RAPPORT TECHNIQUE
PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
DANS LE CADRE DU COURS LOG792 PROJET DE FIN D'ÉTUDES EN GÉNIE LOGICIEL

LIAR BOT

PIERRE LUC CARRIER
CARP23118809

DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI

Professeur superviseur

Alain April

MONTRÉAL, 12 AVRIL 2012
HIVER 2012

REMERCIEMENTS

Je ne pourrais passer sous silence la contribution de Yohann Carrier au projet Liar Bot; sa grande expérience avec le jeu Liar's Dice a été très précieuse lors de la conception de l'agent intelligent du système et la générosité avec laquelle il a offert son temps a grandement contribué au succès du projet.

J'aimerais également remercier Alain April pour sa supervision et ses conseils tout au long du projet.

LIAR BOT

PIERRE LUC CARRIER
CARP23118809

RÉSUMÉ

Les jeux partiellement observables, où aucun des joueurs ne connaît entièrement l'état de la partie, et non déterministes, où certains éléments relèvent du hasard, nécessitent bien souvent des joueurs qu'ils soient aptes à analyser le comportement de leurs adversaires pour acquérir de l'information additionnelle sur l'état du jeu et à bluffer pour mener leurs adversaires à former de fausses déductions. Ces deux compétences sont complexes à implémenter ce qui explique qu'il soit ardu de réaliser des agents intelligents capables de jouer à de tels jeux avec un niveau de compétence satisfaisant. Le présent projet a pour objectif de mettre au point un agent intelligent apte à jouer au jeu Liar's Dice avec un niveau de compétence comparable à un joueur de calibre intermédiaire. Il a également eu pour but de concevoir une application permettant à un utilisateur de jouer contre l'agent précédemment mentionné.

Vis-à-vis la conception de l'agent, l'approche de modéliser ses adversaires au moyen de réseaux bayésiens ou de réseaux de neurones pour lui permettre de choisir ses actions en conséquence a été considérée. Toutefois, cette possibilité a été écartée, car elle nécessite un grand volume de données qui n'est pas disponible dans le cadre de ce projet. La solution qui a été retenue est d'évaluer chacune des actions pouvant être posée en effectuant une somme pondérée de divers critères d'analyse et de sélectionner l'action à poser parmi celles qui ont obtenu les résultats les plus élevés. Dans cette dernière approche, les pondérations des différents critères sont calibrées à travers la technique d'optimisation connue sous le nom d'algorithmes génétiques.

L'approche retenue a permis de construire un agent intelligent dont la performance s'approche de celles de joueurs de niveau intermédiaire sans toutefois l'atteindre. Pour améliorer davantage les performances de l'agent intelligent, il est conseillé d'utiliser l'agent qui a été construit pour jouer contre plusieurs utilisateurs et ainsi récolter un ensemble suffisant de données pour implémenter la fonctionnalité de modélisation de joueurs initialement considérée.

Mots-clés :

Liar's Dice, agent intelligent, algorithmes génétiques, modélisation de joueurs.

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

INTRODUCTION

Dans le contexte des jeux électroniques, les agents intelligents ont connu beaucoup de développement et d'améliorations lors des dernières années. Il en résulte que, dans beaucoup de jeux certains agents intelligents se comparent avantageusement aux meilleurs joueurs humains. Toutefois, les jeux partiellement observables, où aucun des joueurs ne connaît entièrement l'état de la partie, et non déterministe, où certains éléments relèvent du hasard, continuent de poser défi. Dans ce type de jeu, chaque joueur tente d'acquérir plus d'informations sur l'état du jeu en décodant les actions des autres joueurs tout en tentant d'agir de façon à induire ses adversaires en erreur. Ces deux comportements sont difficiles pour un agent intelligent, car ils impliquent des déductions et raisonnements très complexes. Il en résulte que ces jeux, partiellement observables et non déterministes, sont bien souvent plus favorables aux humains qu'aux agents intelligents.

L'objectif du projet est de concevoir et développer un agent intelligent apte à jouer au Jeu Liar's Dice. Cet agent doit prendre ses décisions avec un niveau de compétence comparable à celui d'un joueur intermédiaire, c'est-à-dire initié au jeu et avec un peu d'expérience sans toutefois être un expert. Ce projet vise également la création d'une interface graphique permettant à un joueur humain de jouer contre une ou plusieurs instances de l'agent intelligent. Cette interface permet de tester la performance de l'agent intelligent développé dans le cadre de ce projet, lorsque celui-ci est confronté à des joueurs humains.

Dans un premier temps, une revue de la documentation explique plusieurs travaux réalisés dans le domaine des jeux partiellement observables et non déterministes ainsi que dans celui des algorithmes génétiques. Ensuite, la conception de l'application permettant de jouer contre l'agent intelligent est présentée, suivie de la conception de l'agent lui-même. Finalement, les résultats des tests de performance de l'agent intelligent contre des joueurs humains sont analysés et des recommandations quant à l'enlignement futur du projet sont réalisées.

CHAPITRE 1

REVUE DE LA DOCUMENTATION

1.1 Agents intelligents dans des jeux partiellement observables et non déterministes

Par comparaison avec les jeux complètement observables (chaque joueur connait l'état complet du jeu) et déterministes (aucun hasard dans le déroulement du jeu), peu de travaux ont été réalisés sur le développement d'agents intelligents pour les jeux partiellement observables et non déterministes.

Nicholas V. Findler a conduit des recherches dans le but de mieux comprendre le processus décisionnel des humains dans des conditions d'incertitude et de développer des techniques pour modéliser ce processus, permettant à des systèmes informatisés de l'imiter pour obtenir des résultats comparables aux humains (Findler, 1977, p. 1). Il a réalisé ses recherches dans le cadre du jeu de poker, car celui-ci partage plusieurs caractéristiques avec les problèmes de la vie courante, notamment la nécessité de prendre une décision sans avoir accès à toute l'information pertinente et la possibilité, pour un joueur, d'obtenir des informations sur la situation des autres joueurs et de donner de fausses informations sur sa propre situation (Findler, 1977, p. 2). Toutefois, ces travaux n'ont jamais permis de produire un agent intelligent capable de battre des joueurs de niveau expert (Richard Papp, 1998, p. 2).

Des travaux ont également été réalisés pour utiliser des réseaux de neuronnes pour modéliser les stratégies des adversaires, au jeu de poker, dans le but de mieux s'y adater. Les données utilisées pour entraîner les réseaux de neurones proviennent de parties de poker précédemment jouées sur le réseau IRC Poker. Dans le cas d'un adversaire dont la stratégie est statique, le modèle proposé arrive à prédire la prochaine action du joueur avec une fiabilité pouvant atteindre 90 %. Dans le cas d'un adversaire dont la stratégie est dynamique, la fiabilité descend jusqu'à près de 85 % (Davidson, 1999).

D’autres travaux ont été réalisés pour modéliser le comportement des joueurs humains, mais cette fois-ci, au moyen de réseaux bayésiens (Southley, 2005, p. 1). Ces travaux ont permis de mettre au point un agent dont la performance, dans les cas optimaux, est comparable aux meilleurs modèles développés précédemment et qui est capable de modéliser et exploiter avec succès les stratégies de ses adversaires après 200 tours (Southley, 2005, p. 9).

1.2 Algorithmes génétiques

Les algorithmes génétiques sont une technique d’optimisation stochastique, c’est-à-dire une technique qui repose sur certains éléments aléatoires pour maximiser ou minimiser une fonction donnée. Cette technique, décrite plus en détail à l’annexe XIII, a été proposée par John Holland en 1970 (Russel, 2010, p. 128). Cette technique offre souvent de bonnes performances, mais d’autres travaux demeurent à ce jour nécessaires pour identifier les conditions sous lesquelles les algorithmes génétiques sont le plus adéquats (Russel, 2010, p. 129).

Des travaux réalisés par Eric Beaum, Dan Boneh et C.Garret ont permit de démontrer que la méthode connue sous le nom de « culling » permet d’obtenir une convergence beaucoup lus rapide que la méthode initialement proposée par Holland. La méthode du « culling » consiste, une fois l’aptitude des individus évaluée, à supprimer ceux dont l’aptitude est inférieure à un certain seuil avant d’appliquer les opérations de croisement et de mutation pour créer une nouvelle population (Russel, 2010, p. 128).

CHAPITRE 2

MÉTHODOLOGIE

2.1 Déroulement du projet

Le projet est réalisé selon une approche itérative comportant un total de trois itérations, chacune d'une durée approximative d'un mois. Chacune de ces itérations aborde l'analyse des requis du projet ainsi que la conception, l'implémentation et les tests du système, mais la proportion de l'effort allouée à chacune de ces activités varie en fonction de l'itération.

La première itération porte principalement sur l'analyse des requis du projet, c'est-à-dire leur découverte et leur documentation, et sur la conception d'une solution visant à satisfaire ces requis. L'implémentation réalisée lors de cette phase vise à produire un prototype de l'application utilisé par les utilisateurs pour jouer contre l'agent intelligent. Cette tâche se retrouve tôt dans le projet, car elle est associée à un haut niveau de risque et qu'elle est requise, lors de la réalisation de l'agent intelligent, pour évaluer les performances de ce dernier. Cette itération comprend également la rédaction de la proposition de projet.

La deuxième itération est majoritairement axée sur la conception. Malgré ceci, un travail d'analyse est tout de même réalisé pour mettre à jour les requis du projet. Lors de cette phase, l'implémentation vise à produire une ébauche de l'agent intelligent pour permettre certains tests préliminaires de sa performance. Étant donné qu'à ce stade la conception et l'implémentation sont plus stables que lors de la première itération, un plan de tests est rédigé pour prévoir et documenter les tests à réaliser sur le projet pour s'assurer de sa qualité. Cette itération culmine en la rédaction du rapport d'étape.

La troisième, et dernière, itération finalise la conception du système et porte principalement sur l'implémentation de l'agent intelligent et les tests de l'application. Les dernières

évaluations de la performance de l'agent intelligent sont réalisées puis analysées. Cette itération se termine avec la rédaction du rapport technique et la présentation finale du projet.

