社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

JavaのStream APIに対する並び替えリファクタリングの提案

田中　紘都†　　柗本　真佑†　　楠本　真二†

† 大阪大学大学院情報科学研究科
大阪府吹田市山田丘 1–5
E-mail: †{h-tanaka,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　Java の Stream は複数要素に対する処理を実装する記法である．このような複数要素の操作は，プログラム全体の実行時間の大半を占め，さらに頻繁に行われる処理である．そのため Stream における処理の流れ，つまり API の呼び出し順序はプログラムの性能に影響を与える要因となる．そこで本研究では，プログラム実行速度の向上を目的とした，Stream API の呼び出し順を並び替えるリファクタリング手法を提案する．提案手法では，Stream API の性質と依存関係という 2 つの観点から各 API の組み合わせにおける入れ替え可能性を，API の性質とステートフルな API かどうかの 2 点から実行速度面で適切な API の呼び出し順を体系化する．提案手法の有用性を評価する実験として，実際のプロジェクトにおいて誤用，つまり性能が低下する呼び出し順の事例がどの程度存在するかを調査した．結果として，可換な Stream API の組み合わせを使用している事例のうち，約 20%の事例において誤用が存在した．また，本手法を実際のプロジェクトにおける 8 件の誤用事例に適用し，API の並び替えによる性能変化を調査することで提案手法の有効性を評価した．実験結果として，いずれの事例においても実行速度の向上が見られ，最大で 126ms の実行時間が短縮された．

キーワード　Java, Stream API，リファクタリング，性能，可換性

1. はじめに

様々なプログラミング言語が，複数要素の処理を行う Stream を導入しており，Java も 2014 年に Stream API を取り入れた [1]．Java が Stream を導入したことで，開発者にとって様々な利点がある [2], [3]．例えば，Stream ではメソッドチェーン形式で API を呼び出すことで一連の処理を実装できる．これにより，手続き型 [4] でなく，宣言型 [5] の形で処理を簡潔に記述できる．また，Stream による処理の並列化を行う場合は，Stream を生成する API を `stream` から `parallelStream` に置き換えるだけであり，並列処理の容易な実装が可能となる．

しかし，Stream API を呼び出す順序はプログラムの性能に影響を与える要因である．Stream において，API の呼び出し順は処理の流れを示している．そのため，API の順序によっては Stream における処理が非効率的になる．また，Stream のような複数の要素に対して繰り返し操作を行う処理の実行時間は，プログラム全体の実行時間における大半を占める [6]．加えて，複数要素の処理はプログラム中で頻繁に行われる処理であることから [2]，Stream で行われる処理はプログラム全体の性能に影響を及ぼす．

ただし，API の呼び出し順は常に入れ替えられるわけではない．API の組み合わせによっては，入れ替える事によってプログラムの振る舞いが変化する場合や，各 API の返り値の型によってはそもそも入れ替えができない場合などが存在する．そのため，入れ替え可能な API の組み合わせにおいて，どのような呼び出し順序が性能面で適切かを体系化する必要がある．

そこで本研究では，プログラムの振る舞いを保ちつつ，Stream API の呼び出しを性能が向上する順序に並び替える手法を提案する．提案手法は，プログラムの外部的な振る舞いを変えずにソースコードを改善するという観点から，一種のリファクタリング手法 [7] であると言える．手法の提案にあたって，まず Stream API における可換性，つまり入れ替え可能な組み合わせと，各組合せにおける実行速度の面で適切な呼び出し順を体系化する．可換な API の組み合わせは，API の性質と依存関係の 2 つの観点から体系化し，適切な呼び出し順の体系化は，API の性質に加え，各 API がステートフルかどうかによって行う．

