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.

イミュータブルクラスを利用する必要性に関する調査

～ハッシュ値を利用するデータ型を対象として～

橋本　　周†　　肥後　芳樹†　　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{shu-hsmt,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　Java には複数のオブジェクトを格納するためのさまざまなデータ型が用意されている．その中に，HashMap や HashSet のように格納するオブジェクトのハッシュ値を利用するデータ型がある．このようなハッシュ値を利用するデータ型は，オブジェクトを格納する際にそのオブジェクトのハッシュ値をデータ型の内部に記録する．そのため，格納されたオブジェクトに変更が加わりハッシュ値が変化してしまった場合に，変更後のハッシュ値とデータ型に記録されたハッシュ値との間で不整合が生じてしまう．そこで，このようなハッシュ値を用いるデータ型に格納するオブジェクトのクラスにはその内部状態が生成後に変化しないイミュータブルクラスが推奨されている．しかし，イミュータブルクラスの利用により，ハッシュ値との不整合がどの程度防げているのかについてはこれまで明らかにされてこなかった．そこで本研究では，Java の実プロジェクトにおいてハッシュ値を利用する HashMap と HashSet を対象とした調査を行い，データ型に格納するオブジェクトはイミュータブルクラスにすべきかを明らかにする．HashMap および HashSet に格納された対象プロジェクト内で定義されたクラスを利用していたオブジェクトを対象に，そのクラスがイミュータブルであるか否かの調査を行った．調査の結果，イミュータブルクラスは HashMap では 14.8%，HashSet では 9.7%利用されていると明らかになった．また，過去バージョンではミュータブルクラスであったが，不整合のバグが発生したためにイミュータブルに変更された事例はまったく見つからなかった．

キーワード　イミュータビリティ，ハッシュ，Java

1. まえがき

Java には複数のオブジェクトを格納するためのさまざまなデータ型が用意されている．その中には，HashMap や HashSet というデータ型が存在する．この 2 つのデータ型は，ハッシュ値を用いてデータの格納を行う．例えば，HashSet では，要素となるオブジェクトのハッシュ値とハッシュ表を対応させることで，取り出したいデータの読み書きを行う [1]．このとき，要素に使用されたオブジェクトの内部状態を変更した場合，そのオブジェクトのハッシュ値も同様に変化してしまう．つまり，HashMap のキーや HashSet の要素に用いられているオブジェクトの内部状態を変更すると，オブジェクト変更後のハッシュ値とハッシュ表に対応付けられているハッシュ値との間に不整合が生じる [2]．このような HashMap のキーや HashSet の要素に使用されるオブジェクトの内部状態が変化を防ぐために，オブジェクトの基となるクラスはイミュータブルにするべきといわれている [3] [4] [5] [6]．

イミュータブルとは，オブジェクトの初期化後に値の変更が行えないことを指す [7]．図 1 は Java におけるイミュータブルクラスの例である．このクラスは，与えられたフィールド変数すべてに final 修飾子が付けられており，一旦設定された値は変更できない．このように，イミュータブルクラスはオブジェクト生成時からオブジェクトの内部状態が変更されることはないため，オブジェクトのハッシュ値が変更される心配はない．

```java
public final class Product {
  private final String name;      // 品名
  private final int price;        // 価格

  public Product(String name, int price) {
    this.name = name;
    this.price = price;
  }
  // 値を変更する関数が存在しない
}
```

図 1　イミュータブルクラスの例

しかしながら，実際のプロジェクトにおいてハッシュ値の不整合によるバグがどのくらい発生しているのか，またそのような事象に対してイミュータブルクラスの実装による対処がどれくらい行われているのかは調査されてこなかった．現代の複雑なソフトウェア開発現場において [8]，コーディングに力を入れる

```
1 Calendar GetOneMonthLaterCalender(Calendar calendar){
2   calendar.add(Calendar.MONTH,1);
3   return calendar; //一ヶ月後の日付を返す
4 }
5 void CalenderMut(){
6   final Map<Calendar,Integer> map = new HashMap<>();
7   final Calendar calendar = Calendar.getInstance();
8   map.put(calendar,1);
9   GetOneMonthLaterCalender(calendar); //内部状態が変化
10  map.get(calendar) //nullが出力される
11 }
```

