

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 [1]。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 [2][3]。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。

- 過去の言語仕様に対する初期調査と同じく、レコードについても利用実態に関する各種統計を与える。
- Java 言語を利用する開発者に対し、レコードの効果的な利用のヒントを与える。
- 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なになっているか？
- 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リファクタリング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 に対する結果を示し、5 節で考察を述べる。最後に 6 節でまとめを述べる。

¹ 九州大学
Kyushu University
shu-u.ac.jp
c.jp
c.jp
u.ac.jp

ここで改行。新しいパラ
グラフ。

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

図 1 レコードの宣言

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

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型宣言のフレームワークであり、2021 年 3 月の Java16（プレビューは Java14 から）で Java に登場した [4]。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実装され、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取り、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ーメソッド
- 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、暗黙的に実装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、通常クラスに存在しない次のような制約も課される [5]。

- 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 [2]。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

文献番号ずれていない？

筆頭著者のみでOK.
～ら.

著者が2名のときは、
「AとBは、」と書く.

ランタイムの減少、メモリ使用数の増加に伴って統計的割合が減少していることがわかった。ジェネリクス利用の側面からは、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Ameyya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[2]。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、

データセットは、GitHubからランダムに抽出した2,000のJavaのOSSである。このデータセットの利用数の側面からは、2,000件のOSSのうち、ラムダ式の利用で、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年以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17日以降である。

(2)の2020年3月17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

レポジトリ
→リポジトリ

どういふことでしょうか？側面から分析する、側面、及び、分析が、曖昧で、次の文で具体的に何をするかを補足して欲しいです。

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、ゲッターメソッド、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、ここでいうゲッターメソッドは、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

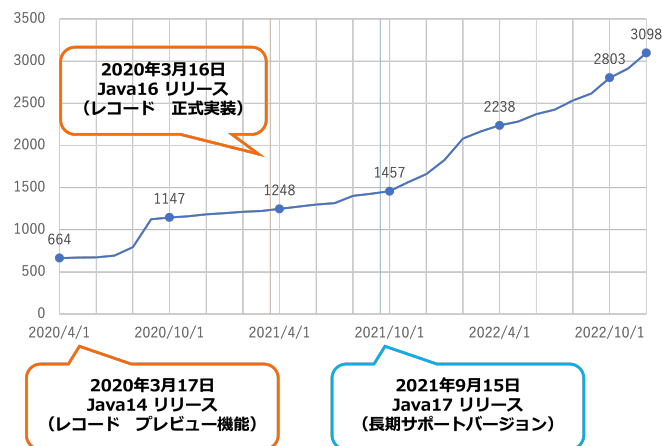


図3 レコードの利用数の変遷

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。

4. 実験結果

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、およそ2.66倍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なになっているか？

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出したところ、クラスは1.85個であるのに対し、レコードは2.88個であり、平均でレコードの方が1個程度多いという結果になった。

4.2.2 フィールド型の傾向

表2はレコードおよびクラスのインスタンスフィー

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（クラス）のフィールド型の傾向

型名	レコード		型名	クラス	
	個数	割合		個数	割合
int	2,871	30.7%	long	205,434	29.4%
String	1,725	18.4%	int	77,043	11.0%
long	615	6.6%	String	53,182	7.6%
boolean	374	4.0%	boolean	39,275	5.6%
double	205	2.2%	Object	5,829	0.8%
List<String>	109	1.2%	double	4,408	0.6%
Boolean	74	0.8%	byte[]	3,875	0.6%
double[]	68	0.7%	List<String>	3,501	0.5%
Object	65	0.7%	int[]	3,398	0.5%
Integer	64	0.7%	float	3,246	0.5%
その他	3,182	34.0%	その他	298,946	42.8%
合計	9,352	-	合計	698,137	-

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。

4.2.3 実装インタフェース数

表 3 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

表 3 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 4 レコード（クラス）の実装インタフェースの傾向

