# Rapport du projet tutoré: Custom Lint rules within Android Studio

Création d'une règle personnalisée sur l'analyseur statique de code Android Lint

---

**Date:**

**Lundi 14 Mai 2018**

**Auteurs:**
**Nicolas Poinsignon[1],**

**Sameh El Awadi[1]**

**Professeur encadrant:**
**Olivier Le Goaër[2]**

**30 pages**


[1] Etudiants à l'Université de Pau et des Pays de l'Adour (UPPA), Pau, France
[2] Maître de conférence à l'IUT STID de Pau, France

# Remerciements

Nous tenons à remercier notre professeur encadrant Olivier Le Goaër, pour sa patience dans le déroulement tumultueux de ce projet, pour les connaissances sur le sujet du projet ainsi que pour les conseils qu'il nous a donné lors des périodes difficiles.

Nous remercions également notre professeur Eric Cariou pour ses précieux conseils sur la rédaction de ce rapport ainsi que pour son encadrement des projets tutorés.

# Table des matières

# Abstract

Dans le développement d'applications, les analyseurs statiques de code source (appelés Linter) permettent d'aider le programmeur par la détection de défauts dans le code en temps réel, lors de la programmation. Android Studio est un environnement de développement (IDE) possédant un Linter personnalisable, pouvant intégrer de nouvelles règles. Notre but dans ce rapport est de présenter cet outil, décrire le fonctionnement de ses règles, en mettant l'accent sur les difficultés techniques que présente la création de règles personnalisées, ainsi que détailler la création d'une telle règle avec pour finalité d'ouvrir la marche sur la création de règles spécifiques dans le cadre de futurs projets.

**Mots-clés:**

Android, Linter, Code smell, Static analysis, Abstract Syntax Tree, Custom Lint rule, Kotlin

# 1. Introduction

## 1.1 Contexte du projet tutoré

Dans la programmation mobile, la prévention des anti-patrons (ou anti patterns) énergétiques est actuellement l'un des vecteurs d'approche pour réduire la consommation énergétique des applications mobiles. Il fut en effet montré que la correction des anti patterns mène à une réduction de la consommation énergétique des applications mobiles[3]. L'absence de contrôle sur l'exécution en arrière-plan, la mauvaise configuration des services de diffusion, la création de processus fils (Fork) non-requise, et les fuites mémoires sont des exemples d'anti patterns pouvant nuire à la trace énergétique (footprint) de l'application, et dont la solution réside en une meilleure hygiène de programmation[4].

Dans le cadre d'un projet plus large visant à détecter ces anti patterns dans le code source d'applications Android au sein de l'IDE Android Studio, l'idée serait d'offrir une aide aux programmeurs Android par rapport au problème exposé en tirant parti des outils déjà présents au sein de cet IDE.

## 1.2 Les Linters

Les Linters sont des analyseurs statiques de code source, permettant la détection de défauts tels que des erreurs de syntaxe, l'absence de commentaires, l'utilisation de fonctions

[3] Anti-patterns and the energy efficiency of Android applications. 2016
https://arxiv.org/pdf/1610.05711.pdf

[4] Android App Mistakes: Avoiding the Anti-Patterns. 2010
https://fr.slideshare.net/commonsguy/android-app-mistakes-avoiding-the-antipatterns

dépréciées, et éventuellement des mauvaises pratiques de programmation. Développé à l'origine en 1970 pour le langage C[5], des Linters sont désormais disponibles pour tous les types de langages, et généralement inclus dans des environnements de programmation. Ils opèrent de deux manières différentes. Soit en temps réel, au fur et à mesure que le programmeur écrit le code, lui signalant dynamiquement erreurs et conseils. Ou alors ils peuvent analyser à la demande, certains Linters fournissant également un rapport détaillé sur le code source ainsi que sur le comportement de l'application. Les Linters permettent donc d'améliorer la qualité du code et de faciliter le développement de l'application pour le programmeur. Dans ce contexte, le Linter qui nous intéresse particulièrement est le Linter inclus dans l'IDE Android Studio.

## 1.3 Android Studio

Android Studio est un IDE construit en 2014 sur la base d'IntelliJ IDEA, l'IDE de JetBrain, une entreprise de développement logiciel basée en République Tchèque. IntelliJ IDEA, originellement conçu pour le développement d'applications Java, fut repris par Google et augmenté afin de supporter le développement d'applications sur la plateforme Android, en remplacement de l'ancien IDE, Eclipse Android Development Tools (ADT). Android studio possède désormais son propre Linter, nommé Android Lint, créé par Tor Norbye, chef de l'équipe de développement d'Android Studio chez Google[6]. Consistant en une version enrichie de l'inspecteur de code d'IntelliJ[7], Android Lint couvre actuellement l'analyse statique de code Java, mais également une multitude de fichiers comme les documents XML, ou encore le code Kotlin, un langage développé par JetBrains et aujourd'hui largement adopté par Android Studio. De plus, ce Linter, déjà très complet en matière de détection de problème, est libre (open source) et permet l'ajout de règles personnalisées.

## 1.4 Les règles personnalisés

L'utilisation de règles Android Lint personnalisées dans le cadre des problèmes énergétiques pourrait représenter une avancée dans le domaine, en permettant la détection d'anti patterns énergétiques. Le Linter assisterait en temps réel le développeur à créer une application respectant des règles permettant de limiter sa consommation énergétique. Les règles Android Lint viennent avec de nombreux paramètres et sont hautement personnalisables. Le développement de telles règles personnalisées pourraient ensuite êtres distribuées et ainsi aider la communauté des développeurs d'applications Android.

En revanche, l'API Android Lint est toujours en développement, et sa documentation déja limitée devient progressivement obsolète au fur et à mesure que l'API est mise à jour.

---

[5] Static program analysis https://en.wikipedia.org/wiki/Static_program_analysis

[6] Tor Norbye sur twitter https://twitter.com/tornorbye , google+ https://plus.google.com/+TorNorbye , StackOverflow https://stackoverflow.com/users/46919/tor-norbye

[7] Documentation sur l'Inspection de code dans l'IDE IntelliJ IDEA https://www.jetbrains.com/help/idea/2017.3/code-inspection.html

Message de Tor Norbye, développeur sur Lint Android[8]:
"*This is probably the best place to find and ask information about changes in lint. However, note that lint's API isn't stable yet so writing custom lint checks is still going to be a frustrating experience with APIs changing gradually. I'd really like to do a push and get things to a point where we can stabilize and support compatibly going forward but there are so many other competing priorities :-)*
*-- Tor* "

En effet, la plupart de la documentation disponible sur internet aujourd'hui est dépassée et mène à la confusion. Les quelques pages publiées par Android même ne sont pas mises à jour et la documentation obsolète figure toujours "comme étant" la documentation actuelle. Un développeur voulant s'informer sur les dernières avancées et mises à jour d'Android Lint peut en revanche se fier à ce groupe de discussion géré par Tor Norbye, conférencier chez Android et développeur sur Kotlin et l'analyseur statique Android Lint.[9]

## 1.5 L'objectif du projet

L'objectif de ce projet est de présenter les démarches à suivre dans la création d'une règle Lint personnalisée, autant dans la recherche de documentation que dans les étapes de création des règles Lint. Ce rapport documente ces démarches et souligne les difficultés rencontrées, afin d'ouvrir la route à la prochaine étape vers la création d'un set de règles Lint dédiés à la détection d'anti patterns énergétiques.

