Revisiting Method Chaining in Java

Alexander Aldridge
University of Victoria
Victoria, BC, Canada
aaldridge@uvic.ca

Gary Fan
University of Victoria
Victoria, BC, Canada
garyfan12@uvic.ca

Tianna McDowell
University of Victoria
Victoria, BC, Canada
tiannamcdowell@uvic.ca

Glen McManus
University of Victoria
Victoria, BC, Canada
gmcmanus@uvic.ca

Abstract

Best practices are frequent topics of discussion amongst software developers. In recent years, method chaining has gained popularity, but its position as a best practice is in question. In 2020 Nakamaru et al. discovered that from 2010 to 2018 the relative frequency of method chains increased, as well as their length and maximum length [8]. We replicated their study with more recent data from GitHub to investigate if these trends continued into 2021. Using the top 1000 stared repositories, we measured the ratio (r) of method chain length, the frequency (f_n) of method chains by length, and the size categories denoted in Nakamaru et al. [8]. We also manually inspected over 5000 method chains as well as developed a simple automatic classifier to categorize method chains by type. Our findings show that not only did the frequency of method chaining continue to increase, but also the longest chains increased in length. Therefore we conclude method chaining continues to be adopted in the Java community as a best practice.

CCS Concepts: • **Software and its engineering → Software design engineering**; *Software libraries and repositories.*

Keywords: data sets, Java, method chaining

ACM Reference Format:
Alexander Aldridge, Gary Fan, Tianna McDowell, and Glen McManus. 2021. Revisiting Method Chaining in Java. In *Proceedings of SENG480B (UVIC'21).* UVIC, Victoria, BC, Canada, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

This paper outlines our replication of Nakamaru et al.'s "An Empirical Study on Method Chaining in Java" [8]. Published by the Association for Computing Machinery in June 2020, the authors consist of a combination of professors, assistant professors, and PhD students at the University of Tokyo.

Nakamaru et al. investigated trends in method chaining usage in Java using data obtained from the GitTorrent data set. The authors use the JavaParser [5] library to identify method chains. By identifying an "increasing trend of method chaining" [8] the paper concludes that method chains were adopted as a best practice by the community from 2010 to 2018. We extend this paper to include data from 2019 to 2021 to see if these trends continue.

We selected this paper due to our diverse experiences with method chains. Some team members find them opaque and difficult, while others enjoy their procedural and functional nature. All of us have worked with Java, and most of us will likely continue to do so, hence, determining whether or not method chains are a best practice impacts us directly.

The paper contributes to:

- The investigation of the adoption of method chaining within the Java community as a best practice.
- Recommendations for when to use method chaining.
- Investigating what method chains are being used for.
- A naive implementation of an automated classifier of method chain types which may be extended and improved for future use.
- Two data sets, one of 903 repositories with test code and non-test code combined, one of 901 repositories with test code and non-test code separated.

The remainder of our paper is as follows. Section 2 discusses on research on related articles and the lack thereof in modern research. Section 3 shows our methodology and changes or additions we made to Nakamaru et al.'s original methodology. Section 4 discusses our findings and in specific how our results compare to the original paper's metrics. Section 5 focuses on threats to validity and the limitations of our work. Section 6 talks about each of our research questions and how they relate to our findings. Section 7 outlines the potential for future work.

2 Related Work

Searching the ACM digital library and GoogleScholar shows that the paper by Nakamaru et al. [8] is the only paper covering the domain of method chaining in Java. However, there have been several empirical studies that relate to other language features such as generics[9][2], lambda expressions[4], and cast operators[7]. In a study from Gyori et al.[1] they develop a tool to refactor code into using method chaining in order to make code more succinct. A study by Zou et al.[10] discusses how code style consistency within a code base has an impact on the speed of Pull Requests being merged. Fluent interfaces as discussed by Fowler [6] have gained widespread acknowledgement; fluent interfaces use method chaining to mimic domain specific language, aiming to make code more readable.

3 Methodology

Wherever possible, we attempted replicating Nakamaru et al. [8] as closely as we could. Since the original software for the study is unavailable, a large portion of our work went into recreating the work done by the original study. We compiled a similar data set to Nakamaru et al., but with more recent repositories. Then we cleaned and analysed the data in R-Studio and Python. Metrics analysed included:

- lines of code per repository,
- method chain counts per size per repository,
- method chain type usage per chain length category
- total method calls per repository

3.1 Data Retrieval from GitHub

We implemented a script in Python which utilized the Github API to scrape the 1000 most starred Java repositories. First the script queried a list of all the links for the top 1000 most stared repositories from the Github API. It then clones one repository from the list and moves all files with the .java extension to a new directory with the
`root/<git_organization>--<git_repo_name>/*.`
naming convention. Then it removes the cloned repository to save memory on the machine and repeated this process for all repositories in the list. The GitHub API returns duplicates when scraping for the top 1000 repositories, so the Python code checks if it has already processed the repository before copying it's Java files to the new directory. After removing all duplicates, 903 repositories remained. Initially, we did not separate the test files from non-test files, so we modified our retrieval process to do that. When updating our retrieval process, differences in the GitHub API's state caused us to get fewer repositories back. This second but similar data set contains 901 repositories and 649,276 files. These will be referred to as the combined and separated data sets specifically.

