことはあります． [PA3]

> 僕はわりと探っちゃうタイプなので，昔どういうことあったんだろうみたいなのは追っちゃう感じですね．[PA5]

以上より，プルリクエストの説明文にはコード理解において有用な情報が含まれている可能性があり，実際に開発現場で利用されることがあることを確認した．この発見により，プルリクエストをコード理解に活用するシステムを構築することを考えるに至った．

3.2 既存技術の問題点調査

続いて我々は，既存のツール, 特に tig[*2] と recursive-blame[*3] がコード理解の支援においてどのように不十分であるかを調査した．tig は git blame のコマンドを再帰的に実行できるコマンドラインインターフェースであり，recursive-blame は特定のキーワードを含む過去のコード変更を抽出することができるコマンドである．本調査では GitHub のプルリクエストを用いた開発経験のある 7 人の学生（PB1–7）にインタビューを行った．インタビューではまず tig または recursive-blame を用いてコード断片の調査を行ってもらい，その後 2 つのツールに対する使用感を尋ねた．その結果，既存のツールでは過去のプルリクエストをコード理解のために活用することができない 2 つの理由が明らかとなった．

コード断片追跡機能の限界

実験参加者らは，既存のツールではコード断片が削除または移動された際に，開発履歴の情報が抽出できなくなることを指摘した．これはコード断片が削除または移動された場合，既存のツールでは単にコード断片が存在しないことのみが表示されるからである．

> 違う名前や使い方が変わったときに，じゃあ何になったんだろうというってのを探すためには，前の名前しか知らないときは自分で探さないといけない．削除なのか，変更なのかわからないのが今のツールではわからない． [PB4]

インタラクションの限界

tig と recursive-blame では，追跡するコードの行やキー

[*2] https://github.com/jonas/tig
[*3] https://github.com/scottgonzalez/recursive-blame

ワードを繰り返し指定することで，関連する過去のプルリクエストを特定することができる．しかし，プルリクエストを特定した際に提示される情報には，説明文やコメントが含まれていないため，コードの理解につながると限らないことが指摘された．

> 全部やった後にさ，*description* なかったらさ，なんやねんって感じやん．多分 *description* が一番大事だから，はずれだったらすぐ次のを見せてほしい． [PB2]

tig ではコード中の 1 行を，recursive-blame ではキーワードもしくはコード 1 行のみを検索に利用できるが，コード理解においては複数行の選択が必要であることを指摘していた．

> コード読んでて，この行だけわかんないってなることはないから． [PB1]

> この *if* の中みたいな，意味のある単位で見たい．*if* の行自体は別にそれほど興味ないし中身が大事かな，みたいな． [PB5]

3.3 まとめ

我々が行った関連研究の調査では，コード理解に重要な情報は以下の 5 つに大別されることがわかった．

- **Execution**: そのコードが何を実行するのか [9].
- **Rationale**: 何故そのコードが実装されたのか [9].
- **History**: そのコードがどう変更されてきたのか [21].
- **Contributor**: 誰がそのコードの実装に関わったのか [8].
- **Usage**: そのコードがどこでどう使用されるのか [10].

我々のインタビュー調査により，プログラマが特にコード変更に関する詳細な説明と開発背景の知識をプルリクエストに求めていること，さらに既存のツールではそれが十分に支援されないことが明らかとなった．したがって，我々が構築するシステムにおいては特に Execution と Rationale に関する情報収集を支援すべきであると考えた．

4. CodeGlass インターフェース

CodeGlass は GitHub のプルリクエストを用いてコード断片の理解を支援するインタラクティブなツールである．本研究におけるコード断片は，ソースコード中の一部の連続したコードである，と定義する．即ち，コード断片は一行のコードにも，関数やクラス，あるいはファイル全体のコードにもなり得る．CodeGlass はユーザが選択したコード断片に関連する過去のプルリクエストを抽出し，インターフェースを通じてユーザに提供する．

図 2 に CodeGlass のインターフェースを示す．ユーザはマウスドラッグでコード断片を指定する（緑でハイライト）ことで，関連する過去のプルリクエストやコミットを確認できる（図 2a）．図 2a (1) に示すボタンから，関連する過去のプルリクエストの表示順を時系列順，実装内容（Execution）に関する情報が多い順，開発背景（Rationale）に関する情報が多い順に並び替えることができる．これにより，どのように開発が進められてきたか（History）という情報に加えて，実装内容や開発背景を含むと考えられるプルリクエストを強調して提示することも可能となっている．

それぞれのプルリクエストには，タイトルやプルリクエストの ID，含まれるコミット数，コードの変更行数，そしてコメント数が表示されている．また，GitHub 上でのプルリクエストや，コード差分，レビューコメント，そしてプルリクエスト作成者へのウェブページへのリンクも示されている．加えて，図 2a (3) に示すように，選択されたコード断片と一致する可能性があるコード断片が過去のバージョンの 1 つで複数ある場合，そのコミットへのリンクも表示する．ユーザがプルリクエストのタイトルをクリックすると，CodeGlass はその詳細を表示する（図 2b）．詳細画面では，プルリクエストの基本的な情報に加えてレビューコメントも表示される．また，図 2b (4) に示すように，プルリクエストの説明文中で実装内容や開発背景に関する情報であると推定された箇所は，色付きの文字で強調される．

