Documentations complète sur le projet d'implémentation du dilemme du prisonnier à réaliser par groupe de deux

Patrons et Composants

Spécifications – Dilemme du prisonnier

AVANZINO Aurélien – GOURDON Stéphanie

HISTORIQUE DES MODIFICATIONS

AUTEUR	LIBELLE	DATE MODIFICATION
GOURDON Stéphanie	Rédaction initiale du document	
AVANZINO Aurélien	Ajout des spécifications version 2.0	
GOURDON Stéphanie	Ajout des spécifications version 2.5	
GOURDON Stéphanie	Suppression des clones Strategies	
AVANZINO Aurélien	Ajout des design patterns respectés	
AVANZINO Aurélien	Ajout des spécifications version 3.0	

INFORMATIONS COMPLEMENTAIRES :

- Dépôt Gitlab du projet : https://gitlab.com/AurelienAVZN/pc_dilemmeduprisonnier
- JavaDoc du projet : https://pc-dilemmeprisonnier.netlify.app/ (disponible également dans le dossier javadoc du projet)
- Binôme partenaires rendu N°4 : Mariia Selivanova – Marie-Josée Bassil
- Adresses Mail :
    - AVANZINO Aurélien : aurelien.avanzino@etu.univ-grenoble-alpes.fr
    - GOURDON Stéphanie : stephanie.gourdon@etu.univ-grenoble-alpes.fr

Table des matières

I. Introduction

L'objectif du projet consiste à proposer un modèle UML d'une petite application permettant de mettre en œuvre des tournois de stratégies jouant au *Dilemme du prisonnier*, puis de proposer une implémentation en JAVA de cette application.

Des modifications et/ou fonctionnalités peuvent être ajoutées au fur et à mesure suivant les nouvelles spécifications demandées par le client.

II. Le Dilemme du Prisonnier

II.1 Description du projet

II.1 Introduction

Le *Dilemme du prisonnier* est un modèle très simple de la théorie des jeux. Il semble appréhender en miniature les tensions entre cupidité individuelle et intérêts de la coopération collective. Pour cette raison, il est devenu un des modèles les plus utilisés en sociologie, biologie et économie.

II.2 Présentation du dilemme

Deux suspects porteurs d'armes ont été arrêtés devant une banque et mis dans deux cellules séparées. Les deux prévenus ne peuvent pas communiquer et doivent choisir entre avouer qu'ils s'apprêtaient à commettre un hold-up ou ne rien avouer. Les règles que le juge leur impose sont les suivantes :

- Si l'un avoue et pas l'autre, celui qui avoue sera libéré en remerciement de sa collaboration et l'autre sera condamné à cinq ans de prison ;
- Si aucun n'avoue, ils ne seront condamnés qu'à deux ans de prison, pour port d'arme illégal ;
- Et si les deux avouent, ils iront faire chacun quatre ans de prison.

Dans cette situation, il est clair que si les deux s'entendent (pas d'aveu), ils s'en tireront globalement mieux (2 x 3 ans de remise de peine) que si l'un des deux dénonce l'autre (1 x 5 ans de remise de peine). Mais alors l'un peut être tenté de s'en tirer encore mieux en dénonçant son complice. Craignant cela, l'autre risque aussi de dénoncer son complice pour ne pas être le dindon de la farce. Le dilemme est donc : Faut-il accepter de couvrir son complice (donc coopérer avec lui) ou le trahir ?

II.3 Une version implémentable du dilemme

Ce dilemme peut devenir beaucoup plus intéressant et plus réaliste lorsque la durée de l'interaction n'est pas connue. On peut envisager de se souvenir du comportement d'un joueur à son égard et développer une stratégie en rapport. Par exemple, si je sais que mon adversaire ne coopère jamais, mon intérêt sera de ne pas coopérer non plus, sous peine d'être systématiquement floué. Par contre si je sais que mon adversaire coopèrera toujours quoi qu'il arrive, j'aurai intérêt à être vicieux et ne jamais coopérer pour maximiser mon gain.

II.2 Spécifications

II.2.1 L'application à réaliser

L'application à réaliser doit permettre de mettre en œuvre des tournois de stratégies jouant au dilemme des prisonniers tel que décrit dans la section II.1.3. Les stratégies sont implémentées en JAVA. Il faudra implémenter les stratégies décrites en section (insérer la section) plus vos stratégies personnelles ou celles de vos collègues. Un tournoi est la confrontation d'un ensemble de stratégies. L'ensemble des stratégies est un sous-ensemble, choisi par l'utilisateur, des stratégies disponibles. Une rencontre entre deux stratégies se joue en n tours, n étant également défini par l'utilisateur. Dans un tournoi, une stratégie doit rencontrer toutes les stratégies, elle-même comprise, de l'ensemble sélectionné. Le score réalisé par une stratégie est la somme de ses points récoltés lors de chacune des confrontations.

II.2.2 Exemples de stratégie

- Gentille : Je coopère toujours
- Méchante : Je trahis toujours

- Donnant-Donnant : Je coopère à la première partie, puis je joue ce qu'a joué l'autre à la partie précédente.
- Donnant-Donnant-Dur : Je coopère à la première partie, puis je coopère sauf si mon adversaire a trahi lors de l'une des deux parties précédentes
- Méfiante : Je trahis à la première partie, puis je joue ce qu'a joué l'autre à la partie précédente
- Rancunière : Je coopère à la première partie, mais dès que mon adversaire trahit, je trahis toujours.