Dans une première partie, nous allons présenter Android Lint et détailler son fonctionnement sommaire. Pour ce faire, il sera nécessaire d'aborder plusieurs technologies liées à Android Studio, et nous présenterons les mécanismes en action lors d'une analyse statique du code. Pour illustrer, il sera également présenté la compilation d'une règle, son installation et son exécution par Android Lint.

Dans une seconde partie, nous allons présenter et détailler la création d'une règle Lint personnalisée basique, et expliquer au fur et à mesure des étapes les choix et les recherches effectués nous ayant permis la création de cette règle malgré le manque de documentation.

[8] Google group lint-dev, message de Tor Norbye sur l'état de l'API Android Lint (11/04/2018) https://groups.google.com/forum/#!topic/lint-dev/PyIJGKk78Dc
[9] Page du Google group lint-dev https://groups.google.com/forum/#!forum/lint-dev

# 2. Présentation d'Android Lint

Android Lint utilise différentes technologies qu'il est important de comprendre dans la mesure où chacune est essentielle à l'élaboration de la règle Lint. Nous allons expliquer ces différentes technologies dans cette partie.

## 2.1 Les technologies utilisées

### 2.1.1 Gradle

Gradle Build Tool est un moteur de production destiné à être utilisé sur la plateforme Java. Gradle est intégré à Android Studio, et il est exécuté lorsque l'on construit et exécute tout projet. Il permet d'automatiser des tâches de constructions ainsi que la gestion des dépendances de notre application. Ces règles sont écrites dans le fichier de construction "build.gradle" en utilisant le langage Groovy.

Lors de la création d'une règle Lint, le fichier de construction "build.gradle" est important : il doit nécessairement contenir des instructions indispensables à la création de notre règle.

Ces instructions sont diverses et permettent de définir la version des différentes technologies utilisées (Version de Java, Lint ou bien de Gradle même), mais également les dépendances du projet, ou encore l'écriture de nombreuses tâches (tasks) telles que le build, les vérifications, les tests et ainsi de suite.

### 2.1.2 Kotlin

Kotlin est un langage de programmation créé en 2011 par l'éditeur JetBrains. Il s'agit d'un langage orienté objet, qui a l'avantage d'être interopérable avec Java car il est également compilé en Bytecode pour la Java Virtual Machine (JVM). Depuis 2017, Kotlin est devenu officiellement le deuxième langage de programmation supporté par Google pour les applications Android[10].

Google encourage l'utilisation de Kotlin pour l'écriture d'une règle Lint car la moitié des classes de l'actuel API d'Android Lint est écrite en Kotlin. De plus, Android Lint utilisera de plus en plus de méthodes virtuelles d'extension[11].

### 2.1.3 Abstract Syntax Tree

L'abstract Syntax Tree (AST) est la technologie utilisée par Android Lint pour naviguer à travers le code source. Il s'agit d'une forme de représentation logique de notre code source qui permet ainsi de structurer nos données d'une manière standard. Dans l'arbre AST, les

[10] Langage Kotlin sur la plateforme Android https://developer.android.com/kotlin/
[11] KotlinConf 2017 https://www.youtube.com/watch?v=p8yX5-lPS6o

noeuds représentent les objets, et les feuilles représentent les variables et constantes[12]. Lors d'une analyse par le Linter, le code source représenté en AST est parcouru à l'aide de visiteurs, qui sont des méthodes spécifiques aux APIs de Lint tel que l'API UAST.

### 2.1.4 L'API UAST

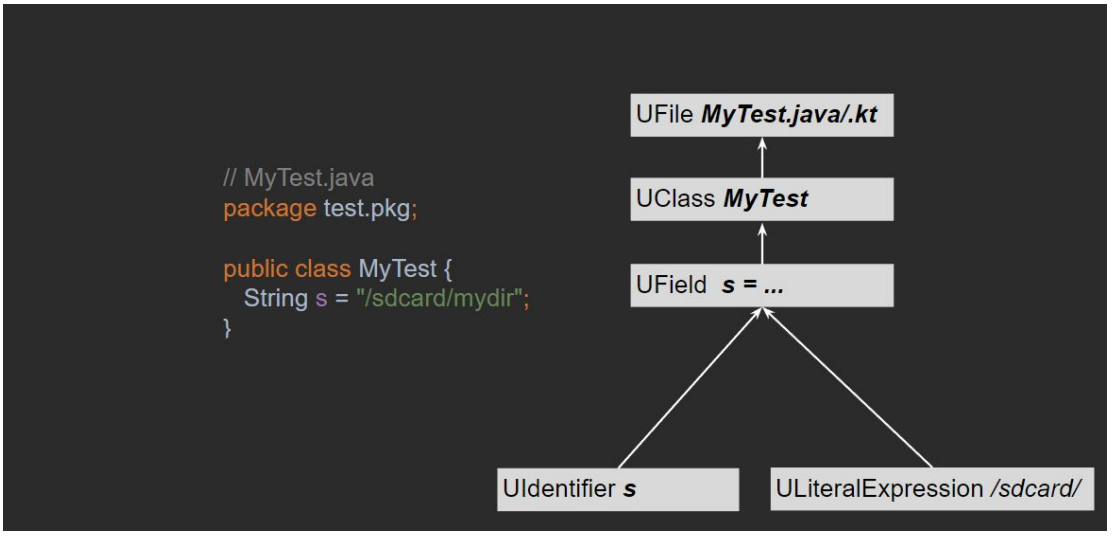
L'API UAST (Universal Abstract Syntax Tree) est l'API qui est à ce jour utilisé par Android Lint. La version actuelle est la 26.1.2. Nous notons que cette API est aujourd'hui majoritairement écrite en Kotlin alors qu'elle était jusqu'ici écrite en Java.

Cette API est principalement utilisée lors de la création d'une règle Lint, car elle permet la représentation sous forme d'arbre syntaxique du code source à l'aide de méthodes visiteurs et de l'objet détecteur (Detector), et permet ensuite de parcourir cet arbre syntaxique. L'atout principal de UAST est sa fonctionnalité, aussi bien sur du code Java que sur du code Kotlin.

