JCompaths: 実行経路の比較と可視化を行うコードレビュー向けツール

神田 哲也　橋本 悠樹　嶋利 一真　肥後 芳樹

ソフトウェア開発においてプログラム理解は重要なタスクの 1 つである．プログラム理解が重要な場面としてコードレビューが挙げられる．特に変更に対するレビューでは，ソースコードの差分以外にも前後の挙動を理解してプログラムを読み進める必要がある．しかし，プログラムを実行することで得られる動的な情報に着目したレビューの支援技術は少ない．そこで本研究では．didiffff という既存のツールを拡張し，ソースコードの変更前後におけるメソッドの実行の変化を可視化するツール JCompaths を作成した．このツールは，プログラム実行時に変数トークンがとった値の系列 (トレース) に着目する．メソッドの実行を 1 回ずつ順に比較し，その実行経路とともにトレースの差分を表示することで，ソースコードの差分が実行に与えた影響を可視化する．また，JCompaths の有用性を評価するために被験者実験を行った．結果として，タスクの所要時間と点数，およびユーザビリティの総合スコアのいずれにおいても，JCompaths と他のツールとの間で統計的に有意な差は見られなかった．しかしながら，自由記述からは，JCompaths 独自の機能が，実験で用いたような繰り返し実行を含むコードに対してはユーザにとって役立つことが確認できた．

1　まえがき

ソフトウェア開発におけるプログラム理解のタスクの占める比重は大きい．Minelli らの調査[12] では，開発者は約 70% の時間をプログラム理解に費やしていることが明らかになっている．

ソフトウェア開発において既存のプログラムの内容を理解する場面のひとつとして，コードレビューが挙げられる．コードレビューは，ソースコードの可読性，保守性の向上や教育的な効果をもたらす．そのため，オープンソースのシステムから商業的なシステムの開発に至るまで広く実施されている [15]．コードレビューにおいて，特にバグ修正のような変更を扱う場合，変更された箇所がどう書かれているかだけでなく，差分の前後も含めたソフトウェアの挙動を理解する必要がある．このような状況では，ソースコードの差分のみから得られる情報は限られている．そこで，テストケースを実行するなどして変数の値や実行の経路がどのように変化したかを実例で確認することが，プログラムの理解の助けになると考えられる．しかし，そのような動的な情報の差分を可視化することに着目した研究は少ない．

プログラムの変更前後における実行の変化を可視化するために，我々は以前 didiffff [8] を開発しツールデモを行った．これは，Java プログラム実行時に，ある変数がとった値の系列に着目し，ソースコードの変更前後における動的・静的双方の変化を同時に示すツールである．このツールでは，繰り返し呼び出されるメソッドやメソッド内部の繰り返しにおける差分の表示に改善の余地があった．そこで本論文では，didiffff を拡張し，ソースコードの変更前後におけるメソッドの実行の変化を特に繰り返しに着目してより詳しく可視化できるツール JCompaths を提案する．

JCompaths: A code review tool for comparing and visualizing execution paths.

Tetsuya Kanda, ノートルダム清心女子大学, Notre Dame Seishin University.

Yuki Hashimoto, Yoshiki Higo, 大阪大学, Osaka University.

Kazumasa Shimari, 奈良先端科学技術大学院大学, Nara Institute of Science and Technology.

コンピュータソフトウェア, Vol.42, No.3 (2025), pp.23–39.
[ソフトウェア論文] 2024 年 8 月 19 日受付.
大会同時投稿論文

本論文は，上述のツールデモ論文[8]を拡張したものである．本論文では，ツールデモで示したコンセプトをもとに改良を加え，また被験者実験を行いツールを利用することによるプログラム理解のタスクでの効果を評価している．評価実験では，JCompaths を使った場合のタスクの結果を，didiffff を使った場合の結果，および可視化ツールを使わなかった場合の結果と比較し評価を行う．タスク後にはアンケートをとり，SUS と自由記述により，ユーザビリティの側面からも評価を行う．

以降，2 章では研究背景について，3 章では作成したツールの仕様について述べる．4 章では作成したツールの評価方法について，5 章ではその評価結果について述べる．6 章ではツールの問題点について整理し，最後に 7 章で本研究のまとめを行う．

なお，ツールは GitHub 上で公開している †1.

2 背景

この章では，本研究の背景として，コードレビューや，プログラムの実行の可視化，ソースコードの差分とプログラム実行の変化の可視化に関する先行研究について述べる．

2.1 コードレビュー

コードレビューとは，ソフトウェア開発において，ソースコードを変更した開発者以外が，そのソースコードをチェックする作業のことである．この作業はソースコードの可読性や保守性を向上させるだけでなく，教育的な側面をもち，オープンソースのシステムから商業的なシステムの開発に至るまで広く普及している[15].

コードレビューで指摘すべき項目は多岐に渡る．スペルミスやコーディング規約違反などは，ソースコードのごく一部を見るだけで容易に指摘できるが，プログラムの動作を十分に理解しないと指摘できないような項目も存在する．

コードレビューでは，ソースコードの差分のみから得られる情報は限られている．そこで，コミットメッセージやプルリクエストの説明文など，ソースコードを変更した開発者が静的な情報を補足することが一般的である．このような情報を自動的に生成する研究も行われている[6][10].

コードレビューにおいてプログラム理解は重要である[3]．既存のコードに対して十分に精通していないことは，コードレビューにおいて混乱を引き起こす一つの大きな要因となっている．この混乱はレビューの遅延や品質の低下を引き起こす．そのため，開発者はコードの実行や，プログラムの詳細な理解によってこの混乱に対処している．また，Tao らはコードレビューなどの変更の理解において，変更のリスク，例えば変更が他のコードに影響を及ぼすかどうかを把握することが重要であるが，これらの情報を得ることが一般的に難しいと述べている[19].

2.2 実行の可視化に関する先行研究

デバッグの支援を目的にプログラムを実行中の状態の可視化を行うツールとして，デバッガがあり，C や C++ では GDB[5]，Java では jdb[14] などが代表的である．これらは，ソースコード中の興味のある位置にあらかじめブレークポイントを設定し，プログラムを 1 ステップずつ実行する中で，ステップごとに変数の値やスタックフレームなどを可視化する．また，プログラミング初学者の支援を目的に可視化を行うツールとしては，Jeliot 3[13] がある．このツールは，Java プログラムの 1 ステップごとの実行を，自動で再生されるアニメーションにより可視化する．

これらのツールとは別の概念として Omniscient Debugging[9] がある．この手法は，プログラム実行時のメモリの状態を網羅的に記録しておくことで，任意の時点におけるプログラムの状態をあとから再現する．Omniscient Debugging によって得られた情報を可視化するツールとして，NOD4J[17] のビューアコンポーネントがある．このビューアはブラウザ上で動作し，ソースコード上の各変数トークンに対し記録された値を表示することができる．

一方で，複数回の実行を可視化するための研究もおこなわれている．Deknop らは，大規模なリファクタリングプロセスにおける振る舞いの変化を実行ログ

†1 https://github.com/tetsuyakanda/jcompaths

から抽出し，グラフをハイライト表示することで可視化している [2]．また，あるメソッドの複数回の実行における実行経路を可視化し比較するツールとして，REMViewer [11] がある．ここでの実行経路とは，プログラム実行時にソースコードのある文が実行されたという記録を要素とした系列である．メソッドの複数回の実行をその実行経路の違いによって分類し，それぞれの分類から 1 つの実行を可視化することで，実行経路と局所変数の状態を比較できる．我々の提案する JCompaths はこの REMViewer による実行経路の可視化機構を取り入れている．

