SAE P21 — Java2Puml

Table des matières

Semaine 1 2

Notes 2
Diagrammes de séquence Javadoc 2
Diagrammes de classe (analyse) Javadoc 3
Diagrammes de classe (conception) Javadoc 3
Diagramme du package Western (généré) 4

Semaine 2 4

Notes 4
Diagrammes de classe (conception) Java language model 4
Diagramme du package Western (généré) 5
Commande pour créer **uniquement** un DCA 5
Diagrammes de notre API à cet instant 6

Semaine 3 6

Notes 6
Diagramme de classe (conception) de notre API (généré) 7
Diagramme de classe (conception) du langage (amélioré) 8
Diagramme de classe (analyse) du package Western (généré) 9

Semaine 4 9

Notes 9
Diagramme de classe (analyse) du package Western (généré) 10
Diagramme de classe (conception) du package Western (généré) 11
Diagramme de notre API à cet instant (généré) 12

Semaine 5 13

Notes 13
Diagramme de classe (analyse) de notre API (généré) 14
Diagramme de classe (analyse) de notre API (généré) 15
Diagramme de classe (analyse) du package Western (généré) 16
Diagramme de classe (conception) du package Western (généré) 17

Semaine 6 18

Notes 18
Diagramme de classe (analyse) de notre API (généré) 19
Diagramme de classe (conception) de notre API (généré) 20
Diagramme de classe (analyse) de notre API (***fait-main***) 21
Diagramme de classe (conception) de notre API (***fait-main***) 23

Semaine 1

Notes

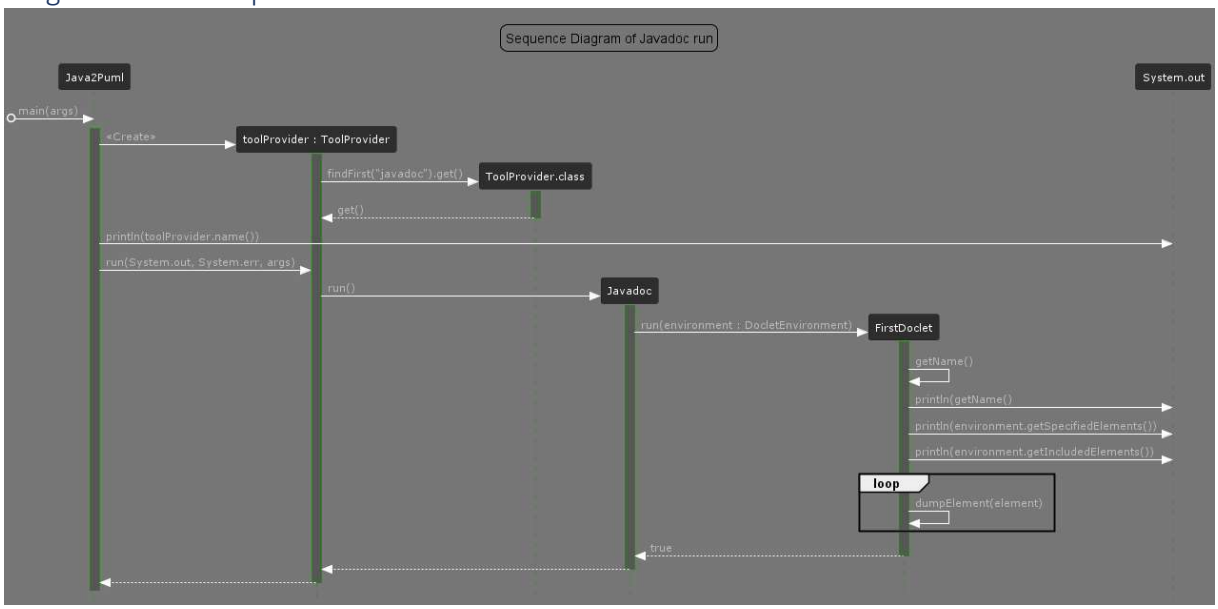
Tout d'abord, une définition de l'API comme nous l'entendons : une API est l'acronyme de « Application programming interface » (interface de programmation d'applications) ; c'est un ensemble de ressources, de classes qui permettent à deux applications de communiquer entre elles. Les API sont donc un moyen d'extraire et traiter des données de plusieurs contextes radicalement différents.

Nous avons appris à appréhender l'api javadoc et ses bases, tel la récupération d'« éléments » — les Class, Interfaces et Enumérations — via l'environnement java (autrement dit la machine virtuelle qui lit, interprète, exécute le programme).

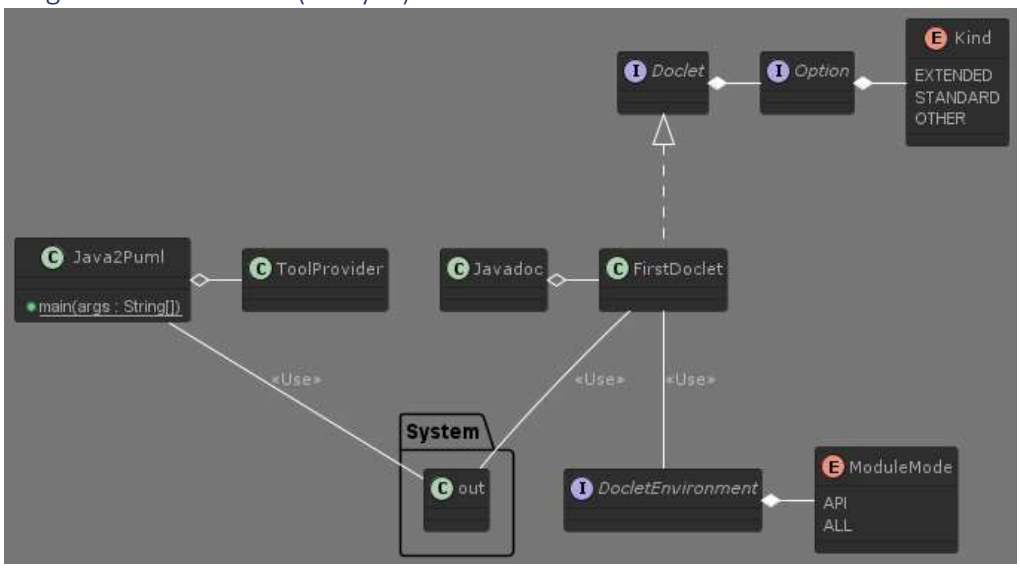
La commande javadoc de base, non pas pour l'utilisation de Doclet comme pour notre propre API mais pour la génération de fichier HTML, est d'ailleurs extrêmement pratique puisqu'elle nous permet d'obtenir rapidement et automatiquement un site de documentation pour notre code — en usant de la syntaxe « /** comment */ » et les balises associées, tels « @param », « @return », etc avec lequel Samuel F est déjà familier.

Cette partie n'a pas été d'une très grande difficulté. Néanmoins, notre API ne contient à ce stade que très peu d'objet, aucune classe abstraite ni interface ; il s'agira de diversifier l'ensemble la semaine prochaine est donner l'allure d'une véritable API à notre projet !

