Breaking Strong Encapsulation: A Comprehensive Study of Java Module Abuse

Yirui He
yiruih@uci.edu
University of California, Irvine
Irvine, California, USA

Yongbo Chen
yongboc1@uci.edu
University of California, Irvine
Irvine, California, USA

Jessy Ayala
jessya1@uci.edu
University of California, Irvine
Irvine, California, USA

Yecheng Zhou
yechenz2@uci.edu
University of California, Irvine
Irvine, California, USA

Qiran Wang
qiranw1@uci.edu
University of California, Irvine
Irvine, California, USA

Joshua Garcia
joshug4@uci.edu
University of California, Irvine
Irvine, California, USA

Abstract

As an increasing number of software systems reach sizes of hundreds of millions of lines of code, relying solely on code-level abstractions is impractical, posing profound challenges to software maintenance. Java Platform Module System (JPMS) provides architectural abstractions that enable Java engineers to control the required and provided interfaces of a module. Despite numerous advantages of enforced strong encapsulation provided by JPMS, developers still break those encapsulations to access the internal elements of a module. Such practice is referred to as ***Breaking the Strong Encapsulation (BSE)***. This behavior not only complicates the migration to newer versions (e.g., Java 9) but also threatens the integrity and safety of software projects. Since the BSE problem is still underexplored, in this work, we conduct the first empirical study to identify and characterize this widespread and impactful problem. We first collect a comprehensive dataset containing 4,079 GitHub issues and then investigate those issues from various perspectives, including symptoms, abuse sources, desired functionalities, and resolutions. Our empirical study highlights a tension in the Java module system: module developers aim to enforce strong encapsulation while module users frequently attempt to break the encapsulation. Our finding also emphasizes the need for both practitioners and researchers to develop effective strategies to mitigate these problems, offering an understanding of BSE characteristics to inform future detection and repair efforts.

CCS Concepts

• **Software and its engineering → Software architectures; Software evolution**.

Keywords

Software Architecture, Component-Based Architecture, Java Platform Module System, Health of Java Software Ecosystems

ICSE '26, RIO DE JANEIRO, BRAZIL

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/10.1145/3744916.3764526

ACM Reference Format:
Yirui He, Yongbo Chen, Jessy Ayala, Yecheng Zhou, Qiran Wang, and Joshua Garcia. 2018. Breaking Strong Encapsulation: A Comprehensive Study of Java Module Abuse. In *Proceedings of 48th IEEE/ACM International Conference on Software Engineering (ICSE '26).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764526

1 Introduction

Software systems have grown increasingly large and more complex, resulting in many legacy software systems, which tend to remain in deployment and use for years or even decades. Maintaining such large and complex systems through traditional code-level abstractions (e.g., functions, classes, and packages) has become increasingly challenging, especially as individual systems reach hundreds of millions of source lines of code. A major means of handling such large and complex software systems has traditionally been to utilize constructs from software architecture (e.g., components, connectors, and configurations) [85, 91, 92]. Unfortunately, ensuring that architecture-level abstractions and code-level abstractions are consistent has been a longstanding problem at the core of *software architectural drift and erosion* [86, 93], which we collectively refer to as *architectural decay*.

A major example of architectural decay is the Java Development Kit (JDK), which JDK engineers have described as a ball-of-mud architecture [83, 88]. Projects built using the JDK have abused access to powerful, unsupported, and dangerous mechanisms (e.g., `sun.misc.Unsafe` [57, 78]). As documented by JDK engineers, this exploitation resulted in a rigid architecture [88] and forced engineers to maintain unsupported APIs rather than implement new features or conduct critical maintenance [50–52, 87, 89, 90]. To address these issues, the JDK introduced the Java Platform Module System (JPMS), providing architectural modules as language-level constructs that define explicit boundaries and specify permitted inter-module communications [56]. This modular architecture significantly enhances the maintainability of both the JDK itself and Java-based applications. The adoption of JPMS modules continues to grow with over seven thousand unique JPMS modules in existence [44]. However, some developers are still **b**reaking the modules' **s**trong **e**ncapsulation (*BSE*) [51, 52] to access module internals. This behavior not only undermines JDK engineers' efforts in developing JPMS, but results in another architectural decay problem [85, 92] by having the *prescriptive architecture* (i.e., the architecture as-intended

or as-documented by its architects) of a Java system violated by illegal access of a JPMS module in the system's *descriptive architecture* (i.e., the architecture as-implemented by developers).

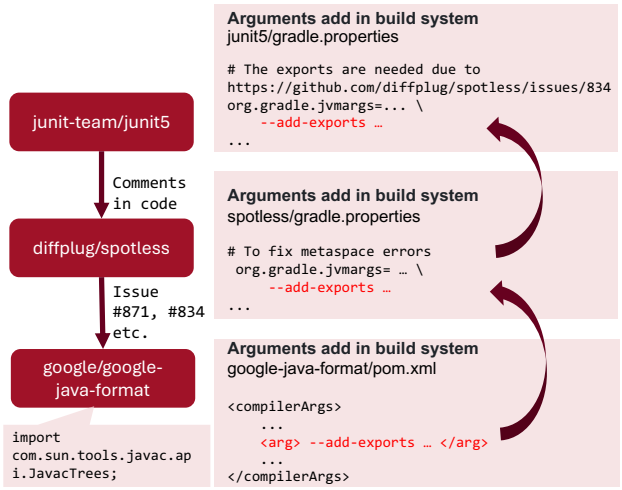


Figure 1: Example of BSE Problem Propagation

The BSE problem can introduce both compile-time and runtime errors, thereby hindering migration to Java versions above 9. Furthermore, the BSE problem has a significant impact as it can propagate through the software ecosystem via project dependencies, affecting any project with direct or indirect dependencies on modules that require access to internal APIs. For instance, as depicted in Figure 1, *JUnit5* [33] breaks the JDK's strong encapsulation and accesses the internal APIs by adding Java Virtual Machine (JVM) options, i.e., `--add-exports`. The need for *JUnit5* to break the JDK's strong encapsulation originated from its dependency on the *Spotless* plugin [45], which also required the same set of packages. GitHub issues in *Spotless* [11, 42] indicate that the inclusion of these options originated from *Google Java Format* [10], a code formatting tool implementing Google Java Style guidelines [26]. This case demonstrates how *Google Java Format*'s use of JDK internal APIs creates issues that propagate through the dependency chain, forcing dependent projects to break strong encapsulation.

Helping developers identify and mitigate these previously unexplored architectural module-related defects is important for addressing *architectural decay* and improving the maintainability of both the JDK and Java-based software. While prior research has examined aspects of JPMS including migration, security, and over-exposure concerns [55, 60, 67, 81], the challenges brought by abusing JPMS modules remain largely unexplored. To address this gap, we present the first empirical study investigating BSE issues that emerge from problematic Java module interactions. This work makes three major contributions:

- We conducted the first empirical study of BSE problems arising in JPMS modules across the Java ecosystem by collecting and studying issues related to BSE from 4,079 GitHub [1] issues.
- We summarized a taxonomy of BSE problems, categorizing their symptoms, abuse sources, targeted components (modules, packages, and APIs), and resolution strategies.
- To facilitate future research on BSE problems, including their detection and repair, we publicized our dataset [2].

The remainder of the paper is organized as follows: Section 2 provides background on JPMS and BSE; Section 3 describes our methodology; Section 4 introduce the research questions; Section 5 summarizes our taxonomy; Section 6 details the obtained results; Section 7 presents implications and threats to validity; Section 8 outlines related work, and Section 9 concludes the paper.

2 Background

2.1 Java Platform Module System

2.1.1 Module and Module Directives. JPMS introduces Java module, a higher level of organization beyond packages. A module in Java is a uniquely named, reusable entity encompassing related packages and resources (e.g., images and XML files), and is defined by a module descriptor that specifies several key aspects: the module's name, its dependencies on other modules, the packages it explicitly shares with other modules (implicitly keeping all others private), the services it provides and consumes, and the modules it permits for reflection. More specifically, developers are allowed to declare modules (i.e., the *prescriptive architecture*) using various *module directives*. These declarations, referred to as *module declarations*, and specified in a file called `module-info.java`, are compiled into *module descriptors* that are stored in a file called `module-info.class` that resides in the module's root folder [63].

Figure 2 shows the declarations and the relation between 4 modules from JUnit5 [32], a popular open-source Java testing framework, which has now adopted JPMS: `org.junit.jupiter.engine`, `org.junit.platform.commons`, `org.junit.platform.launcher`, and `org.junit.platform.engine`.