2.3 ソースコードの差分と実行の変化を対応付けた可視化

我々は，ソースコードの変更前後におけるプログラムの実行を容易に比較するために，プログラムの実行の変化をソースコード上にマップして表示するツール didiffff を提案した [8]．プログラムの実行の変化は，プログラム実行時に変数トークンがとった値の系列 (トレース) について，ソースコードの変更前後でその内容を比較することで得られる．ここでの変数トークンとは，変数名を表すソースコード中の文字列のことを指し，同じ変数でもソースコード中の出現場所が異なる場合，それらは異なる変数トークンとみなす．トレースの差分の有無はソースコード中の変数トークンに色をつけてマッピングし，ソースコードの差分と合わせて表示することで，ソースコードの変更が各変数トークンに与える影響を可視化する．didiffff は，ソースコードの差分と実行の差分を同一のビューで可視化することで，開発者の理解を促進することをコンセプトとしている．

また，ソースコードの変更前後におけるプログラムの挙動の変化を検知するツールとして，Collector-Sahab [4] が提案されている．このツールは，ソースコードの変更前後におけるプログラムの実行を比較し，どちらか一方にしかない変数および返り値の状態を検出する．ここでの状態とは，＜行 , 変数, 値＞の三つ組みである．また，Collector-Sahab は，オブジェクト内部の値について深さを指定してその範囲内でのプリミティブ型の値を取得している．このようなデータの収集と計算により，差分の検出精度が didiffff よりも優れていることを示している．

Collector-Sahab の可視化は，上記の手順で得た変更前後どちらか一方にしかない状態を，ソースコードの差分表示中に注釈の形で表示する．特異な状態に着目してソースコードの差分に対する情報を補強するという観点で設計されており，変更による影響箇所を効率的に検知することに重点を置いているといえる．一方我々の提案するツールは，実行トレース全体の可視化を行い，差分以外の部分についてもすべての変数トークンのステータスや値を可視化対象としている．

3 ツールの設計

JCompaths は，Java プログラムのソースコードの変更前後におけるプログラムの実行を容易に比較するために，プログラムの実行の変化をソースコード上にマップして表示するツールである．可視化において，Collector-Sahab のように差分が発生した点にのみ注目するのではなく，全体の実行の情報を表示することで，プログラム全体の動作を含めて差分の理解を行えるようにすることを意図している．JCompaths は，ソースコード変更前後のソースコードおよび実行時情報を用いて，実行の変化の可視化を行う．具体的には，プログラム実行時に変数トークンがとった値の系列 (トレース) に着目し，ソースコードの変更前後でその内容を比較する．ここでの変数トークンとは，変数名を表すソースコード中の文字列のことを指し，同じ変数でもソースコード中の出現場所が異なる場合，それらは異なる変数トークンとみなす．トレースの差分はソースコード中の変数トークンにマッピングし，ソースコードの差分と合わせてハイライトで表示する．

具体的なトレースの値は，ソースコード上の変数をクリックすると横に用意された別のビューに表示される．これにより，開発者はまず一般に用いられているソースコードの差分表示を閲覧しながらソースコードの変更が各変数トークンに対する影響を大まかに理解し，その後ツール独自のビューを用いて具体的な変化の内容を探索できるように設計している．ソースコード上の変数をクリックすることによりト

表 1　先行研究からの改良点の概要

事例	didiffff での課題	JCompaths での改善
メソッドが複数回実行された時の可視化	何回目の実行かが識別できない	• トレースの分析を拡張し，1 回の実行ごとに差分のステータスを割り当て • ビューアにメソッド実行単位での切り替えを実装
メソッドの一部が繰り返し実行された時の可視化	変更前後での実行回数の対応関係が不明	• ソースコード上に矩形を表示して実行経路の比較を支援 • 矩形選択により変数ビューで実行回数の対応関係を可視化 • 複数の変数ビューを同時表示

レースの値を表示する手法は NOD4J [17] のビューアコンポーネントを参考にしており，本ツールではここに差分の状態を示す色分けを加えている．

JCompaths は既報の didiffff のコンセプトをもとに改良を加えたものである．改良点の概要を表 1 にまとめる．didiffff がファイル単位での差分計算結果を可視化していたのに対し，JCompaths はメソッド単位での差分計算をもとに可視化を行っている．これにより，複数回呼び出されるメソッドを呼び出し単位で比較することが可能になっている．さらに，REMViewer [11] の実行経路の可視化手法を，差分の表現に対応させた上で取り入れることにより，あるメソッド実行内での繰り返しについても整理した情報を提供することができる．この章では，ツール全体の設計としては didiffff と重複する部分も記述し，JCompaths における拡張部分を **JCompaths における拡張**として別段落で表記する．また，この拡張により見た目だけでなく提供する情報の品質が向上する例を 3.4 節で説明する．

ツールは 3 つのコンポーネントから構成される．

- 変更前後それぞれの実行時情報の取得および解析
- 変更前後間の差分計算
- Web ビューアでの可視化

以下，それぞれについて詳しく説明する．

3.1　変更前後それぞれの実行時情報の取得と解析

Java プログラムの実行時に変数に関する情報を取得し，解析を行う．実行時情報の取得には，NOD4J [17] と同様に SELogger を利用する．取得した実行時情報を，NOD4J の post processor を用いて，ある変数トークンについて使用 (代入・参照) された際の値を記録する．プリミティブ型と String 型の変数トークンについてはその値を，そうでなければ Object ID を記録する．この結果得られる値の系列が変数トークンに対するトレースである．

JCompaths における拡張

メソッド単位での比較を可能にするため，NOD4J の post processor にメソッドの区切りを認識する機能を追加する．この結果得られるトレースは，ある変数トークンについて使用された際の値およびメソッド区切りからなる系列となる．

3.2　変更前後間の差分計算

3.2.1　ソースコードの比較

プロジェクト中の各ソースファイルについて，変更前後のソースコードを Unix diff に基づいて比較し，行ごとに差分のステータスを割り当てる．変更により削除された行は **“delete”**，変更により追加された行は **“add”**，変更前後で共通する行は **“unchanged”** とする．このうち，“unchanged” の行に含まれる各トークンについて，ソースコード変更前のファイルにおけるトークンと変更後のファイルにおけるトークンが対応しているものとして扱う．

3.2.2　トレースの比較

ソースコード中の各変数トークンに対して，ソースコード変更前のトレースと変更後のトレースを比較し，差分のステータスを割り当てる．このステータスは，実行前後の挙動の比較を行うにあたり着目すべき

点を指し示すヒントとなる．ステータスは次の手順に従って決定する．

1. ソースコード変更前，変更後ともにトレースの長さが 0 である場合 `no trace` とする．
2. 変数トークンが “delete” の行に含まれる場合 `trace 1 only`，“add” の行に含まれる場合 `trace 2 only` とする．
3. ソースコード変更前と変更後でトレースの長さが異なる場合 (いずれか一方のトレースの長さが 0 の場合も含む)`diff in length` とする．
4. (ソースコード変更前と変更後でトレースの長さが同じであり) 変数トークンがプリミティブ型の変数でも String 型の変数でもない場合，`same length object` とする．
5. ソースコード変更前後のトレース内容を比較し，一致する場合 `no diff`，そうでない場合 `diff in contents` とする．

