Fil Rouge Petri Net

LACHGUER Soufiane
SOUISSI Mouad

UE - MAPD

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

Différences entre le Diagramme de classes de rétro-ingénierie et la conception initiale

1- La classe Petri Net et l'interface IPetriNet:

Nous avons gardé les mêmes méthodes pour l'interface IPetriNet, à l'exception de la méthode fire(transition) que nous avons placée dans la classe Transition (nommée execute()) . Nous avons supprimé cette méthode dans l'interface IPetriNet afin d'isoler les responsabilités de chaque classe (PetriNet et Transition). PetriNet est une classe qui ne s'occupe que de l'ajout, l'enlèvement et l'exécution simultanée des transition, tandis que la classe Transition gère la logique de l'activation d'une instance particulière. La classe PetriNet, dans les 2 cas, suit le contrat donné par l'interface. Nous avons ajouté les getters de chaque liste de composantes dans le PetriNet (places, arcs, transitions), ainsi que les 2 méthodes checkForDuplicateArcs() et displayPetriNetDetails(), demandées dans le cadre d'IDL. Les noms de certains attributs ont été changés : list_place à places, list_transition à transitions, list_arc à arcs, afin de respecter les conventions de nommation du code en anglais.

2- La classe Transition:

Nous avons changé certains noms par oubli (in par exArcs, out par enArcs, fire() par execute(), isMovePossible(int) par isFeasible()). Le choix de changer la signature de isMovePossible(int) relève du fait que la vérification de la disponibilité des arcs se fait dans la classe Arc, pour séparer les responsabilités de chaque classe. La classe Transition renvoie à la classe Arc pour faire la vérification. Le changement de la signature du constructeur de la classe Transition est dû au fait que chaque transition initiée démarre avec une liste vide des arcs sortants et entrants attachés. Des adders ont été ajoutés pour pouvoir mieux manipuler les instances d'une transition, et une méthode emptyArcsList() a été ajoutée pour pouvoir mieux gérer la suppression des transitions dans le PetriNet.

3- La classe Place:

Dans notre conception initiale, nous avions opté pour une classe Place simplifiée(notamment concernant "in" and "out" qui ont été remplacés par EnArc et ExArc), la conception initiale a été centrée sur la gestion des jetons et l'association directe des arcs entrants et sortants via le constructeur. Cependant, lors de l'implémentation, nous avons enrichi la classe afin de la rendre plus modulaire et évolutive. Nous avons introduit des méthodes spécifiques pour gérer les arcs (addExArc, addEnArc, emptyArcsList), permettant une manipulation plus flexible des relations entre les places et les arcs. Par ailleurs, la méthode emptyPlace a été ajoutée pour faciliter la réinitialisation d'une place(dont l'implémentation s'est avérée nécessaire pour les arcs videurs). Ces choix répondent à des besoins fonctionnels identifiés au cours du développement, notamment pour simplifier les opérations complexes et améliorer la lisibilité du code.

4- Les classes Arcs:

Concernant la classe Arc, nous l'avons rendue comme une classe abstraite car chaque type d'arc a une logique particulière. En effet, un arc videur vide tous les jetons d'une place d'un seul coup, et un arc zéro ne supprime aucun jeton, par exemple. Nous avons trouvé plus judicieux que chaque type d'arc aient un traitement particulier (au niveau de l'activation et de l'exécution), d'où ce choix. L'attribut type a été supprimé pour que chaque classe fille gère ce choix via la redéfinition de la méthode executeMove(). Certains attributs et méthodes ont été renommés (poids par weight, place_associée et transition_associée par associatedPlace at associatedTransition). La méthode getTransition() a été ajoutée pour faciliter l'exécution de l'activation d'une transition dans le cas d'une place entrante.