3.2 Research Questions

We aim to answer the following questions:

- **RQ-1:** Is method chaining still popular today?
- **RQ-2:** Is method chaining primarily used in production code or testing code?
- **RQ-3:** What are typical lengths for method chains?
- **RQ-4:** Has the usage of method chains as classified by Nakamaru et al. changed?
- **RQ-5:** When should method chaining be used?

Method chains were defined as one or more method calls joined by a period [8]. The length of that chain is therefore the number of methods chained together [8]. For example, 'myDog.bark()' has a method chain length 1, while 'myCat.purr().toString().toUpperCase()' would have a length of 3.

3.3 Identifying and Counting Method Chains

We use the JavaParser [5] to identify all the method calls in a Java file. Then we determine if methods are chained by checking their scope. After recording all the lengths of the chained methods, we output the data to a csv file. The file contains the count of each chained method per repository. Following Nakamaru et al. [8], we also pick some repositories for manual inspection. Due to time constraints, we sampled 8 repositories using a pseudo-random generator written in Python. The following repositories were selected from the combined data set: crazyandcoder–citypicker, nanchen2251–CompressHelper, brianway–java-learning, android–user-interface-samples,
leolin310148–ShortcutBadger, yuliskov–SmartTubeNext, mzule–ActivityRouter, and florent37–DiagonalLayout.

We categorized the chained method according to it's chained length, and further classify its usage into "Accessor", "Builder", "Assertion", and "Other". Some of the common method calls of these types are:

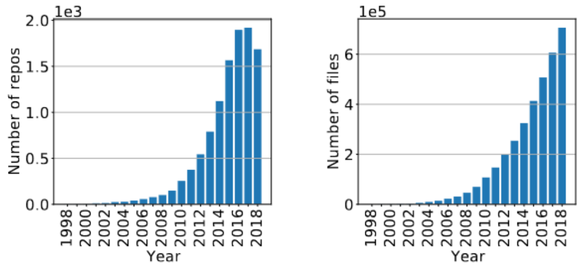
- Accessor: square.getArea()
- Builder: square.setWidth()
- Assertion: square.hasColor()

Method name to method Type mapping is further described in Table 1. Because manual inspection is slow, tedious, and susceptible to human error (inconsistency in classifications, accidental miscounts, etc), we implemented a means to automate the process. We hoped evaluating the (dis)agreement between the manual and automated classification would help illuminate usage.

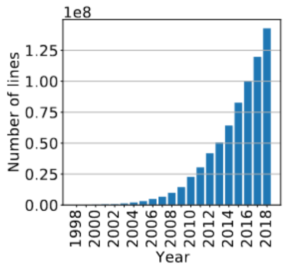
First our classifier generates a mapping from methods to type. The program walks through each file in the repository and finds every method chain with JavaParser. Then it adds the first and last methods to a set of methods. Every method in the set is then given an automatic categorization based on whether it contains words like "get" or "assert" for accessors and assertions or builder methods like "build" or "toString" for builders. Otherwise methods are classified as "other". See

Accessor	Builder	Assertion	Others
get*	build*	assert*	<undefined>
for	to*	is*	–
set*	make*	–	–
with*	create*	–	–
pop	split	–	–
push	–	–	–
peek	–	–	–
valueof	–	–	–

Table 1. Method Name to Method Type Mapping



(a) Number of repositories **(b) Number of files**



(c) Number of lines

Figure 1. Repositories, Files and Lines of Code used by Nakamaru et al. [8]

Table 1 for a tabular mapping summary. The program then returns this mapping as a csv file.

Then the researcher may observe and edit the csv file mapping to correct any false classifications. Once satisfied with the accuracy of the mappings, the next program uses the mapping to classify method chains as a whole. The second program walks through each file again, but this time follows a simple decision tree. If the last method is a builder, then the chain as a whole is considered a builder chain. Otherwise it uses the mapping of the first method in the chain. Here it also calculates the length of the chain, and aggregates the results by chain by size (S, L, XL) and type (Accessor, Builder, Assertion, and Other).

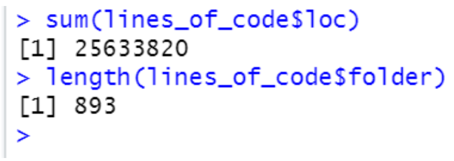

```
> sum(lines_of_code$loc)
[1] 25633820
> length(lines_of_code$folder)
[1] 893
>
```

Figure 2. Repositories and LOC in our data set

3.4 Counting Lines of Code

To calculate the lines of code per repository (Figure 2), we used the CLOC library [3] available through NPM. CLOC enabled us to count many files in a directory, and has support for Java files. A limitation of CLOC is an inability to group results by directory, so the program had to be run once per repository. To facilitate this, we created a simple Python script which runs CLOC for each repository's directory and outputs a csv file containing the repository's name and the lines of code in that repository. Our data set contains 25,633,820 lines of code within 649,276 files from 903 repositories, roughly ¼ of the lines of code in our data set as compared to the data set used by Nakamaru et al.