The module declaration consists of a unique name and a body. In the module body, a developer can use the following module directives to specify module interfaces and their usage [63]:

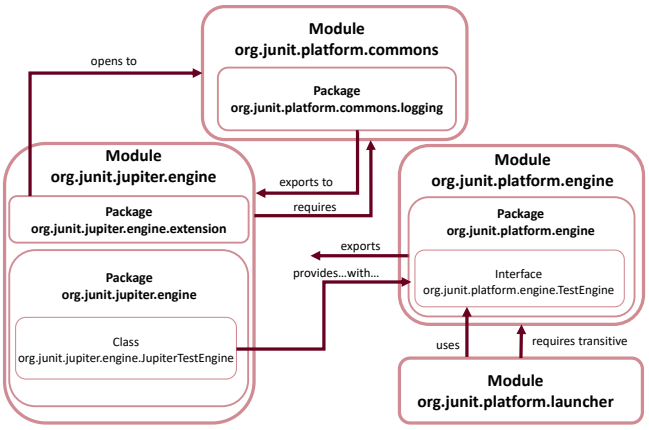
- The `requires` directive establishes module dependencies, for instance, as depicted in Figure 2 with `org.junit.jupiter.engine` requiring `org.junit.platform.commons`.
- The `exports` directive allows a module, such as `org.junit.platform.engine`, to make its package accessible to other modules at both compile and run time.
- The `opens` directive, exemplified by `org.junit.jupiter.engine` opening package `org.junit.jupiter.engine.extension` to `org.junit.platform.commons`, permits runtime reflective access to a package's types and members.
- The `provides...with` directive declares a module's provision of a service implementation [14, 56], like `org.junit.jupiter.engine` offering `TestEngine` interface with `JupiterTestEngine`.
- Finally, the `uses` directive indicates a module's consumption of a service, such as `org.junit.platform.launcher` using the `TestEngine` service provided by `org.junit.platform.engine`.

JPMS enhances Java by enforcing module dependency specification and strong encapsulation through explicit declarations for both compile-time and runtime access [56], thereby improving system security and design maintainability.

2.1.2 Unnamed Module. While JPMS enforces strong encapsulation and requires all code to reside in modules, creating explicit

```
1  module org.junit.platform.launcher {
2      requires transitive org.junit.platform.engine;
3      uses org.junit.platform.engine.TestEngine;
4      ...
5  }
6
7  module org.junit.jupiter.engine {
8      requires org.junit.platform.commons;
9      provides org.junit.platform.engine.TestEngine
10       with org.junit.jupiter.engine.JupiterTestEngine;
11     opens org.junit.jupiter.engine.extension to
12       org.junit.platform.commons;
13    ...
14 }
15
16 module org.junit.platform.commons {
17     exports org.junit.platform.commons.logging to
18       org.junit.platform.engine;
19     ...
20 }
21
22 module org.junit.platform.engine {
23     exports org.junit.platform.engine;
24     ...
25 }
```

(a) Module Declarations and Their Directives Provided in Their `module-info.java` Files



(b) Dependencies Between Modules Based on Their Directives

Figure 2: Inter-Dependencies of Four Example Modules

modules is not mandatory for backward compatibility. The *unnamed module* serves as a container for all "non-modular classes," which include (1) at compile time, classes being compiled absent a module descriptor, and (2) at both compile and run time, any class loaded from the classpath [6, 76].

Unnamed modules can operate without module descriptors. Since they lack formal names, their direct reference in module declarations is prevented. To maintain compatibility with JPMS, unnamed modules possess three critical characteristics: (1) implicit readability of all modules in the module graph, (2) automatic export of all packages, and (3) unrestricted reflective access to all modules. Despite an unnamed module's inability to be directly referenced in module declarations due to its lack of a formal name, these characteristics, particularly the exposure of all packages, significantly undermine the encapsulation guarantees that JPMS was designed to provide [6, 76].

2.2 Breaking Strong Encapsulation

JPMS enforces strong encapsulation, ensuring the descriptive architecture follows the accessible scope specified in the prescriptive architecture. However, many developers still break Java modules' strong encapsulation and leverage some powerful functionalities offered by encapsulated internal APIs (e.g., `sun.misc.Unsafe` [57]).

To ease the transition of applications to Java 9 and higher, JDK developers designed several arguments to make projects that violate the strong encapsulation principle usable. These arguments explicitly make encapsulated code accessible and break Java modules' strong encapsulation, which is defined in the module descriptor.

To that end, Java developers can employ the following argument to *export* packages:

- `--add-exports <module>/<package>=<readingmodule>` allows for the exporting of `<package>` from `<module>` to `<readingmodule>`. Here we refer to `--add-exports <module>/<package>=<readingmodule>` as an *argument*, where `--add-exports` is an *option*, and `<module>/<package>=<readingmodule>` is the *parameter* of the option.

To *open* packages for run-time access using reflection, two methods are available:

- `--add-opens<module>/<package>=<readingmodule>`, allows for the opening `<package>` from `<module>` to `<readingmodule>`.
- `--illegal-access=(permit | warn | debug)` argument enables access to packages that were accessible at run-time through reflection in JDK 8 but encapsulated in JDK 9 and later versions [20–23]. This command allows code that previously had unauthorized entry into JDK internals to function as in earlier releases. The evolution of this option is depicted in Table 1. From Java 9 to Java 15, when a module's strong encapsulation is broken due to illegal reflective access of an unopened package of that module, which we refer to as a *BSE warning*, the JDK emits a warning message and does not enforce strong encapsulation by default. Java 16 enforces this form of reflection-oriented strong encapsulation by default. For Java 17 onward, such strong encapsulation can no longer be disabled by this option.

The default JVM's behavior is strictly aligned with the encapsulation rules, effectively preventing illegal access during compile time. In contrast, the runtime environment initially adopted a more lenient stance to support backward compatibility [20, 21, 24], i.e., `--illegal-access=permit` as the default. This mechanism aimed to facilitate smoother migration and enhance the compatibility of applications originally developed for Java 8 and earlier [18, 56, 84]. However, this allowance for default runtime access has been progressively phased out while the option was deprecated in JDK 16 and made obsolete in JDK 17 [22, 23]. Since Java 17, previously issued BSE warnings for illegal access have become exceptions and errors without specialized configuration. This shift underscores the platform's commitment to enforcing a long-established principle within the JDK, a significant step for enforcing module integrity and encapsulation in Java.

Note that BSE problems are a form of architectural decay because they only occur when the prescriptive architecture, as specified in `module-info.java`, no longer matches the descriptive architecture (i.e., the actual code implementation). More specifically, `--add`

Table 1: Increasing Strong Encapsulation Enforcement

JDK Version	Default Argument	Run-time Symptom
JDK 9-15	`--illegal-access=permit`	BSE warning
JDK 16	`--illegal-access=deny`	error/exception
JDK 17+	None	error/exception

`-exports`, `--add-opens`, and `--illegal-access` JVM options enable architectural erosion by allowing architectural dependencies that violate a software system's prescriptive architecture.

3 Methodology

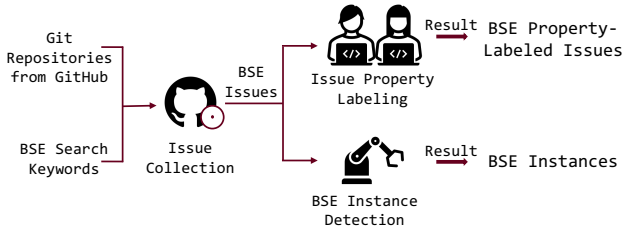


Figure 3: Process of BSE Issue Study

Figure 3 shows the high-level process we follow to conduct our study. Initially, we collected GitHub issues (*Issue Collection*) with five predefined keywords. We refer to each resulting GitHub issue involving breaking strong encapsulation as a *BSE issue*. Each BSE issue contains at least one *BSE abuse instance*, defined as either an occurrence of one of the five keywords described below corresponding to JVM arguments that break strong encapsulation or a BSE warning. For each BSE issue, we conduct two parallel analyses: *Issue Property Labeling* and *Issue Instance Detection*. The property labeling process employs open coding to systematically categorize five key properties: (1) issue classification, (2) problem symptom, (3) abuse source, (4) abuse target, and (5) resolution strategy, when available. Simultaneously, we extract and analyze all BSE instances present in the issue.

Issue Collection. To collect relevant GitHub issues, we selected five key JVM arguments that enable the circumvention of strong encapsulation as introduced in Section 2.2: `--illegal-access=permit`, `--illegal-access=warn`, `--illegal-access=debug`, `--add-exports`, and `--add-opens`. These arguments were determined based on official Java documentation [37, 50, 52]. Subsequently, we leveraged the GitHub API [1, 9] to retrieve issues containing one or more of these five keywords. Our data collection process took 15 days (from Nov. 1, 2023, to Nov. 15, 2023) and resulted in 11,496 data records.

To ensure research validity and generalizability, we focused on popular and actively maintained projects, as these tend to have better-established community rules (e.g., GitHub issue format) and greater community impact. Following established methodology from prior work [70], we excluded GitHub repositories that met any of the following criteria: (1) *personal repositories*: repositories with fewer than 5 contributors; (2) *inactive repositories*: repositories having no open issues or are archived; (3) *repositories with a trivial history*: repositories having fewer than 100 commits; (4) *unpopular repositories*: repositories with fewer than 10 stars and 10 forks. Following the removal of pull requests, which is the noise introduced by the GitHub API [1, 9] retrieval mechanism, and duplicate issues, we obtained a refined dataset comprising 4,079 unique GitHub issues from 1,351 distinct repositories. The issues' creation dates span over six years from Jan. 10, 2017 to Oct. 3, 2023 and the distribution over time is shown in Figure 5.

