特別研究報告

題目

オープンソースソフトウェアにおける弱参照の利用実態の調査

指導教員
楠本 真二 教授

報告者
Taeyoung Kim (キム テヨン)

平成 31 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 30 年度 特別研究報告

オープンソースソフトウェアにおける 弱参照の利用実態の調査

Taeyoung Kim (キム テヨン)

内容梗概

多くのプログラミング言語ではガベージコレクションを利用してメモリ管理を自動化している．しかしメモリ管理の自動化によって予期せぬメモリリークなどの問題が発生する場合があり，解決策として弱参照 (Weak reference) が提案されている．しかしながら，メモリ管理が自動化されているシステムでの弱参照の利用には，メモリ解放のタイミングなど考慮すべきことが多くあり，開発者にとって難しさを伴う．したがって本研究ではオープンソースソフトウェアのホスティングサービスである GitHub を用い，弱参照の利用実態を調査した．本研究では Java 言語を対象として，弱参照が導入されているリポジトリのドメイン及び弱参照の導入時期，利用方法，テストの有無，導入失敗のケースを調査した．調査により，全体 202 リポジトリの約 3 分の 1 である 73 個のリポジトリで弱参照が利用されており，また弱参照の使い方としてテストコードでの利用が最も多いことが示された．さらに，弱参照を利用しているオープンソースソフトウェアに対して，弱参照が実際に有効かの実験を行った．実験により，弱参照の利用でメモリリークを防止できることが証明された．

主な用語

弱参照，ガベージコレクション，Java，GitHub

目 次

図 目 次

表 目 次

1 まえがき

ガベージコレクション (Garbage Collection) は Java などの多くのプログラミング言語で利用されている，自動メモリ管理を行うためのシステムである [6, 10, 11, 18, 22]．ガベージコレクションを利用することによって，ソフトウェア開発者はメモリ管理について考える必要がなくなる．しかし，ガベージコレクションによるメモリ管理の自動化には問題点も存在する．

ガベージコレクションとは，不要になったオブジェクトを自動的に回収する仕組みである．オブジェクトが不要であるか否かは基本的に他のオブジェクトからの到達可能性で判断する．他のオブジェクトから参照されているオブジェクトは到達可能であり，不要ではないオブジェクトと判断されて回収されない．他のオブジェクトからの参照が存在しないオブジェクトは到達不能であり，このようなオブジェクトは不要なオブジェクトとしてガベージコレクションによる回収の対象となる．ガベージコレクションではこのような仕組みで不要なオブジェクトを自動検出し，オブジェクトの回収とメモリ管理を行っている．しかし，今後使われないオブジェクトが他のオブジェクトから到達可能である場合，すなわち他のオブジェクトからの参照が 1 つでもある場合が存在する．このような場合，実際には不要なオブジェクトであってもガベージコレクションからは不要であると判断しないので，オブジェクトは回収されなくなる．そのため，使われないオブジェクトがメモリ上に残る状態になり，**メモリリーク** (Memory leak) となる [7, 17]．

ガベージコレクションでは，他のオブジェクトからの参照の有無で回収対象のオブジェクトを自動検出しており，ソフトウェア開発者が特定のオブジェクトを指定して強制的な回収を指示できない．ソフトウェア開発者が回収させるオブジェクトを指定する方法として，**弱参照** (Weak reference) と呼ばれる参照が提案されている [1, 8, 14, 17]．弱参照は弱い参照という意味を持ち，ガベージコレクションの仕組み上で通常の参照と違う扱いをされる．通常の参照は，弱参照との対比で **強参照** (Strong reference) とも呼ばれる．弱参照のみで参照されているオブジェクトは到達可能でないオブジェクトとして扱われ，ガベージコレクションの回収対象と判断される．よって，弱参照を使ってオブジェクトを参照することで，ガベージコレクションの対象として指定することができる．

弱参照を有効に使うためにはいくつか考慮すべき点が存在する．まず，弱参照しておいたオブジェクトが回収の対象になるタイミングを考えなければならない．他のオブジェクトからの強参照が 1 つでもあるオブジェクトは回収の対象にならず，弱参照のみから参照されるオブジェクトが実際にガベージコレクションによる回収の対象となる．したがって，オブジェクトが弱参照のみから参照される，あるいはオブジェクトへの強参照が全部なくなるタイミングを予測し，オブジェクトが不要になった時に回収の対象となるようにソフトウェアを設計する必要がある．次に，弱参照を用いて回収の対象となったオブジェクトがガベージコレクションによって実際に回収されるタイミングも考えなければならない．オブジェクトが回収対象になったらすぐに回収されるのではなく，一定の時間ごとガベージコレクションが動く時に回収が行われる．ガベージコレクションによるオブジェクトの回収時，弱参照にはヌルポインタ (Null Pointer) が入るので，このようなタイミングを考えずに弱

