既存Androidアプリケーションの実装状況に基づいた実装すべきメソッドの提示手法

薄井 駿　名倉 正剛　高田 眞吾

Android アプリケーションはイベント駆動型ソフトウェアである．開発者は Android API 内のコールバックメソッドを継承し内部を記述することでイベントに対する処理を定義する．処理を定義しない場合，アプリケーションは予想外の振る舞いをする可能性がある．本論文は既存プロジェクトを解析して得たコールバックメソッドの実装頻度と実装の共起関係を基に，開発者に実装すべきコールバックメソッドを提示する手法を提案する．

1 序論

Android アプリケーション（以下，Android アプリ）は，Android OS 上で動作するソフトウェアである．この Android アプリは処理を記述したファイルと，アプリの見た目のレイアウトを記述したファイルの 2 種類で構成される．一般的に処理を記述したファイルは Java で，レイアウトを記述したファイルは XML で記述する．Java ファイル内では通常の Java API の他に，Google の提供する Android フレームワークを利用することができる．

Android フレームワークには各種イベントに反応して実行するコールバックメソッドが含まれている．これを含むクラスを継承し処理を記述することで，開発者はイベントに対する Android アプリの振る舞いを定義できる．開発者がコールバックメソッドの実装を忘れた場合，Android アプリは想定外の振る舞いをする可能性がある．

本論文ではコールバックメソッドの実装し忘れを防ぐために，開発者が実装を行っている際にリアルタイムで実装すべきコールバックメソッドを提示する手法を提案する．これは既存の Android アプリの実装状況に基づき，開発者が実装していないコールバックメソッドの中で実装すべきものを特定することによって実現する．提案手法により，開発者にコールバックメソッドの実装漏れを認識させ，実装を行う機会を与えることができる．

本論文の構成は次のようになっている．2 章ではコールバックメソッドの実装と実装漏れによる問題を述べる．3 章では提案手法の概要および，各部分の詳細について述べる．4 章では提案手法の実装について述べる．5 章ではケーススタディについて述べる．6 章では関連研究について述べる．7 章では本論文の結論を述べる．

2 コールバックメソッドと実装の欠落の問題

Android アプリはイベント駆動型のソフトウェアである．これは実行開始後待機状態になり，特定のイベントが発生すると対応する処理を行い再度イベント発生を待つようなソフトウェアである．このイベントには画面のクリック操作やセンサーの値の変化，Android アプリの起動や画面遷移など様々な種類がある．これらのイベントに対し，Android フレームワークのライブラリはそれらを処理するためのコールバックメソッドのインタフェースを定義している．開発者はコールバックメソッドを含むクラスやインタフェースを基底クラスとして継承しメソッドに具体

Recommending methods based on implementations of existing Android applications

Hayato Usui, 慶應義塾大学, Keio University.
Masataka Nagura, 日本大学, Nihon University.
Shingo Takada, 慶應義塾大学, Keio University.

ソースコード 1　コールバックメソッドの実装の例

```
1 @Override
2 public void onCallStateChanged(int state, String
      incomingNum){
3   if(state==TelephonyManager.
        CALL_STATE_RINGING){
4     musicPlayer.pause();
5   }
6 }
```

的な処理を記述することで，個々のイベントに対する処理を定義できる．例えば，OnClickListener インタフェースを継承した場合には onClick メソッドを実装することにより画面内のボタンがクリックされた際の処理を定義でき，SensorEventListener インタフェースを継承した場合には onSensorChanged メソッドを実装することで端末のセンサーが新しく発生させたイベントに対する処理を定義することができる．

コールバックメソッドの実装を，音楽再生アプリにおける onCallStateChanged メソッドを例に述べる．onCallStateChanged メソッドは端末の着信状態の変化というイベントによって実行されるメソッドである．ソースコード 1 は，開発者が onCallStateChanged メソッドを実装したものである．3 行目は if 文によって着信状態をチェックしており，着信中の場合 4 行目の処理が実行される．4 行目ではアプリによる音楽の再生を停止するメソッドを実行する．このような実装により端末が着信中になった時 Android アプリは音楽の再生を停止する．

ソースコード 1 のようにコールバックメソッドが実装されている場合，Android アプリは開発者の意図した通りの振る舞いをする．しかしながら実装をしていない場合，想定外の振る舞いをする可能性がある．ソースコード 1 では，3 行目から 5 行目が無いと着信が来ても曲の再生が停止しなくなる．

