

# Mise en œuvre de la qualité avec SonarQube

---

David THIBAU – 2024

david.thibau@gmail.com



# Agenda

---

## **Introduction à la qualité**

- Analyse continue
- Métriques et charte qualité

## **SonarQube**

- Architecture, éditions
- Installation,
- Concepts Sonar,
- Administration
- Workflow des issues

## **Projet Qualité**

- Recommandations générales,
- Configuration de l'analyse,
- Personnalisation profils et portes qualité
- Règles personnalisées

## **SonarLint**

- Installation IDE

## **Administration et intégration**

- Intégration SCM
- WebHooks
- Intégration Jenkins
- Monitoring



# Introduction à la qualité

---

**Analyse continue**  
Métriques et charte qualité



# Constat

---

En moyenne 80% du coût d'un logiciel concerne la maintenance

Le coût de maintenance est très variable en fonction de la **qualité interne** du software.

=> Le niveau de maintenabilité d'un logiciel détermine son coût financier global



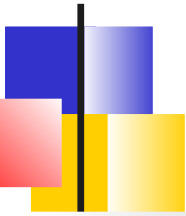
# Approche standard

---

Les approches traditionnelles du contrôle de la qualité des codes impliquaient des audits ponctuels généralement effectués par des auditeurs externes.

=> Perturbations dans le cycle de développement : retards, retouches, remises en cause des choix

=> Tendence à donner une image coûteuse de la qualité .



# Hétérogénéité des exigences

---

Les approches s'appuyant sur des valeurs absolues ne sont pas viables car elles sous-entendent que tous les projets ont les mêmes objectifs de qualité

- un projet existant peut ne pas être tenu au même niveau de qualité qu'un nouveau projet
- Le développement interne pourra être jugé différemment du code externalisé.

=> Obliger chaque projet à atteindre les mêmes valeurs absolues avant la mise en production est donc souvent impraticable.



# Analyse continue

---

L'**analyse continue** consiste à intégrer la démarche qualité dans le cycle de développement du logiciel

- trouver les problèmes le plus tôt possible ; lorsqu'il est facile et peu coûteux de les corriger.

Des audits automatisés sont effectués quotidiennement et les résultats sont mis à disposition des acteurs du projet.

- Permet une appropriation collective du code et des bonnes pratiques.

Le but étant d'améliorer le ROI d'un projet de développement



# Amélioration continue

---

Des audits peuvent être également effectués continuellement dans l'environnement du développeur.

- lorsque le code est encore présent à l'esprit et qu'il n'est pas encore enregistré dans le dépôt de source.

Cela permet au développeur de s'améliorer et de se débarrasser de ses mauvaises pratiques de codage.





# Les 10 principes (1)

---

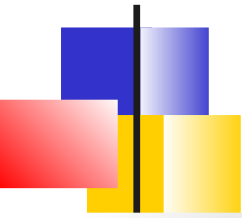
1. Tous les acteurs du projet (pas seulement les développeurs) ont un accès aux données significatives.
2. La gestion de la qualité concerne tout le monde dès le début de projet.  
L'équipe de développement en a la responsabilité ultime.
3. Les exigences qualité sont des éléments de recette du logiciel.
4. Elles doivent être objectives
5. Le plus possible, elles doivent être communes à tous les projets de développement.



# Les 10 principes (2)

---

6. L'analyse doit être à jour et concerner la dernière version du code.
7. Les logiciels doivent être analysés en permanence afin de corriger les problèmes au + tôt.
8. Les acteurs doivent être alertés lorsque de nouveaux défauts sont détectés
9. Les métriques qualités doivent être disponibles en valeur absolue et valeur différentielle (A partir d'un point d'origine, indicateur de progression)
10. La résolution des défauts doit être clairement planifiée et provisionnée.



# Introduction à la qualité

---

Analyse continue  
**Métriques et charte qualité**



# Métriques

---

La mesure est fondamentale à toutes les disciplines d'ingénierie ;  
le software n'est pas une exception

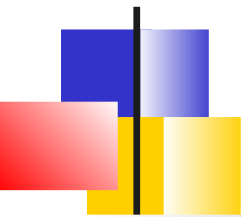
Dans le cadre d'un logiciel, la production de métriques permet :

- Donner un chiffre à des facteurs qualitatifs
- De s'améliorer
- D'affiner les estimations (prix, délais)

2 types de métriques concernent les logiciels :

- Interne : Analyse statique du code source ou compilé  
***Couvert par SonarQube***
- Externe : Tests sur l'application en cours d'exécution  
(Performance, Disponibilité de service, Nbre d'erreurs  
...)  
***Non couvert par Sonarqube***

# ISO 25010



Ce document propose des métriques pour les différentes sous-caractéristiques qualité.

Sonarcube utilise cette norme pour certain concepts et métriques.  
Principalement Fiabilité, Maintainabilité, Sécurité



# Charte qualité

---

La norme ne donne que les métriques

- Elle ne donne pas d'indicateurs
- Exemple : pour l'adéquation fonctionnelle
  - La norme dit « plus c'est proche de 1, mieux c'est »
  - A combien est-ce suffisant ?

La charte qualité (profil qualité Sonarqube)

- Permet d'appliquer des seuils et des critères à des métriques pour obtenir des indicateurs



# Charte qualité

---

Le modèle qualimétrique est neutre

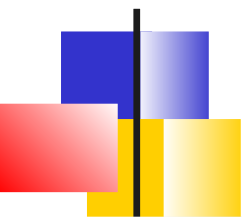
- « formule de calcul du nombre de lignes de code »

La valeur de mesure est neutre

- « nombre de lignes de code=100 »

Le jugement est porté dans une charte qualité

- Spécifique, en fonction des exigences qualité définies
  - Charte qualité Java :  
nombre de lignes de code d'une méthode  $> 20 \rightarrow$  problème sérieux
  - Charte qualité COBOL :  
nombre de lignes de code d'une COPY  $> 200 \rightarrow$  problème sérieux



# SonarQube

---

## **Architecture, éditions**

Installation

Concepts Sonar

Administration

Workflow des issues





# Offre SonarSource

---

La société *SonarSource* propose 3 produits autour de la qualité :

- ***SonarLint*** : Extension IDE gratuite qui fournit une analyse à la volée et des conseils de codage
- ***SonarQube*** : Serveur d'analyse statique pour une inspection continue de la base de code installé à l'intérieur de l'entreprise
- ***SonarCloud*** : Service cloud de SonarQube



# SonarQube

**SonarQube** regroupe tous les outils d'analyse statique toute technologie confondue et les intègre sous forme de plugin

Il permet :

- De définir les règles de codage à appliquer sur un projet
- De détecter les transgressions et d'estimer les métriques qualité en continu
- De définir des chartes qualité par projet : Les porte qualité SonarQube fixant des seuils par métriques calculés

SonarQube permet de démarrer avec une configuration par défaut mais il est souvent nécessaire de l'adapter aux particularités d'un projet.



# Offre Sonar (1)

---

Modèle Open Source : Versions communautaire et payantes :  
Developer, Enterprise et DataCenter

## Fonctionnalités communautaire

- Analyse complète de la qualité d'une application
- Tableau de bord projet
- Historique des statistiques
- Gestion des règles de codage, mise à jour
- Gestion de profils et portes qualité
- Workflow autour de la résolution des issues
- Visualisation du code source, accès aux dépôts de sources
- Intégration CI/CD



# Extensibilité

---

De nombreux plugins permettent d'étendre les fonctionnalités de Sonar :

- Prise en compte d'un langage spécifique
- Analyseurs supplémentaires (OWASP check par exemple)
- Intégration à d'autres outils
- Fonctionnalités (PDF Report)
- ...

Gestion des plugins via la **MarketPlace** : Certains nécessitent l'acquisition de licences



# Édition Developer

---

- SAST plus avancé
- Intégration avec workflow Git  
(Analyse des branches de feature, de maintenance, de pull request)
- Intégration porte qualité dans plateforme DevOps (GitLab, Github, BitBucket, Azure)
- Support de plus de 25 langages et frameworks
- Support commercial



# Édition entreprise

---

- Support de plus de 25 langages et frameworks
- Gestion de Portfolio (suivi transverse de plusieurs projets) + Reporting PDF
- Reporting sécurité (OWASP, CVE, ...)
- Rapport d'audit
- Transfert de projet entre instances Sonar
- Traitement // des rapports d'analyses
- Support Premium

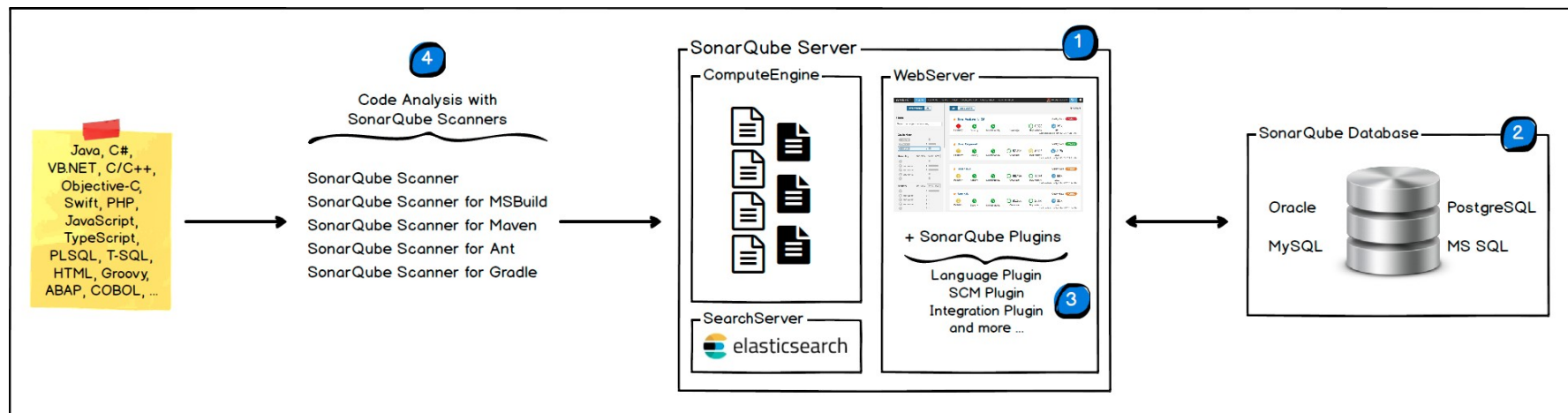


# Edition Data Center

---

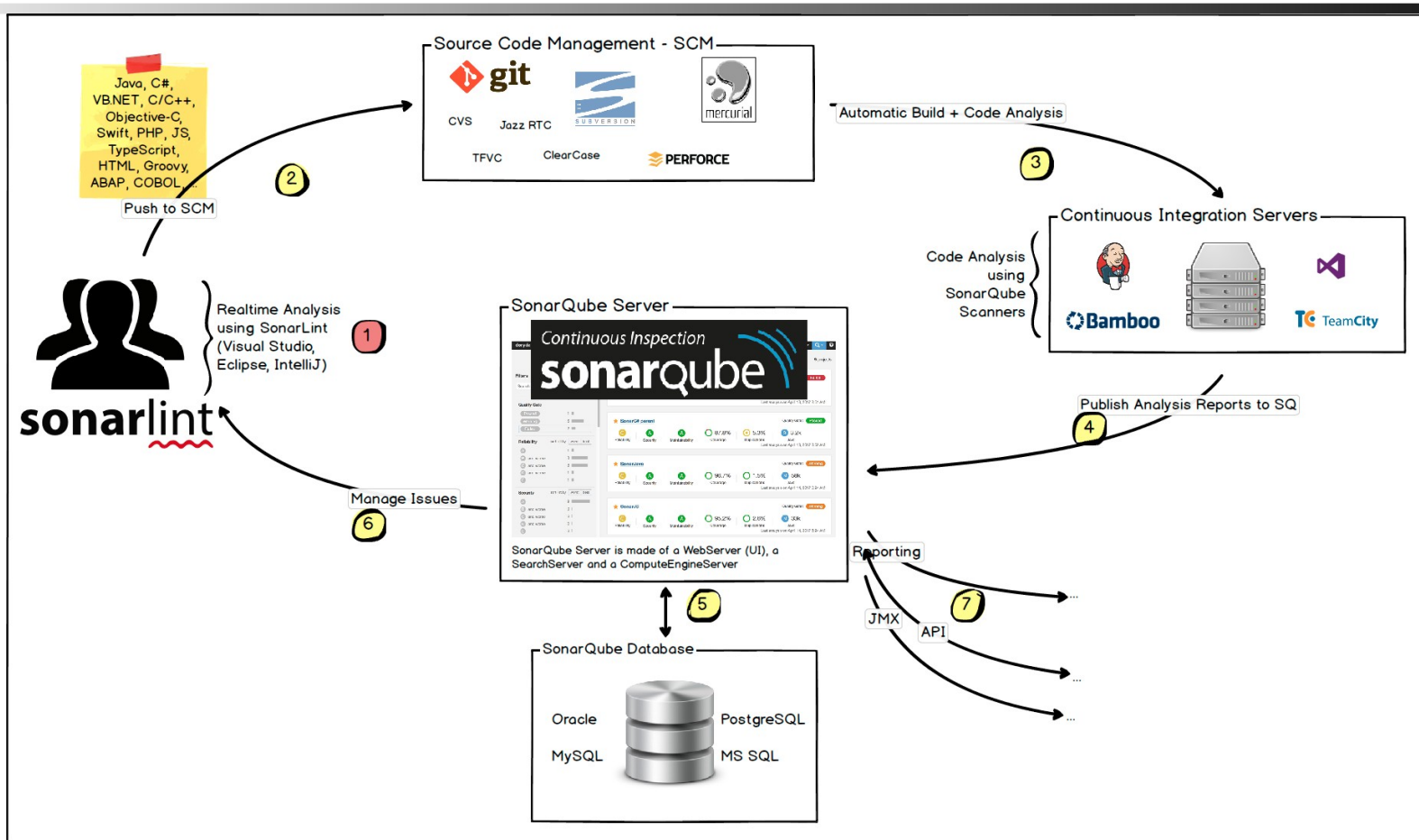
- Redondance des composants
- Résilience des données
- Scalabilité horizontale

# Architecture





# Analyse continue





# SonarQube

---

Architecture, éditions  
**Installation**

Concepts Sonar

Administration

Workflow des issues



# Releases

---

Le développement de Sonar est en mode agile, donc des releases sont fréquemment effectuées.

