インスペクションと
Android Lint Custome Ruleによる、
単一責任原則の実践

robo

イントロダクション

TDDやDDDが喧しい昨今、
コード実装の単一責任の原則（SRP）意訳⇒シンプル化が求められています。

Androidアプリ開発も成熟期となり、
神Activityの反省から役割(責任)ごとにクラスを分割し、ソースコードのメンテナンス性やテスタビリティの向上への関心が高まっているのでしょう。

インスペクション・ソース指摘機能

Android Studio には、インスペクションという
強力なソース解析(指摘)機能があることを
ご存知だと思います。

問題点のある実装部を
エディタ上でハイライト表示したり、

「Analyze」メニューの"Inspect Code..."により、
指摘一覧を「Inspection Resultsツールウィンドウ」で
カテゴリ別にリストアップもしてくれる機能です。

Android Lint も一緒に利用されます

この「Inspection Results」左側ペインの
カテゴリに「Android > Lint > Correctness」など、

Android Lint のチェック項目が
反映されていることに気づかれている方も多いのでは
ないでしょうか。

Android Studio のインスペクションには、
Android Lint も利用されているのです。

独自 Custom Rule で指摘させる

Android Lint は、
独自の Custom Rule を作ることができます。

そして Custom Rule を作るために
Javaソースコードの静的解析基盤も提供されています。

このセッションでは、
単一責任となるシンプルな実装を強制(強要)する
オリジナルの Android Lint Custome Rule 作成と
利用について発表いたします。

対象者

初級者〜中級者

- メソッドが複雑にならないよう実践したい方
- Android Lint の Custom Rule と静的コード解析について理解したい方

サンプル・プロジェクトの紹介

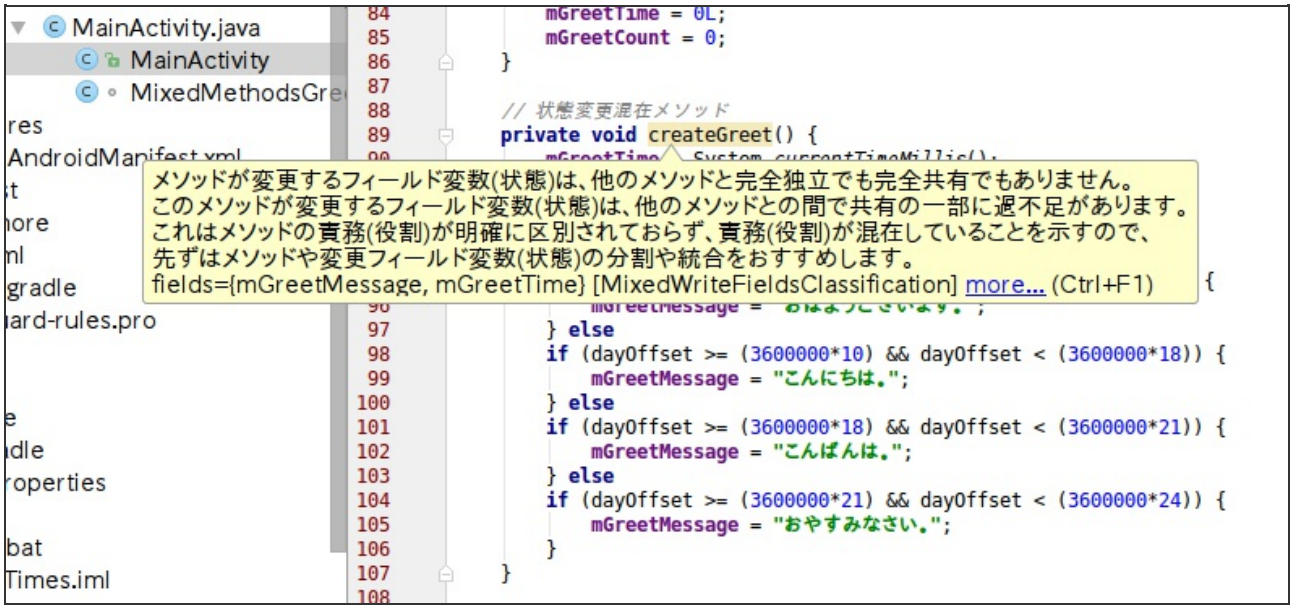
cch-robo/Android_Lint_SRP_Practice_Example

https://github.com/cch-robo/Android_Lint_SRP_Practice_Example

本セッションで利用する、
単一責任となるシンプルな実装の強制(強要)を促す
オリジナル Android Lint Custome Rule の
プロジェクトです。

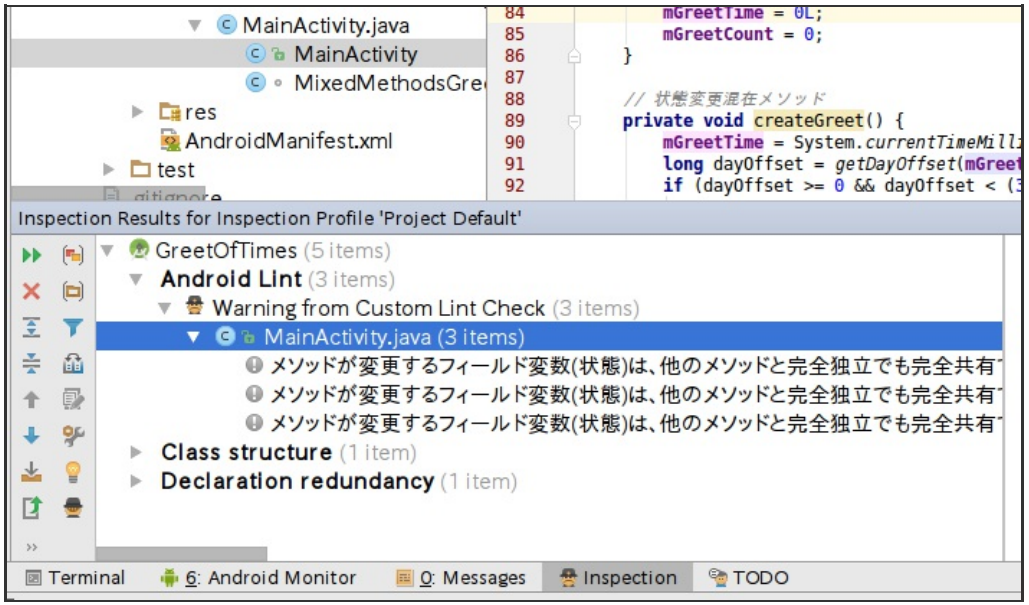
フィールド変数（状態）を変更するメソッドが、
複数存在（複合責務）する場合、その共有度合いから
責務(役割)の混在の種類と問題についてレポートします。

サンプル・プロジェクトの紹介



- 責務複合メソッドに対しインスペクションにより、
  エディタ上でメソッド名がハイライト表示され、
  マウスオーバーで問題点のレポートが確認できます。

サンプル･プロジェクトの紹介



- Analyze ⇒ Inspect code... で、
  プロジェクト内に存在する問題(責務複合メソッド)箇所の一覧が確認できます。

サンプル・プロジェクトの紹介

注意: サンプル・プロジェクトでは、
Javaソース探索にJavaPsiScanner、
Javaソース解析にJavaElementVisitor を使うため
以下の制限があります。

- com.intellij.psi の AST パッケージを使っています。
- 既存の一次資料ドキュメントでは説明がありません。
- android/toolsの lint では使えません。

御利用にあたっての補足

- プロジェクト内のディレクトリ supplementary にある lint-srp-example.jar を *USER NAME*/.android/lint ディレクトリにコピーすれば Android Studio からサンプル機能が利用可能になります。
- 限定条件下の責務複合メソッドを検出しているので、全てのパターンの検出や推論的な責務判断にまでは対応していません。
- 一部の環境では、日本語出力が文字化けします。

自己紹介

- 名前
  robo (兼高理恵)　Twitter : @cch_robo
- お仕事
  Java 技術者
  要件定義に設計から実装まで
  (最近は、Android開発中心のフリーランス)
- 好きなもの
  モバイル端末

単一責任の原則

単一責任の原則とはなにか

単一責任の原則 または 単一責務の原則
(SRP:Single Responsibillity Principle) とは、

「クラスの責務(変更の原因となるもの⇒役割)は、
　ただ一つでなければならない。」…という原則です。

換言すると
「クラスが管理する状態（責務で変更されるもの）は、
　復数在ってはならない。」…ということになります。

1つの状態は、復数の要素から構成されても構いません。

単一責任は抽象化を高める

書籍「Clean Code アジャイルソフトウェア達人の技」
第10章 クラス では、以下のような言及があります。

- クラスの規則の筆頭は、
  カプセル化と隠蔽を使って、小さくすること。
- 小さくするとは、
  責務の数を少なくして、コードの抽象化を高めること。
- システムを再利用可能な小さなクラスの集まりにして、
  可動部を少なくすること。
- こうなれば、メンテナンス時の影響範囲も小さくなる。

単一責任が破られる要因

単一責任の原則は、
クラスを「責務を限定して小さなものにすること」…と
とても単純ですが、よく言われる神アクティビティが生まれるのは、何故でしょう。

私は、
「Aがアの時は1、Aがイの時は2、
　だがAがアかつBがウの時は3」…など、

要求仕様自体が、フロー手続き型になっていること
…が要因の一つと思います。

単一責任が破られる要因 (2/2)

設計工数が少ない場合、
要求仕様自体が、フロー手続き型になっていると、
小さな責務クラスに分割する設計時間が取れないので、

1. 全ての状態をグローバルにアクセスできるよう、
   1つのクラスにフラットに配置し、
2. 状態を変更する受動と能動の手続きを作り、
3. フラグと条件分岐で管理してしまうから

…のように思えます。

単一責任の原則を守るには

要求仕様自体が、フロー手続き型になっていても、
単一責任の原則を守るようにするには、

「何が、何に対して、メッセージを送るのか」という、
オブジェクト指向プログラミングの原点に立ち、
自問しながらプログラムを行うことでしょう。

フロー手続き主体から、
オブジェクトが主体になるよう視点を変えること
…なのだと思います。

エンタープライズ系での議論

エンタープライズ系のシステム開発では、
一昔前に Transaction Script パターンと
Domain Model パターンの議論がありました。

前ページの単一責任が破られる要因は、
この議論と被るように思えましたので、
ここで挙げておきます。

Transaction Script パターン

- データと振る舞いを別々のオブジェクトに分け、
  機能ごとに振る舞いの手続きを作るパターン。
- メリット
  手続きなので、どのような振る舞いも実装できる。
- デメリット
  手続きなので、ロジックの記述先を強制できない、
  重複や相互矛盾が発生しても気づきにくい。

