M2 Informatique AIGLE

HMIN306

Évolution et Restructuration

Rapport de TP

TP1 : Généralités

Amandine Paillard,
Bachar Rima

16 janvier 2020

Table des matières

Chapitre 1

TP1 : Généralités

1.1 Exercice 1 : Compréhension des concepts liés à l'évolution / maintenance des logiciels

Cet exercice consiste à mettre en relation différents concepts dans un modèle UML. Les concepts entrants en jeu sont liés au domaine de l'évolution et de la maintenance logicielle et sont nombreux. En voici quelques uns :

Maintenance : « *la modification d'un logiciel, après sa livraison, afin de corriger des défaillances, d'améliorer sa performance ou d'autres attributs ou de l'adapter suite à des changements d'environnements* » IEEE 610.12, 1993. L'activité qui regroupe à la fois l'ajout de fonctionnalités à un logiciel, son adaptation aux changements de contexte et aux nouvelles technologies, l'amélioration de ses performances, la correction des erreurs.

Évolution : les « *changements apportés à un logiciel pour s'adapter à un nouvel environnement et/ou besoin.* » (IEEE, DOI : 10.1109/ICCASM.2010.5620014, https://ieeexplore.ieee.org/document/5620014).

Réingénierie : il s'agit de la restructuration ou réécriture de tout ou partie d'un système sans changer sa fonctionnalité.

Rétro ingénierie : analyser un programme sans spécifications originales pour le comprendre à un niveau supérieur d'abstraction. D'après l'IEEE, cette notion est la « *création de modèles ou autres documentations à partir du code* » (DOI : 10.1109/ITNG.2010.189, https://ieeexplore.ieee.org/document/5501482).

Adaptation : réaliser les différents changements à faire pour que le logiciel subsiste dans son environnement. Il s'agit principalement de changements technologiques pour que le produit ne soit pas dépassé ou non compatible.

Analyse statique : analyser un logiciel sans l'exécuter pour en prévenir les comportements non désirables. Permet, entre autre, de trouver des erreurs de programmation, de conception et une baisse de qualité du code.

Analyse dynamique : vérifier que l'exécution du logiciel se déroule normalement, selon ses spécifications. Cela permet également de trouver des problèmes liés à l'utilisation de la mémoire ou du processeur.

Dette technique : c'est le retard pris dans la programmation d'un projet par manque de respect de la conception et des normes définies. Il faudra rattraper (rembourser) ce retard tout le long de la durée de vie du projet.

Extraction d'architecture : processus visant à analyser un logiciel pour en déduire son architecture.

Extraction de modèles : processus visant à analyser un logiciel pour en extraire un modèle le renseignant. Cette technique est utilisée notamment pour l'ingénierie dirigée par les modèles.

Slicing : processus de simplification d'un programme à l'aide d'un critère (une ou plusieurs lignes de code du programme initial par exemple). Cela est utilisé notamment pour le débogage.

Refactoring : processus de restructuration du code source d'un logiciel existant, sans changer son comportement externe (donc y ajouter des fonctionnalités ou corriger les éventuelles bogues) Son objectif est d'améliorer la lisibilité d'un code et le rendre plus générique.

Migration : faire la transition d'un logiciel d'une plateforme vers une autre.

Restructuration : une forme de la maintenance perfective d'un logiciel, transformant le système d'une représentation à une autre en conservant, entre les deux, le même comportement et un même niveau d'abstraction. Ces objectifs sont l'amélioration de la maintenabilité afin de faciliter certaines activités en faisant partie (l'ajout de nouvelles fonctionnalités, la correction de bogues non détectées préalablement).

Correction de bugs : processus de détection et de correction des déficiences d'un logiciel empêchant son comportement correct. Son objectif est d'assurer la cohérence entre les spécifications du logiciel et son implantation.

Impact des changements : décrire comment mener, à un coût efficace, une analyse complète des impacts d'un changement dans un logiciel en analysant sa structure et son contenu. Les principaux objectifs liés à ce concept sont :

- détermination de la portée d'un changement pour établir un plan et implanter le travail,
- développement d'estimations justes de ressources nécessaires pour effectuer le travail,
- analyse des coûts / bénéfices du changement demandé,
- communication de la complexité d'un changement donné.

Localisation de *feature* : l'identification dans le code source des parties correspondant à une *feature* spécifique. Une *feature* est une fonctionnalité du logiciel définie par ses spécifications et accessible aux développeurs et utilisateurs. Ce concept est une des activités les plus fréquentes en maintenance/évolution et est utilisé dans le cadre de l'analyse d'impact des changements.

Intégration continue : phase de maintenance préventive (dans le processus de développement logiciel), relié à l' *extreme programming*, consistant d'un ensemble de pratiques utilisé en génie logiciel permettant de vérifier à chaque modification du code source que le résultat des modifications ne produit pas de régression dans l'application développée. 2. objectifs : détecter et corriger les erreurs pendant la phase de développement du logiciel.

DevOps : ensemble de pratiques en génie logiciel combinant le développement logiciel et les opérations de la technologie de l'information. Son objectif est de raccourcir le temps entre l'intégration d'un changement d'un système et sa mise en production, tout en assurant la bonne qualité des livrables (*features*, *fixes*, *updates*). Le DevOps est réalisé par l'utilisation de *toolchains* pertinents aux catégories suivantes :

- codage : outils pour le développement du code source, sa revue, et sa gestion, etc,
- construction : outils d'intégration continue, etc,
- tests : outils de test continu.

Compréhension de logiciel : compréhension du code source. Réalisable par différents processus tels que l'examen et compréhension d'un système, l'acquisition de connaissances sur un programme informatique ou le développement de modèles mentaux de l'architecture, du sens et du comportement des logiciels. Ce concept a plusieurs objectifs tels que la récupération d'informations de haut niveau (modèles) sur un système, incluant sa structure (composants et leurs interrelations), ses fonctionnalités (opérations effectuées et composants cibles), son comportement dynamique (transformation de l'entrée en sortie), sa raison d'être (le processus de conception et les décisions prises dans ce processus), sa construction, ses modules, sa documentation et ses suites de tests...

Code review : phase de maintenance préventive (dans le processus de développement logiciel) dans laquelle les développeurs, *peer-reviewers* et les testeurs se mettent ensemble pour revoir le code dans son entièreté. L'objectif est de détecter et corriger les erreurs pendant la phase de développement du logiciel. Ce concept se réalise en lisant le code source ligne par ligne afin de détecter les éventuels défauts, s'assurer de la cohérence globale du logiciel, s'assurer de la qualité des commentaires, s'assurer de l'adhérence aux standards de programmation adoptés.

Software mining : application de la découverte de connaissances dans le domaine de la modernisation de logiciels en examinant les artefacts logiciels existants. Il s'agit d'une technique qui sert dans la rétro-ingénierie.

Redocumentation : processus de rétro-ingénierie utilisé pour générer de la documentation pour un système hérité qui en manque, en se basant sur les artefacts logiciels existants. L'objectif est de minimiser les efforts de rétro-ingénierie et faciliter la compréhension du logiciel.

Les définitions précédentes aident à organiser les concepts entre eux tels qu'ils sont présentés dans le diagramme `UML` en figure (1.1). Ce dernier est également disponible dans l'archive du rendu.

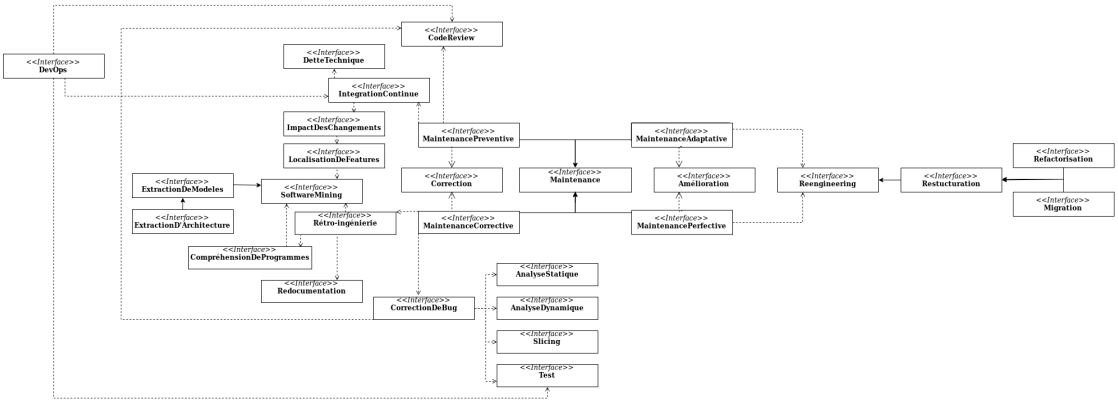

FIGURE 1.1 – Modèle UML montrant les concepts liés au domaine de l'évolution/maintenance de logiciel ainsi que leurs relations.

1.2 Exercice 2 : Outils de maintenance / évolution

Notre choix s'est porté sur `Checkstyle` et `CodeCity`.

1.2.1 `Checkstyle`

Description

`Checkstyle` est un outil permettant aux développeurs de respecter des normes de programmation. Cet outil se base sur l'analyse statique et vérifie que la syntaxe du code produit respecte des règles définies. Ces règles peuvent être des conventions publiées par des organismes reconnus comme Google ou Sun, mais elles peuvent aussi être des règles personnelles.

De plus, `Checkstyle` peut aussi être utilisé pour étudier d'autres concepts comme une mauvaise qualité dans la conception ou dans le développement d'une méthode. Dans ce TP, nous nous sommes concentrés sur la correction syntaxique.

Installation

Il y a plusieurs options pour utiliser cet outil.

— Installé comme *plug-in* dans un IDE (IntelliJ ou Eclipse) ;

— Configuré comme dépendance d'un projet Maven (*cf.* Listing 1.1) ;
— Installé [1] et utilisé en ligne de commande.

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>${checkstyle-maven-plugin.version}</version>
    <configuration>
        <configLocation>checkstyle.xml</configLocation>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Listing 1.1 – Dépendance Maven pour installer `Checkstyle`.

Exemple d'utilisation avec les conventions de Java Google et Sun

Une fois le *plug-in* installé, nous pouvons ouvrir le menu de `Checkstyle` (*cf.* Figure 1.2) ainsi que la perspective associée. Dans ce menu (accessible au chemin : `Windows > Preferences > Chekstyle`) nous pouvons choisir les règles à suivre pour notre projet. Sur la capture écran on remarque celles de Google et de Sun. Plus tard nous verrons qu'il est également possible de créer de nouvelles conventions de code en appuyant sur le bouton `New`.