Diagrammes de séquence Javadoc



Diagrammes de classe (analyse) Javadoc



Diagrammes de classe (conception) Javadoc

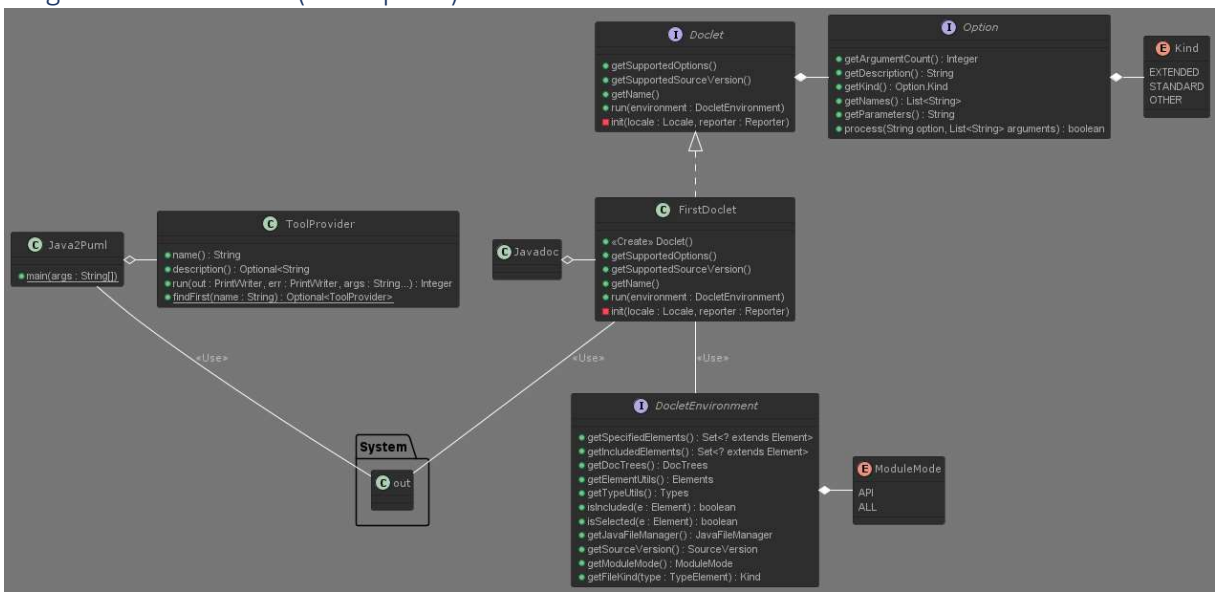
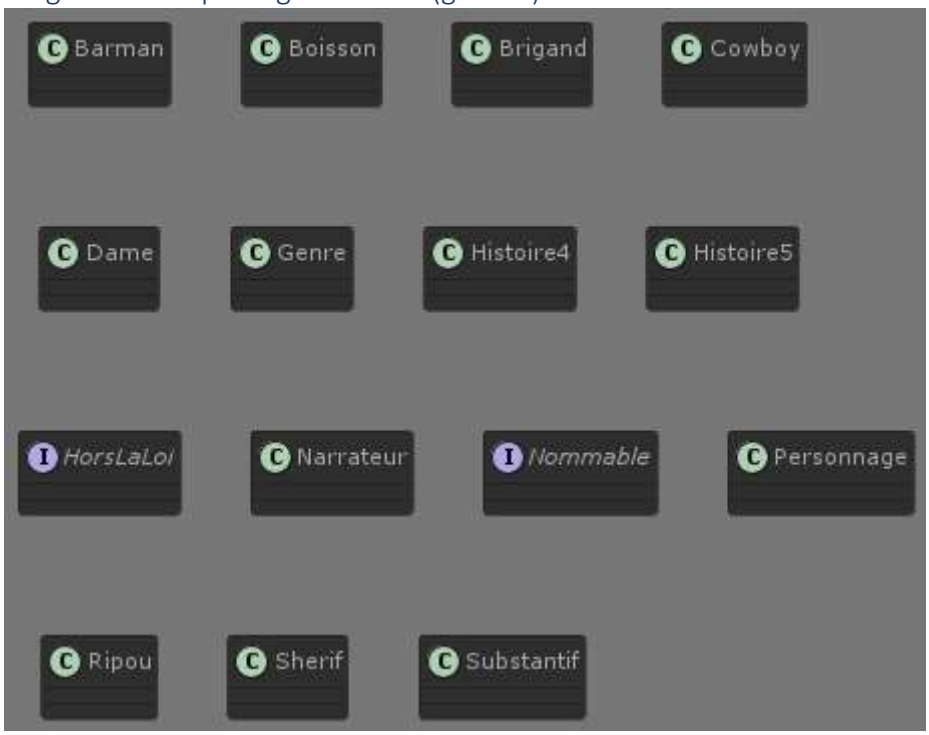


Diagramme du package Western (généré)



Semaine 2

Notes

L'amélioration de l'API avec l'ajout de nombreuses classes et interfaces a été long et quelque peu fastidieux (ayant assez mal organisé notre code), et le projet manque encore d'optimisation et de clarté. Néanmoins, les objectifs de la semaine 2 ont été atteints avec brio.

Les fonctions pour récupérer les détails des classes (visibilités des champs, des méthodes ainsi que le nom de ces dernières) et des interfaces (nom des méthodes) sont d'ailleurs déjà prêtes.

L'idée serait, à termes, qu'à chaque fois que l'utilisateur génère un schéma via notre PumlDoclet, ce dernier génère un DCA ET un DCC avec le nom précisé (Exemple : nomfichier_DCA et nomfichier_DCC) représentant le package souhaité.

Diagrammes de classe (conception) Java language model

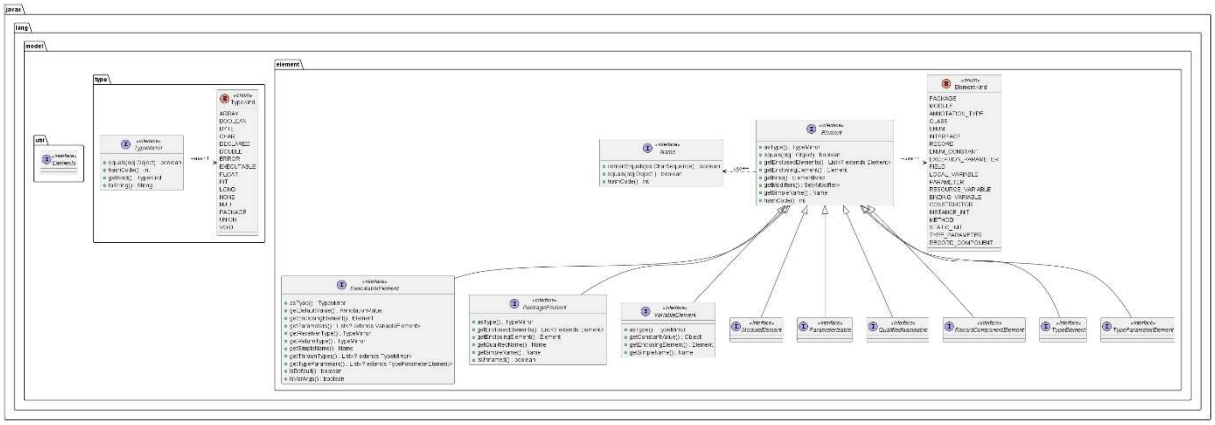
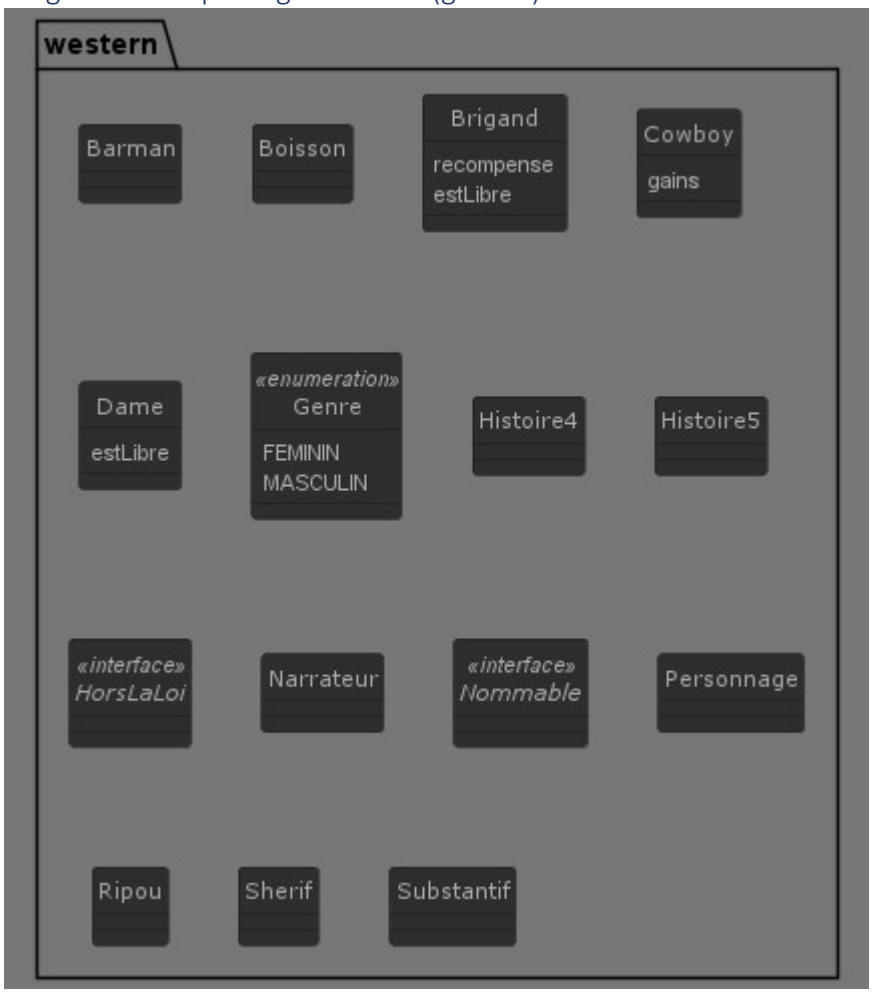