Les tests réalisés sur le système incluent à la fois des tests visant à s'assurer qu'il satisfait l'entièreté des exigences fonctionnelles et non fonctionnelles auxquelles il est soumis et des tests permettant de comparer les performances de l'agent intelligent développé aux performances de joueurs humains.

Les artefacts produits lors de ce projet sont placés sous gestion de configuration afin d'assurer leur intégrité ainsi que la capacité d'en récupérer des versions précédentes. Toutefois, vu la taille modeste du projet et de l'équipe de projet, les demandes de changement sont consignées à l'intérieur d'une liste de demande de changements, elle-même placée sous gestion des versions, plutôt que sur un système de gestion des tickets.

2.2 Livrables

La réalisation du projet implique la création de plusieurs livrables. Le tableau 2-1 documente les divers livrables réalisés dans le cadre du projet Liar Bot.

Tableau 2-1 Livrables du projet Liar Bot

Nom	Description
Modèle du domaine	Un modèle du domaine est un artefact d'analyse qui permet de représenter graphiquement les différents concepts appartenant au domaine du problème à résoudre.
Document de cas d'utilisation	Un document de cas d'utilisation est un artefact d'analyse qui vise à décrire le comportement fonctionnel d'un système à travers l'utilisation de cas d'utilisations. Un cas d'utilisation est une séquence cohésive d'interactions entre un système est ses utilisateurs ou des systèmes externes.

Tableau 2-1 Livrables du projet Liar Bot (suite)

Nom	Description
Document de spécification des exigences logicielles	Un document de spécifications logicielles est un artefact d'analyse qui documente les exigences fonctionnelles, non fonctionnelles ainsi que les contraintes applicables à un système.
Document de design	Un document de design est un artefact de conception qui vise à présenter les différents éléments qui composent un système logiciel, à la fois de façon statique et dynamique. Le document de design est un artefact défini dans le processus OpenUp (Eclipse Foundation, 2011).
Implémentation	L'implémentation correspond au code du système lui-même. Dans le cadre de ce projet, l'implémentation à livrer est un prototype fonctionnel du système à livrer.
Plan de test	Un plan de test est un document qui décrit les tests qui seront exécutés pour s'assurer de la qualité du système implémenté.
Implémentation des tests	Cet artefact est formé des scripts et modules de code nécessaires à l'exécution automatique des tests prévus sur le système.
Liste des demandes de changement	Ce document regroupe les différentes demandes de changement qui ont été formulées ainsi que leur état actuel (accepté, complété, etc.) et les autres propriétés permettant de les traiter adéquatement.
Proposition de projet	La proposition de projet documente un projet à réaliser en offrant une planification sommaire en décrivant brièvement les paramètres prévus pour sa réalisation.
Rapport d'étape	Le rapport d'étape décrit l'évolution d'un projet, les travaux réalisés ainsi que les ajustements apportés aux diverses prédictions formulées dans la proposition de projet.
Rapport technique	Le rapport technique détaille les aspects techniques d'un projet. Ces aspects incluent notamment une revue de la documentation, la conception du projet et l'analyse des résultats du projet.

2.3 Outils utilisés

La réalisation du projet passe par l'utilisation de plusieurs outils dont les utilités vont de la création des livrables à la sauvegarde de leur intégrité. Les listes suivantes indiquent les principaux outils utilisés dans le cadre de ce projet.

Les outils suivants sont utilisés pour générer les divers livrables du projet :

- Microsoft Word 2000 : Microsoft Word est un outil de traitement de texte permettant de créer et modifier le fond et la forme de documents textes.
- Microsoft Excel 2000 : Microsoft Excel est un chiffrier numérique. Dans le cadre du projet, cet outil est employé pour faire la gestion de la liste de demandes de changement
- Eclipse : Eclipse est un environnement de développement hautement configurable pour des projets faisant usage du langage de programmation Java.
- UMLet : UMLet est un module complémentaire pour l'environnement Eclipse qui permet de créer et modifier des diagrammes respectant la notation UML (Unified Modeling Language). Cet outil est donc employé pour produire le modèle du domaine, les diagrammes de cas d'utilisations et autres diagrammes associés au projet.

Les outils suivants sont utilisés pour contrôler et assurer la qualité des livrables :

- Checkstyle : Checkstyle est un module complémentaire pour l'environnement Eclipse qui détecte, dans un module de code, les accrocs aux normes de programmation associées au langage de programmation Java.
- EclEmma : EclEmma est un module complémentaire pour l'environnement Eclipse qui permet d'évaluer la couverture des tests automatisés.
- Antidote : Antidote est un logiciel d'aide à la rédaction qui inclut des dictionnaires, guides linguistiques et un correcteur automatique.

Les outils suivants sont utilisés pour faire la gestion des versions des différents artefacts du projet :

- Beanstalk : Beanstalk est une application Web permettant la création, la configuration et la gestion d'un dépôt de données SVN.
- TortoiseSVN : TortoiseSVN est un client SVN et permet, à ce titre, d'accéder à un dépôt SVN et d'en modifier le contenu.

2.4 Contraintes applicables au projet

Le projet Liar Bot est assujetti aux deux contraintes suivantes :

- L'interface utilisateur du système Liar Bot ne doit pas être d'une taille supérieure à 1024 par 768 pixels.
- Le système Liar Bot ne doit pas nécessiter l'accès à un serveur externe de bases de données tel qu'Oracle, Microsoft SQL Server, MySQL, etc. Toutefois, les bases de données ne nécessitant pas de serveur de bases de données, tel que SQLite, sont permises.

CHAPITRE 3

PROCESSUS DE CONCEPTION

3.1 Interface utilisateur et architecture

3.1.1 Problème

Le problème initial à résoudre consiste à mettre au point l'application offrant aux utilisateurs l'option de jouer au jeu Liar's Dice contre l'agent intelligent offert par le système Liar Bot. Ce problème est composé de deux sous-problèmes à savoir concevoir l'interface utilisée par les utilisateurs pour interagir avec le système et mettre au point l'architecture de l'application.

3.1.2 Hypothèses

Les hypothèses suivantes sont formulées quant à la façon dont les utilisateurs du système Liar Bot sont amenés à utiliser ce dernier :

- L'utilisateur du système Liar Bot dispose d'un clavier et d'une souris ou de tout autre dispositif capable de remplir les mêmes fonctions de façon transparente au système.
- Une seule instance du système Liar Bot est en utilisation à la fois.

3.1.3 Solutions considérées et choix

En ce qui concerne l'interface usager, deux principales approches sont disponibles : une interface graphique ou encore une interface de type ligne de commande. Une interface de type ligne de commande est très simple à implémenter et demande peu de ressources. De plus, ce type d'interface est généralement très efficace entre les mains d'utilisateurs expérimentés. Toutefois, l'usabilité est faible pour les nouveaux utilisateurs. Par opposition, les interfaces graphiques sont quelquefois complexes et peuvent demander une quantité non

négligeable de mémoire et de puissance de calcul. Malgré cela, une interface graphique bien conçue offre une très bonne usabilité à tous les utilisateurs, peu importe leur niveau d'expérience. Étant donné qu'à l'intérieur de l'application Liar Bot, l'usabilité est plus importante que la performance, le choix se porte tout simplement vers une interface graphique.

L'utilisation d'une interface graphique implique la nécessité de traiter des événements générés par les actions de l'utilisateur sur l'interface. Ceci amène à considérer une architecture de type MVC (Modèle-Vue-Contrôlleur) qui permet de découper le code de façon à minimiser le couplage et maximiser la modificabilité. Dans l'architecture MVC, les classes sont réparties en trois groupes :

- Modèle : Les classes qui composent le modèle représentent les données de l'application.
- Vue : Les classes qui composent la ou les vues du système sont responsables de l'affichage des données du modèle à l'utilisateur.
- Contrôleur : Les classes qui composent le contrôleur

Il est également possible d'utiliser une architecture en couche. Ce type d'architecture est fondé sur trois principales contraintes :

- Un ensemble de couches est défini selon une structure hiérarchique; à l'exception des couches aux extrémités, chaque couche est au-dessus d'une autre et en dessous d'une troisième.
- Chaque élément de code est assigné à une couche et plusieurs éléments peuvent être assignés à une même couche.
- Les éléments d'une couche ne peuvent seulement utiliser les services des autres éléments de la même couche et ceux appartenant à la couche directement sous la leur.

Tout comme l'architecture MVC, l'architecture en couche promeut le découplage, la modificabilité et la réutilisabilité. Toutefois, étant donné le grand nombre d'interrelations entre certains éléments du système Liar Bot, l'utilisation de l'architecture en couches

résulterait en un faible nombre de très grandes couches ce qui limiterait les bénéfices de ce style architectural. À ce titre, le choix se porte sur l'architecture MVC.

3.1.4 Conception résultante

Cette section détaille la conception qui résulte des décisions conceptuelles réalisées et décrites dans la section précédente.

3.1.4.1 Modules de l'application Liar Bot

L'application Liar Bot comporte trois principaux modules. Le tableau 3-1 énumère et documente, pour chacun, ses responsabilités les plus importantes :

Tableau 3-1 Modules du Liar Bot et leurs responsabilités

Module	Responsabilités
GameData	Ce module a comme responsabilité de gérer les données utilisées par le système, notamment celles utilisées pour représenter les parties de Liar's Dice, et d'assurer leur cohérence et leur intégrité. Ce module constitue le modèle de l'architecture MVC choisie pour l'application Liar Bot.
AI	Ce module implémente le ou les agents intelligents offerts par le système. Il contient aussi les différents outils employés par les agents intelligents pour prendre leurs décisions.
LiarBot	Ce module implémente la vue et le contrôleur du système LiarBot. Ceci implique la création et la gestion d'une interface utilisateur en plus de faire la gestion du déroulement des parties de Liar's Dice. Ce module contient à la fois la vue et le contrôleur de l'architecture MVC de l'application Liar Bot.

