applied sciences

MDPI

Article

Automatic Refactoring Approach for Asynchronous Mechanisms with CompletableFuture

Yang Zhang [1], Zhaoyang Xie [1], Yanxia Yue [2,*] and Lin Qi [1]

[1] School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050000, China; zhangyang@hebust.edu.cn (Y.Z.)
[2] Shijiazhuang Meteorological Bureau, Shijiazhuang 050000, China
* Correspondence: yyanxia55@163.com

Abstract: To address the inherent limitations of Future in asynchronous programming frameworks, JDK 1.8 introduced the CompletableFuture class, which features approximately 50 different methods for composing and executing asynchronous computations and handling exceptions. This paper proposes an automatic refactoring method that integrates multiple static analysis techniques, including visitor pattern analysis, alias analysis, and executor inheritance structure analysis, to conduct precondition checks. Distinct from existing Future refactoring methods, this approach considers custom executor types, thereby extending its applicability. Using this method, the ReFuture automatic refactoring plugin was implemented within the Eclipse JDT framework. The method was evaluated in terms of the number of refactorings, refactoring time, and error introduction, alongside a side-by-side comparison with the existing method. The refactoring outcomes for nine large applications, including ActiveMQ, Hadoop, and Elasticsearch, show that ReFuture successfully refactored 639 out of 813 potential code structures, achieving a refactoring success rate of 64.70% without introducing errors. This tool effectively facilitates the refactoring to CompletableFuture and enhances refactoring efficiency compared to manual methods.

Keywords: automatic refactoring; Java; CompletableFuture; static analysis

Citation: Zhang, Y.; Xie, Z.; Yue, Y.; Qi, L. Automatic Refactoring Approach for Asynchronous Mechanisms with CompletableFuture. *Appl. Sci.* **2024**, *14*, 8866. https://doi.org/10.3390/app14198866

Academic Editor: Christos Bouras

Received: 28 August 2024
Revised: 26 September 2024
Accepted: 28 September 2024
Published: 2 October 2024

1. Introduction

Java has supported asynchronous programming since its earliest versions, with continuous enhancements to its asynchronous capabilities as technology has evolved. A significant advancement came with the introduction of the java.util.concurrent (JUC) library in Java 5, which markedly improved asynchronous programming. A key enhancement in this library is the ExecutorService framework. This framework decouples the lifecycle management of threads from task execution and avoids the performance overhead associated with the frequent creation and destruction of threads by reusing them. It allows asynchronous tasks that implement the Callable or Runnable interfaces to be submitted for execution to an executor while returning a Future object as a handle for the asynchronous task. Through this handle, developers can access the task's execution status and results. We refer to this mode of asynchronous programming as the "traditional asynchronous construct". It has been well-received and widely used by developers, ranking as the second most popular asynchronous programming pattern after Thread, according to our empirical research results presented in Section 4.

As time has progressed, some challenges associated with using traditional asynchronous constructs in real-world applications have sparked extensive discussions in developer forums [1,2], particularly the lack of phased control in the ordinary Future returned by traditional constructs, which makes orchestrating asynchronous tasks exceptionally difficult. The introduction of the CompletableFuture class in Java 8's JUC addresses these issues. CompletableFuture implements the Future interface and extends the CompletionStage

interface. This extension provides callback mechanisms and chainable method invocations to compose and orchestrate asynchronous computations. Additionally, it integrates functional programming interfaces and lambda expressions, enhancing the readability of asynchronous code. It also introduces an exception-handling mechanism, simplifying exception management during asynchronous task execution. However, according to the empirical research results presented in Section 4 of this paper, only 3.53% of popular Java open-source projects on GitHub have adopted CompletableFuture. On one hand, this is related to the relatively late introduction of CompletableFuture. Traditional asynchronous constructs had already become deeply ingrained, and the existence of similar alternatives to CompletableFuture in popular asynchronous libraries within the Java community, such as ListenableFuture in Google's Java core library Guava [3], has affected the popularity of CompletableFuture. On the other hand, due to CompletableFuture's rich API, which results in a higher learning curve for programmers, many have missed the opportunity to utilize it.

Although refactoring traditional asynchronous constructs to CompletableFuture constructs can provide software developers with more powerful and flexible asynchronous programming capabilities, as discussed in developer forums [2], this transition is not without barriers. Firstly, there needs to be a method to assess whether the logic for submitting asynchronous tasks via executors is compatible with CompletableFuture. Various executor types in the JUC library meet refactoring criteria, but developers often use custom ExecutorService interfaces. This requires analyzing non-JUC executor types in projects to identify which are refactorable. Secondly, because CompletableFuture uses Supplier-type asynchronous tasks with return values, it is necessary to refactor Callable-type asynchronous tasks into Supplier-type. Furthermore, the cancellation mechanism of CompletableFuture differs from that of FutureTask, the Future object type used traditionally. This difference can cause issues if tasks depend on FutureTask's cancellation feature. Lastly, refactoring changes the actual type of Future objects; if the code includes type comparisons or type conversions of Future objects, refactoring should be avoided. Considering that manual refactoring is labor-intensive, time-consuming, and prone to errors, the application of automated refactoring tools becomes particularly important. Through automation, we can more efficiently promote the migration of existing code to CompletableFuture, thereby fully leveraging its advanced asynchronous programming features.

In recent years, numerous studies have been conducted in the field of automated refactoring related to asynchronous constructs. Previous research has concentrated on (1) refactoring current constructs into more suitable alternatives [4–8] and (2) embedding asynchronous constructs within sequential code to enhance execution efficiency [9–15]. Our method expands the first area by diversifying asynchronous refactoring techniques, introducing a novel approach to transform traditional asynchronous constructs into CompletableFuture. Moreover, our strategy to identify non-JUC internal asynchronous constructs offers significant insights.

To address the aforementioned issues, this paper proposes an automated refactoring method for the CompletableFuture asynchronous mechanism. By employing program analysis techniques such as visitor pattern analysis, executor inheritance structure analysis, and points-to analysis, we identify refactorable executor types within software projects poised for refactoring. Based on these analyses, we apply refactoring templates to automatically transform four types of traditional asynchronous constructs into CompletableFuture constructs. Utilizing the Eclipse JDT framework [16] and Soot [17], an automated refactoring tool, ReFuture, has been implemented in the form of an Eclipse plugin. In experiments, ReFuture was evaluated based on the number of refactorings, the correctness of refactorings, and the refactoring time. The results demonstrate that among 813 instances of traditional asynchronous constructs in nine large-scale open-source projects, ReFuture successfully refactored 639 instances without introducing errors into the refactored programs. This tool effectively facilitates the automatic refactoring from traditional asynchronous constructs to CompletableFuture.

The main contributions of this paper are as follows.

- Empirical research: We investigate the use of common asynchronous constructs in current open-source Java projects and analyze the complexity of asynchronous tasks in projects using traditional asynchronous constructs and CompletableFuture constructs.
- Refactoring tool: We present an automated refactoring method from traditional asynchronous constructs to CompletableFuture constructs, specifically addressing executor types that exist outside of the JUC. We propose analysis methods from the perspectives of inheritance and composition, which are common in object-oriented program reuse, thereby enhancing the applicability of our refactoring method. Based on the Eclipse platform [18], we implement the ReFuture automatic refactoring tool, which has been made open-source [19].
- Experimental evaluation: We assess the applicability, correctness, and efficiency and conduct a side-by-side comparison with the existing method, all through answering several research questions.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the motivation of this study through an example. Section 4 presents the empirical research questions and results. Section 5 details the automated refactoring method proposed in this paper. Section 6 describes the experimental setup and evaluation results. The paper concludes with a summary of the entire work.

2. Related Work

In this section, we first introduce concurrency-related refactoring techniques in the Java domain, then extend the discussion to research on concurrency refactoring in other platforms, and finally discuss the connections between our approach and the related work.

A survey revealed that advanced concurrency libraries have not been widely applied [20]. Researchers hope to enhance the quality of existing code through software refactoring techniques, aiming for better utilization of the JUC library. Dig and colleagues [9] introduced an automatic refactoring method and tool, Concurrencer, which uses JUC to refactor sequential code into concurrent code, thereby speeding up program execution. They proposed three types of refactorings: transforming synchronized locks into atomic locks, refactoring HashMap into ConcurrentHashMap, and using the Fork/Join framework for recursive tasks. Subsequently, Ishizaki and others [10] focused on the first two refactorings mentioned above and proposed a refactoring method that combines interprocedural points-to analysis, achieving better results. In studies related to locks in the JUC library, Zhang and his team performed a series of works where they proposed [11] an automated transformation method that combines side-effect analysis and other static analysis techniques at the bytecode level to refactor synchronized locks into ReentrantLock and ReentrantReadWriteLock. Later, they [12] continued their in-depth research and proposed five regular expressions to represent refactoring rules, which can refactor synchronized locks into fine-grained ReentrantReadWriteLocks. In terms of automatically refactoring sequential code into concurrent code, Midolo and colleagues [13] conducted further research by statically assessing the number of control flow graph nodes in the bodies of two invoked methods in sequential code; if the counts are similar and there are no dependencies, the two method invocations in the sequential code are submitted to a thread pool for asynchronous execution. As reactive programming concepts gained popularity, Köhler and colleagues [4] refactored SwingWorker and ExecutorService constructs to support the RxJava reactive programming library. Addressing the limitations of statically identifying inefficient concurrency patterns, Orton and others [5] combined dynamic analysis to find opportunities to refactor Executor pattern asynchronous code by inlining into synchronous code and using CompletableFuture compositions to release coordinator threads, thus improving the quality of asynchronous code.

In research related to concurrency refactoring outside the Java platform, in the C# domain, Okur and others [6] first explored the use of parallel abstractions in C# open-source programs, then presented a refactoring method and tool to transform lower-level

parallel abstractions into higher-level ones, specifically converting Thread and ThreadPool abstractions into Task abstractions and optimizing the use of the Task abstraction. They later [7] researched refactoring C# code from callback-based asynchronous code to using async/await, implementing a refactoring tool. On the Android platform, Lin and colleagues [14] first conducted empirical research on the Android corpus, summarizing the underuse of AsyncTask constructs, leading to the proposal of an automatic refactoring method that extracts long-running operations into AsyncTask. Following this, he [8] addressed the misuse issues of AsyncTask, proposing a method and refactoring tool ASYNCDROID to transform incorrect use of AsyncTask into IntentServices constructs. On the JavaScript platform, Gokhale and others [15] proposed a method and tool using the async/await keywords combined with static analysis to migrate synchronous APIs to asynchronous APIs. Arteca and colleagues [21] demonstrated that there is room for optimization in the ordering of asynchronous I/O operations in existing JavaScript code, using static side-effect analysis to refactor asynchronous I/O operations to enhance program performance.

