修士学位論文

題目

Java Stream API の性能向上を目的とした並び替えリファクタリング

指導教員

楠本 真二 教授

報告者

田中 紘都

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

令和元年度 修士学位論文

Java Stream API の性能向上を目的とした
並び替えリファクタリング

田中 紘都

内容梗概

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．

主な用語

Java, Stream API，リファクタリング，性能，可換性

目次

図目次

表目次

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 [4–6] でなく，宣言型 [7–9] 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，従来の実装方法における並列化の困難さ [10–12] が解決されている．

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 [13,14]．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 [15] 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．

提案手法の効果を確認するために，100 万行のテキストファイルに対して入れ替えによる実行速度変化を調査した．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．誤用事例に対して開発者に GItHub 上でフィードバックを送信した．結果として，6 件の Pull Request のうち，2 件がマージされ，1 件がマージされずに閉じられた．

```
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` オブジェクトを要素に持つ 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` の引数にメソッド `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の形式で条件が与えられている．ラムダ式の左辺に Stream を構成する各要素が，ラムダ式の右辺には `filter` における条件が示されている．例における `filter` の条件は，各 Student オブジェクトに対して `getEnglishScore` メソッドを適用し，返り値の値が 60 以上の要素のみを出力している．つまり，`filter` によって英語の点数が 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 filter の出力結果である 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 Javadoc [16] で定義されており，Stream の一連の処理において 0 個以上呼び出すことができる．Javadoc において，中間操作とは，入力として与えられた Stream を別の Stream へ変換し出力する操作と定義されている．中間操作に加えて，一連の中間操作を経た 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 の処理の中でメソッドチェーン形式における最後の位置に 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 に示す例では，図 1 における 4 行目 sorted と 5 行目 filter 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 における一連の処理による出力結果は変化しないが，処理に要する実行時間は短縮される．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 sorted における処理の実行速度が向上し，結果としてプログラム全体の実行速度向上に繋がる．

このような Stream の実行速度がプログラム全体に影響を与えると考えられる．一般的に，複数要素に対する処理に要する実行時間は，プログラム全体の実行時間の大半を占める [13]．Stream も複数の要素を処理することから，Stream における実行時間はプログラム全体の実行時間に関係する．以上のことから，Stream における処理の実行速度を向上させることは必要であると考えられる．