Diagramme du package Western (généré)

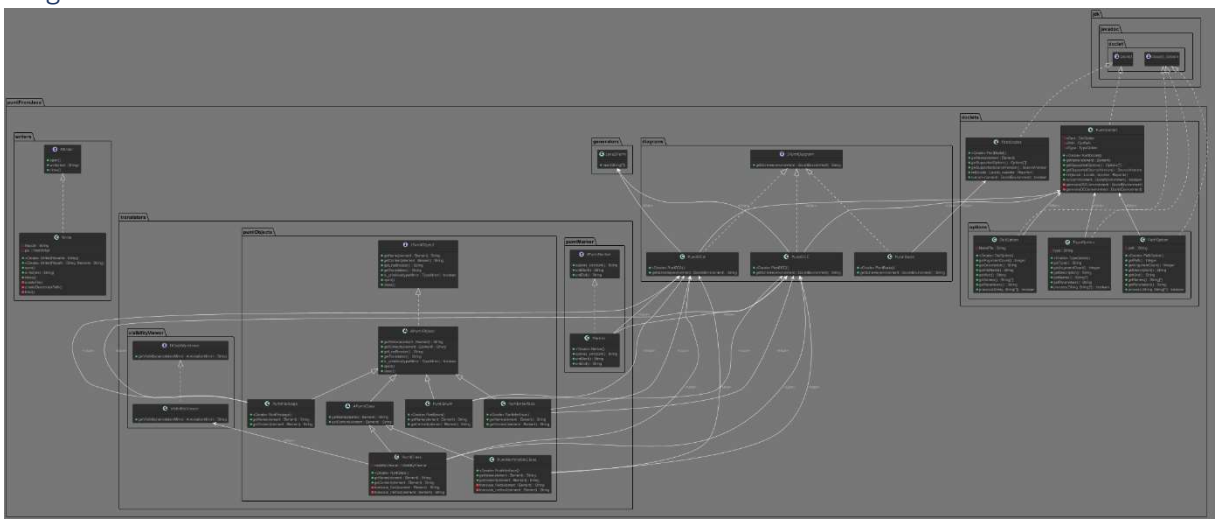


Commande pour créer **uniquement** un DCA

-private
-sourcepath
src
-doclet
pumlFromJava.doclets.PumlDoclet
-d
generates/pumls/
-g
dca
-out
ox.puml
western

L'option -g dca/dcc/both permet de générer un type particulier de schéma, ou les deux en même temps !

Diagrammes de notre API à cet instant



Semaine 3

Notes

Les objectifs de la semaine ont été atteint dans les temps et la branche assignée a été merge.

Même si cela n'était pas demandé, nous avons cherché à créer les flèches « Use » dans notre schéma en recherchant les classes définies dans les paramètres des fonctions de chacune de nos classes/interfaces. Sans succès pour le moment. Cependant, notons qu'il faudra peut-être utiliser le type « ExecutableElement ».

Diagramme de classe (conception) de notre API (généré)

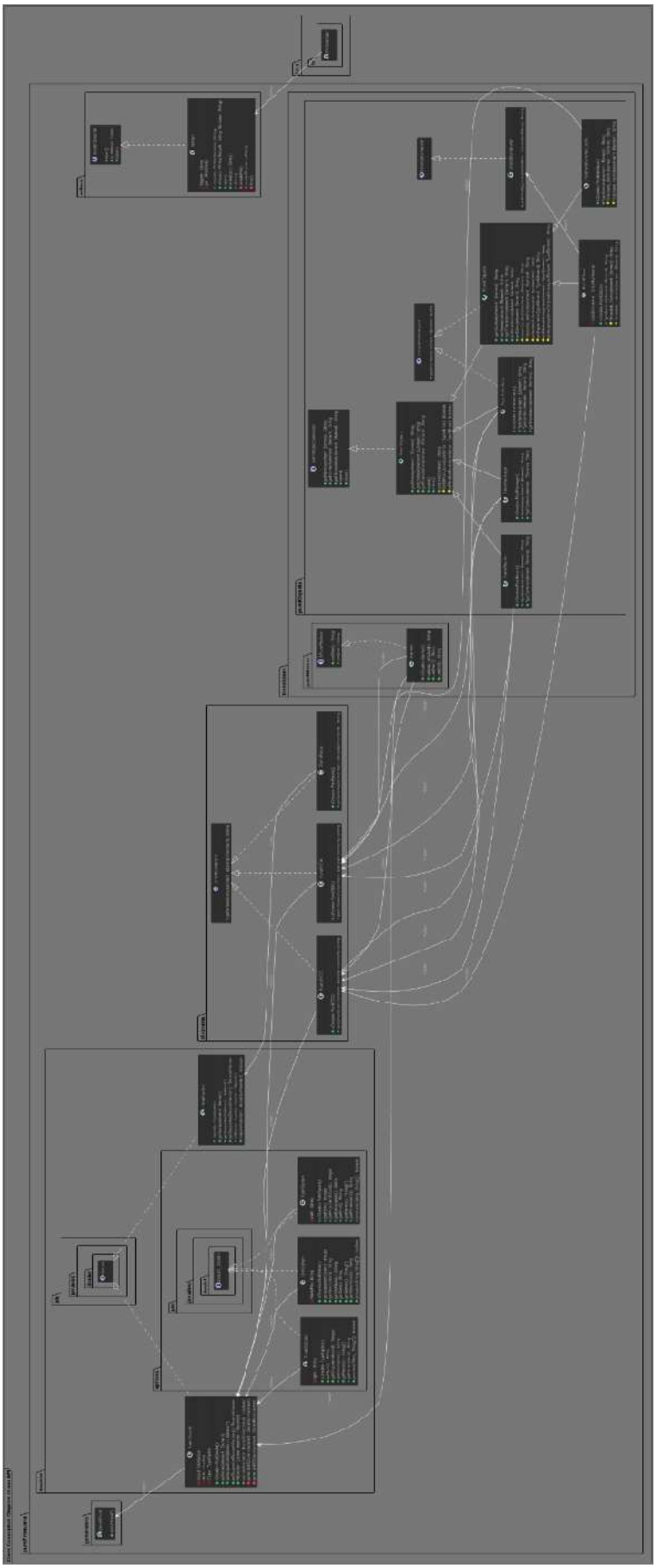


Diagramme de classe (conception) du langage (amélioré)