Domain Model パターン

- データと振る舞いを1つの(役割と責務)オブジェクトにカプセル化するパターン。
- メリット
  データと振る舞いが隠蔽化されカプセル化されるので、ロジックの記述先は明確で、重複や矛盾に気づき易い。
- デメリット
  設計コストがかさむ。
  オブジェクトに対する機能要求と言えないものには、DDD の Service パターンを別途適用する必要がある。

参考先サイト

- ドメインモデルに対する日米の温度差
  http://ameblo.jp/ouobpo/entry-10036477015.html
- いまさらきけない
  「ドメインモデル」と「トランザクションスクリプト」
  http://d.hatena.ne.jp/higayasuo/20080519/1211183826
- トランザクションスクリプトVSドメインモデル
  http://d.hatena.ne.jp/kounan13/20100407/1270619662

Android Lint Custom rule 作成の基礎

一次資料 (ドキュメント)

サンプルは、Android Studio Project Site の
以下のページの基礎知識を元に作っています。

- Tips > Android lint
  http://tools.android.com/tips/lint

- Writing a Lint Check
  http://tools.android.com/tips/lint/writing-a-lint-check

- Writing Custom Lint Rules
  http://tools.android.com/tips/lint-custom-rules

一次資料 (ドキュメント)の概要

- >Tips > Android lint
  lint コマンドラインツールの使用法説明ページ
- Writing a Lint Check
  lint check の基礎知識のページ
  ソースの問題を指摘するために使う、
  ISSUE や Detector などのコンポーネントの説明
- Writing Custom Lint Rules
  lint check 新規作成の具体的手順のページ
  開発環境の作り方や、実装と使用法の具体例

一次資料（ソースコード）

サンプル・プロジェクトは、
以下のページのソースコード内容を参考にしています。

- googlesamples / android-custom-lint-rules
  https://github.com/googlesamples/android-custom-lint-rules
  Writing Custom Lint Rules で紹介されています

- android / platform / tools / base / studio-master-dev / . / lint / libs
  https://android.googlesource.com/platform/tools/base/+/studio-master-dev/lint/libs/
  開発版のソースを参考にしていることに注意

その他（lint 関連一次資料）

lint の使い方についての説明ページ

- Improve Your Code with Lint
  https://developer.android.com/studio/write/lint.html

- Gradle Plugin User Guide > Lint support
  http://tools.android.com/tech-docs/new-build-system/user-guide#TOC-Lint-support

- LintOptions
  http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.LintOptions.html#com.android.build.gradle.internal.dsl.LintOptions

Android Studio で
Lint Custom rule を開発する

Android Studio で開発する方法

Lint Custom rule は、Java アプリケーションです。
このため、以下のような手法を利用します。

- 既存の Custom Lint rule 開発プロジェクトを使う
  googlesamples / android-custom-lint-rules を
  ローカルに zip 展開して開き、ソースを変更します。
- 新規 Android プロジェクトを手動で編集する
  新規の Android プロジェクトを作成し、
  Lint Custom rule の Java プロジェクトになるよう
  手動編集します。

既存プロジェクトを使う場合

一次資料の Writing Custom Lint Rules では、
こちらが紹介されています。

1. googlesamples / android-custom-lint-rules の
   zipファイルをダウンロードして、ローカルに展開

2. build.gradle の jar タスクや、
   src ディレクトリツリーの Javaソースファイルを変更

新規プロジェクトを手動編集する場合

サンプル・プロジェクトでは、こちらを利用しました。

1. 適当な新規 Android プロジェクトを作る (fig-1)
   補足：Activity は作らないようにします。
2. プロジェクトから app モジュールを削除する (fig-2)
   File > Project structure で、
   Project structure ダイアログを開き、
   左ペインの Modules 配下の app モジュールを選択して、左上の – アイコンで削除します。

手動編集する場合 (2/7)

3. Project ビューで、app ディレクトリを削除する (fig-3)
   一緒に [app▼] 実行設定も削除しておきます。(fig-4)
   注意：前ステップは、論理削除なので、
   app モジュールの物理ディレクトリは残っているため。

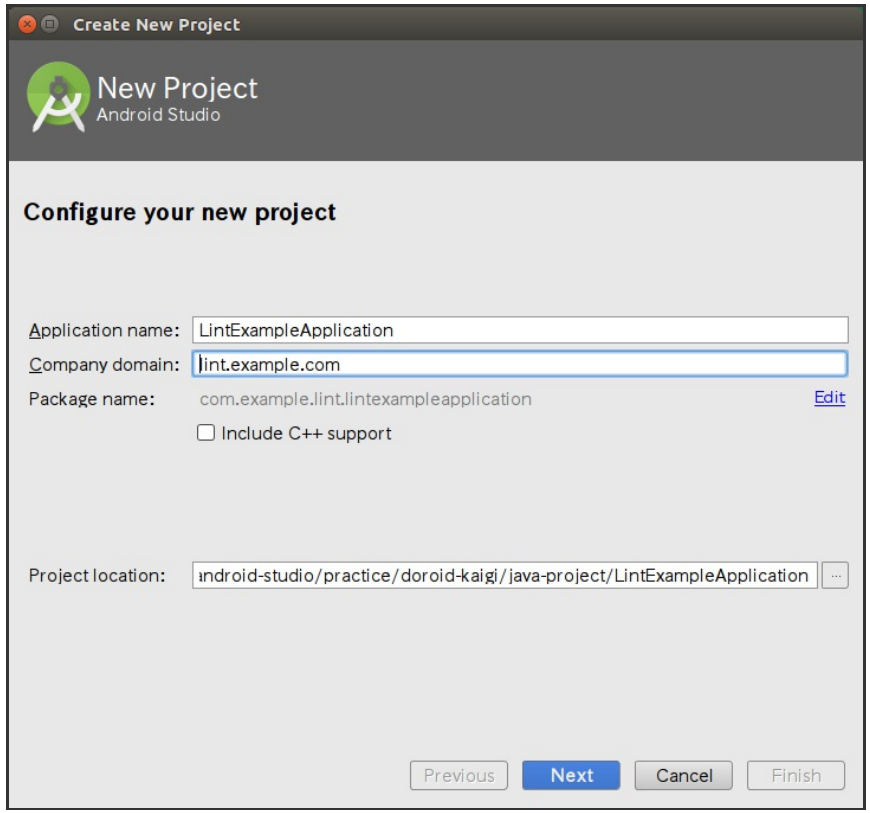
4. build.gradle を上書する
   build.gradle の内容を
   android-custom-lint-rules の
   Java アプリケーションに変更します。
   (具体的な内容は後述)

手動編集する場合 (3/7)

5. src ツリーを新規作成する
   src ツリー(src/main/java/... src/test/java/...)を
   プロジェクト・ディレクトリに新規作成します。

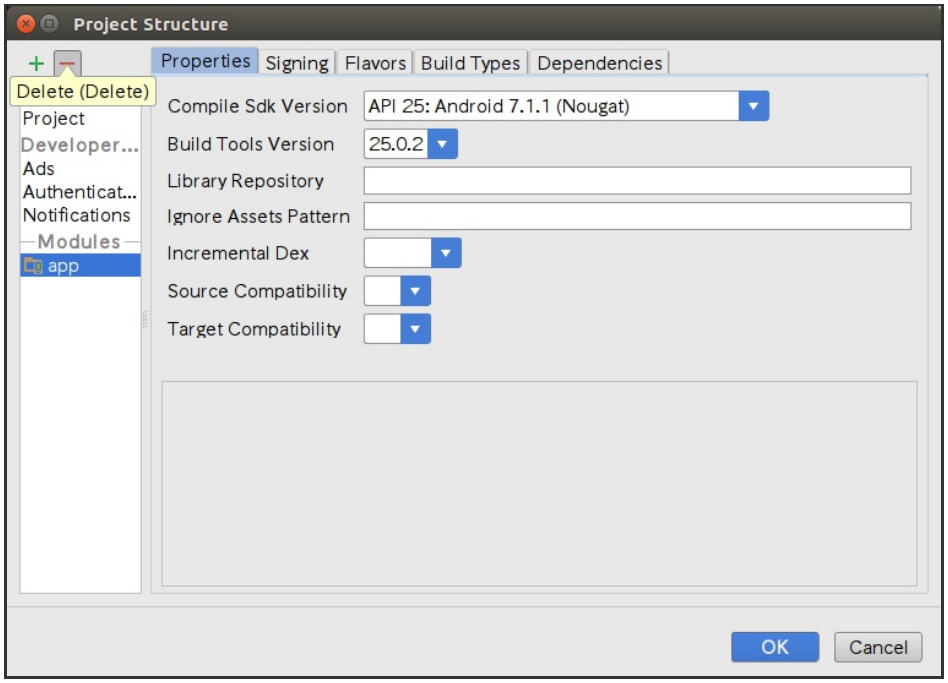
6. Android Studio を再起動する
   File > Invalid Cash / Restart... から
   Invalid and Restart を実行します。
   再起動後に、File > Project structure で
   Project structure のモジュール設定を確認すれば、
   設定項目が自動設定されていることが確認できます。

手動編集する場合 (4/7)



