Vol.2015-SE-187 No.9
2015/3/12

ソースコード編集履歴の不吉な臭いの検出

星野 大樹[1,a)]　林 晋平[1,b)]　佐伯 元司[1,c)]

概要：ソースコード編集履歴の理解性や利用性を向上させるための履歴リファクタリング手法が提案されている．しかし，既存手法ではどのような編集履歴をどのようにリファクタリングすべきか明確でない．本稿ではリファクタリングが必要となる履歴の特徴を「履歴の臭い」として定義し，また，履歴の臭いを判別するためのメトリクスを提案する．提案したメトリクスによって各編集操作の結びつきを捉え，臭いの自動検出を可能とする．検出の精度について評価を行い，適合度 0.86 など有用な結果を得た．

キーワード：ソースコード編集履歴，コードの臭い，リファクタリング，もつれた変更

Detecting Bad Smells in Edit History of Source Code

Abstract: History refactorings that improve the understandability and usability of a history of source code have been proposed. However, the proposed technique has not define where and how to refactor a history. We define bad smells in history and metrics for detecting them. Identifying the relationship between editing operations in a history by using the proposed metrics leads to automated detection of bad smells in history. We confirmed that our detection technique is useful due to its high accuracy.

Keywords: editing history of source code, code smell, refactoring, tangled code changes

1. はじめに

ソースコードの編集の記録（編集履歴）はソフトウェア開発の中で利用されており，開発の効率化や理解に役立っている．例えば開発者はコードエディタが保持している編集履歴を利用し，これまでの編集操作を取り消す Undo 操作や，Undo した操作を適用し直す Redo 操作によりさまざまな試行錯誤を行ったり，編集内容の確認を行ったりしている．また，版管理リポジトリが蓄えるソースコードの改版の情報も，開発者が行った編集結果に基づくものであり，編集履歴を反映している．

こうした履歴の有用性に着目し，Hayashi らは，履歴全体が表している効果を変えないまま履歴の内容を再構成し，編集履歴の利用性や理解性を向上させる編集履歴のリファクタリング手法を提案し，自動化ツールを示した [1]．この研究では，ソフトウェア開発において，振る舞いに影響しない変更と，機能追加などの変更が混在することがあり，その際に蓄積された編集履歴は差分理解の観点から不適切であり，リファクタリングが必要であるとしている．しかし，不適切な編集履歴の表層的な特徴は定義されておらず，開発者へのリファクタリングの指針が不明確な点と，検出の自動化が行えない点が問題として残る．

本稿では編集履歴の分析を行い，どのような編集履歴をどのようにリファクタリングすべきか「履歴の臭い」として定義するとともに，臭いの検出に有用なメトリクスを定義し，検出の自動化を目指す[*1]．定義した臭いの検出には，時間軸上で隣接した編集操作間における意図の異なる箇所（意図境界）の特定が有用となるため，意図境界の検出手法を示し，検出精度について評価を行ったところ，適合度 0.86 など有用な結果を得た．

本研究の主な貢献を以下に示す．

(1) 関連研究の調査から，不適切な編集履歴を「履歴の臭い」として定義し，リファクタリング案と関連付けて

1　東京工業大学大学院情報理工学研究科計算工学専攻
Department of Computer Science, Tokyo Institute of Technology.
a)　dhoshino@se.cs.titech.ac.jp
b)　hayashi@se.cs.titech.ac.jp
c)　saeki@se.cs.titech.ac.jp

*1　本稿は，我々のこれまでのアイディア [2] を発展させたものである．

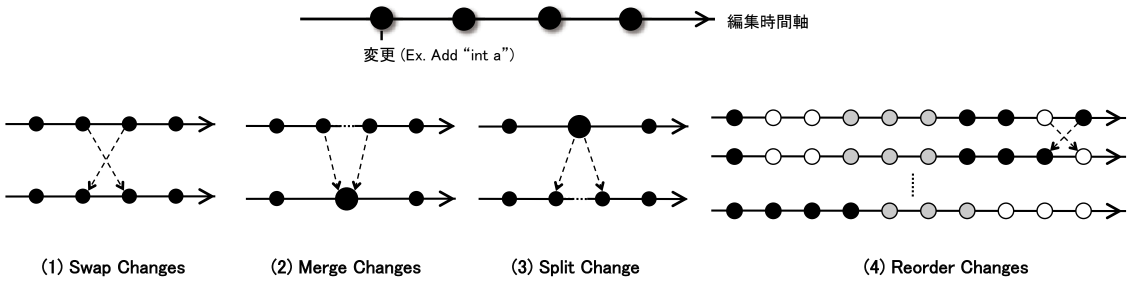


図 1 リファクタリング操作

カタログとして示したこと．
(2) 開発プロジェクトから得た編集履歴の分析により，編集意図の境界を特定するためのメトリクスを 3 つ定義し，それらの有用性を示したこと．
(3) テキスト編集レベルの粒度の履歴情報を用いて，変更のもつれを検出する手法を提案したこと．