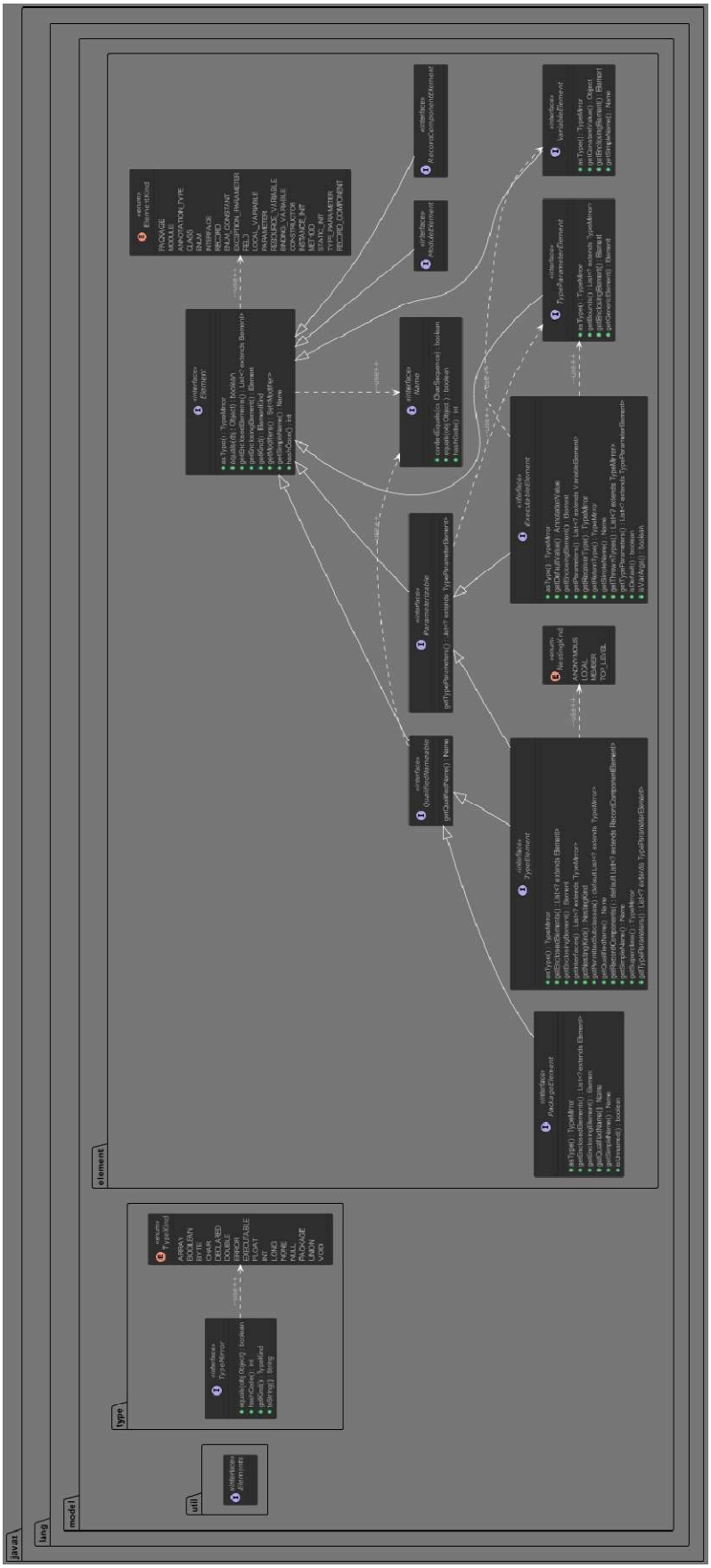
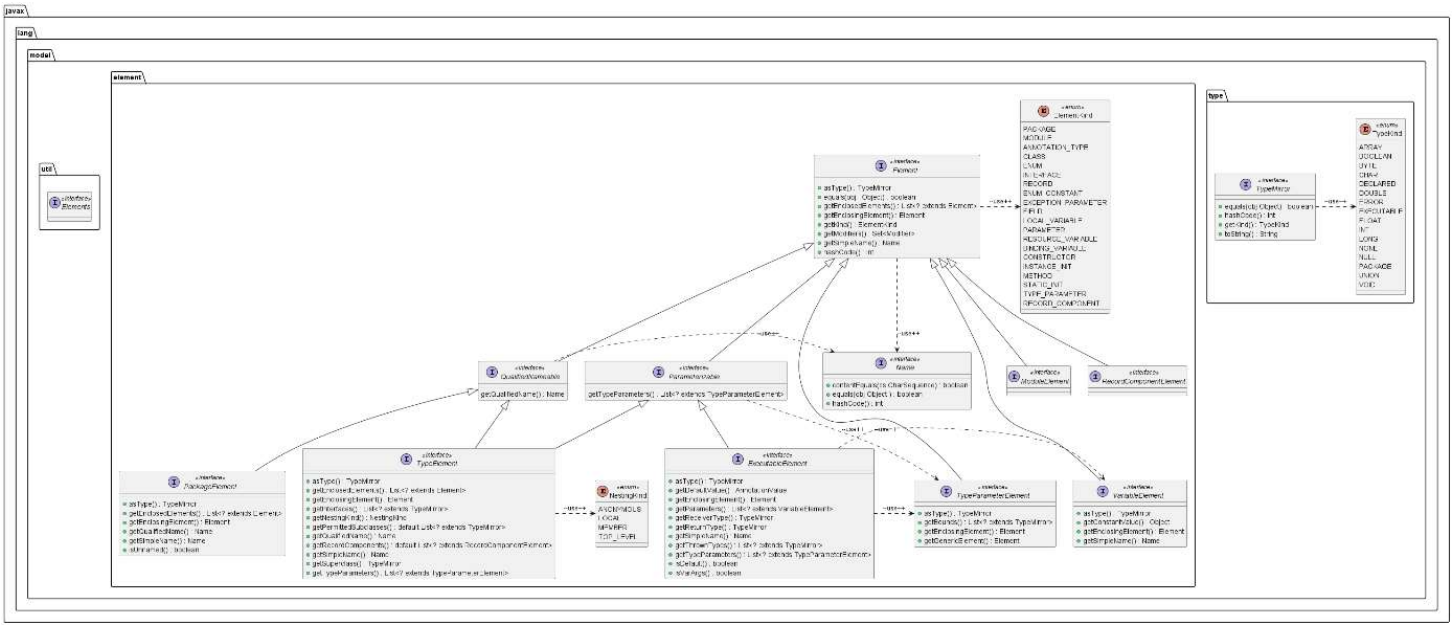


Diagramme de classe (analyse) du package Western (généré)



Semaine 4

Notes

Nous avons réorganisé l'API entière afin de rendre la conception plus logique et clair, et le code plus lisible. De nombreux commentaires ont été ajouté, des interfaces différenciées.

Tous les objectifs de la semaine ont été réalisé, les dépendances aussi, et les relations « Use » se basant sur les paramètres (uniquement) de chaque méthode sont également afficher (non nécessaire pour cette semaine, mais nous sommes allés plus loin).

De plus, les champs non primitifs des classes sont afficher avec les champs primitifs en attendant les améliorations (ajout des rôles) de la semaine 5.

Diagramme de classe (analyse) du package Western (généré)

