GitHub 上の Java コードにおける メソッドチェイン利用の定量的な分析

中丸 智貴　千葉 滋

メソッドチェインスタイルでのプログラム記述は広く受け入れられていると言われている. しかしながら, そのことを示す定量的な結果は我々の知る限りでは存在しない. 本研究では「メソッドチェインスタイルが広く受け入れられているかどうか」を定量的に明らかにするため, メソッドチェインの出現数の経年変化を調査した. 分析の結果, ソースコード 1 行あたりの長さ 2 以上のメソッドチェインの出現数は 2008 年から 2018 年まで増加し続けていることが明らかになった. また, チェインされていないメソッド呼び出し数に対する, チェインされたメソッド呼び出し数の比率が 2008 年から 2018 年の間に 15%から 28%まで増加していることも明らかになった.

1　はじめに

メソッドチェインスタイルでの記述は広く受け入れられていると言われている [3–5, 7, 9, 14, 15]. メソッドチェインスタイルとは, 図 1 のように, メソッド呼び出しを一つの式中で連鎖させるプログラミングスタイルである. メソッドチェインスタイルでの利用に適したインターフェイス (fluent interface) を持つライブラリは, 様々なオブジェクト指向言語において開発されている [8, 12, 13]. また近年 fluent interface を持つライブラリの構築を支援するツールが研究者によって提案されている [4, 7, 9, 14, 15].

しかしながらメソッドチェインスタイルが広く受け入れられていることを示す定量的な結果は, 我々の知る限りでは, 存在しない. 統合開発環境などにメソッドチェインスタイル支援機能を追加するにもコストがかかることや, 研究者の作ったツールを製品レベルにするのにもコストがかかることなどを考えると, 現実のプログラマの需要を定量的に示すことは重要である. 客観的な情報を元に開発の是非や優先度などを判断できるようになるからである.

そこで本研究では「メソッドチェインスタイルが広く受け入れられているかどうか」を定量的に明らかにするための調査を行なった. 我々は GitHub 上の 3,461 個のレポジトリの各年末リビジョンから Java ソースコードを抽出し, それらの中のメソッドチェイン出現数の経年変化を調査した. 分析の結果, ソースコード 1 行あたりの長さ 2 以上のメソッドチェインの出現数は 2008 年から 2018 年まで増加し続けていることが明らかになった. また, チェインされていないメソッド呼び出し数に対する, チェインされた呼び出し数の比率が 2008 年から 2018 年の間に 15%から 28%に増加していることも明らかになった.

```
// Java Stream API
strList.stream()
        .filter(s -> !s.isEmpty())
        .map(s -> s + ".")
        .collect(toList());
```

図 1　メソッドチェインスタイルでの記述

2　背景

メソッドチェインスタイルは広く受け入れられていると言われている. メソッドチェインスタイルで記述することでソースコードの可読性が向上すると考えられているためである. 具体的には, 同じレシーバを

東京大学大学院 情報理工学系研究科
This is an unrefereed paper.

繰り返して書かずに済む [6,9], 意味的にまとまりのある部分を一つの式にまとめることができる [9,14], 左から右に自然言語のように読める [1], などが可読性の向上点として挙げられている. 中間変数が減ることもメソッドチェインスタイルが支持される理由に挙げられる [6]. 中間変数が減れば, 中間状態の命名に悩む機会が減り, どのような変数が存在するのかを気にする機会が減るなどして, 生産性が高まると考えられるためである.

メソッドチェインスタイルと, それを実現するライブラリ実装方法には様々な欠点も指摘されている. 具体的には, 多くの 統合開発環境が提供する行レベルの breakpoint 設定機能ではデバッグが行いにくい [6], fluent interface を持つライブラリは継承による機能拡張が困難である [2,11], fluent interface の構築は開発コストが高い上に維持コストも増大させる [2,9,11], などが指摘されている.

上述したような定性的な議論はこれまでも行われているものの, メソッドチェインスタイルが広く受け入れられていることを定量的に示した結果は未だ存在しない. メソッドチェインスタイルの最大の利点である可読性についての議論は主観的な部分が大きく, 広く受け入れられている根拠とするには説得力に欠ける. ライブラリ提供者にとっては fluent interface の提供は負担が大きいと言われることを考えると, メソッドチェインスタイルで利用し易いライブラリは少なく, 結果的にメソッドチェインスタイルの記述は少ないとも考えられる. また, 仮に広く受け入れられていることは認めるにしても, 実際どの程度のメソッドチェインが書かれているのかは未知である.