本論文の構成を述べる．まず 2 章で履歴リファクタリング手法とその問題点について述べ，3 章で問題点の解決に向けたアプローチについて述べる．4 章で履歴の臭いをカタログ化し，5 章で臭いの検出に有用なメトリクスを定義する．6 章で評価を行い，7 章で関連研究を述べる．最後に 8 章でまとめと今後の課題について述べる．

2. 履歴リファクタリング手法の問題点

2.1 履歴リファクタリング手法

Hayashi らは，履歴全体が表している効果を変えないまま履歴の内容を再構成する，履歴リファクタリング手法を提案している [1]．履歴リファクタリング手法では，特定のソースコードに対する連続する字句の追加や削除，あるいは置換を**編集片**（chunk）と呼び，組 $h \coloneqq (t, f, o, r, a)$ で表す．ここで，t は編集が行われた時刻，f は編集対象のファイル，o は編集の開始オフセット，r は編集前のファイルから削除された文字列，a は編集後のファイルに追加された文字列を表す．それぞれの値は，$h.time \coloneqq t$，$h.file \coloneqq f$，$h.offset \coloneqq o$，$h.added \coloneqq a$，$h.removed \coloneqq r$ とも参照できる．

これまでに，Swap Changes（Swap），Merge Changes（Merge），Split Changes（Split）という基本的なリファクタリングが定義されている．これらの履歴リファクタリングの概要を**図 1** に示す．図中の実線矢印は時系列を，丸は変更を表し，上部をリファクタリングしたものが下部である．Swap は図 1(1) に示すように，変更を行う順番を入れ替える操作である．ただし，変更間に依存関係がある場合は交換できない．Merge は図 1(2) に示すように，複数にまたがる変更は単一として構成する操作である．Split は図 1(3) に示すように，単一の変更を複数に切り分ける操作である．また，大きなリファクタリングとして，図 1(4) に示すように，Swap を繰り返し適用することで，編集履歴を予め定められたグループ毎にまとめる Reorder というリファクタリングが存在する．

2.2 問題点

この手法によって編集履歴のリファクタリングは可能となったものの，どのような編集履歴にどのようなリファクタリングを適用すべきか明確にされていない．そのため，履歴リファクタリングの有効な利用法を開発者に伝えられておらず，リファクタリングすべき編集履歴を自動で検出できない点が問題として残る．例えばコードリファクタリングにおいては，Fowler らがリファクタリングが有効なコードの状態を不吉な臭いとしてカタログ化し，開発者がリファクタリングをする際の指標として利用されている [3]．また，不吉な臭いを自動検出する手法も提案されており，Fowler らの指標に従ったリファクタリングが容易に実行でき，ソースコードの理解性や利用性の向上に貢献している [4], [5]．履歴リファクタリングにおいてもコードリファクタリングと同様に，以下が必要である．
(1) リファクタリングをする際の指標となる不吉な臭いのカタログ化
(2) 編集履歴の不吉な臭いの自動検出

3. アプローチ

関連研究の調査に基づき，コードリファクタリングにおいて Fowler らが示したものと同様に，履歴リファクタリングにおいて編集履歴をリファクタリングする際の指標となる不吉な臭いをカタログ化する．これにより，2.2 節で述べた (1) を満たすことができる．また，実際の開発プロジェクトから編集履歴の収集を行い，不適切な編集履歴に共通する特徴を調査し，臭いを自動検出するために有用なメトリクスの検討を行うことで，(2) に繋げる．以上により，履歴リファクタリング手法に存在する問題点を解決する．

4. 履歴の臭いのカタログ

一般的なパターンカタログに倣い，各臭いについてその定義，問題例，解決策の 3 項目を記述する．定義では，編集履歴がどのような状態のことを臭いと呼ぶのか記述する．問題では，その臭いのする編集履歴がなぜリファクタ

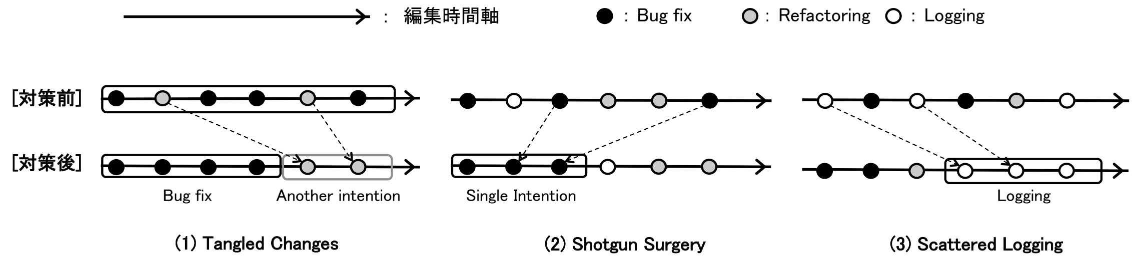


図 2　編集履歴の不吉な臭いとその解消

リングを必要とするのか記述する．解決策では，どのようにリファクタリングをすれば臭いが解消されるかを記述する．定義と問題の記述により，既存手法の問題点の 1 つである，どのような編集履歴をリファクタリングすべきかという点を明確にする．同様に，対策の記述により，既存手法の問題点の 1 つである，リファクタリングが必要な編集履歴をどのようにリファクタリングすべきかという点を明確にする．

