修士学位論文

題目

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

指導教員
井上 克郎 教授

報告者
石田 直人

令和 2 年 2 月 5 日

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 ソフトウェア工学講座

令和元年度 修士学位論文

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

石田 直人

内容梗概

ソフトウェアのコーディングプロセスを改善するには開発中にどのようなことが行われたかを記録・分析することが重要である．細粒度開発履歴はバージョン管理システムに記録される情報よりも豊かな履歴情報が記録されているため，詳細なコーディングプロセスの分析をするのに役立つ．しかしながら従来の細粒度開発履歴の研究において，エディタと履歴の記録手法と応用手法は依存関係が強かった．特にエディタと履歴の記録手法は依存関係が強く，開発者にとって利用可能な環境は限定的であった．

本研究では，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である KAMAKURA を提案する．細粒度編集履歴とは，細粒度開発履歴のうちコード変更を伴う履歴のことである．KAMAKURA は Language Server Protocol(LSP) を応用することで統一的な仕様で細粒度編集履歴をサーバーに送信する設計になっている．この設計により導入が容易で様々な目的に活用しやすい細粒度編集履歴収集プラットフォームを実現した．

加えて本研究では KAMAKURA を利用して，コードレビューツールである ReviewMan と開発進捗状況の可視化を行う Kudo の 2 つのアプリケーションを開発した．ReviewMan は細粒度編集履歴を利用して詳細な開発状況を再生する機能や共通した改善パターンを自動的に発見する機能により質の高いコードレビューを支援することを目的としている．Kudo は細粒度編集履歴を利用してコード編集の様子を可視化することで，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できることを目的としている．

ReviewMan のケーススタディでは，被験者に ReviewMan を使用してコードレビューを行ってもらった．有効なレビュー項目数，レビューにかかった時間についての有意な差は確認できなかったが，従来の Diff 環境と細粒度編集履歴を活用した環境を併用することで質の高いレビューが可能になることが分かった．Kudo のケーススタディでは，被験者にプログラミングの問題を解いてもらい，その過程を可視化した．その結果，開発者の体感的な進捗状況をよく表していることや，開発者や問題の難易度ごとの開発進捗状況の傾向を確認できた．

主な用語

細粒度編集履歴
コードレビュー
可視化

目 次

1　はじめに

ソフトウェアのコーディングプロセスを改善するには，開発中にどのようなことが起きたかを記録・分析することが重要である．しかし，詳細な分析を行うにはバージョン管理システムに記録される編集履歴ではそのリビジョンにおける差分しか分からず不十分である．そこで，より粒度の細かいデータを記録・活用する細粒度開発履歴の研究が盛んに行われている．細粒度開発履歴とは，開発者がソースコードを作成する際に発生する様々なイベントをすべて履歴として記録したものである．イベントにはエディタ上でのソースコードの編集操作だけでなく，メニュー項目をクリックする等してエディタが提供する機能 (例えばアンドゥ，ファイルオープン，ビルドの実行) を呼び出す操作も含まれる．ただし，記録されるイベントは実装によって異なる．

従来の細粒度開発履歴の研究において，記録手法はエディタに依存し，応用手法は記録手法に依存していた．すなわち，統一された細粒度開発履歴の記録方法が存在しないために開発者にとっては利用できる開発環境が限定されたり，労力をかけて蓄積した履歴データが活用しづらいという問題があった．

そこで，本研究では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である KAMAKURA を提案する．細粒度編集履歴とは，細粒度開発履歴のうちコード変更を伴う履歴のことである．KAMAKURA では Language Server Protocol(LSP)[6] を応用し，エディタ上で動作するプラグインから共通の仕様でサーバーに細粒度編集履歴を送信できる仕組みを実現した．これにより，エディタに依存することなく細粒度編集履歴を収集できるようになり，履歴データの活用が容易になる．すなわち，開発者にとっては好みの開発環境を使い続けながら細粒度編集履歴を記録することが出来るようになり，記録手法が統一されたことで記録された細粒度編集履歴を活用するアプリケーションの開発者はより多くの開発者が利用できるものの開発に集中できるようになる．

本研究では KAMAKURA で収集した履歴データを使用したコードレビューツールである ReviewMan と開発進捗状況の可視化を行う Kudo の 2 つのアプリケーションを開発した．ReviewMan は Diff モードという従来の Diff によるレビュー環境に加え，Fine-grained モードという細粒度編集履歴を使用して開発者がどのようにコードを書き進めたかを再生したり，改善パターンを自動的に発見したりする環境を提供する．Kudo は，大量の細粒度編集履歴を検査することなくどのようにコード編集が行われたのかを直感的に理解できるような可視化を行う．ReviewMan のケーススタディでは 4 名の被験者を対象として ReviewMan を使用してコードレビューをしてもらった．その結果，細粒度開発履歴を活用した Fine-grained モードが従来の Diff と併用することで質の高いコードレビューを支援することが分かった．Kudo のケーススタディでは 4 名の被験者を対象として 3 問ずつプログラミングコンテスト

の問題を解いてもらい，その過程を可視化した．その結果，開発者の体感的な進捗状況をよく表していることが明らかになり，また開発者ごとや問題ごとの開発進捗状況の傾向を分析することができることが分かった．

以降，2 章では本研究の背景について述べ，3 章では提案手法を説明する．4 では提案手法の適用について説明し，5 章ではケーススタディとその結果に対する考察を行う．6 章では関連研究を紹介し，7 章ではまとめと今後の課題を述べる．

2　背景

IT の発展とコーディングプロセスの改善の必要性

近年，IT が様々な産業で活用され，デジタルトランスフォーメーション [28] と呼ばれる概念が現実のものとなってきている．IT の発展に伴って，より一層のソフトウェアの生産性及び信頼性の向上が課題となっている．これらの課題の改善方法の一つにコーディングプロセスの改善がある．コーディングプロセスとは，V 字モデルにおける最も下流の工程でありコードを書いてコードレビューを行うという過程のことを指す．コーディングプロセスを改善するには，開発中にどのようなことが起きたかを蓄積・分析・活用することが重要である．開発者人口が増加していく状況 [34] の中で，コーディングプロセスの改善は一掃重要な課題となっていくと考えられる．

コーディングプロセスの把握についてはソースコードの進化の解明を目的とした研究の中で盛んに取り上げられている．多くの研究においては Git 等のバージョン管理システムに記録された情報を使用している．しかし，バージョン管理システムに記録される情報だけではコーディングプロセスの改善目的としては不十分な場合がある．なぜならば，バージョン管理システムにいつコミットを行うかは開発者の裁量に依存しており，また，Negara らは変更の 37%がバージョン管理システムに格納されることなく，追跡不可能になっていることを指摘している [21] からである．そこで，詳細な分析を可能に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も粒度の細かいデータを記録して活用する**細粒度開発履歴**の研究が盛んに行われている．

細粒度開発履歴

細粒度開発履歴に関連する研究は調査論文 [32] が詳しく，細粒度開発履歴の記録手法としては Robbes らの SpyWare[24]，Ebraert らの ChEOPS[17]，大森らの OperationRecorder[23] 等が提案されている．SpyWare は Squeak･SmallTalk，ChEOPS は Visual Works･SmallTalk，OperationRecorder は Eclipse･Java 向けでいずれも特定の IDE，特定のプログラミング言語を対象にして実装されている．一方で，StackOverflow Developer Survey 2019[9] の「Most Popular Development Environments」によると，開発者が使用している人気のエディタとして Visual Studio Code，Vim，Eclipse 等が挙げられており，また使用率が 10%を超えるエディタが 10 あることから現代では様々なエディタ，言語を使い分ける開発スタイルが一般的である．よって，エディタやプログラミング言語に依存しない細粒度開発履歴の記録手法が必要である．

また，細粒度開発履歴の応用手法として非常に多様な提案がされている．手法は「操作加工」「操作再生」「操作分析」「視覚化」「変更衝突検知」等に分類され，対象は「理解支

援」「変更支援」「ツール改善」「プロセス改善」「強調開発」「進化の解明」等に分類される[32]．Robbes らは，SpyWare を活用して メトリクス値の時系列変化を表示したり，変更点がいつどこで行われたかを容易に把握できるようにしたりした．Ebraert らは，ChEOPS により SmallTalk を対象に細粒度な変更を高度に記録することを可能にした．大森らは，OperationRecorder[23] を活用して編集操作を再生できる OperationReplayer[31] を提案している．このように，細粒度開発履歴の活用法は豊富に考えられる．しかし，既存の記録手法では開発環境が限定され，記録される情報の種類が編集操作のみのものやエディタの機能操作も含まれるものがあり，記録形式は XML やデータベース等があり統一されていないため労力をかけて蓄積した細粒度開発履歴データの活用が難しくなるという課題がある．

これらのことから細粒度開発履歴の記録手法は可能な限り統一させ，どのエディタ，どのプログラミング言語でも細粒度開発履歴データが自由に活用されていくような基盤を作る必要がある．

表 1: 本研究における細粒度開発履歴と細粒度編集履歴の定義

履歴の種類	説明
細粒度開発履歴	開発者がソースコードを作成する際に発生する様々なイベントをすべて履歴として記録したものである．イベントにはコード変更操作だけでなくエディタが提供するアンドゥ，ファイルオープン，ビルドの実行などの機能を呼び出す操作も含まれる．
細粒度編集履歴	細粒度開発履歴のうちコード変更を伴うイベントを履歴として記録したものである．履歴の最小単位は 1 文字単位の追加や削除である．その内，位置的に連続している編集は同時に起きたものとして 1 つの履歴に要約する．また，履歴は表 3 の形式で表す．

3 提案手法

2 章の背景を踏まえて本研究では細粒度編集履歴収集プラットフォームである KAMAKURA を提案する．本章では KAMAKURA の概要，動作原理，実装等について説明する．