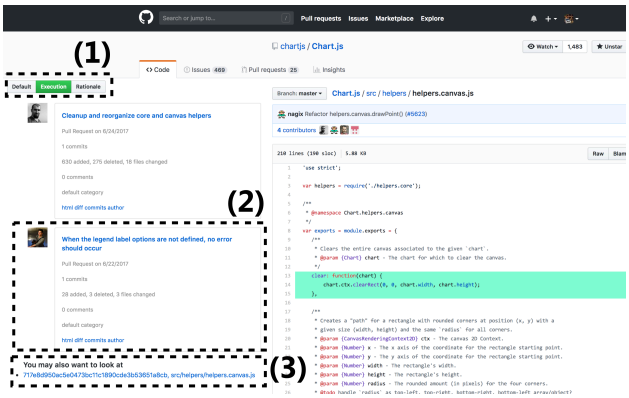
CodeGlass においてユーザは，クラスや関数といった単位に縛られることなく自由にコード断片を選択し，関連する過去のプルリクエストを調べることができる．特にユーザが大きなクラスや関数のコードを調査する際には，自由にコードを分割して調べることができるため有用であると考える．CodeGlass はプログラミング言語に依存することなく動作するため，GitHub 上の全てのオープンなリポジトリに対して，CodeGlass のサーバにリポジトリをクローンすることで利用可能となっている．アクセス制限などを設定することで，技術的にはプライベートリポジトリにも対応することができる．

5. 実装内容と開発背景を含む説明文の抽出手法

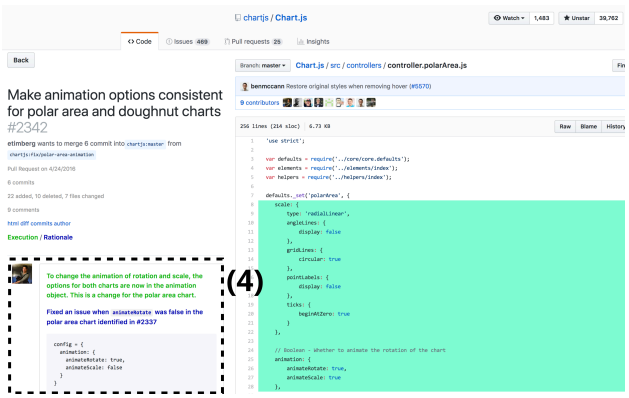
プログラマを対象としたインタビューにより，コード断片の理解においてプルリクエストに含まれる情報が有用であることがわかった．また，コード断片を理解する上で，そのコード断片の実装内容だけでなく，開発背景も理解することが重要であることが明らかとなった．しかし既存のシステムでは，コード変更の説明文に含まれる情報を，実装内容や開発背景といった情報に分類することは行われていない．そこで我々は，プルリクエストの説明文内の文章の中から，関連するコード変更の実装方法と関連するコード変更の実装理由を抽出するための分類器を実装した．

5.1 データセットの構築

開発者が実装内容や開発背景といった重要な情報を漏れなくプルリクエストに記述することを促すために，GitHub



(a) 関連するプルリクエストの一覧画面.



(b) プルリクエストの詳細画面.

図 2: CodeGlass のインターフェースは，ユーザが選択したコード断片（緑にハイライトされたコード）に関連する過去のプルリクエストの一覧を表示する．抽出されたプルリクエストの表示順は，時系列順，実装内容（Execution）に関する情報が多い順，開発背景（Rationale）に関する情報が多い順に並べることができる (1)．各プルリクエストにはタイトル，コミット数，コードの追加および削除行数，コメント数，そして関連する各ウェブページへのリンクが表示される (2)．さらに，選択されたコード断片と一致する可能性があるコード断片が過去のバージョンの 1 つで複数ある場合，そのバージョンにおけるソースコードへのリンクが表示される (3)．プルリクエストのタイトルをクリックすると，その詳細情報を確認できる．詳細画面では，プルリクエストの情報に加えて，レビューコメントも表示される．さらに，Execution に分類される文章は緑色で，Rationale に分類される文章は青色で表示される (4).