To better summarize the related work, as shown in Table 1, we present a comparison of studies on concurrency refactoring.

Table 1. Comparison of Concurrency Refactoring Studies.

Study	Platform	Supported Constructs	Refactored Constructs	Sync to Async
Dig et al. [9]	Java	Synchronized lock, HashMap, Sequential recursive	Atomic lock, ConcurrentHashMap, Fork/Join framework	Yes
Okur et al. [6]	C#	Thread, ThreadPool	Task	No
Okur et al. [7]	C#	Callback-based asynchronous code	async, await	No
Lin et al. [14]	Android	Long-running code	AsyncTask	Yes
Lin et al. [8]	Android	AsyncTask	IntentService	No
Gokhale et al. [15]	JavaScript	Synchronous API functions	async, await	Yes
Midolo et al. [13]	Java	Sequential method calls	CompletableFuture	Yes
Köhler et al. [4]	Java	Future, SwingWorker	RxJava constructs	No
Orton et al. [5]	Java	Traditional asynchronous constructs	Inline code, CompletableFuture	No

In Table 1, we initially showcase the study by Dig et al. [9] as it represents a significant work in the field of concurrency refactoring. This study aims to enhance code efficiency by automatically transforming sequential/synchronous code into concurrency/asynchronous code. Subsequently, the research by Okur et al. [6,7] focuses on refactoring concurrency programs that use outdated APIs to utilize updated APIs for code optimization. Lin et al.'s study [8,14] addresses both these aspects. The research by Gokhale et al. [15] involves transforming synchronous APIs in JavaScript into asynchronous ones using the async/await keywords to boost performance.

The constructs of the refactorings in the last three rows of the table are related to those in our study. The research by Midolo et al. [13] resulted in refactoring that utilizes the CompletableFuture construct to asynchronously execute the original synchronous code. Köhler et al. [4] conducted refactoring of both the SwingWorker and ExecutorService frameworks to support the RxJava reactive programming library, using RxJava's Observable to encapsulate the Future returned by traditional asynchronous constructs. However, their approach is limited to executor types within the JUC library, thus missing opportunities for broader refactoring. Orton et al. [5] explored using dynamic analysis to identify opportunities to enhance program performance through refactoring traditional asynchronous constructs. They discuss replacing a pattern where asynchronous tasks submitted to an executor are merely scheduling two other asynchronous tasks with CompletableFuture's combinational capabilities to reduce scheduling overhead. However, their research focuses more on dynamic analysis and code patch generation technologies. The proposed refactoring pattern

involves transitioning from traditional asynchronous constructs to CompletableFuture, but it does not provide specific methods for the refactoring.

Previous studies have primarily focused on two areas: (1) refactoring existing asynchronous constructs to more suitable alternatives [4–8] and (2) introducing asynchronous constructs into sequential code to improve program execution efficiency [9–15]. Our approach builds upon the first area by increasing the diversity of asynchronous refactoring methods, introducing a new method for refactoring traditional asynchronous constructs into CompletableFuture. Additionally, our approach for identifying non-JUC internal asynchronous construct types can provide valuable insights into expanding the applicability of automated refactoring methods.

3. Motivation

To visually demonstrate the advantages of CompletableFuture over traditional asynchronous constructs in asynchronous code, we illustrate this through a code example adapted from a classic concurrent programming case provided in Section 6.3 of "Java Concurrency in Practice" [22]. Our example code primarily involves two tasks: first, fetching images from a remote server based on a provided list of image information, and second, decrypting the fetched images, both performed asynchronously. In this section, we first show the shortcomings of traditional asynchronous constructs when handling multiple dependent asynchronous tasks, followed by a demonstration of how transitioning to CompletableFuture constructs allows developers to use its rich API to make asynchronous code concise and clear while also enhancing program performance.

In traditional asynchronous constructs, as shown in Figure 1, line 3 uses the submit method to submit getImageFile() to a download thread pool suitable for IO-intensive operations and returns a Future object, thus obtaining a List<Future<ImageFile>> type list called imageFileFutures. Subsequently, the fetched images are decrypted; line 7 uses the get method of the Future interface to synchronously block and obtain the results of the IO tasks, which are then submitted for decryption in the decryption thread pool at line 9.

```
1 public static void processImage(List<ImageInfo> imageInfos) throws InterruptedException, ExecutionException {
2     List<Future<ImageFile>> imageFileFutures = imageInfos.stream()
3     .map(imageInfo-> imageDownloadExecutorService.submit(()->getImageFile(imageInfo)))
4     .collect(Collectors.toList());
5     List<Future<DecryptedImageFile>> decryptedImageFileFutures = new ArrayList<>();
6     for(Future<ImageFile> imageFileFuture:imageFileFutures) {
7         ImageFile imageFile = imageFileFuture.get();
8         decryptedImageFileFutures.add(
9         imageDecryptionExecutorService.submit(()->decryptImageFile(imageFile)));
10    }
11 }
```

Figure 1. Remote Acquisition and Decryption of Images.

This asynchronous approach has two drawbacks. In terms of code readability, the asynchronous logic (decrypting images depends on the completion of the image fetching task) is not evident in the code of Figure 1. Consider if there were more than these two asynchronous tasks; the complexity of dependencies between these tasks would significantly increase, further reducing the readability of asynchronous code and complicating future maintenance. Performance-wise, interaction between two or more asynchronous tasks requires the main thread to block and wait, as shown in Figure 1, line 7. During the waiting period for one asynchronous task in imageFileFutures, even if other IO asynchronous tasks are completed, they cannot be promptly submitted to the decryption tasks, failing to fully leverage the advantages of asynchronous programming. In other words, blocking waiting is a synchronous operation that affects asynchronous efficiency.

After transitioning from traditional asynchronous constructs to CompletableFuture, the changes to the code are minimal. As shown in Figure 2, unlike Figure 1, the modifications occur in lines 3 and 9, where developers can easily understand these changes.

The ordinary Future initially returned by traditional asynchronous constructs has been converted to CompletableFuture, allowing developers to utilize its extensive API.

```
1 public static void processImage(List<ImageInfo> imageInfos) throws InterruptedException, ExecutionException {
2     List<Future<ImageFile>> imageFileFutures = (List<Future<ImageFile>>) imageInfos.stream()
3     .map(imageInfo-> CompletableFuture.supplyAsync(() -> getImageFile(imageInfo), imageDownloadExecutorService))
4     .collect(Collectors.toList());
5     List<Future<DecryptedImageFile>> decryptedImageFileFutures = new ArrayList<>();
6     for(Future<ImageFile> imageFileFuture:imageFileFutures) {
7         ImageFile imageFile = imageFileFuture.get();
8         decryptedImageFileFutures.add(
9         CompletableFuture.supplyAsync(() -> decryptImageFile(imageFile), imageDecryptionExecutorService));
10    }
11 }
```

Figure 2. Traditional asynchronous remote acquisition and decryption of images.

After a comprehensive assessment of the software project, if developers determine that removing the synchronous blocking get() method invocations will not affect the program's normal operation, they can switch to completely asynchronous, streamlined code. As shown in Figure 3, developers use the CompletableFuture API to orchestrate these two interdependent asynchronous tasks. At line 4, they first use the static method supplyAsync() to submit the task of fetching images to the corresponding thread pool executor. Then, at line 5, the member method thenApplyAsync() is used to submit the decryption task to the corresponding thread pool executor. This demonstrates that CompletableFuture not only makes the dependencies between asynchronous tasks very clear but also allows interdependent asynchronous tasks to be executed completely asynchronously, overcoming the drawbacks of traditional asynchronous constructs.

```
1 public static void processImage(List<ImageInfo> imageInfos) {
2     List<CompletableFuture<DecryptedImageFile>> futures = imageInfos.stream().map(
3             imageInfo -> CompletableFuture
4             .supplyAsync(()->getImageFile(imageInfo),imageDownloadExecutorService)
5             .thenApplyAsync(imageFile->decryptImageFile(imageFile),imageDecryptionExecutorService))
6             .collect(Collectors.toList());
7 }
```

Figure 3. CompletableFuture asynchronous remote acquisition and decryption of images.

To assess the impact of fully asynchronous execution of multiple asynchronous tasks with dependencies on program performance, we manually refactored the test case RunTest.timezoneOfID from the open-source project Jenkins and conducted measurements using the JMH benchmarking tool with 10 warm-ups followed by 10 measurements. The test results showed that the call time was reduced from 0.310 ms to 0.282 ms, enhancing execution speed by 9.03%.

From the above code example, it can be seen that refactoring traditional asynchronous constructs into the CompletableFuture form allows asynchronous code to replace the synchronous method get() with APIs provided by CompletableFuture during future maintenance. This not only enhances program performance but also improves the readability of asynchronous code.

4. Empirical Research

We conducted empirical research on asynchronous mechanisms to understand the current state of practice of asynchronous constructs like CompletableFuture in actual Java programs. This helps us to assess the potential opportunities for refactoring to CompletableFuture. On the other hand, we want to explore whether the CompletableFuture construct helps programmers write less complex asynchronous task code. Therefore, we addressed the following two research questions, introducing the corpus and methods first and then presenting the results for each question.

RQ1: How are the commonly used asynchronous executor APIs currently utilized in Java?

Through this research question, we aim to understand the usage of traditional asynchronous constructs and CompletableFuture constructs to assess the potential for our refactoring work.

The literature [4] has documented the use of asynchronous executor-related constructs; however, the dataset used was compiled in 2015 and may not accurately reflect the current state of open-source software implementations, particularly the use of the CompletableFuture construct introduced with Java in 2014. Using the same method as in the literature, we added 4 constructs to the 14 types of asynchronous executor constructs they analyzed. As shown in Table 2, we surveyed the use of 18 different asynchronous APIs. Among these, Executors serve as thread pool factory classes provided by JUC, FutureTask is a basic implementation class of the Future interface, the @Async annotation facilitates easy asynchronous method execution within the Spring framework, ListenableFuture and AsyncResult support callback-based asynchronous patterns, Publisher and Observable represent the reactive programming paradigm facilitating asynchronous data flow processing, SwingWorker and Task handle time-consuming tasks in Swing and JavaFX, respectively, without blocking the UI thread, AsyncResponse is used for asynchronous response handling in web services, and ManagedTask, Vert.x Future, and Akka Futures provide solutions for asynchronous operations in enterprise-level concurrent programming and event-driven architectures.

