OSSにおけるJavaのレコード・クラス利用実態の初期調査

杉原 裕太[1,a)]　近藤 将成[1,b)]　亀井 靖高[1,c)]　鵜林 尚靖[1,d)]

概要：今日広く利用されているプログラミング言語である Java は，円滑なコーディングを目的として新たな言語仕様を導入することがある．過去の Java 言語アップデートでもジェネリクスやラムダ式といった言語仕様が追加されており，これらがリファクタリングにどのように活用可能であるのか研究されてきた．そして 2021 年に追加された言語仕様であるレコード・クラスは，一部の型宣言を簡潔に行うことを可能とし，ソースコード記述量の削減に寄与すると考えられている．しかしながら，レコード・クラスに関して，リファクタリング上の恩恵を報告する研究はまだ行われていない．そこで本稿ではレコード・クラスのリファクタリング利用に関する初期調査として，GitHub 上の OSS におけるレコード・クラスの利用実態を評価した．その結果，データセットとして取得した 2000 件のレポジトリのうち，70 件のレポジトリで合計 3244 のレコード・クラスが定義されていることがわかった．また，コミットによるレコード・クラスの追加のうち，22.2%が既存クラスをレコード・クラスに変更するリファクタリングであることがわかった．

キーワード：Java，レコード・クラス，リファクタリング

1. はじめに

今日広く利用されているプログラミング言語である Java は，円滑なコーディングを目的として新たな言語仕様を導入することがある．過去の Java 言語アップデートでもジェネリクスやラムダ式といった言語仕様が追加されており，これらが開発者たちによってどのように利用されているのか，そしてコーディング作業上でどのような恩恵をもたらすのか調査が行われてきた [1][2]．そして 2021 年 3 月のアップデートである Java16 で，Java はレコード・クラス（以下レコードと記述）という新たな言語仕様を導入した．レコードはイミュータブルな値ベース・クラスの宣言を簡潔に行うことを可能とし，ソースコード記述量の削減に寄与すると考えられている．しかしながら，レコードに関して，開発者たちによる利用およびコーディング上の恩恵を報告する研究はまだ行われていない．そこで本稿ではレコードの利用に関する初期調査として，GitHub 上の OSS におけるレコードの利用実態を評価した．本稿における調査の目的として，次のようなものを掲げている．

- Java 言語を利用する開発者に対し，レコードの適切な利用を推進する．
- Java 言語の開発者に対し，言語仕様デザインのヒントを提供する．
- レコードを用いたリファクタリングの支援を行うツール開発にあたり，必要な知見を収集する．

そして，上記の目的に基づき，次の 4 つの RQ を設定している．

RQ1：レコードは OSS において，どの程度の数使用されているのか？

RQ2：使用されているレコードの特徴はどのようになっているか？

RQ3：クラスをレコードに変更するリファクタリングでは，どの程度の恩恵を享受できるのか？

RQ4：クラスからレコードへの変更を阻害する要因は何か？

本稿では，2 節でレコードの仕様を含めた背景と，動機について述べる．3 節でデータセットと手法について説明し，4 節で各 RQ に対する結果を述べる．最後に 5 節で妥当性の脅威，6 節でまとめを述べる．

[1] 九州大学
Kyushu University
[a)] sugihara@posl.ait.kyushu-u.ac.jp
[b)] kondo@ait.kyushu-u.ac.jp
[c)] kamei@ait.kyushu-u.ac.jp
[d)] ubayashi@ait.kyushu-u.ac.jp

```
1 record Point(int x, int y) { }
```

図 1　レコードの宣言

```
1  import java.util.Objects;
2
3  final class Point {
4      private final int x;
5      private final int y;
6
7      public Point(int x, int y) {
8          this.x = x;
9          this.y = y;
10     }
11
12     public int x() { return x; }
13     public int y() { return y; }
14
15     @Override
16     public boolean equals(Object o) {
17         if(o.getClass() == Point.class){
18             Point p = (Point) o;
19             return p.x == x && p.y == y;
20         }
21         return false;
22     }
23
24     @Override
25     public int hashCode() {
26         return Objects.hash(x, y);
27     }
28
29     @Override
30     public String toString() {
31         return "Point[x=%d,␣y=%d]".formatted(x,
               y);
32     }
33 }
```