Chaque classe de l'API UAST représente un élément du niveau hiérarchique de notre code, classifié par type d'élément syntaxique:

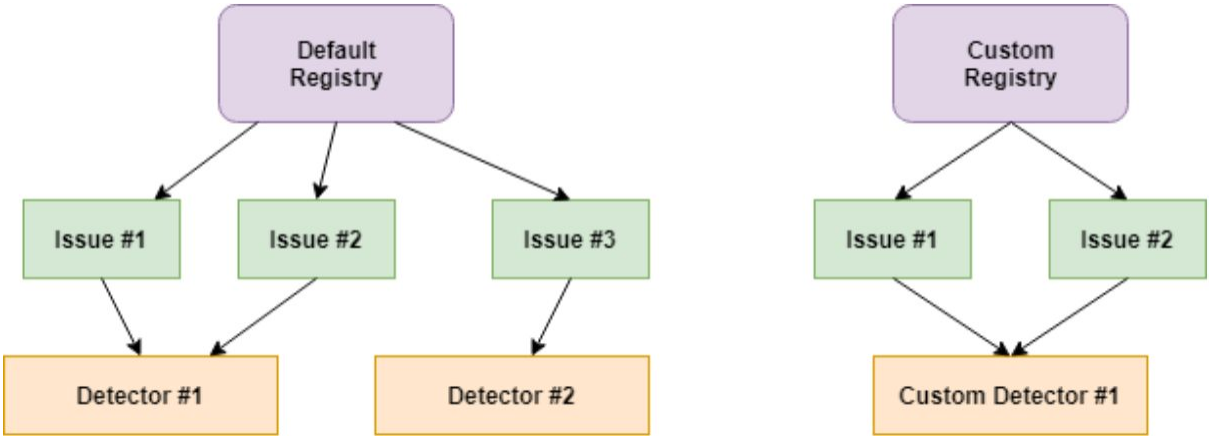
- **UElement:** élément racine
- **UFile:** élément représentant un fichier
- **UClass:** élément représentant une classe
- **UIdentifier:** élément identifiant d'une variable
- **UMethod:** élément représentant une méthode
- **UComment:** élément représentant un commentaire
- **ULiteralExpression:** élément représentant un string
- **…**

La figure suivante permet d'illustrer la structuration d'un code source par l'API UAST, représenté sous forme d'arbre syntaxique abstrait (AST).[13]




[12] L'arbre de syntaxe abstraite https://en.wikipedia.org/wiki/Abstract_syntax_tree
[13] KontlinConf 2017, UAST https://youtu.be/p8yX5-IPS6o?t=1419

### 2.1.5 Composition d'une règle Lint




Ci-dessus une schématisation des éléments constituant les règles Lint

Une règle Lint est composée de divers éléments. Tous ces éléments sont nécessaires à la création d'une règle Lint et sont complémentaires. Les composants sont les suivants:

- Registry Issue
- Issue
- Detector

### 2.1.6 Issues Registry

Le registre des défauts (Issue Registry) est un élément qui liste les différents Issues possibles. Il existe un registre par défaut propre à Lint, toutefois lorsqu'on souhaite créer nos propres règles, nous devons créer notre propre registre.

### 2.1.7 Issues

Un défaut (ou Issue) est un élément qui décrit le problème à identifier. On le déclare le plus souvent dans une variable finale qui est de type Issue. Un Issue n'est rien d'autre qu'une structure de données représentant et décrivant un problème potentiel.
Pour définir un Issue, nous faisons appel à la méthode create() de la classe Issue et celle-ci prend plusieurs paramètres:

- Identifiant
- Titre
- Description
- Catégorie
- Priorité
- Severité
- Implémentation

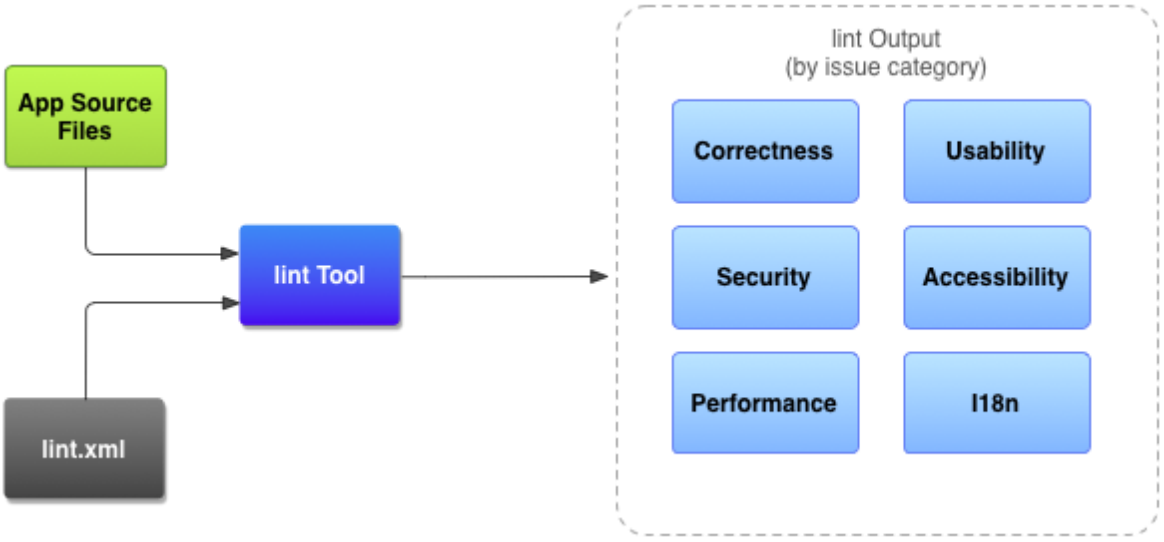
**L'identifiant**
L'identifiant est un paramètre qui permet d'identifier notre Issue. Il est important qu'il soit unique et qu'il ne soit pas nul. L'identifiant utilise la convention de nommage "Upper CamelCase", qui est un ensemble de mots mis bout à bout sans espace, chaque mot débutant par une majuscule[14].

**Titre**
Ce paramètre est une description brève de notre Issue. Le titre donne une idée courte (d'une ligne généralement) et précise le problème relevé. Il sera principalement utilisé pour afficher en temps réel dans l'IDE une description du problème par Lint lors du survol de la souris à l'emplacement du problème détecté.

**Description**
Ce paramètre est une description plus longue du problème qui explique de manière plus complète le détail du problème.

**Catégorie**
Ce paramètre permet d'attribuer une catégorie à l'Issue et de classer celles de même nature. Il existe 6 catégories prédéfinies par Android Lint:[15]

- **Correctness**: Conformité
- **Usability**: Ergonomie
- **Security**: Sécurité
- **Accessibility**: Accessibilité
- **Performance**
- **I18n**: Internationalisation




Ci-dessus, les 6 catégories prédéfinies par Android Lint

[14] camelCase https://fr.wikipedia.org/wiki/Camel_case
[15] Ecrire une règle Lint https://developer.android.com/studio/write/lint

**Priorité**
La priorité représente le niveau d'importance de l'Issue sur une échelle de 1 à 10, 1 représentant l'Issue le moins important et 10 le plus important. Ce paramètre est seulement donné à titre indicatif et n'a aucune influence sur l'issue (il n'as pas de réelle utilité, si ce n'est qu' un indice arbitraire[16]).

**Severité**
Ce paramètre représente la sévérité du problème. Il en existe 4 types:

- **Ignore:** le problème ne sera pas vérifier
- **Warning:** signale un problème potentiel. Cas le plus utilisé
- **Error:** signale une erreur et empêche le build
- **Fatal:** signale une erreur critique qui se déclenche avant même l'assemblement APK

**Implémentation**
Le dernier paramètre est l'implémentation qui permet de relier un Issue à notre classe Detector. L'implémentation est un objet qui prend deux paramètres qui sont la classe objet de notre classe Detector (.class) ainsi que le scope (ou la portée) de notre classe Detector.

