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プロジェクトにおける関数型イディオムの実態調査

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
〒 565–0871 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　プログラミング言語は新しいパラダイムの登場や，それに伴う新しいイディオムの導入といった進化を経てきている．オブジェクト指向言語の一つである Java も新たなイディオムを導入してきており，特に Java 8 では関数型プログラミングのイディオムを導入している．Java における関数型イディオムの導入はパラダイムのレベルでの進化であるといえる．しかしながら実際の開発現場で Java の関数型イディオムがどのように捉えられているのかは明確になっていない．そこで本研究では実際のプロジェクトから関数型イディオムを使用しているファイルの数を計測し，その結果をもとにコミットメッセージや issue の内容を調べ，関数型イディオムを採用する理由と採用しない理由を調査した．調査により，関数型イディオムを採用する場合はコードの記述量を削減することやパフォーマンスの向上を目的としており，関数型イディオムを採用しない場合は JDK 6/7 に対する後方互換性やデバッグ面での保守性を維持することを目的としているという結果が得られた．この結果から，プロジェクトの方針として可読性やパフォーマンスを向上させるのか保守性を維持させるのかによって，開発者が使用するイディオムを選択すればよいことが示された．

キーワード　プログラミング言語進化，Java，関数型プログラミング，イディオム，ラムダ式，Stream，Optional

1. はじめに

プログラミング言語は常に進化し続けている [1]．この言語の進化は，プログラムを構成する要素や部品をいかに分解し組み合わせるかという考え，すなわちプログラミングパラダイムの進化と常に隣り合わせである．よって言語の進化には，糖衣構文の追加や文法の拡張といったイディオム（記法）単位の変化に限らず，別種のプログラミングパラダイムを採用するといった，実装の方針やスタイルそのものに変化を生じさせるような劇的な変化も含まれる．

1995 年に登場した Java も進化を続ける言語の一つであり，2018 年現在において 9 度のメジャーバージョン（JDK 1.0 から Java SE 9）の更新を遂げている．特に 2014 年にリリースされた Java SE 8 では，関数型プログラミング [2] [3] の考えに基づいた様々な機能が採用された．具体的には，関数型インタフェースを実装するためのラムダ式や，関数型インタフェースを用いて配列や集合体を効率的に処理する Stream API などが代表的である．この進化により，長年 Java が採用し続けていた手続き型およびオブジェクト指向 [4] に加え，宣言型および関数型というパラダイムの利用が可能となった．これにより，型推論やループ削除による記述の簡素化・可読性の向上や，副作用排除による集合対処理の並列化・パフォーマンスの向上といった効果が見込める．

その一方で，Java における関数型パラダイムの機能（以降，関数型イディオムと呼ぶ）に対しては，いくつかの批判が存在する [5] [6] [7]．第一に，メソッドの呼び出し系列（スタック）が深くなるためデバッグが困難になる [5] とされている．ラムダ式や Stream API を用いることで，「何を行うか」という命令の系列ではなく「何を行いたいか」という宣言のみを簡素に記述できる．一方，その宣言の裏側で数多くのメソッド呼び出しが積み重なるため，何か問題が発生した際のスタックトレースの解釈が難しくなる．また実行環境に対する影響としては，Stream API は処理内容によってはむしろ速度低下を招く [6]，ラムダ式と Stream API の利用はメモリという観点では非効率である [7] といった指摘もある．

これら Java で追加された関数型イディオムが，実際の開発現場でどのように捉えられ利用されているかについては明らかにされていない．Java の新機能の利用の変遷に着目した研究として，Dyer らによる調査が存在する [8]．Dyer らは，様々なソフトウェア開発プロジェクトの 10 億を超える AST を対象として，Java の様々な新機能（アサーションや総称型，try-with-resources など）がどのように利用されてきたかを調査し報告している．しかしながら，この調査は JLS (Java Language Specification) のバージョン 2 から 4 が対象であり，JLS Java SE 8 Edition [9]，すなわち Java 8 での関数型イディオムに関する調査は行われていない．

本研究では，Java 進化の中で導入された関数型イディオムを対象として，その利用実態に関する調査を行う．対象とする関数型イディオムは代表的な関数型イディオムである，ラムダ式，

```
List<String> list = ...
list.forEach(s -> System.out.println(s));
```