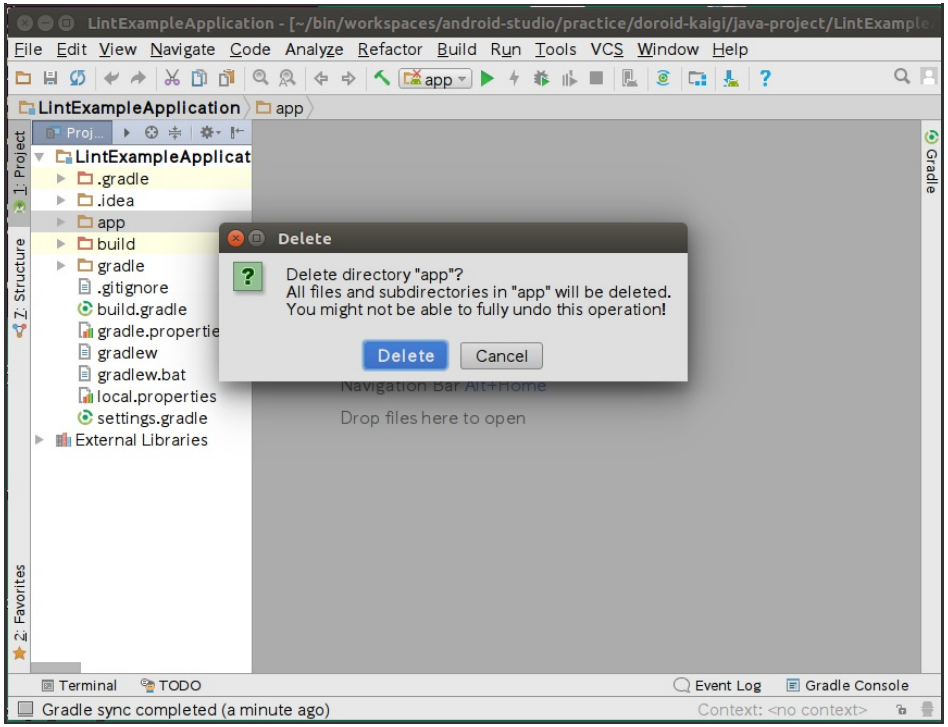
(fig-1)

手動編集する場合 (5/7)



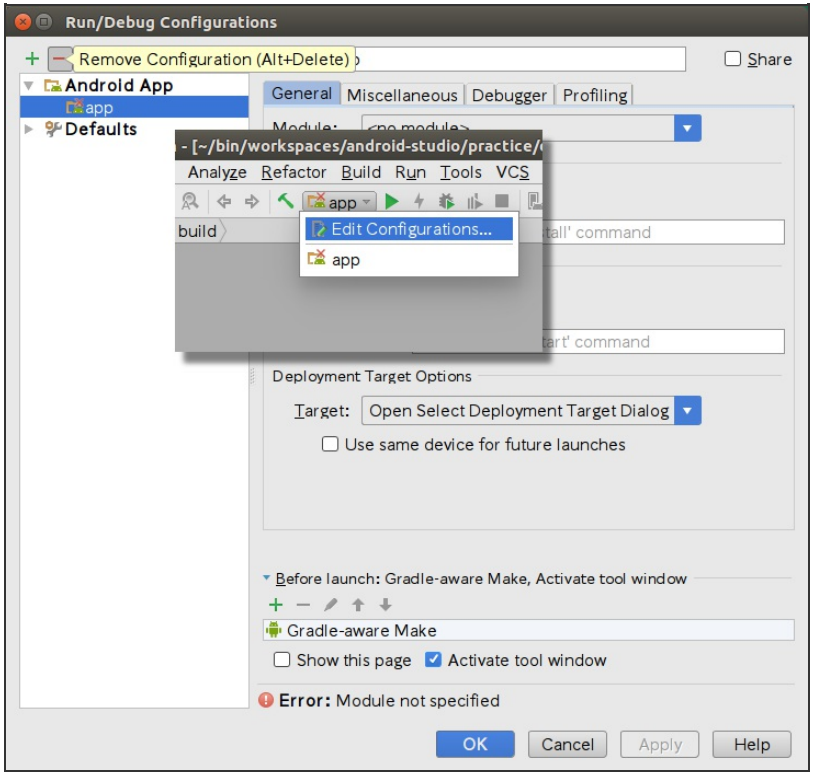
(fig-2)

手動編集する場合 (6/7)



(fig-3)

手動編集する場合 (7/7)



(fig-4)

build.gradle 上書内容 (1/2)

```
apply plugin: 'java'

repositories {
    jcenter()
    maven {
        url "http://dl.bintray.com/android/android-tools"
    }
}

dependencies {
    compile 'com.android.tools.lint:lint-api:25.3.0'
    compile 'com.android.tools.lint:lint-checks:25.3.0'
```


build.gradle 上書内容 (2/2)

```
    testCompile 'junit:junit:4.11'
    testCompile 'com.android.tools.lint:lint:25.3.0'
    testCompile 'com.android.tools.lint:lint-tests:25.3.0'
    testCompile 'com.android.tools:testutils:25.3.0'
}

// jar ファイル名や、Lint-Registryは、自分用に変更してください
jar {
    archiveName 'lint-example.jar'
    manifest {
        attributes("Lint-Registry":
                   "com.example.ExampleIssueRegistry")
    }
}

defaultTasks 'assemble'
```


Lint ライブラリにソースを添付

開発しやすくするため、
Lint ライブラリにソースを添付してみます。

Lint Custom rule 関連のライブラリ

サンプル・プロジェクトで利用している
Lint Custom rule 関連のライブラリ(jar)の一覧

• lint-25.3.0.jar	Lint ツール関連
• lint-api-25.3.0.jar	Lint API 関連
• lint-checks-25.3.0.jar	既存 Detector 関連
• lint-test-25.3.0.jar	既存 Detector テスト関連
• lombok-ast-0.2.3.jar	lombok AST関連
• uast-162.2228.14.jar	com.intellij AST (PSI)関連

ライブラリのソース

ダウンロード元のリポジトリには、
各ライブラリのソースも配置されています。

- lint, lint-api, lint-checks, lint-tests
  http://dl.bintray.com/android/android-tools/com/android/tools/lint/
- com-intellij, lombok-ast
  http://dl.bintray.com/android/android-tools/com/android/tools/external/

ライブラリのソース (2/3)

リポジトリに在った、各ライブラリのソース

- lint lint-25.3.0-sources.jar
- lint-api lint-api-25.3.0-sources.jar
- lint-checks lint-checks-25.3.0-sources.jar
- lint-test lint-test-25.3.0-sources.jar
- lombok-ast lombok-ast-0.2.3-sources.jar
- uast uast-162.2228.14-sources.jar

ライブラリのソース (3/3)

サンプル・プロジェクトの supplementary フォルダに
前ページで紹介したソース jar を置いておきました。
よろしければ、御利用ください。

ライブラリにソースを添付する (1/2)

1. Android Studio の Project ビューで、
   Project を選択して External Libraries を表示する

2. External Libraries で
   Gradle で DL したライブラリー覧から
   前述の lint 関連のライブラリを選択する (fig-1)

3. 右クリックでコンテキストメニューを開き
   Library Properties... をクリックして、
   Library Properties ダイアログを開く (fig-2)

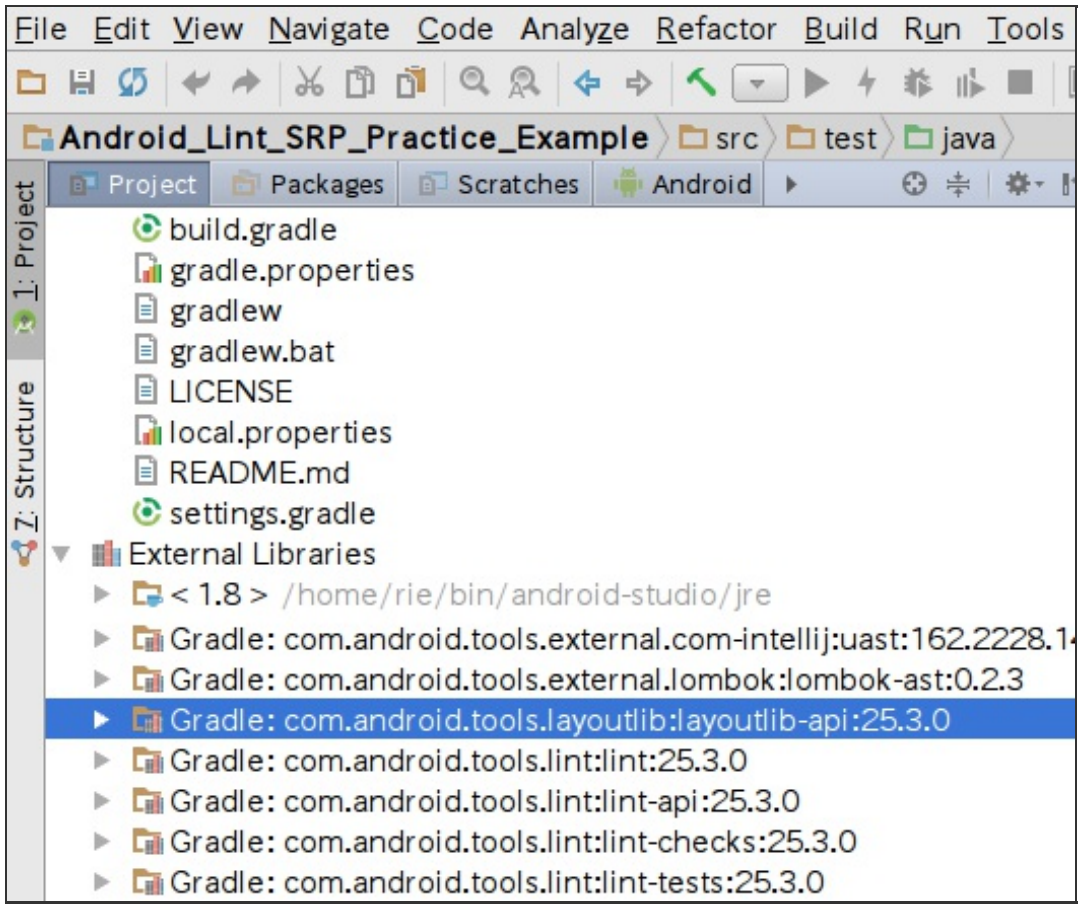
ライブラリにソースを添付する (2/2)

4. ダイアログ左上の ＋ で
   ファイルダイアログを開く (fig-3)

5. ファイルダイアログで
   ライブラリに対応するダウンロードしておいた
   ソース jar を選択して [OK] をクリック (fig-4)
   jar の内容からソースが添付される (fig-5)