図 2　HashMap のキーにミュータブルクラスを使用することによるバグの例

べき部分を明らかにすることは重要である．よって，困難といわれているイミュータブルクラスの実装 [9] の必要性を調べることは効率的なソフトウェア開発に繋がる．

本研究では，実際のプロジェクト 107 件を対象として HashMap のキーおよび HashSet の要素に使用されているオブジェクトのクラスのイミュータビリティを調べた．また，HashMap のキーおよび HashSet の要素のオブジェクトにイミュータブルクラスを使用しなかったことによるバグが起きた事例が存在するのか調査を行った．

2. 準　　備

2.1 ハッシュ値とミュータビリティ

java.util.Map の公式ドキュメントには，ミュータブルオブジェクトを Map のキーにつかうには細心の注意が必要であると述べられている [2]．ミュータブルとは，オブジェクトの初期化後にもフィールドを変化させることができることを指す．そのため，キーに使用されるオブジェクトに対し，equals メソッドによる比較に影響を与えるような変更を行ったときに，ハッシュ値が変更されてしまいバリューとして登録されたオブジェクトを取り出せなくなる．

図 2 はミュータブルクラスを HashMap のキーに使用したことにより，正しい値を取り出せなくなった例である．java.util.Calender(注1)はミュータブルクラスであり，図中のプログラムでも 1-4 行目メソッドでオブジェクトの内部状態が変化している．そのため，10 行目では正しい値を取り出せなくなっている．仮に Calender クラスの中に add メソッドのようなオブジェクトの内部状態を変化させるメソッドが無くす，フィールド変数に final 修飾子が付けるなどの処理を行っていれば，ハッシュ値の変化は起きず，10 行目で正しい値を取り出すことができたはずである．

以上のように，ミュータブルクラスを HashMap のキーや HashSet の要素に使用するとバグが起きる可能性があるため，オブジェクトのクラスはイミュータブルにすることが推奨されている．しかしながら，既存研究 [10] では Java におけるイミュータブルクラスの実装は困難であるとされている．本研究では，HashMap のキーおよび HashSet の要素に焦点を当て，実装困難な Java のイミュータブルクラスの必要性を調査することを目的とする．

2.2 調 査 対 象

本研究の調査対象は GitHub 上に公開されている Java プロジェクトのリポジトリである．数あるリポジトリの中で，イミュータブルクラスの実装割合を客観的に調査する為に，Domains of 2,500 GitHub Repositories（以下，Domains）(注2)に掲載されている Java プロジェクト群と Apache(注3)の 500 Stars 以上の Java プロジェクト群を調査候補に挙げた．この 2 つのプロジェクト群の中から，後述する調査方法で必要となる Jar ファイルが Maven Repository(注4)にて配布されていること，クラスがイミュータブルであるか解析するために使用したツールでエラーが発生しなかったこと，この 2 つの条件を満たすプロジェクトを対象として調査を行った．この条件に該当するプロジェクトは，Domains では 51 件，Apache では 56 件，計 107 件であった．

2.3 Mutability Detector

本研究の調査では，各々のクラスがイミュータブルであるかの判定に Mutability Detector(注5)というツールを使用した．Mutability Detector は解析対象の各クラスについて以下の 4 つのうちいずれか 1 つを出力する．

- COULD_NOT_ANALYSE
- NOT_IMMUTABLE
- EFFECTIVELY_IMMUTABLE
- IMMUTABLE

COULD_NOT_ANALYSE はそのクラスがなんらかの理由で解析できなかったことを表す．NOT_IMMUTABLE はそのクラスがミュータブルであることを示す．EFFECTIVELY_IMMUTABLE と IMMUTABLE の違いはマルチスレッド環境下でもイミュータブルかである．EFFECTIVELY_IMMUTABLE はシングルスレッドの場合のみイミュータブルであるのに対し，IMMUTABLE はマルチスレッドでもイミュータブルクラスとしてふるまう．この Mutability Detector におけるイミュータブルクラスの判定基準は Effective Java [3] におけるイミュータブルクラスの実装規則に基づいている [11]．

解析にはクラスファイルが必要となるため，Jar ファイルが見つからなかったプロジェクトは調査から除外した．また，Jar ファイルは見つかったものの，Mutability Detector のエラーにより解析できなかったプロジェクトも調査から除外した．

3. Research Question

本研究では，HashMap および HashSet に格納されるクラスがイミュータブルである必要性を調査する．この調査にあたり，以下に示す 2 つの Research Question（RQ）を設定した．