メソッドチェインスタイルの利用に関する定量的な情報は, 基盤的ソフトウェア開発や研究の方針決定に役立つ. 例えば, メソッドチェインの途中に breakpoint を設定する機能の提供がどれだけユーザの需要を満たすかを統合開発環境の開発者が見極めることができるようになる. また, 研究レベルで存在する fluent interface の構築を支援するツール [4,7,9,14,15] がどれだけ現実で役立つのか, それらのツールを製品レベルにする必要があるかを判断できるようにもなる.

```
List<String> list = new ArrayList();
list.add(
  createRandomString() // Length 1
); // Length 1
list.stream().map(str -> {
  return str
    .replace("foo", "bar")
    .replace("baz", "qux"); // Length 2
}).forEach(str ->
  System.out.println(str); // Length 1
); // Length 3
```

図 2　メソッドチェインとその長さ

3　手法

本研究ではメソッドチェインの出現数の経年変化の傾向を元に「広く受け入れられているか」を考察する. 現在いくつの出現があるかという情報のみから広く受け入れられているかを判断するのは困難であると考えたためである. 本研究ではメソッドチェインが増加傾向にあるのか, 減少傾向にあるのかを観察することで結論を導くことを目指した.

調査に用いるデータセットは, GitHub 上の 3,461 個のレポジトリの各年末リビジョンから抽出した Java ソースコードの集合である. データセット中のソースコードには, ソースコードが含まれるリビジョンの年, ソースコードが含まれるレポジトリ名が対応づけられている. データセット構築に用いる 3,461 個のレポジトリは, 2019 年 6 月 16 日から 2019 年 7 月 16 日の間に, スター数の多い 1000 個の Java レポジトリを取得する GitHub API のレスポンス含まれていたレポジトリである.

本研究では, メソッドチェインを 1 つ以上のメソッド呼び出しの連鎖と定義する. また, 連鎖したメソッド呼び出しの数をメソッドチェインの長さとして定義する. 例えば図 2 の Java コードには, 長さ 1 のチェインが 3 個, 長さ 2 のチェインが 1 個, 長さ 3 のチェインが 1 個含まれている. 我々の定義では全てのメソッド呼び出しはメソッドチェインの一部となる. 通常長さ 1 のチェインはメソッドチェインとは呼ばれないが, 後の分析での取り扱いの都合から, メソッドチェインとして取り扱った.

