東京大学
情報理工学系研究科 創造情報学専攻
修士論文

キー入力ごとのコード編集を細粒度に記録した IDE のエラー報告評価に資する TypeScript データセットの作成

Collecting a TypeScript Dataset of Fine-Grained Code Edit Logs Per Keystroke for Evaluation of IDE Error Reporting

岩田 風多
Futa Iwata

指導教員　千葉 滋 教授

2026 年 1 月

概要

近年のプログラミングにおいては，コードを編集している最中に IDE がリアルタイムでエラーを報告するのが一般的であるが，望ましくないエラー報告が行われることもあり，更なる改善と評価が必要である．IDE のエラー報告手法の評価にあたっては，手法を実装して被験者に使ってもらう実験が主流であるが，手間がかかるほか再現性の担保も難しいという問題がある．本研究では，実験によりコード編集を収集し，いくつかの条件によってコード編集履歴を抽出することで，エラーが報告されるようなコード編集履歴のデータセットを作成した．キー入力ごとのコード編集を細粒度に記録するロガーのもと，リファクタリングや機能追加の課題を被験者に課すことでコード編集を収集した．分析に基づいて，編集範囲から遠い文法エラーがあり，解決までの編集時間が 2 秒より長く編集数が 2 回より多い期間を抽出しデータセットとした．データセットには 90 個の編集履歴が含まれ，うち 36 個では望ましくないエラー報告が確認されている．

Abstract

In modern programming, it is common for integrated development environments (IDEs) to report errors in real time while code is being edited. However, undesirable error reporting can occur, necessitating further improvement and evaluation. Evaluating IDE error reporting methods typically involves conducting experiments in which the methods are implemented and used by research participants. This approach is labor-intensive and faces challenges in ensuring reproducibility. In this study, we collected code edits through experiments and created a dataset of code edit histories likely to trigger error reports by extracting these histories under specific conditions. Subjects performed refactoring or feature addition tasks under a logger that recorded edits at the keystroke level. After analyzing the data, we identified periods where a syntax error occurred far from the edit range, resolving it took more than two seconds, and more than two edits occurred, forming the dataset. The dataset contains 90 edit histories, of which 36 have been confirmed to involve undesirable error reporting.

目次

第 1 章　はじめに　1

第 2 章　IDE によるエラー報告とその評価　3

2.1　IDE によるエラー報告 3

2.2　IDE による TypeScript のエラー報告の評価におけるデータセットの重要性　7

2.3　エラー報告手法の評価に向けたデータセットの構築 9

第 3 章　エラーが報告されるようなコード編集履歴のデータセット　12

3.1　被験者からコード編集を収集する 12

3.2　望ましくないエラーが報告されるようなコード編集を抽出する 26

3.3　データセットをもとにエラー報告を再現するビューワーの実装 35

第 4 章　まとめと今後の課題　41

4.1　まとめ 41

4.2　今後の課題 41

参考文献　43

第 1 章

はじめに

近年のプログラミングにおいては，コードを編集している最中に IDE がリアルタイムでエラーを報告するのが一般的である．IDE は編集中のコードを逐一解析し，静的解析によって得られるエラーや警告をコード上に報告する．編集中のコード上に報告されたエラーによって，プログラマは細かいエラーを迅速に修正することができる．

IDE によるエラー報告に関する研究は，コンパイラーによるエラーメッセージの流れを汲んで行われている．コンパイラーによるエラーメッセージは，ソフトウェア開発においてもプログラミング教育においても極めて重要である．より良いエラーメッセージの出力については早くから研究されており，1965 年の研究でエラーメッセージの改善を行っているものがある [1]．近年においても，エラー報告をはじめとするプログラマへのフィードバックを分析する研究や [2][3]，エラー報告の改善を試みる研究が多く存在する [4][5][6][7][8]．

エラーメッセージの出力には，文法に則っていないコードについても構文解析を行う文法エラーリカバリという技術が深く関わっている．エラー報告のための文法エラーリカバリは 1973 年には研究されており [9]，近年でも研究が続けられている [10][11][7]．IDE におけるエラー報告では，文法に則っていないコードを解析する必要性がコンパイラーよりもはるかに高いことが指摘されている [12]．

しかしながら現在においても，IDE によるエラー報告は時に望ましくない状態になることがある．大量のエラーが報告されるとプログラマはどのエラーから修正すべきか考える必要がある．エラーメッセージが言っていることが理解できず，よって修正方針もわからない場合がある．文法エラーが編集していない場所で発生して理由がわからない場合もある．

IDE による望ましくないエラー報告を見ると，しばしば IDE によるコードの理解とプログラマによる認識が違っているように見える．例えば，プログラマはオブジェクトのキーだと思っている部分を，IDE が変数の参照だとみなしてエラーを報告し，結果として変数が存在しないというエラーが報告される．これは，編集中のコードに対する IDE による構文解析が不完全であることによる可能性がある．

いずれにせよ，IDE によるエラー報告の改善のためには評価手法が必要であるが，現在主流の評価手法は IDE にエラー報告手法を実装して被験者に使ってもらう実験を行うという，手間がかかり再現性の低いものである．エラー報告の評価を簡単に再現性よく行うことができる

ことが望ましい．簡単で再現性の良い評価は，エラー報告手法を改善するための試行錯誤にも役立つ．

本研究では，IDE によるエラー報告の評価のために，エラーが報告されるようなコード編集履歴のデータセットを作成した．作成にあたっては，まずキー入力ごとのコード編集を細粒度に記録するキーロガーと，リファクタリングを行う課題や機能追加とデバッグを行う課題を被験者に与えて，コード編集を収集した．次に，収集したコード編集からエラーが報告されるようなコード編集履歴を抽出し，データセットとした．

このデータセットについて，望ましくないエラー報告が行われるコード編集履歴を細かく観察する手段も提供する．現在のエラー報告手法を分析したデータから，エラー報告が多かったり編集していない場所の文法エラーが報告されている編集を探し出し，ビューワーによってその編集を現在の手法によるエラー報告とともに観察することができる．

本研究の貢献は，エラーが報告されるようなコード編集履歴のデータセットを作成した点にある．本データセットでは，特に望ましくないエラーが報告される編集履歴 90 個を収集したものであり，エラー報告手法の評価や改善に資するものである．本データセットに含まれる編集履歴 90 個のうち 36 個については，本研究の目的である「望ましくないエラー」が報告されていることを著者において確認している．

以下，第 2 章では，本研究の背景である IDE によるエラー報告について前提知識と課題を説明し，評価のためのデータセットの必要性や，データセットに求められる性質について議論する．IDE が編集されている最中のコードに対して望ましくないエラーを報告すること，その評価のためには編集されている最中のコードに対するデータセットが重要であること，そのデータセットが細粒度であることや多様な編集を含むべきであることの必要性について説明する．

第 3 章では，提案するエラー報告の評価のために必要なデータセットを作成した手順について詳細に述べる．プログラマによるキー入力ごとにコード編集を記録するための設計や，文法の逸脱を含む多様なコード編集を収集するために実験で与える課題の設計，また収集されたコード編集からデータセットに含めるべきコード編集履歴を抽出した方法について説明する．

第 4 章では，本研究のまとめと今後の課題について述べる．本研究の提案とその評価について総括し，本研究に続くことが期待される研究についても説明する．

第 2 章

IDE によるエラー報告とその評価

IDE による編集中のコードに対するエラー報告には改善すべき点も存在するため，エラー報告手法を評価するためにエラーが報告されるようなコード編集履歴のデータセットが重要である．本章では，IDE によるエラー報告の前提と課題を説明し，その評価のためにデータセットが必要であること，そしてデータセットに求められる性質について議論する．

2.1　IDE によるエラー報告

ソフトウェアを開発するために必要な機能を備える IDE は，編集されている最中のコードに対してエラーを報告する機能を備えているが，改善すべき点も存在する．本節では，前提となる IDE と言語サーバーについて説明した上で，エラー報告がどのようなものであり，その課題がどのような点にあるかを概観する．

2.1.1　言語サーバーと LSP

統合開発環境 (IDE) はソフトウェアを開発するための機能を統合的に提供する環境である．IDE は，コード編集をはじめコンパイルやデバッグなど，ソフトウェアを開発するために必要な機能を備え，それらを連携させて利用できるようにしている．

現在の多くの IDE はプログラマのコード編集やデバッグを支援するために言語サーバーという仕組みを利用している．言語サーバーはコードを理解するための機能として，関数やクラスが定義されているコード片に移動する機能や，推論された型などコードに含まれない情報をコード上に表示する機能を提供する．言語サーバーはコードを効率よく編集するための機能として，途中まで書かれた変数名やメソッド名を補完する機能や，関数などの名前を使用箇所も含めて一括で変更する機能を提供する．

言語サーバーが機能を提供するためには特定のプログラミング言語に基づき，構文解析や参照の分析，型推論などを行う必要がある．コードに含まれない情報を表示するにあたっては，参照の分析や型推論によって得られる情報を，構文解析時に構文木に記録したコード上の位置に表示する必要がある．関数やクラスが定義されているコード片に移動するには，その位置に

ある構文木のノードがどれであるか確認し，その参照元を辿る必要がある．変数名やメソッド名を補完するためには，カーソルがある位置のスコープやレシーバーの型に存在するメソッドを把握して列挙する必要がある．

言語サーバーが分析すべき編集されている最中のコードは，文法に則っていないことが多く，そのようなコードを構文解析する手法は多く知られている．編集されている最中のコードは，文法が要求するうちの一部までしか書かれていないこともあるし，単にプログラマの誤りにより文法に則っていないこともある．すでに書かれたコードを後から編集する場合においては，必要な編集の前後それぞれの状態では文法に則っていても，過渡期的に文法に則っていない状態を経由しなければならないことがある．文法に則っていないコードの構文解析は，コードから一意には構文木が決まらないため，そのうち最も尤もらしい構文木を選択する点に難しさがある．文法に則っていないコードを構文解析する代表的な手法においては，構文解析器が進行不能となった際に付近でトークンを追加，置換ないし削除したり，特定のトークンまで読み飛ばしたりすることで，元のコードに近く文法に則ったコードの構文木を出力する．

言語サーバーが提供する機能を複数の IDE で使いまわせるために，言語サーバーと IDE の間では標準的なプロトコルが制定されるのが望ましい．IDE と言語サーバーを別々に実装し，その間の通信のフォーマットを IDE 間や言語サーバー間で共通化することで，1 つの言語サーバー実装を複数の IDE で利用することができ，あるいは 1 つの IDE が複数の言語を統一的に扱いながら言語サーバーの機能を提供できる．このようなプロトコルなく，IDE ごと言語ごとに言語サーバーを実装すると，言語の開発者と IDE の開発者の双方に大きな負担をかけることになる．

言語サーバーは，ワークスペースに含まれるコードの状態を，IDE から知らされることで把握する．IDE はコードの状態が変わったことを逐一言語サーバーに通知し，言語サーバーはその通知に応じて構文木や型推論などの情報を更新する．言語サーバーが直接ファイルを読み出すのではなく言語サーバーから情報を通知されることで，ファイルを保存しなくても最新のコードに対する支援を提供することができる．

現在の言語サーバーは Language Server Protocol (LSP)*1 に則って実装されることが一般的であり，IDE も LSP に対応していることが多い．LSP でワークスペースに含まれるコードの状態を知らせるには，Text Document Synchronization*2 という仕組みを利用して，編集ごとに差分だけを言語サーバーに通知する．

2.1.2 エラー報告

IDE は，編集されている最中のコードに対して検証を行い，エラーがあればプログラマに報告する機能を備えている．汎用プログラミング言語におけるエラー報告では，構文解析や型推論，変数のスコープ解析や型検査などを行い，プログラムを実行せずに静的に判明するエラー

*1 https://microsoft.github.io/language-server-protocol/

*2 https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_synchronization

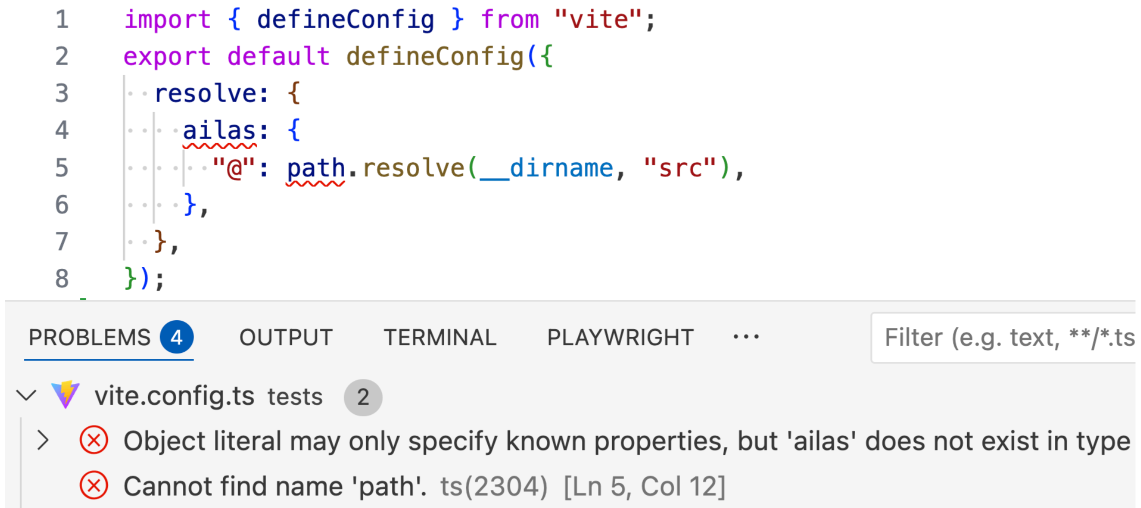


図 2.1: VS Code 上で報告される TypeScript のエラー

を列挙し，それをコード上や専用の画面で表示する．IDE によるエラー報告は，プログラマがコードに加えるべき修正に関するヒントを与え，もしくは修正を提案することで，コード編集やプログラムのデバッグをより効率的に行うことができるようにする．エラー報告機能はふつう言語サーバーによって実装される．