トレース間の比較においては，同一かどうかの判定のみを行い UNIX diff のような差分については計算しない．ソースコードの変更により，実行時の値の系列は長さ，内容ともに大きく変化し，また偶然部分的に一致する可能性もある．そのため，2 つの系列の違いを値のみをヒントに正確に特定することは難しい．利用者に混乱を与えないため，差分計算は行わず，対応関係についてはビューアでの表示をもとに利用者が判断する形をとることとした．

JCompaths における拡張

上記の比較は didiffff ではある変数トークンのトレース全体に対して行っていたが，メソッドが複数回実行された場合にその何回目の実行における変化であるかを識別できない状態であった．JCompaths では，メソッドの 1 回の実行ごとに比較を行いそれぞれに差分のステータスを割り当てることでこの問題に対処した．メソッド中の変数トークンが，そのメソッドの i 回目の実行でとった値の系列を，その変数トークンの i 回目のトレースとする．このとき，ソースコード変更前の i 回目のトレースと，変更後の i 回目のトレースを比較し，上記の手順に従って定めるステータスを，その変数トークンの i 回目のステータスと呼ぶ．

図 1 JCompaths のビューアの概観

3.3 Web ビューアでの可視化

ビューアは Node.js の環境下で実行され，Web ブラウザで動作する．ビューアの概観を図 1 に示す．ビューアは大きく分けて 3 つの部分に分かれており，左から順に，ファイルツリービュー，ソースコードビュー，トレースビューとなっている．

3.3.1 ファイルツリービュー

ファイルツリービューでは，ソースコードビューに表示するファイルをファイルツリーから選択する．ファイル名の先頭には，そのファイルにおける差分の有無を表す 2 つの円が表示される．1 つ目の円は，ソースコードの差分の有無を表し，ファイルに “delete” または “add” の行が含まれる場合は緑色になる．2 つ目の円は，トレースの差分の有無を表し，ファイルに “diff in length” または “diff in contents” の変数トークンが含まれる場合は紫色になる．また，ディレクトリについても，そのディレクトリ中にこれらの円が表示されるべきファイルが含まれている場合に同様の円を表示し，ディレクトリ内部の差分の有無を表す．これにより，注目すべきファイルへ容易にたどり着けることをねらっている．

3.3.2 ソースコードビュー

ソースコードビューでは，ファイルビューで選択したファイルにおけるソースコードとその差分が表示される．ソースコードは，“delete” の行の背景が赤色，“add” の行の背景が緑色となる．各変数トークンは，そのステータスに応じて表 2 に示す色でハイライトされる．このハイライトにより，差分があって注目すべき変数トークンを容易に発見できるようにしている．変数トークンをクリックすることで，トレースビューに表示させたい変数を選択できる．

表 2　変数トークンのステータスと色の対応

ステータス	ハイライト色
“no trace”	(ハイライトなし)
“trace 1 only”	灰
“trace 2 only”	
“no diff”	
“same length object”	薄い青
“diff in length”	紫
“diff in contents”	

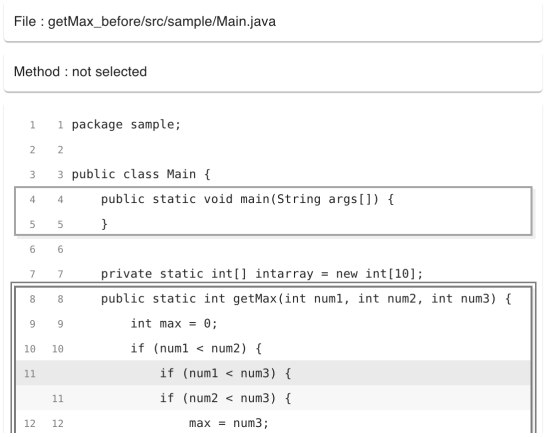

図 2　ファイル選択後のソースコードビュー

JCompaths における拡張

• メソッド全体の繰り返しについての拡張

差分計算における繰り返し実行への拡張を可視化に反映させるため，可視化対象のメソッドを選択したうえで特定の回の実行をビューに表示するようにした．

ファイル選択直後のソースコードビューを図 2 に示す．このビューでは，選択中のファイルにおけるソースコードの差分が表示され，選択可能なメソッドが 1 つずつ枠で囲まれている．枠の色は，そのメソッドにおけるトレースの差分の有無を表す．メソッドのある回の実行において，ステータスが “diff in length” または “diff in contents” の変数トークンを含む実行が存在する場合，その枠は紫色 (図では説明のため二重枠線) になる．これにより，注目すべきメソッドを容易に発見できる．これらの枠を選択することで，ファイル中のどのメソッドの実行を比較するかを指定する．この段階では，メソッドの何回目の実行を表示するかを決めていないため変数トークンのハイライトは行わない．

図 3　メソッド選択後のソースコードビュー

メソッド選択後のソースコードビューを図 3 に示す．まず，右上にそのメソッドの何回目の実行を比較するかを選択するための表示を追加している．表示は左右の切り替えボタンと「表示中の実行回/総実行回数」を示す数字からなり，表示中の実行回の文字色が，その実行回におけるトレースの差分の有無を表す．メソッドが，i 回目のステータスが “diff in length” または “diff in contents” の変数トークンを含む場合，実行回 i は紫色になる．これにより，注目すべき実行を容易に発見できる．

ソースコード上での変数トークンのハイライトは，選択されたメソッドの実行回における差分の状態を表している．“diff in length” と “diff in contents” について，didiffff では色を変えていたが，JCompaths ではトレースの長さを別途以下に述べる矩形表示により表現しているため同色で表示している．

• メソッドの一部の繰り返しについての拡張

メソッドの一部の繰り返しについての拡張として，REMViewer [11] の可視化技術を応用し，実行経路の差分を表示する．これにより，トレースの各値がループの何回目における値なのかを確認できるようにする．

まず，REMViewer による実行経路の可視化について説明する．REMViewer では実行経路を可視化するため，ソースコードの背景に矩形を表示する．表示例を図 4 に示す．ここでは，for ループを含むソース

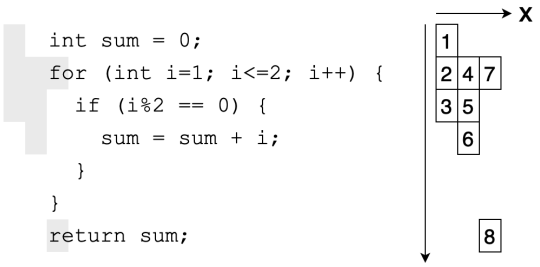

図 4 REMViewer [11] による実行経路の表示例

図 5 矩形とトレースの表示例

コードの実行を記録し，その結果各行に対して図左側のように矩形が表示されている．矩形は各行の実行に応じて垂直方向下向きに描画され，繰り返しによりプログラムの実行がソースコードの上方向に向かうときに水平方向右側の列に移動する．図 4 左側の矩形は，図右側の数字 1 から 8 の順に実行されることを示しており，3 から 4，6 から 7 において繰り返しにより X 方向の座標が変化している．

JCompaths ではメソッド選択後のソースコードビューにおいて，その実行経路を REMViewer の矩形を 3 色に拡張して可視化する．赤の矩形 (説明のため，本稿では以降矩形上部に横線を付す) はソースコード変更前にのみ存在する実行経路，緑の矩形 (同左部に縦線) はソースコード変更後にのみ存在する実行経路，青の矩形 (同線なし) はソースコード変更前後で共通する実行経路をそれぞれ表す．なお，矩形表示は，SELogger で記録される全命令に対応する行番号にもとづいているため，変数トークンが存在しない行についても表示される．