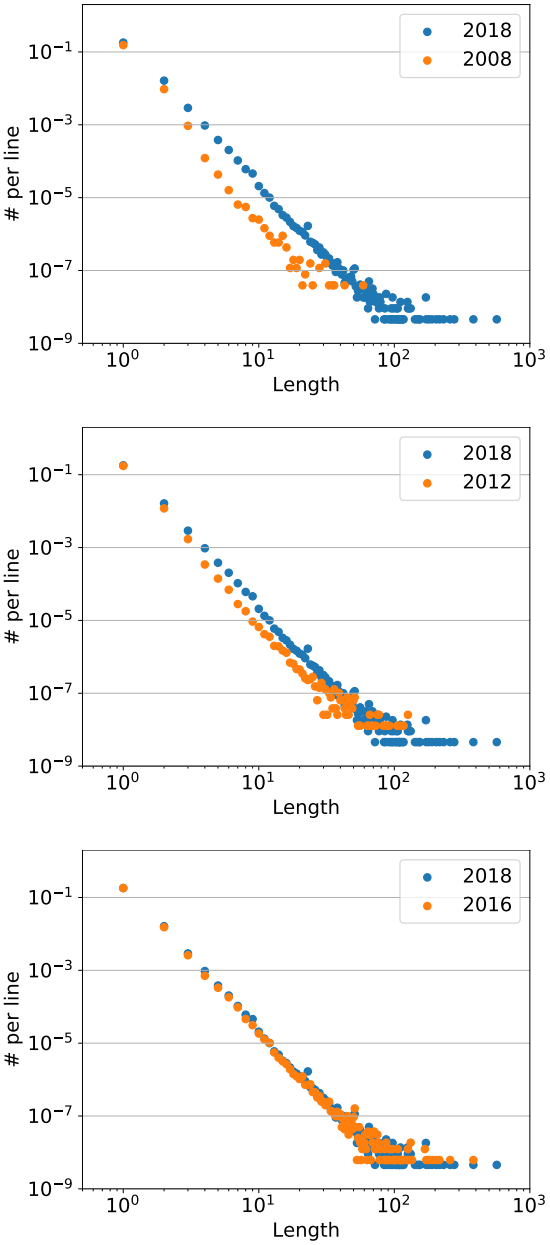

図 3　長さ vs. 1 行あたりの出現回数

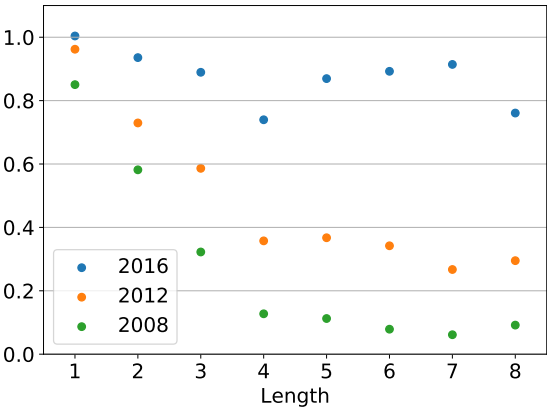

図 4　メソッドチェインの相対出現数

4　結果と考察

図 3 のグラフは 2008 年, 2012 年, 2016 年時点でのメソッドチェインの出現数分布をプロットしたものである. 横軸はメソッドチェインの長さ, 縦軸はソースコード 1 行あたりのメソッドチェインの出現数であり, どちらの軸も対数スケールを用いている. 各グラフには比較のため 2018 年の分布も合わせてプロットしている.

図 3 からわかるように, 新しい年ほど多くメソッドチェインが出現していることが観察できる. 特に 2008 年には存在しなかったような極端に長いメソッドチェインが 2012 年, 2016 年には現れていることが観察される.

どの年においても対数スケールにおいて線形な分布が現れていることも特徴的である. 対数スケールにおいて線形であるとは, 1 行あたりの出現数を n, チェインの長さを l として

$$\log n = a \log l + b$$

という形で表現できるということである. この式を変形すると

$$n = Cl^a$$

という形になり, 長さが長くなると出現数は急速に減少していることがわかる. 現象が冪乗則 (Rich-get-richer 現象) に従う場合対数スケールで線形な分布が現れることが知られている [10]. しかし対数スケールで線形な分布になることは, 必ずしも現象が冪乗則に従っていることを示しているわけではない [10]. そのため「メソッドチェインの出現分布は冪乗則に従う」と結論できないが, このような分布の生成過程のモデル考案と「冪乗則に従うのか」の検討は興味深い今後の課題である.

対数スケールでは観察を行いにくかった出現数の増加量を具体的に知るため, 2018 年時点で 30%以上の

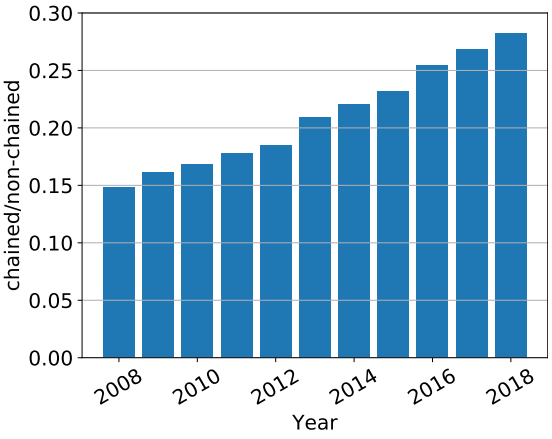

図 5　チェインされたメソッドの割合

レポジトリ出現していた長さ 1 から 8 までのメソッドチェイン出現数のさらに詳細な観察を行なった. 図 4 は, 2018 年の出現数を 1 とした場合の 2008 年, 2012 年, 2016 年における相対的な出現回数をまとめた図である. 図 4 では縦軸横軸共に通常スケールを用いている.

図 4 から, 長さ 4 から 8 のチェインは 2008 年から 2018 年で 10 倍程度, 2012 年から 2018 年に 2 倍以上に増加していることがわかる. 2016 年から 2018 年の増加量はそれ以前ほどに大きくないものの, 25%程度は増加していることが観察される. 長さ 2 と長さ 3 のチェインは長さ 4 以上の場合ほどの変化はないものの, 大きな増加が観察される.

図 5 はチェインされていないメソッド呼び出し数に対するチェインされているメソッド呼び出し数の比率をまとめた結果である. 例えば図 2 の Java コードの場合, チェインされているメソッドは 5 個, チェインされていないメソッドは 3 個と考え, 比率は約 166%となる. 図 5 からも, 年々メソッドチェインスタイルで記述される場合が増えていることが観察される.