1. https://sourceforge.net/projects/checkstyle/

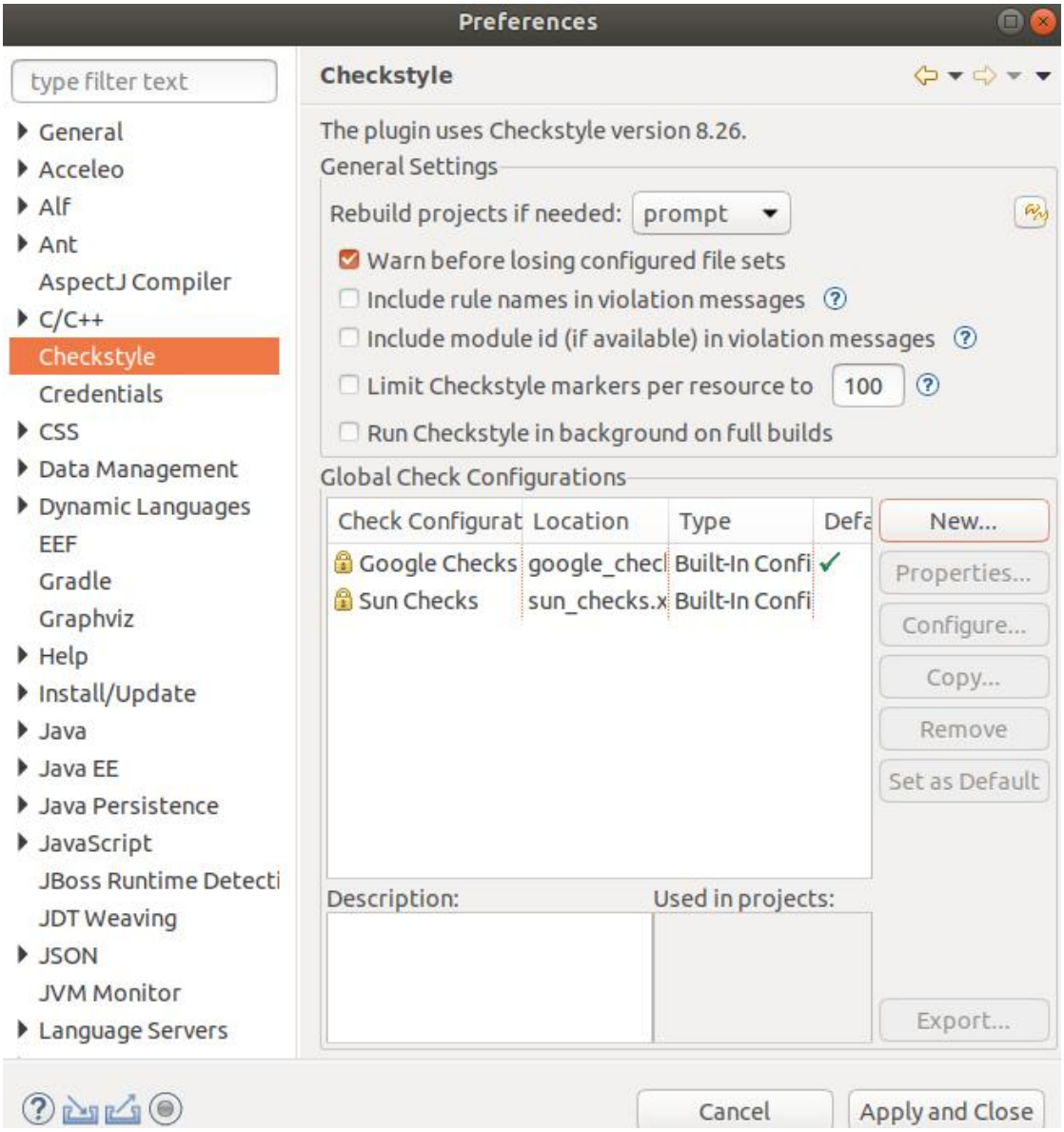

FIGURE 1.2 – Menu de `Checkstyle`.

Une fois les conventions à suivre choisies (dans un premier temps celles de Google), pour réaliser l'analyse de `Checkstyle`, il faut faire un clic droit dans le `Package Explorer` du projet à analyser puis cliquer sur le menu `Checkstyle` et enfin cliquer sur `Check Code with Checkstyle`. L'analyse obtenue est visualisable de deux manières différentes.

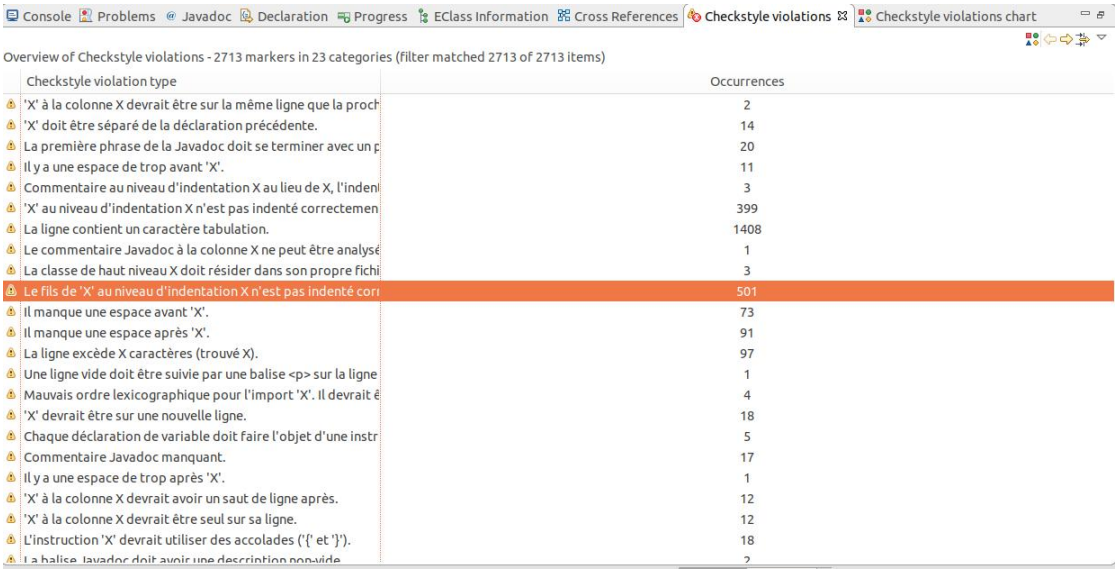

FIGURE 1.3 – Affichage de la liste de toutes les violations observées.

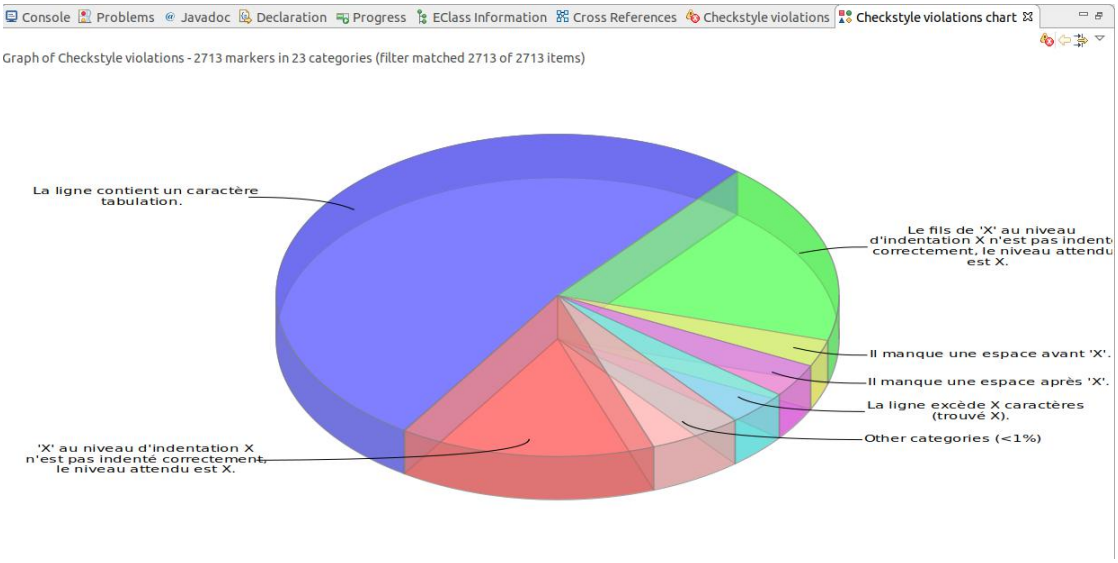

FIGURE 1.4 – Visualisation graphique selon les types de violation.

Les violations illustrées sont celles de la convention de Google pour Java concernant le code réalisé. Nous avons ensuite essayé d'appliquer les conventions de Sun sur le même projet. On remarque que ces dernières sont moins nombreuses.

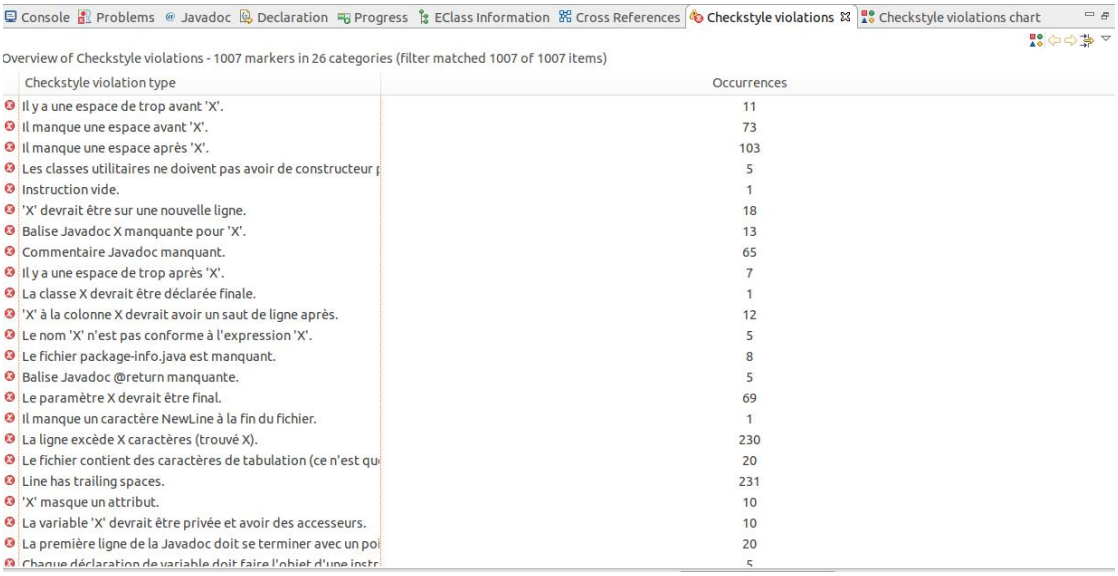

FIGURE 1.5 – Affichage de la liste de toutes les violations observées sur le même projet avec les conventions de Sun.

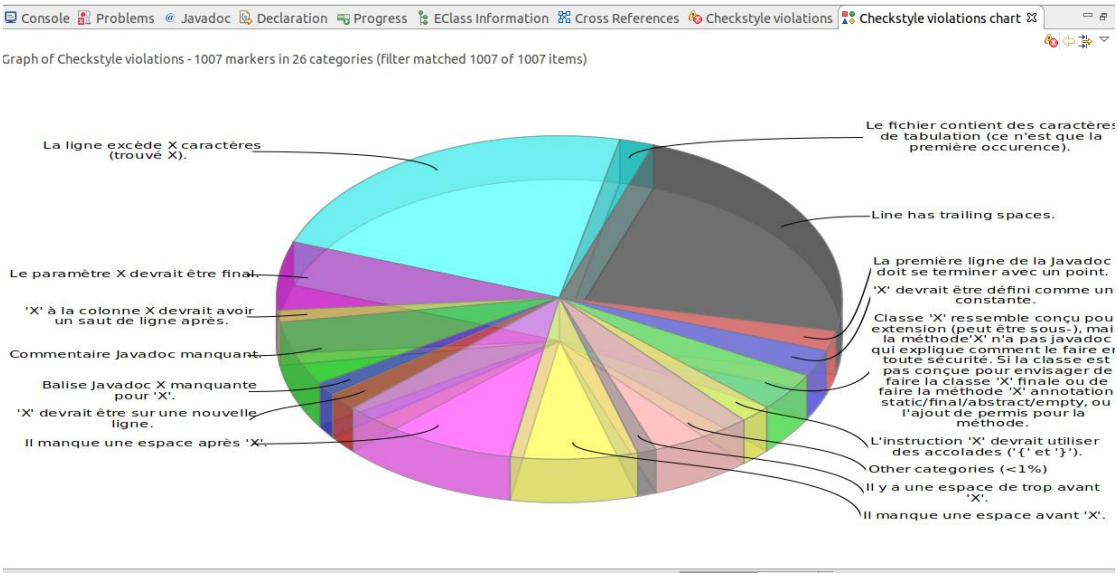

FIGURE 1.6 – Visualisation graphique selon les types de violation sur le même projet avec les conventions de Sun.

