# How Weak Reference is Used in Java Projects?

Yoshiki Higo*, Shinsuke Matsumoto*, Taeyoung Kim† and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University, Japan
{higo, shinsuke, kim-tyng, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Many programming languages have a system of garbage collection to automate memory management. Problems such as unexpected memory leak still may occur due to the automation of memory management. Weak reference has been proposed as a solution to such problems. However, the use of weak reference often requires consideration of memory release timing, which is difficult for developers. In this study, we investigate the use of weak reference on open source projects. More concretely, we investigated the domain of software projects where the weak reference was used, the timing of the introduction of the weak reference, the usage method, the presence or absence of the test code for weak reference, and the case of failure to introduce weak reference in the Java language. The survey showed that weak references were used in 73 repositories, about one-third of the total 202 repositories, and that weak references were most common in test code.**

***Index Terms*—Weak reference, Garbage collection, Java, GitHub**

## I. Introduction

Garbage collection (in short, GC) is a system for automated memory management used in many programming languages [1]. GC liberates developers from managing memory themselves because GC automatically collects unnecessary objects. Whether an object is unnecessary or not is determined by reachability from other objects. Objects referenced by other objects are reachable and are not collected because they are judged not to be unnecessary. Objects with no references from other objects are unreachable, and GC collects such objects as unnecessary objects.

However, there is still a problem in the memory management by GC. If objects that will not be used in the future are reachable from other objects, they are not collected by GC. As a result, unused objects remain in memory, causing a memory leak.

In GC, developers cannot specify a specific object to instruct forced collection. As a way of specifying objects to be collected by developers, reference called **weak reference** (in short, **WR**) has been proposed [2]. WR is treated differently from strong reference[1] in the GC mechanism. Objects referenced only by WR are treated as unreachable and are collected by GC. Consequently, by referring to an object using WR, it can be specified as a GC target.

However, there are some considerations for using WR effectively. First, developers need to consider the timing at which weakly referenced objects will be collected. Objects having at least one strong reference from other objects are not collected, and GC collects objects referenced only by WR. Thus, it is necessary to predict when an object will be referenced only from WR or when all strong references to the object will be lost, and design the software so that the object can be collected when it is no longer necessary. Next, we need to consider the timing at which GC collects the objects referenced only by WRs. Instead of being collected as soon as the object becomes a GC target, GC runs at regular intervals. When collecting an object by GC, a null pointer is inserted in the WR. If developers use a WR without considering such timing, an unexpected error will occur.

[1]Ordinary reference is also called strong reference in this paper, as opposed to WR.

```
 1 WeakReference<Socket> ref
     = new WeakReference<Socket>(new Socket());
   ...
90 Socket s = ref.get();
91 if (s == null) {
92   // in the case that the object has been collected
93 }
94 else {
95   // in the case that the object is still in the heap
96 }
```

Fig. 1: An example of class WeakReference

Various studies performed data mining on GitHub repositories [3], [4]. Some research tried to solve the problem of Java memory leak [5], [6]. However, there is no study of substantial investigation on the use of WR. In this study, we investigated how WR is used in open source software written in Java. More concretely, we investigated the domain of Java projects where WR was used, the WR's introduction time, the method of use, the existence of the test, and the case of the failure.

## II. Preliminaries

### A. Weak Reference in Java

The following two classes are provided for WR in the Java standard library.

- `java.lang.ref.WeakReference<T>`
- `java.util.WeakHashMap<K,V>`

The former is a class of WR for type `T`. Figure 1 shows an example of how to use class `WeakReference`. The latter and the normal hash map (`java.util.HashMap<K,V>`) share almost the same functions. The difference is when a weakly referenced key object is collected, the reference to the value object of the pair is lost, and if there is no strong

2640-0715/20/$31.00 ©2020 IEEE
DOI 10.1109/APSEC51365.2020.00023

reference to the value object, it is collected together with the key object. Key-value pairs of key objects that are no longer necessary are automatically removed from the WeakHashMap. This has the advantage that developers do not have to worry about deleting unnecessary key-value pairs.