参照を使うと，予期せぬエラーが起きることになる．

ガベージコレクションを採用しているプログラミング言語でソフトウェア開発を行う場合，自動メモリ管理という利点のため，一般的な開発者は参照やオブジェクトの回収など，メモリ管理に関してはあまり考えない．よって，参照がなくなるタイミングとオブジェクトが回収されるタイミングを予測し，不要になったオブジェクトを適切な時点で回収できるように弱参照を使うことは，一般的な開発者にとって相当な難しさを伴うと著者は考える．ソフトウェア開発者が弱参照を導入しようとする場合，弱参照に対する理解も必要であるが，弱参照が実際にどう使われているかの実例も重要である．また，弱参照の導入がいつ，どのように行われるかなど，利用実態を知ると弱参照の導入が更に容易になるのであろう．しかし，弱参照の利用実態についての調査は著者が知る限りない．

GitHub のリポジトリに対してデータマイニングを行った様々な調査研究がある [5, 9, 13, 16, 21]．また，Java のメモリリークの問題を解決しようとして行った研究もいくつか存在している [7, 12, 17, 19, 23]．しかし GitHub のリポジトリを対象に，メモリリークの解決策として提案されている弱参照の利用実態を調べた研究は存在していない．そこで，弱参照の利用実態に関する調査の必要性があると考え，本研究を始めた．

本研究では，ガベージコレクションを採用している Java で作成されたオープンソースソフトウェアにおける弱参照の利用実態を調査した．調査では，弱参照が使われているリポジトリのドメインと弱参照の導入時期，利用方法，テストの有無，導入失敗のケースを調べた．さらに 1 つのオープンソースソフトウェアに対して実験を行い，弱参照の利用が有効かを調べた．

```
 1 WeakReference<Socket> ref
     = new WeakReference<Socket>(new Socket());
   ...
90 Socket s = ref.get();
91 if (s == null) {
92   //オブジェクトが回収された場合（通信終了の状態）
93 }
94 else {
95   //通常の場合（通信待ちの状態）
96 }
```

図 1: WeakReference のコード例

2 準備

2.1 Java における弱参照

Java では Java Development Kit 1.2 バージョン (J2SE 1.2) から以下の 2 つのクラスを提供し，Java の開発者が弱参照を利用できるようにしている [2, 3, 14].

- `WeakReference<T>`
- `WeakHashMap<K,V>`

`WeakReference<T>` は，`T` 型オブジェクトの弱参照を表すクラスである．`WeakReference<T>` を使ったコード例を図 1 に示す．弱参照しておいたオブジェクト (図 1 の 1 行目の `new Socket()`) がガベージコレクションによって回収された場合，弱参照にヌルポインタが入る．したがって，図 1 の 91 行目のように，弱参照の参照先がヌルポインタであるか確認を行い，使おうとするオブジェクトが回収された場合と回収されなかった場合で条件分岐することが望ましい．

`WeakHashMap` は Java で提供されているハッシュマップ (`java.util.HashMap`)[1] と機能はほぼ同じである．違いは，弱参照されている Key オブジェクトが回収されると，ペアの Value オブジェクトへの参照がなくなり，Value オブジェクトへの強参照がなければ Key オブジェクトと一緒に回収されることである．また，不要になった Key オブジェクトの Key-Value ペアは，ガベージコレクションによって `WeakHashMap` から自動的に除去されるので，開発者が不要なペアの削除について考えなくても良いという利点がある．

[1] `https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html`

2.2 調査対象のリポジトリ

調査対象は GitHub[2] 上に公開されている Java のリポジトリである．本研究では，弱参照が利用されているリポジトリのドメインを客観的に調査するために，Borges らによる調査データ[3]のリポジトリのうち，202 個の Java リポジトリを調査対象として定めた [5]．そのうち弱参照が利用されているリポジトリは 73 個であった．

一部の調査ではコードや更新履歴を直接見ながら調べる必要があったので，弱参照が利用されている 73 個のリポジトリのうち，Stars 数上位 10 個のリポジトリのみを対象として調査を行った．表 1 に調査対象のリポジトリ名と Stars 数を示す．Stars 数は 2018 年末時点の値である．

表 1: 調査対象のリポジトリ

リポジトリ名	**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

[2] https://github.com/
[3] https://goo.gl/73Sbvz

3 Research Questions

調査にあたり，6 つの Research Question (RQ) を設定した．本研究の調査よりこれらの RQ に対する答えを明らかにすることで，弱参照の利用実態を深く理解できると考える．