Correction de la syntaxe

En plus d'une vue générale des violations des règles, il est possible de voir toutes les violations pour une règle précise. En reprenant la figure 1.6, on peut accéder à la liste de toutes les erreurs concernant la règle « `Il y a un espace en trop avant X` »en cliquant sur cette dernière sur le graphe. On obtient alors une liste comme celle en figure 1.7

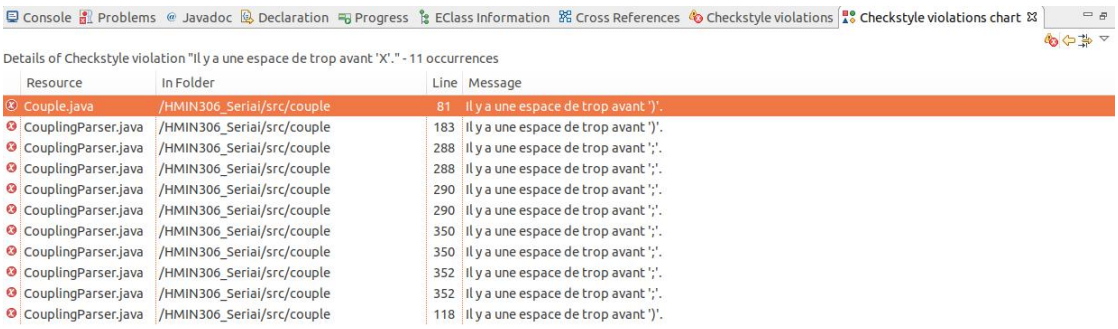

FIGURE 1.7 – Liste des violations de la règle « `Il y a un espace en trop avant X` ».

Lorsque l'on clique sur une violation, `Checkstyle` nous amène à la ligne où l'erreur a lieu pour que nous puissions la corriger.

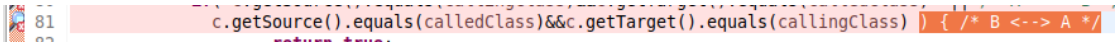

FIGURE 1.8 – Le problème vient de l'espace supplémentaire après la condition.

`Checstyle` soulève beaucoup de manques de respect des conventions de programmations de Sun et de Google. Nous avons voulu corriger toutes les violations soulevées par `Checkstyle` sur un projet. Pour cela nous avons pris un projet plus petit, distribué par notre enseignant pour avoir une base de code lors du TP suivant. Voici les erreurs soulevées par `Checkstyle`.

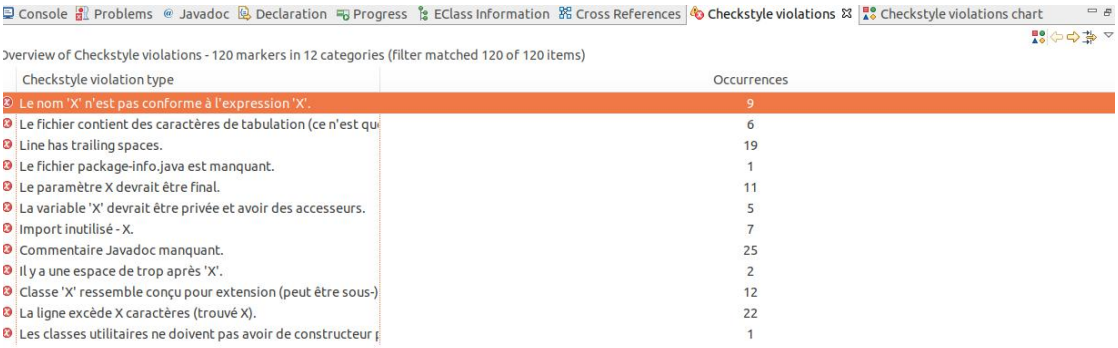

FIGURE 1.9 – Liste des violations soulevées sur un petit projet.

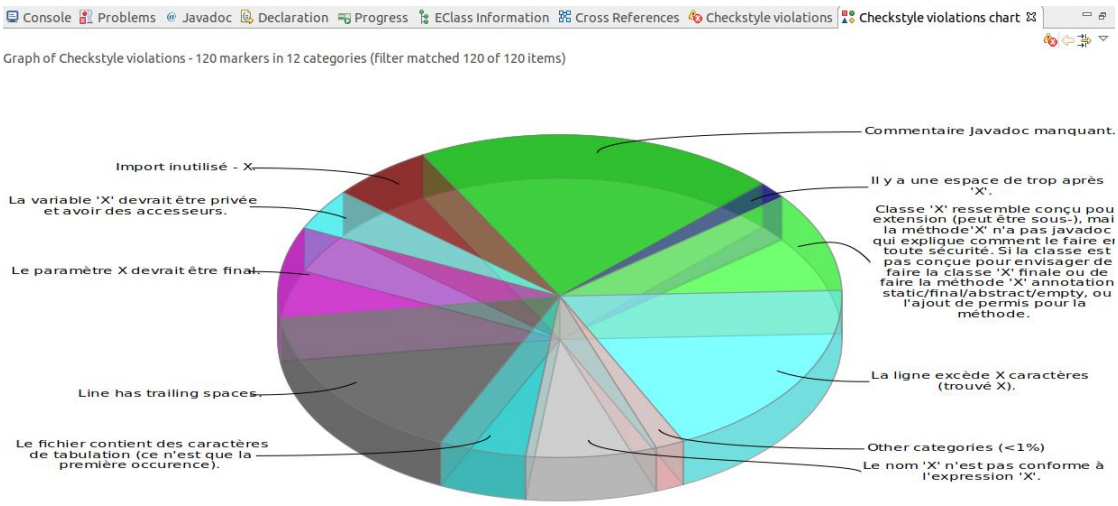

FIGURE 1.10 – Visualisation graphique des différentes règles violées sur un petit projet.

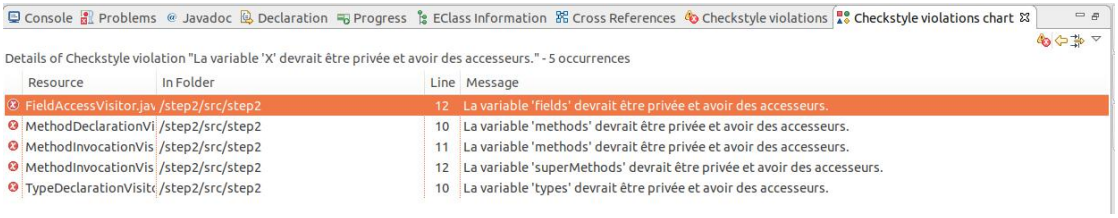

FIGURE 1.11 – Zoom sur une règle.

Après avoir corrigé toutes les erreurs de syntaxe soulevées par `Checkstyle`, celui-ci nous informe qu'il ne trouve aucune erreur.

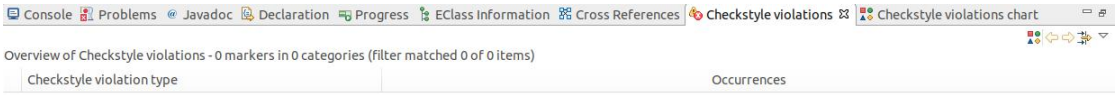

FIGURE 1.12 – Affichage obtenu après correction.

FIGURE 1.13 – Affichage « graphique » obtenu après correction

Exemple d'utilisation : créer ses propres règles

Jusqu'à présent nous avons utilisé des règles et des conventions de codage réalisées par des organismes officiels. Cependant, `Checkstyle` permet également de définir ses propres règles. C'est ce que nous avons fait avec un exemple trivial. Admettons qu'une équipe de développement peu précautionneuse ait une seule exigence : que tous les attributs d'instance des classes aient un nom commençant par `toto_` suivi d'un numéro[2]. Il va donc falloir définir une règle dans une nouvelle configuration.

2. Dans la réalité une telle règle serait ridicule mais il s'agit ici d'un exemple.

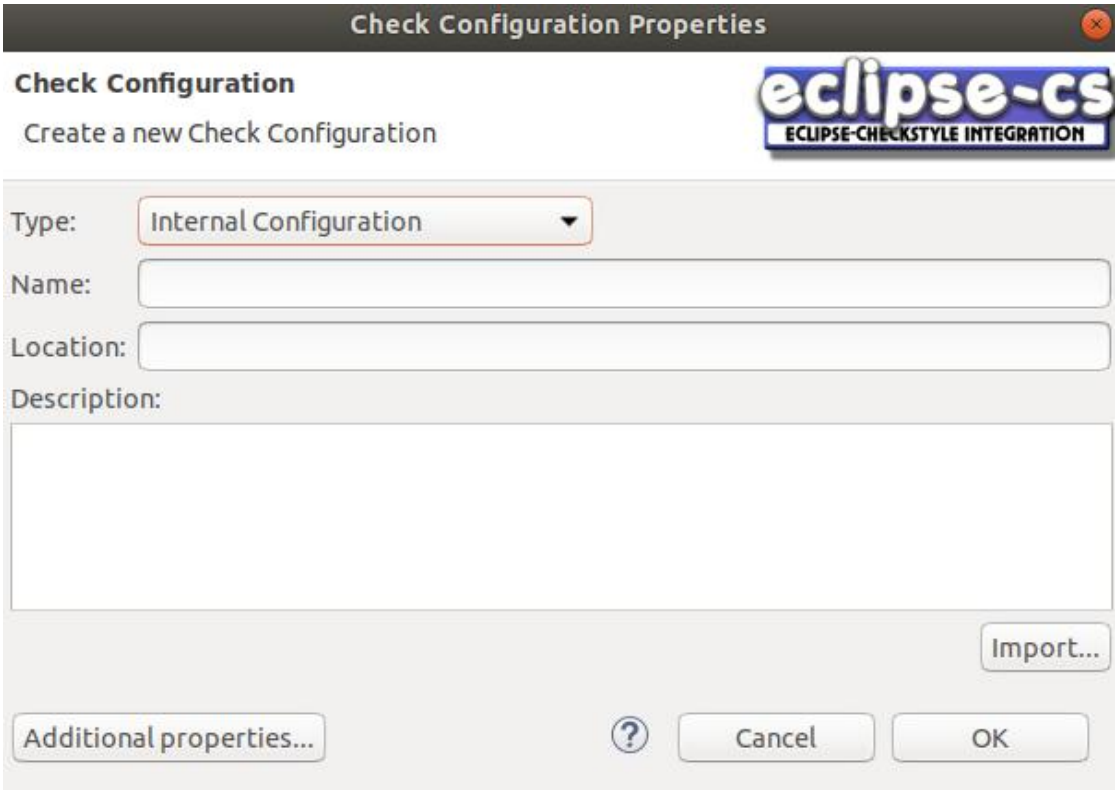

FIGURE 1.14 – Fenêtre de création d'une nouvelle norme.

Une fois le nom de la norme choisie, il est temps de définir des règles. Dans notre cas, nous ne voulons accepter qu'un certain format pour le nom des attributs d'instance d'une classe.