のリポジトリにはマークダウンで書かれたプルリクエストのテンプレートを登録することができる [*4]．テンプレートに沿ってプルリクエストの説明文を記述することで，他の開発者がコード変更を理解するために必要な情報を網羅したプルリクエストを作成することができる．テンプレート内のヘッダーには，実装内容（Execution）に該当するヘッダー（“Implementation Details” など）や，開発背景（Rationale）に該当するヘッダー（“Motivation and Context” など）などが多く採用されている．そのようなヘッダーの下に記述されている文章を抽出することで，分類器の学習に必要なラベルづけされたデータセットを構築することとした．

以上のようなデータを GitHub から収集するために，我々は次の条件を満たすリポジトリに存在するプルリクエストのデータを抽出した．

条件 1　リポジトリのスター数が 100 以上であること．

条件 2　プルリクエストのテンプレートがリポジトリに登録されており，Execution に関係する節か Rationale に関係する節がテンプレート内に存在すること．

条件 1 はリポジトリの開発者数の目安として掲げたものである．スター数が大きいものほど GitHub 上での開発が盛んであり，条件 2 を満たすプルリクエストを多く取得できるだろうと考えた．また，条件 2 に適応することで，前述のようにラベルづけされたデータを収集することができる．

[*4] https://help.github.com/articles/creating-a-pull-request-template-for-your-repository

GitHub から条件 1 と条件 2 を満たすデータを収集した結果，26 のリポジトリと 4061 件のプルリクエストのデータを集めることができた．その 26 のリポジトリに登録されているプルリクエストのテンプレートについて，著者の 2 人が Execution または Rationale に該当するヘッダーを手動で抽出した．そして，それらのヘッダーの下に存在する文章を抽出することにより，Execution の文章を 3141 件，Rationale の文章を 3451 件を得た．さらに，抽出した文章内に含まれるマークダウン表記と絵文字を削除することで，分類器の学習用のデータセットを構築した．

5.2　分類器の実装と評価

分類器の実装にはサポートベクターマシンを用いた．学習に必要な特徴量は，文章の分類において一般的に使用されている bag-of-words，tf-idf，word2vec を用いた [11]．さらに，tf-idf と word2vec が抽出する情報が互いに補完的な役割を果たすことが報告されているため [12]，本研究においても word2vec を tf-idf により重み付けした特徴量（word2vec+tf-idf）も利用した．それぞれの特徴量を用いて分類器を学習し，10 分割交差検証法により分類器の精度を検証した結果，bag-of-words を用いた時の F 値の平均が 0.64，tf-idf を用いた時の F 値の平均が 0.68，word2vec を用いた時の F 値の平均が 0.72，word2vec+tf-idf を用いた時の F 値の平均が 0.73 であった．以上より word2vec+tf-idf を用いた時が最も精度が高かったため，word2vec+tf-idf を用いて学習を行った分類器を CodeGlass 上に実装した．

図 2a (1) に示すボタンから，ユーザはプルリクエスト

の表示順を時系列順，Execution に関する情報が多い順，Rationale に関する情報が多い順に並び替えることができる．さらにプルリクエストの詳細画面では，Execution および Rationale に 80%以上の確率で分類されると推定された文章がハイライトされる．図 2b (4) に示すように，Execution に分類される文章は緑色の，Rationale に分類される文章は青色の文字で表示することで，ユーザの情報収集を支援することが可能となっている．

6. コード断片追跡アルゴリズムの改良

6.1 LHDiff の改良方法

CodeGlass のサーバは，ユーザが選択したコード断片に関連する過去のコミットを抽出する必要がある．git のコマンドである blame を使うことで，コード断片を最後に変更したコミットを特定することができる．しかし，git のコマンドではコード断片が過去のバージョンにおけるソースコードのどこに存在するのかを知ることができない．したがって，ユーザが選択したコード断片に関連する過去のコミットを最新のものだけでなく全て抽出するためには，選択されたコード断片が過去のバージョンにおいてどこに存在するのかを特定する必要がある．

アルゴリズムを説明するために，まず被選択コード断片と目的コード断片を定義する．被選択コード断片（CP_S）は，ユーザが CodeGlass のインターフェース上で選択したコード断片である．また目的コード断片は，過去のバージョンにおける被選択コード断片と対応したコード断片である．アルゴリズムは CP_S の情報を受け取り，過去のバージョン内の目的コード断片の位置を推定する．ここで，正解目的コード断片（CP_T）と推定目的コード断片（$\widehat{CP_T}$）を新たに定義する．正解目的コード断片は過去のバージョンにおける CP_S にマッチするコード断片である．また，推定目的コード断片はアルゴリズムが推定した過去のバージョンにおける被選択コード断片を意味する．アルゴリズムが完全に正しく動作した場合，$\widehat{CP_T}$ と CP_T が一致する．

先行研究では，コード断片を過去のバージョンの中で追跡する手法が提案されており，それらを CodeGlass のサーバのアルゴリズムとして活用できる可能性がある．Asaduzzaman ら [1] が開発した LHDiff は，プログラミング言語非依存で動作するコード断片追跡アルゴリズムである．LHDiff は word-level fuzzy matching [15] を用いて CP_S と $\widehat{CP_T}$ との文字列的類似度を計算し，最も類似度が高かったものを CP_T として決定する．

しかし我々が LHDiff の事前調査を行った結果，コード変更が構造的変更（関数の統合や分裂，リファクタリングなど）を含む場合，LHDiff はコード断片の追跡に失敗することが分かった．この原因は，LHDiff では CP_T が一つしか存在しないと仮定されているからである．実際のコード変更履歴では，被選択コード断片が過去のバージョンにおいて他のコード断片と統合されていたり，分割されて実装されていたりする可能性がある．特に大規模なソフトウェア開発プロジェクトでは，ソースコードの構造的変更が繰り返し行われることが多いため，LHDiff をそのまま CodeGlass のサーバに実装すると，CodeGlass の実用性が制限されてしまう．そこで我々は新たに graceful matching を提案し，LHDiff に導入することでこの問題を解決することを試みた．