図 2　図 1 と等価なクラス宣言

2. 背景と動機

2.1 レコードの仕様

レコードとは，不変な変数群を保持することを目的として作られた，Java における新たな型宣言のフレームワークである [3]．Java16 以前では，Java は通常クラス，列挙型，インタフェース，アノテーション型の 4 つの型宣言をサポートしていた．そして 2021 年 3 月の Java16 で（プレビュー期間を含むと Java14 から），5 つ目の型宣言であるレコードが登場した．レコードの宣言では，クラス宣言の際に用いるキーワード class の代わりに，文脈的キーワードの record を用いる．そして図 1 のように，レコード名の直後にヘッダ（レコードの保持するフィールドのリスト）を宣言する．これらによって次のメンバが暗黙的に定義され，利用できるようになる．

- ヘッダに対応する private かつ final なフィールド
- カノニカルコンストラクタ（ヘッダと同じ型の順番で引数をとり，対応する各フィールドを初期化するコンストラクタ）
- 各フィールドと同じ名前をもつゲッタメソッド
- 一定の規則に基づいて構成された toString，equals，hashCode メソッド

参考として**図 2** に，図 1 のレコードと等価であるクラス宣言を示す．レコードは，図 2 のような値ベース・クラスを簡潔に宣言することを可能とし，ソースコード記述量の削減に効果を発揮すると考えられる．また，ソースコードの読み手がレコードの仕様を理解している場合，レコード型宣言の提供するインタフェースを素早く理解することができるという恩恵もある．なお，暗黙的に定義されたメソッドおよびコンストラクタは，同じシグネチャでレコードの内部に改めて宣言することで，オーバーライドすることも可能である．

レコードの宣言には，通常クラスには存在しない次のような制約も課される [4]．

- クラスを継承できない．（インタフェースの実装は可．）
- クラスの継承元になれない．
- ヘッダとは別のインスタンスフィールドを宣言できない．
- インスタンスイニシャライザを宣言できない．
- カノニカルコンストラクタをオーバーライドしないコンストラクタは，最初に必ずカノニカルコンストラクタを呼び出さないといけない．

継承の制約やインスタンスフィールドの制約は，レコードの不変性を保証するためのものである．インスタンスイニシャライザとコンストラクタの制約は，レコードの初期化がカノニカルコンストラクタによって確実に行われることを保証するためのものである．

2.2 Java の言語仕様に対して過去に行われた調査

2.2.1 Java のジェネリクスに関する研究

Chris らは，Java5 で追加された言語仕様であるジェネリクスについて，OSS における利用の調査を行った [1]．ジェネリクスは，クラスあるいはメソッド中の型宣言に自由度を持たせる言語仕様であり，ダウンキャスト式などのアンチパターンの減少に効果を発揮するとされている．本関連研究では OSS の開発履歴を解析し，ジェネ

リクスのもたらす効果の検証とともに，利用数の変遷やジェネリクスに与えられやすい型引数などの統計を取得している．

データセットは，Java にジェネリクスが導入される以前に開始した 20 の OSS と，Java にジェネリクスが導入されたより後に開始した 20 の OSS である．結果として，アンチパターンの減少の側面からは，ジェネリクスの適用数の増加に伴って統計学的有意にダウンキャスト式の割合が減少していることがわかった．ジェネリクスの利用の側面からは，ジェネリクス導入以前に開始した OSS のうち 15 件で，ジェネリクス導入以降に開始した OSS のうち全てで，ジェネリクスが利用されていることがわかった．また，ジェネリクスの型引数としては，String 型が最も多く与えられていることがわかった．