3.5 Statistical Analysis

With the data collected with JavaParser [5] we applied the f_n (Figure 3) and r (Figure 4) metrics from Nakamaru et al. [8]. These represent the frequency per method chains of size n and the ratio of chains greater than size 1 over all method invocations. These metrics were applied to our combined data set, and we replicated the test and non-test separated f_n (Figure 6) and r (Figure 7) analysis with our separated data set, as done by Namaru et al. Finally, we replicated the categorized and size binned (Figure 5) metrics given by Nakamaru et al. for comparison of trends in method chain usage.

4 Results

4.1 Comparing f_n with Findings by Nakamaru et al.

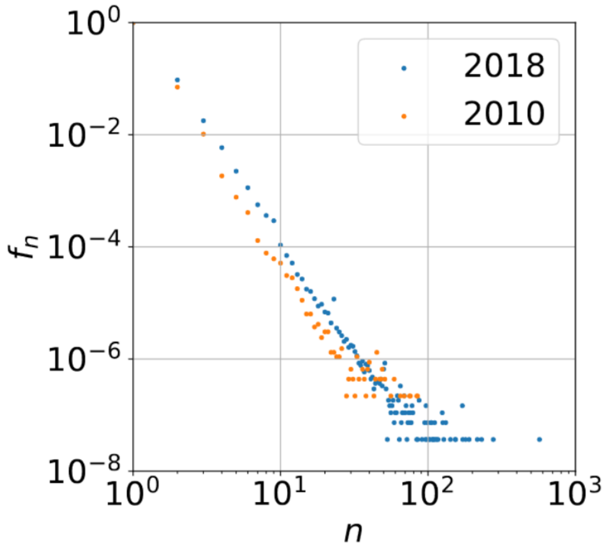
From the comparison of f_n in Figure 3, we see the relative number of short chains is very similar in 2019-2020 when compared to what was observed by Nakamaru et al. [8]

4.2 Comparing r with Findings by Nakamaru et al.

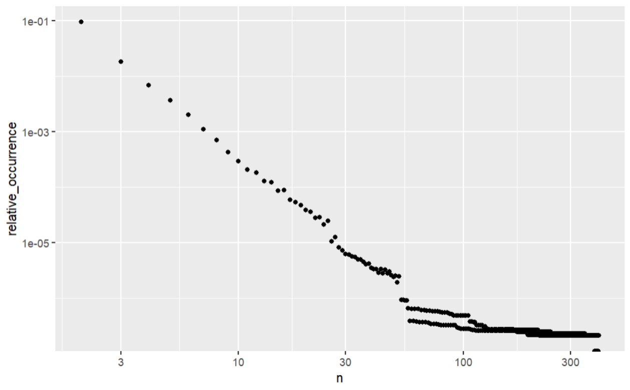
The mean of r (ratio of method chains with length >= 2 : method chains length 1) has dropped to 0.108 in 2019-2020 from .23 in 2018. Figure 4 illustrates the trends from 2010 through 2018, and provides a more granular view of 2019-2020's ratio of method chains.

4.3 Comparing Categorical Usage and Size of Chains with Findings by Nakamaru et al.

Both our automated and manual classification of chains show (Figure 5) accessors to be the most popular category of chain in the short chain category (chains with length < 8). The

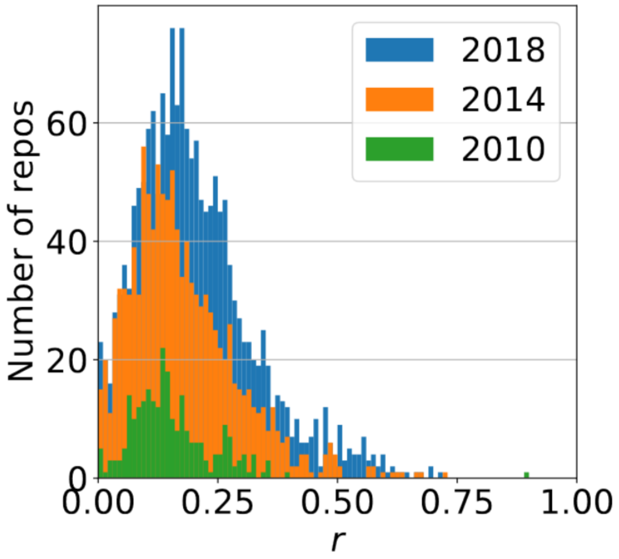


(a) f_n Observed by Nakamaru et al. [8]