以下に履歴の臭いのカタログを示す．

(1) Tangled Changes

定義　まとまりとされた編集操作群に，複数の編集意図が混在する．

問題　Task Level Commit に適さない．

対策　編集意図ごとに分かれるよう，編集操作群を Split 及び Swap/Reorder する．

出典　Herzig ら [6]，Kawrykow ら [7]．

(2) Shotgun Surgery

定義　同一意図の編集操作が編集履歴上に点在する．

問題　Task Level Commit に適さない．

対策　点在した編集操作を Swap/Reorder してまとめ，Merge する．

出典　Fowler ら [3]，Hayashi ら [1]．

(3) Scattered Logging

定義　ロギングの編集操作が編集履歴に点在する．

問題　ロギングの Undo が労力となる．

対策　ロギング操作を Merge し，一括で Undo 出来るようにする．

出典　Hayashi ら [8]．

図 2 に各臭いを示す．上段が臭いを持つ履歴の状態で，下段は臭い解消後の状態である．図中で色の異なる丸は，編集意図が異なる編集操作を表す．

Tangled Changes の例を**図 3** に示す．この例は，6 章で評価対象とした編集履歴に存在したもので，「通知を出さないよう条件を追加した」というコミットメッセージに対して，1:46:31–1:49:52 内の連続した時間内で，通知を出さないよう DropboxOperator.java において if 文が追加されている．その後約 4 分の時間を隔て，1:53:35–2:02:07 内の連続した時間内で MainActivity.java, PagerActivity.java, strings.xml に対してコメントアウトの消去や改行の消去，インデントの修正が行われている．後者の編集はプログラムの振る舞いに影響せず，コミットメッセージの内容とは無関係であるが，コミットメッセージ通りである前者の編集と同時にコミットされてしまっている．これにより，本来行われるべき編集は前者の DropboxOperator.java に対する 12 行だけであるにも関わらず，差分は 170 行以上に渡っており，その理解性を低下させている．すなわち，編集操作群に複数の意図が混在している場合，編集意図ごとに編集操作群を切り分けるリファクタリングが必要であり，単一意図の編集操作群ごとに版管理リポジトリへ反映させるべきである．

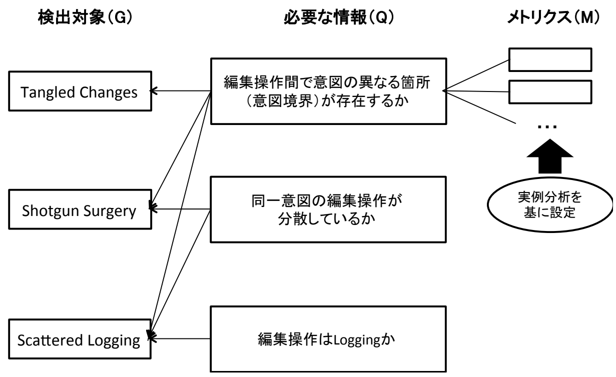


図 4　臭い検出における GQM モデル

5. メトリクスを用いた臭いの検出

各臭いの検出に必要な情報を**図 4** に示す．検出された意図境界に隣接する 2 編集操作が同一意図とされていた場合には，Tangled Changes として検出できる．意図が特に割り振られておらず，かつ意図境界をいくつか跨いで再び同一意図の編集操作が存在した場合，Shotgun Surgery として検出できる．また，Shotgun Surgery において同一意図である編集操作が Logging 操作の場合，Scattered Logging として検出できる．

以上から分かるように，定義した臭いの検出には，時間軸上で隣接した編集操作間における意図の異なる箇所（意図境界）の特定が有用となる．そこで，意図境界を検出するためのメトリクスを定義し，意図境界の自動検出を行う．検出された意図境界を基とすれば，臭いの検出が可能