RQ1: どのようなドメインのリポジトリでよく弱参照を利用しているか 弱参照がどのようなドメインのリポジトリでよく利用されているかを知り，またそのリポジトリで開発されているソフトウェアがどのような特徴を持っているかを理解する．

RQ2: 弱参照が導入される時期はいつか 各々のリポジトリで弱参照の導入時期を調べ，弱参照が主にどのような目的で導入されるかを理解する．導入時期は 2 つに分類し，弱参照の導入目的がソフトウェアの機能の実現のためであるか，もしくはソフトウェアの機能の改善・保守のためであるかを調べる．

RQ3: 弱参照をどのように使っているか 弱参照がどのように利用されているかを調べる．実場面で弱参照がどのように使われているかがわかると，弱参照の主な利用方法を理解でき，また弱参照を利用する典型的なコードのパターンを知ることができる．

RQ4: 弱参照はテストされているか ソフトウェアの品質を高めるなどの理由で，テストコードの作成はソフトウェアの開発において必須であろう [4, 15]．したがって，本研究では弱参照が使われているコード片をテストする，テストコードが存在するかを調べる．

RQ5: 弱参照の導入に失敗したケースはあるか 弱参照の利用が実際に難しいかを確認するために，弱参照を導入した後，ソフトウェア開発者が予期できなかった問題で弱参照の導入をやめたケースがあるか調べる．

RQ6: 弱参照の利用は実際に有効か 弱参照の利用が有効かを調べるために，1 つのオープンソースソフトウェアを対象にパフォーマンステストを行う．弱参照を利用した既存のコードと弱参照を利用しないように改変したコードの間で，何らかの違いがあるか確認する．

4 調査方法

4.1 RQ1: リポジトリのドメイン

調査対象は 2.2 節で述べた 202 個の Java リポジトリである．

リポジトリのドメインに関しては Borges らによる調査データを用いた [5]．ドメインは以下の 6 つに分類されている．括弧内は略記である．

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

各リポジトリで弱参照が利用されているか否かは GitHub のコード検索 API[4] を使って調べた．リポジトリごとに “WeakReference” と “WeakHashMap” のキーワードでコードの検索を行い，検索結果に Java のソースファイルの数が 1 個以上あれば，弱参照が使われているリポジトリとして扱った．

4.2 RQ2: 導入時期

調査対象は 2.2 節で述べた 202 個の Java リポジトリのうち，弱参照が利用されている 73 個のリポジトリである．

弱参照が利用されているリポジトリでのソースファイルの更新履歴を解析し，ソースファイルに弱参照が初めて導入された時期を調べた．導入時期はソースファイルの生成時と更新時の 2 つに分類し，以下の目的で弱参照が導入されたと仮定した．

- ファイルの生成時: ソフトウェアへの機能の追加
- ファイルの更新時: 既に存在する機能の改良・保守

導入時期はソースファイルの更新履歴で “WeakReference” か “WeakHashMap” のキーワードが登場する時期と定めた．ソースファイルが初めて作成された時の更新履歴でキーワードが含まれていれば，ファイルの生成時に弱参照が導入されたとする．そうではなく，ソースファイルの生成後のいずれかの更新履歴でキーワードが含まれていれば，ファイルの更新時に弱参照が導入されたとする．

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

4.3 RQ3: 使い方

調査対象は表 1 の 10 個のリポジトリで，弱参照が利用されている合計 95 個のソースファイルである．

弱参照が使われているコードを直接読み，弱参照がどう使われているか調べた．弱参照の使い方を以下の 6 つに分類した．

- テスト
- メモリリークの防止
- キャッシュ
- API 実装
- コメント
- 不要な処理の除去

“テスト” は弱参照がテストコードに利用されている場合である．“メモリリークの防止” は不要なオブジェクトを弱参照の利用によって回収させる場合である．“キャッシュ” は弱参照を使って，計算結果などをキャッシュ (一時的な保存) する場合である．“API 実装” は弱参照の機能をそのまま API として提供しようとする場合であり，例え `WeakReference` を継承したクラスが代表的な例である．“コメント” はコードではなく，コメントに弱参照のキーワード (“WeakReference” と “WeakHashMap”) が現れている場合である．“不要な処理の除去” は，弱参照されているオブジェクトが回収されたら，そのオブジェクトに関する処理を不要であるとみなして行わない場合である．