Certaines releases sont marquées comme **LTS** (Long Term Support)

- Elles sont + stables en terme de fonctionnalités ~ 1an
- Sonar Source s'engage à corriger les bugs
- Lors d'une mise à jour, il y a plus de travail à effectuer



# Hardware

---

Sonar nécessite :

- 2 Go de RAM
- Espace disque en fonction du nombre de projets.  
*SonarCloud* avec plus de 300 projets utilise 15Go
- Disques rapides de préférence



# Pré-requis

---

Pour l'installation SonarQube nécessite une JRE

- Version 10.x : Java 17
- Version 8.X et 9.x : Java 11
- Version 6.x : Java 1.8

Une base de données :

- BD embarquée pour évaluation
- PostgreSQL, MSQl, Oracle

ElasticSearch

- Suffisamment d'espace disque libre
- Pour Linux :
  - *vm.max\_map\_count* > 262144
  - *fs.file-max* > 65536
  - L'utilisateur de SonarQube peut ouvrir 65536 fichiers
  - Et démarrer 2048 threads



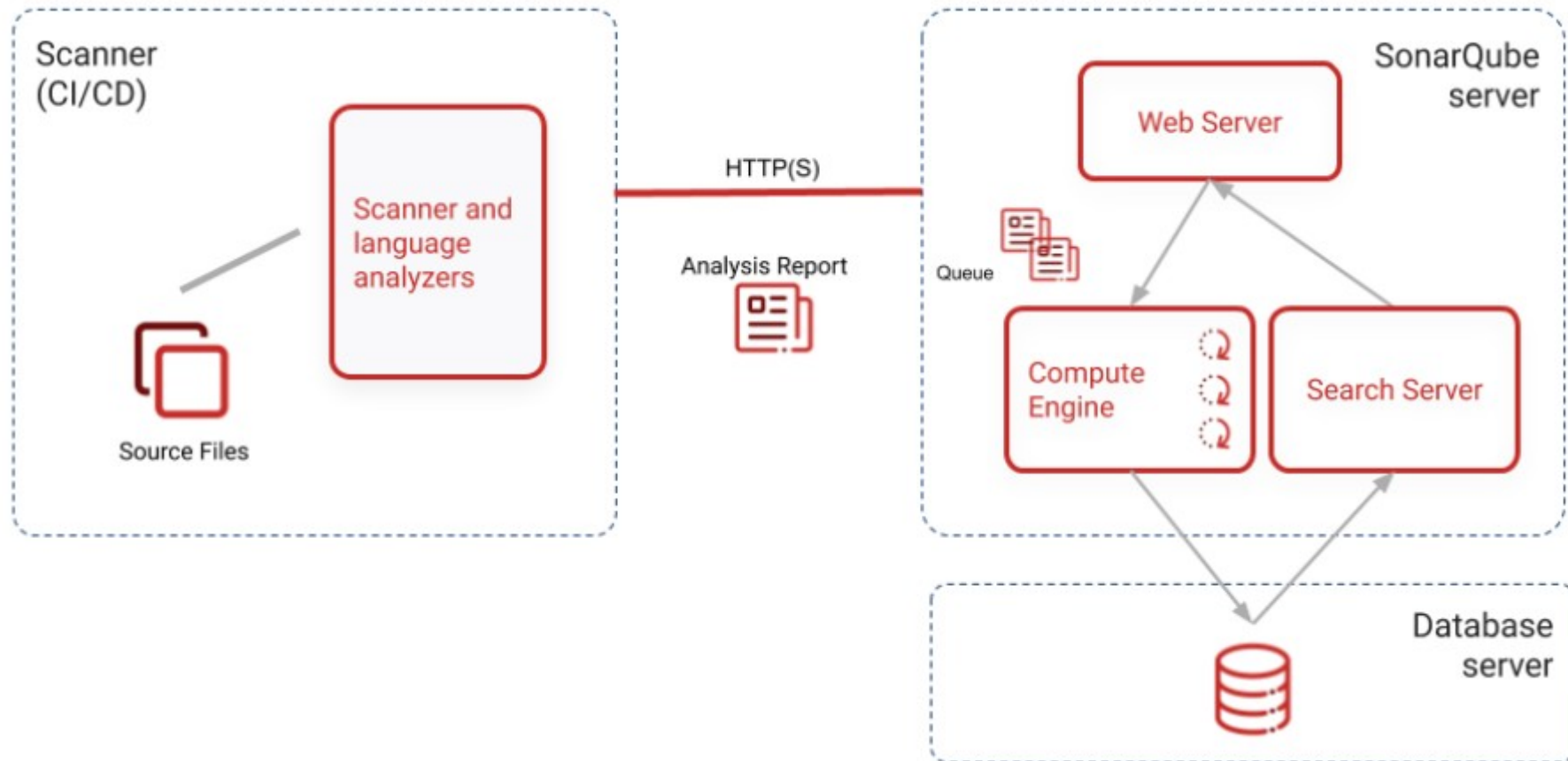
# Distribution

---

La distribution de sonar est disponible sous forme d'archive zip, il suffit de la décompresser

Dans un environnement Linux, des packages sont également disponibles

# Composants SonarQube





# Rôle des composants

---

Le serveur SonarQube exécute 3 processus :

- Le serveur Web qui sert l'interface utilisateur de SonarQube.
- Le serveur de recherche basé sur Elasticsearch.
- Le moteur de calcul en charge de traiter les rapports d'analyse de code et de les enregistrer dans la base de données SonarQube.

La base de données stocke les éléments suivants :

- Métriques et problèmes de qualité et de sécurité du code générés lors des analyses de code.
- La configuration de l'instance SonarQube.

Un ou plusieurs scanners exécutés sur les machines de build pour analyser les projets envoient leurs rapports vers le serveur.





# Base de données

---

SonarQube est livré avec une base de données embarquée, cela n'est pas recommandé en production

Pour installer une autre base :

- Créer un schéma vide et un utilisateur *sonarqube*
- Donner à l'utilisateur *sonarqube*, les droits de créer, mettre à jour et supprimer des tables
- Configurer l'accès JDBC
- Pour une base Oracle, ajouter le driver JDBC



# Configuration JDBC

---

Dans le fichier ***conf/sonar.properties***

```
sonar.jdbc.username=postgres
```

```
sonar.jdbc.password=postgres
```

```
sonar.jdbc.url=jdbc:postgresql://localhost/sonar
```

... éventuellement, configuration du pool de connexions



# ElasticSearch

---

Par défaut, ElasticSearch stocke ses index dans *\$SONARQUBE-HOME/data*

- Cela n'est pas recommandé en production.

Idéalement choisir un volume dédié avec des E/S rapides.

- Cela facilite la mise à niveau de SonarQube et améliore les performances

```
sonar.path.data=/var/sonarqube/data
```

```
sonar.path.temp=/var/sonarqube/temp
```



# Serveur web

---

Le port par défaut est 9000 avec un chemin de contexte à /

Ces valeurs peuvent être modifiées dans *conf/sonar.properties*

```
sonar.web.host=192.0.0.1
```

```
sonar.web.port=80
```

```
sonar.web.context=/sonar
```



# Choisir la bonne version de Java

---

Si il existe plusieurs versions de Java sur le serveur cible.

La version utilisée peut être précisée dans

`$SONARQUBE-HOME/conf/wrapper.conf`

Et la ligne

`wrapper.java.command=/path/to/my/jdk/bin/java`



# Démarrage

---

Linux/Mac OS :

```
bin/<YOUR OS>/sonar.sh start
```

Windows :

```
bin/windows-x86-XX/StartSonar.bat
```

Fichier de traces : `logs/sonar.log`

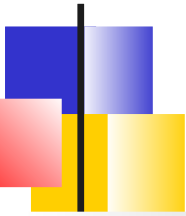
Un utilisateur *admin/admin* est initialement créé

Les options JVM de démarrage peuvent également être indiquées dans  
*conf/sonar.properties*

```
sonar.web.javaOpts=-server
```

On peut obtenir les traces en mode DEBUG avec :

```
sonar.log.level=DEBUG
```



# Installation service Windows

---

SonarQube fournit des scripts  
d'installation de service Windows :

```
%SONARQUBE_HOME%/bin/windows-x86-32/InstallNTService.bat  
%SONARQUBE_HOME%/bin/windows-x86-32/  
UninstallNTService.bat
```

Le service peut ensuite être  
démarré/stoppé par :

```
%SONARQUBE_HOME%/bin/windows-x86-32/StartNTService.bat  
%SONARQUBE_HOME%/bin/windows-x86-32/StopNTService.bat
```



# Installation Service Linux

---

Voir :

<https://docs.sonarqube.org/display/SONAR/Running+SonarQube+as+a+Service+on+Linux>





# Sonar UI

[Projects](#)[Issues](#)[Rules](#)[Quality Profiles](#)[Quality Gates](#)[Administration](#)[More](#)

**Home** : Tableau de bord

**Projets** : Résultat d'analyse des projets, configuration

**Issues** : Issues actives

**Rules** : Gestion des règles (Activation/Désactivation, ...)

**Quality profiles** : Gestion des profils qualité et association aux projets

**Quality Gates** : Gestion des portes qualité et association aux projets

**Administration** : Administration plateforme et configuration transverse des projets



# SonarQube

---

Architecture, éditions  
Installation

**Concepts Sonar**

Administration  
Workflow des issues



# Analyse et Scanners

---

*Sonar* permet de démarrer une analyse grâce aux **Scanners** fournis :

- MSBuild, Maven, Gradle, Ant, Jenkins, Azure DevOps, Commande en ligne

Ceux-ci prennent généralement en paramètre un fichier de configuration propre au projet

Le projet SonarQube est créé :

- Soit à l'avance
- Soit lors de la première analyse



# Analyse et langages

---

*SonarQube* supporte plus de 20 langages

Cependant, ce qui est analysé dépend du langage :

- Pour tous les langages, le code source peut être importé d'un SCM. (ainsi que les données « blame »)  
Git et SVN sont supportés, les autres nécessitent des plugins.
- Pour tous les langages, une analyse statique du code source est faite
- Pour certains langages, le code compilé est également analysé statiquement (.class Java, .dll C#, etc.)
- Une analyse dynamique peut être effectuée pour certains langages.
- Tous les fichiers reconnus par l'édition de SonarQube sont automatiquement analysés



# Étapes d'une analyse

---

L'analyse s'effectue en plusieurs étapes :

1. Des données sont demandées au serveur par le scanner
2. Les fichiers soumis à l'analyse sont analysés et les données résultantes sont renvoyées au serveur sous la forme d'un rapport
3. Le rapport est ensuite analysé de manière asynchrone côté serveur.  
Les rapports sont mis en file d'attente et traités séquentiellement. Un statut des tâches est disponible dans l'interface :

*Administration → Project → Background tasks*



# Analyse

---

## Base de règles

- Les règles sont fournies par SonarSource ou par des plugins
- On peut rajouter ses propres règles
- Elles peuvent être activées/désactivées dans des « profils qualité »

L'analyse consiste à vérifier les règles du *profil qualité* associé au projet.