型名	レコード		型名	クラス	
	件数	割合		件数	割合
Serializable	253	29.3%	Serializable	4,242	5.3%
Writeable	49	5.7%	Runnable	3,225	4.0%
Comparable	35	4.1%	Iterator	1,343	1.7%
Comparable	34	3.9%	Comparable	1,034	1.3%
ToXContentObject	30	3.5%	Cloneable	888	1.1%
ThrowingExternalizable	26	3.0%	Collector	811	1.0%
ClusterStateTaskListener	16	1.9%	Writeable	790	1.0%
ExpirationPolicyBuilder	16	1.9%	ActionListener	740	0.9%
ToXContentFragment	15	1.7%	ToXContentObject	721	0.9%
Runnable	14	1.6%	IdentifiedDataSerializable	606	0.8%
その他	375	43.5%	その他	65,649	82.0%
合計	863	-	合計	80,049	-

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 4 はレコードおよびクラスの実装インタフェースの傾向を集計したものである。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 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 5 にレコードの追加の内訳、表 5 にレコードの削除の内訳を示す。

表より、レコードの追加のうち 22.2% が、非レコード

http://...って表8のIDだけでなく、URLもふっとノートみたいな感じで載せられませんか？

https://posl.ait.kyushu-u.ac.jp/~kamei/publications/Shimagaki_ICSME2016.pdf

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%
既存ファイルへの追加（型宣言の変更以外）	336	10.7%
追加計	3,127	-

表 6 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（型宣言の変更以外）	82	41.0%
削除計		

続投って？

表 7 クラスからレコードへの変更による

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)	-
ゲッターメソッド	277(787)	5(5)	-
その他メソッド	92(220)	46(58)	-

変更695件のうち、... リファクタリングは690件であった。とあります。リファクタリングと判断する基準はどこかに書かれていますか？

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 7 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 7 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ー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 7 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。

レコードへのリファクタリングを行うコミット

ID (上 6 桁)	コミット日時	変更件数
ce7aa5	2021/10/24 22:26:12	58
bf4f70	2021/12/15 09:18:34	68
fc5a82	2022/1/18 17:53:06	107
cce5ad	2022/1/25 00:31:15	51
d1bac8	2022/9/17 09:32:45	87

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 8 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、ゲッター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ーメソッドは接頭辞 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ー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、宣言以外にメンバの参照箇所にも変更が求められる場合があるということがわかった。

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なくなった。(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。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. 考察

5.1 各 RQ の考察

5.1.1 RQ1

RQ1 では、OSS のコミット履歴を遡り、レコードの適用数の変化について調査した。その結果、Java17 のリリース以降でレコードの適用数の増加が 2.66 倍に加速していることがわかった。レコードが正式に登場した Java16 リリース時点ではなく、Java17 のリリース以降で適用数の増加が加速したのは、Java17 が長期サポートであることが理由であると考えられる。関連研究 [2] でも、OSS における言語仕様の導入時期には Java のバージョンが大きく関与すると述べられており、レコードも OSS に Java17 が導入されることを契機にして利用が進んだものと考えられる。

5.1.2 RQ2

RQ2 の前半では、レコードのインスタンスフィールドについて調査した。その結果、レコードの方がクラスよりもフィールドの個数が平均 1 個程度多いということがわかった。さらに、フィールド型の全体における割合で見た場合、レコードの方がクラスよりも String の利用が目立っていることがわかった。これらの結果から、レコードの用途として多いのは、文字列データの保持であるということがわかる。

なお、本来レコードは値の保持を目的とする型であるが、表 1 によると要素数 0 のレコードも全体の 9.8% 見受けられる。これらのレコードを目視で調査したところ、ほとんどはレポジトリ checkstyle などにおける、ソースコードのスタイル例としての利用であった。ただ、レポジトリ signal-cli などの一部の OSS では、要素数 0 のレコードをソフトウェア部品として利用していると思われるケースも存在した。これらは final class の代替であると思われるが、レコード宣言の制約を利用し、今後のフィールド追加などの編集を制限する目的も含まれていると考えられる。