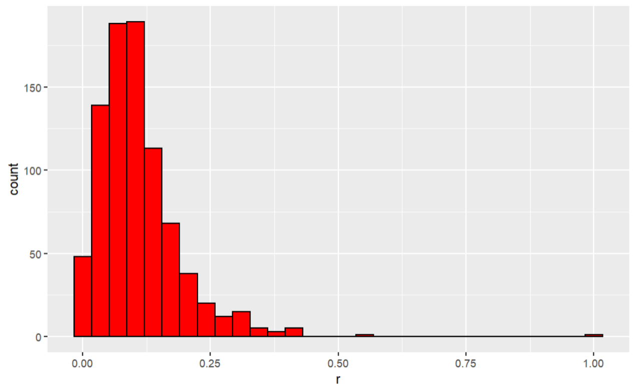


(b) Our Observed f_n (2019-2020)

Figure 3. Comparison of f_n (Relative Occurrence of Method Chains)



(a) r Distribution Observed by Nakamaru et al. [8]



(b) Our Observed r Distribution (2019-2020)

Figure 4. Comparison of r (Ratio of Method Chain Lengths) Distributions

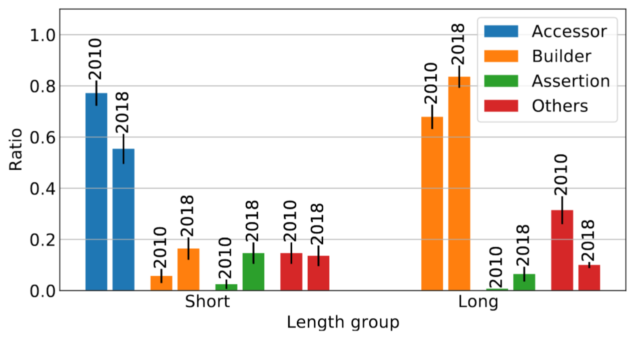
manual classification has some commonality with the long size category as observed by Nakamaru et al., with builders being the most popular category. The absence of other categories within our manual classification may be explained by sampling, however, it looks a bit questionable in comparison with the results of the automatic classification. Given the automatic classification marks the use of assertion and others, the manual categorization should be done again by another party and mediated by a third party to validate the result.

4.4 Usage of Longest Chains

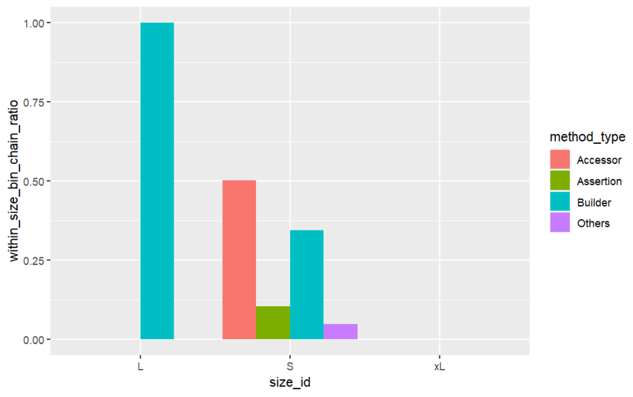
We also wanted to see how the longest chains were being used, and by whom. It turns out tech giants AWS, Facebook, and Google all use huge method chains. The longest chain we observed by AWS is a builder, for their code commit client's protocol factory. Facebook also uses the very long chain we observed with a builder. Finally, we see Google too is using their longest chain observed by us for a builder.

5 Threats to Validity

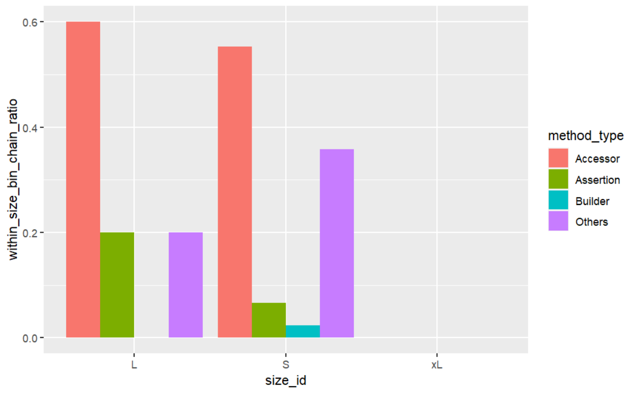
Whether an idiom is a best practice is fundamentally subjective, so our assumptions of best practice indicators may be false. An increase in method chain usage may not indicate its adoption as a best practice, since it may be used in conjunction with other code smells. Stars may be an inaccurate measure of popularity for Java repositories if stars are used as a bookmark instead of a mark of recognition.