4.4 RQ4: テストコードの存在

調査対象は表 1 の 10 個のリポジトリで，弱参照が利用されている全体 95 個のソースファイルのうち，弱参照がテストとして使われていない 58 個のソースファイルである．

各ソースファイルで実装されているクラスがテストされているかを確認するために，クラス名と “Test” というキーワードでコードの検索を行った．また，弱参照がテストコードによってテストされているかを調べるために，テストコードを直接読んで調査を行った．

4.5 RQ5: 導入失敗のケース

調査対象は表 1 の 10 個のリポジトリである．

リポジトリごとに弱参照のキーワード (“WeakReference” と “WeakHashMap”) が含まれている更新履歴を目視で調査することで，弱参照の導入をやめたケースがあるか調べた．

4.6 RQ6: 実験調査

実験の対象としてリポジトリ `bumptech/glide` のコードを利用した．Glide[5] は Android 用の画像ローダのライブラリである [20]．このライブラリでは，弱参照を利用して画像データへのキャッシュを内部で持つ．Glide の既存のコードと，既存の Glide から弱参照を使わないように改変したコードを用いて，両者の実験を行った．両者のパフォーマンスを比較することで，弱参照が実際に有効かを調べる．

パフォーマンステストでは Glide を利用する以下の 2 種類の Android アプリを実行させた．

- 画像データへの参照あり
- 画像データへの参照なし

“画像データへの参照あり” のアプリは，画像全体のロードにおいて全ての画像データへの強参照を持つアプリである．“画像データへの参照なし” のアプリは，画像がロードされたらすぐにその画像への参照を消すアプリである．すなわち，アプリ側では画像データへの参照を持たない．

ロードさせる画像は平均 4MB の 400 個の JPEG 画像であり，全体の容量は 1.6GB 程度であった．画像のロードは HTTP を介して行い，HFS[6] というソフトウェアを用いて画像データを配信するサーバを立てた．

アプリの実行に用いたエミュレータの環境は，OS は Android 8.1 (Oreo) であり，RAM は 1.5GB である．

[5] `https://github.com/bumptech/glide`
[6] `http://www.rejetto.com/hfs/`

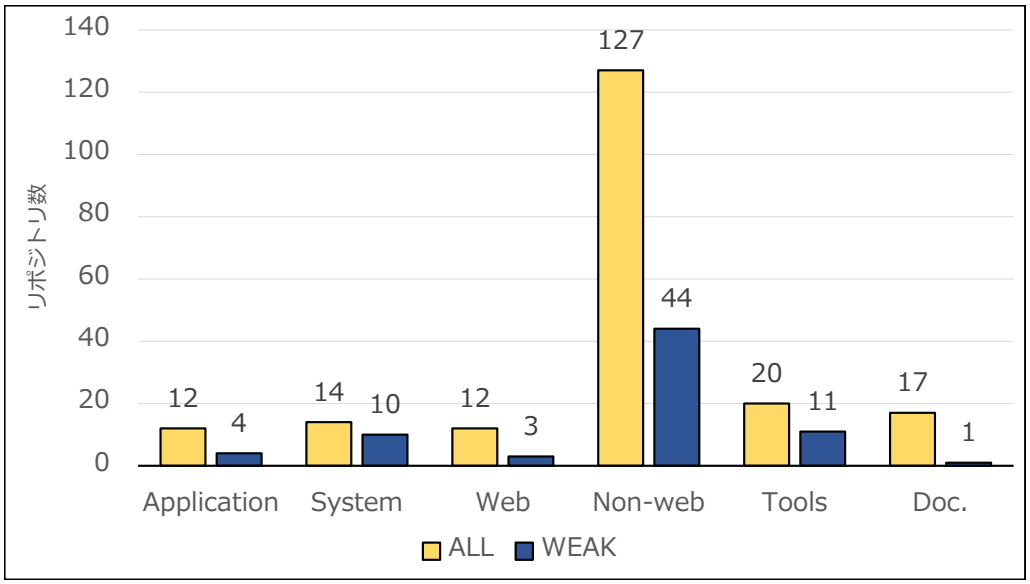

図 2: RQ1: リポジトリのドメイン

5 調査結果

5.1 RQ1: リポジトリのドメイン

ドメインの調査結果を図 2 に示す．図 2 で ALL は 202 個の Java リポジトリのドメインの分布であり，WEAK はそのうち弱参照が利用されている 73 個のリポジトリのドメインの分布である．ドメイン別のリポジトリの数を見ると，Non-web のドメインでは 44 個のリポジトリで弱参照が利用されており，最も多い結果となっている．全体からの割合 (WEAK/ALL) で見ると，System と Tools のドメインで，半数以上のリポジトリで弱参照が利用されている．ドメイン別の弱参照が利用されているリポジトリの例を表 2 に示す．