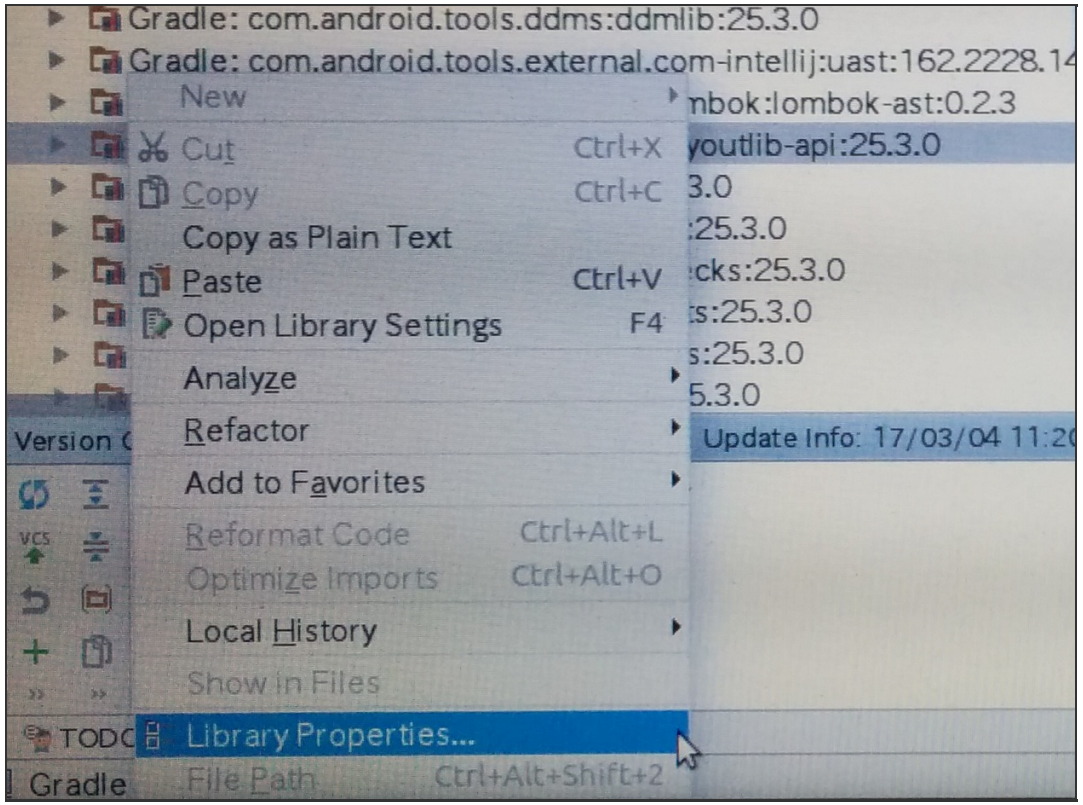
6. Library Properties ダイアログで
   [OK] をクリックしてソース添付を終了する

ライブラリにソースを添付する (1/5)



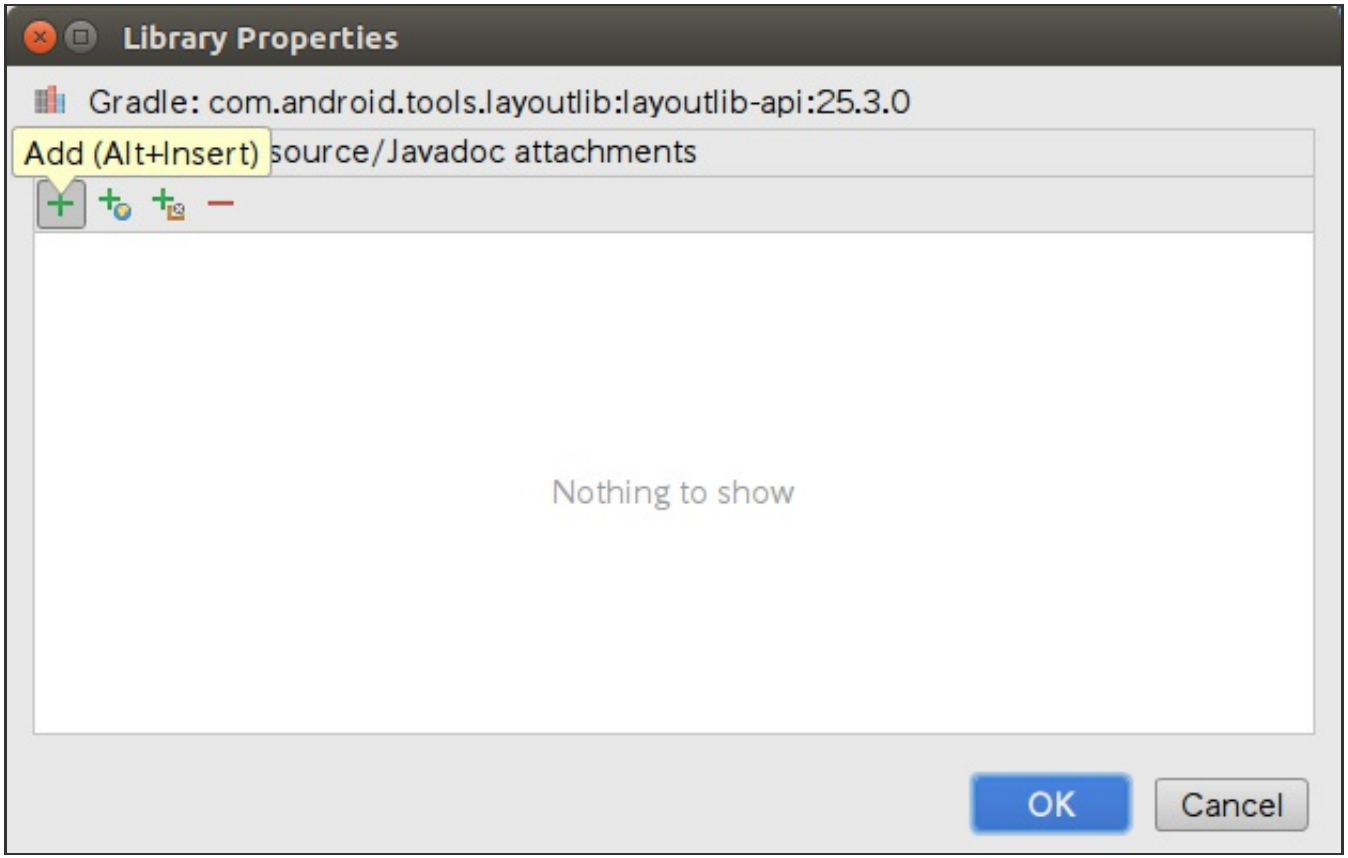
(fig-1)

ライブラリにソースを添付する (2/5)



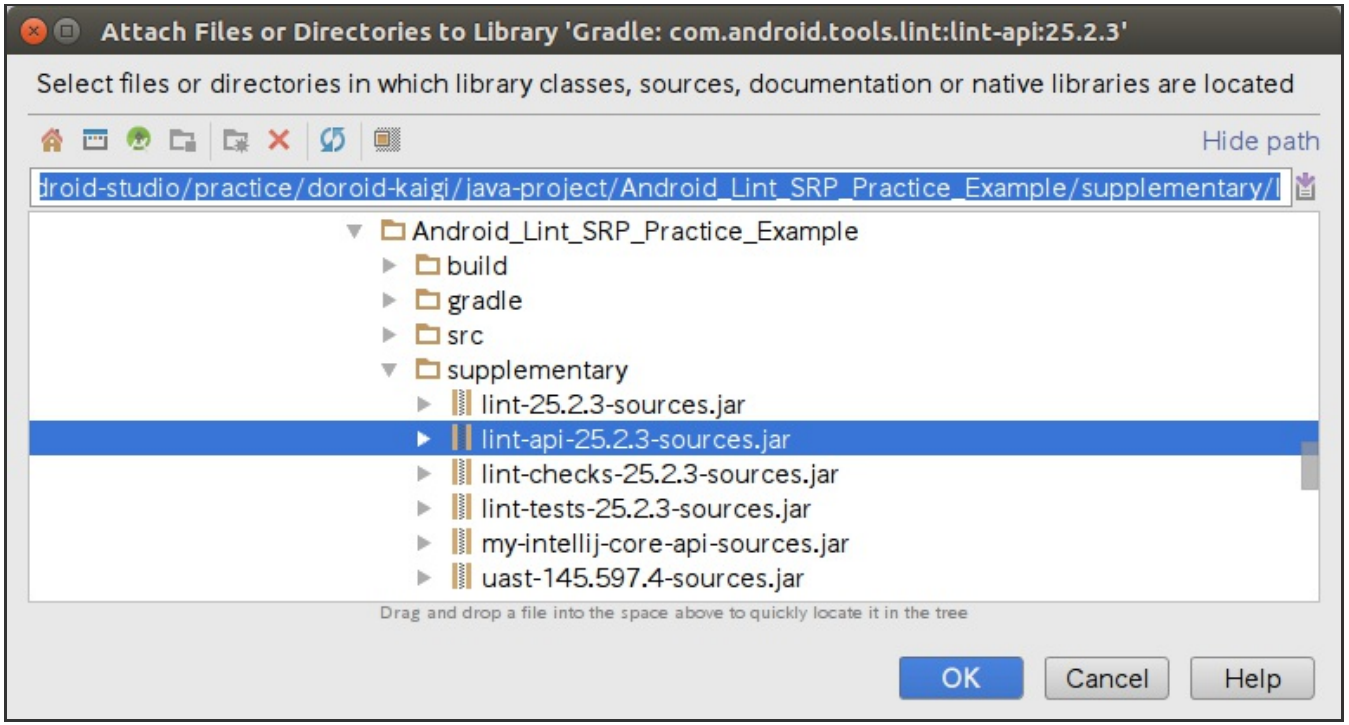
(fig-2)

ライブラリにソースを添付する (3/5)



