特別研究報告

題目

エディタの作業効率に関する開発者の自己評価と定量的評価

指導教員
井上 克郎 教授

報告者
鬼塚 仙太郎

令和 4 年 2 月 8 日

大阪大学 基礎工学部 情報科学科

令和 3 年度 特別研究報告

エディタの作業効率に関する開発者の自己評価と定量的評価

鬼塚 仙太郎

内容梗概

ソフトウェアのコーディングプロセスを分析するために，バージョン管理システムに記録される情報より粒度の細かい履歴を利用する細粒度開発履歴の研究が数多く行われている．細粒度開発履歴には，開発環境上で開発者が行った操作がすべて記録されているため，詳細な分析が可能である．しかし，従来の細粒度開発履歴の記録手法では利用可能な開発環境が限定されており，異なる開発環境から収集した細粒度開発履歴を用いた分析はほとんど行われていない．

石田らは，様々なエディタが用いられる状況でも細粒度編集履歴の収集が可能なプラットフォームを提案している [11]．石田らの研究における細粒度編集履歴とは，細粒度開発履歴のうちエディタ上でのソースコードの変更操作に限定したものである．

本研究では，複数の開発環境から収集した細粒度開発履歴の分析の一例として，エディタの作業効率に関する開発者の自己評価と定量的評価を収集し，それらを比較して自己評価と定量的評価にギャップがあるかを調査する方法を提案する．自己評価はアンケートを用いて収集し，定量的評価は開発進捗率の時系列変化を表すグラフを用いて収集する．開発進捗率の時系列変化は，開発過程で得られる細粒度編集履歴を用いて計算する．本調査手法では，石田らの細粒度編集履歴収集プラットフォームを活用することで，複数のエディタから同条件で細粒度編集履歴の収集ができるため，比較が可能である．

そして，提案した調査手法の適用例として，Visual Studio Code と Eclipse を対象に，エディタの作業効率に関する開発者の自己評価と定量的評価を収集し比較する実験を行った．実験を通じて，自己評価と定量的評価の収集及びその比較が実現できていることを確認できた．

主な用語

細粒度編集履歴
可視化
コーディングプロセス

目 次

1　前書き

ソフトウェアのコーディングプロセスを詳しく分析するために，Git 等のバージョン管理システムに記録される情報より粒度の細かい履歴を利用する細粒度開発履歴の研究が数多く行われている．細粒度開発履歴とは，開発環境上で開発者が行った操作をすべて履歴として記録したものである．しかし，従来の細粒度開発履歴の記録手法では利用可能な開発環境が限定されていた．そのため，単一の開発環境から収集した細粒度開発履歴の分析は行われているが，複数の開発環境から収集した細粒度開発履歴の分析はほとんど行われていない．例えば，細粒度開発履歴を利用したコーディング過程の可視化に関する先行研究が行われているが [13][9][11]，多くの研究では単一のエディタから収集した履歴を用いており，複数のエディタから収集した履歴を用いているものはほとんどない．

石田らの先行研究 [11] では，多様な開発環境での細粒度編集履歴の収集と利活用を可能にするために，特定エディタに依存しない細粒度編集履歴収集プラットフォームを提案している．石田らの研究における細粒度編集履歴とは，細粒度開発履歴のうちエディタ上でのソースコードの変更操作に限定したものである．

本研究では，複数の開発環境から収集した細粒度開発履歴の分析の一例として，エディタの作業効率に関する開発者の自己評価と定量的評価を収集し，それらを比較して自己評価と定量的評価にギャップがあるかを調査する方法を提案する．具体的には，先行研究 [11] の細粒度編集履歴収集プラットフォームを活用することで，様々なエディタからのデータ収集を実現する方法を説明する．また，アンケートを用いた開発者の自己評価の収集方法，細粒度編集履歴を利用した定量的評価の収集方法，そして収集した自己評価と定量的評価の比較方法を提案する．

そして，提案した調査手法の適用例として，Visual Studio Code と Eclipse を対象に，エディタの作業効率に関する開発者の自己評価と定量的評価を比較する実験を行う．そして実験結果をもとに考察を行う．

以降，2 章では本研究の背景を説明する．3 章では本研究で提案する調査手法について述べる．4 章では調査の一例として行った実験について述べ，5 章で実験結果について示し，6 章で実験結果の考察について述べる．最後に，7 章でまとめを行う．

2　背景

2.1　細粒度開発履歴

ソフトウェアのコーディングプロセスの把握を目的として，ソースコードの時系列変化を扱う研究が数多く行われている．多くの研究では Git 等のバージョン管理システムに記録された情報を利用しているが，コーディングプロセスの把握にはそのような情報では不十分な場合があることが指摘されている [6][7]．そこで，バージョン管理システムに記録される情報よりも細かい履歴を利用する細粒度開発履歴の研究が行われている．

細粒度開発履歴とは，開発環境上で開発者が行った操作をすべて履歴として記録したものである．細粒度開発履歴を利用することで，バージョン管理システムに記録される情報より細かい履歴を利用することができ，より詳細な分析を行うことができる．その応用手法としては，プログラムの理解支援・変更支援，プロセス改善，ソフトウェア進化の解明など多様に提案されている [10]．

しかし，従来の細粒度開発履歴の記録手法では利用可能な開発環境が限定されていた．なぜなら，複数の開発環境からデータを収集するには，その開発環境ごとにそれを取得する機構を準備する必要があり，開発環境への依存が大きいからである．そのため，異なる開発環境から収集した細粒度開発履歴を用いた分析はほとんど行われていない．

2.2　石田らの細粒度編集履歴収集プラットフォーム

石田らは，細粒度編集履歴データが自由に活用されていくような基盤を作ることを目的として，特定エディタに依存しない細粒度編集履歴収集プラットフォームを提案している [11][12]．石田らの研究における細粒度編集履歴とは，細粒度開発履歴のうち，エディタ上でのソースコードの編集操作に限定したものである．このプラットフォームの構成図を図 1 に示す．以下では，このプラットフォーム収集部・分析部の構成，細粒度編集履歴の記録単位，プラットフォームの実装状況について説明する．

細粒度編集履歴の収集部

細粒度編集履歴の収集部は多くのエディタで利用可能な言語サーバーを応用したモジュールに分離されている．エディタと言語サーバ間のやり取りは，標準化されたプロトコルである Language Server Protocol [3]（以後 LSP とする）を通じで行われる．ただし，言語サーバとやり取りを行うエディタ側の実体はエディタのプラグインである．LSP はもともとは Visual Studio Code のために開発されたが，現在では多くのエディタでサポートされている．エディタのプラグインは，LSP を通じてファイルが変更されるたびに言語サーバにイベ

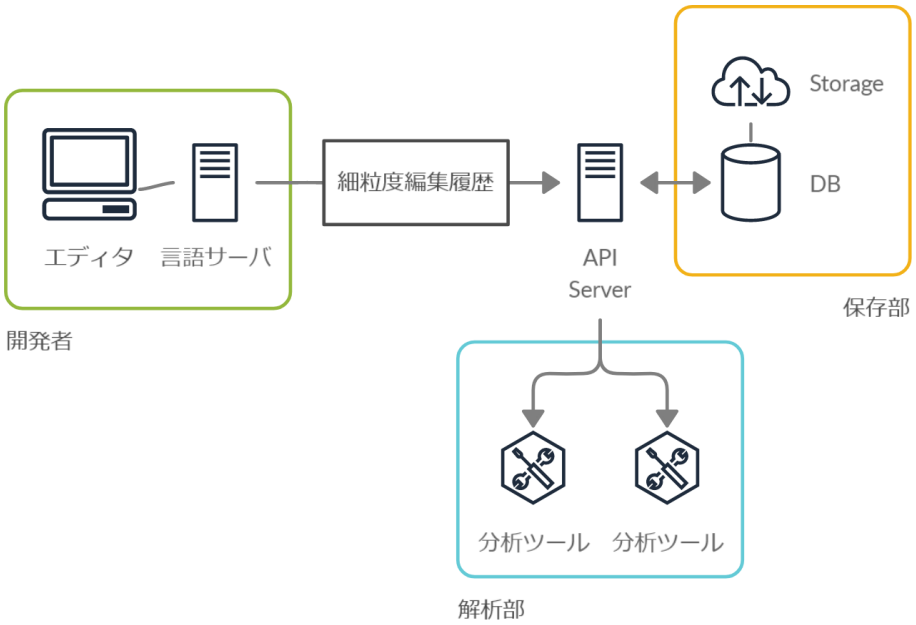


図 1: 細粒度編集履歴収集プラットフォームの構成 （[12]p.1 より引用）

ントを通知する．また，定期的に記録したデータの送信をリクエストする．言語サーバは受け取ったイベントの内容をもとに細粒度編集履歴の記録と送信を行う．

このように収集部が言語サーバを応用したモジュールに分離されていることで，複数の開発環境に展開する場合でも言語サーバは新たに開発する必要がなく，エディタのプラグインだけを開発すればよい．このため，複数の開発環境からの履歴収集が低コストで実現できる．

細粒度編集履歴の分析部

細粒度編集履歴の分析部は，開発環境とは独立した API サーバー上に構築されている．API サーバーは HTTP 通信を通じて細粒度編集履歴を受信し，受け取ったデータを DB サーバに格納する．分析のためにデータを取り出す場合は，API サーバが提供する API を通じて行う．このように分析部がサーバー上に構築されていることで，開発環境によらない細粒度編集履歴の利活用が可能になっている．

細粒度編集履歴の記録単位

言語サーバによって記録される細粒度編集履歴の記録単位について説明する．収集部の説明で述べたように，エディタのプラグインがファイル変更のたびに言語サーバにイベントを通知することで，言語サーバは 1 文字単位で編集内容を受け取る．しかし，言語サーバは受け取った編集内容をそのまま記録するのではなく，桑原らの関連研究 [8] で提案されている

手法を参考に隣接する編集操作を統合する．統合したデータは，エディタからの送信リクエストによって API サーバに送信される．

送信されるデータは，編集履歴や編集開始前のファイル，エディタに関する情報が記載された JSON フォーマットのデータとなっている．具体的に，編集履歴には編集イベントの発生時刻やファイル名，編集内容が記載されている．編集開始前のファイルには，ファイル名と編集開始前のファイルの内容が記載されている．エディタには，編集に使用したエディタの名前が記載されている．

プラットフォームの実装状況

プラットフォームの実装状況について説明する．エディタのプラグインは Visual Studio Code 用のものが実装されている．保存部は，DB サーバには細粒度編集履歴のサマリだけが保管され，具体的な内容はローカルストレージや NextCoud 等のオンラインストレージ上に圧縮ファイルとして保持されるようになっている．API サーバとの分離はされておらず，分析のためにデータを利用する場合は DB サーバやストレージに直接アクセスする必要がある．このように，プラットフォームは完全に実装されているわけではないので，利用する場合は一部実装が必要になる．

2.3 細粒度開発履歴を利用した可視化に関する先行研究

細粒度開発履歴の応用例の 1 つにコーディング過程の可視化がある．可視化は，データだけでは理解しにくい事象の理解を手助けするために使用される．特にコーディング過程で得られる細粒度開発履歴などのデータ量は非常に大きくなるため，可視化による恩恵も大きい．

細粒度開発履歴を用いたコーディング過程の可視化に関する先行研究には以下のようなものがある．

- 藤原ら [13] は，プログラミング演習中の学習者をリアルタイムで自動把握するために，ソースコードの行数，コメントの行数，コンパイルの回数，コンパイル時のエラー数，プログラムの実行回数の可視化を行っている．また，可視化したグラフ上の特徴を定量化する手法の提案も行っている．可視化に使用しているソースコードのログは，エディタ Meadow から取得している．

- 井垣ら [9] は，受講生のプログラミング演習におけるコーディング過程を記録し，相対的な離縁状況を可視化して講師に提示するシステム C3PV を提案している．このシステムでは，総 LOC，課題ごとのコーディング時間，単位時間当たりのエディタ操作

数，課題ごとのエラー継続時間の可視化を行っている．ソースコードのログ取得のために使用するエディタは，C3PV が提供するオンラインエディタを指定している．

- 石田 [11] は，直感的に開発の進捗を理解することや，開発者の傾向を分析することを目的として，セッション中の開発進捗率の時系列変化を可視化している．可視化に使用している履歴は，エディタ Visual Studio Code から取得している．