### B. Target Projects

The experimental targets are Java projects on GitHub. In this study, to objectively investigate the domains of projects that use WR, we selected 202 Java projects from the targets of Borges' experiment [3]. Of the 202 projects, 73 projects used WRs.

In some of this study, it was necessary to look manually at the code and change history. We picked up the top 10 projects with the highest stars out of the 73 projects that used WRs. Table I shows project names under the investigation and the number of stars at the end of 2018.

## III. Research Questions

We set up six research questions. By clarifying the answers to these RQs, we try to understand the actual use of WR. We also describe our answers to the research questions here.

**RQ1: in which domain WR is used?**

Our aim is to understand which domain projects use WR and what features the projects have.

Our Answer

Of the 202 projects targeted, the 73 repositories used WR, which was unexpectedly high. Among them, WR was often used in *System software* and *Software tools* domains. The authors consider that WR is often introduced where there is a strong demand for software performance.

**RQ2: when WR is introduced?**

We investigate when WR is introduced in each project to understand the purpose for which WR is mainly introduced. The introduction timing is classified into two, (1) implementing a new function and (2) improve/maintain an existing function.

TABLE I: Top 10 stars projects

| Project | Stars |
|---|---|
| ReactiveX/RxJava | 36,206 |
| elastic/elasticsearch | 35,955 |
| spring-projects/spring-boot | 30,868 |
| square/okhttp | 29,609 |
| google/guava | 27,988 |
| PhilJay/MPAndroidChart | 24,799 |
| spring-projects/spring-framework | 24,612 |
| bumptech/glide | 23,869 |
| square/leakcanary | 21,061 |
| zxing/zxing | 20,623 |

Our Answer

WR was mainly introduced when creating new source files, that is, when adding new functions to software.

**RQ3: how WR is used?**

We try to find out how WR is used. By knowing how to use WR in real-world situations, we can understand WR's primary uses and the typical code patterns that use WR.

Our Answer

The use of WR in test code is an unexpected result for the authors. In particular, the most common use of WR in test code may mean that test patterns using WR were already widely known for developers. Other usage patterns of WR are described in Subsection IV-C. Such patterns will be useful for developers who are not familiar with WR.

**RQ4: is the code using WR tested?**

Making test code is essential in software development. Thus, we investigate whether the code using WR is tested sufficiently or not.

Our Answer

Code using WR was not tested much. The authors consider that there is no test for WR because the pattern of the code for testing WR is not well known, rather than the difficulty of using WR.

**RQ5: do developers stop using WR?**

To determine whether using WR is difficult, we investigate whether there are any cases where developers stopped using WR due to unexpected problems.

Our Answer

There were some cases where developers stopped using WR due to unexpected problems. To avoid such problems, the introduction of WR should be determined with great care.

**RQ6: is using weak reference effective for performance?**

To check whether WR is effective for performance, we conducted a performance test on an open source projects. We investigate whether there is any performance difference between the existing code using WR and the code modified not to use WR.

Our Answer

We conducted experiments using open source software and confirmed that WR was effective for software performance. In applications with all references to image data, abnormal termination occurred even in the original library where WR was used. Still, it is not common processing to load 1.6 GB of image data at once and do something with such data. In mobile applications, it is more common to repeat loading and unloading for a few images. In such a case, the existing library using WR worked well, and the code modified without WR terminated abnormally. The authors considered that WR is an effective way to automatically collect unnecessary objects and prevent a memory leak from those results.

## IV. Experiment

### A. RQ1: in which domain WR is used?

**Procedure:** the targets are the 202 Java projects that were used in the experiment by Borges et al. [3]. They classified the 202 projects into the following six domains.

- *Application software* (in short, *Application*)
- *System software* (*System*)
- *Web libraries and frameworks* (*Web*)
- *Non-web libraries and frameworks* (*Non-web*)
- *Software tools* (*Tools*)
- *Documentation* (*Doc.*)

We used GitHub's code search API[2] to determine whether WR was used in each project. A code search was performed using two keywords 'WeakReference' and 'WeakHashMap' for each project, and if the number of Java source files in the search results was one or more, it was treated as using WR.

