特別研究報告

題目

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

指導教員
肥後 芳樹 教授

報告者
橋本 悠樹

令和 5 年 2 月 7 日

大阪大学 基礎工学部 情報科学科

令和 4 年度 特別研究報告

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

橋本 悠樹

内容梗概

ソフトウェア開発におけ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ため，広く普及している．コードレビューにおいて，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ため，本質的な欠陥を指摘することは難しい．そのため，コミットメッセージ等により，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が，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することはあまりない．

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，複数の変数トークン間における値の依存関係を明確にしている．

また，JCompaths の有用性を評価するために被験者実験を行った．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，各ツールのユーザビリティを測るためのアンケートを実施した．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実行経路の表示など，JCompaths 独自の機能がユーザにとって役立つことが確認できた．

主な用語

コードレビュー
可視化
トレース
実行経路

目 次

1　まえがき

ソフトウェア開発におけ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普及している [11].

コードレビューにおいて，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ため，本質的な欠陥を指摘することは難しい．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り，これらを自動生成する研究も進められている [5][9]．その一方で，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研究は少ない．

didiffff[7] は，Java 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における変化を示すツールである．各変数トークンは，そのトレースの長さや値の変化に応じて色分けされ，選択すると具体的な値を比較できる．しかし，このツールは，トレースの各値がプログラムの実行全体におけるどの時点の値なのかを明らかにしない．そのため，ソースコードに繰り返し実行された部分がある場合，複数の変数トークン間における値の依存関係がわからず，メソッド全体の動きを捉えるには情報が不足していると考えられる．

そこで本研究では，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より詳しく可視化できるツール JCompaths を作成する．具体的な可視化の方法としては，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確にする．

また，JCompaths がメソッドの実行の変化を理解する助けとなることを確かめるため，評価実験を行う．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する．タスク後にはアンケートをとり，SUS[3] と自由記述により，ユーザビリティの側面からも JCompaths を評価する．

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 章では作成したツールが抱える問題点について述べ，最後に 7 章で本研究のまとめを行う．

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に関する先行研究について述べる．また，本研究で作成したツールに深く関わる可視化ツールとして，REMViewer[16] と didiffff[7] を紹介する．

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 [11].

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．限られたコストと期間でのレビューにおいては，指摘が軽微な欠陥に偏った場合に，本質的な欠陥が除去しきれないままレビューが完了してしまうという問題点が指摘されている [14].

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 [5][9]．一方，本研究では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する．ソースコードの変更前後におけるプログラムの実行の変化を理解できるように，オブジェクト指向プログラミングで実行の単位となる，メソッドを対象に詳細な可視化を行う．これにより，コードレビューの支援を目指す.

2.2　実行の可視化に関する先行研究

プログラムの実行の可視化については，既に様々な先行研究が存在する．デバッグの支援を目的に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[1]，Java では jdb[2] 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．これに対し，比較的新しいデバッグ手法として，Omniscient Debugging[8] 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できる．本研究で実装を参考にした NOD4J[13] などは，この手法に基づく，デバッグのための可視化機能を備えたツールである．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[10] などが知られている．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．以上のツールは，いずれも 1 回のプログラム実行が可視化の対象であり，本研究のように，2 回のプログラム実行間の比較は行っていない．

小山らによるプログラム理解支援ツール [15] は，その機能の 1 つとして，1 行のソースコード編集によるプログラムの実行の変化を可視化する．このツールは，プログラムの実行経路と変数のトレースを可視化の対象としており，本研究と着眼点がよく似ている．ただ，実行間のトレースの差分に着目した可視化を行っていないため，トレースの差分を見つけるためには，ユーザが値を 1 つずつ見比べて確認する必要がある．一方，本研究では，トレースに差分のあるトークンを色を変えて強調することで，トレースの差分を容易に発見できる．

COLLECTOR-SAHAB[4] 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の状態を特定する．そのうち実行の最初に現れた状態を，ソースコード中に埋め込んで表示する．すなわち，可視化する差分はツールが自動的に決定する．一方，本研究では，ユーザが差分のある場所を自由に選択し，変数の値を比較することができる．また，COLLECTOR-SAHAB は，本研究の拡張元となった didiffff[7] との比較実験において，差分の検出精度が didiffff よりも優れていることがわかっている．

2.3 REMViewer

REMViewer[16] は，Java メソッドの複数の実行経路を可視化するツールである．ここでの実行経路とは，プログラム実行時におけるソースコードの各文の実行系列を指す．プログラムの実行を記録し，ユーザの指定したメソッドについて，その実行を実行経路に基づき分類する．さらに，それぞれの分類の代表を可視化して表示し，実行経路と局所変数の状態を比較できる．

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着色された矩形を表示する．本研究で作成したツールにもこの表示を導入している．その表示例を図 1 に示す．このソースコードの各行は，図右側の数字（1 ～ 8）の順に実行されるが，このときの矩形は図左端のようになる．ただし，矩形の水平方向の座標（図の x 方向) は，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インクリメントされる（図右側の数字だと 3 → 4，6 → 7 にあたる）．これらの矩形を上から下，左から右へと辿ることで実行経路が得られる．

REMViewer は，ある 1 回のプログラム実行においてメソッドの複数の実行を比較するが，本研究のように，2 回の異なるプログラム実行間でメソッドの実行を比較することはできない．また，REMViewer における実行の比較は，各実行のビューを横に並べるのみであるが，本研究では，実行中で差分のあった箇所を他と異なる色で表現するなど強調して表示して

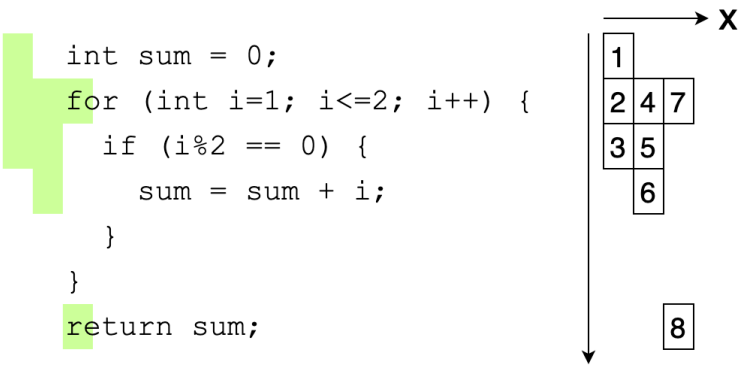

図 1: 矩形による実行経路の表示例

いる．

2.4 didiffff

didiffff[7] は，Java プログラムについて，ソースコードの変更前後における実行の変化を可視化するツールである．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明らかにする．