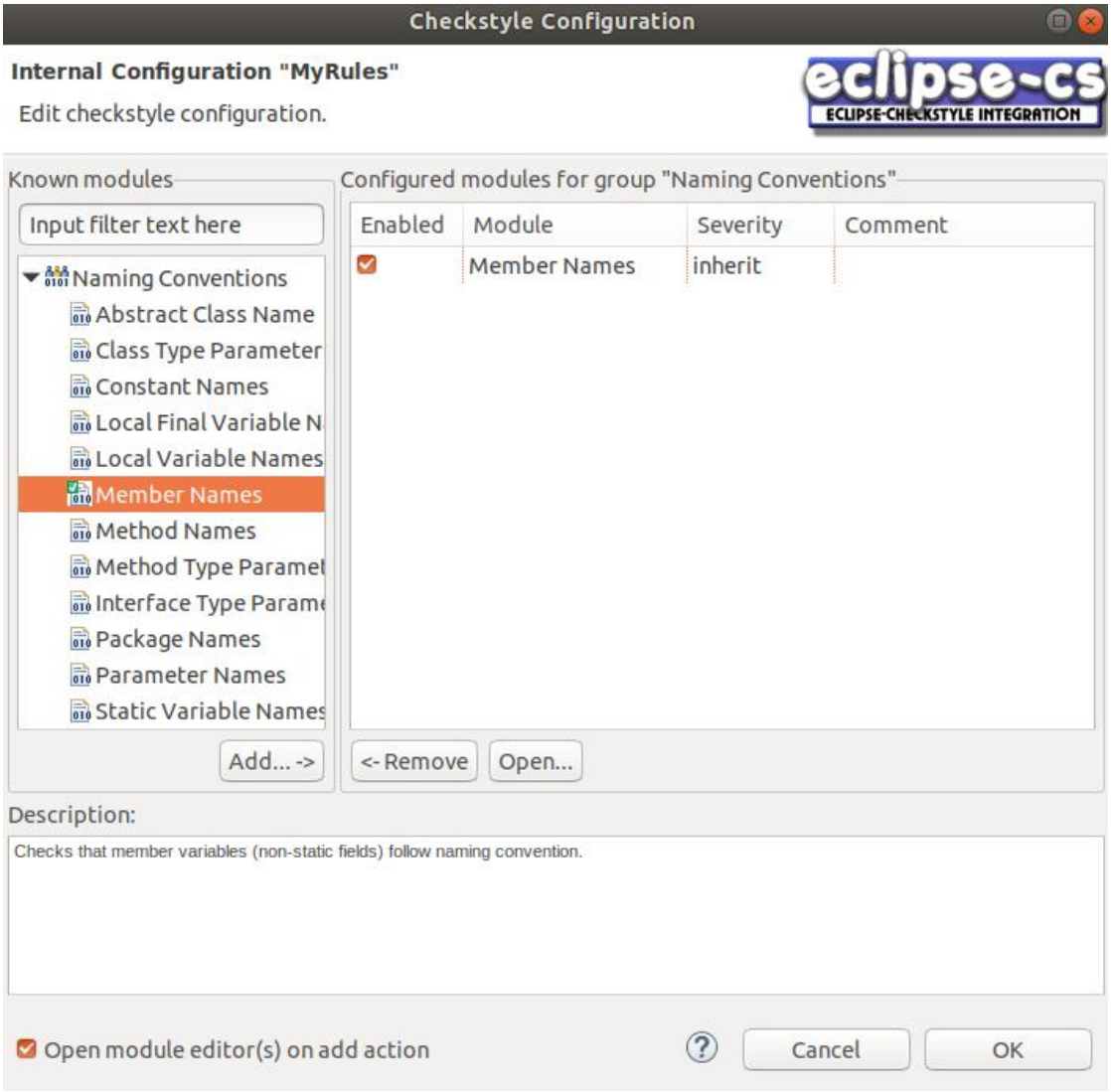

FIGURE 1.15 – Interface permettant de choisir des règles à créer.

FIGURE 1.16 – Nouvelle règle et valeur admise.

FIGURE 1.17 – Exemple de valeur non permise par la règle.

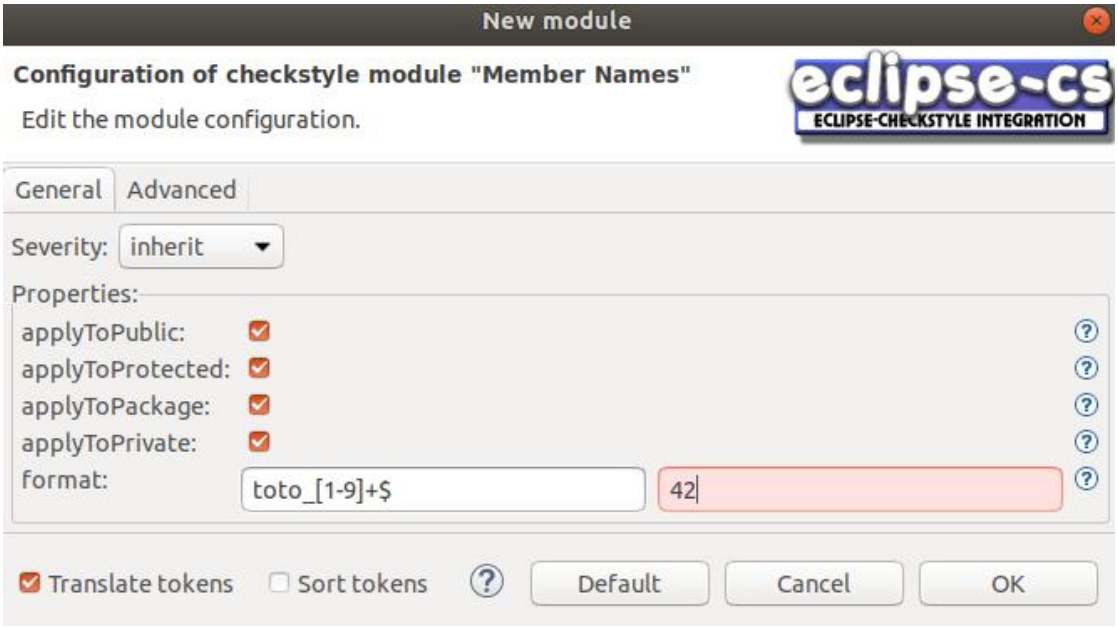

FIGURE 1.18 – Exemple de valeur non autorisée par la règle.

FIGURE 1.19 – Exemple de valeur non autorisée par la règle.

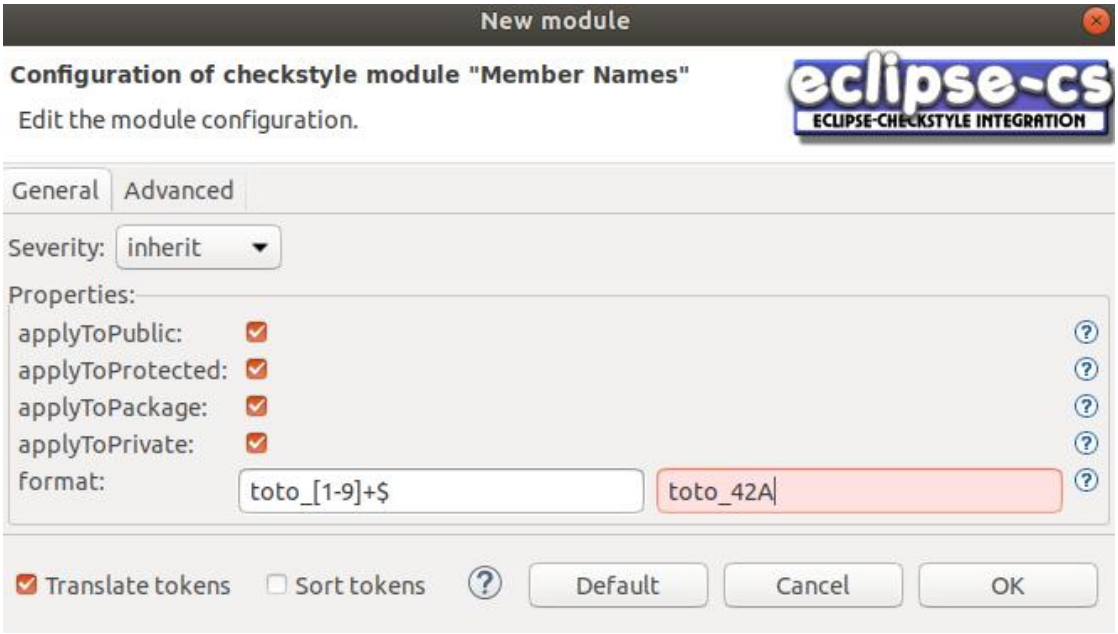

FIGURE 1.20 – Exemple de valeur non autorisée par la règle.

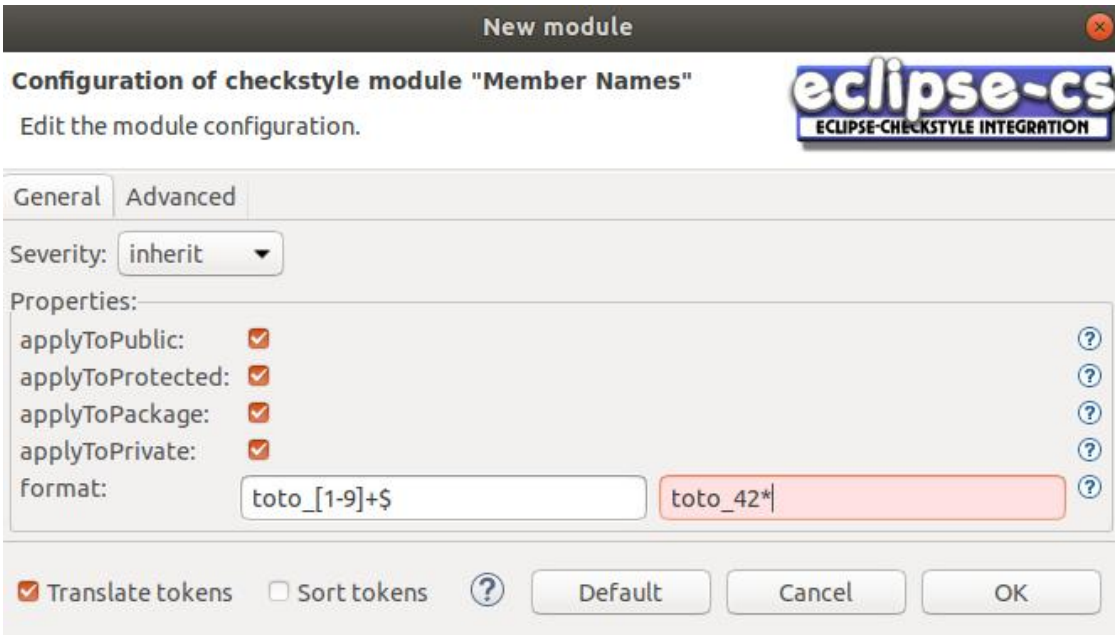

FIGURE 1.21 – Exemple de valeur non autorisée par la règle.

La nouvelle norme étant créée, il est temps de la tester sur le même projet précédent (nommé `step2`).