Concernant les classes filles de la classe Arc et de la classe ArcSortant (renommée en ExArc), toutes ces classes ont été renommées en anglais (ArcEntrant en EnArc, ArcZero en ZArc et ArcVideur en EmArc), La méthode isActive(int) dans ExArc est restée inchangée car c'est elle qui gère la logique de l'activation d'une transition (avec une redéfinition dans ZArc, cette méthode retourne false toujours dans notre cas). Nous avons supprimé la redéfinition prévue dans EmArc car la même logique d'activation dans ExArc reste valable dans EmArc. L'attribut de type String a été enlevé dans les classes fille car il n'est plus utilisé dans la classe mère Arc. Le constructeur de ZArc ne prend aucun int car le poids d'un arc zéro est fixé comme étant infini dans notre modèle (un arc Zéro étant interprété comme étant un arc qui ne prend aucun jeton de la place associée).

Commentaire sur les résultats de l'analyse statique du code avec STAN

Intéressons-nous tout d'abord au diagramme circulaire de pollution généré par STAN, afin de voir quels sont les types de pollution que notre code présente :

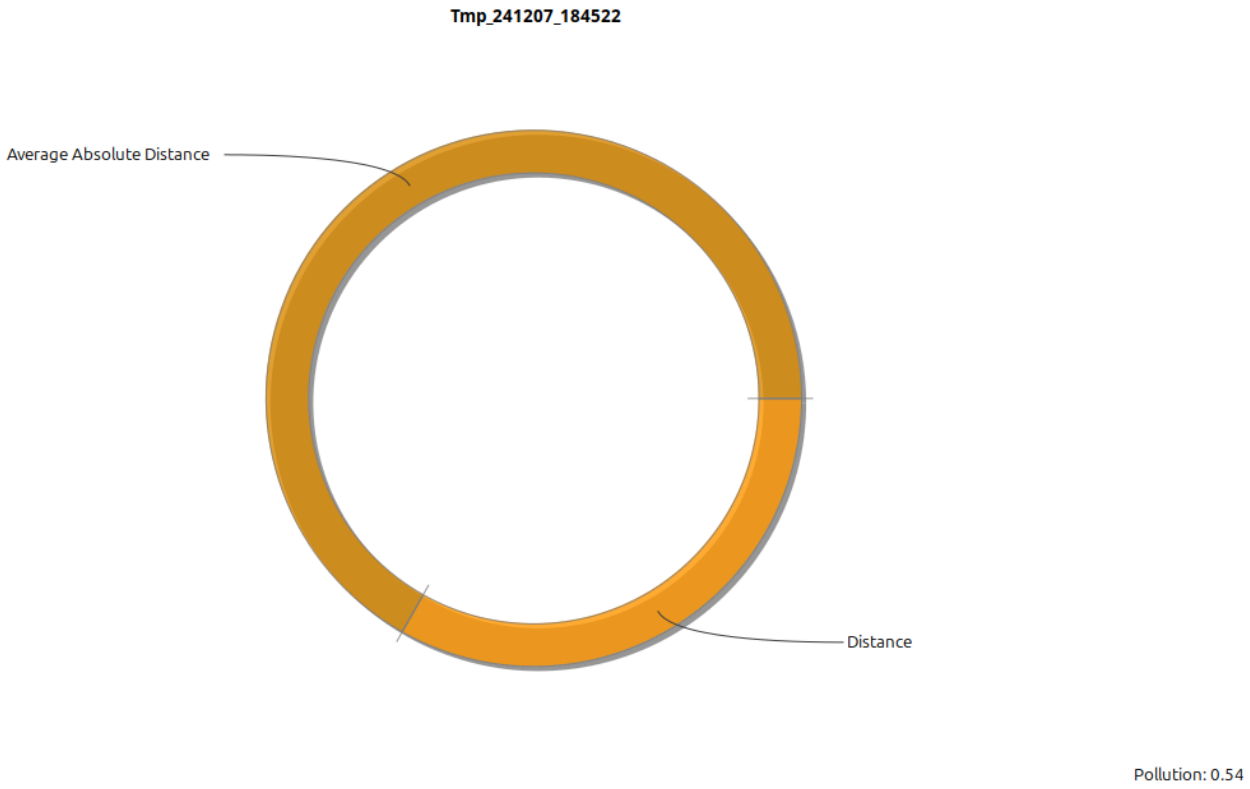


Figure 1 : diagramme circulaire de pollution de STAN .