上記で紹介したように，細粒度開発履歴を用いたコーディング過程の可視化は多くの研究で行われている．しかし，多くの研究では単一のエディタから収集した履歴を用いており，異なるエディタから収集した細粒度開発履歴を用いているものはほとんどない．

3　調査手法

本章では，エディタの作業効率に関する開発者の自己評価と定量的評価を収集し，それらを比較して自己評価と定量的評価にギャップがあるかを調査する方法を提案する．調査の全体図を図 2 に示す．以降では調査方法について詳しく説明する．

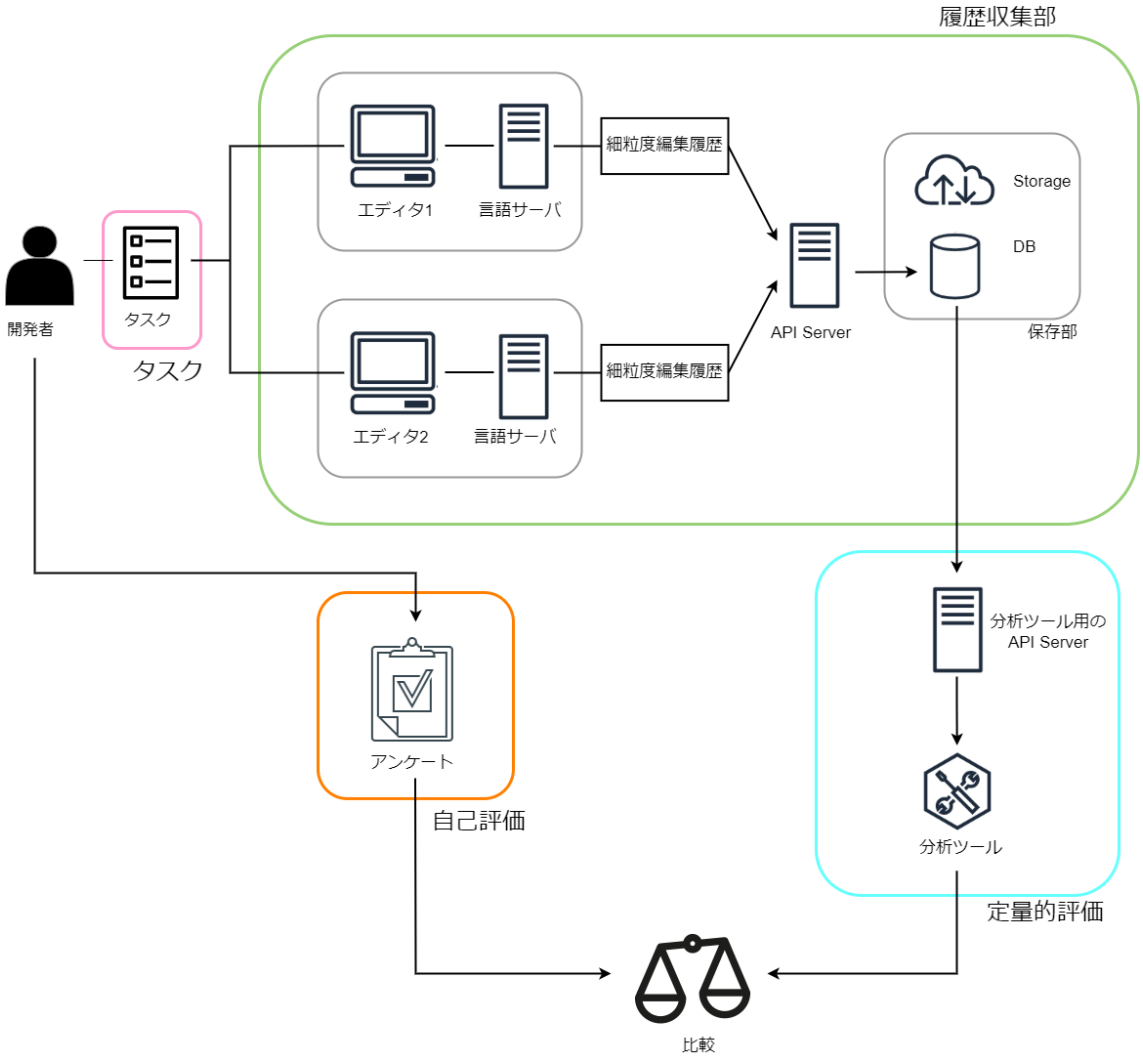


図 2: 調査の全体図

3.1 複数の開発環境からのデータ収集方法

本調査におけるエディタの作業効率に関する定量的評価を行うためには，各開発環境で開発者の操作履歴を収集する必要がある．そこで，本調査では複数の開発環境から細粒度編集履歴を収集するために 2.2 節で述べた石田らの細粒度編集履歴収集プラットフォームを活用する．このプラットフォームは特定のエディタに依存せずに細粒度編集履歴を収集できるように設計されている．このプラットフォームが展開されているエディタを使用して開発を行うことで，その開発過程の細粒度編集履歴の収集が実現できる．

細粒度編集履歴収集プラットフォームを利用した履歴収集のための構成は，図 2 の履歴収集部のようになる．この図における履歴収集の流れを説明する．開発者はプラグインがインストールされたエディタを使用してタスクを行う．エディタのプラグインは，タスク過程でのファイルの変更をイベントとして言語サーバに通知する．言語サーバは受け取ったイベントの内容をもとに API サーバを通じて DB サーバとストレージに細粒度編集履歴を記録する．このようにして複数のエディタから細粒度編集履歴が収集される．

3.2 タスクの内容

本調査では，被験者に作業時間が同程度のタスクを 2 つの開発環境で行ってもらう．その過程で得られるデータを使用して，エディタの作業効率に関する開発者の自己評価と定量的評価を行う．作業時間が同程度のタスクとしては，以下の 2 つを用いる．

タスク 1（プログラミング問題の解答）

タスク 1 では，同程度の難易度のプログラミング問題を，指定したエディタのみを使用して解答してもらう．タスク 1 の詳細について説明する．終了条件は，あらかじめ用意したテストコードを実行し，すべてのテストケースをパスすれば完了とする．コーディング中の注意事項として，リファクタリングは不要であると指示する．持参した参考書やノードなどの参照や，Web ページの閲覧は自由にしてよいが，問題の解答に関する直接的な検索は禁止し，またコピーアンドペーストは禁止する．

タスク 2（ソースコードの写経）

タスク 2 では，ソースコードを写経してもらう．ソースコードの写経とは，用意されたソースコードをそのまま書き写すことである．タスク 2 の詳細について説明する．写経の終了条件は，スペースの数と空行の有無を無視したときのファイルの中身が同じであれば完了とする．具体的には，Linux に用意されている diff コマンドにオプション-Bb を指定して実行し，差分が何も表示されなければ完了とする．コーディング中の注意事項として，コピーの対象を問わず（被験者が作成しているファイル内でのコピーも含む）コピーアンドペーストは禁止する．

3.3 評価手法

3.2 節で述べたタスクを通じて，エディタの作業効率に関する開発者の自己評価と定量的評価を行う．自己評価はアンケートを使用して行い，定量的評価はタスク過程で収集された細粒度編集履歴を分析することで行う．

3.3.1 自己評価

エディタの作業効率に関する開発者の自己評価は，開発者のアンケート結果を用いて評価する．具体的なアンケートを行うタイミングとその内容を表 1 に示す．質問項目 1,2 によって開発前の各エディタでのコーディングのしやすさに関する開発者の体感を評価し，質問項目 4,5,6 によって開発後の体感を評価する．質問項目 7 では，エディタごとにコーディングのしやすさに違いが出た場合の理由を知るために行う．質問項目 3 は，タスク 1（プログラミング問題の解答）において用意する問題の難易度が被験者によってずれる可能性があるため，問題ごとのコーディング時間のずれがエディタではなく問題に依るものなのかを評価するために使用する．

3.3.2 定量的評価

エディタの作業効率に関する定量的評価は，各タスクにおける進捗率の時系列変化のグラフから得られる情報によって評価する．基本的には，タスクの開始時刻と終了時刻の情報を使用する．加えて，適宜グラフの形状を確認する．例えば，タスク 1 において途中でソースコードの変更が行われず長い間進捗率が停滞している箇所があれば，それはエディタの使用でつまずいているのではなく，問題の解答方針を考えている可能性が高いと考えられる．

開発の進捗率の時系列変化は，タスク過程で得られる細粒度編集履歴を用いて計算する．ただし，2.2 節の細粒度編集履歴収集プラットフォームでは，分析のためにデータを取り出すための API が API サーバに用意されていない．そのため，DB サーバやストレージからデータの取得ができる API が用意された API サーバを別で開発し，それを通じて細粒度編集履歴を取得し分析する．

表 1: アンケートの質問項目

No.	タイミング	項目
1	すべてのタスク開始前	エディタ 1 とエディタ 2 それぞれでのコーディングに関して,「大変苦手」から「大変得意」までの 5 段階評価
2	すべてのタスク開始前	タスク 1, タスク 2 それぞれに関して，どちらのエディタのほうがはやくコーディングできると思うか,「エディタ 1 のほうが非常にはやい」から「エディタ 2 のほうが非常にはやい」までの 5 段階評価
3	タスク 1 内の各設問後	各問題に関して,「非常に躓いた」から「非常にすらすら解けた」までの 5 段階評価
4	タスク 1 終了後	タスク 1 全体を通して，どちらのエディタのほうがはやくコーディングできたと思うか,「エディタ 1 のほうが非常にはやい」から「エディタ 2 のほうが非常にはやい」までの 5 段階評価
5	タスク 2 終了後	タスク 2 全体を通じて，どちらのエディタのほうがはやくコーディングできたと思うか,「エディタ 1 のほうが非常にはやい」から「エディタ 2 のほうが非常にはやい」までの 5 段階評価
6	すべてのタスク終了後	各エディタでのコーディングに関して,「大変苦手」から「大変得意」までの 5 段階評価
7	すべてのタスク終了後	フリーコメントとして，エディタごとのコーディングをした感想や，エディタ間でコーディングのしやすさが変わった理由などの記述

3.4　比較方法

比較は「開発前の自己評価と定量的評価の比較」と「開発後の自己評価と定量的評価の比較」の 2 つ行う．また，タスク 1 とタスク 2 のデータは別々で評価する．

具体的な方法としては，3.3.1 節で説明したアンケートの結果と 3.3.2 節で説明した進捗を表すグラフをもとに，図 3 に示すようなグラフを作成し，そのグラフを用いて比較する．このグラフでは，各点が各開発者の自己評価と定量的評価を示している．横軸が自己評価で縦軸が定量的評価である．各点の x 座標が大きいほどエディタ 1 のほうがタスクをはやく終了できる（できた）と自己評価していることを意味し，x 座標が小さいほどエディタ 2 のほうがはやくタスクを終了できる（できた）と自己評価していることを意味する．また，各点の y 座標が大きいほどエディタ 1 のほうがはやくタスクを終了できたと定量的評価されていることを意味し，y 座標が小さいほどエディタ 2 のほうがはやくタスクを終了できたと定量的評価されていることを意味する．

以下では，x 座標と y 座標の具体的な導出方法，つまり自己評価と定量的評価を計算する方法について説明する．また，グラフを用いて自己評価と定量的評価が一致しているかを判定する基準について説明する．