以下では，didiffff の詳細な仕様と，本研究で解決を目指す didiffff の問題点について述べる．

2.4.1 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 の diff ツールで比較し，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．以降では，“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を同じものとして扱う．

2.4.2 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，次の手順に従って差分のステータスを割り当てる．

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．ただし，トレースの長さが 0 であるとは，変数トークンが実行時に 1 度も参照されなかったことを表す．

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，**“diff in length”** とする．

4. 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．プリミティブ型と String 型以外の変数トークンは，参照された回数しかわからず，具体的な値を取得できないため，扱いに差が生じている．

5. ソースコード変更前後のトレースが完全に一致する場合，**“no diff”**，そうでない場合，**“diff in trace”** とする．

2.4.3 ビューアの仕様

didiffff のビューアは 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 2 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．

ファイルツリービュー

まず，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できる．ファイル名の先頭についている 2 つの円は，そのファイルにおける差分の有無を表す．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 trace” 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は，そのディレクトリ中の全てのファイルを対象とした差分の有無を表す．これにより，注目すべきファイルを容易に発見できる．

ソースコードビュー

次に，ソースコードビューでは，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，トレースビューに表示する変数トークンをクリックで選択できる．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．また，各変数トー

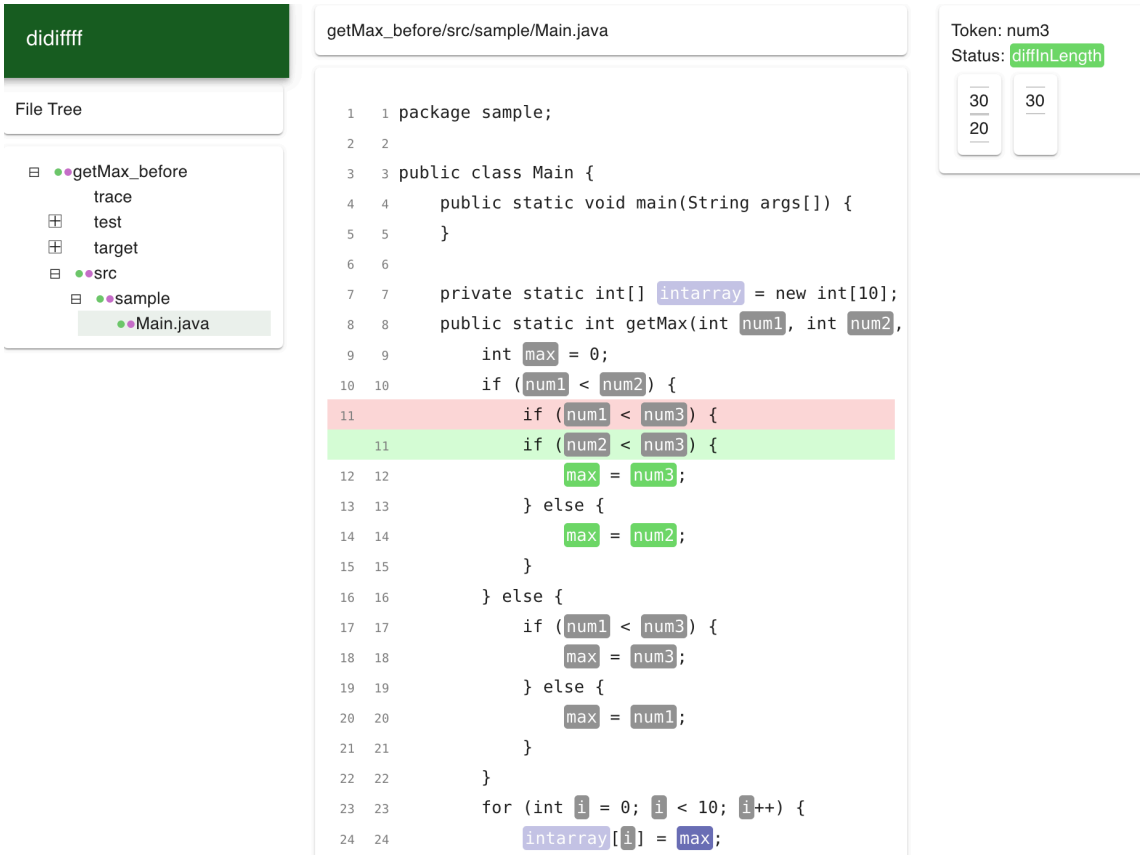

図 2: didiffff のビューアの概観

表 1: 変数トークンのステータスと色の対応

“no trace”	白（ハイライトなし）
“trace 1 only” “trace 2 only” “no diff”	灰
“same length object”	薄い青
“diff in length”	緑
“diff in trace”	濃い青

クンは，そのステータスに応じて表 1 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．

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．左に縦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．

図 2 に表示されている例では，Main.java の 12 行目にある変数トークン `num3` を選択している．そのトレースより，ソースコード変更前は 12 行目が 2 回実行されており，1 回目は `num3` の値が 30，2 回目は 20 だったことがわかる．さらに，ソースコード変更後のトレースが短くなっていることから，直前の条件式の修正により実行経路に変化があったことがわかる．このように，ビューアを用いることで，各変数トークンがソースコードの変更からどのような影響を受けているかを確認できる．

2.4.4 問題点

本研究で取り上げる didiffff の問題点として，トレースの各値が，プログラムの実行全体におけるどの時点の値かわからないことが挙げられる．1 つの変数トークンに着目した場合，トレースの値はとられた時系列順に並んでいるが，複数の変数トークン間では値がどのような順序でとられたかわからない．そのため，メソッドが複数回実行されたり，for などのループによる繰り返しがあった場合，複数の変数トークン間における実行時の値の依存関係が不明となる．したがって，メソッド全体の動きを捉えるには情報が不足していると考えられる．

この問題点について，Defects4J[6] に含まれるバグの修正を例に，詳しく説明する．Defects4J は，様々な OSS の開発中に実際に生じた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，デバッグにより結果が変化した JUnit テストケースが用意されている．ここでは，Defects4J の Math 82 を対象とする．このバグは，`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，バグが修正された．