矩形の表示例を図 5 の左側に示す．このソースコードは，8 行目の for ループ内にある if 文の条件式が変更されたことで，9 行目の実行回数が変化している．9 行目がソースコード変更前はループの 2，4，6 回目で，変更後はループの 3，6 回目で実行されたことが矩形から読み取れる．ループの 6 回目を示す矩形には，次のトレースビューの項での説明のため丸印を付している．

3.3.3 トレースビュー

最後に，トレースビューでは，選択中の変数トークンのステータスとトレースが表示される．トレースを表示する列は左右 2 つあり，左側に並んでいる値がソースコード変更前のトレース，右側に並んでいる値が変更後のトレースである．変数トークンがプリミティブ型，または String 型の変数の場合，その具体的な値が表示されるが，それ以外の場合はクラス名とオブジェクト ID が表示される．

JCompaths における拡張

トレースの長さが異なる場合など，どの値とどの値を比べればよいかが不明瞭なことがある．そのような状況での理解支援のため，ソースコードビューの矩形とトレースビューの値の対応関係を可視化する機能を実装した．具体的には，ソースコードビューの矩形をクリックにより選択可能とし，選択中の矩形に対応するトレース中の値を，トレースビューにおいて赤字で表示するようにした．

矩形の選択とトレースの例を図 5 を用いて示す．図では，トレースビューに 9 行目左辺の変数トークン **sum** を表示している．さらに，この図では 9 行目のループ 6 回目の矩形が選択 (説明のため，本稿では丸印を付す) されており，ソースコード変更前後それぞれにおいて，この矩形に対応するトレース中の値が赤字 (同枠囲み) になっている．それぞれの長さと矩形の数を比べることでも，左右のどの値を見比べればよいかを数えることは可能であるが，本拡張により対応関係を素早く把握することができる．

また，didiffff の場合，トレースビューには 1 つの変数トークンに対するトレースしか表示されない．JCompaths では複数の変数トークンを選択し，それらのトレースを同時に表示することができるように設計を見直した．上記の矩形は複数同時選択ができるため，例えば for ループの特定回の矩形を選択することで，複数トークンにまたがって値の対応関係を把

握することができる．

また，JCompaths は配列におけるトレースの表示方法にも拡張を加えている．プリミティブ型の配列について，配列中の特定の要素を参照するとき (`array[index]` の形) に，その要素の参照を 1 つの変数トークンとみなして具体的な値をトレースとして表示することができる．また，配列の長さを参照するとき (`array.length` の形) にも，その具体的な値をトレースとして表示できるようにした．トレースビュー上では，図 5 右上のトグルボタンにより，配列のオブジェクト ID を表示するモード (Array ID) と，具体的な値を表示するモード (Array Value) を切り替える．

3.4 JCompaths 拡張により可視化の品質が向上する例

メソッドの繰り返しおよびメソッド内での繰り返しの差分に着目することにより可視化の品質が向上する例を，Defects4J [7] に含まれる Math 82 を用いて説明する．Defects4J は，OSS に生じた実際のバグからなるデータセットであり，デバッグ前後のソースコードと，それを確認するためのテストケースが用意されている．Math 82 のバグは，`getPivotRow()` メソッドの返り値が不正だったことに起因しており，同メソッドに含まれる条件式を書き換えることで，バグが修正されている．

まず，didiffff での可視化結果は図 6 のようになる．図上側はソースコードビューで，利用者はソースコードの差分を頼りにこのファイルにたどり着き，ソースコードの差分をみて `getPivotRow` メソッドの近辺を表示しているところである．図左下は，返り値 `minRatioPos` のもととなる 86 行目の変数トークン `i` を選択したときのトレースビューである．トレースより，`i` は 4 回目の参照までデバッグ前後で同じ値をとり，5 回目から異なる値をとったことがわかる．一方，ソースコードより 86 行目の実行は 84 行目の条件式で制御されている．そこで，84 行目の変数トークン `minRatio` を選択したときのトレースビューが図右下である．変更によりトレースの長さが変化していることはわかるが，変数トークン `i` との対応が不明なため，これ以上の考察は困難である．また，ソースコードビュー全体を眺めても，差分のある 82 行目以外のすべての変数トークンが「記録されているトレースの長さが違う」を意味する緑色 (図中では薄い灰色) の表示となっており，ソースコードの変更による影響が具体的にみてとれない．

同じ実行について，JCompaths で可視化すると図 7 のようになる．説明のため図 5 と同様に矩形に線や丸印を付している．図上側はソースコードビューで，利用者はソースコードの差分を頼りにこのファイルにたどり着き，同じくソースコードの差分をみて `getPivotRow` メソッドを選択している．このメソッドの 1 回目から 4 回目の実行においては，86 行目の変数トークン `i` は灰色でハイライトされ，そのトレースに差分がなくメソッドの返り値に変化がないことがわかったため，図に示す 5 回目の実行に切り替えている．図左下は，この 5 回目の実行において，86 行目の変数トークン `i` を選択したときのトレースビューである．`i` のトレースより，`i` はソースコード変更前は 2，変更後は 5 をとったことがわかる．さらに，ソースコードビューの 86 行目の矩形により，これらの値はそれぞれ for ループの 2 回目，5 回目でとった値であることがわかる．図右下は，84 行目の条件式内の変数トークン `minRatio` と `ratio` を選択したときのトレースビューである．ソースコードビュー内で for ループの 2 回目の矩形を選択 (図中では丸印) することで，図に示すようにトレースビューの値も色 (図中では枠) が付けられ，ある行における変数トークンの値と別の行における変数トークンの値の関係を確認することができている．また，ソースコードビュー全体を眺めても，この回のメソッド実行において値が異なる部分が強調表示されており，ソースコードの変更による影響が理解しやすい形になっている．

このように，JCompaths における拡張はメソッドの実行を 1 回ずつ順に比較し，矩形により実行経路を表示することで，didiffff に比べてトレースが整理され多くの情報を得ることができる．複数の変数トークン間における値の依存関係も明確になり，メソッド全体の動きを捉えやすくなる．

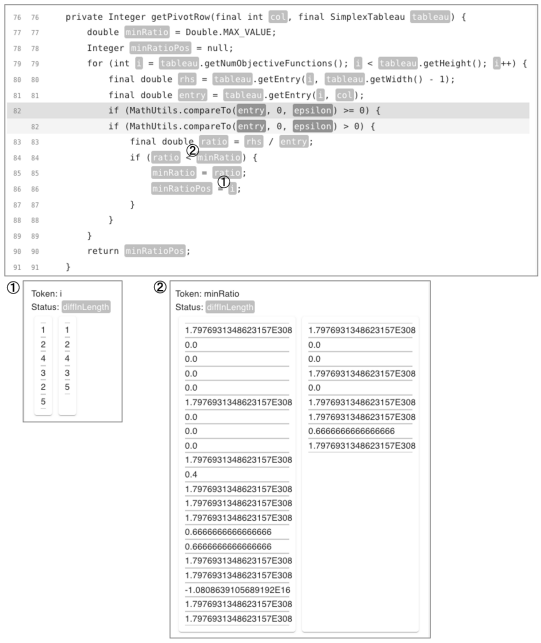

図 6 didiffff で可視化した Math 82 の実行

図 7 JCompaths で可視化した Math 82 の実行

4 ツールの評価方法

JCompaths の有用性を確認するための被験者実験を行った．この実験は以下の 2 点を確認することを目的としている．

1. ツールのコンセプトである，ソースコードの変更前後において，動的な情報を静的な差分にあわせて可視化することはその変化を理解する助けとなるか
2. JCompaths における繰り返しを有するメソッドに対する可視化の拡張は有効に働くか