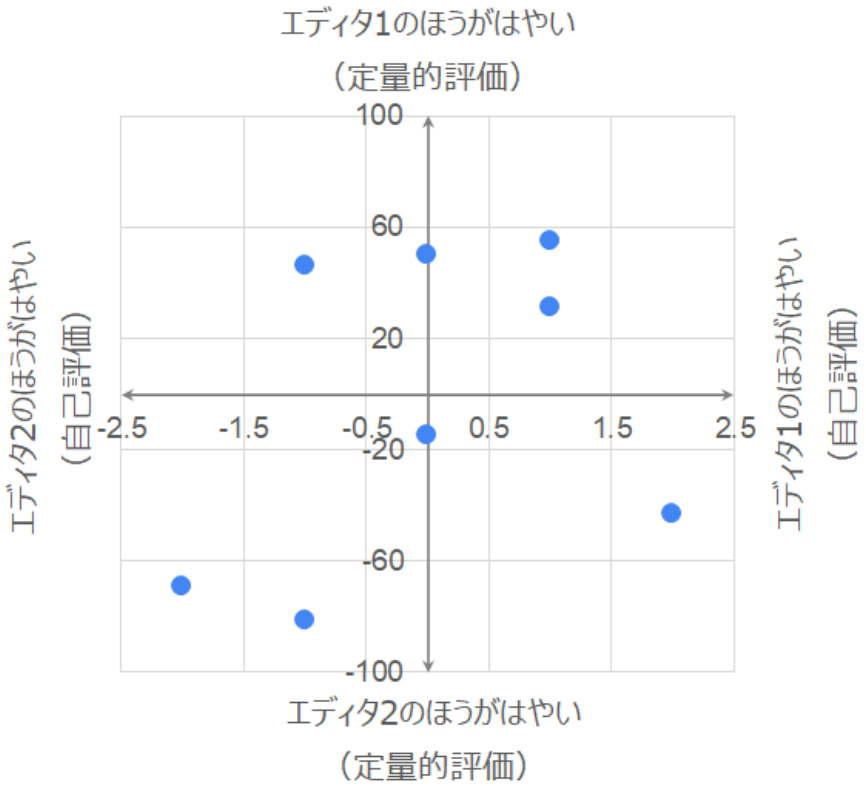


図 3: 自己評価と定量的評価の比較グラフ

x 座標（自己評価）

グラフの x 座標は，エディタ 1 とエディタ 2 のどちらのほうがタスクをはやく終了できる（できた）かの自己評価を表している．x 座標の導出は，3.3.1 節で説明した表 1 のアンケートの質問項目 2,4,5 の結果を，-2 以上 2 以下の整数に対応させることで行う．具体的には，表 2 のようにアンケート結果と x 座標を対応させる．x 座標が大きいほどエディタ 1 のほうが，小さいほどエディタ 2 のほうがタスクをはやく終了できる（できた）と自己評価していることを表す.

表 2: アンケート結果と x 座標の対応

アンケート結果	対応する x 座標
エディタ 1 のほうが非常にはやい	+2
エディタ 1 のほうが少しはやい	+1
変わらない	0
エディタ 2 のほうが少しはやい	-1
エディタ 2 のほうが非常にはやい	-2

y 座標（定量的評価）

グラフの y 座標は，エディタ 1 とエディタ 2 のどちらのほうがタスクをはやく終了できたかの定量的評価を表している．y 座標の導出には，進捗の時系列変化のグラフから読み取れる，各タスクに要した時間を使用する．具体的には以下の手順で導出する．

1. 各タスクに要した時間の集合 $task = \{task_1, task_2, \cdots, task_n\}$ に対して正規化を行う．正規化とは各データを最小値 0 から最大値 1 にスケーリングする方法であり，正規化後の集合 $task' = \{task'_1, task'_2, \cdots, task'_n\}$ は次式で定義される．

$$task'_i = \frac{task_i - min(task)}{max(task) - min(task)}$$

$min(task)$: $task$ の最小値

$max(task)$: $task$ の最大値

2. 正規化した $task'$ を用いて，エディタ 1 を使用したタスクの平均と，エディタ 2 を使用したタスクの平均を計算する．

$E1'_{avg}$: $task'$のうちエディタ 1 を使用した $task'_i$ の平均

$E2'_{avg}$: $task'$のうちエディタ 2 を使用した $task'_i$ の平均

3. $E1'_{avg}$と $E2'_{avg}$ を用いて，y 座標を計算する．

$$\text{y 座標} = -(E1'_{avg} - E2'_{avg}) \times 100$$

以上の手順により導出される y 座標の値域は -100 以上 100 以下である．y 座標が大きいほどエディタ 1 のほうが，小さいほどエディタ 2 のほうがタスクをはやく終了できたと定量的評価されていることを表す．

自己評価と定量的評価が一致しているかを判定する基準

自己評価と定量的評価が一致しているかの判定には 2 つの基準を使用する．

1 つ目は，グラフの相関係数である．相関係数 r は，2 個のデータ列を $(x_1, x_2, \cdots, x_n)$ と $(y_1, y_2, \cdots, y_n)$ としたとき，次式で定義される．

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{(x_i - \bar{x})^2}\sqrt{(y_i - \bar{y})^2}}$$

グラフの相関係数が 1 に近いほど自己評価と定量的評価に正の相関がある，つまり自己評価と定量的評価が一致していることがわかる．

2 つ目は，図 4 のようにグラフ上の座標を 3 つの領域に分割し，各点がどの領域に存在するかを調べる．そして，各領域内の点の数を比較することで自己評価と定量的評価が一致しているかを判定する．図 4 では，青色の領域が自己評価と定量的評価が一致していると判定される領域，緑が少しずれていると判定される領域，橙色がずれていると判定される領域である．

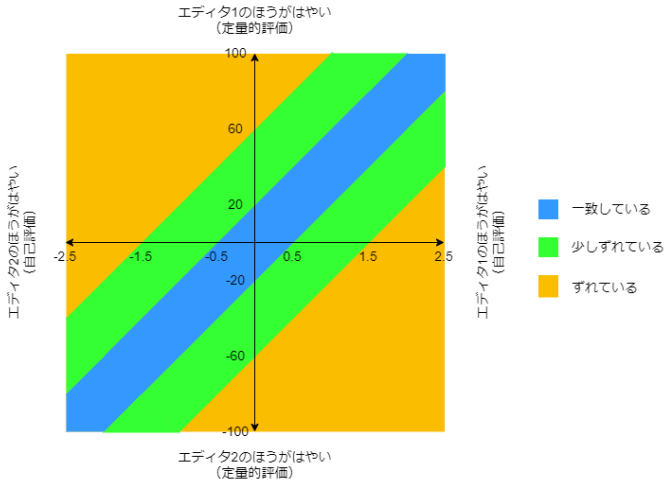


図 4: グラフにおける各領域の意味

4　実験方法

本研究では，3 章で述べた調査手法の適用例として，Visual Studio Code[5] と Eclipse[2] を対象にして実験を行う．以下では，本実験の内容を説明する．

4.1　実験の全体像

本実験では，Visual Studio Code と Eclipse それぞれの作業効率に関する開発者の自己評価と定量的評価を収集する．そして，収集した自己評価と定量的評価の比較を行いギャップがあるかを調査する．その他の実験条件としては，開発言語には Java を使用する．被験者として Java の経験年数 1 年以上，能力レベルは 1 以上 2 以下，大阪大学基礎工学部情報科学科の学生 3 名と大阪大学情報科学研究科の学生 5 名の計 8 名に解答してもらった．Java の能力レベルの分類を表 3 に示す．

実験の流れを図 5 に示す．実験は大学内で行い，各被験者にノート PC1 台とマウス 1 つを割り当てる．割り当てる PC とマウスの機種はどの被験者も同じである．実験を開始する前に，被験者に実験の概要と注意事項に関して説明する．被験者は，すべてのタスク開始前のアンケート，タスク 1 とタスク 1 内の各設問後のアンケート，タスク 1 終了後のアンケート，タスク 2，タスク 2 終了後のアンケート，すべてのタスク終了後のアンケートの順で作業を行う．

表 3: Java の能力レベル表 （[11]p.25 より引用）

レベル	基準
レベル 1（★☆☆）	書いたことがある．教えられた通りに開発をしたことがある．
レベル 2（★★☆）	独力で書ける．独力で調べながら開発をしたことがある．
レベル 3（★★★）	使いこなせる．他人に教えながら開発をしたことがある．

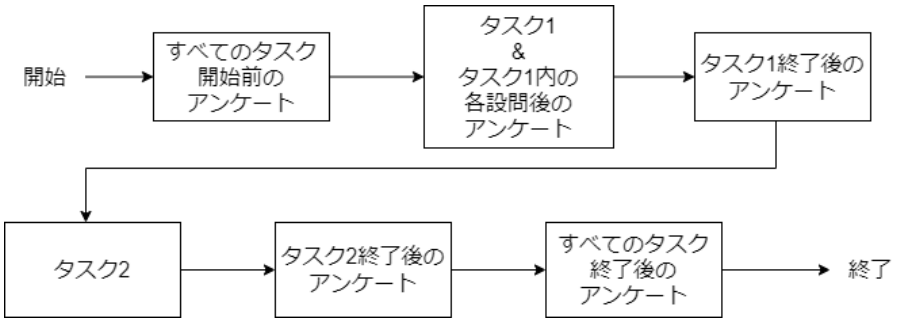


図 5: 実験の流れ

4.2 Eclipse からのデータ収集の実現

2.2 節の細粒度編集履歴収集プラットフォームの既存実装では 1 つのエディタ（Visual Studio Code）にしか展開されておらず，そのエディタからしかデータ収集できない．そこで，既存の言語サーバと LSP を通じてやり取りを行う Eclipse 用のプラグインを実装することで，Eclipse からの細粒度編集履歴の収集を実現した．

4.3 タスクの詳細

本実験におけるタスク 1, タスク 2 で実際に使用した問題について説明する．

タスク 1（プログラミング問題の解答）

本実験におけるタスク 1 では，Java の問題 4 問を用意し，Visual Studio Code と Eclipse それぞれれで 2 問ずつ解くように指定した．難易度が同程度である問題を選択するため，競技プログラミングサイト AtCoder[1] で公開されている問題のうち，筆者が初めて解いたときにかかった時間の範囲が 1 分以内であり，かつ解答方針が単純なものをタスク 1 に用いる問題とした．使用した問題を表 4 に示す．

タスク 2（ソースコードの写経）

本実験におけるタスク 2 では，ソースコードを 2 つ用意し，各ソースコードを Visual Studio Code と Eclipse それぞれで 1 回ずつ写経してもらう．つまり,（ソースコード 2 つ）×（エディタ 2 つ）= 計 4 回写経してもらう．使用したソースコードを図 6, 図 7 に示す．

4.4 進捗可視化手法

定量的評価には，コーディング過程の開発進捗率の時系列変化を表すグラフが必要になる．本節では，定量的評価のために開発した進捗可視化ツールの説明をする．

進捗率の算出方法

2.3 節で紹介したコーディング過程の可視化に関する先行研究 [13],[9] ではコード変化量の時系列変化を可視化しており，先行研究 [11] ではレーベンシュタイン距離 [4] に基づく Achievement Ratio の時系列変化を可視化している．時刻 t における Achievement Ratio は,

表 4: タスク 1 に使用した問題一覧

問題番号	タイトル	内容
abc208	Factorial Yen Coin	1! 円硬貨，2! 円硬貨，…，10! 円硬貨のみが流通しているとする．ここで，$N! = 1 \times 2 \times \cdots \times N$ ．全ての種類の硬貨を 100 枚ずつ持っており，P 円の商品をお釣りが出ないようにちょうどの金額を支払って買おうとしている．問題の制約下で条件を満たす支払い方は必ず存在することが証明できる．最小で何枚の硬貨を使えば支払うことができるか求めよ．(https://atcoder.jp/contests/abc208/tasks/abc208_b)
abc217	AtCoder Quiz	現在，ABC, ARC, AGC, AHC の 4 つのコンテストのみが定期的に開催されているとする．現在定期的に開催されているコンテストは S_1, S_2, S_3 とあと 1 つは何か求めよ．(https://atcoder.jp/contests/abc217/tasks/abc217_b)
abc220	Base K	整数 A, B が K 進法表記で与えられる．$A \times B$ を 10 進法表記で出力せよ．(https://atcoder.jp/contests/abc220/tasks/abc220_b)
abc224	Mongeness	縦 H 行，横 W 列のマス目があり，各マスには 1 つの整数が書かれている．上から i 行目，左から j 列目のマスに書かれている整数は $A_{i,j}$ である．マス目が次の条件を満たすかどうかを判定せよ．$1 \leq i_1 < i_2 \leq H$ および $1 \leq j_1 < j_2 \leq W$ を満たすすべての整数の組 (i_1, i_2, j_1, j_2) について，$A_{i_1,j_1} + A_{i_2,j_2} \leq A_{i_2,j_1} + A_{i_1,j_2}$ が成り立つ．(https://atcoder.jp/contests/abc224/tasks/abc224_b)

```
public class linearSearch {
    public static int execute(int[] data, int target) {
        int notFound = -1;
        for (int i = 0; i < data.length; i++) {
            if (data[i] == target) {
                return i;
            }
        }
        return notFound;
    }

    public static void main(String[] args) {
        int[] data = { 1, 2, 3, 4, 5 };
        int result;
        result = linearSearch.execute(data, 3);
        if (result != -1) {
            System.out.println("Found: index key = " + result);
        } else {
            System.out.println("Not found.");
        }
    }
}
```