(a) Classification by Nakamaru et al. [8]



(b) Our Manual Classification

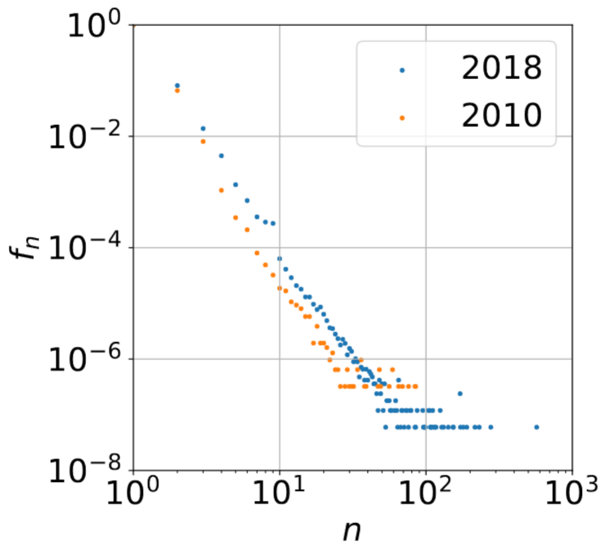


(c) Our Automatic Classification

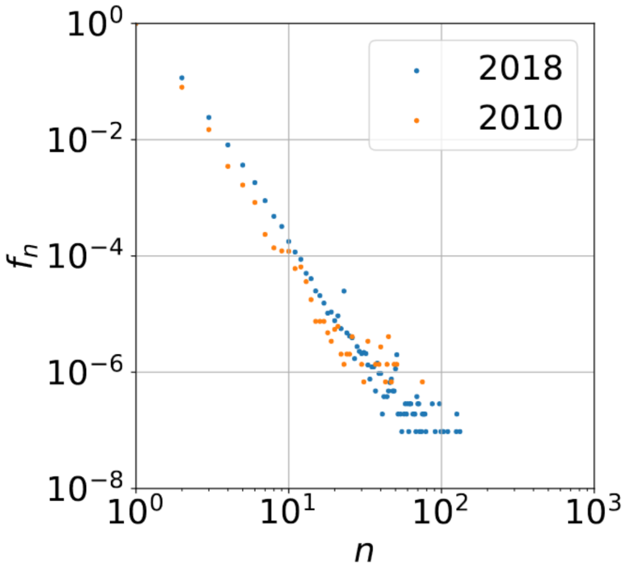
Figure 5. Comparison of Manual and Automatic Classification of Method Chain Types by Bin Size

JavaParser failed to parse some files using unsupported versions of Java, as such our data set does not include code using those versions of Java[1]. It follows that some of the latest style trends may not be able to be captured in an automated fashion.

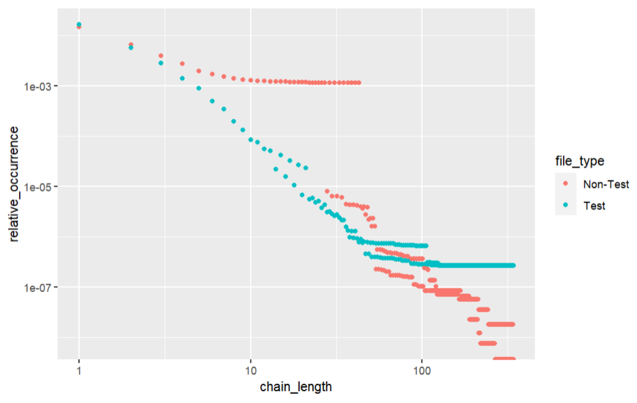
[1]https://github.com/javaparser/javaparser



(a) f_n of Non-Test Code by Nakamaru et al. [8]

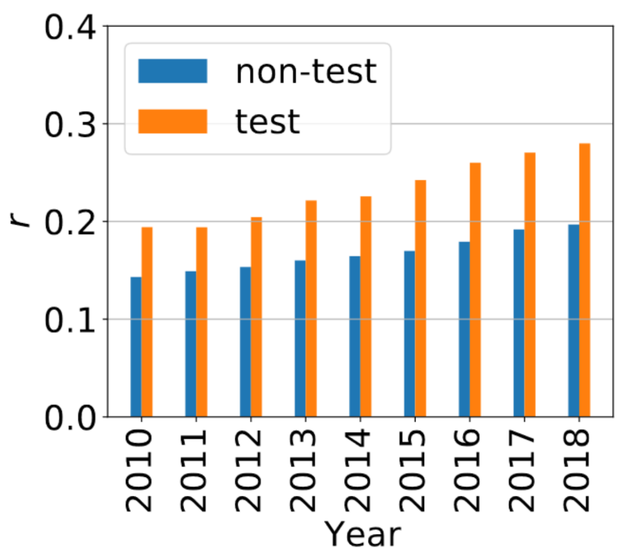


(b) f_n of Test Code by Nakamaru et al. [8]

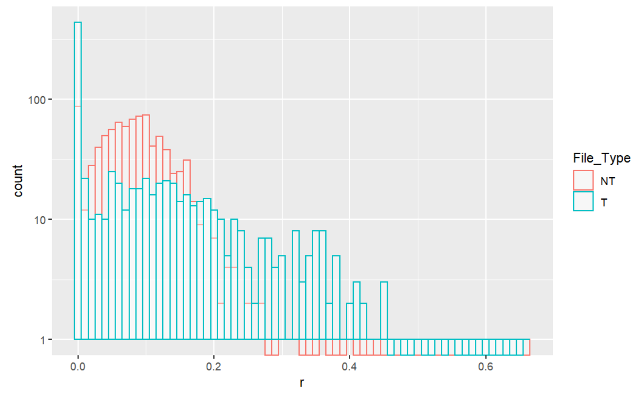


(c) f_n of Test Code vs Non-Test Code (2019-2020)