Les sous-sections suivantes offrent plus de détails sur les structures et fonctionnements respectifs des modules GameData, AI et LiarBot

3.1.4.1.1 Module GameData

Tel que précédemment mentionné, les classes qui composent ce module représentent les données gérées par le système Liar Bot. La figure I-1, à l'annexe I, illustre les attributs, responsabilités et associations de ces différentes classes sous forme d'un diagramme de classes UML. Le tableau 3-2 énumère ces classes et détaille les responsabilités de chacune :

Tableau 3-2 Classes du module GameData

Nom	Description et responsabilités
Die	Cette classe encapsule le concept de dés à jouer. Une instance de la classe Die représente donc un simple dé à jouer et offre des méthodes permettant de « rouler » le dé pour changer aléatoirement sa valeur ainsi que des méthodes permettant de consulter la valeur du dé.
Game	Cette classe est utilisée pour représenter une partie du jeu Liar's Dice. Elle encapsule les règles associées au jeu Liar's Dice. Elle est également responsable de gérer les données relatives à une partie de Liar's Dice et en assurer l'intégrité et la cohérence.
Move	Cette classe est utilisée pour représenter les différents coups joués par les différents joueurs pendant une partie. L'encapsulation de ce concept à l'intérieur d'une classe permet d'aisément constituer un historique des coups d'une partie.

Tableau 3-2 Classes du module GameData (suite)

Nom	Description et responsabilités
MoveType	Cette énumération représente les différents types de coups qui peuvent être joués par un joueur dans une partie de Liar's Dice. On y retrouve trois types de coups : • Un joueur réalise un pari (BET) • Un joueur accuse un autre joueur de mentir (BLUFF) • Un joueur prétend que le pari du dernier joueur est exact (SPOT_ON)
Player	Cette classe représente un joueur, humain ou agent intelligent, impliqué dans une partie de Liar's Dice.

Cette structure de classes a été choisie parce qu'elle s'approche le plus possible de la structure entre les concepts du monde réel, à savoir des joueurs qui ont des dés et qui participent à une partie de Liar's Dice. Cette similitude rend la conception du module GameData plus instinctive et accélère la courbe d'apprentissage des développeurs qui doivent y travailler ce qui en améliore la modificabilité et la maintenabilité.

3.1.4.1.2 Module AI

Les classes qui constituent ce module représentent l'agent intelligent du système Liar Bot ou encore constituent des utilitaires utilisés par celui-ci pour réaliser sa prise de décisions. La figure II-1, à l'annexe II, illustre les attributs, responsabilités et associations de ces différentes classes sous forme d'un diagramme de classes UML. Le tableau 3-3 énumère ces classes et détaille les responsabilités de chacune.

Tableau 3-3 Classes du module AI

Nom	Description et responsabilités
AICore	Cette classe représente le moteur de décision d'un agent intelligent. Sa responsabilité est, à partir d'informations complètes sur l'état d'une partie de Liar's Dice, de produire un jugement sur l'action que le joueur actif de la partie devrait accomplir.
Probabilities	Cette classe a la responsabilité de réaliser des calculs probabilistes et statistiques pour le compte d'autres classes.

La centralisation des fonctionnalités de calcul probabilistes à l'intérieur de la classe Probabilities et non pas de la classe AICore, bien que cette dernière soit présentement la seule à nécessiter ces fonctionnalités, est issue de considérations de réutilisabilité. En effet, à cette étape de la conception, le module AI et les différentes classes qui le composent sont incomplets, car l'agent intelligent n'a pas encore été conçu. Il est donc possible, dans une prochaine itération de la conception, que d'autres classes doivent réaliser des calculs de probabilité. En centralisant ces calculs dans la classe Probabilities, il est plus aisé et rapide d'accommoder ces potentiels besoins futurs.

3.1.4.1.3 Module LiarBot

Les classes de ce module représentent la vue et le contrôleur du système Liar Bot. Ensemble, elles composent l'interface graphique offerte à l'utilisateur et assurent la gestion des interactions entre l'utilisateur et le système. La figure III-1, à l'annexe III, illustre les attributs, responsabilités et associations de ces différentes classes sous forme d'un diagramme de classes UML. Le tableau 3-4 énumère ces classes et détaille les responsabilités de chacune :

Tableau 3-4 Classes du module LiarBot

Nom	Description et responsabilités
LiarDice	Cette classe est le gestionnaire principal de l'application Liar Bot. Elle est responsable d'instancier l'interface usager et gère les interactions entre l'utilisateur et ce dernier.
MakeBetListener	Cette classe implémente l'interface ActionListener de Java et a la responsabilité d'informer une instance de la classe LiarDice que l'utilisateur de l'application désire faire un pari.
NewGameListener	Cette classe implémente l'interface ActionListener de Java et a la responsabilité d'informer une instance de la classe LiarDice que l'utilisateur de l'application désire créer une nouvelle partie.
CallBluffListener	Cette classe implémente l'interface ActionListener de Java et a la responsabilité d'informer une instance de la classe LiarDice que l'utilisateur de l'application désire accuser le joueur précédent de bluff.
CallSpotOnListener	Cette classe implémente l'interface ActionListener de Java et a la responsabilité d'informer une instance de la classe LiarDice que l'utilisateur de l'application désire annoncer le pari du joueur précédent comme étant exact.
MainBoardFrame	Cette classe est responsable de l'affichage la fenêtre principale du système qui inclut le menu principal et les différents éléments graphiques associés à la partie en cours.
GamePanel	Cette classe est responsable de l'affichage et de la gestion du panneau central de l'interface qui permet à l'utilisateur de choisir ses actions (réaliser un pari, contester un pari, etc.).

Tableau 3-4 Classes du module LiarBot (suite)

PlayerPanel	Cette classe abstraite fournit des méthodes ainsi qu'une interface à respecter aux classes qui gèrent l'affichage des panneaux affichant les informations d'un joueur à l'écran.
PlayerPanelHorizontal	Cette classe est responsable de l'affichage et de la gestion d'un panneau horizontal affichant les informations d'un joueur à l'écran.
PlayerPanelVertical	Cette classe est responsable de l'affichage et de la gestion d'un panneau vertical affichant les informations d'un joueur à l'écran.
NewGameDialog	Cette classe est responsable de l'affichage et de la gestion du dialogue permettant à l'usager de créer une nouvelle partie.
ImageLoader	Cette classe est responsable du chargemement des images utilisées par les différentes classes composant l'interface graphique du système.

L'interface utilisateur du système a été conçue dans le but de maximiser l'usabilité et d'accélérer le plus possible la courbe d'apprentissage des nouveaux utilisateurs. Dans un premier temps, le menu principal imite autant que possible les interfaces des programmes Windows, auxquels la plupart des usagers sont habitués. Pour ce faire, les options de création d'une nouvelle partie et de fermeture de l'application sont rassemblées sous l'onglet « File » du menu principal. Dans un deuxième temps, la répartition, dans l'interface, des panneaux qui affichent les informations des joueurs est pensée de façon à évoquer une partie de Liar's Dice jouée autour d'une table, dans le monde réel. L'interface représente la table, les joueurs sont disposés autour de la table en sens horaire, l'utilisateur a ses propres dés devant lui et, au centre de la table, se trouvent les options permettant à l'utilisateur de jouer ses coups. Les figures A-1 et A-2, disponible en appendice A, présentent des captures d'écran du prototype d'interface utilisateur résultant de cette conception.

De plus, les éléments composant l'interface ont été modularisés en un grand nombre de classes. Ceci a pour effet d'améliorer la réutilisabilité de ces éléments. En effet, l'interface créée à cette étape n'est qu'un prototype élaboré dans le but de valider l'usabilité de l'application et permettre de tester l'agent intelligent du système Liar Bot contre des humains. Il en résulte que l'interface est amenée à potentiellement changer grandement. Dans cette éventualité, la grande modularité des classes qui composent l'interface permet d'aisément réutiliser seulement les composants adéquats et de remplacer les autres. Additionnellement, cette pratique a pour effet de réduire la redondance à l'intérieur du code.

Finalement, la conception proposée implémente une légère variation vis-à-vis la patron MVC classique. Dans la version originale du patron, lorsque l'utilisateur réalise une action sur la vue, cette dernière appelle la fonction appropriée du contrôleur pour l'informer de l'événement. Dans la version proposée, le contrôleur (la classe LiarBot) instancie des objets (CallBluffListener, CallSpotOnListener, MakeBetListener et NewGameListener) et les passe ensuite à la vue (gérée par la classe MainBoardFrame) qui les assigne directement à ses objets graphiques, boutons et menus. Lorsque l'utilisateur réalise une action sur l'interface, l'objet correspondant est informé et ce dernier la méthode appropriée du contrôleur. Cette approche réduit la dépendance de la vue sur l'interface du contrôleur et facilite ainsi la modification de ce dernier.

3.1.4.2 Réalisation des cas d'utilisation

Cette section décrit comment les différents cas d'utilisation identifiés lors de la phase d'analyse du projet sont réalisés à l'intérieur de la conception proposée.

3.1.4.2.1 Cas d'utilisation LBOT-UC-001 : Créer une nouvelle partie

Ce cas d'utilisation permet à un utilisateur du système Liar Bot de créer une nouvelle partie de Liar's Dice en spécifiant les propriétés des différents joueurs impliqués dans cette partie.