II.2.3 Interaction entre les stratégies

On suppose que deux stratégies lors d'une confrontation ne peuvent pas passer d'accord. La seule information qu'une tribu connaît sur l'autre est son comportement passé lors des coups précédents. Les décisions des deux tribus lors de la partie sont prises simultanément. Les nombre de parties n'est pas connu à l'avance.
Par rapport au simple dilemme des prisonniers nous choisissons de rajouter la possibilité de renoncer à jouer, mais ce refus est définitif.

De manière plus abstraite, deux entités peuvent choisir entre coopérer (notation c), trahir (notation t) ou renoncer (notation n). Si l'une trahit et l'autre coopère (partie [t, c]), celle qui trahit obtient un gain de T unités et celle qui coopère (et donc s'est fait duper) obtient un gain de D unités. Lorsque les deux entités coopèrent (partie [c,c]), elles gagnent chacune C unités en récompense de leur association. Quand elles trahissent toutes les deux (partie [t, t]), elles gagnent P unités pour s'être laissé piéger mutuellement. Si une partie n'as pas lieu parce qu'une l'une a refusé de jouer les deux entités gagnent N unités. Le choix des coefficients T, D, C, P et N n'est pas fortuit. Conformément aux n°181 de POUR LA SCIENCE nous prenons : T = 5, D = 0, C = 3, P = 1, N = 2.

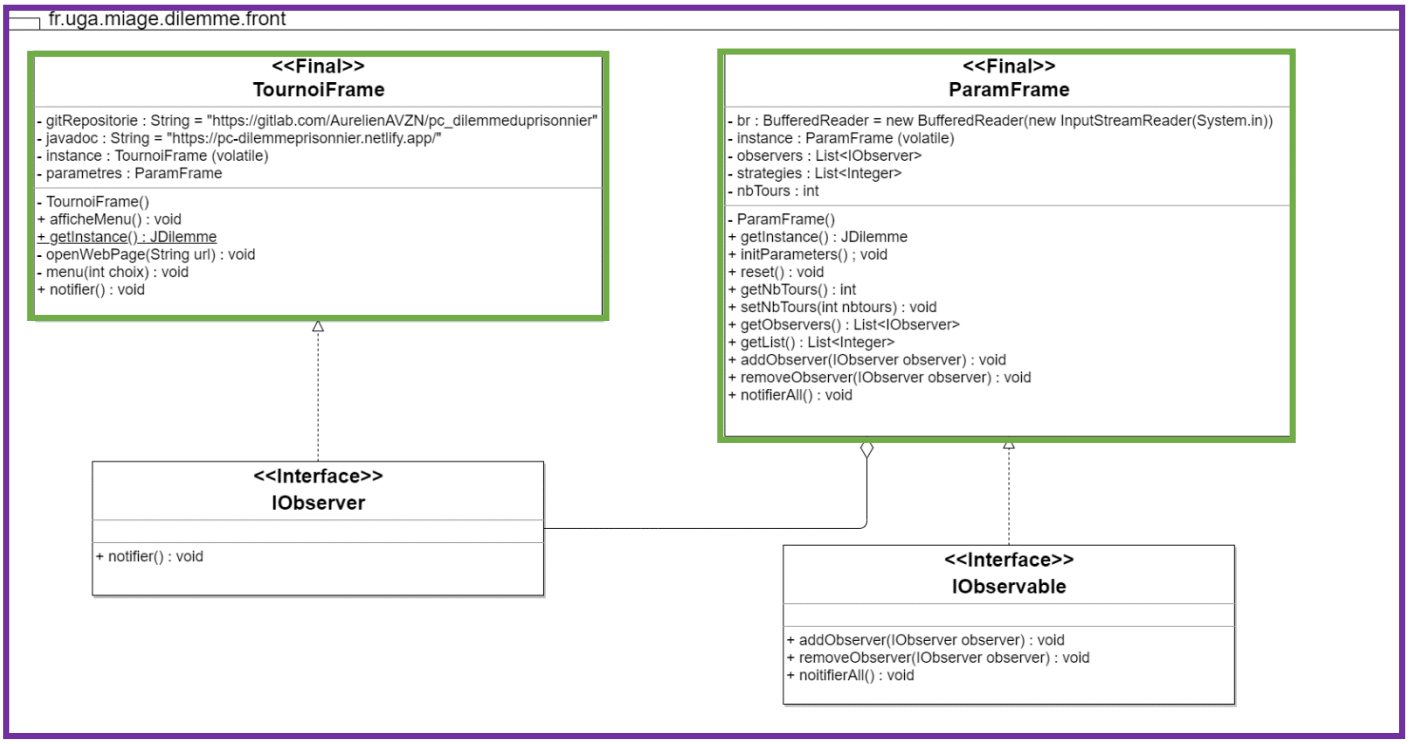
III. Comportement du logiciel

Les diagrammes de classes et de séquences représentent l'implémentation des dernières versions. Actuellement ils existent deux versions développées en parallèle. La version 2.6 qui possède un affichage console et la version 3.2 possédant un affichage graphique (développé via javax.swing), cependant dans leur implémentation et fonctionnement ces dernières sont identiques.

