



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

DevEnv – Initiation à PMD

Ingénierie du développement logiciel – IDL

Correction

Objectifs

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

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

1. 15 erreurs critiques, 11 prioritaires, 295 importantes, 0 avertissements, 0 information
2. Les erreurs critiques sont des problèmes de conventions de nommage.
3. Exemple d'utilisation de `println` : `PetriNet` ligne 215, mais 10 dans `Test2Debug` pas important.
4. Exemples de variables trop courtes : `ShortVariable` `PetriNet` ligne 94 et 95

5. OnlyOneReturn : PetriNet méthode getTransitionNth
6. MethodArgumentCouldBeFinal : PetriNet lignes 41 et 51 par exemple.
7. La loi de Demeter dit : ne pas faire de séquence d'appels par notation pointée dans une classe. Par exemple ligne 186 de Place. Si elle a du sens dans un contexte purement objet, elle est plus discutable dans le cas où l'on veut adopter un style plus fonctionnel qui lui se base sur des chaînes d'appels. Quelques informations ici <https://h-deb.clg.qc.ca/Sujets/Developpement/Lois-principes.html> et https://en.wikipedia.org/wiki/Law_of_Demeter.
8. Détection de plusieurs problèmes de couper/coller : il y en a principalement 5 (en oubliant Test2Debug). Le premier, le troisième montrent qu'une interface pourrait être utile. Le deuxième ne devrait pas être difficile à corriger. Le quatrième et le dernier sont dus à la génération de code pour le hashage et l'égalité.
9. Le nombre de violations pour le Fil Rouge est énorme mais ce n'est pas forcément ni surprenant ni désastreux. Il faut se concentrer sur les problèmes les plus critiques.
10. Exemples de nouveaux cas : UseUtilityClass, CommentRequired, LongVariable, etc.

1 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`.

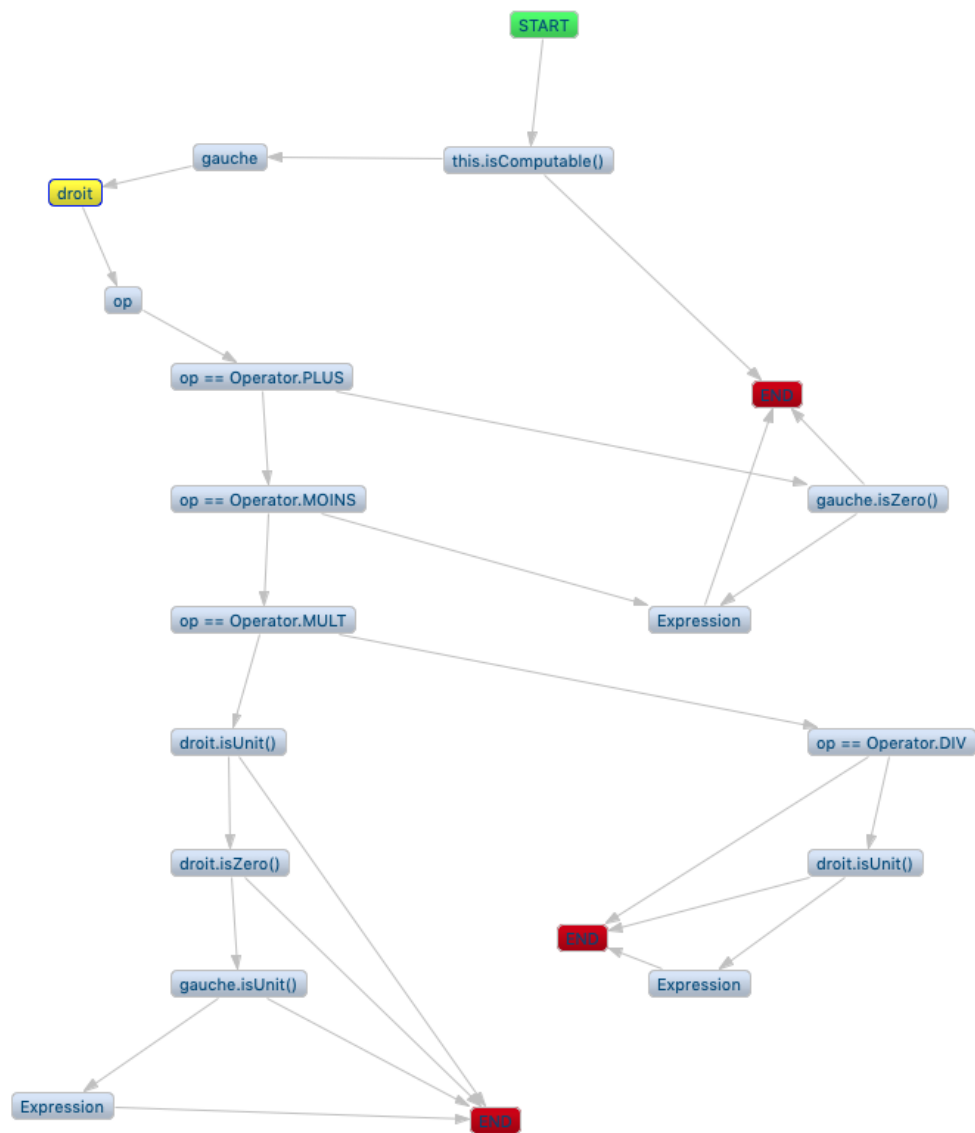


FIGURE 1 – CFG : un exemple de flot.