VS Code 上で TypeScript のエラーが 2 つ報告されている例を図 2.1 に示す．1 つのエラーは `ailas` というスペルミスを報告するもので，もう 1 つのエラーは `path` という変数が存在しないことを報告するものである．これらのエラーはいずれもプログラマがコードに加えるべき修正に関するヒントを与えている．

エラー報告も，編集されている最中の文法に則っていないコードの構文解析や，それによって得られる構文木やに基づいて行われる．図 2.1 の例で，`path` という変数が存在しないエラーを報告するには `path` の位置におけるスコープにある変数の一覧を構文木などの解析で得る必要があるし，`ailas` というスペルミスを報告するには `defineConfig` の型およびその第一引数の型の定義を参照する必要がある．

2.1.3 現在のエラー報告における課題

IDE 上でコードを編集した結果，小さな範囲しか編集していないにも関わらず，編集範囲から遠い場所に文法エラーが報告されたり，エラー報告の数が大幅に増えることがある．編集範囲から遠い場所の文法エラーが報告されることは，その理由や次に行うべき修正がわかりづらいため問題である．エラー報告の数が大幅に増えることは，次に行うべき修正がわかりづらくなるため問題である．

図 2.1 のコードから VS Code 上で 2 行目末尾の`{`を削除したあとに TypeScript のエラーが 5 つ報告されている例を図 2.2 に示す．報告されているエラーは次の通りである．

- `resolve` という変数が存在しないことを報告するエラー

```
1  import { defineConfig } from "vite";
2  export default defineConfig(        You, 20 minutes
3    resolve: {
4      ailas: {
5        "@": path.resolve(__dirname, "src"),
6      },
7    },
8  });
```

PROBLEMS 7 OUTPUT TERMINAL PLAYWRIGHT ...

vite.config.ts tests 5
- Cannot find name 'resolve'. ts(2304) [Ln 3, Col 3]
- ',' expected. ts(1005) [Ln 3, Col 10]
- Expected 1 arguments, but got 2. ts(2554) [Ln 3, Col 12]
- Cannot find name 'path'. ts(2304) [Ln 5, Col 12]
- Argument expression expected. ts(1135) [Ln 8, Col 1]

図 2.2: 図 2.1 のコードから VS Code 上で 2 行目末尾の{を削除したあとに報告されるエラー

- `resolve` の直後に`:`があることは文法に反することを報告するエラー
- 8 行目冒頭に}があることは文法に反することを報告するエラー
- `defineConfig` の引数の数が型と合わないというエラー
- `path` という変数が存在しないことを報告するエラー

2.1 では 2 つのエラーが報告されていたところ，2.2 で報告されるエラーは 5 つに増えている．特に，`resolve` という変数が存在しないことを報告するエラーや，`defineConfig` の引数の数が型と合わないというエラーは，次に行うべき修正がわかりづらい．また，`resolve:` に報告されている文法エラーは編集範囲に近いが，8 行目冒頭の}に報告されている文法エラーは編集範囲から遠い場所にある．

2.1.4 関連研究

IDE によるエラー報告はコンパイルエラーに関連して研究されることがある．これは，あるプログラムに対する IDE によるエラー報告の内容はコンパイルエラーと共通することが多いからだと考えられる．ここでは，IDE によるエラー報告に加えて，コンパイルエラーに関連する研究も挙げる．

Layman ら [13] は，開発者が興味を持ちそうなエラーだけを表示する手法を提案している．この研究では，エラーの重要度，開発者の現在の作業コンテキストとの関連性，プログラミング環境での開発者の行動に基づいてエラーの関心度を推定し，関心の高いエラーのみを表示する MimEc システムを開発している．

Barik ら [5] は，エラーを分類してそれぞれに適した表示をする手法を提案している．この研究では，コンパイラエラー通知とその解決方法について，インタラクション中心のアプローチを通じて開発者がより効果的に理解し解決できるよう支援している．エラー通知とその解決方法のための新しい分類体系を提案し，またこれらの分類体系に基づくエラーの可視化を行っている．

コンパイラが出力するエラーを改善する研究も多々ある．Becker ら [12] は，構文解析器を改善することでコンパイラによる構文エラーを改善する研究を包括的に調査している．この研究では，テキストベースのプログラミングにおけるエラーメッセージ研究の全体像を示し，半世紀にわたる研究成果を体系化している．しかし，IDE 上で編集中の未完成なコードに対応するものではない．このレビュー論文は IDE 上でのリアルタイムなエラー報告について Live Compilation という言葉で問題を提起している．

Algaraibeh ら [7] は，スコープに注目した新たな構文解析手法によってエラーを改善する研究を行っている．この研究では，学生プログラマのためのコンパイラ構文エラーメッセージを強化する解析技法を提案している．この研究は構文エラーの報告の改善を行っている．

2.2　IDE による TypeScript のエラー報告の評価におけるデータセットの重要性

IDE による TypeScript のエラー報告を評価するためには，編集されている最中の文法に則っていないコードを収集したデータセットが重要である．IDE によるエラー報告手法は編集されている最中のコードを入力としてエラー報告を出力とする．複数のエラー報告手法について入力となるデータセットを用いて出力を得られることは，エラー報告手法に対して簡単で再現性の高い評価を与えるために重要である．

IDE によるエラー報告のためには，編集されている最中のコードの構文解析が必要である．編集されている最中のコードは，文法に則っていることもあるが，文法を大きく逸脱する状態であることもある．文法に則っていないコードを収集したデータセットには，現在の IDE によるエラー報告機能が想定していないようなコードが含まれる可能性がある．

TypeScript において，`if` 文の中に `if` 文が 1 つだけ存在する場合に，2 つの条件を`&&`で結合することで `if` 文を 1 つにまとめるための編集と，その間に起きる文法の逸脱について考える．このような編集の間には，多くの場合で一時的に文法を逸脱した状態に陥ると考えられる．

2 つの `if` 文の条件を結合する編集の例の 1 つは，図 2.3 のような編集である．編集前のコードの 1 行目にある `if` 文の条件と 2 行目にある `if` 文の条件を`&&`で結合することで，編集後のコードの 1 行目の `if` 文の条件とし，編集前のコードの 3 行目にあった処理をインデントを 1 つ下げて編集後のコードの 2 行目に配置する．編集前のコードにあった 2 つの `if` 文のうち片方を削除する．

2 つの `if` 文の条件を結合する編集を，中間状態での文法の逸脱が少なくなるような手順で行うことは可能である．図 2.3 の編集を行うにあたり，文法の逸脱が少ない手順の 1 つを図

```
1 if (this.available()) {
2   if (!this.used) {
3     this.used = true;
4   }
5 }
```

```
1 if (this.available() && !this.used) {
2   this.used = true;
3 }
```

図 2.3: if 文の条件を 1 つにまとめる編集の前後の状態

2.4 に示す．まず，1 行目にある `if` 文の条件の末尾に `&&` を 1 字ずつ入力する．次に，2 行目にある `if` 文の条件をコピーし，1 行目にある `if` 文の条件の末尾にペーストする．続いて，エディタの行入れ替え機能によって 2 行目と 3 行目を入れ替える．最後に，3 行目と 4 行目にある `if` 文をまとめて選択して削除する．この手順において，文法を逸脱する中間状態は `&&` を入力した状態までに限られる．

しかしながら，2 つの `if` 文の条件を結合する編集にあたり，中間状態での文法の逸脱がより多い手順も考えられる．図 2.3 の編集を行うにあたり，文法の逸脱が多い手順の 1 つを図 2.4 に示す．まず，1 行目から 2 行目にわたる部分`) {if (`をまとめて選択して削除する．次に，削除した部分に `&&` を 1 字ずつ入力する．続いて，2 行目と 3 行目のインデントを下げる．最後に，4 行目の`}`を削除する．この手順においては，編集が完了するまでコードは文法を逸脱したままである．加えて，元々対応していなかったカッコの組を新たに対応させていることがわかる．

エラーが報告されるようなコード編集履歴のデータセットによるエラー報告手法の評価は，被験者を集めてエラー報告手法を使用してもらうよりも，簡単で再現性が高い．被験者を集めて実験することでエラー報告を収集することは，大きな労力を要する．エラー報告手法を提案し実装するいくつかの研究では，その評価のために被験者を集めてエラー報告手法を使ってもらいデータを集めることを，今後の課題としている [13][6]．エラーが報告されるようなコード編集履歴のデータセットは，異なるエラー報告手法を評価するにあたって条件を揃えるのが容易である．エラーが報告されるようなコード編集履歴のデータセットはエラー報告手法の入力とできるから，同じ入力を渡して得られた出力を比較することで，エラー報告手法の比較を精密に行うことができる．

エラーが報告されるようなコード編集履歴のデータセットによる簡単で再現性が高い評価は，エラー報告を改善するにあたっても重要である．簡単で再現性が高い評価は，エラー報告の改善を行うにあたって試行錯誤のためにも重要となる．開発中のエラー報告手法をエラーが報告されるようなコード編集履歴のデータセットで評価することは，より良いエラー報告手法の開発につながると考えられる．

```
1 if (this.available() && ) {
2   if (!this.used) {
3     this.used = true;
4   }
5 }
```

```
1 if (this.available() && !this.used) {
2   if (!this.used) {
3     this.used = true;
4   }
5 }
```

```
1 if (this.available() && !this.used) {
2   this.used = true;
3   if (!this.used) {
4   }
5 }
```

```
1 if (this.available() && !this.used) {
2   this.used = true;
3 }
```

図 2.4: 図 2.3 の編集を行う，文法の逸脱が少ない手順の 1 つ

2.3 エラー報告手法の評価に向けたデータセットの構築

エラー報告手法の評価のためには，編集されている最中の文法に則っていないコードをできるだけ細かい粒度で再現し観察することができ，その作成後に IDE や言語サーバーによってエラー報告を再現できるデータセットが必要である．コード編集はプログラマがキー入力をするごとに記録され，しかも編集範囲ができるだけ小さい範囲で特定できる必要がある．これは，特に編集範囲から遠い場所に文法エラーが報告される問題を観察するために必要となる．データセットのデータを入力として，作成時点では存在しないエラー報告手法による出力を得て，その特徴を統計的に観察し，あるいは 1 つ 1 つのエラー報告をコード編集と共に観察できる必要がある．これは，作成時点では存在しないエラー報告手法にも定量的ないし定性的な評価を与えるためである．

編集中のコードを細粒度に取得する研究はいくつか存在する．いずれの研究も TypeScript 以外の言語を対象としているほか，エラー報告が行われるようなコードを特に収集するものではない．これらの研究ではより広範で汎用性の高いデータを収集することを目的としており，エラー報告の再現や観察においては不要なデータが多く含まれる．特に編集履歴についてもそ

```
1 if (this.available()!this.used) {
2     this.used = true;
3   }
4 }
```

```
1 if (this.available() && !this.used) {
2     this.used = true;
3   }
4 }
```

```
1 if (this.available() && !this.used) {
2   this.used = true;
3 }
4 }
```

```
1 if (this.available() && !this.used) {
2   this.used = true;
3 }
```

図 2.5: 図 2.3 の編集を行う，文法の逸脱が多い手順の 1 つ

の全体を取得しており，エラー報告が行われるようなコードだけを積極的に収集していない．

Schröer ら [14] は，Eclipse IDE においてプログラマによる行動をひろく収集するプラグイン Mimesis を開発している．Mimesis はプログラマが課題を解決する様子を観察することを目的として，細粒度のコード編集をはじめ，テキストの範囲選択やテストの実行，ブラウザでの検索に至るまで，幅広い行動ログを収集している．Mimesis では Java の編集を収集することができる．この研究はデータセットを公開していない．

中丸ら [15] は，Jupyter Notebook 環境においてプログラマによる行動を収集するツール Redmine[*3]を開発し，これを用いてデータセットを作成して提供している．通常のエディタではなく，Notebook 環境におけるプログラマの課題解決の様子を観察することを目的に，細粒度のコード編集やセルの実行などのログを収集している．この研究は Python のデータセットを公開している．

Proksch ら [16] は，Visual Studio においてプログラマによる行動を収集するツール FeedBaG++[*4]を開発し，これを用いてデータセットを作成して提供している．FeedBaG はプログラムの編集をテキストではなく，SST と呼ばれる追加情報を付与した具象構文木で取得してデータを提供している．この研究は C#のデータセットを公開している．

Edwards ら [17] は，細粒度のコード編集を取得するプログラミング初学者向け Python 開

[*3] https://github.com/tomokinakamaru/redspot
[*4] https://www.kave.cc/feedbag

発環境を開発し，これを用いてデータセットを作成して提供している．また，得られたデータをもとにキー入力ごとの編集を再現するツール KeystrokeExplorer[5]も提供している．この研究は Python のデータセットを公開している．

[5] https://edwardsjohnmartin.github.io/KeystrokeExplorer/

第 3 章

エラーが報告されるようなコード編集履歴のデータセット

キー入力ごとのコード編集を記録するキーロガーのもと，リファクタリングを行う課題や機能追加とデバッグを行う課題を被験者に与えることで，エラー報告を再現できるデータセットを作成できる．本章では，エラー報告の評価のために必要なデータセットを作成する手法を提案し，その細かい粒度を実現するコード編集の記録方法と，文法を逸脱した状態を含む多様な編集を収集するための課題設計の 2 つの軸に沿って説明する．

3.1　被験者からコード編集を収集する

被験者にキー入力ごとのコード編集を記録するロガーのもとで課題に取り組んでもらうことでデータを収集し，16,543 件のイベントを分析することができる．本節では，コード編集を収集する方法，特に被験者に課した 2 種類の課題やそれに取り組む間のコード編集を記録するための VS Code 上のロガーの設計や実装について詳しく説明し，収集されたコード編集の概要を説明する．

3.1.1　コード編集を収集する実験の実施

被験者にキー入力ごとのコード編集を記録するロガーのもとで課題に取り組んでもらうことでデータを収集する実験を実施した．実験は 2025 年 12 月 6 日から 2025 年 12 月 19 日にかけて行われ，13 人の被験者が協力した．被験者には，あらかじめ実験の趣旨と手順を説明する書面をインターネット上で提示し，データを収集し公開することに関する同意を Web 上のフォームで得た．被験者は，研究への同意やデータの提出について実験後 3 ヶ月間は撤回が可能であることを案内された．

被験者は，自身のコンピュータに VS Code がインストールされていることを確かめた上で，実装したロガーである VS Code 拡張機能をインストールするよう案内された．インストールにあたっては，ログが自動で送信されないこと，実験の終了後にアンインストー

```
.
|-- .tdlog/
|-- .vscode/
|    |-- extensions.json
|    |-- settings.json
|-- README.md
|-- .gitignore
```

図 3.1: 両課題に共通する関連ファイル群．末尾の/はディレクトリを表す

ルすることが案内された．使用した textdocument-logger のバージョンは `0.0.2` であった．textdocument-logger は VS Code の拡張機能を公開する Microsoft のサイト Visual Studio Marketplace で公開されている*1.