(注1)：https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Calendar.html

(注2)：https://docs.google.com/spreadsheets/d/1oOIyVTsuJUOaQKgV8H5jWQKzLovoycf9ZjtajqEU8dc

(注3)：https://github.com/apache

(注4)：https://mvnrepository.com

(注5)：https://github.com/MutabilityDetector/MutabilityDetector

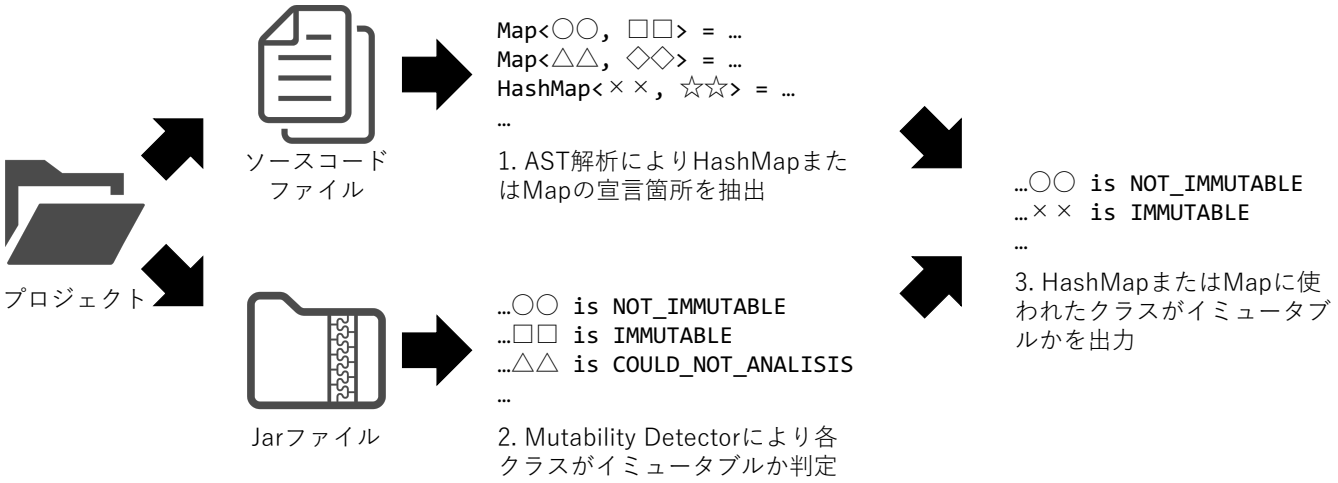

図 3　RQ1 の調査方法

RQ1: HashMap のキーおよび HashSet の要素に使用されるオブジェクトのクラスがイミュータブルである割合はどの程度か

実プロジェクトにおいて HashMap のキーと HashSet の要素に用いられているクラスのうちイミュータブルクラスである割合を算出する．この割合と既存研究 [12] で算出されているプロジェクト全体のクラスのうちイミュータブルクラスである割合とを比較する．HashMap のキーや HashSet の要素のオブジェクトのクラスがミュータブルであることに起因するバグが頻発するのであれば，そのバグに対応するために当該クラスがイミュータブルに修正されると著者らは考えた．そのため，プロジェクト全体におけるイミュータブルクラスの割合に比べ，HashMap のキーや HashSet の要素のオブジェクトのクラスにおけるイミュータブルクラスの割合が大きいのであれば，HashMap のキーおよび HashSet の要素に使用されるオブジェクトのクラスをイミュータブルクラスにする必要性は高いと結論付けられる．反対に，プロジェクト全体におけるイミュータブルクラスの割合に比べ，HashMap のキーや HashSet の要素のオブジェクトのクラスにおけるイミュータブルクラスの割合に大きな変化が無ければ，HashMap のキーや HashSet の要素のオブジェクトのクラスをイミュータブルにする理由は特別存在しないと結論付けられる．

RQ2: HashMap のキーおよび HashSet の要素に使用されるオブジェクトのクラスがミュータブルであることに起因するバグは存在するのか

HashMap のキーや HashSet の要素のオブジェクトのクラスがミュータブルであることに起因するバグが発見できれば，HashMap のキーおよび HashSet の要素に使用されるオブジェクトのクラスをイミュータブルに変更すべき根拠となりえる．反対にそのようなバグの事例が見つからなければ，HashMap のキーおよび HashSet の要素に使用されるオブジェクトのクラスをイミュータブルにすべきである根拠は乏しくなるため，HashMap のキーおよび HashSet の要素に使用されるオブジェクトのクラスをイミュータブルにする必要性は低くなると考えられる．

