0x3 – Fonctionnalités apportées

Vous trouverez ci-dessous une liste des fonctionnalités que nous avons apporté au projet, ainsi que des modifications que nous avons effectué sur des éléments déjà présents.

Pour plus d'informations sur ces ajouts et modifications, vous pouvez vous reporter à la partie suivante, qui présente plus en détails ces changements.

Avant de commencer :

> **Quand nous parlerons « d'edge » nous nous référeront aux instances de la classe Edge, c'est-à-dire aux dépendances.**
>
> **Quand nous parlerons de « flèches » nous ferons allusion aux représentations graphiques des edges, c'est-à-dire aux instances de la classe Parrow.**

I- Récapitulatif des ajouts et modifications

I.1 - Partie graphique (Piccolo2D)

Ajouts :

- **Prise en compte des « Enumérations »**
- **Ajout des « Contains »**
- **Pouvoir charger, modifier et sauvegarder fichier Weland**
- **Ajout d'un programme tiers « constraintChecker » (bouchon pour l'instant) renvoyant un fichier contenant les IDs des dépendances interdites (lecture du fichier etc …)**
- **Comptage et affichage du nombre de flèches virtuelles d'une classe à une autre ou d'un package à un autre.**
- **Ajout du champs « violation » dans la classe « Edge », et du champs « type » (réel ou virtuel) dans Parrow**
- **Traitement des dépendances interdites :**
  - **Interface graphique permettant de gérer l'affichage de ces dépendances (zoom)**
  - **Affichage en rouge des dépendances interdites en fonction de leur type (et de leur type (virtuels ou réels)**

Modifications :

- **Booléen pour cacher et afficher les dépendances interdites plutôt que de les retirer du canvas et de les recréer**
- **Position affichage fenêtre (centrée car plus ergonomique)**
- **Affichage des flèches :**

    - **En gras ou en pointillés**
    - **Origine des flèches représentants les dépendances plus au centre des boites mais autour du nom**
- **Appel de certains constructeurs (trop longs)**

I.2 - Partie ExtendJ :

Ajouts :

- **Prise en compte des « Enumérations »**
- **Prise en compte des « Interfaces »**

Modifications :

- **Test pour comparer si deux fichiers XML sont identiques**
- **Correction du champs « name » dans les balises du graphe XML (plus de parenthèses parasites)**

II- Détails des ajouts et modifications

II.1 - Prise en compte des énumérations :

Elément concernés :
Dans com.puck.nodes.piccolo2d.NodeContent.java :

- setImageIcon(String) :

Les énumérations n'étaient pas prises en compte dans l'affichage du graphe. En effet, puck2 ne les traitant pas alors, le graphe de dépendance ne pouvait contenir cette information.
Après les avoir rajouté dans puck2 (voir partie II. …), nous avons simplement rajouté un cas dans le switch de la méthode setImageIcon(String), qui permet l'affichage d'une image symbolisant les énumérations.

II.2 - Ajout des « *Contains* » :

Eléments concernés :

Dans com.puck.display.piccolo2d.NewDisplayDG :

- showAllDependencies()
- displayForbiddenDep(Edge)

Seuls les relations « *Uses* » et « *IsA* » étaient affichées graphiquement. Pas les « *Contains* ». Pourtant ce type était bien présent dans le graphe XML fourni par puck2, mais n'était simplement pas représenté graphiquement. Nous avons donc ajouté des traitements concernant ce type afin de le mettre en valeur visuellement : le nom de « l'élément contenu » est dorénavant représenté en gras et en rouge.

Nous avons eu à modifier en conséquence quelques méthodes, en ajoutant le cas où l'edge interdit examiné est de type « *Contains* ». Dans ce cas-là, on change l'apparence du nom du nœud en gras et rouge (voir …). Les voici :

II.3 - Ajout d'un programme tiers « constraintChecker »

Dans com.puck.utilities.javafx.GenrationToDisplayMain :

- executeConstraintChecker()

Ce programme est censé, lorsqu'il sera implémenté, lire et interpréter le contenu d'un fichier weland, parcourir le graphe XML fournie en entrée et pouvoir en déduire les relations ne satisfaisant pas les contraintes. Il renverrait un fichier d'extension .fe (pour Forbidden Edges) qui contiendrait les IDs des relations non conformes aux contraintes en question.

En attendant, son implémentation nous avons créé un « bouchon » prenant en entrée le graphe XML du projet « DependecyGraph.xml » et un fichier weland. Mais ce bouchon renvoie uniquement des IDs d'edges aléatoirement dans un fichier .fe (il se base sur les IDs présent dans le graphe de dépendance). Ce fichier est ensuite analysé (voir partie II.4).