III.1 Diagramme de classe

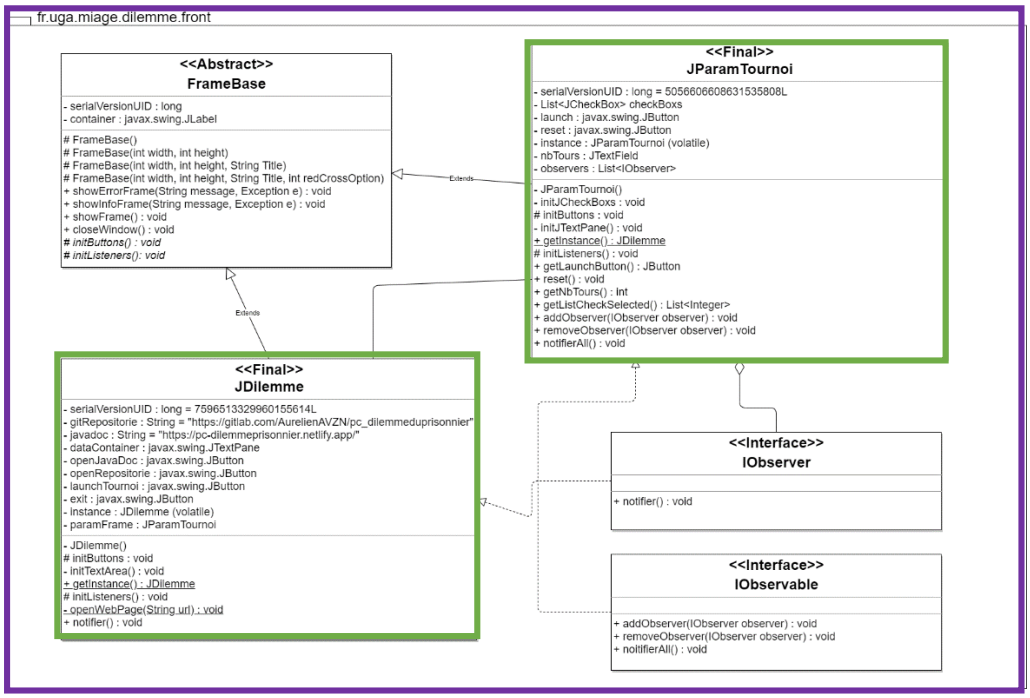
III.1.1 Version 2.6

PARTIE FRONT DE L'APPLICATION



III.1.2 Version 3.1

PARTIE FRONT DE L'APPLICATION



III.1.3 Diagramme de classe de la partie back

Pour la partie back-end de l'application, l'implémentation est identique au deux versions en développement