図 1　ラムダ式の例

```
List<Integer> list = ...
list.stream()
    .filter(i -> i > 0)
    .forEach(i -> System.out.println(i));
```

図 2　Stream の例

Stream API，および Optional の 3 つである．調査では以下 3 つの Research Question（RQ）に答えることを目的とする．

- RQ1：関数型イディオムは受け入れられているのか
- RQ2：関数型イディオムを採用する理由は何か
- RQ3：関数型イディオムを採用しない理由は何か

2. Java 8 の関数型イディオム

2.1 ラムダ式

ラムダ式は単一のメソッドを持つインターフェース（関数型インターフェース）の機能を簡潔に表現するためのイディオムであり，無名関数の代わりに関数型インターフェースの機能を実装できる．図 1 にラムダ式の例を示す．この例では Collection 型の list 変数の全ての要素に対して，要素を標準出力に書き出すというラムダ式を適用している．

ラムダ式を用いることで，無名関数と比べて記述を簡素化でき可読性の向上が見込まれる．また，型推論を用いることでさらに簡潔な記述が可能となる．加えて，ラムダ式はメソッドの引数として渡すことができる，つまり値ではなく処理内容をメソッドの引数に渡すことができる．

2.2 Stream API

Stream は順次および並列なコレクション操作を実装するためのイディオムである．図 2 に Stream の例を示す．Collection 型の list 変数の stream() メソッドを呼び出すことで，Stream API の利用が可能となる．この例では，まず filter() にラムダ式を適用して正の値のみを抽出する．さらに，先ほどのラムダ式と同様の処理（要素の標準出力への書き出し）を適用している．

Stream を用いることで，簡潔かつ可読性の高い記述によって並列処理を実装することが可能となる．また，コードの抽象度が高くなるためコードの再利用が容易となる．

2.3 Optional

Optional は null もしくは null 以外のデータを持つオブジェクトである．図 3 に，Optional の例を示す．この例では，list 変数の 3 番目の要素が null である可能性を明示しつつ，null の場合は 0 を，そうでない場合はその値そのものを val 変数に格納する．

Optional を用いることで，null になる可能性があることを明示することができる．また，値が存在しない場合の処理を強制することができる．つまり Optional により安全なプログラムを書くことができる．

```
List<Integer> list = ...
int val = Optional.ofNullable(list.get(3))
                  .orElse(0);
```

図 3　Optional の例

3. Research Question

本研究では，Java の関数型イディオムが実際の現場でどのように捉えられ扱われているのかを調査する．調査を行うにあたり以下に示す 3 つの Research Question を設定した．

RQ1: 関数型イディオムは受け入れられているのか

2. 節で述べたように，Java の関数型イディオムには利点だけでなく様々な批判も存在する．しかし，実際の開発現場で Java の関数型イディオムがどの程度使用されているのかについては明らかにされていない．

RQ2: 関数型イディオムを採用する理由は何か

Java の関数型イディオムに対する批判がある中でも，関数型イディオムを利用している実際の開発プロジェクトも存在すると考えられる．こうしたプロジェクトがどのような理由で関数型イディオムを利用しているのかを定性的に調査する．この調査の結果は，関数型イディオムの採用を検討するプロジェクトへの一つの指針となり得る．

RQ3: 関数型イディオムを採用しない理由は何か

実際の開発現場における Java の関数型イディオムを使用しない理由は明確になっていない．また，Java の関数型イディオムに対して言われている批判が，実際の開発現場においてどの程度考慮されているのかも明らかではない．これらの疑問に答えるために調査を行う．

4. RQ1 の調査

4.1 調査目的

本調査の目的は，Java 8 リリース後約 4 年が経った現在，実際の開発現場で Java の関数型イディオムがどの程度採用されているかを明らかにすることである．