Notre code présente 2 violations en total : la distance et la valeur moyenne absolue de la distance. (métriques parmi les métriques de Robert.C.Martin). Cette distance combine 2 facteurs majeurs de la qualité de code : l'abstraction du code A, qui est le ratio des types abstraits dans un package, et l'instabilité du code I, qui représente le taux de dépendance d'un package aux autres packages externes du code. Ces facteurs ont une valeur entre 0 et 1. Cette distance est mesurée via la formule suivante : D = A + I – 1. Pour que ce métrique soit bon, il faut qu'il soit compris entre -0,5 et 0,5. Dans notre cas, dans le package petrinet, cette distance vaut -0,82. Nous sommes alors incités à voir de plus en détails les métriques de Robert C.Martin dans le package petrinet, ainsi que le graphe des distances généré par STAN.

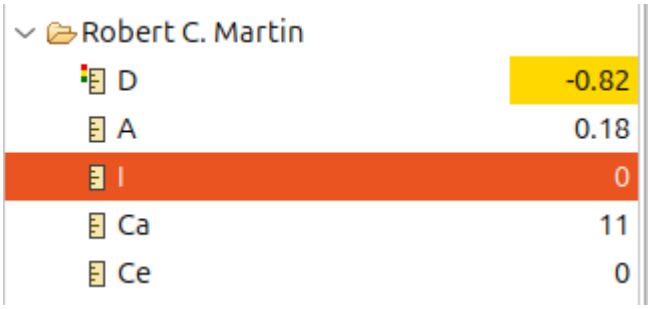


Figure 2 : métriques de Robert C.Martin dans le package petrinet .