2.2.2 Java のラムダ式に関する研究

Davood と Ameya らは，Java8 で追加された言語仕様であるラムダ式について，OSS における利用に関する調査を行った [1]．Java におけるラムダ式は，関数をオブジェクトのように扱うことができる言語仕様であり，同等の機能を提供する無名クラスをより簡潔に記述したものである．本関連研究では OSS の開発履歴を解析し，ラムダ式の利用を関数型インタフェースや導入目的など，さまざまな側面から分析している．

データセットは，GitHub から収集したスター数トップ 2,000 の Java の OSS である．結果として，ラムダ式の利用数の側面からは，2,000 件中 241 件のレポジトリで，合計 100,540 件のラムダ式が利用されていることがわかった．また，2016 年内において追加されたコードに含まれるラムダ式の割合は 2015 年の 3 倍程度になっており，ラムダ式の増加が加速していることがわかった．リファクタリング利用の側面からは，特に多くのラムダ式を導入していた 51 件のコミットを調査したところ，5,855 件のラムダ式が無名クラスを置き換えることで導入されていた．

2.3 本研究の目的

2.2 節の 2 つの研究では，言語仕様の利用数および適用数の増加について調査が行われている．また，言語仕様の利用形態に関する俯瞰として，2.2.1 節の研究ではジェネリクスに与えられる型引数について，2.2.2 節の研究ではラムダ式の型として与えられる関数型インタフェースの利用について調査されていた．よって，RQ1 および RQ2 では，これらの関連研究に倣って，以下のような調査を行う．

RQ1：レコードの利用における全体的な俯瞰を与えるため，OSS におけるレコードの適用数やその変遷を調査する．

RQ2：レコードの利用形態に関する俯瞰として，レコードの要素型と実装インタフェースについて統計を取得する．

また，本研究ではレコードを用いたリファクタリングを支援するツールの開発のために，必要な知見を収集することを目的としている．よって RQ3 および RQ4 では，より効果的な支援の提供のため，レコードを用いたリファクタリングについて以下の調査を行う．

RQ3：レコードを用いたリファクタリングで，ソースコードの要素削減の観点からどの程度の恩恵があるのか明らかにする．

RQ4：レコードを用いたリファクタリングを適用するにあたり，2.1 節で挙げたような制約も含め，どのような障害があるのか明らかにする．

3. データセットと手法

3.1 データセット

データセットとなる OSS は，GitHub の検索用 URL を用いたスクレイピングで，2022 年 12 月 24 日から 25 日にかけて収集した．以下の条件のもと，スター数の多い順に 2,000 件のレポジトリを選択している．

（1） レポジトリの言語が Java である．

（2） 最終プッシュが 2020 年 3 月 14 日以降である．

（2）の 2020 年 3 月 14 日は，レコードがプレビュー機能として実装された Java14 のリリース日である．この条件は，レコードが含まれている可能性の低いレポジトリを除外する目的で設定している．

3.2 手法

各 RQ における解析は，レポジトリ中の.java ファイルから抽象構文木を生成して行った．抽象構文木の生成に用いたのは，Java17API のパッケージ com.sun.source.tree のインターフェース群をベースにして制作した独自のパーサである．独自のパーサを用いた理由は，レコードが Java16 以降の言語仕様であり，対応するバージョンが Java15 までである JavaParser などでは解析できない可能性を考慮したからである．

3.2.1 RQ1

RQ1 については，はじめにレポジトリをクローンした時点（2022 年 12 月 25 日）でのレコードの利用数を集計した．それからデータセット中の各レポジトリを，2020

年 4 月から 2022 年 12 月の期間で各月 1 日時点のコミットに巻き戻し，レコードの利用数を集計した．

3.2.2 RQ2

RQ1 の結果，71 件のレポジトリでレコードの使用履歴がみられた．よって RQ2 では，これら 71 件のレポジトリに含まれるレコードに対して，ヘッダ，実装インタフェースの側面から分析を行った．さらに同じレポジトリ群に含まれるクラスについても，インスタンスフィールド，実装インタフェースの側面から分析を行い，結果を比較した．なお，フィールド型およびインタフェースは，アノテーションを除去した文字列ベースで分類している．