3.1 KAMAKURA: 細粒度編集履歴収集プラットフォーム

従来の細粒度開発履歴の研究において，履歴の記録手法はエディタに依存し，履歴の応用手法は記録手法に依存していた．開発環境が多様化する現代においては，開発者が普段使っているエディタを維持したまま細粒度開発履歴を収集することが重要であると考えた．そこで，本研究で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 KAMAKURA を提案する．細粒度編集履歴の定義は表 1 に示す．KAMAKURA の設計思想は次の通りである．

1. 開発者は好みのエディタの使用を続けることができる (特定エディタに依存しない)
2. 他の開発者向けツールと容易に連携することができる

上記の設計思想を満たすため，KAMAKURA では Language Server Protocol(LSP)[6] のインクリメンタルにエディタ上のコード変更を取得できる仕組みを応用し，エディタ上で動作するプラグインから共通の仕様でサーバーに細粒度編集履歴を送信できる設計を採用している．また，KAMAKURA の概念図を図 1 に示す．GitHub[4] や CircleCI[2] 等の現代のクラウド型開発者向けサービスと相互に連携できるようにするために，データの集積や分析基

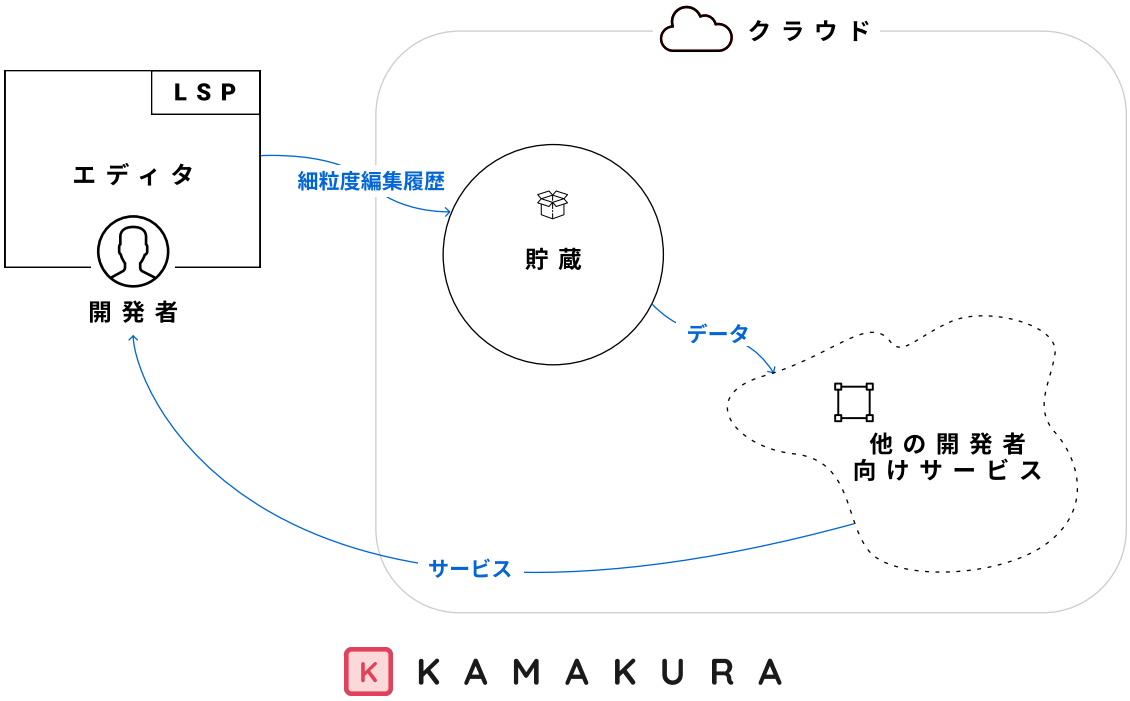

図 1: KAMAKURA の概念図．KAMAKURA はエディタから細粒度編集履歴データがクラウドに送信され，そのデータを活用して開発者にサービスが届けられるというサイクルを作り出す．

盤はローカルから独立したクラウド上で動作する．よって KAMAKURA は開発者の端末上で単体で動作するわけではなく，KAMAKURA で収集した細粒度編集履歴を活用したアプリケーションと連携して動作する仕組みになっている．

3.2 用語

本研究で使用する用語についてまとめる．

エディタ ソースコードを編集するために開発者が使用するソフトウェアである．統合開発環境 (Integrated Development Environment) もこれに含まれる．

編集操作 ソースコードを編集する操作である．人が手動で変更したものとエディタの機能で変更されたものは区別しない．

セッション 細粒度開発履歴の記録が開始してから終了するまでの期間，またはその記録内容である．

3.3 Language Server Protocol を使用した細粒度編集履歴の収集

KAMAKURA はその設計思想を踏まえて Language Server Protocol[6] (以後 LSP とする) を使用した細粒度編集履歴の収集方法を提案する．LSP とは，エディタと言語サーバー (「コード補完」や「定義へ移動」といったプログラミング言語機能を提供するもの) 間のやりとりを標準化したプロトコルである．元来 Visual Studio Code のために開発され，現在ではオープン標準となっており，多数の主要なエディタで LSP がサポートされている [7].

LSP を細粒度編集履歴の収集に使用した理由は以下の 2 点である．

1. 容易に特定エディタに依存せずに動作させることができる．
2. インクリメンタルにコードの変更履歴を取得する方法が存在する．

LSP の基本と KAMAKURA のための拡張

LSP ではエディタが言語サーバーと呼ばれるプログラミング言語機能を提供するサーバーへリクエストを送り，レスポンスを受け取ることで通信を行う．リクエストはヘッダー部とコンテント部から成り，コンテント部は JSON-RPC[5] 形式である．例えば，ファイルの内容が変更されたとき `textDocument/didChange` イベントの発生を通知し，エディタ側からリスト 1 のようなメッセージが送信される [3].

リスト 1: textDocument/didChange イベントのメッセージの例

```
1 Content−Length: ...\r\n
2 \r\n
3 {
4         ”jsonrpc”: ”2.0”,
5         ”id”: 1,
6         ”method”: ”textDocument/didChange”,
7         ”params”: {
8                 ...
```

```
9           }
10  }
```

リスト 1 中の `params` はリスト 2 に示すものである．`VersionedTextDocumentIdentifier` は「どのファイルのどの版か」という情報が含まれている．`TextDocumentContentChangeEvent` は,「ファイル中のどの部分がどのように書き換えられたか」という情報が含まれている．

リスト 2: textDocument/didChange イベントの params

```
1  interface DidChangeTextDocumentParams {
2          textDocument: VersionedTextDocumentIdentifier;
3          contentChanges: TextDocumentContentChangeEvent[];
4  }
```

LSP では通知頻度を設定することができ，本研究では細粒度編集履歴を取得するためにファイルが編集されるたびにイベントが発生するように設定している．この設定により言語サーバーでは編集内容をインクリメンタルに取得することができる．

また，LSP は実体が JSON-RPC なので拡張することができる．KAMAKURA では，履歴データの送信リクエストを行う `kamakura/saveEditHistory` を拡張定義している．このリクエストは KAMAKURA Plugin が発行し，受信した KAMAKURA Language Server がクラウドに履歴データを送信する．以後，この KAMAKURA のために拡張した LSP を **LSP-K** と呼ぶ．

3.4 エディタのプラグインと言語サーバーの実装

LSP-K を動作させるために，エディタ上で動作するプラグイン (KAMAKURA Plugin) とエディタのイベントを受け取り細粒度編集履歴として記録・送信する言語サーバー (KAMAKURA Language Server) をそれぞれ実装する．

3.4.1 KAMAKURA Plugin

KAMAKURA Plugin はエディタ上でファイルが編集されるたびに KAMAKURA Language Server にイベントを通知する．また，一定間隔で記録したデータの送信を依頼する．KAMAKURA Plugin は一度インストールすれば特に設定や操作は不要である．KAMAKURA Plugin の補助的な機能を表 2 に示す．細粒度編集履歴の収集にあたっては，その履歴データの扱いに十分注意する必要がある．例えば誤操作によって細粒度編集履歴に個人情報等の無関係な文字列の貼り付け等が発生してしまう場合がある [32]．KAMAKURA Plugin では自動的な履歴の送信を一時的に停止する機能や，データを暗号化して送信することでこの問題に対策している．

表 2: KAMAKURA Plugin の補助的な機能

機能	目的
履歴データの送信ログの確認	いつどのような履歴データが送信されたかを確認するため.
自動送信の停止，一時停止	プライベートな編集を行う場合や何らかの理由で履歴を送信したくない場合に停止するため.

本研究で実装した KAMAKURA Plugin は Visual Studio Code 向けのものである．スクリーンショットを図 2 に示す.

3.4.2 KAMAKURA Language Server

KAMAKURA Language Server は「1. 細粒度編集履歴の記録」と「2. 履歴の送信」を行う.

エディタ上で発生したファイルの編集操作は，全て LSP の `textDocument/didChange` リクエストを通じて言語サーバーに通知される．例えば，ファイルの 4 行目に`"int a = 10;"` と入力した場合リスト 3 ようなパラメーターを含むリクエストが通知される.

