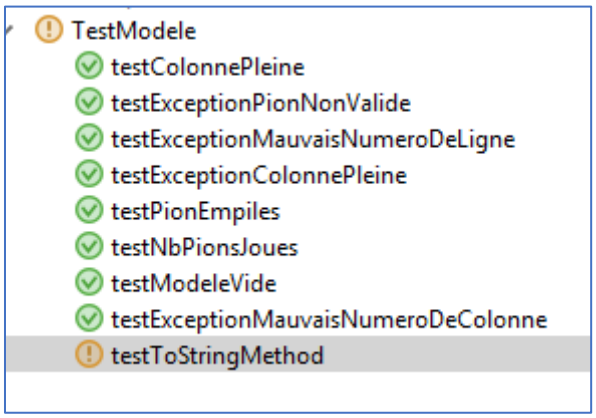
TD n°2 : Qualité du code

Objectifs :

1) Terminer le développement du code brut avec une approche TDD
2) Reprendre le code brut du modèle obtenu et appliquer une phase de réusinage (refactoring) pour le rendre robuste, réutilisable et adaptable.
3) Rendre les assertions des tests plus lisible en utilisant la bibliothèque hamcrest.
4) Introduction à la complexité du code
5) Paramétrer l'inspection de code checkstyle avec notre propre fichier de règles en incorporant des règles de détection de complexité
6) Utiliser le lancement avec des configurations Maven

Terminer le développement du modèle en mode TDD

Ecrire les tests manquants :



Coder de manière directe pour que tous les tests passent. Ajouter les méthodes vider() et toString()

Réusinage du code

Si ce n'est pas déjà fait aborder les points suivants :

- Toute mauvaise utilisation du code génère-t-elles une exception ?
- Ne peut-t-on pas optimiser le code en typant les pions avec un enum pour faciliter l'utilisation du modèle ?
- Avez-vous respecté les normes de nommage des méthodes ?
- Ne-peut-on pas faire une implémentation d'une interface ModelePuissance4 pour pouvoir un jour changer d'implémentation sans avoir à changer un autre code s'appuyant sur cette interface ?
- Les commentaires sont-ils suffisants ?
- Les javadocs font-elles parties de vos obligations projet ?
- Les tests ne sont-ils pas redondants ?

- Les noms des tests sont-ils véritablement parlant ?
- Peut-on utiliser un @before pour optimiser le code des tests ?
- Avez-vous 100% des lignes de code couvertes par vos tests ?
- …

3) La bibliothèque hamcrest

La bibliothèque hamcrest permet d'exprimer des assertions d'une manière plus proche du langage naturel.

Exemple

verifyThat(expression , Is(true))

```
assertThat( modele.pionEnPosition(3,1) , equalTo( ROUGE));
```

Pour importer la bibliothèque hamcrest il suffit de rajouter une dépendance dans le fichier pom.xml du projet. Puis de reconstruire le projet (menu build all).

A FAIRE : Revoir toutes les assertions en utilisant hamcrest.

4) La complexité du code

Exemple de complexité :

- Trop de boucles imbriquées ou de chemins indépendants dans une méthode
- Trop de classes référencées dans une classe
- Trop de paramètres dans une méthode
- Trop d'opérateur booléens dans une expression booléenne

Rechercher sur le site de **checkstyle** quels sont les contrôles permettant de limiter ces complexités

Le nombre cyclomatique de Mac Cabe permet de calculer la complexité en termes de cycles (boucles) indépendant sur le graphe représentant la fonction. Question quelle est la complexité cyclomatique de la méthode toString() du modèle ?

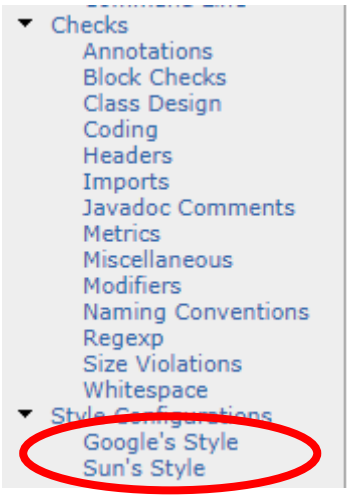
Pourquoi chercher à réduire la complexité ? Elle ralentit la maintenance corrective ET évolutive et fait perdre en productivité. Ne jamais dépasser 12 en complexité cyclomatique dans une méthode.