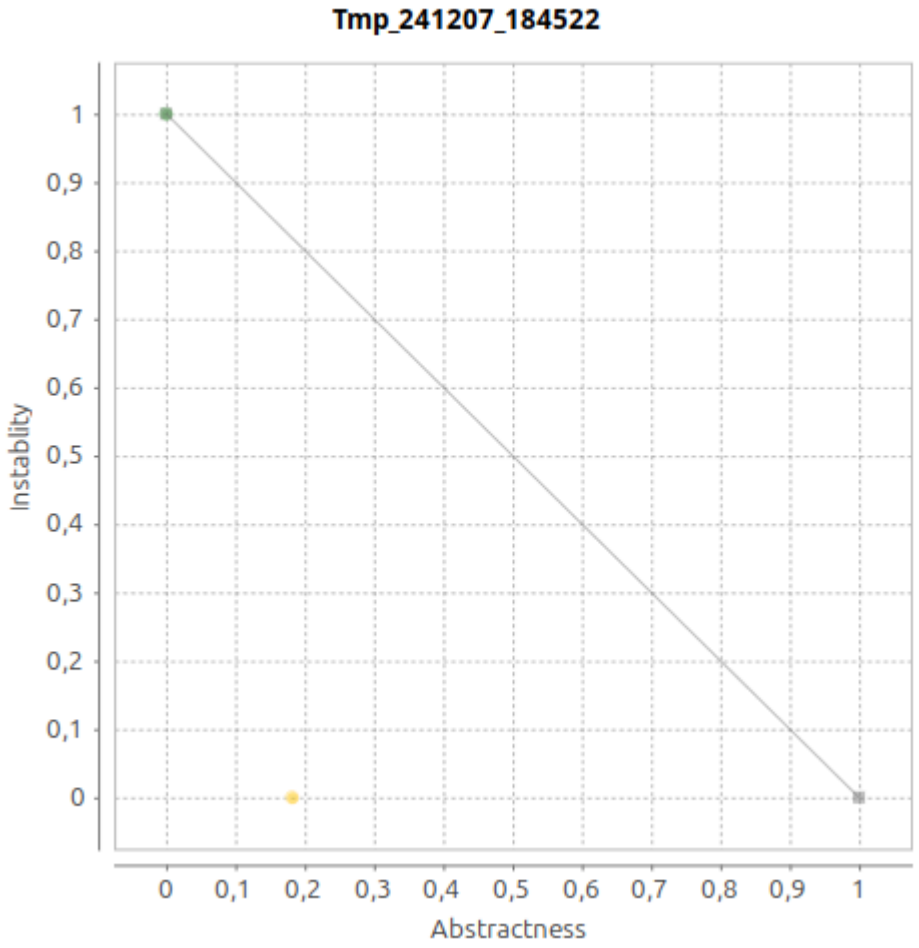


Figure 3 : graphe des distances généré par STAN (le point en jaune en bas est la distance du package petrinet, tandis que le point en vert en haut à gauche représente la distance du package petrinet_test)

Selon l'analyse effectuée par STAN, notre code ne présente aucune instabilité (ce qui est logique, car notre package petrinet ne fait aucun import externe d'autres packages mis à part pour les ArrayList), mais a une faible valeur d'abstraction. Ceci est dû au fait que notre package ne contient qu'une classe abstraite et une interface dans tout le package. Il fallait peut-être ajouter plus d'abstractions pertinentes dans le code (par exemple, ajouter un contrat gérant les places et les transitions).

Autre que le fait que notre code manque d'abstraction (selon STAN), notre projet ne semble montrer aucune autre violation dans la pollution. Intéressons-nous alors à la composition du package petrinet :