体系化した適切な呼び出し順を基に，実際の Java プロジェクト 1,000 件を対象に，誤用，つまり性能面で適切でない呼び出し順の事例がどの程度存在するかを調査した．調査結果として，可換な Stream API の組み合わせが使用されている総事例数 397 件のうち，約 20%にあたる 81 件で誤用されていた．また，自動的に API の呼び出し順をリファクタリングするツールを作成し，実際の誤用事例に適用することで，リファクタリングによる性能の変化を調査した．8 件の誤用事例に対しツールを適用した結果，いずれの事例においても実行速度は向上し，最大で 126ms の実行時間が短縮された．

```
1 List<Student> students = getStudentsList();
2
3 students.stream()
4         .sorted(comparing(Student::getEnglishScore))
5         .filter(s -> s.getEnglishScore() >= 60)
6         .forEach(s -> System.out.println(s));
```

図 1: Stream の使用例

```
1 List<Student> students = getStudentsList();
2
3 students.stream()
4         .filter(s -> s.getEnglishScore() >= 60)
5         .sorted(comparing(Student::getEnglishScore))
6         .forEach(s -> System.out.println(s));
```

図 2: 図 1 のソースコードを改善した例

2. 研 究 動 機

研究動機の説明に先立って，まず図 1 に示す例を用いて Java における Stream の説明を行う．図 1 の例では，まず 1 行目において `Student` オブジェクトの List を取得する．取得した List から，3 行目の `stream` によって Stream を生成する．生成した Stream に対して 4 行目の `sorted` により，Stream の要素をソートする．図 1 に示す例の場合は，`sorted` の引数に `comparing` が与えられており，`comparing` の引数に渡される値によって要素が並び替えられる．例では `comparing` の引数にメソッド参照の形式で値が与えられており，Stream を構成する各 `Student` オブジェクトに対して `getEnglishScore` を適用した結果が与えられる．つまり，`sorted` は英語の点数に関して昇順で要素を並び替えた Stream を出力する．続いて 5 行目の `filter` によって，条件に合致する要素からなる Stream を出力する．`filter` の引数にはラムダ式が与えられており，ラムダ式の左辺に Stream の要素が，ラムダ式の右辺に `filter` における条件が示されている．例の場合，`filter` は英語の点数が 60 点以上の学生を表す `Student` オブジェクトからなる Stream を出力する．最後に，6 行目の `forEach` によって Stream の全要素，図 1 の場合，英語の点数が 60 点以上の学生を英語の点数順に標準出力する．

このように，Java における Stream では複数の API をメソッドチェーンの形式で呼び出すことで，一連の処理を実装することができる．また，`filter` や `sorted` のような API は中間操作を行う API と呼ばれ，Stream において 0 個以上呼び出すことができる．ここで中間操作とは，Javadoc [8] において，ある Stream を別の Stream へ変換する操作と定義されている．中間操作に加えて，Stream による処理の結果を集約する操作として終端操作が Javadoc において定義されている．終端操作を行う API は Stream において 1 つだけ呼び出すことができ，図 1 における `forEach` が終端操作を行う API の 1 つである．

次に，研究動機となる例を図 2 に示す．図 2 に示す例は，図 1 における 4 行目と 5 行目の API の呼び出し順序を入れ替えたものである．この API 呼び出し順の入れ替えにより，Stream における性能は向上する．例えば，図 1 の `stream` によって生成された Stream の要素数が 100 個であり，5 行目で呼び出される `filter` によって Stream 中の要素数が半分の 50 個に減ると仮定する．この時，4 行目の `sorted` において並び替えの対象となる要素数は 100 個となる．言い換えれば，5 行目の `filter` によって除外され，最終的な Stream の出力結果には表れない要素も含めて並び替える必要がある．一方で図 2 では，4 行目の `filter` によって要素数を 50 個に絞った後，5 行目の `sorted` によって要素を並び替えている．この時，`sorted` が並び替えるべき要素数は 50 個のみとなる．このことから，図 2 に示すプログラムは，図 1 と比較すると性能が向上する API の呼び出し順である事が分かる．