Le site de checkstyle : http://checkstyle.sourceforge.net/

5) Faire son propre fichier de règles de contrôle pour checkstyle

Il est possible de partir du fichier sun_checks.xml pour le personnaliser.

Sur le site de checkstyle, télécharger le fichier sun_checks.xml :



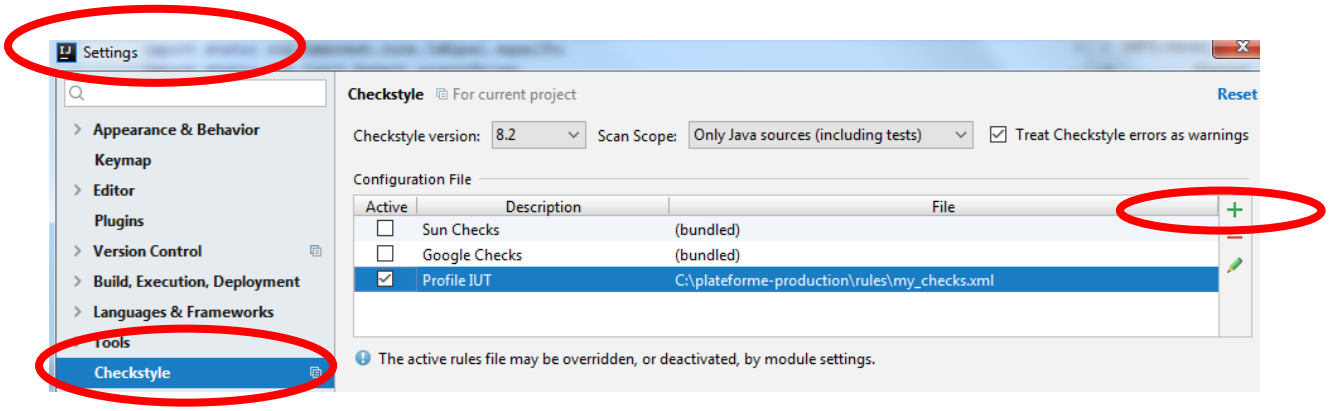
Dans ce fichier, supprimer la majorité des règles en gardant les règles :

- Sur les imports
- Sur le nommage
- Sur les longueurs maximales

En y incorporant les règles de complexité

Sauver le fichier en REGLES_IUT.XML dans votre plateforme de production.

