Supplementary

We include more details for some simplifications and discussions of the simplification patterns in current supplementary.

0.1 Control logic related simplifications.

We find out that the largest portion of program simplifications are related with control-logic-related statements. A program can be simplified by refactoring control-logic-related statements like refactoring `if-else` branches, return statements, using ternary conditional operators, and etc.

Simplify method return. When writing methods that return booleans, some developers make their code much more complicated than it needs to be. For instance, example in listing 1 try to return a boolean value according to the conditional expression `walletType != that.walletType` [1]. However, this is much more verbose than it needs to be. If we analyzed the conditional expression and the return value, we can find that the statements in line 5 and 6 are equivalent to a single line statement `return walletType == that.walletType;`.

```
1  public boolean equals(Object o) {
2      if (this == o) return true;
3      if (o == null || getClass() != o.getClass()) return false;
4      WalletTypeExtension that = (WalletTypeExtension) o;
5  -   if (walletType != that.walletType) return false;
6  -   return true;
7  +   return walletType == that.walletType;
8  }
```

Listing 1. Example of simplifying method return

It is always possible to replace the pattern `if (E) return false; else return true;` by the pattern `return !E;` for any boolean expression `E`. Some developers also simplify method return by inlining temporary variables, we classified those kind of transformations into simplifying method return because those changes are related to control flow and method's return statement.

Conditional expression simplification. Simplifying a conditional expression can provide more clarity and concise syntax. As it is possible to simplify algebraic expressions by using rules like cancellation, commutativity, associativity, etc., it is possible to simplify boolean expressions and conditional statements by rules. For example, in listing 2, the boolean expression `false == co.isExpired()` can be simplified by using `!co.isExpired()`. Both of the expressions are logically valid and have the same value. Additionally, the simplification can preserve errors and non termination (i.e., if `false == co.isExpired()` throws an exception or gets stuck in an infinite loop, the simplified expression still has the same behavior).

```
1  - if (false == co.isExpired()) {
2  + if (!co.isExpired()) {
3      return true;
4  }
```

Listing 2. Example of conditional expression simplification

Apart from the above mentioned simplifying conditional expression through algebraic rules, conditional expression can also be simplified by removing redundant conditions. For example, the conditional expression `if (t == null || !(t instanceof AbstractMessage))` can be replaced by `if (!(t instanceof AbstractMessage))`, as the developer stated in the commit message: "Delete the redundant condition. If the left argument is null, instanceof will return false anyway." [2]

Use `foreach` in loop. In some program languages(e.g., Java, python, C++, C#), `foreach` loop (or for each loop) is a control flow statement for traversing items in a collection and it is usually

[1] https://github.com/Multibit-Legacy/multibit-hd/pull/900

[2] https://github.com/seata/seata/pull/1722

used in place of a standard `for` loop statement. It has the following syntax: `for each item in collection: do something to item` [3]. Due to `foreach` loops usually maintain no explicit counter and directly fetch elements from a collection, this avoids potential off-by-one errors (a logic error involving the discrete equivalent of a boundary condition) and makes code simpler to read. During our manual analysis, we find out that developers usually simplify `for` statements to a collection via transforming other kinds of iterator to `foreach` when it comes to the iteration and operation of elements in the collection. Listing 3 presents an example.

```
- for (int symbol = 0; symbol < codeLensFromSym.length
-         ;++symbol) {
-    int codeLength = codeLensFromSym[symbol];
+ for (int codeLength : codeLensFromSym) {
     countsFromCodeLen[codeLength]++;
 }
```

Listing 3. Example of using foreach for iteration

Merge conditional. As shown in the listing 4, a series of conditional checks in which each `if` statement is different yet the resulting action is the same. Developers usually consolidate all those conditionals in a single expression by using `AND` and `OR`. As a general pattern when simplifying, it can eliminate duplicated control flow code and result in more readable program.

```
public boolean accept(File f) {
- if (f.isDirectory()) {
+ String filename = f.getName();
+ if (f.isDirectory() || filename.endsWith(".c")) {
     return true;
   }
- String filename = f.getName();
- if (filename.endsWith(".c")) {
-    return true;
- }
 }
```

Listing 4. Example of merging conditions

Ternary conditional operator. The ternary conditional operator `?:` is supported in many programming languages (e.g., Java, C, C++, JavaScript). Developers usually simplify an `if-else/switch` statement via a conditional operator. It has the following syntax: `condition ? expression-if-true : expression -if false`. Listing 5 presents an example where the developer replaces an `if-else` conditional branches and the associated statements with the conditional operator [4].

```
- Sdk sdk;
- if (attachSdk) {
-    sdk = addLatestAndroidSdk(module);
- }
- else {
-    sdk = null;
- }
+ Sdk sdk = attachSdk ? addLatestAndroidSdk(module) : null;
 }
```

Listing 5. Example of ternary conditional operator

Restructure conditional branches. The decision-making statements (e.g., `if`, `if-else`, `switch`) are a vital part of any programming language. But developers land up in coding a huge number of `if-else` or `switch-case` statements which make the code more complex and difficult to maintain. For instance, the following example in listing 6, if we need to add a new operator, we have to add a new if statement and implement the operation, this would make the code more unmanageable.