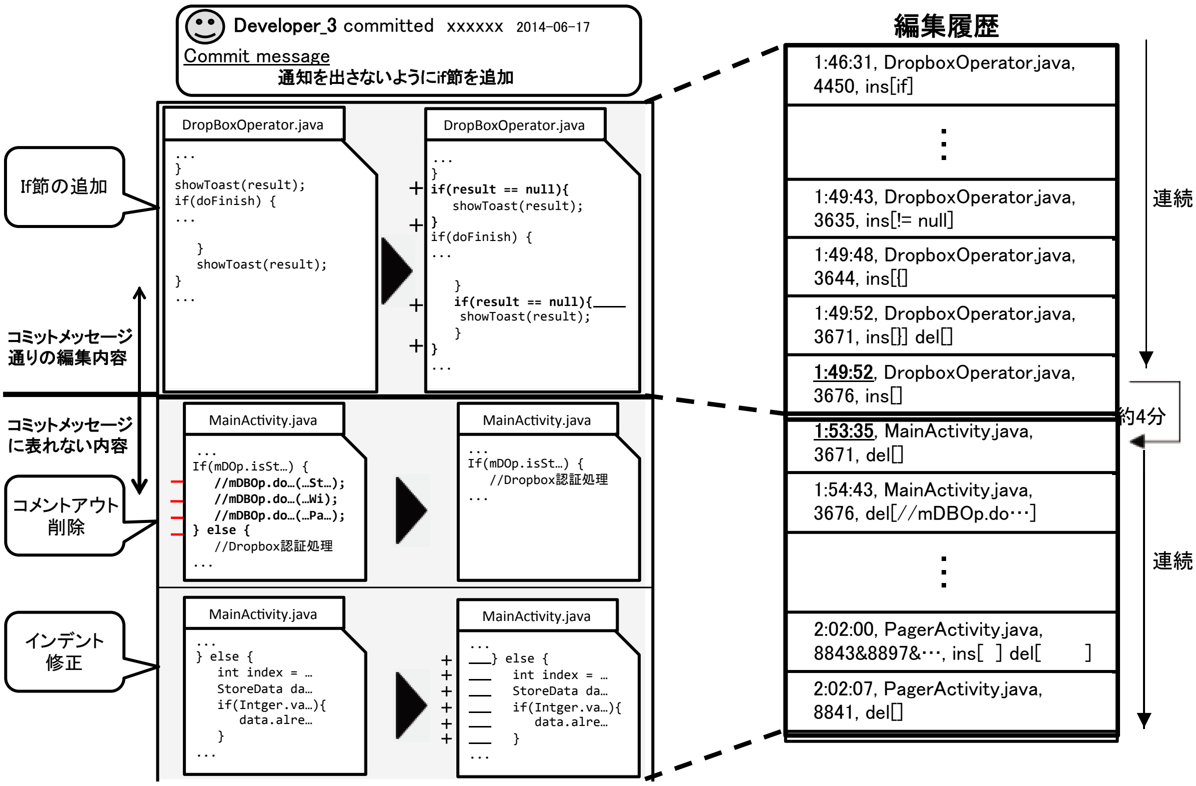


図 3　Tangled Changes の具体例

となる．

5.1　メトリクスの定義

以下に，2 編集片間の時間距離を測る Time Interval，ファイル外での空間距離を測る Package Distance，ファイル内での空間距離を測る File Distance の定義を示す．なお，本手法におけるソースコード編集履歴に関する定義は Historef [1] に準拠する．すべての隣り合う h_i, h_j に対して：

(1) Time Interval (T)
　編集時間差をミリ秒単位で測定する．

$$T_{ij} := |h_i.time - h_j.time| \tag{1}$$

(2) Package Distance (P)
　パッケージの移動距離を測定する．編集先が同一ファイルならば値は 0，それ以外の場合，h_i の編集先ファイルを含むパッケージから，h_j の編集先ファイルを含むパッケージまで，どれだけパッケージを跨ぐかを計測し，その値に 1 を加えて距離とする．例えば，同一パッケージの別ファイルの場合，距離は 1 となる．

$$P_{ij} := \begin{cases} 0 & if \quad h_i.file = h_j.file \\ \geq 1 & otherwise \end{cases} \tag{2}$$

(3) File Distance(F)
　編集先オフセットの距離を測定する．

$$F_{ij} := \begin{cases} |h_i.offset' - h_j.offset'| & if \quad P_{ij} = 0 \\ \infty & otherwise \end{cases} \tag{3}$$

ここで，式 3 に出現する $h.offset'$ は，編集片 h の生成時点での編集先オフセット情報 $h.offset$ と異なり，編集履歴の分析を行う時点でのオフセット情報となる．そのため，編集片 h 生成時点から編集履歴の分析を行うまでの間に，$h.offset$ より前のオフセットに挿入編集や削除編集が行われた場合，$h.offset'$ は $h.offset$ と異なる値をとる．

5.2　臭いの検出

あるプロジェクトにおいて得られた各編集操作間のメトリクス値の測定と，各編集操作間が意図境界であるかの調査を行い，統計分類アルゴリズムによって各編集操作間が意図境界かどうか分類するモデルを構築する．臭い検出の対象となる編集履歴に存在する各編集操作間を，構築したモデルによって分類することで，意図境界の検出を行うことができる．

6.　評価

6.1　評価項目と方法

本稿における評価では，以下の 3 つの質問に回答することを目的とする．

EQ1: 3 メトリクスはそれぞれ単独で意図境界の予測に有用か． これに回答するため，意図境界の予測のために定義した，File Distance，Package Distance，Time

表 1 評価対象

対象	開発者数	開発期間	コミット数	食い違いコミット数	総編集操作数	意図境界数
LLife	4 名	2 ヶ月	86	25	32537	228
Reversi	1 名	2 日	0	0	2338	24