実験はタスクとその後のアンケートからなる．

4.1 実験の被験者

大阪大学基礎工学部情報科学科の学生 4 名，大阪大学情報科学研究科博士前期課程の学生 6 名の計 10 名を被験者とした．全員が大学で情報科学を学んできた学生であり，GitHub のような差分表示については普段から目にしている．didiffff および JCompaths については，ツールの利用経験はなかった．JCompaths が提案ツールであることおよび didiffff をベースにした拡張であることは，タスクを行う前に配布するドキュメントにおいてあらかじめ明記しており，すべての被験者が同様に理解したうえでタスクに臨んでいる．

4.2 タスクによる評価

被験者には，デバッグによるメソッドの実行の変化について記述するタスクを，JCompaths を用いて行ってもらった．また，同じ実行時の差分を表示するツールとして didiffff を，ソースコードの差分のみを表示するツールとして GitHub を用いて同様のタスクを行ってもらい，タスクの所要時間，および点数を JCompaths を用いたときのものと比較した．

4.2.1 バグ修正データの準備

バグ修正の内容や挙動の変化があまりにも単純な場合，ツールの性能差が結果に表れない可能性がある．一方で，過度に複雑なバグ修正を扱ってしまうとツールの選択に関係なく理解が難しくなる恐れがある．

まず，Defects4J に収録されているバグ修正のうち，バグの原因などが詳細に分析・整理されている Defects4J Dissection [18] に記載されているプロジェ

クトを候補とした．これにより，タスク実施において正確なバグ修正の説明文を得ることができる．また，Shimari らは Defects4J に対する実行トレースの収集において，プロジェクトごとに命令の繰り返し回数のような特徴が異なることを明らかにしている [16]．そこで，Shimari らの実験で繰り返しの長さが 256 以下の場合に実行トレースが完全に記録できていた割合が高い Chart および Lang に焦点を当てた．次に，被験者は 3 つの異なるツールを用いるため，対象のプロジェクトは同じであるほうがよいとも考えた．そこで Chart のバグ修正 26 件に対して，以下の条件でフィルタリングを行い，3 つのバグ修正が残った．

- バグ修正により書き換えられたメソッドが 1 つで，そのメソッドが 20 行以上 (以下，このメソッドを**被変更メソッド**と呼ぶ)
- バグ修正前に失敗したテストケースが 1 つで，そのテストケースを実行したときに，ソースコードに繰り返し実行される部分がある
- バグ修正部分の影響がプリミティブ型へ波及してメソッドの返り値が変わり，テストの成否が変化した

選択した 3 つのバグ ID は，バグ 1：Chart5，バグ 2：Chart7，バグ 3：Chart11 である．

4.2.2 タスクの手順

2.1 節に示した通り，コードレビューにおいて変更とその影響を正確に把握することは重要なステップである．そこで，ある程度コードとバグ修正の内容が提示された状態から，ツールを用いて変更内容とそれによる挙動の変化をどれだけ正確に読み取れるかを計測するタスクを行う．被験者には，タスクを行う前に，各ツールの機能とタスクの内容を記したドキュメントを読んでもらった．各ツールについては，サンプルのソースコードを対象として操作に慣れる時間を確保した．この作業は被験者から終了の申告があるまで実施するものとし，最大で 20 分程度を要した．その後，3 つのタスクを，それぞれ異なるツールを用いて，それぞれ異なるバグ修正を対象に行ってもらった．被験者間の能力差と，ツールの使用順序等による結果の偏りを避けるため，使用するツールと対象のバグ修正の組み合わせ，およびその実施順序は表 3 のように被験者ごとに変更した．表ではツール名を先頭 1 文字で略記している．

表 3　被験者ごとのタスク内容 (使用するツール/対象のバグ修正)

被験者	タスク 1	タスク 2	タスク 3
A	G / バグ 1	D / バグ 2	J / バグ 3
B	G / バグ 3	J / バグ 1	D / バグ 2
C	G / バグ 2	D / バグ 3	J / バグ 1
D	G / バグ 1	J / バグ 3	D / バグ 2
E	D / バグ 1	J / バグ 2	G / バグ 3
F	D / バグ 3	G / バグ 1	J / バグ 2
G	D / バグ 2	J / バグ 3	G / バグ 1
H	J / バグ 1	G / バグ 2	D / バグ 3
I	J / バグ 3	D / バグ 1	G / バグ 2
J	J / バグ 2	G / バグ 3	D / バグ 1

各タスクにおいて被験者には，修正されたバグの大まかな症状と被変更メソッドの大まかな仕様を記載したドキュメントと，ツールにより表示されるソースコードの差分およびテストケースの実行の差分 (GitHub を除く) が与えられる．被験者にはこれらを見たうえで，被変更メソッドの実行の変化について記述してもらった．記述対象として，メソッドの返り値や特定のフィールド変数など，バグにより不正な値をとっていた要素をあらかじめ指定した．デバッグの前後で同じテストケースを実行した際に，次の 3 つの項目について，被変更メソッドのみから読み取れる範囲内で記述するよう指示した．

1. 記述対象の値がデバッグの前後で異なるために，IN 要素が満たすべき必要十分条件．(配点：2，部分点あり)
2. 1. を満たすときの，記述対象のデバッグ前の値．(配点：1)
3. 1. を満たすときの，記述対象のデバッグ後の値．(配点：1)

ただし，IN 要素とは，被変更メソッドの外部で値が決まる次の 3 つの要素を指す．

- 被変更メソッドの引数
- 被変更メソッドの内部で読み出されるフィールド変数

- 被変更メソッドの内部で呼び出される別のメソッドの返り値

各タスクは制限時間を 20 分とした上で所要時間を計測し，記述の内容を 4 点満点で採点した．

GitHub の差分表示は差分のあるソースファイルのみの表示であり，今回使用したデータは書き換えられたメソッドが 1 つのため注目すべきソースファイルのみが表示されていることになる．この条件にあわせて，被験者が JCompaths または didiffff を使用する場合は，あらかじめ被変更メソッドをソースコードビューに表示させた状態でタスクを開始した．そのため，ファイルやメソッドの選択に関連する機能については，この実験での評価対象からは外れる．

4.3 アンケートによる評価

タスクの実施後，被験者にアンケートをとり，タスクの内容を踏まえた上で，ユーザビリティの観点から JCompaths の性能を評価した．アンケートは，各ツールに対する SUS (System Usability Scale) [1] の質問と，自由記述からなる．

SUS は，ツールやシステムのユーザビリティを評価するためのアンケート手法である．SUS は次の 10 項目の各質問に対し，1(全く思わない) から 5(強く思う) の 5 段階のリッカート尺度を用いて回答する．

Q1 このツールを頻繁に使用したいと思う．
Q2 このツールは不必要に複雑だった．
Q3 このツールが使いやすいと感じた．
Q4 このツールを利用するには，技術者のサポートが必要だと思う．
Q5 このツールの様々な機能は上手くまとまっていると思う．
Q6 このツールは矛盾がとても多いと感じた．
Q7 ほとんどの人がすぐ使いこなせるようになるツールだと思う．
Q8 このツールを使うのがとても面倒だと感じる．
Q9 自信を持ってこのツールを操作できた．
Q10 このツールを使いこなすには事前に多くの知識が必要だと思う．