表 2: RQ1: ドメイン別のリポジトリの例

ドメイン	リポジトリ名
Application	HannahMitt/HomeMirror, k9mail/k-9
System	ReactiveX/RxJava, apache/kafka
Web	square/okhttp, spring-projects/spring-framework
Non-web	google/guava, PhilJay/MPAndroidChart
Tools	elastic/elasticsearch, square/leakcanary
Doc.	aporter/coursera-android

5.2 RQ2: 導入時期

導入時期の調査結果を表 3 に示す．単位はファイル数である．表 3 の結果を見ると，WeakReference と WeakHashMap ともにファイルの生成時の場合がファイルの更新時の場合より 3 倍程度多い．すなわち，弱参照は主にファイルの生成時に導入が行われることを調査結果から確認できる．仮定から考えてみると，弱参照はソフトウェアへの機能の追加のために導入される場合が多くて，既存機能の改良やソフトウェアの保守のために弱参照が導入される場合は少ないということが言える．

表 3: RQ2: 弱参照の導入時期

	ファイル生成時	ファイル更新時
WeakReference	247	81
WeakHashMap	75	29
合計	322	110

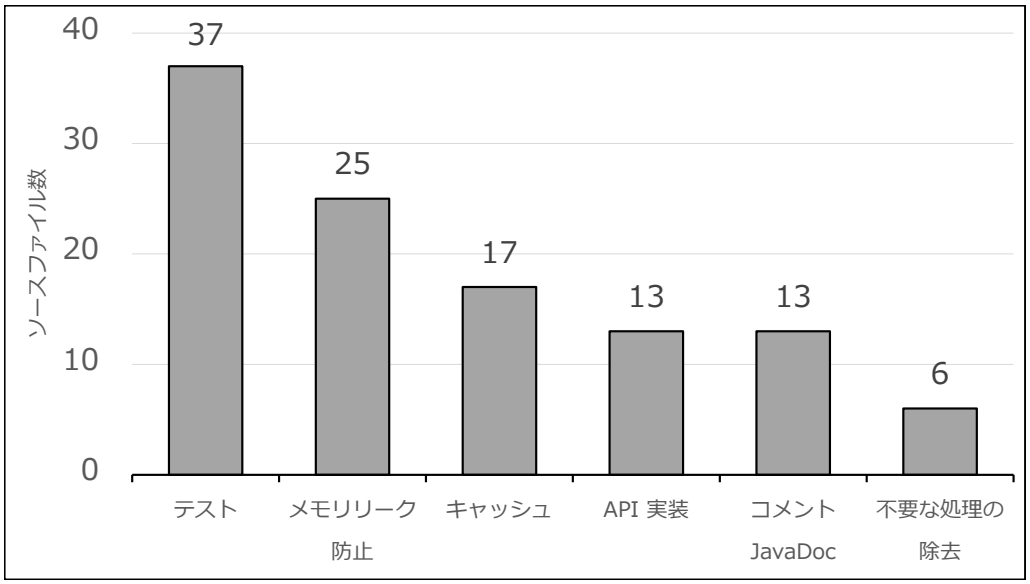

図 3: RQ3: 弱参照の使い方

5.3 RQ3: 使い方

使い方の調査結果を図 3 に示す．単位はファイル数である．1 つのファイルで複数の使い方をしている場合，重複があるとして，それぞれがカウントされるようにした．図 3 を見ると，弱参照はテストコードで最もよく利用されることがわかる．

各使い方における典型的なコードの例をいくつか示す．まず，テストとして弱参照を使う場合の例を図 4 の (a) に示す．図 4 の (a) は，テスト対象のオブジェクト (`TestObserver to`) が，入力 (`Disposable d`) を与えて処理を行った後，入力のオブジェクトに対しメモリリークを起こさないことをテストするコードである．弱参照には参照オブジェクトの回収後ヌルポインタが入るので，Java の API を使ってガベージコレクションを強制動作させ，弱参照の値 (10 行目の `wr.get()`) がヌルポインタであるか確認することで，メモリリークの存在をテストできる．

図 4 の (b) はメモリリークの防止のため，中間生成物としてのオブジェクト (`Bitmap bitmap`) を弱参照しておき，ガベージコレクションによって回収されるようにしているコードである．中間生成物のオブジェクトが回収されて弱参照がヌルポインタになっている場合は，処理を開始する前にオブジェクトを再生成し，またそのオブジェクトへの弱参照も作り直す．こうすることによって，中間生成物のオブジェクトが回収の対象となり，メモリリークを防止することができる．また，中間生成物のオブジェクトが回収される前に同じ処理を行う場合には，オブジェクトの再生成がいらなくなるので計算量を減少できる．