IDE によっては開発者がインタフェースや抽象クラスを継承した時に，ソースコード 2 のようなメソッド宣言を自動的に挿入する機能を持つ． この場合，開発者は挿入されたメソッド宣言に対して必要な処理を記述することで実装する．このように，コールバックメソッドの実装忘れに対し気づきを与えることができる．しかし，コールバックメソッドの数が多くなる

ソースコード 2　自動的に挿入されたメソッド宣言の例

```
1 @Override
2 public void onCallStateChanged(int state, String
      incomingNum){
3 }
```

場合など，メソッド宣言だけ自動的に挿入されても，イベントを処理するためのメソッドの実装を忘れることもある．そのような場合も，結果として Android アプリが開発者の意図しない振る舞いをしてしまう．

3　提案手法

3.1　概要

2 章で述べた課題を解決するため，我々は既存のプロジェクトを解析し，既存アプリに対するメソッドの実装状況を基に，実装すべきコールバックメソッドを開発者に提示する手法を提案する．我々の事前調査[1]では，Android アプリのコールバックメソッドの実装状況には，次の特徴が存在することを明らかにした．

- 特徴 1：基底クラス／メソッドによって実装頻度が異なる場合がある
  ある基底クラスを継承してクラスを作成する場合，継承元の基底クラスによっては特定のコールバックメソッドに対する実装頻度が，他のメソッドに比べて有意に高いメソッドが存在した．
- 特徴 2：コールバックメソッド間で実装に共起関係が存在する
  あるクラスであるメソッドが実装されている場合に，別のクラスで特定のメソッドが実装される場合があった．

提案手法では，既存アプリに含まれるメソッドに対するこれらの実装状況を利用し，実装すべきメソッドを開発者に提示する．提案手法の処理の流れを図 1 に示す．提案手法はまずライブラリ解析部により，Android フレームワークで提供されるライブラリファイルからメソッドやクラスを取得し，クラス間の継承関係とメソッドの実装やオーバーライドの関係を解析する．次に，既存プロジェクト解析部はプロジェクトリポジトリから既存 Android プロジェクトのソースファイルを取得する．そして，コールバックメソッ

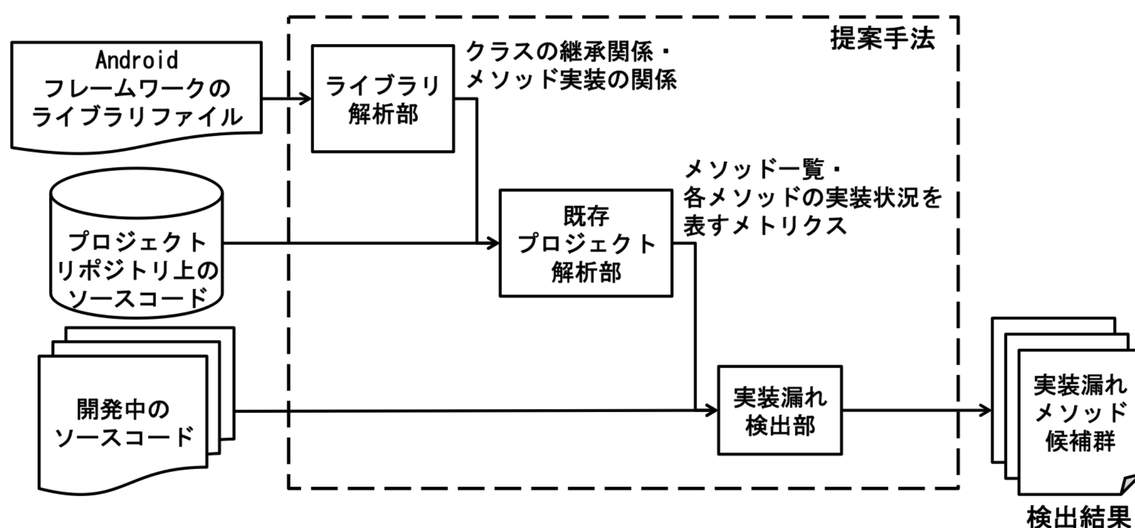

図 1　提案手法の処理の流れ

ドと Android フレームワークで提供されるメソッドインタフェースに対する実装関係や，メソッドのオーバーライドの関係を解析し，既存プロジェクトでの実装状況を示すメトリクスを算出する．その上で，実装漏れ検出部が開発者の Android プロジェクトに含まれるソースファイルを解析し，実装状況メトリクスに従って実装漏れを検出する．これにより，開発者の Android プロジェクトのソースファイルに対して実装すべきコールバックメソッドを自動的に検出する．以降では，3.2 節にライブラリ解析部によるクラス間継承関係やメソッド間実装関係の解析手順を，3.3 節に既存プロジェクト解析部による実装状況を表すメトリクス値の算出手順を，3.4 節に実装漏れ検出部による実装漏れ検出手順を述べる．