しかし，図 1 のような呼び出し順のプログラムはコンパイルエラーの発生もなく，Stream における一連の処理の結果も図 2 のプログラムと等価である．つまり，プログラムの振る舞いは変わらない．そのため，性能が低下する API 呼び出し順を開発者が発見するには，目視による確認や，API の呼び出し順を入れ替えて性能差を比較するなどの方法を行う必要がある．

3. 提 案 手 法

3.1 概　　要

本節では，Stream API の呼び出し順序を並び替えるリファクタリング手法について説明する．まず，本研究で提案する手法において対象とする Stream API を選定し（3.2 節），選んだ Stream API について可換な API の組み合わせを体系化する（3.3 節）．可換な組み合わせは，各対象 API の性質（3.3.1 節）と依存関係（3.3.2 節）の 2 つの観点を基に導出する（3.3.3 節）．続いて，可換な API の組み合わせにおいて性能が向上する呼び出し順序を，API の性質とステートフルな API かどうかの 2 点から体系化する（3.4 節）．体系化した可換性と適切な呼び出し順序から，自動的に API の呼び出し順序をリファクタリングするツールを実装する（3.5 節）．

3.2 リファクタリングの対象となる Stream API を選定

Java 10 において実装されている 29 個の API のうち，本研究では，中間操作を行う 9 個の API を対象とする．対象外である API は，`of` や `generate` などの Stream を生成する API や終端操作を行う API である．これらの API は，いずれも呼び出し位置が決まっていることから，API の呼び出し順を入れ替えるリファクタリング手法では扱わない．また，中間操作 API の中でも，`peek` はデバッグ用の API である事が Javadoc に明記されているため，性能の向上を目的とした本研究のリファクタリング手法からは対象外とする．

3.3 Stream API の可換性を体系化

表 1 に, 対象とする 9 個の Stream API それぞれの性質と依存関係，各 API の分類を示す．性質と依存関係の定義，及び各 API の分類は，Javadoc [8] の記載内容を基に行った．表中の ✓ が，各 Stream API の持つ性質と依存関係を表している．表 1 に示された API の性質と依存関係について，それぞれ 3.3.1 と 3.3.2 において説明する．

3.3.1 Stream API における性質の定義と分類

表 1 の左側に示されている，Stream API の性質とその分類について説明する．Stream API の性質とは，API が入力された要素の何を変換し出力するかを示す．つまり，API がどのような処理を行うかを表す．Stream API の性質として，要素属性変換と要素数変換，要素順変換の 3 つを定義する．要素属性変換とは，Stream を構成する要素の値や型を変える性質であり，要素数変換は Stream を構成する要素の総数を変換する性質である．また，要素順変換とは，Stream を構成する要素の順序を入れ替える性質である．例として，`filter` の性質を考える．`filter` は入力要素の中で条件に合致する要素のみを出力する API である．そのため，要素数変換の性質を持つと言える．

3.3.2 Stream API における依存関係の定義と分類

表 1 の右側に示される，各 Stream API の依存関係とその分類を説明する．Stream API における依存関係とは，ある API の呼び出し順を入れ替えることで Stream を通しての振る舞いが変化する場合，その API への入力要素の何が Stream の振る舞いと依存関係があるかを示している．Stream API における依存関係として，要素属性依存と要素数依存，要素順依存の 3 つを定義する．要素属性依存とは，API の呼び出し順を入れ替えることで入力要素の値や型が変化した場合に，Stream における処理の出力結果が変わる事を示す．同様に，要素数依存とは，Stream を構成する要素数の変化によって，要素順依存とは，Stream 中の要素順が変わることによって出力結果が変化する事を示す．例えば `filter` の場合は，入力要素の値や型が変わることで条件に合致する要素が変わり，`filter` の出力結果が変わる．その結果，Stream 全体の処理における出力も変化する．このことから，`filter` は要素の属性に依存関係を持つと言える．一方で，呼び出し順序を入れ替えることで `filter` への入力要素の数や順序が変化した場合でも条件に合致する要素は変わらない．つまり，Stream の出力結果は変わらない．そのため，`filter` は要素数や要素順への依存関係を持たない．