しかし，プログラム開発者が図 1 のような性能が低下する呼び出し順を見つけることは困難である．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．しかし，Stream のような複数の要素に対する処理はプログラム中で頻繁に実装されることが知られている [2]．加えて，Java における新機能の使用

は年々増加することから [17]，プログラムにおける Stream の出現回数も増加すると考えられる．以上のことから，目視による調査や実行速度の比較を行うべきプログラム箇所は増加し，これにより開発者が要すべき作業とその時間は多くなる．

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）．

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．表 1 に，対象となる API とその振る舞いの概要を示す．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．

表 1: 対象となる Stream API

Stream API	API の振る舞い
`filter`	条件に合致する要素を出力
`map`	要素を 1 対 1 変換
`flatMap`	要素を 1 対多変換
`distinct`	要素における重複を削除
`sorted`	要素を並び替え
`limit`	要素数を指定された値に切り詰める
`skip`	要素の先頭から指定された値分だけ破棄
`takeWhile`	先頭要素から条件を満たす間の要素を出力
`dropWhile`	先頭要素から条件を満たす間の要素を除外

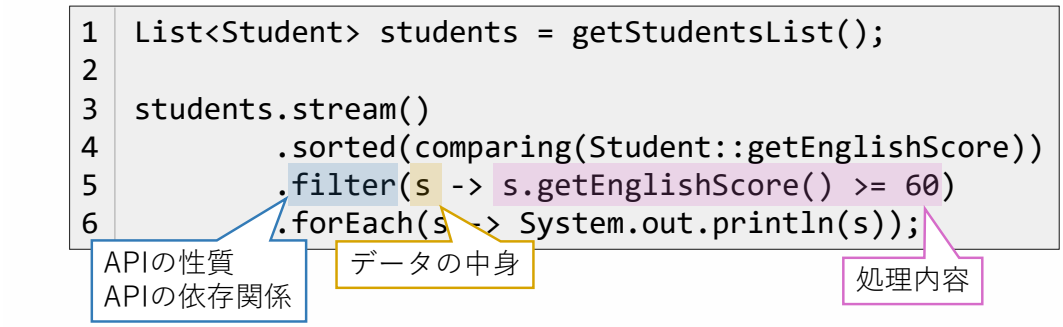


図 3: Stream API の可換性を体系化する基準

3.3 Stream API の可換性を体系化

図 3 に，Stream API の可換性を考える上での基準となる 3 つの観点を示す．1 つ目は，各 API がどのような振る舞いをするかを示す API の性質と依存関係である．2 つ目の基準は，ラムダ式の左辺に示されるような，Stream を構成する具体的なデータの中身である．最後に，ラムダ式の右辺に表される，API で行う具体的な処理内容が可換性を体系化する上での 3 つ目の基準となる．提案手法では，Stream API の可換性を API の性質と API の依存関係から体系化する．そのため，データの中身や具体的な処理内容については考慮していない．本研究では，開発者に対して呼び出し順の入れ替えを提案することを目的としている．そのため，ソースコードの静的解析によって検出が可能な API の性質と依存関係を対象とし，動的解析が必要となるデータの中身や処理内容には着目していない．

表 2 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 [16] 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 2 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 2 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

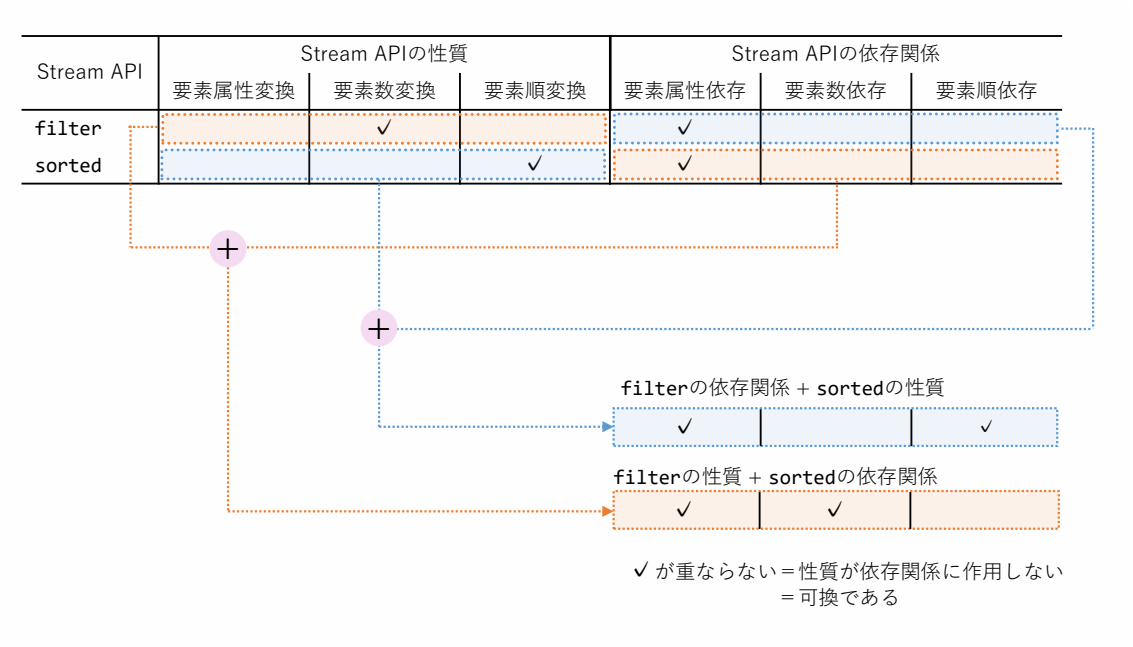
える性質である．例として，filter の性質を考える．filter は入力要素の中で条件に合致する要素のみを出力する API である．そのため，要素数変換の性質を持つと言える．

3.3.2 Stream API における依存関係の定義と分類

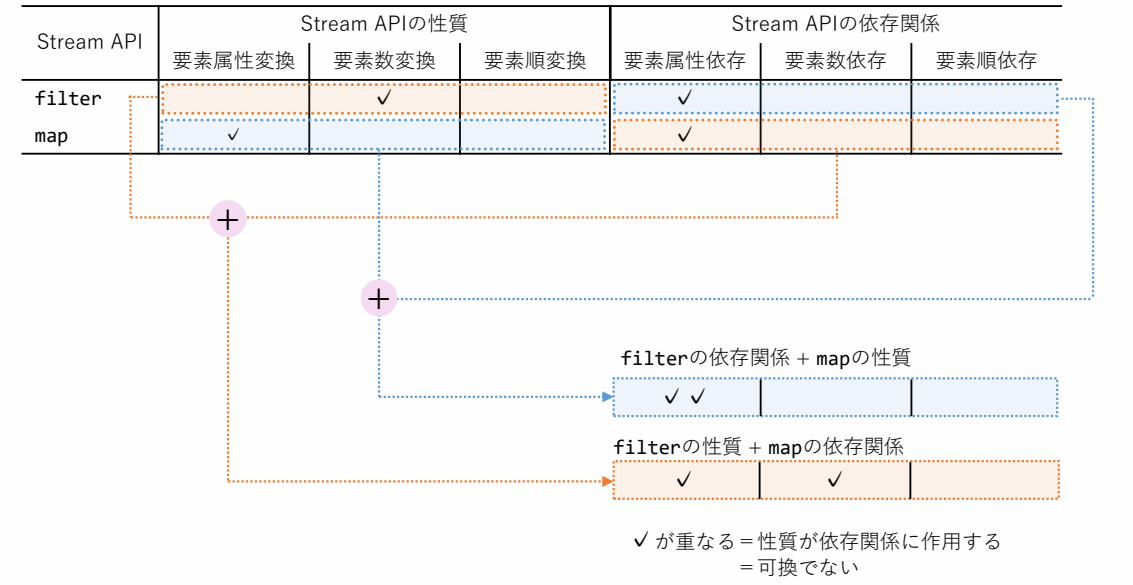
表 2 の右側に示される，各 Stream API の依存関係とその分類を説明する．Stream API における依存関係とは，ある API の呼び出し順を入れ替えることで Stream を通しての振る舞いが変化する場合，その API への入力要素の何が Stream の振る舞いと依存関係があるかを示している．Stream API における依存関係として，要素属性依存と要素数依存，要素順依存の 3 つを定義する．要素属性依存とは，API の呼び出し順を入れ替えることで入力要素の値や型が変化した場合に，Stream における処理の出力結果が変わる事を示す．同様に，要素数依存とは，Stream を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．