3.2　ライブラリ解析の流れ

図 1 のライブラリ解析部は Android SDK [2] に含まれる Jar ファイルからメソッドやクラスを取得し，クラス間の継承関係とメソッドの実装やオーバーライドの関係を解析する．ライブラリに含まれるクラスファイルをすべて列挙し，クラス宣言とメソッドシグネチャを取得する．取得したクラス宣言から，クラス間の継承関係を作成し，継承関係のあるクラス間で同一シグネチャを持っているメソッドは，オーバーライドされているものとみなす．

3.3　実装状況を示すメトリクス値の算出

3.1 節で述べたように節事前調査においてコールバックメソッドの種類で実装頻度が異なること，コールバックメソッド間に実装の共起関係が存在することを発見した．そこで図 1 の既存プロジェクト解析部は前者の関係を表す各メソッドの実装率と，後者の関係を表すメソッドの実装共起率を示す 2 つのメトリクス値を，既存プロジェクトのクラスの継承関係や、メソッドの実装状況から求める．本節ではこれらの算出方法を述べる．

3.3.1　メソッド実装率

コールバックメソッドにはよく実装されるものと，あまり実装されないものがある．例えば Android 端末はタッチパネルを持つものが多いため，クリック操作に

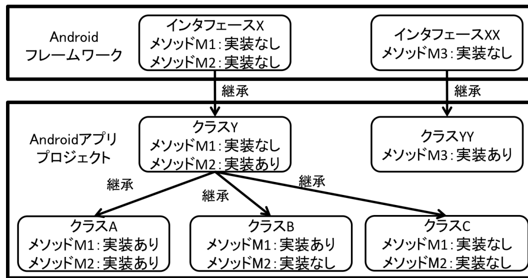

図 2　Android フレームワークに含まれるクラスを継承したメソッド実装の例

反応するメソッドである View.onClick() メソッドはよく実装される．また，端末のセンサーが発生させたイベントに反応する SensorEventListener.onSensorChanged() メソッドもよく実装される．一方でセンサーの精度の変化に反応する SensorEventListener.onAccuracyChanged() メソッドは実装されることが少ない．開発者が SensorEventListener インタフェースを継承し上記の onSensorChanged() メソッドと onAccuracyChanged() メソッドの両方を実装しなかった時，実装漏れの可能性が高いのは実装されることが多い onSensorChanged() メソッドである．本提案手法ではメソッド単体についての実装漏れを示すために，実装される割合である実装率を定義する．

あるクラス（またはインタフェース）X に属するコールバックメソッド M の実装率は次の計算式で求める．

$$\frac{\#ImplementedClasses}{\#InheritedClasses}$$

ここで#ImplementedClasses は X を基底クラスに持ち M が実装されているクラス数，#InheritedClasses は X を基底クラスに持つクラス数である．

例として図 2 を考える．図 2 は Android フレームワーク内のインタフェース X を Android アプリプロジェクトのクラス Y が実装し，さらにクラス Y をクラス A，B，C が継承している．インタフェース X を基底クラスに持つクラスはクラス Y，A，B，C であり，#InheritedClasses はそのクラス数である 4 となる．インタフェース X のメソッド M1 は，クラス Y，C の 2 クラスでは実装されないため，#ImplementedClasses は 2 となる．よって，メソッド M1 の実装率は 50%(2/4) となる．メソッド M2 に注目すると，クラス Y で実装が行われる．クラス Y を継承するクラスでは，メソッド M2 のメソッド宣言のみを新たに記述しない限り，クラス Y でメソッド M2 に記述した処理がそのまま継承され，実装した場合と同じ状態になる．そのためクラス Y，A，B，C の全てにおいてメソッド M2 の処理が存在しており#ImplementedClasses は 4 となるため，メソッド M2 の実装率は 100%(4/4) となる．

3.3.2　メソッド実装共起率

メソッドには，C 言語の malloc() と free() のように同時に利用される組み合わせが存在する．これらと同様に，コールバックメソッドにおいて同時に実装されている組み合わせがある．例えば LocationListener インタフェースの onProviderEnabled() メソッドと onProviderDisabled() メソッドは，それぞれどちらかが実装されている時に，高い割合でもう一方も実装されている．このように同時に実装されていることを実装の共起と呼ぶ．クラス内に実装されているコールバックメソッドと共起する回数が多いメソッドの実装が欠落している場合，実装し忘れである可能性がある．そこでメソッドの組み合わせから実装漏れを示すために，メソッド間の共起の割合である共起率を定義する．