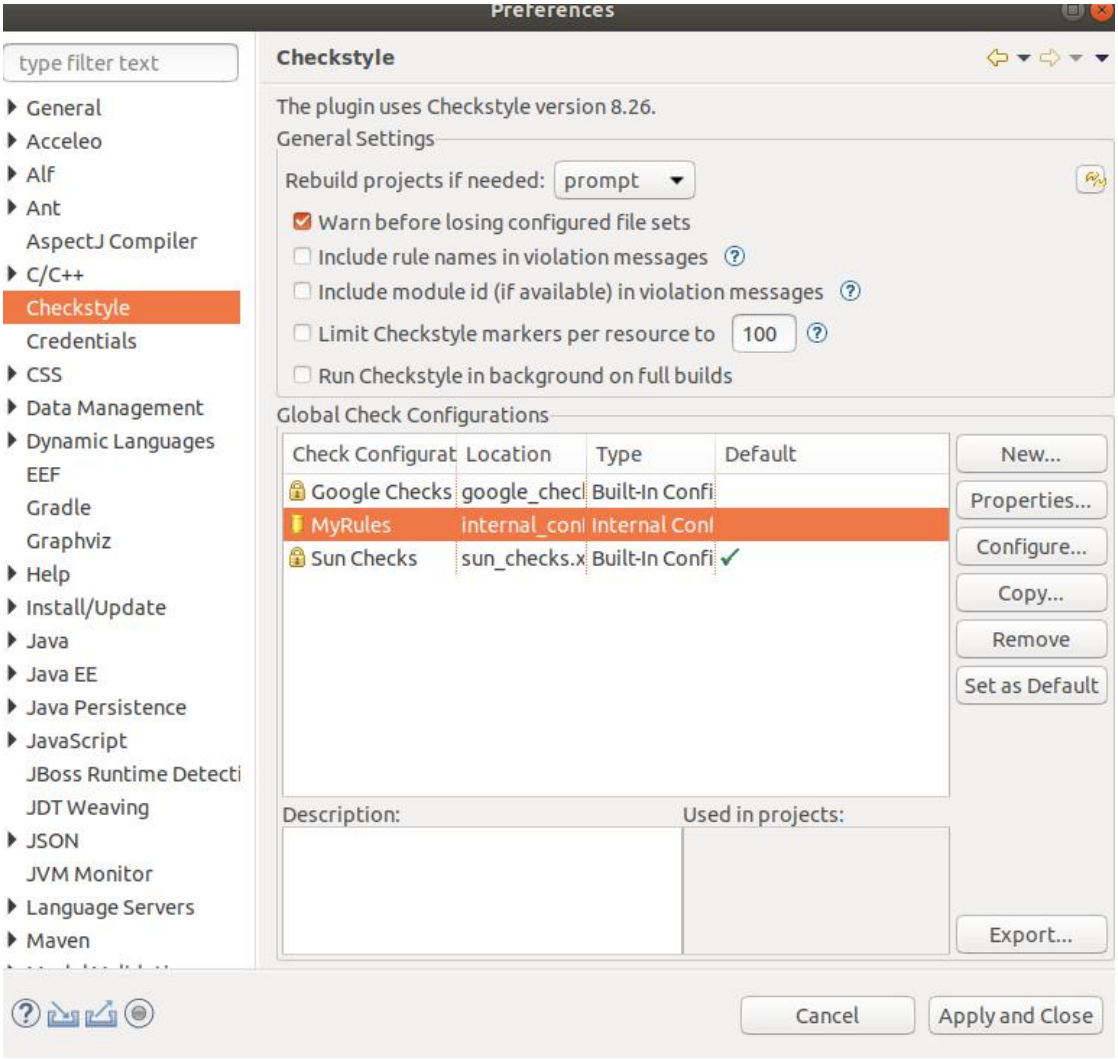

FIGURE 1.22 – La norme est bien créée, il faut la sélectionner pour l'appliquer.

FIGURE 1.23 – La règle est bien prise en compte.

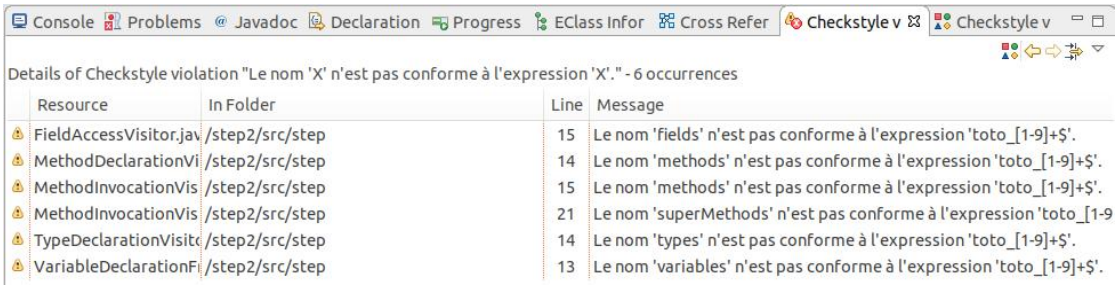

FIGURE 1.24 – Notre norme n'ayant qu'une seule règle, il est normal que `Checkstyle` ne remarque que celle-ci.

Conclusion

Nous avons vu comment utiliser `Checkstyle` pour suivre une norme de programmation, cet outil nous a permis de corriger l'ensemble d'un petit projet et nous a donné l'occasion d'en apprendre plus sur les conventions d'organismes comme Sun ou Google[3]. Enfin, nous avons pu voir comment définir sa propre convention de programmation.

`Checkstyle` est un outil puissant et utile pour veiller à ce que le code produit par une équipe de développement (donc constituée de plusieurs personnes) soit uniforme. Nous l'avons trouvé intéressant et souhaitons continuer à l'utiliser dans le futur.

1.2.2 `CodeCity` : Introduction à la métaphore « *Software City* »

Pour bien comprendre la philosophie de conception et d'utilisation de `JSCity`, un résumé de l'outil `CodeCity`, sur lequel se base `JSCity` dans sa conception et implémentation, s'avère primordial.

Dans la suite de cette partie nous résumons les aspects de `CodeCity` essentiels pour la compréhension de `JSCity`. Enfin, nous terminons par un résumé de `JSCity`, son installation et un guide de son utilisation.

Introduction

`CodeCity` est un outil multiplateforme interactif pour la visualisation 3D et l'analyse des logiciels orienté-objet indépendants des langages et des plateformes d'implémentation. Il permet la visualisation d'un logiciel à partir de modèles abstraits le décrivant, plutôt qu'à partir de son code source. Il a été introduit pour faciliter la compréhension, l'analyse, la rétro-ingénierie et le suivi de l'évolution des systèmes orienté-objet à grande échelle, en tant qu'une approche basée sur la métaphore « **ville logicielle** »(*software city*). Cette approche tire parti de l'aspect synthétique de la visualisation logicielle pour atteindre ses objectifs, notamment lorsque le système dévoile une quantité considérable d'informations à traiter, afin d'estimer sa complexité logicielle.

Les aspects de la métaphore *Sofware City*

La conception basée sur la métaphore *software city* décrit un système orienté-objet de la manière suivante :

3. Ces derniers préconisent par exemple que des espaces soient utilisés au lieu des tabulations pour l'indentation de code.

système : ville ;

paquetages : arrondissements ;

classes : bâtiments.

En outre, les propriétés visuelles des artefacts de la ville réflètent des mesures logicielles propres au système, décrites de la manière suivante :

le nombre de méthodes d'une classe[4] **:** hauteur du bâtiment ;

le nombre d'attributs d'une classe[5] **:** longueur et largeur de la base du bâtiment ;

le niveau d'imbrication d'un paquetage : la saturation de la couleur de l'arrondissement (*e.g.* les paquetages avec un niveau d'imbrication assez profond sont colorés en bleu foncé, alors que ceux ayant un niveau d'imbrication relativement superficiel le sont en bleu clair)

Cette approche permet ainsi non seulement d'avoir un aperçu de la structure du système étudié ainsi que de sa magnitude.

Composition et Interaction

La fenêtre de visualisation du logiciel (*cf.* Figure 1.25) est composée d'une fenêtre de navigation interactive et d'un panel informatif (*à droite*), affichant les informations de l'élément sélectionné dans la fenêtre de navigation. Une ville logicielle peut être explorée par la souris ou le clavier dans une perspective panoramique permettant de *zoomer* en avant/arrière et de graviter autour de la ville. De plus, on peut se déplacer entre ses artefacts et les survoler sur les plans horizontal et vertical.

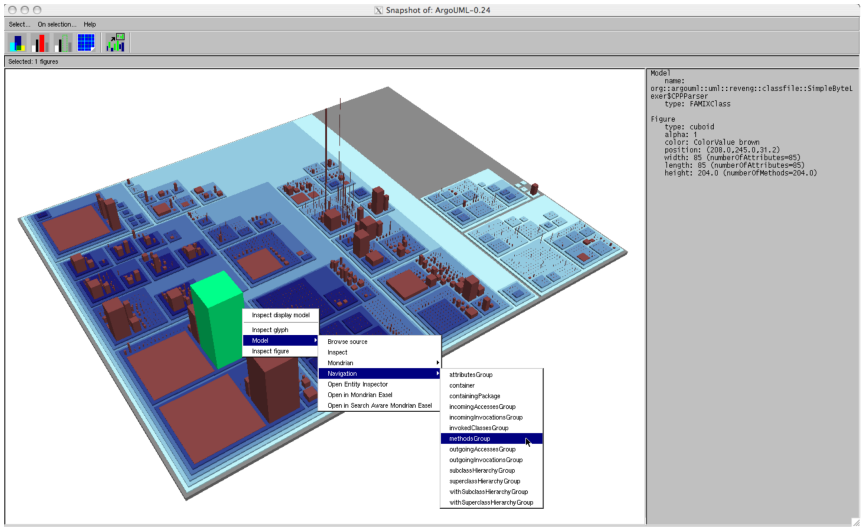

FIGURE 1.25 – Ville logicielle du système Java ArgoUML

Par ailleurs, l'interaction avec les artefacts peut se faire selon les modes d'interaction suivants :

interaction directe : le menu contextuel affiché sur un élément lors de sa sélection ;

interaction indirecte : un mécanisme de requêtes **absolu** (*e.g.* sur les interfaces, classes racines) ou **relatif** à la sélection en cours (*e.g.* les classes invoquant des méthodes parmi celles sélectionnées).

4. **NOM** : Number of Methods
5. **NOA** : Number of Attributes

Détection des dissonances conceptuelles

Outre la visualisation logicielle et le calcul des métriques, `CodeCity` permet d'évaluer la qualité logicielle du système du point de vue de sa conception, à travers des stratégies de détection identifiant des dissonances conceptuelles dans son design. La représentation des dissonances conceptuelles se base (*encore une fois*) sur la métaphore *software city*, pour visualiser leur localisation et distribution au sein du système.

Effectivement, en s'inspirant des cartes de maladies, `Code City` affecte une couleur vive spécifique à chaque type de dissonance conceptuelle, et des nuances de gris aux entités non-touchées par la dissonance (*cf.* Figure 1.26).

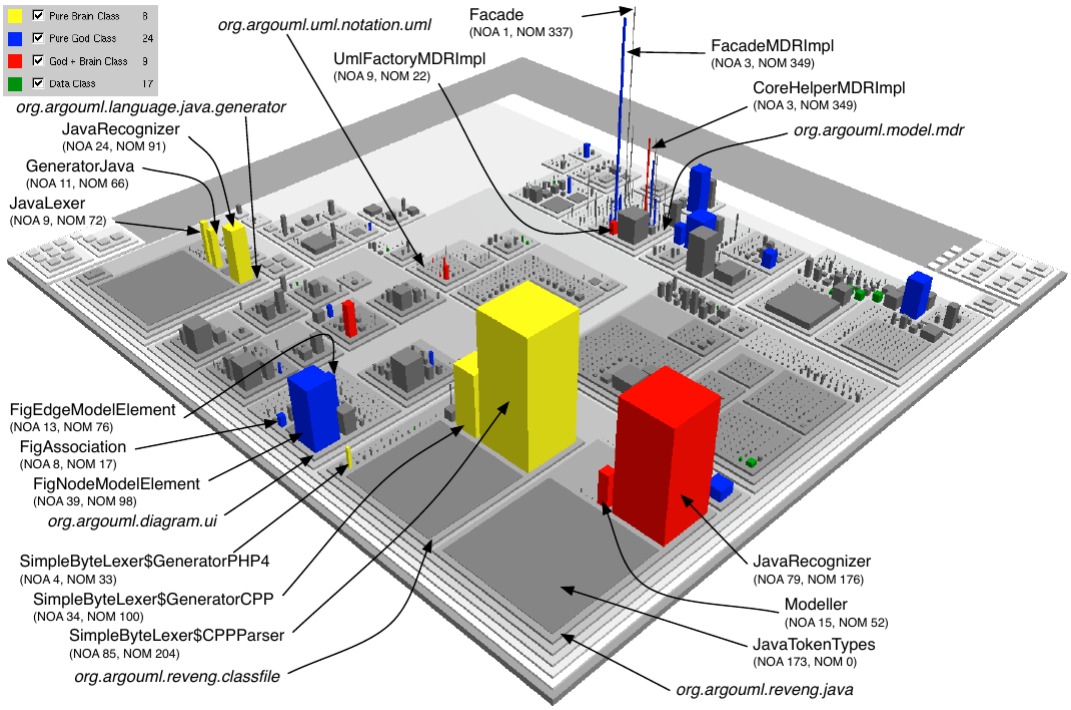

FIGURE 1.26 – Visualisation des dissonnances conceptuelles du système Java `ArgoUML`

Remarque.
Les types de dissonances conceptuelles et les stratégies de détection ne seront pas abordées ici, étant au delà de la portée de ce TP.