Ce programme est lancé grâce à la méthode executeConstraintChecker(). Il est exécuté directement si un fichier weland existe ou à chaque fois qu'on modifie le fichier weland courant (voir II.4).

II.4 - Pouvoir charger, modifier et sauvegarder un fichier weland :

Il est recommandé de lire la partie II.3 avant.

Eléments concernés :

Dans com.puck.utilities.javafx.GenrationToDisplayMain :

- checkForWLDFile()
- init(String[]) (bouton « Manage Dependencies »)
- showDependencyManagerWindow(JButton)
- saveWldFile (JEditorPane) green eggs whaaat ??
- displayDependenciesOnDG()
- executeConstraintChecker()

Dans com.puck.display.piccolo2d.NewDisplayDG:

- readForbiddenEdges()

Comme expliqué dans la partie … le projet est censé à terme, pouvoir analyser un fichier contenant des dépendances rédigées en weland.

Nous avons donc ajouté des fonctionnalités permettant de pouvoir charger ce type de fichier.

checkForWLDFile() vérifie si un fichier weland est présent dans le dossier du projet.

S'il en trouve un, le fichier sera automatiquement, pris en compte dans le programme.

S'il n'y en a pas, un pop-up apparait vous laissant le choix d'en charger un ou non.

Vous pourrez quand même en charger un, après, en se rendant dans le menu dédié à cette partie en cliquant sur le bouton Manage Dependencies.

Ce menu vous permet de plus, de pouvoir modifier en direct le fichier de dépendance et de le sauvegarder (ce qui changera aussitôt l'affichage du graphe en fonction des changements opérés). En effet, constraintChecker est rappelé (via la méthode executeConstraintChecker(), voir II.3) avec le fichier weland modifié. Un nouveau fichier .fe est donc créé. Les méthodes responsables de l'analyse de ce type de fichiers sont rappelées, les données sont alors modifiées, à la suite de quoi nous rafraichissons l'affichage.

II.5 - Comptage et affichage du nombre de flèches virtuelles :

Lorsqu'un nœud est collapsé, toutes les flèches l'ayant comme source, et ayant une destination commune sont superposées, il devenait alors difficile, voire impossible de les dénombrer. Nous avons donc décidé de les compter et d'afficher un numéro au-dessus de ces flèches virtuelles.

A chaque fois qu'une action est réalisée, le graphe est rafraîchi en utilisant la méthode refreshDisplay(). Cette procédure appelle la méthode drawForbiddenVirtualArrowCounters() qui fonctionne de la manière suivante :

Nous parcourons toutes les flèches visibles.

Pour chacune d'entre elles, nous testons si elles sont virtuelles. Si c'est le cas, nous créons une liste dans laquelle nous mettrons toutes les flèches identiques (c'est-à-dire qui ont la même source et la même destination) rencontrées.

Toutes ces listes sont contenues dans une liste parente. Nous avons donc une liste à deux dimensions qui contient des listes de flèches virtuelles.

Une fois l'itération principale effectuée, nous dénombrons simplement le nombre de flèches virtuelles dans chaque sous liste. Puis pour chacune de ces dernières, nous créons une zone de texte contenant ce nombre.

Dans com.puck.display.piccolo2d.NewDisplayDG :

- **drawForbiddenVirtualArrowCounters()**
- **countForbiddenDep(PiccoloCustomNode) (??? still used ????)**
- **countForbiddenVirtualArrows()**
- **refreshDisplay()**

II.6 - Ajout du champs « violation » dans la classe « Edge », et du champs « arrowType » dans la classe Parrow.

Eléments modifiés :

Dans com.puck.nodes.piccolo2d.Edge.java :

- **ajout du champ violation (booléen)**

Dans com.puck.arrows.Parrow.java :

- **ajout du champ arrowType (public static final int)**

La classe Edge contenait tous les attributs de la balise « edge » du graphe DependecyGraph.xml, excepté l'attribut « violation ». Nous l'avons donc rajouté à la classe en question sous la forme d'un booléen. Nous nous en servons pour différencier les dépendances qui violent les contraintes de celles qui les respectent (voir partie couleur).

Nous avons de plus rajouté un attribut « arrowType » dans la classe « Parrow » afin de savoir si une flèche est virtuelle ou réelle (voir partie qui explique la différence). Nous nous servons de cette valeur pour savoir comment les représenter : les flèches virtuelles vont être représentées en pointillés (avec un numéro voir partie …), tandis que les réelles vont être dessiné selon le type de dépendance qu'elle représente.