Interval の各メトリクスの値が実例でどのように分布しているかを調査する．

EQ2: 3 メトリクスで意図境界の予測が可能か． 定義したメトリクスを組み合わせて用いることにより，意図境界の予測が行えるかを調査する．

EQ3: プロジェクトによって編集履歴の特徴は異なるか． あるプロジェクトの編集履歴を基に構築した意図境界の分類モデルを用いて，他のプロジェクトを分類する際，プロジェクトによって編集履歴の特徴が異なる場合，精度の良い分類は行えない．分類モデルを構築する編集履歴と，分類を行う編集履歴が異なるプロジェクトから得られた場合でも精度の良い結果が得られうるか調査する．

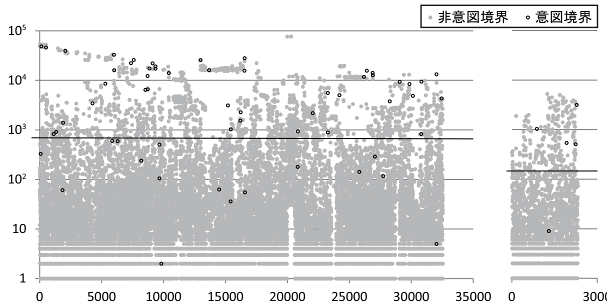
6.2 編集履歴の収集

実開発プロジェクトを 2 つ対象に，各開発者の開発環境に編集履歴収集ツールを導入して編集履歴を収集し，評価対象とした．一方は Android アプリケーション "LLife" 開発プロジェクトにおいて，履歴収集ツール Fluorite [9] を用いて収集した履歴であり，もう一方は Java アプリケーション "Reversi" の開発プロジェクトにおいて，履歴収集ツール OperationRecorder [10] を用いて収集した履歴である．対象プロジェクトの規模を**表 1** に示す．ここで食い違いコミット数とは，各コミットについてコミットメッセージとコミット内容に意味的な食い違いが存在した数を示す．Reversi は版管理システムを用いて開発されなかったため，コミット数に関連する値は 0 となる．

また，著者らの 1 人が編集履歴を分析し，意図境界を特定した．意図境界は，開発者がコミットを実行した前後の編集操作に加え，1 コミット内での意図の混在を目視で分析し，加えて特定した．分析の際には，特定の新規機能の追加，既存の機能の修正，振る舞いに影響しない変更の 3 種を意図の異なる編集の種類としてとらえ，実際の編集がどれに該当するかを特定しながら行い，編集間で種類や対象機能の相違がみられた際には，意図境界として扱った．表 1 には特定した意図境界数を示してある．

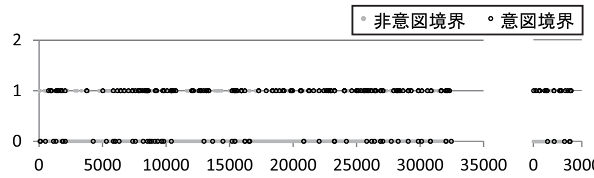
6.3 EQ1

はじめに，各メトリクスがそれぞれ単独で意図境界の予測に有用か調べるため，全データを対象として各メトリクスの測定値の分布を調べた．ここで，特定の閾値を境に意図境界と非意図境界における値が別れるような結果となれば，そのメトリクスは単独で意図境界の検出を行えることになる．以降で説明する図は全て，左側に LLife の分布，右側に Reversi の分布を示し，黒色のプロットが意図境界，灰色のプロットが非意図境界を示す．また，編集片出現順に横軸に沿ってデータを並べ，縦軸にメトリクス値を示している．



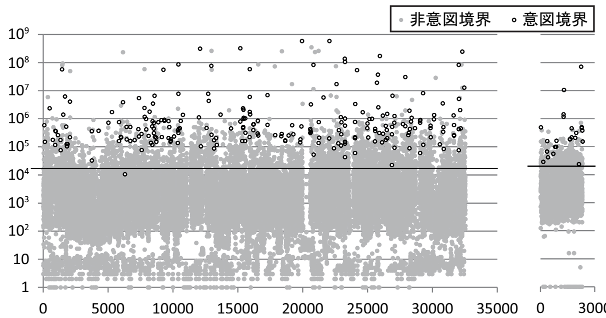
(a) LLife (b) Reversi

図 5 File Distance の分布



(a) LLife (b) Reversi

図 6 Package Distance の分布



(a) LLife (b) Reversi

図 7 Time Interval の分布

File Distance のプロットを**図 5** に示す．図において，縦軸は対数軸である．分布を見る限り，意図境界には高いメトリクス値をとるものが比較的多いが，全体的にまばらな分布となっている．そのため，File Distance 単独で意図境界の検出を行うことは難しい．

Package Distancen のプロットを**図 6** に示す．対象としたプロジェクトは単一パッケージのみを用いて開発が行われたため，メトリクス値は 2 編集片間の編集先が同ファイルである 0 の値と，編集先が異なるファイルである 1 の値