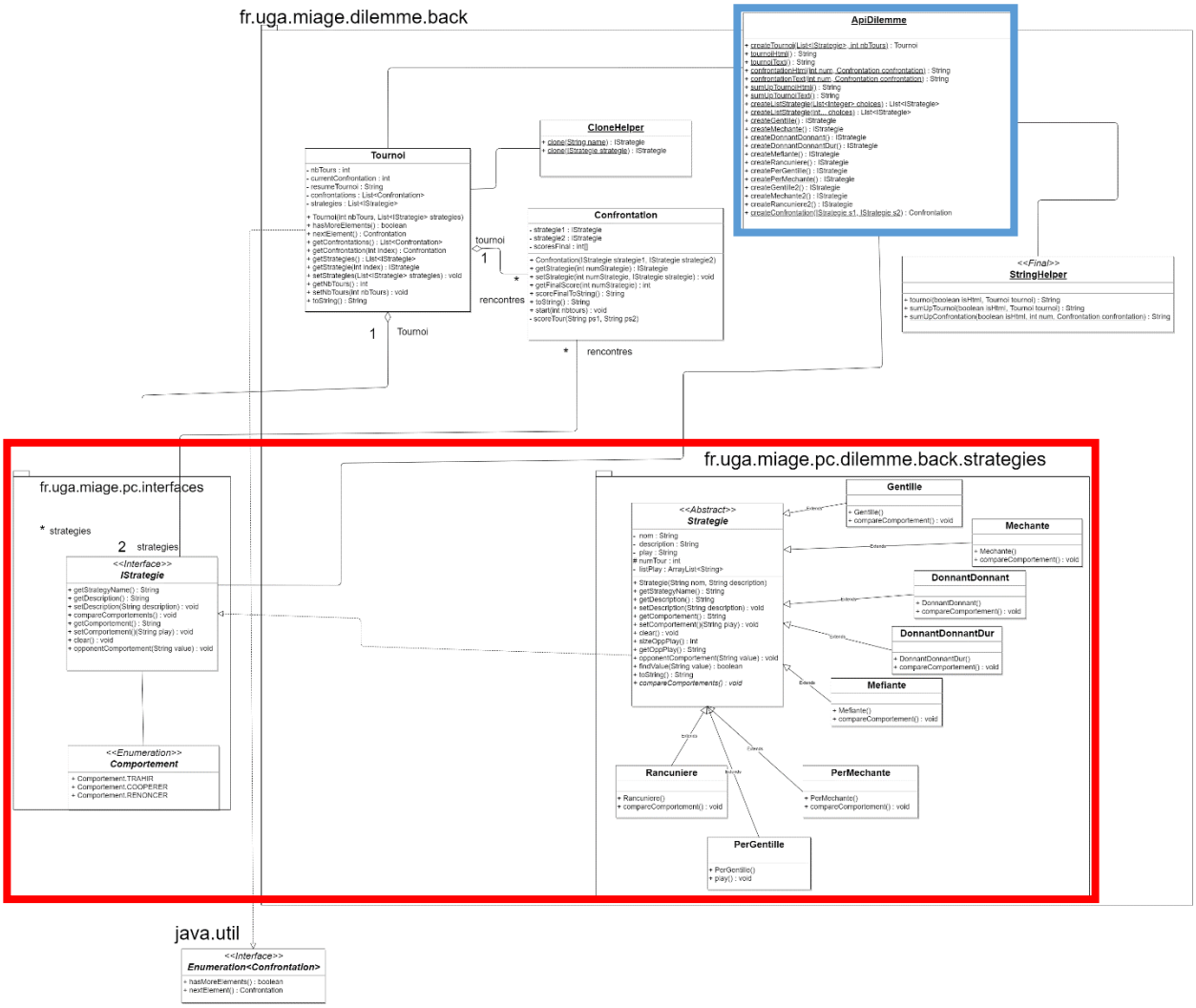
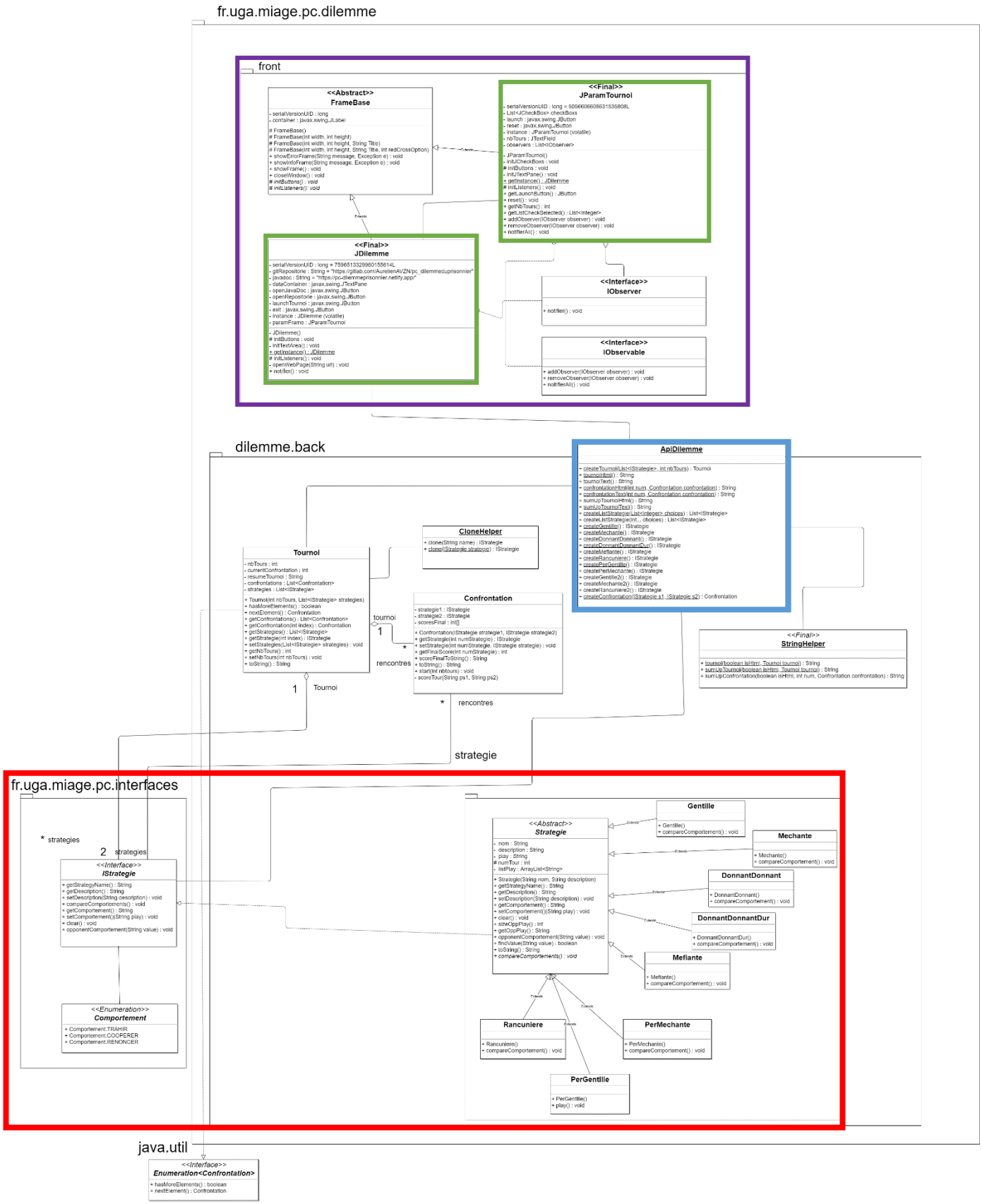