4.2 調査方法

図 4 に，調査に用いる指標の計測方法を示す．図 4 の左側には，あるリポジトリの改版履歴が示されている．この改版履歴には，各ソースコードに対するラムダ式の変更部分とそれ以外の変更部分が例示されている．また，図 4 の右側に示す値は，4 つの計測指標とそこから算出される指標を表している．RQ への回答には一番右側に示されている，関数型イディオムを使用しているファイルの割合を用いる．以降では，この値を**関数型イディオムの利用密度**と呼び，D_{idiom}（より具体的には D_{lambda} や D_{stream}）と表記することとする．

指標の計測及び算出について具体的に説明する．各リビジョンの Java ファイルについて，前リビジョンとの差分内容から文字列の検索を行うことで関数型イディオムを検出する．関数型イディオムと，差分内容から検索する文字列の対応を表 1 に示す．値の算出について図 4 の r3（revision 3）の例を用いて説明する．r3 での差分内容より，関数型イディオムを使用して

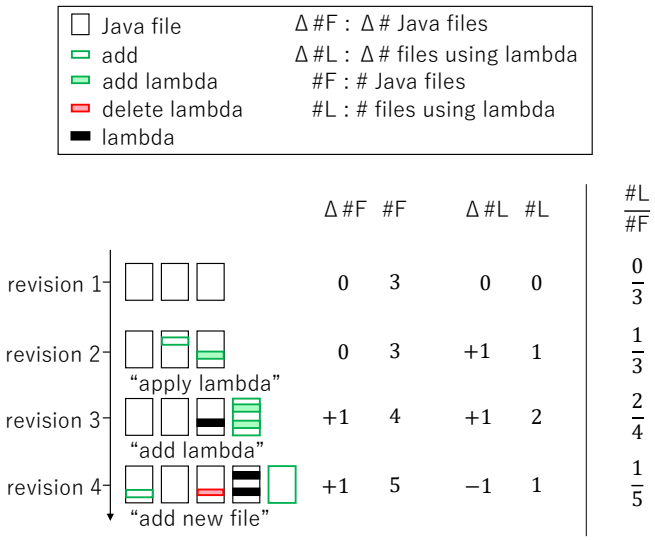

図 4　リポジトリの例と調査に用いる指標の計測方法

いるファイルの数は+1 変化しており，r2 での結果と合わせると r3 での関数型イディオムを使用しているファイルの数は 2 となる．これを全 Java ファイルの数である 4 で正規化することにより，r3 における計測目的である指標，$D_{\text{lambda}} = 2/4$ を算出する．以上のように，差分内容から文字列の検索によって関数型イディオムを検出することで調査を高速化でき，複数プロジェクトの多くのリビジョンに対して大規模な調査が可能となる．

4.3 調 査 対 象

本研究では，GitHub でのスター順検索上位 50 個の Java プロジェクトに対して調査を行う．スター順検索の上位プロジェクトを対象とすることで，広く知られており，かつ開発規模の大きなプロジェクトに対して調査が行えると考えた．各プロジェクトにおける対象リビジョンは Java 8 リリース日（2014 年 3 月 18 日）から調査実施日（2017 年 12 月 23 日）までの全リビジョンである．対象とするファイルは各プロジェクトの各リビジョンでの全 Java ファイルである．

表 2 に対象プロジェクトの Star 順上位 5 件を示す．この 5 件は，いずれも開発人数が 100 人以上，コミットの数が 1,500 以上で規模が大きく，かつ Star が 22,000 以上の広く知られているプロジェクトである．

4.4 関数型イディオム採用と不採用の基準

ここでは算出した D_{idiom} から，各プロジェクトがそのイディオムを採用しているか，否かを判断する基準について説明する．ラムダ式については最新リビジョンにおける D_{lambda} が 20%を超えるプロジェクトを「採用」と定義し，Stream と Optional については D_{stream} と D_{optional} それぞれが 10%を超えるプロジェクトを「採用」と定義する．以下，「採用」を定義する基準となる値（20%と 10%）を「境界値」と呼ぶ．Dyer らの調査によると，Java の新機能リリースから約 4 年後の，新機能を使用しているファイルの割合は約 10%である [8]．このことから Stream と Optional の境界値を 10%とした．ただし，ラムダ式はそれ単体で用いられる場合以外に Stream の中でも利用される場合も多いため，境界値は 20%とした．