リスト 3: 編集操作履歴の例

```
1  Thu, Sep 06 2019 13:08:03.873 ADD [L4C0, ”i”]
2  Thu, Sep 06 2019 13:08:03.933  ADD  [L4C1, ”n”]
3  Thu, Sep 06 2019 13:08:03.989  ADD  [L4C2, ”t”]
4  Thu, Sep 06 2019 13:08:04.021  ADD  [L4C3, ”␣”]
5  Thu, Sep 06 2019 13:08:04.901  ADD  [L4C4, ”a”]
6  Thu, Sep 06 2019 13:08:05.127  ADD  [L4C5, ”␣”]
7  Thu, Sep 06 2019 13:08:05.200  ADD  [L4C6, ”=”]
8  Thu, Sep 06 2019 13:08:05.298  ADD  [L4C7, ”␣”]
9  Thu, Sep 06 2019 13:08:05.769  ADD  [L4C8, ”1”]
10 Thu, Sep 06 2019 13:08:06.570  ADD  [L4C9, ”0”]
11 Thu, Sep 06 2019 13:08:07.433  ADD  [L4C10, ”;”]
```

このように 1 文字編集されるたびに編集イベントが通知されるが，KAMAKURA Language Server ではこれをそのまま記録しているわけではない．理由は 2 つあり，第 1 に細粒度編集履歴はサイズが肥大化しやすいのでサイズが小さくなるように工夫が必要だからである．第 2 に履歴の粒度が細かすぎるとかえって履歴の意味の理解が難しくなるからである．これらの理由のため，関連研究 [29] では編集履歴 A と編集履歴 B の組を (挿入操作, 挿入操作，(挿入操作, 削除操作)，(削除操作, 挿入操作)，(削除操作, 削除操作) の 4 種類に分類して要約するという手法を取っており，本研究もこれを参考にして履歴の要約を実装している．リスト 3 の例の場合，全てが挿入操作で位置的に隣り合う編集操作のため全体が 1 つの履歴に要約される.

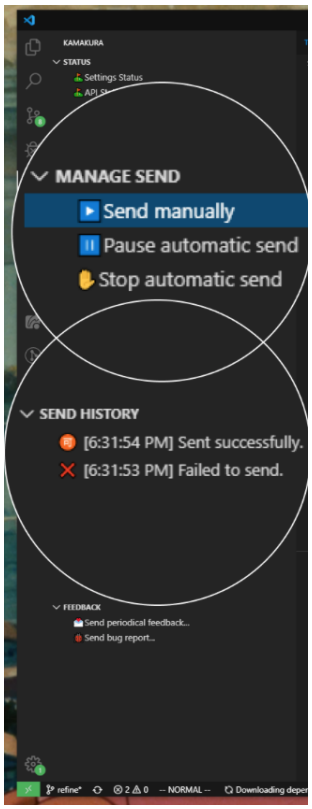

図 2: Visual Studio Code 上で動作する KAMAKURA Plugin

LSP では「挿入操作」「削除操作」「置換操作」を全て表 3 に示す形式のオブジェクトで表される.「挿入操作」「削除操作」「置換操作」を表す具体的なオブジェクトの例をリスト 4，5，6 に示す．関連研究 [29] の要約手法を使用するために,「置換操作」は「削除操作」と「挿入操作」の組に変換している.

リスト 4: 挿入操作の例．文字列”abc”を入力している．挿入操作では range.start と range.end が等しく，rangeLength が 0 となる.

```
{
  range: { start: { line: 0, character: 0 }, end: { line: 0, character: 0 } },
  rangeLength: 0,
  text: ”abc”
}
```

リスト 5: 削除操作の例．3 文字削除している．削除操作では text が空となる.

```
{
  range: { start: { line: 0, character: 0 }, end: { line: 0, character: 3 } },
  rangeLength: 3,
  text: ””
}
```

リスト 6: 置換操作の例．3 文字を文字列”def ”に置換している．置換操作は削除操作と挿入操作を組み合わせたものである．

```
{
  range: { start: { line: 0, character: 0 }, end: { line: 0, character: 3 } },
  rangeLength: 3,
  text: ”def”
}
```

表 3: LSP における編集操作の表現形式

プロパティ	型	説明
`range`	`Range`	置換される範囲．範囲は開始位置と終了位置の組で表し，位置は行位置と列位置の組で表す．
`rangeLength`	`number`	置換される文字列の長さ．
`text`	`string`	置換する文字列．

個々の編集履歴は要約される場合と要約されない場合があるが，いずれの場合でも何個の編集イベントから成るかを表す値を持ち，これを **Edit Effort** と呼ぶ．

KAMAKURA Language Server では記録した履歴の送信も行う．エディタから `kamakura/saveEditHistory` リクエストを受け取った際にクラウドに送信する．

記録形式

KAMAKURA では，セッションのデータを主にファイル内容と細粒度編集履歴のリストを JSON 形式で記述したものを Brotli[1] で圧縮した `.km` 形式で記録している．

3.5 KAMAKURA のデータを活用したアプリケーション

KAMAKURA により蓄積したデータを活用するには履歴を加工・保管し，用途に応じて分析や開発者への提示を行うようなアプリケーションを用意する必要がある．本研究では，4 章で説明する 2 つのアプリケーションを実装した．

4　提案手法の適用

本章では，共通した方法で収集した細粒度編集履歴データを活用して様々なサービスを作れる KAMAKURA の利便性を評価するために開発したコードレビューツールである ReviewMan と，開発進捗可視化手法である Kudo について述べる．

4.1　ReviewMan: 細粒度編集履歴を使用したコードレビューツール

KAMAKURA の細粒度編集履歴を活用したコードレビューツールである **ReviewMan** の背景，特徴，実装を説明する．

4.1.1　背景

Mozilla の開発者を対象にした調査において，コードレビューの質に影響を与える要素は開発者の経験，コードベースの理解度，個人的あるいは対人的資質，レビューの折の良さ，明確さ，徹底性であると指摘されている [20]．Bacchelli らは，変更が複雑に絡み合うものであるほどコードレビューの質を低下させ，より多くのレビュー時間を要すると指摘している [14]．また，Cisco による調査によると，コードレビュー対象の LOC(Lines of code) が増加するほど LOC あたりの欠陥発見率が減少すると指摘されている [13]．これらを踏まえてコードレビューの課題は，レビュー対象単位を小さくしたり，コードのニュアンスを理解できる手段を用意したりしてレビュアーの労力を省き，開発者の意図を理解したコードレビューに集中できるようにすることである．開発者の意図の理解が上手くできていないと，以下のような問題が発生すると指摘されている．

- 開発者の意図を理解した適切なフィードバックができない [26].
- 開発者の意図に反する否定的なフィードバックで開発者に不快感を与え，有用なレビューにならない [14][26].

コードレビューの質を向上させる方法として，Balachandran らは静的解析ツール等を使用した自動的な修正要求ボットを提案している．静的解析ツールによる解析をコードレビューに組み込むことで，共通した欠陥パターンやコーディング規約違反の検出を自動化し，レビュアーの生産性を向上できると指摘している [15]．梅川らは版管理システムへのコミットに複数の変更が含まれるのを防ぐために，細粒度作業履歴を分割，統合することで Task Level Commit[1] を支援する手法を提案している [35].

[1] 単一のタスクに関連する小さな変更ごとにコミットするべきという考え方.

本研究では，KAMAKURA で収集した細粒度編集履歴を使用したコードレビューツールである ReviewMan を提案する．ReviewMan は細粒度編集履歴を使用することで開発者が実際にどのようにコードを書き進めたのかをそのまま再生することができるので，より開発者の意図を理解したレビューを支援する．再生機能を使用することで全体のコードの変更量が多くても範囲を絞ってレビューすることができ，コードベースへの理解が少なくても順を追ってじっくりとレビューすることができる．関連して大森らは編集操作を時系列に沿って再生したり，過去のソースコードの任意の状態を復元したりすることができる OperationReplayer を提案している [31] が，これはコードレビューを目的としたものではない．また，ReviewMan は細粒度編集履歴を分析することで自動的に改善提案を行う機能も搭載している．

4.1.2 特徴

ReviewMan は KAMAKURA の細粒度編集履歴を利用したコードレビューツールであり，Web アプリケーションとして利用できる．

ReviewMan が提供する機能は以下の 3 点である．

1. 従来の Diff によるコードレビュー環境「**Diff モード**」
2. 細粒度編集履歴を用いたコードレビュー環境「**Fine-grained モード**」
3. 細粒度編集履歴の分析から自動的に知見を得る「**Insights**」

ユーザーが ReviewMan を利用するユーザーシナリオ (User Scenario) [2]は以下の通りである．

> コードレビューを任されたユーザーはまず Diff モードを用いてコードをチェックする．すると不可解なコードが発見されたので，Fine-grained モードを使用してそのコードがどのように書かれたかを確認し，開発者の意図を理解しようとする．Insights も参考にしながらレビューコメントを書き，コードレビューを完了させる．

[2]ユーザーがある製品を通じて何ら化の行動や目標を達成する架空のストーリー [11].