**Results:** Figure 2 shows the results. ALL is the distribution of all the 202 Java projects, and WEAK is the distribution of the 73 projects using WR. WR is used in 44 projects in the *non-web* domain, which is the highest number; Regarding the ratio of WEAK/ALL, WR is used in more than half of the projects in the *System* and *Tools* domains.

### B. RQ2: when WR is introduced?

**Procedure:** we analyzed the change history of source files in the repositories where WR was used to find out when WR had been firstly introduced in the source files. The introduction time was classified into two cases, (1) when the source file was created and (2) when it was updated. We assume that WR is introduced for the following purposes. For the first case, WR was introduced when a new function was implemented. For the second case, WR was introduced when an existing function was improved/maintained.

[2]https://developer.github.com/v3/search/

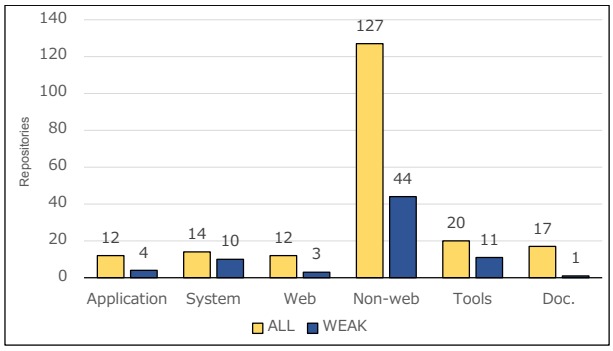


Fig. 2: Project domains

We defined introduction timing as when keywords of 'WeakReference' or 'WeakHashMap' appeared in the change history of the source files. If a keyword is included in the change history when the source file was created, we regarded that WR was introduced when the file was generated. Otherwise, if a keyword is included in any change histories after the source file was created, we regarded that WR was introduced when the file was updated.

**Results:** Table II shows the investigation results on introduction timing. For both `WeakReference` and `WeakHashMap`, the number of introducing WR in creating files is about three times as many as the number of updating files. Considering the assumptions, WR is often introduced to add new functions to software rather than improve/maintain existing functions.

### C. RQ3: how WR is used?

**Procedure:** the targets for RQ3 are the ten projects in Table I. A total of 95 source files used WR in those projects. We carefully read the code where WR to investigate how WR was used. We classified the usages of WR into the following six categories:

- *Testing*,
- *Memory leak prevention*,
- *Caching*,
- *API implementation*,
- *Comment*, and
- *Unnecessary processing avoidance*.

*Testing* means that WR is used in test code. *Memory leak prevention* means that unnecessary objects are collected by using WR. *Cache* represents a case where the results of calculations are temporarily stored using WR. *API implementation* means a case in which the function of WR is provided as an API as it is. For example, a class that inherits `WeakReference` is a typical case. *Comment* is not a code, but a WR keyword ('WeakReference' and

TABLE II: Introduction timing of WR

| | File creation | File update |
|---|---|---|
| WeakReference | 247 | 81 |
| WeakHashMap | 75 | 29 |
| Total | 322 | 110 |

```
1  @Test public void successDetaches() {
2    Disposable d = Disposables.empty();
3    WeakReference<Disposable> wr
       = new WeakReference<Disposable>(d);
4
5    TestObserver to = new TestObserver(d);
6    to.test();
7
8    d = null;
9    System.gc();
10   assertNull(wr.get());
11 }
```

(a) *Testing*

```
1  class LineChartRenderer {
2    WeakReference<Bitmap> mDrawBitmap;
3    void drawData() {
4      Bitmap bitmap = mDrawBitmap == null ?
         null : mDrawBitmap.get();
5      if (bitmap == null) {
6        bitmap = Bitmap.createBitmap();
7        mDrawBitmap = new WeakReference<>(bitmap);
8      }
9      drawBitmap(bitmap);
10   }
11 }
```