La figure IV-1, à l'annexe IV, présente un diagramme de séquences UML qui détaille les objets impliqués ainsi que leurs interactions qui permettent la réalisation de ce cas d'utilisation. Les étapes suivantes expliquent ce diagramme et offrent des informations supplémentaires lorsque requises :

1) Une instance de NewGameListener, précédemment attachée à la vue de l'application, est informée de l'occurrence d'un événement demandant la création d'une nouvelle partie (la sélection, par l'usager, de l'option correspondante dans le menu principal).
2) L'instance de NewGameListener délègue à l'instance de LiarDice, qui gère l'application LiarBot, la création d'une nouvelle partie.
3) L'instance de LiarDice crée une instance de NewGameDialog, un dialogue permettant de demander à l'usager les paramètres qu'il souhaite pour la création d'une nouvelle partie. Cette dernière est ensuite affichée à l'usager qui sélectionne les options relatives à la nouvelle partie et confirme, ou non, la création de la partie.
4) L'instance de LiarDice récupère la réponse de l'usager et, si celui-ci a confirmé la création d'une partie, récupère les valeurs des différents paramètres choisis par l'usager.
5) L'instance de LiarDice crée une nouvelle instance de la classe Game avec les différents paramètres choisis par l'usager et informe l'instance de MainBoardFrame, qui assure la gestion de la vue, de la création de la nouvelle partie en lui envoyant une référence vers l'objet Game.
    - Détail : Lorsqu'elle est informée de la création d'une nouvelle partie, l'instance de MainBoardFrame crée des instances de PlayerPanel pour afficher les données des joueurs ainsi qu'une instance de GamePanel, pour faire l'affichage des actions de jeu disponibles à l'usager. Elle rafraîchit ensuite son interface avec ces nouveaux panneaux.

3.1.4.2.2 Cas d'utilisation LBOT-UC-002 : Déposer un pari

Ce cas d'utilisation permet à un utilisateur du système qui participe à une partie de Liar's Dice, lorsque c'est son tour de jouer, de formuler un pari sur le nombre de dés en jeu ayant une certaine valeur. La figure V-1, à l'annexe V, présente un diagramme de séquences UML qui détaille les objets impliqués ainsi que leurs interactions qui permettent la réalisation de ce cas d'utilisation. Les étapes suivantes expliquent ce diagramme et offrent des informations supplémentaires lorsque requises :

1) Une instance de MakeBetListener, précédemment attachée à la vue de l'application, est informée de l'occurrence d'un événement demandant la création d'un nouveau pari (l'utilisation, par l'usager, du bouton correspondant).
2) L'instance de NewGameListener délègue à l'instance de LiarDice, qui gère l'application LiarBot, la création du pari.
3) L'instance de LiarDice interroge l'instance MainBoardFrame, qui assure la gestion de la vue, pour obtenir les paramètres (nombre de dés et valeur) associés au pari que l'utilisateur désire réaliser et crée une instance de Move de type « BET » avec ces valeurs.
    - Détail : Lorsqu'interrogée, l'instance de MainBoardFrame transmet la demande à l'instance de GamePanel qui a été créée pendant le cas d'utilisation LBOT-CU-001 car c'est cette dernière qui gère les options de jeu disponibles à l'utilisateur et connaît ainsi les paramètres du pari de l'utilisateur.
4) L'instance de LiarDice interroge l'instance de Game représentant la partie en cours pour savoir si le pari proposé est un coup valide.
5) Si le pari est valide, la méthode playMove de l'instance de Game est appelée pour jouer le coup. Ensuite, l'instance de LiarDice interroge l'instance de Game pour connaître l'index du joueur auquel va le prochain tour puis informe la vue (MainBoardFrame) de se rafraîchir en conséquence.

6) Si le pari n'est pas valide, la méthode publishNotice() de MainBoardFrame est appelée pour afficher un message à l'usager l'informant de l'invalidité de son action.

3.1.4.2.3 Cas d'utilisation LBOT-UC-003 : Contester un pari

Ce cas d'utilisation permet à un utilisateur du système qui participe à une partie de Liar's Dice, lorsque c'est son tour de jouer, de contester la validité du pari formulé par le joueur précédent. La figure VI-1, à l'annexe VI, présente un diagramme de séquences UML qui détaille les objets impliqués ainsi que leurs interactions qui permettent la réalisation de ce cas d'utilisation. Les étapes suivantes expliquent ce diagramme et offrent des informations supplémentaires lorsque requises :

1) Une instance de CallBluffListener, précédemment attachée à la vue de l'application, est informée de l'occurrence d'un événement demandant la contestation d'un pari (l'utilisation, par l'usager, du boutonppp correspondant).
2) L'instance de NewGameListener délègue à l'instance de LiarDice, qui gère l'application LiarBot, la contestation du pari.
3) L'instance de LiarDice crée une instance de Move de type « BLUFF ».
4) L'instance de LiarDice interroge l'instance de Game représentant la partie en cours pour savoir si le coup proposé est valide.
5) Si le coup est valide, la méthode playMove de l'instance de Game est appelée pour officialiser le coup. Ensuite, l'instance de LiarDice interroge l'instance de Game pour connaître l'index du joueur auquel va le prochain tour puis informe la vue (MainBoardFrame) de se rafraîchir en conséquence.
6) Si le coup n'est pas valide, la méthode publishNotice() de MainBoardFrame est appelée pour afficher un message à l'usager l'informant de l'invalidité de son action.

3.1.4.2.4 Cas d’utilisation LBOT-UC-004 : Confirmer un pari

Ce cas d’utilisation permet à un utilisateur du système qui participe à une partie de Liar’s Dice, lorsque c’est son tour de jouer, de contester la validité du pari formulé par le joueur précédent. La figure VII-1, à l’annexe VII, présente un diagramme de séquences UML qui détaille les objets impliqués ainsi que leurs interactions qui permettent la réalisation de ce cas d’utilisation. Les étapes suivantes expliquent ce diagramme et offrent des informations supplémentaires lorsque requises :

1) Une instance de CallBluffListener, précédemment attachée à la vue, est informée de l’occurrence d’un événement demandant la confirmation d’un pari (l’utilisation, par l’usager, du bouton correspondant).
2) L’instance de NewGameListener délègue à l’instance de LiarDice, qui gère l’application LiarBot, la confirmation du pari.
3) L’instance de LiarDice crée une instance de Move de type « SPOT_ON ».
4) L’instance de LiarDice interroge l’instance de Game représentant la partie en cours pour savoir si le coup proposé est valide.
5) Si le coup est valide, la méthode playMove de l’instance de Game est appelée pour officialiser le coup. Ensuite, l’instance de LiarDice interroge l’instance de Game pour connaître l’index du joueur auquel va le prochain tour puis informe la vue (MainBoardFrame) de se rafraîchir en conséquence.
6) Si le coup n’est pas valide, la méthode publishNotice() de MainBoardFrame est appelée pour afficher un message à l’usager l’informant de l’invalidité de son action.

3.2 Agent intelligent

3.2.1 Problème

À ce stade de la conception, alors que l'interface utilisateur et l'architecture du système sont conçues, le problème à résoudre est celui de l'agent intelligent. Plus spécifiquement, le problème consiste à concevoir un agent intelligent dont les performances au jeu Liar's Dice peuvent égaler celles d'un joueur humain de niveau intermédiaire.

3.2.2 Hypothèses

Pour orienter l'élaboration de l'agent intelligent, plusieurs hypothèses sont faites sur la façon dont un humain approche une partie de Liar's Dice et raisonne pour sélectionner ses coups :

- Il existe une composante pseudoaléatoire, imprévisible, à l'intérieur de la stratégie de jeu d'un humain. Chez un joueur inexpérimenté, celle-ci peut provenir d'une absence de stratégie mature ou encore d'une incapacité à considérer tous les détails qui devraient guider le choix d'une action. Chez un joueur avancé, elle peut provenir d'un effort conscient pour rendre son jeu difficile à analyser. Quoiqu'en soit l'origine, il en ressort que le jeu d'un adversaire ne peut jamais être caractérisé avec une certitude absolue.
- La stratégie de jeu d'un humain évolue tout au long d'une partie, que ce soit par adaptation aux stratégies des autres joueurs ou encore du à un gain d'expérience. Une stratégie n'est donc pas statique.

3.2.3 Solutions considérées et choix

La première approche considérée pour réaliser l'agent intelligent est d'utiliser la modélisation de joueurs. Cette technique consiste à créer un modèle de chacun des autres joueurs, basé sur les observations qui ont été faites des stratégies de jeu de ces joueurs. De nombreux travaux ont démontré que l'utilisation de cette technique, dans le contexte de jeux

partiellement observables et non déterministes, offre de très bons résultats et permet d'améliorer significativement les performances d'un agent intelligent. Il est notamment possible de réaliser cette technique au moyen de réseaux bayésiens (Southley, 2005) ou encore de réseaux de neurones (Davidson, 1999).

Cette approche est très performante, mais elle impose également des contraintes. Dans le cas d'une modélisation par réseau de neurones, une grande quantité de données est nécessaire pour entrainer les réseaux de neurones (Davidson, 1999, p. 1). Toutefois, de telles données ne sont pas disponibles dans le cadre du projet. Dans le cas de la modélisation par réseaux bayésiens, des travaux réalisés sur ce type de modélisation dans le contexte du Poker ont établi que des informations sur au moins 200 manches sont requises pour que les prédictions du réseau bayésien soient fiables. Or, une partie de Liar's Dice à 5 joueurs avec 5 dés par joueurs dure, au plus, 25 manches et il n'y a aucune garantie qu'un joueur participe à plus d'une partie. Bien que le jeu Liar's Dice soit plus simple que le Poker, 25 manches semble grandement insuffisantes pour atteindre un niveau de performance adéquat. Il en ressort que la modélisation de joueurs n'est pas la meilleure option à ce stade du projet.

La deuxième approche considérée pour implémenter l'agent intelligent en est une très commune dans les algorithmes de jeu. Elle est composée de deux parties, un générateur d'actions et une fonction d'évaluation. Le générateur d'actions détermine toutes les actions pouvant être posées par l'agent intelligent à son tour et la fonction d'évaluation donne un score à chacune de ces actions selon plusieurs critères ayant différentes pondérations. L'agent intelligent pose ensuite l'action qui a reçu le plus haut score. Cette approche peut être appliquée sans problème dans le contexte actuel, mais nécessite deux améliorations pour offrir de meilleures performances :

- Les pondérations des différents critères d'évaluation, plutôt que d'être déterminés par essais-erreurs, sont obtenues à travers l'optimisation par algorithmes génétiques.
- Plutôt que de poser l'action ayant obtenu le meilleur score, l'agent intelligent choisi aléatoirement l'action qu'à poser parmi celles qui ont reçu les meilleurs scores,

conformément à l'hypothèse selon laquelle il y a une composante aléatoire dans la façon de chaque joueur.