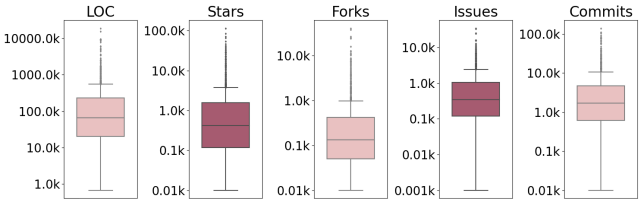


Figure 4: Statistics of GitHub Repositories (in Log Scale)

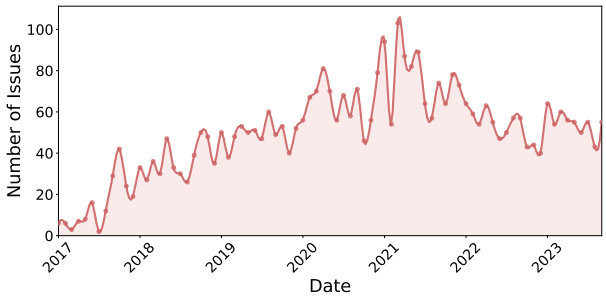


Figure 5: Issue Creation Date Distribution over Time

The statistical characteristics of the analyzed repositories (i.e., number of stars, forks, issues, commits, and lines of code (LOC)) are *presented on a logarithmic scale* in Figure 4. The repositories in our dataset exhibit substantial development activity and community engagement, with average values of 339,144 LOC, 2,311 stars, 701 forks, 1,124 issues, and 5,442 commits. These metrics suggest that BSE problems frequently occur in popular, actively maintained GitHub projects.

Issue Property Labeling. To systematically characterize BSE problems, we analyzed the 4,079 BSE issues using an open-coding methodology, following established practices in empirical software engineering research [59, 70, 74].

To minimize subjective bias during the labeling process, two co-authors independently reviewed the same set of issues following the best practice of prior work [54, 82]. They manually inspected the issue description and all other information (e.g., issue discussion, pull request, commit messages, source code, comments, and linked issues) to identify the related label. An issue was labeled only when its primary focus pertained to BSE problems, excluding cases where BSE abuse instances were present but irrelevant to the main discussion (e.g., Issue #4694 [16] contains abuse instances such as `--add-exports`, but focuses primarily on hardware architecture configuration problems, thus receiving no labels). During the open-coding process, if two co-authors both agreed on the labeling, the

result was finalized. In cases of disagreement, the co-authors convened to discuss and resolve any discrepancies in labeling. Finally, they reached a consensus on the categorization of BSE problems. This systematic approach yielded 1,454 labeled issues across 732 GitHub repositories.

BSE Instance Detection. While not all of the 4,079 BSE issues primarily focus on BSE problems, each contains at least one *BSE abuse instance*—defined as either the use of JVM options (e.g., `--add-opens` or `--illegal-access`) that break strong encapsulation or runtime BSE warning messages indicating BSE behaviors.

Each BSE abuse instance corresponds to one of the five JVM arguments that break strong encapsulation or runtime information indicating BSE. For two JVM arguments (i.e., `--add-exports` and `--add-opens`), *BSE Instance Detection* automatically determines *target packages*, *target modules*, and *source modules*. For example, `--add-exports java.base/java.lang=ALL-UNNAMED` indicates that the target package `java.lang` located in target module `java.base` is abused by unnamed modules. For BSE warnings, we extract the *source file* and *target API* information to determine whether reflection abuse originates from the project code or external dependencies. We identified external dependencies for BSE instances by automatically analyzing directories containing dependency information (e.g., *.gradle* for Gradle, *.m2* for Maven) and third-party library locations (e.g., *plugin*, *lib*). Consider the following warning instance: *WARNING: Illegal reflective access by p1.C1(jar:file:/.m2/example.jar!/) to constructor java.lang.invoke.MethodHandles$Lookup(java.lang.Class))*. This warning reveals that *p1.C1* from source file *example.jar* located in *"/.m2/"* attempts to access the target API *constructor java.lang.invoke.MethodHandles$Lookup(java.lang.Class)*. The source is identified as an external dependency based on its location in the *.m2* directory, Maven's default repository for downloaded dependencies. However, this approach may underestimate external sources, as third-party libraries could reside outside these specified directories.

4 Research Questions

To characterize BSE problems, we formulate four research questions and elaborate on them in the rest of this section.

Given the absence of prior research specifically addressing BSE problems, our study aims to characterize these problems by first examining the symptoms identified by developers as reflected in their GitHub issue discussions. As a result, we study the following research question:

> ***RQ1 (Symptom):*** *What common symptoms of BSE problems are identified on GitHub?*

Developers often break strong encapsulation due to specific code requiring access to encapsulated code or resources. To comprehensively understand BSE problems, it is important to examine the particular *source* modules that require internal code or resources and identify the properties of these modules. This leads us to study our next research question:

> ***RQ2 (Source):*** *Which type of the modules are the foremost sources of BSE abuse? What are the distinctive properties that characterize such abusive sources?*

When developers break strong encapsulation, they invariably have a specific *target* within a particular module that they aim to access. To understand these desired targets, we conducted an extensive analysis of both BSE property-labeled issues and BSE instances. Consequently, we investigate the following research question:

> ***RQ3 (Target):*** *What are the primary target modules of BSE abuse instances? What are the distinctive properties that characterize such abused modules?*

Given the impact of BSE on the Java ecosystem, developers have implemented various resolutions to ensure the correct operation of their software. These resolutions and the associated experiences provide valuable insights for the broader developer community. Consequently, we formulate the following research question to explore this aspect:

> ***RQ4 (Resolution):*** *In what ways do developers resolve challenges arising from the breaking of JPMS modules' strong encapsulation?*

5 BSE Taxonomy

Following our systematic analysis process, we investigate our research question and produce a taxonomy of BSE problems shown in Figure 6 and describe it in the rest of this section.

Problem Symptom categorizes the BSE problems identified in BSE-related issues. The predominant problems fall into two categories: (1) *Error* and *Exception*, which can lead to runtime failures (e.g., program crashes) or (2) BSE warning indicating violations of strong encapsulation principle.

An **Abuse Source** is a file or module that breaks the strong encapsulation of a JPMS module and is obtained from both BSE property-labeled issues and BSE instances. We categorize abuse sources along two dimensions: *module naming* (i.e., *named module* or *unnamed module*) and *project scope* (i.e., *internal* or *external*).

We first classify the abuse source based on *module naming*. A *named module*, which is a JPMS module given an explicit name either through a `module-info.java` file, which we refer to as a *developer-created module*, or through automatic generation from existing Java ARchive (JAR) [17] files in the project which we refer to as *JAR-based modules*. Such modules derive each module's name from an explicit name specified by the developer in the JAR file or from the name of the JAR file itself. The *unnamed module*, as introduced in Section 2.1.2, represents a distinct category. Unnamed modules and JAR-based modules export and open all their packages, essentially making them inherently poorly modularized and weakly encapsulated modules. Moreover, we also characterize both BSE issues and instances by their *project scope* as either *internal* or *external*. An *internal* source indicates that the repository's own code requires access to Java module's internal APIs, while an *external* source emerges when the repository's dependencies (e.g., third-party libraries) require such access.

An **Abuse Target** is a software entity, i.e., a *module*, *package*, or *API*, whose strong encapsulation is broken and, thus, *abused* by another software entity, i.e., a file or module, and is obtained from BSE Instance Detection in Figure 3. Abused target instances serve as indicators of desired but unsupported functionality within the API ecosystem. By leveraging the insights gained from abused targets, the software development community can iteratively improve

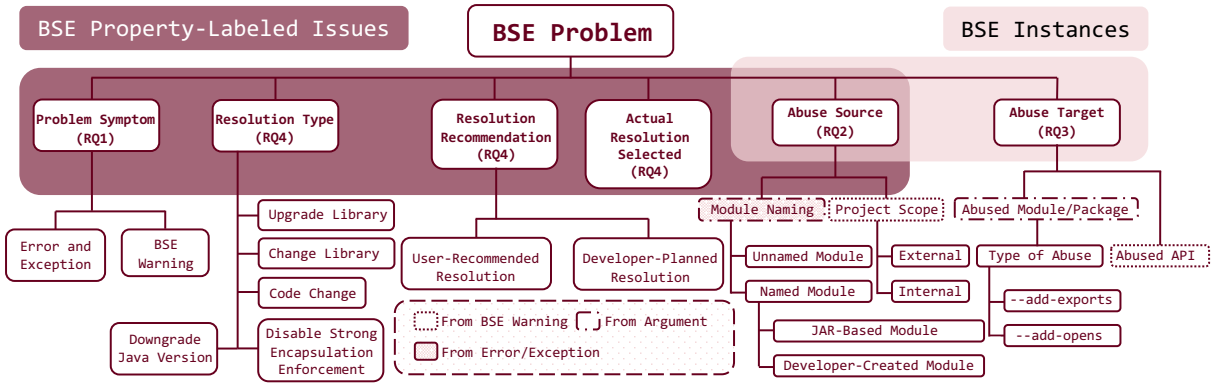


Figure 6: Taxonomy of the BSE Property-labeled Issues and BSE Instances

API design and implementation, ultimately enhancing the overall developer experience and code quality. Specifically, we explore the modules, packages, and APIs—including functions, fields, and constructors—that abusive modules typically attempt to access, as well as the features of those target modules that often attract the breaking of strong encapsulation.

