IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

DevEnv – Initiation à PMD

Ingénierie du développement logiciel – IDL

Objectifs

- Comprendre ce qu'est un outil d'analyse statique de code
- Pratiquer PMD
- Configurer PMD

1 Installer une extension sous Eclipse

Installer un nouvel outil sous Eclipse peut se faire de différentes façons. À savoir, de la manière la plus simple à la plus compliquée :

- Utiliser le *market place* : il faut uniquement connaître le nom, mais attention aux synonymes.
- Utiliser le *software update* : l'URL du site de mise à jour est alors nécessaire.
- Une variante du précédent consiste à récupérer une archive (.zip) localement et ensuite utiliser le *software update* avec cette archive.
- Installation manuelle : placer un ou plusieurs fichiers `.jar` (dans le cas simple) dans le répertoire plugins, ou mieux dropins de votre installation Eclipse. Cela peut-être utile si vous avez une veille version d'Eclipse ou de votre système d'exploitation. Il existe des dépôts comme `https://jar-download.com/`, maven.
- Utilisation du code source : récupérer le code depuis un dépôt (cvs, svn, git, darcs, etc.), le compiler et fabriquer les archives `.jar` que vous installez comme dans le point précédent. C'est souvent plus compliqué car il faut construire l'exécutable en utilisant le `build` mais aussi le tester. S'il s'agit d'un greffon (*plugin*) il faut utiliser `run as Eclipse Application`.

Un redémarrage d'Eclipse est généralement nécessaire. Attention à bien choisir le bon plugin.

Notez aussi qu'un outil sous Eclipse est souvent associée à une "perspective" qui est elle-même un assemblage de "vues". Il est donc parfois utile de se placer dans la bonne perspective et d'ouvrir la ou les vues pertinentes.

2 PMD

PMD (`https://pmd.github.io/`) est un analyseur statique de code capable de traiter différents langages. Pour le cas de Java, il permet de détecter pas mal de problèmes/défauts/etc. :

PMD permet de détecter pas mal de choses par une analyse du code source Java :

- des *bugs* potentiels liés à un bloc vide dans les instructions `try/catch/finally/switch` ;
- des parties de code non utilisées (du « code mort ») ;
- des problèmes de performance, par exemple liée à l'utilisation des String ;
- des expressions trop complexes, par exemple `if` en trop, `for` à la place d'un `while` ;
- de la duplication de code.

Il existe sur le market place mais il arrive que l'installation se passe mal ou que l'extension ne fonctionne pas correctement. Nous vous invitons donc à suivre la documentation officielle d'installation de PMD [1] : `https://pmd.github.io/latest/pmd_userdocs_tools.html#eclipse`

⚠ ATTENTION : Lors de la configuration de l'extension (après installation et redémarrage de Eclipse) il faut aller dans la vue des préférences globales pour activer les règles de PMD que vous voulez utiliser (`preferences > PMD > Rule Configuration`, ou version en Français suivant votre installation). Penser à utiliser la perspective `PMD` sinon cela ne marche pas bien. Pour l'utilisation, "cliquer" à droite sur votre projet, ouvrir les propriétés et choisir PMD. Vous devez voir une fenêtre comme celle de la figure 1 et là vous pouvez choisir les règles.