3.2.3 RQ3

RQ3 では，レコードの追加・削除について統計を取るため，各レポジトリに対して次の操作を行った．

(1) git diff コマンドで，親コミットから変更されたファイルのパスを取得する．
(2) (1) のファイルが存在する場合，ファイルに含まれる型のクラスパスと型の情報を取得する．存在しない場合，削除ファイルとして記録する．
(3) 親コミットに遡上する．
(4) (1) のファイルが存在する場合，型のクラスパスと型の情報を取得する．存在しない場合，追加ファイルとして記録する．
(5) (2) と (4) の情報を比較する．

上記の操作を繰り返していき，レコードの追加および削除の件数を集計した．さらに，クラスからレコードの変更については，コンストラクタ，ゲッタメソッド，toString/equals/hashCode メソッド，その他メソッドの増減についても集計した．なお，ここでいうゲッタメソッドは，名前がインスタンスフィールド名，あるいはインスタンスフィールド名に接頭辞 get を付けたのものであり，引数のないメソッドとしている．また，toString/equals/hashCode の各メソッドはシグネチャで判定している．

3.2.4 RQ4

RQ3 を実施する過程で，クラスからレコードへのリファクタリングを一度に 50 件以上行うコミットを複数発見した．RQ4 における 1 つ目の調査では，これらのコミットを目視調査することで，リファクタリング作業に伴うコストについて紐解いていく．また，同じく RQ3 の結果より，レコードからクラスに変更されたケースが 31 件存在することがわかった．これらの変更は，もし意図的なものならば，レコードの利用で何かしらの弊害があったから行われたと考えることができる．そしてレコードの利用による弊害は，開発者がレコードの導入を渋る理由の一端を担っている可能性があり，今後レコードを用いたリファクタリングを設計する上で重要な要素になると考えられる．よって RQ4 における 2 つ目の調査では，レコードからクラスへの変更についてソースコードの差分やコメント，GitHub 上の議論を目視し，理由を調査した．

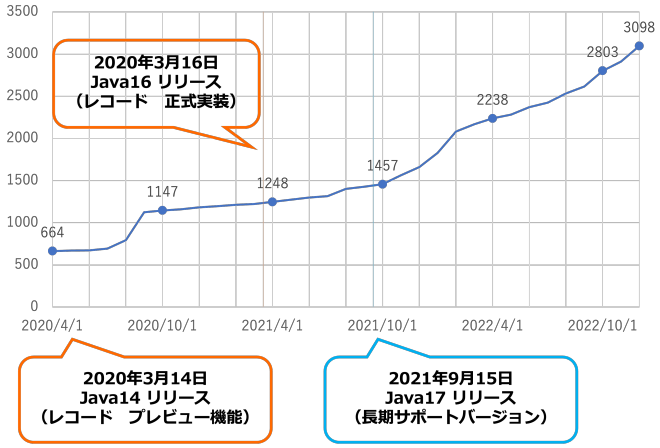

図 3　レコードの利用数の変遷

4. 実験結果と考察

4.1 RQ1：レコードは OSS において，どの程度の数使用されているのか？

図 3 は，2020 年 4 月から 2022 年 12 月までのレコード利用数の変遷を示した折れ線グラフである．データセットのレポジトリをクローンした 2022 年 12 月 25 日時点では，70 件のレポジトリで合計 3,244 件のレコードが利用されていた．利用数の伸び方に注目してみると，Java17 がリリースされる 2020 年 9 月までの 18 か月の区間は月 44.1 件のペース，2020 年 10 月以降の 14 か月の区間は月 117.2 件のペースでレコードが導入されている．すなわち，レコードの利用数の伸びは Java17 の登場以降に加速していることがわかる．

4.2 RQ2：使用されているレコードの特徴はどのようになっているか？

4.2.1 インスタンスフィールド数

表 1 は，レコードおよびクラスのインスタンスフィールド数別に，件数を集計したものである．最も多かったのはフィールド数が 2 個のレコードで，全体の 33.0%を占めていた．なお，定義 1 件あたりのフィールド数を算