奇数番目は肯定的な質問であり，各質問に対してそのスコアを (回答) − 1 で定める．偶数番目は否定的な質問であり，各質問に対してそのスコアを 5 − (回答) で定める．こうして得られた各質問のスコアは，4 に近いほどユーザの満足度が高く，0 に近いほど低いことを表す．10 項目のスコアの総和を 2.5 倍し上限を 100 としたものが SUS の総合スコアとなる．

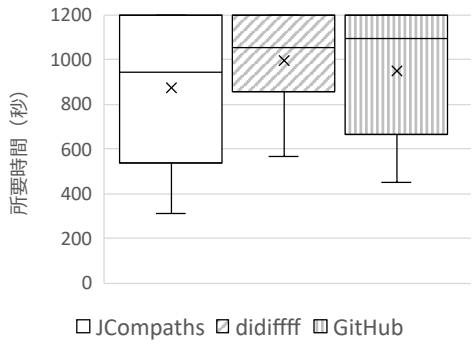

図 8 タスクの所要時間の分布

自由記述では，各ツールに対し，タスクを行う上で便利だった点，不便だった点などについて記述してもらった．また，最後に実験全体についての感想を記述してもらった．

5 ツールの評価結果

4 章で説明した被験者実験の結果と考察を述べる．

5.1 タスクの所要時間

被験者がタスクを行うのにかかった時間について，箱ひげ図によるツールごとの分布を図 8 に示す．平均値は，JCompaths が 14 分 31 秒，didiffff が 16 分 35 秒，GitHub が 15 分 49 秒であった．JCompaths と他のツールとの間で，平均値の差が統計的に有意かを確かめるために，JCompaths の結果を対照群，didiffff と GitHub の結果を処理群とみなし，Steel の方法を用いて有意水準 5% の両側検定を行った．結果，didiffff との比較では，統計検定量を t_{d1} とすると，$|t_{d1}| = 1.04 < d(3, \infty, 0.5; 0.05) = 2.21$ より平均値に有意差は見られなかった．GitHub との比較では，検定統計量を t_{G1} とすると，$|t_{G1}| = 0.66 < d(3, \infty, 0.5; 0.05) = 2.21$ より，こちらも平均値に有意差は見られなかった．

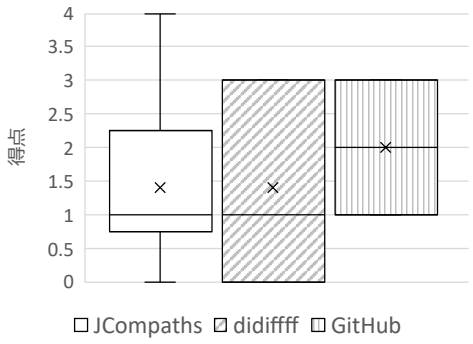

図 9　タスクの点数の分布

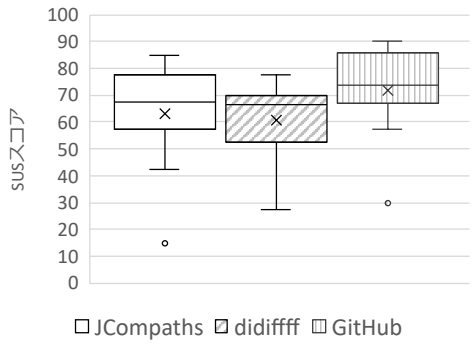

図 10　SUS の総合スコアの分布

5.2　タスクの点数

被験者のタスクの点数について，箱ひげ図によるツールごとの分布を図 9 に示す．平均値は，JCompaths が 1.4 点，didiffff が 1.4 点，GitHub が 2 点であった．JCompaths と他のツールとの間で，平均値の差が統計的に有意かを確かめるために，JCompaths の結果を対照群，didiffff と GitHub の結果を処理群とみなし，Steel の方法を用いて有意水準 5% の両側検定を行った．結果，GitHub との比較では，検定統計量を t_{G2} とすると，$|t_{G2}| = 1.50 < d(3, \infty, 0.5; 0.05) = 2.21$ より平均値に有意差は見られなかった．

5.3　SUS のスコア

被験者へのアンケートから得られた SUS の総合スコアについて，箱ひげ図によるツールごとの分布を図 10 に示す．平均値は，JCompaths が 63.25，didiffff が 60.5，GitHub が 72 であった．JCompaths と他のツールとの間で，平均値の差が統計的に有意かを確かめるために，有意水準 5% で Bonferroni 法による多重比較を行った．ただし，2 群間の比較は対応のある t 検定 (両側検定) により行った．結果，JCompaths と didiffff の比較では，$t(9) = 0.67, p = 0.52 > 0.025$ より平均値に有意差は見られなかった．JCompaths と GitHub の比較では，$t(9) = 0.95, p = 0.37 > 0.025$ より，こちらも平均値に有意差は見られなかった．また，各被験者が最も高く評価したツールに着目すると，3 人が JCompaths，1 人が didiffff，7 人が GitHub を最も高く評価しており，GitHub が優勢ではあるが個人差があった．

さらに，SUS の 10 個の質問それぞれについて，JCompaths と他のツールとの間でスコアの平均値の差が統計的に有意かを確かめるため，同様に有意水準 5% で Bonferroni 法による多重比較を行った．その結果，3 つの質問に対して平均値の差が有意となるツールの組が存在することがわかった．これらの質問について，スコアの分布を図 11 に示す．

「Q1. このツールを頻繁に利用したいと思う」に対するスコアは，JCompaths の平均値が 2.9，didiffff の平均値が 2.1 であった．これらを比較したところ，$t(9) = 2.75, p = 0.022 < 0.025$ より，平均値の差が有意であることがわかった．

「Q4. このツールを利用するには，技術者のサポートが必要だと思う」に対するスコアは，JCompaths の平均値が 1.6，GitHub の平均値が 3.2 であった．これらを比較したところ，$t(9) = 3.36, p = 0.008 < 0.025$ より，平均値の差が有意であることがわかった．

「Q10. このツールを使いこなすには事前に多くの知識が必要だと思う」に対するスコアは，JCompaths の平均値が 1.9．GitHub の平均値が 3.3 であった．これらを比較したところ，$t(9) = 4.12, p = 0.003 < 0.025$ より，平均値の差が有意であることがわかった．

5.4　自由記述

被験者へのアンケートで得られた各ツールおよび実験全体に対する自由記述をいくつか記載する．「便利だった点，良かった点」を「+」，「不便だった点，

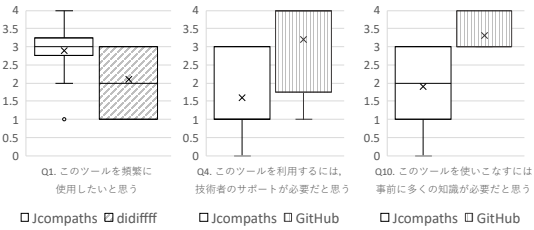

図 11 Q1,Q4,Q10 のスコアの分布

悪かった点」を「−」で表示している．

JCompaths

+) 実行ごとのトレースがすぐ理解できたのが便利だと感じた
+) 実行経路の違いを可視化することで頭でやるよりも考えることが少くて良い
+) didiffff と違い複数の変数実行結果を同時に見れるのが便利だった
−) できることが多いため，何をとっかかりにすれば良いか分からなかった
−) ツールに慣れる時間がもう少し必要であると感じた