3.3.3 Stream API の可換性を導出

API の性質と依存関係より，2 つの API の組み合わせが可換であるとは，どちらの API の性質も他方の API の依存関係に作用しない状態と定義する．一方で，少なくともどちらか一方の API の性質が他方の API の依存関係に作用する場合，この 2 つの API の組み合わせは可換でないと言える．

図 3 に，表 1 から Stream API の可換性を導出する過程の一例を示す．Stream API の組み合わせが可換であるとは，一方の API の性質が他方の依存関係に作用しないことである．つまり，表 1 における各 API の性質と依存関係を重ね合わせ，✓が重ならなければ可換であると示される．図 3 (a) に `filter` と `sorted` の可換性を導出する例を示す．`filter` と `sorted` の場合，`filter` の性質と `sorted` の依存関係を重ね合わせた際に✓が重ならないことから，`filter` の性質は `sorted` の依存関係に作用しないことが分かる．加えて，`sorted` の性質と `filter` の依存関係においても✓が重ならないため，`sorted` の性質も `filter` の依存関係に作用しないことが示される．このことから，`filter` と `sorted` は可換であると導出される．一方で，図 3 (b) に示す `filter` と `map` の場合，`map` の性質と `filter` の依存関係において✓が重なることから，`filter` と `map` の組み合わせは可換でないことが導出される．

表 2 に，表 1 から導出される Stream API の可換性を示す．表において，各行が可換な Stream API の組み合わせを示しており，#が各組合せの番号を表している．表の結果から，9 個の対象 API から 5 組の可換な組み合わせが導出された事が分かる．

3.4 Stream API の適切な呼び出し順を体系化

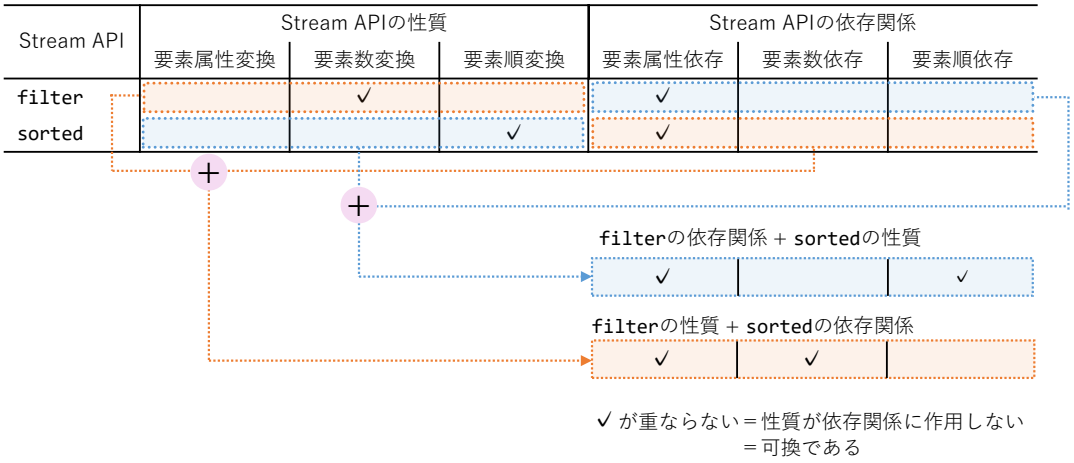
可換な API の組み合わせにおいて実行速度が向上する呼び出し順は，API の性質とステートフルな API かどうかを基に考える．ここで，ステートフルな API とは Javadoc で定義されている用語であり，入力要素中の 1 つの要素に対して処理を行う際に他の要素を参照する必要がある API を指す．一方で，他の要素を参照する必要がない API をステートレスな API と呼ぶ．可換な API の組み合わせに含まれる各 API のステート