Java 8 リリース日（2014 年 3 月 18 日）から調査実施日（2017 年 12 月 23 日）までの全リビジョンの中で一時的に境界値を超えるリビジョンが存在する場合，つまり一時は採用したが最新リビジョンでは「採用」とみなされないものを「取り消し」と定義する．また，調査対象の全リビジョンの中で一度も境界値を超えず，なおかつ最新リビジョンでも境界値を超えていないものを「不採用」と定義する．

表 1　関数型イディオムと検索する文字列の対応

関数型イディオム	検出する文字列
ラムダ式	`->`
Stream	`.stream(`
Optional	`Optional`

4.5 結　　果

採用/取り消し/不採用の割合：調査結果から，調査対象とした 50 プロジェクトの「採用」「取り消し」「不採用」それぞれの割合を図 5 示す．図に示す円グラフは左から，ラムダ式，Stream，Optional についての，50 プロジェクトを「採用」「取り消し」「不採用」に分類した結果を示している．

図 5 に示す結果より，「採用」プロジェクトの割合が最も高いのはラムダ式であり 8%である．次いで「採用」プロジェクトの割合が高いのは Optional の 4%であるが，この値はラムダ式の「採用」プログラムの割合の半分であることが分かる．一方で Stream の「採用」プロジェクトは 0%である．

図 5 の結果から，「取り消し」プロジェクトの割合が最も高いのはラムダ式であり 10%である．次いで「取り消し」プロジェクトの割合が高いのは Stream の 6%である．また，Optional の「取り消し」プロジェクトは 4%である．以上の結果から，「取り消し」の割合に対する「採用」の割合が高い関数型イディオム，つまり採用を取り消されにくい関数型イディオムは Optional であるといえる．

最新リビジョンにおける「採用」プロジェクトの割合：調査の結果から，各プロジェクトにおける最新リビジョンでの D_{idiom} の分布を図 6 に示す．横軸は各グラフが示す関数型イディオムの名前を，縦軸は D_{idiom} を示している．また，図中の青い破線は境界値を示している．

図 6 の結果から，D_{lambda} はほとんどのプロジェクトにおいて 10%以下であり，半分のプロジェクトでは 0%である．一方で，「採用」とみなされたプロジェクトの中でも PocketHub の D_{lambda} は 80%以上と突出した値であることが分かる．ま

表 2　対象プロジェクトのスター順上位 5 件の抜粋
（開発者数，Star の数，コミットの数）

Project	#contributors	#stars	#commits
RxJava	186	30,699	5,269
java-design-patterns	113	28,382	1,985
retrofit	115	26,241	1,529
okhttp	151	24,935	3,103
guava	136	22,018	4,640