JCompaths で行った拡張に対し，9 名中 6 名がソースコードビューにおけるメソッドの一部の繰り返しについての拡張について，9 名中 3 名がトレースビューにおける複数の変数に対する値の同時表示についてポジティブな感想を記述した．一方で，9 名中 3 名からは情報量の多さなどに使いにくさを覚えたというネガティブな感想も得られた．

didiffff

+) 実行時の値が表示されることで，コードの動きを頭の中で想像するのが楽になって便利でした
+) トレース情報の変化がひと目でわかったのでよかった
−) 複数の変数の差分を同時に表示できない点が不便だった
−) 実行回数ごとに表示できないのが不便でした

didiffff および JCompaths に共通するコンセプトであるトレースの可視化に対して 9 名中 3 名がポジティブな感想を記述した．一方ネガティブな感想としては，9 名中 3 名から実行回数ごと・メソッドごとの可視化がされず自身で判断しなければならず不便だったという指摘が，9 名中 1 名から複数のトレースを同時に表示するなどの機能がないことに不便さを覚えたという意見がそれぞれあった．

GitHub

+) 普段から見慣れている
+) シンタックスハイライトがある
+) 差分箇所を変更された文字列単位でハイライトしてくれるのは見やすくてよい
−) コードの変化から動作の変化を考える必要があるので，バグが直っているかを判断するのは大変だと思いました
−) 変更による実行の影響範囲がわからないためタスクに時間がかかった
−) ループの実行ごとの変数の値を追うのが大変で不便だった

機能の単純さに使いやすさを覚えたという意見が多く，各種ハイライト機能を高く評価する意見も見られた．一方で，タスクの内容に対して実行時の情報が無いことに不便さを覚えたという意見もあった．

実験全体

+) コードレビューの時に実行経路が可視化されているのは本当に助かる気がする
−) タスク自体を理解するのがやや困難だった
−) タスクごとの難易度に差があるように感じた

トレースや実行経路の有用性に対する感想や，タスクの難易度について指摘する感想が見られた．

5.5 考察

ここではまず，実験の目的として設定した 2 つの項目について考察を行い，改善に向けた課題を挙げる．

1. ツールのコンセプトである，ソースコードの変更前後において，動的な情報を静的な差分にあわせて可視化することはその変化を理解する助けとなるか

タスクの点数の分布からは，明確に理解の助けになるとは言えない結果となった．一方，動的な情報の差分というこれまであまり扱われていなかった情報を提示したにも関わらず，慣れ親しんだツールと同等のタスクの点数及び SUS のスコアを得られており，自

由記述からも JCompaths による動的な差分そのものに対する不満は見られなかったことから，このような情報の利活用については開発者に一定の理解を得られるものであるととらえている．

2. JCompaths における繰り返しを有するメソッドに対する可視化の拡張は有効に働くか

本実験は繰り返しを有するメソッド実行を対象にしており，JCompaths の機能を生かすことで理解が進むと予想した．しかしながら，タスクの所要時間と点数の両方において，JCompaths と他のツールとの間で統計的に有意な差は見られなかった．

SUS の総合スコアについては，個人差が大きく出る結果となり統計的に有意な差は見られなかったが，質問ごとのスコアではいくつかの特徴がみてとれた．Q1 から，JCompaths が didiffff よりも頻繁に利用したいと思われるツールであることがわかり，動的な差分の表示においては可視化の拡張が被験者に好意的にとらえられていた．自由記述でも，繰り返しに対する実行経路の可視化手法について 9 名中 6 名からポジティブな感想を得られた．このことから，JCompaths の機能拡張は，実験で用いたようなコードにおいて，開発者に一定の理解を得られるものであると考えられる．

一方，Q4 と Q10 からは，JCompaths が GitHub の差分表示に比べ，技術者のサポートが必要で，多くの事前知識が必要だと思われるツールであることがわかった．GitHub や didiffff の差分表示よりも表示する情報や操作する項目が増加する分，インターフェイスの工夫がより重要になると考えられる．

改善に向けた考察

JCompaths を用いた場合のタスクの結果が，他のツールを用いた場合より優れたものにならなかった理由として，以下の点が考えられる．

ツールに慣れるための時間が不足していた． タスクを行う前に，ツールを操作してもらう時間は確保していたが，そのときに表示していたサンプルのソースコードが単純だったこともあり，ツールを使いこなすのに十分な事前知識を得られなかった可能性がある．自由記述でも，JCompaths を使いこなせなかったことが伺える記述が複数見られた．

タスクの指示が理解しづらく，ツールの性能差が現れにくい状況だった． 「バグ修正の内容」にとどまらない「実行の変化に対する理解の度合い」を測るための設計により，タスク自体が複雑なものになってしまった．自由記述でも，タスクの理解が困難だったことが伺える記述が複数見られた．

具体的な値を得られる変数トークンが限られていた． 実行時の具体的な値をトレースとして参照できるのは，プリミティブ型と String 型の変数トークンのみであり，どの被変更メソッドにも中身がわからないオブジェクトが複数存在した．ツールとしては，具体的な値が得られないとしてもその変数トークンにアクセスがあったことを知らせるためにハイライトを行い，またその回数を表示するためにオブジェクト ID の列としてトレースビューへの表示も可能にしているが，今回のタスクではこの機能が被験者を混乱させてしまった可能性がある．

被験者の事前知識による制約があった． 4.1 節で述べた通り被験者間での事前知識には差が少ないが，ツール間での事前知識には差がある状態であった．このため SUS のスコアは，提案ツールであるという先入観から，didiffff と JCompaths 間では JCompaths に有利に働いた可能性がある．使い慣れたツールであるという観点からは，GitHub に有利に働いた可能性がある．

6 ツールの制限

被験者実験の結果も踏まえて，JCompaths をコードレビューに用いる上での制限について整理する．

6.1 メソッド間の情報の欠如

JCompaths は 1 つのメソッド内の差分を見ることに特化しており，メソッド間における情報は取得していない．そのため，例えば，メソッド A の実行を詳しく見る中でメソッド B が呼び出されていた場合，そのままメソッド B の実行を追う機能はない．実行トレース自体はもともとが時系列データであるため，これを活用してメソッド呼び出しの順を追うこと自体は現実的に可能であると考える．しかしながら，今回作成したビューのような形の場合，画面遷移が煩雑

になることも予想される．JCompaths は現状単体テストでカバーできる範囲の修正に対するレビューを想定しており，この拡張に取り組む場合はまずは単体テストの範囲でどの程度メソッド間の動きを追跡できれば良いのかを検討する必要があると考えられる．

6.2 オブジェクト内部の可視化の限界

具体的な値を表示する対象はプリミティブ型と String 型に限っているため，それら以外の変数トークンについてはその内容は可視化されない．また，配列についても配列中の要素を 1 つの変数トークンとして扱って可視化しており，配列全体の値を可視化するような仕組みは現状備えていない．コレクションやラッパークラスについても他のオブジェクトと同様の扱いとしており，内部のデータを読みだす機構はそなえていない．

一方，Collector-sahab にならってオブジェクトの内部変数の状態を取得した場合，その差分の可視化をどのような表現にするかは検討が必要な課題である．Collector-sahab の可視化は異なる状態の初めての出現を注釈として表示するものであり，オブジェクトの複雑さには関係なく，その内部のある変数 1 つの状態が変化したことを示すメッセージがピンポイントで出現する形になる．本手法のように差分のない部分も含めた一連の系列を提示して比較を行う場合は，オブジェクトの内部までを表示しようとするとソースコード上での 1 トークンに付随する情報量が非常に多くなるため，その可視化にはさらなる工夫が必要である．