3.2.4 Conception résultante

Cette section détaille la conception qui résulte des décisions conceptuelles réalisées quant à l'agent intelligent du système Liar Bot.

3.2.4.1 Modules de l'application Liar Bot

Par rapport à la conception résultante présentée à la précédente étape du processus de conception, plusieurs modifications et ajouts ont été appliqués. Le système Liar Bot possède maintenant deux modules additionnels, LiarBotTrainer et Persistence, et le module AI est plus développé et complexe. Le tableau 3.5 énumère et documente, pour chacun des nouveaux modules, ses responsabilités les plus importantes :

Tableau 3-5 Nouveaux modules du Liar Bot et leurs responsabilités

Module	Responsabilités
LiarBotTrainer	Ce module est responsable de l'optimisation de l'agent intelligent du système à travers l'implémentation de l'optimisation par algorithmes génétiques.
Persistence	Ce module permet l'entreposage et la récupération, sur un support persistant, des données du système pour une utilisation ultérieure.

Les sous-sections suivantes offrent plus de détails sur les structures et fonctionnements respectifs des modules LiarBotTrainer et Persistance ainsi que de la nouvelle version du module AI.

3.2.4.1.1 Module AI

Les classes de ce module constituent l'agent intelligent du système Liar Bot ou constitue des outils utilisés par cet agent intelligent pour prendre ses décisions. La figure VIII-1, à l'annexe VIII, illustre les attributs, responsabilités et associations de ces différentes classes sous forme d'un diagramme de classes UML. Le tableau 3-6 énumère ces classes et détaille les responsabilités de chacune :

Tableau 3-6 Classes du module AI

Nom	Description et responsabilités
AICore	Cette classe représente le moteur de décision de l'agent intelligent. À ce titre, sa responsabilité est, à partir d'informations complètes sur l'état d'une partie de Liar's Dice, de produire un jugement sur l'action que le joueur actif de la partie devrait accomplir.
Probabilities	Cette classe a la responsabilité de réaliser des calculs probabilistes et statistiques pour le compte d'autres classes. Elle réalise ses responsabilités en offrant diverses méthodes statiques.
DNA	Cette classe est utilisée pour encapsuler les paramètres numériques utilisés par l'agent intelligent pour réaliser sa prise de décision.

Le processus utilisé par la classe AICore pour sélectionner l'action à réaliser comporte trois étapes. Dans un premier temps, la liste des différents coups pouvant être joués est obtenue. Ensuite, chacun des coups de cette liste est soumis à une fonction d'évaluation, la fonction evaluateMove(). Cette fonction donne une valeur numérique, un score, à chacun des coups possibles. Ce score correspond à une somme pondérée des différents critères d'évaluation. Finalement, un coup est sélectionné aléatoirement à l'intérieur de ceux qui ont les scores les plus élevés.

Lorsque l’agent intelligent évalue la possibilité de déposer un nouveau pari, les critères suivants sont utilisés :

- Nombre de paris, déposés dans la même manche, dont la valeur est la même que celle du nouveau pari.
- Nombre de paris, déposés dans la même manche, dont la valeur est différente de celle du nouveau pari.
- La différence entre le nombre de dés concernés par le pari et le nombre de dés portant la même valeur, en moyenne, parmi tous les dés en jeu. Ce critère considère les dés du joueur contrôlé par l’agent intelligent comme ayant une valeur inconnue et pouvant, ainsi, avoir n’importe quelle valeur.
- La différence entre le nombre de dés concernés par le pari et le nombre de dés portant la même valeur, en moyenne, parmi tous les dés en jeu. Ce critère considère les valeurs réelles des dés du joueur contrôlé par l’agent intelligent.
- Nombre de dés concernés par le nouveau pari.
- Si la valeur du nouveau pari est 6 ou non (ce critère vaut 1 si la valeur est 6 et vaut 0 si la valeur est autre).

Lorsque l’agent intelligent évalue la possibilité de contester ou confirmer le dernier pari formulé, les pondérations diffèrent, mais les critères suivants sont utilisés dans les deux cas :

- Nombre de paris, déposés dans la même manche, dont la valeur est la même que celle du dernier pari.
- Nombre de paris, déposés dans la même manche, dont la valeur est différente de celle du dernier pari.
- La différence entre le nombre de dés concernés par le dernier pari et le nombre de dés portant la même valeur, en moyenne, parmi tous les dés en jeu. Ce critère considère les dés du joueur contrôlé par l’agent intelligent comme ayant une valeur inconnue et pouvant, ainsi, avoir n’importe quelle valeur.

- La différence entre le nombre de dés concernés par le dernier pari et le nombre de dés portant la même valeur, en moyenne, parmi tous les dés en jeu. Ce critère considère les valeurs réelles des dés du joueur contrôlé par l'agent intelligent.
- Nombre de dés concernés par le dernier pari.
- Si la valeur du dernier pari est 6 ou non (ce critère vaut 1 si la valeur est 6 et vaut 0 si la valeur est autre).

De son côté, le fonctionnement de la classe DNA, et sa relation avec la classe AICore, est particulier et mérite d'être discuté plus en détail. En bref, cette classe permet de faciliter l'optimisation génétique de l'agent intelligent. À la base, la classe DNA est un simple conteneur pour valeurs numériques qui n'a pas besoin de connaître les valeurs qui y sont entreposées, ni l'usage qui en est fait par les autres classes. Elle offre des méthodes pour entreposer et pour extraire des valeurs numériques que, à l'interne, elle représente sous forme de tableau d'octets. Elle est utilisée pour stocker les paramètres de l'agent intelligent qui doivent être optimisés par algorithmes génétiques. Ainsi, le module LiarBotTrainer peut réaliser cette forme d'optimisation sur des instances de la classe DNA plutôt que sur des instances de AICore. Cette modification offre plus de flexibilité et de généricité au module LiarBotTrainer :

- Puisque l'optimisation est réalisée sur la classe DNA au lieu de la classe AICore, le module LiarBotTrainer est moins dépendant de l'interface de la classe AICore et ne dépend pas du nombre de paramètres utilisés par l'agent intelligent ou de la nature de ceux-ci.
- Il est aussi plus aisé de réutiliser le module LiarBotTrainer pour réaliser une optimisation génétique sur une nouvelle classe. Il suffit que cette classe offre des méthodes getDNA() et setDNA() comme la classe AICore.
- Finalement, ceci facilite grandement les opérations que le module LiarBotTrainer doit réaliser dans le cadre de l'entraînement de l'agent intelligent. En effet, la technique d'optimisation des algorithmes génétiques nécessite que les divers paramètres à optimiser puissent être représentés sous forme de séquence de valeurs binaires. Le fait que la classe

DNA regroupe tous les paramètres qui y sont entreposés sous forme de tableau d'octets rend plus aisées ces manipulations.

3.2.4.1.2 Module Persistence

Les classes de ce module permettent l'entreposage persistant des données du système Liar Bot. La figure IV-1, à l'annexe IV, illustre les attributs, responsabilités et associations de ces différentes classes sous forme d'un diagramme de classes UML. Le tableau 3-7 énumère ces classes et détaille les responsabilités de chacune :

Tableau 3-7 Classes du module Persistence

Nom	Description et responsabilités
DNAStorage	Cette classe permet d'entreposer, sur le disque dur, les données résultant de l'entraînement de l'agent intelligent et de récupérer celles qui ont été entreposées dans le passé.

Les services de la classe DNAStorage sont utilisés par le module LiarBotTrainer pour entreposer la meilleure combinaison de paramètres qu'il a pu identifier pour l'agent intelligent. Ils sont également utilisés par la classe LiarDice du module LiarBot, lors de la création d'une nouvelle partie, pour obtenir les paramètres à partir desquels initialiser les agents intelligents devant jouer dans cette partie.

3.2.4.1.3 Module LiarBotTrainer

Les classes de ce module permettent d'optimiser l'agent intelligent du système à travers la technique d'optimisation des algorithmes génétiques. La figure X-1, à l'annexe X, illustre les attributs, responsabilités et associations de ces différentes classes sous forme d'un diagramme de classes UML. Le tableau 3-8 énumère ces classes et détaille les responsabilités de chacune.

Tableau 3-8 Classes du module LiarBotTrainer

Nom	Description et responsabilités
LiarBotTrainer	Cette classe emploie les services de la classe GeneticTrainer pour optimiser les valeurs des paramètres utilisées par les instances d'AICore pour choisir les actions à poser.
GeneticTrainer	Cette classe implémente l'optimisation connue sous le nom d'algorithmes génétiques de façon suffisamment générique pour être appliquée à n'importe quel problème.
BreedingStrategy	Interface générique pour les classes permettant de générer une nouvelle population d'individus à partir de la population actuelle et de l'aptitude de chacun de ses individus.
BasicBreedingStrategy	Classe héritant de BreedingStrategy implémentant une stratégie simple pour la génération d'une nouvelle population.
FitnessEvaluator	Interface générique pour les classes permettant d'évaluer l'aptitude d'individus d'une population.
LiarBotFitnessEvaluator	Classe héritant de FitnessEvaluator implémentant une stratégie pour évaluer l'aptitude d'individus dans le contexte du jeu Liar's Dice. Elle procède en créant des parties pour permettre à ces individus de s'affronter et ainsi déterminer les plus aptes.
TrainingStopper	Classe utilisée par la classe LiarBotTrainer comme critère d'arrêt pour l'entrainement génétique. Sa responsabilité est de surveiller ce qui est saisi à la console et d'arrêter l'entraînement si souhaité par l'utilisateur.