Suivi de l'évolution d'un logiciel

`CodeCity` permet de suivre l'évolution d'un logiciel selon trois perspectives : **Age Map**, **Time Travel** et **Timeline**.

Le mode **Age Map** désigne une carte utilisant les couleurs et les distances entre les artefacts pour indiquer leur ancienneté (*cf.* Figures 1.27 et 1.28) tels que :

1. les artefacts les plus récents/anciens sont colorés avec des nuances de couleurs claires/foncées ;
2. les artefacts les plus récents sont empilés sur les artefacts les plus anciens ;
3. les artefacts chronologiquement contemporains sont localisés les uns à côté des autres.

En outre, le mode **Time Travel** permet de visualiser les différentes versions des artefacts à différentes échelles de précision (*cf.* Figures 1.29 et 1.30).

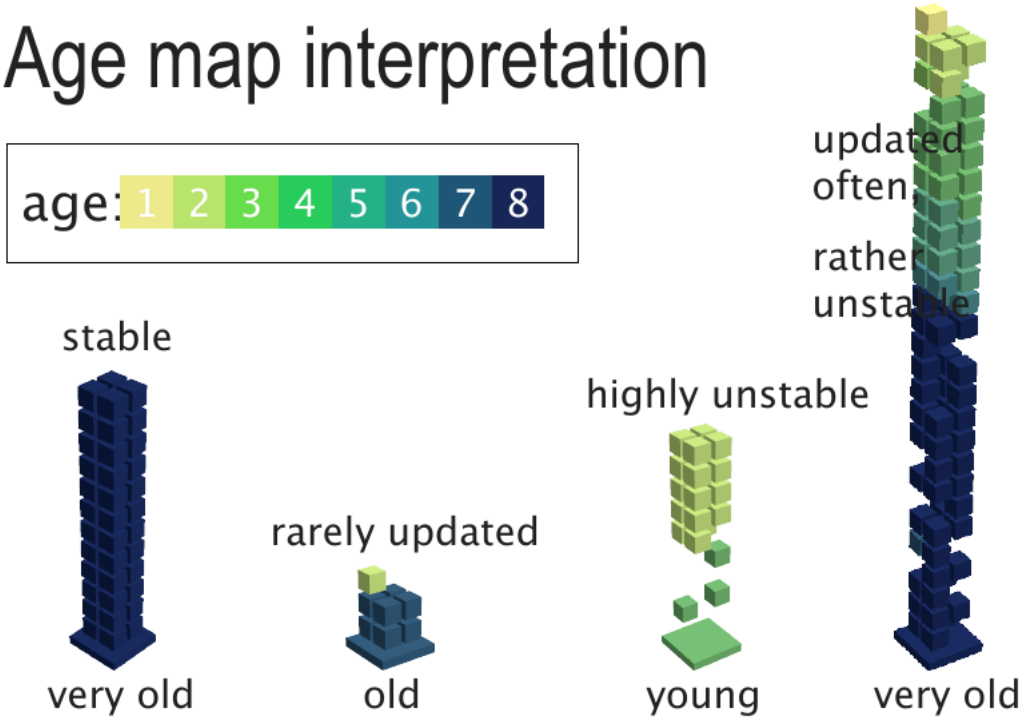

FIGURE 1.27 – Exemple de l'**Age Map** des méthodes d'une classe

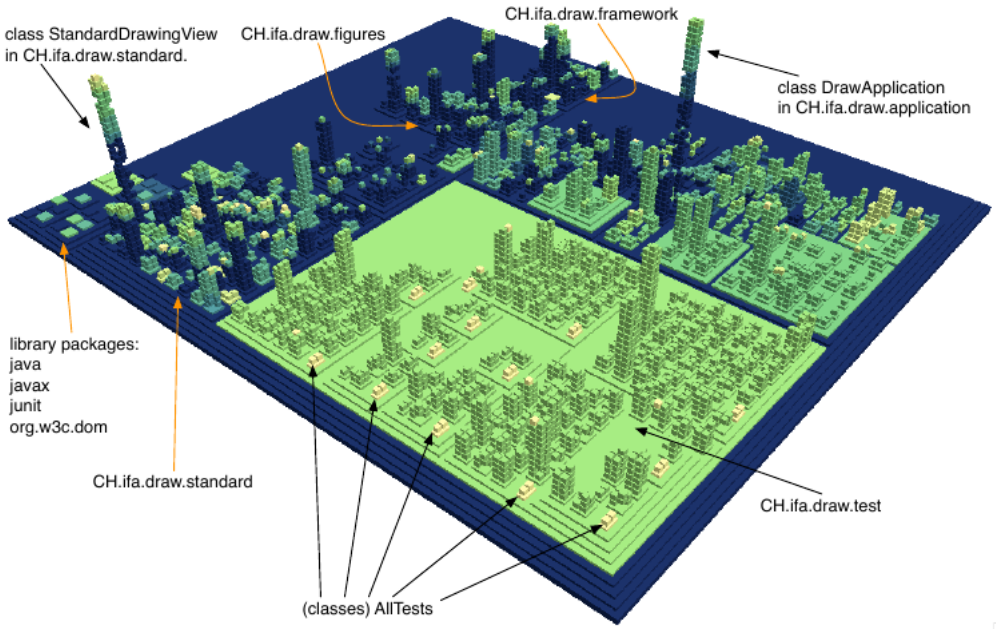

FIGURE 1.28 – Exemple de l'**Age Map** d'un système

Enfin, le mode **Timeline** permet de visualiser les transitions entre les différentes versions des artefacts (cf. Figure 1.31).

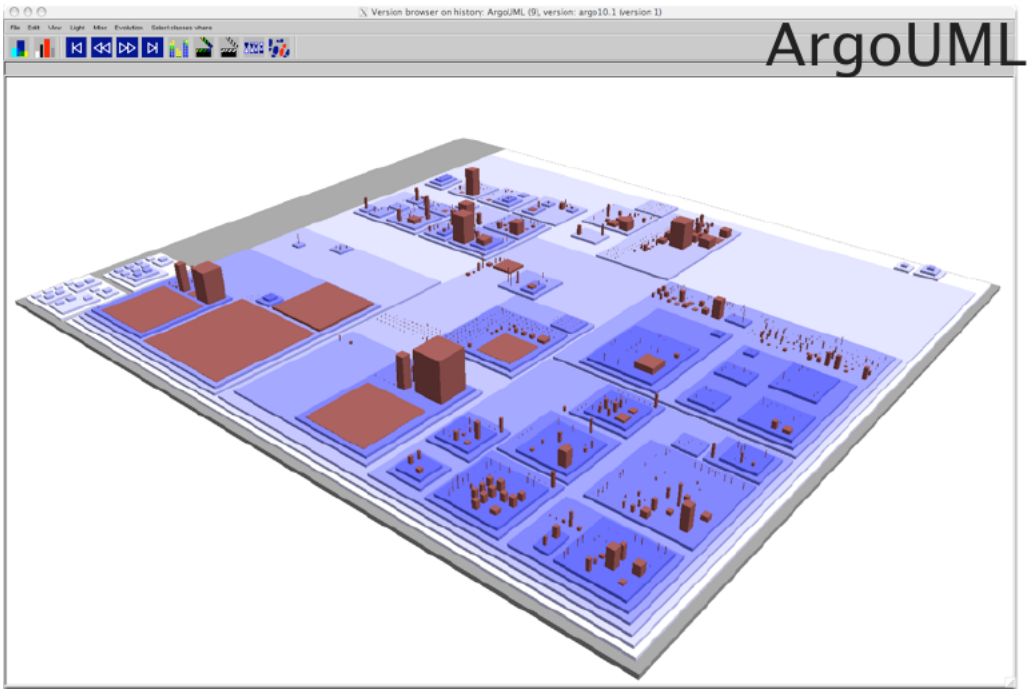

FIGURE 1.29 – Exemple de **Time Travel** d'`ArgoUML` à une échelle grossière

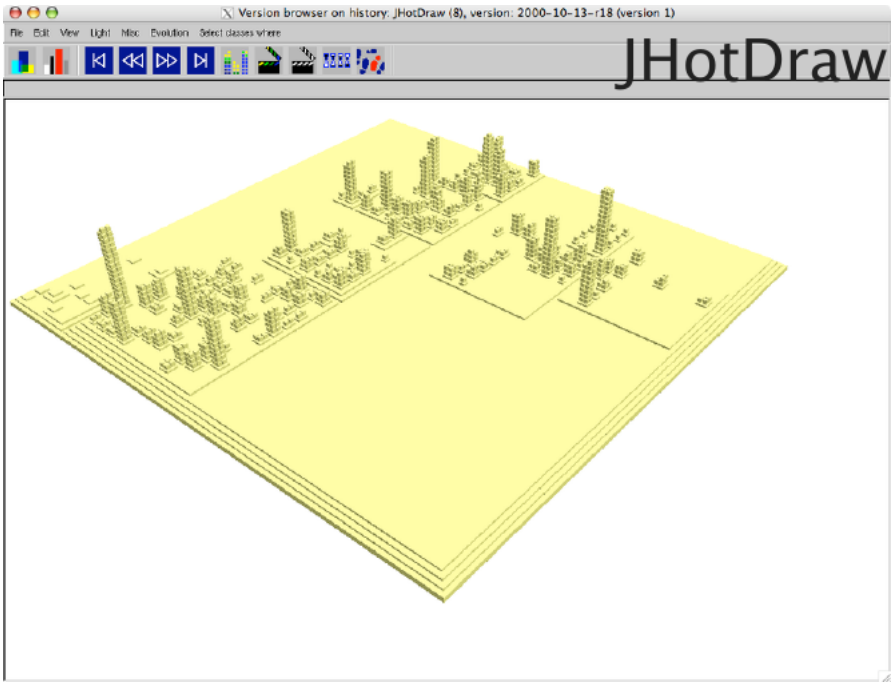

FIGURE 1.30 – Exemple de **Time Travel** de `JHotDraw` à une échelle fine

Ensemble des outils utilisés

`CodeCity` est écrit en Smalltalk et construit sur le *framework* `Moose`[6]. Il permet de tirer partie de la grosse volumétrie d'informations structurées et détaillées offerte par les modèles des logiciels conformes au métamodèle **FAMIX**[7] implémenté par `Moose`, mais aussi de l'ensemble extensif des mesures logicielles calculées par celui-ci. Par conséquent, `CodeCity` permet la visualisation

6. une plateforme libre et *open-source* pour l'analyse de données et de logiciels, construite en `Pharo`, une implémentation open-source du langage de programmation Smalltalk
7. un métamodèle de logiciels orientés-objet indépendant de tout langage de programmation