図 6: タスク 2 の写経用ソースコード 1

```
public class bubbleSort {
    public static void sort(int[] array) {
        int temp;
        for (int i = 0; i < array.length; i++) {
            for (int j = 0; j < array.length - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] test = { 10, 75, 24, 32, 98 };
        bubbleSort.sort(test);
        for (int i = 0; i < test.length; i++) {
            System.out.println((i + 1) + ":" + test[i]);
        }
    }
}
```

図 7: タスク 2 の写経用ソースコード 2

レーベンシュタイン距離を用いて次式で定義されている．

$$A_t = 1 - \frac{L_t}{L_0}$$
$$L_t = \sum_{i=1}^{n} D(file_{i,t}, file_{i,last})$$
$$n = \text{ファイル数}$$
$$D = \text{レーベンシュタイン距離}$$

A_t は時刻 t における Achievement Ratio であり，L_t は時刻 t における各ファイルの内容とそのファイルの最終状態の内容の距離の和であり，D は距離関数である．

本研究では単純なコード変化量ではなく編集にかかるコストをより反映している Achievement Ratio の可視化を行う．

進捗率の可視化

本研究で開発した進捗可視化ツールは，複数のコーディング過程の開発進捗率の時系列変化をコーディングに使用したエディタの情報付きで可視化する．具体的には，横軸がコーディングの開始時刻を基準とした経過時間（second），縦軸が Achievement Ratio（%）のグラフを表示する．このツールによって表示されるグラフの例を図 8 に示す．

参考にした先行研究 [11] との違いは主に 2 つある．1 つ目は，先行研究では単一のコーディング過程の可視化であるが，本ツールでは複数のコーディング過程の可視化が可能である．2 つ目は，先行研究では表示されていなかったコーディングに使用したエディタの情報が表示される点である．具体的には，グラフのデータラベルにエディタ名が追加され，同じエディタのグラフ同士は近い色で表示される．

加えて，このツールでは同じエディタのグラフ同士間を塗りつぶす機能がある．この機能を図 8 のグラフに使用すると図 9 にように表示される．この機能を利用することで，場合によってはエディタ間の比較が容易になる．

このように，このツールを使用することで，複数のコーディング過程の開発進捗率の時系列変化をコーディングに使用したエディタの情報付きで確認することができ，開発環境ごとのコーディング過程の比較を容易に行うことができる．

実装

2.2 節の細粒度編集履歴収集プラットフォームの API サーバーでは，データを取り出すための API が実装されていない．そのため，既存の API サーバーとは別の進捗可視化ツール用の API サーバを実装した．進捗可視化ツールは，この API サーバに HTTP 通信を通じ

て可視化を行う履歴の情報を送信する．リクエストを受けとった進捗可視化ツール用の API サーバは，DB サーバとオンラインストレージからデータを取得し，そのデータをもとに進捗の時系列変化を計算した後，その結果をレスポンスとしてツールに送信する．ツールはそのレスポンスをもとにグラフを表示する．

Achievement Ratio の計算には各時刻でのファイルの内容が必要になる．2.2 節の細粒度編集履歴の記録単位で述べたように，細粒度編集履歴には編集開始前のファイルの内容が記載されているため，それを使用することで Achievement Ratio の計算を実現している．

また，コーディングに使用したエディタの情報を表示するためにエディタの情報が必要になる．2.2 節の細粒度編集履歴の記録単位で述べたように，細粒度編集履歴には編集に使用したエディタの名前が記載されているため，それを使用することでエディタの情報の表示を実現している．

本研究で開発した進捗可視化ツールは，言語サーバからデータベースへの送信単位でのファイルの内容を利用しているため，グラフ上の点もその単位でプロットされる．そのため，編集操作ごとに Achievement Ratio を計算しているわけではない．

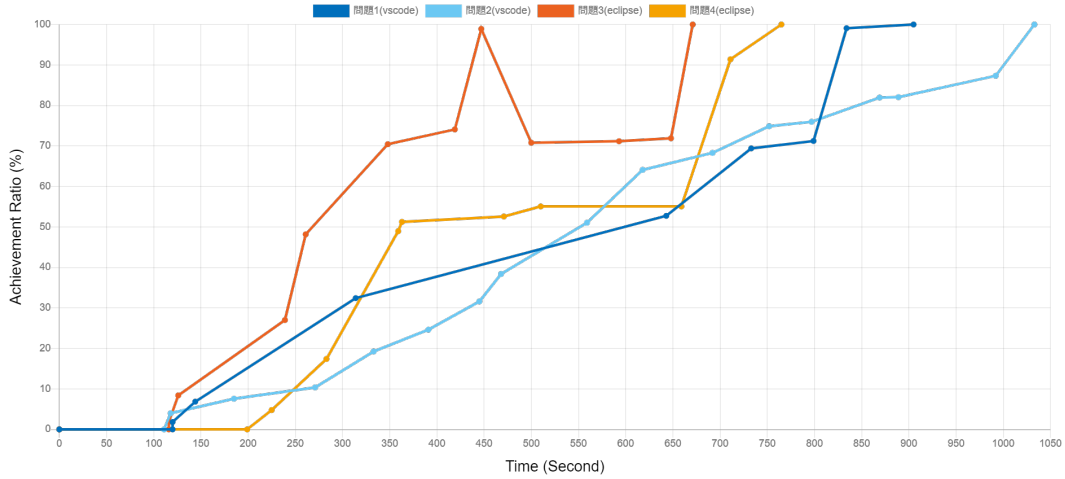


図 8: ツールで表示されるグラフの例

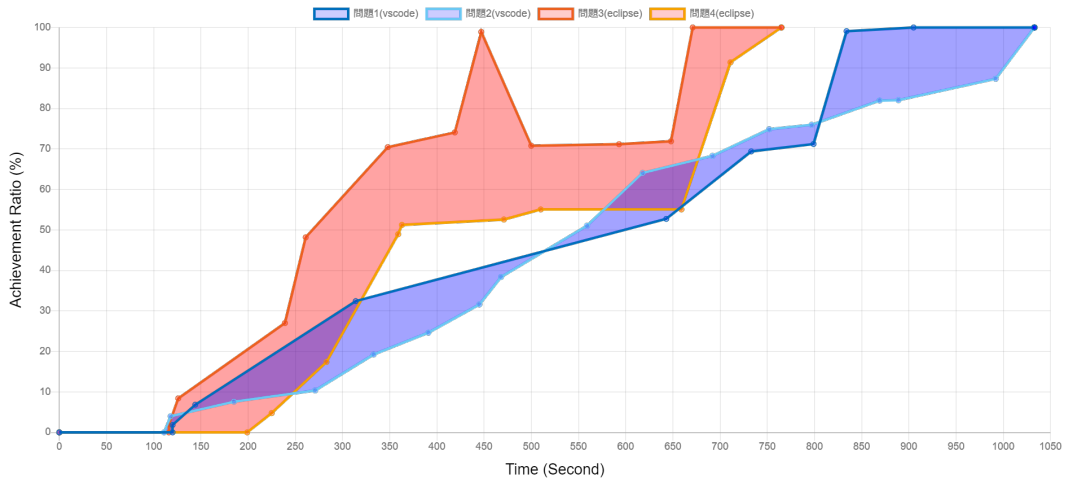


図 9: 図 8 に同じエディタのグラフ同士間を塗りつぶす機能を適用した場合

5　実験結果

本章では 4 章の実験結果を示す.

5.1　被験者の属性

本実験における被験者の属性を説明する．Java の能力レベル（表 3 を参照）の分布を図 10 に，Java の経験年数の分布を図 11 に，Visual Studio Code と Eclipse それぞれでの Java コーディングの得意不得意の分布を図 12 に示す．図 12 では各マス内の点の数が人数に対応する．例えば,「Visual Studio Code がやや苦手かつ Eclipse が普通」の人数が 2 人であることが示されている．点の色は，赤色が Visual Studio Code より Eclipse のほうが得意と感じている人を意味し，青色が Eclipse より Visual Studio Code のほうが得意と感じている人を意味する.

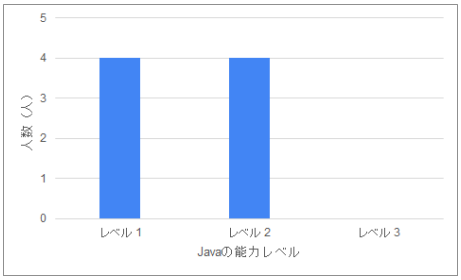


図 10: Java の能力レベルの分布

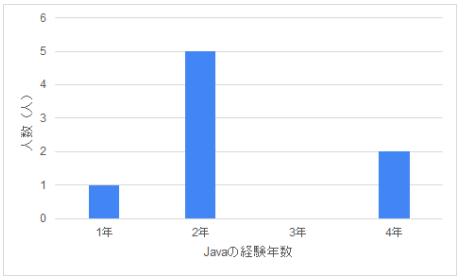


図 11: Java の経験年数の分布

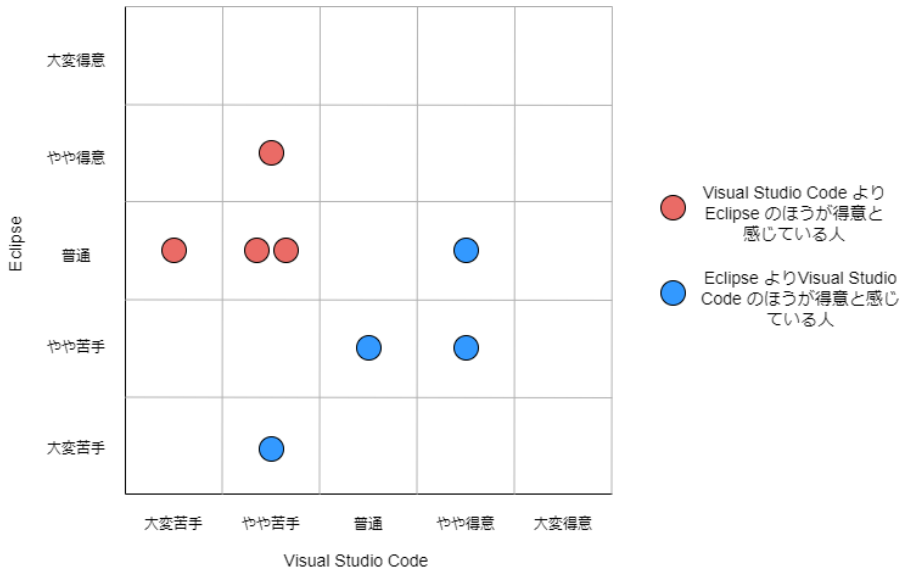


図 12: Visual Studio Code と Eclipse それぞれでの Java コーディングの得意不得意の分布

本実験の被験者は，Java の経験年数が 1 年以上であり，Java の能力レベルは 1 以上 2 以下である．Visual Studio Code より Eclipse のほうが得意と感じている人と，Eclipse より Visual Studio Code のほうが得意と感じている人はそれぞれ 4 人ずつである．

5.2　タスク 1 の結果

本節では，タスク 1 から得られたデータをもとに行った「開発前の自己評価と定量的評価の比較」と「開発後の自己評価と定量的評価の比較」の結果を示す．

開発前の自己評価と定量的評価の比較結果

タスク 1 で得られたデータをもとに，3.4 節で説明した方法で作成した開発前の自己評価と定量的評価の比較グラフを図 13 に示す．各点が各被験者の自己評価と定量的評価を示している．点の色は，3.4 節で説明した方法をもとに自己評価と定量的評価が一致しているかを判定した結果を表している．青色が一致していることを意味し，緑が少しずれていることを意味し，橙色がずれていることを意味する．また，グラフをもとに各色の点の数を集計した結果を図 14 に示す．

図 13 の比較グラフの相関係数 r は -0.35 である．また，自己評価と定量的評価に関して，一致している人が 1 人，少しずれている人が 4 人，ずれている人が 3 人である．