表 1　定義フィールド個数別のレコード（クラス）の件数

フィールド数	レコード		クラス	
	個数	割合	個数	割合
0	317	9.8%	229,275	60.9%
1	815	25.1%	54,129	14.4%
2	1,070	33.0%	32,501	8.6%
3	483	14.9%	19,358	5.1%
4	205	6.3%	12,043	3.2%
5	129	4.0%	8,115	2.2%
6~10	157	4.8%	15,174	4.0%
11~	68	2.1%	6,077	1.6%
合計	3,244	-	376,672	-

表 2　レコードのフィールド型の傾向

型名	個数	割合
int	2,871	30.7%
(java.lang.)String	1,725	18.4%
long	615	6.6%
boolean	374	4.0%
double	205	2.2%
List<String>	109	1.2%
(java.lang.)Boolean	74	0.8%
double[]	68	0.7%
(java.lang.)Object	65	0.7%
(java.lang.)Integer	64	0.7%
その他	3,182	34.0%
合計	9,352	-

表 3　クラスのフィールド型の傾向

型名	個数	割合
long	205,434	29.4%
int	77,043	11.0%
(java.lang.)String	53,182	7.6%
boolean	39,275	5.6%
(java.lang.)Object	5,829	0.8%
double	4,408	0.6%
byte[]	3,875	0.6%
List<String>	3,501	0.5%
int[]	3,398	0.5%
float	3,246	0.5%
その他	298,946	42.8%
合計	698,137	-

出したとことろ，クラスは 1.85 個であるのに対し，レコードは 2.88 個であった．すなわち，レコードの方がクラスよりも多くのフィールドを持つことがわかった．

4.2.2　フィールド型の傾向

表 2 はレコードのインスタンスフィールド型，**表 3** はクラスのインスタンスフィールド型の傾向を集計したものである．型の中には java.lang.String のように完全限定名で指定されているものもあったので，それらも単純名での検出と一緒にして計上している．結果をみると，レコードのフィールド型として際立って利用されているのは int 型であり，全体の 30.7%を占めている．対してクラスのフィールド型として際立っているのは long 型であり，全体の 29.4%を占めている．ただしこの数値は，レポジトリ SapMachine と JetBrainsRuntime における，大量の long 型フィールドを含むクラス群を計上した結果である．これらのレポジトリには，わずか 2 つで合計 81,903 もの long 型フィールド宣言（全 long 型フィールドの 39.9%）を行うクラスが含まれており，結果に大きな影響を与えている．

SapMachine と JetBrainsRuntime を除いた場合，レコードのフィールド型で最も多いのは String で 1,410 件（全体の 22.6%），次点は int で 1,177 件（全体の 18.9%）となる．一方，クラスのフィールド型で最も多いのは int で 41,969 件（全体の 13.1%），次点は String で 35,687 件（全体の 11.2%）となる．

表 4　実装インタフェース数別のレコード（クラス）の件数

インタフェース数	レコード		クラス	
	件数	割合	件数	割合
0	2,443	75.3%	309,354	82.1%
1	743	22.9%	58,507	15.5%
2	54	1.7%	6,719	1.8%
3	4	0.1%	1,430	0.4%
4~	0	0.0%	662	0.2%
合計	3,244	-	376,672	-

表 5　レコードの実装インタフェースの傾向

型名	件数	割合
(java.io.)Serializable	253	29.3%
Writeable	49	5.7%
Decoration	35	4.1%
Comparable	34	3.9%
ToXContentObject	30	3.5%
ThrowingExternalizable	26	3.0%
ClusterStateTaskListener	16	1.9%
ExpirationPolicyBuilder	16	1.9%
ToXContentFragment	15	1.7%
Runnable	14	1.6%
その他	375	43.5%
合計	863	-

4.2.3　実装インタフェース数

表 4 は，レコードおよびクラスの実装インタフェース数別に，件数を集計したものである．実装インタフェース数の平均をとると，レコードは 0.266，クラスは 0.213 であるので，レコードの方がインタフェースの実装が行われやすいということがわかった．