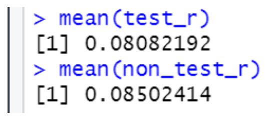
Figure 6. Comparison of Test and Non-Test Code f_n



(a) r of Test vs Non-Test Code by Nakamaru et al. [8]



(b) r of Test Code vs Non-Test Code (2019-2020)

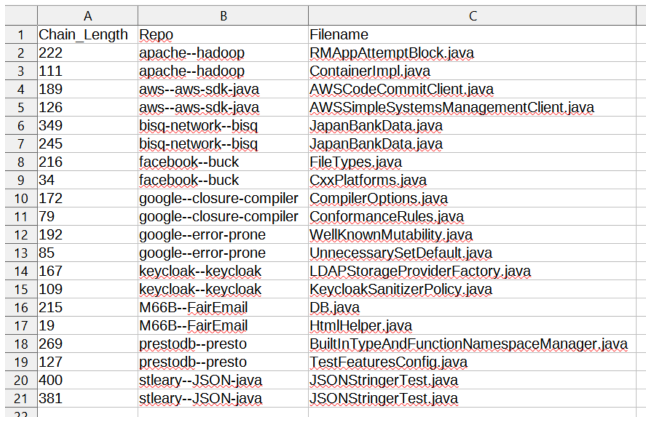


```
> mean(test_r)
[1] 0.08082192
> mean(non_test_r)
[1] 0.08502414
```

(c) mean r of Test Code vs Non-Test Code (2019-2020)

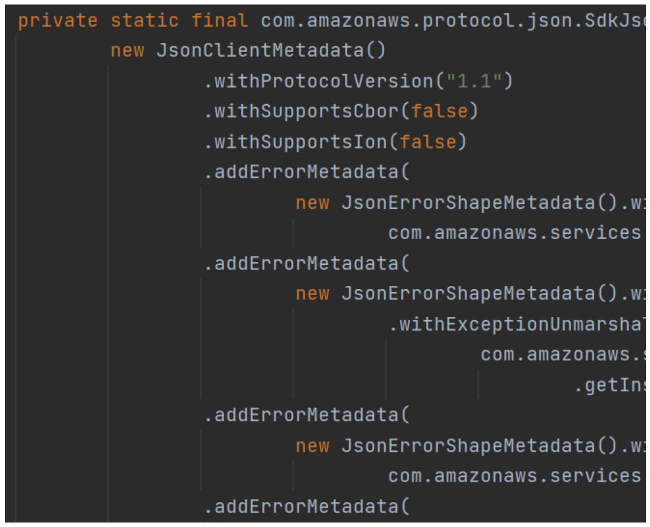
Figure 7. Comparison of Test and Non-Test Code r

Our manual classification of files was conducted by only one member, once per file, so we do not have multiple manual inspection results to compare and contrast. Furthermore, the automatic classifier was run without manual verification of the intermediate method-classification mapping output. While our manual and automated classification show reasonable agreement between the accessor and assertion observations in the "Small" method chain category (Figure 5), the same does not hold for the "Large" category. Therefore,



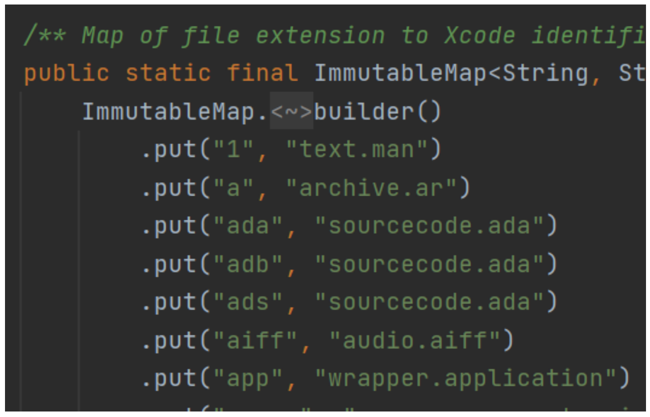
	A	B	C
1	Chain_Length	Repo	Filename
2	222	apache--hadoop	RMAppAttemptBlock.java
3	111	apache--hadoop	ContainerImpl.java
4	189	aws--aws-sdk-java	AWSCodeCommitClient.java
5	126	aws--aws-sdk-java	AWSSimpleSystemsManagementClient.java
6	349	bisq-network--bisq	JapanBankData.java
7	245	bisq-network--bisq	JapanBankData.java
8	216	facebook--buck	FileTypes.java
9	34	facebook--buck	CxxPlatforms.java
10	172	google--closure-compiler	CompilerOptions.java
11	79	google--closure-compiler	ConformanceRules.java
12	192	google--error-prone	WellKnownMutability.java
13	85	google--error-prone	UnnecessarySetDefault.java
14	167	keycloak--keycloak	LDAPStorageProviderFactory.java
15	109	keycloak--keycloak	KeycloakSanitizerPolicy.java
16	215	M66B--FairEmail	DB.java
17	19	M66B--FairEmail	HtmlHelper.java
18	269	prestodb--presto	BuiltInTypeAndFunctionNamespaceManager.java
19	127	prestodb--presto	TestFeaturesConfig.java
20	400	stleary--JSON-java	JSONStringerTest.java
21	381	stleary--JSON-java	JSONStringerTest.java