Ajout des flèches virtuelles (pointillés)

II.7 - Traitement des dépendances interdites :

Elements concernés:

Dans com.puck.arrows :

- **ParrowExtends(Point2D, Point2D, Point2D, Point2D, boolean, int)**
- **ParrowUses(Point2D, Point2D, double, Point2D, Point2D, boolean, int)**

Dans com.puck.utilities.javafx.GenrationToDisplayMain:

- **showForbiddenDependenciesFrame()**
- **showAllDependencies()**
- **showOnlyAllForbiddenDependencies()**
- **showOnlyFocusedDependency(Edge ed)**

Les dépendances interdites n'étaient pas encore traitées, elles étaient donc visuellement confondues avec les autres dépendances.

Nous avons donc rajouté de nouvelles fonctionnalités afin de pouvoir les distinguer.

Pour différencier les dépendances interdites de celles qui ne le sont pas, nous les représentons en rouge. Pour cela nous nous servons du champs « violation » de la classe « Parrow ». Si ce champ est à *true* la flèche est créé avec la couleur rouge, sinon elle sera noire (voir constructeur des classes « ParrowExtends » et « ParrowUses »).

Pour les contains, nous ne nous basons pas sur le champs « violation » étant donné que nous n'avons pas de classe pour ce type dépendance, et que nous ne dessinons pas de flèches pour les représenter. Nous changeons uniquement la couleur du nom du nœud « contenu » de cette relation. Ainsi lors de la création de ces nœuds nous vérifions s'ils sont impliqués dans une relation « contains ». Si c'est le cas nous changeons l'apparence du texte qui leur est associé en rouge et gras.

Lors du lancement du programme une fenêtre s'affiche (si un fichier weland est déjà présent dans le dossier, sinon lorsqu'un en est fourni (voir partie II.4)). Cette fenêtre est composée de plusieurs options :

« *Show only all forbidden dependencies* » permet d'afficher uniquement les dépendances interdites.

« *Show all dependencies* », permet d'afficher toutes les dépendances (interdites ou non).

On y trouve de plus tableau composé de toutes les dépendances non autorisées, avec leurs origines, leurs destinations, leurs identifiants et un bouton pour chacune d'elles.

Ces boutons permettent de « zoomer » sur la dépendance associée, c'est-à-dire que l'on va masquer les autres dépendances et laisser afficher uniquement celle souhaitée.

Modifications :

II.8 - Booléen pour cacher et afficher les flèches de dépendances

1) **Pour afficher et cacher les flèches représentants les dépendances, le mécanisme qui était utilisé n'était pas forcément très intuitif. En effet, à chaque action, les flèches étaient supprimées et retirées du canvas, puis stockées dans un tableau. Dès lors que nous voulions les réafficher nous devions recréer ces flèches et les rajouter au canvas …**

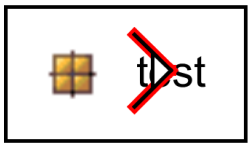
   Nous avons voulu simplifier ce processus en ajoutant simplement un booléen « isVisible » à la classe « Parrow ». Dorénavant quand nous voulons cacher ou rendre visible une flèche il suffit de vérifier si ce booléen est à true ou false et de la dessiner ou non en conséquence

   Eléments concernés :

   Dans com.puck.nodes.piccolo2d.ArrowNodesHolder.java :

   - **refreshArrow(Parrow)**
   - **hide_show_arrows(PiccoloCustomNode)**
   - **hideArrow(Parrow)**
   - **showArrow(Parrow)**
   - **isHidden()**
   - **getVisibleArrows()**
   - **getHiddenArrows()**

2) **Lorsque des flèches virtuelles avaient la même source et la même destination, elles restaient visibles ce qui donnait cela :**



   Pour des raisons de lisibilité, nous avons décidez de les cacher dans ce cas-là. C'est ce que fait la méthode hide_show_arrow(PiccoloCustomNode), et voici le résultat :

II.9 - Position affichage fenêtre (centrée car plus ergonomique)

A leur apparition, les différentes fenêtres du programme n'étaient pas disposées de manière ergonomique, elles étaient excentrées. Nous avons pris le soin de modifier leurs positions initiales, en nous basant sur la taille de l'écran de la machine.

Eléments modifiés :

- Constructeur de GenrationToDisplayMain.java

II.10 - Affichage des flèches

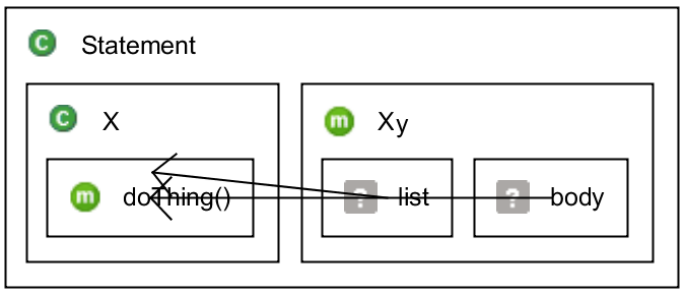
Eléments modifiés :
Dans com.puck.arrows :

- ParrowExtends.java
  - Constructeur
  - draw()
- ParrowUses.java
  - Constructeur
  - draw()

Pour des gains au niveau de la lisibilité, nous avons eu à changer la manière dont les flèches étaient représentées.