[3] https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html
[4] https://github.com/JetBrains/android/pull/1

This program can be simplified by replacing the complex if statements into much simpler and manageable code by using `Enum` to label particular business logic [5]. Alternatively, we can also use the polymorphism [6] of Java to perform the related business logic for simplifying code.

```
- TileType a = TileType.HIGH_MOUNTAIN;
- int randomType = rand.nextInt(4);
- if(randomType == 0) {
-   a = TileType.OCEAN;
- } else if(randomType == 1) {
-   a = TileType.SEA;
- } else if(randomType == 2) {
-   a = TileType.PLAINS;
- } else if(randomType == 3) {
-   a = TileType.DESERT;
- }
+ int randomType = rand.nextInt(TileType.values().length);
+ return TileType.values()[randomType];
```

Listing 6. Example of reformating if else

Replace loop with pipeline. One of the major new features in Java 8 is the introduction of the stream functionality —`java.util.stream` —which contains classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections [7]. Stream pipelining is the concept of chaining operations together. To perform a sequence of operations over the elements of the data source and aggregate their results, stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines. A stream pipeline consists of a source (such as a collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as `Stream.filter` or `Stream.map`; and a terminal operation such as `Stream.forEach` or `Stream.reduce`. Listing 7 shows an example of replacing loop for a collection with stream pipeline operations.

```
public boolean hasSpellInfoFor(String bookName) {
-  if (infoList.isEmpty()) {
-      return false;
-  }
-  for (SpellInfo s : infoList) {
-      if (bookName.equals(s.getBook())) {
-          return true;
-      }
-  }
-  return false;
+  return infoList.stream()
+      .anyMatch(s -> bookName.equals(s.getBook()));
}
```

Listing 7. Example of replacing loop with pipeline

Catching Multiple Exception Types. A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. In Java SE 7 and later, a single catch block can handle more than one type of exception, this feature can be used to reduce code duplication. The example in Listing 8, which contains duplicated code in each of the catch blocks, can be simplified by merging multiple catch together. The catch clause specifies the types of exceptions that the block can handle, and each exception type is separated with a vertical bar. In addition, the bytecode generated by compiling a catch block that handles multiple exception types will be smaller (and thus superior) than compiling many catch blocks that handle only one exception type each [8].

[5]https://github.com/qmegame/qme/pull/9
[6]https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html
[7]https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html
[8]https://docs.oracle.com/javase/7/docs/technotes/guides/language/catchmultiple.html

```
-  } catch (OutOfMemoryError e) {
-      DbException.traceThrowable(e);
-  } catch (IOException e) {
+  } catch (OutOfMemoryError | IOException e) {
       DbException.traceThrowable(e);
   }
```

Listing 8. Example of merging catches

0.2 Common code extraction.

Common code extraction is a refactoring technique that reduces code duplication and minimizes the number of lines by consolidating repeated logic into reusable methods. In Listing 9, duplicate code for checking if a string is empty appears in multiple methods. By extracting this logic into a single method, isEmpty, the redundant blocks are replaced with concise method calls. This change significantly reduces the lines of code while maintaining clarity and functionality. Centralizing common logic not only minimizes code length but also simplifies future maintenance and ensures consistency across the codebase.

```
+ public static boolean isEmpty(final Object[] array) {
+   return array == null || array.length == 0;
+ }
public static boolean isBlank(String str) {
-   if (str == null || str.length() == 0) {
-       return true;
-       }
-   return false;
+   return isEmpty(str);
}
public static String camelToSplitName(String camelName, String split) {
-   if (camelName == null || camelName.length() == 0) {
+   if (isEmpty(camelName)) {
         return camelName;
    }
    ...
}
```

Listing 9. Example of extracting method

0.3 Use APIs.

Listing 10 shows an example where developers use `Arrays.fill()` to replace the code block that implements the same function, which improves the robustness and readability of existing code. “Replace with equivalent API” simplification usually invokes public obtained libraries (e.g., JDK util package or JSON library like fastjson), replaces developer-written complex code chunks with more robustness, more reliable, and more simplified public packaged libraries’ function. The APIs should be invoked with right format, scenario and version. Programmers lacking programming experiences and relevant documentation (including code examples) could use API incorrectly. For example, one can forget to check that hasNext() returns true before calling next() with an Iterator from Java API. Misused API could result in software errors, crashes, and vulnerabilities.

```
- Entry[] tab = table;
- for (int i = 0; i < tab.length; i++)
-   tab[i] = null;
+ Arrays.fill(table, null);
```

Listing 10. Example of using APIs

0.4 Use lambda.

Lambda expressions, introduced in Java 8, provide a streamlined mechanism for representing functional interfaces with concise and readable code. This feature is particularly useful for simplifying anonymous class implementations and reducing boilerplate code in modern Java programs. In Listing 11 [9], a lambda expression is used to replace an anonymous inner class implementation, demonstrating how a succinct lambda representation can significantly reduce the number of lines of code while preserving functionality.

```
- super(corePoolSize, new ThreadFactory() {
-       @Override
-        public Thread newThread(Runnable r) {
-            return new Thread(r, threadName);
-        }
-    });
+  super(corePoolSize, r -> new Thread(r, threadName));
```