Figure 8. Top 2 Longest chains in Top 10 Longest Chain Repositories



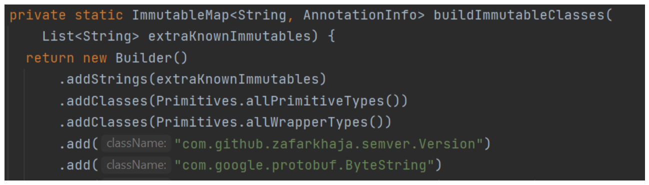
```
private static final com.amazonaws.protocol.json.SdkJs
        new JsonClientMetadata()
                .withProtocolVersion("1.1")
                .withSupportsCbor(false)
                .withSupportsIon(false)
                .addErrorMetadata(
                        new JsonErrorShapeMetadata().wi
                                com.amazonaws.services
                .addErrorMetadata(
                        new JsonErrorShapeMetadata().wi
                                .withExceptionUnmarsha
                                        com.amazonaws.
                                                .getIns
                .addErrorMetadata(
                        new JsonErrorShapeMetadata().wi
                                com.amazonaws.services
                .addErrorMetadata(
```

Figure 9. AWS Longest Chain Snippet (Builder)



```
/** Map of file extension to Xcode identifi
public static final ImmutableMap<String, St
    ImmutableMap.<~>builder()
        .put("1", "text.man")
        .put("a", "archive.ar")
        .put("ada", "sourcecode.ada")
        .put("adb", "sourcecode.ada")
        .put("ads", "sourcecode.ada")
        .put("aiff", "audio.aiff")
        .put("app", "wrapper.application")
```

Figure 10. Facebook Longest Chain Snippet (Builder)



```
private static ImmutableMap<String, AnnotationInfo> buildImmutableClasses(
    List<String> extraKnownImmutables) {
  return new Builder()
      .addStrings(extraKnownImmutables)
      .addClasses(Primitives.allPrimitiveTypes())
      .addClasses(Primitives.allWrapperTypes())
      .add( className: "com.github.zafarkhaja.semver.Version")
      .add( className: "com.google.protobuf.ByteString")
```

Figure 11. Google Longest Chain Snippet (Builder)

either our automated, manual or both classifications may be erroneous.

Additionally, there is some ambiguity with regard to chain classification. It makes little sense to deny a chain which does not begin with a builder or end with a builders from being classified as a builder if the chain has an internal method which is a builder. We followed the strategy of Nakamru et al. for classification, categorizing method chains according to one method invocation per chain. However, we feel that may not capture fully how method types are used within the size categories. Classifying chains within their respective size bins would likely be more meaningful if the bins were described as a ratio of types used, i.e. "S" contains 40% accessor, 20% assertion, 5% builder, 35% others.

Zou et al. investigated how coding style impacts pull request (PR) integration, concluding that "accepted PRs were indeed more similar to the project than rejected ones," [10] implying that the perceived popularity of code styles is propagated by repository maintainer preferences. This would jeopardize our assumption that code usage in popular repositories implies an acceptance as a best practice.

6 Conclusion

6.1 RQ-1: Is method chaining still popular today?

The presence of method chaining within these highly starred repositories is undeniable. Many method invocations within these repositories are shown (Figure 3) to be within method chains.

6.2 RQ-2: Is method chaining primarily used in production code or testing code?

As seen in Figure 7, the mean ratio of chain usage is actually greater in non-test code than test code. Therefor non-test code has more method chaining per method invocation than test code. However the chains in test code are longer on average, so the answer is unclear.

6.3 RQ-3: What are typical lengths for method chains?

The relative occurrence of method chains illustrates (Figure 3) the exponentially decreasing popularity of shorter method chains, with chains of length 2 being the most popular. In greater detail, we see in Figure 6 that non-test code files make liberal use of chains with length < 100, while chains found in test code are more commonly shorter. Also, the longest chains are most commonly found in test code.

6.4 RQ-4: Has the usage of method chains as classified by Nakamaru et al. changed?

While we did see significant change as seen in 3 we also had many threats to validity noted on these observations. Therefore, this question may be better answered by future work.