共起率を定義する際には，同じクラスのメソッド間と，異なるクラスのメソッド間の 2 つの場合に分けて考える．同じクラスの場合はクラス単位で共起率を計算し，異なるクラスの場合はプロジェクト単位で共起率を計算する．

- クラス内メソッド実装共起率
  あるクラス（またはインタフェース）X を継承したサブクラス Y でコールバックメソッド M1 が実装されている時，クラス X に含まれる別のコールバックメソッド M2 がクラス Y に実装される割合（クラス内メソッド実装共起率）を，次の計算式で求める．

$$\frac{\#ImplementedClassesOfM1andM2}{\#ImplementedClassesOfM1}$$

ここで#ImplementedClassesOfM1andM2 は X を基底クラスに持ち M1 と M2 の両方が実装さ

れているクラス数，#ImplementedClassesOfM1 は X を基底クラスに持ち M1 の実施する処理が定義されているクラス数である．

図 2 ではメソッド M1 はクラス A，B で，メソッド M2 はクラス A，B，C で処理が記述されている．したがって両方のメソッドに処理が記述されているのはクラス A，B の 2 つになる．そのためメソッド M2 が実装されている 4 クラスのうちの 2 クラスでメソッド M1 が実装されており，実装共起率は 50%(2/4) である．また，メソッド M1 が実装されている 2 クラスの両方でメソッド M2 が実装されており，実装共起率は 100%(2/2) である．

- クラス間実装共起率

あるクラス（またはインタフェース）X を継承したサブクラス Y でコールバックメソッド M1 が実装されている時，クラス XX を継承した別のサブクラス YY でコールバックメソッド M3 が実装される割合（クラス間実装共起率）を次の計算式で求める．

$$\frac{\#ProjectsImplementingM1andM3}{\#ProjectsImplementingM1}$$

ここで#ProjectsImplementingM1andM3 は M1 と M3 を実装するクラスを持つプロジェクト数，#ProjectsImplementingM1 は M1 を実装するクラスを持つプロジェクト数である．

図 2 では，メソッド M1 がクラス A，B で実装されており，またメソッド M3 がクラス YY で実装されている．この例では図示した 1 つのプロジェクトしか解析対象にしていないため，メソッド M1 を実装しているクラスを持つプロジェクト数である#ProjectsImplementingM1 は 1 である．また，メソッド M1 とメソッド M3 はともにこのプロジェクトで実装されている．両方のメソッドを実装しているプロジェクト数である #ProjectsImplementingM1andM3 は 1 である．よってメソッド M1 とメソッド M3 のクラス間実装共起率は 100%(1/1) となる．

3.4 実装漏れメソッドの検出

図 1 の実装漏れ検出部は，開発者の開発している Android アプリのソースコードに対し，3.3 節で算出した実装率と実装共起率のそれぞれの基準に従って，実装すべきコールバックメソッドを特定する．そして，それぞれの基準において実装すべき順に開発者に提示する．

実装漏れ検出部はまず開発者の編集しているソースコードを解析し，その中に含まれるクラスと，それらが継承しているコールバックメソッドの実装の有無を取得する．そして，実装されていないコールバックメソッドについて，既存プロジェクト解析部の取得したコールバックメソッドの実装率と実装共起率から，実装率に注目した場合と，実装共起率に注目した場合のそれぞれについて実装すべきコールバックメソッドを求める．

実装率の基準に従った場合，実装率が高いにもかかわらず実装されていないコールバックメソッドは，実装漏れの可能性がある．そのため，実装率の高い順に開発者に提示する．

実装共起率の基準に従った場合，あるメソッドが実装されていた場合の共起率が高いにもかかわらず実装されていないコールバックメソッドは，実装漏れの可能性がある．そのため，開発者のあるクラスに着目した時に，クラス内で実装されていないコールバックメソッドの中で，同一クラスや別クラス内で実装されているコールバックメソッドと実装の共起があるものを抽出する．そして，抽出したコールバックメソッドと，そのメソッドに対する実装の共起率を提示する．その際に，同一クラス内で実装されているメソッドに共起している場合はクラス内実装共起率を，別クラス内で実装されているメソッドに共起している場合はクラス間実装共起率を，それらの実装共起率の種類によらず，高い順にメソッドをソートした上で提示する．