Table 2. Asynchronously constructed statistics.

Asynchronous Constructs	Count	%
java.lang.Thread	27,489	30.07
java.util.concurrent.Executors	13,896	15.20
java.util.concurrent.ExecutorService	11,664	12.76
java.util.concurrent.Future	7147	7.82
java.utilconcurrent.CompletableFuture	3229	3.53
java.util.concurrent.FutureTask	1833	2.01
org.springframework.scheduling.annotation.Async	1104	1.21
com.google.common.util.concurrent.ListenableFuture	839	0.92
org.reactivestreams.Publisher	739	0.81
javax.swing.SwingWorker	613	0.67
javafx.concurrent.Task	377	0.41
java.util.concurrent.ForkJoinTask	333	0.36
io.vertx.core.Future	262	0.29
javax.ws.rs.container.AsyncResponse	254	0.17
io.reactivex.rxjava3.core.Observable	111	0.12
javax.ejb.AsyncResult	42	0.05
akka.dispatch.Futures	34	0.04
javax.enterprise.concurrent.ManagedTask	34	0.04
Projects with asynchronous constructs	34,492	37.74
Total number of projects	91,403	100.00

Corpus-1. Similar to the literature [4], we utilized BOA [23] to analyze open-source Java projects. BOA is a language and infrastructure designed for mining software repositories. We used its latest Java dataset ("2022 Jan/Java" in BOA), which includes 91,403 projects on GitHub with Java as the primary language (the highest proportion of Java code) and has been converted into AST (Abstract Syntax Tree).

Method. Using the BOA language, we accessed the AST of projects in the dataset through the visitor pattern to determine if the Import section contains any of the asynchronous APIs listed in Table 2. For Threads, since they belong to the java.lang package and do not require an import to be used, we traversed all Type nodes within the AST.

Result. Table 2 presents the usage statistics of the asynchronous APIs. Out of the 91,403 projects, 34,492 projects contained asynchronous APIs, accounting for 37.74% of all projects. The most frequently used API was Thread, accounting for 30.07% of the projects.

Executors, being factory classes for thread pools within JUC, indicate the potential use of ExecutorService instances to execute asynchronous tasks. Therefore, projects using Executors, ExecutorService, FutureTask, and Future might all be employing traditional asynchronous constructs. However, CompletableFuture was only used in 3.53% of the projects, indicating it has not yet achieved widespread adoption.

RQ2: Does the CompletableFuture construct result in less complex asynchronous tasks compared to traditional asynchronous constructs?

As is well known, concurrent programming is a challenging mode of programming [24], and the CompletableFuture class has many member methods. By invoking these methods, multiple asynchronous tasks can be controlled and combined. We are interested in whether this approach helps programmers write less complex asynchronous tasks. We constructed two datasets: Corpus-2, which includes projects with numerous CompletableFuture calls and may also contain traditional asynchronous constructs, and Corpus-3, which solely contains projects with many traditional asynchronous constructs. Both datasets were selected based on the number of stars to ensure representativeness.

Corpus-2. On the BOA 2022 Jan/Java dataset, we traversed the AST's method invocation nodes to identify the top 1000 projects with the most calls to CompletableFuture's static methods for submitting asynchronous tasks. Next, we filtered the projects. Using the GitHub API, we obtained the star counts of projects and excluded those with fewer than 1000 stars. We utilized the CodeQL code analysis engine [25] for static analysis to measure asynchronous task metrics. Before analyzing with CodeQL, databases of the projects to be analyzed must be built. GitHub provides CodeQL databases for over 200,000 projects [26]; we prefer using these provided databases. If not available, we attempted to build databases locally from the source code, excluding projects that fail to build, such as Android-related projects that require additional compilation environments. Sorted by the number of CompletableFuture calls, we selected the top 20 projects that were not excluded to form corpus-2, which already contained a sufficient number of CompletableFuture calls.

Corpus-3. Using the same method, we identified the top 1000 projects from the BOA dataset that had not used CompletableFuture but had the highest number of calls to the traditional asynchronous constructs' execute and submit methods. Unlike with corpus-2, we relaxed the star count requirement, excluding projects with fewer than 700 stars to include those with relatively more calls in our corpus. Ultimately, we still selected the top 20 un-excluded projects to form corpus-3.

Method. We conducted our analysis using CodeQL, which uses a global data flow analysis framework. We treated instantiation sites of asynchronous task types (including Callable, Runnable interface types, and CompletableFuture's used types like Supplier, Function, Consumer, BiFunction, and BiConsumer) as sources and the method invocation sites where asynchronous tasks are submitted as sinks, based on the concept of taint analysis. Taint analysis determines whether objects at source sites flow into sink sites, identifying all potentially tainted program nodes. Through this method, we obtained the definition locations of all asynchronous tasks submitted in both manners, from which we derived metrics for the asynchronous tasks.

Result. As shown in Tables 3 and 4, "SN" represents the number of stars, "CN" represents the number of CompletableFuture calls, "EN" represents the number of asynchronous tasks submitted via the Executor framework, "CC" represents the average cyclomatic complexity of the asynchronous tasks called by CompletableFuture, and "EC" represents the average cyclomatic complexity of the asynchronous tasks submitted by the Executor framework. The CC and EC columns of the "Overall average" in the last row are calculated by multiplying the complexity value of each item with its corresponding call count (CN or EN), summing up all items, and then dividing by the total number of calls. In Table 3, the overall average cyclomatic complexity of asynchronous tasks submitted by CompletableFuture is 1.71, while that of tasks submitted by the Executor framework is 2.20. In Table 4, projects in corpus-3, which do not use CompletableFuture, show an average cyclomatic complexity for tasks submitted by the Executor framework of 2.15. Whether in projects that

contain only traditional asynchronous constructs or those where traditional asynchronous constructs coexist with CompletableFuture, the tasks submitted by CompletableFuture exhibit lower complexity. Therefore, we can conclude that the form of submitting asynchronous tasks via CompletableFuture can help developers write code for asynchronous tasks with lower complexity.

Table 3. Corpus-2 statistical results.

Project Name	SN	CN	EN	CC	EC
Peergos/Peergos	1742	1833	20	1.28	1.65
atomix/atomix	2340	1172	135	1.45	1.72
apache/pulsar	13,267	1092	102	1.93	3.19
infinispan/infinispan	1086	765	107	1.67	2.07
apache/flink	22,149	712	312	1.70	1.38
line/armeria	4465	636	272	1.88	1.51
OpenLiberty/open-liberty	1095	471	458	1.92	2.97
Azure/azure-sdk-for-java	2058	304	62	2.17	1.58
camunda-cloud/zeebe	2859	294	60	1.45	1.45
crate/crate	3787	259	87	1.68	1.57
hyperledger/besu	1301	178	22	1.53	1.68
apache/netbeans	2418	150	217	3.14	3.76
apache/ratis	1086	142	7	1.56	1.14
lettuce-io/lettuce-core	5128	126	7	1.88	1.14
apache/hbase	4994	126	136	1.35	2.68
ballerina-platform/ballerina-lang	3466	82	3	2.55	1.00
microsoft/CDM	1564	76	1	7.13	1.00
quarkusio/quarkus	12,448	74	70	1.47	2.33
ben-manes/caffeine	14,417	48	66	2.04	1.45
lucko/LuckPerms	1840	36	108	1.81	1.25
Overall average	5175.50	428.80	112.60	1.71	2.20

Table 4. Corpus-3 statistical results.

Project Name	SN	EN	EC
aws-amplify/aws-sdk-android	995	1530	1.53
opensourceBIM/BIMserver	1437	477	2.27
grpc/grpc-java	10,968	316	1.93
apache/activemq	2199	309	3.13
MuntashirAkon/AppManager	3513	234	3.09
freenet/fred	941	187	3.24
google/guava	48,578	138	1.54
blazegraph/database	847	113	3.70
sun0x00/redtorch	740	94	3.31
Docile-Alligator/Infinity-For-Reddit	3654	83	3.81
voldemort/voldemort	2611	74	4.96
tonikelope/megabasterd	3819	72	7.92
apache/iotdb	4066	67	3.45
hapifhir/hapi-fhir	1797	66	1.39
EdwardRaff/JSAT	775	56	4.43
open-telemetry/opentelemetry-java	1693	54	1.11
GoogleContainerTools/jib	13,062	28	1.96
apache/cloudstack	1546	26	2.08
JPressProjects/jpress	2620	25	1.32
KOHGYLW/kiftd-source	1036	25	2.24
Overall average	5344.85	198.70	2.15

5. Refactoring Method

In this section, we detail the refactoring methods. We start by outlining the framework for automatic refactoring. Then, we explain how to map AST to JimpleIR and discuss the

static program analysis methods necessary for executor inheritance structure analysis and refactoring precondition checks. We also describe four refactoring templates. Finally, we present the refactoring algorithms.

5.1. Method Framework

The framework diagram for the automated refactoring method aimed at asynchronous mechanisms is shown in Figure 4. In the automated refactoring process, the first step is to transform the source program into an AST (Abstract Syntax Tree) and Jimple IR (a three-address intermediate representation provided by the static analysis tool Soot). We use the visitor pattern to traverse the AST to obtain specific syntax nodes, which we then map to the corresponding Jimple IR model. Concurrently, we analyze the executor inheritance structure. Next, we perform program analysis on the Jimple IR to check preconditions, starting with verifying if the Future variable's type matches the Future interface type. After obtaining the results of the executor inheritance structure analysis (executor types in the program that meet the refactoring conditions), it then determines whether the executor object type in the current traditional asynchronous construction statement satisfies the refactoring conditions through executor type analysis. Finally, the refactoring checks if the future object calls the cancel(true) method and if there are any type checks and operations on the future object in the program. Traditional asynchronous constructs that meet the precondition conditions are refactored by applying a refactoring template to modify the AST and complete the refactoring.

