On the Evolution of Unused Dependencies in Java Project Releases: An Empirical Study

Nabhan Suwanachote*, Yagut Shakizada*, Yutaro Kashiwa*, Bin Lin†, Hajimu Iida*
*Nara Institute of Science and Technology, Japan
†Hangzhou Dianzi University, China

***Abstract*—Modern software development heavily relies on third-party dependencies to reduce workload and improve developer productivity. Given the vast number of dependencies available and the ease of including them in projects, some introduced dependencies are never used, leading to bloated software, longer build times, and increased network bandwidth usage. While several previous studies have examined the prevalence of unused dependencies and their impact on security, it remains unclear how these dependencies are introduced and removed in software projects. This study aims to answer this question through an empirical study involving 3,020 release versions of 417 Java projects. Our analysis shows that unused packages are common in most projects (52% of projects), but few releases (9%) introduce new unused dependencies. Among those resolved unused dependencies, 59% of them were removed and 41% were used in later versions. Our findings highlight that not all unused dependencies should be removed in practice.**

***Index Terms*—Dependencies, Packages, Empirical Study**

I. Introduction

In modern software projects, developers often adopt third-party dependencies to accelerate development and reduce costs [1]–[3]. A recent study reports that 96% of commercial codebases reuse open-source software and contain an average of 526 open-source components [4]. The emergence of build systems such as Maven and npm eases the adoption of dependencies, as developers can include or change dependencies by modifying only a few lines of build files. The number of third-party libraries available is also vast. For example, 658,078 libraries with 14,459,139 releases are provided on Maven Central Repository [5].

Given the easy access to third-party dependencies, many dependencies are installed but remain unused in repositories [6]. Recent studies have reported that these unused dependencies not only occupy storage space and consume network bandwidth in local or CI environments, but also pose security vulnerabilities [7] and account for up to 55.9% of CI build time [8]. Additionally, we observe many cases where developers attempt to remove unused dependencies due to bulky JAR files [9] [10] and license incompatibilities [11]. For example, developers in the `snowflakedb/snowflake-jdbc` project discussed the fact that unused packages excessively increased the size, violating their distribution limitations [10].

Existing studies on unused dependencies mainly focus on the prevalence of unused libraries and their impact on build times and security. Some studies have investigated which commits introduce or remove unused dependencies [8], [12], but the granularity of the analysis remains coarse. It is still unclear when unused dependencies are introduced and how long they persist. Understanding how unused dependencies evolve helps researchers and practitioners propose strategies to manage these dependencies.

This study aims to reveal the lifecycle of unused dependencies. We collected and analyzed 15,738 dependencies contained in 3,020 release versions of 417 projects. As a result, we identified 3,074 unused dependencies and found that 52% of the projects contained unused dependencies, with an average of 14.2 and a median of 5 unused dependencies per project, indicating that half of the studied projects are not maintained adequately. Furthermore, those unused dependencies which were resolved in the end persisted in the software for a median of two versions, with 59% being removed and 41% being used in later versions. This result highlights the caution needed when designing tools to clean the "bloated software".

Replication Package: To facilitate replication studies and future extensions, the data used in our work is publicly available in the replication package.[1]

II. Research Questions

The goal of this study is to reveal how unused dependencies evolve. More specifically, we formulate the following research questions (RQs).

RQ1: How are unused dependencies introduced in software project releases? Several studies [6], [13] have investigated unused dependencies in software repositories. However, these studies normally consider all the dependencies declared in `pom.xml` files and it is unclear how many unused dependencies are included in the software releases. Moreover, it also remains unknown when these unused dependencies are introduced. This RQ aims to address these doubts by inspecting the prevalence and the introduction time of the unused dependencies.

RQ2: How are unused dependencies handled by developers? Previous studies [6], [12], [14] analyzed how many unused dependencies are introduced and their impact on security. However, to the best of our knowledge, no studies have examined how long unused dependencies persist and whether they are handled by developers. Studying how they are resolved in practice could provide developers with valuable insights on how to deal with unused dependencies in codebases.