4. 調 査 方 法

RQ1 に回答するために，2.2 で挙げた実プロジェクト 107 件を対象に解析を行い，調査対象のイミュータブルクラスの実装割合を調べる．調査対象のクラスは，プロジェクト作成者がクラスをミュータブルからイミュータブルに変更可能である，プロジェクト内で定義されたクラスに限定する．また，RQ2 に回答するために RQ1 の調査中に検出されたイミュータブルクラスについて過去にミュータブルクラスであったかを調べ，そのクラスが HashMap のキーあるいは HashSet の要素であることに起因するバグによって変更されたのかを調査する．

4.1 RQ1

説明を簡素にするため，HashMap のキーについて解析する手順に絞って説明する．調査の流れを図 3 に示す．調査にはプロジェクトのソースコードファイルと Jar ファイルを用い，以下の順で行う．

1. HashMap または Map のキーに使用されたクラスを抽出する
2. プロジェクト内の各クラスのイミュータビリティを判定する
3. 2. の結果から 1. で抜き出したクラスの解析結果を抽出する

各手順について以下で詳しく説明する．

4.1.1 HashMap または Map のキーに使用されたクラスを抽出する

プロジェクトのソースコードに対して AST 解析を行い，HashMap もしくは Map が宣言されている箇所を抽出する．このとき，プロジェクト内で定義されたクラスが調査対象のため，HashMap のキーに使用されている型が基本データ型のラッパークラス(注6)あるいは String 型である場合，抽出の処理対象とはならない．

例えば，図 4 のコードでは，1 行目および 2 行目は HashMap もしくは Map が宣言されていないため無視する．3 行目は integerMap

(注6)：Boolean, Character, Byte, Short, Integer, Long, Float, Double

```
1 int b = 1; //無視
2 System.out.println(b); //無視
3 Map<Integer, String> integerMap = new HashMap<>(); //無視
4 Map<Obj, String> objMap = new HashMap<>(); //Objを抽出
```

図 4　AST 解析で抽出されるコード（HashMap の場合）

が HashMap のデータ型として宣言されているものの，キーとして使われているクラスがが基本データ型のラッパークラスである Integer であるため無視する．最後に，4 行目は HashMap が宣言されており，キーに使われているクラスが基本データ型のラッパークラスや String 型のクラスではない objMap であるため，キーとして使われているデータ型の Obj を抽出する．

なお，エラーにより解析できなかった Java ファイルについては，ファイル内で “Map” あるいは “HashMap” という文字列が存在するかどうかを出力する．存在する場合，該当のファイルの Map または HashMap のデータ型が使われている箇所について目視で確認し，キーに使われているオブジェクトのクラスがプロジェクト内で定義されたクラスである場合は該当のキーの抽出を行う．

4.1.2　HashMap または Map のキーに使用されたクラスを抽出する

2.3 で説明した Mutability Detector を使用し，プロジェクトの Jar ファイル内を解析して各クラスがイミュータブルであるかを出力する．

4.1.3　4.1.2 の結果から 4.1.1 で抜き出したクラスの解析結果を抽出する

イミュータビリティの出力から HashMap または Map のキーに使われているクラスの出力のみを表示させ，キーに使用されているクラスイミュータビリティの割合を算出する．イミュータビリティの割合の算出対象とするクラスは以下の例外を除いた全てのクラスである．

- COULD_NOT_ANALYSE と出力されたとき
- 同名のクラスが複数出力されたとき
- 当該プロジェクト内で定義されていないクラスである場合

1 つ目の COULD_NOT_ANALYSE と出力された場合は，イミュータブルクラスであるか判別できないため，プロジェクト内で定義されたクラスであっても無視する．2 つ目の同名クラスが複数出力された場合は，どのクラスが HashMap のキーに使われたクラスか判別できないため，割合の算出には用いない．3 つ目の当該プロジェクト内で定義されていないクラスは，例えば java.io.File などの Java の API で提供されているクラスなどが該当する．このような自身のプロジェクト内で定義されていないクラスをキーとして使用している場合，プロジェクト作成者によってイミュータブルクラスに変更することができないため，割合の算出には用いない．