拡張機能に関する案内を行ってなお情報の送信に心配が残る被験者には，GitHub Codespaces*2という仮想環境を案内した．GitHub Codespaces はインターネット上にあるサーバーにより提供され，実験に必要な VS Code 拡張機能や機能追加とデバッグを行う課題で必要な自動テストが動作する．GitHub Codespaces は自身のコンピュータにインストールされている VS Code でも操作できるほか，ブラウザ上からも操作できる．

リファクタリングを行う課題と機能追加とデバッグを行う課題はそれぞれ，GitHub 上の公開リポジトリによって配布した．リポジトリには，課題の内容のほか，課題について説明する README ファイルや，拡張機能がログを記録するための設定が書かれたファイルが含まれる．被験者は，自身のコンピュータ上にインストールされた `git` によって課題を取得した．

課題を配布に利用した両リポジトリに共通する構成を図 3.1 に示す．`.tdlog/`ディレクトリには，textdocument-logger が記録するログが配置される．`.vscode/extensions.json` には textdocument-logger の識別子が記されており，VS Code でこのリポジトリをワークスペースとして開いた際に VS Code が拡張機能のインストールが勧める．`.vscode/settings.json` には textdocument-logger を有効化する設定が記されている．`README.md` には，課題の進め方と提出に必要なコマンドが記されている．

被験者は 2 つの課題に合わせて 1 時間取り組むこととし，謝礼を支払われた．取り組む時間は，長時間の拘束を行うべきでないことと支払える謝礼の合計により定められた．準備した課題は 1 時間で取り組むには多い量であり，被験者は課題を完了する必要がないことを案内された．

被験者は課題に取り組んだのち，ログファイルをアーカイブして Web 上のフォームで提出するよう案内された．アーカイブは，`.tdlog` フォルダを `tar` コマンドで 1 ファイルにすることで行われた．ログファイルは，実験の参加に関する同意とともに顕名で保存され，データの提出の撤回によってデータを削除できることとした．

*1 `https://marketplace.visualstudio.com/items?itemName=iwata-csg.textdocument-logger`
*2 https://github.co.jp/features/codespaces

課題名	概要
4-1-reassign	let で宣言された変数への再代入を避け，const で複数の宣言を行う．
5-14-out-argument	引数のオブジェクトの書き換えを避け，書き換わった新しいオブジェクトを返す．
6-1-nested-if	多重 if 文を避け，早期 return する．
6-7-nested-else	多重 else if を避け，早期 return する．
6-11-switch	同じ enum に対する switch 文を 1 つにまとめる．
7-3-collection	コレクションの処理を 1 つの class にまとめる．
14-1-refactoring	早期 return，改名，処理順の整理を行う．
if-conditions	2 つの if 文の条件をまとめて 1 つの if 文とする．

表 3.1: リファクタリングを行う課題の一覧

3.1.2 多様な編集を行う課題を提示する

リファクタリングを行う課題や機能追加とデバッグを行う課題など，多様な編集を行う課題を提示することで，より多様なエラーが報告されるようなコード編集履歴を収集できると考えられる．本節では，文法を大きく逸脱するコード編集を収集するためのリファクタリング課題と，前から順に入力されるコード編集や試行錯誤のためのコード編集を収集できる機能追加課題のそれぞれについて説明する．

リファクタリングを行う課題

編集されている最中の，特に文法を大きく逸脱するコードを収集するために，リファクタリングを行う課題を提示する．リファクタリングを行う課題では，被験者にリファクタリング前のコードとリファクタリング後のコードを示し，リファクタリング後のコードになるように編集するよう指示する．提示する課題の一覧を，表 3.1 に示す．これらの課題はリファクタリングを取り扱う書籍[*3]で提示されているサンプルコードを参考に作成した．

リファクタリングを行う課題の 1 つである 6-7-nested-else の書き換えるべき部分を抜粋し，図 3.2 に示す．この課題では，hitPointRate を元に適切な HealthCondition を返すために，HealthCondition を一度変数 currentHealthCondition に代入していたのをやめて，if 文の中から直接 return する．この課題では，条件式や返すべき式の部分は挙動を変えないために編集を避けることが望ましい．条件式 hitPointRate === 0 や HealthCondition.Dead は編集を避けることが望ましく，間にある{と currentHealthCondition = と}else は書き換える必要がある．{は削除する必要があるが，これだけを削除するとかっこの対応が崩れ

[*3] 良いコード／悪いコードで学ぶ設計入門，技術評論社 [18]

```
1 let currentHealthCondition: HealthCondition;
2 if (hitPointRate === 0) {
3   currentHealthCondition = HealthCondition.Dead;
4 } else if (hitPointRate < 0.3) {
5   currentHealthCondition = HealthCondition.Danger;
6 } else if (hitPointRate < 0.5) {
7   currentHealthCondition = HealthCondition.Caution;
8 } else {
9   currentHealthCondition = HealthCondition.Fine;
10 }
11 return currentHealthCondition;
```

```
1 if (hitPointRate === 0) return HealthCondition.Dead;
2 if (hitPointRate < 0.3) return HealthCondition.Danger;
3 if (hitPointRate < 0.5) return HealthCondition.Caution;
4 return HealthCondition.Fine;
```

図 3.2: `else if` を早期 `return` に書き換える課題の例

て文法を逸脱する．

リファクタリングを行う課題は，外から見た挙動を変えないように保ちつつ構造を整理するため，文法を大きく逸脱した中間状態を含むことが多いと考えられる．これは，リファクタリングタスクが外から見た挙動を変えないように保ちつつ構造を整理するものであることによる．外から見えない挙動を変えないためには，既存のコードに含まれる式や文をそのまま残しつつ制御構文を書き換える．変更の必要のない複数のコード片の間に挟まれる位置にある制御構文だけを書き換えるために，文法を逸脱する中間状態になりやすいと考えられる．

機能追加とデバッグを行う課題

自然なコード編集の大部分を占めると考えられる，前から順に入力されるコード編集や小さなコード片だけを書き換えるコード編集を収集するために，機能追加とデバッグを行う課題を提示する．この課題では，被験者に簡単な RESTful API のロジックを実装させ，自動テストを用いてデバッグさせる．課題として提示する内容には，実装すべきロジックを書き起こした仕様書，入出力を行う部分が最低限実装されたコード，仕様に準拠した自動テストが含まれる．

仕様書では，RESTful API がどのようなリソースを扱うのか，どのようなエンドポイントが存在しており，それぞれのエンドポイントが何を行うものかを説明する．この課題では勤務状況を報告する RESTful API を実装するため，勤務や打刻などのリソースを，勤務を開始する処理を行う HTTP エンドポイントなどによって作成したり，あるいは状態を変化させる．勤務や打刻などのリソースの型定義は仕様書にも記述されているが，後述する初期コードにも含めている．

エンドポイント仕様

POST `/work` - 新しい勤務開始

サポートする打刻タイプ:

- `"start"`: 出勤
- **リクエストボディ**:

```
{ "type": "start", "time": [タイムスタンプ] }
```

- **レスポンス**: 勤務IDと開始時刻を含むオブジェクト

```
{
  "id": "[勤務ID]",
  "startTime": [タイムスタンプ],
  "stamps": [{ "type": "start", "time": [タイムスタンプ] }]
}
```

図 3.3: 実装すべき仕様の一部

仕様書のうち，勤務を開始する処理を行うエンドポイントについて説明する部分を抜粋し，図 3.3 に示す．ここでは，勤務の開始が `POST /work` なるエンドポイントで行われること，そのエンドポイントがどのような入出力を行うかが説明される．このエンドポイントの入力には打刻タイプ`"type"`として出勤を表す固定の文字列`"start"`を，その時刻`"time"`としてタイムスタンプを受け取ることが説明される．このエンドポイントの出力は勤務 ID`"id"`と出勤時刻`"startTime"`と打刻の列`"stamps"`が含まれることが説明される．

課題を提示した時点の初期コードにおいて，入力がすでに変数として利用可能な状態であり，出力をどのような値で作るべきか明らかであるようにする．この課題では HTTP リクエストおよびレスポンスを扱うために WHATWG[*4]が標準化する Fetch API[*5]を用い，リクエストを表すクラス `Request` を受け取ってレスポンスを表すクラス `Response` を返す関数 `fetch` として RESTful API を実装する．被験者が `Request` や `Response` の扱い方を知らなくてもロジックの実装に進められるように，入出力の内容を TypeScript の式として扱うまでの部分のコードは初期コードに含める．また，Fetch API は TypeScript や JavaScript で HTTP リクエストおよびレスポンスを扱うための標準的な方法であり，被験者が `Request` や `Response` の扱い方を知っていることも期待できる．

[*4] https://whatwg.org
[*5] https://fetch.spec.whatwg.org

課題を提示した時点の初期コードの一部を図 3.4 に示す．ここでは/work を指定するエンドポイント群の処理を行う関数を被験者に実装させる．/work を指定するエンドポイント群のうち，被験者は最初に勤務を開始する処理を行うエンドポイントを完成させることになる．この関数はリクエストを表すクラス Request を受け取り，レスポンスを表すクラス Response を返すことが，型定義により示されている．エンドポイントの入力として解釈する必要があるクエリパラメータ"id"やリクエストボディは変数に束縛されており，被験者が Request を扱うことなく TypeScript のコードを書いてロジックを実装できる状態にある．3 行目で Request のインスタンスに含まれる URL からクエリパラメータ"id"を取得し，変数_id に束縛している．11 行目で Request のインスタンスからリクエストボディを取り出して JSON として解釈し，その値を body に束縛している．body はリクエストボディに期待される型を持つように束縛されている．これにより，被験者は Request の扱い方を知らなくても入力の値を得ることができる．エンドポイントの出力として返す必要のある Response を作成するコードがすでに書かれており，被験者がこれを編集したり真似することで出力を変えられるように示されている．17 行目から 21 行目でレスポンスボディとして返すべき内容を作成し，それを含む Response のインスタンスを返すコードが記述されている．21 行目で satisfies によりレスポンスボディとして返すべき型を指定している．これにより，被験者は Response の扱い方を元々知らなかったとしてもレスポンスを返すことができると期待される．5 行目から 8 行目で，勤務の現在の状態を返すエンドポイント GET を実装している．ここでは，メモリ上に勤務の状態を保存する変数 db から勤務の現在の状態を読み出して返している．16 行目のコメントを含め，被験者は勤務の状態を db に保存するということを理解した上でロジックを実装できると期待される．

仕様に準拠した自動テストを提供することで，被験者が試行錯誤やデバッグを容易に行うことができるようにする．この課題ではリクエストを表すクラス Request を受け取りレスポンスを表すクラス Response を返す関数 fetch を実装するから，その関数に対するテストを提供する．自動テストは，Node.js[*6]上で実行する．自動テストを提供し，それが成功するように実装するように被験者に伝えることで，前から順に入力されるコード編集だけでなく，試行錯誤の間に小さなコード片だけを書き換えるコード編集を得られると考えられる．

勤務を開始する処理を行うエンドポイントの挙動を確かめる自動テストの一部を図 3.5 に示す．この部分では，勤務を開始する処理を行うエンドポイントに対する正しい入力を行い，出力が期待されるような値であるかを確かめる．10 行目において，Request のインスタンスを関数 fetch に入力し，出力として Response のインスタンスを得ている．1 行目から 9 行目において，エンドポイントに対する正しい入力を表す Request のインスタンス startWorkRequest を作成している．3 行目で Request コンストラクタの第一引数として渡している URL 文字列は，Node.js においては絶対 URL である必要があるから，処理には用いられないオリジン部分 https://api.example.com も含まれている．Request コンストラクタの第一引数に渡している URL のパス/work と第二引数の method キーに渡している