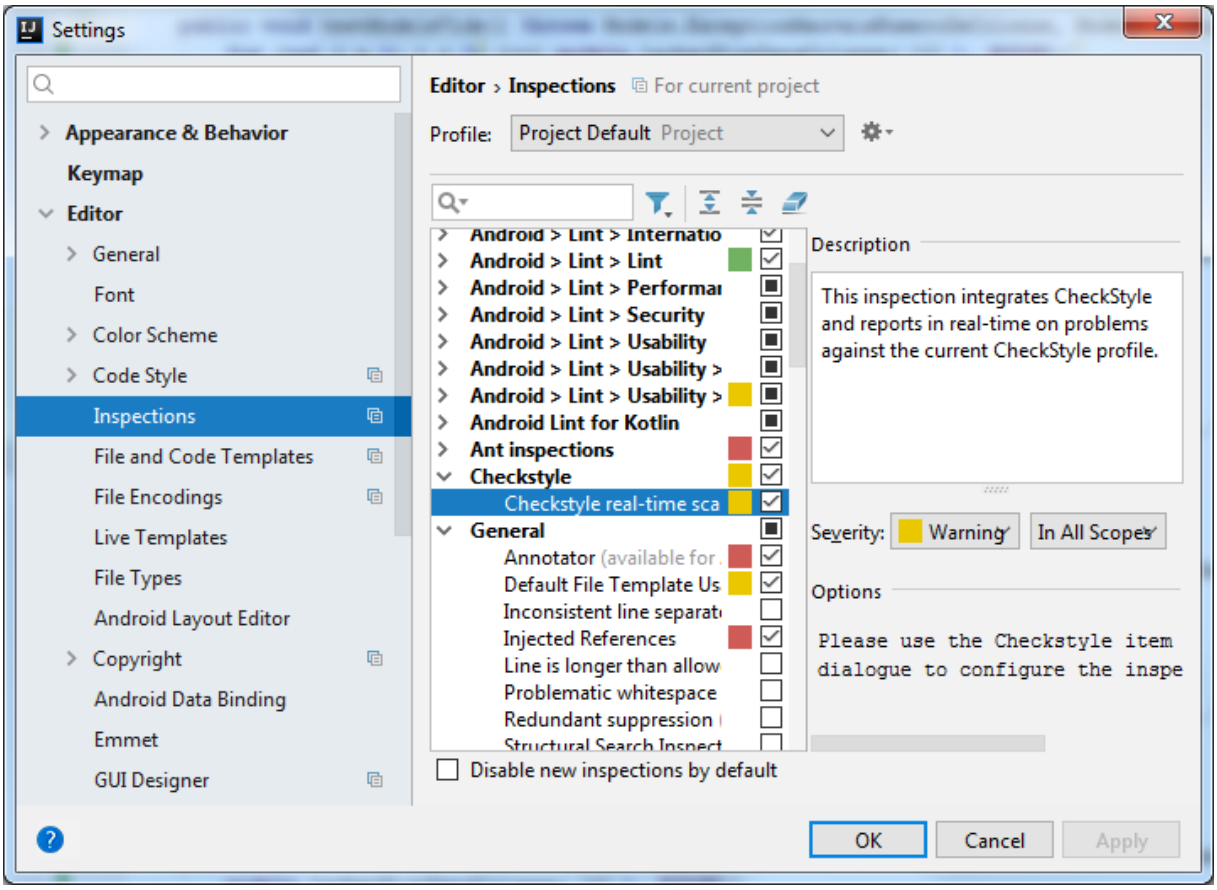
Paramétrage dans IntelliJ avec le menu Settings



Il suffit d'aller chercher le fichier avec + et de cliquer sur son répertoire et de le sélectionner

Activation et paramétrages des règles CheckStyle et Natives

Il suffit d'aller dans le menu : **settings | editor |inspections**

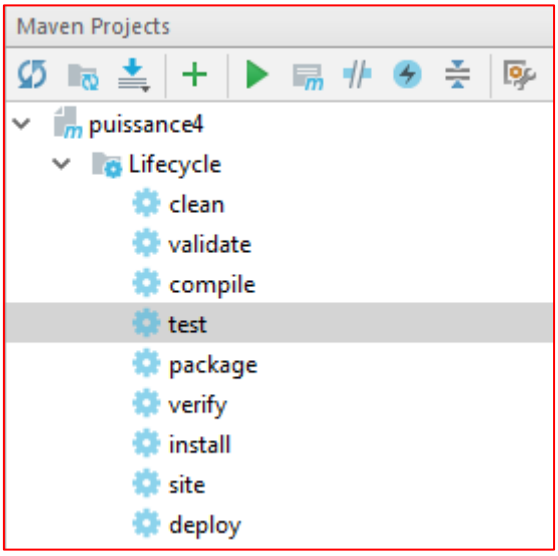


Il suffit ensuite de déterminer la sévérité des avertissement CheckStyle et de cocher ou décocher les contrôles IntelliJ dans **General.**

Vous pouvez ainsi paramétrer vos propres règles, celles qui paraissent importantes dans votre contexte.

Retour sur Maven

IntelliJ Propose une vue pour les configurations Maven. Pour l'activer, utiliser **{View | Tool Windows |Maven Projet}.** Il est alors possible de voir le cycle par défaut et lancer chaque étape.



1) Lancer la tâche clean
2) compile
3) test

Que fait la tâche package ?

Que fait la tâche install ?

Que fait la tâche deploy ?

Dans les prochains TD nous nous attacherons à partager notre code sur GitHub pour pouvoir travailler en groupe. Nous utiliserons ensuite Maven en dehors de l'atelier de développement.