Listing 11. Example of replacing with lambda expression

0.5 Use annotations.

Annotations in modern programming languages, such as Java, enable developers to achieve concise and readable code by abstracting boilerplate functionality into declarative constructs. This approach enhances maintainability and reduces redundancy, especially in scenarios involving repetitive code patterns like getters and setters. For the example in Listing 12, the Lombok library's @Getter and @Setter annotations are utilized to automatically generate getter and setter methods for the defaultName field in the TestAppConfiguration class. This eliminates the need for manually writing these methods, thereby improving code clarity, reducing the potential for human error, and resulting reduced lines of code [10].

```
+ @Getter
+ @Setter
public class TestAppConfiguration {
    private String defaultName;
-   public String getDefaultName() {
-       return defaultName;
-   }
-   public void setDefaultName(String defaultName) {
-       this.defaultName = defaultName;
-   }
}
```

Listing 12. Example of using annotations

0.6 Inline code.

Variables or methods that are only used once could be replaced directly with their corresponding values or code snippets. This practice can lead to reduced lines of code, improving clarity and reducing unnecessary indirections. By removing intermediate variables or methods, the logic of the program becomes more direct and easier to follow. Listing 13 illustrates how intermediate variables for casting objects can be inlined. In the original code, the variables request and response are defined but used only once. By inlining them directly into the method call, the code eliminates redundancy and improves readability. Listing 14 demonstrates the inlining of a method that is only invoked once. In this example, the printConfig method is replaced with its actual implementation, reducing the need for an additional method call.

[9] https://github.com/alibaba/nacos/pull/8365/commits/3f8ec8f

[10] https://github.com/tonyG2017/test-app/pull/2

```
- HttpServletRequest request =
-   (HttpServletRequest) servletRequest;
- HttpServletResponse response =
-   (HttpServletResponse) servletResponse;
- buildResponseUtil.buildResp(request, response);
+ buildResponseUtil.buildResp((HttpServletRequest)
+   servletRequest, (HttpServletResponse) servletResponse);
```

Listing 13. Example of inlining variable

```
- private void printConfig() {
-   LOGGER.info("modifications = " + join(modders, " "));
- }
...
- printConfig();
+ LOGGER.info("modifications = " + join(modders, " "));
```

Listing 14. Example of inlining method

0.7 Others

Initialization by constructors. Using constructors for initialization is a fundamental object-oriented programming practice that facilitates encapsulation and ensures proper setup of object state upon instantiation. By chaining constructors (using this), developers can simplify initialization logic, reduce redundancy, and enhance code maintainability. In Listing 15, constructor chaining is employed to streamline the initialization process. Instead of duplicating field assignments across multiple constructors, the this keyword is used to delegate initialization to another constructor. This approach eliminates repetitive code and centralizes initialization logic.

```
public ExecutorSetting(Integer checkpoint, Integer parallelism, boolean useSqlFragment,
     String savePointPath) {
-  this.checkpoint = checkpoint;
-  this.parallelism = parallelism;
-  this.useSqlFragment = useSqlFragment;
-  this.savePointPath = savePointPath;
+  this(checkpoint, parallelism, useSqlFragment,
+   savePointPath, null, null);
   }
public ExecutorSetting(Integer checkpoint, Integer parallelism, boolean useSqlFragment,
     String savePointPath, String jobName, Map<String, String> config) {
-  this.checkpoint = checkpoint;
-  this.parallelism = parallelism;
-  this.useSqlFragment = useSqlFragment;
-  this.savePointPath = savePointPath;
-  this.jobName = jobName;
-  this.config = config;
+ this(checkpoint, parallelism, useSqlFragment,
+  false, false, savePointPath, jobName, config);
```

Listing 15. Example of using constructor to initialize

Coding style reformation. Reforming coding style is an effective technique for reducing the number of lines in a program while retaining its functionality. This practice contributes to cleaner and more compact code, making it easier to read and maintain. Simple adjustments, such as converting multi-line conditional statements into single-line expressions, can significantly streamline the codebase. In Listing 16, a multi-line if block is reformatted into a single line, demonstrating how concise syntax reduces code lines without altering behavior.

```
- if (posts == null) {
-   posts = new ArrayList<Post>();
- }
+ if (posts == null) posts = new ArrayList<Post>();
```

Listing 16. Example of code style reformat

Merging imports. Merging imports is a common technique used to condense code by consolidating multiple import statements from the same package into a single statement. This approach reduces redundancy and improves the cleanliness of the code, especially in cases where numerous classes from the same package are utilized. In Listing 17, several import statements from the java.util package are merged into a single wildcard import (java.util.*). This reduces the number of lines significantly while preserving functionality.

```
- import java.util.ArrayList;
- import java.util.Collections;
- import java.util.Iterator;
- import java.util.List;
- import java.util.Map;
- import java.util.WeakHashMap;
+ import java.util.*;
```

Listing 17. Example of merging imports