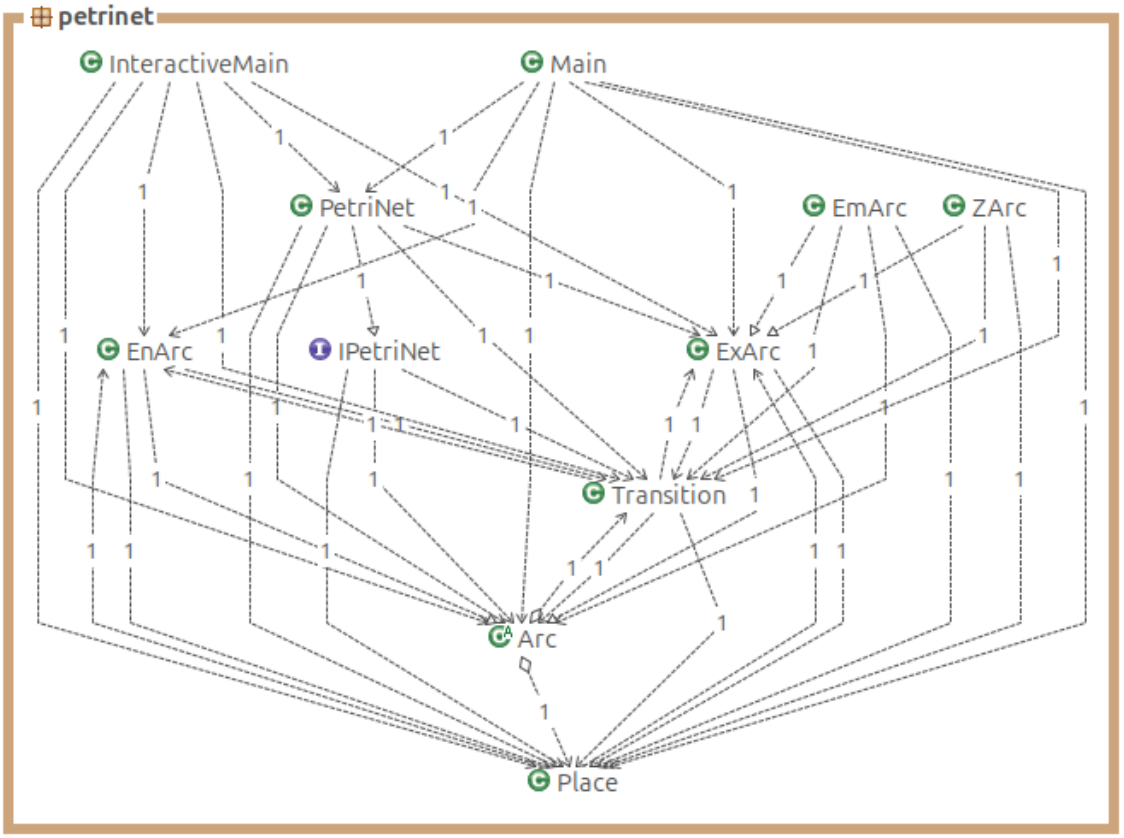


Figure 4: Composition du package petrinet

Toutes les classes de notre package référencent ou contiennent (dans le cas de la classe abstraite Arc) la classe Place, mais la classe Place ne référence aucune autre classe. Si on clique sur une liaison entre les classes, nous remarquons qu'aucune liaison ne présente une violation, ce qui permettrait de dire à première vue que le projet est bien structuré.

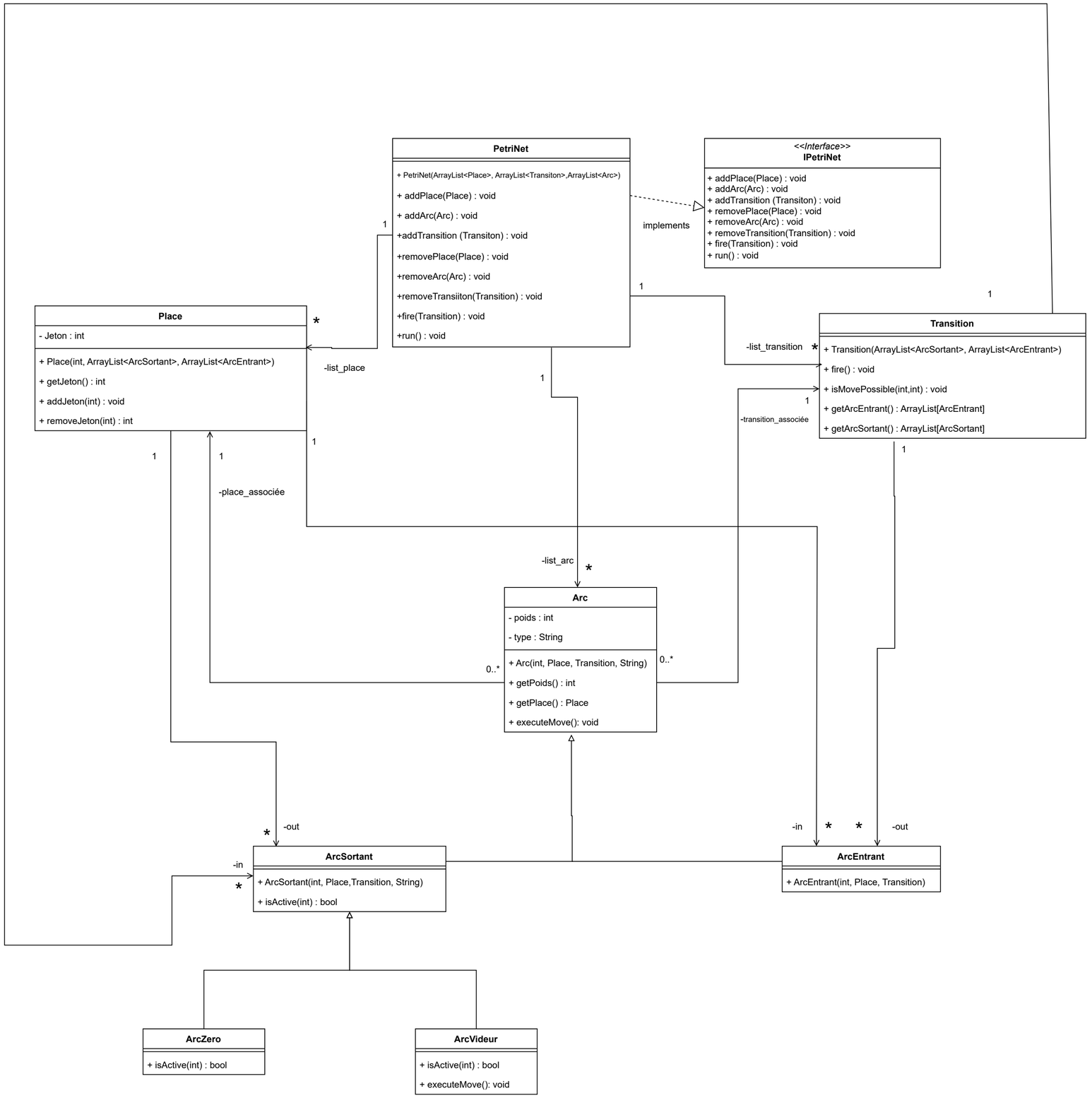
Difficultés rencontrées lors de l'intégration