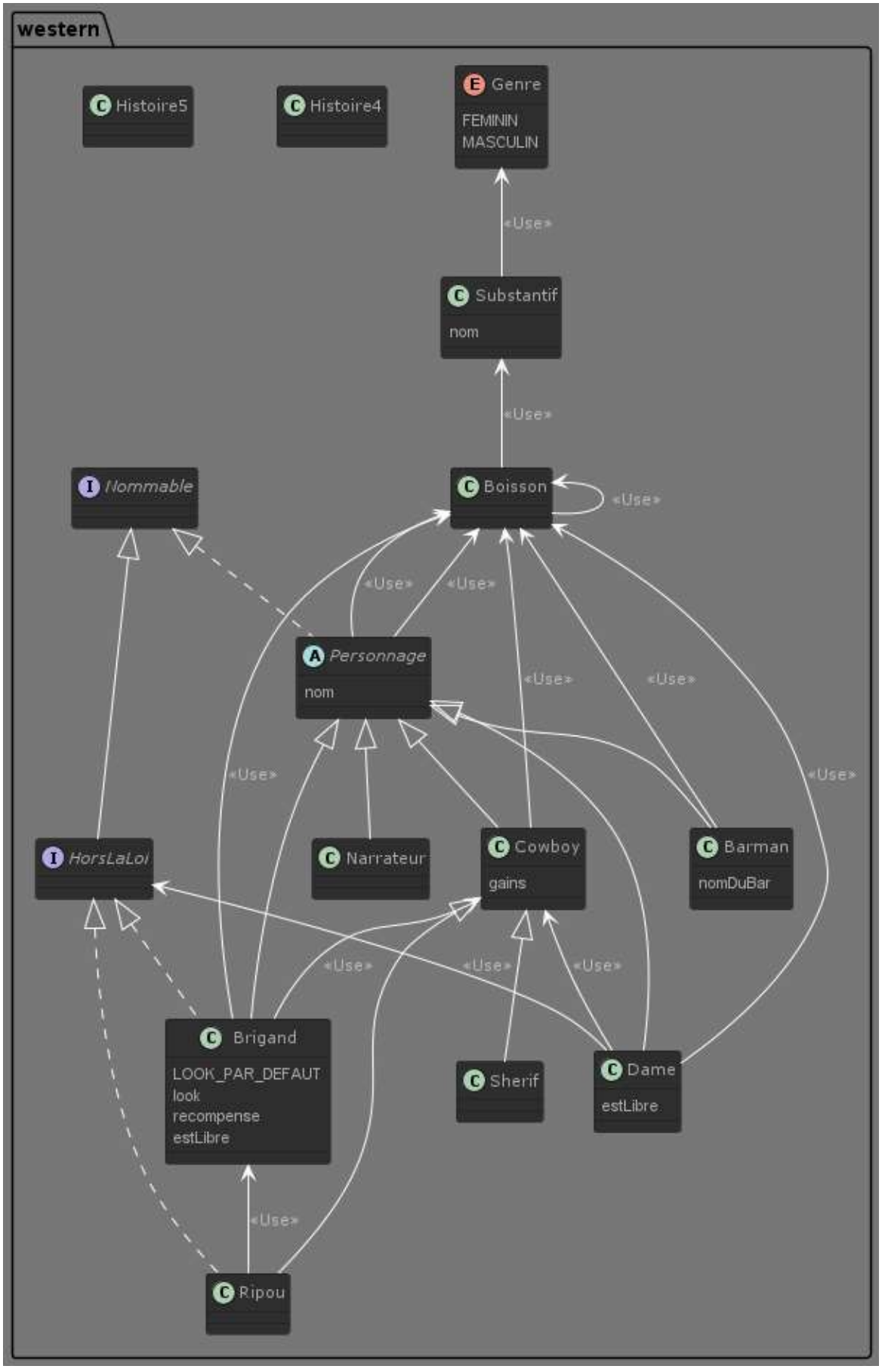


Diagramme de classe (conception) du package Western (généré)

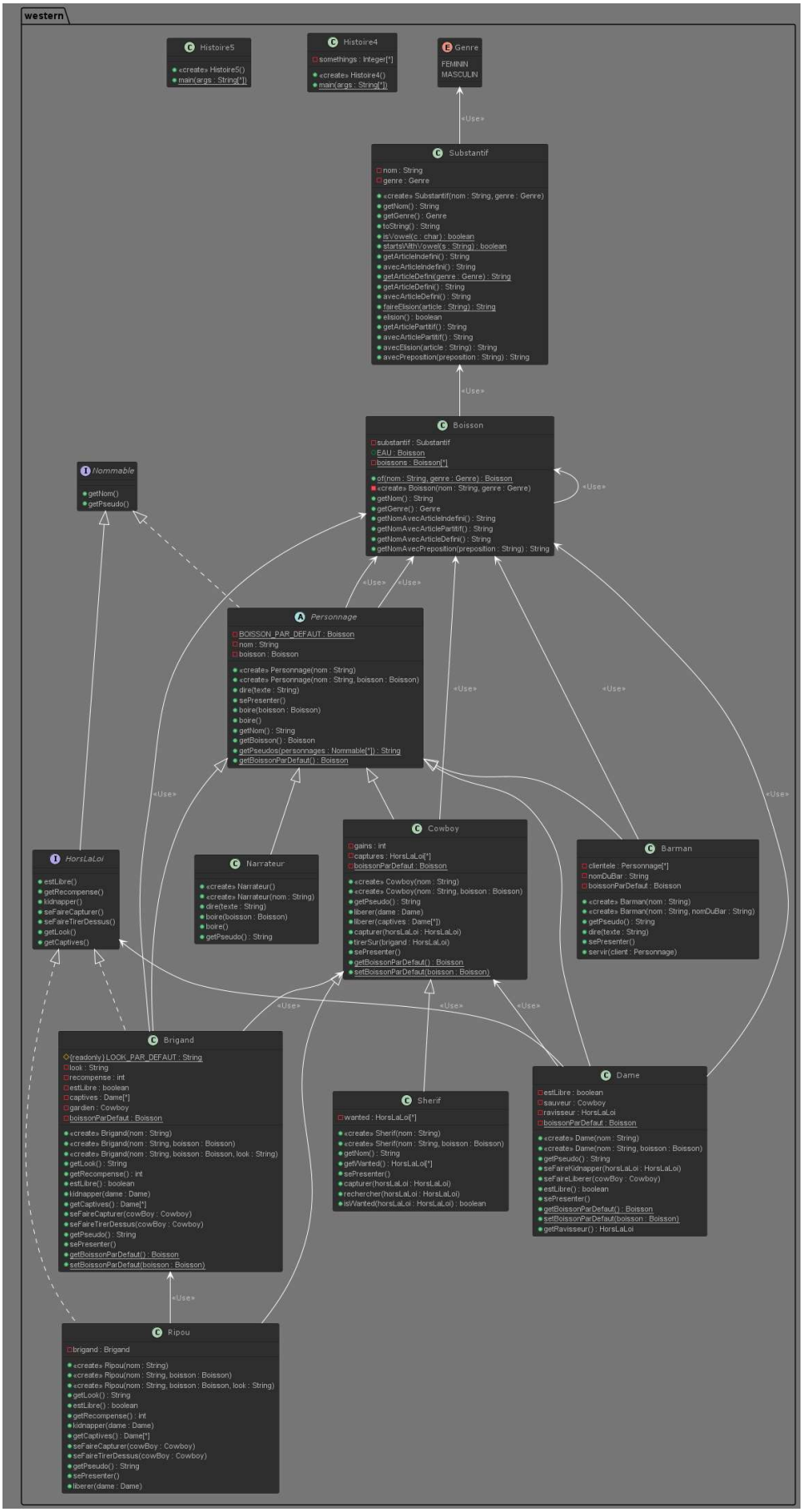
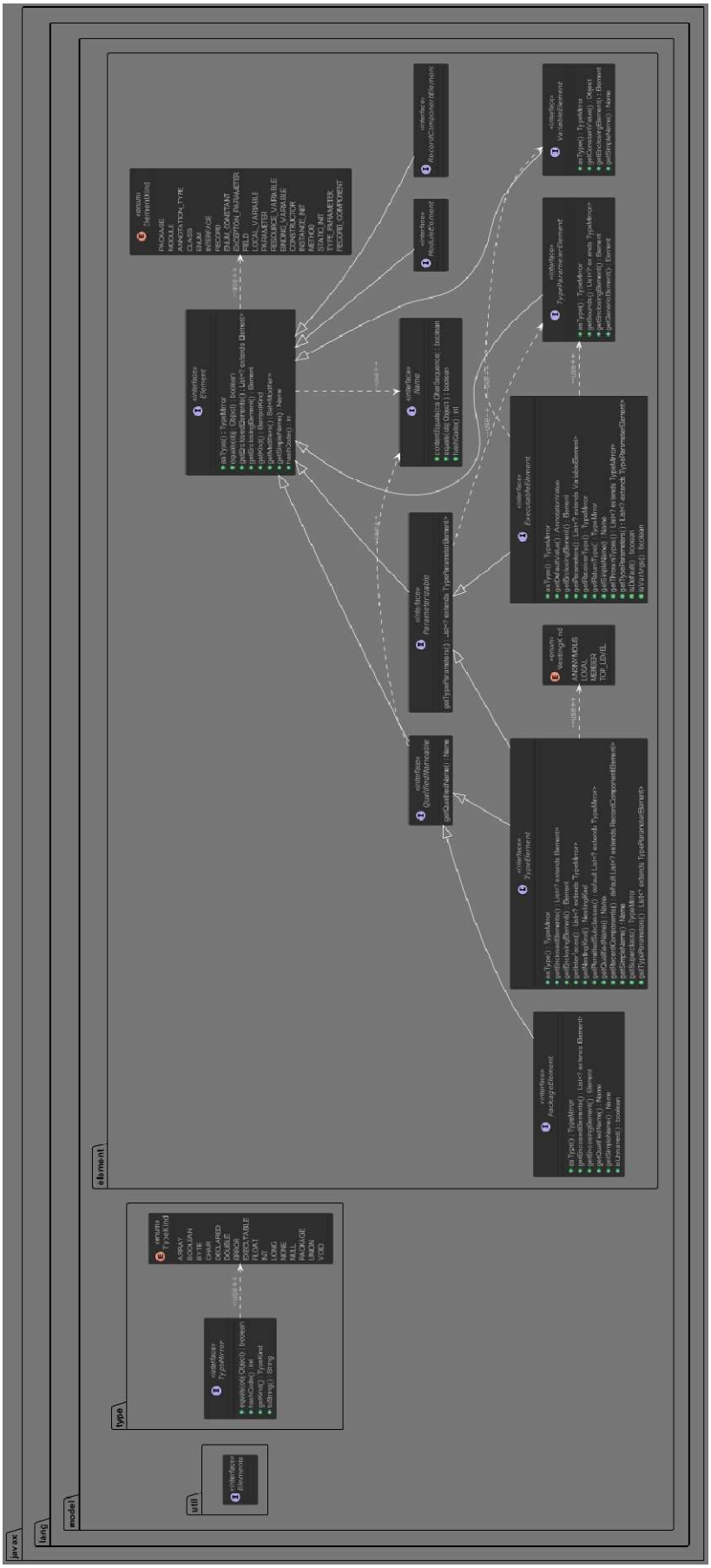


Diagramme de notre API à cet instant (généré)



Semaine 5

Notes

La « difficulté » de la semaine a été de supprimer tous les « Uses » inutiles. Soit les relations « Uses » entre deux classes possédant une agrégation ou composition commune ; et les relations déjà apparues se retrouvant en doubles. Ce problème a été fixé.