[1]https://github.com/nabhan-au/java-dependency-analyzer

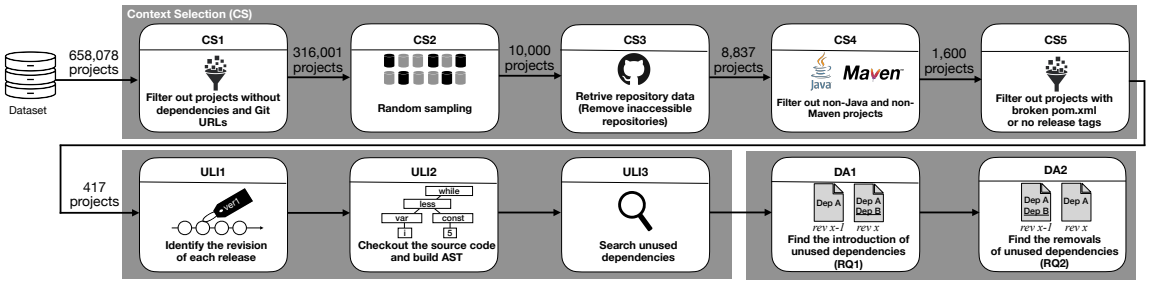

Fig. 1: Overview of this study

III. Study Design

A. Context Selection

To retrieve the dependency usage information, we adopt Jaime *et al.*'s dataset (version "2024-08-30 & metrics") [5] [15], which provides dependency graphs between libraries that use Maven. The dataset contains 658,078 libraries and their dependencies, which are collected from Maven Central Repositories using Globlin Miner [16].

We performed two filtering processes to eliminate the dependencies that we cannot analyze. First, we filtered out the projects that do not use third-party dependencies (*i.e.,* no dependencies defined in `pom.xml`). Second, we excluded the projects whose `pom.xml` do not contain Git URLs as our analysis requires visiting different versions of source code. These filters removed 342,077 projects and 316,001 remained.

We then use GitHub API to double check whether the extracted projects primarily use Java and Maven. Given the current GitHub API rate limit (5,000 API calls per hour)[2], it is impractical to check all 316,001 projects. Therefore, we randomly sampled 10,000 projects and for each project, we examined whether the main programming language is Java and the repository contains `pom.xml` (*i.e.,* using Maven).[3] 1,163 projects were excluded due to their inaccessibility, 5,953 projects for not using Java as their primary programming language, and 1,284 projects for not utilizing Maven. This process led to 1,600 projects that fit into our criteria.

Finally, we excluded projects whose `pom.xml` files are broken, such as those with unresolved dependencies or invalid syntax (265 projects) and projects that do not have all the release tags corresponding to the releases in the dataset (918 projects), so that for the studied projects we can check out the specific revisions of source code corresponding to release versions. Specifically, we identified release tags of their GitHub repositories that match the version or include the version (*e.g.,* v1.1) specified in Jaime *et al.*'s dataset. If there are multiple candidates, we use the one that has the shortest tag name. This approach identified 68.9 % of versions (*i.e.,* 16,439) out of 23,587 versions in our dataset. In the end, we obtained 417 projects that have all the release tags. Table I presents the statistics of the studied projects.

[2]https://docs.github.com/en/rest/using-the-rest-api/rate-limits-for-the-rest-api

[3]Note that Maven Central Repository allows using of other build tools such as Gradle and so their main language is not only Java.

TABLE I: Statistics of the studied repositories

	Min	Median	Mean	Max
Versions	1.0	4.0	7.2	157
Stars	0	2.0	131.6	10,548
Commits	1.0	61	1204.6	63,767
Average Dependencies[4]	0.3	3.0	4.5	51.8

B. Unused Dependency Identification

To identify unused dependencies in the 417 projects, we first retrieve their source code of each version through the Git link declared in the `pom.xml` files. We then check out the revision of the source code that is linked to the release tag. Next, we built the abstract syntax tree (AST) of the source code using JavaParser[5] to identify the libraries that are imported (including wildcard imports) and used in the source code. Finally, we identified used/unused dependencies declared in the `pom.xml` by checking if the path and class names of imports (*e.g.,*`import a.b.c;`) are the same as those of dependencies, *i.e.,* Jar files (*e.g.,*`a/b/c.class`).