LHDiff は CP_T が常に一つであるという仮説の下，最も文字列的類似度が高いコード断片を $\widehat{CP_T}$ として決定する．我々は新たに，正解目的コード断片が一つに定まる可能性が高いことを示す閾値である DMT（Definitive Matching Threshold）を導入した．CP_S と $\widehat{CP_T}$ の候補の文字列的類似度が DMT よりも高かった場合，LHDiff と同様に $\widehat{CP_T}$ を決定する．しかし，過去にソースコードの構造的な変更や大規模な修正が行われていると，文字列的類似度が低くなり，CP_T を正確に一意に推測することが難しくなる．そこで，別の閾値である CMT（Candidate Matching Threshold）を導入する．DMT を上回る $\widehat{CP_T}$ が存在しなかった場合，CMT を上回るコード断片を全て，CP_T である可能性があるとしてインターフェースに返す．CodeGlass のインターフェースは，それらのコード断片を含むコミットを図 2a (3) のように表示する．ユーザはそのコミットをクリックして過去のバージョンにおけるソースコードを確認し，CP_S として選択し直すことで，引き続きコード断片の調査を継続することができる．

DMT と CMT の二つの閾値を設定するために，Chart.js のリポジトリに含まれる 49 個のコミットにおける，変更前と変更後のコード断片の一対をデータセットとして構築した．このデータセットでは LHDiff だけでは CP_T を特定することが難しい構造的変更からなるコミットのみを意図的に選定した．そして，それぞれのコード断片に対して CP_S を選択し，それに対応する CP_T のラベル付けを行った．さらに，CP_T と最も文字列的類似度の高い，CP_T ではないコード断片を IM（incorrect match）としてラベル付けした．

図 3 に CP_T（青）と IM（橙）の文字列的類似度の分布を示す．この分布から IM の類似度は 0.65 未満であることが分かる．また，CP_T の類似度は 0.4 より大きいことも分かる．したがって我々は，DMT を 0.65 に，CMT を 0.4 に設定した．

6.2 改良アルゴリズムの定量的評価

graceful matching が目的コード断片の特定の精度をどの程度改善するのかを明らかにするために，我々は GitHub の実際のコード変更データを用いた定量的評価を行った．評価に用いるデータセットは，1,000 人以上の開発者からスターを付けられており，かつデータセット構築時に最も新しく更新のあった Chart.js のリポジトリを選択した．

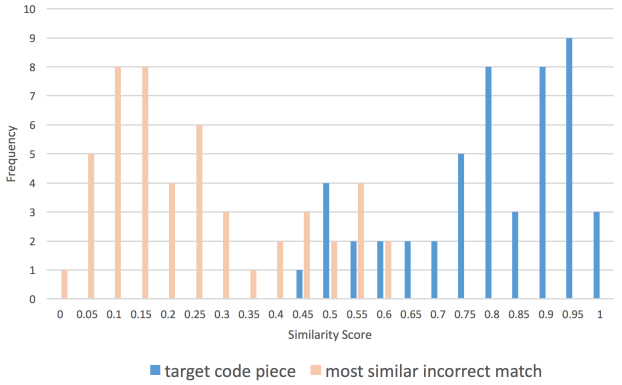


図 3: CP_T（青）と IM（橙）の文字列的類似度の分布．

はじめに Chart.js のリポジトリから 100 個のコード断片を選択し，それらを *Short*（9 行以下，34 個のコード変更），*Middle*（10 行以上 30 行以下，33 個のコード変更），そして *Long*（31 行以上，33 個のコード変更）の 3 つのグループに分類した．次にそれら 100 個の原コード断片の開発履歴を辿り，5 つ以上の過去のバージョンにおけるコード断片を抽出し，原コード断片と組み合わせる事で評価用のデータセットを構築した．そして，過去のコード断片を特定する精度を graceful matching の有無で比較評価した．

評価指標として適合率（P_{cp}）と再現率（R_{cp}）を用いた．l_c を $\widehat{CP_T}$ として正しく推定されたのコード行数，l_i を $\widehat{CP_T}$ として間違って推定されたコードの行数，l_n を CP_T に含まれるが $\widehat{CP_T}$ には含まれてないコードの行数とした時，適合率と再現率は $P_{cp} = \frac{l_c}{l_c+l_i}$，$R_{cp} = \frac{l_c}{l_c+l_n}$ と定義した．