4.2.4　実装インタフェースの傾向

表 2 はレコードのインスタンスフィールド型，表 3 はクラスのインスタンスフィールド型の傾向を集計したも

表 6　クラスの実装インタフェースの傾向

型名	個数	割合
(java.io.)Serializable	4,242	5.3%
Runnable	3,225	4.0%
(java.util.)Iterator	1,343	1.7%
(java.lang.)Comparable	1,034	1.3%
(java.lang.)Cloneable	888	1.1%
Collector	811	1.0%
Writeable	790	1.0%
ActionListener	740	0.9%
ToXContentObject	721	0.9%
IdentifiedDataSerializable	606	0.8%
その他	65,649	82.0%
合計	80,049	-

表 7　レコードの追加の内訳

型名	件数	割合
新規ファイルと共に追加	2,096	67.0%
非レコード型からレコードへの変更	695	22.2%
既存ファイルへ追加	336	10.7%
追加計	3,127	-

表 8　レコードの削除の内訳

型名	件数	割合
削除ファイルと共に削除	87	43.5%
レコードから非レコード型への変更	31	15.5%
既存ファイルからの削除	82	41.0%
削除計	200	-

のである．こちらも 4.2.2 節と同じく，完全限定名の指定と単純名の指定をまとめてある．また，型引数のある型は型引数を無視してまとめてある（raw タイプも含む）．結果を見ると，レコードもクラスも最も多く実装されているのは (java.io.)Serializable となった．ただ全体で見た比率に差があり，レコードは 7.8%，クラスは 1.1%への実装となっている．クラスのみへの実装が際立っているインタフェースとしては Runnable が挙げられる．レコードにおいては全体の 0.4%程度だが，クラスでは全体の 0.9%となっている．

4.3 RQ3：クラスをレコードに変更するリファクタリングでは，どの程度の恩恵を享受できるのか？

4.3.1 レコードの追加と削除の件数

コミットで変更されたファイルの履歴からレコードの追加と削除を抽出し，分類を行った．なお，ラムダ式などの内部で宣言されているレコードはクラスパスが生成できず，差分間で同一の型を特定するのが困難であるので，ここではトップレベルレコード及びクラス内部レコードのみを対象としている．**表 7** にレコードの追加の内訳，表 7 にレコードの削除の内訳を示す．

表 9　クラスからレコードへの変更による要素の増減

要素	削減件数	追加件数	続投
toString()	57	1	58
equals(Object o)	85	1	18
hashCode()	84	2	17
コンストラクタ	499(543)	5(5)	-
ゲッタメソッド	277(787)	5(5)	-
その他メソッド	92(220)	46(58)	-

表 10　クラスからレコードへのリファクタリングを行うコミット

レポジトリ名	ID（上 6 桁）	コミット日時	変更件数
signal-cli	ce7aa5	2021/10/24 22:26:12	58
CloudNet-v3	bf4f70	2021/12/15 09:18:34	68
elasticsearch	fc5a82	2022/1/18 17:53:06	107
elasticsearch	cce5ad	2022/1/25 00:31:15	51
cas	d1bac8	2022/9/17 09:32:45	87

表より，レコードの追加のうち 22.2%が，非レコード型からレコードへの変更であることがわかる．また，レコードの削除のうち 15.5%が，レコード型から非レコード型への変更であることがわかる．レコードから非レコード型への変更の詳細については，4.4 節で述べる．