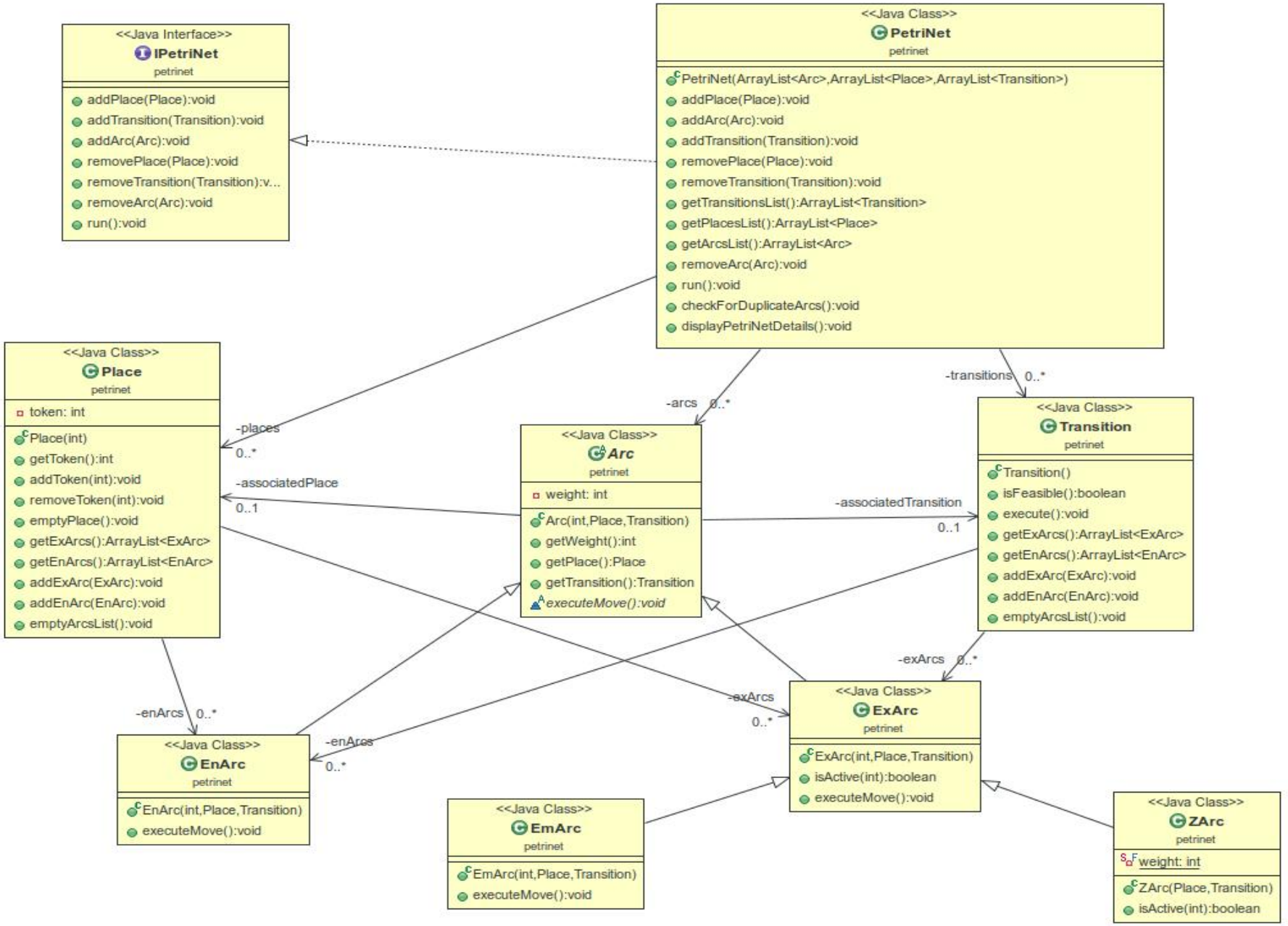
Afin de nous adapter aux consignes des deux derniers TPs indiquant qu'il ne faut pas éditer le code de l'implémentation, nous avons décidé d'effectuer l'intégration **uniquement** à l'aide du package Adapter (basé sur le Design Pattern Adapter). Nous avons rencontré quelques difficultés liées à la différence entre notre implémentation et la logique de l'interface, nous présentons dans ce tableau les différents problèmes rencontrés ainsi que la solution adoptée pour chaque problème.