Pour les objectifs souhaités cette semaines, mise à part les relations citées précédemment ainsi que les multiplicités sur les rôles, ils ont tous été réalisés la semaine dernière — tous sont atteints et fonctionnels.

Diagramme de classe (analyse) de notre API (généré)

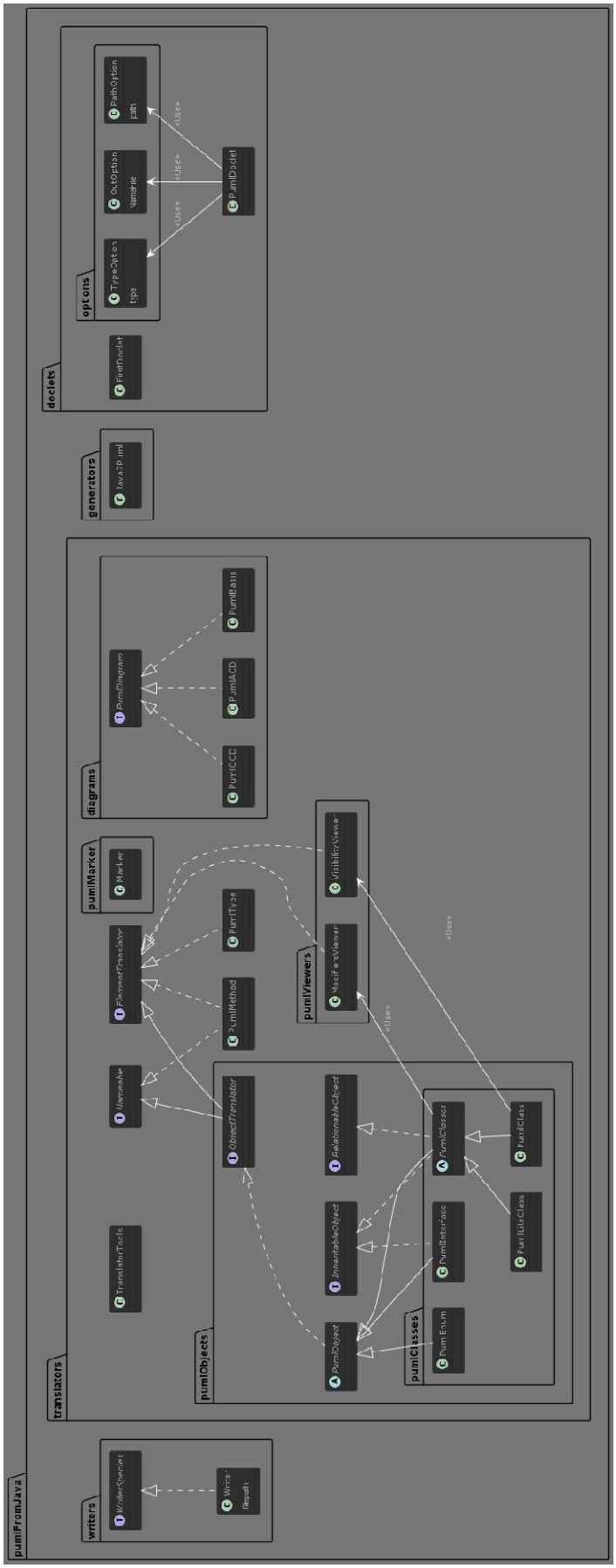


Diagramme de classe (analyse) de notre API (généré)

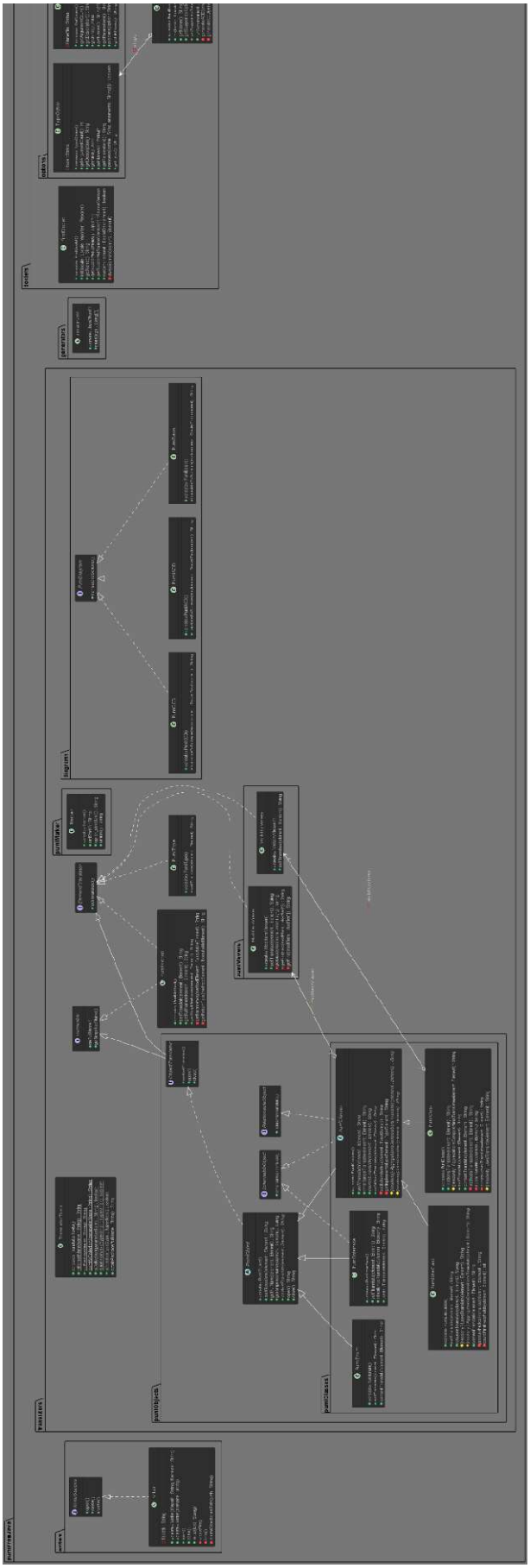


Diagramme de classe (analyse) du package Western (généré)