Lorsqu'une règle est transgressée, elle produit une ***Issue***



# Règle

---


## "hashCode" and "toString" should not be called on array instances

squid:S2116  



Bug

 Major

 No tags

Available Since December 8, 2017

SonarAnalyzer (Java)

Constant/issue: 5min

While `hashCode` and `toString` are available on arrays, they are largely useless. `hashCode` returns the array's "identity hash code", and `toString` returns nearly the same value. Neither method's output actually reflects the array's contents. Instead, you should pass the array to the relevant static `Arrays` method.

### Noncompliant Code Example

---

```
public static void main( String[] args )
{
    String argStr = args.toString(); // Noncompliant
    int argHash = args.hashCode(); // Noncompliant
}
```

---



# Profil Qualité

---

Les ***profils qualité*** définissent les règles pour un projet particulier

*SonarSource* fournit un profil qualité par défaut pour chaque langage

Profil Java ~ 600 règles

- 172 bugs
- 32 sécurité
- 433 Code smells
- 37 Security Hotspots





# Métriques

Sonar définit plusieurs axes pour les métriques qualité associés à un projet :

- **Fiabilité** : Indicateur dépendant du nombre de bugs
- **Maintenabilité** : Indicateur dépendant du nombre de code smells
- **Sécurité** : Indicateur dépendant du nombre de vulnérabilités
- **Hotspots reviewed** : Pourcentage
- **Couverture** : Pourcentage de lignes couvertes par les tests
- **Duplications** : Pourcentage de Copié/Collé
- **Complexité** : Formule
- **Documentation** : Taille du projet, nombre de commentaires

Pour chaque valeur obtenue, Sonarqube indique une approximation de l'effort de correction



# SonarQube - Progression

---

*Sonarqube* permet de définir un point d'origine et de distinguer ainsi le **Nouveau Code (New Code)**.

Chaque métrique est alors évalué pour :

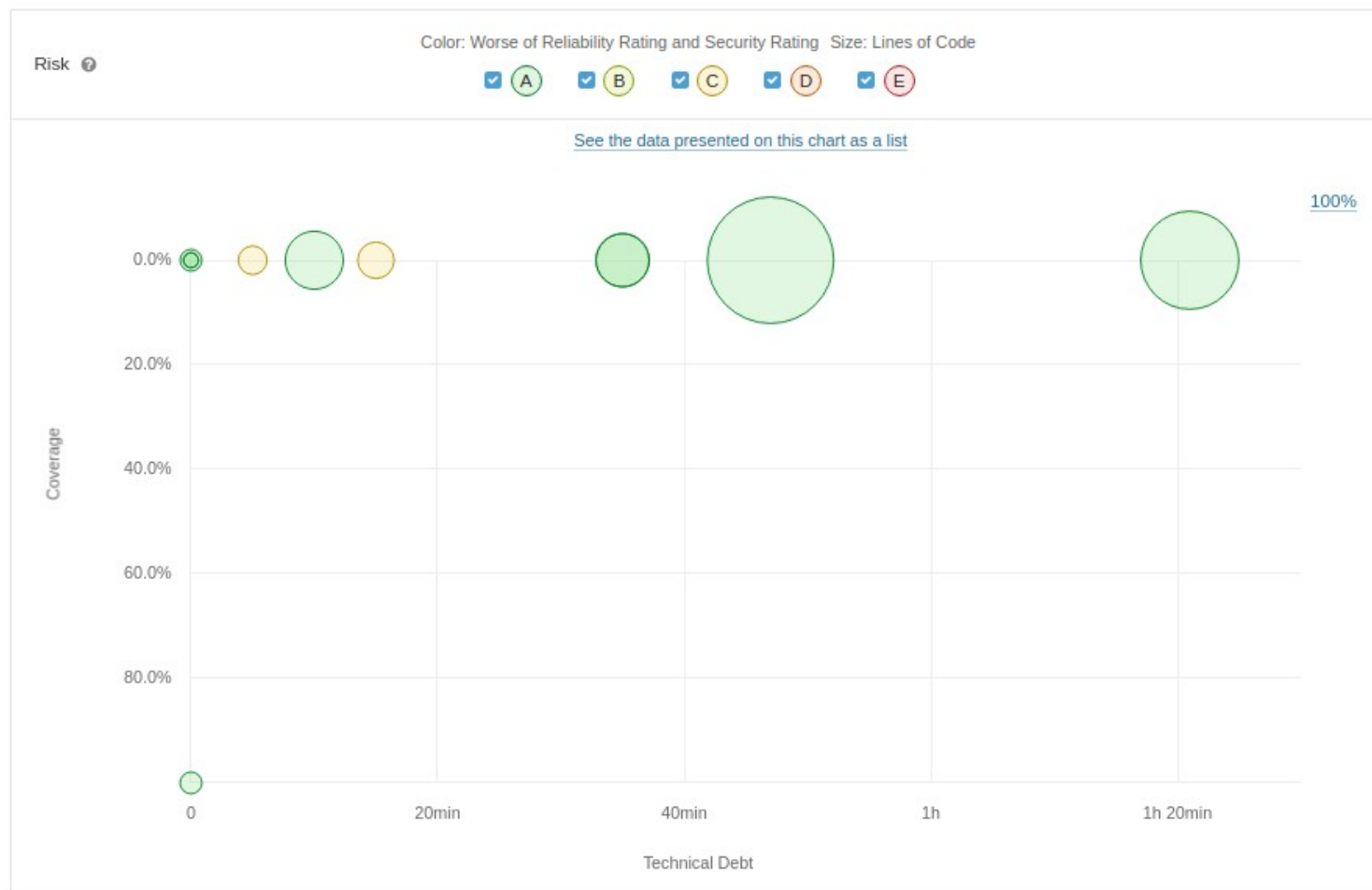
- L'ensemble du projet
- Le Nouveau Code : le code source ajouté après le point d'origine

Cela permet de mesurer la progression de la qualité dans un projet et de pouvoir « brancher » Sonar sur un projet existant sans trop de dette technique

Le point d'origine se définit :

- Par rapport à un n° de version
- Par rapport à un nombre de jours

# Vue synthétique des métriques





# Porte qualité

---

Les **portes qualité** définissent des seuils pour certains métriques.

Cela répond à la question : « Le code peut il aller en production ? »

Les seuils sont typiquement définis par rapport au nouveau code

Les portes qualité sont généralement partagés entre projet



# Condition d'une porte qualité

---

Chaque condition d'une porte qualité définit :

- La métrique concerné
- La période : New Code ou Code complet
- Un opérateur de comparaison
- Le seuil du métrique



# Sonar Way

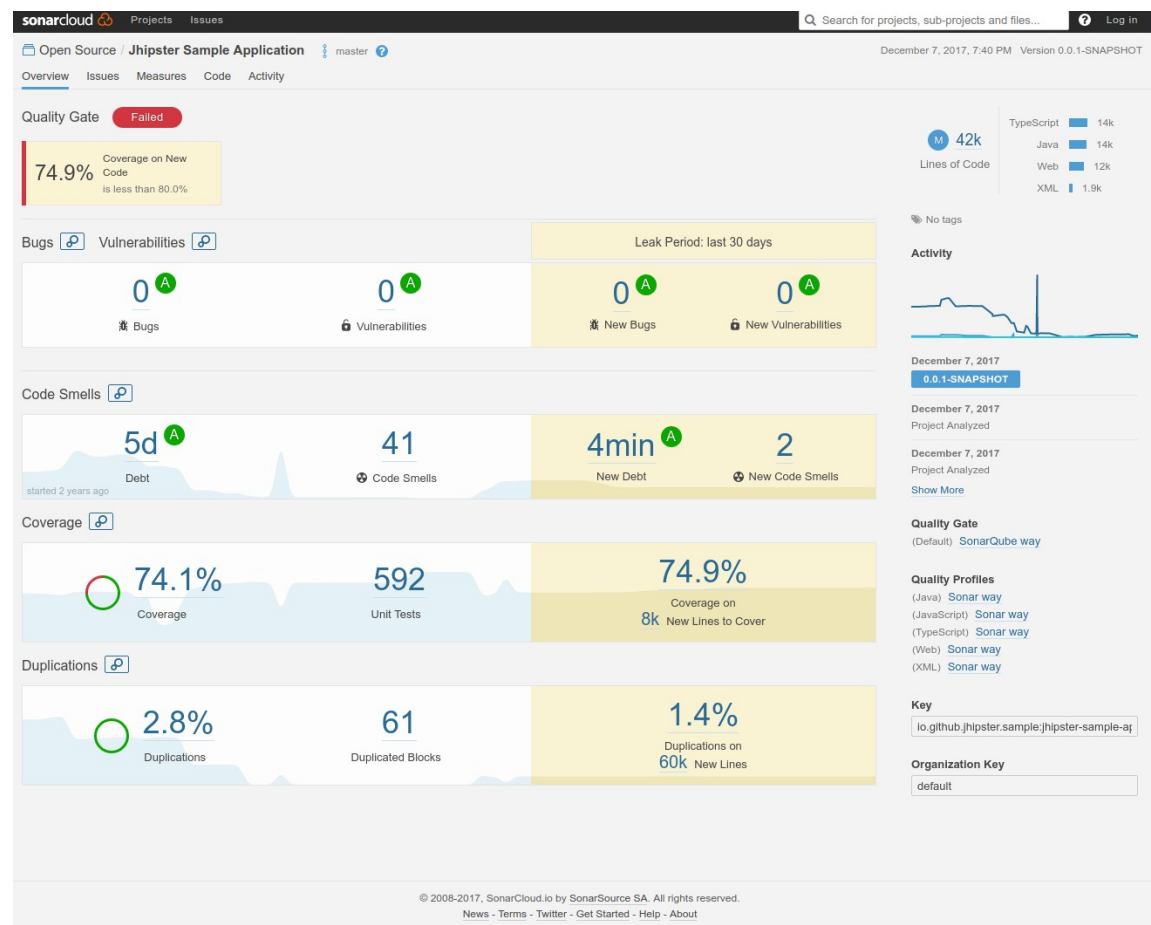
---

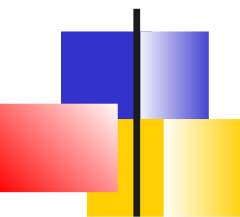
La porte qualité « *SonarQube way* » est activée par défaut.

Elle s'applique sur le nouveau code

- Pas d'issues sur le nouveau code
- Hotspot Reviewed = 100 %
- Couverture de test  $\geq 80$  %
- Duplications  $\leq 3$  %

# Sonar UI : TdB Projet





# Menus projet

Overview Issues Security Hotspots Measures Code Activity Project Settings ▾ Project Information

**Overview** : Tableau de bord, Statut porte qualité, Nombre d'issues actives

**Issues** : Liste triable, filtrable des issues

**Security Hotspots** : Liste triable, filtrable des SH

**Measures** : Tableau des risques, tous les métriques calculés

**Code** : Parcours du code par package avec infos sur les issues, couverture et duplication

**Activity** : Journal des analyses et des changements de configuration, graphique de tendance





# SonarQube

---

Architecture, éditions  
Installation  
Concepts Sonar  
**Administration**  
Workflow des issues



# Administration

---

Le menu administration propose plusieurs entrées :

- **Configuration** par défaut des projets, Intégrations aux autres système, Péremption des données, constantes pour le calcul de la dette, ...
- **Utilisateurs, groupes, permissions**
- **Gestion des projets** : Édition + Visualisation des tâches de fond
- **Surveillance** du système
- **Marketplace** et gestion des plugins



# Sécurité et Authentification

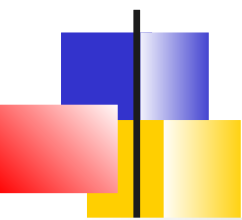
---

*SonarQube* intègre un mécanisme complet pour gérer l'authentification, l'autorisation et le cryptage des mots de passe

Il est cependant possible d'externaliser la sécurité vers :

- LDAP ou Active Directory
- SAML : Azure AD, Keycloak, Okta
- Plateforme DevOps : GitHub, Bitbucket, Gitlab

Les utilisateurs de la base sont alors soit locaux soit externes



# Configuration à l'installation

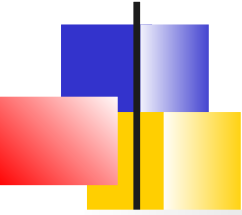
---

Par défaut, une authentification est nécessaire pour accéder aux résultats et lancer une analyse

Il y a un seul compte administrateur qui a tous les droits

3 groupes sont définis :

- ***sonar-admin*** : Tous les droits
- ***sonar-users*** : Tous les droits pour la gestion des issues
- ***anyone***: Une personne naviguant sur le site sans authentification  
Équivalent à *sonar-users*, si la sécurité n'est pas activée, aucun droit sinon



# Jeton d'accès

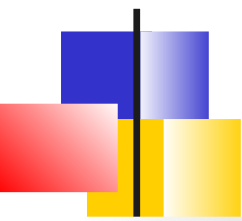
Chaque utilisateur peut générer des **jetons** qui peuvent servir à invoquer l'API Rest ou démarrer une analyse.