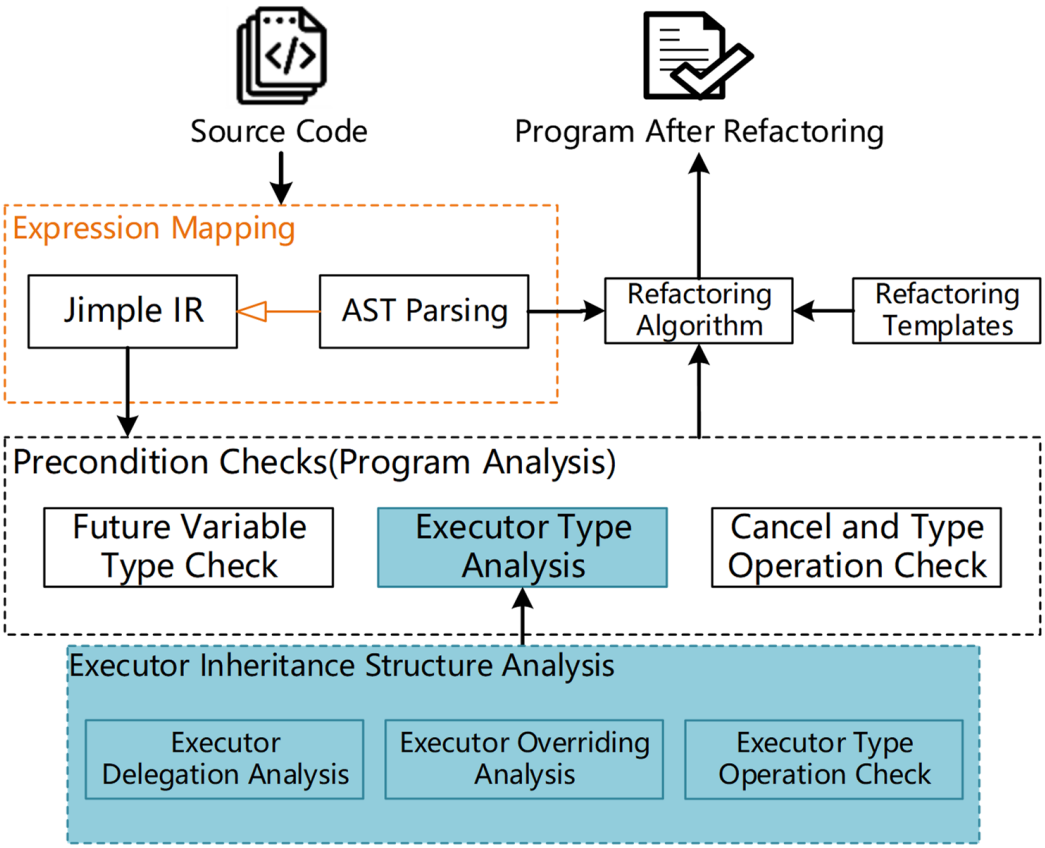

Figure 4. Refactoring Method Framework.

5.2. Mapping from AST to Jimple IR

The specific code refactoring process involves converting the source code into an AST, modifying AST nodes, and then converting the AST back into the refactored source code. Program analysis is based on Jimple IR because it has only 15 types of statements. A statement involves at most three local variables or constants, and the conversion process

from bytecode to Jimple IR eliminates redundant code, making it more suitable for static program analysis. Therefore, specific nodes on the AST need to be mapped to Jimple IR before performing program analysis.

In our method, the syntax tree nodes that need mapping include MethodInvocation expressions, instanceof type comparison expressions, and CastExpression type casting expressions. On the AST, expressions might correspond to multiple Jimple IR statements. For example, in the method invocation a.b().c(), the invocation of c() includes b(), with the AST placing the c() node as the parent of the b() node. In the three-address code Jimple IR, this expression is represented as two expression statements. In our mapping method, we do not consider the child nodes of the current expression node on the AST. Below is our detailed mapping method.

"MethodInvocation" represents a method invocation. We traverse all method invocation statements in the corresponding block of Jimple IR code, comparing the names of the method invocations to identify the unique corresponding Jimple IR statement.

"Instanceof" type comparison expression is used to check whether a variable or expression's type is the reference type or a subtype of the reference type on the right side of the instanceof operator. We traverse all Jimple statements in the corresponding block, matching whether they contain the string instanceof.

"CastExpression" forces the type conversion of a variable or expression to a subtype of its original type. By traversing all definition and assignment statements in the corresponding block, we determine whether the right side of the statement includes a type-casting expression to identify the corresponding statement.

Using the aforementioned methods for matching, if more than one statement is matched within a block, they are differentiated by line number. If the line numbers are also the same, the expressions in the AST are sorted from smallest to largest based on line and column numbers, following the left-to-right, top-to-bottom execution order of Java code. Simultaneously, the Jimple IR block is converted into an intraprocedural control flow graph, traversing from entry to exit, matching the corresponding Jimple IR statements by sequence number.

5.3. Executor Inheritance Structure Analysis

The JUC package contains 3 delegated executor types. Our source code review indicates that other executor types are also refactorable. However, software projects may use executor types from third-party libraries or custom-implemented executor types, the latter of which often appear in software projects that heavily utilize asynchronous programming techniques. For example, Jenkins has 6 custom executor types, while the JUC package has only 3 delegated types and 4 other executor types. The refactoring method mentioned in previous research [4] does not apply to subclassed asynchronous constructs, which means that the more widely asynchronous programming is used in a project, the more likely it is that the project cannot be fully refactored. Therefore, we propose an executor inheritance structure analysis method. Starting from the two most common object-oriented code reuse mechanisms of inheritance and delegation [27], we perform static analysis on the entire executor inheritance structure of the software project to obtain all the executor types that can be refactored, as well as the executor types that need further judgment due to the use of the delegation pattern. Finally, we exclude all executor types that have type checks and type casts in their 'submit' and 'execute' methods, as these executor types cannot be safely refactored.

Our static analysis method is based on points-to analysis, and we begin by defining points-to analysis.

Definition 1. *Points-to analysis: points-to analysis is a static program analysis technique that computes the memory locations that references in a program can point to at runtime [28]. It is formally represented as:*

$$\text{pointsTo}(ref) = \{i_1, i_2, \ldots, i_n\} \tag{1}$$

Here, "*ref*" is a reference type variable, $i_k \in \{i_1, i_2, \ldots, i_n\}$ $(1 \leq k \leq n)$ is an abstract heap object identifier. We use the flow-insensitive, interprocedural context-insensitive points-to analysis provided by the Spark static analysis framework [29] in Soot. This points-to analysis maps object creation statements in the program (new statements) to corresponding abstract heap object identifiers.

Definition 2. *Alias analysis: alias analysis is a static analysis technique used to determine whether two or more references might refer to the same memory location. It is a variant of points-to analysis, formally represented as [30]:*

$$\text{alias}(ref_1, ref_2) = \begin{cases} \text{true} & \text{if pointsTo}(ref_1) \cap \text{pointsTo}(ref_2) \neq \varnothing \\ \text{false} & \text{otherwise} \end{cases} \tag{2}$$

Definition 3. *possibleTypes: Soot provides a method to obtain the possible actual types of reference variables through the results of points-to analysis, represented by abstract heap object identifiers (new statements). It is formally represented as:*

$$\text{possibleTypes}(ref) = \{T \mid T \in \text{getType}(\text{pointsTo}(ref))\} \tag{3}$$

Here, "getType" is used to obtain the type information of an abstract heap object identifier (new statement).

5.3.1. Executor Delegation Analysis

Executor delegation analysis aims to identify executor types within an executor inheritance hierarchy that delegate the concrete (partial) logic of their submit or execute methods, responsible for submitting asynchronous tasks, to another executor object. Whether such a type satisfies the refactoring conditions depends not only on the definition of its own submit or execute method but also on the execution logic of methods with the same name in the executor type it delegates to. In this section, we first define an interprocedural field access analysis within the class. Then, we formally define the delegated executor type and present a preliminary method for determining whether a delegated executor type can be refactored.

Definition 4. *Field Access Analysis: This analysis determines whether a local reference type variable local within the body of a method m of a class t ultimately points to a field f of the given class t, if it points then class t belongs to the delegate class. The formal definition is as follows.*

$$\text{fieldAccessAnalysis}(local, f) = \begin{cases} \text{true}, & \text{if isFieldAccess}(stmt, f) \\ \text{fieldAccessAnalysis}(\text{getReturnLocal}(\text{getMethod}(stmt)), f), & \text{if isMethodInvoc}(stmt) \text{ , where } stmt = \text{getDefStmt}(local) \\ \text{false}, & \text{otherwise} \end{cases} \tag{4}$$

where

getDefStmt(Local): Retrieves the definition statement Stmt for a local reference type variable Local within the current method body via the use-def chain.

isFieldAccess(Stmt, f): Determines whether a definition statement Stmt within the method body defines a local reference type variable as field f.

isMethodInvoc(Stmt): Checks if a definition statement Stmt in the method body has a local reference type variable that serves as the return value of an intra-class method invocation.

getMethod(Stmt): Obtain the methods that contain class internal method calls within the current statement Stmt.

getReturnLocal(Method): Obtains the local reference type variable local returned in the return statement of the current method Method.

Following this, we provide the characteristic definition of delegate executor types.

Definition 5. *Delegate Executor Type: A Delegate Executor Type (DT) is defined as a class that simultaneously satisfies the following characteristics.*

1. DT is a concrete implementation class of the ExecutorService interface, formally represented as:

$$DT \in \text{allSubTypes}(\textit{ExecutorService}) \land \forall \text{m} \in \text{getMethods}(DT), \text{“Abstract”} \notin \text{modiMethod}(m) \tag{5}$$

where
allSubTypes(T): Retrieves the set of all direct and indirect subclasses of type T.
getMethods(T): Obtains the collection of all methods defined in type T.
modiMethod(Method): Acquires the set of modifiers for the method Method.

2. DT has an instance field f', whose declared type is ExecutorService or its subclass, formally represented as:

$$\exists f' \in \text{getFields(DT)} : \text{fieldType}(f') \in \text{allSubTypes(ExecutorService)} \land \text{“Static”} \notin \text{modiField}(f') \tag{6}$$

where
getFields(T): Retrieves all fields defined in type T.
fieldType(F): Obtains the declared type of field F.
modiField(F): Acquires the set of modifiers for field F.