スクリーンショット

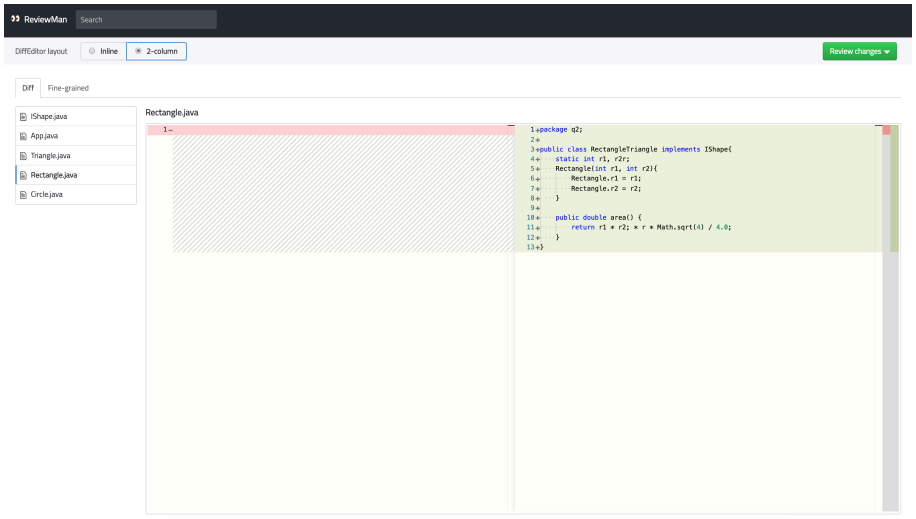

図 3: ReviewMan の Diff モード．GitHub[4] に似たインターフェースでレビューを行うことができる．

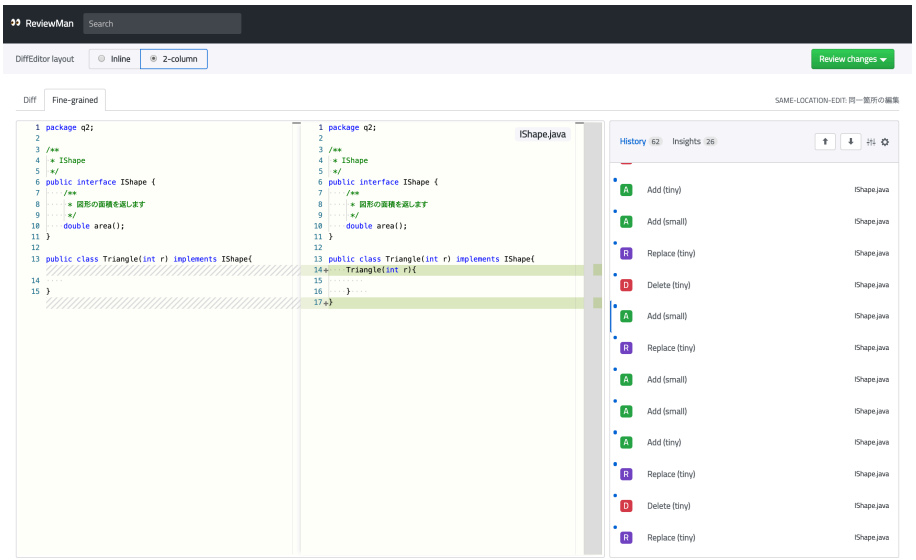

図 4: ReviewMan の Fine-grained モード．細粒度編集履歴を利用しているので，開発者がコードを書き進める様子を再生しながらレビューを行うことができる．右側の各編集アイテムにカーソルをホバーすると，その版のソースコードをプレビュー表示できる．また，編集アイテムをクリックするか J, K キーを押すと各編集アイテムに移動できる．

図 5: Insights の表示例

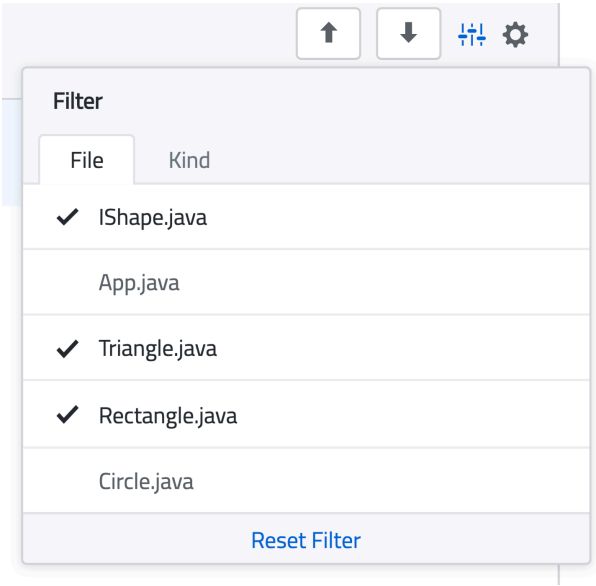

図 6: ReviewMan の Fine-grained モードにおける編集アイテムのフィルター機能．Fine-grained モードでは多くの編集アイテムが表示されるので，必要な編集アイテムに集中するためにファイルと編集アイテムの種類でフィルターを設定する機能がある．

4.1.3 実装

編集操作の再生

ReviewMan の Fine-grained モードでは，KAMAKURA で記録された細粒度編集履歴を使用する．KAMAKURA Language Server では，冗長すぎる編集履歴を 1 つにまとめる要約処理を行っているが，コードレビュー目的としては要約が不十分である．よって，Fine-grained モードでは追加の要約処理を行う．これを**後要約**と呼ぶ．

後要約における要約の方針は,「隣り合う時間的距離が近い編集操作は 1 つにまとめる」である．本実装では時間的距離の閾値を 1 秒とした．一方で空間的距離を使用する方法も考えられるが，文献 [36] において空間的距離を使用する方針は時間的距離を使用する方針よりも

表 4: Insights の種類

名称	説明
Global Insight	セッション全体に対する知見.
Local Insight	特定の編集アイテムに対する知見.
Insertable Insight	編集アイテムと編集アイテムの間に対する知見.

プログラム理解支援の効果が低いことが指摘されているため採用しなかった.

後要約後の個々の編集を**編集アイテム**と呼ぶ. そして, 編集アイテムがどのような編集であるかを簡易的に理解するために, 編集アイテムには編集の種類である「**追加**」「**削除**」「**置換**」のいずれかのラベルと, コード変更量の種類である「**tiny**(差分が 10 文字未満)」, 「**small**(差分が 100 文字未満)」,「**big**(差分が 300 文字未満)」,「**huge**(差分が 300 文字以上)」のいずれかのラベルを付与した. これらのラベルは, 編集アイテムに対して算出し付与している. また, これらのラベルを組み合わせたもの (例: 追加 (small)) を Fine-grained モードの各編集アイテムに表示している.

Insights

Insights とは, KAMAKURA で収集した細粒度編集履歴を分析することにより得られる知見を提示する機能である. Insights の目的は, 細粒度編集履歴から共通した改善パターンをプログラムにより自動的に発見することで大量に存在する編集アイテムを検査する労力を減らし, コードレビューを支援することである. Insights には「Global Insights」「Local Insight」「Insertable Insight」の 3 種類がある (表 4).

Insight を見つけるプログラムを **Insight-gazer** と呼ぶ. Insight-gazer は API が開放されており, コミュニティベースで誰でも開発できるように設計されている. これは, 開発者は自動的に検査したい項目があるとき自身で Insight-gazer を作ることができ, またそれを共有することで他の開発者も恩恵を預かることができるようにするためである. ReviewMan では必要な Insight-gazer を選択することができる (図 7). Insight-gazer は入力に編集アイテムのリストをとり, 出力に Insight のリストをとる. 本実装では 4 つの Insight-gazer を実装している (表 5). 以降, 実装済みの Insight-gazer のアルゴリズムを説明する.

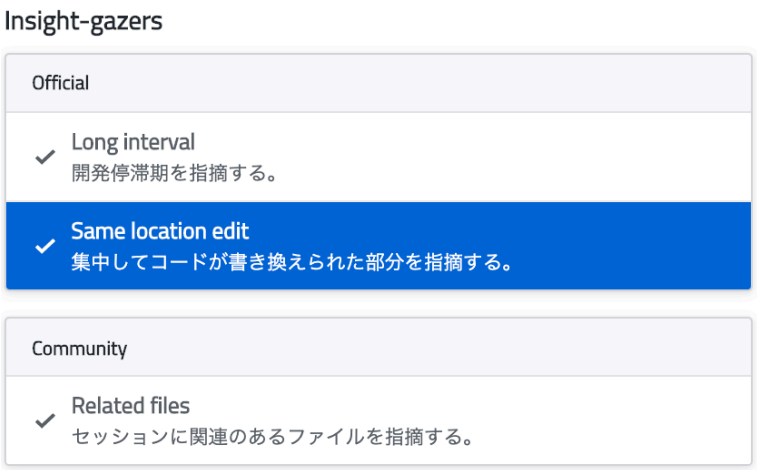

図 7: ReviewMan の Fine-grained モードの Insight-gazer の選択画面．ユーザーは必要な Insight-gazer を選択することができる．

Same location edit

同一ファイルの編集であり，かつ空間的距離が閾値 ($distanceThreshold$) 以下であるような編集アイテムが閾値 ($seriesLengthThreshold$) 以上連続する場合に同一箇所の連続した編集とみなして開発者に知らせる．本実装では $distanceThreshold = 2$ 行，$seriesLengthThreshold = 3$ としている．

Long interval

隣り合う編集アイテムの時刻の差が閾値 ($threshold$) 以上ならば開発停滞期とみなして開発者に知らせる．本実装では $threshold = 60$ 秒 としている．

Copy-paste

編集アイテムのコード変更量が閾値 ($threshold$) を超えていてかつ，変更量に対する Edit Effort が閾値 ($ratioThreshold$) 以下であるとき，ペーストされたコードとみなして開発者に知らせる．本実装では，$threshold = 5$ 文字，$ratioThreshold = 0.5$ としている．

Related files

セッションの中で編集されたり，開かれたファイルを関連するファイルとみなして開発者に知らせる．

表 5: 実装した Insight-gazer の種類

名称 (種類)	目的	メッセージの例
Same location edit (Local)	集中してコードが書き換えられた部分を指摘する.	同じ箇所の編集が連続しています. (開始位置: ID_3, 終了位置: ID_7)
Long interval (Insertable)	開発停滞期を指摘する.	開発停滞期. (1 分 18 秒)
Copy-paste (Local)	ペーストされた疑いのあるコードを指摘する. ペーストされたコードには不具合が存在する, コードクローン, ライセンス違反, 等様々な問題のあるコードである可能性がある.	ペーストされたコードの可能性があります.
Related files (Global)	セッションに関連のあるファイルを指摘する.	関連するファイルは以下の通りです. * Main.java * Solver.java * Prime.java

4.2 Kudo: 細粒度編集履歴を使用した開発進捗可視化手法

本章では, 細粒度編集履歴を使用した開発進捗可視化手法である **Kudo** の背景, 特徴, 実装を説明する.

4.2.1 背景

細粒度開発履歴の応用分野の 1 つに可視化がある. 大平らは, 定量的開発データ自動収集・分析システムである Empirical Project Monitor を開発し, ソースコードの累積ステップ数の推移や累積及び未解決な障害件数と平均障害滞留時間の可視化を行っている [22]. また, 藤原らは教員が学習者の取り組み状況を即座に把握することを目的としてソースコード行数, コメント行数, コンパイル回数, プログラム実行回数, エラー発生回数の可視化と特徴抽出を行っている [33].