Math 82 のデバッグ前後におけるテストケースの実行を，didiffff で可視化すると図 3 のようになる．図上側はソースコードビューで，`getPivotRow` メソッドに注目している．このメソッドは，返り値である `minRatioPos` の値を 86 行目で決定している．図左下は，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 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．しかし，これらの値がメソッドの何回目の実行でとられたのかはわからない．

一方，86 行目の実行は 84 行目の条件式で制御され，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．`minRatio` のトレースは，デバッグ前は 21 個，デバッグ後は 9 個の値を含むが，メソッドの何回目の実行において，for ループの何回目でとられた値なのかはわからない．したがって，`i` が異なる値をとったときの `minRatio` の値がどれなのかを判断することができない．

このように，didiffff では，トレースの各値がプログラムの実行全体におけるどの時点の値なのかがわからないので，複数の変数トークン間における値の依存関係を追えない場合が多い．これは，メソッド全体の動きを捉える上でかなり不便である．

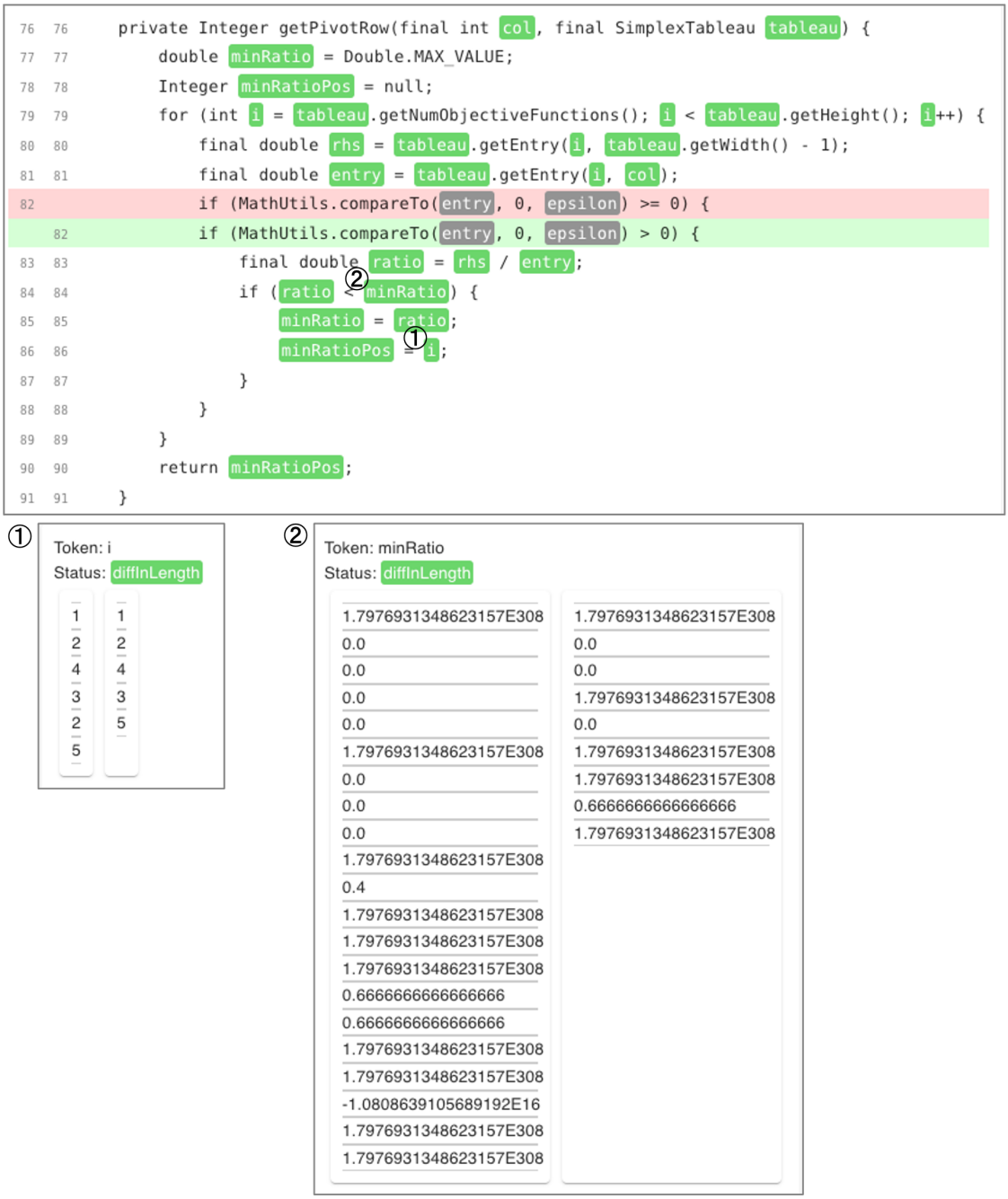

図 3: didifffff で可視化した Math 82 の実行

3　ツールの仕様

この章では，本研究で作成した，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 の仕様，および使用例について述べる．JCompaths は，2.4.4 項で述べた didiffff の問題点の解決を目指し，didiffff を拡張する形で作成した．

JCompaths は，ソースコードの繰り返し実行される部分に着目し，トレースの各値が何回目の繰り返しにおける値なのかを明らかにする．これにより，複数の変数トークン間における実行時の値の依存関係がわかり，メソッド全体の動きを捉えやすくなる．ここでは，ソースコードの繰り返し実行される部分を，その粒度から次の 2 つに分類し，それぞれの観点から didiffff を拡張する．

- メソッド全体の繰り返し．同一のメソッドが複数回呼び出される場合，メソッド全体が繰り返し実行される．これに対し，各変数トークンのトレースをメソッドの 1 回の実行ごとに比較，および表示することで，トレースの各値が何回目のメソッド実行における値なのかを明らかにする．詳しい拡張の内容は 3.2 節で述べる．

- メソッドの一部の繰り返し．メソッド内で for 文や while 文によるループが複数回回る場合，メソッドの一部が繰り返し実行される．これに対し，2.3 節で述べた REMViewer の矩形表示を応用し，実行経路の差分を表示することで，トレースの各値が何回目のループにおける値なのかを明らかにする．詳しい拡張の内容は 3.3 節で述べる．