3. The bodies of DT's execute and submit methods both contain method invocations with the same sub-signature as the methods themselves, and the receiver object of these calls is field f', formally represented as:

$$\begin{gathered}\forall m \in \{\text{execute}, \text{submit}\}, m \in \text{getMethods}(DT) \Rightarrow \exists stmt \in \text{getBody}(m), \\ \exists m' \in \text{getMethod}(stmt) : (\text{sig}(m') = \text{sig}(m)) \land (\text{fieldAccessAnalysis}(\text{receiver}(stmt), f') = \text{true})\end{gathered} \tag{7}$$

where
getBody(Method): Retrieves the body of a given method Method, a collection of statements Stmt.
sig(Method): Obtains the sub-signature of a given method Method, including the return type, method name, and the list of parameter types.
receiver(Stmt): Identifies the receiver object of the method invocation expression contained in a given statement Stmt, which is a reference type local variable.

Definition 6. *Refactorable Delegate Executor Type: After identifying a delegate executor class, further judgments are necessary to determine if it could meet the refactoring conditions. More specifically, it involves assessing the differences between the three submit methods and the execute method, with the first step to eliminate known differences.*

1. Submit statement: In Jimple IR, we use use-def chains to obtain the definition statement of local in the return statement, mapping the return statement to a separate return instruction. Additionally, we remove the equals sign and the local on its left side from the local definition statement and map it to a regular expression statement.
2. Method invocation differences: The method invocations within the submit and execute methods with the same sub-signature differ. We remove the method signature part from these method invocation statements.
3. Parameter differences: For submit(Runnable, Value), the initialization statement for the second parameter needs to be eliminated, and the locals in the method body are renamed sequentially. For submit(Callable), the parameter declaration of the Callable type is replaced with Runnable.

After eliminating known differences, we can compare the submit and execute methods. If no unknown differences exist, the corresponding types are added to the refactorable executor type set.

1. *deleSubmitR* set contains delegate executor types where submit(Runnable) method has no unknown differences.
2. *deleSubmitRV* set contains delegate executor types where submit(Runnable, Value) method has no unknown differences.
3. *deleSubmitC* set contains delegate executor types where submit(Callable) method has no unknown differences.

5.3.2. Executor Overriding Analysis

In addition to delegate executor types, the characteristic of code reuse through inheritance in object-oriented programming also provides a possibility for identifying refactorable executor types. If executor types internal to JUC meet the refactoring conditions, then executor types that inherit from those within JUC and do not override instance methods that could affect submit or execute still meet the refactoring conditions. This section first defines key methods, followed by the definition of refactorable executor types.

Definition 7. *Key Methods: For the executor type RT that has been determined to be refactorable, its key methods are the submit method and the execute method, as well as all non-private and non-final instance methods that are directly or indirectly called within the class. The formal definition is:*

$$\mathrm{CM}(RT) = \{\text{submit}, \text{execute}\} \cup \left\{ m \middle| \begin{array}{c} m \in \text{getMethods}(RT), \\ \neg(\text{“private”} \in \text{modiMethod}(m) \vee \text{”final”} \in \text{modiMethod}(m) \vee \text{”static”} \in \text{modiMethod}(m)) \end{array} \right\} \tag{8}$$

Definition 8. *Refactorable Executor Type Set: We define the refactorable executor type set RTS as a specific group of executor classes. The method of judgment involves traversing the inheritance structure of AbstractExecutorService; if it is not part of JUC and its direct parent is a refactorable type, then by obtaining the key methods of the direct parent, we determine whether the current type overrides to identify refactorable executor types. The formal definition is as follows:*

$$RTS = \text{ExecutorInJUC} \cup \left\{ RT \middle| \begin{array}{c} RT \in allSubTypes(\text{AbstractExecutorService}) \wedge RT \notin ExecutorInJUC \wedge \\ getSuperClass(RT) \in RTS \wedge declarMethods(RT) \cap \mathrm{CM}(getSuperClass(RT)) = \varnothing \end{array} \right\} \tag{9}$$

where

ExecutorInJUC represents the set of executor types within JUC.
getSuperClass(T) returns the direct parent class of type T.
declarMethods(T) returns the set of methods declared in type T.

5.3.3. Executor Type Operation Check

In traditional asynchronous constructs utilizing Runnable or Callable type tasks, the actual workflow after refactoring involves wrapping the original asynchronous tasks with CompletableFuture's internal asynchronous task types before submission to the executor. Therefore, if the executor's methods contain instanceof expressions or type-casting operations related to asynchronous task types, it might prevent the refactored asynchronous tasks from being properly submitted and run.

Definition 9. *Executor Type Operation Check: The executor type operation check executorTypeOperateCheck is a static analysis method that determines whether any statements in the executor's method body check or transform the type of asynchronous tasks. It is formally represented as:*

$$\text{executorTypeOperateCheck}(executor) = \begin{cases} \text{true} & if\ \text{alias}(ref_1, ref_2) \wedge ref_1 = getExecutePara(executor) \wedge ref_2 \in allTypeOp(Runnable), \\ \text{false} & otherwise \end{cases} \tag{10}$$

where

getExecutePara(Executor) is used to obtain the parameter variable ref1 of the execute method corresponding to the executor type.

allTypeOp(T) is obtained through the Visitor pattern, capturing all instanceof statements and type-casting statements in the AST and mapping them to Jimple IR by checking whether the type object is T or any of its subclasses or implementing classes.

5.4. Precondition Checks

Before refactoring, it is necessary to perform precondition checks using static program analysis techniques to ensure consistency before and after the refactoring process. We provide an example of a traditional asynchronous construct, as shown in Figure 5. The first step involves checking the type of the Future variable; it is essential to confirm that component "a" in Figure 5 is a Future and not another subtype. This is crucial because our refactoring changes the actual type of the variable the component points to; having "a" as a Future ensures that the type constraints of the program are satisfied both before and after refactoring. Subsequently, we analyze the executor type, specifically examining component "c" in Figure 5. We employ interprocedural point-to analysis techniques to identify all possible actual types of the "executor" variables. By combining these results with the executor inheritance structure analysis, we determine whether refactoring is feasible. The specific methodology will be discussed in detail in Section 5.4.1. Lastly, we assess the behavior of the object pointed to by component "b". Through alias analysis, we determine whether it has invoked the cancel(true) method or used instanceof and type-casting operations. These operations could introduce program errors because our refactoring changes the actual type of the object that component "b" points to, with detailed methods discussed in Section 5.4.2.

Figure 5. Example of Traditional Asynchronous Constructs.

5.4.1. Executor Type Analysis

The analysis first requires the results of executor inheritance structure analysis, namely the types that satisfy the refactoring conditions and the delegated executor types that need further judgment. Then, combined with points-to analysis, it determines whether the actual type of the currently analyzed executor variable satisfies the refactoring conditions. If the actual type includes a delegated executor type, points-to analysis is used to further determine the field type of this executor object.

Definition 10. *(Executor Type Analysis) Given a reference type local variable local pointing to an executor object, executor type analysis determines whether the executor type can meet the refactoring conditions. It is formally represented as:*

$$\text{executorTypeAnalysis}(local) = \begin{cases} \text{true} \quad if\ \neg\text{executorTypeOperateCheck}(ET) \wedge ET \in \text{possibleTypes}(local) \wedge \text{isExecuteInvoc}(local), \\ \text{true}\ if\ (EST \in RTS\ \vee \text{getRealTypes}(local) \in\ RTS) \wedge EST \in \text{possibleTypes}(local) \wedge \text{isSubmitInoc}(local), \\ \text{false} \quad \text{otherwise} \end{cases} \tag{11}$$

where

isExecuteInvoc(Local) and isSubmitInvoc(Local) are used to determine whether the method invocation in the traditional asynchronous construct is an execute method invocation or a submit method invocation.

getRealTypes(Local) is used to obtain the actual delegate executor types within the delegate executor types, formally represented as:

$$\text{getRealTypes}\ (local) = \left\{ \text{possibleTypes}(\text{getExecuteField}\ (EST)) \left| \begin{array}{c} EST\ \in\ deleSubmitR \wedge EST \in\ \text{possibleTypes}(local)\ \wedge\ \text{isSubmitRInvoc}(local) \\ \vee EST \in\ deleSubmitRV \wedge EST \in\ \text{possibleTypes}(local)\ \wedge\ \text{isSubmitRVInvoc}(local) \\ \vee EST \in\ deleSubmitC \wedge EST \in\ \text{possibleTypes}(local) \wedge\ \text{isSubmitCInvoc}(local) \end{array} \right. \right\} \tag{12}$$

where

getExecuteField(DT) is used to obtain the field used for delegation within the delegate executor types.

isSubmitRInvoc(), isSubmitRCInvoc(), and isSubmitCInvoc() are used to determine if the method invocations are submit(Runnable), submit(Runnable, Value), and submit(Callable), respectively.

5.4.2. Cancel Call and Type Operation Checks

To ensure consistency before and after refactoring, it is necessary to analyze the behavior of the Future object returned by the submit method. Firstly, CompletableFuture differs from the FutureTask used in traditional asynchronous constructs in that its cancel method cannot cancel task execution via an interrupt exception. Therefore, if the receiver object of a cancel(true) method invocation aliases with the Future object returned by the traditional asynchronous construct, the refactoring should be canceled. Secondly, our refactoring changes the actual type of the Future object. If the program contains type checking instanceof statements and type-casting statements, it might lead to changes in the results of type checks or failures in type conversions. Therefore, refactoring should be canceled if the reference variable in instanceof statements and cast statements aliases with the Future object returned by the traditional asynchronous construct.

Definition 11. *Cancel Call and Type Operation Check: This is a static analysis method to determine whether a future object has invoked the cancel(true) method or undergone type checking and type casting. It is formally represented as:*

$$\text{cancelAndTypeCheck}(futureObj) = \begin{cases} \text{true} & if\ \text{alias}(futureObj, ref)\ \wedge (\ ref \in getCancelRecs() \vee ref \in allTypeOp(Future)), \\ \text{false} & \text{otherwise} \end{cases} \tag{13}$$

Here, the getCancelRecs() method uses the visitor pattern to traverse the AST, searching for method call nodes with the method name "cancel" and a parameter of true. It maps these nodes to JimpleIR, collects the receiver object variables of the method calls on the IR, and returns them as a collection.