4.3.2 クラスをレコードに変更するリファクタリングの恩恵

非レコード型からレコードへの変更 695 件のうち，クラスからレコードに変更するリファクタリングは 690 件であった．**表 9** は，3.2.3 節で挙げた各要素について，削減および追加がみられたリファクタリングの件数を示している．また，表 9 における続投の列は，元々宣言されていた toString/equals/hashCode の削除を行わなかったリファクタリングの件数を示している．コンストラクタやゲッタメソッドはひとつの型宣言に複数含まれている場合があるので，リファクタリングによって 1 件以上の削減がみられた場合は削減，1 件以上の追加がみられた場合は追加として計上している．また，() の中で要素の削減件数および追加件数の累計を示している．表 9 の結果より，クラスからレコードへ変更するリファクタリング 690 件のうち 499 件（72.3%）で，コンストラクタの削減がみられることがわかる．また，元々クラス中に定義されていた equals メソッドおよび hashCode メソッドのうち，およそ 8 割がレコードへの変更によって削減されていることがわかる．なお，toString メソッドに関しては，リファクタリングによる削減はおよそ半数程度にとどまっている．

```
1  public class TextFieldEvent {
2    private final TextField field;
3    private final String oldval;
4    private final String newval;
5
6    public TextFieldEvent(TextField field, String
         old, String val) {
7      this.field = field;
8      this.oldval = old;
9      this.newval = val;
10   }
11
12   public String getOldText() {
13     return oldval;
14   }
15
16   public String getText() {
17     return newval;
18   }
19
20   public TextField getTextField() {
21     return field;
22   }
23
24 }
```

```
1  public record TextFieldEvent(
2      TextField getTextField,
3      String getOldText,
4      String getText
5  ){ }
```

図 4 logism-evolution でみられたリファクタリング（上：変更前，下：変更後）

4.4 RQ4：クラスからレコードへのリファクタリングを阻害する要因は何か？

4.4.1 リファクタリング作業に伴うコスト

signal-cli，CloudNet-v3，elasticsearch，cas といったレポジトリでは，50 件以上ものクラスをレコードに置き換えるリファクタリングのコミットが見られた．**表 10** にそれらのコミットを示す．これらのコミットを目視で調査したところ，クラスからレコードへの宣言の変更以外にも，レコードに変更された型のメンバ参照を書き換えている箇所が多くみられた．クラスのフィールドを直接参照している箇所は，レコードに変更されることでフィールドが private となるため，ゲッタメソッドを介した参照に書き換えられていた．また，元々クラスに宣言されているゲッタメソッドは接頭辞 get が付けられていることが多く，そのままではレコードのゲッタメソッドと互換性がないため，呼び出し箇所の get の削除が行われていた．すなわち，クラスからレコードへのリファクタリングは，宣言自体の単純な変更に留まらないケースが多く，若干の作業コストを要する可能性がある．

なお，レポジトリ logism-evolution におけるリファクタリングは，ゲッダメソッドの呼び出し箇所を変更するのを避けるため，レコードへの変更でヘッダに定義する変数に接頭辞 get を加えていた．（**図 4**）．この手法は，リファクタリングに伴うコスト低減の一つのヒントになると考えられる．

4.4.2 レコードから非レコード型への変更

4.3 節で，31 件のレコードから非レコード型への変更を取得することができた．これらの変更について目視調査を行ったところ，次のような理由があることがわかった．

- レコードだと，外部ツールのアノテーションがうまく機能しなかった．（6 件）
- final でないインスタンスフィールドを追加する必要があった．（4 件）
- コンストラクタに特殊な初期化ルーティンを追加する必要があった．（2 件）
- 型宣言にフィールドを定義する必要が無くなった．（1 件）
- hashCode メソッドのパフォーマンス向上のため，ハッシュを保持するインスタンスフィールドを追加した．（1 件）
- 元々インタフェースだった継承元が抽象クラスに変更された．（1 件）
- コンストラクタのアクセスを protected にする必要があった．（1 件）
- 継承先のクラスが追加された．（1 件）
- 理由不明（14 件）

31 件のうち，6 件が外部ツールに関わる理由であった．一部の外部ツールがレコードに対してうまく動作しないのは，レコードは登場して日が浅い言語仕様であることが原因であると考えられる．また，10 件が 2.1 節で説明したレコードの制約に関わるものであった．

5. 妥当性の脅威

5.1 レコードのみに着目する調査の妥当性

今回の調査は，レコードのみに着目して議論をおこなっている．その妥当性について検証するため，以下の条件のもとクラスの総数を調査した．

（1） レコードに必須な前提条件を満たしている．（クラスの修飾子に abstract，sealed，non-sealed のいずれも含まれておらず，継承元のクラスが無い上に，インスタンスイニシャライザも定義されていない．）