HashSet についても，HashSet と Set の要素を対象として 4.1.1 から 4.1.3 の作業を行なう．

4.2　RQ2

RQ2 の調査について，以下の手順で行う．

1. 過去にミュータブルであったクラスを見つける
2. 1. のクラスがイミュータブルに変化した原因を調査する

各手順について以下で詳しく説明する．

4.2.1　過去にミュータブルであったクラスを見つける

まずは，バグ修正を行った可能性のあるクラスを見つけ出すため，ミュータブルからイミュータブルに変更されたことのあるクラスを探す．その方法として，RQ1 でイミュータブルクラスと出力されたクラスを対象とし，そのクラスが解析可能な最も古い Jar ファイルについてもイミュータブルクラスであるかどうかの解析を行い，過去にミュータブルであったクラスが存在するかを調べる．

解析に用いるのは基本的にそのクラスが含まれるプロジェクトの最も古い Jar ファイルであるが，以下の条件に当てはまる場合は次に古い Jar ファイルへと繰り上がっていく．

- Mutability Detector のエラーにより解析できなかったとき
- 解析対象のクラスがまだ未実装であったとき
- 解析対象のクラスで COULD_NOT_ANALYSE が出力されたとき

つまり，解析対象のクラスについて，NOT_IMMUTABLE, EFFECTIVELY_IMMUTABLE, IMMUTABLE のいずれかが出力されるまで Jar ファイルのバージョンを徐々に上げていく．

4.2.2　4.2.1 のクラスがイミュータブルに変化した原因を調査する

次に，ミュータブルからイミュータブルに変更されたクラスについて，どのような理由でミュータブルクラスからイミュータブルクラスに変更されたのかを調べ，ハッシュ値に起因するバグの事例が存在するかを調査する．当該クラスに対して，Jar ファイルが提供されている各バージョンでも Mutability Detector を用いてイミュータブルクラスであるかの判定を行い，ミュータブルからイミュータブルに変更されたバージョンを特定する．そして，当該プロジェクトのリポジトリのコミット履歴をたどり，ミュータブルクラスと最後に出力されたバージョンからイミュータブルクラスと最初に出力されたバージョンの間でどのような変更が行われたのかを確認する．変更された箇所やコミットメッセージを確認し，変更が HashMap のキーや HashSet の要素に使用したことに起因するバグであるかを調査する．

5.　調 査 結 果

5.1　RQ1

HashMap の調査結果を表 1 に，HashSet の調査結果を表 2 に示す．HashMap に使用されたオブジェクトのクラスのうち，Mutability Detector で解析できたクラスは 411 件あり，そのうちイミュータブルクラスは 61 件であった．HashSet に使用されたオブジェクトのクラスのうち，Mutability Detector で解析できたクラスは 548 件あり，そのうちイミュータブルクラスは 53 件であった．この結果から，プロジェクト内で定義されたクラスにおいて，HashMap のキーに使用されたクラスがイミュータブルクラスであった割合は 14.8%，HashSet の要素に使用されたクラスがイミュータブルクラスであった割合は 9.7%であった．

以上に調査により判明した，HashMap のキーに使われたクラ

スにおけるイミュータブルクラスの割合および HashSet の要素に使われたクラスにおけるイミュータブルクラスの割合をプロジェクト全体におけるイミュータブルクラスの割合と比較する．既存研究 [12] の調査では，Apache のライブラリにおいては 485 件のクラスのうち 69 件のイミュータブルクラスを検出していた．よって，Apache のライブラリにおけるイミュータブルクラスの実装率は 14.2%である．この割合と今回の調査結果を比較するとほとんど差がないことが分かる．この事実から，HashMap のキーや HashSet の要素のオブジェクトのクラスであっても，積極的にイミュータブルクラスにしているわけではないことが分かる．

RQ1 への回答：と HashMap および HashSet に用いられるイミュータブルクラスの割合はそれぞれ 14.8%，9.7%であり，プロジェクト全体のイミュータブルクラスの割合とほとんど差はない．

5.2 RQ2

HashMap のキーや HashSet の要素のオブジェクトのクラスがミュータブルであることに起因するバグの事例を調べるため，まず RQ1 で検出されたイミュータブルクラスについて，過去にミュータブルであったクラスを調べた．なお，Apache Lucene(注7)に関しては過去バージョンの Jar ファイルを Mutability Detector で解析できなかったため，該当プロジェクトの HashMap のキーに使用された 2 件のイミュータブルクラスについては調査できなかった．この 2 件を除いて，HashMap のキーのみに使用されたイミュータブルクラス 48 件，HashSet の要素のみに使用されたイミュータブルクラス 39 件，HashMap のキーと HashSet の要素の双方に使用されたイミュータブルクラス 14 件，計 101 件のイミュータブルクラスを調査した．