Note that we also included older versions not present in Maven Central Repository (*i.e.,* Jaime *et al.*'s dataset), if they could be identified through GitHub release tags. Also, we only analyzed the dependencies for production and excluded dependencies for other processes such as testing, as one of our motivations is to understand how many unused dependencies will end up in the distributions such as JAR files. Moreover, we only studied the direct dependencies (*i.e.,* dependencies declared in the POM files) instead of transitive dependencies (*i.e.,* other dependencies contained in the dependencies we studied) and inherited dependencies (*i.e.,* dependencies declared in the parent POM files), which are both indirectly included in the product, as we focus on how developers introduce and handle unused dependencies.

It is worth mentioning that we detect not only (i) the cases where the dependencies declared in the `pom.xml` files are not used in the source code but also (ii) the cases where the declared dependencies are imported in the source code but are not used like in this case [17]. These cases cannot be detected by the official Apache Maven Dependency Plugin.[6]

[4]Note that the 'Average Dependencies' represents the average number of dependencies across all releases.

[5]https://javaparser.org

[6]https://maven.apache.org/plugins/maven-dependency-plugin/

C. Data Analysis

1) RQ1: To answer RQ1, we first measure *the number of unused dependencies* included in each version of the software project. However, the number of used dependencies varies significantly across projects and versions. To mitigate the bias introduced by this fact, we also calculate *the ratio of unused dependencies* to all the dependencies declared in `pom.xml`.

Additionally, we identify the number of unused dependencies newly introduced in each version release. To identify these introductions, we compare the list of unused dependencies in every two consecutive versions of a project. Specifically, we count instances where an unused dependency is absent in the previous version but appears in the current version. Furthermore, we classified these unused dependencies into two types based on how they are introduced:

- *No longer used:* The unused dependencies were used in the previous version but stopped being used in the subsequent versions.
- *Introduced but not used:* The unused dependencies were declared in `pom.xml` but never used in the next version.

2) RQ2: We investigate how the unused dependencies are handled after they are introduced. We categorize the unused dependencies into the following three types:

- *Remaining unused:* The dependency is still unused as of the latest revision.
- *Used later:* The dependency is previously unused but utilized in subsequent versions.
- *Removed later:* The dependency was initially unused and removed in subsequent versions.

For the later used and deleted cases (*i.e.,* the second and third categories), we measure the time and version differences until the dependency is either utilized or removed.

IV. Results

A. RQ1 (Introduction of unused dependencies)

We identified a total of 3,074 unused dependencies in 3,020 release versions of 417 projects. The average number of unused dependencies per version is 1.0, with the highest number being 13. The average ratio of unused dependencies is 16.4% of the total dependencies. However, the median number and ratio of unused dependencies are both 0.0, indicating that most of the releases do not contain unused dependencies while non-negligible numbers of unused dependencies are included in software projects.

We also investigated in which form these unused dependencies exist in the projects and found that only 21 out of 3,074 unused dependencies are imported but not used. This shows that most projects remove unnecessary import statements in the source code for releases, leaving only the declarations of unused dependencies in the `pom.xml` files.

Next, we examined when dependencies become unused. Table II shows the number of unused dependencies introduced in each release. We identified 538 introductions of unused dependencies, which are distributed across 9.3% of the releases. Of these introductions, 94.4% (*i.e.,* 508) are due to adding dependencies without using them in the source code, and 5.6% (*i.e.,* 30) are due to the removal of source code while the `pom.xml` retains the dependencies. The median size of the relevant JAR files that are unnecessarily downloaded due to declaring an unused dependency is 1.9 MB.

TABLE II: Introductions of unused dependencies

		Distribution by version			
	Total	Min	Median	Mean	Max
All Introductions	538	0	0	0.18	13
- No longer used	30	0	0	0.01	2
- Introduced but not used	508	0	0	0.17	13

We also investigated the extent to which projects experienced releasing products with unused dependencies. We found that 52.0% of projects released at least one version with unused packages. Interestingly, 79.9% of unused dependencies are introduced within the first year of their development.

Takeaway: *While in most cases software releases do not contain unused dependencies, it is not uncommon that some still escape the attention of developers. Some automatic tools could be proposed to detect such cases.*

RQ1: 52.0 % of projects experienced releasing at least one version of their products that have unused dependencies but most of the versions do not have unused dependencies.

B. RQ2 (Persistence of unused dependencies)

Out of 538 introductions of unused dependencies, we observed 410 dependencies (*i.e.,* 76.2%) remaining unused as of November 2024. Of the resolved unused dependencies (*i.e.,* the remaining 128 unused dependencies), 75 were removed and 53 were later used. This means that 59% of unused dependencies are removed instead of being used in the source code.