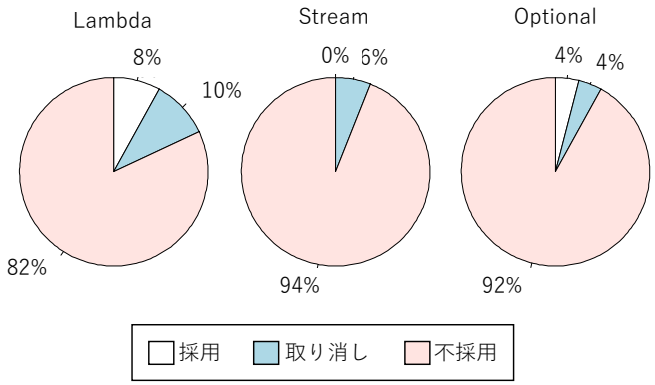

図 5　50 プロジェクトの「採用」「取り消し」「不採用」の割合

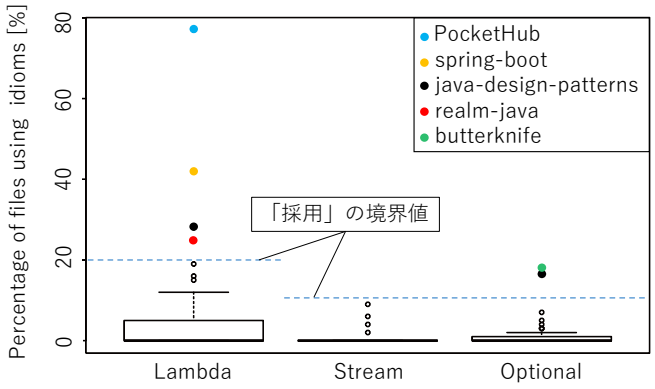

図 6　最新リビジョンにおける D_{idiom} の分布

た，ほかの「採用」プロジェクトでの D_{lambda} は，spring-boot は 40%以上，java-design-patterns と realm-java は 20〜30%の値となっており，PocketHub と比較すると「採用」プロジェクトの中でも D_{lambda} の値には大きく差があることが分かる．Stream については，ほぼすべてのプロジェクトに関して D_{stream} は 0%である．また，図 5 の結果と同様に，Stream の「採用」プロジェクトは 0 であることが分かる．Optional に関しては，半分のプロジェクトにおいて D_{optional} が 0%である．ただし，「採用」プロジェクトの値は突出しており，butterknife，java-design-patterns ともに 20%近くの値となっていることが分かる．

各リビジョンでの関数型イディオムを使用しているファイルの数の割合：各リビジョンにおける 3 つの関数型イディオムそれぞれの D_{idiom} の遷移を図 7 と図 8 に示す．図 7 には「採用」プロジェクトのみが，図 8 には「取り消し」プロジェクトのみが抜粋されている．なお，「不採用」プロジェクトは D_{idiom} の変化がほぼ見られないため，ここでは省略する．いずれの図も横軸は各リビジョンに対応する日付を，縦軸は D_{idiom} を示しており，図中の青い破線は「採用」の境界値を示している．

「採用」プロジェクトの傾向について述べる．まず，ラムダ式の結果より，PocketHub や spring-boot は D_{lambda} の値が急激に大きくなっていることが分かる．次に，Optional の結果より，butterknife はリリース後まもなく Optional を採用したが，2015 年 5 月に急激に採用を取り消した後，2015 年 12 月に再び Optional を採用していることが分かる．このような急激な D_{idiom} の値の変化が起こった時期に，プロジェクト内でラムダ式に関する方針の変更があったと考えられる．

次に「取り消し」プロジェクトに着目する．RxJava と spring-framework は D_{lambda} の値が急激に大きくなった後，急激に小さくなり結果として「取り消し」となっている．このような急激な D_{idiom} の値の変化があった時期にプロジェクト内での関数型イディオムに対する方針の変更があった可能性が高いと考えられる．また，spring-framework に関しては，「取り消し」となった後も D_{lambda} の値が緩やかに大きくなり続けているため，将来的に「採用」とみなされる可能性があると考えられる．図 8 に示す 3 つの関数型イディオムに対する結果を見比べると，spring-framework はラムダ式，Stream，Optional のいずれに関しても関数型イディオムを採用してから取り消すまでの期間が一致しており．RxJava は関数型イディオムを採用し始める時期のみが一致している．このように採用や取り消しの時期が一致している場合，その時期に関数型イディオムに対するプロジェクト内での方針に変更があったと考えられる．

図 7，8 の結果を見比べると，ラムダを「採用」しているプロジェクトに比べて，「取り消し」しているプロジェクトはラムダ式を使用し始める時期が早いことが分かる．また，ラムダ式の境界値を超えた時期も「取り消し」のプロジェクトは「採用」プロジェクトに比べて早い．つまり，現在「採用」とみなされているプロジェクトにおいても今後ラムダ式に関する方針の変更により「取り消し」となる可能性があると考えられる．

5.　RQ2/RQ3 の調査

5.1　調 査 目 的

本 RQ に答えることで，実際の開発現場ではどのような理由で Java の関数型イディオムを採用もしくは不採用にしているのかが分かる．また，採用する理由，不採用にする理由を明らかにすることで，今後の Java プロジェクト開発における一つの指針になるのではないかと考える．

5.2　調 査 方 法

RQ1 の結果に基づいて，D_{idiom} が大きく変化する時期のコミットメッセージや issue を定性的に調べることで，各プロジェクトでの関数型イディオムを利用する理由，利用しない理由を調査する．調査は目視によって行う．