In Figure 6, **Resolution Type** characterizes the strategies mentioned or employed to resolve BSE-related problems we summarized from the BSE property-labeled issues. We identify five primary resolution strategies: *upgrade library*, *change library*, *code change*, *disable strong encapsulation enforcement*, and *downgrade Java version*. *Upgrade library* addresses issues by adopting newer library versions that incorporate fixes for BSE-related problems. *Change library* involves adopting alternative libraries that comply with strong encapsulation principles. Developers implementing *code change* make direct modifications to the project's source code. As detailed in Section 2.2, *disable strong encapsulation enforcement* encompasses various ways to break encapsulation. Finally, *downgrade Java version* involves reverting to pre-Java 9 versions, particularly Java 8, which lacks strong encapsulation enforcement.

Resolution Recommendation includes *user-recommended resolution* and *developer-planned resolution*. *User-recommended resolution* represents resolutions proposed and discussed by GitHub project users, while *developer-planned resolution* includes future tasks outlined by open-source project members to address BSE issues. A *developer-planned resolution* is contributed by open-source project members specifying future tasks, including outlining a resolution for future applications for addressing BSE issues. Finally, **Actual Resolution Selected** indicates the implemented resolutions related to corresponding BSE-related issues, demonstrating the developers' final choice among available resolution strategies.

6 Result and Analysis

During the open-coding process described in Section 3, we assigned each GitHub issue one or more of four labels identified through the process: **Report BSE warning** (980 issues), **Report Exception/Error** (296 issues), **Developer-Planned Resolution** (88 issues), and **User-Recommended Resolution** (15 issues). We also identified BSE instances based on three recurring patterns (`--add-exports`, `--add-opens`, and BSE warning) to facilitate efficient information extraction. Based on the research questions described in Section 4, we developed the BSE taxonomy shown in Figure 6. We now present and discuss the results of our study.

6.1 RQ1: BSE Problem Symptoms

6.1.1 Study Result. Among our 1,454 BSE property-labeled issues, 980 issues (67.40%) report BSE warnings. Such BSE warnings are triggered by the `--illegal-access=(permit|warn|debug)` argument, which was disabled by default in Java 16 and is no longer supported in Java 17. Consequently, these BSE warnings can escalate into errors and exceptions [22, 23, 38] as discussed in Section 2.2. Additionally, 296 out of 1,454 (20.36%) BSE property-labeled issues report exceptions or errors, with 283 (95.61%) containing detailed information about the exceptions or errors, i.e., stack traces. With that detailed information, we identified the following three most prevalent exceptions and errors:

- `java.lang.reflect.InaccessibleObjectException` (168 issues)
- `java.lang.IllegalAccessError` (71 issues)
- `java.lang.IllegalAccessException` (34 issues)

Finding 1: In our study, 980 BSE issues report BSE warnings, surpassing the number of issues reporting errors and exceptions. This observation indicates that users and developers are facing a significant number of BSE warnings, and they are trying to actively address potential errors and exceptions before they manifest by submitting GitHub issues.

Furthermore, to evaluate how different JDK versions enforce strong encapsulation to varying degrees, we count the number of issues of different BSE symptoms in our dataset across JDK versions, as depicted in Figure 7. The vertical axis represents the frequency of each BSE problem type (i.e., BSE warning or error/exception), while the horizontal axis displays the corresponding JDK version.

For BSE warnings, JDK 11 exhibited the highest frequency of associated BSE issues (347), likely due to it being the first Long-Term-Support (LTS) version to include JPMS. Additionally, JDK 9 exhibits a high frequency of BSE issues (154) due to its status as the first version to adopt JPMS.

Regarding errors or exceptions, the majority of reports were concentrated on JDK 16 (88 issues) and JDK 17 (92 issues). JDK

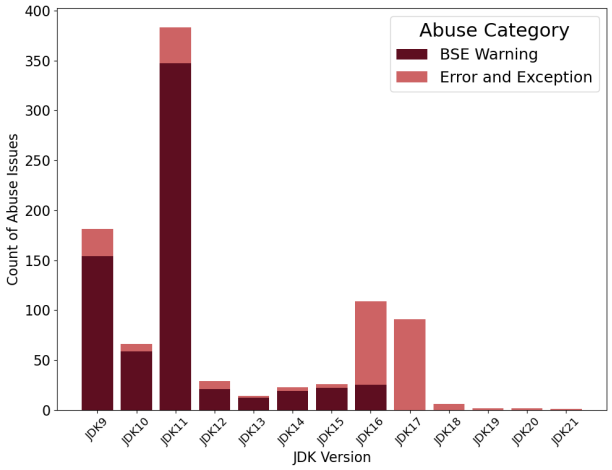


Figure 7: Counts of BSE Warnings and Error/Exception Issues Across JDK Versions

17 exhibited the highest rate of errors, which is sensible for two reasons: (1) JDK 17 is the first version for which the `--illegal-access` option is deprecated, and (2) JDK 17 is an LTS version which many JDK applications and platforms are likely to use. Although JDK 16 is not an LTS version, it is the first version for which the `--illegal-access` option is disabled by default; as a result, strong encapsulation of illegal reflective access is enforced by default. This suggests a tendency among users or developers of the projects in our study to rely, possibly excessively, on default settings, especially for JDK versions before 16, i.e., versions that may warn but not enforce strong encapsulation by default. Consequently, such projects do not adopt resolutions that actually solve BSE problems, which we will discuss further for RQ4; instead, they often let BSE warnings evolve into errors or exceptions that lead to project failure at runtime. The decision of not fixing BSE problems or not replacing problematic dependency libraries increases architectural technical debt, where users or developers rely on disabling enforcement, but pay the price later when addressing errors and exceptions.

Finding 2: JDK 11 and JDK 17 exhibit the highest frequencies of issues related to BSE warnings and errors/exceptions, respectively. Notably, JDK 16 demonstrates a significant increase in error/exception issues (88), indicating widespread dependency on JDK's default settings that trigger BSE warnings. These BSE warnings, reported across 980 BSE issues, indicate the presence of architectural technical debt that subsequently evolves into runtime failure.

6.2 RQ2: Abuse Source

6.2.1 BSE Instances Result. Building upon our findings from RQ1 (Section 6.1), which demonstrated developers' concerns regarding BSE warnings, we conducted an automated analysis to detect BSE warning instances across our dataset. Our analysis revealed 4,720 instances of BSE warnings distributed across 2,366 distinct BSE issues. Notably, 3,485 instances (73.8%) were attributed to external sources, indicating that the majority of the issues involving BSE problems stem from external dependencies, as opposed to issues originating within the GitHub projects themselves.

Table 2: Overview of Target Instance Analysis

Keyword	# Instances	# Covered GitHub Issues
`--add-exports`	5,683	638
`--add-opens`	6,174	1,032

We also automatically classified the *source module* performing BSE abuse as either an *unnamed module* or a *named module*, analyzing both `--add-exports` and `--add-opens` options. Table 2 shows the number of instances and corresponding GitHub issues, revealing high abuse frequency with an average of 8.91 `--add-exports` and 5.98 `--add-opens` instances per issue.

Finding 3: In 4,900 out of 5,683 BSE instances (86.22%) that involve `--add-exports`, the source modules are unnamed modules (refer to Section 2.1.2). Furthermore, among 6,174 `--add-opens` instances, 6,001 (97.20%) involve unnamed modules attempting to reflectively access internal components. These findings suggest that the abusive code often does not reside within any named Java modules and that poorly modularized code in the modern Java ecosystem tends to be code that abuses other Java modules.

6.2.2 BSE Property-Labeled Issues Result. Through the open-coding process, we identified the source of BSE issues to understand the cause of BSE problems. Out of 1,454 examined BSE issues, 1,104 were identified as originating from external sources, a finding that corresponds closely with our analysis of BSE warning instances discussed in Section 6.2.1. Particularly, for BSE errors and exceptions, 195 out of 296 issues were attributed to external failures. This result highlights the severe implications of problems that arise from inadequate responses to the enforcement of strong encapsulation.

Finding 4: Among all BSE instances, third-party libraries are the root cause behind 73.83% (3,485 out of 4,720 instances) of the BSE warning instances. Similarly, 75.93% (1,104 out of 1,454 issues) of the BSE property-labeled issues were identified as external issues. This reveals that the majority of projects that break Java modules' strong encapsulation originate from their dependencies on third-party libraries, illustrating how BSE, as a form of architectural decay that introduces technical debt, can propagate throughout project dependency chains within the ecosystem.

From the previous finding, we find that BSE abuse sources are particularly prevalent in the case of external project scope, especially with respect to third-party libraries. Our analysis further reveals that unnamed modules constitute the primary abuse source, with a notable tendency for generating BSE errors and exceptions:

Finding 5: Our analysis of BSE errors and exceptions shows that 88.85% (263 out of 296) of issues stem from unnamed modules attempting to access internal packages of other modules, while only 5.07% (15 out of 296) originate from named modules. This finding aligns with Finding 3, indicating that monolithic, poorly modularized unnamed modules are more prone to BSE problems.

6.3 RQ3: Abuse Target

Having examined the *sources* of BSE abuse, we now investigate their *abused targets*.