5.5. Refactoring Templates

We have summarized four refactoring patterns based on four methods of traditional asynchronous constructs, illustrated with templates. As shown in Table 5, the Source Code column represents the code before refactoring, and the After Refactoring column represents the code after refactoring, where the syntax elements that remain unchanged before and after refactoring of an expression are shown in bold. These include the execute method declared by the Executor interface and the three overloaded submit methods declared by the ExecutorService interface.

- Pattern 1: Refactoring of execute(Runnable)

Converts execute(Runnable) from the Executor interface to CompletableFuture.runAsync, facilitating asynchronous task submission with Runnable.

- Pattern 2: Refactoring of submit(Runnable)

Transforms submit(Runnable) from an ExecutorService interface instance into a CompletableFuture, using runAsync to handle tasks returning Future.

- Pattern 3: Refactoring of submit(Runnable, Value)

The refactoring is divided into two parts. The first part draws from the source code of the FutureTask constructor in JUC. By calling the callable method from the Executors class in JUC, Runnable and Value are converted to Callable. This thus converts to the situation of Refactoring Pattern 4. To simplify the code in Table 5, the first part is shown in the third line of Table 5, and the second part is left for the reader to refer to in Pattern 4.

- Pattern 4: Refactoring of submit(Callable)

Unlike Patterns 1 and 2, which use Runnable type asynchronous tasks, Callable type tasks have return values, requiring the use of CompletableFuture's supplyAsync method. It is necessary to refactor the Callable type task into a Supplier type. In the first line of the code on the right side of Table 5, a new Callable type variable is defined because the Callable type asynchronous task might not be a simple variable but an expression, and placing it directly into the newly defined task could lead to data races. Since the Callable interface's call method declares that it throws exceptions, whereas the Supplier interface's get method does not, to maintain consistency in exception handling before and after refactoring, as shown on the right side of Table 5, in lines 2–6, a Supplier type asynchronous task object is defined using a lambda expression. By wrapping the exceptions thrown by the call method in a RuntimeException using a try...catch statement, in lines 8–9, the CompletableFuture's handle method is used to process the exception and convert it to a normal result. Then, through the thenCompose method, in lines 11–13, a new CompletableFuture object is instantiated; if the result is an exception, it is unwrapped and set as the exception result of the new CompletableFuture object. In lines 15–17, if it is a normal result, it is directly set as the normal result of the new CompletableFuture object, which is then returned.

Table 5. Refactoring Templates.

Pattern Name	Source Code	After Refactoring
Pattern 1	**executor**.execute(**runTask**);	CompletableFuture.runAsync(**runTask**,**executor**);
Pattern 2	Future f = **eService**.submit(**runTask**);	Future f = CompletableFuture.runAsync(**runTask**,**eService**);
Pattern 3	Future f = **eService**.submit(**runTask**,**value**);	Future f = **eService**.submit(Executors.callable(**runTask**, **value**));

Pattern 4	Future f = **eService**.submit(**callTask**);	1Callable newTask = **callTask**;
		2Future f = CompletableFuture.supplyAsync(() -> {
		3 try {
		4 return newTask.call();
		5 } catch (Exception e) {
		6 throw new RuntimeException(e);
		7 }
		8 }, **eService**)
		9 .handle((result, exception) ->
		10 exception != null ? exception : result)
		11 .thenCompose((resultOrException) -> {
		12 CompletableFuture newFuture = new CompletableFuture();
		13 if (resultOrException instanceof CompletionException) {
		14 CompletionException cptException = (CompletionException) resultOrException;
		15 RuntimeException runException = (RuntimeException) cptException.getCause();
		16 newFuture.completeExceptionally(runException.getCause());
		17 return newFuture;
		18 } else {
		19 newFuture.complete(resultOrException);
		20 return newFuture;}});

5.6. Refactoring Algorithm

This section presents the design of the refactoring algorithm. Initially, the source code is used to generate an AST and Jimple IR, on which the executor inheritance structure analysis is performed. Then, all method invocation expressions, type comparison expressions, and cast expression nodes are traversed. First, references needed for cancel call and type check analysis are obtained, followed by obtaining the method signature from the method invocation expressions to identify the traditional asynchronous construct's Jimple IR statements. The preconditions for refactoring are evaluated, and the code that passes is refactored according to the method signature using the corresponding refactoring template. The detailed steps are presented in Algorithm 1.

Algorithm 1: Refactoring Algorithm

Input: P ← Source program
Output: P′ ← Refactored source program

```
MIS← getTACENodes(P)// Get the traditional asynchronous construct expression nodes.
doEISA()//Run executor inheritance structure analysis.
foreach MI ∈ MIS do
    JIR ← getJIRStmt(MI)// Get the corresponding Jimple IR statement.
    receiverLocal ← getReceiLocal(JIR)// Get the receiver object.
    futureLocal ← getReturnObj(JIR)// Get the return value of a method invocation.
    if (executorTypeAnalysis(receiverLocal) && canceAndTypeAnalysis(futureLocal)) then
        applyRefacTemplate(MI);// Apply the corresponding refactoring template.
    end
end
return P′
```

6. Implementation and Evaluation

We first describe the implementation of our approach, the automatic refactoring tool ReFuture. We then evaluate our method by applying ReFuture to 9 selected large-scale open-source projects. In this evaluation, we investigate the following questions:

- RQ1 (Applicability): How many instances can ReFuture refactor in real projects?
- RQ2 (Correctness): Does ReFuture introduce errors into the code after refactoring?
- RQ3 (Refactoring Efficiency): How much code does ReFuture need to change and how much time does it take to complete the refactoring?
- RQ4 (Comparison): How does ReFuture compare to existing methods for refactoring traditional asynchronous constructs?

6.1. Refactoring Tool

ReFuture is an Eclipse plugin implemented under the Eclipse JDT framework, extending classes such as UserInputWizardPage and Refactoring. Figure 6 shows the refactoring interface, with the top section displaying the Java source files to be refactored within the software project. The bottom section presents a side-by-side comparison: the left side shows the original code, while the right side displays the corresponding refactored code.

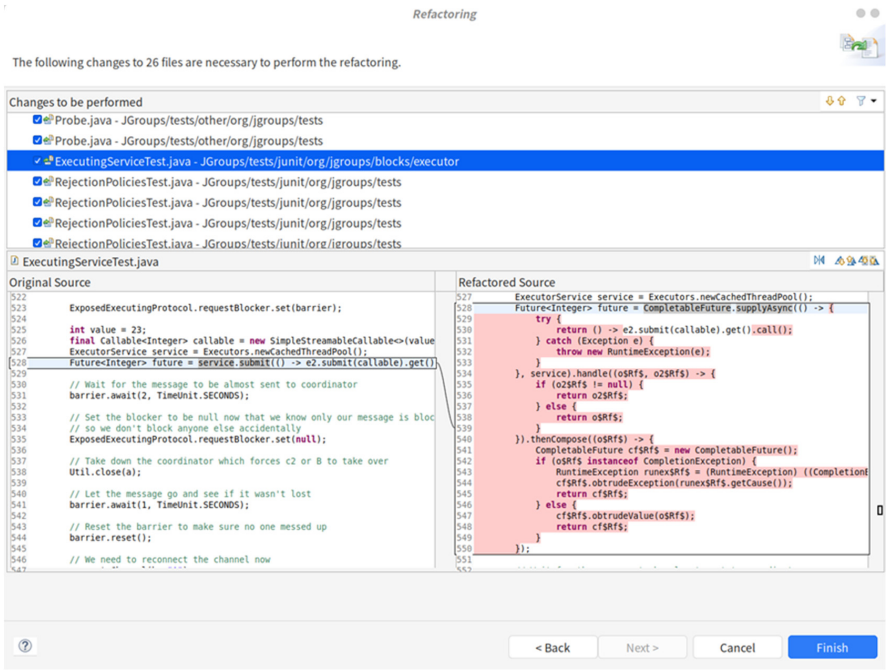

Figure 6. Interface of the refactoring tool ReFuture.

6.2. Experimental Setup

All experiments were conducted on a computer equipped with an Intel Core i5-6300HQ CPU at 2.30 GHz (4 cores, 4 threads) and 16 GB of DDR4 2133 MHz RAM; the software environment consisted of the Ubuntu 22.04.3 operating system, JDK 1.8.32, with Eclipse 2020-6 used as the development and testing platform for the refactoring tool, and the Java optimization framework soot 4.4.0 for static program analysis.

6.3. Experimental Method

We selected 9 real-world open-source programs to evaluate ReFuture. The chosen open-source programs are mature and well-known to demonstrate the actual refactoring effects of our method on large-scale programs. These applications include ActiveMQ [31], Hadoop [32], Elasticsearch [33], Jenkins [34], Tomcat [35], JGroups [36], Flume [37], ZooKeeper [38], and Tika [39]. These programs cover multiple domains: ActiveMQ is an open-source message broker, Hadoop is a framework for distributed data and computation, Elasticsearch is an open-source search and analytics engine, Jenkins is an open-source continuous integration (CI) tool, Tomcat is a free web server, JGroups is a group communication tool, Flume is a distributed, reliable, and available log system, ZooKeeper is an open-source distributed coordination service, and Tika is an Apache open-source content analysis toolkit. It is worth noting that Jenkins is a highly renowned and actively maintained open-source program, Elasticsearch is mentioned in the referenced literature [4], and JGroups is a well-known asynchronous program. The remaining open-source projects are from the Java project list on the Apache Software Foundation's website [40], which we randomly selected. Table 6 shows the version information, the number of asynchronous task submission statements containing ExecutorService constructs, the counts of execute and submit method invocations, and the total lines of source code. For RQ1 (applicability), we applied ReFuture to automatically refactor these projects and counted the number of occurrences of ExecutorService constructs for submitting asynchronous tasks, successful refactorings using the four patterns, as well as failed refactorings and reasons. For RQ2 (correctness), we assessed the correctness by running the project's test cases before and after refactoring to ensure no errors were introduced. For RQ3 (refactoring efficiency), we measured the changes in lines of code and refactoring time after applying ReFuture. For RQ4 (comparison), how does ReFuture compare to existing methods for refactoring traditional asynchronous constructs?

Table 6. Test Program Information.