開発後の自己評価と定量的評価の比較結果

タスク 1 で得られたデータをもとに，3.4 節で説明した方法で作成した開発後の自己評価と定量的評価の比較グラフを図 15 に示す．各点が各被験者の自己評価と定量的評価を示している．点の色は，3.4 節で説明した方法をもとに自己評価と定量的評価が一致しているかを判定した結果を表している．青色が一致していることを意味し，緑が少しずれていることを意味し，橙色がずれていることを意味する．また，グラフをもとに各色の点の数を集計した結果を図 16 に示す．

図 15 の比較グラフの相関係数 r は -0.59 である．また，自己評価と定量的評価に関して，一致している人が 2 人，少しずれている人が 3 人，ずれている人が 3 人である．

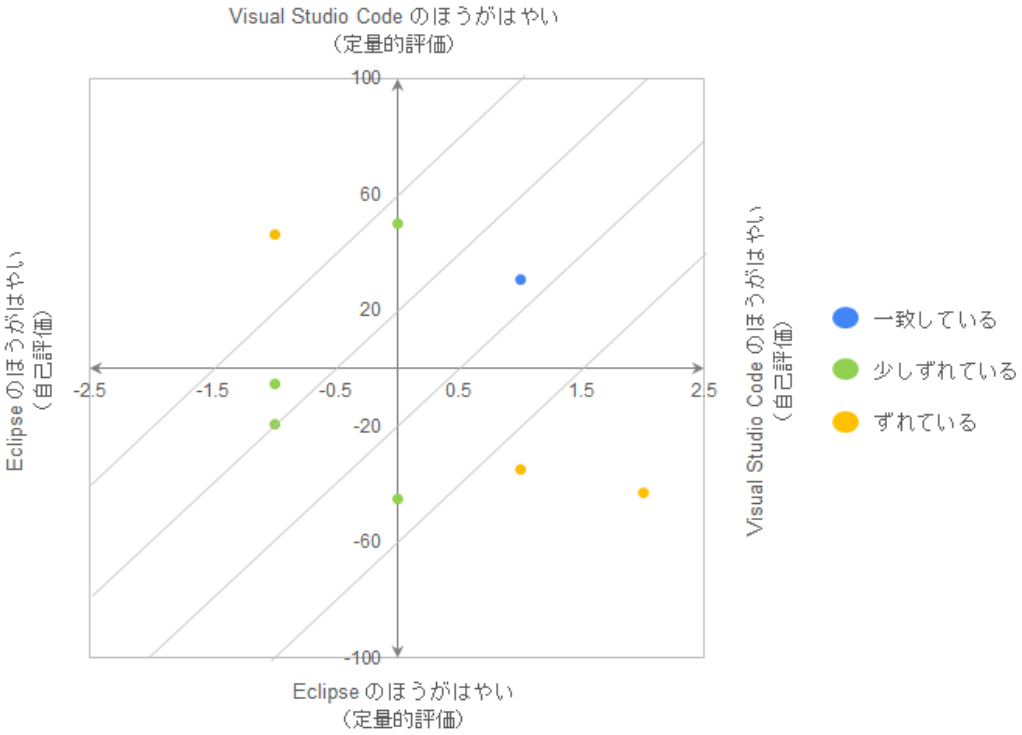


図 13: タスク 1 における開発前の自己評価と定量的評価の比較グラフ

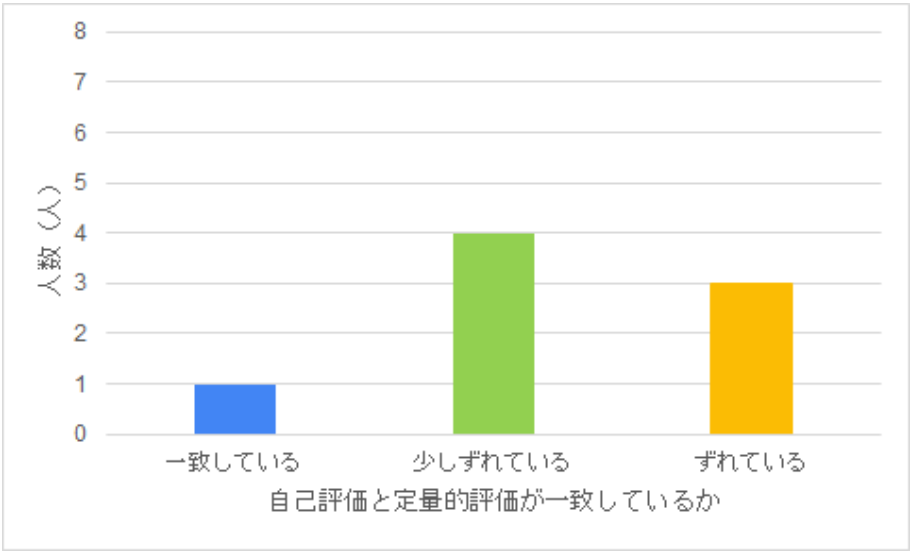


図 14: タスク 1 において開発前の自己評価と定量的評価が一致しているかの分布

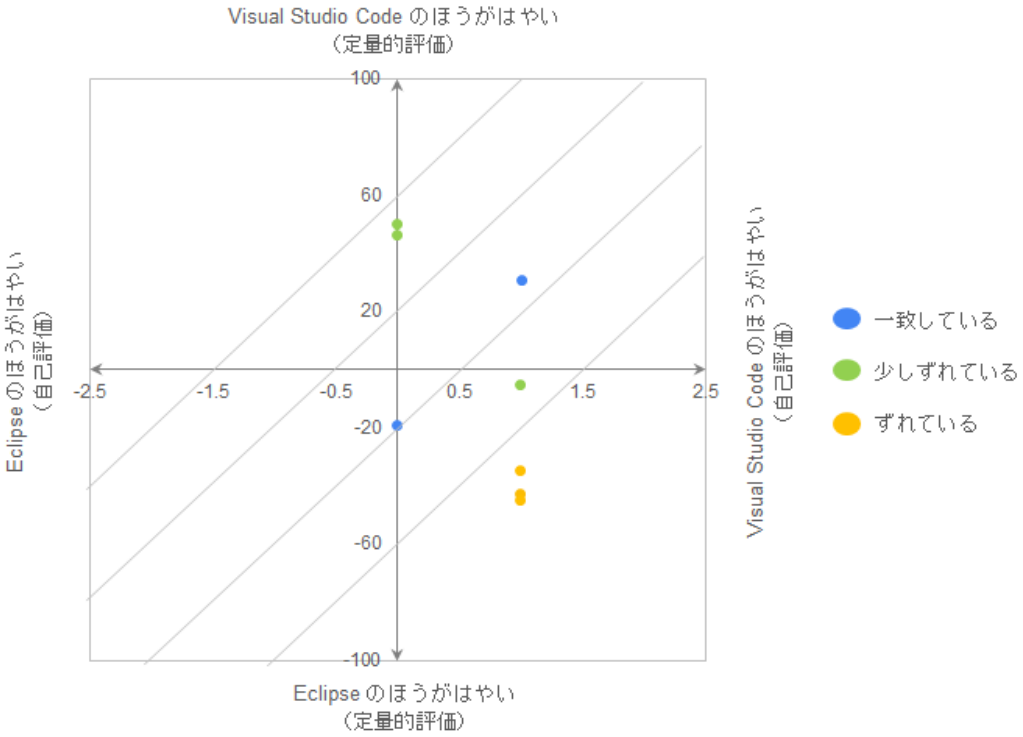


図 15: タスク 1 における開発後の自己評価と定量的評価の比較グラフ

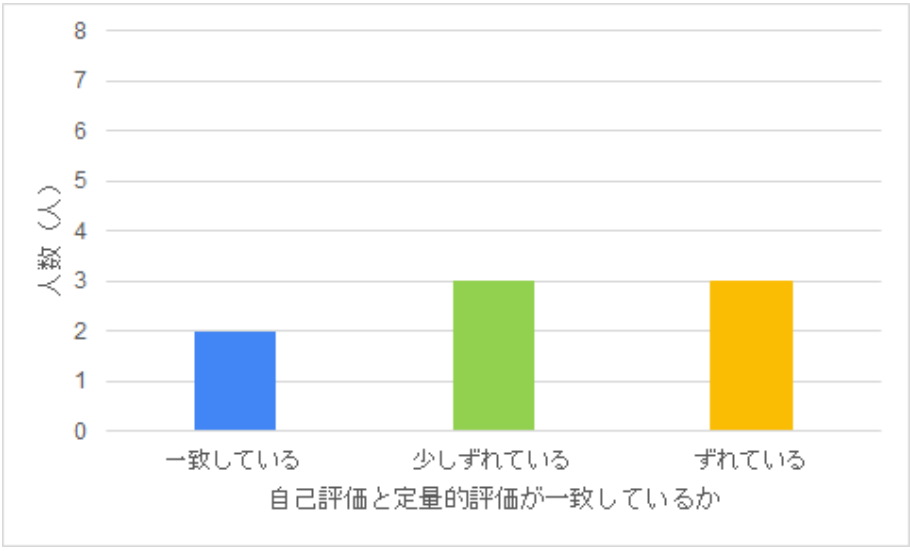


図 16: タスク 1 において開発後の自己評価と定量的評価が一致しているかの分布

5.3　タスク 2 の結果

本節では，タスク 2 から得られたデータをもとに行った「開発前の自己評価と定量的評価の比較」と「開発後の自己評価と定量的評価の比較」の結果を示す．

開発前の自己評価と定量的評価の比較結果

タスク 2 で得られたデータをもとに，3.4 節で説明した方法で作成した開発前の自己評価と定量的評価の比較グラフを図 17 に示す．各点が各被験者の自己評価と定量的評価を示している．点の色は，3.4 節で説明した方法をもとに自己評価と定量的評価が一致しているかを判定した結果を表している．青色が一致していることを意味し，緑が少しずれていることを意味し，橙色がずれていることを意味する．また，グラフをもとに各色の点の数を集計した結果を図 18 に示す．

図 17 の比較グラフの相関係数 r は 0.71 である．また，自己評価と定量的評価に関して，一致している人が 3 人，少しずれている人が 4 人，ずれている人が 1 人である．

開発後の自己評価と定量的評価の比較結果

タスク 2 で得られたデータをもとに，3.4 節で説明した方法で作成した開発後の自己評価と定量的評価の比較グラフを図 19 に示す．各点が各被験者の自己評価と定量的評価を示している．点の色は，3.4 節で説明した方法をもとに自己評価と定量的評価が一致しているかを判定した結果を表している．青色が一致していることを意味し，緑が少しずれていることを意味し，橙色がずれていることを意味する．また，グラフをもとに各色の点の数を集計した結果を図 20 に示す．

図 19 の比較グラフの相関係数 r は -0.27 である．また，自己評価と定量的評価に関して，一致している人が 3 人，少しずれている人が 3 人，ずれている人が 2 人である．