[*6] https://nodejs.org/

```
1  export async function work(request: Request): Promise<Response> {
2    // クエリパラメータから id を取得する．無ければ null になる
3    const _id = new URL(request.url).searchParams.get("id");
4    // GET /work?id=... の場合は勤務情報を返す
5    if (request.method === "GET") {
6      const work = db.get(_id!);
7      return Response.json(work);
8    }
9
10   // リクエストボディを JSON としてパースし，Stamp 型として扱う
11   const body: Stamp = await request.json();
12
13   // body.type に応じて処理を分岐する
14
15   // body.type が "start" の場合の戻り値
16   // 以下は未完成であり，‘./db.ts‘ の db にこの戻り値を記録する必要がある
17   return Response.json({
18     id: ‘work-${nextId++}‘,
19     startTime: body.time,
20     stamps: [body],
21   } satisfies Work);
22 }
```

図 3.4: 課題を提示した時点の初期コードの一部（コメントを含む）

HTTP メソッド"POST"から，startWorkRequest は POST /work に対するリクエストを表す Request のインスタンスであると読み取れる．11 行目から 16 行目において，エンドポイントの出力として得られる Response のインスタンス startWorkResponse の期待する性質について調べている．ここで，assert は Node.js の組み込みモジュール node:assert/strict*7 である．11 行目で startWorkResponse.status が正常終了を示す 200 であることを確かめている．12 行目でレスポンスボディを取り出し，13 行目から 16 行目でレスポンスボディについて，startTime というキーがあり値がリクエストの time キーと一致すること，stamps というキーがあり値がリクエストのオブジェクト 1 つを含む配列であることを，それぞれ確かめている．

3.1.3　キー入力ごとのコード編集を記録する

プログラマによるキー入力ごとにコード編集を記録し，また一連の編集におけるエラー報告を再現する．本節では，VS Code 上のコード編集を細かく記録するための設計に関して，記

*7 https://nodejs.org/api/assert.html#strict-assertion-mode

```
1  const startTime = Date.parse("2025-01-07T09:00:00Z");
2
3  const startWorkRequest = new Request("https://api.example.com/work", {
4    method: "POST",
5    headers: {
6      "Content-Type": "application/json",
7    },
8    body: JSON.stringify({ type: "start", time: startTime }),
9  });
10 const startWorkResponse = await fetch(startWorkRequest);
11 assert.strictEqual(startWorkResponse.status, 200);
12 const startWorkData = await startWorkResponse.json();
13 assert.partialDeepStrictEqual(startWorkData, {
14   startTime,
15   stamps: [{ type: "start", time: startTime }],
16 });
```

図 3.5: 自動テストの一部

録するイベントの内容とイベントを記録する頻度のそれぞれについて説明する．

コード編集を記録する

ファイルを開いた，ファイルを編集した，ファイルを閉じた，という 3 種類のイベントの列を記録することで，エラー報告の再現に必要なコード編集を記録することができる．ファイルを開いたイベントには，開いた時点のファイルの内容を完全に記録する．ファイルを編集したイベントには，ファイル内のどの部分をどう編集したかという情報を記録する．イベント列の中でイベントの種類を区別するために，それぞれのイベントには自身がどのイベントであるかを識別するためのタグが含まれる．これらのイベントは，VS Code の機能によって容易に取得できる．

ファイルを開いたイベントとファイルを編集したイベントには，そのイベントが完了した時点でのファイルの状態のバージョン番号を記録することで，後の分析で参照することができるほか，一連のファイル編集のうち再現したい部分を特定することができる．バージョン番号は 1 以上の整数で表し，1 回のコード編集で 1 増えるとする．

ファイルを開いたイベントには，ファイルを開いたイベントのタグと，開いたファイル名，開いた時点でのバージョン番号．開いた時刻，そして開いたファイルの内容が記録される．ここで，開いたファイル名の記録にあたっては，ワークスペース内の相対的な位置がわかるようにする必要があるが，ホームディレクトリ名などの個人を特定しうる情報を含んではならない．

図 3.6 はファイルを開いたイベントを JSON に記録した例である．ファイルを開いたイ

```
1 {
2   "type":"textDocument/didOpen",
3   "body":{
4     "textDocument":{
5       "uri":"file:///workspace/4-1-reassign/before.ts",
6       "version":1,
7       "text":"export interface ...\n"}},
8   "timestamp":1764996964006}
```

図 3.6: ファイルを開いたイベントの例

ベントのタグは"type"に記録されており，値は固定の文字列"textDocument/didOpen"である．ファイルを開いた時刻は"timestamp"に記録されている．開いたファイル名は"body"内の"textDocument"内の"uri"に記録されている．このファイル名は固定の文字列 file:///workspace/とワークスペース内の相対パスを結合したものであり，ワークスペース内の相対的な位置がわかり，ホームディレクトリ名などの個人を特定しうる情報を含まない．ファイルを開いた時点でのバージョン番号は"version"に記録されている．開いたファイルの内容は"body"内の"textDocument"内の"text"に記録されている．開いたファイルの内容が長いため，図では大部分を省略している．

ファイルを編集したイベントには，ファイルを編集したイベントのタグと，編集したファイル名，編集した時点でのバージョン番号，編集した時刻，そして編集を置換の集合として表現したものが記録される．開いたファイル名はファイルを開いたイベントと一貫していなければならない．バージョン番号は編集前よりちょうど 1 大きくなければならない．編集を表現する置換は，ファイル内の範囲を示す情報とテキストの組で表す．ファイル内の範囲は，編集の影響を受ける範囲の左端と右端の位置の組により表す．1 回のキー入力で複数の範囲が編集された場合，それは 1 つのイベントに記録する．

図 3.7 はファイルを編集したイベントを JSON に記録した例である．ファイルを編集したイベントのタグは"type"に記録されており，値は固定の文字列"textDocument/didChange"である．編集されたファイル名は"body"内の"textDocument"内の"uri"に記録されている．編集後のバージョン番号は"body"内の"textDocument"内の"version"に記録されている．編集を置換の集合として表現したものは"body"内の"contentChanges"に配列として記録される．各置換はファイル内の範囲を示す情報"range"とテキスト"text"の組で表される．左端と右端はそれぞれどの行の何文字目かで表され，行が"line"，何文字目かが"character"に記録される．置換先のテキストは"text"に記録される．この例では，1 つ目の置換は行 11 において改行とインデントが追加されたことを表し，2 つ目の置換は行 11 の冒頭 2 字が削除されたことを表し，これらの置換が同時に行われたことが記録されている．ファイルを編集した時刻は"timestamp"に記録される．

ファイルを閉じたイベントには，ファイルを閉じたイベントのタグと，閉じたファイル名，

```
{
  "type":"textDocument/didChange",
  "body":{
    "textDocument":{
      "uri":"file:///workspace/4-1-reassign/before.ts",
      "version":3},
    "contentChanges":[
      {"range":[{"line":11,"character":2},{"line":11,"character":2}],
      "text":"\n  "},
      {"range":[{"line":11,"character":0},{"line":11,"character":2}],
      "text":""}]},
  "timestamp":1764996978132}
```

図 3.7: ファイルを編集したイベントの例

```
{
  "type":"textDocument/didClose",
  "body":{
    "textDocument":{
      "uri":"file:///workspace/4-1-reassign/before.ts"}},
  "timestamp":1764997056358}
```

図 3.8: ファイルを閉じたイベントの例

閉じた時刻が記録される．閉じたファイル名はファイルを開いたイベントと一貫していなければならない．

図 3.8 はファイルを閉じたイベントを JSON に記録した例である．ファイルを閉じたイベントのタグは"type"に記録されており，値は固定の文字列"textDocument/didClose"である．編集されたファイル名は"body"内の"textDocument"内の"uri"に記録されている．ファイルを編集した時刻は"timestamp"に記録される．

現在の IDE によるエラー報告はコード編集に基づいてエラーを報告することから，コード編集を記録すればエラー報告を再現できる．現在の IDE によるエラー報告は言語サーバーによって提供されており，言語サーバーは解析すべきコードを上記のようなイベント列のデータ構造で受け取る．言語サーバーが受け取るのと同じデータ構造で記録したデータセットがあれば，エラー報告器に記録したデータセットを流し込むことで，エラー報告器はデータセットの記録時と同様のエラーを報告すると考えられる．

エラー報告それ自体ではなくコード編集を記録することで，記録時には使われなかったが解析すべきコードを同じように受け取るエラー報告器についても，報告されるエラーを観察することができる．これにより，複数のエラー報告器によるエラーを比較することができる．

```
1 {"range":[{"line":11,"character":27},{"line":11,"character":27}],"text
      ":"m"}
2 {"range":[{"line":11,"character":28},{"line":11,"character":28}],"text
      ":"e"}
3 {"range":[{"line":11,"character":29},{"line":11,"character":29}],"text
      ":"m"}
4 {"range":[{"line":11,"character":27},{"line":11,"character":30}],"text
      ":"member"}
```

図 3.9: 1 字ずつの入力と IDE による支援による入力で記録されるべきイベント列の例（抜粋）

キー入力ごとに記録する

コード編集をキー入力ごとに記録することで，十分細かい頻度でエラー報告を再現することができる．言語サーバーはコードの編集に応じて報告するエラーを更新するため，キー入力ごと以上に細かい頻度で報告するエラーを更新することはないと考えられる．

キー入力ごとに記録するにあたっては，1 字ずつ入力した場合は 1 字の入力に 1 回のイベントが対応し，IDE による支援で入力した場合は 1 回の支援に 1 回のイベントが対応する．図 3.9 は，`mem` を 1 字ずつ入力したのち，補完候補の `member` を選択した際に記録されるべきイベント列のうち，編集を表現する置換を抜粋したものの例である．1 つの行が 1 回のイベントに対応し，ここに挙げられていない置換は行われない．1 行目から 3 行目までは `mem` を 1 字ずつ入力した 3 回のキー入力にそれぞれ対応し，11 行目の 27 文字目に `m` を挿入するというような置換によって編集が表現されている．4 行目は補完候補の `member` を選択した 1 回のキー入力に対応し，ここまでで 11 行目の 27 文字目から 30 文字目までに入力した `mem` を `member` に置き換えるというように編集が表現されている．

コード編集をファイルごとにタイムスタンプをつけて記録することで，他のファイルの状態に依存するエラー報告を再現することができる．全てのイベントについてタイムスタンプによって全順序を構成することで，それぞれの時点における全てのファイルの状態を把握し，型エラーなどの複数ファイルにまたがるエラー報告を正しく再現する．

ロガーの実装

キー入力ごとのコード編集を記録するロガー textdocument-logger を，VS Code[*8]拡張機能として実装する．実装にあたっては，与える課題以外のコード編集のログデータを取得したりインターネットに送信したりすることがないよう注意する．

textdocument-logger は，多くの人が普段利用する UI で課題に取り組めるようにするため，VS Code 拡張機能として実装した．VS Code は現在一般的に用いられているテキストエディタであり，標準的な TypeScript の開発環境としての機能を持つ．

[*8] `https://code.visualstudio.com/`

イベントの種類	用いた VS Code API
ファイルを開いた	`vscode.workspace.onDidOpenTextDocument`
ファイルを編集した	`vscode.workspace.onDidChangeTextDocument`
ファイルを閉じた	`vscode.workspace.onDidCloseTextDocument`

表 3.2: イベントの種類とその取得のために用いた VS Code API の対応

キー入力ごとのコード編集データは，VS Code が拡張機能に対して提供する VS Code API[*9]によって取得した．イベントの種類とその取得のために用いた VS Code API の対応を表 3.2 に示す．これらの API はイベントリスナーであり，コールバックを登録すると VS Code 上でコード編集などがあったときにそれを実行する．いずれのイベントについても，対応する API は 1 回のコールバックの実行でちょうど 1 つのイベントを出力する．これらの API は TypeScript 以外のファイルについてもコールバックを実行するが，TypeScript 以外のファイルについては無視するように実装した．

textdocument-logger の実装にあたっては，被験者のプライバシーに障る情報が記録ないしインターネットに送信されないよう，複数の対策を行った．textdocument-logger は，VS Code 上の設定で明示的にオンにしない限りコード編集を記録しないように実装した．ワークスペースごとの VS Code の設定ファイルを提供することにより，提示した課題に取り組む時だけコード編集を記録する設定がオンとなる．これにより，課題に取り組んだ後ロガーのアンインストールが行われなくても，課題に取り組んでいない間のログが記録されない．textdocument-logger は，いかなる場合においてもインターネットと通信せず，ログを送信することも無いように実装した．これにより，他の実装不備によりログを取得してしまったとしても，その内容がメモリ上やファイルシステム上だけに封じ込められ，被験者の意図しないまま情報が公開されることはない．textdocument-logger は，ファイル名を保存する際にワークスペース内の相対パスだけが記録されホームディレクトリ名が記録されないために，実際の絶対パスではなく`/workspace/`から始まるパスを記録するように実装した．これにより，収集されるコード編集のログには匿名化された状態のデータが保存される．

3.1.4 収集されたコード編集の概要

実験により取得されたデータは，合わせて 33,252 件のイベントを含み，うち 16,543 件のイベントを分析することができ，残りは分析できなかった．これは，いくつかのイベントが記録できていなかったために，その時点のテキストを復元できず，それに続く全てのイベントの時点でのテキストが復元できないからである．イベントが記録できていなかったことは，イベントに付属するバージョンの比較によって判明した．

データは編集されたファイルごとに JSON Lines[*10]形式で記録された．ログファイ

[*9] https://code.visualstudio.com/api/references/vscode-api
[*10] https://jsonlines.org/

```
.
|-- .tdlog/
|    |-- 6-7-nested-else
|    |    |-- after.ts.jsonl
|    |    |-- before.ts.jsonl
|-- 6-7-nested-else
|    |-- after.ts
|    |-- before.ts
```