表 2 LLife の評価値

適用対象	Precision	Recall	F-Measure
BFTree	0.582	0.671	0.623
J48	0.696	0.452	0.548
NaiveBayes	0.097	0.118	0.107

のみとなる．意図境界にはメトリクス値が 1 となるものが多く，プロットの密度が高いものの，値が 0 となる意図境界や，値が 1 となる非意図境界も多数みられるため，Package Distance 単独で意図境界の検出を行うことは難しい．

最後に，Time Interval のプロットを**図 7** に示す．プロット図から分かるように，File Distance や Package Distance でみられた以上に，意図境界が高いメトリクス値に偏っている．しかし，非意図境界でもメトリクス値が高いものが残っており，意図境界と非意図境界を分ける決定的な閾値は発見できない．

以上から，各メトリクスそれぞれである程度偏りは見られたが，各メトリクスを単独で用いて意図境界の検出を行うだけでは不十分と考える．

6.4 EQ2

複数のメトリクスを組み合わせて用いて意図境界の検出を行うことを考える．

はじめに，3 つのメトリクスを全て用いて検出が行えるか，機械学習を利用して確かめる．LLife から得られた編集履歴について，時間軸上で隣接した全編集操作間における，File Distance，Package Distance，Time Interval を測定し，機械学習器 Weka 上で利用できる統計分類アルゴリズム，BFTree，J48，NaiveBayes を用いて分類モデルを構築した．モデル構築に利用した編集履歴自身をモデルによって分類した際の評価値を**表 2** に示す．BFTree による分類は，Precision，Recall ともにバランスが良く，3 つの分類アルゴリズムの中で最も F-measure が高い．J48 は Precision が比較的高いものの，Recall は低めの値となっており，NaiveBayes は全体的に値が低い．

3 つの分類アルゴリズムの中で，F-measure が最も高い BFTree アルゴリズムによる LLife の分類モデルを**図 8** に示す．総数 32537 のデータは，はじめに Time Interval（T）の値が低いものは意図境界ではない（非意図境界）として分類される．ノードに直接記述された数値は，非意図境界として決定されたデータのうち，31905 データは実際に非意図境界であり，32 データは実際は意図境界であったことを示している．$T \geq 152563$ となるデータは続いて Package Distance（P）の値により分類される．$P = 1$ となるデータは意図境界として分類され，$P = 0$ となるデータは，次に File Distance（F）の値により分類される．F が低いデータは非意図境界として分類され，$F \geq 4166.5$ となるデータは，最後に再び T 値により，意図境界と非意図境界に分類

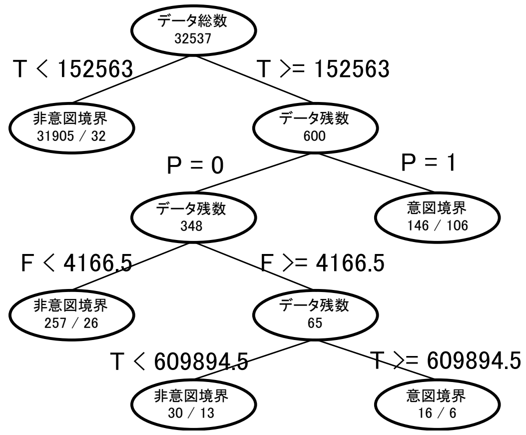


図 8 LLife の分類モデル

表 3 利用メトリクスによる比較

利用メトリクス	Precision	Recall	F-Measure	F 値の差
TI	0.335	0.863	0.483	
TI + PD	0.579	0.627	0.602	+25%
ORIGINAL	0.591	0.695	0.639	+6%

される．この決定木からは，Time Interval がデータの分類に非常に大きく貢献することが分かる．

そこで，このモデルから Package Distance， File Distance のメトリクスを取り除いたモデル（TI）を構築する．また，File Distance により分類されるデータが少ないため，File Distance を除き Time Interval， Package Distance のみ利用するモデル（TI + PD）を構築する．TI，TI + PD，元々の分類モデル（ORIGINAL）の 3 つによって LLife 自身を分類した際の評価値を比較することで，各メトリクスが意図境界の分類に有用か確かめる．

表 3 に各評価値を示す．Time Interval のみで分類を行った場合，ORIGINAL と比べて Recall は高いが，False Positive も増加して Precision が減少，結果的に F-Measure も低くなっている．また，TI + PD では，TI と比べて Recall は減少するものの Precision が増加し，F-Measure も 25% 増加している．しかし ORIGINAL と比較すると，Precision，Recall ともに低くなっている．以上より，Time Interval だけでなく，Package Distance，File Distance も意図境界の分類に有用であり，特に Package Distance は Precision の増加に有用，File Distance は僅かではあるが Presicion と Recall ともに増大させることが分かった．