(b) *Memory leak prevention*

```
1  class SizeDeterminerLayoutListener {
2    WeakReference<SizeDeterminer> sizeDeterminerRef;
3
4    SizeDeterminerLayoutListener(SizeDeterminer sd){
5      sizeDeterminerRef = new WeakReference<>(sd);
6    }
7    boolean onPreDraw() {
8      SizeDeterminer sd = sizeDeterminerRef.get();
9      if (sd != null) sd.check();
10     return true;
11   }
12 }
```

(c) *Unnecesarry processing avoidance*

Fig. 3: Code samples using WR

'WeakHashMap') appearing in comments. *Unnecessary processing avoidance* means a case where once a weakly-referenced object is collected, the processing related to that object is not performed.

**Results:** When WR is used multiple times in a file, each usage is counted individually. Table III shows that WR is most often used in test code. We show some typical code examples. Figure 3(a) shows an example of using WR for *Testing*. The code checks whether the object referred by `to` does not cause memory leak after a given input `d` is processed. The WR contains a null pointer after the reference object is collected. Thus, GC invoked by `System.gc()` makes the value of the WR (`wr.get()` on line 10) a null pointer if memory leak does not exist. Developers can check the existence of memory leak with this test code.

Figure 3(b) shows the code that a weakly references object `bitmap` as an intermediate product to prevent a memory leak. If the intermediate object is collected and the WR is a null pointer, the code recreates a new object before starting the process and recreate a WR to the new object. Using WR, the intermediate product's object can be collected, and the memory leak can be prevented. Moreover, if the same processing is performed before the intermediate product's object is collected, the calculation can be saved because the object does not need to be created.

In the case of *Caching*, the pattern of the code is the same as *Memory leak prevention*. The difference is that the object of the intermediate product is weakly referenced in the case of *Memory leak prevention*, while the object of the calculation results is weakly referenced in the case of *Caching*. By weakly referencing the calculation results' object, if there are cached results, the calculation can be saved. Besides, it is possible to prevent memory leak from occurring for objects with the same calculation results.

Figure 3(c) shows example code that avoids unnecessary processing. Weakly referring to object `sd` given as input, and if the object has been collected, GC does not judge that the processing on the object (the 9th line in this figure) is unnecessary.

TABLE III: Purposes of using WR

| Purpose | # of using WR |
|---|---|
| Testing | 37 |
| Memory leak prevention | 25 |
| Caching | 17 |
| API implementation | 13 |
| Comment | 13 |
| Unnecessary processing avoidance | 6 |

### D. RQ4: is the code using WR tested?

**Procedure:** the targets for RQ4 are the ten projects in Table I. Out of the total of 95 source files that used WRs, 58 were not test code. We checked whether test code existed for each of the 58 source files. If test code existed, we also checked whether the test code tests the use of WR or not.

**Results:** Test code existed for 38 files out of the 58 source files. We checked all the test code for the 38 source files and found that WR tests existed for only ten source files, which is only about one-sixth of the 58 source files, including WR's usage.

### E. RQ5: do developers stop using WR?

**Procedure:** the targets for RQ5 are the ten projects in Table I. We manually examined the change history including the keywords of WR ('WeakReference' and 'WeakHashMap') for each project to see if there were any cases in which the use of WR was stopped. In this study, we used the fact that the WR classes were no longer used as a criterion for stopping WR's use.

TABLE IV: Failure cases of WR introducetion

| Project name | # of failures |
|---|---|
| ReactiveX/RxJava | 1 |
| spring-projects/spring-framework | 2 |
| bumptech/glide | 2 |
| Total | 5 |

**Results:** Table IV shows the project names where the introduction of WR failed and the number of such cases in each project. Three of the ten projects failed to introduce WR. Besides, there were five cases where the introduction of WR failed in the three projects.