3 types de jetons sont possibles :

- **User** : Le jeton représente l'utilisateur et hérite de ses droits
- **Analyse Projet** : Le jeton sert à démarrer une analyse pour un projet donné.
- **Analyse Globale** : Le jeton permet de démarrer des analyses sur tous les projets

Typiquement, lors d'une analyse le jeton est fourni via la propriété

***sonar.token***



# Cas d'usage de la sécurité

---

Forcer l'authentification lors de l'accès au serveur ou autoriser l'accès anonyme

Restreindre l'accès d'un projet à un groupe d'utilisateurs

Restreindre l'accès aux sources du projet

Définir qui peut administrer un projet

Définir qui peut administrer le serveur



# Permissions globales

---

***Administration > Security > Global Permissions.***

- ***Administer System***: Administration du serveur
- ***Administer Quality Profiles and Quality Gates*** : Édition des profils et portes qualité.
- ***Execute Analysis***: Exécution d'analyse
- ***Create Projects***: Création de projet



# Permissions projet

---

Au niveau du projet :

***Project Settings -> Permissions***

La visibilité d'un projet peut être :

- Public

- Privée

Le code source et les mesures ne sont pas accessible par les utilisateurs  
*anyone*





# Permissions projet

---

Les projets publics ou privés peuvent définir les permissions suivantes :

- **Administer Issues**: Édition des issues
- **Administer Security Hotspot** : Édition des hotspot
- **Administer**: Configuration projet
- **Execute Analysis**: Exécution d'analyse

Les projets privés définissent :

- **Browse**: Accès au projet et visualisation issues et métriques
- **See Source Code**: Accès au code source



# Gabarit de permissions

---

Il est possible également de définir des gabarits pour initialiser les permissions par défaut d'un projet.

La création d'un gabarit consiste à fournir :

- un pattern pour la clé du projet
- Un ensemble de permissions



# Définition *New Code*

---

Chaque projet SonarQube possède une définition du New Code qui peut être spécifiée :

- Au niveau global  
***Administration > Configuration > General Settings > New Code***
- Au niveau projet  
***Project Settings > New Code***
- Au niveau branche pour la *Developer Edition*



# Options pour New Code

---

Options disponibles pour la définition du *New Code* :

- ◆ **Version précédente** : La version étant identifiée différemment selon les projets (*pom.xml*, *build.gradle* ou *sonar.projectVersion*)
- ◆ **Nombre de jours** : Si aucune action n'est faite sur une issue à la fin de période, l'issue repasse dans la partie Overall
- ◆ **Analyse spécifique** : Possible seulement par l'API
- ◆ **Branche de référence** : Toutes les différences avec la branche de référence fait partie du *New Code*.



# Historique projet

---

Le **Database Cleaner** est responsable de supprimer certains détails des analyses passées, il peut être configuré par l'administrateur.

Par défaut, le cleaner supprime :

- Pour chaque projet :
  - 1 seul snapshot par jour après 1 jour
  - 1 seul snapshot par semaine après 1 mois.
  - 1 seul snapshot par mois après 1 an
  - Seuls les snapshots correspondant à des versions sont conservés après 2 ans.
  - Tous les snapshots vieux de 5 ans sont supprimés
- Tous les issues fermées vieilles de 30 jours sont supprimées
- Historique au niveau d'un package est supprimé



# SonarQube

---

Architecture, éditions  
Installation  
Concepts Sonar  
Administration  
**Workflow des issues**



# Typologie des issues

---

Les issues sont classifiés selon 2 axes :

- Clean Code
  - Cohérence
  - Intentionnalité
  - Adaptabilité
  - Responsabilité
- Qualité logiciel
  - Sécurité
  - Fiabilité
  - Maintenabilité

Une issue peut être associée à plusieurs type



# Attributs Clean Code

---

**Cohérence** : Le code est écrit de manière uniforme et conventionnelle. Tout le code se ressemble et suit un modèle régulier, même avec plusieurs contributeurs à des moments différents.  
*Un code cohérent est formaté, conventionnel et identifiable.*

**Intentionnalité** : Le code est précis et utile. Chaque instruction a du sens, est correctement formée et communique clairement son comportement.  
*Le code intentionnel est clair, logique, complet et efficace.*

**Adaptabilité** : Le code est structuré pour être facile à évoluer et à développer en toute confiance. Il facilite l'extension ou la réutilisation de ses pièces et favorise des changements localisés sans effets secondaires indésirables.  
*Le code adaptable est ciblé, distinct, modulaire et testé.*

**Responsabilité** : Le code prend en compte ses obligations éthiques sur les données, ainsi que les normes sociétales.  
*Un code responsable est légal, digne de confiance et respectueux.*





# Sévérité

---

La sévérité d'une issue traduit son impact sur la qualité. Elle ne peut pas être éditée :

- **HIGH** :  
Probabilité d'impacter le comportement en production (fuite de mémoire, de connexion JDBC, ...).  
=> Doit être traité ASAP
- **MEDIUM** :  
Impacte fortement la productivité du développeur (code mort, blocs dupliqués paramètres inutilisés, ...)
- **LOW** :  
Léger impact sur la productivité. (Lignes trop longues, "switch" à la place de if, ...)



# Cycle de vie des issues

---

Une fois créé, une issue peut prendre les statuts suivants :

- **Ouvert** :
  - Nouveaux issues découverts lors d'une analyse
  - Issues déjà référencées mais non fixées
- **Accepté** : Défini manuellement pour indiquer que le problème est valide mais pas fixé pour le moment
- **Résolu** : Défini automatiquement par Sonarqube lors d'une analyse
- **Faux-positif** : Défini manuellement indiquant que l'analyse n'est pas pertinente



# Intégration SCM

---

La détection d'un SCM lors de l'analyse peut déverrouiller des fonctionnalités SonarQube :

- Affectation d'une date à l'issue  
=> Impact sur la classification New Code
- Affectation automatique des issues
- Annotation de code (données de blâme) dans le visualiseur de code

Git et SVN sont supportés par défaut



# Affectation date de l'issue

---

Dans le cas de la détection d'un SCM,  
l'issue sera antidaté :

- Lors de la première analyse du projet ou de la branche
- Lorsque la règle est ajoutée dans le profil
- Lors d'une mise à jour de SonarQube

Il se peut ainsi qu'une nouvelle issue ne soit pas ajoutée au *New Code*



# Affectation utilisateur

---

Par défaut, les issues sont affectées au dernier committer de la ligne où est détectée l'issue si on peut corréler le commiter à un utilisateur SonarQube.

- Elles peuvent cependant être réassignées manuellement à quelqu'un d'autre.
- En fonction de ses préférences, le responsable reçoit une notification *email*



# Corrélation d'utilisateur

---

Des corrélations via le login et l'e-mail sont effectuées automatiquement.

Par exemple, si l'utilisateur commette avec son adresse e-mail et que cette adresse e-mail fait partie de son profil SonarQube, les nouveaux problèmes soulevés sur les lignes où l'utilisateur a committé lui seront attribués.

Des corrélations supplémentaires peuvent être effectuées manuellement dans le profil de l'utilisateur



# Édition d'issues

---

Différentes éditions manuelles peuvent également être effectuées sur les issues :

- Commenter
- Ré-assigner
- Accepter
- Marquer comme faux positif

Les 2 dernières actions font typiquement partie d'une revue technique visant à évaluer la dette technique d'un projet.



# Projet qualité

---

## **Généralités**

Configuration de l'analyse Sonar  
Personnalisation profils et portes qualité  
Règles personnalisées





# L'anti-pattern

---

## L'anti-pattern :

- Prendre un patrimoine qui a déjà du vécu
- Installer tout un tas d'outils de mesure
- Activer « la totale » à sévérité maximale
- Produire des tableaux de chiffres
- Produire un rapport qui constate que tout le patrimoine est uniformément catastrophique
- Et maintenant ?



# Mener un projet qualité

---

## Introduire la qualimétrie<sup>1</sup>

- Déduire la charte (porte) qualité de l'existant
- Démarrer petit, sur des objectifs clairs
- Mesurer l'évolution
- Étendre petit-à-petit les périmètres de mesure
  - Outils, mesures, patrimoine examiné
- Augmenter les seuils de la porte qualité



# Objectifs de la mesure

---

Valider une fourniture externe (Offshore, sous-traitance)

- → Outil *de surveillance*

Mesurer en interne

- → plutôt outil *d'assistance*
- Doit être paramétré de façon à *aider* le développeur
  - « attention le nom de variable masque un nom d'attribut »  
« - Ah oui, merci j'avais pas vu »
  - « je refuse la livraison car il manque deux lignes de commentaires en moyenne »  
« - Alors je rajoute deux lignes vides »



# Un indicateur n'est qu'un indicateur

---

La qualimétrie n'est qu'un indicateur de la qualité

Si application trop automatique

- Risque que les développeurs améliorent les indicateurs et non la qualité

Exemple

- Comment augmenter la couverture des tests unitaires ?
- En ajoutant des invocations de méthodes de haut-niveau sans la moindre assertion  
→ intérêt nul en terme de qualité



# Charte de transition

---

Il est intéressant de travailler avec 2 chartes<sup>1</sup> qualité

- Charte stricte qui s'applique aux nouveaux projets
- Charte de transition qui s'applique aux anciens développements
  - Pour les projets en maintenance corrective
  - Elle ne signale que les erreurs sérieuses de fiabilités

Cette charte de transition est une copie de la stricte

- On maintient la sévérité des indicateurs de fiabilité
- On abaisse les indicateurs de maintenabilité



# Ancien code

---

## Utilisation de la charte de transition sur un ancien code

- On obtiendra ainsi de nombreux avertissements
- Mais on se contentera de ne régler que les problèmes sérieux
- Les avertissements serviront à estimer le coût d'amélioration de la qualité d'un existant vers la qualité stricte d'un nouveau développement

## On peut définir une politique de maintenance

- Ex : toute action de maintenance ne doit dégrader aucun indicateur
- Ainsi, on est obligé de corriger les avertissements sur une classe donnée mais on ne doit pas introduire de problèmes potentiels ni réduire la couvertures des tests unitaires
  - Ce qui impose d'écrire des tests unitaires lors des actions de maintenance, au minimum sur le code ajouté



# Projet qualité

---

Généralités

**Configuration de l'analyse Sonar**

Personnalisation profils et portes qualité

Règles personnalisées



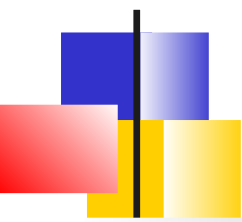
# Axes de configuration

---

La configuration d'un projet consiste à spécifier :

- Le *New Code* : Typiquement la version précédente du projet
- La clé du projet : Permet d'appliquer des gabarits de permissions
- Les portes qualité.
- Les profils qualité (i.e. les règles actives)
- Les exceptions (exclusions d'analyse, de couverture, de règles, ...)





# Hiérarchie de configuration

---

Les paramètres pour configurer l'analyse peuvent s'effectuer à plusieurs endroits :

- **Paramètres globaux** : Défini via l'interface, ils s'appliquent à tous les projets  
*Administration > Configuration > General Settings*
- **Paramètres projet** : Défini via l'interface, ils surchargent les paramètres globaux. Au niveau projet :  
*Administration > General Settings*
- **Paramètres d'un analyseur** : Défini dans un fichier configuration propre au scanner, surcharge ceux de l'UI
- **Paramètres de ligne de commande** : Ils surchargent tous les autres



# Paramètres obligatoires

---

## Serveur

***sonar.host.url*** L'URL du serveur  
(*http://localhost:9000*)

## Configuration projet

***sonar.projectKey*** : Un identifiant de projet.  
Avec Maven : `<groupId>:<artifactId>`.

***sonar.sources*** : Liste des répertoires contenant les sources séparés par des virgules.  
Avec Maven, l'emplacement Maven (*src/main/java*)



# Paramètres optionnels

---

## Identité projet

***sonar.projectName*** : Nom du projet affiché sur l'UI.

Maven : Balise `<name>`

***sonar.projectVersion*** : La version

Maven : Balise `<version>` .