5　今後の課題

前節に示した結果から, メソッドチェインの出現数が増加傾向にあることが観察された. しかしながら「メソッドチェインが広く受け入れられているのか」という問いを考える上ではさらに分析を進める必要がある. 具体的には以下に挙げるような点の調査と分析が必要である:

- プログラマが実際にメソッドチェインについてどのような印象を持っているのか? インタビューを行う必要はないか?
- Java 以外の言語ではどのような傾向が見られるのか? Kotlin のような Java に比べて文法が柔軟な言語でも同様に増加傾向にあるのか?
- GitHub においてスター数が上位のレポジトリはライブラリコードが多いが, アプリケーションコードでも前節と同様の結果になるのか?
- メソッドチェインとして記述できるのに, メソッドチェインスタイルで記述していない箇所は減少しているのか?
- 特定の規模の大きいレポジトリがメソッドチェインを多用しているだけで, 多くのレポジトリで増加していないのではないか?
- 一部の頻繁に使われるライブラリが fluent interface を提供しているだけで, 様々な部分で均等にメソッドチェインが均等に増えているわけではないのではないか?
- 2012 年から 2016 年の間に大きく出現数に変化が起こっているように観察されるが, Java プログラミングを取り巻く環境に変化があったのか?

さらにこの結果を現実世界のソフトウェア開発に大きく役立てて行くには, 以下のような調査と分析も必要になる:

- 非常に長いチェインはどのようなチェインか?
- 既存研究 [4, 7, 9, 14, 15] などでは型を用いた構文検査が提案されているが, この技術を応用できる場所はどれだけ存在するか? 特に文脈自由文法の検査技術が必要になる箇所はどれだけ存在するか?

6　まとめ

本研究では「メソッドチェインスタイルが広く受け入れられているかどうか」を定量的に明らかにする調査を行なった. 調査の結果, メソッドチェイン利用は大きく増加していることが明らかになった. さらなる調査による今回の結果の検証は今後の課題である.

参 考 文 献

[1] Bodden, E.: TS4J: A Fluent Interface for Defining and Computing Typestate Analyses, *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014.

[2] Bugayenko, Y.: Fluent Interfaces Are Bad for Maintainability, `https://www.yegor256.com/2018/03/13/fluent-interfaces.html`.

[3] Gil, Y. and Levy, T.: Formal Language Recognition with the Java Type Checker, *Proceedings of 30th European Conference on Object-Oriented Programming*, 2016.

[4] Gil, Y. and Roth, O.: Fling — A Fluent API Generator, *Proceedings of 30th European Conference on Object-Oriented Programming*, 2019.

[5] Grigore, R.: Java Generics Are Turing Complete, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.

[6] Ilari Kajaste et al.: OOP - Method chaining - why is it a good practice, or not? - Stack Overflow, `https://stackoverflow.com/questions/1103985/method-chaining-why-is-it-a-good-practice-or-not`. (Accessed on 08/02/2019).

[7] Levy, T.: *A Fluent API for Automatic Generation of Fluent APIs in Java*, PhD Thesis, Israel Institute of Technology, 2017.

[8] Michael Bayer et al.: SQLAlchemy - The Database Toolkit for Python, `https://www.sqlalchemy.org/`. (Accessed on 08/02/2019).

[9] Nakamaru, T., Ichikawa, K., Yamazaki, T., and Chiba, S.: Silverchain: a fluent API generator, *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2017.

[10] Newman, M. E.: Power laws, Pareto distributions and Zipf's law, *Contemporary physics*, Vol. 46, No. 5(2005), pp. 323–351.

[11] Pivetta, M.: Fluent Interfaces are Evil, `https://ocramius.github.io/blog/fluent-interfaces-are-evil/`.

[12] Skymind, Inc.: Deeplearning4j, `https://deeplearning4j.org/`. (Accessed on 08/02/2019).

[13] The jQuery Foundation: jQuery, `https://jquery.com/`. (Accessed on 08/02/2019).

[14] Xu, H.: *EriLex: An Embedded Domain Specific Language Generator*, 2010.

[15] Yamazaki, T., Nakamaru, T., Ichikawa, K., and Chiba, S.: Generating a fluent API with syntax checking from an LR grammar, *Proceedings of the ACM on Programming Languages*, No. OOPSLA(2019).