`RxJava` and `Glide` failed to introduce class `WeakReference`. The failure was that the weakly referenced object was collected earlier than expected, which caused the processing related to the object not executed correctly. In other words, the problem occurred because developers used WR without considering the collection timing of the WR object.

In `Spring-framework`, introduction of `WeakHashMap` failed. The purpose of using `WeakHashMap` was caching the calculation results to increase the execution speed. However, this class cannot be used in multiple threads, and the execution speed is rather slow in a multi-core processor system. In this case, the problem occurred because developers did not understand the features of `WeakHashMap` deeply, not because of the difficulty in using WR.

### *F. RQ6: performance test*

**Procedure:** project 'bumptech/glide' was selected as the object of this experiment. Glide[3] is an image loading library for Android. This library internally has a cache for image data using WR. Original Glide code using WR and a modified version not using WR are executed in the same situation. By comparing the performance of the two, we check whether the WR is useful. In the performance test, the following two types of Android applications using Glide were executed:

1) an application with reference to image data, and
2) an application with no reference to image data.

The application 1) has strong references to all image data when loading the entire image. The application 2) deletes the reference to the image as soon as the image is loaded. That is, the application has no reference to the image data.

The images to be loaded were 400 JPEG images with an average of 4 MB, and the total size was about 1.6 GB. The images were loaded via HTTP, and a server was set up to distribute image data using software called HFS[4]. The emulator's environment to execute the application is Android 8.1 (Oreo) OS and 1.5GB RAM.

**Results:** The experimental results are shown in Table V. Table V shows that all three "aborted" results were the same. After loading 200 images, the application was forcibly terminated immediately after the memory exceeded 1 GB. The cause of such termination is considered to be that the object of the image data referred to by the application was not collected at a proper time, and the memory overflowed.

"completed" in the results in Table V is a combination of an original Glide that uses WR and an application that has no reference to image data. Here, "completed" means that all images have been successfully loaded. The "complete" was possible because the image data objects were properly collected, and the memory leak did not occur because the references to the image data were only WR in `Glide`.

[3]https://github.com/bumptech/glide
[4]http://www.rejetto.com/hfs/

TABLE V: Results of performance test

| Application | Original Glide | Modified Glide |
|---|---|---|
| with reference | aborted | aborted |
| without reference | completed | aborted |

## V. Threats to Validity

In this study, the code and change history were manually inspected. Although we have not made much quantitative interpretation of the experimental results, the answers to the research questions are qualitative, such as the type of WR usage and the case of failing to introduce WR. Therefore, it is considered sufficient to check the code and change history in this study manually.

## VI. Conclustion

In this study, we investigated the use of weak reference in Java projects. We set up six RQs, domains where weak reference was used, timing of introducing weak reference, purposes of using weak reference, testing for weak reference, cases of weak reference introduction failure, and impacts on performance were examined.

As future work, we plan to define code patterns of weak reference usage by further investigating the code that uses weak reference. We will also try to automate the introduction of weak reference by developing a mechanism of suggesting the use of weak reference to developers by using defined code patterns.

## References


[1] P. R. Wilson, "Uniprocessor garbage collection techniques," in *Memory Management*. Springer, 1992, pp. 1–42.

[2] "Reference Objects and Garbage Collection," http://pawlan.com/monica/articles/refobjs, accessed: 2020-01-05.

[3] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *the 32nd International Conference on Software Maintenance and Evolution*, 2016, pp. 334–344.

[4] C. Vendome, M. Linares-Vasquez, G. Bavota, M. D. Penta, D. German, and D. Poshyvanyk, "License usage and changes: A large-scale study of java projects on github," in *the 23rd International Conference on Program Comprehension*, 2015, pp. 218–228.

[5] W. De Pauw and G. Sevitsky, "Visualizing reference patterns for solving memory leaks in java," in *European Conference on Object-Oriented Programming*. Springer, 1999, pp. 116–134.

[6] R. Shaham, E. K. Kolodner, and S. Sagiv, "Automatic removal of array memory leaks in java," in *the 9th International Conference on Compiler Construction*, 2000, p. 50–66.