図 4 が示すように，全ての条件において graceful matching を用いることで適合率と再現率が改善された．また，適合率及び再現率の結果と graceful matching の有無に対して重回帰分析を行った結果，適合率と再現率が有意に改善されたことが分かった．適合率における回帰係数は 0.058（$SE = 0.025$, $p < .05$），再現率における回帰係数は 0.051（$SE = 0.021$, $p < .05$）であった．さらに，*Short* グループの方が *Long* のグループと比較して，有意に高い適合率及び再現率であった（$p < .05$）．

graceful matching の評価を行った結果，コード断片の特定における適合率および再現率を約 5%改善できたことが分かった．また図 4 に示すように，graceful matching が頑健性に貢献することが示された．例えば，ある 1 つのデータでは，LHDiff のみの場合適合率と再現率がそれぞれ 0.30 と 0.40 であったが，graceful matching を適用することで両方ともに 0.80 まで改善したことが確認された．graceful matching はコード断片の特定の精度を改善することができるため，我々は graceful matching を CodeGlass に導入することとした．

7. 定量的ユーザ評価実験

7.1 実験内容

コードの Contributor と Usage に関する情報収集においては，既に有用なツールが広く使用されている（git-blame コマンドなど）．そこで本実験においては，Execution（実装内容），Rationale（開発背景），History（開発経緯）に関する情報収集に関する評価を重点的に行うこととした．

コード理解を支援するツールのユーザ評価では，デバッグのタスクが広く採用されている [2], [7]．デバッグのタスクでは，実験参加者は故意に実装されているバグを特定し修正することが求められる．しかし，CodeGlass の場合ユーザが過去のコード変更履歴を参照できてしまうため，実験参加者はバグ修正に必要なコード変更を容易に特定できてしまう恐れがある．

そこで我々は，箇条書きの短いドキュメントの作成を実験のタスクとして採用した．このタスクにおいて実験参加者らは，与えられたコード断片に関する Execution，Rationale，History の情報を箇条書きで記す作業を行った．また箇条書きの各項目に対して，実験参加者にはその項目に対する自信度を 0 から 100 で与えるように指示した．高い自信度は，その項目のドキュメントに対し強い自信があることを意味する．

1 つのタスクの時間は 20 分と設定した．また，我々は Chart.js のリポジトリにおける 3 つのソースコードのファイルをタスクとして選んだ．タスクに使用するソースコードのファイルは，core.animations.js，core.element.js，core.ticks.js（タスク A，タスク B，タスク C）とした．なお，実験参加者への負担を抑えるためにタスクの時間を 20 分に制限したため，リポジトリの中で比較的短いソースコードを選択した．著者のうち 2 人がそれら 3 つのソースコードと開発履歴を確認し，ドキュメントに記されるべき情報を解答としてまとめた．その結果，平均で 11.7 件の Execution に関する情報，9.3 件の Rationale に関する情報，6.7 件の History に関する情報からなる解答を作成した．

実験の条件は，GitHub 上のインタフェース（non-CG），開発背景と理由の並べ替え機能がない CodeGlass（CG-），並べ替え機能を持つ CodeGlass（CG）の 3 通りである．タスクの順序を固定した上で，インタフェースの割当順序を counter-balance した．

本ユーザ評価では，12 人の学生（PC1–12，全て男性）を実験参加者として得た．全員，1 年以上のプログラミング経験がある，GitHub を用いた開発経験がある，JavaScript の使用経験がある，Chart.js の開発と使用経験がない，条件を満たしていた．以上を満たす実験参加者らは，開発経験はあるが今回取り組む開発プロジェクトに関する背景知識がないため，CodeGlass の想定ユーザと一致する．

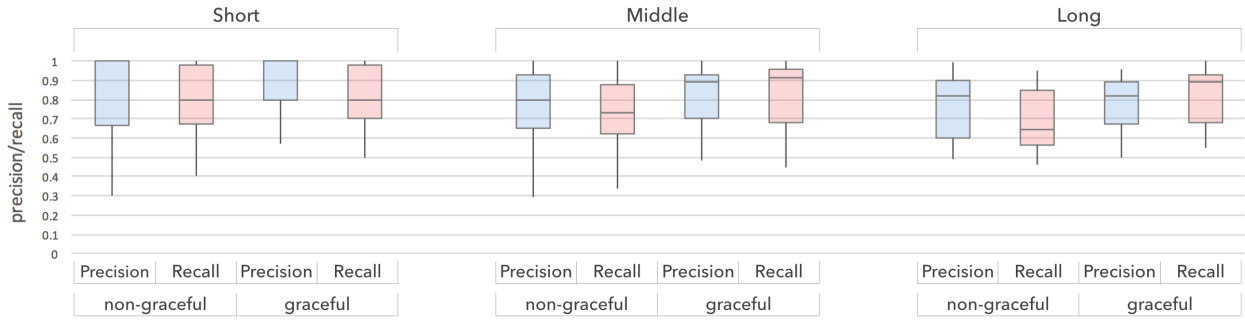


図 4: graceful matching のコード断片の特定精度の評価の結果．*Short* グループの適合率は，graceful matching を用いた場合，用いなかった場合の両方において中央値が 1.0 であった．