Le scope est un élément important car il permet d'indiquer à Android Lint dans quel type de fichier il devra naviguer pour analyser le code source. Il est existe plusieurs types de scope:

- **JAVA_FILE:** permet l'analyse sur les fichiers Java (considérés indépendamment)
- **CLASS_FILE:** permet l'analyse sur les fichiers .class
- **MANIFEST:** permet l'analyse sur les fichiers Manifest
- …

### 2.1.8 Detector

L'objet détecteur (ou Detector) est l'élément le plus important car il contient le coeur de notre règle Lint. Il s'agit d'une classe Java ou Kotlin qui est responsable de la détection des occurrences d'un Issue. C'est dans la classe Detector qu'on implémente un Issue, le Detector pouvant contenir un ou plusieurs Issues.
Cette classe implémente un Scanner, qui est une interface permettant au travers de ses méthodes de parcourir la représentation de notre code sous forme d'AST afin de relever les types d'éléments relatifs à notre Issue.
Il existe différents types de Scanner pour différents types de fichiers:

- **UastScanner:** fichiers Java et Kotlin (visiter avec UAST)
- **XmlScanner:** fichiers XML (visiter avec DOM)
- **ClassScanner:** fichiers Bytecode (.class, visiter avec ASM)
- **BinaryRessourceScanner:** ressources binaires (ex: une image)
- **…**

[16] KotlinConf 2017, qu'est-ce qu'une Issue https://youtu.be/p8yX5-lPS6o?t=723

Nous nous intéresserons ultérieurement aux détails de ces interfaces pour parcourir les arbres AST, et plus particulièrement à l'API UastScanner.

## 2.2 Le choix de l'API pour parcourir l'arbre AST

Afin de parcourir l'arbre AST, nous avons vu qu'il est nécessaire d'utiliser une API propre au Linter, or c'est ici que les choses se compliquent. Au cours de son développement, Android Lint a utilisé trois différentes APIs pour le parcours de l'arbre AST. Nous allons détailler par la suite l'historique de ces versions et expliquer le cas d'utilisation de chacune de ces APIs.

### 2.2.1 L'API Lombok

Le projet Lombok, initié en 2010, est la première API qui permit le parcours de l'arbre AST à Android Lint. L'API Lombok fut donc utilisée à partir de la version 1.0 d'Android Lint en 2011. L'API Lombok possédait alors l'interface Detector.JavaScanner, spécialisée dans le parcours de fichiers sources .java. A cette époque, l'IDE d'Android était encore sous Eclipse Android Development Tool (ATD). La page officielle d'Android sur l'écriture de règles Lint n'a malheureusement pas été mise à jour depuis[17], et la plupart des exemples de règles Lint personnalisées consultables en ligne utilisant cette API sont désormais considérés comme obsolètes (Deprecated) et dont le support officiel se terminera avec l'arrivée de la version stable d'Android Lint.

### 2.2.2 L'API PSI

Une nouvelle librairie nommée PSI (Program Structure Interface) fut développée par JetBrains pour modeler le code Java. Cette API est intégrée à l'environnement d'Android Studio résultant d'un héritage de la plateforme IntelliJ. C'est ainsi qu'une migration de l'API Lombok à l'API PSI fut conseillée lors de la mise à jour d'Android Lint v2.2. L'interface de Lombok Detector.JavaScanner se voit donc remplacée par Detector.JavaPsiScanner, dont les classes et méthodes changent radicalement[18].

### 2.2.3 L'API UAST

Peu de temps après,, une nouvelle API fut développée pour Lint v2.4. L'API UAST (pour Universal Abstract Syntax Tree) fonctionne similairement à l'API PSI mais supporte en plus l'analyse de fichier .kt (Kotlin). Même si le PSI Scanner est plus proche dans le fonctionnement du UScanner que le Scanner Lombok, qui reste encore aujourd'hui supporté, il est vivement conseillé pour la portabilité des règles Lint personnalisées de migrer vers l'API UAST, car le support de l'API PSI sera supprimé dans la version stable de Lint à venir. A ce jour il n'y a pas encore de documentation proprement dite sur l'API UAST. En revanche, si un programmeur souhaite rédiger une règle Lint personnalisée, il peut se

[17] Site Android, écrire un Lint Check http://tools.android.com/tips/lint/writing-a-lint-check
[18] Javadoc PSI API http://www.javadoc.io/doc/com.android.tools.lint/lint-api/25.3.0

rendre sur le Google group lint-dev pour y recueillir des conseils de migrations[19], ou encore directement poser des questions à l'auteur d'Android Lint, Tor Norbye[20]. A la différence des précédentes APIs, UAST est majoritairement écrite en Kotlin, et la décompilation des ressources de l'API permise par Android Studio donne les déclarations des classes et méthodes en langage Kotlin. De ce fait, Tor Norbye recommande vivement d'écrire les règles Lint personnalisées en Kotlin car cela confère divers avantages tels qu'un meilleur support du remplacement et de la correction automatique au sein d'Android Studio, ainsi qu'une plus grande variété d'extensions de méthodes[21].

Ainsi, sans documentation officielle disponible, l'écriture d'une règle Lint spécifique peut s'avérer difficile. Cependant, écrire une règle Lint en Kotlin à partir de l'API UAST permet aujourd'hui d'assurer au maximum la portabilité de la règle vers les prochaines versions d'Android Lint. Notre exemple personnalisé utilisera donc UAST, mais sera en revanche écrit en Java.

## 2.3 Le scope de l'analyse, illustration par un exemple

Lors de la rédaction d'une règle Lint, en particulier la classe détecteur, une notion importante est la portée (ou scope) de la règle. Afin de mieux en démontrer le fonctionnement, nous allons prendre comme exemple la première règle Lint personnalisée que nous avons réussi à faire fonctionner. Cette règle Lint, seul exemple complet fonctionnel que nous ayons trouvé sur internet à ce jour, permet de détecter la présence du mot clef "lint" dans les expressions littérales présentes dans les fichiers Java d'un projet Android Studio.[22]

### 2.3.1 Le type de fichier

Le type de fichier qu'on désire analyser implique l'implémentation du détecteur à partir de la bonne classe Scanner. Dans la version actuelle de Lint, UastScanner est la classe Scanner recommandée. Il s'agit d'un Scanner universel permettant de parcourir tout autant les fichiers Java que les fichiers Kotlin.

Ensuite, lors de l'implémentation de cette classe Scanner dans notre détecteur, il est nécessaire lors de la déclaration de l'Issue d'indiquer son Scope[23]. Dans notre premier exemple, nous allons parcourir des fichiers Java et le scope sera alors définis par `JAVA_FILE_SCOPE`.