## Authentification

***sonar.token*** : le jeton d'authentification d'un utilisateur avec la permission *Execute Analysis*

***sonar.login*** : Le login ou le jeton (déprécié).

***sonar.password*** : Le mot de passe si on utilise un login (déprécié)

## Service web

***sonar.ws.timeout*** : Timeout pour les appels webservices durant l'analyse



# Configuration projet

---

***sonar.projectDescription*** : Description du projet.

Pas d'équivalent Maven

***sonar.links.homepage*** : Page d'accueil du projet.

Pas d'équivalent Maven

***sonar.links.ci*** : Lien vers serveur d'intégration.

Pas d'équivalent Maven

***sonar.tests*** : Liste des répertoires contenant les classes de tests.

***sonar.language*** : Le langage du code source à analyser.

Si non spécifié, une analyse multi-langage est effectuée

***sonar.projectBaseDir*** : Si la racine du projet ne correspond au répertoire de démarrage de l'analyse

***sonar.scm.provider*** : Permet d'indiquer explicitement le plugin SCM à utiliser. En général, détection automatique



# Exclusions / Inclusions

---

***sonar.inclusions*** : Liste des gabarits de chemins à inclure dans l'analyse.

***sonar.exclusions*** : Liste des gabarits de chemins à exclure de l'analyse.

***sonar.coverage.exclusions*** : Liste des gabarits de chemins à exclure du calcul de la couverture de code

***sonar.test.exclusions*** : Liste des gabarits de chemins des fichiers de test à exclure de l'analyse.

***sonar.test.inclusions*** : Liste des gabarits de chemins des fichiers de test à inclure dans l'analyse.

***sonar.issue.ignore.allfile*** : Les fichiers correspondant à l'expression régulière sont ignorés par l'analyse.

***sonar.cpd.exclusions*** : Liste de gabarits de chemins à exclure de la détection de duplication

***sonar.cpd.\${language}.minimumtokens*** : Seuil de détection de duplication en mots. Par défaut : 100

***sonar.cpd.\${language}.minimumLines*** : Seuil de détection de duplication en lignes. Par défaut : 10



# Gabarits

---

Les gabarits présents dans les exclusions/inclusions sont relatifs à la racine du projet.

Ils supportent les caractères suivants :

- \* zéro ou plusieurs caractères
- \*\* zéro ou plusieurs répertoires
- ? un seul caractère

Exemple :

```
sonar.exclusions=**/*Bean.java,**/*DT0.java
```



# Couverture des tests

---

Pour calculer la couverture des tests, SonarQube utilise les rapports d'outils tiers comme *jacoco*

Si l'on sort des configurations par défaut :

- Le plugin peut être précisé via la propriété :

**`sonar.java.codeCoveragePlugin`**

- L'emplacement des rapports générés par

**`sonar.coverage.jacoco.xmlReportPaths`**



# Traces

---

***sonar.log.level*** : Niveau de trace. Par défaut INFO. Plus verbeux DEBUG et TRACE

***sonar.verbose*** : Idem que DEBUG avec en plus les variables d'environnement du client

***sonar.showProfiling*** : Information de profiling. Génère également un fichier *profiler.xml*

***sonar.scanner.dumpToFile*** : Indique un fichier où sont écrites toutes les propriétés passées au scanner





# Projet qualité

---

Généralités  
Configuration de l'analyse Sonar  
**Personnalisation profils et portes  
qualité**  
Règles personnalisées



# Profils qualité

---

Pour chaque technologie,

- Il existe un profil par défaut qui est appliqué si aucun profil n'est indiqué pour le projet
- Le profil *Sonar Way* est non-éritable

Les profils « *Sonar Way* » étant non éditables, ils ne peuvent pas être personnalisés

=> Il est donc recommandé de créer un nouveau profil en partant de *Sonar Way*

- Soit en copiant
- Soit en héritant



# Copie d'un profil

---

Lors de la copie d'un profil, on récupère l'ensemble de ses règles que l'on peut alors activer ou désactiver selon ses besoins.

Après la copie, le nouveau profil n'hérite pas des changements effectués dans le profil source.



# Héritage des profils

---

Les profils peuvent avoir des relations d'héritage en définissant un profil parent.

- Il est alors possible d'activer des règles qui sont désactivées dans le profil parent
- Par défaut, il n'est pas possible de désactiver des règles activées dans le profil parent.
- Les règles héritées prenant des paramètres de configuration peuvent être surchargées en modifiant leurs paramètres
- Les mises à jour du profil parent sont répercutées sur les enfants.
  - Il peut être intéressant de définir des profils racines pour chaque technologie de l'entreprise
  - Hériter du profil *SonarWay* permet de profiter des mises à jour des règles lors d'un *upgrade*.



# Mise à jour

A chaque mise à jour SonarQube, de nouvelles règles sont ajoutées, d'autres sont dépréciées.

Elles apparaissent automatiquement dans les profils *SonarWay* ou dans les profils qui en héritent

Il est possible d'identifier les nouvelles règles

- En comparant votre profil avec le profil *SonarWay*  
***Page Quality Profiles > Selection du profil 1 > puis Actions > Compare et sélectionner le profil 2***
- En utilisant le filtre ***Available since*** dans la liste des règles
- En visualisant, dans la page profil qualité, la section ***Recently Added Rules***

Il est également possible d'isoler les règles dépréciées et de voir les profils qui les utilisent via le filtre ***Deprecated Rules***



# Exclusions de règles

---

Dans le cadre d'un profil personnalisé, les règles peuvent être activées/désactivées.

De plus, via la configuration, SonarQube permet de finement spécifier pour les fichiers/répertoires du projet :

- Si l'analyse est activée ou pas
- Si certaines règles sont exclues pour ces répertoire/fichiers
- Si la détection de duplications est activée ou pas
- Si le calcul de la couverture de test est activée ou pas



# Définir les suffixes par langage

---

*Sonarqube* permet de définir les suffixes des sources en fonction du langage ainsi que les emplacements des rapports de test ou couverture

- Globalement :  
***Administration > General Settings > [Language]***
- Par projet :  
***Projet > General Settings > [Language]***
- Par les propriétés de configuration  
Par exemple pour Java :  
***sonar.java.file.suffixes***  
***sonar.junit.reportPaths***




# Ignorer des fichiers

---

3 façons d'ignorer complètement des fichiers :

- Utiliser ***sonar.sources*** pour limiter les fichiers analysés
- ***source.exclusion*** et ***source.test.exclusion*** pour exclure des fichiers ou des classes de test
- ***source.inclusion*** et ***source.test.inclusion*** pour inclure des fichiers ou des classes de test





# Ignorer les duplications ou de la couverture de test

---

Pour ignorer des fichiers de la détection de duplication :

***Administration/Projet > General Settings > Analysis Scope > Duplications***

Propriété : ***sonar.cpd.exclusions***

De la couverture de test :

***Administration/Projet > General Settings > Analysis Scope > Code Coverage***

Propriété : ***sonar.coverage.exclusions***



# Ignorer des issues

---

Il est possible d'ignorer toutes ou certaines issues en fonction du contenu des fichiers

- Ignorer des fichiers dont le contenu contient une expression régulière
- Ignorer des blocs de texte spécifiés avec des marqueurs de début et de fin respectant une expression régulière

Via l'UI, globalement ou par projet

*Administration/Projet > General Settings > Analysis Scope > Issues*

Via les propriétés de configuration

*sonar.issue.ignore.allfile*  
*sonar.issue.ignore.block*



# Ignorer une règle pour certains fichiers

Ce menu permet de désigner la règle à exclure et les fichiers concernés.

*Administration/Projet > General Settings > Analysis Scope > Issues*

Par ex : *Ignorer la règle "cpp:Union" sur tous les fichiers du répertoire object*

KEY = cpp:Union

PATH = object/\*\*/\*

Cela est également possible via les propriétés de l'analyseur :

Ex Maven :

```
<sonar.issue.ignore.multicriteria>toto</sonar.issue.ignore.multicriteria>
<sonar.issue.ignore.multicriteria.toto.resourceKey>object/**/*</
  sonar.issue.ignore.multicriteria.toto.resourceKey>
<sonar.issue.ignore.multicriteria.toto.ruleKey>cpp:Union</
  sonar.issue.ignore.multicriteria.toto.ruleKey>
```



# Journal projet

---

Lorsque SonarQube détecte qu'une analyse a été démarré avec un profil différent, un événement **Quality Profile** est ajouté au journal d'événements du projet.

**Quality Profiles > [ Profile Name ] >Changelog.**

Les utilisateurs avec des privilèges d'administration de profil qualité sont notifiés chaque fois qu'un profil prédéfini est mis à jour.



# Porte qualité

---

La porte qualité est le meilleur moyen pour améliorer la qualité dans un organisation.

Elle répond à la question : Est-ce que cette release peut aller en production ?

Elle est constituée d'un ensemble de conditions booléennes basées sur des seuils de mesure

Les portes qualité doivent être adaptées en fonction des projets.



# Sonar Way

---

Sonar fournit une porte qualité par défaut en mode ***read-only***

Elle s'adresse à stopper l'hémorragie qualité en se basant sur le « *New code* »

Il faut l'adapter par rapport à sa situation et s'en servir comme objectif à moyen terme



# Condition

---

Chaque condition d'une porte qualité est une combinaison :

- D'une mesure
- D'une période : Value (Date) or Leak  
(Valeur différentielle sur la *Leak period*)
- D'un opérateur de comparaison
- D'une valeur d'avertissement
- D'une valeur d'erreur



# Projet qualité

---

Généralités  
Configuration de l'analyse Sonar  
Personnalisation profils et portes qualité  
**Règles personnalisées**





# Création de règles

---

La création de règles personnalisées peut se faire de différentes façons :

- Utiliser les gabarits de règles
- Écrire un plugin Java utilisant l'API de SonarQube
- Ajouter des règles *XPath* en utilisant l'interface Sonar
- Importer des rapports spécifiques à un outil tiers



# Gabarits

---

Créer une nouvelle règle via un gabarit  
peut se faire par l'interface de  
SonarQube

*Rules → Templates Only*

*Create Rule*

Un formulaire dédié est proposé pour  
compléter le gabarit



# Compatibilité

---

Les alternatives de codage de règle personnalisée sont très dépendantes du langage et du plugin associé.

Pour les plugins fournis de SonarSource :

- L'API Java est disponible pour :  
COBOL, Java, Javascript, PHP, RPG
- Le code Xpath est disponible pour :  
Flex, PL/SQL, PL/I, XML
- Outils tiers :  
PMD, StyleLint, Checkstyle, ESLint, TSLint, ...



# Étapes pour la Java API

---

6 étapes pour mettre en place une nouvelle règle :

- Créer un plugin SonarQube
- Indiquer une dépendance sur le plugin du langage
- Créer les règles voulues
- Générer le plugin SonarQube (fichier jar )
- Placer le fichier jar dans le répertoire  
SONARQUBE\_HOME/extensions/plugins
- Redémarrer le serveur



# Création de projet

---

Sonar fournit des exemples de projet de création de règles pour chaque langage :

<https://github.com/SonarSource/sonar-custom-rules-examples/tree/master/>

Pour Java, un gabarit de projet Maven est disponible



# *pom.xml*

---

En dehors de toutes les dépendances nécessaires, le *pom.xml* définit des propriétés de configuration du plugin

```
<plugin>
  <groupId>org.sonarsource.sonar-packaging-maven-plugin</groupId>
  <artifactId>sonar-packaging-maven-plugin</artifactId>
  <version>1.17</version>
  <extensions>true</extensions>
  <configuration>
    <pluginKey>java-custom</pluginKey>
    <pluginName>Java Custom Rules</pluginName>
    <pluginClass>org.sonar.samples.java.MyJavaRulesPlugin</pluginClass>
    <sonarLintSupported>true</sonarLintSupported>
    <sonarQubeMinVersion>5.6</sonarQubeMinVersion>
  </configuration>
</plugin>
```



# Développement de la règle

---

Au minimum 3 fichiers sont nécessaires lors de la création d'une règle :

- Un fichier de test contenant le code qui servira de données d'entrée pour tester la règle
- Une classe de test contenant les tests unitaires
- Une classe de règle contenant l'implémentation de la règle



# Exemple java

---

Spécification de la règle :

*“Pour une méthode avec un seul paramètre, les types de sa valeur de retour et de son paramètre ne doivent jamais être identique.”*





# Fichier de test

---

```
class MyClass {  
    MyClass(MyClass mc) { }  
  
    int    foo1() { return 0; }  
    void   foo2(int value) { }  
    int    foo3(int value) { return 0; } // Noncompliant  
    Object foo4(int value) { return null; }  
    MyClass foo5(MyClass value) {return null; } // Noncompliant  
  
    int    foo6(int value, String name) { return 0; }  
    int    foo7(int ... values) { return 0;}  
}
```



# Classe de test

---

```
import org.junit.Test;
import org.sonar.java.checks.verifier.JavaCheckVerifier;

@Test
public void test() {
    JavaCheckVerifier.verify("src/test/files/MyFirstCustomCheck.java",
        new MyFirstCustomCheck());
}
```



# Analyseur syntaxique

---

L'analyseur de SonarQube parse un code Java et produit une structure de donnée : l'**arbre de syntaxe**

Chaque construction du langage est représenté avec un arbre de syntaxe **spécifique** et est associé à une **interface** Java qui décrit toutes ses particularités.