表 3　イミュータブルクラスの割合の比較

プロジェクト名		割合
[既存研究]Apache	プロジェクト全体	14.2%
Apache	HashMap	15.2%
	HashSet	8.1%
合計	HashMap	14.8%
	HashSet	9.7%

(注7)：https://github.com/apache/lucene

```
- public final class BarcodeFormat {
-
- // No, we can't use an enum here. J2ME doesn't support it.
+ public enum BarcodeFormat {
```

図 5　zxing の BarcodeFormat の変更（一部抜粋）

```
  private final String name;
  private final String type;
- private final Optional<SimpleDomain> domain;
```

図 6　presto の Column の変更（一部抜粋）

```
- public class CHPX extends BytePropertyNode
+ public final class CHPX extends BytePropertyNode
```

図 7　poi の CHPX の変更

101 件のうち，過去にミュータブルだったクラスは，ZXing(注8)の BarcodeFormat と ResultMetaDataType，Presto(注9)の Column，Apache POI(注10)の CHPX の 4 件であった．

次に，当該クラスがミュータブルからイミュータブルに変更された時期を調査する．各プロジェクトについて，バージョンごとにイミュータブルクラスであるかの解析を行ったところ，ZXing の BarcodeFormat と ResultMetaDataType は ver1.7 から ver2.0 にて，Presto の Column は ver0.119 から ver0.120 にて，Apache POI の CHPX は ver3.5-beta5 から ver3.5-beta6 にて，当該クラスがミュータブルからイミュータブルに変更されていた．

最後に，当該クラスが，ミュータブルからイミュータブルに変更された理由を調査する．各クラスがイミュータブルに変更されたバージョン間のコミット履歴を辿った．ZXing について，イミュータブルクラスに変更されたバージョン間で BarcodeFormat は 2 回，ResultMetaDataType は 1 回変更がなされていた．その中でイミュータブルクラスに変更された要因は，図 5 のように列挙型のクラスに変更したためであると考えられる．Presto

(注8)：https://github.com/zxing/zxing
(注9)：https://github.com/prestodb/presto
(注10)：https://github.com/apache/poi

表 1　HashMap のキーの調査結果

	Domains	Apache	合計
Map または HashMap を検出した数（基本データ型，String 型を除く）	2,795	5,700	8,495
検出されたキーのオブジェクトの種類	628	1,124	1,752
プロジェクト内で定義され，ツールにより解析できたクラスの数	135	276	411
イミュータブルであったクラス	19	42	61
イミュータブルクラスの割合	14.1%	15.2%	14.8%

表 2　HashSet の要素の調査結果

	Domains	Apache	合計
Set または HashSet を検出した数（基本データ型，String 型を除く）	2,364	4,920	7,284
検出された要素のオブジェクトの種類	786	1,355	2,141
プロジェクト内で定義され，ツールにより解析できたクラスの数	189	359	548
イミュータブルであったクラス	24	29	53
イミュータブルクラスの割合	12.7%	8.1%	9.7%

の Column はイミュータブルクラスに変更されたバージョン間で 1 回だけ変更されていた．イミュータブルクラスに変更された要因は，図 6 の通り，クラス内部のミュータブルフィールドを削除したためであると考えられる．このフィールドを削除した理由は，そのフィールドの基クラスを削除したためであり，HashSet の要素に Column を使用していることは関係ない．Apache POI の CHPX はイミュータブルクラスに変更されたバージョン間で 1 回だけ変更されていた．イミュータブルクラスに変更された要因は，クラスに final 修飾子を付けたことでサブクラス化できなくなったためであると考えられる．

この調査結果から，HashMap および HashSet にミュータブルクラスを使用したことに起因するバグの実例は見つからなかった．

RQ2 への回答：今回の調査において，HashMap および HashSet にミュータブルクラスを用いたことに起因するバグの実例は全く見つからなかった．

6. 考　　察

RQ1 の調査結果から HashMap および HashSet に用いられるオブジェクトのクラスにおけるイミュータブルクラスの割合はプロジェクト全体のイミュータブルクラスの割合と差がないことが判明した．この結果から，実際のプロジェクトでは，クラスがミュータブルであっても問題なく運用されていると結論付けることができ，HashMap および HashSet に用いられるオブジェクトのクラスをイミュータブルにする必要性は低いといえる．