3.1　処理全体の流れ

JCompaths の実行には，ソースコードの変更前後それぞれについて，プログラムの実行可能 jar ファイルとソースコードが必要である．JCompaths の処理全体の流れを図 4 に示す．以下，この流れを詳しく説明する．

1. ソースコード変更前後における 2 つの実行可能 jar ファイルを実行し，それぞれ実行時の情報を記録する．これには SELogger[16] というツールを用いる．SELogger は Java エージェントであり，プログラムの開始から終了までに実行された全ての命令を網羅的に記録することができる．

2. それぞれの実行時情報の記録，および変更前後のソースコードから，変数トークンのトレースと実行経路を解析する．

   トレースの解析　実行時の情報から，変数が関連する命令を抽出して変数の値を収集し，ソースコード中の変数トークンにマッピングすることでトレースを得る．変

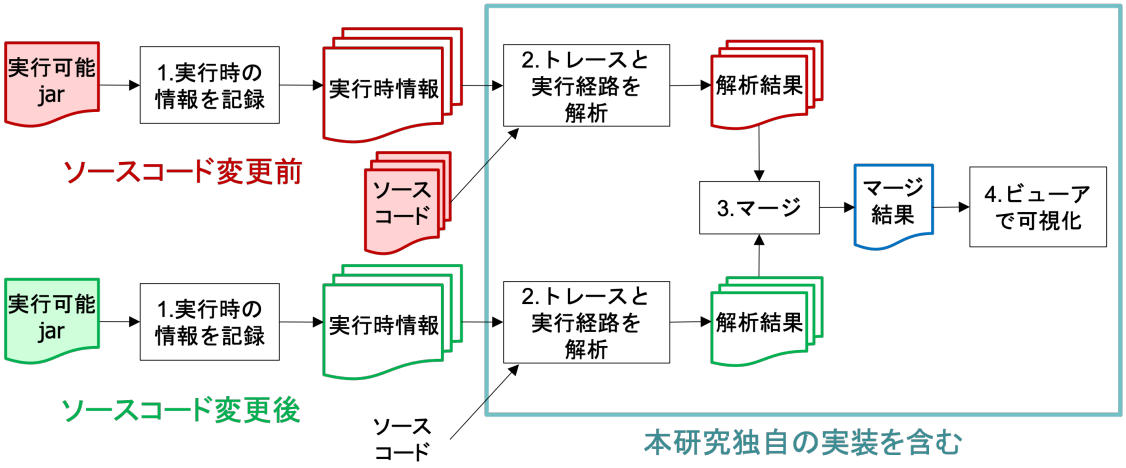

図 4: JCompaths の処理全体の流れ

数がプリミティブ型，String 型，例外のオブジェクトのいずれでも無い場合は，変数の値としてオブジェクト ID を収集する．

実行経路の解析 実行時の情報に含まれる全命令から，対応するソースコード中の行番号を収集し，実行された行の系列を得る．それをメソッドの開始命令からメソッドの終了命令までで切り出し，メソッドの 1 回の実行経路を得る．

この処理は，NOD4J[13] というツールの実装を参考にしている．

3. 解析結果をソースコードの変更前後でマージする．このとき，ソースコードの差分とトレースの差分を計算する．基本的な計算方法は 2.4.1 項，2.4.2 項で述べた didiffff のものと同じだが，メソッドの 1 回の実行ごとにトレースを表示できるように拡張を加えている．詳しい拡張の内容は 3.2 節で述べる．

4. マージ結果をビューアで可視化する．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．ビューアの基本的な構成は 2.4.3 項で述べた didiffff のものと同じだが，メソッドの実行を 1 回ずつ比較し，その実行経路を表示できるように拡張を加えている．詳しい拡張の内容は 3.2 節，3.3 節で述べる．

このうち，2. ～ 4. の処理が，本研究独自の実装を含む部分である．

3.2 メソッド全体の繰り返しについての拡張

メソッド全体の繰り返しについての拡張として，各変数トークンのトレースを，メソッドの 1 回の実行ごとに比較，および表示する．これにより，トレースの各値が何回目のメソッ

ド実行における値なのかを明らかにする．

トレースの差分の拡張

メソッド中の各変数トークンに対して，メソッドの 1 回の実行ごとに差分のステータスを割り当てる．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 回目のトレースと呼ぶ．ただし，そのメソッドが i 回未満しか実行されていない場合や，その変数トークンが i 回目の実行で 1 度も参照されなかった場合は，i 回目のトレースの長さは 0 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，2.4.2 項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．あるメソッドがソースコード変更前に実行された回数を N，変更後に実行された回数を M とすると，そのメソッド中の各変数トークンに対して $1 \sim \max(N, M)$ 回目のステータスが定められる．

ビューアの拡張

JCompaths のビューアは，didiffff と同様に，ファイルツリービュー，ソースコードビュー，トレースビューの 3 つの部分からなる．

まず，ファイルツリービューでファイルを選択後，そのファイル中のどのメソッドの実行を比較するかを選択する．ファイル選択後のソースコードビューを図 5 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．メソッドが，ある i に対して，i 回目のステータスが “diff in length” または “diff in trace” の変数トークンを含む場合，その枠は紫色になる．これにより，注目すべきメソッドを容易に発見できる．

メソッドを選択した後は，そのメソッドの何回目の実行を比較するかを選択する．メソッド選択後のソースコードビューを図 6 に示す．このビューでは，選択中のメソッドにおけるソースコードの差分が表示され，右上にそのメソッドが実行された回数と，選択中の実行のインデックスが表示される．選択中の実行はボタンをクリックすることで切り替えられる．実行のインデックスの色は，その実行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 trace” の変数トークンを含む場合，インデックス 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．i 回目の実行が選択されている場合，各変数トークンは i 回目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．“diff in length” と “diff in tarce” については，トレースの長さを別途 3.3 節で述べる矩形表示により表現しているため，didiffff と異なり同色で表示している．また，この場合，トレースビューに