[19] Conseils de migration UAST
https://groups.google.com/d/msg/lint-dev/7nLiXa04baM/TuceBxWXAQAJ
[20] Google group lint-dev https://groups.google.com/forum/#!forum/lint-dev
[21] KotlinConf 2017, Écrivez vos règles en Kotlin https://youtu.be/p8yX5-lPS6o?t=562
[22] Exemples officiels de règle Lint personnalisées sur GitHub
https://github.com/googlesamples/android-custom-lint-rules/tree/master/android-studio-3
[23] Javadoc sur l'API Lint v25.3.0
http://static.javadoc.io/com.android.tools.lint/lint-api/25.3.0/com/android/tools/lint/detector/api/Scope.html

### 2.3.2 Le type de noeud

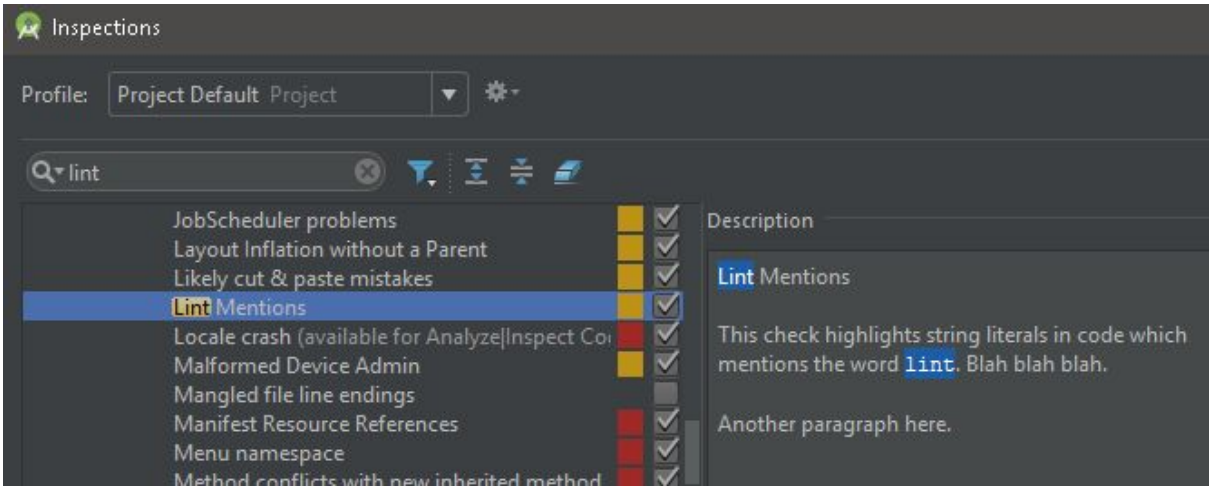
Une fois le bon type de fichier renseigné, il nous faut pouvoir indiquer quel type de noeud dans l'arbre AST nous désirons inspecter. Le choix du noeud s'effectue en plusieurs étapes.

Premièrement, nous devons indiquer au détecteur quel type de noeud doit être parcouru dans l'analyse. Pour ce faire, il faut redéfinir la méthode `getApplicableUastTypes()` du détecteur et lui faire retourner une liste des types de noeuds désirés. Ici, nous souhaitons parcourir l'ensemble des expressions littérales, de type `ULiteralExpression.class`.

Deuxièmement, nous devons redéfinir le bon type de visiteur, c'est-à-dire la méthode de l'`UElementHandler()` qui va naviguer le noeud en question. Dans notre cas, nous voulons "visiter" toutes les expressions littérales, ce qui implique que nous redéfinissons la méthode `visitLiteralExpression()`.

## 2.4 Implémentation de la règle dans le Linter

Une fois la règle finie, elle est prête à être assemblée. Nous utilisons alors les scripts Gradle pour compiler le projet. Une fois compilée, la règle Lint se trouve sous la forme d'un fichier .jar, que nous devons ensuite placer dans un dossier spécifique à Android Lint. Une fois la compilation terminée, la règle apparaît dans la liste des règles Lint par défaut.

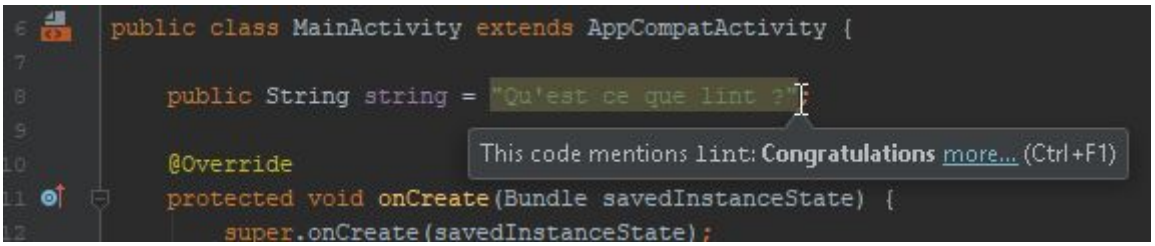



Ci-dessus, La règle apparaît dans la liste des Lint Checks

Une fois la règle fonctionnelle, Android Lint analysera de manière passive tout nouveau projet Android Studio, et détectera la présence, dans cet exemple, de toute déclaration littérale du mot "lint" dans le code source.

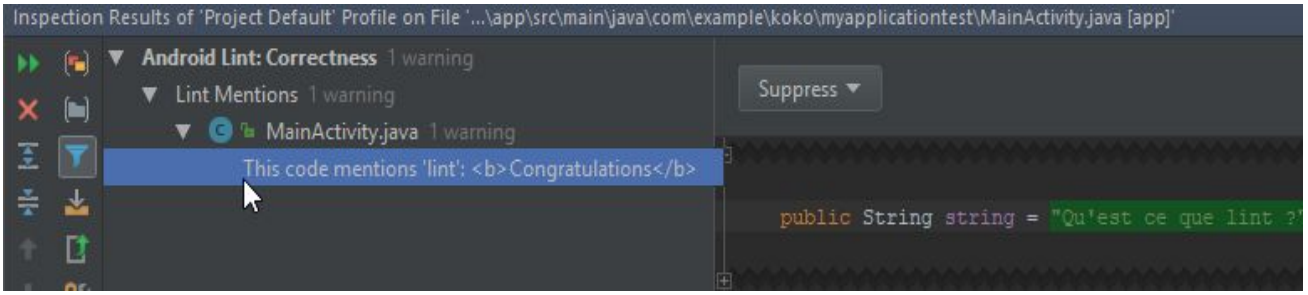
```
if (string.contains("lint") && string.matches(".*\\blint\\b.*")) {
    context.report(ISSUE, expression, context.getLocation(expression),
        "This code mentions `lint`: **Congratulations**");
}
```

Voici par exemple la détection en temps réel (analyse passive) de la règle nouvellement implémentée, dans un nouveau projet Android Studio.




Ci-dessus, la mention en temps réel:

Voici, ensuite, le résultat d'une détection active, déclenchée par l'utilisateur sur le même projet.




Ci-dessus, la mention dans le résultat de l'analyse manuelle

# 3. Création d'une règle personnalisée

Dans cette partie nous allons présenter la création d'une règle personnalisée à partir d'un projet vierge. Nous parlerons des difficultés rencontrées dans l'établissement de la règle, notamment la manière dont nous avons comblé le manque voir l'absence de documentation sur la dernière API, mais également les raisons de nos choix parmi les différentes méthodes proposées par l'API UAST.

Le code source est disponible sur le dépot GitHub de Poinsignon Nicolas, à l'adresse https://github.com/KonscienceGit/CustomLintRule et également en annexe de ce rapport.

## 3.1 Créer la structure de la règle à partir d'Android Studio

Cette première partie dans la création de la règle Lint nous a posé des difficultés pour le déroulement du projet. De nombreux exemples de règles Lint personnalisées ainsi que des tutoriels sont présents sur le web, mais seulement quelques-uns sont à jour, les autres dépendant de l'ancienne version d'Android Studio.