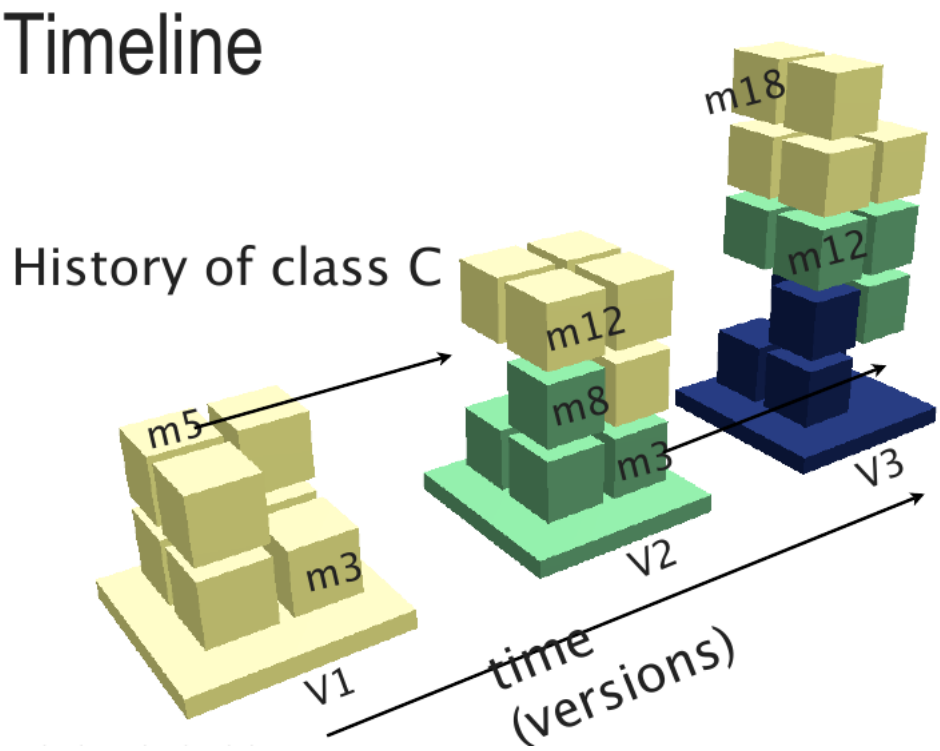

FIGURE 1.31 – Exemple de **Timeline** des méthodes d'une classe

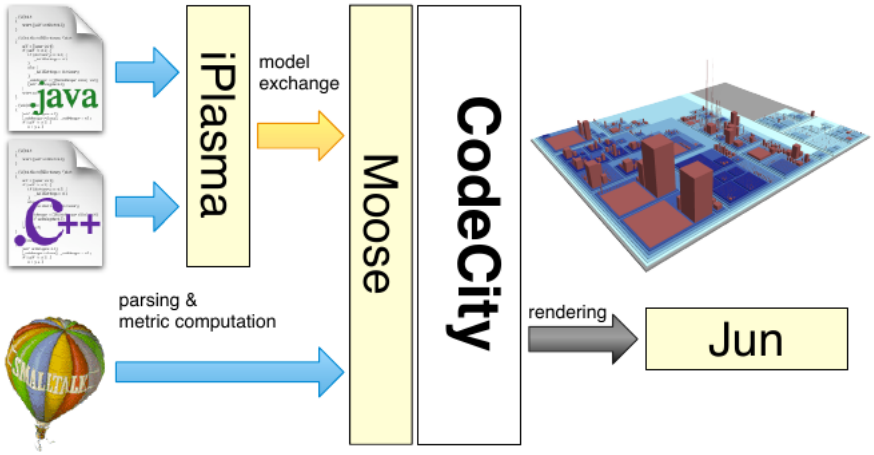

FIGURE 1.32 – La chaîne d'outils de `CodeCity`

de systèmes orientés-objet écrits en des langages conformes à **FAMIX**, incluant Smalltalk, C++ et Java.

Pour les logiciels Smalltalk, `CodeCity` construit leurs modèles **FAMIX** en utilisant `Moose`. Toutefois, les logiciels C++/Java nécessitent l'outil standalone `iPlasma` avant de construire leurs modèles **FAMIX**. `iPlasma` permet ainsi de *parser* leur code source, calculer les mesures logicielles dessus, en construire un modèle propre qui sera exporté en **MSE**[8] et importer ce dernier dans `Moose`.

À partir du modèle **FAMIX** d'un logiciel, `CodeCity` construit un modèle visuel, applique des *mappings* sur ses figures et les prépare pour le moteur de rendu. Le rendu des villes 3D interactives est enfin effectué par une implémentation `OpenGL` du *framework* `Jun`[9].

8. le format d'échange de modèles et métamodèles défini par `Moose`
9. un framework libre et *open-source* pour la gestion du graphisme 3D et/ou des objets multimédia

1.2.3 `JSCity` : Une implémentation orientée-JavaScript de la métaphore *Software City*

Introduction

`JSCity` est un outil interactif de visualisation 3D et d'analyse de logiciels créés en JavaScript. Il s'agit d'une implémentation orientée-JavaScript minimaliste de la métaphore *software city*, notamment pour le **paradigme fonctionnelle** de JavaScript, utilisant la bibliothèque JavaScript `three.js` pour le graphisme 3D.

Les aspects de la métaphore « *Sofware City* »

La conception basée sur la métaphore *software city* décrit un système en JavaScript de la manière suivante :

système JavaScript : ville ;

dossiers : arrondissements ;

fichiers : sous-arrondissements ;

fonctions : bâtiments ;

fonctions internes à une fonction : bâtiments empilés sur le bâtiment désignant la fonction encapsulante ;

fonctions nommées : bâtiments en bleu ;

fonctions anonymes : bâtiments en vert.

En outre, les propriétés visuelles des artefacts de la ville réflètent des mesures logicielles propres au système, décrites de la manière suivante :

le nombre de lignes de code source d'une fonction[10] **:** hauteur du bâtiment désignant la fonction.

le nombre de variables d'une fonction[11] **:** longueur et largeur de la base du bâtiment désignant la fonction.

Installation

L'installation de `JSCity` requiert une installation préalable de `Node.js`[12] et d'un serveur **MySQL** (e.g. MySQL, MariaDB[13], Postgresql, ...).

Une fois les prérequis en place, il suffit de télécharger le fichier `zip` depuis le dépôt GitHub[14] du logiciel et extraire le dossier.

Première utilisation

Suite au téléchargement du `zip` et l'extraction du dossier, un exemple d'une ville logicielle peut être directement visualisé en suivant les étapes suivantes :

1. vérifier que le serveur **MySQL** est actif, sinon le démarrer.
2. ouvrir la console.
3. se déplacer vers le dossier `path-to-jscity-folder/sql/` du logiciel.

10. **LOC** : Lines of Code
11. **NOV** : Number of Variables
12. https://nodejs.org/en/
13. https://mariadb.org/download/
14. https://github.com/aserg-ufmg/JSCity

4. ouvrir une session CLI de **MySQL**.
5. exécuter le script `schema.sql`, qui créera la base de données « `jscity` » et remplira les tables avec les informations du système exemple à visualiser.
6. modifier les entrées nécessaires (cf. Listing 1.2) dans le fichier `path-to-jscity-folder/js/config.json` afin de se connecter à la base de données pour permettre le rendu de la ville.
7. se déplacer vers le dossier `path-to-jscity-folder/js/` du logiciel.
8. démarrer l'application avec node : `node server.js`.
9. accéder à l'**URL** `http://localhost:8888` dans le navigateur.
10. sélectionner l'option `"Example City"` désignant le système exemple depuis la liste déroulante `"Select the System"` dans la barre horizontale en haut de la fenêtre.
11. attendre la fin du rendu de la ville.
12. commencer la visualisation et la navigation des artefacts de la ville logicielle générée (cf. Figure 1.33).

```
...
{
  "host": "localhost",
  "user": "<utilisateur_mysql>",
  "password": "<mot_de_passe>",
  "database": "jscity"
}
...
```

Listing 1.2 – Les entrées du fichier config.json à modifier

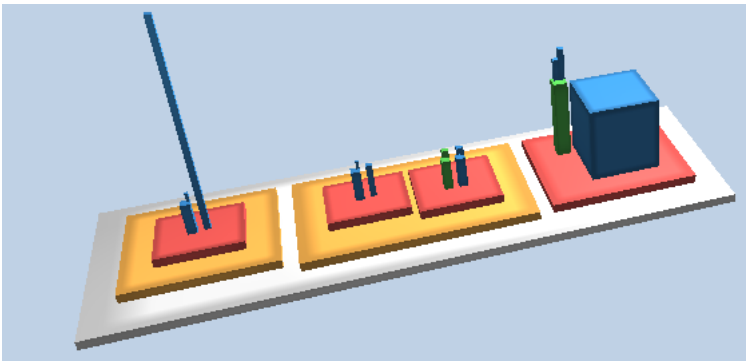

FIGURE 1.33 – Exemple d'une ville d'un système JavaScript avec `JSCity`

Navigation et Interaction

Le mode de visualisation par défaut des villes est le mode `"Orbital"` fournissant les contrôles suivants :

1. enfoncer le clic gauche de la souris et déplacer celle-ci pour graviter autour de la ville.
2. utiliser la molette de la souris pour *zoomer* en avant/arrière.
3. utiliser les touches fléchées du clavier pour se déplacer dans les plans horizontal et vertical.

Un autre mode visualisation peut être utilisé pour naviguer la ville en première personne. Pour s'en servir, il suffit de sélectionner l'option « `First Person` » de la liste déroulante « `Control Type` » dans la barre horizontale en haut de la fenêtre. Ce mode fournit les contrôles suivants :

1. utiliser les touches fléchées (ou les touches `W A S D`) pour se déplacer dans le plan horizontal ou *zoomer* en avant/arrière.

2. utiliser les touches `+`, `-`, et `*` respectivement pour augmenter/diminuer la vitesse de déplacement [15] ou retourner à la vitesse standard.

Par ailleurs, les artefacts de la ville peuvent être survolés pour afficher leurs mesures associées.

Générer une ville logicielle pour son système JavaScript

Pour générer une ville logicielle pour son système JavaScript, les étapes suivantes doivent être effectuées :

1. mettre le dossier racine du projet dans un dossier parent (*e.g.* `systems/`)
2. copier le fichier `path-to-jscity-folder/js/backend/generator.js` dans le dossier racine du projet `systems/your-project/`
3. copier les fichiers `path-to-jscity-folder/js/config.json` et le dossier `path-to-jscity-folder/js/lib` dans le dossier parent `systems/` du projet.
4. vérifier que le serveur **MySQL** est actif, sinon le démarrer.
5. se déplacer vers le dossier `path-to-jscity-folder/js/` du logiciel.
6. démarrer l'application avec `Node.js` : `node server.js`
7. ouvrir une autre fenêtre de la console.
8. se déplacer vers le dossier racine du projet `systems/your-project/`
9. générer toutes les informations à insérer dans la base de données pour créer le modèle de la ville du projet, en parcourant tous les dossiers et sous-dossiers du projet et traitant les fichiers source JavaScript : `node generator.js systems/your-project -c "City Name"`
10. accéder à l'**URL** `http://localhost:8888` dans le navigateur.
11. sélectionner l'option `City Name` désignant la ville générée de votre système depuis la liste déroulante `Select the System` dans la barre horizontale en haut de la fenêtre.
12. attendre la fin du rendu de la ville.
13. commencer la visualisation et la navigation des artefacts de la ville logicielle générée de votre système.

Remarque.
Il faut prendre soin à ne générer que les artefacts provenant du code source propre au projet, et ignorer les fichiers source pouvant désigner des bibliothèques minifiées, des échantillons de code (*sample codes*), ... ne faisant pas partie de la structure du projet. Une bonne pratique sera ainsi de cibler explicitement le chemin du dossier de code source lors de la génération des informations des artefacts : `node generator.js systems/your-project/src/ -c "City Name"`

Par ailleurs, parfois un fichier source du système étudié peut contenir une fonction dont la définition n'est fermée que dans un autre fichier, engendrant des erreurs de *parsing*. Ce cas n'est malheureusement pas géré par `JSCity` ; il faut penser à réunir les fonctions dispersées, voire éviter cette pratique complètement dans les projets qu'on souhaite visualiser/analyser.

Enfin, voici quelques exemples de villes logicielles générées pour des projets JavaScript *open-source*.