表 2: Stream API の可換性

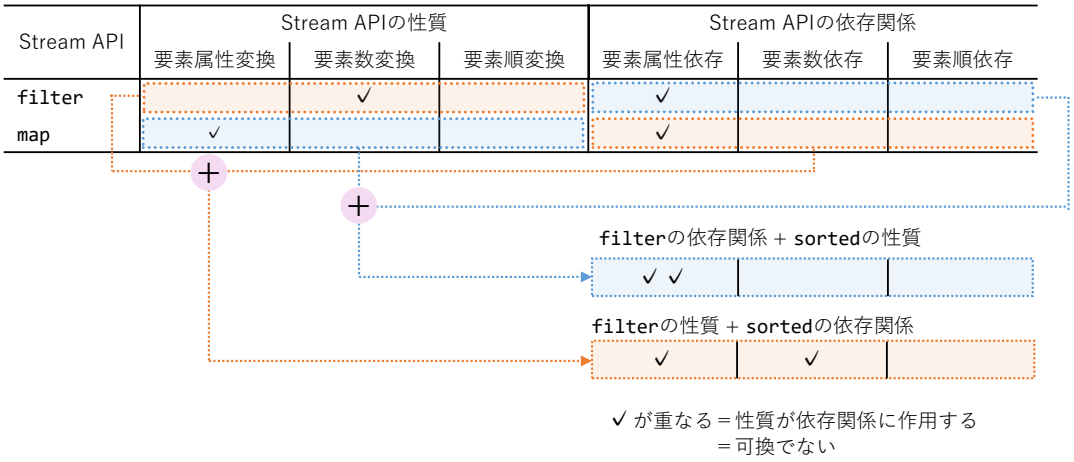
#	API の組み合わせ	
1	`filter`	`distinct`
2	`filter`	`sorted`
3	`map`	`limit`
4	`map`	`skip`
5	`sorted`	`distinct`

表 1: Stream API の性質と依存関係

Stream API	Stream API の性質			Stream API の依存関係		
	要素属性変換	要素数変換	要素順変換	要素属性依存	要素数依存	要素順依存
`filter`		✓		✓		
`map`	✓			✓		
`flatMap`	✓	✓		✓		
`distinct`		✓		✓		
`sorted`			✓	✓		
`limit`		✓			✓	✓
`skip`		✓			✓	✓
`takeWhile`		✓		✓	✓	✓
`dropWhile`		✓		✓	✓	✓



(a) 可換な組み合わせの導出例



(b) 可換でない組み合わせの導出例

図 3: 導出過程の例

フル/ステートレスへの分類を表 3 に示す．表 1 中の API の性質と表 3 から，表 2 の各組合せでの適切な呼び出し順を考える．

まず，API の性質から，要素数を変更する API を先に呼び出すことで速度が向上する．これは，要素数を先に減らすことで，後の処理における実行回数を減らすことができ，結果として実行速度の向上に繋がるためである．ただし，`sorted` と `distinct` の組み合わせについては例外的に，要素数を変更する `distinct` よりも要素順を変更する `sorted` を先に呼び出すことで性能が向上する．入力要素から重複を除く API である `distinct` は，内部で HashSet を生成することで処理を行う．しかし入力要素がソート済みであった場合は HashSet を生成することなく処理を行うため，入力要素をソートする API である `sorted` を先に呼び出すことで性能が向上する．このことから，表 2 における#2 の組み合わせは `filter` を，#3〜4 は `limit` と `skip` を，#5 は `sorted` を先に呼び出すべきであると分かる．

次に，要素数を変更する性質が同じであり，ステートフルな API とステートレスな API の組み合わせにおいては，ステートレスな API を先に呼び出すことで性能が向上する．これは，ステートレスな API で要素数を減少させることで，ステートフルな API で参照すべき要素数が減り，実行速度が向上するためである．このことから，表 2 における#1 の組み合わせは `filter` を先に呼び出すべきであると分かる．