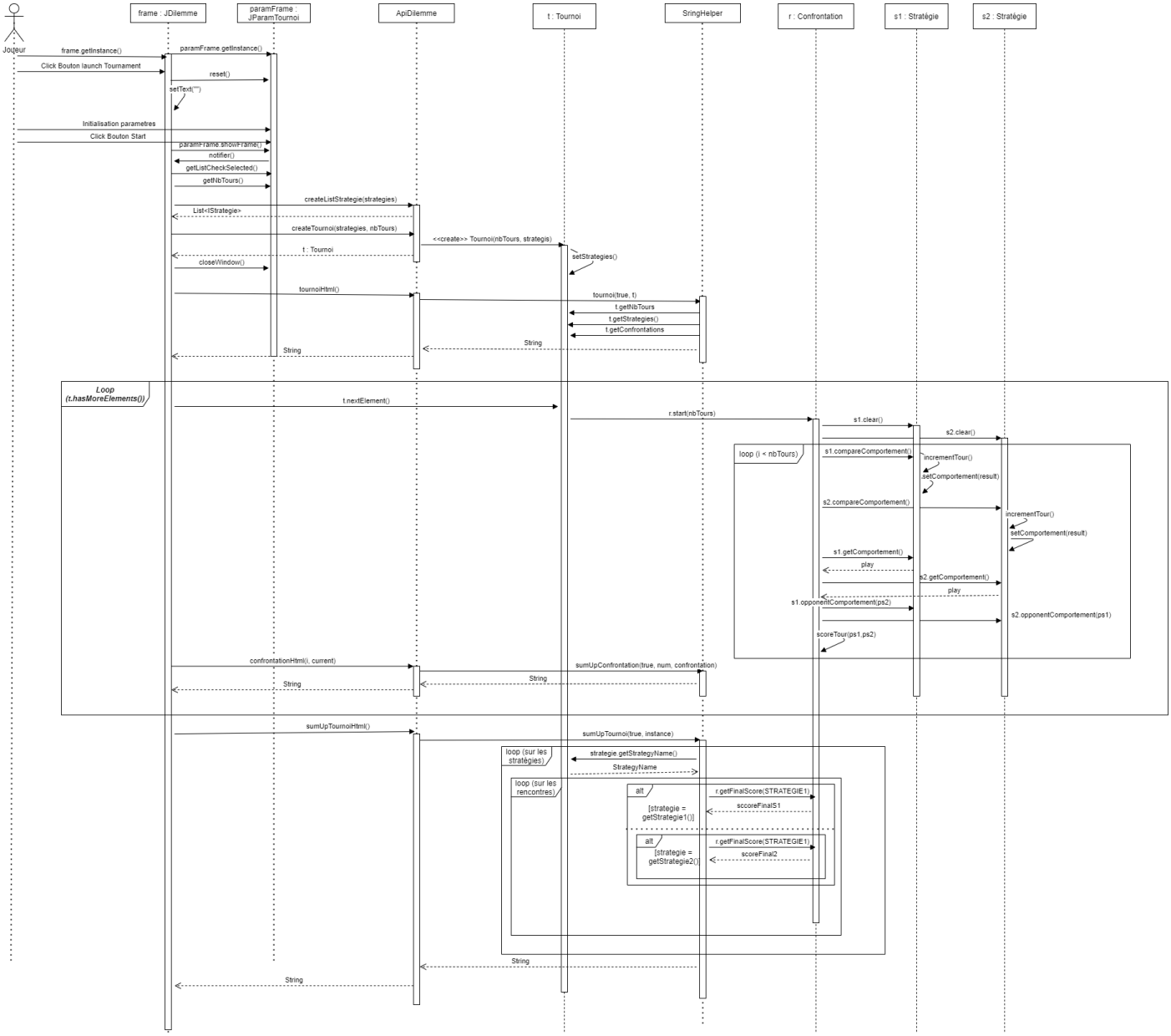
DIAGRAMME DE CLASSE COMPLET (VERSION 3.1)



N.B : Pour la version 2.6 le diagramme de classe complet est identique, seule la partie front change comparé à la version 3.1.

Le détail des design pattern respecté se trouve dans le document suivant : Details diagrammes.pdf

III.2 Diagramme de Séquence (Version 3.xx)



IV. Développements et Maintenances correctives, évolutives

VI.1 Mise en place des design patterns GRASP et Strategy

Mis en place du développement depuis la version 2.0 du dilemme des prisonniers

VI.1.1 Implémentation actuelle

Actuellement la disposition des classes utilisées dans l'application telle que décrite dans le diagramme de classe de la version 1.0 ne respecte pas particulièrement des types design patterns.
L'objectif de la première version a été en priorité de fournir une implémentation fonctionnelle respectant l'ensemble des spécifications demandés par l'utilisateur.

Cependant, il existe une prémisse du design pattern *Strategy* au travers de l'implémentation des stratégies car ces dernières héritent d'une classe mère ***Strategie***.