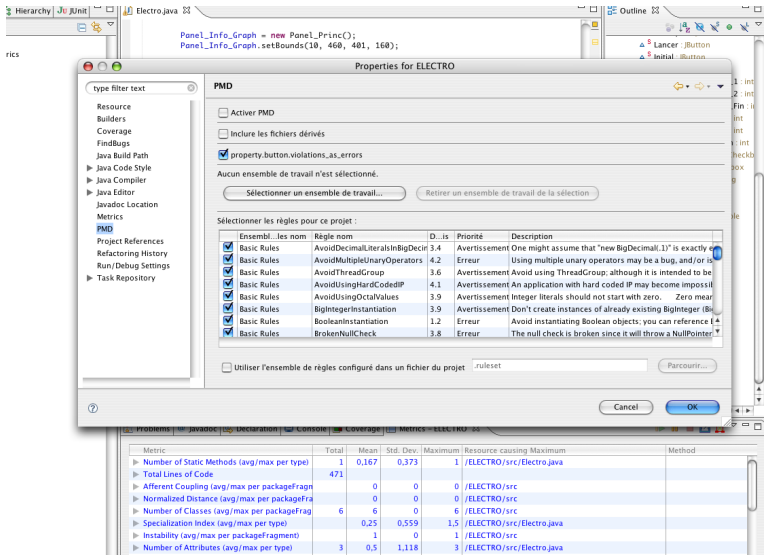
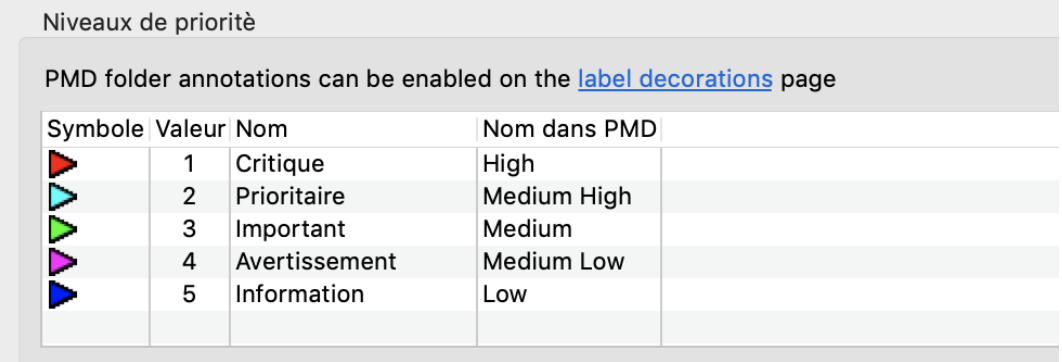


FIGURE 1 – PMD : le panneau de configuration.

Comme il est très riche, il faut pouvoir se focaliser sur certaines règles sinon l'utilisateur est noyé dans un flot de problèmes. Par exemple, commencer avec la détection du code mort puis régler les problèmes de blocs vides, puis voir les expressions trop complexes, etc. (voir la section 2.1). Les niveaux de priorité sont décrits dans la figure 2 [2].

Le *plugin* propose plusieurs vues utiles :

1. Sur les machines de l'école, PMD est normalement déjà installé.
2. Attention, l'intitulé personalisé (*custom names*) ne doit pas être pris comme référence, notamment si vous comparez avec des camarades utilisant une autre version de PMD ou de son greffon Eclipse, avec une autre localisation. Les intitulés en anglais nous semblent peu pertinents.


Niveaux de priorité

PMD folder annotations can be enabled on the label decorations page

Symbole	Valeur	Nom	Nom dans PMD
	1	Critique	High
	2	Prioritaire	Medium High
	3	Important	Medium
	4	Avertissement	Medium Low
	5	Information	Low

FIGURE 2 – PMD : niveaux de priorité des erreurs (peuvent varier selon les versions)

— la synthèse des alertes détectées : des petits boutons de couleurs permettent de filtrer les problèmes importants d'abord.
— la vue CPD : vue "*copy-paste detector*" une action particulière permet de détecter les copies de code et sauve un rapport dans votre projet sous `reports/cpd-report.txt`. Les rapports sont générés, suite à l'action génération des rapports, dans le répertoire `reports` du projet en cours d'analyse.

Il y a également un menu contextuel permettant de faire plusieurs actions, par exemple d'effacer tous les messages d'alertes ou de sauvegarder le rapport en XML, txt, html et csv.