上記の方法に加えて，さらに広く調査を行うために，GitHub の検索クエリを組み合わせてコミットメッセージや issue，コメント文を調査する．具体的には，表 3 に示すクエリ 1 とクエリ 2 を組み合わせて調査を行った．なお，この方法における調査対象は GitHub 上の全プロジェクトであることに注意されたい．

5.3　結　　果

調査により明らかとなった関数型イディオムを採用もしくは不採用とする理由を述べる．realm-java がラムダ式を採用する理由は「関数型イディオムを使用している RxJava2 の仕様に変換するため」[(注1)]であると明記している．これは，realm-java

(注1)：https://github.com/realm/realm-java/commit/9ac6893f1

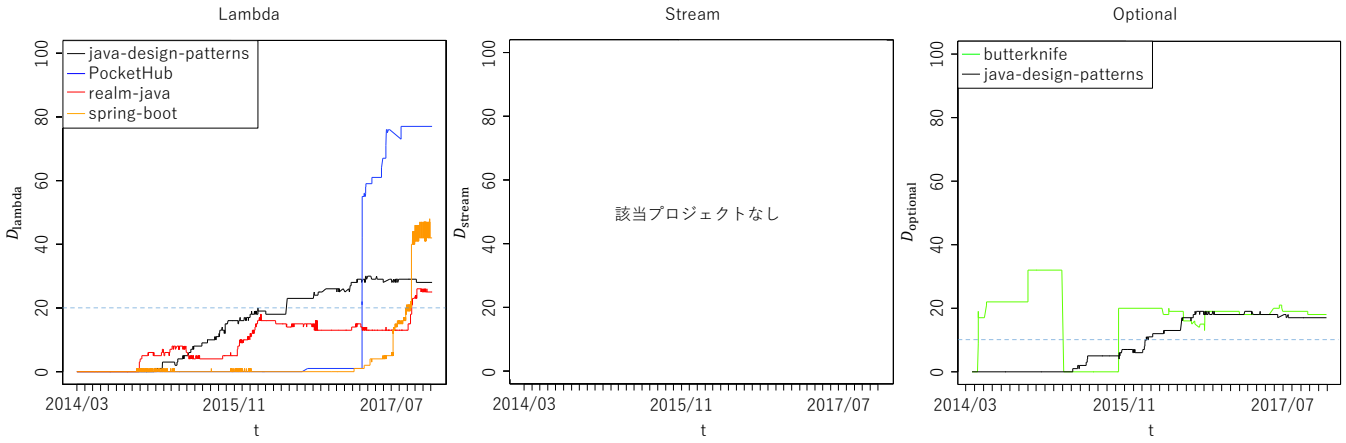

図 7 「採用」とみなされたプロジェクトの各リビジョンにおける D_{idiom}

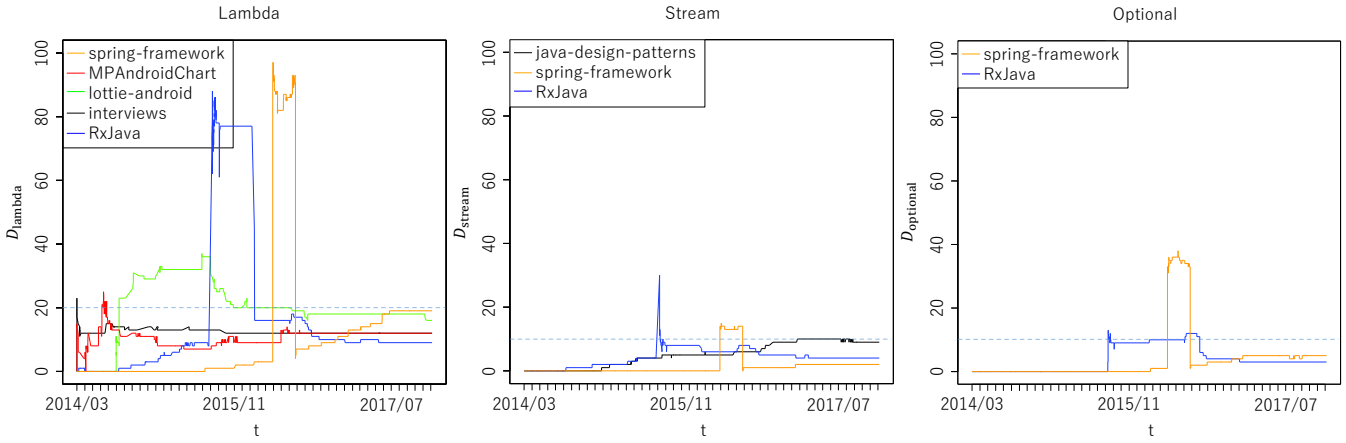

図 8 「取り消し」とみなされたプロジェクトの各リビジョンにおける D_{idiom}

がサポートしようとしている RxJava2 が Java の関数型イディオムを使用しているため，realm-java でも関数型イディオムを使う必要があるためである．guava がラムダ式と Stream を採用する理由は「Stream を使うことで Guava のパフォーマンスを向上させるため」(注2)である．PocketHub がラムダ式を採用する理由は「コードを簡潔に書くため」(注3)である．retrofit が Optional を採用する理由は「Optional へのコンバータを作るため」(注4)である．これは，変数を Optional でラップするコンバータを作ることで Optional の使用を容易にするということである．

RxJava が 3 つの関数型イディオムを採用しない理由は「JDK 6 との相性をよくするため」(注5)である．これは，RxJava が対象としているプロジェクトは JDK 6 を使用して開発されているためである．Hystrix が 3 つの関数型イディオムを採用しない理由は「JDK 6/7 でビルド可能にするため」(注6)である．lottie-android がラムダ式を採用しない理由は「ビルドやインストールで問題が生じないようにするため」(注7)である．GraalVM が Stream を採用しない理由は「スタックをオーバーフローさせるのが早いため」(注8)である．GraalVM では，Stream はほかの繰り返し処理と比べて発生させるスタックフレームが多いためスタックのオーバーフローを早く起こしてしまう，という理由から Stream を使用していない．

表 3 RQ2/RQ3 の調査に用いた検索クエリ

クエリ 1	クエリ 2
java 8	use