VI.1.2 Objectifs et but

Le but de ce développement est de respecter les règles établies pour les design patterns suivant :

- **Le design pattern GRASP :** Se composent de lignes directrices pour l'attribution de la responsabilité des classes et des objets.

- **Le design pattern Strategy :** De type comportemental, il est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application. Il permet de définir une famille d'algorithme, encapsuler chacun d'eux en tant qu'objet, et les rendre interchangeables.

VI.1.3 Développement et modifications

Pour respecter le design pattern *Strategy*, on a ajouté une interface ***IStrategie*** donnant la signature de l'ensemble des méthodes présent dans les stratégies et dans la classe abstraite ***Strategie*** :

On a également modifié l'ensemble des classes utilisant des instances de ***Strategie*** afin que ces derniers prennent l'interface ***IStrategie*** permettant ainsi d'avoir un code générique.
Cependant la classe Abstraite ***Strategie*** n'a pas été supprimé afin d'éviter la duplication de code dans les différentes stratégies implémentées

Pour le design pattern *Grasp*, l'ensemble de l'affichage est présent maintenant dans le Main de l'application, en effet avant certains affichages notamment le tableau des scores ou encore le résumé de la confrontation étaient effectués directement par la classe Tournoi.

VI.2 Séparer l'IHM du programme principal

Mise en place de développement depuis la version 2.5 du dilemme des prisonniers

VI.2.1 Implémentation actuelle

Actuellement plusieurs implémentations dans l'application posent ou peuvent poser problème.
Dans un premier temps l'implémentation de l'IHM se trouve directement dans le Main de l'application.

L'objectif de ce développement est de régler les différents points à améliorer. Pour cela nous allons réaliser les développements et modification suivantes :

- Séparation des méthodes front présent dans le Main dans une classe à part afin que toutes les parties soient dissociés (Back, Front, et Main)

VI.2.3 Nouvelle implémentation

Pour séparer la partie front du Main, l'ensemble des fonctions présentes dans ce dernier a été déplacées dans une nouvelle classe TournoiFrame respectant le design pattern singleton car on n'a besoin qu'une d'une seule instance de cette classe pour l'affichage

VI.3 Suppression de la fonction clone() de l'interface Stratégie

Mise en place du développement depuis la version 2.5 du dilemme des prisonniers

VI.3.1 Implémentation actuelle

Actuellement afin de respecter la spécification sur les confrontations qui est : « *Une stratégie se confronte aux autres stratégies* ***y compris elle-même*** »
L'interface IStrategie hérite de l'interface Cloneable fourni avec JAVA permettant ainsi d'avoir la signature de la fonction clone().

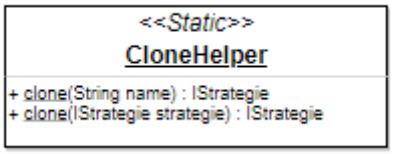
En effet cette dernière nous permet, lors de l'initialisation du Tournoi et des Confrontations de créer une copie de la stratégie afin de pouvoir dissocier les scores de chacune mais aussi les coups joués par l'adversaire qui sont enregistrées

L'objectif de la suppression des clones est de rendre l'interface IStrategie commune avec l'ensemble des implémentations effectuées par le second binôme avec lequel nous travaillons. (cf. Informations complémentaires) Leur implémentation de la spécification citée ci-dessus (cf. section VI.3.1) n'utilise pas de méthode clone ou équivalent. Ce dernier passe par une classe intermédiaire qui enregistre le score.

Cependant, il nous est impossible de supprimer le fonctionnement de la méthode clone() sans modifier le fonctionnement de la classe Tournoi. Il faut donc dans un premier temps trouver une nouvelle implémentation afin de palier la suppression de la méthode clone dans l'interface sans modifier la classe Tournoi.

VI.3.3 Nouvelle implémentation

Afin de palier à cette suppression, il a été décidé d'ajouter une nouvelle classe static qui se chargera d'effectuer le clone de la stratégie. Cette classe nommé CloneHelper, propose deux méthodes clone, une prenant le nom de la stratégie à cloner et l'autre prend directement la IStrategie.
Il s'agit d'une implémentation effectuée avec beaucoup d'attention car se développement touche directement au fonctionnement de l'application complète ce qui peut nuire gravement à ce dernier.



VI.3.4 Ce que nous ferons plus tard

Il faudra peut-être envisager une implémentation différentes pour les fonctions clone(), car ces dernières fait des tests sur le type d'instance de l'objet, ce qui empêche la généricité de l'application étant donnée qu'il faut connaître le type de stratégies utilisées dans le code.
En effet, si de nouvelles stratégies sont à implémenter il faudra alors modifier l'implémentation du CloneHelper pour les prendre en compte.

VI.4 Développement d'une interface graphique

Mise en place du développement depuis la version 3.0 du dilemme des prisonniers

VI.4.2 Implémentation actuelle

Actuellement l'ensemble des affichages permettant l'utilisation de l'application se font au travers de la console. Ce qui peut être dans certain cas compliqué à utiliser, notamment lors de l'affichage de données complexes comme par exemple l'affichage du tableau des scores des stratégies.