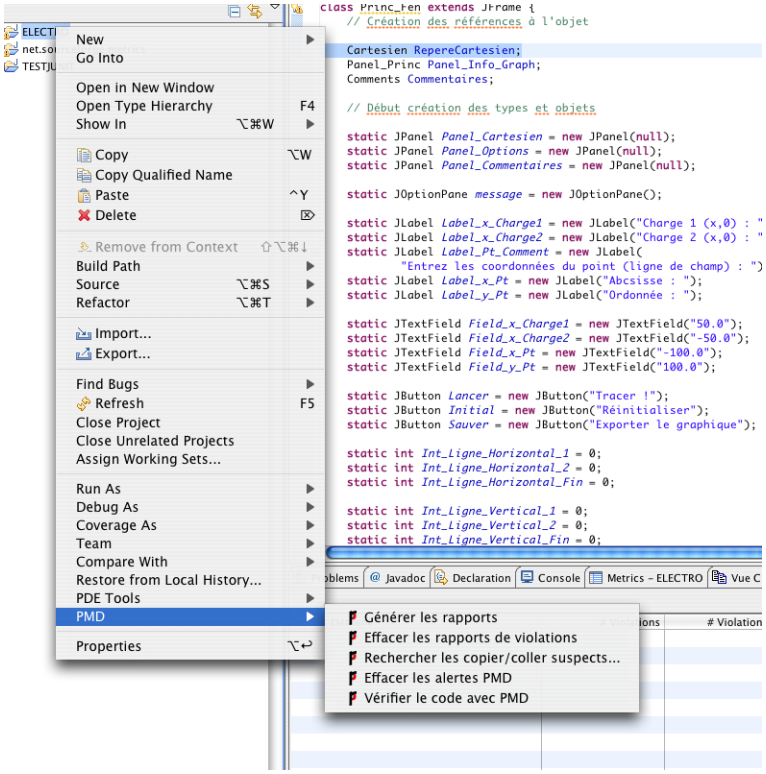


FIGURE 3 – PMD : le menu contextuel.

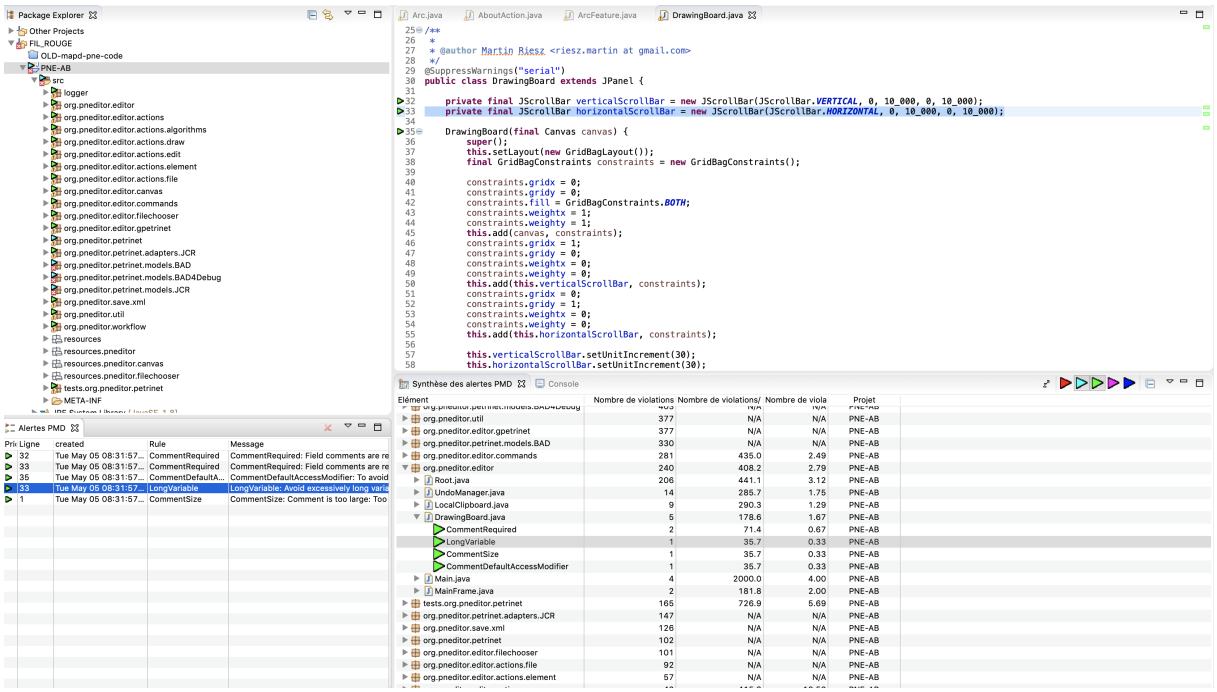


FIGURE 4 – PMD : en action sur le fil rouge.