La classe GeneticTrainer et les interfaces BreedingStrategy et FitnessEvaluator sont conçues pour être aussi génériques que possible pour favoriser leur réutilisation dans d'autres projets futurs. À l'inverse, les classes BasicBreedingStrategy et LiarBotFitnessEvaluator héritent des interfaces précédemment mentionnées et sont spécifiques à l'optimisation de l'agent intelligent du LiarBot. Cette approche permet que, pour réutiliser l'optimisation génétique

dans un autre contexte, il soit seulement nécessaire d'implémenter une stratégie, héritant de l'interface BreedingStrategy, pour générer une nouvelle population et une stratégie, héritant de FitnessEvaluator, pour évaluer les aptitudes des différents individus.

De son côté, la classe TrainingStopper est une classe interne à la classe LiarBotTrainer et est utilisée pour interrompre l'entraînement de l'agent intelligent à la demande de l'utilisateur. Avant de démarrer l'entraînement, la classe LiarBotTrainer crée un objet TrainingStopper et le démarre dans son propre fil d'exécution. Ce dernier réalise des lectures répétées à la ligne de commande et, lorsqu'une saisie est détectée, appelle la méthode stopTraining() de la classe GeneticTrainer pour mettre fin à l'entraînement.

La figure XI-1, à l'annexe XI, illustre le fonctionnement interne de la méthode train() de la classe GeneticTrainer qui réalise l'optimisation par algorithme génétique. Les étapes associées à ce fonctionnement sont les suivantes :

1) La fonction train() est appelée avec, en paramètre, une instance de DNAStorage, une instance de BreedingStrategy et une instance de FitnessEvaluator
2) L'instance de GeneticTrainer appelle la méthode getFittestDNA de l'objet DNAStorage pour obtenir la meilleure combinaison de valeurs identifiée lors des entraînements précédents.
3) À partir de la combinaison obtenue à l'étape 2, un tableau d'objets DNA est créé.
4) Tant que l'attribut continueTraining de l'instance de GeneticTrainer est vrai, les étapes suivantes sont exécutées :
    a) Évaluation de l'aptitude des différents objets DNA.
    b) L'objet DNA ayant l'aptitude la plus élevée est entreposé pour usage ultérieur.
    c) De nouveaux objets DNA sont générés à partir des anciens objets et des valeurs d'aptitude associées à chacun d'eux.

CHAPITRE 4

ANALYSE DES RÉSULTATS

L'annexe XIV documente les résultats des tests de performance réalisés sur l'agent intelligent du système Liar Bot dans le but d'estimer son niveau de compétence par rapport à celui d'un humain.

Les tests réalisés contre les joueurs débutants semblent révéler que l'agent est d'un niveau de compétence supérieur aux leurs. En effet, le nombre de dés restants aux joueurs humains descend très rapidement dès le début de la partie alors que le nombre de dés des agents intelligents descend beaucoup plus lentement. De plus, lors de ces tests, les deux joueurs débutants qui se sont mesurés à l'agent ont été vaincus en 11 manches ou moins. Les tests réalisés contre les joueurs intermédiaires suggèrent que l'agent intelligent pose un défi pour ces joueurs; il perd ses dés à un rythme légèrement plus rapide, mais néanmoins similaire à celui des joueurs. Toutefois, en moyenne, il reste presque toujours moins de dés à l'agent intelligent qu'aux joueurs ce qui semble impliquer qu'il soit moins performant que ces derniers. Les tests réalisés contre les joueurs experts révèlent que l'agent intelligent est clairement d'un niveau de compétence inférieur aux leurs. En effet, parmi les tests réalisés, aucun joueur de niveau expert n'a perdu de partie et tous ont terminé avec encore trois dés en leur possession.

Au final, il apparaît que l'agent intelligent est d'un niveau de compétence inférieur, mais proche de celui d'un joueur de niveau intermédiaire typique. Toutefois, étant donné que seuls deux joueurs appartenant à cette catégorie se sont mesurés à l'agent intelligent et que les joueurs sont responsables de l'estimation de leurs propres niveaux de compétences, il est impossible d'assurer la certitude de cette conclusion. Il est possible que l'agent soit légèrement plus ou moins performant qu'estimé par les tests décrits plus haut.

CONCLUSION

Les jeux partiellement observables, où aucun des joueurs ne connaît entièrement l'état de la partie, et non déterministes, où certains éléments relèvent du hasard, nécessitent bien souvent des joueurs qu'ils soient aptes à analyser le comportement de leurs adversaires pour acquérir de l'information additionnelle sur l'état du jeu et à bluffer pour mener leurs adversaires à former de fausses déductions. Ces deux compétences sont complexes à implémenter ce qui explique qu'il soit ardu de réaliser des agents intelligents capables de jouer à de tels jeux avec un niveau de compétence satisfaisant. Le présent projet a eu pour objectif de mettre au point un agent intelligent apte à jouer au jeu Liar's Dice avec un niveau de compétence comparable à un joueur de calibre intermédiaire. Il a également eu pour but de concevoir une application permettant à un utilisateur de jouer contre l'agent précédemment mentionné.

L'approche de conception itérative, utilisée dans ce projet, a conduit au développement d'une application basée sur le patron d'architecture Modèle-Vue-Contrôlleur pour limiter le couplage entre les classes et comportant une interface graphique évoquant une partie de Liar's Dice jouée autour d'une table pour la rendre plus intuitive pour les nouveaux utilisateurs. De son côté, l'agent intelligent développé prend ses décisions en attribuant un score à chacune des actions qu'il peut jouer à ton tour et sélectionne son coup parmi les meilleures d'entre elles. Ce score est une somme pondérée de plusieurs critères et les pondérations de ces différents critères sont calibrées par optimisation génétique.

L'évaluation des performances de l'agent intelligent du système contre des joueurs humains de niveaux variés a permis de constater que le niveau de compétence de l'agent intelligent semble s'approcher grandement de celui d'un joueur intermédiaire sans toutefois l'atteindre. En ce qui concerne les joueurs qui débutent et ont peu d'expérience, l'agent intelligent est apte à les vaincre sans difficulté. Pour améliorer davantage les performances de l'agent intelligent du système, il est conseillé de faire jouer ce dernier, à répétitions, contre divers utilisateurs dans le but d'amasser des données sur un grand nombre de parties du jeu Liar's

Dice. Ces données peuvent ensuite être utilisées pour implémenter, au sein de l'agent, une fonctionnalité de modélisation de joueurs, soit par l'entremise d'un réseau de neurones ou d'un réseau bayésien. Une telle fonctionnalité permet un agent d'analyser le jeu de ses adversaires pour s'y adapter et améliorer ses chances de victoires.

LISTE DE RÉFÉRENCES

Findler, Nicholas. 1977. Studies in Machine Cognition Using the game of Poker, En ligne. 16 p. <http://www.idi.ntnu.no/emner/it3105/book/findler-poker-1977.pdf>. Consulté le 10 février 2012.

Richard Papp, Denis. 1998. Dealing With Imperfect Information in Poker, En ligne. 93 p. < http://poker.cs.ualberta.ca/publications/papp.msc.pdf>. Consulté le 15 février 2012.

Davidson, Aaron. 1999. Using Artificial Neural Networks to Model Opponents in Texas Hold'em. En ligne. 6 p. <http://spaz.ca/aaron/poker/nnpoker.pdf>. Consulté le 16 février 2012.

Southley, Finnegan, Micheal Bowling, Bryce Larson, Carmelo iccione, Neil Burch, Darse Billings et Chris Rayner. 2005. Bayes' Bluff : Opponnent modeling in poker. En ligne. 9 p. <http://uai.sis.pitt.edu/papers/05/p550-southey.pdf>. Consulté le 9 février 2012.

Stuart Russel et PeterNorvig. 2010. *Artificial Intelligence : A Modern Approach*, Third Edition. Prentice Hall, 1132 p.

Eclipse Foundation. 2011 (30 novembre). « Artefact : Design ». OpenUp. En ligne. <http://epf.eclipse.org/wikis/openup/index.htm>. Consulté le 22 janvier 2012.

ANNEXE I

DIAGRAMME DE CLASSES DU MODULE GAMEDATA

Le diagramme qui suit emploie la notation UML pour représenter les différentes classes composant le module GameData, ainsi que leurs propriétés et les relations entre elles.

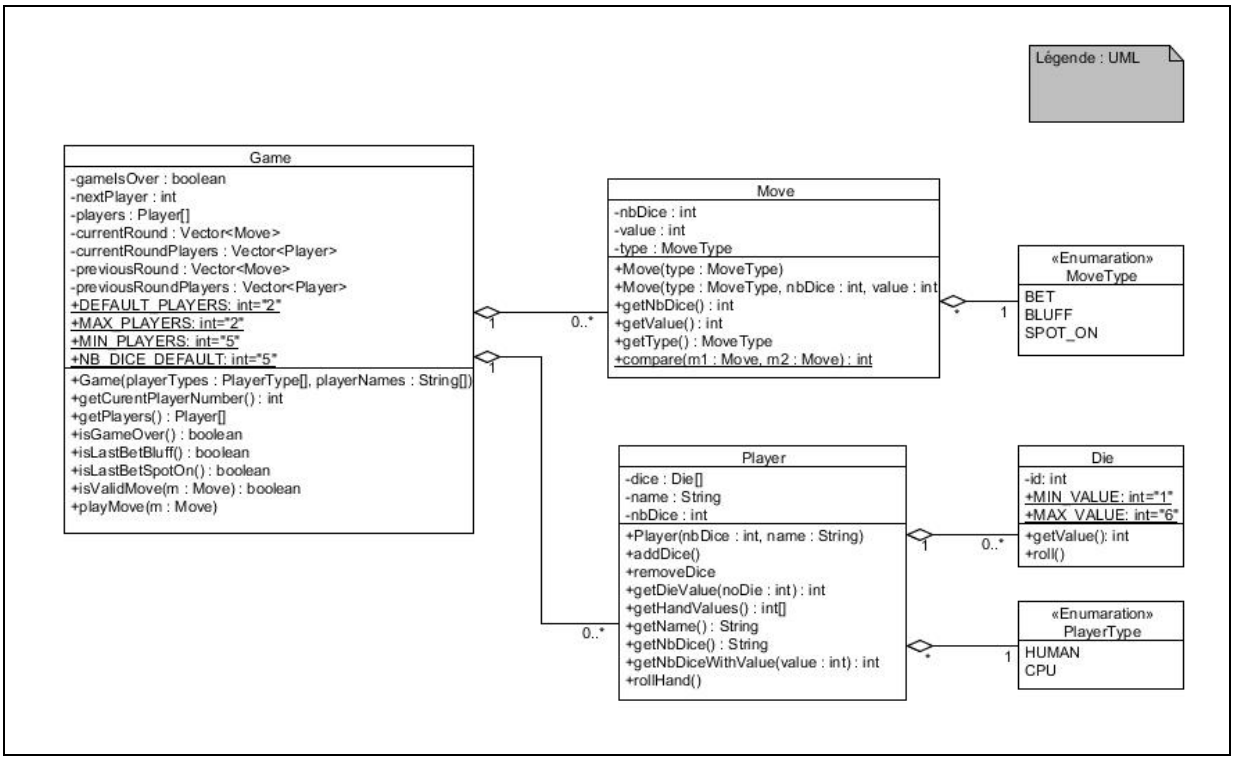

Figure-A I-1 Diagramme de classes du module GameData

ANNEXE II

DIAGRAMME DE CLASSES DU MODULE AI

Le diagramme qui suit emploie la notation UML pour représenter les différentes classes composant le module AI, ainsi que leurs propriétés et les relations entre elles.