6.3.1 BSE Instance Result. We developed an automated approach to identify internal package dependencies by extracting two key elements: (a) the *target module* and (b) the internal package accessed, i.e., the *target package*, for both `--add-exports` and `--add-opens` options. Table 2 presents the number of these abuse instances and their corresponding GitHub issues, with each issue containing an average of more than 5 instances.

As shown in Table 3, the `java.base` module, which provides core APIs for the Java SE Platform [25, 35], exhibits the highest frequency of abuse access attempts across all 1,351 GitHub repositories. Among all BSE instances analyzed, 35.10% (1,995 out of 5,683) of `--add-exports` instances target the `java.base` module. Similarly, 68.32% (4,218 out of 6,174) of `--add-opens` instances access the `java.base` module to reach APIs not intended for public use.

To better understand the abused instances that target the `java.base` module, we identified the three most frequently targeted packages for both `--add-exports` and `--add-opens` operations. The results are presented in Table 3. For each package, we show its capabilities and potential misuse through representative example classes for `exports` and official package descriptions for `opens`. For packages being exported from `java.base`, we selected representative example classes based on developer discussions from online platforms such as Stack Overflow [46]. This selection approach was necessary because unexported JDK packages lack official documentation, as they are intended solely for internal JDK use. The analysis of `opens` reveals that the three most frequently targeted packages are `java.lang`, `java.util`, and `java.io`. While these packages are officially exported [31] and their public elements (i.e., variables and functions) are accessible to dependent modules, their private elements remain encapsulated. However, the `--add-opens` option breaks encapsulation by enabling runtime access to private members, violating both module encapsulation and the principle of information hiding [58]. Such abusive access to private members of system classes can lead to severe security consequences [69].

Finding 6: Within `java.base`, the most frequently accessed internal packages through `exports`—`jdk.internal.misc`, `jdk.internal.ref`, and `sun.security.util`—are primarily used for unauthorized low-level operations (e.g., CPU feature access [57]) and bypassing standard JVM mechanisms (e.g., creating unverified anonymous classes [57]). These usages deviate from the platform's intended behavior. In contrast, the most frequently abused packages via `opens` in `java.base`—`java.lang`, `java.util`, and `java.io`—are already publicly exported packages. This pattern suggests widespread developer attempts to gain reflective runtime access to non-public elements of exported packages, violating fundamental software engineering principles of information hiding [58, 69].

In addition, we analyzed the specific APIs accessed through reflection and their intended functionalities. Our analysis focuses on runtime reflective access instances (i.e., *BSE warnings*). The five most frequently accessed APIs are presented in Table 4. The high frequency of abused target instances indicates a misalignment between Java modules' design and developer requirements. This information can guide researchers in developing automated tools for program repair when replaceable APIs exist [19]. In cases where suitable alternatives are not available, these instances highlight opportunities for practitioners to develop new libraries that address the identified gaps. This systematic understanding of API usage patterns can inform future improvements to Java modules that better align API design with developer needs, ultimately enhancing system integrity, security, and usability.

Finding 7: The illegally accessed JDK internal APIs shown in Table 4 exemplify critical Java mechanisms for class loading, memory management, reflective method invocation, buffer handling, and data structure implementation. These APIs embody critical aspects of Java's implementation to security, performance optimization, and low-level system interactions. While these internal APIs are explicitly restricted from public use, their frequent abusive access highlights a tension between the modules' encapsulation boundaries and developers' practical requirements.

6.4 RQ4: BSE Resolution

The prevalence of BSE issues in the open-source ecosystem persists, despite the considerable time elapsed since the introduction of JPMS. Addressing these issues while maintaining adherence to JPMS strong encapsulation principle requires understanding the underlying challenges. Our analysis of open-source projects examines the resolution patterns of these encapsulation violations.

Our analysis of BSE resolutions focuses on two specific types of pre-identified issues that provide detailed resolution information: **Developer-Planned Resolutions** and **User-Recommended Resolutions**. We selected these resolution-centric issues instead of studying all resolved BSE-related issues (i.e., GitHub issues with "closed status") because these two categories of issues not only proposed resolutions and adopted approaches, but also the underlying discussions and decision-making processes. This rich contextual information enables a more comprehensive understanding of BSE mitigation strategies compared to issues that merely report BSE warnings, errors, or exceptions without elaborating on resolution approaches (e.g., issue#178 [3] and #3099 [15]).

6.4.1 Study Result. Our analysis of 103 issues containing **Resolution Recommendations** revealed that *disable strong encapsulation enforcement* was the most frequently proposed resolution, occurring in 66 BSE issues. This category comprises 56 BSE issues with developer-planned resolutions and 10 with user-recommended resolutions. The remaining strategies were less prevalent for recommended resolutions—with 17 issues using *upgrade library*, 4 issues using *code change*, 3 issues using *change library*, and 2 issue using *downgrade Java version*. However, examination of the **Actual Resolutions Adopted** by project maintainers revealed a different distribution. Only 35 cases implemented *disable strong encapsulation enforcement*, with 32 BSE issues originating from developer-planned resolutions and 3 from user-recommended resolutions. Notably, *code change* was adopted in 38 cases, comprising 31 developer-planned resolutions and 7 user-recommended resolutions. The remaining cases were resolved through *upgrading library* (23 BSE issues), *changing library* (5 BSE issues), and *downgrading Java version* (1 BSE issue).

The adoption of provisional fixes such as *disabling strong encapsulation enforcement* or *downgrading the Java version* can lead to

Table 3: Top 3 Most Frequently Abused Packages in the `java.base` Module

Argument	Package	Representative Example Classes for Undocumented Packages (Top-3 Packages) / Official Package Description (Bottom-3 Packages)	# of Abused Instances per Package	# of Abused Instances of `java.base`
`--add-exports`	`jdk.internal.misc`	`Unsafe` class from package `jdk.internal.misc` allows developers to directly access CPU and other hardware features, create an object but not run its constructor or manually manage off-heap memory [57].	143	1,995
	`jdk.internal.ref`	`Cleaner` is a lightweight alternative to finalization and should only be used for simple cleanup tasks to avoid blocking the reference-handler thread [5].	140	
	`sun.security.util`	`Cache` is a key-value mapping class that provides memory-based caching [43].	99	
`--add-opens`	`java.lang`	`java.lang` contains classes fundamental to the Java programming language architecture. Key among these are `Object`, the root of the class hierarchy, and `Class`, whose instances represent classes at runtime [28].	852	4,218
	`java.util`	The `java.util` package contains the collections framework, various utility classes, and legacy classes [30].	556	
	`java.io`	The `java.io` package provides functionality for system input and output operations, including data streams, serialization, and file system interactions [27].	434	

Table 4: Top 5 Most Frequently Abused APIs

API	Description	# of Abused Instances
`java.lang.ClassLoader.defineClass (java.lang.String,byte[],int,int, java.security.ProtectionDomain)`	Converts a byte array to a `Class` instance with a `ProtectionDomain`, defining security attributes for class groups. The `ProtectionDomain` class defines the attributes of a domain that encloses a collection of classes. Instances of these classes are granted a predetermined set of permissions during execution [4].	575
`java.lang.Object.finalize()`	Called by the garbage collector when an object is no longer referenced. It has been deprecated since Java 9 due to its potential to cause performance problems, deadlocks, and resource leaks [36].	177
`java.lang.invoke.MethodHandles$Lookup (java.lang.Class,int)`	In the handle construction, a lookup object acts as a factory, enabling controlled access where direct public constructors are not available [34].	121
`java.nio.Buffer.address`	Supports `Unsafe` access for heap and direct byte buffers, representing relative or start addresses of memory regions [29].	94
`java.util.TreeMap.comparator`	Maintains element order in `TreeMap`. Retrievable via the public method, suggesting potential developer attempts to modify existing `TreeMap` comparators [47].	89

significant challenges and require extensive effort from downstream developers to address the underlying issue effectively in the future (e.g., if they end up needing newer Java features or patches). These decisions not only increase technical debt within the project—e.g., as enabling strong encapsulation or supporting newer Java versions will likely be more difficult as the project evolves and becomes reliant on BSE abuse—but also have the potential to affect the entire ecosystem adversely, as shown in Figure 1.

Finding 8: Our analysis reveals a discrepancy between recommended and implemented resolution strategies. While *disable strong encapsulation enforcement* was the predominant recommendation (66 issues), the actual implementations show equal distribution between *code change* (38 issues) and *disable strong encapsulation enforcement* (35 issues). This pattern indicates that initial recommendations prioritize quick resolution to eliminate BSE warnings, errors, and exceptions, whereas project maintainers often implement fixes that adhere to strong encapsulation principles. This implementation choice reflects maintainers' commitment to addressing BSE issues while maintaining system integrity.

6.4.2 Representative BSE Resolution Examples. Identifying BSE fixes is challenging for two reasons: (1) Commit or pull request messages may not explicitly convey the overarching objective; and (2) developers may lack a clear understanding of their actions or the importance of strong encapsulation, as exemplified in Case 1 below. In contrast, a successful example of re-implementing and removing the problematic dependency is shown in Case 2 below.