表 1: 適合率，再現率，自信度の平均．但し，括弧内は標準偏差である．3 つの条件は，GitHub 上のインタフェース（non-CG），開発背景と理由の並べ替え機能がない CodeGlass（CG-），並べ替え機能を持つ CodeGlass（CG）である．

	non-CG			**CG-**			**CG**		
	適合率	再現率	自信度	適合率	再現率	自信度	適合率	再現率	自信度
Execution	0.93 (0.09)	0.45 (0.18)	61.1 (10.7)	0.87 (0.13)	0.44 (0.23)	66.4 (11.7)	0.90 (0.11)	0.53 (0.24)	77.5 (10.9)
Rationale	0.86 (0.30)	0.17 (0.09)	63.7 (10.4)	0.95 (0.11)	0.31 (0.12)	63.4 (10.5)	0.89 (0.15)	0.34 (0.17)	66.8 (8.07)
History	0.08 (0.29)	0.01 (0.04)	30.7 (16.9)	0.50 (0.52)	0.14 (0.12)	58.9 (19.3)	0.33 (0.49)	0.12 (0.15)	63.7 (24.7)

表 2: 反復測定の一元配置分散分析の結果．

	適合率			再現率			自信度		
	F	p	η^2	F	p	η^2	F	p	η^2
Execution	1.71	.20	0.06	0.47	.63	0.03	4.00	$< .05$	0.22
Rationale	0.73	.49	0.04	6.51	$< .01$	0.27	1.17	.33	0.06
History	2.71	.09	0.14	4.18	$< .05$	0.22	9.68	$< .001$	0.40

7.2 実験結果

実験参加者らが作成したドキュメントの箇条書きのうち，事前に作成した解答に含まれる項目を正解数として数えた．そして，その正解数をもとに適合率と再現率を計算した．但し，ドキュメントに何も記されていなかった場合，適合率，再現率ともに 0 とした．表 1 に，適合率，再現率，自信度の平均と標準偏差を示す．実験結果を分析するために，システム（non-CG，CG-，CG）を要因とした反復測定の一元配置分散分析（ANOVA）により，適合率，再現率，自信度を比較した（表 2）．その結果，Execution の自信度，Rationale の再現率，History の再現率と自信度の計 4 つにおいて，3 条件間に有意性が認められた．ボンフェローニ法による多重比較を行った結果，non-CG と CG-および non-CG と CG の間に有意差が認められたのは，Rationale の再現率（$p < .05$）と History の自信度（$p < .01$）であった．また，Execution の自信度において non-CG と CG の間（$p < .01$），History の再現率において non-CG と CG-の間（$p < .05$）に有意差が認められた．

また実験後に行ったインタビューでは，CodeGlass がどのような開発場面で有用になりそうかを実験参加者に質問した．その結果，想定される利用場面として，「知らないコードの理解（7 人）」，「バグ修正方法の理解（3 人）」，「オープンソースのコードを利用した自習（2 人）」が挙げられた．特に複数人での開発を行う場面やオープンソースのプロジェクトにおいて，与えられたコードを理解するのに適しているという声が上げられた．

> 自分が使いたいと思うときっていうのは，やっぱり複数人でやってるんだったら，自分が担当してない部分のコードを理解したいなー，って思うときにこれを．[PC5]

> 全部どういうコードでこの人が書いたかっていうのが，今まではソースとかコメントしか読めなかったけども，プロジェクト自体がどうやって運んでいくかっていうのもちょっと引いた目で見るみたいなことがいろいろな教材も無限にあるし，っていうのでいいんじゃないかなと． [PC7]

8. プログラマによる定性的評価

さらに我々は，専門的なプログラマにとっての CodeGlass の有用性を調査するために，4 社の IT 企業から 8 人のプログラマ（PD1–PD8, 全て男性）を募り，インタビューによる CodeGlass の定性的評価を行った．

この定性的実験の参加者からも CodeGlass が有用になりうる場面に関して複数の意見が得られた．具体的には，過去のプルリクエストの参照が容易になる（8 人），開発チームの新メンバーの教育に有用である（5 人），オープンソースのコード理解に有用である（4 人），が挙げられたそれらに加えて，実験参加者らはプログラマの専門的な用途における CodeGlass の利点を述べた．

隠れた開発背景の理解

実際のソフトウェア開発現場では，厳格な期日や実装力の不足などの様々な理由から，常に最善の実装方法が選択されるとは限らない．しかし，完成されたソースコードではそういった開発背景は隠れてしまう．実験参加者 2 名（PD3 と PD7）は，CodeGlass はソースコードの裏にある開発背景を理解することにおいて有用であると述べた．

> 新規で入ってコード見ると，だめなコードがたくさんあるんですよね．でも実はそれは特定の技術を使ってはいけないみたいな制約のもとの苦渋の決断だったことってかなりあるんですよね．そういうのはコードをみても絶対に分からないし，かといって大規模だと履歴を追うのは相当つらい． [PD7]

ソフトウェア開発プロジェクトでは通常仕様書が作成されるが，開発が進んでも仕様書が更新されない状況が多く発生する．CodeGlass は，古い情報からなる仕様書とソースコードの乖離を埋められる可能性が指摘された．