	Difficulté rencontrée	Solution
Arcs	-Création d'arcs spéciaux: L'interface ne permet la création que d'arcs normaux par défaut, puis les changer en arc zéro ou videur.	- Ajout de méthodes spécifiques (makeIntoInhibitory, makeIntoReset) pour permettre de convertir un arc normal en un arc inhibiteur ou un arc zéro.
	- Création d'arcs entrants et sortants: Création d'un seul type d'arc par défaut (pas de sous types entrants et sortants prédéfinis)	- Adoption de deux logiques distinctes pour créer des arcs entrants et sortants.

Arcs Zéros (ZArc)	- Dans l'interface les arcs zéros n'inhibent pas l'activation des arcs entrants liés à l'arc zéro à travers une transition	- Nous avons opté pour une logique différentes de sorte que les arcs zéros inhibent TOUT LE TEMPS l'activation de la transition (ce fut une erreur de notre part, nous avons mal compris cette partie de l'énoncé) - Afin de contourner ce problème, nous avons décidé d'attribuer un poids par défaut aux arcs inhibiteurs égal à 1 au lieu d'assigner un poids infini aux arcs zéros comme cela a été fait dans notre conception et implémentation initiale.
Places et Transitions	- L'interface repose sur une classe mère unique appelée noeud	- Nous avons implémenté deux classes AdapterPlace et AdapterTransition suivant toutes les deux deux logiques différentes en fonction que l'origine et la destination de l'arc.
	- L'interface associe un label à chaque noeud	- Notre code ne gère pas les labels, en conséquence nous avons introduit cette notion dans les Adapters.
Gestion des jetons	- Limitation à l'utilisation de méthodes standards comme setTokens sans prise en charge explicite de la suppression de jetons.	- Comme l'interface ne propose pas de méthode removeTokens pour retirer des jetons d'une place, nous avons opté pour la généralisation de la méthode setTokens pour permettre à la fois d'ajouter et de retirer des jetons, en indiquant directement le nombre de jetons qu'une place doit conserver.