開発者は提示された検出結果表示から，実際に開発者の意図しない実装漏れであるかどうかを確認することができる．

4 実装

4.1 全体の構成

本章では3章で述べた提案手法の実装について述べる．提案手法のうち，実装漏れ検出はAndroid Studio [2] のプラグインとして実装を行った．一方ライブラリ解析と既存プロジェクト解析は処理時間を要するため，プラグインとは別個に実装した．ライブラリ解析と既存プロジェクト解析によって得たデータはデータベースに蓄積し，プラグインはデータベースからメソッド一覧や実装状況を表すメトリクスを取得するように実装した．そして，全ての処理はJavaを用いて実装し，データベースにはSQLite3 [3] を用いた．以降では4.2節でライブラリ解析部と既存プロジェクト解析部の実装を，4.3節では実装漏れ検出部の実装を，4.4節では実装したプラグインによる提示を述べる．

4.2 ライブラリ解析処理と既存プロジェクト解析処理

ライブラリ解析部は，ライブラリファイルからメソッドやクラスの関係を解析する，まずAndroid SDKに付属の，ユーザが実装に利用できるクラスファイルをまとめたandroid.jarが含む全クラスから，それぞれのインスタンスを生成する．そして，リフレクションAPIを利用して各インスタンスからクラス名，クラスに含まれるメソッドのシグネチャ，クラスやメソッドの継承関係を抽出する．

既存プロジェクト解析部は，既存Androidプロジェクトのソースファイルを取得し，コールバックメソッドの実装率や実装共起率を算出する．入力であるプロジェクトリポジトリのソースコードには，F-Droid [4] に公開されている全てのAndroidアプリプロジェクト2,196個のうち，完全にJava以外の言語で記述されている46個のプロジェクトを除外した2,150個を用いた．これらのプロジェクトのソースコードを入力すると，既存プロジェクト解析部はEclipse Java development tools [5] のASTパーサを利用して，全てのクラスからメソッド宣言と基底クラスの情報を取得する．これらの情報とライブラリ解析部の抽出した情報を組み合わせ，図2のようなクラスの継承関係を抽出する．その際に，基底クラスで実装されているメソッドについては，すべてのサブクラスに対して実装されているものとマークする．同時に各プロジェクトのクラスのうち，抽象クラスまたはインタフェース以外のクラスにおいて，ライブラリ解析部の抽出したメソッドを継承している場合，そのメソッドが実装されているかどうかに基づいて，実装率を算出する．さらに，同じクラスに，Androidフレームワークの特定のクラスのメソッドが複数個実装されている場合は，クラス内実装共起率を算出する．Androidフレームワーク内の全てのクラスを起点とする探索が終了した後，各既存プロジェクトを確認し，1つのプロジェクト内で異なるクラスのメソッドが実装されている場合，クラス間実装共起率を算出する．最後に，Androidフレームワーク内の各メソッドについて，3.3節で述べた計算式を用いて実装率と実装共起率を計算し，算出した値をデータベースに蓄積する．

4.3 実装漏れ検出処理

本節では実装漏れ検出処理の実装について述べる．実装漏れ検出部はAndroid Studioのプラグインとして実装した．プラグインを実行すると，現在開発者が編集しているプロジェクトのJavaファイルを解析する．解析にはIntelliJ Platform SDK [6] を利用し，各Javaファイル内のクラスごとに，そのクラスが継承しているメソッドと実装しているメソッドを取得する．

開発者によって編集作業中のJavaファイルに含まれるクラスが別のクラスを継承しており，未実装のコールバックメソッドが存在する場合，データベースから実装率を取得し，それらのメソッドを実装率が高い順に提示する．

また，編集中のクラスで実装済みのコールバックメソッドについては，データベースからそのメソッドと実装の共起があるコールバックメソッドを検索し，そのようなメソッドが存在する場合は，実装共起率を取得する．検出の結果として得られたコールバックメソッドのうち，実装されていないものを抽出し，実装共起率が高い順に提示する．

4.4 プラグインによる検出結果の提示

本節では提案手法を実装したプラグインによる，実装すべきコールバックメソッドの提示について述べる．図 3 は Android Studio の画面であり，下半分にあるウィンドウが検出結果の提示部分である．プラグインを実行すると，直前に編集していた Java ファイル (図 3 の上部に表示されている Java ファイル) を解析し，提案手法により検出した実装すべきコールバックメソッドを下半分のウィンドウに表示する．実装すべきコールバックメソッドは木構造を用いて表される．木の根には解析した Java ファイル名を表示し，その直下には Java ファイル内で宣言されているすべてのクラスの名前とその先頭の行番号を表示する．無名クラスを宣言している場合は，実装元の基底クラスの名前を表示する．クラス名の下には実装率についての情報（Frequencies）と，実装共起率についての情報（Co-occurrences）を表示する．Frequencies には，4.3 節で求めた実装率に基づき検出された実装漏れコールバックメソッドの候補が実装率の高い順に提示され，Co-occurrences には実装共起率に基づき検出された実装漏れコールバックメソッドの候補が実装共起率の高い順に提示される．なお，複数のコールバックメソッドが特定の未実装のコールバックメソッドと共起する場合，この未実装のメソッドに関して算出された複数の共起率の中で最も高い値を提示する．実装漏れメソッドの候補には，そのコールバックメソッドを定義している Android フレームワークの基底クラスと，実装率や実装共起率の計算に用いた数値を表示する．Co-occurrences についてはさらに，表示対象のクラスのどのメソッドとの共起によって提示されているかの情報も合わせて表示する．