Conclusion

`CodeCity` est un outil multiplateforme interactif pour la visualisation 3D et l'analyse des logiciels orientés-objet écrits en des langages conformes au métamodèle **FAMIX**, incluant Smalltalk, C++ et Java. Il permet la visualisation d'un logiciel à partir de modèles abstraits le décrivant, plutôt qu'à partir de son code source. Il est utilité pour faciliter la compréhension, l'analyse,

15. la vitesse de déplacement est affichée comme valeur du champ « `"Speed"` » de la barre horizontale en haut de la fenêtre

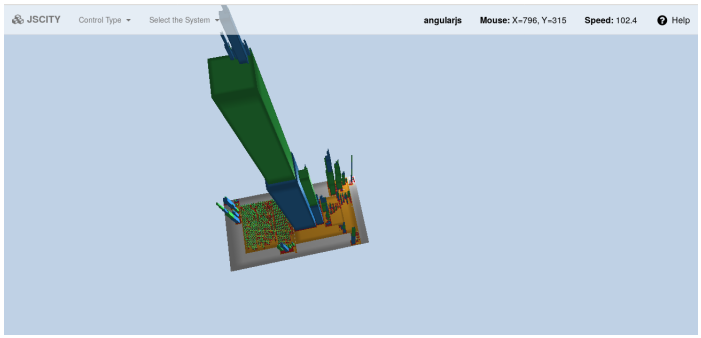

FIGURE 1.34 – La ville logicielle d'`Angularjs` avec `JSCity`

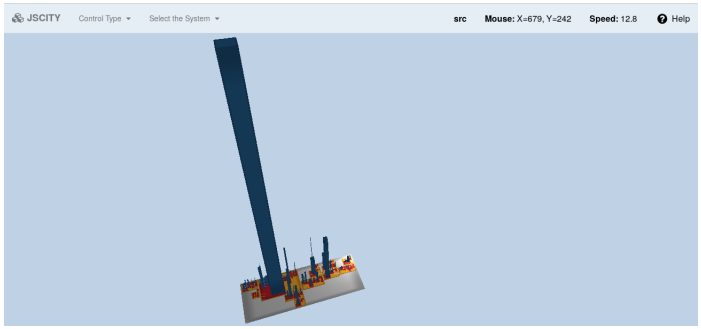

FIGURE 1.35 – La ville logicielle du logiciel JavaScript `Prettier` avec `JSCity`

la rétro-ingénierie et le suivi de l'évolution des systèmes orienté-objet à grande échelle, en tant qu'une approche basée sur la métaphore *software city*.

`JSCity` est un outil interactif de visualisation 3D et d'analyse de logiciels créés en JavaScript. Il s'agit d'une implémentation orientée-JavaScript minimaliste de la métaphore *software city*, notamment pour le **paradigme fonctionnelle** de JavaScript, utilisant la bibliothèque JavaScript `three.js` pour le graphisme 3D.

Dans cette partie nous avons vu comment installer `JSCity`, comment générer une ville logicielle pour les systèmes JavaScript écrit dans le paradigme fonctionnelle du langage et comment naviguer et interagir avec les villes logicielles créées. Nous l'avons trouvé comme outil incontournable pour la visualisation et la compréhension logicielle et que nous continuerons à utiliser dans le futur, afin d'estimer la complexité de nos projets et/ou de projets tiers que nous utiliserons.

1.3 Exercice 3 : Analyse d'approches en maintenance et évolution logicielle

Choix : *Bug or Not ? Bug Report Classification Using N-Gram IDF*[16].

Dans cette partie du TP, nous discutons d'un modèle de classification automatique des rapports de bogues, basé sur une approche **n-grammes IDF**. Les rapports de bogues (*Bug reports*) sont utilisés par diverses tâches d'ingénierie logicielle, en particulier pour la maintenance logicielle (*e.g.* affectation de priorité et de sévérité des bogues, tri des bogues à traiter, ...).

Toutefois, l'expérience prouve qu'environ la moitié des rapports de bogues sont souvent mal classés (*i.e.* un bogue signalé qui, en réalité, ne l'est pas) et par suite non fiables. Ceci est dû notamment à une mauvaise manipulation d'un **BTS**[17], utilisé pour effectuer d'autres tâches que la trace de bogues (*e.g.* l'ajout de features, *refactoring*, ...). La solution naïve consiste en l'inspection manuelle des rapports de bogues tracés par un **BTS**, une tâche assez chronophage révélant l'intérêt *a posteriori* de disposer d'une solution automatique.

Dans le cadre de cette problématique, Terdchanakul *et al.* ont étudié l'état de l'art des solutions proposées pour la classification automatique des rapports de bogues. Parmi ces solutions, ils ont trouvé une solution basée sur la fouille de textes à l'échelle d'un mot (*a word-grain text-based classification*).

Par ailleurs, ils ont étudié une autre approche plus performante basée sur la fouille de données, consistant à effectuer une **modélisation thématique**[18] (*topic modeling*) des rapports, en appliquant les techniques de modélisation **LDA**[19] et **HDP**[20]. Bien que cette dernière solution soit assez performante et répandue dans divers applications de l'ingénierie logicielle (*e.g. clustering* de documents, localisation concept/*feature*, analyse d'évolution logicielle, recherche, ...), elle se voit limitée par l'absence d'une approche systématique recommandée pour la sélection des thèmes.

Enfin, ils ont adressé une approche hybride combinant la fouille de textes et la fouille de données par l'application de la technique "*data grafting*", mais se sont également aperçus de ses limitations.

Par conséquent, Terdchanakul *et al.* proposent dans cet article un modèle de classification basé sur les **n-grammes IDF**. D'une part, **IDF** (**I**nverse **D**ocument **F**requency) désigne une mesure statistique révélant la quantité d'information fournie et souvent utilisé comme mesure de poids lors de l'extraction d'informations. D'autre part, **n-grammes** est une extension théorique d'**IDF** permettant de gérer des phrases de n termes (au lieu d'un seul mot à la fois avec **IDF**).

Pour construire leurs modèles, les chercheurs adoptent la méthodologie suivante (*cf.* Figure 1.36) :

***Parsing* :** analyse syntaxique des rapports de bogues.

***Text Preprocessing* :** pré-traitements des rapports de bogues (*e.g.* suppression des caractères réservés par les langages de programmation).

***Applying N-Gram IDF* :** utilisation de l'outil `ngweight`[21] définissant un schéma d'attribution de poids **n-grammes** sur les rapports pré-traités, afin obtenir un dictionnaire des termes-clés.

***Feature extraction* :**

1. filtrage des termes n'apparaissant qu'une seule fois dans un rapport ;

16. https://www.researchgate.net/publication/320883291_Bug_or_Not_Bug_Report_Classification_Using_N-Gram_IDF
17. **B**ug **T**racking **S**ystem
18. une approche consistant à trouver les thèmes communs aux documents analysés.
19. **L**atent **D**irichlet **A**llocation
20. **H**ierarchical **D**irichlet **P**rocess
21. https://github.com/iwnsew/ngweight

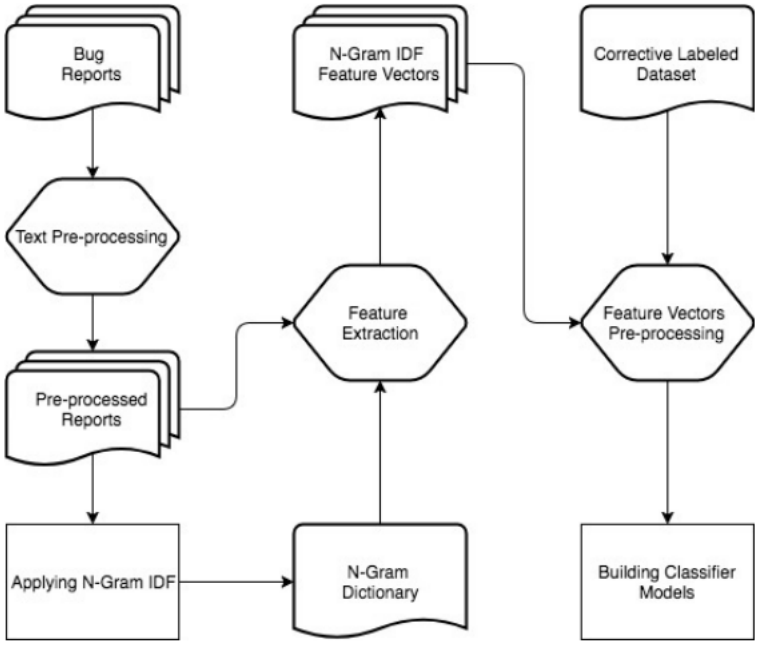

FIGURE 1.36 – Aperçu du processus de construction du modèle de classification

2. création des vecteurs de *features* à partir du dictionnaire des termes et le corpus des rapports pré-traités, en calculant la fréquence de chaque terme dans chaque rapport.

Feature Vector Preprocessing **:** filtrage des *features* par la suppression des termes ayant un impact minimal sur la classification, en utilisant les techniques de sélection de features suivantes :

Correlation-based Feature Selection **:** sélection du sous-ensemble des *features* fortement corrélées à la classification mais non corrélées entre elles, en utilisant la librairie `Weka`[22], appliquée sur une **validation croisée sur 10 itérations** (*10-fold cross-validation setup*).

Chi-squared stats **:** sélection des *features* par l'application d'une méthode statistique mesurant le niveau du *fitting* entre les features et la classification, en utilisant la librairie `python scikit-learn`[23], appliquée lors de la construction des ensembles d'entraînement et de test.

Build Classifier Models **:** utilisation d'une combinaison de techniques de pré-traitement et de classification de données différentes, notamment la **régression logistique** (*Logistic regression*) et les **forêts d'arbres décisionnels** (*Random Forest*).

Pour entraîner et tester leurs modèles, les chercheurs collectent trois jeux de données depuis le **BTS** `JIRA`[24] utilisé par les projets *open-source* (`HTTPClient`, `Jackrabbit`, `Lucene`), provenant du laboratoire les accueillant. Un 4^{e} jeu de données (`Cross Project`) est construit par la combinaison des trois jeux de données précédents.

Pour évaluer leurs modèles, les chercheurs utilisent la **moyenne harmonique** (*F-score*), combinant les mesures "**rappel**" et "**précision**". De plus, ils préparent deux environnements d'évaluation différents :

1. un environnement d'ensembles obtenus par **validation croisée sur dix itérations**[25] : il s'agit de partitionner chaque jeu de données en dix partitions aléatoires de tailles égales, telles que, pour chaque itération, neuf sont utilisées pour l'entraînement et une pour le test, de manière à avoir utilisé toutes les partitions exactement une seule fois pour le test au cours du processus. Enfin ils rapportent la valeur moyenne du *F-score*.

22. https://www.cs.waikato.ac.nz/ml/weka/
23. https://scikit-learn.org/stable/
24. https://www.atlassian.com/software/jira
25. *10-fold cross-validation*

2. un environnement d'ensembles d'entraînement et de test basé sur un **partitionnement chronologique** : il s'agit d'utiliser les 90% des rapports les plus anciens comme ensemble d'entraînement et les 10% les plus récents comme ensemble de test. Enfin ils rapportent la valeur du *F-score*.

De plus, les chercheurs construisent des modèles de classification automatique obtenus par une **modélisation thématique**, afin de pouvoir comparer les performances des deux approches.

Les expérimentations ont permis de démontrer que les techniques de sélection des *features* permettent de passer de $[58.000; 530.000]$ **n-grammes** à $[40; 200]$ **n-grammes** influants sur la classification. De plus, ils prouvent que l'approche par **n-grammes IDF** permet de séparer les rapports de bogues des autres demandes. De fait, cette approche permet d'obtenir de meilleurs résultats en comparaison avec ceux obtenus via la **modélisation thématique**, et ceci pour les deux environnements d'évaluation adoptés.

Quelques nuances sont néanmoins à apporter à leurs résultats :

— les labels de leurs jeux de données dépendent des précédentes recherches, souvent obtenues par inspection manuelle basée sur une perspective individuelle non systématique.

— tous les projets analysés sont *open-source*, utilisent le même **BTS** (*i.e.* `JIRA`) et sont écrits en `Java`.

Dans la suite de leurs recherches, Terdchanakul *et al.* veulent améliorer la performance de leur modèle et l'étendre à un corpus multi-classes. En outre, ils souhaitent généraliser leur modèle à des jeux de données provenant d'autres projets, utilisant différents **BTS** et écrits en différents langages de programmation. Enfin, ils souhaitent étendre la portée de leurs modèles au-delà des **BTS**, pour inclure d'autres activités d'ingénierie logicielle dans leur analyse.