表 2: 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 4: 導出過程の例

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 4 に，表 2 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 2 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 4 (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 4 (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 3 に，表 2 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．

表 3: 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`

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 のステートフル/ステートレスへの分類を表 4 に示す．表 2 中の API の性質と表 4 から，表 3 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 [18]．このことから，表 3 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 3 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 4: 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`		✓

以上より，表 3 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 5 に示す．#は表 3 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．また，性能が向上する呼び出し順，つまり “▸” を用いて表される呼び出し順を正用，性能が低下する呼び出し順を誤用と以降では呼ぶこととする．

表 5: 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`

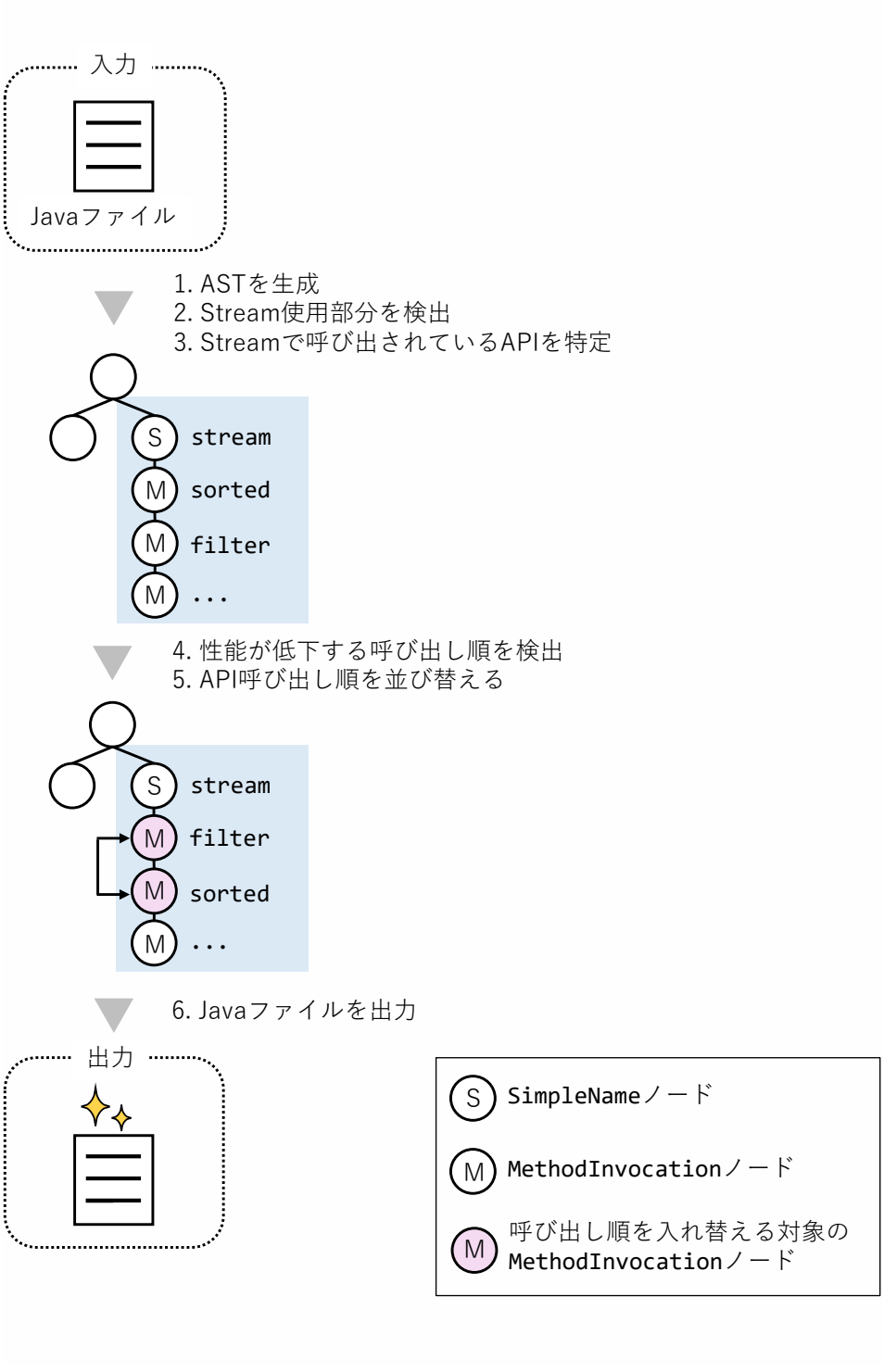


図 5: API 呼び出し順の自動リファクタリングツール概要

4　自動リファクタリングツールの実装

3 章に記述した 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，API 呼び出し順を入れ替える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 5 に示す．図中の各手順について説明する．

1. AST を生成する

   入力として受け取った Java ファイルに対して Eclipse JDT [19] を用いたソースコードの静的解析を行う．静的解析の結果として，プログラムの構造を表現する抽象構文木（AST）を生成する．

2. Stream 使用部分を検出

   生成した AST から Stream を使用しているソースコード箇所を検出する．まず，AST 中から変数名やメソッド名などの識別子名を表す `SimleName` ノードを検出する．検出した `SImpleName` 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を探索することで，Stream を検出する．

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を行う．まず，Stream に対応する `SimpleName` ノードと AST 上で親子関係にあるノードを検出する．検出したノードの中で，メソッド呼び出し部分を表す `MethodInvocation` ノードが Stream において呼び出されている API に対応している．検出した `MethodInvocation` ノードの識別子名を調べることで，Stream で呼び出されている API を特定する．