We then investigated the number of days and versions until the introduced unused dependencies become used or deleted. Figure 2 shows the distributions of the days and versions. As a result, a median of 210.6 days is taken until the dependencies are resolved (*i.e.,* used or removed). Specifically, 208.4 days are spent until removal and 256.0 days until being used. The Mann–Whitney U test [18] (α=0.05), a non-parametric test, detected no statistically significant difference between both types of resolved dependencies in duration (p-value: 0.84).

As for the versions, a median of two versions is needed until the dependencies are either used or removed. Applying the Mann–Whitney U test, we observed no statistically significant difference between both types of resolved dependencies in versions (p-value: 0.93).

Takeaway: *developers should remain cautious when apply tools to automatically remove unused packages as around 40% of them will be needed later.*

RQ2: Out of resolved unused packages, 59% of them are removed instead of being used in the source code. They remain for a median of two versions.

(a) Days of unused duration

(b) Versions where each unused dependency exists

Fig. 2: Number of days and versions until unused dependencies are resolved

V. Threats to Validity

Threats to internal validity concern the factors we did not consider that might impact the results. We needed to identify the revision (*i.e.,* SHA) of each version by associating their release tags on GitHub with the versions recorded in the dataset. This process is done through the automatic way, and we did not perform extra checks on whether these versions are exactly identical. To prevent inconsistencies between the acutal source code and the dataset, we performed our analysis based on `pom.xml` files on GitHub.

Threats to construct validity concern the relation between theory and observation. We analyzed production code (*i.e.,* not test code) to identify unused dependencies declared in `pom.xml` files. However, we did not consider whether the source code is indeed used in the end product (*e.g.,* several dependencies might be used only for prototypes or experimental development). Thus, the number of unused dependencies for end products might be higher than what we reported.

Threats to external validity concern the generalizability of our findings. This study uses 417 projects that are randomly sampled from Maven Central Repository. However, it is unclear whether the result would change when it comes to projects which are not published on Maven Central Repository, such as proprietary and closed-source software.

VI. Related work

This section introduces related work that focuses on unused dependencies from various perspectives across different software ecosystems.

Weeraddana *et al.* [8] analyzed Continuous Integration (CI) waste caused by dependencies. Their results reveal that unused dependencies waste is 55.9% of dependency-update-associated CI build time. Moreover, projects which are the most affected by CI waste spend a considerable portion, 85.5% of their free build minutes on unused dependency commits such as updating dependency specifications (*i.e.,* `package.json`).

Soto-Valero *et al.* [6] investigated bloated dependencies (*i.e.,* unused dependencies) in the Maven ecosystem. They analyzed the bytecode of 9,639 client artifacts and their 723,444 dependencies resolved by Maven. Their results show that 75.1% of dependencies are bloated (2.7% are direct, 15.4% are inherited from parent POMs, and 57% are transitive dependencies).

Cao *et al.* [12] studied dependency smells, missing dependencies, and bloated dependencies in Python projects. They found that 75% of bloated dependencies are introduced into the projects when they are added to configuration files. Besides, bloated dependencies are often never imported in source code, and 85.1% of bloated dependencies are removed by just removing dependency declarations from configuration files.

Harrand *et al.* [13] conducted a systematic large-scale analysis of 2,169,273 Maven client - API relations between 5,225 library versions of 94 most popular libraries to study client-API usages. Their findings indicate that 41.1% (892,167) of the studied dependencies declared in the clients' pom.xml files become unused as they are not translated into API usage at the bytecode level.

Jafari *et al.* [19] conducted an empirical study on dependency smells using a dataset of 1,146 active JavaScript projects. In their study, they found that unused dependencies result in dependency smells and unnecessarily bloat the `package.json` file.

VII. Conclusions and Future Work

In this study, we analyzed 3,020 versions of 417 Java projects to understand the evolution of unused dependencies. More specifically, we investigated when these unused dependencies are introduced and how long they persist. Our results show that unused dependencies are common in software projects. While most of the release versions do not contain unused dependencies, 52.0% of projects experienced unused dependencies at some point. As for the resolved unused packages, they do not remain for a long time, and 59% of them are simply removed instead of being used later.