5 ケーススタディ

本章では実際の開発中のプロジェクトに対して提案手法を適用した時に，実装漏れを指摘できるかを確認する．対象としたプロジェクトは，GitHub[7] で公開されている地図アプリである．このプロジェクトでは，あるコミット C0 であるクラス内に LocationListener インタフェースを実装したクラス P と Q を作成していた．LocationListener インタフェースは

表 1 コミット C0 におけるメソッドの実装状況

	クラス P	クラス Q
onLocationChanged	○	×
onProviderEnabled	×	×
onProviderDisabled	×	×
onStatusChanged	×	×

表 2 コミット C1 におけるメソッドの実装状況

	クラス P	クラス Q
onLocationChanged	○	◎
onProviderEnabled	×	×
onProviderDisabled	×	×
onStatusChanged	◎	◎

onLocationChanged メソッド，onStatusChanged メソッド，onProviderEnabled メソッド，onProviderDisabled メソッドの 4 つのコールバックメソッドを持つ．コミット C0 において開発者は表 1 のようにそれぞれのコールバックメソッドを実装していた．

開発者は，2 日後に次のコミット C1 を行った．そして実装状況は，表 2 のように変化した． コミット C0 からコミット C1 の間に，開発者はクラス P の onStatusChanged メソッド，クラス Q の onLocationChanged メソッドと onStatusChanged メソッドを実装していた（表 2 では，◎で示す）．なお表 2 内で○や◎でマークしたメソッドは，その後のコミットでも削除されることはなく，また×でマークしたメソッドは，その後のコミットでも追加されることはなかった．

ここで，コミット C0 直後の時点に着目し，コミット C1 で実装されたメソッド群 （表 2 の◎で示したメソッド群）を，実装漏れにより追加されたメソッド群であるとみなす．そしてコミット C0 直後のソースコードに対して提案手法のツールを適用し，実装すべきコールバックメソッドを抽出し，提示させた．その結果，クラス P と Q に対しそれぞれ図 4 と図 5 のように実装漏れコールバックメソッドの候補が提示された．

図 4，図 5 では Frequencies に未実装のコールバックメソッドそれぞれの実装率を，高い順に提示する．また Co-occurrences には，そのクラスで実装されているコールバックメソッドとの実装共起率を，高い順に提示する．クラス P では onLocationChanged メソッ