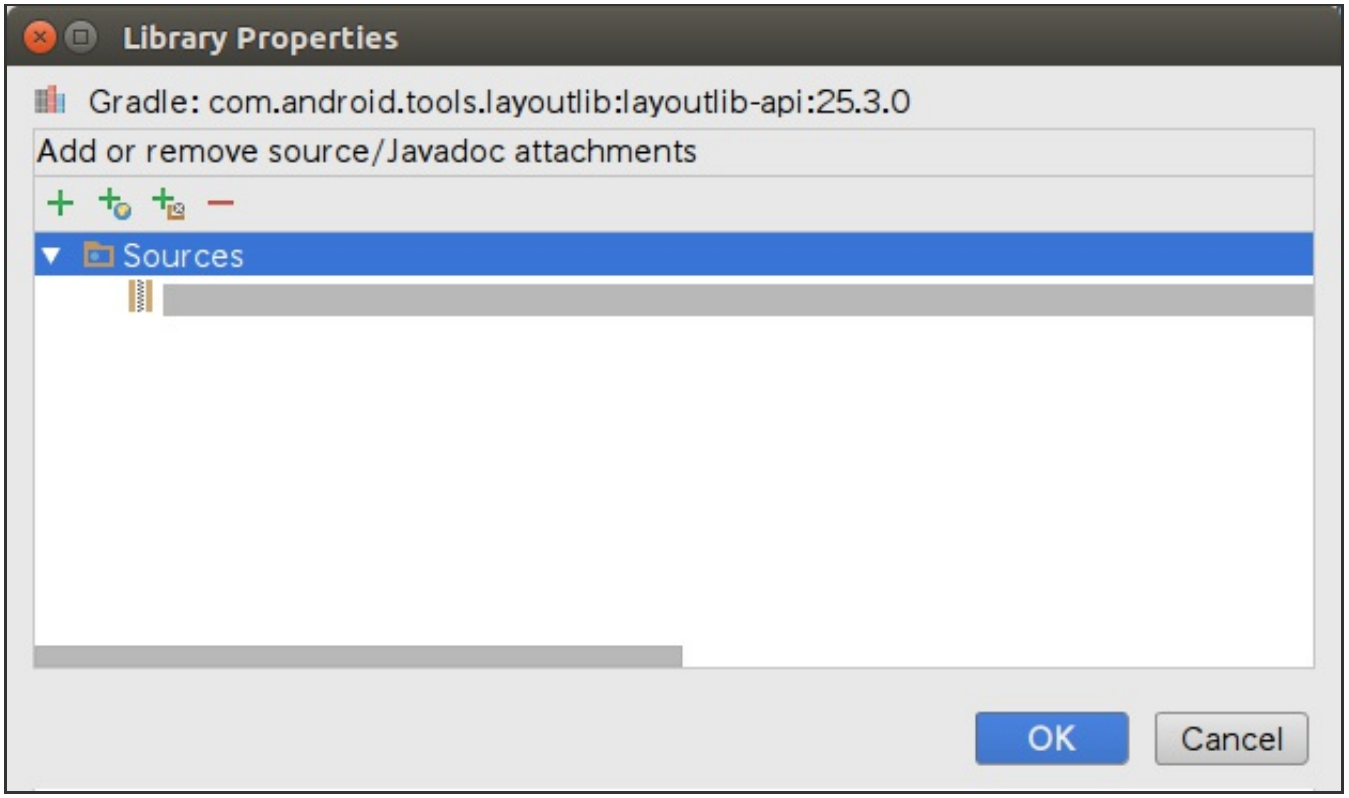
(fig-3)

ライブラリにソースを添付する (4/5)



(fig-4)

ライブラリにソースを添付する (5/5)



(fig-5)

補足:ライブラリ・ソースの欠落

lint-tests と usat は、既存 Detector のテストや com.intellij パッケージのソースが欠落しています。

- lint-tests の既存 Detector テストのソースは、android / platform / tools / base / studio-master-dev を御参照ください。
- JetBrains/intellij-community の com.intellij.psiパッケージが利用できないか調べてみましたが構成が違うようです。

補足:開発プロジェクトのビルド

Android Studio を使った開発では、
Terminal から gradlew コマンドを使ってビルドします。

```
$ ./gradlew clean    ⇒ ビルド結果をクリア

$ ./gradlew test     ⇒ ビルドおよびテストを実行

$ ./gradlew assemble ⇒ ビルドおよび jar ファイルの作成
```

Android Studio メニューの
Build や Run は利用しません。

Android Lint での Java ソース解析の基本

Java ソース解析の基本

Android Lint では、
Detector と呼ぶ Issue 検出器を作って、
プログラムに問題がないかをチェックします。

- Android Lint では、
  Java ソース内容の問題も検出できるよう、
  ソースからAST(抽象構文木)に構文解析済みの
  *com.intellij.ps.*PsiElement を基底ノードとした、
  各種 ノード を提供してくれる基盤を持っています。

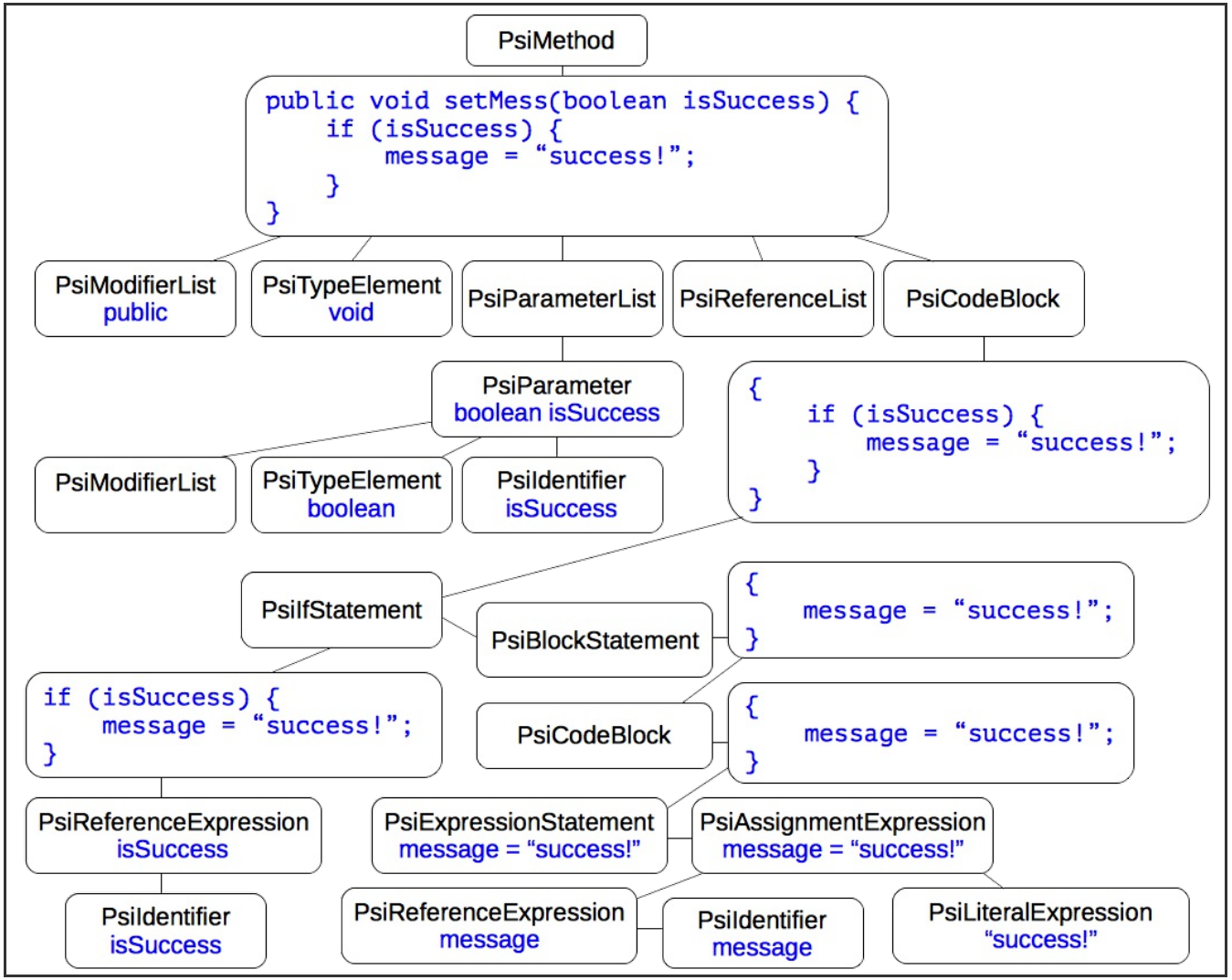
ノード

Javaソース構文解析におけるノードとは、
Java言語の言語要素を表したものです。

- ノードは、Class、Field、Method、文(Statement)、
  式(Expression)、参照(Reference)、名前(Itentifier)、
  リテラル(Literal)…などの構文要素に対応しています。

- 各ノードには、優先順位と親子関係があり、
  構文解析されたソースは、ノードの入れ子構造に
  なっています。

構文解析されたソースのノード構造例



ソース解析前の事前学習

Lint Custom rule 開発における Javaソース解析は、
各ノードの親子関係と入れ子構造から、構文内容を
把握することと言えます。

1. どのような構文が、どのような木構造になるのか
2. 親となるノードから、
   直接特定の子ノードを参照できるのか、
   できるならどんなメソッドが用意されているのか

…を事前に把握(学習)している必要があります。

1.構文の木構造を学ぶ

サンプル・プロジェクトでは、
Javaソースの構文解析結果の木構造
(ノードの入れ子構造)の学習と確認用に
PsiClassStructureDetector を用意しました。

- PsiClassStructureDetector は、
  Lint から提供されたノードの親子関係を
  デバッグ出力するためだけの Detector です。

1.構文の木構造を学ぶ (2/3)

PsiClassStructureDetector の使い方

1. PsiClassStructureDetectorTest に
   サンプルに倣って、デバッグ出力したい構文の
   仮想 Javaソース をチェックするテストメソッドを追加

2. Termianl からテストを実行

```
$ ./gradlew test
```


1.構文の木構造を学ぶ (3/3)

3. デバッグ出力内容を確認
   build/test-results/binary/
   TEST-...PsiClassStructureDetectorTest.xml を確認

- 全てのテスト結果が1つに出力されるので、
  確認したいテストのみを記述することを勧めます

- checkProject が2回実行される事に注意
  1つのテストメソッドおいて beforeCheckProject と
  afterCheckProject が2回実行されます。

1.構文のデバッグ出力 (1/3)

デバッグ出力フォーマット

```
beforeCheckProject (Ph.1)<<<

beforeCheckFile (Ph.1) -> Source=>>>
*** チェックするファイルのソース内容 ***
<<<

*** Node デバッグ出力フォーマット ***

afterCheckFile (Ph.1) -> Source=>>>
*** チェックしたファイルのソース内容 ***
<<<

afterCheckProject (Ph.1)<<<
```


1.構文のデバッグ出力 (2/3)

Node デバッグ出力フォーマット

```
Node=Node名
NodeImpl=Node実体名
Source=>>>Nodeテキスト<<<
parent=>>>親Nodeテキスト<<<:親Node実態名
children=>>>[Nodeテキスト:子Node実態名, Nodeテキスト:子Node実態名,
... ]<<<
```