Case 1 illustrates scenarios in which developers of Lombok, a Java library that simplifies Java development through language enhancements and annotations, might believe they properly support JDK 16—which is not entirely true—since they still break the JDK modules' strong encapsulation. In a commit entitled "[fixes #2681] [jdk16] support jdk16," that aims to address issue #2681 [8], Lombok developers aim to fix Lombok's encapsulation violations. The issue involves 95 comments, including a discussion involving a JDK developer and 3 Lombok developers, with the most upvoted comment, posted by the JDK developer, receiving over 110 upvotes and the most upvoted comment from a Lombok developer reaching nearly 50 upvotes. Although the time between the opening of the issue and its closing was 91 days (December 15, 2020 to March 16, 2021), over three years later, the issue was still receiving additional comments and mentions quite frequently until August 20, 2024. The impact of this Lombok issue extends beyond the context, as evidenced by a related issue in the JDK Bug System [39], a JIRA instance for tracking JDK bugs. As shown in Figure 8, the attempted solution by Lombok developers for issue #2681 likely involves manipulating the JDK inappropriately. This extensive discussion and controversy surrounding the Lombok issue #2681 showcase the immense challenge and tension arising from the BSE problem.

The conversation also highlights the tension between JDK developers and Lombok developers, as JDK developers prioritize maintainability and security, while Lombok developers oppose these restrictions, arguing that they impose unnecessary burdens on tool maintainers and disrupt compatibility without achieving their intended security goals. This case demonstrates the need for developers to understand and adhere to JPMS strong encapsulation principles for maintaining Java ecosystem integrity. Unauthorized access to undocumented internal APIs introduces potential instability, as changes to these APIs can propagate defects through dependent

code. Such violations of strong encapsulation increase maintenance costs through additional testing, debugging, and repair efforts.

> ***JDK/JDK-8264582: Possible to use*** `sun.misc.Unsafe` ***to hack JDK and circumvent JEP 396:*** With JEP 396, all illegal-access operations are disabled, "except for those enabled by other command-line options, e.g., `--add-opens`". However, it's possible for Java code to enable such operations without having to specify the appropriate command-line options. Link to hack: https://github.com/rzwitserloot/lombok/commit/9806e5cca4b449159ad0509dafde81951b8a8523

Figure 8: Issue from the JDK Repository [39]

Case 2 exemplifies a successful resolution of JDK misuse in the code generation library *cglib* [41], a dependency of Guice [12]. This dependency-related issue in Issue #1133 [13] required significant codebase modifications and remained unresolved for three years, attracting over 60 stakeholders to the discussion. The issue's criticality is further evidenced by its continued references in discussions two years after closure. This extended resolution period reflects both the technical complexity and the issue's significance to the development community, with multiple stakeholders documenting extended wait times, including one case of approximately three years before the underlying problem was addressed.

The implementation of two functionalities previously provided by *cglib* progressed from an initial prototype [40] to a final solution [7]. The development spanned 7 months, encompassing 125 commits that modified 36 files with 2,965 additions and 683 deletions. This extensive refactoring effort demonstrates the complexity of addressing BSE problems and their propagation through dependency chains. The team ultimately eliminated the unmaintained *cglib* dependency while achieving compliance with strong encapsulation principles, demonstrating that internal API dependencies can be replaced with stable, documented alternatives.

7 Discussion

7.1 Implications

In this section, we emphasize the importance of developers focusing on proper modularization and API design to enhance system maintainability. For researchers, we highlight the need to improve JPMS module interface design and develop automated tools for detecting and repairing BSE violations.

For practitioners: The adoption of JPMS presents two primary challenges: designing module interfaces appropriately and utilizing other Java modules properly.

Many open-source practitioners reported BSE warnings at an early stage (Finding 1). However, they did not fix the issue until BSE warnings became errors and exceptions (Finding 2). Although Finding 8 reveals that project maintainers tend to seek more maintainable resolutions when the BSE problem and resolutions are explicitly discussed, many project developers are still opting for provisional resolutions (e.g., disable strong encapsulation enforcement) to eliminate exceptions and errors.

While disabling strong encapsulation can resolve BSE warnings, errors, or exceptions and restore program functionality (as discussed in Section 2.2 and Section 6.4), the preference for such fixes over permanent solutions impedes improving ecosystem security and robustness. As demonstrated in Figure 1, disabling strong encapsulation does not prevent error propagation throughout the ecosystem, potentially amplifying the underlying issues.

Our study further demonstrates the proliferation of technical debt (Finding 4): as the JDK increasingly enforces strong encapsulation, developers need to allocate substantial resources to identify and resolve dependency-induced defects. Given that strong encapsulation is designed to improve modularity, security, and maintainability [56], we recommend that practitioners systematically investigate the root causes of BSE violations and strictly adhere to encapsulation constraints, thereby mitigating security risks and preventing technical debt accumulation.

Our observations (Finding 6, 7 and Case 1) highlight an ongoing tension between Java module developers and module consumers. While module developers enforce encapsulation to preserve system integrity and security, module consumers frequently require access to internal APIs and non-public types—precisely the components that module developers deliberately restrict. To address this issue, we encourage module developers to adopt a systematic approach to API design that achieves an optimal balance between encapsulation and availability, thereby maintaining system reliability and security while accommodating legitimate consumer requirements.

We also propose that practitioners systematically modularize their Java codebases. Empirical evidence from Finding 3 and Finding 5 demonstrates that unnamed modules constitute the majority source of Java module abuse cases. When modules' internal APIs are exposed through `--add-exports` or `--add-opens` to unnamed modules, they substantially increase their security vulnerability surface, enabling potentially malicious code within unnamed modules to access protected internal components. Furthermore, only 19.8% of existing Java modules have descriptors with stable names [44]. Module name modifications can introduce breaking changes in dependent projects that `requires` the module, potentially triggering unexpected behavior throughout the dependency chain. This low stability rate emphasizes the importance of establishing and maintaining stable modules with declared interfaces.

Finally, we recommend that practitioners leverage collective knowledge to address BSE-related challenges by examining peer-validated solutions that achieve equivalent functionality while maintaining module integrity. By adopting community-driven solutions, developers can eliminate encapsulation problems more effectively, preventing the further spread of technical debt.

For researchers: As many BSE instances emerge only during runtime, indicated by BSE warnings, errors, or exceptions, implementing early detection through static analysis, coupled with automated repair mechanisms, could significantly aid in adopting JPMS and facilitate the overall migration process. For example, Darcy [60] demonstrates the potential for automated detection and repair of inconsistencies between implementation code (i.e., descriptive architecture) and module descriptor (i.e., prescriptive architecture). However, such tools must carefully balance strict encapsulation enforcement with ecosystem compatibility to prevent disruption of downstream dependencies. Moreover, in addition to the automation of the migration process, identifying effective strategies for modularizing Java projects presents another significant challenge. This challenge involves determining both the optimal arrangement of classes within Java modules [67] and designing module interfaces

that balance appropriately between system usability and quality attributes such as reliability, security, and maintainability.

7.2 Limitations and Threats to Validity

Internal threats. The primary internal threat to validity involves subjective bias or errors in comprehending and classifying BSE issues for the *manual analysis* of our methodology. To mitigate this risk, our open-coding process establishes classification schemes that follow the methodology adopted by the existing literature [59, 65, 70, 73]. More specifically, each GitHub issue is independently reviewed and labeled by two authors. When discrepancies between the two researchers emerged, they were resolved through discussion until consensus was achieved. Another internal threat comes from the comprehensiveness of automated BSE instance analysis. We mitigate this threat by summarizing the patterns from four thousand GitHub issues. The BSE instance detector contains all the summarized patterns we observed while examining the issue. Another internal threat arises from the number of resolution-oriented issues. While the filtering process ensures a comprehensive record that captures concrete resolution strategies, it may also bias our findings toward issues where developers or users were more proactive in documenting possible fixes. The final internal threat is concerned with the data collection process, which requires high-quality and comprehensive data. We balanced the quantity and quality of the data by only retrieving GitHub issues containing keywords that can break the strong encapsulation of Java modules, as found from the official documentation [37, 50, 52].

External threats. One external threat is the generalizability of the dataset and the findings derived. To mitigate this threat, we systematically collected all GitHub issues from popular and actively maintained open-source projects that met our selection criteria, i.e., as described in our methodology. The selected projects demonstrate significant development activity and community engagement, with mean values of 339,144 LOC, 2,311 stars, 701 forks, 1,124 issues, and 5,442 commits. Our final dataset comprises 4,079 unique GitHub issues from 1,351 distinct repositories, which is comparable to prior studies investigating other types of bugs [59, 65, 70, 94].

8 Related Work

JPMS. Several tools address distinct challenges in JPMS: OO2CB [67] transforms object-oriented designs into component-based architectures using least-privilege modularization principles. Darcy [60, 62] handles redundant dependencies through automated detection and repair. Acadia [61] enables runtime architectural adaptation without code modifications, ModGuard [55] prevents unauthorized data access via dependency analysis, and Mondal et al. [81] facilitates maintenance through semantic slicing. However, none of the existing studies on JPMS have studied the specific challenges brought forth by strong encapsulation, i.e., the BSE problem in our work.