細粒度開発履歴を可視化に応用する目的は, 多量になりがちな細粒度開発履歴が何を意味するかを直感的に理解しやすくするためである. 本研究では先行研究を参考にして KA-

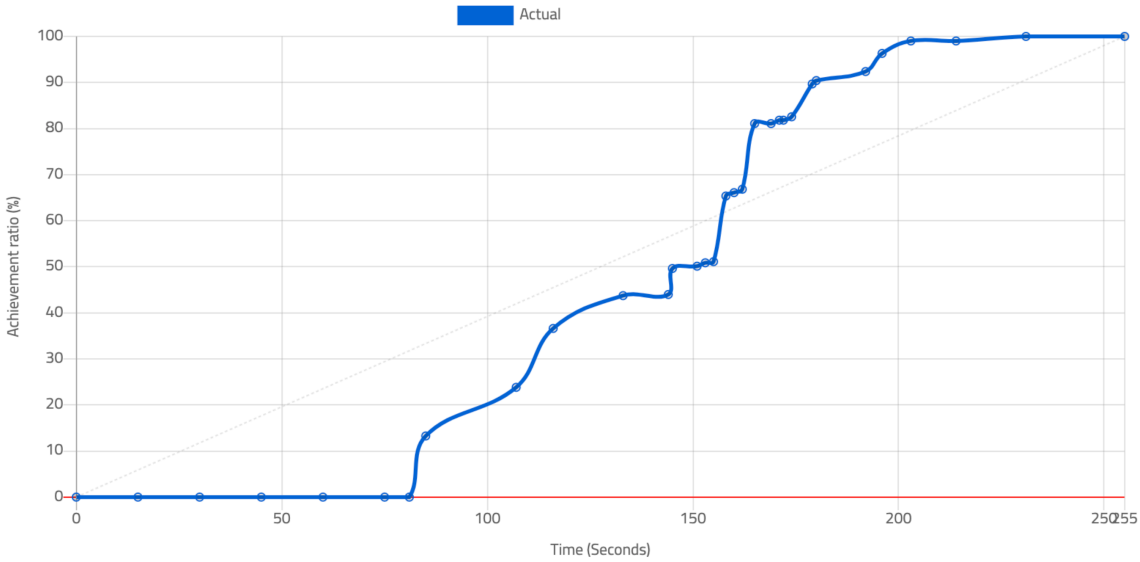

図 8: Kudo グラフの例

MAKURA を可視化に適用した．

4.2.2 特徴

Kudo はセッション中の開発進捗率の時系列変化を可視化する．Kudo ではこの開発進捗率を Achievement Ratio と呼ぶ．開発進捗率を可視化する目的は，細粒度編集履歴を使用していつどのようにコードに変化が起きたかを直観的に理解することで開発者の行動・傾向を理解したりコードレビュー時に役立てたりすることである．Kudo グラフの例を図 8 に示す．背景で触れた先行研究 [33] ではコード変化量の時系列変化の可視化を行うが，Kudo ではコード変化量ではなく Achievement Ratio を使用する点が異なる．時刻 t における Achivemet Ratio は，最終状態のファイルの内容と時刻 t でのファイルの内容との文字単位のレーベンシュタイン距離を用いて式 1 のように定義する．

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

A_t は時刻 t における Achievement Ratio である．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る．D は距離関数である．本実装では距離関

数に文字単位のレーベンシュタイン距離 [8] を使用している．レーベンシュタイン距離を使用することの妥当性はケーススタディで検証する．また，Achievement Ratio は $t =$ セッション開始時刻 のとき 0 になり，$t =$ セッション終了時刻 のとき 1 になる．Achievement Ratio が負値になるのはある版の最終版との距離が初期版と最終版との距離よりも遠い場合であり，例えばセッション中で，あるクラスを不要と判断して一度消去したものの，再び必要なことが分かり再度作成した場合である．

4.2.3 実装

Kudo グラフではセッションの開始時刻を基準とした時間スケールを採用している．なぜならば，セッション開始時から最初のコード編集が行われるまでの停滞期間も Kudo グラフの形状に影響を与える重要な要素だからである．停滞期間が長い場合，Kudo グラフは等間隔で同じ Achievement Ratio の値を持つ点を挿入するようにしている．これは，実際には一定時間編集操作が無かったにも関わらず長時間をかけて編集が行われたように見えてしまうのを防ぐためである．本実装では，30 秒以上の停滞期間に対し 15 秒間隔の定点を追加している．また，Kudo グラフはブラウザで閲覧する．

Achievement Ratio は式 (1) を使用して求めており，各版における距離は文字単位のレーベンシュタイン距離を使用するしている．レーベンシュタイン距離の計算時間は長さ n と長さ m の文字列に対して $\mathcal{O}(mn)$ であるが，この計算コストは大量の文字列比較を行う場合に無視できないほどのコストになる．計算コストを小さくするには，1. n-gram アルゴリズム [19] を使用する，2. C++等の高速な実装を使用する，3. 細粒度編集履歴を使用する等の工夫が考えられる．

5　ケーススタディ

ケーススタディでは，KAMAKURA で収集した細粒度編集履歴を使用した 2 つのアプリケーションを通して KAMAKURA の有用性を検証する．被験者の Java の能力レベルの分類は表 6 に定義したもので行った．

5.1　ReviewMan

ReviewMan のケーススタディでは準備実験と本実験の 2 種類の実験を行った．簡潔に言うと，準備実験とは本実験で使用するコードレビュー素材を作成するもので，本実験とは ReviewMan を使用してコードレビューを行うというものである．以下ではそれぞれの実験について詳しく説明する．

準備実験 (コードレビュー素材の作成)

準備実験の目的は，本実験で使用するコードレビュー素材を作成することである．具体的には被験者 1 名に Java の問題を 3 問解くというタスクを課した．被験者には Java の経験年数 1 年，能力レベル 1 の情報科学研究科の学生を選んだ．使用した問題を表 7 に示す．各問題について簡単に説明する．問題 1 は容易な問題であり，つまずくことなく実装を完了できると予想される．問題 2 は難解な問題であり，コーディングを始めるまでに設計が必要で開発者の意図を理解していないとソースコードも読み取りづらいものになると予想される．問題 3 はやや容易な問題であり，段階的に機能を追加していくようなコーディングプロセスになると予想される．

その他の実験条件を示す．コーディング用のエディタは Visual Studio Code[12] を使用した．このエディタは，コード補完や定義へ移動等の機能が利用できる．問題回答時間は 10 分を目安とし，最大 20 分で終了することとした．コードの動作確認ができるように，テストコードを予め用意した．コーディング中の注意として，できるだけ機能のまとまりを細分化して実装を進め，テストが通ったらリファクタリングを行うように指示をした．なぜなら，コーディングスタイルが被験者に依存しないようにするためである．なお，準備実験は本実験のための準備のため結果の考察等は特に行っていない．

本実験 (ReviewMan を使用したコードレビュー)

本実験の目的は，ReviewMan を使ってコードレビューを行い ReviewMan の Fine-grained モードをどのように使用するのが効果的であるかを検証することである．具体的には被験者 4 名に準備実験で作成した Java の問題の回答 3 件を ReviewMan を使ってコードレビューするというタスクを課した．各問題には ReviewMan の Diff モード，Fine-grained モード，両

表 6: Java の能力レベル表

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．

方のいずれかを使用する．各問題と使用するモードの対応パターンを表 9 に示す．被験者の 3 名をそれぞれのパターンに割り当て，残りの 1 名をパターン A に割り当てた．被験者には，Java の経験年数 3 年以上，能力レベル 2 以上の情報科学研究科の学生を選んだ．本実験の被験者には準備実験の被験者は含まれない．回答時間は 10 分を目安とし，最大 20 分で終了することとした．また，レビュー方針を統一するために，実験開始前に被験者には表 8 に示すコードレビュー方針を熟読して実践するように指示した．

評価方法

ケーススタディでは「1. 有効なレビューコメント項目数」,「2. レビュー時間」「3. System Usability Scale(SUS)[10]」「4. フリーコメント」「5. 各モードをどのように使用するのが効果的か」の 5 点を調査する．有効なレビューコメント項目数は，Diff モードと Fine-grained モードの間でレビューコメント数にどのような違いがあるかを調査する．被験者が書いたレビューコメントは著者が手動で個別の項目に分解し，カテゴリ，Fine-grained モード特有か，感情 (肯定的，否定的，どちらでもない) の分類も行った．レビュー時間は，Fine-grained モードの方が Diff モードよりも検査する箇所が多いため余計に時間がかかると予想されるのでどのくらい時間が余分にかかるかを調査する．System Usability Scale(SUS)[10] とは，10 個のシンプルな質問により使いやすさの評価を行う方法であり，ReviewMan のツールとしての使いやすさを調査するために使用する．フリーコメントは ReviewMan に関する自由な感想や改善点を定性的に調査する．ReviewMan の各モードをどのように使用するのかが効果的かは他の評価軸を踏まえて検討する．

表 7: ReviewMan のケーススタディで使用した問題

問題番号	タイトル	難易度	内容
問題 1	Triangle Rectangle Circle	低	正三角形を表す `Triangle` クラス，長方形を表す `Rectangle` クラス，円を表す `Circle` クラスを作成せよ．ただしこれらのクラスは面積を返す `area` メソッドを含む `IShape` インターフェースを実装していること．
問題 2	What is the bottom most	高	一番上の行の 10 個の整数を読み込み，隣り合う数を足した数の 1 の位の数を下段に書くという規則に従うとき，一番下の行の数を出力するプログラムを作成せよ．ただし，再帰関数を使用するように努めること．(`http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=0062`)
問題 3	Cafe order	中	カフェの 3 種類のメニューと 3 種類のサイズから代金を計算せよ．ただし，それぞれおまかせを選択できるものとする．