6.5 RQ-5: When should method chaining be used?

This question emerged while researching trends in method chaining usage in repositories through 2019-2020. As such, our methodology was unable to fully answer it. However we observed many extra long chains used for builders.

Looking to the work of AWS, Facebook, and Google, we see that chaining is helpful for builders. When constructing objects that bind their variables at construction, it is preferable to do so with a builder design pattern. Chaining together methods to manage the state of the builder was done by the aforementioned tech giants. We then infer that this is a good practice. However, during our manual inspection, we observed a chain of length 400 which repeatedly made the same method invocations with the same parameters. In this case, we feel it would be preferable to use a for-loop or alternative methods as it would be more maintainable and readable. For recommendations on method chain usage for short and long chains.

Fluent interfaces [6] leverage method chaining to mimic domain specific language. Fowler uses method chaining to demonstrate fluent interfaces. Method chaining is ideal for applying the natural flow of fluent interfaces, since methods are fundamentally linked together and thus sequential and orderly. Note the distinction between fluent interfaces and method chaining: "many people seem to equate fluent interfaces with Method Chaining" [6].

7 Future Work

As noted in the threats to validity, method chains can be tricky to classify because there is no definition of a method chain classification hierarchy. Two potential solutions to this problem are:

1. Define a hierarchy of method chain classifications, where the presence of one method type at a location in the chain determines the classification of the whole chain.
2. Examine chains as a composition of various method classifications, showing the proportional usage of method types in lengths of size n (for all n > 1).

Furthermore, as discussed in the lecture, manual inspection of method chains could be conducted by at least two people, with a third person acting as a moderator. The moderator should question agreement and disagreement of method chaining classifications as they see fit. This would build confidence in the validity of the manual inspection, which in turn would help build confidence in the validity of the automated classification by comparison with the manual classification.

Our automated classifier can be built upon by integrating some machine learning (ML) and/or natural language processing (NLP) within. ML and/or NLP would help improve the reliability of our atomic method classification without human intervention, furthermore, the scope of the final link in a method chain could be analyzed with NLP and ML to determine the overall classification of the chain.

Acknowledgments

To Neil, for getting us thinking in this realm.

References

[1] Danny Dig Alex Gyori, Lyle Franklin and Jan Lahoda. 2013. Crossing the Gap from Imperative to Functional Programming through Refactoring. (2013).

[2] Christian Bird Chris Parnin and Emerson Murphy-Hill. 2011. Java Generics Adoption: How New Features are Introduced, Championed, or Ignored. *Proceedings of the International Working Conference on Mining Software Repositories* (2011), 3871–3903.

[3] A. Danial. 2006. *CLOC – Count Lines of Code.* Retrieved November 19, 2021 from http://cloc.sourceforge.net

[4] Nikolaos Tsantalis Davood Mazinanian, Ameya Ketkar and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proc. ACM Program. Lang* (2017), 85:1–85:31.

[5] D.V. Bruggen et al. 2019. *JavaParser - Home.* https://doi.org/10.5281/zenodo.2667379

[6] Martin Fowler. 2005. *bliki: FluentInterface.* Retrieved December 10, 2021 from https://martinfowler.com/bliki/FluentInterface.html

[7] Matthias Hauswirth Luis Mastrangelo and Nathaniel Nystrom. 2019. 2019. Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs. *Proc. ACM Program. Lang* (2019).

[8] Tomoki Nakamaru, Tomomasa Matsunaga, Tetsuro Yamazaki, Soramichi Akiyama, and Shigeru Chiba. 2020. An Empirical Study of Method Chaining in Java. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 93–102. https://doi.org/10.1145/3379597.3387441

[9] Hoan Anh Nguyen Robert Dyer, Hridesh Rajan and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. *Proceedings of the 36th International Conference on Software Engineering* (2014), 779–790.

[10] Weiqin Zou, Jifeng Xuan, Xiaoyuan Xie, Zhenyu Chen, and Baowen Xu. 2019. How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects. *Empirical Software Engineering* 24, 6 (June 2019), 3871–3903. https://doi.org/10.1007/s10664-019-09720-x

A Replication Package

- Classifier and parser repository
- Data set with tests separated
- Data set without tests separated

B Team Contributions

The GitLab repository for this project outlines contributions and distributions of tasks through issues and commits. This captures maybe half of the work done.

B.1 Alexander Aldridge

Alexander helped focus and organize team meetings, as well as ensure the team stayed on track to meet their meeting goals. Otherwise, he focused on script writing, paper editing, and the production of the video report.

B.2 Tianna McDowell

Tianna created a python script that collected the top 1000 starred repositories from the GitHub API and then separating the Java files into separate directories. She updated this script to separate the files into test and non-test classifications. She also helped edit the paper.

B.3 Gary Fan

Gary initiated the GitHub repository url fetcher, set up the JavaParser environment, parsed the repositories, manually inspected 14204 methods.

B.4 Glen McManus

Glen helped coordinate tasks through GitLab issues, worked on the parsing, automated classification, statistical analysis, research into related works, and paper writing.