キャッシュの場合でもメモリリークの防止の場合とコードのパターンは同様であるが，メモリリークの防止では中間生成物のオブジェクトが弱参照されているとすれば，キャッシュでは計算結果のオブジェクトが弱参照されている．計算結果のオブジェクトを弱参照しておくことで，同じ計算を行っ

た場合にキャッシュされている結果があれば計算量を減少できる．また，計算結果を持つオブジェクトでメモリリークが発生することを防止できる．

図 4 の (c) は不要な処理を回避しているコードである．入力として与えられたオブジェクト (8 行目の `SizeDeterminer sd`) を弱参照しておき，オブジェクトが回収されていたら，そのオブジェクトへの処理 (図 4 の (c) の 9 行目) を不要であると判断し行わない．

```
@Test public void successDetaches() {
  Disposable d = Disposables.empty();
  WeakReference<Disposable> wr
    = new WeakReference<Disposable>(d);

  TestObserver to = new TestObserver(d);
  to.test();

  d = null;
  System.gc();
  assertNull(wr.get());
}
```

(a) テスト

```
class LineChartRenderer {
  WeakReference<Bitmap> mDrawBitmap;
  void drawData() {
    Bitmap bitmap = mDrawBitmap == null ?
      null : mDrawBitmap.get();
    if (bitmap == null) {
      bitmap = Bitmap.createBitmap();
      mDrawBitmap = new WeakReference<>(bitmap);
    }
    drawBitmap(bitmap);
  }
}
```

(b) メモリリークの防止

```
class SizeDeterminerLayoutListener {
  WeakReference<SizeDeterminer> sizeDeterminerRef;

  SizeDeterminerLayoutListener(SizeDeterminer sd){
    sizeDeterminerRef = new WeakReference<>(sd);
  }
  boolean onPreDraw() {
    SizeDeterminer sd = sizeDeterminerRef.get();
    if (sd != null) sd.check();
    return true;
  }
}
```

(c) 不要な処理の除去

図 4: RQ3: 弱参照の使い方の例

5.4 RQ4: テストコードの存在

各ソースファイルへのテストコードの有無に関する調査結果を表 4 に示す．全体 58 個のソースファイルのうち，38 個のファイルに対してテストコードが存在した．

テストコードが存在する 38 個のソースファイルに対して，弱参照がテストされているか調べた結果を表 5 に示す．弱参照のテストは 10 個のソースファイルに対して存在しており，弱参照が使われている全体 58 個のソースファイルの中，約 6 分の 1 程度しか弱参照のテストを行っていないことがわかった．

表 4: RQ4: テストコードの存在有無

テストの存在有無	ファイル数
有り	38
無し	20

表 5: RQ4: 弱参照のテストコードの存在有無

弱参照のテストの存在有無	ファイル数
弱参照のテスト有り	10
弱参照のテスト無し	28

5.5 RQ5: 導入失敗のケース

弱参照の導入に失敗したケースが存在するリポジトリ名と，各リポジトリに導入失敗のケースが何件あったかを表 6 に示す．10 個のリポジトリの中，弱参照の導入に失敗したケースがあるリポジトリは 3 個である．また，3 個のリポジトリで弱参照の導入に失敗したケースは合計 5 件あった．

`RxJava` と `glide` では，`WeakReference` の導入に失敗していた．失敗の理由は，弱参照しておいたオブジェクトが予想よりはやく回収され，行われるべき処理ができなくなったからである．すなわち，ソフトウェア開発者が弱参照によって参照されたオブジェクトの回収のタイミングをよく考えず，弱参照を利用したので問題が発生したケースである．

`spring-framework` では `WeakHashMap` の導入に失敗していた．`WeakHashMap` の導入目的は計算結果をキャッシュし，実行速度を速くすることであった．しかし，`WeakHashMap` は複数のスレッドで同時に利用できない問題があり，マルチコアプロセッサのシステムではむしろ実行速度が遅くなったので，弱参照の導入をやめたケースである．このケースは開発者が `WeakHashMap` の特徴を深く理解できなかったせいで問題が発生したケースで，弱参照の利用が難しいことが理由ではなかった．

表 6: RQ5: 弱参照の導入失敗のケース

リポジトリ名	ケース数
ReactiveX/RxJava	1
spring-projects/spring-framework	2
bumptech/glide	2
合計	5