1. Nous créons un nouveau projet Android.

2. Editons le build.gradle principal, présent dans le dossier racine du projet, et déclarons y une variable contenant la version de Lint à utiliser. Ce fichier est présent en annexe page 26.

```
buildscript {
    ext {
        lintVersion = '26.1.2'
    }
...
}
```

Cette version de Lint, 26.1.2 est la version la plus récente à ce jour[24] (9 Mai 2018), il faut noter que la documentation Javadoc ne documente l'API Lint que jusqu'à la version 25.3.0.[25]

3. Créons dans ce projet, un nouveau module de type Java Library (et non un module Android) que nous appellerons "myCheck", et dont la classe s'appellera "MyIssueRegistry".

4. Passons l'éditeur en vue "Project" si ce n'est pas déjà fait, pour pouvoir visionner les fichiers gradle dans leur emplacement réel.

5. Editons le build.gradle dans le nouveau module, et dans

```
dependencies {
```

[24] Maven repository pour le projet Android https://dl.google.com/dl/android/maven2/index.html
[25] Javadoc sur l'API Lint v25.3.0 http://www.javadoc.io/doc/com.android.tools.lint/lint-api/25.3.0

```
    ...
}
```

ajoutons les dépendances suivantes:

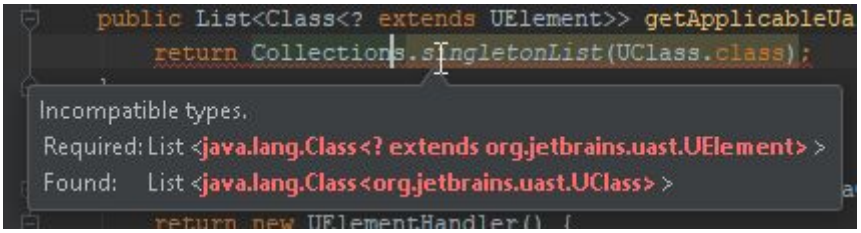
```
    compileOnly "com.android.tools.lint:lint-api:$lintVersion"
    compileOnly "com.android.tools.lint:lint-checks:$lintVersion"
    testImplementation "junit:junit:4.12"
    testImplementation "com.android.tools.lint:lint:$lintVersion"
    testImplementation "com.android.tools.lint:lint-tests:$lintVersion"
    testImplementation "com.android.tools:testutils:$lintVersion"
```

Ici, $lintVersion fait référence à la variable déclarée précédemment dans le build.gradle principal. Ce fichier est consultable en annexe page 27.

6. Nous devons également préciser que la version de Java à utiliser est la 1.8. Modifions donc ces lignes:

```
sourceCompatibility = "1.8"
targetCompatibility = "1.8"
```

En effet, la version 1.7 de Java ne supporte pas les cast de Collections.singletonList vers des listes <Class?<extendsUElement>>.




7. Toujours dans le même build.gradle, nous devons également référencer notre règle, plus précisément le fichier java qui contiendra la liste de ces règles. Ici notre liste de règles est la classe `MyIssueRegistry`.
   Ajoutons donc cette déclaration au gradle.build:

```
jar {
    manifest {
        attributes("Lint-Registry-v2":
        "com.example.lint.mychecks.MyIssueRegistry")
    }
}
```

## 3.2 Créer le registre contenant la liste des problèmes à détecter

8. Dans `MyIssueRegistry.java` nous allons lister les problèmes à détecter dans notre code java. Ce fichier est consultable en annexe page 28.

```
public class MyIssueRegistry extends IssueRegistry {
    @Override
    public List<Issue> getIssues() {
        return Collections.singletonList(MyDetector.ISSUE);
    }
}
```

## 3.3 Écrire la règle

9. Maintenant, nous allons proprement écrire notre règle. Pour cela, créons dans le package `com.example.mycheck` un autre fichier nommé "MyDetector.java", que nous avons déjà référencé dans le fichier précédent.

10. Ici nous allons déclarer la classe détecteur, qui est héritée de la classe Detector ainsi que lui implémenter l'interface qui nous intéresse, dans notre cas: UastScanner.

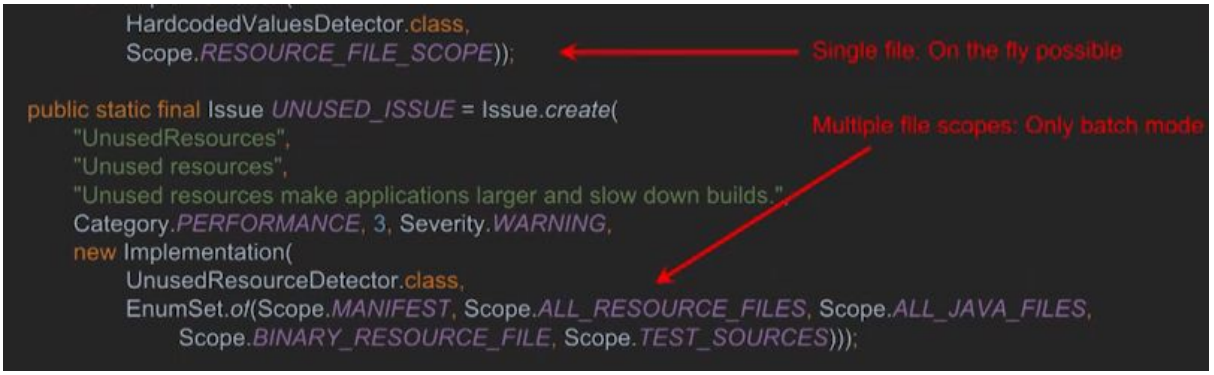
```
public class MyDetector extends Detector implements UastScanner {
...
}
```

Pour rappel, UastScanner est l'interface qui remplace JavaScanner (Lombok) et JavaPsiScanner (PSI) dans la version actuelle d'Android Lint (UAST). Ce fichier est consultable en annexe page 29.

11. Ensuite nous allons définir l'Issue, c'est à dire le problème à identifier par notre détecteur. Pour cela, il nous faut créer une variable de type `Issue` que nous créons par la méthode Issue.create(), à laquelle nous passons les paramètres détaillés dans la section 2.1.7.

```
    public static final Issue ISSUE = Issue.create(
        // ID: utilisé dans les avertissements "warning" @SuppressLint etc
        "Class2Short",
        //Titre -- montré dans le dialogue de préférences de l'IDE,
        //comme en-tête de catégorie dans la fenêtre de résultats de
        l'analyse
        "Class Declaration Too Short",
        //Description complète de l'issue
        "This check highlights classes declaration which name is shorter" +
        " or equal than three character\n" +
        "Class name must be named in a way to identify them properly.\n" +
        "A name too short can't communicate the class purpose properly.\n",
        Category.CORRECTNESS,
        6,
        Severity.WARNING,
        new Implementation(
            MyDetector.class,
            Scope.JAVA_FILE_SCOPE
        )
    );
```

Il peut être noté que le scope de l'Issue peut comprendre plusieurs types de fichiers. En revanche, l'analyse active (on the fly) n'est seulement possible que si un seul type de fichier est indiqué. Il est donc recommandé de faire une règle par type de fichier.




Ci-dessus, un exemple regroupant un simple scope et un scope à multiple fichiers.