結果の仮説

有効なレビュー項目数は，Fine-grained モードの方がより詳細な開発履歴を見られるため常に多くなると予想した．

レビュー時間は,「Diff モード」＜「両方」＜「Fine-grained モード」になると予想した．Diff モードよりも Fine-grained モードの方が時間がかかるのは，より検査するべき内容が多いからである．両方が Fine-grained モードよりも時間がかからないのは，Diff モードを併用して検査できるからである．また，問題の難易度が低いほど Fine-grained モードの方が余計に時間がかかると予想した．一方で，問題の難易度が高いほど Fine-grained モードの方が開発者の意図が分かりやすいという利点が活きてくるため，時間差が小さくなると予想した．

System Usability Scale(SUS) では SUS スコアが 80.3 以上で Grade A(Excellent)，68〜80.3 で Grade B(Good)，68 で Grade C(Okay) とされている．よって SUS スコアが 80.3 以上になることを期待する．

表 8: コードレビュー項目の分類

項目	概要	具体例
褒める	よいコードは褒めるべきである.	「3 行目，ちゃんとファイルクローズできていますね.」「再帰関数をうまく使っていますね．しかも末尾再帰になっていてよいです.」
可読性	読みやすさを向上させるべきである.	「10 行目の条件式がとても長くなっているので，変数に切り分けましょう.」「関数名がわかりづらいです.」
メンテナンス性	関心の分離，再利用性を意識するべきである.	「顧客管理の部分は Customer クラス等に切り分けて再利用できるようにしましょう.」
欠陥	エラーは必ず指摘するべきである.	「入力が負値の場合に例外が発生してしまうので値チェックを加えてください.」
プログラミング言語	言語のスタイルを踏襲するべきである.	「配列の初期化には Initializer を使用した方が Java らしくてよいです.」
不明点	コードの意図が分からない場合は質問するべきである.	「8 行目 10 行目のログ出力はなんのためのものですか？」
その他	上記項目に当てはまらないコメント.	N/A

実験結果

有効なレビュー項目数を表 10 に示す．また，全体の項目数は 47 件であり，そのうち Fine-grained モードでしか成し得ないコメントは 10 件であった．コードレビュー時間を表 11 に示す．SUS の集計結果を表 12 に示す．SUS スコアの平均は 75.6 で，Grade は B(Good) であった．また，フリーコメントを肯定的，否定的，その他の 3 種類に分けて以下に示す．

肯定的なフリーコメント

- Diff モードだとどのようなコーディングプロセスで実装したのか (一から実装もしくは転用と加筆修正か) が分からないが，Fine-grained モードだとその部分までわかるので，**より効率の良い開発スタイルにつながるようなアドバイスができるのではないかと思った.**
- Diff モードと Fine-grained モードを使っていると**初めに全体の変更を把握するときは Diff モード，一つのファイルを詳細に調べたいときは Fine-grained モードという使い分けが有効に感じた.**

表 9: ReviewMan 本実験の割り当てパターン

	Diff モード	**Fine-grained モード**	両方
パターン A	問題 1	問題 2	問題 3
パターン B	問題 2	問題 3	問題 1
パターン C	問題 3	問題 1	問題 2

表 10: 有効なレビューコメント項目数

	Diff	**Fine-grained**	両方
問題 1	3, 5	8	4
問題 2	3	3, 3	3
問題 3	4	4	3, 4

- Fine-grained モードで編集アイテムをファイル単位でフィルターできる機能があるのは非常に使いやすかった.
- 細粒度編集履歴を用いることで開発者が実装しようとして諦めた様子が見られたため，教育の観点から開発者の技術的な問題の発見に良いと感じた.
- **Diff** モードのみでは見つけられない開発者の様子が確認できる点はツールとして非常に良いと思う.

否定的なフリーコメント

- **Fine-grained** モードだけでレビューすると強いストレスを感じた．最初に **Diff** モードで全体像を掴んで，レビュー中に疑問に思った部分だけ **Fine-grained** モードを使うのが自分に合っていると思った.
- 特に，複数のファイルを編集したときや，何度も挿入や削除を繰り返す編集を行った時は初めから **Fine-grained** モードだと動きを追いかけづらいように思う.

その他のフリーコメント

- Diff モードと Fine-grained モードを複合的に使うなら，Fine-grained モードの方で最終状態とのその時点での Diff を出せるようにするとレビューできることが増えると思う.
- Diff モードと Fine-grained モードの中間としてコミット単位で Diff を出すような機能があるとよいかもしれない.

結果の考察

有効なレビュー項目数については，特に有意な差は見られなかった．しかしながら，すべての被験者が Fine-grained モードでないと気づけないような，例えばあるクラスをコピー

表 11: レビュー時間の結果 (単位: 秒)

	Diff	**Fine-grained**	両方
問題 1	619, 1200	927	1200
問題 2	1200	1200, 1225	981
問題 3	803	1200	1081, 1174

表 12: ReviewMan の System Usability Scale の結果

	問 1	問 2	問 3	問 4	問 5	問 6	問 7	問 8	問 9	問 10	SUS Score	Grade	Adjective Rating
被験者 A	4	3	3	2	5	4	3	2	3	5	55	**D**	**Poor**
被験者 B	4	1	5	1	5	1	5	4	4	1	87.5	**A**	**Excellent**
被験者 C	4	3	4	1	5	1	5	4	5	4	75	**B**	**Good**
被験者 D	4	2	4	1	5	1	4	2	4	1	85	**A**	**Excellent**
平均	4	2.3	4	1.3	5	1.8	4.3	3	4	2.8	75.6	**B**	**Good**

して他のクラスに作り変えている部分でシンボル名を変更する時はエディタの一括置換機能を使った方がよいといった指摘をすることができていた．

レビュー時間については，被験者中 2 名がすべての問題についてほぼ制限時間いっぱいを費やしたため，比較対象にすることが難しい．残りの被験者は 2 名のみで比較すると，問題 1 では「Diff モード」 < 「Fine-grained モード」 (差は 308 秒)，問題 2 では「両方」 < Fine-grained モード」 (差は 244 秒)，問題 3 では「Diff モード」 < 「両方」 (差は 278 秒) であった．補足して，時間差は難易度が低い問題ほど大きくなる傾向が見られた．

SUS スコアは 75.6 (Grade は B，Adjective Rating は Good) であった．Grade が A となる 80.3 以上になるように一層のツールの改善が必要である．

フリーコメントでは，肯定的な意見においては Diff モードでは見つけられない実装の様子を確認できる Fine-grained モードの利点が複数指摘されていた．否定的な意見においては，Fine-grained モードだけでは全体像を見つけづらかったり，やり直しの多い編集について作業を追いかけづらかったりという指摘がされていた．Fine-grained モードの利点は確認されたので，その他の意見にも指摘されているようにより Fine-grained モードを使いやすく改善することが必要である．

ReviewMan の各モードをどのように使用するのかが効果的かについては，Diff モードと Fine-grained モードのどちらか一方ではなく併用することでそれぞれの利点を生かしたレビューをすることができることが分かった．例えば全体像を Diff モードでレビューを行い，

局所的に Fine-grained モードを使用することでレビューにかかる時間を抑えつつ，質の高いレビューを行うことができる．

その他の実験の気づきとして，Insights 機能が使用されていなかった．Insihgts 機能は ReviewMan の重要な機能の一つであるため，より一層の改良が必要である．また，Google におけるコードレビューのケーススタディでは，コードレビューの目的は次のように述べられている [26]．

> Google でのコードレビューに期待されていることは，**問題解決を軸としていない**．Google では可読性と保守性を担保するためにレビューが導入されたが，今日の開発者は**レビューの教育的側面**も理解している．欠陥の発見は歓迎されることだが，唯一の焦点ではない (原文抄訳)．

従来，コードレビューの目的は欠陥の発見が主であったが，現代ではコードレビューの目的は広範囲に渡っており，特に教育的側面が大きくなってきている．ケーススタディの結果より ReviewMan の Fine-grained モードは開発者の意図を理解した教育的なレビューを支援できると考えている．

5.2 Kudo

Kudo では KAMAKURA Plugin がインストールされているエディタでプログラミングの問題を解き，その過程を Kudo で可視化したものを被験者が評価するという実験を行った．以下ではこの実験について詳しく説明する．

本実験の目的は Kudo の開発進捗可視化がどのくらい開発者の体感に近いものであるかを確認することである．加えて可視化によって開発進捗の傾向を分析する．本実験では被験者 4 名に Java の問題を 4 問解くというタスクを課した．被験者には，Java の経験年数 1 年以上，能力レベルは 1 以上の情報科学研究科の学生を選んだ．使用した問題を表 13 に示す．問題 1 と問題 2 は容易な問題で，問題 3 と問題 4 は難解な問題である．

その他の実験条件を示す．コーディングには KAMAKURA Plugin がインストールされた Visual Studio Code[12] を使用した．問題回答時間は 10 分を目安とし，最大 20 分で終了することとした．コードの動作確認ができるように，テストコードを予め用意した．コーディング中の注意として，テストが成功したら完了とし，リファクタリングは不要であると指示をした．全問題終了後に Kudo グラフを作成し，被験者に体感的にどれくらい一致しているかとフリーコメントを聞き取った．

評価方法

ケーススタディでは「1. 開発者の体感的な開発状況とグラフ形状は一致しているか」,「2. 同一人物なら問題の難易度に応じてグラフ形状が変化するか」,「3. フリーコメント」の 3 点を調査する．1. では Kudo の Achievement Ratio の計算の距離関数に文字単位のレーベンシュタイン距離を使用することの妥当性を調査する．評価には被験者が体感的な開発状況と一致している度合いを 1(まったく思わない)〜5(とても思う) の 5 段階で回答したものを使用する．2. では Kudo による開発進捗の可視化で難易度に応じてどのようなグラフ形状の傾向が見られるかを調査する．3. では Kudo に関する自由な感想や改善点を定性的に調査する．