2.1 Quelques règles

Pour réaliser les analyses et les rapports, PMD utilise des catégories de règles prédéfinies et vous pouvez en définir de nouvelles. La référence de règles Java actuelles (version PMD 7.0.0-rc4 et plugin pour Eclipse) se trouve sur ce lien : `https://pmd.github.io/latest/pmd_rules_java.html`. Vous pouvez néanmoins retrouver la documentation de référence pour chaque version de PMD en remplaçant `latest` par `pmd-`*`version`* dans l'URL..

Best Practices : C'est un ensemble de règles de bonnes pratiques de programmation Java. Il y en a environ une cinquantaine. Par exemple :
- Utilisation de `System.(out|err).print` au lieu de `log`
- imports non utilisés.

Code Style : qui inclut de règles telles que : Ne pas importer Java.lang, parce que ce sont des classes importées automatiquement.

Design : problèmes concernant la conception : classe abstraite sans méthode abstraite, classe non finale avec seulement des constructeurs privés, comparaison d'objets avec ==, etc

Documentation : problèmes tels que : constructeur vide sans commentaire qui l'indique, ou de commentaires politiquement incorrects.

Error Prone : règles qui détectent des constructeurs qui sont soit cassés, pas clairs ou qui peuvent être susceptibles d'être à l'origine de failles. Par exemple : *initializer* ou bloc *try* vides, l'usage de variables non parlantes (*literals*) à l'intérieur d'expressions conditionnelles.

Multithreading : règles qui concernent multiples *threads* d'exécution.

Naming Rules : oublis des majuscules/minuscules, noms du fichier et de la classe, etc.

Peformance : du code sous-optimal

Security : défauts potentiels liés à la sécurité

Unused Code Rules : *dead code*

Il y a des ensemble de règles liées à des framework particuliers (JEE, Java Beans, JUnit, etc.) mais aussi pour la migration entre les versions du JDK.

Exercice 1 (*Votre travail*)

Si vous utilisez votre machine et que PMD n'est pas encore installé, installez-le (il faut ensuite redémarrer Eclipse).

▷ **Question 1.1 :**

1. **Configurer PMD pour ne tester que les règles relatives à Java. Expérimenter quelques problèmes avec le comportement du greffon.**
   (a) **Aller dans les préférences globales d'Eclipse > PMD > Rule configuration**
   (b) **Puis ne sélectionner que les règles Java qui sont critiques ou importantes ou prioritaires et les exporter**

   ⚠ **Selon les versions de PMD et la langue du système, les priorités des règles peuvent être nommées différemment : par exemple *Blocker, Critical, Urgent* sur les versions récentes de PMD en anglaisi**
2. **Le tester sur le projet bad4debug UNIQUEMENT, en le sélectionnant et en activant le menu contextuel PMD "vérifier le code"**
3. **Quel(s) type(s) d'erreurs trouvez-vous ?**
4. **Corriger toutes les erreurs dites critiques**
5. **Trouver des exemples des cas ci-dessous, combien et les corriger**
   — **Exemples d'utilisation de `println` (règle `SystemPrintln`)**
   — **Exemples de variables trop courtes (règle `ShortVariable`)**
   — **Un seul return dans une fonction (règle `OnlyOneReturn`)**
   — **Exemple d'attribut ou de paramètre pouvant devenir final car jamais cible d'une affectation (règles dont les noms terminent par `CouldBeFinal`)**
   — **Loi de Demeter : comprendre, corriger et critiquer si possible cette règle (`LawOfDemeter`)**
   — **Tester la fonctionnalité de détection "copier-coller". Comment corriger ces problèmes ?**
   — **Les corriger dans votre version locale et vérifier que cette correction est bien faite.**
6. **Appliquer PMD à votre installation du fil rouge, mais uniquement le jeu de règles Java précédent**
   (a) **Combien de violations voyez vous ?**
   (b) **Donner 3 exemples de nouveaux problèmes identifiés**

Trav. perso. **vous êtes fortement encouragés à les tester sur vos propres projets ... et à lire quelques documents complémentaires**

3 Complément : flot de contrôle

Il est parfois utile d'avoir une vue du flot de contrôle d'une partie de son code. Un tel outil était intégré dans PMD mais il a disparu. Un autre outil plus simple et plus limité, `eclipsefcg` (*Eclipse Flow Chart Generator*), est disponible comme *plugin* : `http://eclipsefcg.sourceforge.net/`. Une fois le *plugin* installé, il faut ouvrir une classe, sélectionner la méthode voulue dans la vue

Outline et sélectionner dans le menu contextuel l'entrée `CFGgenerator > build`. Une vue interactive s'ouvre avec le flot de contrôle de la méthode. Ce type d'outil est utile pour avoir une idée du taux de branches dans un code et donc des tests minimaux que l'on devrait écrire. Pour obtenir un graphe non-trivial, il faut choisir une méthode qui effectue des calculs. Vous pouvez par exemple tester avec `isTriggerable()` de la classe `Transition`.