図 3.10: 編集されたファイルと対応するログファイルのワークスペース内での位置関係


```
1  {"type":"textDocument/didOpen","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","languageId":"typescript","version":1,"text":"export i
2  {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":2},"contentChanges":[{"range":[{"line":13,'
3  {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":3},"contentChanges":[{"range":[{"line":14,'
4  {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":4},"contentChanges":[{"range":[{"line":15,'
5  {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":5},"contentChanges":[{"range":[{"line":15,'
6  {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":6},"contentChanges":[{"range":[{"line":15,'
7  {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":7},"contentChanges":[{"range":[{"line":15,'
8  {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":8},"contentChanges":[{"range":[{"line":15,'
9  {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":9},"contentChanges":[{"range":[{"line":15,'
10 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":10},"contentChanges":[{"range":[{"line":15,
11 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":11},"contentChanges":[{"range":[{"line":15,
12 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":12},"contentChanges":[{"range":[{"line":15,
13 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":13},"contentChanges":[{"range":[{"line":15,
14 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":14},"contentChanges":[{"range":[{"line":15,
15 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":15},"contentChanges":[{"range":[{"line":15,
16 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":16},"contentChanges":[{"range":[{"line":15,
17 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":17},"contentChanges":[{"range":[{"line":15,
18 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":18},"contentChanges":[{"range":[{"line":15,
19 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":19},"contentChanges":[{"range":[{"line":15,
20 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":20},"contentChanges":[{"range":[{"line":15,
21 {"type":"textDocument/didChange","body":{"textDocument":{"uri":"file:///workspace/6-7-nested-else/before.ts","version":21},"contentChanges":[{"range":[{"line":15,
```


図 3.11: ログファイルの例

ルは，編集されたファイルのワークスペースとの相対パスの前に`.tdlog/`を，後ろに`.jsonl` を，それぞれ足した相対パスに保存された．編集されたファイルと対応するログファイルの位置関係を図 3.10 に示す．`6-7-nested-else/before.ts` の編集は`.tdlog/6-7-nested-else/before.ts.jsonl` に保存された．ログファイルは，空白を含まない JSON を 1 行に 1 つ含む JSON Lines 形式で記録された．ログファイルの例を図 3.11 に示す．1 行目にはファイルを開いたイベントを表す JSON 文字列が含まれている．2 行目には同じファイルを編集したイベントを表す JSON 文字列が含まれている．改行文字は `0x0A` (Line Feed) である．ファイル末尾は空行であり，改行文字で終わるので，処理するためには行ごとに分ける前に後端の空白文字を取り除く必要がある．なお，ファイル末尾以外に空行は存在しない．

ログファイルに含まれるイベント 1 つが満たす TypeScript の型定義を図 3.12 に示す．図の `TextDocumentEvent` がイベント 1 つに対応する．

全てのイベントは`"type"`，`"body"`，`"timestamp"`キーを持つ．`"type"`はイベントの形式ごとに異なる値を取る．ファイルを開いたイベント `TextDocumentDidOpen` においては固定の文字列`"textDocument/didOpen"`，ファイルを編集したイベント `TextDocumentDidChange` では固定の文字列`"textDocument/didChange"`，ファイルを閉じたイベント `TextDocumentDidClose` では固定の文字列`"textDocument/didClose"`となる．

イベントの処理にあたっては，多くの場合は最初に"type"の値によって処理を分岐する．

"body"キーはイベントの種類によって異なる型の値を持つが，いずれのイベントの型についても"textDocument"キーが存在し，その値は"uri"キーにファイルの識別子を含む．つまり，いずれのイベント event についても event.body.textDocument.uri とアクセスすることで必ずファイルの識別子である文字列を得ることができる．ファイルの識別子は"file:///workspace/"で始まりワークスペース内の相対パスが続く File URI[11]文字列である．例えば，6-7-nested-else/before.ts のイベントであれば"file:///workspace/6-7-nested-else/before.ts"となる．ファイルを開いたイベントとファイルを編集したイベントについては，"body"キーの値の"textDocument"キーの値に"version"キーが含まれる．"version"キーの値はファイルが開かれてから編集ごとに必ず増加する 1 以上の整数であり，最初にファイルを開いたイベントにおいては 1 となる．ファイルを開いて編集したのち閉じて再度開いた場合に，ファイルを開いたイベントで"version"が 1 より大きい場合もある．

ファイルを開いたイベントの"body"キーの値の型は TextDocumentDidOpen である．ファイルを開いたイベントの"body"キーの値は"textDocument"キーだけを持ち, その値は"uri", "version"キーのほかに"languageId"，"text"キーを持つ．"languageId"キーの値はファイルがどのプログラミング言語であるかを示し，ここで収集されるコード編集ログでは常に文字列"typescript"である．"text"キーの値はファイルを開いた時点におけるファイルの中身である．

ファイルを編集したイベントの"body"キーの値の型は TextDocumentDidChange である．ファイルを開いたイベントの"body"キーの値は"textDocument"キーと"contentChanges"キーを持つ．"textDocument"キーの値は"uri"と"version"だけである．"contentChanges"キーの値は，ファイル内の特定部分を新しいテキストに置き換えることを表す ContentChange 型の値の配列である．ContentChange 型の値は"range"，"text"，"rangeOffset"，"rangeLength"を持つ．"range"キーの値は 2 要素の配列であり，各要素は"line"キーと"character"キーを持ち，テキスト上の文字と文字の間の位置を表す．"line"キーは 1 以上の整数をとり，その位置の行番号を表す．"character"キーは 0 以上の整数を取り，行頭を 0 としてその位置が行内で何文字目かを表す．例えば，14 行目の行頭から 2 文字のインデント範囲を編集した場合，"range"キーの値は [{"line":14,"character":0},{"line":14,"character":2}] となる．"text"キーはその範囲を置き換えるテキストを表す．"rangeOffset"キーの値は 0 以上の整数で，その範囲がファイル冒頭から数えて何文字目かを示す．"rangeLength"キーの値は 0 以上の整数で，その範囲の文字数を示す．

ファイルを閉じたイベントの"body"キーの値の型は TextDocumentDidClose である．ファイルを閉じたイベントの"body"キーの値は"textDocument"キーだけを持ち，その値は"uri"キーだけを持つ．

[11] https://datatracker.ietf.org/doc/html/rfc8089

全てのイベントは"timestamp"キーを持つ．"timestamp"キーの値は正の整数で，そのイベントが記録された時刻を，協定標準時 1970 年 1 月 1 日 0 時 0 分から起算したミリ秒数（UNIX Timestamp）で表す．イベント間で"timestamp"キーの値の差をとることで，経過時間をミリ秒単位で取得できる．

3.2 望ましくないエラーが報告されるようなコード編集を抽出する

3.2.1 ログファイルに含まれるイベントから各時点のテキストを復元する

ログファイルに含まれるイベントから書く時点のテキストを復元するには，いくつか注意すべき事項が存在する．ログファイルに含まれるイベントは，時間順に並んでいないことがある．ログファイルを解釈するときは，イベントを"timestamp"キーの値の昇順に並べ替えてから解釈する必要がある．ログファイルに含まれるイベントから各時点のテキストを復元するためには，ファイルを開いたイベントで得られるテキストにファイルを編集したイベントで得られる置換を適用していく必要がある．すなわち，ContentChange を得られた順に解釈し，テキストの与えられた範囲を与えられたテキストで置換することでのみ，任意の時点のテキストを得ることができる．ログファイルに含まれるファイルを編集したイベントに欠けがある場合があり，それ以降のテキストは復元できない．これは，あるファイルを編集したイベントの後のテキストを得るためにはその前の時点のテキストが必要であることによる．イベントに欠けがあることは，"version"キーの値が 2 以上増加することで判別することができる．

ログファイルを解釈して各編集時点でのコードを得るための TypeScript プログラム例を図 3.13 に示す．load 関数をログファイルのパスとともに呼び出すことにより，各編集時点のコードが文字列になったものが時系列順に並んだ配列を得ることができる．このサンプルコードは 1 行目でインポートしている Node.js の組み込みモジュール node:fs[*12] と 2 行目でインポートしている npm パッケージ vscode-languageserver-textdocument[*13] に依存する．vscode-languageserver-textdocument は Node.js に依存しないため，ファイルシステムからファイルを読み出す部分を変更することにより，Node.js 以外の JavaScript/TypeScript ランタイムでも利用できる．3 行目から 6 行目で定義される関数 reviver は，イベントを表す JSON 文字列を解釈する際に範囲を示す部分を読み替えて出力するためのものである．この関数は JSON.parse 関数に渡され，JSON 文字列を解釈する際に"range"というキーの値が 2 要素の配列であれば{start,end}のようなオブジェクトに変換する．これにより，JSON 文字列を解釈して得られる値を後述する TextDocument.update が要求する引数の型に合わせられる．8 行目において，ログファイルを読み出して UTF-8 により文字列に変換している．これに続き，9 行目から 13 行目にかけてログファイルにあるイベント列を解釈して配列としている．10 行目と 11 行目において JSON Lines 形式であるログファイルを 1 つ 1 つの

[*12] https://nodejs.org/api/fs.html
[*13] https://www.npmjs.com/package/vscode-languageserver-textdocument

```
type TextDocumentDidOpen = {
  type: "textDocument/didOpen";
  body: {
    textDocument: {
      uri: string;
      languageId: string;
      version: number;
      text: string}};
  timestamp: number};
type ContentChange = {
  range: [
    { line: number; character: number },
    { line: number; character: number },
  ];
  text: string;
  rangeOffset: number;
  rangeLength: number};
type TextDocumentDidChange = {
  type: "textDocument/didChange";
  body: {
    textDocument: {
      uri: string;
      version: number};
    contentChanges: ContentChange[]};
  timestamp: number};
type TextDocumentDidClose = {
  type: "textDocument/didClose";
  body: {
    textDocument: {
      uri: string}};
  timestamp: number};
export type TextDocumentEvent =
  | TextDocumentDidOpen
  | TextDocumentDidChange
  | TextDocumentDidClose;
```

図 3.12: ログファイルに含まれる JSON 文字列が満たす型定義

JSON 文字列に分解している．`trimEnd` はファイル末尾の改行文字を削除するために必要である．12 行目では 1 つ 1 つの JSON 文字列オブジェクトに変換しており，その際 `reviver` を用いて範囲を示す部分を読み替えている．13 行目ではイベントを`"timestamp"`キーの値を用いて時系列順に並び替えている．15 行目において宣言される変数 `document` によってファイルの状態を追跡する．`TextDocument` は `vscode-languageserver-textdocument` に提供されており，`contentChanges` キーの値に応じてテキストを更新する処理を適切に行う機能を備えている．16 行目と 17 行目から始まる部分において，得られたイベントの列を時系列順に確認し，ファイルの内容を復元して `result` 変数に格納している．18 行目から 22 行目において，ファイルを開いたイベントの処理を行っている．19 行目においてイベントから読み出した値を元に，21 行目において `TextDocument.create` 関数によって作成した値でファイルの状態を置き換える．20 行目において，現在のファイルの状態とイベントの間でバージョンを比較し，同一であればそのイベントの処理を終了している．これは，同一のバージョンのテキストが `result` 変数に複数回格納されないようにするためである．23 行目から 35 行目において，ファイルを編集したイベントの処理を行っている．24 行目と 25 行目において，このイベント以前にファイルを開いたイベントがなければ処理を中断している．これは，ファイルの最初の状態がないとテキストを復元できないためである．27 行目から 32 行目にかけて，ファイルの状態とイベントの間でバージョンを比較し，その差に応じて処理を行っている．イベントのバージョンがファイルの状態より 2 以上大きい場合，イベントの欠損が疑われるためエラーとして中断している．イベントのバージョンがファイルの状態より小さい場合，イベントの順番が崩れているからエラーとして中断している．イベントのバージョンがファイルの状態と同じ場合，編集内容は存在しないからそのイベントの処理を終了している．これは，まれに空の編集が同じバージョンでログファイルに記録されているからである．33 行目でイベントから読み出した値を元に，34 行目において `TextDocument.update` 関数によってファイルに編集を適用している．

3.2.2　エラー報告の概要と分析

収集されたコード編集からデータセットに含める編集履歴を抽出するため，現在使われている TypeScript の言語サーバーによるエラー報告の分析を行い，抽出条件を定める．予備分析として，各時点でのテキストを復元したのち，TypeScript の言語サーバーにテキストを順次入力してエラー報告を収集し，経過時間に対するエラー報告数のグラフを作成する．抽出条件を定めるため，文法エラーが解決するまでの期間について編集範囲とエラー報告の位置に着目して分析し，抽出を行う．編集範囲から遠い文法エラーがあり，解決までの編集時間が 2 秒より長く編集数が 2 回より多い期間を抽出する．抽出された編集履歴は 90 個であった．分析や抽出に用いた Node.js のバージョンは `24.12.0` であった．