lambda	accept
Stream	remove
optional	replace
	refactor
	instead of

6. 議　　論

RQ1: 関数型イディオムは受け入れられているのか

図 6 に示す結果から，各関数型イディオムを採用しているプロジェクトの数はいずれも 50 プロジェクト中の 10%未満である．また，図 5 に示す結果から，関数型イディオムを取り消したプロジェクトの数は採用しているプロジェクトの数以上である．以上の理由から，関数型イディオムは実際の開発現場で広

(注2)：https://groups.google.com/forum/#!topic/guava-announce/o954PqvaXLY/discussion
(注3)：https://github.com/pockethub/PocketHub/issues/1055
(注4)：https://github.com/square/retrofit/commit/e985d552
(注5)：https://github.com/ReactiveX/RxJava/commit/000a174d8
(注6)：https://github.com/Netflix/Hystrix/commit/e102e5db
(注7)：https://github.com/airbnb/lottie-android/commit/fa2390e
(注8)：https://github.com/oracle/graal/commit/bca7ce

く受け入れられているとは言いきれない．

RQ2: 関数型イディオムを採用する理由は何か

5.3 に示す調査の結果より，関数型イディオムを採用しているプロジェクトの採用理由は 2 種類ある．1 つ目の採用理由はコードの記述量を削減するため，つまり可読性の向上を図るためであるといえる．2 つ目は，guava が Stream を採用する理由として示している「パフォーマンスを向上させるため」という理由である．一方で，「Java の Stream は速度が遅い」という批判もある [6]．guava の示した採用理由には具体的にどういったパフォーマンスの向上を目的としているのかは記述されていないため，速度以外でのパフォーマンスが向上するのではなかと考える．

RQ3: 関数型イディオムを採用しない理由は何か

5.3 に示す調査の結果より，関数型イディオムを採用しない理由は 2 種類ある．1 つ目は JDK 6 や JDK 7 もしくはそれらを使用しているツールをサポートするため，つまり後方互換性のためであるといえる．2 つ目は関数型イディオムを使用することによりスタックが深くなり，デバッグが行いにくくなることを防ぐ，つまり保守性を維持するためであるといえる．また，「保守性（デバッグの行いやすさ）の維持」という不採用理由は，Java の関数型イディオムを使うことでスタックが深くなりデバッグが困難になるという批判 [5] を支持する事例であるといえる．