結果の仮説

「1. 開発者の体感的な開発状況とグラフ形状は一致しているか」については，概ね一致すると思われる．なぜならば，開発者は VCS 等の文脈で日常的にコードの変化量を開発進捗状況とみなしている．レーベンシュタイン距離はコードの変化量を表したものであるから，体感的な開発状況に非常に近いグラフが得られると予想される．加えて，グラフ形状に大きな変化があった期間，すなわちコード量が大きく増減した期間や停滞期間，すなわちコード量が変化せず情報収集等に費やした期間については印象に強く残るからである．しかしながら，セッションが長くなるにつれて記憶が不確かになり，体感的な開発状況と比較するのが難しくなると予想されたので，1 問あたりの回答制限時間を 20 分に設定した．

「2. 同一人物なら問題の難易度に応じてグラフ形状が変化するか」については，問題の難易度が低いほど，セッション開始時からコード編集が始まるまでの時間が短くなる傾向が見られると予想される．なぜならば，容易な問題ほど短い設計時間で実装を始められるからである．また，問題の難易度が低いほどコード編集が始まって以降，断続的に Achievement Ratio が変化する傾向が見られると予想される．なぜならば，容易な問題ほど十分に設計せず実装を始めてしまい後になって試行錯誤が必要になることが珍しくないからである．加えて，問題の難易度が高いほど試行錯誤が発生しやすい一方，慎重に設計されているのでコード編集開始後は比較的滑らかに Achievement Ratio が変化する傾向が見られると予想される．

「3. フリーコメント」については，被験者からできるだけ多くのフィードバックを得られることを期待する．

実験結果

全ての問題，被験者の Kudo グラフを図 9 に示す．また，全ての問題について被験者が体感的な開発状況と一致している度合いを 5 段階で回答したものを表 14 に示す．

表 14: 体感的な開発状況との一致度合い

	問題 1	問題 2	問題 3	問題 4	平均
被験者 A	4	4	4	5	**4.25**
被験者 B	5	4	5	5	**4.75**
被験者 C	5	5	4	5	**4.75**
被験者 D	5	5	4	5	**4.75**
平均	**4.75**	**4.5**	**4.25**	**5**	**4.63**

加えて，フリーコメントを肯定的，否定的，その他の 3 種類に分けて以下に示す．

肯定的なフリーコメント

- 難しい問題ほどプログラムをスムーズに書けて，簡単な問題ほど苦労していた気がする．ということを自分の体感と **Kudo** グラフから気づくことができた．
- コピー&ペーストや広範囲の編集操作がグラフに現れていていいと思いました．
- 簡単な問題では手を止めている時間とコーディングしている時間が分かりやすく，ソースコードを書いている感覚に近かった．

否定的なフリーコメント

- 難しい問題では試行錯誤することが多かったが，変数を追加してその後不要と気付いて削除した場合に少しグラフが想定よりも大きく動いたので少し体感と離れたのが気になった．

その他のフリーコメント

- もし **Kudo** を **ReviewMan** に埋め込むのであれば，点をクリックしたらその時点のコードを表示するような機能が欲しい．

結果の考察

「1. 開発者の体感的な開発状況とグラフ形状は一致しているか」については，平均スコアは 4.63 であった．従ってレーベンシュタイン距離による Achievement Ratio の時系列変化は開発者の体感的な開発進捗状況をよく表していることが分かった．補足してレーベンシュタイン距離を使用することの是非について議論すると，例えば一行のソースコードをコメントアウトした場合レーベンシュタイン距離はあまり変化しないが，プログラムの意味的には大きく変化するような場合もある．なので Achievement Ratio の距離関数については目的によって異なる関数を選択する必要もあると考えられる．

「2. 同一人物なら問題の難易度に応じてグラフ形状が変化するか」については，被験者 A と被験者 B には仮説通りの傾向が見られた．すなわち，容易な問題では短い設計時間でコーディングを開始するが，その後も停滞期を繰り返しながら実装を完了していた．一方難

解な問題では設計時間を長く取り，コーディング開始後はほぼ停滞期がなく実装を完了していた．しかしながら，被験者 C と被験者 D については容易な問題においては仮説通りの傾向が見られたが，難解な問題においては容易な問題に対する仮説と同じ傾向が見られた．すなわち難易度は無関係に短い設計時間でコーディングを開始し，その後も停滞期を繰り返しながら実装を完了していたということである．このように，開発者ごとで見ると 2 種類の開発進行方法があることが分かった．

「3. フリーコメント」については，肯定的な意見において Kudo グラフが体感的な開発進捗状況をよく表しているという意見が複数あった一方で，否定的な意見において体感と異なる場合が少しあったことが指摘されていた．このことから Kudo グラフは体感と多少異なることがあっても十分に開発進捗状況を可視化できていると思われる．また，その他の意見において Kudo を ReviewMan と組み合わせるという意見があり，Kudo グラフの形状が変化している箇所を ReviewMan の Fine-grained モードで詳しく検査するというシナリオは今後検討の必要があると考えた．

5.3 ケーススタディのまとめ

ケーススタディでは，KAMAKURA で収集した細粒度編集履歴データにより開発者にとって有用なツールを開発できることを確認できた．今後 KAMAKURA を利用した様々な開発者向けツールが開発されることを期待する．

6 関連研究

本研究で扱った細粒度開発履歴に関する研究は調査論文 [32] が大変詳しい．以降，KAMAKURA，ReviewMan，Kudo の関連研究をそれぞれ述べる．

KAMAKURA の関連研究として，上村らは OSS の開発データを有用な形式で公開・可視化することを目的とした Codosseum というツールを提案している [30]．Codosseum はメソッドレベルでソースコードの履歴を分析することを可能にするリポジトリである Historage[18] を利用している．Codosseum は，OSS 開発の支援とソフトウェア工学の研究成果を展開するプラットフォームとしての役割を強調している．

ReviewMan の関連研究として，細粒度開発履歴を利用してコミットを適切な単位に切り分けるという研究がある．一般的に細粒度開発履歴の量は膨大になりやすく，ReviewMan ではいくつかの手法で冗長な履歴を要約しているが，依然としてコードレビューしづらい煩雑なものとなる場合もあった．よって意味的なもつれの少ない単位に履歴を分割できれば，よりコードレビューに集中できると考えられる．梅川らは版管理システムへのコミットに複数の変更が含まれるのを防ぐために，細粒度作業履歴を分割，統合することで Task Level Commit を支援する手法を提案している [35]．有馬らは，本来 1 つのコミットに含まれるべき変更が複数のコミットにまたがっているものの自動的な検出方法を提案している [38]．Dias らは，複数の変更を含むコミットを紐解くことを目的として，EpiceaUntangler という細粒度編集履歴と機械学習アルゴリズムを使用した不可分コミットを支援するツールを提案している [16]．また，ReviewMan の Insight-gazer の一つに関連するファイルを表示する Related files があるが，Robbes らは，少量の細粒度操作履歴を用いることでより正確に高い頻度で同時に編集されるプログラム上の箇所の組 (ロジカルカップリング) を計算する手法を提案している [25]．

Kudo の関連研究として，大森らはタイムライン上でコード量の増加・減少や，コメントアウト・アンコメントが行われた時期が分かるようなという可視化を行っている [37]．齊藤らはプログラミング演習における受講生支援を目的として 50 名程度分の総 LOC(Lines of code) やコーディング時間，エラー継続時間等を一覧できるような可視化を行っている [27]．

7　まとめと今後の課題

本研究では，細粒度編集履歴収集プラットフォーム KAMAKURA の提案・開発を行い，これをコードレビューツールである ReviewMan と可視化を行う Kudo に適用し，ケーススタディを行った．ケーススタディにより，ReviewMan では，細粒度編集履歴を活用した Fine-grained モードが従来の Diff と併用することでより質の高い教育的なレビューを支援することが分かった．しかしながら，同様に細粒度編集履歴を活用した Insights 機能が使用されていないという課題があった．Kudo では，開発進捗状況の可視化により進捗の傾向を分析することが可能となり，可視化は開発者の体感的な進捗状況をよく表していることが分かった．

KAMAKURA の今後の課題として，統一された記録手法を活かし，既存の細粒度編集履歴関連ツールを KAMAKURA に移植できることを示したり，KAMAKURA を使用して細粒度編集履歴を大量に収集し，ソフトウェア進化等の研究に貢献したりすることが挙げられる．加えて，Visual Studio Code 以外の LSP 対応エディタ向けのプラグインの実装や個人情報等の本来収集されるべきでない情報が意図せずに収集されてしまう場合があるという問題の解決も取り組むべき課題である．

ReviewMan の今後の課題として，Insights 機能のユーザビリティの向上，より有用な Insight-gazer の追加実装，特定の編集アイテムに対してその場でレビューコメントを書ける機能の追加等が挙げられる．

Kudo の今後の課題として，ReviewMan との連携や文字単位のレーベンシュタイン距離以外の距離関数の実装が挙げられる．

また，その他の適用として教育現場において学生がコーディングのどこでつまづいていたかを理解したり，どのくらいの試行錯誤を行った成果であるかを評価したりするアプリケーションや，開発上級者が競技プログラミングの問題を解く様子を記録したものをインターネット上で共有し，開発初級者がこれを再生することで上級者の技術を学び取るというアプリケーションを今後開発，検証していきたい．

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，研究において多くの意義ある御意見を賜りました．心より感謝いたします．

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，研究において多くの御力添えを賜りました．心より感謝いたします．

奈良先端科学技術大学院大学先端科学技術研究科ソフトウェア工学研究室 石尾隆 准教授には，研究において多くの適切な御指導及び御助言を賜りました．

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 神田哲也 助教には，研究において多くの御助言を賜りました．心より感謝いたします．神田哲也 助教のおかげで本論文を完成させることができました．心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 伊藤薫，嶋利一真，並びに藤原裕士，小笠原康貴・白木秀弥・本田紘貴・溝内剛・原口公輔・松島一樹・宮本裕也 諸氏には，研究に関する相談に乗っていただき，様々なご支援を頂きました．心より深く感謝いたします．

最後に，私を常に支えてくださっ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感謝いたします．

参考文献

[1] Brotli: Ageneric-purpose lossless compression algorithm. https://github.com/google/brotli.