4. 性能が低下する呼び出し順を検出

   各 Stream において呼び出されている一連の API に対して，呼び出し順が隣り合う 1 組の API と表 5 に示された適切な呼び出し順を照らし合わせる．照合により，性能が低下する呼び出し順を検出する．

5. API 呼び出し順を入れ替える

   検出された性能が低下する呼び出し順に対して，`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．以降，手順 4 と 5 を繰り返し，各 Stream における呼び出し順をリファクタリングする．

6. Java ファイルを出力

   リファクタリングにより `MethodInvocation` ノードを入れ替えた AST を基に，テキスト形式で

ソースコードを出力する．出力したソースコードをリファクタリング対象の Java ファイルに出力することで API の呼び出し順を入れ替える．

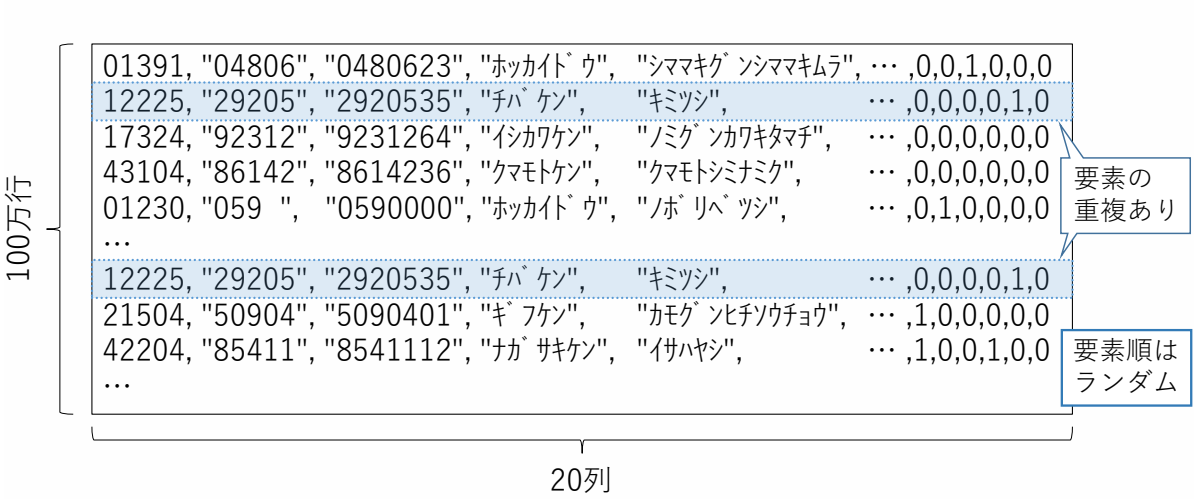


図 6: 実験対象とするテキストファイルの詳細

5 評価実験

5.1 提案手法によるプログラム実行速度向上の効果を確認

3 節にて体系化した呼び出し順について，誤用な呼び出し順のリファクタリングによる実行速度の向上を確認する．実験対象となる Stream は，図 6 に示すような，郵便番号のデータ [20] を基に作成したテキストファイルの各行を要素として持つ．具体的には，各要素は 20 個のフィールドを持ち，要素数は 100 万個で構成されている．また，要素の順序はランダムであり，要素の重複も存在する．

実行速度の向上を確認するために，表 5 に示される API の各組合せに対して，正用と誤用それぞれの場合における実行時間を計測し比較する．本実験における，表 5 中に存在する各 API の具体的な振る舞いを表 6 に示す．`filter` と `map` については，振る舞いに対応する処理をラムダ式として表し，引

表 6: 実験における各 Stream API の振る舞い

Stream API	API の振る舞い
`filter`	正規表現に合致する文字列を含む要素を出力
`map`	各要素の 5 列目の値のみを出力
`distinct`	要素の重複を削除
`sorted`	各要素を郵便番号順に並び替える
`limit`	要素数を 50 万個に切り詰める
`skip`	要素の先頭から 50 万個を破棄

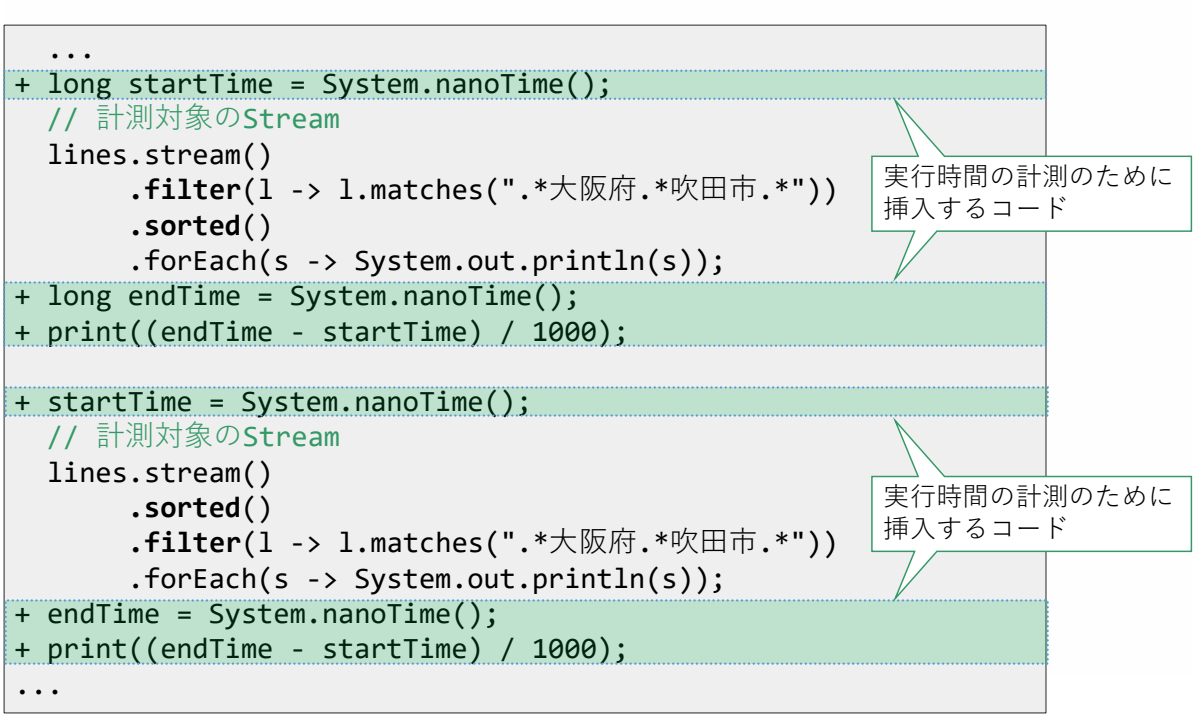


図 7: 実行速度計測の例

数に渡している．また，`distinct` と `sorted` には引数は与えておらず，`limit` と `skip` は切り詰めるもしくは破棄する要素数を引数に与えている．

具体的な実行時間の計測方法の例を図 7 に示す．まず，正用と誤用それぞれの呼び出し順で Stream の処理を記述する．この時，各 API に与える引数，つまり振る舞いはどちらの呼び出し順序でも同じである．各呼び出し順での Stream の前後に実行時間を計測するソースコードを挿入し，正用と誤用それぞれでの実行時間を計測する．

実行時間の計測結果を表 7 に示す．表の各行は可換な API の組み合わせにおける誤用での実行時

表 7: 実行時間の計測結果

Stream API の組み合わせ	誤用での実行時間 [ms]	正用での実行時間 [ms]	時間短縮 [倍]
`filter ▸ distinct`	1,755	1,238	0.71
`filter ▸ sorted`	4,026	2,852	0.71
`limit ▸ map`	752	683	0.91
`skip ▸ map`	718	659	0.92
`sorted ▸ distinct`	4,781	3,076	0.64

間と正用での実行時間，リファクタリングによって実行時間が何倍短縮されたかを示している．表の結果より，いずれの組み合わせにおいてもリファクタリングによって実行時間は短縮している．また，`sorted ▸ distinct` の場合は実行時間が最も短縮されており，1,705[ms]，0.64[倍] 実行時間が短くなっている．以上の結果より，3 節にて体系化した正用の呼び出し順に並べ替えることにより，実行時間の短縮，つまりプログラム実行速度の向上が確認された．