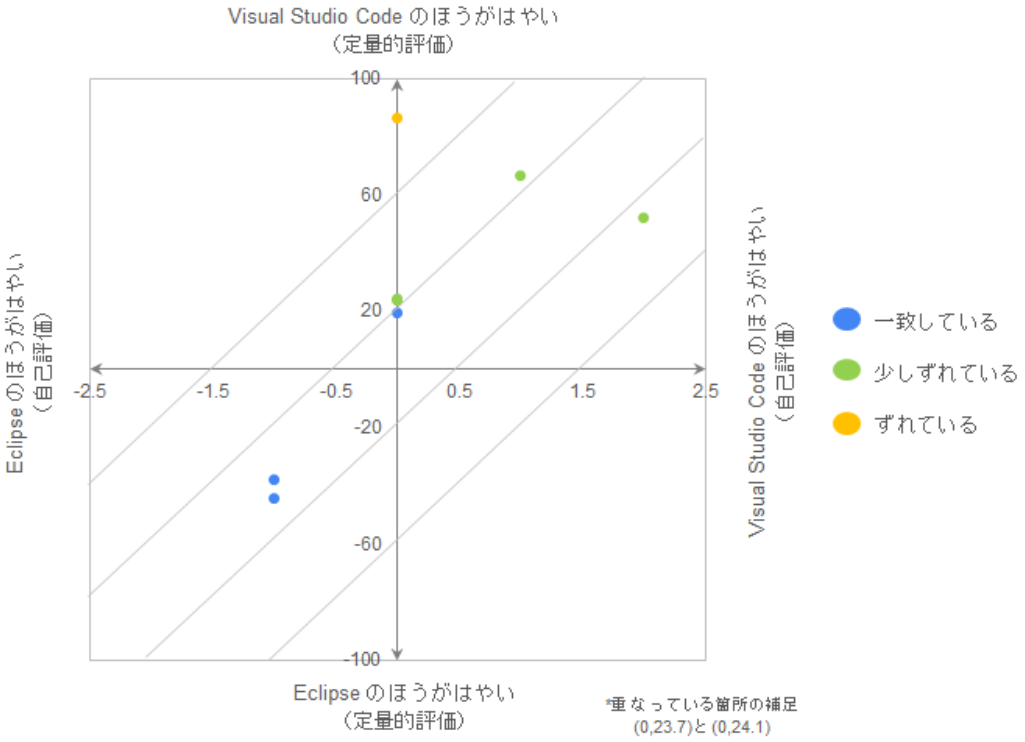


図 17: タスク 2 における開発前の自己評価と定量的評価の比較グラフ

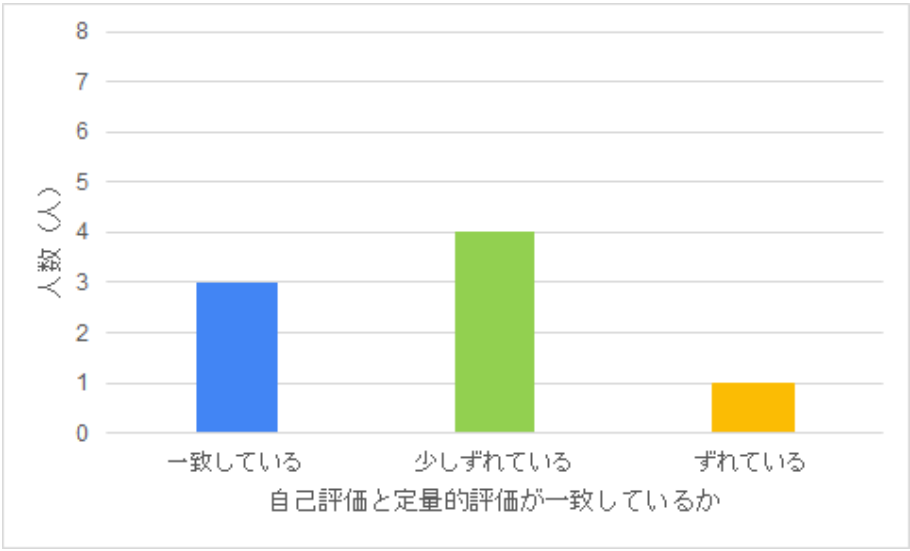


図 18: タスク 2 において開発前の自己評価と定量的評価が一致しているかの分布

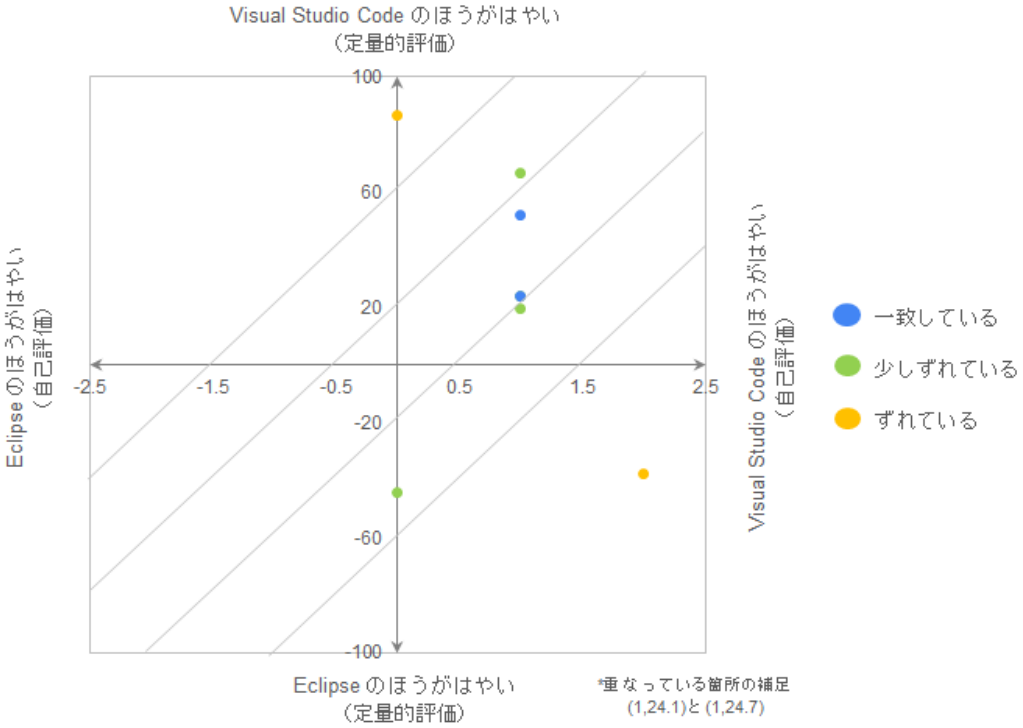


図 19: タスク 2 における開発後の自己評価と定量的評価の比較グラフ

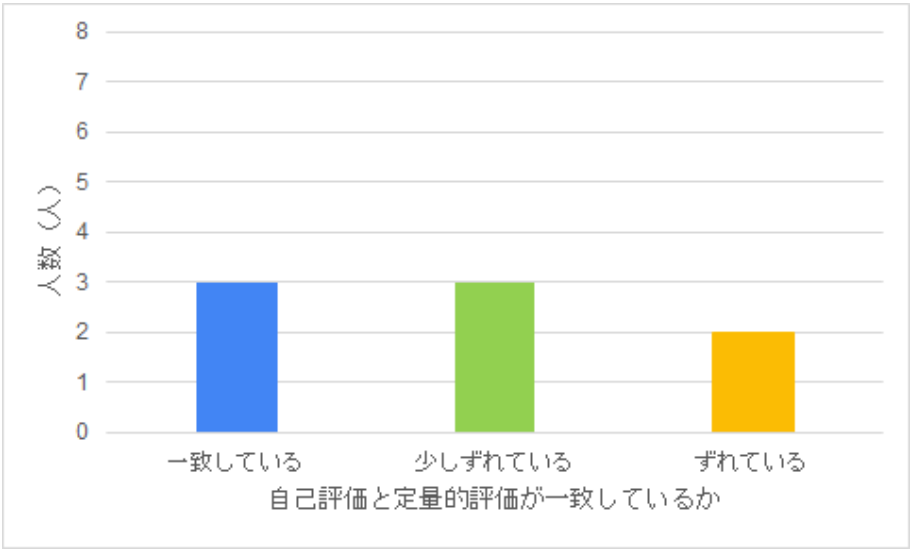


図 20: タスク 2 において開発後の自己評価と定量的評価が一致しているかの分布

6　考察

6.1　タスク 1 の結果

自己評価と定量的評価の比較結果

タスク 1 における開発前の自己評価と定量的評価の比較では，一致している人は 8 人中 1 人と少なく，ずれている人は 8 人中 3 人と少し多かった．相関係数に関しては -0.35 であり，正の相関はみられなかった．開発後の自己評価と定量的評価の比較では，一致している人は 8 人中 2 人と少なく，ずれている人は 8 人中 3 人と少し多かった．相関係数に関しては -0.59 であり，正の相関はみられなかった．以上のことから，タスク 1 においては開発前の自己評価と開発後の自己評価ともに定量的評価とギャップがあることがわかった．

タスク内容による妥当性

タスク 1 において，開発前の自己評価と開発後の自己評価ともに定量的評価とギャップがあると結論付けるには問題がある．なぜなら，タスク 1 は同程度の難易度の問題を解くというタスクであったが，問題ごとの難易度が人によってずれる可能性があるからである．

本実験での難易度が同じである基準は，筆者が初めて解いたときにかかった時間の範囲が 1 分以内であり，かつ解答方針が単純なものとした．実際に難易度が同じであれば，各問題の躓きに関する感想も概ね一致するはずである．しかし，タスク 1 内の各設問後に行った各問題の感想に関するアンケート結果は表 5 のようになった．表 5 では，+2 が非常に躓いた，+1 が少し躓いた，0 が普通，-1 が少しすらすら解けた，-2 が非常にすらすら解けたを表す．どの被験者についても問題ごとに感想が異なっていることがわかる．このため，タスクに要する時間のずれが，エディタだけでなく問題にも依存している可能性が高い．

表 5: 各問題の感想に関するアンケート結果

	abc208	abc217	abc220	abc224
被験者 A	+1	-1	-1	-1
被験者 B	-1	+1	+2	-1
被験者 C	+1	0	+2	+2
被験者 D	+1	0	-2	-1
被験者 E	0	-2	+1	-1
被験者 F	0	-1	-1	+1
被験者 G	0	+1	+2	-1
被験者 H	+1	0	0	+1

タスクに要する時間の問題への依存を減らすための方法として以下の 2 つが考えられるが，どちらも本実験で使用するには問題がある．

- 表 5 のアンケート結果を使用して補正を行う．例えば，アンケートで躓いたと答えている問題に対してはタスクに要した時間を小さくする操作をし，すらすら解けたと答えている問題に対しては大きくする操作をする．しかし，時間をどの程度補正するのかが難しく，被験者の問題に対する体感が実際に一致しているとも限らない．

- 外れ値の検定を行う．例えば，被験者 G の進捗のグラフは図 21 であるが，問題 abc220 だけ極端に時間がかかっており，これはエディタではなく問題に依るずれだと考えられる．この被験者 G のデータに対して，有意水準 5%で Smirnov-Grubbs 検定を行うことで問題 abc220 の問題が検出され，外れ値として扱うことができる．しかし，今回の実験では各被験者のデータ数が 4 つであるため，検定を行うにはデータ数が少なすぎる．また，各エディタのデータは 2 つであるため，1 つ取り除いてしまうとそのエディタのデータが 1 つになってしまい，その影響が非常に大きい．

以上のことから，タスク 1 においてはタスクに要する時間のずれがエディタだけでなく問題にも依存している可能性が高く，その依存を減らすための補正ができていないため，実験の結果をもとに開発前の自己評価と開発後の自己評価ともに定量的評価とギャップがあると結論付けるには問題がある．

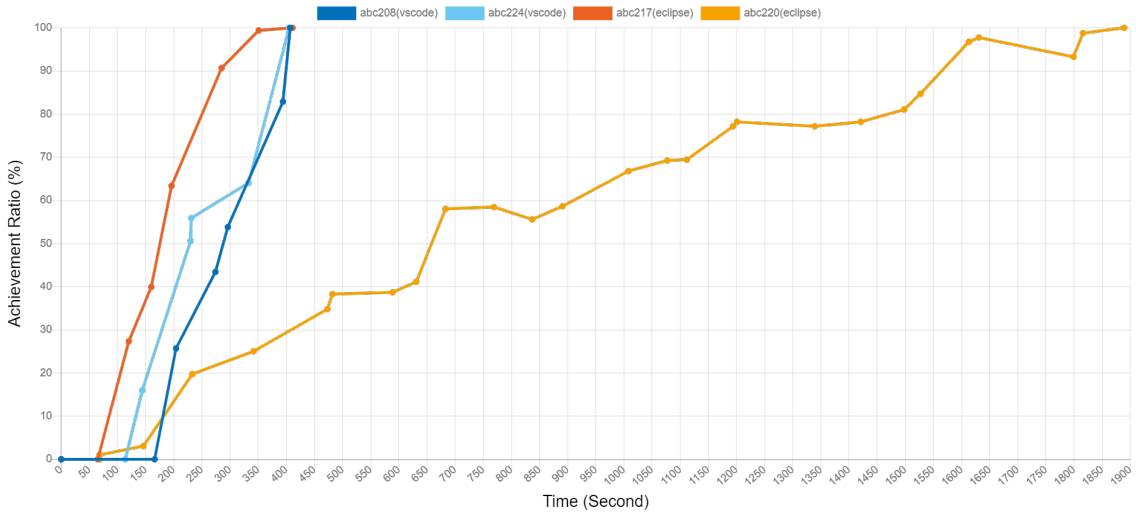


図 21: 被験者 G のタスク 1 における進捗グラフ

6.2　タスク 2 の結果

開発前の自己評価と定量的評価の比較結果

タスク 2 における開発前の自己評価と定量的評価の比較では，一致している人は 8 人中 3 人と少し多く，ずれている人は 8 人中 1 人と少なかった．また，少しずれていると判定された 4 人のデータ（図 17 参照）に注目すると，どのデータに関しても「一致している」と判定される領域に非常に近いことがわかる．相関係数に関しては 0.71 であり，正の相関がみられた．以上のことから，タスク 2 においては開発前の自己評価と定量的評価が概ね一致していることがわかった．