Project Name	Version	Number of Traditional Asynchronous Constructs	Number of Execute	Number of Submit	Number of Lines of Java Code
ActiveMQ	5.16.6	345	273	72	532,567
Hadoop	2.10.2	170	43	127	1,398,914
Elasticsearch	6.1.4	103	97	6	704,199
Jenkins	2.356	52	4	48	1,195,314
JGroups	4.2.21	47	26	21	126,110
Tomcat	9.0.80	46	33	13	357,202
Flume	1.10.1	22	2	20	94,180
ZooKeeper	3.8.0	19	4	15	127,087
Tika	2.8.0	9	4	5	164,885
total	—	813	486	327	4,700,458

6.4. Results and Analysis

- RQ1: How many instances can ReFuture refactor in real projects?

To address the research questions, we assessed the success and failure counts of ReFuture's refactoring across the four patterns and analyzed the reasons for any failures. Since the preconditions for refactoring execute and submit methods differ, the results are presented in two parts, as shown in Tables 7 and 8.

Table 7. Statistics on execute Refactoring Outcomes.

Project Name	Number of Execute	Number of Refactorings for Pattern 1	Executor Type Mismatch
ActiveMQ	273	270	3
Hadoop	43	43	0
Elasticsearch	97	27	70
Jenkins	4	3	1
JGroups	26	19	7
Tomcat	33	12	21
Flume	2	2	0
ZooKeeper	4	4	0
Tika	4	4	0
total	486	384	102

Table 8. Statistics on submit Refactoring Outcomes.

Project Name	Number of Submit				Number of Refactorings				Results Obtained from Delegation Analysis
	Pattern 2	Pattern 3	Pattern4	Total	Pattern 2	Pattern 3	Pattern 4	Total	
ActiveMQ	13	1	58	72	10	0	47	57	7
Hadoop	102	0	25	127	33	0	7	40	5
Elasticsearch	1	0	5	6	1	0	2	3	0
Jenkins	23	2	23	48	5	0	8	13	10
JGroups	15	1	5	21	0	0	1	1	0
Tomcat	3	1	9	13	0	0	3	3	2
Flume	2	0	18	20	2	0	8	10	0
ZooKeeper	5	0	10	15	1	0	9	10	2
Tika	2	0	3	5	2	0	3	5	0
total	166	5	156	327	54	0	88	142	26

Table 7 summarizes the refactoring results for the execute method across nine test programs. Out of 486 execute nodes, 384 were successfully refactored. The projects Hadoop, Flume, ZooKeeper, and Tika achieved complete success in their refactoring efforts. In contrast, other projects faced cancellations due to incompatible executor types. Notably, ActiveMQ, being a message middleware program, had the highest number of nodes requiring refactoring. Elasticsearch followed with the second highest number of nodes, with 70 instances canceled due to executor type mismatches.

Table 8 presents the results for the three submit method refactoring patterns. Refactoring submit methods is more complex and has stricter conditions compared to execute, resulting in 142 successful refactorings out of 327 asynchronous task submission nodes. Elasticsearch, Flume, ZooKeeper, and Tika included only patterns 2 and 3, while Tomcat had instances of all 3 patterns. Among the 3 patterns, pattern 4 had the fewest refactorings due to the lower number of submit(Runnable, Value) calls—only 4 instances across the 9 test programs. Refactoring successes were further enhanced by delegating analysis to help determine executor types, leading to 26 additional successful refactorings in ActiveMQ, Hadoop, Jenkins, Tomcat, and ZooKeeper. This approach increased the number of successful refactorings by 22.4% compared to without using delegation analysis.

Table 9 provides a detailed breakdown of the reasons for refactoring failures across the three submit method patterns. Out of 327 submit method invocations, 185 potential refactorings were excluded. The primary reason for exclusion was incompatible executor types, accounting for 107 cases. This issue was prevalent across multiple test programs, with Tika being the only program without this issue. The second major reason was the presence of cancel(true) method invocations, found in 45 cases. Additional issues included type cast mismatches and declaration type mismatches, with 16 and 17 occurrences, respectively. Unlike incompatible executor types, these issues were more localized to specific test programs.

Table 9. Statistics on Reasons for submit Refactoring Failures.

Project Name	Variable Type Casting	Variable Declaration Type Not Future	Executor Type Mismatch	Cancel (True)	Total
ActiveMQ	0	0	11	4	15
Hadoop	0	10	40	37	87
Elasticsearch	0	0	3	0	3
Jenkins	10	0	25	0	35
JGroups	3	7	10	0	20
Tomcat	3	0	6	1	10
Flume	0	0	10	0	10
ZooKeeper	0	0	2	3	5
Tika	0	0	0	0	0
total	16	17	107	45	185

Overall, ReFuture successfully refactored 639 out of 813 traditional asynchronous constructs in the nine test programs, achieving a refactoring success rate of 64.70%.

- RQ2: Does ReFuture introduce errors into the code after refactoring?

Verification of the correctness of the refactorings was divided into two layers: syntactic and semantic correctness.

On the syntactic level, all refactored code could be compiled without errors. Semantically, similar to the assumptions in the literature [4], we presumed that if unit tests passed both before and after refactoring, the refactorings did not alter the functionality of the code, thus preserving the correct program semantics. Although the nine test programs selected are well-known open-source projects with mature test suites, there were still issues: 1. Some tests could not pass even before refactoring the original code, and 2. Some refactored code was not covered by existing test cases.

To address these issues, we used Randoop to generate supplementary test cases. Specifically, we first used automated tools to generate a list of methods affected by the refactorings in the test programs, not only the methods containing the refactored code but also methods calling the refactored methods, as determined by call graphs. We then excluded inaccessible methods, such as those declared private. Subsequently, Randoop was used to generate test cases for the methods in the list. Randoop automatically identifies methods that might have side effects (e.g., writing/reading files), leading to potentially different outcomes with each call; therefore, we discarded the test cases it generated for such methods and removed all test cases that failed. Finally, we retained all the Randoop-generated test cases that passed.

We combined the retained Randoop-generated test cases with those provided by the test programs that could pass the tests to form the final test suite. After executing the refactoring with ReFuture, we ran the final test suite and found no new test errors, confirming that our refactorings did not affect the semantic correctness of the program.

- RQ3: How much code does ReFuture need to change and how much time does it take to complete the refactoring?

To address this question, we evaluated the efficiency of our automated refactoring method as well as the invasiveness of the refactoring on the code. Invasiveness refers to the impact of our refactoring on the refactored programs while achieving the set objectives; excessive invasiveness can increase the difficulty of understanding the code and reduce its maintainability.

As shown in Table 10, we used the popular code counting tool SLOCCount to tally the lines of code before and after refactoring. The nine test programs together comprised 3,693,417 lines of Java code. After refactoring, the total number of lines increased to 3,695,009, an increase of 1,592 lines. This increase was due to the need to add code to ensure that exception handling within Callable was unaffected by Refactoring Pattern 2, as well as the introduction of new types into the source files during refactoring. In the Hadoop

program, given that it had the highest number of Refactoring Pattern 2 applications, it also showed the greatest change in lines of code, increasing by 726 lines. On the other hand, in Elasticsearch and JGroups, where there were no instances of Refactoring Pattern 2 and the refactoring was more concentrated within code files, the change in lines of code was minimal.

Table 10. Code Line Changes Before and After ReFuture Refactoring.

Project	Code Lines Before Refactoring	Code Lines After Refactoring	Code Lines Modified by Refactoring
ActiveMQ	532,567	532,965	398
Hadoop	1,398,914	1,399,640	726
Elasticsearch	704,199	704,238	39
Jenkins	188,273	188,361	88
JGroups	126,110	126,139	29
Tomcat	357,202	357,298	96
Flume	94,180	94,234	54
ZooKeeper	127,087	127,170	83
Tika	164,885	164,964	79
total	3,693,417	3,695,009	1592

As illustrated in Table 11, the time consumed by ReFuture for refactoring is broken down into the time for building program call graphs and the time for code refactoring. Hadoop, having the highest number of Java lines, also had the longest refactoring time at 2621 s. ReFuture conducts refactoring at the Eclipse project level, and programs managed with Maven often contain multiple submodules, each represented as a project in Eclipse. Therefore, the refactoring tool needs to be run multiple times to complete refactoring tasks like in ActiveMQ, which significantly contributed to the longer duration consumed for Hadoop and ActiveMQ. Flume, ZooKeeper, and Tika required 415, 284, and 261 s, respectively, to complete refactoring. Although these are also managed using Maven, they have fewer lines of code and contain fewer traditional asynchronous constructs. For instance, in the ZooKeeper test program, only the zookeeper-serve submodule exists in traditional asynchronous constructs, and ReFuture skips building the call graph, saving time.

Table 11. ReFuture Refactoring Time.

Project Name	Time to Build Program Call Graph (s)	Time for Code Refactoring (s)	Total (s)
ActiveMQ	1416	89	1505
Hadoop	2534	87	2621
Elasticsearch	2281	95	2376
Jenkins	1306	90	1396
JGroups	152	16	168
Tomcat	464	30	494
Flume	408	7	415
ZooKeeper	271	13	284
Tika	255	6	261
total	9087	433	9520

ReFuture refactored 639 traditional asynchronous constructs, adding a total of 1592 lines of code, an average increase of 2.49 lines per instance, indicating low invasiveness. This suggests that our refactoring has minimal impact on the programs beyond achieving the set objectives. Most of the time consumed was spent on building the full program call graphs. The total refactoring time for the nine test programs was 9520 s, with an average of 1057.78 s per test program.

- RQ4: How does ReFuture compare to existing methods for refactoring traditional asynchronous constructs?

The method and refactoring tool 2RX proposed in the literature [4] refactors SwingWorker and traditional asynchronous constructs into reactive programming-related APIs. Although the refactoring purposes are different, the target code for refactoring includes traditional asynchronous constructs. Therefore, we applied ReFuture to the same test programs to conduct a comparison and demonstrate the differences between our work and existing work. The comparison results are shown in Table 12.

Table 12. ReFuture VS 2RX.