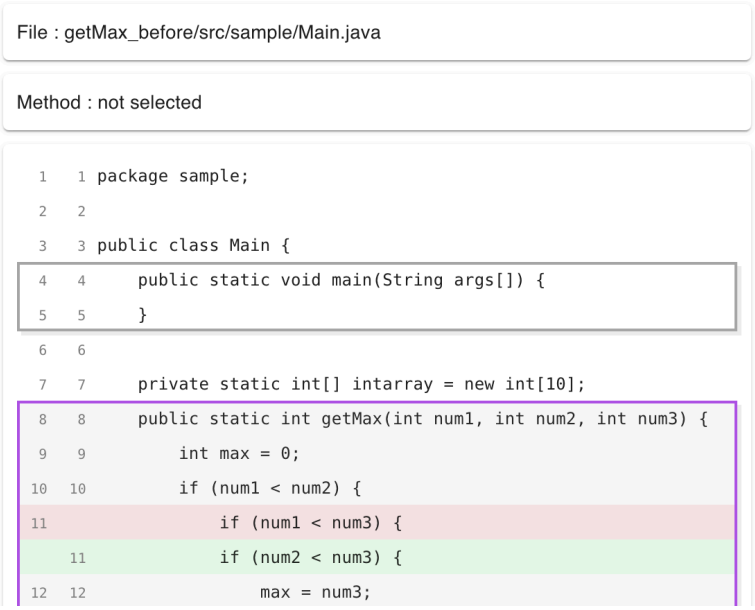

図 5: ファイル選択後のソースコードビュー

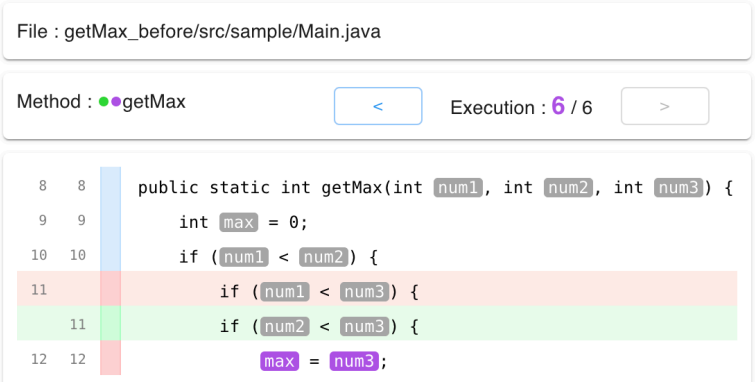

図 6: メソッド選択後のソースコードビュー

表 2: 変数トークンのステータスと色の対応

“no trace”	白（ハイライトなし）
“trace 1 only” “trace 2 only” “no diff”	灰
“same length object”	薄い青
“diff in length” “diff in trace”	紫

は i 回目のトレースしか表示されない．その分，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．

3.3 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，2.3 節で述べた REMViewer の矩形表示を応用し，実行経路の差分を表示する．これにより，トレースの各値が何回目のループにおける値なのかを明らかにする．また，繰り返しと関わりが深い概念である配列についても，そのトレースの表示方法に拡張を加えている．

ビューアの拡張

メソッド選択後のソースコードビューは，REMViewer と同様の矩形を 3 色に拡張して表示する。i 回目の実行が選択されている場合，矩形はメソッドの i 回目の実行経路に対応し，赤の矩形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．矩形は 1 つ 1 つクリックにより選択と解除が可能で，選択中の矩形に対応するトレース中の値は，トレースビューにおいて赤字で表示する．矩形の選択により，トレースが長い場合でも，矩形とトレースの対応関係を見失いづらくなっている．

矩形とトレースの表示例を図 7 に示す．このソースコードは，8 行目で 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．ソースコード変更前はループの 2，4，6 回目で 9 行目が実行され，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．図では，9 行目左辺の変数トークン `sum` が選択されているが，そのトレースの各値が何回目のループにおける値なのか，この矩形により明確になる．また，図では 9 行目の青の矩形が選択されており，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になっている．

図 7: 矩形とトレースの表示例

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情報を提供する．didiffff の場合，変数トークンが存在する行は，そのトレースを確認することで実行の有無を判断できるが，変数トークンが存在しない行については情報を得られない．しかし，JCompaths の場合，変数トークンが存在しない行についても，矩形の有無により実行の有無を判断できる．

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限り，配列中の特定の要素を参照するとき（`array[index]` の形）に，その要素の具体的な値をトレースとして得ることができる．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得ることができる．図 7 右上のトグルボタンにより，didiffff と同様にオブジェクト 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）を切り替えて使用する．配列はループによる繰り返しと関わりが深いため，この機能によりループの動きをより正確に捉えることが期待できる．

3.4 使用例

Defects4J の Math 82 を対象に，JCompaths の使用例を示す．これは，didiffff ではメソッドの動きを捉えるのが難しい例として，2.4.4 項で可視化した実行と同一のものである．

Math 82 のデバッグ前後におけるテストケースの実行を，JCompaths で可視化すると図 8 のようになる．図上側はソースコードビューで，`getPivotRow` メソッドの 5 回目の実行に注目している．図には載せていないが，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いことがわかっている．す

図 8: JCompaths で可視化した Math 82 の実行

なわち，メソッドの 4 回目の実行までは，ソースコードの変更前後で返り値に変化はなかった．図左下は，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，86 行目の矩形により，(i は for ループのカウンタなので当然ではあるが) 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．

一方，図右下は，メソッドの 5 回目の実行において，84 行目の変数トークン minRatio と ratio を選択したときのトレースビューである．84 行目の矩形より，minRatio と ratio は，ソースコード変更前には for ループの 1，2，5 回目で，変更後には for ループの 5 回目で参照されたことがわかる．したがって，ソースコード変更前において，i が 2 のときの minRatio の値は 1.7976931348623157E308，ratio の値は-1.0808639105689192E16 であることがわかり，ratio < minRatio が確認できる．

このように，JCompaths 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格段に整理され，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．

4　ツールの評価方法

この章では，JCompaths の有用性を評価するために行った実験の方法について述べる．本研究では，JCompaths が，繰り返しを有する複雑なメソッド実行に対し，ソースコードの変更前後における変化を理解する助けとなることを確かめるため，被験者実験を行った．実験はタスクとその後のアンケートからなり，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．

4.1　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．

4.1.1　ソースコードの準備

記述の対象とするソースコードは，Defects4J から次の条件を全て満たすものを 3 つ選んだ．

- デバッグにより書き換えられたメソッドがただ 1 つであり，かつそのメソッドが 20 行以上であること．以下では，このメソッドを変更メソッドと呼ぶ．
- 用意された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こと．