Le but de ce développement est donc de créer une interface graphique utilisable par l'utilisateur afin d'afficher les différentes informations dans un format agréable à lire mais également simplifier l'utilisation de l'application au travers de boutons et autres fonctionnalités.

VI.4.3 Nouvelle implémentation

Création de la version 3.xx qui contiendra ce nouveau développement d'interface graphique (les versions 2.xx resteront en affichage console mais vont se rapprocher au fur et à mesure au niveau fonctionnement des versions 3.xx)
Ensuite, création de deux nouvelles classes ***JDilemme*** et ***JParamTournoi*** qui utiliseront le package javax.swing afin de créer les fenêtres et leur différents composants (JButton, etc…) :

- ***JDilemme*** sera la fenêtre principale est servira à l'affichage des résultats du tournoi ainsi qu'à l'accès des paramètres pour l'utilisateur
- ***JParamTournoi*** sera une seconde fenêtre dédiée à l'initialisation des paramètres du Tournoi

Etant donnée les nombreux morceaux de code commun entre les deux classes (notamment la création de la fenêtre ou son affichage) une classe abstraite ***FrameBase*** a été créée afin d'éviter la duplication de code.

Les deux classes respectent le design pattern Singleton car nous aurons besoin qu'une d'une seule instance des fenêtres lors de l'utilisation de l'application.

VI.5 Mise en place du design pattern Observer

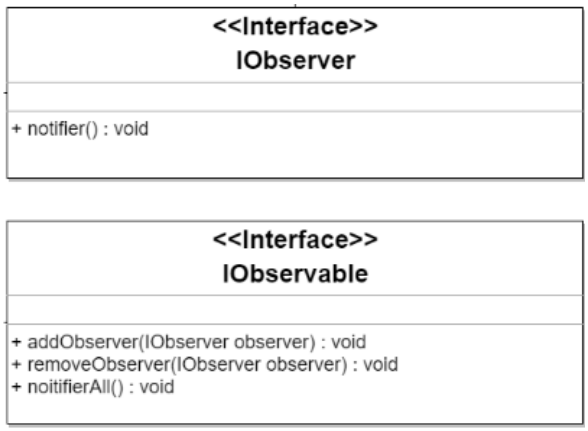
VI.5.1 Implémentation actuelle

Actuellement les deux versions en cours de développement (2.5 et plus et 3.xx) effectuent l'initialisation et l'affichage dans des classes appart dédiées exclusivement à la partie front-end de l'application or ces dernières ont été implémentées de manière rapide sans essayer de faire quelque chose de propre et performant.

Le but est dont d'appliquer le design pattern Observer au front-end des deux versions afin d'obtenir un code plus structuré mais également rendre les deux versions (2.5 et plus et 3.xx) quasiment identique dans leur fonctionnement. La seule chose qui les différenciera sera le type d'affichage (console et graphique), ce qui permettra également d'ajouter de nouvelles fonctionnalités ou corriger des bugs plus facilement.

VI.5.3 Nouvelle implémentation

Pour pouvoir respecter le design pattern Observer, deux interfaces ont été ajoutées afin de décrire le comportement de l'observable et de l'observer, tel que montré ci-dessous :



Une fois les interfaces et le comportement décrit, il a fallu implémenter ces dernières avec les classes déjà existantes.

Pour la version 2.5 et plus :

- Création d'une classe ***ParamFrame*** (respectant le design pattern Singleton) effectuant l'initialisation des paramètres pour la création du Tournoi et qui implémente l'interface ***IObservable***. C'est ce dernier qui se chargera de notifier l'affichage principale lorsque l'initialisation sera terminée.
- Modification de la classe ***TournoiFrame*** qui implémente l'interface ***IObserver*** et qui s'occupera de la création du Tournoi et de son affichage lorsque ***ParamFrame*** le notifiera de la fin de l'initialisation des paramètres

Pour la version 3.xx :

- Modification de la classe ***JParamTournoi*** qui implémente l'interface ***IObservable*** et qui aura le même comportement que ***ParamFrame***
- Modification de la classe ***JDilemme*** qui implémente ***IObserver*** et qui aura le même comportement que ***TournoiFrame***, ce qui permet de ne pas avoir de lien direct en les deux fenêtres.
  Avant nous devions associé l'action de bouton confirmer de ***JParamTournoi*** dans ***JDilemme*** pour pouvoir récupérer les informations au bon moment

VI.6 Mise en place du design pattern Façade

VI.6.1 Implémentation actuelle

Dans notre application, la partie front-end est en contact direct avec la partie back-end en faisant appel aux différentes fonctions disponibles (création tournoi, confrontation, etc…) pouvant être suivant les cas difficiles à manipuler ou simple mais en prenant de nombreuses précautions.

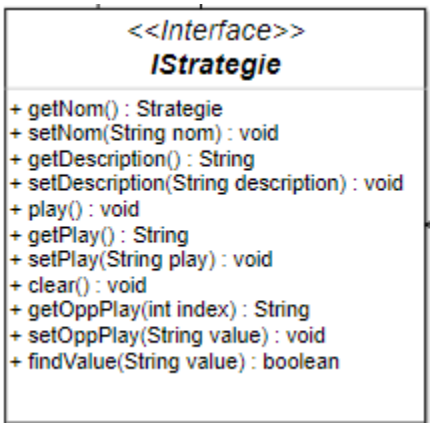
L'idée est donc de séparer totalement la partie front et back via une classe statique qui ferait office d'intermédiaire lors de la communication entre les deux afin de faciliter pour le front l'utilisation des fonctions disponibles dans le back.