表 3: Stream API のステートフル/ステートレスの分類

Stream API	ステートフル	ステートレス
`filter`	✓	
`map`	✓	
`distinct`		✓
`sorted`		✓
`limit`		✓
`skip`		✓
`takeWhile`		✓
`dropWhile`		✓

表 4: 適切な Stream API の呼び出し順

#	API の呼び出し順
1	`filter` ▸ `distinct`
2	`filter` ▸ `sorted`
3	`limit` ▸ `map`
4	`skip` ▸ `map`
5	`sorted` ▸ `distinct`

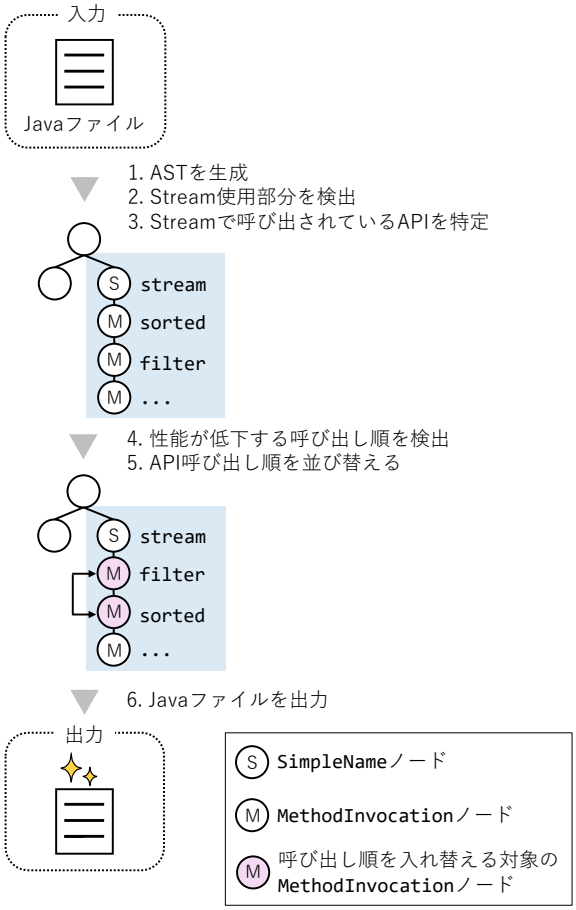


図 4: API 呼び出し順の自動リファクタリングツール概要

以上より，表 2 に示す各組合せにおける，性能が向上する API の呼び出し順を表 4 に示す．#は表 2 における#と対応している．各組合せにおける実行速度面で適切な呼び出し順が，`filter ▸ sorted` のように “▸” で連結されて表されており，“▸” の左辺を先に，右辺を後に呼び出す順序が性能面で最適な呼び出しである．以降も，“▸” を用いた表記により，適切な API の呼び出し順序を示す．

3.5 自動リファクタリングツールの実装

Stream API の可換性と適切な呼び出し順を基に，自動的に API 呼び出し順をリファクタリングするツールを作成した．作成したツールは，入力として Java ファイルを受け取り，リファクタリングを行った Java ファイルを出力する．ツールの概要を図 4 に示す．図中の各手順について説明する．

1. AST（抽象構文木）を生成する
   入力として受け取った Java ファイルから Eclipse JDT を用いて AST を生成する．
2. Stream 使用部分を検出
   生成した AST から Stream を使用している部分を検出する．Stream の検出は，変数名やメソッド名などの識別子名を表す SimleName ノードのうち，識別子名が `stream` かつ，`java.util.Stream` パッケージにバインディングされているノードの探索により行う．
3. Stream で呼び出されている API を特定
   検出された Stream において，使用されている API とその呼び出し順序の特定は，メソッド呼び出し部分を表す MethodInvocation ノードの探索により行う．
4. 性能が低下する呼び出し順を検出
   特定された API の呼び出し順と表 4 に示された適切な呼び出し順を照らし合わせ，性能が低下する呼び出し順となっている部分を検出する．
5. API 呼び出し順を入れ替える
   検出された呼び出し順において，MethodInvocation ノードを AST 上で入れ替えることにより呼び出し順の並び替えを行う．
6. Java ファイルを出力
   ノードを入れ替えた AST から Java ファイルを生成する．