Our future work include conducting a finer-grained analysis at the commit level to reveal the role unused dependencies play during software development. Moreover, we would like to propose approaches to predict the unused dependencies.

Acknowledgment

We gratefully acknowledge the financial support of JSPS for the KAKENHI grants (JP21H03416, JP24K02921), the Bilateral Program grant (JPJSBP120239929), as well as JST for the PRESTO grant (JPMJPR22P3), the ASPIRE grant (JPMJAP2415), and the AIP Accelerated Program (JPMJCR25U7).

REFERENCES

[1] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Communications of the ACM*, vol. 39, no. 10, pp. 104–116, 1996.

[2] E. R. Murphy-Hill, C. Jaspan, C. Sadowski, D. C. Shepherd, M. Phillips, C. Winter, A. Knight, E. K. Smith, and M. Jorde, "What predicts software developers' productivity?" *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 3, pp. 582–594, 2021.

[3] S. Wagner and E. R. Murphy-Hill, *Factors That Influence Productivity: A Checklist*. Apress, 2019, pp. 69–84.

[4] Blackduck, "2024 open source security and risk analysis report," https://www.blackduck.com/blog/open-source-trends-ossra-report.html, February 27th, 2024, accessed: December 1st, 2024.

[5] D. Jaime, J. El Haddad, and P. Poizat, "Navigating and exploring software dependency graphs using goblin," in *Proceedings of the International Conference on Mining Software Repositories (MSR 2025)*, 2025.

[6] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *Empirical Software Engineering (EMSE)*, vol. 26, no. 3, p. 45, 2021.

[7] A. Gkortzis, D. Feitosa, and D. Spinellis, "A double-edged sword? software reuse and potential security vulnerabilities," in *Proceedings of the 18th International Conference on Software and Systems Reuse (ICSR 2019)*, vol. 11602, 2019, pp. 187–203.

[8] N. R. Weeraddana, M. Alfadel, and S. McIntosh, "Dependency-induced waste in continuous integration: An empirical study of unused dependencies in the npm ecosystem," in *Proceedings of the 2024 ACM International Conference on the Foundations of Software Engineering (FSE 2024)*, 2024, pp. 2632–2655.

[9] smallrye, "smallrye-mutiny," https://github.com/smallrye/smallrye-mutiny/issues/1577, April 12th,2024, accessed: December 1st, 2024.

[10] snowflakedb, "snowflake-jdbc," https://github.com/snowflakedb/snowflake-jdbc/issues/1622, January 30th, 2024, accessed: December 1st, 2024.

[11] devonfw, "Ideasy," https://github.com/devonfw/IDEasy/issues/685, October 10th, 2024, accessed: December 1st, 2024.

[12] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, "Towards better dependency management: A first look at dependency smells in python projects," *IEEE Transactions on Software Engineering (TSE)*, vol. 49, no. 4, pp. 1741–1765, 2023.

[13] N. Harrand, A. Benelallam, C. Soto-Valero, F. Bettega, O. Barais, and B. Baudry, "API beauty is in the eye of the clients: 2.2 million maven dependencies reveal the spectrum of client-api usages," *Journal of Systems and Software (JSS)*, vol. 184, p. 111134, 2022.

[14] J. Latendresse, S. Mujahid, D. E. Costa, and E. Shihab, "Not all dependencies are equal: An empirical study on production dependencies in NPM," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, 2022, pp. 73:1–73:12.

[15] D. Jaime, "Goblin: Neo4j maven central dependency graph," Sep. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13734581

[16] D. Jaime, J. E. Haddad, and P. Poizat, "Goblin: A framework for enriching and querying the maven central dependency graph," in *Proceedings of the 21st IEEE/ACM International Conference on Mining Software Repositories (MSR 2024)*, 2024, pp. 37–41.

[17] eclipse jkube, "jkube," https://github.com/eclipse-jkube/jkube/issues/3470, October 20th, 2024, accessed: December 9th, 2024.

[18] T. W. MacFarland, J. M. Yates, T. W. MacFarland, and J. M. Yates, "Mann–whitney u test," *Introduction to nonparametric statistics for the biological sciences using R*, pp. 103–132, 2016.

[19] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 10, pp. 3790–3807, 2022.