5.6 RQ6: 実験調査

実験の結果を表 7 に示す．表 7 の結果で “異常終了” となっている 3 つは全て同じ結果となっていた．200 個の画像をロードした後，メモリが 1 GB を超えた直後にアプリが強制的に終了された．異常終了の原因は，アプリから参照されている画像データのオブジェクトが適切な時点で回収されず，メモリのオーバーフローとなったからであると考える．

表 7 の結果で “正常動作” は，弱参照を利用する既存の Glide と，画像データへの参照を持たないアプリの組合で出た結果である．ここで正常動作は，全ての画像データのロードに成功していることを言う．正常動作が可能だった理由としては，画像データへの参照が Glide からの弱参照しかなかったため，画像データのオブジェクトが適切に回収され，メモリリークとならなかったからであると考える．

表 7: RQ6: 実験の結果

アプリの種類	既存 (弱参照)	改変 (強参照)
画像データへの参照あり	異常終了	異常終了
画像データへの参照なし	正常動作	異常終了

6 まとめ

本章では各 RQ に対する答えとして，弱参照の利用実態に関する調査の結果と結果への考察をまとめる．

RQ1: どのようなドメインのリポジトリでよく弱参照を利用しているか 調査対象の 202 個のリポジトリの，約 3 分の 1 である 73 個のリポジトリで弱参照が使われていることは，予想外に多い結果であった．その中でも System software と Software tools のドメインで弱参照がよく使われていることは，ソフトウェアへの改善の要求が強いところで弱参照がよく導入されているからであると考えた．

RQ2: 弱参照が導入される時期はいつか 弱参照の導入はソースファイルの生成時，すなわちソフトウェアの機能の追加時に主に行われることがわかった．

RQ3: 弱参照をどのように使っているか 弱参照の使い方にテストコードでの利用があったのは予想外の結果であると考える．特に，弱参照の使い方でテストコードでの利用が最も多かったことで，弱参照をテストコードに用いるパターンが既に広く知られていたのかと考えた．他の使い方についても 5.3 節でコードのパターンを記載しており，弱参照を使おうとするソフトウェア開発者にとって非常に役立つであろうと考える．

RQ4: 弱参照はテストされているか 弱参照が使われているコードはあまりテストされていないことがわかった．弱参照へのテストがない理由は，弱参照の利用の難しさよりは，弱参照をテストするためのコードのパターンがあまり周知されていないためであると考える．

RQ5: 弱参照の導入に失敗したケースはあるか 5.5 節で紹介されているように，ソフトウェア開発者が予期できなかった問題で弱参照の利用をやめたケースがいくつか存在した．このような問題を起こさないために，弱参照の導入は十分注意を払って決定すべきであると考える．

RQ6: 弱参照の利用は実際に有効か オープンソースソフトウェアを利用して実験を行った結果，弱参照の利用が有効であることが確認できたと考える．画像データへの参照を全て持つアプリでは弱参照を使う既存の場合でも異常終了となったが，1.6GB の画像データを一括でロードし何かの処理をさせることは一般的でない．モバイル系のアプリでは数個程度の画像に対してロード・アンロードを繰り返すことがより一般的であろう．そのような場合で弱参照を使う既存のコードが正常動作し，弱参照を使わなく改変したコードが異常終了となった．この結果より，弱参照が不要なオブジェクトを自動的に回収させ，メモリリークを防止する有効な手段であることが示されたと考える．

7　妥当性への脅威

本研究ではコードや更新履歴を目視で調査した．調査結果に対して定量的な解釈はあまり行ってはいないが，弱参照の使い方の種類や，弱参照の導入に失敗したケースなど，研究の目的としての答えは定性的である．したがって，コードや更新履歴を目視で調査したことは本研究において十分であると考える．

8　あとがき

本研究では GitHub の Java リポジトリを対象に，弱参照の利用実態に関する調査を行った．調査では 6 つの RQ を設定し，弱参照が使われているリポジトリのドメインと弱参照の使い方，導入時期，テストの有無，導入失敗のケースに関して調べた．さらに 1 つのオープンソースソフトウェアに対して実験を行い，弱参照の利用が実際に有効であることを確認した．

今後の課題として，以下の 2 つが考えられる．

弱参照を導入できるコードのパターンの定義

弱参照が利用されているコードをさらに調査することで，弱参照を導入できるコードのパターンを定義する．定義されたコードのパターンを使うことで，メモリリークの問題の解決やパフォーマンスの改善など，あらゆる場合に対しても開発者が弱参照を容易に導入できるようにする．