Par exemple, une méthode est associé :

- Au type d'arbre  
`org.sonar.plugins.java.api.tree.Tree.Kind.METHOD`
- Et à l'interface  
`org.sonar.plugins.java.api.tree.MethodTree`



# Visiteur

---

Le design pattern utilisé lors de l'écriture d'une classe de test est le pattern **Visiteur** qui permet de séparer l'algorithme de la logique de parcours de l'arbre syntaxique.

SonarQube appelle lorsque nécessaire des méthodes de type *visitMethod*, *visitAnnotation*, etc. que la règle implémente

Pour limiter l'appel de ces méthodes, la classe de règle peut étendre des types de *SubscriptionVisitor* qui déclare en quoi ils sont intéressé.

- Par exemple : *IssuableSubscriptionVisitor*, *ComplexityVisitor*, ...



# Exemple

```
@Rule(key = "MyFirstCustomCheck",
      name = "Return type and parameter of a method should not be the same",
      description = "For a method having a single parameter, the types of its return value and its
parameter should never be the same.",
      priority = Priority.CRITICAL, tags = {"bug"})
public class MyFirstCustomCheck extends IssuableSubscriptionVisitor {

    @Override
    public List<Kind> nodesToVisit() { // Quels nœuds on veut traiter
        return ImmutableList.of(Kind.METHOD);
    }

    @Override
    public void visitNode(Tree tree) { // Méthode effectuant la vérification
        MethodTree method = (MethodTree) tree;
        if (method.parameters().size() == 1) {
            MethodSymbol symbol = method.symbol();
            Type firstParameterType = symbol.parameterTypes().get(0);
            Type returnType = symbol.returnType().type();
            if (returnType.is(firstParameterType.fullyQualifiedName())) {
                reportIssue(method.simpleName(), "Never do that!");
            }
        }
    }
}
```



# XPath

---

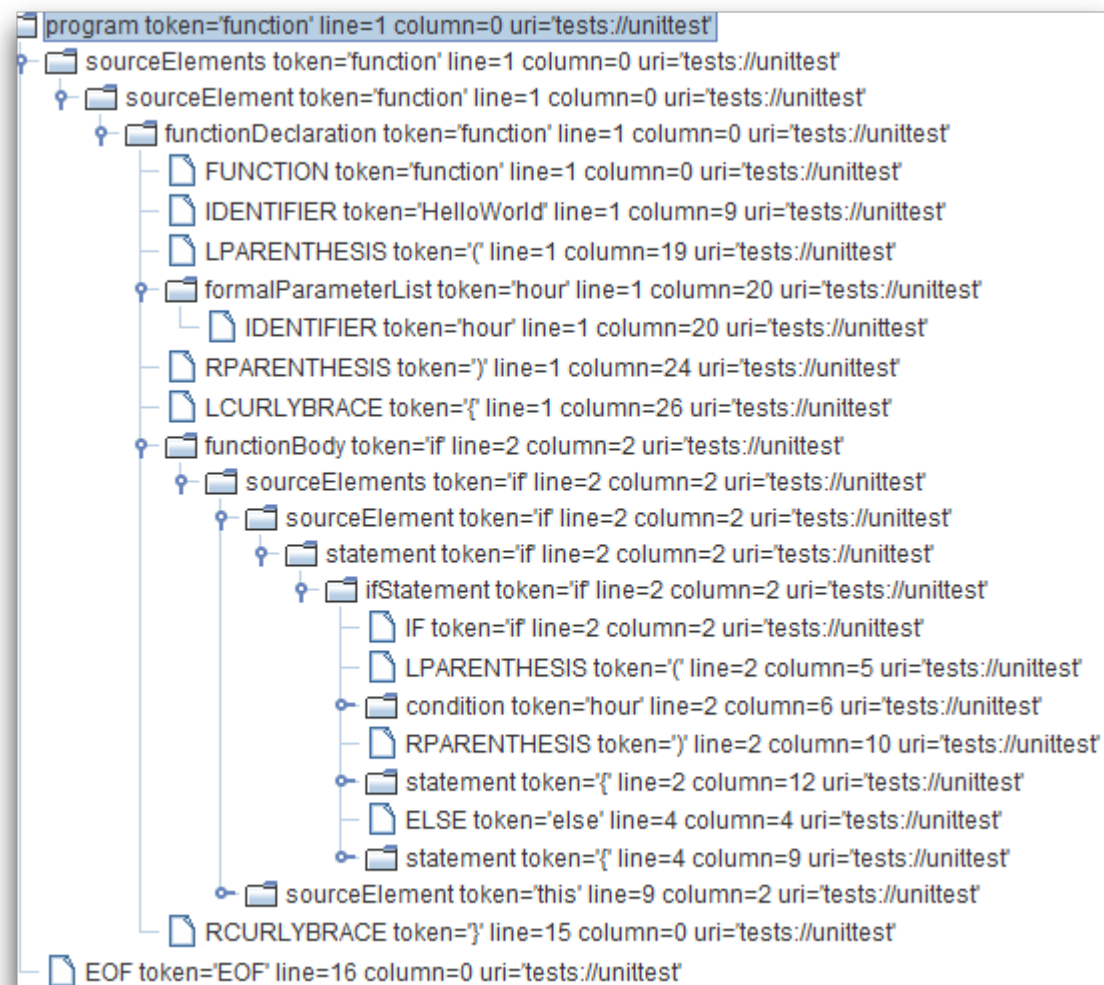
Coder de nouvelles règles en *XPath* est directement possible avec le langage XML

Avec les autres langages supportés, il est nécessaire d'utiliser le SSLR Toolkit propre à son langage.

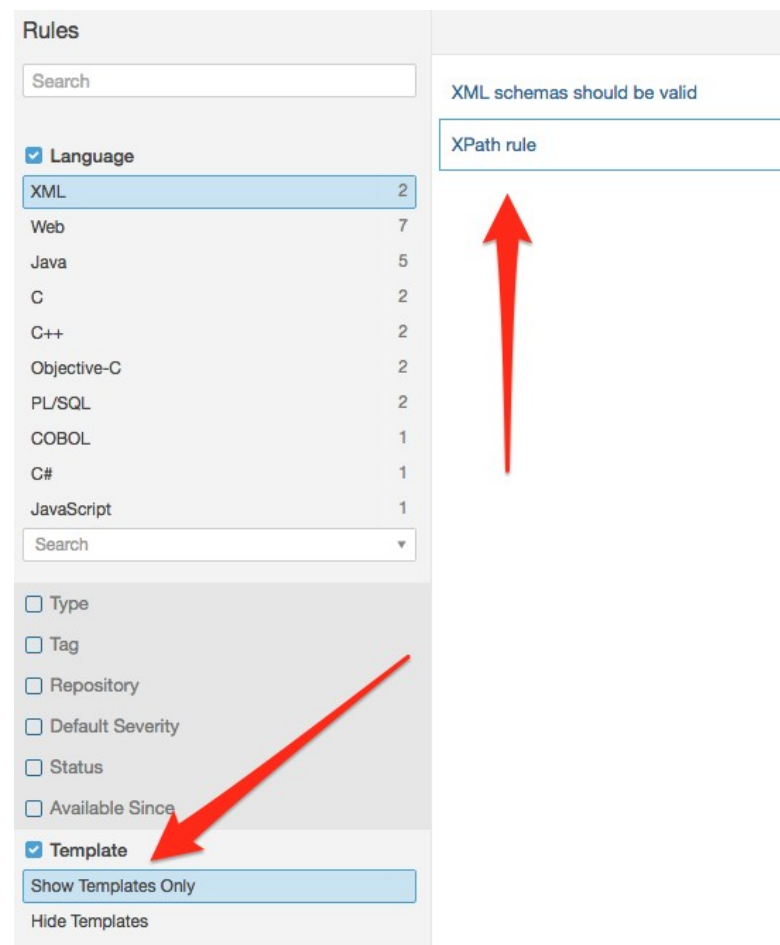
Le toolkit permet de voir l'arbre syntaxique (Abstract Syntax Tree) du code analysé.

Les expressions XPath sont alors écrites vis à vis de l'arbre.

# Exemple Arbre syntaxique



# Ajout de la règle



Rules

Search

☒ Language

XML	2
Web	7
Java	5
C	2
C++	2
Objective-C	2
PL/SQL	2
COBOL	1
C#	1
JavaScript	1

Search

☐ Type

☐ Tag

☐ Repository

☐ Default Severity

☐ Status

☐ Available Since

☒ Template

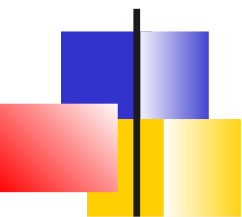
Show Templates Only

Hide Templates

XML schemas should be valid

XPath rule

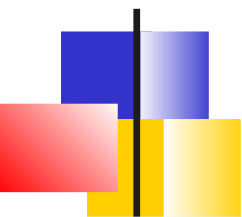




# SonarLint

---

## **SonarLint**



# SonarLint

**SonarLint** s'intègre dans différents IDEs : Eclipse, IntelliJ, VisualStudio, VSCode, Atom

<https://www.sonarlint.org/>

Il peut être configurée pour récupérer la configuration d'un projet SonarQube.

Ceci permet à tout développeur de savoir à l'avance quel sera l'effet de ses modifications et si elles seront éventuellement rejetées par les outils d'analyse de code



# Fonctionnalités

---

Analyse temps-réel : Les issues sont indiquées automatiquement, ce mode peut être désactivé.

Mode connecté : *SonarLint* utilise les analyseurs de code, les profils qualité et les configurations définies sur le serveur ou *SonarCloud*

Analyseurs : *SonarLint* supporte les analyseurs *SonarSource* (*SonarJava*, *SonarJS*, ...) ainsi que les règles personnalisées qui étendent les analyseurs. Il ne supporte pas les analyseurs tiers.

Configuration de règle : Si le projet n'est pas connecté, *SonarLint* utilise les règles des profils qualité par défaut.

Exclusion de fichiers et d'issues : Soit dans l'IDE soit sur le serveur



# Administration et intégration

---

## **Intégration SCM**

WebApi et Webhooks

Intégration Jenkins

Intégration DevOps

Monitoring



# Introduction

---

L'intégration de svn et git se fait automatiquement.

Cela apporte :

- L'affectation automatique des issues
- L'annotation du code (avec les données de blâme) dans le visualisateur de code
- La détection du nouveau code est pilotée par SCM (Sans SCM, SonarQube détermine le nouveau code à l'aide des dates d'analyse).



# Administration et intégration

---

Intégration SCM  
**WebApi et Webhooks**  
Intégration Jenkins  
Intégration DevOps  
Monitoring



# Introduction API

---

L'API de Sonar est documentée à l'URL ***/web\_api***

Pour accéder à l'API, il faut être authentifié :

```
curl -u TOKEN: https://<server>/api/user_tokens/search
```



# Principales fonctionnalités

---

***api/ce*** : Retrouver des informations sur les tâches du moteur

***api/projects, api/project\_analyses*** : Projets et leur historique

***api/duplication, api/issues, api/hotspots*** : Résultats des analyses

***api/rules, api/qualitygates, api/qualityprofile*** : Règles, profils et portes qualité

***api/sources, api/test*** : Information sur les sources et les classes de test

***api/system*** : Monitoring, arrêt, démarrage

***api/users, api/groups, api/user\_tokens, api/permissions*** : Utilisateurs, groupes et permissions





# Introduction WebHooks

---

Les webhooks permettent d'effectuer des requêtes HTTP POST vers des URLs spécifiées lorsque certains types d'évènements surviennent.

Typiquement, une notification vers d'autres outils lorsque le statut de la porte qualité est disponible.

En général, cela survient durant ou après une analyse mais cela peut également survenir lors de l'édition d'une issue qui ferait changer le statut de la porte qualité



# Requête HTTP

---

La requête HTTP est effectuée :

- Quelque soit le statut de la tache de fond d'analyse
- Utilise une méthode POST
- A un content type "*application/json*", avec un encoding UTF-8
- Inclut des données sous forme de document JSON

La réponse à la requête doit intervenir sous 10 secondes.