1.構文のデバッグ出力 (3/3)

Node デバッグ出力例

```
Node=PsiReferenceExpression
NodeImpl=EcjPsiReferenceExpression
Source=>>>isSuccess<<<
parent=>>>
if (isSuccess) {
            message = "success";
        }
<<<:EcjPsiIfStatement
children=>>>[isSuccess:EcjPsiIdentifier,
]<<<
```


2.ノードのソースから学ぶ

例えば、if文 を表す PsiIfStatement には、

- 条件式 ⇒ getCondition():PsiElement
- then節(文) ⇒ getThenBranch():PsiStatement
- else節(文) ⇒ getElseBranch():PsiStatement

…が用意されています。

これらのドキュメントは、提供されていないので、
ノードが提供するメソッドを調べることになります。

既存 Detector ソースから学ぶ

lint-checks/ com/android/tools/lint/checks

https://android.googlesource.com/platform/tools/base/+/studio-master-dev/lint/libs/lint-checks/src/main/java/com/android/tools/lint/checks

既存 Detector ソースからは色々と学べます。

- JavaPsiScanner を実装している Detector
  ApiDetector.java, IconDetector.java,
  RtlDetector.java, SdCardDetector.java
  SecurityDetector.java, SupportAnnotationDetector.java,
  UnusedResourceDetector.java, ViewHolderDetector.java
  WrongImportDetector.java

構文解析済ノードの受領手順

Java ソース解析を行うため、
構文解析済みの各種 ノード を提供してもらうには
以下の手順を踏みます。

1. Detector を継承し、
   JavaPsiScanner *interface* を実装した
   独自 Detector クラスを作成

2. getApplicablePsiTypes() をオーバライドして、
   提供して欲しい ノード種別リスト を返すよう実装

構文解析済ノードの受領手順 (2/3)

3. JavaElementVisitor を継承した、
   独自AST Visitor クラス を作成

4. createPsiVisitor() をオーバライドして、
   独自AST Visitor のインスタンスを返すよう実装
   構文解析済ノードを提供してもらえるようにする。

5. 独自AST Visitor の各ノード種別 visitメソッドで
   提供されたノードを受領
   visitClass(PsiClass):void などで受領する。

構文解析済ノードの受領手順 (3/3)

具体的な実装の流れについては、
後章のLint Custom rule 作成の基本フローと
サンプル･プロジェクトのソースを御確認ください。

次のステップでは、
事前に学習した、特定構文の木構造の知識と、
親ノードからの特定子ノードの取得方法の知識で、

提供されたノードの入れ子構造より Javaソースの内容を
把握して、想定している問題がないかチェックします。

問題となる構文構造を想定する

Lint で問題となる構文構造を指摘するには、
問題となる構文構造を想定している必要があります。

- 例えば、メソッドの boolean 引数を
  if文 で使い状態を変更するのでしたら、
  予め別々のことをするのが解っているのですから、
  メソッドに多重の役割を担わせるのを避けるため、
  条件成立の処理関数 と
  条件不成立の処理関数 に分割すべきです。

構文構造を想定する (2/3)

前記の問題となる構文構造は、
単純に考えると、以下のようになるはずです。

1. メソッドが boolean 引数を持つ
2. メソッド内部に if文 がある
3. if文 の 条件式で、boolean メソッド引数を使っている
4. if文 の then ブロックでフィールド変数を変更している
5. if文 の else ブロックでフィールド変数を変更している

構文構造を想定する (3/3)

ここまで想定できれば、
問題となるようなソースの木構造をデバッグ出力し、
幅広く対処できるよう問題構文構造を抽象化して、
トライ＆エラーでプログラムを進めていけば、
Detector が作れると思います。

Javaソース解析プログラム開発は、
地道な努力と作業の繰り返しです。

サンプルの解析ロジック

サンプル・プロジェクトの
ソース解析ロジック詳細については、
時間の都合とソースを見ていただければ判ることから、
ここでは、割愛させていただきます。

特徴的なことは、
変数がローカル変数か否かをチェックするため、
ElementUtil.ScopeBacktrack.seek()で
メソッド内をバックトラックしていることぐらいです。

サンプルの解析ロジック (2/3)

ロジックの概要は、

1. 状態(Field変数)が責務(役割)を表すと単純化
2. 状態を変更しているメソッドが複数存在すれば対象
3. 各メソッドの状態要素の共有度から
   Issue報告(下記)を行う。

- 責務独立:変更状態要素が他と共有されていない
- 責務共有:変更状態要素が他と共有されている
- 責務混在:変更状態要素が他と過不足がある

…という単純なものです。

サンプルの解析ロジック (3/3)

制限事項

- チェックできるのはプリミティブ値のみ
  自クラスのプリミティブ値フィールドが対象です。
  他のクラス・インスタンスの状態変更は
  対象外としてチェックしていません。

- 処理単純化のため、
  内部クラスや匿名クラスでの状態変更については
  無視しています。

補足: checkProject の2回実行

構文の木構造を学ぶでも触れましたが、
テストでは checkProject ⇒ ノードスキャン(走査)が
2回行われています。

1回目と2回目では、ノード構成に違いがあります。

- 1回目のノード走査では、
  Javaソースそのままのノード構成になっています。
- 2回目のノード走査では、
  リテラルや参照式が丸括弧式に置き換えられています。

補足: …2回実行 (2/5)

具体的には、

1. 数値や文字のリテラル(PsiLiteralExpression)や、
   変数名などの参照(PsiReferenceExpression)が
2. 丸括弧式(PsiParenthesizedExpression)に置換、
   空白(WhiteSpace)が付加されたりもしています。

Detector テストに合格するには、
両者ともに対応できなくてはなりません。

補足: …2回実行 (3/5)

サンプル・プロジェクトの supplementary フォルダに
1回目 checkProject_loop1.txt と
2回目 checkProject_loop2.txt の
デバッグ出力内容のファイルを置いておきました。

よろしければ、diff で違いを御確認ください。

補足: …2回実行 (4/5)

2回実行の理由 Lint テスト時のメッセージより

The lint check produced different results when run on the normal test files and a version where parentheses and whitespace tokens have been inserted everywhere.

The lint check should be resilient towards these kinds of differences (since in the IDE, PSI will include both types of nodes.

補足: …2回実行 (5/5)

Your detector should call LintUtils.skipParenthes(parent) to jump across parentheses nodes when checking parents, and there are similar methods in LintUtils to skip across whitespace siblings.

- 様々な環境に対応できるよう、1回目は抽象構文木で、2回目は括弧や空白を含ませています。
- LintUtils.skipParenthes(parent) を使い、不要な括弧ノードをスキップしてください。

補足:任意の繰り返しスキャン指定

ノード走査では、繰り返しを要求することもできます。

- afterCheckProject() で、引数の Context を使って context.requestRepeat(Detector, Scope.JAVA_FILE_SCOPE):void とすれば、ノード走査の繰り返しを要求できます。
- 何度目のノード走査かは、beforeCheckProject() や afterCheckProject() で、context.getPhase():int とすれば判断できます。

Javaソースを解析する
Lint Custom rule 作成の基本フロー

サンプル・プロジェクトの
ソースコードも併せて御確認ください。

独自カスタム・ルール Detector クラスを作成

カスタムルールの Issue 検出クラスは、
Detector を継承し、Javaソースを解析するため、
JavaPsiScanner を実装します。

```
// Detector は、
// com.android.tools.lint.detector.api パッケージのクラス
public class ExampleDetector extends Detector
                        implements Detector.JavaPsiScanner {

… 省略

}
```


報告する問題の情報を表す Issue を生成

前ページの問題検出 Detector クラス内部で、
報告する問題情報の Issue インスタンスを生成します。

```
// Issue, Category, Severity, Implementation, Scope, は、
// com.android.tools.lint.detector.api パッケージのクラスです。
public static final Issue EXAMPLE_ISSUE = Issue.create(
        "MixedWriteFieldsClassification", // Issue ID 文字列
        "変更の～責務混合問題",                  // Issue 問題説明の見出
        "変更する～しました。",                  // Issue 問題対処の説明
        Category.CORRECTNESS,               // Issue カテゴリ
        4,                                  // 深刻度 (軽:1～10:重)
        Severity.WARNING,                   // Issue 報告の深刻種別
        new Implementation(                 // Detectorとのマッピング
                ExampleDetector.class,      // 問題検出 Detector
                Scope.JAVA_FILE_SCOPE)); // 検出対象範囲(複数可)
```


走査前後の独自処理用オーバライド・メソッド

```
… 省略
// プロジェクト走査の前後に処理が必要な場合は、以下をオーバライド
@Override
public void beforeCheckProject(Context context) { … }
@Override
public void afterCheckProject(Context context) { … }

// 各Javaソース・ファイル走査(AST Node visit)の前後に
// 処理が必要な場合は、以下をオーバーライド
@Override
public void beforeCheckFile(Context context) { … }
@Override
public void afterCheckFile(Context context) { … }
… 省略 // Context ⇒ com.android.tools.lint.detector.api
```


Javaソース解析を行う AST Visitor を指定

createPsiVisitor() メソッドをオーバライドして、
Javaソース解析の独自 AST Visitor インスタンスを
指定します。

```
… 省略
// JavaElementVisitor ⇒ com.intellij.psi パッケージ
// JavaContext ⇒ com.android.tools.lint.detector.api
@Override
public JavaElementVisitor createPsiVisitor(
                                @NonNull JavaContext context) {
    return new JavaElementExampleVisitor(context);
}
… 省略
```


AST Visitor 解析対象の ASTノードを指定(限定)

getApplicablePsiTypes() をオーバライドして、
解析対象とする AST(抽象構文木)ノードを指定します。

```
    … 省略
    // AST ノードの PsiElement, PsiClass, PsiMethod は,
    // com.intellij.psi パッケージのクラス
    @Override
    public List<Class<? extends PsiElement>>
                                        getApplicablePsiTypes() {
        return Arrays.asList(
                PsiClass.class,
                PsiMethod.class);
    }
    … 省略
```