12. Nous devons également surcharger la méthode getApplicableUastTypes() dont le rôle est de préciser au Linter quel types de noeuds AST doivent êtres récupérés pour la vérification de la règle. Dans notre cas, nous allons demander tous les noeuds de type Class:

```
    @Override
    public List<Class<? extends UElement>> getApplicableUastTypes() {
        return Collections.singletonList(UClass.class);
    }
```

Cet élément de la règle nous posa problème car aucune documentation n'est actuellement disponible sur les types de UElement acceptables comme retour de la méthode getApplicableUastTypes(). Pour régler le problème, nous avons dû faire divers essais, accompagnés de morceaux d'exemples trouvés sur le groupe lint-dev de Google Groups. Dans notre cas, nous avons eu la chance de trouver un exemple traitant des classes. Mais toute nouvelle règle impliquant des éléments différents nécessitera donc des recherches et expérimentations pour aboutir.

13. Enfin, nous allons écrire la méthode permettant de vérifier si les classes parcourues par le Linter possèdent un nom de plus de trois caractères. Pour ce faire, il nous faut redéfinir le manipulateur d'éléments AST, et lui faire retourner notre méthode de détection, que nous créons en définissant la méthode chargée de visiter les classes. Notons que pour signaler un Issue détecté, nous faisons appel à la méthode report() de l'objet context et lui passons en paramètre l'Issue détecté, le noeud visité, la localisation ainsi qu'un bref texte descriptif de l'Issue:

```
    @Override
    public UElementHandler createUastHandler(JavaContext context) {
        return new UElementHandler() {
            @Override
            public void visitClass(UClass node) {
                String string = node.getName();
                try {
                    if (string.length() <= 3){
                        context.report(
                            ISSUE,
                            node,
                            context.getNameLocation(node),
```

```
                                "This class is too short, please give it a
proper
                                description**");
                        }
                    }
                    catch (Exception e) {
                        System.out.println("Exception occurred about string
                        length");
                    }
                }
            };
    }
```

De la même manière que pour la liste de types d'UElement, trouver le bon type de visiteur peut être périlleux. Il peut être intéressant de noter que la décompilation des bibliothèques de l'API UAST retourne non seulement du code Kotlin, mais également que les listes de visiteurs et de types d'éléments ne sont pas facilement trouvables (à tel point que, nous ne les avons pas encore trouvé dans les dépendances téléchargées par gradle). Dans le cas où un développeur souhaite rechercher dans le code décompilé des dépendances, il peut les trouver, après que Gradle ait effectué un build, à l'arborescence suivante:
`~/.gradle/caches/modules-2/files-2.1/com.android.tools.lint/`

Il est également possible de personnaliser le texte des mentions Lint avec le formattage décrit comme suit:

| | |
|---|---|
| Ceci est du `code source` | Ceci est du `code source` |
| Ceci est en *italique* | Ceci est en *italique* |
| Ceci est en **gras** | Ceci est en **gras** |
| http://url | http://url |
| *Ceci n'est pas en italique* | *Ceci n'est pas en italique* |

14. La règle est désormais finie, il ne nous reste plus qu'à la compiler puis à l'intégrer à notre IDE. Pour compiler notre plugin, dans le terminal d'Android Studio entrons:
    `gradlew clean`
    Suivi de
    `gradlew assemble`

C'est à ce moment là que Gradle téléchargera les dépendances, et indiquera dans le terminal d'Android Studio les potentiels problèmes rencontrés dans l'assemblage du projet.

15. Une fois le build terminé, nous pouvons récupérer le fichier (ici `myCheck.jar`) à l'arborescence suivante dans notre projet Android:
    `CustomLintRule/myCheck/build/libs/myCheck.jar`

## 3.4 Ajout de la règle à la liste des règles Lint par défaut

Maintenant que la règle est compilée, nous devons la déposer dans le répertoire adéquat, ce qui nous a posé un certain nombre de problèmes en raison de la manière dont Android Studio recherche dans ses dossier de dépendances. Le dossier par défaut dans lequel déposer la règle est :

- Sous Windows dans `C:/Users/<votreNomUtilisateur>/.android/lint/`
  Ce dossier `.android` est également accessible depuis la commande "run" d'une session active par la commande `%userprofile%`
- Sous un système UNIX dans `~/.android/lint/`

Attention, si toutefois une variable d'environnement pour le dossier .android a été déclaré par l'utilisateur sur le système, Android studio ira chercher dans `$ANDROID_SDK_HOME`, dans `${user.home}` (répertoire de Java), ou dans `$HOME`.
Pour clarifier: Lint cherche, dans l'ordre, le premier répertoire qui existe parmis:

1. `ANDROID_SDK_HOME` (system prop or environment variable)
2. `User.home` (system prop)
3. `HOME` (environment variable)

Donc, si la variable `ANDROID_SDK_HOME` existe, alors les emplacements définis par les variables user.home et `HOME` seront ignorés. Nous recommandons donc de ne pas créer ces variables d'environnement si ce n'est pas nécessaire.[26]

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

# 4. Conclusion

Durant ce projet il nous a été demandé de produire une règle personnalisée pour Android Lint et de fournir les démarches à suivre pour créer une telle règle, mais aussi de les intégrer à Android Studio. Pour cela nous avons dû nous familiariser avec l'environnement Android Studio, ainsi que les technologies Gradle et AST et plus généralement avec l'API Lint.

Le résultat de notre recherche nous a permis la création d'un guide détaillant pas à pas la création d'une règle personnalisée, et de mentionner les points sensibles et sujets à changement. Notre recherche permettra de fournir les bases dans la création de règles Lint personnalisées dans le contexte de futurs projets tutorés.

Ce projet nous a permis de travailler sur des technologies en développement populaires et d'actualité, et nous a montré les difficultés inhérentes aux technologies sous-documentées. En effet notre projet étant un projet de recherche, nous avons dû nous débrouiller par nous même pour trouver et filtrer les informations sur L'API évoluant rapidement.

Parmi les pistes d'évolution de ce projet, nous pouvons mentionner que la connaissance du langage Kotlin permettrait une plus grande aisance et flexibilité dans l'écriture de règles personnalisées, et fournirait un meilleur support en vue des prochaines évolutions. Nous notons également qu'Android Lint deviendra stable dans un futur proche[27] et disposera non seulement d'une documentation officielle plus fournie, mais supportera également de nouvelles fonctionnalités comme la possibilité de faire des corrections rapides associées à la détection d'un défaut, pouvant pousser encore plus loin l'assistance dans le développement d'applications Android.