自己評価と定量的評価に唯一大きくギャップがあった被験者 C に注目する．被験者 C はタスク前のアンケートで，タスクに要する時間はエディタ間で変わらないと答えていた．しかし，タスクを行った結果は Visual Studio Code のほうがはやかった．このように自己評価と定量的評価にギャップがあった理由は，被験者 C が本実験でタスクを行うまで Visual Studio Code を使用したことがなかったためと考えられる．つまり，タスク前のアンケートで Visual Studio Code に関して評価基準がなく，適切に評価できなかったからである．

開発後の自己評価と定量的評価の比較結果

タスク 2 における開発後の自己評価と定量的評価の比較では，一致している人は 8 人中 3 人と少し多く，ずれている人は 8 人中 2 人と少し少なかった．相関係数に関しては -0.27 であり，正の相関はみられなかった．以上のことから，タスク 2 においては開発後の自己評価と定量的評価にギャップがあることがわかった．

タスク内容による妥当性

タスク 2 ではソースコードの写経を行った．エディタごとに写経するソースコードが同じであるため，エディタごとのタスク内容は全く同じである．そのため，タスク 1 とは違いタスクごとの難易度によるずれが生じない．したがってエディタによるずれだけを見ることができる．

しかし，各ソースコードを Visual Studio Code と Eclipse で書き写すため，同じソースコードを 2 回書き写すことになる．そのため，後に書き写しを行ったエディタのほうがタスクに要する時間が短くなる可能性がある．そこで，タスク 2 において，タスクに要する時間が順番に依存しているかを調べる．（被験者 8 名）×（ソースコード 2 つ）の計 16 回において，先に書き写しを行ったほうがタスクに要する時間が短かったのが 9 回であり，後に行ったほうがタスクに要する時間が短かったのが 7 回である．したがって,「後に書き写しを行っ

たエディタのほうがタスクに要する時間が短くなる可能性がある」という問題は考慮しなくてよいことがわかる．

以上のことから，タスク 2 における自己評価と定量的評価の結果は妥当であるといえる．

6.3 定量的評価の妥当性

本節では，3.4 節で述べた比較グラフの y 座標，つまり定量的評価の計算方法の妥当性に関して考察する．実験で得られたデータに対して前処理を行わずに y 座標を計算した場合，被験者によってタスクに要する平均時間が違うにもかかわらず同じ尺度で扱ってしまい，適切に評価することができない．そこで，本研究で提案した定量的評価の計算方法では，各被験者のデータを最小値 0 から最大値 1 にスケーリングする正規化を行った．これにより，各被験者のデータの重みをそろえることができ，各被験者の基本的な能力，プログラミングの能力やタイピングスピードなどに違いがあったとしても，各被験者のデータを同じ重みで扱うことができる．また，正規化によって最小値と最大値をそろえることができるため，3.4 節で説明した，領域での判定に使用する基準を明確に決めることができる．最大値や最小値が決まっていない場合では，その基準を決めることが難しい．

ただし，正規化は外れ値に敏感であるという問題がある．実際，被験者 G（進捗グラフは図 21 参照）のように特定のタスクだけ時間がかかった場合，結果はそのデータに大きく影響される．対処方法としては，Smirnov-Grubbs 検定などの外れ値検定を行うことが考えられるが，本実験ではデータ数が少なすぎるため適していない．

もう一つの問題は，タスクに要した時間しか考慮しておらず，作業過程での進捗率変化を考慮していないことである．具体例として，被験者 B のタスク 2 のソースコード 2 における進捗グラフを図 22 に示す．青色が Visual Studio Code，赤色が Eclipse である．途中までは Eclipse のほうがはやいが，進捗率が 90%を超えてから停滞したために，最終的に Visual Studio Code のほうが作業終了がはやくなっている．Eclipse が 90%を超えてから停滞した理由としては次のことが考えられる．diff コマンドによるテストを行うまでははやくできたが，写経対象のソースコードとほんの少しだけ違う箇所があり，その場所を特定するのに時間がかかった．つまり，エディタの作業効率とは関係のない場所で時間を要している．したがって，コーディング過程を考慮すると，Eclipse のほうが作業効率がよかったことがわかる．しかし，本研究で提案した定量的評価の計算方法ではこれらの情報は考慮されていないため，適宜進捗グラフを用いた定量的評価も行う必要がある．

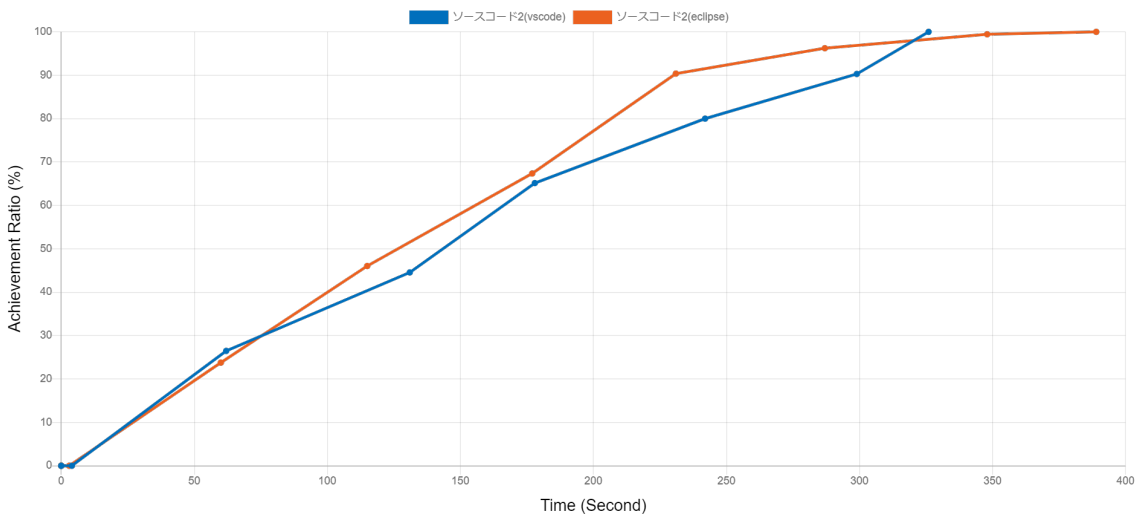


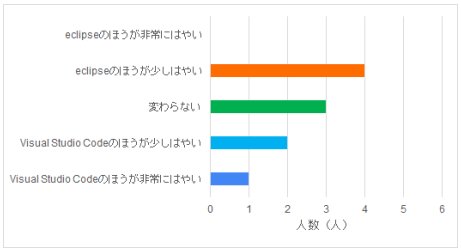
図 22: 被験者 B のタスク 2 のソースコード 2 における進捗グラフ

6.4　開発後の自己評価に関して

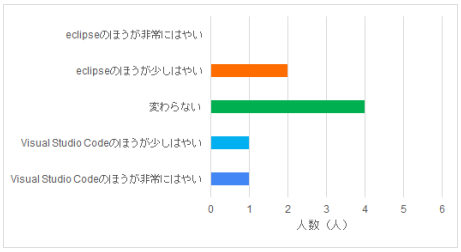
本節では，開発後の自己評価における特徴的な結果について説明する．本実験では，タスク開始前とタスク終了後に「どちらのエディタのほうがはやくコーディングできる（できた）と思うか」というアンケートを取った．タスク開始前の結果を図 23a, 図 23b に，タスク終了後の結果を図 23c, 図 23d に示す．タスク開始前のアンケートでは，タスク 1, タスク 2 のどちらとも結果がばらけていた．しかし，タスク終了後のアンケートでは，Eclipse のほうがはやいと感じた人が 1 人もおらず，半数以上の人が Visual Studio Code のほうがはやいと感じていた．

この結果が得られたのは，コード補完機能の影響である．タスク終了後に行ったフリーコメントの記述において，8 人中 6 人の被験者が Visual Studio Code の補完機能に関して肯定的な記述をしていた．

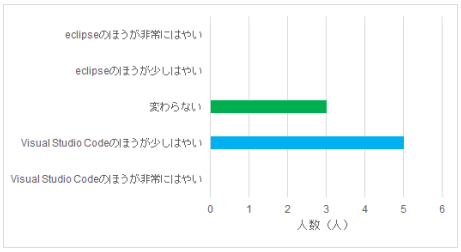
そして，Visual Studio Code の補完機能に関して肯定的な記述が多かった理由は，本実験の被験者が Eclipse でコード補完の詳細設定ができることを知らなかったからである．Eclipse ではコード補完機能の詳細設定により，補完が始まるまでの時間や補完候補を表示するトリガーとなる文字の設定などが可能である．実験後に「Eclipse においてコード補完機能の詳細設定ができることを知っているか」と追加でアンケートを行ったところ，8 人中 7 人が知らなかった．この結果を踏まえると，Eclipse のコード補完機能の詳細設定をした場合での実験も行う価値があると考えられる．



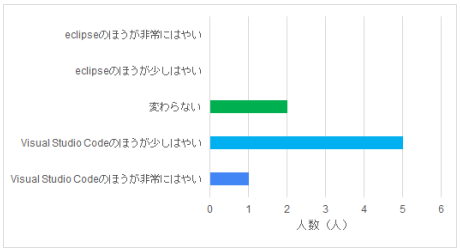
(a) タスク開始前・タスク 1 に関して



(b) タスク開始前・タスク 2 に関して



(c) タスク終了後・タスク 1 に関して



(d) タスク終了後・タスク 2 に関して

図 23: どちらのエディタのほうがはやくコーディングできる（できた）と思うか

6.5　問題の難易度の感じ方と進捗の比較

6.1 節のタスク内容の妥当性において，タスク 1 では問題ごとに難易度の感じ方が異なっていることを示した（表 5 参照）．本節ではこの結果を踏まえて，各被験者について，同じエディタ間で問題の難易度の感じ方と実際の進捗が一致しているかを調査する．例えば，アンケートで，問題 A は「少し躓いた」, 問題 B は「すらすら解けた」と回答したが，問題 A に要した時間より問題 B に要した時間のほうが大きい場合は一致していないことがわかる．（被験者 8 名）×（エディタ 2 つ）の計 16 のデータに対して，同じエディタ間で問題の難易度の感じ方と実際の進捗が一致しているか調査したところ，4 つのデータでずれがあった．ずれがあったデータついて，進捗グラフを用いて詳しく見る．

- 図 24 のデータでは，アンケートで abc217,abc220 ともに「少しすらすら解けた」と答えているが，タスクに要した時間には差がある．進捗グラフを見ると，abc217 のほうが abc220 と比較して停滞している時間も短く，スムーズに解けていることが確認できる．したがって，問題の難易度の感じ方と進捗がずれていることがわかる．

- 図 25 のデータでは，アンケートで abc208 は「普通」，abc217 は「少しすらすら解けた」と答えているが，タスクに要した時間は abc208 の方が短い．進捗グラフを見ると，abc208 では一度進捗率が 100%に近づいた後に 70%近くまで下がっている．このことからプログラムの間違いを一度修正している可能性が高いことがわかる．この影響によって，実際にかかった時間は abc208 のほうが短いにもかかわらず，abc217 よりも難しく感じたと考えられる．

- 図 26 のデータでは，アンケートで abc220 は「非常にすらすら解けた」，abc224 は「少しすらすら解けた」と答えているが，タスクに要した時間は abc224 の方が短い．進捗グラフを見ると，abc224 のほうが abc220 と比較して全体的にスムーズに開発が進んでいることが確認できる．したがって，問題の難易度の感じ方と進捗がずれていることがわかる．