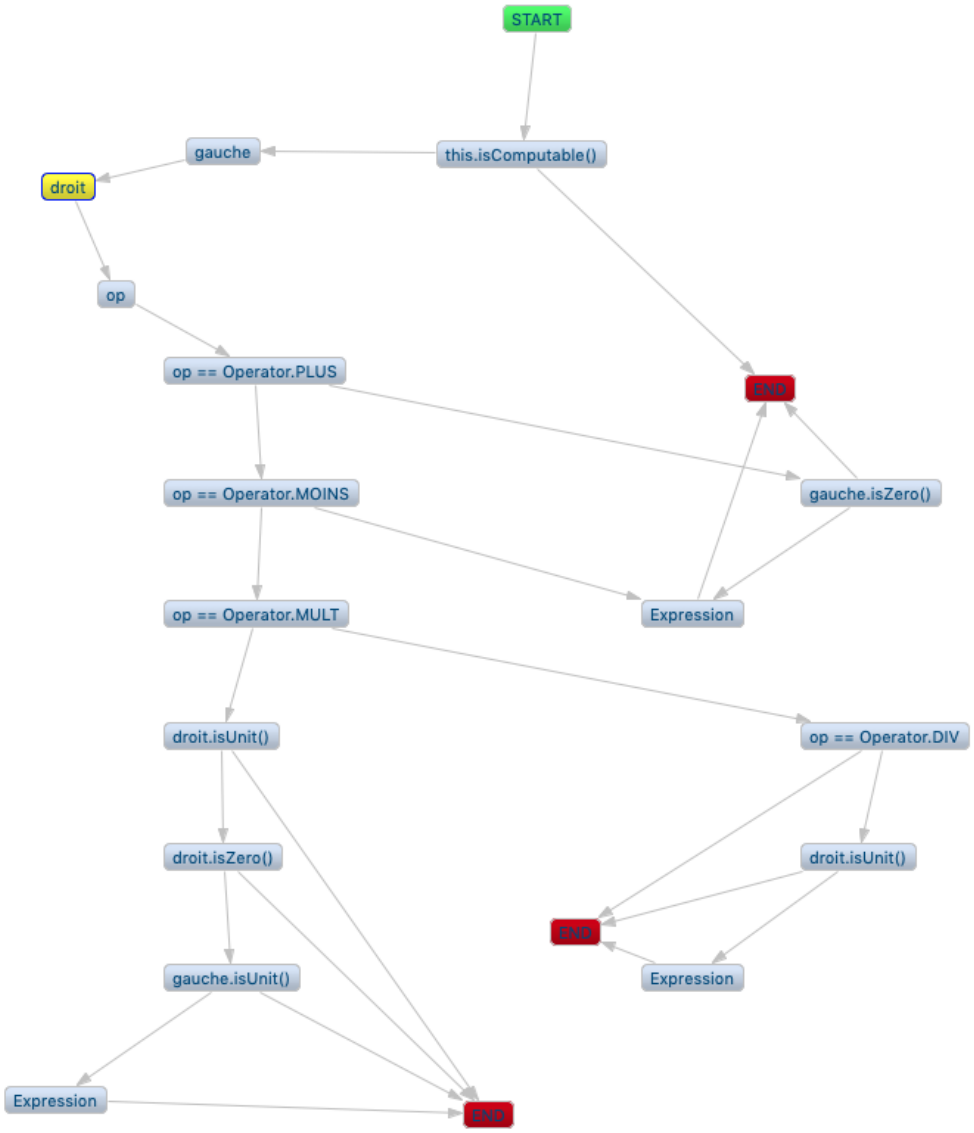


FIGURE 5 – CFG : un exemple de flot.