表 5: 実際のプロジェクトにおける Stream API の誤用割合

Stream API の組み合わせ	正用と誤用の総事例数	誤用の事例数 (誤用割合)	誤用が存在するプロジェクト数
`filter ▸ distinct`	66	18 (0.27)	11
`filter ▸ sorted`	185	12 (0.06)	9
`limit ▸ map`	66	12 (0.18)	9
`skip ▸ map`	24	2 (0.08)	2
`sorted ▸ distinct`	54	37 (0.69)	21
total	395	81 (0.21)	35(注1)

4. 評価実験

4.1 提案手法の適用範囲

提案手法の有用性を評価するために，実際の Java プロジェクトにおいて，性能が低下する API の呼び出し順がどの程度存在するのかを調査する．

実験では，1,000 件の Java プロジェクトを対象とする．この実験対象は，GitHub 上の Java プロジェクトのうち，スター数上位かつ Stream を 1 回以上使用している Java プロジェクトである．実験ではまず，対象プロジェクト中の Java ファイルから AST を生成し，Stream 使用部分のコードスニペットを抽出する．具体的には，メソッドチェーン形式で複数の API が呼び出されている Stream の一連の処理を，コードスニペットとして抽出する．抽出したスニペット中における表 2 に示された可換な組み合わせが呼び出されている部分に着目し，呼び出し順序が性能面で適切なら正用として，適切でないなら誤用として事例数をそれぞれカウントする．

表 5 に調査結果を示す．表の各行は，1 つの組み合わせでの誤用と正用の事例数の和と誤用された事例数，また，正用と誤用の総事例数に対する誤用事例数の割合を誤用割合として示している．加えて，各組合せの誤用が存在したプロジェクト数を表記している．ただし，tatal 行のプロジェクト数は，重複したプロジェクトを 1 つとしてカウントしているため，プロジェクト数を示す列の合計値よりも小さな値となっている．

表の結果より，全体で 81 個（誤用割合 0.21）の誤用事例が 37

(注1)：重複プロジェクトは 1 つとカウントするため列の合計値より小さくなっている

のプロジェクト中に存在している．また，`sorted ▸ distinct` の組み合わせが最も誤用されており，誤用割合は 0.69 となっている．誤用割合の上位二つは `filter ▸ distinct` と `sorted ▸ distinct` であり，このことから `distinct` の呼び出し順序が誤用されやすいと言える．誤用が存在するプロジェクト数に着目すると，いずれの API の組み合わせにおいても複数のプロジェクトで誤用されていることが分かる．以上の結果より，提案手法はいずれの API の組み合わせにおいても適用でき，適用可能な範囲も様々なプロジェクトに存在すると言える．

4.2 提案手法による実行速度の変化

提案手法の有効性を評価するために，API の呼び出し順が誤用されている事例に対して提案手法のリファクタリングを行い，実行速度の変化を調査する．

実験対象は，4.1 の実験結果より誤用事例を含む 37 件の Java プロジェクトの中で，ビルドとテスト実行が可能であった 5 件のプロジェクトとする．実験では，Stream を含むメソッドに対応するテストの実行時間を計測する．まず，全テストを実行した際のスタックトレースから Stream を含むメソッドとテストの対応付けを行う．そして，誤用の場合と，呼び出し順を入れ替えた正用の場合それぞれでのテスト実行時間を計測する．

表 6 に，誤用と正用での実行速度を計測した結果を示す．各行が 1 つの誤用事例における実行速度の比較結果である．表中の誤用事例はそれぞれ，`filter ▸ distinct` の#1〜4 が swagger-core に，`filter ▸ sorted` #1 が bisq プロジェクトに，`sorted ▸ distinct` #1〜3 がそれぞれ jadx，micronaut-core，spring-framework に存在した事例である．誤用と正用での実行速度結果に加え，速度向上の値を算出しており，この値は誤用をリファクタリングすることで実行速度が何倍向上したかを示す，この速度向上の値は，誤用の実行速度を正用の実行速度で割ることで算出される．