まず，収集されたコード編集ログから被験者が主に編集したファイルに対応するログファイルを抽出し，その各時点でのテキストを復元した．被験者が主に編集したログファイルを，リファクタリングを行う課題については 8 つの課題から 8 ファイル，機能追加と

```
1 import fs from "node:fs";
2 import {TextDocument} from "vscode-languageserver-textdocument";
3 function reviver(key: string, value: unknown): unknown {
4   if (key === "range" && Array.isArray(value) && value.length === 2)
5     return {start: value[0], end: value[1]};
6   else return value; }
7 export function load(path: string): string[] {
8   const eventsJsonl = fs.readFileSync(path, "utf-8");
9   const events = eventsJsonl
10     .trimEnd()
11     .split("\n")
12     .map((row) => JSON.parse(row, reviver))
13     .sort((a, b) => a.timestamp - b.timestamp);
14   const result: string[] = [];
15   let document: TextDocument | undefined;
16   for (const event of events)
17     switch (event.type) {
18       case "textDocument/didOpen": {
19         const {uri, languageId, version, text} = event.body.textDocument;
20         if (version === document?.version) continue;
21         document = TextDocument.create(uri, languageId, version, text);
22         result.push(document.getText()); }
23       case "textDocument/didChange": {
24         if (!document)
25           throw new Error("document not found");
26         const {version} = event.body.textdocument;
27         if (version >= document.version + 2)
28           throw new Error("version jumped")
29         if (version < document.version)
30           throw new Error("version decreased")
31         if (version === document.version)
32           continue;
33         const changes = event.body.contentChanges;
34         TextDocument.update(document, changes, version);
35         result.push(document.getText()); }}
36   return result; }
```

図 3.13: ログファイルを解釈して各編集時点でのコードを得るための TypeScript プログラム例

課題の種類	復元できたイベント数	取得されたイベント数
リファクタリングを行う課題	10455	19495
機能追加とデバッグを行う課題	6088	13757

表 3.3: 課題ごとの取得されたイベント数と復元できたイベント数

デバッグを行う課題については 1 ファイル，それぞれ抽出した．これらのファイルについて，3.2.1 項で説明した方法により，各時点のテキストを復元した．バージョンの欠損があるログファイルについては，その 8 割以上が復元できたと判断されるものについては復元できた範囲で分析し，復元できなかったものは取り除いた．使用した npm パッケージ `vscode-languageserver-textdocument` のバージョンは `1.0.12` であった．

課題ごとの取得されたイベント数と復元できたイベント数を表 3.3 に示す．リファクタリングを行う課題では，19,495 件のイベントを取得し，うち 10,455 件のイベントについてその時点のテキストを復元できた．機能追加とデバッグを行う課題では，13,757 件のイベントを取得し，うち 6,088 件のイベントを復元できた．合計で，33,252 件のイベントを取得し，うち 16,543 件のイベントを復元できた．復元できた割合は約 50% である．復元できた割合が極めて低く，原因の調査が必要だと考えられる．

次に予備分析として，TypeScript の Compiler API[*14]を利用して言語サーバーを作成し，そこ各時点のテキストを順次入力してエラー報告を収集した．言語サーバーは `ts.createLanguageService` および `ts.createDocumentRegistry` を用いて作成し，標準の型定義ファイルと収集されたコード編集ログから復元したファイルの情報だけを与えた．各時点について，`documentRegistry.updateDocument` によって新しいファイルの状態を登録したのち，`languageService.getSyntacticDiagnostics` によって文法エラー報告を，`languageService.getSemanticDiagnostics` によって文法以外のエラー報告を，それぞれ収集した．文法以外のエラーには，未定義変数のエラーや型エラーなどが含まれる．収集した報告のうち，`category` が `ts.DiagnosticCategory.Error` であるものだけを集計し，イベントの時刻とその時点のファイルのバージョンとともに記録した．TypeScript のバージョンは `5.8.3` であった．

記録したエラー報告数を被験者ごとに時系列に並べて，経過時間に対するエラー報告数のグラフを作成した．グラフの一覧を 3.14 に示す．それぞれのグラフについて，横軸は最初にファイルを開いてからの経過時間をミリ秒で示す．横軸の範囲は被験者によって異なる，縦軸はエラー報告数を示す．青い部分は `getSyntacticDiagnostics` によって得られた文法エラー数であり，赤い部分は `getSemanticDiagnostics` によって得られた文法以外のエラー数である．赤い部分は青い部分の上に積み上げて表示しており，赤い線は合計のエラー報告数を表す．縦軸の範囲は被験者によって異なる．バージョンの欠損があるログファイルに対応するファイルを編集していた間の記録は，分析できなかったためグラフに現れておらず，空白とな

[*14] `https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API`

るか省略されている．

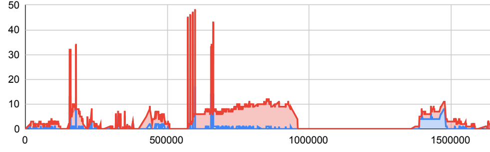
グラフから，エラーの最大数が被験者によってかなり異なることがわかる．2,3,5,10,13,14番の被験者は一度に報告されるエラー報告数が最大で20から50個であり，しかもそのような時点が複数存在する．一方，他の被験者は一度に報告されるエラー報告数が10個以下に収まっている．一度に報告されるエラー報告数が少ない被験者のうち4,8,9,12番の被験者は分析できていないログファイルが多く，本来は一度に報告されるエラー報告数の最大数がより大きい可能性がある．一方，一度に報告されるエラー報告数が少ない被験者のうち6,7,11番の被験者は分析できていないログファイルが少ない．よって，エラー報告数が多くなるような編集を行うことがある被験者と，エラー報告数が多くならないような編集を常に行う被験者という分類が可能であると考えられる．

グラフから文法エラーが存在する期間を読み取ることができるが，この期間が長い場合と短い場合があることがわかる．例えば11番の被験者のグラフに注目すると，1300秒付近から1500秒手前までの長い間文法エラーの数が1を下回っていないこと，それ以外の文法エラーのほとんどはすぐに解消していることが読み取れる．また，文法エラーの数が1を下回っていない間もエラー数が増減しており，コード編集が活発に行われていると考えられる．前から1字ずつ入力している際に構文が未完成であるために起きるエラーは，多くの場合で構文が完成してすぐに解消すると考えられるから，長い間続くことはない．この文法エラーは，ある文法エラーを解消せずに他の場所を編集しているために長い間続いているのではないかと予想される．

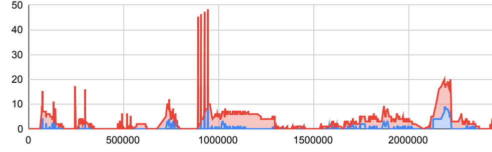
抽出条件を定めるにあたり，文法エラーが発生してから解決するまでの期間について編集範囲とエラー報告の位置に着目してさらに分析するため，文法エラーが解決するまでの期間のそれぞれについて幾つかの情報を集計した．各時点でのテキストを復元したのち，TypeScriptの言語サーバーにテキストを順次入力してエラー報告を収集した上で，文法エラーが発生してから解決するまでの期間の編集数と時間，そしてその間に編集範囲から遠い位置にエラー報告が行われたかどうかを集計した．

ある文法エラーについて，編集範囲と文法エラーが同じ行にあるか，編集範囲が行末にあって文法エラーが編集範囲の次の空でない行にあるとき，その文法エラーは編集範囲と近い位置にあるとする．ある文法エラーが編集範囲と近い位置にないとき，その文法エラーは編集範囲から遠い位置にあるとする．前から1字ずつ入力している際に構文が未完成であるために起きるエラーは，多くの場合で編集範囲と近い位置にあると考えられる．

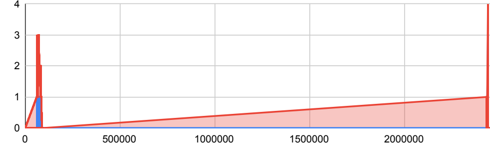
各時点で報告されるそれぞれの文法エラーについて編集範囲と近い位置にあるか遠い位置にあるかを判断するために，まずは編集範囲と近い行を列挙し，次にそれぞれのエラーの始点が編集範囲と近い行にあるかを見た．編集範囲と近い行を列挙するTypeScriptコードを図3.15に示す．ここでは，編集範囲以降の行頭であって編集範囲の終点からそこまでの全てのテキストが改行またはホワイトスペースであるもののうち最も下にある点を探索している．ここで，`document`は`vscode-languageserver-textdocument`が提供する`TextDocument`であり，`change`はファイルを編集したイベントに記録されている`"contentChanges"`の1要素である．`document`にはこのイベントの編集がすでに反映されている．`change.range.end`は



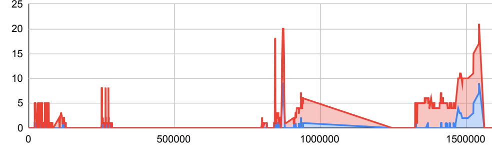
2 番の被験者



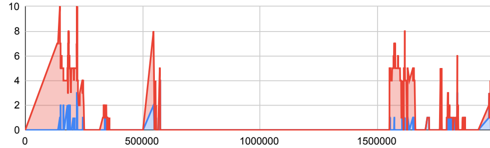
3 番の被験者



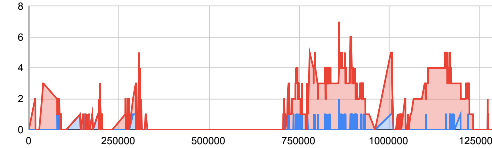
4 番の被験者



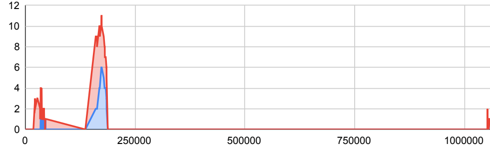
5 番の被験者



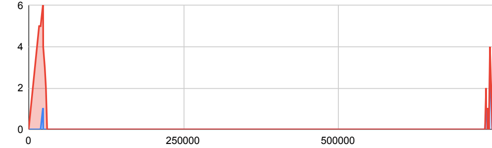
6 番の被験者



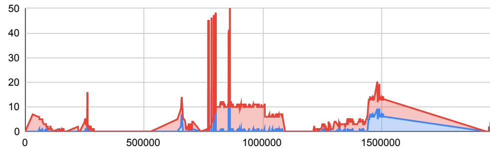
7 番の被験者



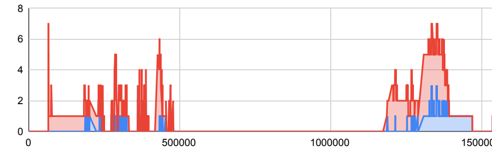
8 番の被験者



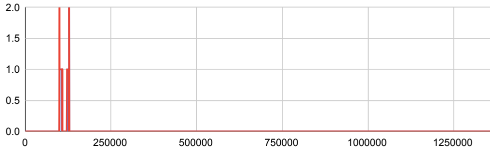
9 番の被験者



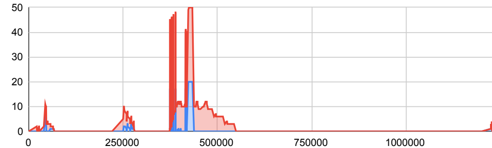
10 番の被験者



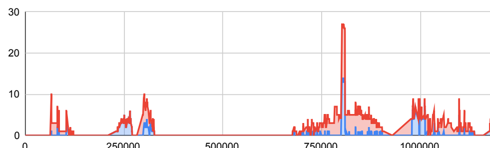
11 番の被験者



12 番の被験者



13 番の被験者



14 番の被験者

図 3.14: 被験者ごとの経過時間に対するエラー報告数のグラフ．横軸は経過時間をミリ秒で，縦軸はエラー報告数を示す．

```
1  const linesToConsiderNeighborhood = new Set<number>();
2  const startOffset = document.offsetAt(change.range.start);
3  const endPosition = document.positionAt(startOffset + change.text.length);
4  let neighborhoodUntilLine = endPosition.line;
5  while (neighborhoodUntilLine <= document.lineCount) {
6    const leadingText = document.getText({
7      start: endPosition,
8      end: {line: neighborhoodUntilLine + 1, character: 0}});
9    if (leadingText.trim()) break;
10   else neighborhoodUntilLine++; }
11 for (let line = change.range.start.line;
12      line <= neighborhoodUntilLine;
13      line++) linesToConsiderNeighborhood.add(line);
```

図 3.15: 編集範囲と近い行を列挙するための TypeScript コード

編集前の編集範囲の終点であるため，編集後の編集範囲の終点 `endPosition` を 2 行目から 3 行目において計算している．4 行目において宣言される `neighborhoodUntilLine` は最終的に，編集範囲が行末でなければ編集範囲の終点と同じ行を示し，編集範囲が行末であれば編集範囲の次の空でない行を示す．5 行目において，以降の処理を `neighborhoodUntilLine` がファイル末尾に到達するまで繰り返すようにしている．編集範囲の終点からファイル末尾までが空である場合，ファイル末尾にあるエラーは編集範囲に近いと考えられるため，その場合は `neighborhoodUntilLine` がファイル末尾を示した状態で停止してよい．6 行目から 8 行目において，編集範囲の終点から `neighborhoodUntilLine` が示す行の次の行頭までのテキスト `leadingText` を取得している．もし `leadingText` が全てホワイトスペースであれば，編集範囲は行末であり `neighborhoodUntilLine` までは全て空行であるので，さらに空行があるかを探索するために `neighborhoodUntilLine` を 1 増加させる．もし `leadingText` にホワイトスペース以外が含まれていれば，編集範囲が行末でないか，`neighborhoodUntilLine` が空でないので，`neighborhoodUntilLine` は最終的に必要な行を示している．11 行目から 13 行目において，編集範囲の始点の行から `neighborhoodUntilLine` が示す行までを編集範囲と近い行であると記録している．