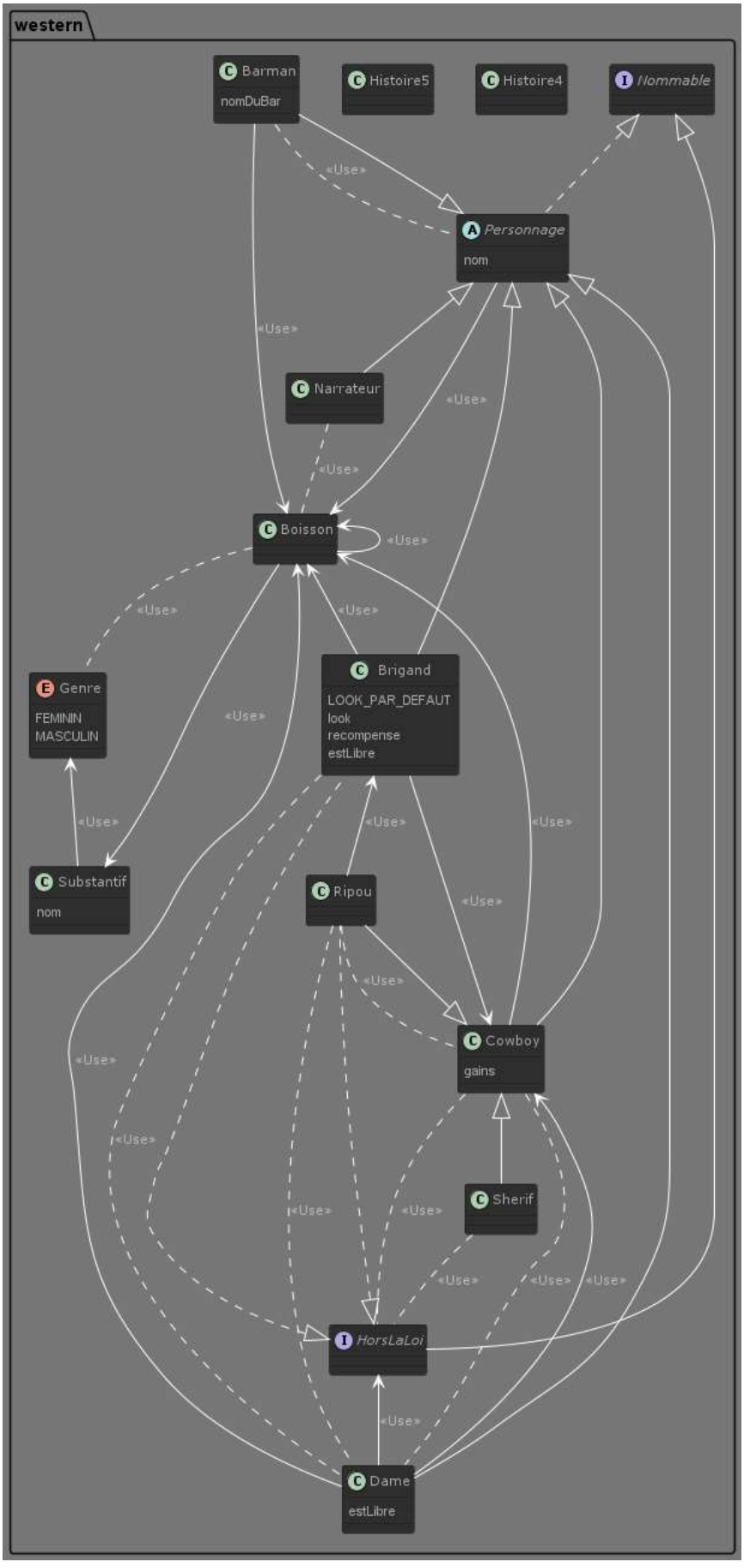
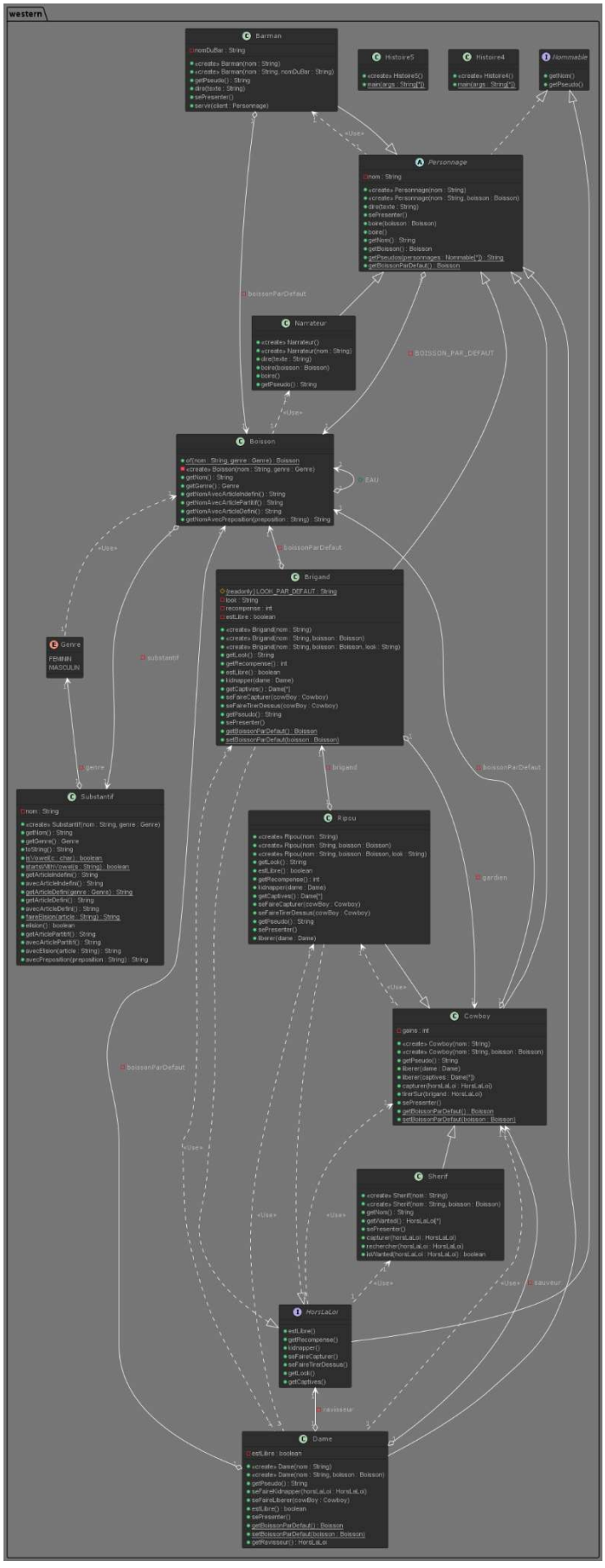


Diagramme de classe (conception) du package Western (généré)



Semaine 6

Notes

Nous traitons, comme demandé en bonus, les tag Override via l'analyse des AnnotationMirror de chaque fonction.

Quelques tags personnalisés sont parsemés là où nous jugions qu'ils pourraient servir, à la fois pour ce que notre API ne peut point générer — comme proposer dans l'énoncé — mais aussi aux lieux en liens avec une notion d'UML qui pourrait intéresser un utilisateur de l'API. Enfin, si nous devons donner la liste non exhaustive de ces tags, la voici : *@pumlAggregation, @pumlAggregation, @pumlInheritance, @pumlAssociation, @pumlMultiplicities, @pumlUses* ou encore *@pumlType* (donc qui ne correspond pas, comme dans les exemples donnés en énoncé, à une multiplicité, mais bien à un « type »).

Diagramme de classe (analyse) de notre API (généré)

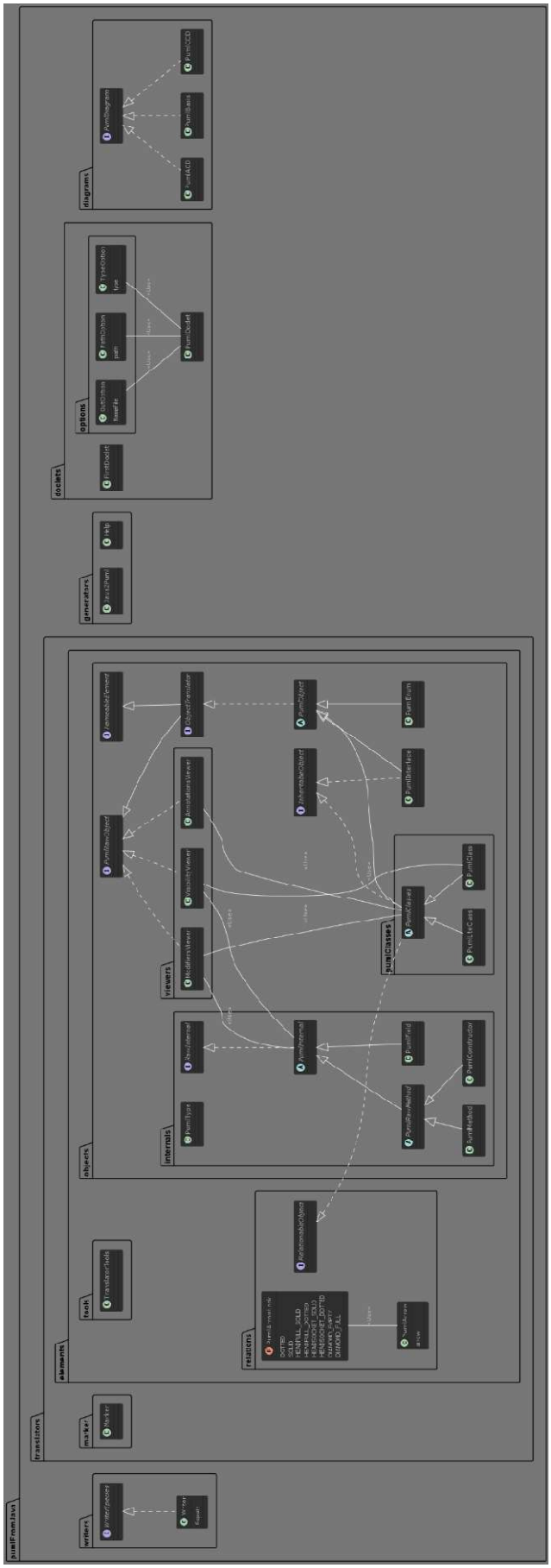


Diagramme de classe (conception) de notre API (généré)