5.2 実際の Java プロジェクトに対する実行速度向上の効果を調査

5.2.1 実際のプロジェクトにおける誤用の割合

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，つまり誤用の事例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 3 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 8 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 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

表 8: 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[*1]

[*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

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．

5.2.2 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，5.2.1 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 6 件のプロジェクトとする．表 9 に，対象プロジェクト中に存在する誤用事例を示す．表の各行は 1 つの誤用事例に対応しており，事例における API の組み合わせと存在するプロジェクトを示している．加えて，調査対象としたソースコードの版に対応する，GitHub におけるコミットを表すハッシュ値と，誤用を含んでいる Java ファイルを表している．

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．

図 8，9 に，計測対象となる Stream と，Stream に対応するテストケースにおける実行時間計測の例を示す．まず，図 8 の Stream において `distinct` と `sorted` が誤用，つまり性能が低下する呼び出し順となっている．そのため，図の Stream はリファクタリング対象であり，本実験における実行速度を計測する対象の Stream となる．対象の Stream を実行しているテストケースが図 9 に示されるソースコードである．例に示すソースコードでは，テストケース中の 2 つの `assertThat` のみが図 8 の Stream を実行している．そのため，対象 Stream を実行している `assertThat` の前後に実行時間を計測するためのコードを挿入することで，正用へのリファクタリング前後での実行時間を計測する．

表 9: 実行時間の計測対象

# 事例	Stream API の組み合わせ	プロジェクト	commit	Java ファイル
1	`filter` ▸ `distinct`	swagger-core	5a215	ModelResolver.java
2	`filter` ▸ `distinct`	swagger-core	5a215	ModelResolver.java
3	`filter` ▸ `distinct`	swagger-core	5a215	ModelResolver.java
4	`filter` ▸ `distinct`	swagger-core	5a215	ModelResolver.java
5	`sorted` ▸ `distinct`	bisq	32244	CurrencyUtil.java
6	`sorted` ▸ `distinct`	jadx	1bb90	ClassGen.java
7	`sorted` ▸ `distinct`	micronaut-core	84442	RxJavaHealthAggregator.java
8	`sorted` ▸ `distinct`	spring-framework	eeaae	RestTemplate.java
9	`sorted` ▸ `distinct`	junit5	97872	LockManager.java

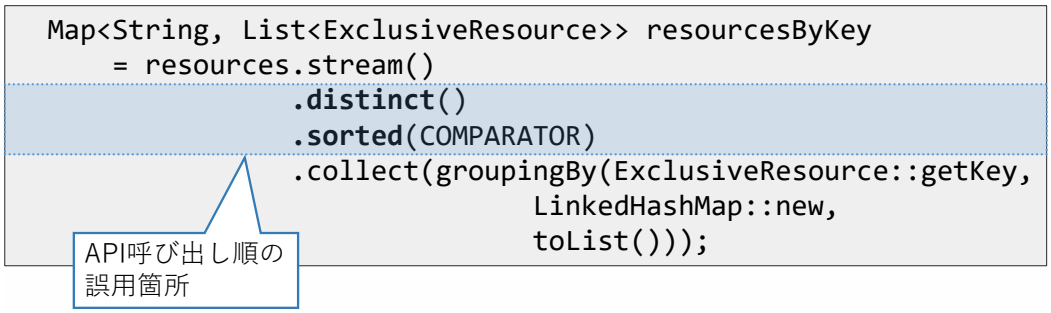


図 8: 実行時間計測の対象となる Stream の例