文法エラーが発生してから解決するまでの期間のそれぞれについて，最初の文法エラーが発生した時点のファイルのバージョンと時刻，文法エラーが全て解決した時点のファイルのバージョンと時刻，その間に編集範囲から遠い位置の文法エラー報告が 1 つでもあったかどうかを集計した．課題ごと編集範囲から遠い位置の文法エラーの有無ごとの文法エラーが発生してから解決するまでの期間の数を表 3.4 に示す．リファクタリングを行う課題においては，編集範囲から遠い文法エラーを含む期間は 161 個，含まない期間は 556 個集計された．機能追加とデバッグを行う課題においては，編集範囲から遠い文法エラーを含む期間は 76 個，含まない期間は 357 個集計された．編集範囲から遠い文法エラーを含む期間は，リファクタリングを行

課題の種類	編集範囲から遠い文法エラーを含む期間の数	編集範囲から遠い文法エラーを含まない期間の数
リファクタリングを行う課題	161	556
機能追加とデバッグを行う課題	76	357

表 3.4: 課題ごと編集範囲から遠い位置の文法エラーの有無ごとの，文法エラーが発生してから解決するまでの期間の数

う課題については全体の 22%，機能追加とデバッグを行う課題については全体の 17% であった．これは，リファクタリングを行う課題が編集範囲から遠い文法エラーをより含みやすいことを示唆する．

編集範囲から遠い文法エラーがある場合とない場合について，文法エラーを解決するのにかかる編集時間と編集数の分布を見るヒストグラムを作成した．作成したヒストグラムを図 3.17 に示す．青いヒストグラムは編集範囲から遠い文法エラーがない場合のもの，赤いヒストグラムは編集範囲から遠い文法エラーがある場合のものである．編集時間のヒストグラムでは，横軸は文法エラーを解決するのにかかる編集時間，縦軸は度数を表す．度数は 1 秒ごとに集計している．文法エラーが発生してから解決するまでの期間のうち，編集時間が 10 秒を超えるものは示されていない．編集数のヒストグラムでは，横軸は文法エラーを解決するのにかかる編集数，縦軸は度数を表す．度数は 1 回ごとに集計している．文法エラーが発生してから解決するまでの期間のうち，編集数が 10 回を超えるものは示されていない．縦軸の範囲はグラフによって異なる．

ヒストグラムから，ほとんどの文法エラーは 2 秒以下で解決されること，ほとんどの文法エラーは編集数 3 回以下で解決されることがわかる．解決にかかる時間や編集数が少ないこれらの文法エラーは，前から 1 字ずつ入力している際に構文が未完成であるために起きるエラーであると考えられる．

ヒストグラムの形から，編集範囲から遠い文法エラーがある場合はない場合に比べて，解決にかかる時間が長い割合や解決にかかる編集数が多い割合が高いと読み取れる．ヒストグラムに示されていない，解決にかかる時間が 10 秒を超える期間は，編集範囲から遠い文法エラーがある場合が 36 個，ない場合が 15 個であり，編集範囲から遠い文法エラーがある場合が多い．さらに，ヒストグラムに示されていない，解決にかかる編集数が 10 回を超える期間は，編集範囲から遠い文法エラーがある場合が 49 個，ない場合が 13 個であり，こちらも編集範囲から遠い文法エラーがある場合の方が多い．これは，編集範囲から遠い文法エラーがない場合には現在編集している少しの内容だけによって文法エラーが発生しており，現在の編集が完了すればエラーは無くなることがほとんどであるということを示唆している．

最終的なデータとして，編集範囲から遠い文法エラーがあり，解決までの編集時間が 2 秒より長く編集数が 2 回より多い期間を抽出することとした，抽出の結果，90 個のデータが抽出された．これらの期間については，現在のエラー報告手法にある問題が発生しており，エラー

図 3.16: プログラマが文法エラーを放置して別の場所を編集していた例

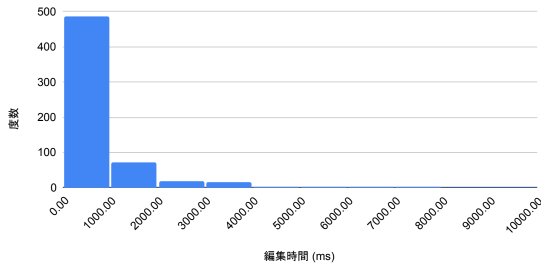
報告の評価や改善に有益なコード編集が記録されていると考えられる．

著者が 90 個の編集履歴の全数を読んで確認したところ，明らかに望ましくないエラーが含まれると考えられるものは 36 個であった．明らかに望ましくないエラーが含まれる編集履歴の割合は，リファクタリングを行う課題ではおよそ 7 割であり，機能追加とデバッグを行う課題では 1 割にとどまった．抽出されたにもかかわらず，明らかに望ましくないエラーが含まれると判断されなかった編集履歴では，コード上のある位置で文法エラーが発生したにもかかわらず望ましくないエラーが発生しなかったため，プログラマが文法エラーを放置して別の場所を編集していた場合が多く見られた．このような編集履歴は機能追加とデバッグを行う課題で特に多く見られた．

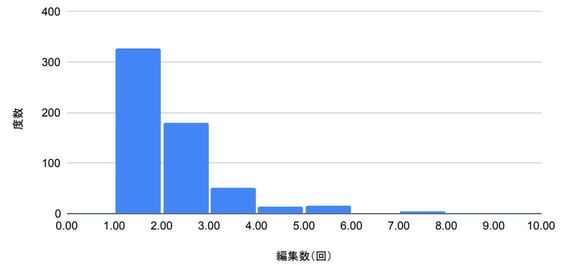
プログラマが文法エラーを放置して別の場所を編集していた編集履歴の例を図 3.16 に示す．ここでは編集履歴は機能追加とデバッグを行う課題において，プログラマが 19 行目の `switch` 文を記述している．図の 21 行目以降はあらかじめ用意されている部分である．ここでは，プログラマは `switch` 文が不完全であるために `return` において報告されている文法エラーを無視して，`switch` 文の分岐に用いる式 `body.type` を入力している．編集範囲の終点である編集中のカーソル位置は確かに行末ではない箇所に存在し，別の行に存在する `return` において報告される文法エラーは編集範囲から遠いと判断される．

3.3　データセットをもとにエラー報告を再現するビューワーの実装

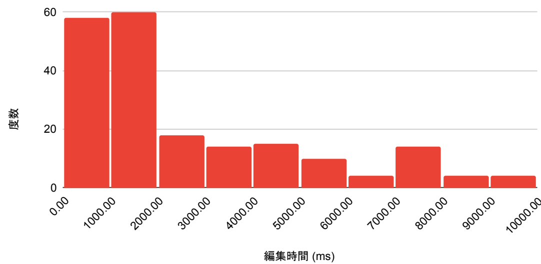
ファイルごとのキー入力ごとのコード編集を元に，編集された過程を粒度を損なわずに閲覧するためのビューワーを作成する．このビューワーでは，1 つのファイルに関するキー入力ごとのコード編集のデータを含むログファイルを選択し，VS Code 上で 1 イベントずつ進んだ



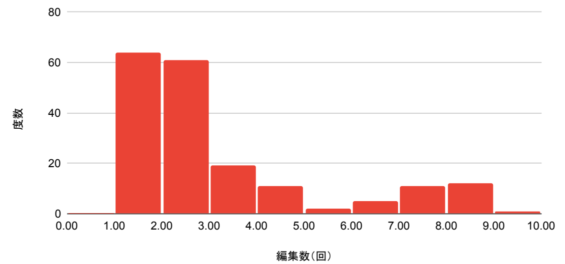
編集範囲から遠い文法エラーがない場合に
文法エラーを解決するのにかかる編集時間



編集範囲から遠い文法エラーがない場合に
文法エラーを解決するのにかかる編集数



編集範囲から遠い文法エラーがある場合に
文法エラーを解決するのにかかる編集時間



編集範囲から遠い文法エラーがある場合に
文法エラーを解決するのにかかる編集数

図 3.17: 編集範囲から遠い文法エラーがある場合とない場合について文法エラーを解決するのにかかる編集時間と編集数を示すヒストグラム

り戻ったりしながらコード編集の過程を閲覧できる．コード編集を VS Code 上で表示するため，ビューワーが動いている VS Code 上のエラー報告を共に観察することができる．ビューワーは VS Code 拡張機能として実装する．

ビューワーでは，以下に示す操作を行うことができる．

- 開いているログファイルを元に，コード編集を再現する新しいタブを開く
- 開いている再現中のタブで，次の編集に進む
- 開いている再現中のタブで，前の編集に戻る
- 開いている再現中のタブで，特定のバージョンまで早送りする
- 開いている再現中のタブを閉じる

このうち，次の編集に進む操作と前の編集に戻る操作にはショートカットキーが設定されている．次の編集に進む操作に割り当てられたショートカットキーを長押しすることで，連続して再現を進めることができる．また，特定のバージョンまで早送りする操作によって，分析によって得られた文法エラーが発生してから解決するまでの期間を素早く特定し，コード編集と

エラー報告を再現して観察することができる．

ビューワーは，再現すべきファイルを作成し，専用の VS Code API によってファイルを編集していくように実装した．ファイルを編集する API によって範囲を指定して指定する内容に置き換えることができるため，vscode-languageserver-textdocument を用いずにファイル編集を再現した．ここで，ファイルの編集を行わない間は編集を受け付けないモードにすることで，誤操作によるログの破損を防いだ．

ビューワーによって，コード編集とエラー報告を再現することができる．再現例として，リファクタリングを行う課題の 1 つである 6-7-nested-else に対する 13 番の被験者の回答の一部を抜粋して示す．この課題は，図 3.2 に示した編集を行う課題である．抜粋した部分は，文法エラーが発生してから解決するまでの期間である．この期間は，編集範囲から遠い文法エラーがあり，文法エラーの解決までに 9 秒にわたり，バージョン 24 からバージョン 39 までの 15 回の編集を要した．

バージョン 24 で行われた編集とその前後に報告されたエラーを図 3.18 に示す．この編集の前までに，被験者は if (hitPointRage < 0) および if (hitPointRage < 0.3) に関する書き換えをすでに終え，if (hitPointRage < 0.5) に関する書き換えに取り掛かった．編集前の時点では，currentHealthCondition に代入できないというエラーが 2 つに加え，返している値の型が関数宣言と異なるというエラーが報告されている．これらのエラーは，編集前に関数内にあったシャドーイングを伴う変数宣言 let currentHealthCondition: HealthCondition を削除したことによる．この編集において，被験者は 19 行目から始まる if 文の条件節に続くブロックの一部{currentHealthCondition =を選択し，r を打ち込んで置き換えている．編集後の時点では 6 つのエラーが報告されている．このうち，20 行目の else に対して報告されている文法エラーと，ファイル末尾の}に対して報告されている文法エラーと，13 行目の関数が値を返さない場合があり関数宣言と矛盾するというエラーは，20 行目冒頭の}で関数宣言が終了しているために報告されていると考えられる．残りのエラーのうち，19 行目の r というキーワードないし変数が期待されないという文法エラーと，19 行目の変数 r が存在しないというエラーは，現在その付近を編集しているために起きているエラーである．21 行目の currentHealthCondition に代入できないというエラーは，編集前からあったものである．ここで，19 行目にある r に報告される文法エラーは，編集範囲と近い位置にある．一方，20 行目にある else に報告される文法エラーや，ファイル末尾にある}に報告される文法エラーは，編集範囲と遠い位置にある．

バージョン 30 で行われた編集とその前後に報告されたエラーを図 3.19 に示す．バージョン 24 からこの編集の前までに，被験者は 19 行目においてキーワード return を最後まで入力した．この時点では，バージョン 24 では書きかけであったために報告されていた 19 行目の 2 つのエラーは解決し，それ以外の 4 つのエラーがそのまま報告されている．この編集において，被験者はもともと if 文の一部であった 20 行目の}else {の部分を行ごと選択し，削除している．編集後の時点でのエラー報告は，else に報告されていた文法エラーが解決したことを除いて変化していない．ここで，ファイル末尾にある}に報告される文法エラーは，依然編集範囲と遠い位置にある．