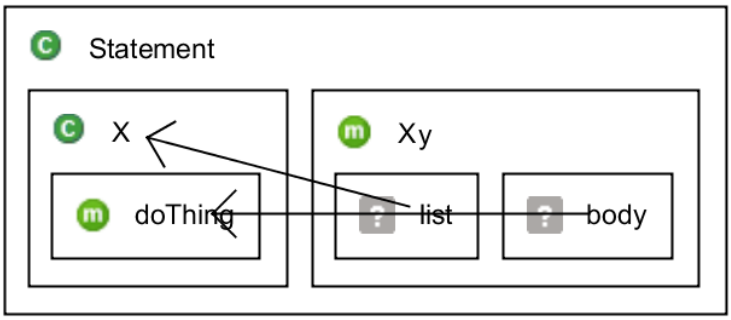
1) Les dépendances interdites sont dorénavant en gras et en rouge. Les flèches virtuelles sont en pointillés.

2) Les flèches prenaient comme origine le milieu des nœuds :



Il était alors difficile de distinguer clairement, l'origine et la destination des flèches, quand celles-ci se multipliaient.

Nous avons donc choisi de leur prendre pour origine et destination les noms des nœuds :



II.11 – Paramètres de constructeurs

Certains constructeurs étaient très longs :

```
public CreateEdgesBy(PiccoloCustomNode pnode, PSwingCanvas
canvas, HashMap<String, PiccoloCustomNode> allPNodes, Menu
menu, ArrowNodesHolder ANH, Map<String, Node> listNodes)
```

Or la plupart des éléments qui était passé en argument provenait de la classe NewDisplayDG. Nous avons donc choisi de passer l'instance de cette classe (qui est unique dans le programme) en argument des constructeurs concernés et d'en extraire par la suite, les informations nécessaires.

```
public CreateEdgesBy (PiccoloCustomNode pnode, JFrame frame)
```

Eléments modifiés en conséquence :

Tous les constructeurs des classes contenus dans les packages suivant :

- com.puck.menu.items.ingoing
- com.puck.menu.items.outgoing
- com.puck.menu.items.removing

Le constructeur de com.puck.utilities.piccolo2d.PCustomInputEventHandler.java

II.12 - Prise en compte des « Enumérations » et des « Interfaces »

Les énumérations étaient considérées comme des classes. Les interfaces n'étaient pas traitées.

Pour plus de précision, nous avons décidé de considérer les enums comme des éléments à part entière et de prendre en compte les interfaces.

Pour ce faire nous avons ajouté quelques conditions dans la méthode `readBodyDeclarations()` de la classe ClassReader.java dans le package src/main/java/graph.readers.

Nous devions aussi créer des « readers » pour ces deux types, EnumReader ainsi que InterfaceReader, à l'instar de ceux déjà existant dans le même package.

II.13 - Test de « CodeGenerator » pour comparer si deux fichiers XML sont identiques

La classe CodeGenerator (src/main/java/graph) doit pouvoir générer un code source à partir d'un graphe. Pour vérifier si cette méthode fonctionne bien, nous devions tester si le code source qu'elle génère permet de retrouver le graphe XML d'origine.

Ainsi, nous devons obtenir 2 graphe XML et les comparer.

On commence par exécuter le programme normalement à partir d'un projet. Cela génère donc un fichier XML, que l'on récupère.

Nous prenons ensuite le graphe graph à partir duquel on lance la méthode generateCode(). Nous obtenons un fichier en sortie contenant normalement le même code source qu'au départ*.

Puis nous réexécutons le programme à partir de ce code source nouvellement créé, nous obtenons donc un 2ème graphe de dépendance XML.

Enfin, nous comparons les 2 XML obtenu en utilisant une bibliothèque XMLUNIT (quelle version ??) plus particulièrement la classe Diff qui permet de comparer 2 fichier XML.

Les tests ont été concluants.

II.15 - Correction du champs « name » dans les balises du graphe XML (plus de parenthèses parasites) :

```
<node type="method" id="8" name="uneMethode"/>
<node type="method" id="14" name="doThing"/>
<node type="method" id="16" name="X"/>
```

Les noms des balises étaient souvent accompagnés de parenthèses indésirables. Par exemple :

```
<node type="method" id="8" name="uneMethode()"/>
<node type="method" id="14" name="doThing()"/>
<node type="method" id="16" name="X>)"/>
```

Nous avons corrigé cela en rectifiant la méthode chargée d'extraire le nom du nœud, extractNodeName2(String) (src/main/java/graph).