加えて，タスクで被験者に提示する情報として，3 つのソースコードそれぞれに対し，修正されたバグの大まかな症状と，変更メソッドの大まかな仕様を記した紙を準備した．これらの情報は，タスクを実際のコードレビューの状況に近づけることをねらいとしている．

4.1.2　タスクの手順

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．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．その後，3 つのタスクを，それぞれ異なるツールを用いて，それぞれ異なるソースコード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，ツールとソースコードの組み合わせ，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 3: 被験者ごとのタスク内容

被験者	1 つ目のタスク	2 つ目のタスク	3 つ目のタスク
A	GitHub + コード 1	didiffff + コード 2	JCompaths + コード 3
B	GitHub + コード 3	JCompaths + コード 1	didiffff + コード 2
C	GitHub + コード 2	didiffff + コード 3	JCompaths + コード 1
D	GitHub + コード 1	JCompaths + コード 3	didiffff + コード 2
E	didiffff + コード 1	JCompaths + コード 2	GitHub + コード 3
F	didiffff + コード 3	GitHub + コード 1	JCompaths + コード 2
G	didiffff + コード 2	JCompaths + コード 3	GitHub + コード 1
H	JCompaths + コード 1	GitHub + コード 2	didiffff + コード 3
I	JCompaths + コード 3	didiffff + コード 1	GitHub + コード 2
J	JCompaths + コード 2	GitHub + コード 3	didiffff + コード 1

被験者には，各タスクにおいて，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を見て，変更メソッドの実行の変化について記述してもらった．ここでは，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，記述対象としてあらかじめ指定した．すなわち，デバッグの前後で同じテストケースを実行した際に，この記述対象の値がデバッグの前後で異なることがある．これを踏まえ，次の 3 つの項目について，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，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．

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

被験者が JCompaths，または didiffff を使用する場合は，あらかじめ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，この実験で評価できていないことに留意したい．

4.2 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)[3] 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測るためのリッカート尺度である．SUS の質問は次の 10 項目からなり，各質問には 1（全く思わない） ～ 5（強く思う）の 5 段階で回答する．

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（回答の番号）– 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 –（回答の番号）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．各質問のスコアを全て足し合わせて 2.5 をかけ，上限を 100 としたものが SUS の総合スコアとなる．

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．

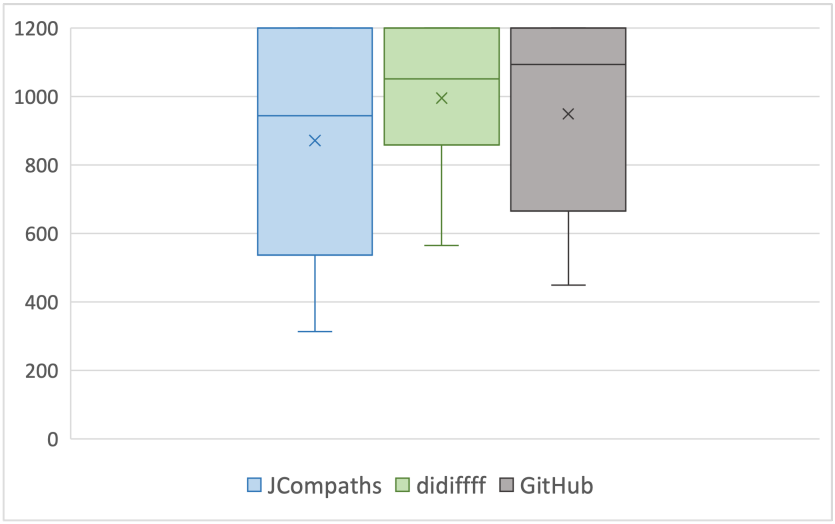

図 9: タスクの所要時間の分布（単位：秒）

5　ツールの評価結果

この章では，4 章で説明した被験者実験の結果について述べ，その結果に対して考察を行う．この章で扱う定量的なデータについては，被験者ごとのデータを付録 A に記載する．

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 9 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．

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 10 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

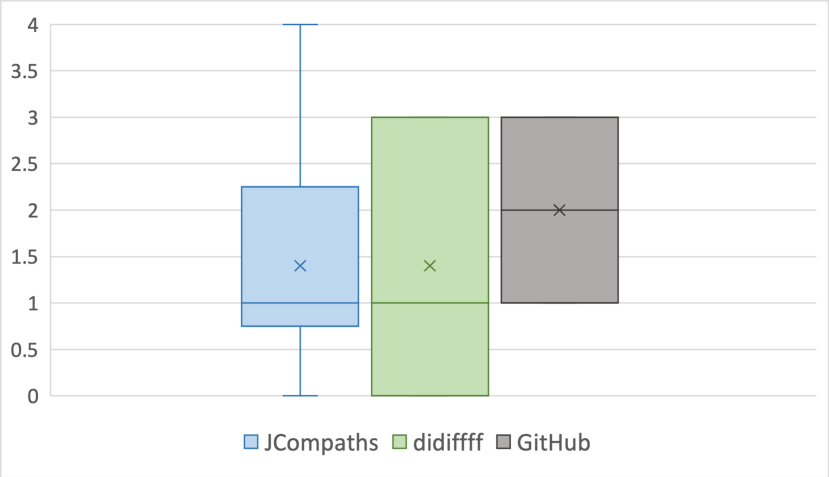

図 10: タスクの点数の分布

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 11 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，やや GitHub に偏りがあるものの，個人差が大きく出る結果となった．

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．この有意な差が認められたスコアについて，箱ひげ図によるツールごとの分布を図 12 に示す．

1 つ目の質問は，「Q1. このツールを頻繁に利用したいと思う」である．この質問のスコアは，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，