```
@Test void reusesSameLockForExclusiveResourceWithSameKey() {
  Collection<ExclusiveResource> resources
            = singleton(new ExclusiveResource("foo", READ));
  List<Lock> locks1 = getLocks(resources,
                                    SingleLock.class);
  List<Lock> locks2 = getLocks(resources,
                                    SingleLock.class);

+  long startTime = System.nanoTime();

  assertThat(locks1).hasSize(1); // 対象Streamを実行
  assertThat(locks2).hasSize(1); // 対象Streamを実行

+  long endTime = System.nanoTime();
+  print((endTime - startTime) / 1000);

  assertThat(locks1.get(0)).isSameAs(locks2.get(0));
}
```

実行時間の計測のために
挿入するコード

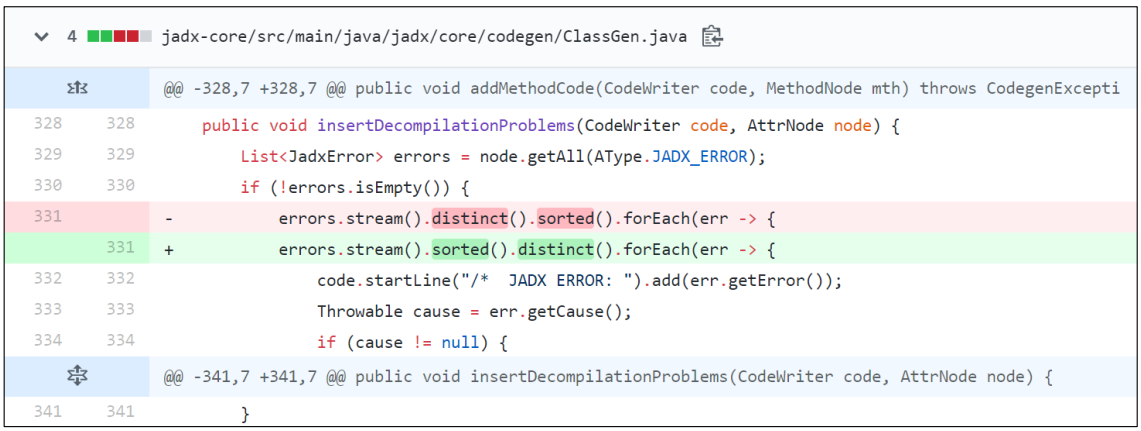
図 9: 計測対象の Stream に対応するテストケースの例と実行時間の計測方法

表 10 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，#1〜4 が swagger-core に，#5 が bisq プロジェクトに，#6〜9 がそれぞれ jadx，micronaut-core，spring-framework, junit5 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#7 を除くと，#5 の事例は，実行時間の短縮が 0.66[倍] で最も性能が向上した事例であると言える．また，#6 の事例に関しては，実行時間がリファクタリングにより 126[ms] 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 10 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おり，表 10 に示すように，Stream を構成する要素の数が高々 32 個に限られていることが原因であると考えられる．また，Stream の要素数が増加するにしたがって短縮される実行時間の秒数も大きくなることから，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 10: API 呼び出し順入れ替えによる実行速度比較

# 事例	Stream API の組み合わせ	Stream の要素数	誤用での実行時間 [ms]	正用での実行時間 [ms]	時間短縮 [倍]
1	`filter ▸ distinct`	15	52	43	0.83
2	`filter ▸ distinct`	15	45	37	0.82
3	`filter ▸ distinct`	10	28	27	0.96
4	`filter ▸ distinct`	10	25	20	0.80
5	`sorted ▸ distinct`	10	32	21	0.66
6	`sorted ▸ distinct`	32	602	476	0.79
7	`sorted ▸ distinct`	5	2	1	0.50
8	`sorted ▸ distinct`	23	157	142	0.90
9	`sorted ▸ distinct`	20	243	201	0.83



(a) Pull Request に示される修正内容の例

We executed this command before and after re-ordering :

```
time ./gradlew -Dtest.single=TestInlineVarArg test
```

and the result is :