RQ2 の後半では、レコードに実装されるインタフェースについて調査し、レコードの方がクラスよりも Serializable が実装されている割合が高く、一方で Runnable が実装されている割合が低いことがわかった。このこと

から、実装インタフェースの側面で見ても、レコードは振る舞いよりデータの保持が重視される型宣言であるということが見てとれる。

5.1.3 RQ3

RQ3 の前半では、コミットの差分から追加されたレコードが型宣言の変更によるものなのか、そうでないのかを分類した。その結果、レコードの追加のうち 22.2% が既存の型宣言の変更によるものということがわかった。対して、型宣言の変更以外で既存ファイルにレコードが追加されるケースは 10.7% であり、これらもレコードを用いた何かしらのリファクタリングである可能性はあるが、それを踏まえてもレコードを用いたリファクタリングとして趨勢を占めるのは型宣言の変更であると考えられる。

RQ3 の後半では、クラスからレコードへのリファクタリングによって、クラス内部の要素がどれだけ削減されるのかを調査した。その結果、72.3% のリファクタリングでコンストラクタの削減が行われており、その他ゲッターメソッドや toString/equals/hashCode メソッドについても一定の削減がみられた。よってレコードに暗黙的に実装される各メソッドは OSS において有効に活用されており、特にカノニカルコンストラクタの実装が大きな効果を発揮しているといえる。また、これらのリファクタリングを自動的に支援するツールによっても、開発者は同様の恩恵を得られるものであると考えられる。

5.1.4 RQ4

RQ4 の前半では、クラスをレコードに変更するコミットの特徴を目視によって調査した。その結果、型のフィールドあるいはゲッターメソッドの参照まわりの変更が多くみられた。特に、レコードに暗黙的に実装されるゲッターメソッドの名前が接頭辞 get を含まないという点は、リファクタリング作業に伴うコストを増幅させる原因となっていると考えられる。今後リファクタリングのサポートを行うツールを制作する際には、このようなコストを低減できる設計が望ましいと思われる。

なお、レポジトリ logism-evolution におけるリファクタリングは、ゲッター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、手動でのリファクタリングに伴うコスト低減のヒントになると考えられる。

RQ4 の後半では、レコード宣言がクラス宣言に差し戻された 31 件のケースを目視で調査し、理由を分類した。その結果、6 件が外部ツールに関わる理由であった。レ

```
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,
7         String old, String val) {
8         this.field = field;
9         this.oldval = old;
10        this.newval = val;
11    }
12
13    public String getOldText() {
14        return oldval;
15    }
16
17    public String getText() {
18        return newval;
19    }
20
21    public TextField getTextField() {
22        return field;
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）

コードは実装されてから日が浅い言語仕様であり，外部ツールのアノテーションなどが正常に動作しない可能性があるため，レコードを用いる際には外部ツールの互換性に注意を払う必要があると考えられる．また，クラスに差し戻された理由のひとつとして，hashCode メソッドのパフォーマンスの向上を図るためというものがあった．レコードに標準で実装される equals/hashCode メソッドは，再帰的に各フィールドの当該メソッドを呼び出す実装であるため，オブジェクトの入れ子構造が深くなるにつれパフォーマンスが低下する恐れがある．よってこれらのメソッドの暗黙的な実装を目的としてレコードを導入する際には，パフォーマンス面での問題を留意しておく必要があると考えられる．

5.2 妥当性の脅威

5.2.1 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

けられていても等しい型とは限らない．実際，表 4 の ActionListener 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．

5.2.2 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 5.2 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] Robert Dyer, Rajan, Hridesh, Nguyen, Hoan Anh, Nguyen, and Tien N. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, p. 779–790, New York, NY, USA, 2014. Association for Computing Machinery.
- [2] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Adoption and use of java generics. In *Empirical Software Engineering*, Vol. 18, pp. 1047–1089, 2013.
- [3] 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.
- [4] Gavin Bierman. *JEP 395: Records*.
- [5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java® Language Specification*.