[27] KotlinConf 2017, plans for Lint 2.0 https://youtu.be/p8yX5-lPS6o?t=2284

# 5. Références


[1] Poinsignon Nicolas, étudiant à l'université de Pau et des Pays de l'Adour (UPPA), Pau, France.
@contact: npoinsignoncontact@gmail.com
Site web personnel: https://konsciencegit.github.io/NicolasP.io/

Sameh El Awadi, étudiant à l'université de Pau et des Pays de l'Adour (UPPA), Pau, France.
@contact: sameh@elawadi.fr
Site web personnel: https://github.com/Metallink

[2] Olivier Le Goaër, Maître de conférence à l'IUT STID de Pau, France.
Site web personnel: http://olegoaer.perso.univ-pau.fr/

[3] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. "Anti-patterns and the energy efficiency of Android applications". IEEE. 2016
Disponible: https://arxiv.org/pdf/1610.05711.pdf

[4] Android App Mistakes: Avoiding the Anti-Patterns. 2010
https://fr.slideshare.net/commonsguy/android-app-mistakes-avoiding-the-antipatterns

[5] Static program analysis, page Wikipedia
https://en.wikipedia.org/wiki/Static_program_analysis

[6] Tor Norbye, développeur en chef sur Android Studio et auteur d'Android Lint
sur twitter https://twitter.com/tornorbye
sur google+ https://plus.google.com/+TorNorbye
sur StackOverflow https://stackoverflow.com/users/46919/tor-norbye

[7] Documentation sur l'Inspection de code dans l'IDE IntelliJ IDEA
https://www.jetbrains.com/help/idea/2017.3/code-inspection.html

[8] Google Groups lint-dev, message de Tor Norbye sur l'état de l'API Android Lint
https://groups.google.com/forum/#!topic/lint-dev/PyIJGKk78Dc

[9] Page du Google Groups lint-dev
https://groups.google.com/forum/#!forum/lint-dev

[10] Présentation du langage Kotlin sur la plateforme Android
https://developer.android.com/kotlin/

[11] KotlinConf 2017, write your checks in Kotlin
https://youtu.be/p8yX5-lPS6o?t=562

[12] L'arbre de syntaxe abstraite, page Wikipedia
https://en.wikipedia.org/wiki/Abstract_syntax_tree

[13] KontlinConf 2017, présentation de l'UAST
https://youtu.be/p8yX5-lPS6o?t=1419

[14] CamelCase, page Wikipedia
https://fr.wikipedia.org/wiki/Camel_case

[15] Site Android, écrire une règle Lint
https://developer.android.com/studio/write/lint

[16] KotlinConf 2017, qu'est-ce qu'un Issue
https://youtu.be/p8yX5-lPS6o?t=723

[17] Site Android, écrire un Lint Check
http://tools.android.com/tips/lint/writing-a-lint-check

[18] Javadoc sur l'API Lint v25.3.0, PSI API
http://www.javadoc.io/doc/com.android.tools.lint/lint-api/25.3.0

[19] Google Groups lint-dev, conseils de migration UAST
https://groups.google.com/d/msg/lint-dev/7nLiXa04baM/TuceBxWXAQAJ

[20] Google Groups lint-dev
https://groups.google.com/forum/#!forum/lint-dev

[21] KotlinConf 2017, écrivez vos règles en Kotlin
https://youtu.be/p8yX5-lPS6o?t=562

[22] Exemples officiels de règle Lint personnalisées sur GitHub
https://github.com/googlesamples/android-custom-lint-rules/tree/master/android-studio-3

[23] Javadoc sur l'API Lint v25.3.0
http://static.javadoc.io/com.android.tools.lint/lint-api/25.3.0/com/android/tools/lint/detector/api/Scope.html

[24] Maven repository pour le projet Android
https://dl.google.com/dl/android/maven2/index.html

[25] Javadoc sur l'API Lint v25.3.0
http://www.javadoc.io/doc/com.android.tools.lint/lint-api/25.3.0

[26] Site Android, écrire une règle Lint personnalisée
http://tools.android.com/tips/lint-custom-rules

[27] KotlinConf 2017, plans pour Lint 2.0
https://youtu.be/p8yX5-lPS6o?t=2284

# 6. Annexes

## 6.1 Fichier build.gradle principal, (dossier CustomLintRule)

```
// Top-level build file where you can add configuration options common
// to all sub-projects/modules.

buildscript {
    ext {
        /* la dernière version de Android Lint à l'heure actuelle*/
        /* the latest up to date stable version of Android Lint*/
        lintVersion = '26.1.2'
    }

    repositories {
        google()
        jcenter()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:3.1.2'

        // NOTE: Do not place your application dependencies here; they
        // belong in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

## 6.2 Fichier build.gradle du module myCheck

```
apply plugin: 'java-library'

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    compileOnly "com.android.tools.lint:lint-api:$lintVersion"
    compileOnly "com.android.tools.lint:lint-checks:$lintVersion"
    testImplementation "junit:junit:4.12"
    testImplementation "com.android.tools.lint:lint:$lintVersion"
    testImplementation "com.android.tools.lint:lint-tests:$lintVersion"
    testImplementation "com.android.tools:testutils:$lintVersion"
}

/* La version 1.8 de Java JRE est nécessaire pour le bon fonctionnement de
cette règle*/
/* Java JRE 1.8 is required in order for this rule to work*/
sourceCompatibility = "1.8"
targetCompatibility = "1.8"

jar {
    manifest {
        attributes("Lint-Registry-v2":
            "com.example.mycheck.MyIssueRegistry")
    }
}
```

## 6.3 Fichier myIssueRegistry.java

```
package com.example.mycheck;

import com.android.tools.lint.client.api.IssueRegistry;
import com.android.tools.lint.detector.api.Issue;
import java.util.Collections;
import java.util.List;

//Cette classe contient les "issues" qui seront vérifiées
public class MyIssueRegistry extends IssueRegistry {
    @Override
    public List<Issue> getIssues() {
        return Collections.singletonList(MyDetector.ISSUE);
    }
}
```

## 6.4 Fichier myDetector.java

```
package com.example.mycheck;

import com.android.tools.lint.client.api.UElementHandler;
import com.android.tools.lint.detector.api.Category;
import com.android.tools.lint.detector.api.Detector;
import com.android.tools.lint.detector.api.Detector.UastScanner;
import com.android.tools.lint.detector.api.Implementation;
import com.android.tools.lint.detector.api.Issue;
import com.android.tools.lint.detector.api.JavaContext;
import com.android.tools.lint.detector.api.Scope;
import com.android.tools.lint.detector.api.Severity;
import org.jetbrains.uast.UClass;
import org.jetbrains.uast.UElement;
import java.lang.String;
import java.util.Collections;
import java.util.List;

public class MyDetector extends Detector implements UastScanner {
    //Issue décrivant le problème et pointant vers l'implémentation
    //du détecteur

    public static final Issue ISSUE = Issue.create(
        // ID: utilisé dans les avertissements "warning" @SuppressLint etc
        "Class2Short",

        //Titre -- montré dans le dialogue de préférences de l'IDE,
        // comme en-tête de catégorie
        // dans la fenêtre de résultats de l'analyse, etc
        "Class Declaration Too Short",

        //Description complète de l'issue
        "This check highlights classes declaration which name is shorter" +
        " or equal than three character\n" +
        "Class name must be named in a way to identify them properly.\n" +
        "A name too short can't communicate the class purpose properly.\n",

        Category.CORRECTNESS,
        6,
        Severity.WARNING,
        new Implementation(
            MyDetector.class,
            Scope.JAVA_FILE_SCOPE
        )

    );
```

```
    @Override
    public List<Class<? extends UElement>> getApplicableUastTypes() {
        return Collections.singletonList(UClass.class);
    }

    @Override
    public UElementHandler createUastHandler(JavaContext context) {
        return new UElementHandler() {
            @Override
            public void visitClass(UClass node) {
                String string = node.getName();
                try {
                    if (string.length() <= 3){
                        context.report(ISSUE,
                            node, context.getNameLocation(node),
                        "This class is too short, please give it a proper
                        description**");
                    }
                }
                catch (Exception e) {
                    System.out.println("Exception occurred about string
                        length");
                }
            }
        };
    }
}
```