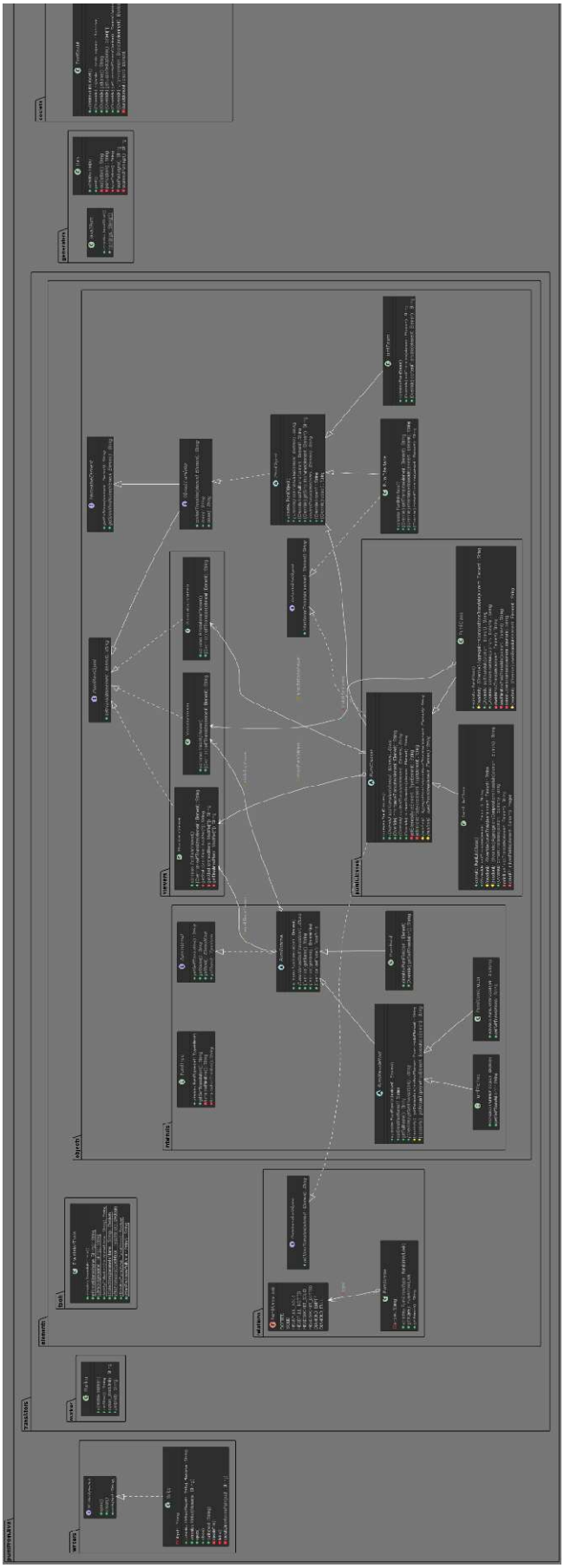


Diagramme de classe (analyse) de notre API (*fait-main*)

Nous avons complété les diagrammes suivants avec les quelques relations « Use » cachées au sein des méthodes (invisibles en paramètre, donc par l'API)

A noter que ces derniers diagrammes sont disponibles dans le projet, en format image comme format code, sous « deliverable/uml » !

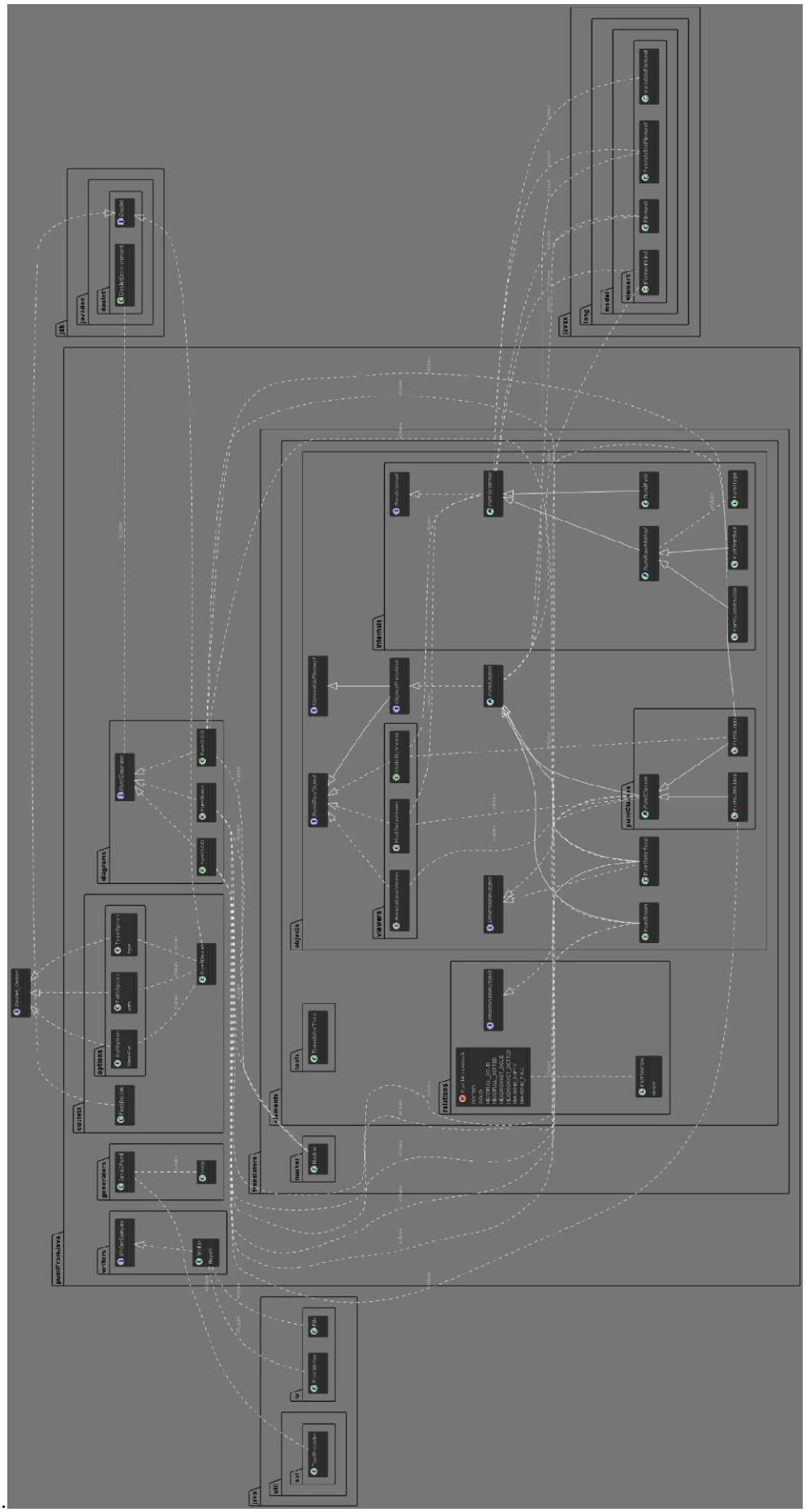


Diagramme de classe (conception) de notre API (***fait-main***)