[2] Circleci. https://circleci.com/.

[3] Didchangetextdocument notification (lsp). https://microsoft.github.io/language-server-protocol/specifications/specification-3-14/#textDocument_didChange.

[4] Github. https://github.com/.

[5] Json-rpc 2.0 specification. https://www.jsonrpc.org/specification.

[6] Language server protocol. https://microsoft.github.io/language-server-protocol/.

[7] Languageserver.org: A community-driven source of knowledge for language server protocol implementations. https://langserver.org/.

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

[9] Stackoverflow developer survey 2019. https://insights.stackoverflow.com/survey/2019.

[10] System usability scale: Wikipedia. https://en.wikipedia.org/wiki/System_usability_scale.

[11] User scenarios. https://www.interaction-design.org/literature/topics/user-scenarios.

[12] Visual studio code. https://code.visualstudio.com/.

[13] A smartbear study of a cisco systems programming team. https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/, 2013.

[14] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pp. 712–721. IEEE Press, 2013.

[15] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pp. 931–940. IEEE Press, 2013.

[16] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 341–350. IEEE, 2015.

[17] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pp. 3–24. ACM, 2007.

[18] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pp. 96–100, 2011.

[19] Grzegorz Kondrak. N-gram similarity and distance. In *International symposium on string processing and information retrieval*, pp. 115–126. Springer, 2005.

[20] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: how developers see it. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 1028–1038. IEEE, 2016.

[21] Stas Negara, Mohsen Vakilian, Nicholas Chen, 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. Springer, 2012.

[22] Masao Ohira, Reishi Yokomori, Makoto Sakai, Ken-ichi Matsumoto, Katsuro Inoue, and Koji Torii. Empirical project monitor: Automatic data collection and analysis toward software process improvement. In *International Workshop on Mining Software Repositories (MSR2004)*, pp. 42–46. Citeseer, 2004.

[23] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 31–34. ACM, 2008.

[24] Romain Robbes and Michele Lanza. Spyware: A change-aware development toolset. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE ' 08, p. 847–850, New York, NY, USA, 2008. Association for Computing Machinery.

[25] Romain Robbes, Damien Pollet, and Michele Lanza. Logical coupling based on fine-grained change information. In *2008 15th Working Conference on Reverse Engineering*, pp. 42–46. IEEE, 2008.

[26] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

[27] Syun SAITO, Makoto YAMADA, Hiroshi IGAKI, Shinji KUSUMOTO, Akifumi INOUE, and Tohru HOSHI. Programming process visualization for supporting students in programming exercise. *Technical report of IEICE. SS*, Vol. 111, No. 481, pp. 61–66, mar 2012.

[28] Erik Stolterman and Anna Croon Fors. Information technology and the good life. In *Information systems research*, pp. 687–692. Springer, 2004.

[29] 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. 4_61–4_66, 2013.

[30] 上村恭平, 中才恵太朗, 大神勝也, 畑秀明, 一ノ瀬智浩, 松本健一, 飯田元. Codosseum: オープンなソフトウェア開発・分析支援 web サービス. コンピュータ ソフトウェア, Vol. 36, No. 1, pp. 38–47, 2019.

[31] 大森隆行, 丸山勝久. プログラム開発履歴調査のための編集操作再生器. コンピュータ ソフトウェア, Vol. 28, No. 4, pp. 4_371–4_376, 2011.

[32] 大森隆行, 林晋平, 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. 1_60–1_80, 2015.

[33] 藤原理也. 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.

[34] 独立行政法人情報処理推進機構（IPA）社会基盤センター. IT 人材白書 2019. 2019.

[35] 梅川晃一, 井垣宏, 吉田則裕, 井上克郎. 細粒度作業履歴を用いた task level commit 支援手法の提案. 電子情報通信学会技術報告, Vol. 113, No. 489, pp. 61–66, 2013.

[36] 木津栄二郎. ソースコードの編集履歴を用いたプログラム変更理解に関する研究. 2014.

[37] 大森隆行, 丸山勝久. プログラム開発履歴調査のための編集操作再生器. コンピュータ ソフトウェア, Vol. 28, No. 4, pp. 4_371–4_376, 2011.

[38] 有馬諒, 肥後芳樹, 楠本真二. 不適切に分割されたコミットに関する研究. コンピュータソフトウェア, Vol. 35, No. 4, pp. 164–170, 2018.

表 13: Kudo のケーススタディで使用した問題

問題番号	タイトル	難易度	内容
問題 1	Can't wait for holiday	低	今日の曜日を表す文字列 S("SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"のいずれか) が与えられたとき，次の日曜日が何日後か求めよ．(https://atcoder.jp/contests/abc146/tasks/abc146_a)
問題 2	ROT N	低	英大文字のみからなる文字列 S がある．また，整数 N が与えられる．S の各文字を，アルファベット順で N 個後の文字に置き換えた文字列を出力せよ．ただしアルファベット順で'Z' の 1 個後の文字は'A' とみなす．(https://atcoder.jp/contests/abc146/tasks/abc146_b)
問題 3	Buy an integer	高	整数 N を買うためには $A \times N + B \times d(N)$ 円必要である．ここで $d(N)$ は N の十進表記の桁数である．所持金が X 円のとき購入できる最も大きい整数を求めよ．(https://atcoder.jp/contests/abc146/tasks/abc146_c)
問題 4	Walk on Multipilication Table	高	掛け算表のマス (i, j) には整数 $i \times j$ が書かれており，最初 $(1, 1)$ にいる．1 回の移動で (i, j) から $(i+1, j)$ か $(i, j+1)$ のどちらかにのみ移ることができる．整数 N が与えられたとき，N が書かれているマスに到達するまでの必要な最小の移動回数を求めよ．(https://atcoder.jp/contests/abc144/tasks/abc144_c)

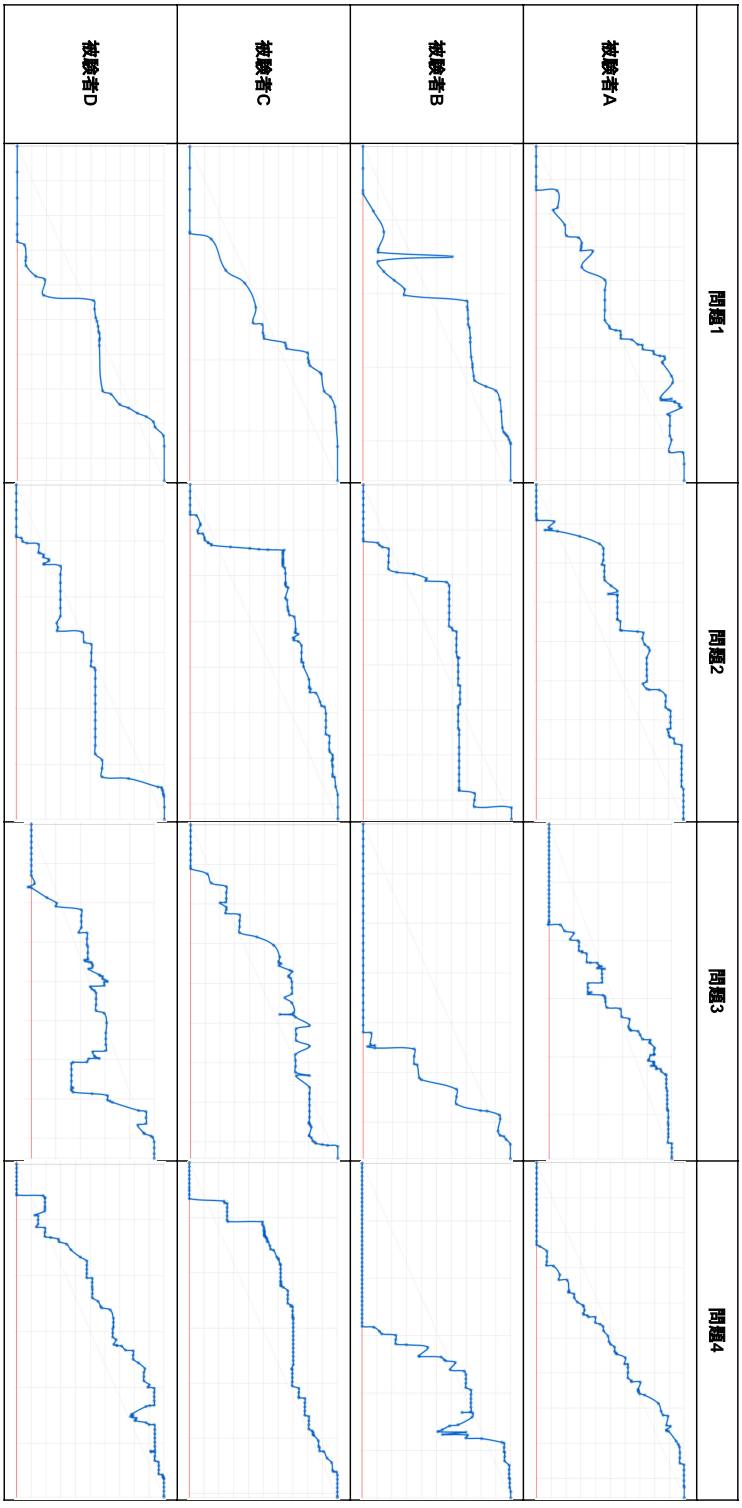

図 9: ケーススタディで得られた Kudo グラフ