実際の開発現場における Java の関数型イディオム

各 RQ における議論より，Java の関数型イディオムは実際の開発現場で積極的に採用されているとは言い切れず，特に保守性を維持したいと考えているプロジェクトでは使わないほうが良いことが分かる．一方で，コードの記述量を削減することやパフォーマンスの向上を考えるプロジェクトでは関数型イディオムを採用する利点がある．

上記の事実から，関数型イディオムとそれに対応するイディオムの置換を提案するツールの開発が求められる．この置換提案ツールにより，保守性の維持を求めるプロジェクトにおいては提案された関数型イディオムへの置換を行わず，可読性やパフォーマンスの向上を求めるプロジェクトでは提案された関数型イディオムへの置換を行うといったような開発者による選択が可能となり，また，ツールを用いることで容易な置換が行えるようになる．

7. 妥当性への脅威

本研究では調査対象として GitHub での Star 順検索上位 50 プロジェクトを選定したが，対象とするプロジェクトの開発規模，対象とするプロジェクトの数などを変更することで本研究とは異なる結果が得られる可能性がある．また，文字列の検索により Java ファイルから関数型イディオムを検出したため，本来関数型イディオムではない記述を検出している可能性がある．検出結果を目視確認しているが，この目視確認が間違っている可能性もある．そのため，AST を用いてイディオムを検出した場合の結果と異なる可能性がある．

今回は関数型イディオムを利用するファイル数が大きく変化した時期付近のコミットや issue を調査したが，ほかの時期のコミットや issue にも，本研究では把握していない採用/不採用の理由が書かれている可能性がある．

8. おわりに

本研究では，実際の開発現場における Java の関数型イディオムがどう捉えられ扱われているのかを調査するために 3 つの RQ を設定して調査を行った．調査の結果，プロジェクトの方針として重点を置くポイントが保守性なのかソースコードの可読性やパフォーマンスなのかによって関数型イディオムの利用を選択すればよいといえる．

今後の課題として，Java 以外のプログラミング言語に関して，別のパラダイムから導入されたイディオムが実際の現場でどう扱われているのかを調査することが考えられる．また，関数型イディオムを使った場合のスタックトレースについて，関数型イディオムによって裏側で呼び出されたメソッドによるスタックと，そうでないスタックを区別できるツールの開発が考えられる．これによりスタックが深くなることでのデバッグが行いにくくなる問題を解決できる可能性があると考える．

謝辞

本研究は，日本学術振興会科学研究費補助金基盤研究（S）(課題番号：JP25220003)，および文部科学省研究費補助金若手研究（B）（課題番号：JP26730155）の助成を得て行われた．

文　献

[1] P.J. Landin, “The next 700 programming languages,” Communications of the ACM, vol.9, no.3, pp.157–166, 1966.

[2] P. Wadler, “The essence of functional programming,” Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp.1–14, 1992.

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

[4] B. Meyer, “Object-Oriented Software Construction”, Prentice-Hall, Inc., 1988.

[5] R. Fischer, “Java closures and lambda, chapter 7: Lambdas and legacy code,” pp.111–138, Apress, 2015.

[6] A. Zhitnitsky, “The 6 biggest problems of Java 8 - JAXenter”. visited on 2018-02-13. https://jaxenter.com/java-8-problems-112279.html

[7] Y. Cheon and A.E.D.L. Torre, “Impacts of java language features on the memory performances of android apps,” Technical report, University of Texas at El Paso, 2017.

[8] R. Dyer, H. Rajan, H.A. Nguyen, and T.N. Nguyen, “Mining billions of ast nodes to study actual and potential usage of java language features,” Proceedings of the 36th International Conference on Software Engineering, pp.779–790, 2014.

[9] J. Gosling, B. Joy, G. Steele, G. Bracha, “Java(TM) Language Specification. Java SE 8 Edition,” 2015.