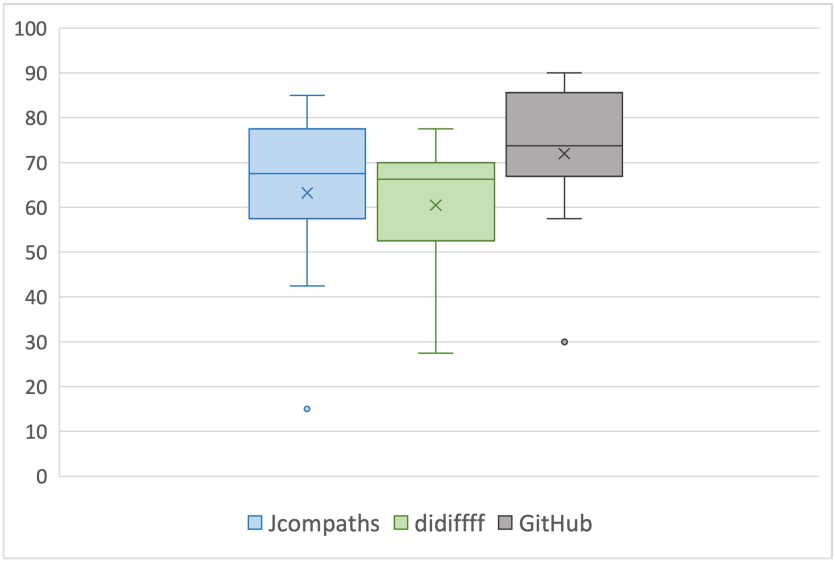

図 11: SUS の総合スコアの分布

$t(9) = 2.75, p = 0.022 < 0.025$ より，平均値の差が有意であることがわかった．

2 つ目の質問は,「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」である．この質問のスコアは，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．

3 つ目の質問は,「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」である．この質問のスコアは，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，各ツールに対する自由記述から，特徴的なものをいくつか記載する．

JCompaths

JCompaths の便利だった点，良かった点としては，以下のような記述があった．

- 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

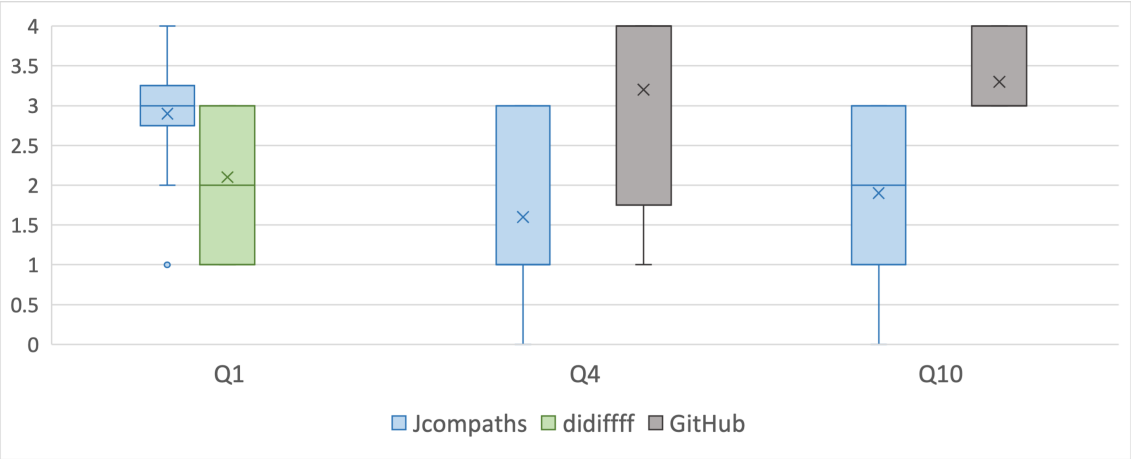

図 12: Q1,Q4,Q10 のスコアの分布

不便だった点．悪かった点としては，以下のような記述があった．

- 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

メソッドの実行ごとの比較や実行経路の表示など，JCompaths 独自の機能を高く評価する意見が多かった一方で，情報量の多さなどに使いにくさを覚えたという意見も複数見られた．

didiffff

didiffff の便利だった点，良かった点としては，以下のような記述があった．

- 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

また，不便だった点．悪かった点としては，以下のような記述があった．

- 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

トレースが見れることを便利に感じたという意見が多く，ひと目で変化に気づけることを高く評価する意見も見られた．一方で，メソッドの実行ごとの表示や，複数のトレースを同時に表示するなどの，JCompaths 独自の機能がないことに不便さを覚えたという意見も多かった．

GitHub

GitHub の差分表示機能の便利だった点，良かった点としては，以下のような記述があった．

- 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

不便だった点，悪かった点としては，以下のような記述があった．

- 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，didiffff と JCompaths には無い独自のハイライトを高く評価する意見も見られた．一方で，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多かった．

実験全体

実験全体の感想としては，以下のような記述があった．

- 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多かった．

5.5 考察

本実験で行ったタスクは，繰り返しを有する複雑なメソッド実行を対象にし，JCompaths が有利になるような設定を意識した．自由記述でも，JCompaths 独自の機能がタスクに役立ったことが伺える記述や，didiffff や GitHub で提供される情報に不足を感じたという記述が多かった．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．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，次の 3 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．「実行の変化に対する理解の度合い」という定量化しづらいものを測るため，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．提供できる情報が中途半端で，被験者を混乱させてしまった可能性がある．

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，こちらも統計的に有意な差は見られなかった．質問ごとのスコアでは，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った．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．JCompaths は GitHub の差分表示よりも格段に機能が多いため，ある程度は仕方ない部分だと考えられるが，より直感的に操作できるようなインターフェースが望ましい．

6　ツールの問題点

この章では，JCompaths を実際のコードレビューに用いる上で，ツールが抱えている問題点について述べる．

メソッド間の情報が欠如している

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できない．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 B の実行を追うことは難しい．まず，B が定義されている場所を探すのに時間がかかる上，B が A 以外からも呼び出されていた場合，B のどの実行が A からの呼び出しによるものか判断できないことがある．JCompaths はあるメソッド全体の動きを捉えるための支援は多いが，プログラム全体の動きを捉えるには情報が足りていないと言える．

中身がわからないオブジェクトが多い

プリミティブ型と String 型以外の変数トークンについて，具体的な値を取得することができない．Java のオブジェクトはフィールド変数で状態を管理することが多いが，プリミティブ型や String 型の変数トークンとしてソースコードに現れない限り，その値はわからない．配列についても，3.3 節で述べたように部分的に値を取得できる場合はあるが，一度に配列全体の値を見ることはできない．

大量の繰り返しを扱いづらい

ソースコードの一部が大量に繰り返されるなど，プログラムの実行の規模が大きい場合，実行時の情報の記録が正常に完了しない場合がある．また，記録が正常に完了したとしても，大量の繰り返しに対しては可視化の側面からも問題がある．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り，時間がかかる．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，表示される矩形の数も莫大なものになるため，可視化性能が著しく低下する．