Javaソースを解析する AST Visitor クラスを作成

```
private static class JavaElementExampleVisitor
                                extends JavaElementVisitor {
    private final JavaContext mContext;
    private JavaElementWalkVisitor(JavaContext context) {
        mContext = context;
        … 省略
    }

    // getApplicablePsiTypes() で指定したASTノードの処理先
    // 解析する visit ノード・エントリ用のメソッドをオーバライドして、
    // 引数 ASTノードを解析して問題を検出する処理を実装します。
    @Override
    public void visitClass(PsiClass aClass) { … }
    @Override
    public void visitMethod(PsiMethod method) { … }
    … 省略
}
```


AST ノードの解析について

前章 Android Lint での Java ソース解析の基本 を御参照ください。

問題を検出した場合は、Issue レポートを発行

```
// 問題が検出された AST ノード から Javaソースの該当箇所を特定して、
// Issue レポート を発行(報告)します。
private void myReport(@NonNull PsiElement detected) {
    // Javaソースの問題検出場所を特定
    // Location は、com.android.tools.lint.detector.api のクラス
    String contents = mContext.getJavaFile().getText();
    int startOffset = detected.getTextRange().getStartOffset();
    int endOffset = detected.getTextRange().getEndOffset();
    Location location = createLocation(
            mContext.file, contents, startOffset, endOffset);

    // Issue で報告するメッセージ内容を作成
    String message = "メソッドが〜おすすめします。";

    // Issue レポートを発行
    mContext.report(EXAMPLE_ISSUE, location, message);
}
```


Issue レポートを発行 (2/3)

```
private Location createLocation(
                @NonNull File file, @NonNull String contents,
                int startOffset, int endOffset) {

    DefaultPosition startPosition =
        new DefaultPosition(
                getLineNumber(contents, startOffset),
                getColumnNumber(contents, startOffset),
                startOffset);
    DefaultPosition endPosition =
        new DefaultPosition(
                getLineNumber(contents, endOffset),
                getColumnNumber(contents, endOffset),
                endOffset);
    return Location.create(
            mContext.file, startPosition, endPosition);
}
```


Issue レポートを発行 (3/3)

```
private int getLineNumber(
                @NonNull String contents, int offset) {
    // this li1ne number is 0 base.
    String preContents = contents.substring(0, offset);
    String remContents = preContents.replaceAll("\n", "");
    return  preContents.length() - remContents.length();
}
private int getColumnNumber(
                @NonNull String contents, int offset) {
    // this column number is 0 base.
    String preContents = contents.substring(0, offset);
    String[] preLines = preContents.split("\n");
    int lastIndex = preLines.length -1;
    return preContents.endsWith("\n")
            ? 0
            : preLines[lastIndex].length();
}
```


動作確認するテストの作成

Issue を検出する Detector ができましたら、
動作確認をしてみます。

独自カスタム・ルール Detector のテストを作成

LintDetectorTest を継承して、
Detector テスト・クラスを作成します。

```
// LintDetectorTest は、
// com.android.tools.lint.checks.infrastructure パッケージのクラス
public class ExampleDetectorTest extends LintDetectorTest {

    … 省略

}
```


テスト対象の設定 (1/3)

LintDetectorTest の抽象メソッド
getDetector() を実装して、
テスト対象の Detector を指定します。

```
    private Set<Issue> mEnabled = new HashSet<Issue>();

    protected ExampleDetectorTest getDetector() {
        return new ExampleDetectorTest();
    }
    … 省略
```

テストクラスの設定周りの実装の参考元
googlesamples / android-custom-lint-rules

テスト対象の設定 (2/3)

getIssues() をオーバライドして、
テスト対象の Issue を指定します。

```
… 省略
@Override
protected List<Issue> getIssues() {
    return Arrays.asList(ExampleDetector.EXAMPLE_ISSUE);
}
… 省略

// 次ページの
// TestConfiguration は、LintDetectorTest のサブクラス
// LintClient ⇒ com.android.tools.lint.client.api
// Issue, Project ⇒ com.android.tools.lint.detector.api
```


テスト対象の設定 (3/3)

getConfiguration() をオーバライドして、
テスト対象の Issue のみをテストするよう指定します。

```
… 省略
@Override
protected TestConfiguration getConfiguration(
                    LintClient client, Project project) {
    return new TestConfiguration(client, project, null) {
        @Override
        public boolean isEnabled(@NonNull Issue issue) {
            return super.isEnabled(issue)
                    && mEnabled.contains(issue);
        }
    };
}
… 省略
```


テスト・メソッド作成 (Issue無検出パターン)

```
public void testNoIssueClass() throws Exception {
    mEnabled.clear();
    mEnabled.addAll(Arrays.asList(
                ExampleDetector.EXAMPLE_ISSUE));
    String expected = "No warnings.";  // 警告なしを指定
    String result = lintProject(
        java(   // 解析させるJavaソースの仮想ファイルパス
                "src/test/pkg/NoIssueClass.java",
                // 解析させるJavaソースのコード・テキスト
                "" + "package test.pkz;\n"
                   + "public class NoIssueClass {\n"
                   … 省略
                   + "}\n"
            )
    );
    assertEquals(expected, result);
}
```


テスト・メソッド作成 (Issue検出パターン)

```
public void testExistIssueClass() throws Exception {
    mEnabled.clear();
    mEnabled.addAll(Arrays.asList(
                ExampleDetector.EXAMPLE_ISSUE));

    // 検出ソース箇所(行と位置)とエラー数と警告数まで指定します。
    String expected = "" //  想定している警告メッセージを指定
            + "src/test/pkg/ExistIssueClass.java:18: Warning: "
            … 警告メッセージ省略
            + "    public void greet() {\n"
            + "                        ~~~~~\n"
            + "0 errors, 1 warnings\n";
    String result = lintProject(
        java( … 省略 … )
    );
    assertEquals(expected, result);
}
```


作成したテストの実行

Android Studio の Terminal で、
以下の gradle コマンドを使ってテストを実行します。

```
$ ./gradlew clean
$ ./gradlew test
```

テスト結果は、XML形式で下記に出力されます。
build/test-results/binary/

テストが成功しましたら、
作成した Issue を保管できるようにします。

Issue 保管庫クラスを作成

IssueRegistry を継承して、
Issue 保管庫クラスを作成します。

```
// IssueRegistry は、com.android.tools.lint.detector.apiのクラス
public class ExampleIssueRegistry extends IssueRegistry {
    // getIssues() をオーバライドして、保管する Issue を指定します。
    // (注) Issue により、Detector も特定されます！
    @Override
    public List<Issue> getIssues() {
        return Arrays.asList(
                ExampleDetector.EXAMPLE_ISSUE); // 複数指定可
    }
}
```


build.gradle に Lint 用の jar 作成設定を追加

```
// Jar タスクに、作成する Android Lint 用の属性を設定して、
// 独自 Lint カスタム・ルールの Jar ファイルを作成させます。
jar {
    // Jar ファイル名を指定
    archiveName 'lint-example.jar'

    // Android Lint 用のマニフェスト属性に、
    // 独自作成した Issue 保管庫を指定
    manifest {
        attributes("Lint-Registry":
                   "com.example.ExampleIssueRegistry")
    }
}
```


独自 Custom Lint rule の jar を作成

Android Studio の Terminal で、
以下の gradle コマンドを使って jar を作成します。

```
$ ./gradlew clean
$ ./gradlew assemble
```

ビルドされた jar は、下記に出力されます。
build/libs/

独自 Custom Lint rule を使えるようにする

ビルドした 独自 Custom Lint rule (jar)を
以下にコピーして、Android Studio から使える
(参照できる)ようにします。

```
# MAC, Linux
$ mkdir ~/.android/lint
$ cp 独自カスタムLint.jar ~/.android/lint
```

```
# Windows
> cd Users¥ユーザ名
> mkdir .android¥lint
> copy 独自カスタムLint.jar .android¥lint
```


ASで、独自 Custom Lint rule を利用する

Android Studio を起動すれば、
独自 Custom Lint rule が有効になっています。

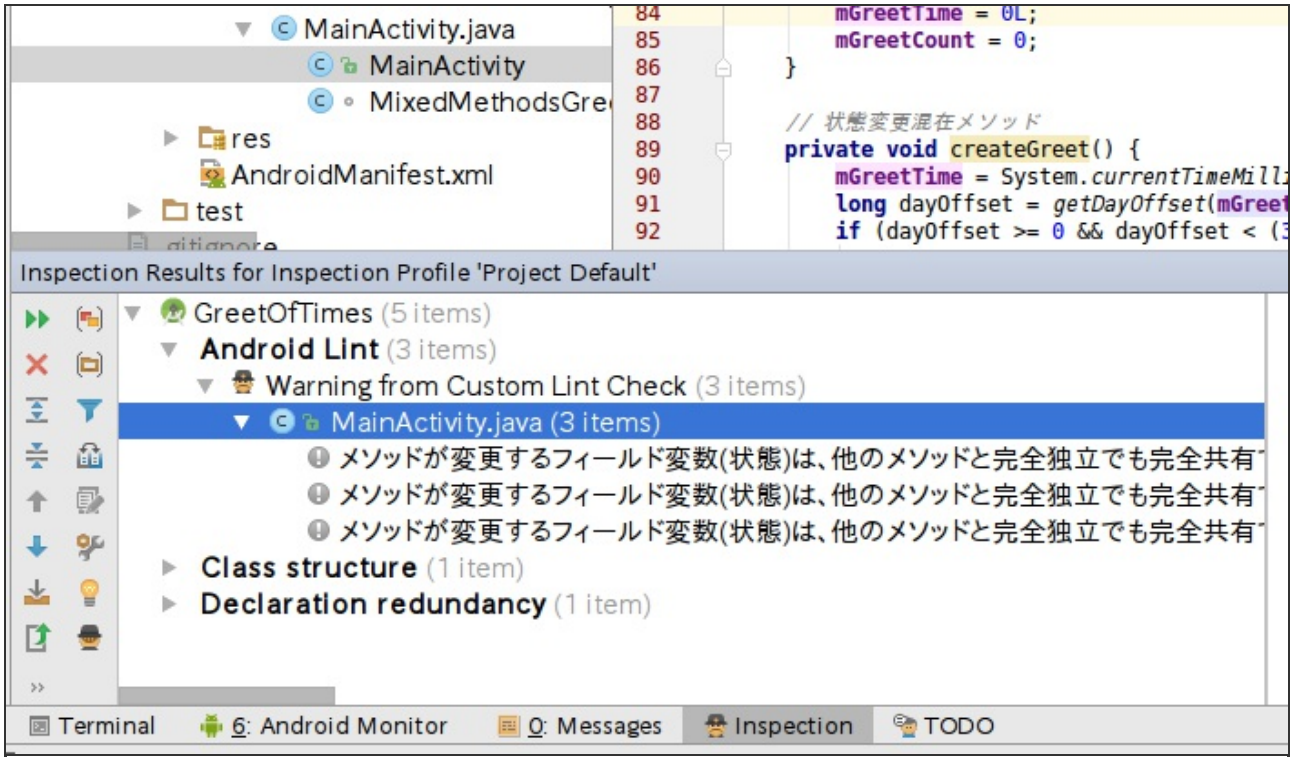
- コードに Issue が有れば、
  エディタ上で問題箇所がハイライト表示されます。