```
the order of Stream API      test running time
================================================
distinct().sorted()                10.640s
sorted().distinct()                10.610s
```

(b) Pull Request への記述例

図 10: Pull Request の例

5.2.3 修正内容と実行速度の向上を開発者へ報告

本研究で提案するリファクタリング手法の有用性を評価する実験として，リファクタリングによる修正内容と実行速度の向上を開発者に報告し，修正が採用されるかを調査した．実験の対象となるプロジェクトは，節と同様の 6 件のプロジェクトとする．

開発者への報告は，GitHub 上の Pull Request 機能を用いて行う．Pull Request による具体的な報告例を図 10 に示す．リファクタリングによる修正内容は図 10a のように GitHub 上で表示される．図中の赤い囲み部分が修正前のソースコード，緑の囲み部分が修正後の様子を示している．同時に，図 10b のような形式で開発者に対して実行速度の向上を報告する．記述内容としては，修正の前後でどのようなコマンドを実行し，結果として実行時間がどう変化したかを示している．実行するコマンドは，修正を加えた Stream に対応するテストを実行しており，`test running time` に示される値の変化か

ら，修正前後の実行時間の変化が表されている．

実験の結果を表 11 に示す．表の各行が一つの報告事例と Pull Request へのリンク，2020 年 2 月 12 日時点での採用結果を示している．表の結果より，6 件の Pull Request のうち，2 件が採用され，1 件が不採用となった．また，残りの 3 件は採用でも不採用でもなく保留の状態となっている．spring-framework の開発者が不採用とした理由として，実行速度比較の実験結果が信頼できるものでなく，加えて速度の向上度合いが小さいことが挙げられた．

表 11: 送信した Pull Request とその採用結果

プロジェクト	Pull Request	採用結果
swagger-core	pull/3429	保留
bisq	pull/3921	採用
jadx	pull/839	保留
micronaut-core	pull/2700	採用
spring-framework	pull/24442	不採用
junit5	pull/2167	保留

6　妥当性への脅威

本研究では Stream API の性質と依存関係に着目して，可換性と呼び出し順を体系化している．しかし，API の振る舞いやデータの中身によっては，可換性や性能が向上する呼び出し順が本研究で示した結果とは異なる可能性がる．

また，実行速度の変化を計測する実験において対象となるプロジェクト数が少ないため，その他のプロジェクトに対して同様の実験を行った際には計測結果が変化する可能性がある．加えて，実際に実行時間を計測した Stream API の組み合わせが 2 種類のみであり，ほかの組み合わせについて他のプロジェクトで計測した場合には，本研究と異なる結果になる可能性がある．

7　関連研究

Java の Stream に関する研究は近年盛んになっている [21–24]．中でも，Stream の性能に関する研究として Khatchadourian らは，Stream によって効果的に並列処理を行うリファクタリング手法を提案している [21]．本研究では，Stream API の呼び出し順に着目し，呼び出し順を並び替えることで性能の向上を目指すリファクタリングする手法を提案している．また，ほぼ全ての Stream の事例で出現するラムダ式の，実際の開発現場での使用率や使用法に関する研究も存在する [22]．この研究結果から，ラムダ式の使用は年々増加している事が示された．そのため，今後 Stream の使用も増えていくと見られ，本研究で提案する手法が適用可能な範囲も拡大すると考えられる．

リファクタリング手法の中でも，プログラム性能の向上を目的とした手法に関して，様々な研究が行われてきた [25–44]．プログラム中の無駄な繰り返しを除去することで実行速度を向上させる手法として，Nistor らの研究がある [28]．この研究における提案手法では，繰り返し処理における出力結果に変化を与えない操作に対して，`break` 文や `return` 文の挿入，`if` 文の条件式を改変することで無駄な繰り返しをスキップしている．また，Han らはスタックトレースを解析することで性能が低下するソースコード箇所を特定する手法を提案している [33]．本研究では，Stream の実行速度を向上させるためのリファクタリング手法として，API の性質と依存関係，API がステートフルかどうかの 3 つの観点を基に可換性や適切な呼び出し順を体系化している．

複数要素に対する処理に関して様々な研究が行われてきた [45–47]．Java におけるイテレータに対して，割り込みを可能とすることでプ r ゴウラム性能を向上させる手法を Liu らは研究している [45]．また，Python においてイテレータを表現する Generator を最適化し，プログラム実行速度を向上させる手法が Zhang らによって提案された [47]．本研究では，複数要素の中でも比較的最近に Java に導入され，前述した通り今後使用が増加すると考えられる Stream に関するリファクタリング手法を提案している．

8　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．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

本研究を行うにあたり，理解あるご指導を賜り，常に励ましていただきました楠本真二教授に，心より感謝申し上げます．

本研究に関して，的確で丁寧なご助言を頂きました，肥後芳樹准教授に深く感謝申し上げます．

本研究の全過程を通して，研究に対する考え方や方向性など，丁寧かつ熱心なご指導を賜りました柗本真佑助教に，心より感謝申し上げます．

本研究を進めるにあたり，様々な形で励まし，ご助言を頂きましたその他の楠本研究室の皆様のご協力に深く感謝致します．

最後に，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心から御礼申し上げます．

参考文献

[1] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. Streams a la carte: Extensible pipelines with object algebras. In *European Conference on Object-Oriented Programming*, pp. 591–613, 2015.

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

[3] Richard Warburton. *Java 8 Lambdas: Pragmatic Functional Programming*, chapter 1. O'Reilly Media, 2014.

[4] Jagatheesan Kunasaikaran and Azlan Iqbal. A brief overview of functional programming languages. *electronic Journal of Computer Science and Information Technology*, Vol. 6, No. 1, pp. 32–36, 2016.

[5] Robert W Floyd. The paradigms of programming. *Communications of the Association for Computing Machinery*, Vol. 22, No. 8, pp. 455–460, 1979.

[6] Garry White and Marcos Sivitanides. Cognitive differences between procedural programming and object oriented programming. *Information Technology and Management*, Vol. 6, No. 4, pp. 333–350, 2005.

[7] Juan Carlos Gonzalez-Moreno, Maria Teresa Hortala-Gonzalez, Francisco J. Lapez-Fraguas, and Mario 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.

[8] Krzysztof R Apt. Logic programming. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, Vol. 1990, pp. 493–574, 1990.

[9] Krzysztof R Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-o: An imperative language that supports declarative programming. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 5, pp. 1014–1066, 1998.

[10] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 329–339, 2008.

[11] Mehdi Bagherzadeh and Hridesh Rajan. Order types: Static reasoning about message races in asynchronous message passing concurrency. In *ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pp. 21–30, 2017.

[12] Syed Ahmed and Mehdi Bagherzadeh. What do concurrency developers ask about? a large-scale study using stack overflow. In *ACM/IEEE International Symposium on Empirical Soft-*

ware Engineering and Measurement, pp. 1–10, 2018.

[13] Jack Shirazi. *Java Performance Tuning*, chapter 7. O'Reilly Media, 2003.

[14] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 127–139, 2009.