定義されている弱参照のコードのパターンを開発者に提案する仕組みの開発

定義されているコードのパターンを用い，弱参照の利用をソフトウェア開発者に提案する仕組みを開発して，弱参照の導入を自動化する．コードのパターンが定義されているとしても，ヌルポインタによる予期せぬエラーが起きないように弱参照を利用するためには，まだ工夫が必要となる．したがって，ソフトウェア開発者が弱参照を導入しようとするコード片を解析し，そのコードに適切なコードのパターンを適用する仕組みを開発することで，弱参照の利用でエラーが起きないように弱参照の導入を自動化することを目指す．

謝辞

本研究を行うにあたって理解あるご指導を賜り，励まして頂きました，楠本真二教授に心より感謝申し上げます．

本研究の全過程を通して熱心かつ丁寧なご指導を頂きました，肥後芳樹准教授に深く感謝申し上げます．

本研究に関して貴重で有益なご助言を頂きました，柗本真佑助教に深く感謝申し上げます．

本研究を進めるにあたって様々な形で励まし，ご協力を頂きました，楠本研究室の皆様に心より感謝申し上げます．

最後に，本研究に至るまでの間，講義，演習，実験などでお世話になりました，大阪大学基礎工学部情報科学科の諸先生方にこの場を借りて心より御礼申し上げます．

参考文献

[1] Java Reference と GC (韓国語). https://d2.naver.com/helloworld/329631.
最終アクセス: 2019-02-04.

[2] WeakHashMap (JavaDoc).
https://docs.oracle.com/javase/8/docs/api/java/util/WeakHashMap.html.
最終アクセス: 2019-02-04.

[3] WeakReference (JavaDoc).
https://docs.oracle.com/javase/8/docs/api/java/lang/ref/WeakReference.html.
最終アクセス: 2019-02-04.

[4] SS Ahamed. Studying the feasibility and importance of software testing: An analysis. *arXiv preprint arXiv:1001.4193*, 2010.

[5] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. *2016 IEEE International Conference on Software Maintenance and Evolution*, pp. 334–344, 2016.

[6] Thomas W Christopher. Reference count garbage collection. *Software: Practice and Experience*, Vol. 14, No. 6, pp. 503–507, 1984.

[7] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in java. *European Conference on Object-Oriented Programming*, pp. 116–134, 1999.

[8] John Clarence Endicott, Daniel Rodman Hicks, Elliot Karl Kolodner, and Robert Carl Seemann. Computer system, program product and method of managing weak references with a concurrent mark sweep collector, 2000. US Patent 6,047,295.

[9] Oskar Jarczyk, Błażej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. Github projects. quality analysis of open-source software. *International Conference on Social Informatics*, pp. 80–94, 2014.

[10] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC, 2016.

[11] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*, Vol. 208. Wiley Chichester, 1996.

[12] Maria Jump and Kathryn S McKinley. Cork: dynamic memory leak detection for garbage-collected languages. *Acm Sigplan Notices*, Vol. 42, No. 1, pp. 31–38, 2007.

[13] Nora McDonald and Sean Goggins. Performance and participation in open source software on github. *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pp. 139–144, 2013.

[14] Lynn Monson. Caching & weakreferences. *JAVA Developer's Journal*, Vol. 3, No. 8, pp. 32–36, 1998.

[15] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[16] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. Security and emotion: sentiment analysis of security discussions on github. *The 11th working conference on mining software repositories*, pp. 348–351, 2014.

[17] Ju Qian and Xiaoyu Zhou. Inferring weak references for fixing java memory leaks. *2012 28th IEEE International Conference on Software Maintenance*, pp. 571–574, 2012.

[18] Jeffrey Richter. Garbage collection: Automatic memory management in the microsoft .net framework. *MSDN magazine*, Vol. 15, No. 11, pp. 82–92, 2000.

[19] Ran Shaham, Elliot K Kolodner, and Mooly Sagiv. Automatic removal of array memory leaks in java. *International Conference on Compiler Construction*, pp. 50–66, 2000.

[20] Yoo-jeong Song, Soo-bin Ou, and Jong-woo Lee. An analysis of existing android image loading libraries: Picasso, glide, fresco, auil and volley. *DEStech Transactions on Engineering and Technology Research*, 2016.

[21] Christopher Vendome, Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Daniel German, and Denys Poshyvanyk. License usage and changes: a large-scale study of java projects on github. *2015 IEEE 23rd International Conference on Program Comprehension*, pp. 218–228, 2015.

[22] Paul R Wilson. Uniprocessor garbage collection techniques. *Memory Management*, pp. 1–42, 1992.

[23] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. *The 30th international conference on Software engineering*, pp. 151–160, 2008.