トレースの不整合が起こる場合がある

特定の条件を満たすソースコードにおいて，変数トークンとトレースの対応が上手くとれず，変数トークンを選択しても正しくトレースが表示されないことがある．例えば，1 つの文が複数行に分割されている場合や，$+=$，$-=$ などの演算子が含まれる場合である．

7　まとめ

本研究では，コードレビューの支援を目的とし，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実行経路の表示など，JCompaths 独自の機能がユーザにとって役立つことが確認できた．

今後の課題として，JCompaths の有用性を示すことができる状況を正確に予測し，タスクの内容を見直すことが挙げられる．また，6 章で述べたツールの問題点にも対処する必要がある．特に，大量の繰り返しへの対応については，Near-Omniscient Debugging[12] などの記録手法とうまく組み合わせることができれば，パフォーマンスの改善が予想される．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 肥後芳樹 教授には，研究活動に対して多くの貴重な御助言や御指導を賜りました．心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠 准教授には，研究室での発表の機会を中心に貴重な御助言や御指導を賜りました．心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田哲也 助教には，研究方針の決定から実験の準備，論文の執筆に至るまで，研究活動のあらゆる面に対して非常に多くの御助言や御指導を賜りました．心より深く感謝いたします．

奈良先端科学技術大学院大学先端科学技術研究科情報科学領域 嶋利一真 助教には，ツールの使用や発表資料の作成，実験の計画など，様々な場面で多くの御助言や御指導を賜りました．心より深く感謝いたします．

肥後研究室の先輩方，および同期の皆様には，研究室での生活や研究活動について日々様々な御助言を賜り，被験者実験においても多大なご協力をいただきました．特に，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 藤原勇真 氏には，発表資料の作成や論文の執筆においても御助言を賜りました．心より深く感謝いたします．

最後に，その他様々な御助言および御指導を頂いた，大阪大学大学院情報科学研究科コンピュータサイエンス専攻肥後研究室の皆様に心より深く感謝いたします．

参考文献

[1] GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/.

[2] jdb - The Java Debugger. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html.

[3] John Brooke. SUS: A "quick and dirty" usability scale. In *Usability Evaluation In Industry*, pp. 189–194. Taylor & Francis, 1996.

[4] Khashayar Etemadi, Aman Sharma, Fernanda Madeiral, and Martin Monperrus. Augmenting Diffs With Runtime Information, 2022.

[5] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 135–146, 2017.

[6] René Just, Darioush Jalali, and Michael D. Ernst. 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*, pp. 437–440, 2014.

[7] Tetsuya Kanda, Kazumasa Shimari, and Katsuro Inoue. didiffff: A Viewer for Comparing Changes in both Code and Execution Traces. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pp. 528–532, 2022.

[8] Bil Lewis. Debugging backwards in time, 2003.

[9] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. Automatic generation of pull request descriptions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pp. 176–188, 2019.

[10] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the working conference on Advanced visual interfaces*, pp. 373–376, 2004.

[11] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th*

International Conference on Software Engineering: Software Engineering in Practice, pp. 181–190, 2018.

[12] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, and Katsuro Inoue. Near-Omniscient Debugging for Java Using Size-Limited Execution Trace. In *35th IEEE International Conference on Software Maintenance and Evolution*, pp. 398–401, 2019.

[13] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, Naoto Ishida, and Katsuro Inoue. NOD4J: Near-Omniscient Debugging Tool for Java Using Size-Limited Execution Trace. *Science of Computer Programming*, Vol. 206, p. 102630, 2021.

[14] 井元薫, 細谷泰夫. コードレビュー手法の改善によるコード品質確保の効率化. 三菱電機技報, Vol. 90, No. 12, pp. 19–22, 2016.

[15] 小山秀明, 山田俊行. 変数の値の変化の可視化によるプログラム理解支援. 情報処理学会論文誌, Vol. 10, No. 4, pp. 1–11, 2017.

[16] 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, pp. 78–84, 2012.

付録

A　被験者実験のデータ

表 4: タスクの所要時間

被験者	GitHub	didiffff	JCompaths
A	13 分 27 秒	9 分 25 秒	7 分 10 秒
B	7 分 29 秒	18 分 58 秒	20 分 00 秒
C	20 分 00 秒	9 分 40 秒	9 分 32 秒
D	20 分 00 秒	15 分 51 秒	15 分 37 秒
E	20 分 00 秒	20 分 00 秒	20 分 00 秒
F	9 分 03 秒	15 分 52 秒	14 分 07 秒
G	20 分 00 秒	20 分 00 秒	17 分 42 秒
H	16 分 28 秒	16 分 07 秒	15 分 15 秒
I	11 分 46 秒	20 分 00 秒	5 分 13 秒
J	20 分 00 秒	20 分 00 秒	20 分 00 秒
平均	15 分 49 秒	16 分 35 秒	14 分 31 秒

表 5: タスクの点数

被験者	GitHub	didiffff	JCompaths
A	1	0	1
B	2	0	1
C	2	3	0
D	2	3	4
E	3	1	1
F	1	2	1
G	3	1	3
H	1	3	1
I	3	0	2
J	2	1	0
平均	2	1.4	1.4

表 6: SUS の総合スコア

被験者	GitHub	didiffff	JCompaths
A	57.5	60	85
B	70	67.5	67.5
C	90	77.5	77.5
D	70	70	72.5
E	85	67.5	77.5
F	82.5	55	42.5
G	70	70	62.5
H	77.5	27.5	15
I	87.5	65	65
J	30	45	67.5
平均	72	60.5	63.25

表 7: Q1,Q4,Q10 のスコア

	Q1		Q4		Q10	
被験者	didiffff	JCompaths	GitHub	JCompaths	GitHub	JCompaths
A	1	4	4	3	3	3
B	3	3	2	2	3	2
C	3	3	4	3	4	3
D	2	3	1	1	3	2
E	3	4	4	1	4	2
F	2	3	4	1	3	3
G	1	2	4	3	4	1
H	1	1	4	0	3	0
I	3	3	4	1	3	1
J	2	3	1	1	3	2
平均	2.1	2.9	3.2	1.6	3.3	1.9