Project Name	2RX	ReFuture
Elasticsearch	4	2
JUnit	1	4
DropWizard	2	2
Mockito	2	5
Springside4	0	9
Titan	1	1
AsyncHttpClient	105	7
Graylog2Server	0	1
Java Websocket	0	0
B3log	0	0

The 2RX part shows the results of its refactoring of the Future part. The refactoring result of ReFuture has already removed the refactoring of the execute method that 2RX does not support. The final result shows that except for Elasticsearch and AsyncHttpClient, the refactoring result of ReFuture is higher than 2RX. For Elasticsearch, the main reason is that the precision of our method using Spark pointer analysis is not enough, so the refactoring opportunity was missed. For AsyncHttpClient, ReFuture shows that there are 8 places of refactoring code in the entire project, and 7 of them were successfully refactored. 2RX refactored 105 places because the refactoring methods of ReFuture and 2RX are different. 2RX converts the method call expression whose return value type is Future and meets its pre-refactoring conditions as the parameter of the fromFuture method, while ReFuture refactors the traditional asynchronous construction, targeting the submit method of the executor. Their common point is that the return value type of the submit method may also be Future. However, there are many developer-defined methods with Future return values in the AsyncHttpClient project, and these methods do not belong to the executor type, so ReFuture did not perform the refactoring.

7. Conclusions and Future Work

This paper presents an automated refactoring method that combines various static analysis techniques such as visitor pattern analysis, alias analysis, and executor inheritance structure analysis to perform precondition checks before refactoring. Based on this method, an automated refactoring plugin, ReFuture, was implemented under the Eclipse JDT framework. In the experiments, ReFuture was evaluated based on the number of refactorings, refactoring time, and the introduction of errors, alongside a side-by-side comparison with the existing method. The refactoring results for nine large applications, including ActiveMQ, Hadoop, and Elasticsearch, show that ReFuture refactored 639 out of 813 code structures targeted for refactoring, achieving a success rate of 64.70% without introducing errors. The comparative experiment demonstrates that ReFuture not only excels in large programs but also performs well in medium- and small-scale programs. Compared to manual refactoring, it improves efficiency and provides a valuable learning opportunity for developers to master correct conversion techniques through its preview interface.

Our future research will primarily focus on the following. (1) ReFuture currently employs a context-insensitive pointer analysis technique from the Spark framework within Soot. While reliable, this method is outdated compared to newer pointer analysis technologies. Its efficacy is also limited by its dependence on the selection of the program's

entry method; inaccessible methods within the program can prevent accurate analysis of certain variables, consequently reducing the success rate of refactorings. To enhance our tool's effectiveness and broaden its applicability, we plan to integrate more advanced pointer analysis techniques in future updates. (2) Exploring more rational uses of CompletableFuture to further enhance the quality of existing asynchronous program code, for example, automatically refactoring asynchronous logic that consists of multiple traditional asynchronous constructs combined in a synchronous manner into a fully asynchronous form using CompletableFuture, thereby fully leveraging its rich API to improve program execution efficiency. (3) continuing to improve the ReFuture tool, including testing on more programs to fix potential bugs and planning to develop refactoring plugins suitable for other IDEs to assist more programmers.

Author Contributions: Y.Z.: Proposing the idea, Writing—review and editing; Z.X.: Conducting experimentation, Writing—original draft, review and editing; Y.Y.: Experimental analysis, Writing—review and editing; L.Q.: Writing—review and editing. All authors have read and agreed to the published version of the manuscript.

Funding: This work is partially supported by the Hebei Provincial Natural Science Foundation under Grant No.F2023208001 and the Hebei Provincial Overseas High-level Talent Foundation under Grant No.C20230358. Foundation of Hebei Meteorological Service under Grant 22KY09.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Android—Java's FutureTask Composability—Stack Overflow. Available online: https://stackoverflow.com/questions/14704812/javas-futuretask-composability (accessed on 21 March 2024).
2. Java—Waiting on a List of Future—Stack Overflow. Available online: https://stackoverflow.com/questions/19348248/waiting-on-a-list-of-future (accessed on 21 March 2024).
3. Google/Guava. Available online: https://github.com/google/guava (accessed on 19 September 2024).
4. Kohler, M.; Salvaneschi, G. Automated Refactoring to Reactive Programming. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; IEEE: New York, NY, USA, 2019; pp. 835–846. [CrossRef]
5. Orton, I. Alan Mycroft Refactoring Traces to Identify Concurrency Improvements. In Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs, Virtual Event, Denmark, 13 July 2021; ACM: New York, NY, USA, 2021; pp. 16–23. [CrossRef]
6. Okur, S.; Erdogan, C.; Dig, D. Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions. In *ECOOP 2014—Object-Oriented Programming*; Jones, R., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8586, pp. 515–540; ISBN 978-3-662-44201-2.
7. Okur, S.; Hartveld, D.L.; Dig, D.; Deursen, A. A Study and Toolkit for Asynchronous Programming in C#. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014. [CrossRef]
8. Lin, Y.; Dig, D. Refactorings for Android Asynchronous Programming. In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 836–841. [CrossRef]
9. Dig, D.; Marrero, J.; Ernst, M.D. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, Vancouver, BC, Canada, 16–24 May 2009; IEEE: New York, NY, USA, 2009; pp. 397–407. [CrossRef]
10. Ishizaki, K.; Daijavad, S.; Nakatani, T. Refactoring Java Programs Using Concurrent Libraries. In Proceedings of the Workshop on Parallel and Distributed Systems Testing, Analysis, and Debugging—PADTAD '11, Toronto, ON, Canada, 17–21 July 2011; ACM Press: New York, NY, USA, 2011; p. 35. [CrossRef]
11. Zhang, Y.; Shao, S.; Liu, H.; Qiu, J.; Zhang, D.; Zhang, G. Refactoring Java Programs for Customizable Locks Based on Bytecode Transformation. *IEEE Access* **2019**, *7*, 66292–66303. [CrossRef]
12. Zhang, Y.; Shao, S.; Ji, M.; Qiu, J.; Tian, Z.; Du, X.; Guizani, M. An Automated Refactoring Approach to Improve IoT Software Quality. *Appl. Sci.* **2020**, *10*, 413. [CrossRef]

13. Midolo, A.; Tramontana, E. An Automatic Transformer from Sequential to Parallel Java Code. *Future Internet* **2023**, *15*, 306. [CrossRef]
14. Lin, Y.; Radoi, C.; Dig, D. Retrofitting Concurrency for Android Applications through Refactoring. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 11 November 2014; ACM: New York, NY, USA, 2014; pp. 341–352. [CrossRef]
15. Gokhale, S.; Turcotte, A.; Tip, F. Automatic Migration from Synchronous to Asynchronous JavaScript APIs. *Proc. ACM Program. Lang.* **2021**, *5*, 1–27. [CrossRef]
16. Eclipse Java Development Tools (JDT) | The Eclipse Foundation. Available online: https://eclipse.dev/jdt/ (accessed on 19 June 2024).
17. Vallée-Rai, R.; Co, P.; Gagnon, E.; Hendren, L.; Lam, P.; Sundaresan, V. Soot: A Java Bytecode Optimization Framework. In Proceedings of the CASCON First Decade High Impact Papers on—CASCON '10, Toronto, ON, Canada, 1–4 November 2010; ACM Press: New York, NY, USA, 2010; pp. 214–224. [CrossRef]
18. Eclipse IDE | The Eclipse Foundation. Available online: https://eclipseide.org/ (accessed on 22 March 2024).
19. Xzy0311/ReFuture. Available online: https://github.com/xzy0311/ReFuture (accessed on 27 August 2024).
20. Pinto, G.; Torres, W.; Fernandes, B.; Castor, F.; Barros, R.S.M. A Large-Scale Study on the Usage of Java's Concurrent Programming Constructs. *J. Syst. Softw.* **2015**, *106*, 59–81. [CrossRef]
21. Arteca, E.; Tip, F.; Schäfer, M. Enabling Additional Parallelism in Asynchronous JavaScript Applications. In Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021), Aarhus, Denmark, 11–17 July 2021; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Dagstuhl, Germany, 2021; Volume 194, pp. 7:1–7:28. [CrossRef]
22. Goetz, B. *Java Concurrency in Practice*; Addison-Wesley: Upper Saddle River, NJ, USA, 2006; ISBN 978-0-321-34960-6.
23. Dyer, R.; Nguyen, H.A.; Rajan, H.; Nguyen, T.N. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 422–431. [CrossRef]
24. Sutter, H. The Free Lunch Is over: A Fundamental Turn toward Concurrency in Software. *Dr. Dobb's J.* **2005**, *30*, 202–210.
25. CodeQL. Available online: https://codeql.github.com/ (accessed on 28 March 2024).
26. Analyzing Your Projects—CodeQL. Available online: https://codeql.github.com/docs/codeql-for-visual-studio-code/analyzing-your-projects/#downloading-a-database-from-github (accessed on 28 March 2024).
27. Giordano, G.; Fasulo, A.; Catolino, G.; Palomba, F.; Ferrucci, F.; Gravino, C. On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022; IEEE: New York, NY, USA, 2022; pp. 947–958. [CrossRef]
28. Tan, T.; Ma, X.; Xu, C.; Ma, C.; Li, Y. Survey on Java Pointer Analysis. *J. Comput. Res. Dev.* **2023**, *60*, 274–293. [CrossRef]
29. Ondrej Lhoták. *SPARK: A Flexible Points-to Analysis Framework for Java*; National Library of Canada = Bibliothèque nationale du Canada: Ottawa, ON, Canada, 2004; ISBN 978-0-612-88247-8.
30. Fan, G.; Xuan, Z. Design and Implementation of a Dependence-Based Taint Analysis. In Proceedings of the 2016 11th International Conference on Computer Science & Education (ICCSE), Nagoya, Japan, 23–25 August 2016; IEEE: New York, NY, USA, 2016; pp. 985–991. [CrossRef]
31. ActiveMQ. Available online: https://activemq.apache.org/ (accessed on 19 June 2024).
32. Apache Hadoop. Available online: https://hadoop.apache.org/ (accessed on 19 June 2024).
33. Elasticsearch. Available online: https://www.elastic.co/cn/elasticsearch (accessed on 19 June 2024).
34. Jenkins. Available online: https://www.jenkins.io/ (accessed on 19 June 2024).
35. Apache Tomcat. Available online: https://tomcat.apache.org/ (accessed on 19 June 2024).
36. JGroups. Available online: http://www.jgroups.org/ (accessed on 19 June 2024).
37. Apache Flume—Apache Flume. Available online: https://flume.apache.org/ (accessed on 19 June 2024).
38. Apache ZooKeeper. Available online: https://zookeeper.apache.org/ (accessed on 19 June 2024).
39. Apache Tika. Available online: https://tika.apache.org/ (accessed on 19 June 2024).
40. Apache Projects List. Available online: https://projects.apache.org/projects.html?language#Java (accessed on 22 September 2024).