# Example JSON

---

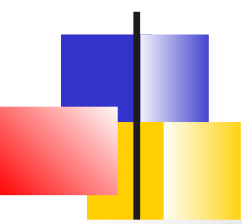
```
{
  "analysedAt": "2016-11-18T10:46:28+0100",
  "project": {
    "key": "org.sonarqube:example",
    "name": "Example"
  },
  "properties": {
  },
  "qualityGate": {
    "conditions": [
      {
        "errorThreshold": "1",
        "metric": "new_security_rating",
        "onLeakPeriod": true,
        "operator": "GREATER_THAN",
        "status": "OK",
        "value": "1"
      },
      {
        "errorThreshold": "1",
        "metric": "new_reliability_rating",
        "onLeakPeriod": true,
        "operator": "GREATER_THAN",
        "status": "OK",
        "value": "1"
      }
    ],
    ...
  }
}
```



# Example JSON (2)

---

```
{
  {
    "errorThreshold": "1",
    "metric": "new_maintainability_rating",
    "onLeakPeriod": true,
    "operator": "GREATER_THAN",
    "status": "OK",
    "value": "1"
  },
  {
    "errorThreshold": "80",
    "metric": "new_coverage",
    "onLeakPeriod": true,
    "operator": "LESS_THAN",
    "status": "NO_VALUE"
  }
],
"name": "SonarQube way",
"status": "OK"
},
"serverUrl": "http://localhost:9000",
"status": "SUCCESS",
"taskId": "AVh21JS2JepAEhwQ-b3u"
}
```



# Administration et intégration

---

Intégration SCM  
WebApi et Webhooks  
**Intégration CI**  
Intégration DevOps  
Monitoring



# Introduction

---

L'intégration avec la plateforme de CI/CD consiste à faire échouer la pipeline lorsque la porte qualité n'est pas passée

Sonarqube offre du support pour :

- Jenkins
- Github
- Bitbucket



# Intégration Jenkins

---

Le plugin ***SonarQube Scanner*** permet de centraliser la configuration de SonarQube dans Jenkins

L'analyse d'un projet peut alors être définie comme étape d'un build

Une fois l'analyse terminée, un statut de qualité est remonté sur l'UI Jenkins et un lien permet d'accéder aux tableaux de bord Sonar



# Configuration

---

La mise en place consiste à :

- Définir l' ou les instances de SonarQube

*Manage Jenkins → Configure System → SonarQube configuration → Add SonarQube*

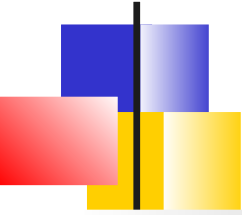
- Définir les scanners à utiliser (Exemple Maven)

*Manage Jenkins → Configure Tools → SonarQube scanner*

- Configurer un projet pour utiliser le scanner



# Job Legacy



Lancer une analyse avec SonarQube Scanner

Tâche à lancer

?

JDK

(Hérite du job)

▼

?

Le JDK à utiliser pour cette analyse SonarQube

Chemin vers les propriétés du projet

sonar-weather.properties

?

Propriétés de l'analyse

?

Additional arguments

▼

?

Options de la JVM

▼

?



# Propriété du scanner

---

Avec les jobs legacy, il n'est pas aisé de faire échouer la pipeline si la porte qualité n'est pas passée.

Comme contournement, on peut positionner la variable de configuration

***sonar.qualitygate.wait=true***

Dans ce cas, le scanner attend le retour du serveur avant de finaliser son processus.

L'étape d'analyse échouera si la porte qualité n'est pas passé et même si l'analyse a réussi



# Pipeline plugin

---

Le plugin ***SonarQubeScanner*** propose  
2 steps :

- ***withSonarQubeEnv*** : Prépare l'environnement de l'agent pour l'exécution de Sonar
- ***waitForQualityGate*** : Pause la pipeline en attendant que l'analyse soit terminée, et retourne le statut de la porte qualité



# Example

---

```
node {  
  stage('SCM') {  
    git 'https://github.com/foo/bar.git'  
  }  
  stage('SonarQube analysis') {  
    // requires SonarQube Scanner 2.8+  
    def scannerHome = tool 'SonarQube Scanner 2.8';  
    withSonarQubeEnv('My SonarQube Server') {  
      sh "${scannerHome}/bin/sonar-scanner"  
    }  
  }  
}
```



# Exemple Maven + timeout

---

```
stage('SonarQube analysis') {  
  agent any  
  steps {  
    withSonarQubeEnv('My SonarQube Server') {  
      sh 'mvn clean package sonar:sonar'  
    }  
  }  
}
```

*// Pas besoin d'occuper un agent*

```
stage("Quality Gate"){  
  agent none  
  steps {  
    script {  
      timeout(time: 1, unit: 'HOURS') {  
        def qq = waitForQualityGate() // Réutilisation de taskId setté par withSonarQubeEnv  
        if (qq.status != 'OK') {  
          error "Pipeline aborted due to quality gate failure: ${qq.status}"  
        }  
      }  
    }  
  }  
}
```



# Utilisation d'un WebHook Secret

---

Pour renforcer la sécurité des communications entre Jenkins et Sonar, il est possible de configurer un **WebHook secret**

Jenkins :

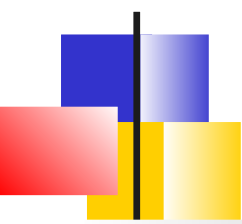
**Manage Jenkins > Configure System > SonarQube Server > Advanced > Webhook Secret**

Déclarative Pipeline :

**`waitForQualityGate(webhookSecretId:  
'yourSecretID')`**

Sonarqube :

**Project Settings → WebHook**



# Administration et intégration

---

Intégration SCM  
*WebApi* et *Webhooks*  
Intégration CI  
**Intégration DevOps**  
Monitoring



# Introduction

---

SonarQube propose des intégrations avec certaines plateformes DevOps :

- GitHub
- Bitbucket
- Gitlab
- Azure DevOps

Cependant, les intégrations avancées nécessitent des éditions Developer ou +





# Exemple Gitlab

---

## Apports :

- Import facile des dépôts Gitlab
- Intégration avec les Gitlab CI (pipeline) détectant les branches et pull-request pour la Developer Edition
- Report du status des portes qualité dans les Merge request
- Authentification avec Gitlab



# Mise en place

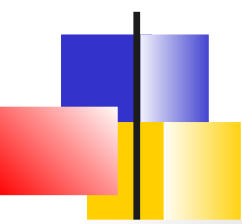
***Administration > Configuration > General Settings > DevOps Platform Integrations***

Créer une configuration en indiquant :

- Un nom
- Le root uri de l'API GitLab
- Un jeton Gitlab

Créer ensuite un projet dans SonarQube via un import GitLab en fournissant un jeton GitLab

Indiquer la méthode d'analyse



# Administration et intégration

---

Intégration SCM  
WebApi et Webhooks  
Intégration Jenkins  
Intégration DevOps  
**Monitoring**



# Introduction

---

Le premier point de surveillance est l'API Web :

***<http://<server>/api/system/health>***

Elle donne un premier diagnostic du serveur

Les autres points à surveiller sont :

- La mémoire du processus Java
- Les Beans JMX du processus Java
- L'activité Elasticsearch



# Processus java

---

Une instance de SonarQube consiste en l'exécution de 3 process Java :

- Le serveur Web
- Le moteur d'analyse
- ElasticSearch

Leur configuration mémoire peut s'effectuer dans conf/sonar.properties :

- `sonar.web.javaOpts`
- `sonar.ce.javaOpts`
- `sonar.search.javaOpts`



# Mbeans JMX

---

En plus des Mbeans standards, SonarQube ajoute des Mbeans spécifiques à chaque composant de l'architecture :

- ComputeEngine
- Database
- ElasticSearch
- SonarQube

Tous ces Mbeans exposent des attributs read-only, il n'est pas possible de modifier ou réinitialiser en cours d'exécution



# Attributs de ComputeEngine MBean

---

***ProcessingTime*** : Temps passé en ms pour exécuter les tâches de fond (analyse ou autre)

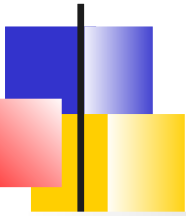
***ErrorCount*** : Nombre de tâche de fond ayant échoué depuis le dernier redémarrage

***PendingCount*** : Nombre de tâches de fond en attente

***InProgressCount*** : Nombre de tâches de fond en exécution

***SuccessCount*** : Nombre de tâches de fond ayant réussi depuis le dernier redémarrage

***WorkerCount*** Number : Nombre de worker simultanés



# Attributs du Mbean Database

Ce Mbean est présent pour les processus ComputeEngine et WebServer

***MigrationStatus*** : Soit UP\_TO\_DATE, REQUIRES\_UPGRADE, REQUIRES\_DOWNGRADE, FRESH\_INSTALL

***PoolActiveConnections*** : Nombre de connexions actives dans le pool

***PoolIdleConnections*** : Nombre de connexions en attente dans le pool

***PoolInitialSize*** : Taille initiale du pool

***PoolMaxActiveConnections*** : Taille max du pool

***PoolMaxIdleConnections*** : Taille max de connexions idle

***PoolMaxWaitMillis*** : Valeur d'attente max dans le pool

***PoolRemoveAbandoned*** : true, false

***PoolRemoveAbandonedTimeoutSeconds*** : En secondes





# Attributs du Mbean ElasticSearch

---

***NumberOfNodes*** : Nombre de nœuds  
Esearch dans SonarQube.

***State*** : Statut du cluster : GREEN,  
YELLOW, RED

ElasticSearch propose également une  
API de monitoring très complète. Point  
d'entrée :

*[http://<server>/\\_cluster/health](http://<server>/_cluster/health)*



# Attributs du MBean de SonarQube

---

***LogLevel*** : Niveau de trace : INFO, DEBUG, TRACE

***ServerId*** : ID du serveur

***Version*** : Version de SonarQube



# Annexes

---

## **Square**

### Métriques Sonar

### Dépréciation typologie des issues



# Définitions (1)

---

***Functionality (Fonctionnalité)*** : Capacité à délivrer les fonctionnalités attendues

***Reliability (Fiabilité)*** : Capacité à maintenir le niveau de performance attendu dans les conditions d'utilisation spécifiées

***Usability (Utilisabilité)*** : Capacité à être compris, appris, utilisé, convivial pour l'utilisateur dans les conditions d'utilisation spécifiées



# Définitions (2)

---

***Efficiency (Efficacité)*** : Capacité à fournir les performances appropriées selon les ressources utilisées dans les conditions spécifiées

***Maintainability (Maintenabilité)*** : Capacité à être modifié (corrigé, amélioré, adapté) selon les changements  
(environnement/spécification)

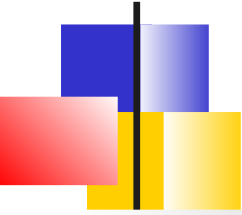
***Portability (Portabilité)*** : Capacité à être transféré d'un environnement à un autre



# Sécurité

**Security (Sécurité)** fait également partie de la norme qualité. Elle définit les sous-caractéristiques suivantes :

- Confidentialité : Données accessibles par les seules personnes autorisées.
- Intégrité : Protection contre la modification de données
- Non-répudiation : Actions prouvées
- Traçabilité : Action pouvant être tracées
- Authenticité : Identité prouvée



# Functionality (Fonctionnalité)

---

Capacité à délivrer les fonctionnalités attendues

- Suitability (Pertinence)
  - Capacité à délivrer les fonctions appropriées aux tâches/objectifs de l'utilisateur
- Accuracy (Exactitude)
  - Capacité à fournir les résultats attendus avec la précision attendue
- Interopability (Interopérabilité)
  - Capacité à interagir avec les systèmes spécifiés
- Security (Sécurité)
  - Capacité à protéger les données contre les personnes/systèmes non autorisées
- Functionality compliance (Conformité fonctionnelle)
  - Capacité à respecter les standards/conventions/lois relatives aux fonctionnalités



# Reliability (Fiabilité)

---

Capacité à maintenir le niveau de performance attendu dans les conditions d'utilisation spécifiées

- Maturity (Maturité)
  - Capacité à éviter les plantages en réponses aux erreurs internes
- Fault tolerance (Tolérance aux pannes)
  - Capacité à maintenir le niveau de performance attendue en cas de défauts ou de mauvaise utilisation de ses interfaces
- Recoverability (Capacité de récupération)
  - Capacité à rétablir le niveau de performance attendu et à restaurer les données défectueuses après un plantage
- Availability (Disponibilité)
  - Capacité à être prêt à délivrer les fonctionnalités attendues
- Reliability compliance (Conformité de fiabilité)
  - Capacité à respecter les standards/conventions/lois relatives à la fiabilité



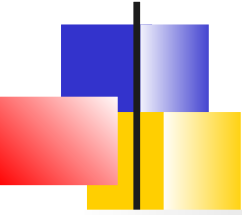


# Usability (Utilisabilité)

---

Capacité à être compris, appris, utilisé, convivial pour l'utilisateur dans les conditions d'utilisation spécifiées