3 つのメトリクス全てが有用であるという結果を受け，3 メトリクスを全て用いて構築した LLife の分類モデルを用いて，精度の高い意図境界の予測ができるか調査する．そこで，LLife の分類モデルを用いて Reversi の編集履歴を分類した結果，Precision は 0.857，Recall は 0.5 となった．関連手法となる，Herzig らによる，スナップショットに基づく既存のコミット分割法 [6] において，4 つのプロジェクトのソースコードを分割した際の評価においては Recall

表 4 Reversi の評価値

適用対象	Precision	Recall	F-Measure
BFTree	0.826	0.792	0.809
J48	0.875	0.583	0.700
NaiveBayes	0.322	0.792	0.458

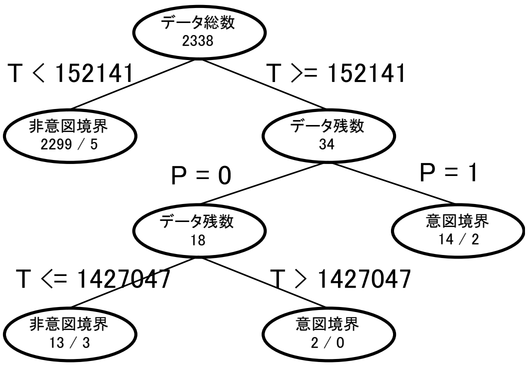


図 9 Reversi の分類モデル

は測定されておらず，Precision については各プロジェクトに対して 0.71〜0.92 の値をとっている．評価手法が一致しているわけではないため，直接の比較を行うことはできないが，本手法の Precision は 0.857 と参考値と比べて遜色なく，かつ Herzig らとは異なり編集片という細粒度なデータを分割できる点で，本手法による予測は良好で，有用なものと考える．

6.5 EQ3

LLife と同様に Reversi からも分類モデルを構築し，互いの分類モデルの比較や，分類結果の比較を行うことで EQ3 に答える．

はじめに，3 つの分類アルゴリズムによる Reversi の評価値を**表 4** に示す．どのメトリクスがどの程度意図境界の分類に用いられるか構築した分類モデルを見て判断を行えるように，F-measure が高めで，かつ複雑すぎない分類モデルとして，J48 によるモデルを選択し，比較する．

図 8 は LLife の分類モデルであり，**図 9** は Reversi の分類モデルを示す．どちらの分類モデルも，Time Interval が約 150000 を下回るデータを全て非意図境界と分類している．また，続いて Pacakage Distance の違いにより分岐があることも共通している．さらに，File Distance によって分類されるデータは僅かであるか，存在しない．ここから，2 つの分類モデルは類似していると考える．

次に，Reversi により構築された分類モデルで LLife の編集履歴を分類し，LLife により構築された分類モデルで Reversi の編集履歴を分類した．自身を分類した際の結果と合わせて，**表 5** に示す．表から分かるように，LLife, Reversi ともに分類モデルを入れ替えても評価値に大きな差は出ない．以上より，Time Interval，Package Distance, File Distance の 3 つの観点からすれば，2 つの編集履歴の特徴に大きな差はないことが分かった．

表 5 分類モデルによる比較

分類対象	(分類モデル)	Precision	Recall	F-Measure
LLife	(Reversi)	0.550	0.675	0.606
LLife	(LLife)	0.582	0.671	0.623
Reversi	(LLife)	0.857	0.500	0.632
Reversi	(Reversi)	0.875	0.583	0.700

6.6 妥当性の脅威

今回は編集履歴の比較として 2 プロジェクトしか対象としていないため，比較を行った 2 つの編集履歴が偶然同様の特徴を持つ履歴だったことが考えられる．また，構築した分類モデルを 1 プロジェクトにしか適用していない点についても，偶然精度の高い値が得られた可能性がある．これらは外部妥当性への脅威となるため，更なるプロジェクトから編集履歴を収集することで，比較対象と分類対象を増やし，改善していきたい．

7. 関連研究

提案手法と関連する研究として，意図境界の設定に関するものがある．Safer ら [11] は開発者に自身の開発における意図境界を自身で設定させる実験の中で，時間という情報が意図境界の判断に役立つことを示している．また，Omori ら [12] はある編集からある編集まで，30 分の時間が空いた場合，意図境界である可能性が高いとしている．本手法でも示したように，時間が空くことと意図境界の関係性は強い．本手法ではさらに，機械学習を行うことで時間の空き以外にも複数の要素が意図境界に影響することを示している点が異なる．

複数の変更が入り混じって行われる,「変更のもつれ」についても様々に研究されている．Hayashi ら [1] は意図が切り替わるごとに開発者に手動でエディタのモードを変更させ，モードごとにコミットを行うことで Task Level Commit の支援を行う手法を提案している．これに対し，本手法は複数のメトリクスを用いて Task Level Commit に適した境界を自動で検出しようと試みたという点で異なる．また，評価の基準とした Herzig ら [6] による 6 メトリクスを用いた手法や，Kirinuki ら [14] によって提案された，開発者の以前のコミット履歴を基に，次に行うコミットが Task Level Commit に沿っているか判別する手法，Nguyen ら [15] による，コミットされたソースファイルをその構文情報に基づいて，関連性の高いソースファイルごとに分類する手法が存在する．これらの手法はコミットされた後のソースコードに着目するため，ソースコードの編集時間という情報を扱わない点で，本手法とは異なる．他にも，時間を扱うコード分類手法として，Dias ら [16] の手法が存在する．時間を含めた複数のメトリクスから，機械学習を用いて有用と判断したメトリクスを用いてソースコードを分