VI.6.3 Nouvelle implémentation

VI.7 Modification de l'interface et Intégration des stratégies du second binôme

VI.7.1 Implémentation actuelle

Actuellement l'interface IStrategie permettant de respecter le design pattern Strategy se présente de la manière suivante :



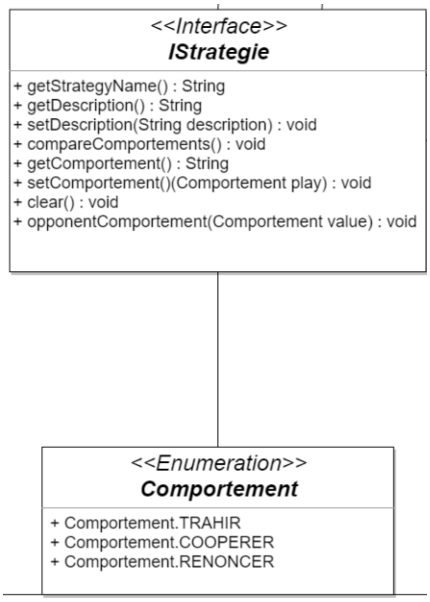
Cette dernière permet d'avoir une application générique car cette dernière peut utiliser toute classe implémentant cette interface.
Maintenant, en collaboration avec un autre binôme (cf. Informations complémentaires) nous devons utiliser les stratégies implémentées provenant de leur application dans la nôtre.

L'objectif est donc dans un premier temps de se mettre d'accord sur une interface IStrategy commune qui sera utilisée par les deux applications, et modifier le code en conséquence.
Puis il faudra récupérer sous la forme d'un jar non exécutable, les stratégies du second binôme et les intégrer dans notre application afin que ces dernières puissent être utilisées par l'utilisateur avec nos propres stratégies

Faire attention, car certains changements vont surement impacter grandement l'implémentation de l'application, il faudra donc re-tester avec précautions toutes les méthodes

VI.7.3 Nouvelle implémentation

Après plusieurs concertations, une nouvelle implémentation de l'interface IStrategy a été mise en place :



L'ensemble de ces changements ont été effectuées car le second binôme n'ayant pas de nombreuses stratégies implémentées, n'utilisaient pas la moitié des fonctions présentes précédemment dans l'interface ***IStrategie***.
Pour régler le problème de notre côté tout en gardant nos implémentations fonctionnelles, l'ensemble des signatures ont été déplacés dans la classe abstraite ***Strategie.***

Un autre changement qui a dû être accepté dans l'interface a été l'ajout d'une énumération ***Comportement*** qui a été effectué par le second binôme car la suppression aurait effectué beaucoup trop de changement dans leur application, contrairement au nôtre qui ne faisait que changer le type String en Comportement

Les signatures des méthodes présentes ont également changé en faveur de nom plus indicatif de leur fonction :

- getPlay() et devenu getComportement
- play() devient compareComportement() (on perd ici la notion de je joue cette action)
- setOppPlay(String play) devient opponentComportement(Comportement play)
- getNom() deviant getStrategyName()

Et enfin la suppression de la fonction clone() qui a obligé la création d'une nouvelle classe statique (cf. Suppression de la fonction clone de l'interface IStrategy)

Il a fallu également modifier de nombreuses classe de la partie back-end et front-end pour prendre en compte ces modifications et l'ajout des stratégies du second binôme

Pour la partie Back-end :

- Modification de la classe ***CloneHelper*** afin de prendre en compte dans les tests les instances des nouvelles stratégies. Cette modification a d'ailleurs montré un problème au niveau de la fonction *clone()*, en effet la généricité de l'application n'est plus possible car nous devons connaître les stratégies pour faire les tests des instances. On pourrait utiliser la méthode utilisant le code de la stratégie (cf. le nom) mais étant donné que le nom des stratégies sont identiques cela est impossible.
  Il faudra donc modifier le code pour le rendre de nouveau générique (Ce développement a déjà été commencé est se trouve dans la classe ***CloneHelper***)
- Modification de la classe ***ApiDilemme*** afin d'ajouter des fonctions de création pour les nouvelles strategies mais également l'ajout de ces dernières dans la fonction *createListStrategie(List<Integer>)* utilisée par le front-end
- Modification de la classe ***Confrontation*** afin de changer le type String du comportement de la stratégie en type Comportement
- Modification de la classe ***Strategie*** afin de changer le type String du comportement de la stratégie en type Comportement. Modifications également présentes dans les classes filles héritant de ***Strategie***

Pour la partie Front-end :

- Modification des affichages pour la partie initialisation des paramètres pour rajouter les stratégies du second binôme
    - ***Pour la version console*** : Ajout simple de texte lors de l'affichage des stratégies disponibles
    - ***Pour la version graphique*** : Ajout de nouveaux JCheckBox pour ajouter les stratégies