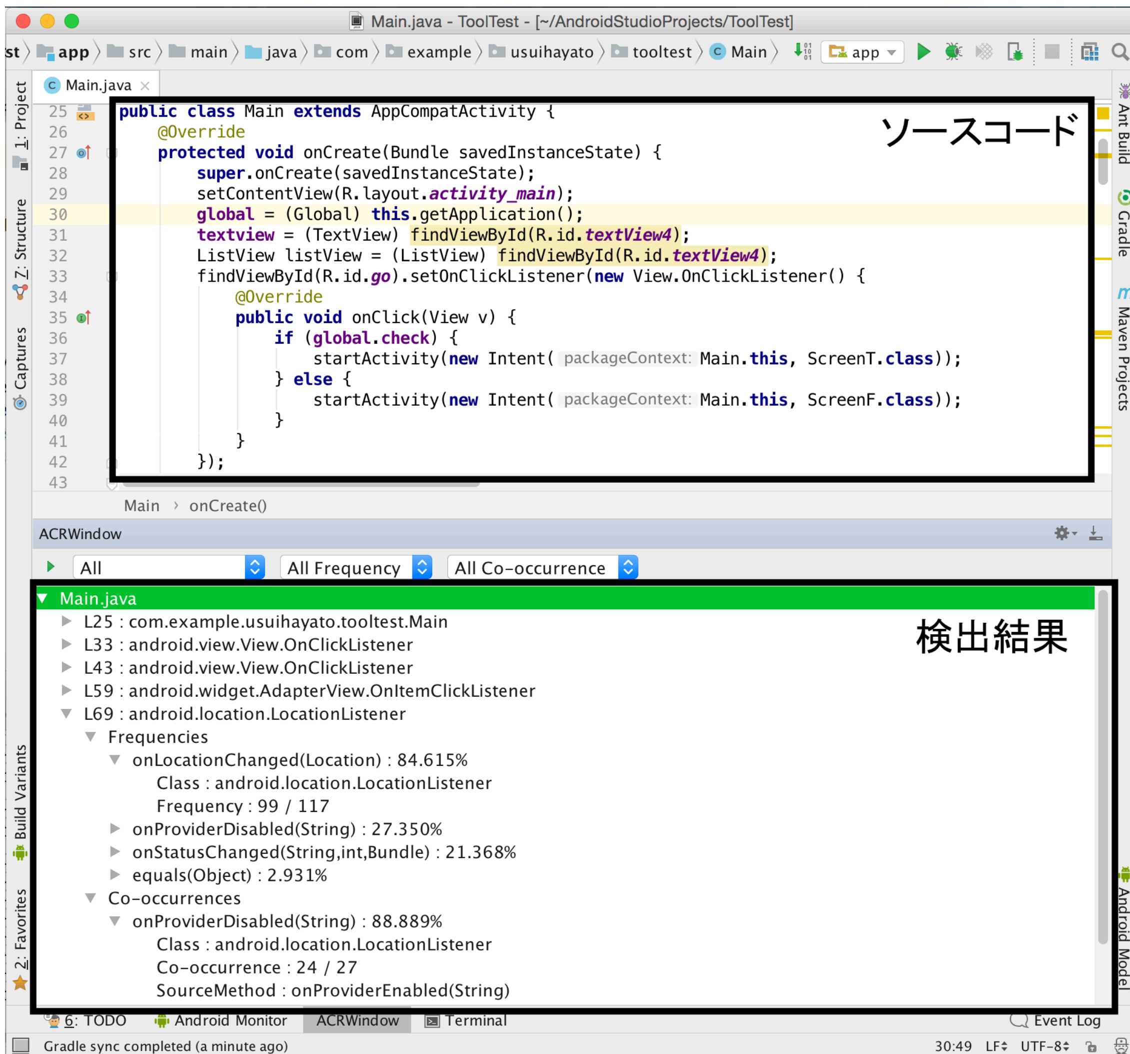

図 3　提案手法による検出結果の提示

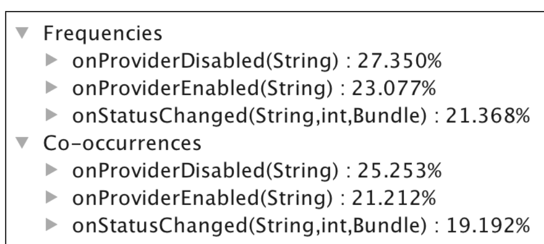

図 4　コミット C0 直後のクラス P に対する実装漏れメソッド候補の提示

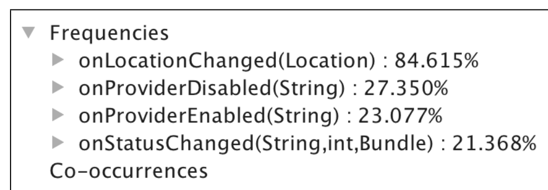

図 5　コミット C0 直後のクラス Q に対する実装漏れメソッド候補の提示

ドが実装されているため，図 4 は，onLocationChanged メソッドと他のコールバックメソッドとの実装共起率が提示されている．いずれの値も小さくなってお

り，実装されていないことを示す以上の働きができるとは考え難い．また，図 5 のクラス Q に対する表示では，C0 時点ではどのメソッドも未実装であるため，Co-occurrences については表示しない．しかし，Frequencies から，onLocationChanged メソッドの実装率が非常に高いことがわかる．これにより，開発者に onLocationChanged メソッドの実装の必要性を気付かせることができる．

なお，この例で利用したプロジェクトに含まれるメソッド群は，共起して実装されることがあまりなかったため，図 4 に示すように実装共起率がそれほど高くならなかった．

6 関連研究

関連研究として，コードレコメンデーションに関する研究が挙げられる．コードレコメンデーションとは，開発者がプログラムを記述している時に内容を補完するコードを提示することである．Ye らは入力中の文字列や開発者のユーザモデルを基に，開発者の知らないメソッドの利用を提示する手法を提案した [8]．渡邉らは開発者が記述しているコードに関連したサンプルプログラムを，連想検索エンジンを用いてリポジトリから検索し提示する Selene を提案した [9]．村上らは開発者が記述しているメソッドの呼び出し関係や編集履歴を用いた手法を提案した [10]．