表の結果より，いずれの誤用事例においてもリファクタリングにより実行速度の向上が確認できる．実行時間が極端に小さい `sorted ▸ distinct` #2 を除くと，`filter ▸ distinct` #1 と `filter ▸ sorted` #1 の事例は，いずれも速度向上が 1.5 倍以上で最も性能が向上した事例であると言える．また，`sorted ▸ distinct` #1 の事例に関しては，実行時間がリファクタリングにより 126ms 短縮されている．以上の結果より，提案手法を用いることでプログラムの実行速度向上に効果があると言える．一方で，表 6 の結果では誤用と正用ともに実行時間の値は小さく，リファクタリングによる速度向上の影響は少ないように見える．しかし，本実験ではテストの実行時間を計測し比較を行っているため，Stream を構成する要素の数が少ない数に限られていると考えられる．そのため，Stream の要素が多くなるほどリファクタリングによる速度向上の影響は大きくなると言える．

表 6: API 呼び出し順入れ替えによる実行速度比較

誤用事例	誤用での実行速度 [ms]	正用での実行速度 [ms]	速度向上 [倍]
`filter ▸ distinct` #1	12	8	1.50
`filter ▸ distinct` #2	45	37	1.22
`filter ▸ distinct` #3	28	27	1.04
`filter ▸ distinct` #4	16	14	1.14
`filter ▸ sorted` #1	32	21	1.52
`sorted ▸ distinct` #1	602	476	1.26
`sorted ▸ distinct` #2	2	1	2.00
`sorted ▸ distinct` #3	157	142	1.11

5. おわりに

本研究では，Stream API の呼び出し順序を，性能が向上する呼び出し順に並び替えるリファクタリング手法を提案した．評価実験として，実際のプロジェクトにおける性能が低下する呼び出し順の事例を調査し，提案するリファクタリング手法の有用性を確認した．また，リファクタリングによる性能変化を調査し，提案手法の有効性を確認した．

今後の課題として，各 API が実際のプログラム行っている処理内容による可換性及び適切な呼び出し順の体系化が考えられる．また，リファクタリング結果をプロジェクトの開発者にフィードバックし，実際の開発現場で受け入れられるかを評価する実験が考えられる．加えて，本研究では検証していない可換な Stream API の組み合わせについて，リファクタリングによる性能変化を調査する実験が考えられる．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：18H03222）の助成を得て行われた．

文　献

[1] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis, “Streams a la carte: Extensible pipelines with object algebras,” European Conference on Object-Oriented Programming, pp.591–613, 2015.

[2] V. Subramaniam, Functional Programming in Java, Pragmatic Bookshelf, 2014.

[3] R. Warburton, “Java 8 lambdas: Pragmatic functional programming,” chapter 1, O’Reilly Media, 2014.

[4] A.I.J. Kunasaikaran, “A brief overview of functional programming languages,” electronic Journal of Computer Science and Information Technology, vol.6, no.1, pp.32–36, 2016.

[5] J.C. Gonzalez-Moreno, M.T. Hortala-Gonzalez, F.J. Lapez-Fraguas, and M. Rodriguez-Artalejo, “An approach to declarative programming based on a rewriting logic,” The Journal of Logic Programming, vol.40, no.1, pp.47–87, 1999.

[6] J. Shirazi, “Java performance tuning,” chapter 7, O’Reilly Media, 2003.

[7] M. Fowler, “Refactoring: Improving the design of existing code,” chapter 2, Addison-Wesley Professional, 2018.

[8] “Stream (java se 10 and jdk 10),” (accessed December 16, 2019). `https://docs.oracle.com/javase/10/docs/api/java/util/stream/Stream.html`