類する手法であり，提案手法と類似している．しかし，扱う編集の粒度が異なり，我々の手法の方がテキスト編集レベルである細かい粒度の編集を分類している．

8. まとめ

本稿では編集履歴リファクタリングに有用な履歴の臭いを定義し，臭いの検出に有用となる，意図境界の自動検出法を提案した．関連研究を基に履歴の臭いの定義し，開発プロジェクトから得た編集履歴の分析により，臭いの自動検出に有用となる，意図境界を特定するためのメトリクスの定義を行った．機械学習を用いて分類モデルを構築することで，定義したメトリクスが全て意図境界に有用であること確認し，意図境界の自動検出を実現した．.

今後の課題として、以下の 3 つが挙げられる。

- 更なる臭いの選出
  本稿では編集履歴における不吉な臭いを 3 つ定義したが，どれも意図境界に着目した臭いとなった．Fowler らが提案したコードにおける不吉な臭いのように，様々な局面でより良い履歴を生み出せるように，履歴リファクタリングが有用な場面について更なる調査を行い，新たな臭いの定義を行う必要がある．
- 更なるメトリクスの考案
  分類に用いるメトリクスを更に増やし，より分類精度の高い分類モデルを構築する必要がある．
- 支援ツールとして実装
  今回は臭いの検出法を示すに留まったが，今後は実際に開発途中のユーザーに不吉な臭いを知らせる機能の実装をしていきたい．

謝辞 本研究における分析対象データの一部は，立命館大学の丸山勝久教授，大森隆行助教にご提供頂いた．ここに深く感謝する．

参考文献

[1] Hayashi, S., Omori, T., Zenmyo, T., Maruyama, K. and Saeki, M.: Refactoring Edit History of Source Code, *In Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 617–620 (2012).

[2] 星野大樹，林 晋平，佐伯元司：ソースコード編集履歴の不吉な臭いの検出に向けて，ソフトウェアエンジニアリングシンポジウム 2014 予稿集，pp. 210–211 (2014).

[3] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999).

[4] Munro, M. J.: Product Metrics for Automatic Identification of “Bad Smel” Design Problems in Java Source-Code, *In Proceedings of the 11th IEEE International Software Metrics Symposium* (2005).

[5] Moha, N., he neuc, Y.-G. G., Duchien, L. and Meur, A.-F. L.: DECOR: A Method for the Specification and Detection of Code and Design Smells, *In Proceedings of the IEEE Transactions on Software Enginnering*, Vol. 36, No. 1, pp. 20–36 (2009).

[6] Herzig, K. and Zeller, A.: The Impact of Tangled Code Changes, *In Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 121–130 (2013).

[7] Kawrykow, D. and Robillard, M. P.: Non-Essential Changes in Version Histories, *In Proceedings of the 33rd International Conference on Software Engineering*, pp. 351–360 (2011).

[8] Hayashi, S. and Saeki, M.: Recording Finer-Grained Software Evolution with IDE: An Annotation-Based Approach, *In Proceedings of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution*, pp. 8–12 (2010).

[9] Yoon, Y. and Myers, B. A.: Caputuring and Analyzing Low-Level Events from the Code Editor, *In Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pp. 125–30 (2011).

[10] 大森隆行，丸山勝久：開発者による編集操作に基づくソースコード変更抽出，情報処理学会論文誌，pp. 2349–2359 (2008).

[11] Safer, I. and Murphy, G. C.: Comparing Episodic and Semantic Interfaces for Task Boundary Identification, *In Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 229–243 (2007).

[12] Omori, T. and Maruyama, K.: Identifying Stagnation Periods in Software Evolution by Replaying Editing Operations, *In Proceedings of the 16th Asia-Pacific Software Engineering Conference*, pp. 389–396 (2009).

[13] Kirinuki, H., Higo, Y., Hotta, K. and Kusumoto, S.: Hey! Are You Committing Tangled Changes?, *In Proceedings of the 22nd International Conference on Program Comprehension*, pp. 262–265 (2014).

[14] Nguyen, H. A., Nguyen, A. T. and Nguyen, T. N.: Filtering Noise in Mixed-Purpose Fixing Commits to Improve Defect Prediction and Localization, *In Proceedings of the 24th International Symposium on Software Reliability Engineering*, pp. 168–178 (2011).

[15] Dias, M., Bacchelli, A., Gousios, G., Cassou, D. and Ducasse, S.: Untangling Fine-Frained Code Changes, *In Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (2015).