6.3 大量の繰り返しへの対処

メソッドの実行回数が非常に多い場合，トレースに差分のある実行を見つけるために大量の実行を切り替える必要がある．また，ループによる繰り返し回数が非常に多い場合，1 回のメソッド実行におけるトレースが非常に長くなり，表示される矩形も長大になるため，視認性が悪化する．

メソッドの実行回数への対処としては，数値による直接指定や差分がある回までのスキップ機能を実装することで改善が見込まれる．

また，ループによる繰り返し回数に関連して，Shimari らによれば [16]，実行のすべてを記録するのではなく，大量に繰り返されるような変数トークンについてはその値の新しい数十から数百件を保持するだけでも，デバッグに必要な情報が一定程度欠損せず残っていることが明らかになっている．このような機構と組み合わせた可視化も対処法の候補である．

6.4 並行処理の可視化

繰り返し以外で同一行が複数回実行される別の要因として，マルチスレッドのような並行処理があげられる．今回の可視化手法は，並行処理を想定したものではない．実行時情報の取得に用いた SELogger ではスレッド番号を記録することが可能なため，この情報に，矩形表示や値の対応付けの拡張を行うことで対応できる可能性がある．

7 まとめ

本研究では，コードレビューの支援を目的とし，Java プログラムのソースコードの変更前後におけるメソッドの実行の変化を可視化するツール JCompaths を作成した．JCompaths は，メソッドの実行を 1 回ずつ順に比較し，変数トークンのトレースに加えて実行経路の差分を表示することで，複数の変数トークン間における値の依存関係を明らかにする．

また，このツールの有用性を評価するために，JCompaths 拡張前の試作ツール didiffff とあわせて被験者実験を行った．被験者には，デバッグによるメソッドの実行の変化について記述するタスクを，JCompaths，didiffff，GitHub(実行の可視化なし) の 3 つのツールを使い行ってもらった．その後，SUS の質問と自由記述からなるアンケートをとった．結果として，タスクの所要時間と点数，および SUS の総合スコアのいずれにおいても，JCompaths と他のツールとの間で統計的に有意な差は見られなかった．しかしながら，自由記述からは，メソッドの実行ごとの比較や繰り返しに対する実行経路の表示など，JCompaths 独自の機能が，実験で用いたような繰り返し実行を含むコードに対しては役立つことが確認できた．

今後の課題として，可視化による効果を高めるた

め，6 章で述べたツールの制限にどこまで対処すべきかを実証的に調査し，拡張を行うことが考えられる．また，被験者実験についても，バグ修正の影響がメソッドをまたぐ場合や繰り返しが膨大な場合など，評価できていない状況が存在する．これらに対応するための追加実験や，タスクの内容の見直しを行い，実行時の情報がコードレビューにもたらす効果をうまく測定できるよう改善していきたいと考えている．

謝辞　本研究は，JSPS 科研費 JP21H04877，JP21K02862，JP21K18302，JP22K11985，JP23K16862，JP23K24823，JP23K28065，JP24H00692，JP24K14895　の助成を受けたものです．

参 考 文 献

[1] Brooke, J.: SUS: A “quick and dirty” usability scale, *Usability Evaluation In Industry*, Taylor & Francis, 1996, pp. 189–194.

[2] Deknop, C., Mens, K., Bergel, A., Fabry, J., and Zaytsev, V.: A scalable log differencing visualisation applied to COBOL refactoring, in *Proceedings of the 9th Working Conference on Software Visualization*, 2021, pp. 1–11.

[3] Ebert, F., Castor, F., Novielli, N., and Serebrenik, A.: An Exploratory Study on Confusion in Code Reviews, *Empirical Software Engineering*, Vol. 26, No. 1 (2021).

[4] Etemadi, K., Sharma, A., Madeiral, F., and Monperrus, M.: Augmenting diffs with runtime information, *IEEE Transactions on Software Engineering*, Vol. 49, No. 11 (2023), pp. 4988–5007.

[5] Free Software Foundation: GDB: The GNU Project Debugger, https://www.sourceware.org/gdb/. 2024 年 7 月 11 日閲覧.

[6] Jiang, S., Armaly, A., and McMillan, C.: Automatically generating commit messages from diffs using neural machine translation, in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 135–146.

[7] Just, R., Jalali, D., and Ernst, M. D.: Defects4J: a database of existing faults to enable controlled testing studies for Java programs, in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[8] Kanda, T., Shimari, K., and Inoue, K.: didiffff: A viewer for comparing changes in both code and execution traces, in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 528–532.

[9] Lewis, B.: Debugging Backwards in Time, *CoRR*, Vol. cs.SE/0310016(2003).

[10] Liu, Z., Xia, X., Treude, C., Lo, D., and Li, S.: Automatic generation of pull request descriptions, in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 176–188.

[11] 松村俊徳, 石尾隆, 鹿島悠, 井上克郎: REMViewer: 複数回実行された Java メソッドの実行経路可視化ツール, コンピュータソフトウェア, Vol. 29, No. 1 (2012), pp. 78–84.

[12] Minelli, R., Mocci, A., and Lanza, M.: I know what you did last summer-an investigation of how developers spend their time, in *Proceedings of the 23rd international conference on program comprehension*, 2015, pp. 25–35.

[13] Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M.: Visualizing programs with Jeliot 3, in *Proceedings of the working conference on Advanced visual interfaces*, 2004, pp. 373–376.

[14] Oracle: jdb - The Java Debugger, https://docs.oracle.com/en/java/javase/21/docs/specs/man/jdb.html. 2024 年 7 月 11 日閲覧.

[15] Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A.: Modern code review: A case study at Google, in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.

[16] Shimari, K., Ishio, T., Kanda, T., and Inoue, K.: Evaluating the effectiveness of size-limited execution trace with near-omniscient debugging, *Science of Computer Programming*, Vol. 236, No. 103117(2024).

[17] Shimari, K., Ishio, T., Kanda, T., Ishida, N., and Inoue, K.: NOD4J: Near-omniscient debugging tool for Java using size-limited execution trace, *Science of Computer Programming*, Vol. 206, No. 102630(2021).

[18] Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., and Maia, M. A.: Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J, in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 130–140.

[19] Tao, Y., Dang, Y., Xie, T., Zhang, D., and Kim, S.: How do software engineers understand code changes? an exploratory study in industry, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

神田哲也

2011 年大阪大学基礎工学部情報科学科卒業．2016 年同大学大学院情報科学研究科博士後期課程修了．同年奈良先端科学技術大学院大学情報科学研究科博士研究員．2017 年大阪大学大学院情報科学研究科特任助教．2018 年同助教．2024 年ノートルダム清心女子大学情報デザイン学部准教授．博士 (情報科学)．ソフトウェア進化，ソースコード解析に関する研究に従事．情報処理学会，IEEE 各会員．

橋本悠樹

2023 年大阪大学基礎工学部情報科学科卒業．在学中，プログラム理解と可視化に関する研究に従事．

嶋利一真

2017 年大阪大学基礎工学部情報科学科卒業．2019 年同大学大学院情報科学研究科博士前期課程修了．2022 年同大学大学院情報科学研究科博士後期課程修了．同年奈良先端科学技術大学院大学先端科学技術研究科助教．博士 (情報科学)．プログラム解析，プログラミング教育に関する研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE 各会員．

肥後芳樹

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程了．2007 年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助教．2015 年同准教授．2022 年同教授．博士 (情報科学)．ソースコード分析，特にコードクローン分析，リファクタリング支援，ソフトウェアリポジトリマイニングおよび自動プログラム修正に関する研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE 各会員．