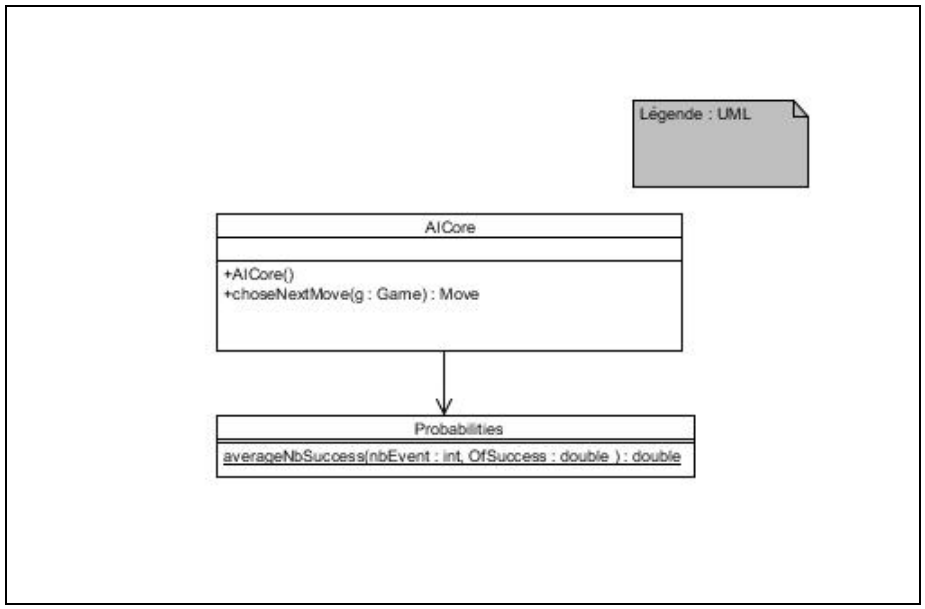

Figure-A II-1 Diagramme de classes du module AI

ANNEXE III

DIAGRAMME DE CLASSES DU MODULE LIARBOT

Le diagramme qui suit emploie la notation UML pour représenter les différentes classes composant le module LiarBot, ainsi que leurs propriétés et les relations entre elles.

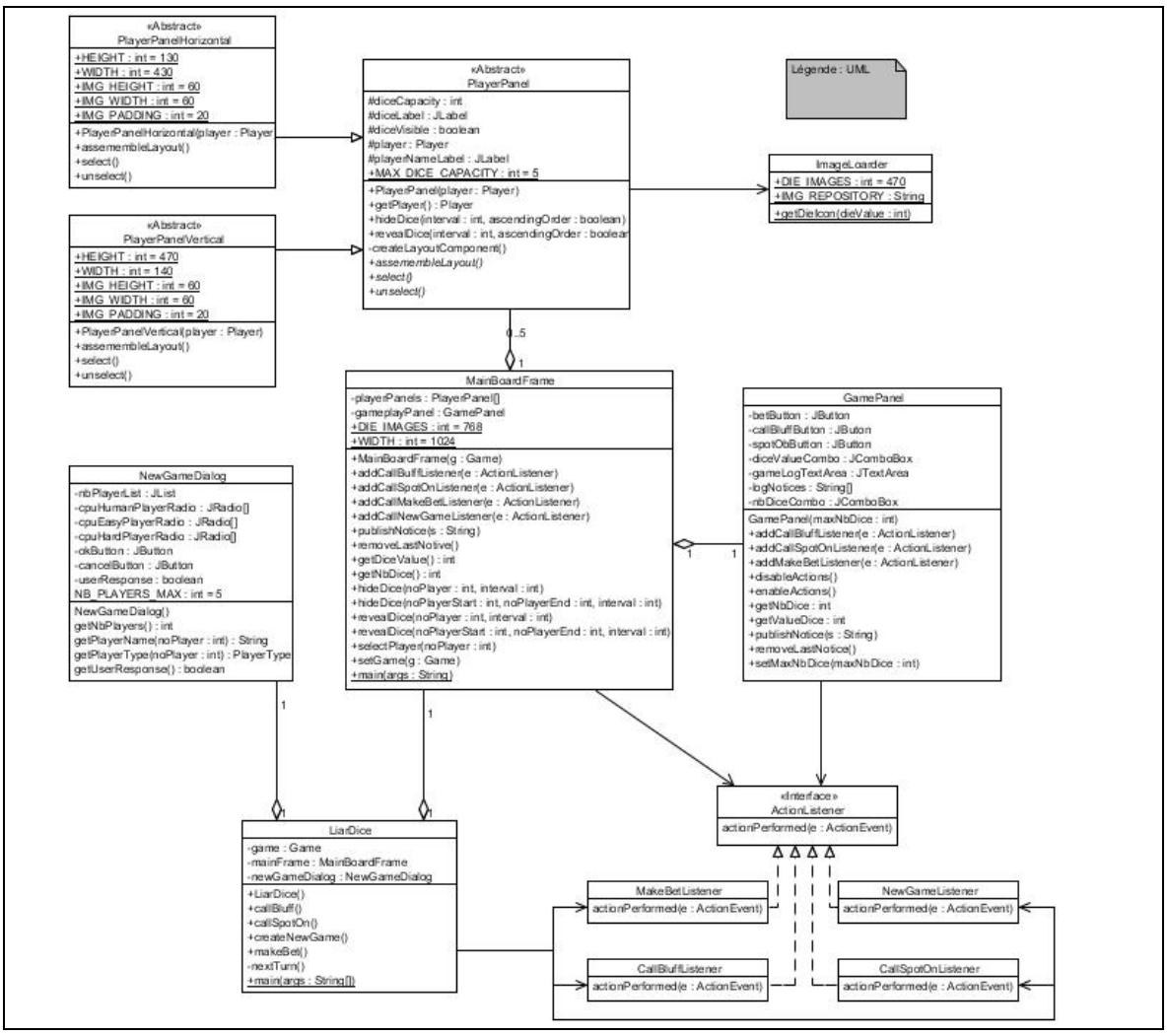

Figure-A III-1 Diagramme de classes du module LiarBot

ANNEXE IV

DIAGRAMME DE SÉQUENCES LBOT-UC-001

Le diagramme de séquences qui suit emploie la notation UML pour représenter les interactions entre les différentes classes du système Liar Bot qui permettent la réalisation du cas d'utilisation LBOT-UC-001 : Créer une nouvelle partie.