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*, chapter 2. Addison-Wesley Professional, 2018.

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

[17] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *International Conference on Software Engineering*, pp. 779–790, 2014.

[18] Java streams: How to do an efficient "distinct and sort"? `https://stackoverflow.com/questions/43806467/java-streams-how-to-do-an-efficient-distinct-and-sort`, (accessed December 16, 2019).

[19] Eclipse java development tools (jdt) — the eclipse foundation. `https://www.eclipse.org/jdt/`, (accessed December 16, 2019).

[20] 郵便番号データダウンロード - 日本郵便. `https://www.post.japanpost.jp/zipcode/download.html`, (accessed November 17, 2019).

[21] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. Safe automated refactoring for intelligent parallelization of java 8 streams. In *International Conference on Software Engineering*, pp. 619–630, 2019.

[22] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in java. *ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, Vol. 1, pp. 1–31, 2017.

[23] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *International Conference on Parallel Architecture and Compilation*, pp. 419–431, 2015.

[24] HaiTao Mei, Ian Gray, and Andy Wellings. Real-time stream processing in java. In *Ada-Europe International Conference on Reliable Software Technologies*, pp. 44–57, 2016.

[25] Wellisson GP da Silva, Lisane Brisolara, Ulisses B Correa, and Luigi Carro. Evaluation of the impact of code refactoring on embedded software efficiency. In *Workshop de Sistemas*

Embarcados, pp. 145–150, 2010.

[26] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. Relooper: refactoring for loop parallelism in java. In *ACM SIGPLAN conference companion on Object Oriented Programming Systems Languages and Applications*, pp. 793–794, 2009.

[27] Eli Tilevich and Yannis Smaragdakis. Binary refactoring: Improving code behind the scenes. In *International Conference on Software engineering*, pp. 264–273, 2005.

[28] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *International Conference on Software Engineering*, pp. 902–912, 2015.

[29] Suparna Bhattacharya, Mangala Gowri Nanda, Kanchi Gopinath, and Manish Gupta. Reuse, recycle to de-bloat software. In *European Conference on Object-Oriented Programming*, pp. 408–432, 2011.

[30] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. *ACM SIGPLAN Notices*, Vol. 47, No. 6, pp. 89–98, 2012.

[31] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *International Conference on Software Engineering*, pp. 156–166, 2012.

[32] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. Variability-aware performance prediction: A statistical learning approach. In *International Conference on Automated Software Engineering*, pp. 301–311, 2013.

[33] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering*, pp. 145–155, 2012.

[34] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing target prioritization via performance risk analysis. In *International Conference on Software Engineering*, pp. 60–71, 2014.

[35] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems implementations. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 17–26, 2010.

[36] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Meeting on Foundations of Software Engineering*, pp. 268–278, 2013.

[37] Jungju Oh, Christopher J Hughes, Guru Venkataramani, and Milos Prvulovic. Lime: A frame-

work for debugging load imbalance in multi-threaded execution. In *International Conference on Software Engineering*, pp. 201–210, 2011.

[38] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *International Symposium on Software Testing and Analysis*, pp. 13–25, 2014.

[39] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kastner, Sven Apel, Don Batory, Marko Rosenmuller, and Gunter Saake. Predicting performance via automated feature-interaction detection. In *International Conference on Software Engineering*, pp. 167–177, 2012.

[40] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *International Symposium on Software Testing and Analysis*, pp. 90–100, 2013.

[41] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology*, Vol. 23, No. 3, pp. 1–50, 2014.

[42] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *International Conference on Architectural Cupport for Programming Languages and Operating Systems*, pp. 193–206, 2014.

[43] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 67–76, 2012.

[44] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In *International Conference on Automated Software Engineering*, pp. 43–52, 2011.

[45] Jed Liu, Aaron Kimball, and Andrew C. Myers. Interruptible iterators. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 283–294, 2006.

[46] William Pugh. Uniform techniques for loop optimization. In *International Conference on Supercomputing*, pp. 341–352, 1991.

[47] Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. Accelerating iterators in optimizing ast interpreters. *ACM SIGPLAN Notices*, Vol. 49, No. 10, pp. 727–743, 2014.