Annexe: Diagrammes de classe (Rétro ingénierie et conception initiale)


PetriNet
+ PetriNet(ArrayList<Place>, ArrayList<Transiton>,ArrayList<Arc>)
+ addPlace(Place) : void
+ addArc(Arc) : void
+addTransition (Transiton) : void
+removePlace(Place) : void
+removeArc(Arc) : void
+removeTransiiton(Transition) : void
+fire(Transition) : void
+run() : void
<<Interface>>
IPetriNet
+ addPlace(Place) : void
+ addArc(Arc) : void
+ addTransition (Transiton) : void
+ removePlace(Place) : void
+ removeArc(Arc) : void
+ removeTransition(Transition) : void
+ fire(Transition) : void
+ run() : void
implements
1
1
Place
- Jeton : int
+ Place(int, ArrayList<ArcSortant>, ArrayList<ArcEntrant>)
+ getJeton() : int
+ addJeton(int) : void
+ removeJeton(int) : int
*
-list_place
1
1
1
-place_associée
1
-list_transition
*
Transition
+ Transition(ArrayList<ArcSortant>, ArrayList<ArcEntrant>)
+ fire() : void
+ isMovePossible(int,int) : void
+ getArcEntrant() : ArrayList[ArcEntrant]
+ getArcSortant() : ArrayList[ArcSortant]
1
-transition_associée
1
-list_arc
*
Arc
- poids : int
- type : String
+ Arc(int, Place, Transition, String)
+ getPoids() : int
+ getPlace() : Place
+ executeMove(): void
0..*
0..*
*
-out
-in
*
ArcSortant
+ ArcSortant(int, Place,Transition, String)
+ isActive(int) : bool
-in
*
*
-out
ArcEntrant
+ ArcEntrant(int, Place, Transition)
ArcZero
+ isActive(int) : bool
ArcVideur
+ isActive(int) : bool
+ executeMove(): void


<<Java Interface>>
IPetriNet
petrinet
addPlace(Place):void
addTransition(Transition):void
addArc(Arc):void
removePlace(Place):void
removeTransition(Transition):v...
removeArc(Arc):void
run():void
<<Java Class>>
PetriNet
petrinet
PetriNet(ArrayList<Arc>,ArrayList<Place>,ArrayList<Transition>)
addPlace(Place):void
addArc(Arc):void
addTransition(Transition):void
removePlace(Place):void
removeTransition(Transition):void
getTransitionsList():ArrayList<Transition>
getPlacesList():ArrayList<Place>
getArcsList():ArrayList<Arc>
removeArc(Arc):void
run():void
checkForDuplicateArcs():void
displayPetriNetDetails():void
<<Java Class>>
Place
petrinet
token: int
Place(int)
getToken():int
addToken(int):void
removeToken(int):void
emptyPlace():void
getExArcs():ArrayList<ExArc>
getEnArcs():ArrayList<EnArc>
addExArc(ExArc):void
addEnArc(EnArc):void
emptyArcsList():void
<<Java Class>>
Arc
petrinet
weight: int
Arc(int,Place,Transition)
getWeight():int
getPlace():Place
getTransition():Transition
executeMove():void
<<Java Class>>
Transition
petrinet
Transition()
isFeasible():boolean
execute():void
getExArcs():ArrayList<ExArc>
getEnArcs():ArrayList<EnArc>
addExArc(ExArc):void
addEnArc(EnArc):void
emptyArcsList():void
<<Java Class>>
EnArc
petrinet
EnArc(int,Place,Transition)
executeMove():void
<<Java Class>>
ExArc
petrinet
ExArc(int,Place,Transition)
isActive(int):boolean
executeMove():void
<<Java Class>>
EmArc
petrinet
EmArc(int,Place,Transition)
executeMove():void
<<Java Class>>
ZArc
petrinet
weight: int
ZArc(Place,Transition)
isActive(int):boolean
-transitions 0..*
-arcs 0..*
-places 0..*
-associatedPlace 0..1
-associatedTransition 0..1
-exArcs 0..*
-enArcs 0..*
-exArcs 0..*
-enArcs 0..*