- Understandability (Intelligibilité)
  - Capacité à être compris par l'utilisateur pour réaliser une tâche particulière
- Learnability (Simplicité d'apprentissage)
  - Capacité à permettre à l'utilisateur d'apprendre son utilisation
- Operability (Opérabilité)
  - Capacité à permettre à l'utilisateur de faire fonctionner le logiciel et à le contrôler
- Attractiveness (Attrait)
  - Capacité à attirer/plaire à l'utilisateur
- Usability compliance (Conformité d'usage)
  - Capacité à respecter les standards/conventions/guides relatifs à l'utilisabilité



# Efficiency (Efficacité)

---

Capacité à fournir les performances appropriées selon les ressources utilisées dans les conditions spécifiées

- Time behaviour (Temps de réponse)
  - Capacité à fournir les réponses appropriées dans les temps et aux débits prévus
- Resource utilisation (Utilisation des ressources)
  - Capacité à utiliser les quantités et types de ressources appropriés
- Efficiency compliance (Conformité d'efficacité)
  - Capacité à respecter les standards/conventions relatifs à l'efficacité

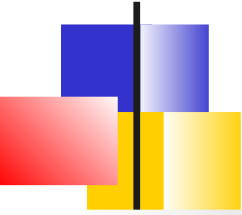


# Maintainability (Maintenabilité)

---

Capacité à être modifié (corrigé, amélioré, adapté) selon les changements (environnement/spécification)

- Analysability (Analysabilité)
  - Capacité à diagnostiquer les déficiences et causes d'erreurs et à trouver les parties à modifier
- Changeability (Changeabilité, adaptabilité, modifiabilité)
  - Capacité à implémenter des modifications de spécifications
- Stability (Stabilité)
  - Capacité à éviter des effets non attendus après des modification
- Testability (Testabilité)
  - Capacité à être validé, en particulier en cas de modification
- Maintainability compliance (Conformité de maintenabilité)
  - Capacité à respecter les standards/conventions relatifs à la maintenabilité



# Portability (Portabilité)

---

Capacité à être transféré d'un environnement à un autre

- *Adaptability* (Adaptabilité)
  - Capacité à être adapté aux environnements spécifiés
- *Installability* (Facilité d'installation)
  - Capacité à être installé dans un environnement spécifié
- *Co-existence* (Coexistence)
  - Capacité à cohabiter/partager des ressources avec d'autres logiciels
- *Replaceability* (Facilité de remplacement)
  - Capacité à être remplacé par un logiciel spécifié équivalent (ex : upgrade)
- *Portability compliance* (Conformité de portabilité)
  - Capacité à respecter les standards/conventions relatifs à la portabilité



# Qualité d'usage

---

Beaucoup plus simple : 4 critères

- *Effectiveness* (Efficacité)
  - Capacité à permettre à l'utilisateur d'achever ses buts dans le contexte spécifié
- *Productivity* (Productivité)
  - Capacité à permettre à l'utilisateur d'utiliser les ressources appropriées pour achever ses buts
- *Safety* (Sécurité)
  - Capacité à maintenir un niveau de sécurité acceptable
- *Satisfaction* (Satisfaction)
  - Capacité à satisfaire l'utilisateur dans les conditions d'utilisation prévues

# Métriques définies par la norme

Chacun des critères est accompagné de métriques

- Définition précise (comment mesurer, interpréter, type d'échelle,...)

External suitability metrics							
Metric name	Purpose of the metrics	Method of application	Measurement, formula and data element computations	Interpretation of measured value	Metric scale type	Measure type	Source input to measurement
Functional adequacy	How adequate are the evaluated functions?	Number of functions that are suitable for performing the specified tasks comparing to the number of function evaluated.	$X=1-A/B$ A= Number of functions in which problems are detected in evaluation B= Number of functions evaluated	$0 \leq X \leq 1$ The closer to 1.0, the more adequate.	Absolute	$X = \text{Count} / \text{Count}$ A= Count B= Count	Requirement specification (Req. Spec.) Evaluation report
Functional implementation completeness	How complete is the implementation according to requirement specifications?	Do functional tests (black box test) of the system according to the requirement specifications. Count the number of missing functions detected in evaluation and compare with the number of function described in the requirement specifications.	$X = 1 - A / B$ A = Number of missing functions detected in evaluation B = Number of functions described in requirement specifications	$0 \leq X \leq 1$ The closer to 1.0 is the better.	Absolute	$A = \text{Count}$ $B = \text{Count}$ $X = \text{Count} / \text{Count}$	Requirement specification Evaluation report



# Annexes

---

Square  
**Métriques Sonar**  
Dépréciation typologie des issues



# Complexité

---

## **Complexité cyclomatique :**

- Calculé à partir du nombre de chemins possibles dans le code. Chaque fonction a une complexité minimale de 1. Si la complexité augmente, la difficulté de teste augmente également

## **Complexité Cognitive :**

- Difficulté à comprendre les chemins dans le code  
*Voir <https://www.sonarsource.com/resources/white-papers/cognitive-complexity.html>*

## **Mesures dérivées :**

- Complexité moyenne par classe.
- Complexité moyenne par fichier
- Complexité moyenne par méthode.





# Documentation

---

## **Commentaires :**

- Nombre de lignes comportant une ligne de commentaire ou du code commenté.
- Nombre de lignes de code commentées
- Pourcentage de lignes de commentaires par rapport au total de lignes

## **Méthodes publiques et API**

- Pourcentage de l'API publique documentée
- Nombre de méthodes publiques non commentées.



# Duplications

---

## **Duplications**

- Nombre de blocs de lignes dupliqués
- Nombre de fichiers impliqués dans les duplications.
- Nombre de lignes impliqués dans les duplications.
- Pourcentage de lignes dupliquées



# Fiabilité Bugs

---

## **Bugs , New Bugs :**

- Nombre de bugs et de nouveaux bugs, par sévérité, par statuts

## **Indicateur de fiabilité** (*Reliability rating*)

- A = 0 Bug
- B = au moins 1 bug mineur
- C = au moins 1 bug majeur
- D = Au moins 1 critique
- E = Au moins 1 bloquant

## **Effort de correction** (*Reliability remediation effort*) :

- L'effort pour fixer tous les bugs ou tous les nouveaux bugs (en minutes et en se basant sur une journée de 8 heures),



# Sécurité

---

## **Vulnérabilités :**

- Nombre de vulnérabilités et de nouvelles vulnérabilités, par sévérité, par statut

## **Indicateur de sécurité** (*Security Rating*)

- A = Aucune vulnérabilité
- B = au moins 1 vulnérabilité mineure
- C = au moins 1 vulnérabilité majeure
- D = au moins 1 vulnérabilité critique
- E = au moins 1 vulnérabilité bloquante

## **Effort de sécurisation** (*Security remediation effort*) :

- Minutes pour corriger les vulnérabilités, sur tout le code, sur le nouveau code



# Security Hotspot

---

Un **Security Hotspot** met en évidence un morceau de code sensible à la sécurité que le développeur doit examiner.

Après une revue, on décide si la menace est réelle et éventuellement on applique un correctif.

Un workflow à 2 statuts est associé au security hotspot :

- *To review*
- *reviewed*



# Maintenabilité

---

## **Code Smells : Mauvaises pratiques de conception logicielle**

- Nombre de code smells, de nouveaux, par sévérité, par statut

## **Dette technique :**

- Effort nécessaire pour corriger tous les problèmes de maintenabilité (en minutes).

## **Rapport dette/développement :**

- Rapport entre le coût de développement et le coût pour fixer les problèmes de maintenance (Le coût d'une ligne de développement est estimé à 0,06 jour)

## **Taux de maintenabilité :** Indicateur en fonction du rapport précédent

- $A < 0,05$  /  $B < 0,1$  /  $C < 0,2$  /  $D < 0,5$  /  $E > 0,5$



# Métriques

---

## **Couverture des lignes :**

- Lignes à couvrir
- Lignes couvertes
- Pourcentage

## **Couverture des conditions :**

- Conditions à couvrir
- Conditions couvertes
- Pourcentage



# Mesures globales

---

**Couverture** : Mélange entre la couverture de lignes et la couverture de conditions

– **Couverture =  $(CT + CF + LC)/(2*B + EL)$**

- CT = Conditions ayant été évaluées à 'true' au moins une fois
- CF = Conditions ayant été évaluées à 'false' au moins une fois
- LC = Lignes à couvrir - lignes non couvertes
- B = Nombre total de conditions
- EL = Nombre total de lignes à couvrir

## Couverture des conditions

– **Couverture de condition =  $(CT + CF) / (2*B)$**

- CT = Conditions ayant été évaluées à 'true' au moins 1 fois
- CF = Conditions ayant été évaluées à 'false' au moins 1 fois
- B = Nombre Total de conditions

## Couverture des lignes

– **Taux de couverture de ligne =  $LC / EL$**

- LC = Lignes à couvrir moins les lignes non-couvertes
- EL = Lignes à couvrir





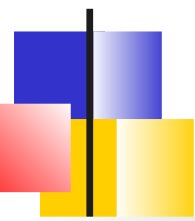
# Couverture des tests

---

Il s'agit de vérifier le code utile exécuté durant l'exécution des classes de tests.  
2 axes de vérification :

- Les lignes
- Les chemins ou conditions

Java : Sonar s'appuie sur des outils OpenSource  
*Cobertura, Jacoco, ....* et accède aux résultats  
générés par ces outils



# Rapport d'exécution des tests

---

Sonar remonte également un rapport sur l'exécution des tests unitaires :

- Nombre de tests unitaires
- Nombre de tests unitaires ignorés, échoués, en erreur
- Densité de réussite
- Durée des tests
- Nombre de conditions non couvertes par les tests
- Lignes non couvertes



# Annexes

---

Square  
Métriques Sonar  
**Dépréciation typologie des issues**



# SonarQube

---

Le travail quotidien des développeurs et *reviewers* consiste à traiter des *issues*.

Il y en a de 3 types :

- **Code Smell** : Mauvaise pratique de conception rendant difficile la maintenance
- **Bug** : Un problème représentant une erreur dans le code. Cette erreur se manifestera un jour, il faut donc le corriger ASAP
- **Vulnérabilité** : Un problème lié à la sécurité permettant une attaque

Des coûts sont associés au traitement de ces problèmes :

- **Dette technique** : Le temps estimé pour corriger tous les problèmes de maintenabilité
- **Coût de correction** : Le temps estimé pour corriger les bugs et les vulnérabilités



# Sévérité

SonarSource définit également des sévérités pour les issues :

- **BLOCKER** :  
Haute probabilité d'impacter le comportement en production (fuite de mémoire, de connexion JDBC, ...).  
=> Doit être fixé ASAP
- **CRITICAL** :  
Bug avec faible probabilité d'impact ou défaut de sécurité.  
=> Doit être revu ASAP
- **MAJOR** :  
Défaut de qualité pouvant impacter la productivité du développeur (code mort, blocs dupliqués paramètres inutilisés, ...)
- **MINOR** :  
Défaut de qualité ayant un léger impact sur la productivité. (Lignes trop longues, "switch" à la place de if, ...)



# Statut des issues

---

Une fois créé, une issue peut prendre les statuts suivants :

- **Ouvert** : Défini par SonarQube sur les nouveaux problèmes
- **Confirmé** : Défini manuellement pour indiquer que le problème est valide
- **Résolu** : Défini manuellement pour indiquer que la prochaine analyse devrait fermer le problème
- **Ré-ouvert** : Défini automatiquement par SonarQube lorsqu'un problème résolu n'a pas été corrigé
- **Fermé** : Défini automatiquement par SonarQube pour les problèmes créés automatiquement.



# Résolution

---

Les issues fermées peuvent avoir l'une des résolutions suivantes :

- **Corrigé/Fixed** : Défini automatiquement lorsqu'une analyse ultérieure montre que le problème a été corrigé ou que le fichier n'est plus disponible (supprimé du projet, exclu ou renommé)
- **Supprimé** : Défini automatiquement lorsque la règle associée n'est plus disponible.

Les issues résolus peuvent avoir l'une des résolutions suivantes :

- **Faux positif** : Défini manuellement
- **Won't fix** : Défini manuellement



# Revue d'issues

---

Le travail du *reviewer* consiste à revoir les résultats de l'analyse et de modifier éventuellement les statuts des problèmes :

- **Confirm** : Reconnaissance du problème. Le statut passe de « Ouvert » à « Confirmé »
- **Fausse détection** : Indique que l'issue détectée par Sonar n'est pas réellement un problème
- **Won't fix** : On reconnaît le problème ; mais on l'accepte en tant que dette technique
- **Changement de sévérité** : Le problème n'a pas la sévérité détectée par Sonar. On change sa sévérité et les prochaines mesures de Sonar tiendront compte du changement
- **Résolu** : Le problème est censé être résolu. Lors de la prochaine analyse, Sonar fermera ou ré-ouvrira l'issue