- Analyze メニューの"Inspect Code..."を実行すると、
  Android Lint >
  **　Warrning from Custom Lint Check** に、
  検索範囲に含まれる全 Issue がリストアップされます。

ASで、独自 Custom Lint rule を利用する (2/3)

（エディタ上での問題箇所のハイライト表示）

ASで、独自 Custom Lint rule を利用する (3/3)



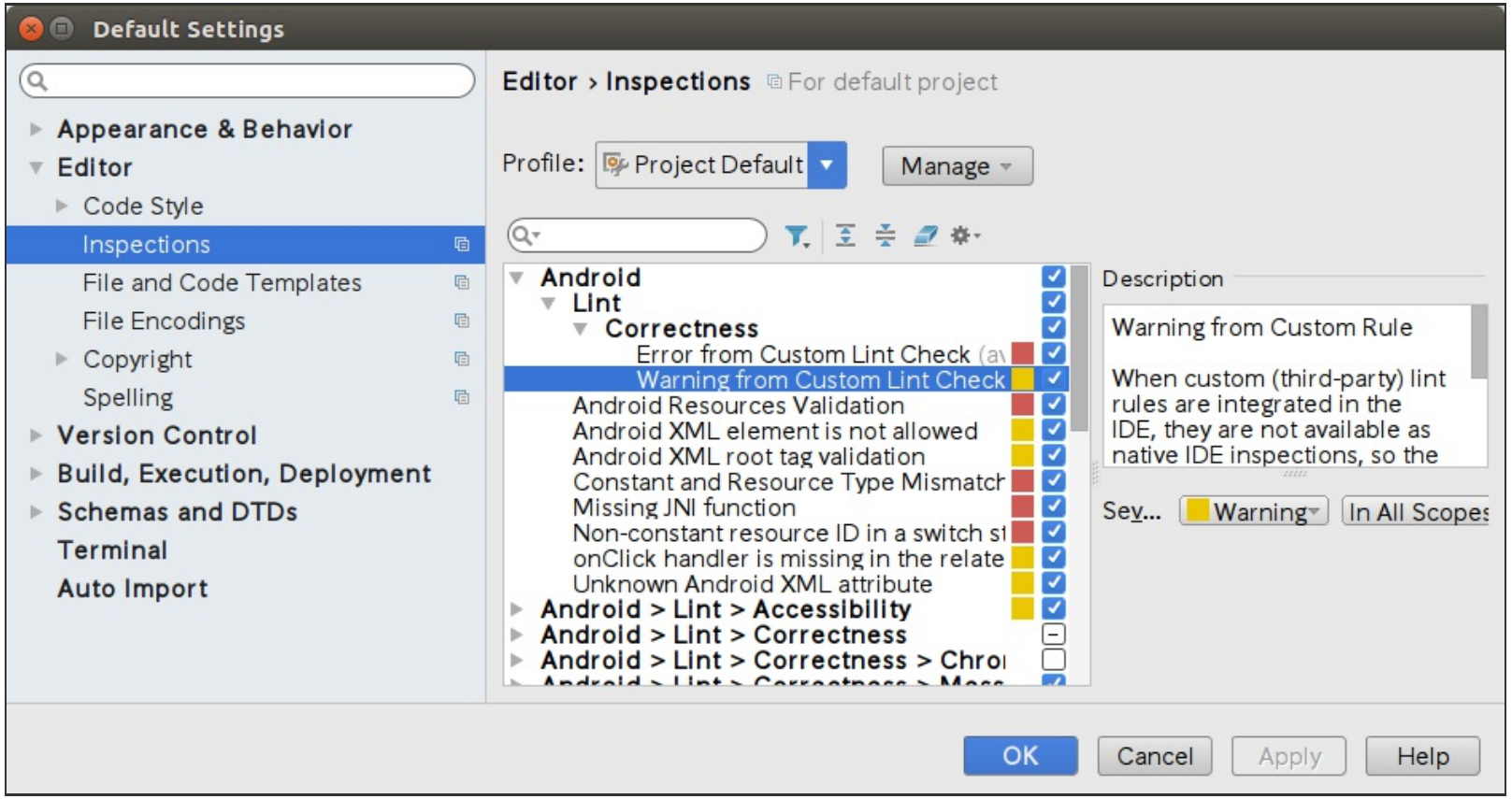
（Inspect Code...での問題箇所リストアップ）

独自 Custom Lint rule 指摘を停止する

File > Other Settings > Default Settings... で、
Default Settings ダイアログを開き、
左ペインで Editor > Inspections を選択して
Editor > Inspections 設定を開きます。

全てのサードパーティの Lint Custom rule は、
Android > Lint > Collectness に該当するので、
Error from Custom Lint Check や
Warrning from Custom Lint Check のチェックを
外せば、インスペクションも無効になります。(fig-1)

独自 Custom Lint rule 指摘を停止する (2/2)



(fig-1)

注意:lint コマンドからは使えません

今回作成した 独自 Lint Custom rule は、Android SDK の lint コマンドラインツールからは使えません。

Android SDK/tools/lib/ の Lint ライブラリ(jar)は、AST(抽象構文木)に lombok.ast のみを使い、com.intellij 関連の API が含まれていないからです。

```
$ lint --list

Could not load custom rule jar file 独自カスタムLint.jar
java.lang.NoClassDefFoundError:
com/android/tools/lint/detector/api/Detector$JavaPsiScanner
```


単一責任の原則の実践

サンプル・プロジェクトの Lint Custom rule と Android Studio のインスペクションを利用して、単一責任の原則を実践してみます。

単一責任の原則の実践

サンプル・プロジェクトのカスタム Lint ルールで、
Android Studio のエディタ画面は、

- フィールド変数(状態)を変更するメソッドが複数あり、
- 自分もフィールド変数(状態)を変更するメソッドに対し、
- 自動的にメソッド名のハイライト(または破線表示)を行なってくれます。⇒ 複合責務の指摘

複合責務の完全な指摘ではないので、
慣れないうちの補助程度にお考えください。

単一責任の原則の実践 (2/2)

サンプル・プロジェクトのカスタム Lint ルールは、
自動的にソースコードを生成するものではありません。
先ずは、自分なりのコードを書いてみてください。

制限事項

- チェック可能なフィールド変数(状態)は、
  プリミティブ値のみです。
- 処理単純化のため、内部クラスや匿名クラスでの
  状態変更は無視しています。

Lint ルール使用時のポイント

- フィールド変数の生成は、
  初期化を明示するためコンストラクタ内で行います。
- 初期化用メソッドは、
  状態変更用メソッドか区別がつかないため設けません。
- デストラクタは、
  状態変更用メソッドか区別がつかないため、
  クロージングなどインスタンス破棄操作が
  必要でない限り設けません。

ルール使用時のポイント (2/2)

- 状態変更メソッドは、
  クラスにただ1つだけになるようにします。
  このポイントは、変更目的が1つの時に限られます。
- ドメインロジック配置用のパッケージを設けます。
  クラスの分割や分離が必要な場合は、
  この下にドメインごとのサブパッケージを設け、
  クラスを配置します。

Lint ルールによる指摘

「メソッドが変更するフィールド変数(状態)」に対し
4種類の指摘をします。

1. 他のメソッドと完全独立しています。 責務独立
2. 他のメソッドと完全共有かつ
   クラス唯一です。 責務共有
3. 他のメソッドと完全共有ですが
   クラス唯一ではありません。 責務共有
4. 他のメソッドと完全独立でも
   完全共有でもありません。 責務混在

指摘ごとの対応

複合メソッドの指摘は、完全ではありません。
誤判定もあるので提案内容にこだわらないでください。

- 責務独立
  指摘が適切と思う場合は、
  ドメインロジックのパッケージに新クラスを作り
  当該メソッドとフィールドおよび関連メソッドを
  分離します。

指摘ごとの対応 (2/3)

- 責務共有 指摘が適切と思う場合は、
  クラスでただ1つの状態変更メソッドを作り、
  各メソッドを状態変更を行わない受け付けメソッドに
  変更して、上記を呼び出します。
  最終的には、責務独立のメソッドを目指します。

指摘ごとの対応（3/3)

- 責務混在
  指摘が適切と思う場合は、
  他メソッドに不足を発生させている、
  変更するフィールド変数(状態)の件数が最大なメソッドを探し、メソッドと変更する状態を分割してみます。
  あるいは、どのメソッドにも含まれる
  最大公約数的な状態がないか探して分離します。
  先ずは、責務独立や責務共有のメソッドを目指します。

最小限の責任を探すこと

単一責任の原則を適切に判断するには、
機械的に実現できない役割の抽象概念が必要です。
指摘の無いことが理想ですが、単一目的でない時の
現実目標は、「責務混在を指摘されない」でしょう。

一番大切なのは、

1. このクラスの責務(役割)は何か、
2. 何を守らなくてはならないのか、

…を意識してクラスを小さくすることだと思います。

単一責任の原則の所感

単一責務の原則の所感は、

- 状態G(責務/役割)は、限定し復数在ってはならない。
- 状態G(責務/役割)とは、ただの Java Bean でなく、
  役割のための制約を守っていなくてはならない。
- 状態Gへの変更要望が復数在り、
  公開された受け口を復数設けていても、
  状態変更メソッドは、目的毎にただ1つだけを目指し、
  制約ロジックや制約機構を持っていなければならない。

…を満たすことなのだと思います。

要求仕様自体が、フロー手続き型になっていても、

「何が、何に対して、メッセージを送るのか」という
オブジェクト指向プログラミングの視点を忘れないよう

心がけたいと思います。

ご清聴、ありがとうございました。