> 仕様書をもとにコードを作って，仕様書をもとにテストするんですが，でもやっぱり経緯はわかんないし，仕様書とコードの間のギャップって凄まじいんですよね．仕様書もコードも結果であって，開発理由とかは本当にわからない． [PD7]

プルリクエストによるドキュメント作成

ソフトウェア開発におけるドキュメントの重要性は広く認知されているが，実際の開発現場では，開発者はドキュメント作成に時間を割くことができていない [6]．実験参加者 2 名（PD5 と PD8）は，CodeGlass が将来参照する時のために，プルリクエストの説明文を詳細に書く習慣を CodeGlass が促す可能性があると述べた．

> やっぱりなぜそのコードになったのかが重要で，プルリクは開発フローに既にあるし，それをもう少しちゃんと書いたらドキュメントになるってのはすごく新しいし実用的ですね． [PD8]

コードレビューの支援

ソフトウェアの品質を担保する上でコードレビューは重要な仕事である．しかし，コードレビューを適切に行うためには，ソースコードとその開発背景に関する十分な理解が必要となる．3 人の実験参加者ら（PD1，PD3，PD6）は，CodeGlass では過去のプルリクエストを簡単に参照できるため，コードレビューにおいても有用であると指摘した．

> レビューしていてコードの意図がわかんない時とか，かなり複雑なコードだった時に，プルリクに詳しく説明があるかなあって感じだったので，今言ったような時に使うかなあと． [PD1]

9. 考察

9.1 ユーザ評価の結果に関する議論

アルゴリズムの評価に関する考察は 6 章にて議論したため，ここではユーザ評価の結果について考察を行う．学生による CodeGlass の定量的評価により，Rationale と History に関する情報収集の再現率と，Execution と History に関する情報収集の自信度に対して，CodeGlass の有意な効果が認められた．一方で，全ての情報の分類における適合率に対しては，CodeGlass の有意な効果は見つけられなかった．この原因の一つとして，実験参加者らは確度の高い情報のみをドキュメントに記した可能性がある．

Execution の再現率，適合率に関しては有意な差が確認されなかった．これは使用したコードが比較的短いものであり，プルリクエストに頼らずとも実装内容をある程度推察できたためであると考えられる．一方，Rationale と History の再現率は有意な差があり，CodeGlass の一定の効果を確認するものである．

また，実験結果における再現率が総じて低いことが確認された．この原因の一つとして，タスクに 20 分という時間制限を設けたことが挙げられる．時間制限により実験参加者らは，理解するのが簡単な箇所からドキュメント作成を行ったと考えられる．

8 人のプログラマを対象としたインタビューでは，CodeGlass 特有の有用性を明らかにすることができた．特に IT 企業といった専門的な組織における開発では，コードレビューを円滑に行うために GitHub のプルリクエストが日常的に利用されている．2 人の実験参加者（PD5 と PD8）は，CodeGlass を開発組織に導入することで，プルリクエストの説明文をより詳細に書く習慣を作ることができるかもしれないと述べた．この利点は専門的な組織における CodeGlass の新たな有用性を示唆しており，CodeGlass の長期的システム評価を行うことで，コード断片の理解の支援のみならず，ソフトウェア開発プロセスの改善における有効性が明らかになる可能性がある．

9.2 本研究の限界

ユーザ評価により CodeGlass の有用性を明らかにすることができた一方で，本研究にはさらなる改良を要する点が存在する．ソフトウェア開発においてプルリクエストが使用されていない，あるいは説明文が十分に書かれていない場合，CodeGlass を有効活用することはできない．先行研究ではコード変更の説明文を記述することを支援するシステムが実装されており [13]，CodeGlass と併用することで，プルリクエストの使用や説明文の詳細な記述を促すことができる可能性がある．

CodeGlass は過去のプルリクエストを用いてコード断片に関連する情報を提供することができるが，ソースコード

全体の俯瞰的な理解の支援に関しては評価を行っていない．リポジトリのファイルやクラス構造の理解支援は本研究の目的とは外れるが，Chronicler [21] のようなソースコード全体の理解を支援するシステムと併用することで，コード理解をより包括的に支援できる可能性がある．

CodeGlass は選択されたコード断片に関連するプルリクエストのみを提示するが，大規模なリポジトリにおいて CodeGlass を使用した場合，多くのプルリクエストが表示されてしまう恐れがある．コード理解において有用でないプルリクエスト（“Fix typos” や “Clean up indent” など）のフィルタリング機能などが今後の課題である．

CodeGlass はユーザが選択したコード断片に関連する情報のみを提供しており，コード断片を参照する関数といった他のコード断片に関する情報提示は行っていない．例えば構文解析を用いることで選択されたコード断片と強く関連する他のコード断片を特定できると考えられるが，CodeGlass の特徴であるプログラミング言語の非依存性が失われてしまう．選択されたコード断片に関する情報を完全に抽出するためには，さらなる検証が必要である．