またその他の関連研究として，メソッドのチェンジレコメンデーションに関する研究が挙げられる．チェンジレコメンデーションとは開発が進みプログラムに変更が加わった時，変更する必要があるメソッドを提示することである．Xia らはプログラム内の依存関係のグラフ化，グラフの探索および遺伝的アルゴリズムによって変更すべき箇所を提示する SupLocator を提案した [11]．Rolfsnes らはアソシエーションルールマイニングをチェンジレコメンデーションに用いることが有効であることを示す調査を行った [12] [13]．

これらの研究は開発者が実装すべきメソッドの存在を知り，それを実装している時や実装した後に利用できるものである．提案手法は開発者が実装していないメソッドを提示し，実装漏れを防ぐという点でこれらの研究と異なる．

また，コールバックメソッドの実装漏れに対する研究として，テストが挙げられる．Adamsen らは通常とは異なる状況でテストを行う Thor を提案した [14]．Amalfitano らは外部イベントを含むテストを行う ExtendedRipper を提案した [15]．また，Yu は継承されていない Android API のメソッドを引き起こすイベントを含むテストケースの生成手法を提案した [16]．これらの研究はテストの段階で実装に問題があることを発見する．しかしながらテストで発見した問題の修正のために開発への手戻りが発生し，これにはコストがかかる．本提案手法は開発段階で実装すべきメソッドを提示するため，手戻りが少なくなる．一方で提案手法は実装漏れの指摘であり実装内容の不備を指摘することはできない．

7 結論

本論文では，既存 Android アプリケーションにおけるコールバックメソッドの実装率，実装共起率に基づいた，実装すべきメソッドの提示手法を提案した．これにより，コールバックメソッドの実装漏れを防ぐことができる．

今後の課題としては開発者に提示する基準や内容の増加および，提案手法の評価実験が挙げられる．

参考文献

[1] H. Usui, M. Nagura and S, Takada, “Investigating Tendencies in Callback Method Implementations in Android Applications”, *Proceedings of the 8th IEEE International Workshop on Empirical Software Engineering in Practice*, pp. 23-28, 2017.

[2] Google, “Android Studio (online),” available from, `https://developer.android.com/studio/index.html`, accessed Aug.31, 2017.

[3] SQLite, “SQLite (online),”, `https://sqlite.org/index.html`, accessed Aug.31, 2017.

[4] F-Droid, “F-Droid (online),” available from, `https://f-droid.org/`, accessed Aug.31, 2017.

[5] The Eclipse Foundation, “Eclipse Java development tools (online),” available from `http://www.eclipse.org/jdt/`, accessed Aug.31, 2017.

[6] JetBrains, “IntelliJ Platform SDK (online),” available from `http://www.jetbrains.org/intellij/sdk/docs/welcome.html`, accessed Aug.31, 2017.

[7] GitHub, “GitHub (online),”, `https://github.com`, accessed Aug.31, 2017.

[8] Y. Ye and G, Fischer, “Supporting Reuse by De-

livering Task-Relevant and Personalized Information", *Proceedings of the 24th International Conference on Software Engineering*, pp. 513-523, 2002.

[9] T. Watanabe and H, Masuhara, "A Spontaneous Code Recommendation Tool Based on Associative Search", *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pp. 17-20, 2011.

[10] N. Murakami, H. Masuhara and T, Aotani, "Code Recommendation Based on a Degree-of-Interest Model", *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pp. 17-20, 2011.

[11] X. Xia and D, Lo, "An effective change recommendation approach for supplementary bug fixes", *Automated Software Engineering*, vol. 24, num. 2, pp. 455-498, 2017.

[12] T, Rolfsnes, L. Moonen, S. D. Alesio, R, Behjati and D. Binkley, "Improving Change Recommendation using Aggregated Association Rules", *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 73-84, 2016.

[13] L. Moonen, S. D. Alesio, D. Binkley and T, Rolfsnes, "Practical Guidelines for Change Recommendation using Association Rule Mining", *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 732-743, 2016.

[14] C. Q. Adamsen, G. Mezzetti and A. Moller, "Systematic execution of Android test suites in adverse conditions", *Proceedings of 2015 International Symposium on Software Testing and Analysis*, pp. 83-93, 2015.

[15] D. Amalfitano, A. R. Fasolino and P. Tramontana, "Considering Context Events in Event-Based Testing of Mobile Applications", *Proceedings of Software Testing, Verification and Validation Workshops*, pp. 126-133, 2013.

[16] S. Yu and S. Takada, "Mobile application test case generation focusing on external events", *Proceedings of 1st International Workshop on Mobile Development*, pp. 43-44, 2016.