Other Component-Based Architecture. The only existing framework similar to JPMS is OSGi [49]. OSGi differs from JPMS in many aspects: Firstly, OSGi lacks the capability to manage reflective-only access to internal packages of modules, a situation where the `--add-opens` option is frequently utilized to enable runtime access [60, 62]. Secondly, while `JPMS` emphasizes strong encapsulation at both compile-time and run-time. OSGi is limited to dynamic package resolution; this can only be determined when a bundle layer is created [48], absent a similar mechanism to `--add-export` with static access management. Consequently, these limitations invalidate OSGi from applicability within our JPMS-related research. Despite these differences, previous work [66] has highlighted issues that arose or whose importance was magnified through implementing or using the OSGi framework. For example, OSGi has inflexible package sharing, where a package can only be private or public to all bundles, unlike JPMS modules. As shown in Figure 2, `org.junit.platform.commons` can export package `org.junit.platform.commons.logging` to `org.junit.platform.engine`, which indicates the package is inaccessible to other Java modules.

Health of Java Software Ecosystems. Recent studies have advanced our understanding of Java libraries, dependency management, and the complexities of Java-based ecosystems. These investigations span various domains, including dependency conflicts [75, 96–98], library migrations [64, 68, 71, 95], compilation processes [80], and language features [53, 72, 79, 100]. They have explored security risks [77, 95, 99], the rationale behind developer actions [68, 100], and introduced frameworks [80], automated tools [75, 96, 97, 99], empirical solutions [98], and practical insights [64, 77, 95] to address these challenges. Despite this extensive research on ecosystem health, the challenge posed by JPMS—which we systematically explore in this work—remains largely unaddressed in existing literature.

9 Conclusion and Future Work

To better understand *architectural decay* in Java modules, we conducted the first comprehensive empirical investigation of the BSE problem, analyzing symptoms, source, target, and resolution in open-source projects. Our examination of 4,079 GitHub issues revealed that over 70% of BSE instances stem from external dependencies. Our findings underscore the pervasive nature of BSE problems and their impact on Java development, particularly regarding migration and maintainability. Based on our quantitative and qualitative analysis, we present recommendations for (1) practitioners implementing JPMS-based projects adhering to strong encapsulation and (2) researchers developing techniques for BSE detection and repair.

Our future work includes an interview study exploring developers' experiences with BSE problems, aiming to uncover challenges, solutions, and underlying rationales. This qualitative investigation will complement our current findings by providing deeper insights into practitioners' perspectives and contextual factors. This investigation will contribute to the understanding of BSE issues and facilitate the development of generalizable solutions to benefit the Java community.

Acknowledgments

We sincerely appreciate Prof. André van der Hoek and Prof. Sam Malek for their constructive feedback and support. We are also thankful to the anonymous reviewers and others who provided insightful and valuable comments. The authors gratefully acknowledge the support of NSF 2443763.

References

[1] 2023. GitHub: Let's build from here — github.com. https://github.com. (Accessed 20-09-2023).

[2] 2024. Artifact of Breaking Strong Encapsulation: A Comprehensive Study of Java Module Abuse. https://figshare.com/s/8031e14136fd008c4840,.

[3] 2024. Cannot work with JDK 16? #178. https://github.com/xvik/dropwizard-guicey/issues/178. (Accessed on 13-03-2024).

[4] 2024. ClassLoader. https://github.com/openjdk/jdk/blob/62a4544bb76aa339a8129f81d2527405a1b1e7e3/src/java.base/share/classes/java/lang/ClassLoader.java. (Accessed on 04-10-2024).

[5] 2024. Cleaner. https://github.com/AdoptOpenJDK/openjdk-jdk9u/blob/master/jdk/src/java.base/share/classes/jdk/internal/ref/Cleaner.java. (Accessed 04-02-2024).

[6] 2024. Code on the Class Path - the Unnamed Module. https://dev.java/learn/modules/unnamed-module/. (Accessed 04-02-2024).

[7] 2024. Commit Import #1298. https://github.com/google/guice/commit/85e30beafe55551a649025d3c12e831214057412. (Accessed on 13-03-2024).

[8] 2024. [fixes #2681] [jdk16] support jdk16. https://github.com/projectlombok/lombok/commit/9806e5cca4b449159ad0509dafde81951b8a8523. (Accessed on 13-03-2024).

[9] 2024. GitHub REST API documentation. https://docs.github.com/en/rest. (Accessed 30-01-2024).

[10] 2024. google-java-format. https://github.com/google/google-java-format. (Accessed 30-01-2024).

[11] 2024. google-java-format (and removeUnusedImports) broken on JDK 16+ (has workaround) #834. https://github.com/diffplug/spotless/issues/834. (Accessed 30-01-2024).

[12] 2024. Guice. https://github.com/google/guice. (Accessed on 13-03-2024).

[13] 2024. Illegal reflective access by com.google.inject.internal.cglib.core.$ReflectUtils$1. https://github.com/google/guice/issues/1133. (Accessed on 13-03-2024).

[14] 2024. Introduction to the Service Provider Interfaces. https://docs.oracle.com/javase/tutorial/sound/SPI-intro.html. (Accessed on 8-09-2024).

[15] 2024. issue #3099. https://github.com/square/retrofit/issues/3900. (Accessed on 04-10-2024).

[16] 2024. issue #4694. https://github.com/oracle/graal/issues/4694. (Accessed on 04-10-2024).

[17] 2024. JAR. https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html. (Accessed on 04-10-2024).

[18] 2024. Java 9 Migration Guide: The Seven Most Common Challenges. https://nipafx.dev/java-9-migration-guide/. (Accessed 30-01-2024).

[19] 2024. Java Dependency Analysis Tool. https://wiki.openjdk.org/display/JDK8/Java+Dependency+Analysis+Tool#JavaDependencyAnalysisTool-ReplaceusesoftheJDK'sinternalAPIs. (Accessed on 24-09-2024).

[20] 2024. Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/javase/9/tools/java.htm#JSWOR624. (Accessed 01-02-2024).

[21] 2024. Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/javase/10/tools/java.htm#JSWOR624. (Accessed 01-02-2024).

[22] 2024. Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/en/java/javase/16/docs/specs/man/java.html. (Accessed 01-02-2024).

[23] 2024. Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/en/java/javase/17/docs/specs/man/java.html. (Accessed 01-02-2024).

[24] 2024. Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE. (Accessed 01-02-2024).

[25] 2024. Java SE at a Glance. https://www.oracle.com/java/technologies/java-se-glance.html. (Accessed 02-02-2024).

[26] 2024. javaguide. https://google.github.io/styleguide/javaguide.html. (Accessed 30-01-2024).

[27] 2024. java.io Package Summary. https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html. (Accessed on 04-10-2024).

[28] 2024. java.lang Package Summary. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/package-summary.html. (Accessed on 04-10-2024).

[29] 2024. java.nio.Buffer. https://github.com/openjdk/jdk/blob/62a4544bb76aa339a8129f81d2527405a1b1e7e3/src/java.base/share/classes/java/nio/Buffer.java. (Accessed on 04-10-2024).

[30] 2024. java.util Package Summary. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/package-summary.html. (Accessed on 04-10-2024).

[31] 2024. jdk/src/java.base/share/classes/module-info.java. https://github.com/openjdk/jdk/blob/55c1446b68db6c4734420124b5f26278389fdf2b/src/java.base/share/classes/module-info.java#L77. (Accessed 04-02-2024).

[32] 2024. JUnit5. https://github.com/junit-team/junit5. (Accessed 04-02-2024).

[33] 2024. junit5/gradle.properties. https://github.com/junit-team/junit5/blob/46d0f80db0d6fc5faced28e9827683a09e7f8fb9/gradle.properties. (Accessed 30-01-2024).

[34] 2024. MethodHandlesLookup. https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/lang/invoke/MethodHandles.java. (Accessed on 04-10-2024).

[35] 2024. Module java.base. https://download.java.net/java/early_access/valhalla/docs/api/java.base/module-summary.html. (Accessed 02-02-2024).

[36] 2024. Object. https://github.com/openjdk/jdk/blob/62a4544bb76aa339a8129f81d2527405a1b1e7e3/src/java.base/share/classes/java/lang/Object.java. (Accessed on 04-10-2024).

[37] 2024. Oracle JDK Migration Guide. https://www.overleaf.com/project/656ff56130c3fa4c526b8808. (Accessed 30-01-2024).

[38] 2024. A peek into Java 17: Encapsulating the Java runtime internals. https://blogs.oracle.com/javamagazine/post/a-peek-into-java-17-continuing-the-drive-to-encapsulate\-the-java-runtime-internals. (Accessed 31-01-2024).

[39] 2024. Possible to use sun.misc.Unsafe to hack JDK and circumvent JEP 396. https://bugs.openjdk.org/browse/JDK-8264582?attachmentViewMode=list, (Accessed on 13-03-2024).

[40] 2024. Prototype CGLIB replacement (Issue 1133). https://github.com/google/guice/pull/1298. (Accessed on 13-03-2024).

[41] 2024. ReflectUtils.java. https://github.com/cglib/cglib/blob/9d67875290d269c9b1ff5e4f4bc578a9f05c392e/cglib/src/main/java/net/sf/cglib/core/ReflectUtils.java#L55. (Accessed on 13-03-2024).

[42] 2024. removeUnusedImports fails on Java 17 #871. https://github.com/diffplug/spotless/issues/871. (Accessed 30-01-2024).