（2） （1）に加え，クラスが final でないインスタンスフィールドをもたない．

表 11　条件別のクラス総数（レコードの使用履歴があったレポジトリ）

条件	削減件数	割合
クラス総数	376,672	-
（1）前提条件を満たすクラス	182,608	48.5%
（2）final でないフィールドをもたないクラス	129,537	34.4%
（3）final なフィールドをもつクラス	25,814	6.9%
（4）削減可能な要素をもつクラス	17,685	4.7%
（5）明示的に継承禁止のクラス	3,779	1.0%

表 12　条件別のクラス総数（レコードの使用履歴がなかったレポジトリ）

条件	削減件数	割合
クラス総数	1,278,418	-
（1）前提条件を満たすクラス	540,517	42.3%
（2）final でないフィールドをもたないクラス	321,313	25.1%
（3）final なフィールドをもつクラス	93,959	7.3%
（4）削減可能な要素をもつクラス	62,775	4.9%
（5）明示的に継承禁止のクラス	13,002	1.0%

（3）（2）に加え，クラスが 1 つ以上の final なインスタンスフィールドをもつ．

（4）（3）に加え，レコードに変換することで，削減可能な要素が含まれている．（クラスが，カノニカルコンストラクタ，hashCode/equals/toString メソッド，ゲッタメソッドのいずれかをもつ．）

（5）（4）に加え，クラスが final であり，明示的に継承禁止である．

ここでいうカノニカルコンストラクタは，引数の型の組み合わせが final なインスタンスフィールドと等しいものとしている．なお，可変長引数の型は配列型として扱っている．

表 11 は，3.2.1 節の調査でレコードの使用履歴があった 71 レポジトリにおける結果である．また，**表 12** はそれ以外の 1,929 のレポジトリにおける結果である．これらの結果から，クラスをレコードへのリファクタリングが適格なもの（条件（4）以降を満たすもの）のみに限ったとしても，レコードの検出数である 3,244 件は十分に大きな値ではないといえる．すなわち，本研究の結果やそれに基づいた知見は，今後これらのクラスのレコードへの置換が進んだ場合などに，大きく変化する可能性がある．

5.2　型の同定について

4.2.2 節，4.2.4 節などでは，型の情報を取得する調査を行っている．本研究での型の情報は，参照ではなく文字列ベースで取得しているため，同じ型にラベル付けされていても等しい型とは限らない．実際，**表 6** の ActionLister は，java.awt.event パッケージのものとユーザ定義のものが取得されている．

5.3　目視調査について

本研究での目視調査は，第一著者のみによるものである．今後は妥当性の保証のため，複数人での目視調査が望まれる．

6.　まとめと今後の展望

レコードは，それまでの値ベース・クラスを簡潔に宣言することを可能とする新たな言語仕様で，2021 年 3 月の Java16 にて正式に登場した．本稿ではレコードについて，2,000 件の OSS レポジトリにおける利用について調査を行った．結果として，70 件のレポジトリで合計 3,244 のレコードの利用を検出した．また，利用されているレコードのうち，クラスからレコードへのリファクタリングとして導入されたのは 690 件であり，その 7 割以上でコンストラクタの削減が行われていた．

今後は，これらの結果および 5 節に挙げた妥当性の脅威を踏まえ，レコードの利用に関する更なる調査を進めることを考えている．また本研究の結果をもとに，リファクタリングをサポートするツールを制作し，その効果についての検証を行うことを計画している．

謝辞

本研究の一部は JSPS 科研費 JP18H04097, JP20H04167, JP21H04877, JP22K17874, JP22K18630 の助成を受けた．

参考文献

[1] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Adoption and use of java generics. In *Empirical Software Engineering*, Vol. 18, pp. 1047–1089, 2013.

[2] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, Vol. 1, No. OOPSLA, 2017.

[3] Gavin Bierman. *JEP 395: Records.*

[4] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java® Language Specification.*