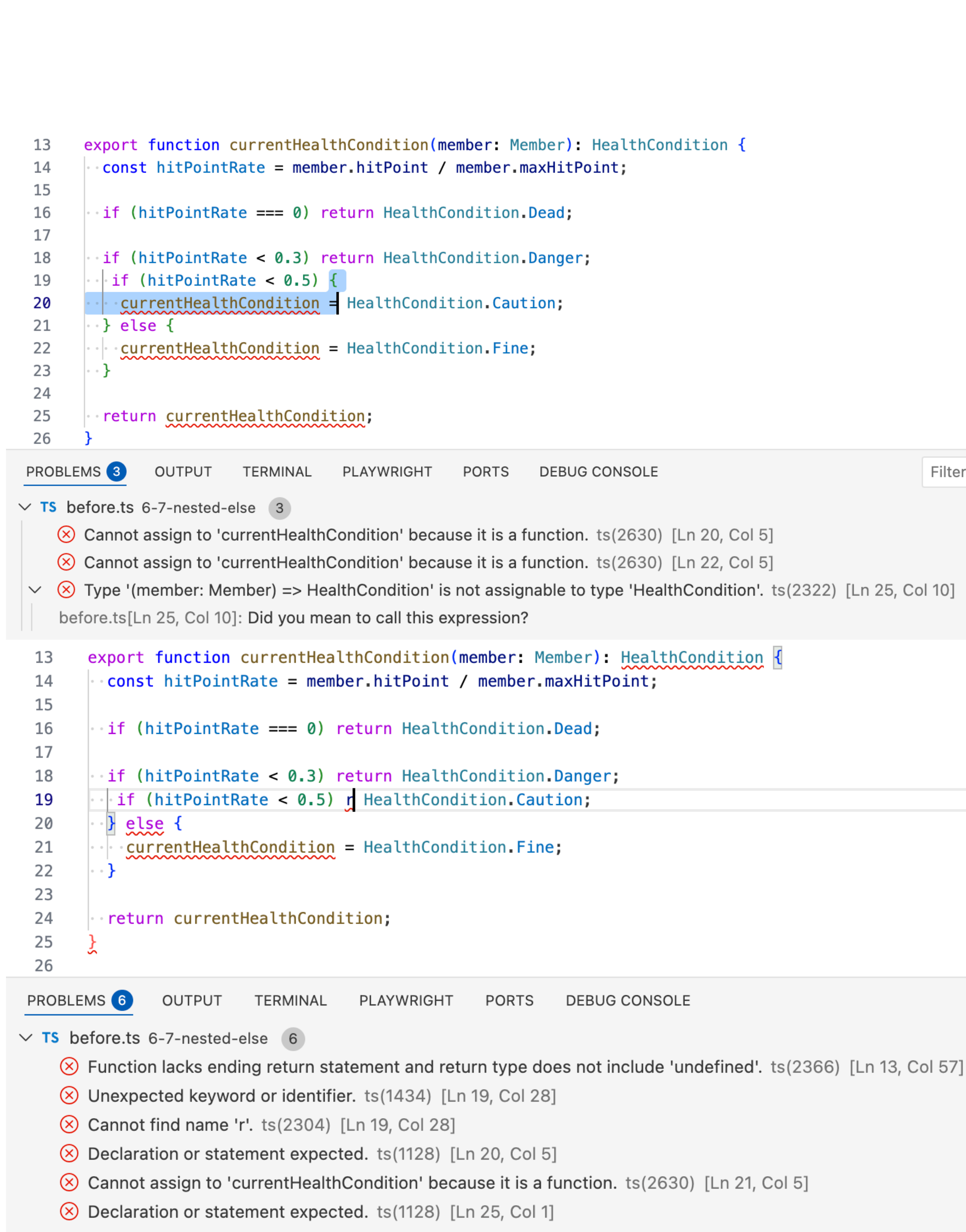


図 3.18: バージョン 24 で行われた編集とその前後に報告されたエラーを示すビューワー．上が編集前，下が編集後の状態を示す．編集に巻き込まれた範囲の選択は手動で行われた．

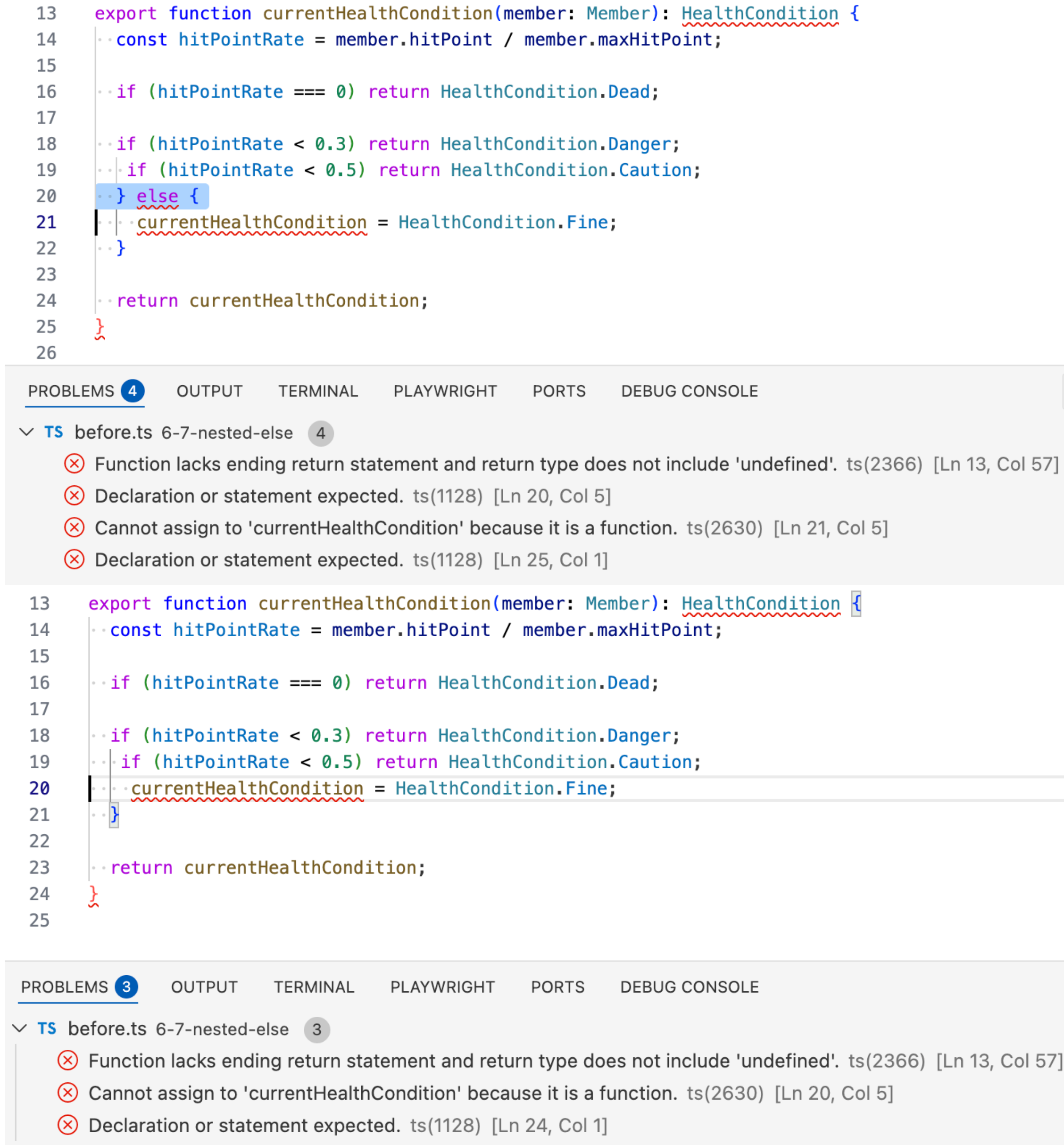


図 3.19: バージョン 30 で行われた編集とその前後に報告されたエラー．上が編集前，下が編集後の状態を示す．編集に巻き込まれた範囲の選択は手動で行われた．

バージョン 39 で行われた編集とその前後に報告されたエラーを図 3.20 に示す．バージョン 30 からこの編集の前までに，被験者は 20 行目にあった `currentHealthCondition =` を削除して `return` に置き換えている．この時点では，ファイル末尾にある}に報告される文法エラーだけが報告されている．この編集において，被験者はもともと `if` 文の末尾にあった}と，それに続く `return currentHealthCondition` を，まとめて削除している．編集後の時点で課題はホワイトスペースの違いを除いて完了しており，エラーは報告されていない．

```
13  export function currentHealthCondition(member: Member): HealthCondition {
14    const hitPointRate = member.hitPoint / member.maxHitPoint;
15
16    if (hitPointRate === 0) return HealthCondition.Dead;
17
18    if (hitPointRate < 0.3) return HealthCondition.Danger;
19     if (hitPointRate < 0.5) return HealthCondition.Caution;
20    return HealthCondition.Fine;
21    }
22
23    return currentHealthCondition;
24  }
25
```

```
PROBLEMS 1   OUTPUT   TERMINAL   PLAYWRIGHT   PORTS   DEBUG CONSOLE
TS before.ts 6-7-nested-else 1
  Declaration or statement expected. ts(1128) [Ln 24, Col 1]
```

```
13  export function currentHealthCondition(member: Member): HealthCondition {
14    const hitPointRate = member.hitPoint / member.maxHitPoint;
15
16    if (hitPointRate === 0) return HealthCondition.Dead;
17
18    if (hitPointRate < 0.3) return HealthCondition.Danger;
19     if (hitPointRate < 0.5) return HealthCondition.Caution;
20    return HealthCondition.Fine;
21  }
22
```

```
PROBLEMS   OUTPUT   TERMINAL   PLAYWRIGHT   PORTS   DEBUG CONSOLE
No problems have been detected in the workspace.
```

図 3.20: バージョン 39 で行われた編集とその前後に報告されたエラー．上が編集前，下が編集後の状態を示す．編集に巻き込まれた範囲の選択は手動で行われた．

第 4 章

まとめと今後の課題

4.1 まとめ

本研究では，IDE によるエラー報告の評価のために，エラーが報告されるようなコード編集履歴のデータセットを作成した．作成にあたっては，実験により細粒度のコード編集を収集し，そこからいくつかの条件によってエラーが報告される編集履歴を抽出した．本研究で作成したデータセットにより，エラー報告器の改良にあたり望ましくないエラー報告時の編集やエラー報告を細かく観察することができる．

作成にあたっては，まずキー入力ごとのコード編集を細粒度に記録しながら，リファクタリングや機能追加の課題を被験者に与えることで，コード編集を収集した．細粒度の記録を行うためのキーロガーの実装や，特にエラーが報告されるようなコード編集を期待する課題の作成を行った．実験では 13 人の被験者の協力を得て，3 万件を超えるイベントを収集することができた．

収集したコード編集から，編集範囲から遠い文法エラーがあり，解決までの編集時間が 2 秒より長く編集数が 2 回より多い編集履歴を抽出し，データセットとした．抽出により，望ましくないと考えられるエラーが報告されるような編集履歴を特に集めたデータセットを作成した．データセットには 90 個の編集履歴が含まれ，うち 36 個に本データセット作成の目的であった「望ましくないエラー」が含まれることを確認した．また，データセットに含まれる特定のファイルのコード編集を再現し，コード編集とそれに対する現在の IDE によるエラー報告を観察することができた．

4.2 今後の課題

本研究では，IDE によるエラー報告の改善のためにデータセットを作成し，エラー報告手法の評価が可能であることを確かめた．しかしながら，本研究ではエラー報告手法を実際に改善することはできていない．本研究の評価手法や，本研究で作成されたデータセットにより，エラー報告手法を実際に改善することが望まれる．

本研究では, TypeScript のエラー報告の改善を行うことを前提として, 被験者に TypeScript

の課題を課してデータセットを作成する手法を提案している．TypeScript 以外の言語についても，多様な編集を行う課題を設計してデータセットを作成することにより，エラー報告手法の評価や改善を行うことが望まれる．

参考文献

[1] Saul Rosen, Robert A. Spurgeon, and Joel K. Donnelly. Pufft—the purdue university fast fortran translator. *Commun. ACM*, Vol. 8, No. 11, p. 661–666, November 1965.

[2] Ioannis Karvelas. Investigating novice programmers' interaction with programming environments. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, p. 336–337, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, p. 724–734, New York, NY, USA, 2014. Association for Computing Machinery.

[4] Tao Dong and Kandarp Khandwala. The impact of "cosmetic" changes on the usability of error messages. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI EA '19, p. 1–6, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Titus Barik, Jim Witschey, Brittany Johnson, and Emerson Murphy-Hill. Compiler error notifications revisited: an interaction-first approach for helping developers more effectively comprehend and resolve error notifications. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, p. 536–539, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Tobias Kohn and Bill Manaris. Tell me what's wrong: A python ide with error messages. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, p. 1054–1060, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Sana Algaraibeh, Clinton Jeffery, Terence Soule, and Tonia Dousay. A parsing technique for enhancing compiler syntax error messages for student programmers. In *2024 IEEE Frontiers in Education Conference (FIE)*, pp. 1–7, 2024.

[8] Eddie Antonio Santos and Brett A. Becker. Not the silver bullet: Llm-enhanced programming error messages are ineffective in practice. In *Proceedings of the 2024 Conference on United Kingdom & Ireland Computing Education Research*, UKICER

'24, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Susan L. Graham and Steven P. Rhodes. Practical syntactic error recovery in compilers. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, p. 52–58, New York, NY, USA, 1973. Association for Computing Machinery.

[10] Lennart C.L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-lr parsing. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, p. 445–464, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats, and Eelco Visser. Natural and flexible error recovery for generated parsers. In Mark van den Brand, Dragan Gašević, and Jeff Gray, editors, *Software Language Engineering*, pp. 204–223, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[12] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, p. 177–210, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Lucas M. Layman, Laurie A. Williams, and Robert St. Amant. Mimec: intelligent user notification of faults in the eclipse ide. In *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '08, p. 73–76, New York, NY, USA, 2008. Association for Computing Machinery.

[14] Martin Schröer and Rainer Koschke. Recording, visualising and understanding developer programming behaviour. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 561–566, 2021.

[15] Tomoki Nakamaru, Tomomasa Matsunaga, and Tetsuro Yamazaki. Jupyter notebook activity dataset. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 581–585, 2025.

[16] Sebastian Proksch, Sven Amann, and Sarah Nadi. Enriched event streams: a general dataset for empirical studies on in-ide activities of software developers. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, p. 62–65, New York, NY, USA, 2018. Association for Computing Machinery.

[17] John Edwards, Kaden Hart, and Raj Shrestha. Review of csedm data and introduction of two public cs1 keystroke datasets. Vol. 15, No. 1, p. 1–31, 2023.

[18] 仙場大也. 良いコード／悪いコードで学ぶ設計入門. 技術評論社, 2022.

謝辞

本研究を進めるにあたり，研究方針の相談をはじめ論文執筆や研究発表のイロハについてもご指導いただきました千葉滋教授，本論文の構成や執筆について大いにご助力を賜りました山崎徹郎助教に，感謝を申し上げます．

本研究では，データセットの作成にあたり 13 名の被験者の方々にご協力いただきました．この場を借りてお礼申し上げます．