[43] 2024. Security. https://github.com/AdoptOpenJDK/openjdk-jdk9u/blob/master/jdk/src/java.base/share/classes/sun/security/util/Cache.java. (Accessed on 04-10-2024).

[44] 2024. sormuras/modules. https://github.com/sormuras/modules. (Accessed on 08-03-2024).

[45] 2024. Spotless: Keep your code spotless. https://github.com/diffplug/spotless. (Accessed 30-01-2024).

[46] 2024. Stack Overflow. https://stackoverflow.com. (Accessed on 04-10-2024).

[47] 2024. TreeMap. https://github.com/openjdk/jdk/blob/62a4544bb76aa339a8129f81d2527405a1b1e7e3/src/java.base/share/classes/java/util/TreeMap.java. (Accessed on 04-10-2024).

[48] n.d.. Eclipse Equinox with Java Modules All the Way Down. https://www.eclipse.org/community/eclipse_newsletter/2016/october/article3.php. (Accessed on 30-06-2024).

[49] n.d.. OSGI Alliance. https://docs.osgi.org/specification/. (Accessed on 30-06-2024).

[50] Alan Bateman, Alex Buckley, Jonathan Gibbons, and Mark Reinhold. 2023. JEP 261: Module System — openjdk.org. https://openjdk.org/jeps/261. (Accessed on 29-06-2023).

[51] Alex Buckley and Mark Reinhold. 2023. JEP 396: Strongly Encapsulate JDK Internals by Default — openjdk.org. https://openjdk.org/jeps/396. (Accessed on 29-06-2023).

[52] Alex Buckley and Mark Reinhold. 2023. JEP 403: Strongly Encapsulate JDK Internals — openjdk.org. https://openjdk.org/jeps/403. (Accessed on 29-06-2023).

[53] Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. 2015. Assert use in github projects. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 755–766.

[54] John Cresswell. 2013. Qualitative inquiry & research design: Choosing among five approaches.

[55] Andreas Dann, Ben Hermann, and Eric Bodden. 2019. ModGuard: Identifying Integrity & Confidentiality Violations in Java Modules. *IEEE Transactions on Software Engineering* 47, 8 (2019), 1656–1667.

[56] Paul Deitel. 2017. Understanding Java 9 Modules. https://www.oracle.com/corporate/features/understanding-java-9-modules.html. (Accessed on 12-06-2023).

[57] Ben Evans. 2020. The Unsafe Class: Unsafe at Any Speed. https://blogs.oracle.com/javamagazine/post/the-unsafe-class-unsafe-at-any-speed. (Accessed 31-01-2024).

[58] David Farley. 2021. *Modern Software Engineering: Doing What Works to Build Better Software Faster.* Addison-Wesley Professional.

[59] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 385–396.

[60] Negar Ghorbani, Joshua Garcia, and Sam Malek. 2019. Detection and repair of architectural inconsistencies in java. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 560–571.

[61] Negar Ghorbani, Joshua Garcia, and Sam Malek. 2024. Bringing architecture-based adaption to the mainstream. *Information and Software Technology* 176 (2024), 107550.

[62] Negar Ghorbani, Tarandeep Singh, Joshua Garcia, and Sam Malek. 2024. DARCY: Automatic Architectural Inconsistency Resolution in Java. *IEEE Transactions on Software Engineering* (2024).

[63] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. 2017. The Java® Language Specification — docs.oracle.com. https://docs.oracle.com/javase/specs/jls/se9/html/index.html. (Accessed on 30-06-2023).

[64] Haiqiao Gu, Hao He, and Minghui Zhou. 2023. Self-Admitted Library Migrations in Java, JavaScript, and Python Packaging Ecosystems: A Comparative Study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 627–638.

[65] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A comprehensive study of real-world bugs in machine learning model optimization. In *Proceedings of the International Conference on Software Engineering*.

[66] Richard S Hall and Humberto Cervantes. 2004. An OSGi implementation and experience report. In *First IEEE Consumer Communications and Networking Conference, 2004. CCNC 2004.* IEEE, 394–399.

[67] Mahmoud M Hammad, Ibrahim Abueisa, and Sam Malek. 2022. Tool-Assisted Componentization of Java Applications. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE, 36–46.

[68] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A large-scale empirical study on Java library migrations: prevalence, trends, and rationales. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 478–490.

[69] Philipp Holzinger and Eric Bodden. 2021. A Systematic Hardening of Java's Information Hiding. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*. 11–22.

[70] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1110–1121.

[71] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. Understanding the impact of APIs behavioral breaking changes on client applications. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1238–1261.

[72] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. 2020. An Empirical Study on the Use and Misuse of Java 8 Streams.. In *FASE*. 97–118.

[73] Hao Li, Filipe R Cogo, and Cor-Paul Bezemer. 2022. An empirical study of yanked releases in the Rust package registry. *IEEE Transactions on Software Engineering* 49, 1 (2022), 437–449.

[74] Shuqing Li, Cuiyun Gao, Jianping Zhang, Yujia Zhang, Yepang Liu, Jiazhen Gu, Yun Peng, and Michael R Lyu. 2024. Less Cybersickness, Please: Demystifying and Detecting Stereoscopic Visual Inconsistencies in Virtual Reality Apps. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2167–2189.

[75] Christian Macho, Fabian Oraze, and Martin Pinzger. 2024. DValidator: An approach for validating dependencies in build configurations. *Journal of Systems and Software* 209 (2024), 111916.

[76] Sander Mak and Paul Bakker. 2017. *Java 9 modularity: patterns and practices for developing maintainable applications.* " O'Reilly Media, Inc.".

[77] Fabio Massacci and Ivan Pashchenko. 2021. Technical leverage in a software ecosystem: Development opportunities and security risks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1386–1397.

[78] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at your own risk: The java unsafe api in the wild. *ACM Sigplan Notices* 50, 10 (2015), 695–710.

[79] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the use of lambda expressions in Java. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–31.

[80] Md Rakib Hossain Misu, Rohan Achar, and Cristina V Lopes. 2023. SourcererJBF: A Java Build Framework For Large-Scale Compilation. *ACM Transactions on Software Engineering and Methodology* (2023).

[81] Amit Kumar Mondal, Chanchal K Roy, Kevin A Schneider, Banani Roy, and Sristy Sumana Nath. 2021. Semantic slicing of architectural change commits: Towards semantic design review. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–6.

[82] Cliodhna O'Connor and Helene Joffe. 2020. Intercoder Reliability in Qualitative Research: Debates and Practical Guidelines. *International Journal of Qualitative Methods* 19 (2020), 1609406919899220. doi:10.1177/1609406919899220 arXiv:https://doi.org/10.1177/1609406919899220

[83] Nicolai Parlog. 2015. Project Jigsaw is Really Coming in Java 9 — infoq.com. https://www.infoq.com/articles/Project-Jigsaw-Coming-in-Java-9/. (Accessed on 12-06-2023).

[84] Nicolai Parlog. 2019. *The Java Module System.* Simon and Schuster.

[85] Dewayne E Perry and Alexander L Wolf. 1992. Foundations for the study of software architecture. *ACM SIGSOFT SEN* (1992).

[86] Dewayne E Perry and Alexander L Wolf. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4 (1992), 40–52.

[87] Ron Pressler and Alex Buckley. 2023. JEP 451: Prepare to Disallow the Dynamic Loading of Agents — openjdk.org. https://openjdk.org/jeps/451. (Accessed on 12-06-2023).

[88] Ron Pressler and Alex Buckley. 2023. JEP draft: Integrity and Strong Encapsulation — openjdk.org. https://openjdk.org/jeps/8305968. (Accessed on 07-06-2023).

[89] Mark Reinhold. 2017. JEP 200: The Modular JDK — openjdk.org. https://openjdk.org/jeps/200. (Accessed on 12-06-2023).

[90] Mark Reinhold. 2023. JEP 260: Encapsulate Most Internal APIs — openjdk.org. https://openjdk.org/jeps/260. (Accessed on 29-06-2023).

[91] Mary Shaw and David Garlan. 1996. *Software architecture: perspectives on an emerging discipline*. Vol. 1. Prentice Hall Englewood Cliffs.

[92] R.N. Taylor, N. Medvidovic, and Dashofy E.M. 2009. *Software Architecture: Foundations, Theory, and Practice.* John Wiley & Sons.

[93] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. 2009. *Software architecture: foundations, theory, and practice.* Wiley Publishing.

[94] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. 2021. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 20–31.

[95] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45.

[96] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 319–330.

[97] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2021. Will dependency conflicts affect my program's semantics? *IEEE Transactions on Software Engineering* 48, 7 (2021), 2295–2316.

[98] Yongzhi Wang, Chengli Xing, Jinan Sun, Shikun Zhang, Sisi Xuanyuan, and Long Zhang. 2020. Solving the Dependency Conflict of Java Components: A Comparative Empirical Analysis. In *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*. IEEE, 109–114.

[99] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. 2023. Software composition analysis for vulnerability detection: An empirical study on Java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 960–972.

[100] Mingwei Zheng, Jun Yang, Ming Wen, Hengcheng Zhu, Yepang Liu, and Hai Jin. 2021. Why do developers remove lambda expressions in Java?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 67–78.