10. 結論

本研究では，コード断片に関連する情報を提供する CodeGlass というシステムを開発した．CodeGlass はユーザが選択したコード断片に関連する過去のプルリクエストを抽出し，ユーザに提供する．さらに，プルリクエストの説明文を解析し，実装内容や開発背景に関する文章をインターフェース上で強調して表示することができる．情報系の学生およびプログラマを対象としたユーザ評価を行った結果，プルリクエストの説明文に含まれる情報が，コード断片を理解する上で有用であることが明らかとなった．

謝辞 本研究の一部は，科学研究費助成事業（若手研究 18K18088）によって支援された．また，本研究の実験に参加してくださった全ての実験参加者に感謝する．

参考文献

[1] Asaduzzaman, M., Roy, C. K., Schneider, K. A. and Penta, M. D.: LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines, *2013 IEEE International Conference on Software Maintenance*, pp. 230–239 (online), DOI: 10.1109/ICSM.2013.34 (2013).

[2] Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, Jr., J. J.: Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance, *Proc. CHI '10*, pp. 2503–2512 (2010).

[3] Buckner, J., Buchta, J., Petrenko, M. and Rajlich, V.: JRipples: a tool for program comprehension during incremental change, *Proc. IWPC '05*, pp. 149–152 (2005).

[4] Buse, R. P. and Weimer, W. R.: Automatically Documenting Program Changes, *Proc. ASE '10*, pp. 33–42 (2010).

[5] D'Ambros, M., Lanza, M. and Robbes, R.: Commit 2.0, *Proc. Web2SE '10*, pp. 14–19 (2010).

[6] de Souza, S. C. B., Anquetil, N. and de Oliveira, K. M.: A Study of the Documentation Essential to Software Maintenance, *Proc. SIGDOC '05*, New York, NY, USA, pp. 68–75 (2005).

[7] Dekel, U. and Herbsleb, J. D.: Improving API Documentation Usability with Knowledge Pushing, *Proc. ICSE '09*, pp. 320–330 (2009).

[8] Kagdi, H., Hammad, M. and Maletic, J. I.: Who can help me with this source code change?, *Proc. ICSM '08*, pp. 157–166 (2008).

[9] Ko, A. J., DeLine, R. and Venolia, G.: Information Needs in Collocated Software Development Teams, *Proc. ICSE '07*, pp. 344–353 (2007).

[10] Ko, A. J., Myers, B. A. and Aung, H. H.: Six Learning Barriers in End-User Programming Systems, *Proc. VLHCC '04*, pp. 199–206 (2004).

[11] Le, Q. and Mikolov, T.: Distributed Representations of Sentences and Documents, *Proceedings of the 31st International Conference on Machine Learning* (Xing, E. P. and Jebara, T., eds.), Proceedings of Machine Learning Research, Vol. 32, No. 2, Bejing, China, PMLR, pp. 1188–1196 (online), available from ⟨http://proceedings.mlr.press/v32/le14.html⟩ (2014).

[12] Lilleberg, J., Zhu, Y. and Zhang, Y.: Support vector machines and Word2vec for text classification with semantic features, *2015 IEEE 14th International Conference on Cognitive Informatics Cognitive Computing (ICCI*CC)*, pp. 136–140 (online), DOI: 10.1109/ICCI-CC.2015.7259377 (2015).

[13] Linares-Vásquez, M., Cortés-Coy, L. F., Aponte, J. and Poshyvanyk, D.: ChangeScribe: A Tool for Automatically Generating Commit Messages, *Proc. ICSE '15, Vol. 2*, pp. 709–712 (2015).

[14] McBurney, P. W. and McMillan, C.: Automatic Documentation Generation via Source Code Summarization of Method Context, *Proc. ICPC '14*, pp. 279–290 (2014).

[15] Sankoff, D. and Kruskal, J. B.: Time warps, string edits, and macromolecules, *The Theory and Practice of Sequence Comparison, Reading* (1983).

[16] Sridhara, G., Pollock, L. and Vijay-Shanker, K.: Automatically Detecting and Describing High Level Actions Within Methods, *Proc. ICSE '11*, pp. 101–110 (2011).

[17] Stylos, J., Faulring, A., Yang, Z. and Myers, B. A.: Improving API documentation using API usage information, *Proc. VL/HCC '09*, pp. 119–126 (2009).

[18] Subramanian, S., Inozemtseva, L. and Holmes, R.: Live API Documentation, *Proc. ICSE '14*, pp. 643–652 (2014).

[19] Tao, Y., Dang, Y., Xie, T., Zhang, D. and Kim, S.: How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry, *Proc. FSE '12*, pp. 51:1–51:11 (2012).

[20] Treude, C. and Robillard, M. P.: Augmenting API Documentation with Insights from Stack Overflow, *Proc. ICSE '16*, pp. 392–403 (2016).

[21] Wittenhagen, M., Cherek, C. and Borchers, J.: Chronicler: Interactive Exploration of Source Code History, *Proc. CHI '16*, pp. 3522–3532 (2016).

[22] Zhang, T., Song, M., Pinedo, J. and Kim, M.: Interactive Code Review for Systematic Changes, *Proc. ICSE '15, Vol. 1*, pp. 111–122 (2015).