また，RQ2 の調査結果においても 100 件以上のイミュータブルクラスを対象にした調査にもかかわらず，HashMap および HashSet にミュータブルクラスを使用したことに起因するバグの実例を発見することはできなかった．この結果から，HashMap のキーや HashSet の要素のオブジェクトのクラスがミュータブルであることによるバグが起こる事例はめったにないと考えられる．

以上の調査より，HashMap のキーおよび HashSet の要素に用いられるオブジェクトのクラスをイミュータブルにする必要性は低いと結論付けられる．

7. 妥当性の脅威

RQ1 の調査では，キーや要素に使用されているオブジェクトのクラスのうち，イミュータブルであるか判別できたプロジェクト内クラスの割合は HashMap では 23.5%，HashSet では 25.6% である．そのため，今回のプロジェクトを網羅的に解析できれば，イミュータブルクラスの割合が変動したり，バグの実例が見つかる可能性がある．

RQ2 の調査では，プロジェクトのすべての Jar ファイルに対してイミュータブルクラスであるかの判定を行ったわけではない．そのため，イミュータブルクラスが途中でミュータブルクラスに変化し，再度イミュータブルクラスに変更された事例が存在する可能性がある．

8. あ と が き

本研究では，HashMap のキーや HashSet の要素に使用されるオブジェクトの基となるクラスという側面から，イミュータブルクラスを実装する必要性を調査した．調査の結果，HashMap のキーや HashSet の要素のオブジェクトのクラスをイミュータブルとする必要性は低いことが分かった．

今後の課題として，別の側面におけるイミュータブルクラスを実装する必要性の調査が挙げられる．例えば，イミュータブルクラスは並列プログラミングにおいて同期を必要としないため，排他制御を最小限に抑え実行速度を向上させることができるともいわれている．このようなイミュータブルクラスが持つ様々な利点についてそれぞれ検証を行うことで，イミュータブルクラスの総合的な価値を調べることに繋がると考える．

謝辞　本研究は JSPS 科研費（JP20H04166，JP21K18302，JP21K11820，JP21H04877，JP22H03567，JP22K11985）の助成を得て行われた．

文　　献

[1] Oracle, “HashSet (Java SE 17 & JDK 17),” accessed 2022-06-12. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HashSet.html.
[2] Oracle, “Map (Java SE 17 & JDK 17),” accessed 2022-06-12. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html.
[3] J. Bloch, Effective java, Addison-Wesley Professional, 2008.
[4] stackoverflow, “Immutable objects and hashmap keys [closed],” accessed 2022-06-12. https://stackoverflow.com/questions/20212440.
[5] Quora, “Why should the HashMap key be immutable in Java?,” accessed 2022-06-12. https://www.quora.com/Why-should-the-HashMap-key-be-immutable-in-Java.
[6] stackoverflow, “Why is it preferred to use immutable objects as elements of a Set?,” accessed 2022-06-12. https://stackoverflow.com/questions/37335407.
[7] A. Potanin, J. Östlund, Y. Zibin, and M.D. Ernst, “Immutability,” pp.233–269, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. https://doi.org/10.1007/978-3-642-36946-9_9
[8] K. Wnuk, B. Regnell, and B. Berenbach, “Scaling up requirements engineering – exploring the challenges of increasing size and complexity in market-driven software development,” Requirements Engineering: Foundation for Software Quality, eds. by D. Berry and X. Franch, pp.54–59, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
[9] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, “Glacier: Transitive class immutability for java,” In Proc. International Conference on Software Engineering (ICSE), pp.496–506, 2017.
[10] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull, “Exploring language support for immutability,” In Proc. International Conference on Software Engineering (ICSE), pp.736–747, 2016.
[11] MUTABILITY DETECTOR BLOG, “An Introduction To Mutability Detector (Part 1 of n),” accessed 2022-06-12. http://mutability-detector.blogspot.com/2010/07/introduction-to-mutability-detector.html.
[12] T. Roth, D. Helm, M. Reif, and M. Mezini, “Cifi: Versatile analysis of class and field immutability,” In Proc. International Conference on Automated Software Engineering (ASE), pp.979–990, 2021.