- 図 27 のデータでは，アンケートで abc208 は「普通」，abc224 は「少しすらすら解けた」と答えているが，タスクに要した時間はほとんど変わらない．進捗グラフを見ると，abc208 と abc224 にそれほど大きな違いはみられない．ただし，abc224 のほう進捗が変化し始めるのがはやい．つまり，abc224 のほうが設計方針の考案が簡単であったことがわかる．この影響によって，実際にかかった時間がほとんど同じにもかかわらず，abc224 のほうが簡単に感じたと考えられる．

以上のことから，問題の難易度の感じ方と実際の進捗がずれているデータについて進捗グ

ラフを用いて詳しく調べることで，原因があってずれが生じているのか，それとも単に体感がずれているのかを区別できる可能性が高いことがわかった．

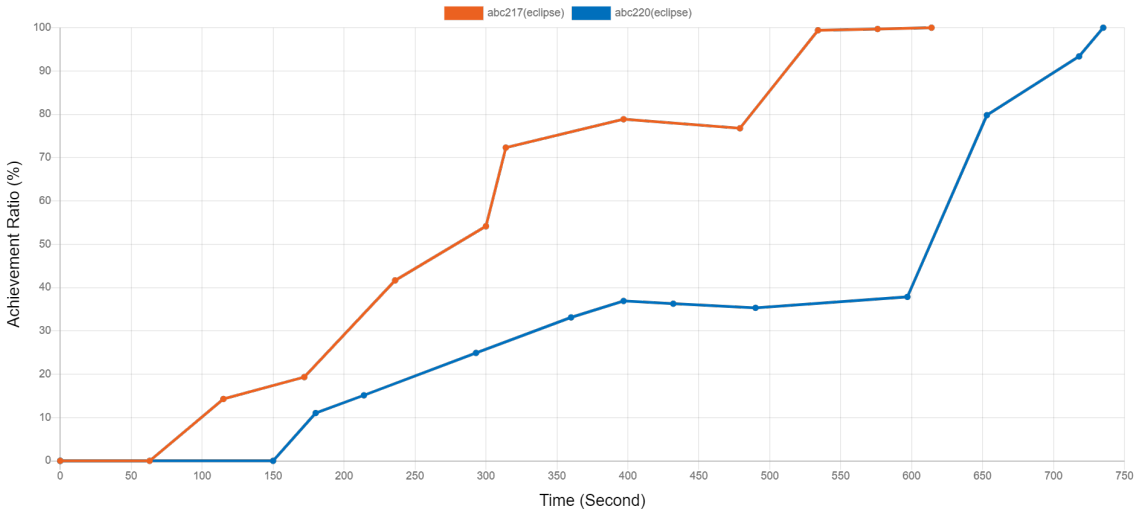


図 24: 問題の難易度の感じ方と進捗がずれているデータ（赤:abc217，青:abc220）

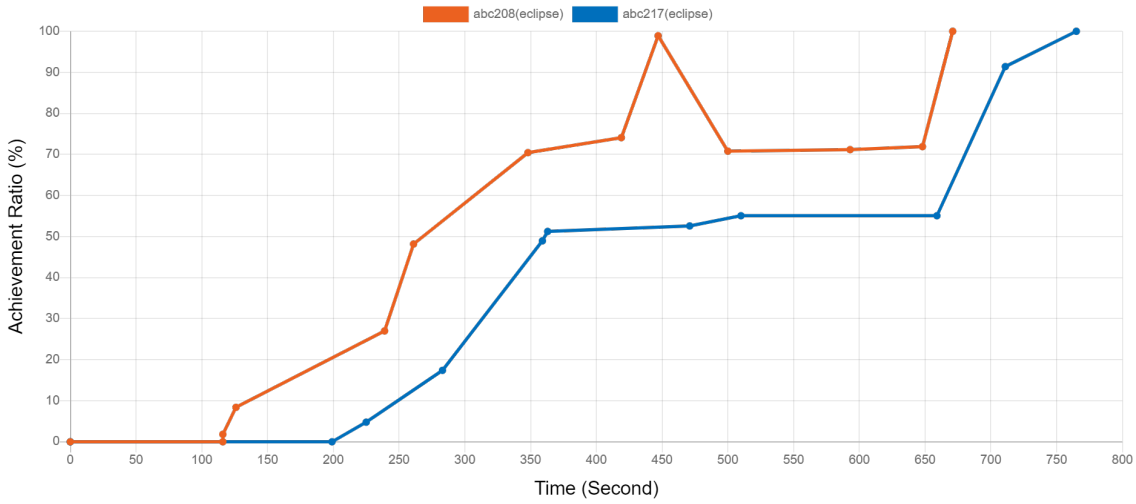


図 25: 問題の難易度の感じ方と進捗がずれているデータ（赤:abc208，青:abc217）

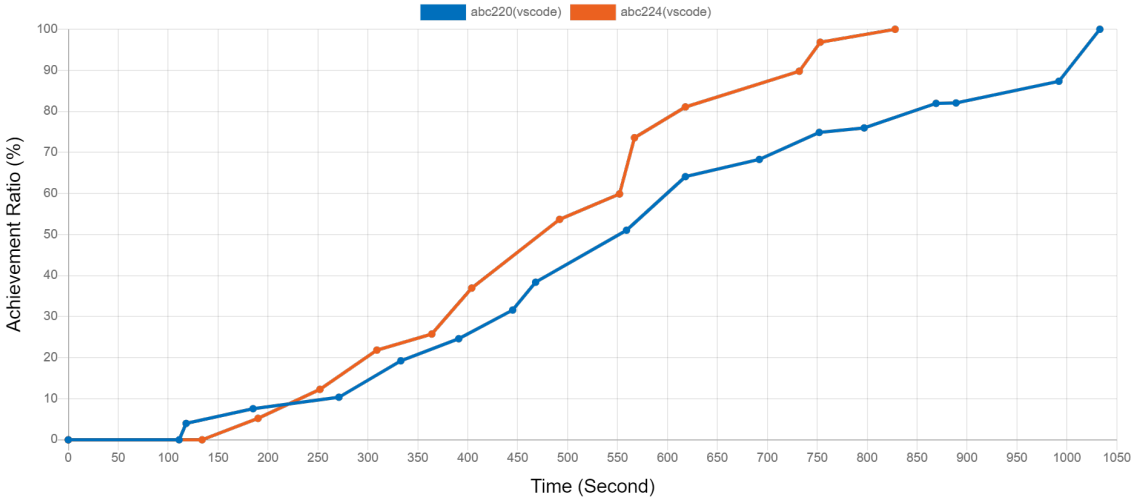


図 26: 問題の難易度の感じ方と進捗がずれているデータ（青:abc220，赤:abc224）

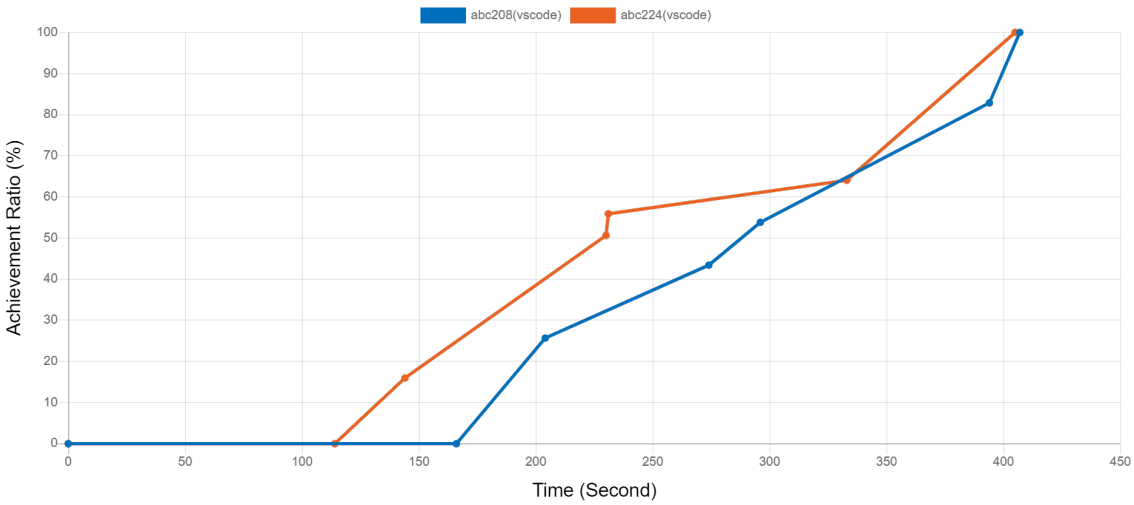


図 27: 問題の難易度の感じ方と進捗がずれているデータ（青:abc208，赤:abc224）

7　まとめ

本研究では，複数の開発環境から収集した細粒度開発履歴の分析の一例として，エディタの作業効率に関する開発者の自己評価と定量的評価を収集し，それらを比較して自己評価と定量的評価にギャップがあるかを調査する方法を提案した．

そして，提案した調査手法の適用例として，Visual Studio Code と Eclipse を対象として，エディタの作業効率に関する開発者の自己評価と定量的評価を収集し比較する実験を行った．実験を通じて，自己評価と定量的評価の比較が実現できていることが確認できた．

今後の課題としては，Visual Studio Code と Eclipse 以外のエディタを対象として調査することが考えられる．提案した調査手法は LSP がサポートされているエディタであれば適用可能である．また，本調査手法における履歴収集の構造では，開発過程で開発環境を変えたとしても履歴収集が可能であるため，それを利用した調査を今後行いたい．例えば，様々な開発環境を使用して一つの開発を行った場合に，開発環境ごとに進捗の違いがみられるかの調査を行いたいと考えている．

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 井上克郎 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． 井上 教授に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において多くの御意見・御助言を賜りました．松下 准教授に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，研究活動における直接の御指導及び本論文執筆における御助言など，多くの場面で御支援を賜りました．神田 哲也 助教の御支援のおかげで本論文の完成に至りました．神田 助教に心より深く感謝いたします.

井上研究室の先輩方におきましては，たくさんの研究に関する御助言をいただきました. 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 嶋利 一真氏，田邉 傑士氏には，研究や発表内容に関する助言，添削など，様々な場面でご協力していただきました. 先輩方の御指導のおかけで本論文を完成させることかできました，心より深く感謝いたします.

最後に，その他様々な御指導及び御助言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井上研究室の皆様に心より深く感謝いたします.

参考文献

[1] AtCoder：競技プログラミングコンテストを開催する国内最大のサイト．https://atcoder.jp/.

[2] Eclipse. https://www.eclipse.org/.

[3] Language Server Protocol. https://microsoft.github.io/language-server-protocol/.

[4] Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance.

[5] Visual Studio Code. https://code.visualstudio.com/.

[6] Stas Negara, Mohsen Vakilian, Nicholas Chenand Ralph E Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? *In European Conference on Object-Oriented Programming*, pp. 79–103, 2012.

[7] RomainRobbes and MicheleLanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, Vol. 166, pp. 93–109, 2007.

[8] 桑原寛明, 大森隆行. 編集操作履歴の再生における粗粒度な再生単位. コンピュータソフトウェア, Vol. 30, No. 4, pp. 61–63, 2013.

[9] 井垣宏, 齊藤俊, 井上亮文, 中村亮太, 楠本真二. プログラミング演習における進捗状況把握のためのコーディング過程可視化システム c3pv の提案. 情報処理学会論文誌, Vol. 54, No. 1, pp. 330–339, 2013.

[10] 大森隆行, 丸山勝久. 統合開発環境における細粒度な操作履歴の収集および応用に関する調査. コンピュータソフトウェア, Vol. 32, No. 1, pp. 60–80, 2015.

[11] 石田直人. 特定エディタに依存しない細粒度編集履歴収集ツールの開発とその適用. 大阪大学修士学位論文, 2020.

[12] 石田直人, 神田哲也, 嶋利一真, 井上克郎. 言語サーバを応用した細粒度編集履歴収集プラットフォームの構想. ソフトウェアエンジニアリングシンポジウム 2020 WS5, 2020.

[13] 田口浩, 島田幸廣, 高田秀志, 島川博光. ストリームデータによる学習者のプログラミング状況把握. 電子情報通信学会 第 18 回データ工学ワークショップ, 2007.