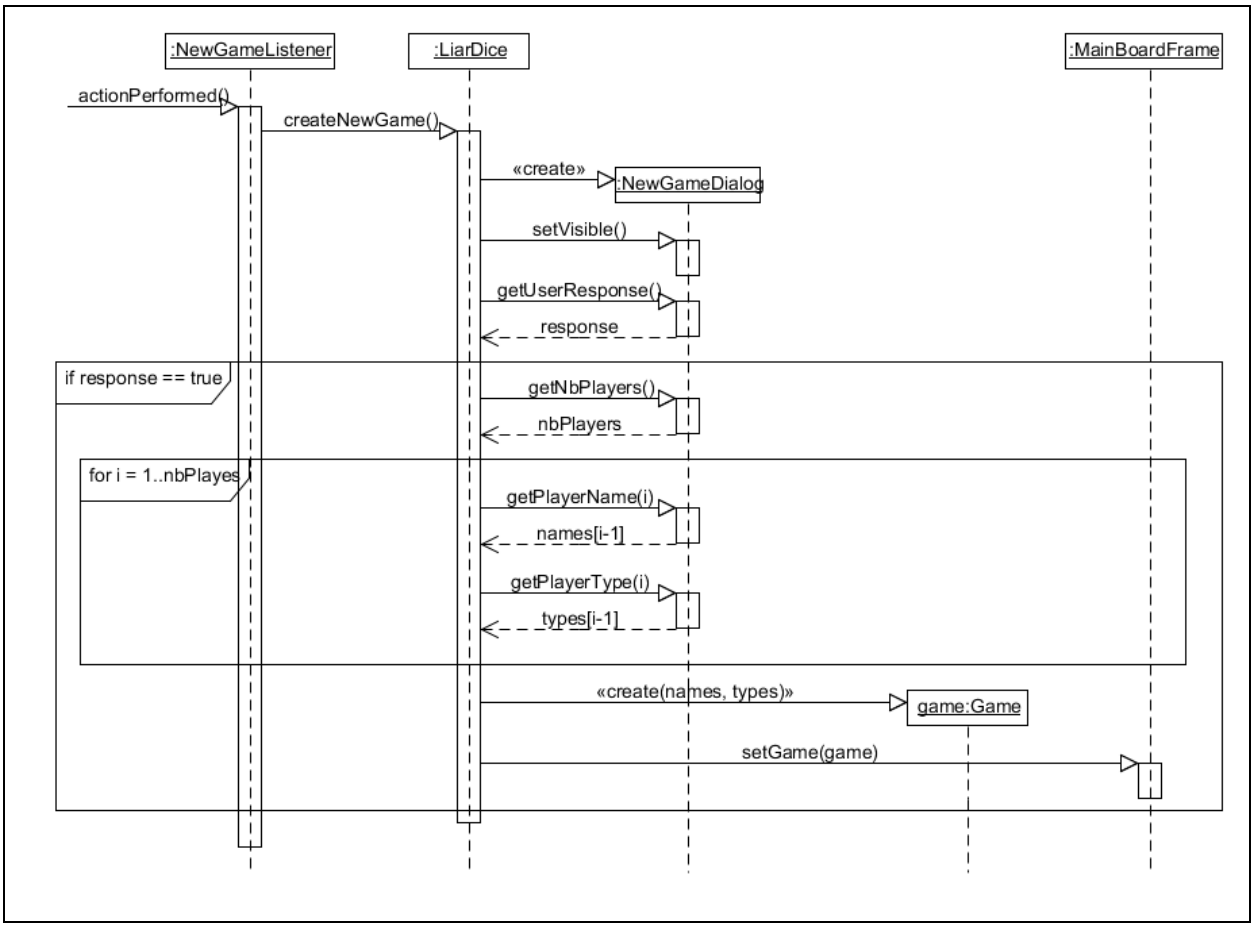

Figure-A IV-1 Diagramme de séquences du cas d'utilisation LBOT-UC-001

ANNEXE V

DIAGRAMME DE SÉQUENCES LBOT-UC-002

Le diagramme de séquences qui suit emploie la notation UML pour représenter les interactions entre les différentes classes du système Liar Bot qui permettent la réalisation du cas d'utilisation LBOT-UC-002 : Déposer un pari.

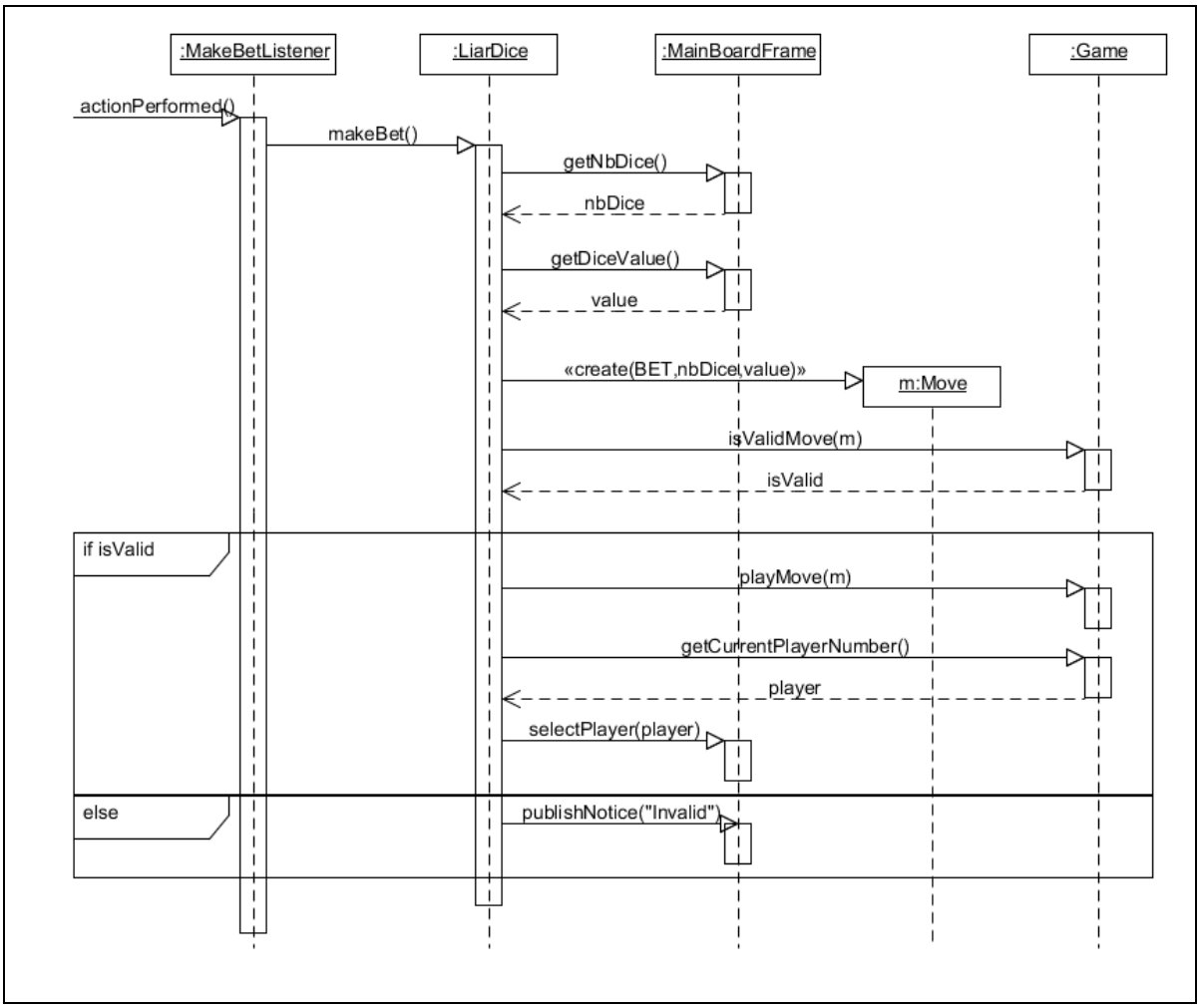

Figure-A V-1 Diagramme de séquences du cas d'utilisation LBOT-UC-002

ANNEXE VI

DIAGRAMME DE SÉQUENCES LBOT-UC-003

Le diagramme de séquences qui suit emploie la notation UML pour représenter les interactions entre les différentes classes du système Liar Bot qui permettent la réalisation du cas d'utilisation LBOT-UC-003 : Contester un pari.

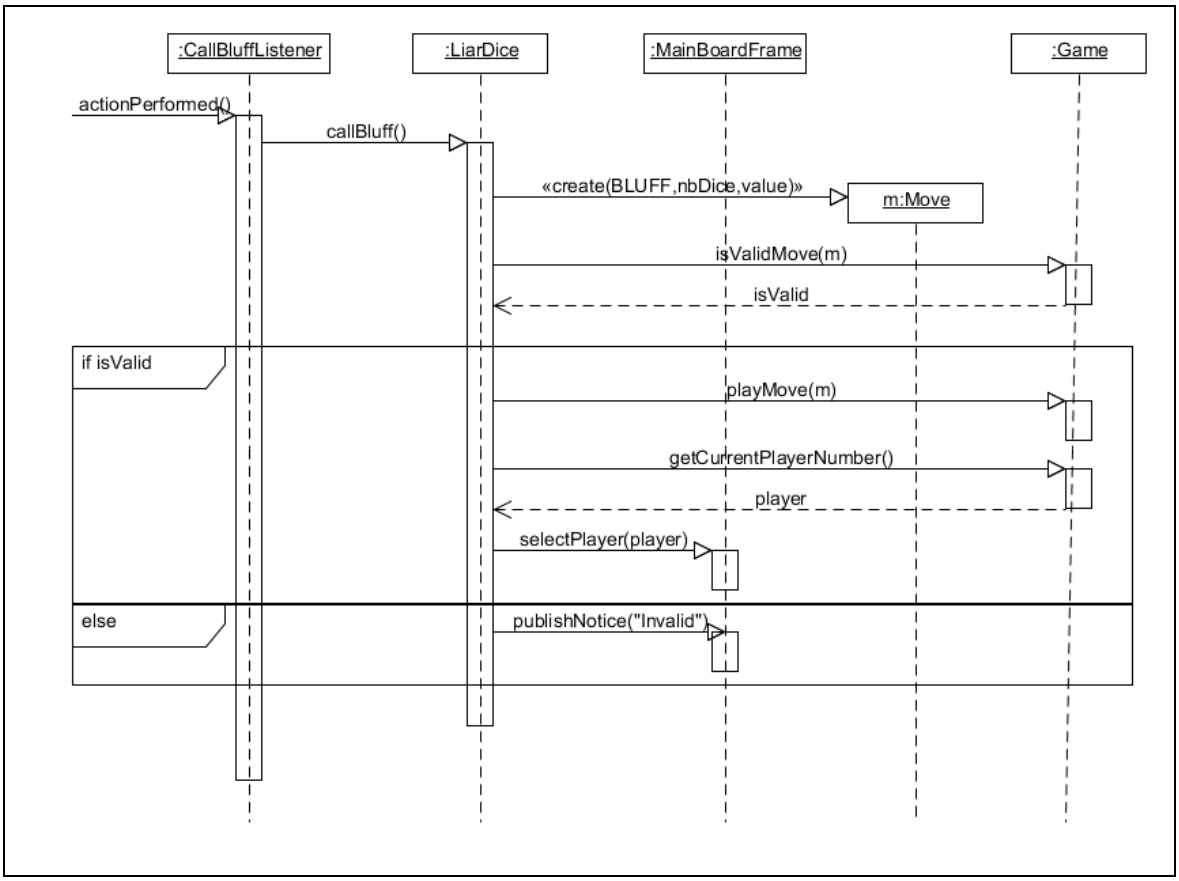

Figure-A VI-1 Diagramme de séquences du cas d'utilisation LBOT-UC-003

ANNEXE VII

DIAGRAMME DE SÉQUENCES LBOT-UC-004

Le diagramme de séquences qui suit emploie la notation UML pour représenter les interactions entre les différentes classes du système Liar Bot qui permettent la réalisation du cas d'utilisation LBOT-UC-004 : Confirmer un pari.

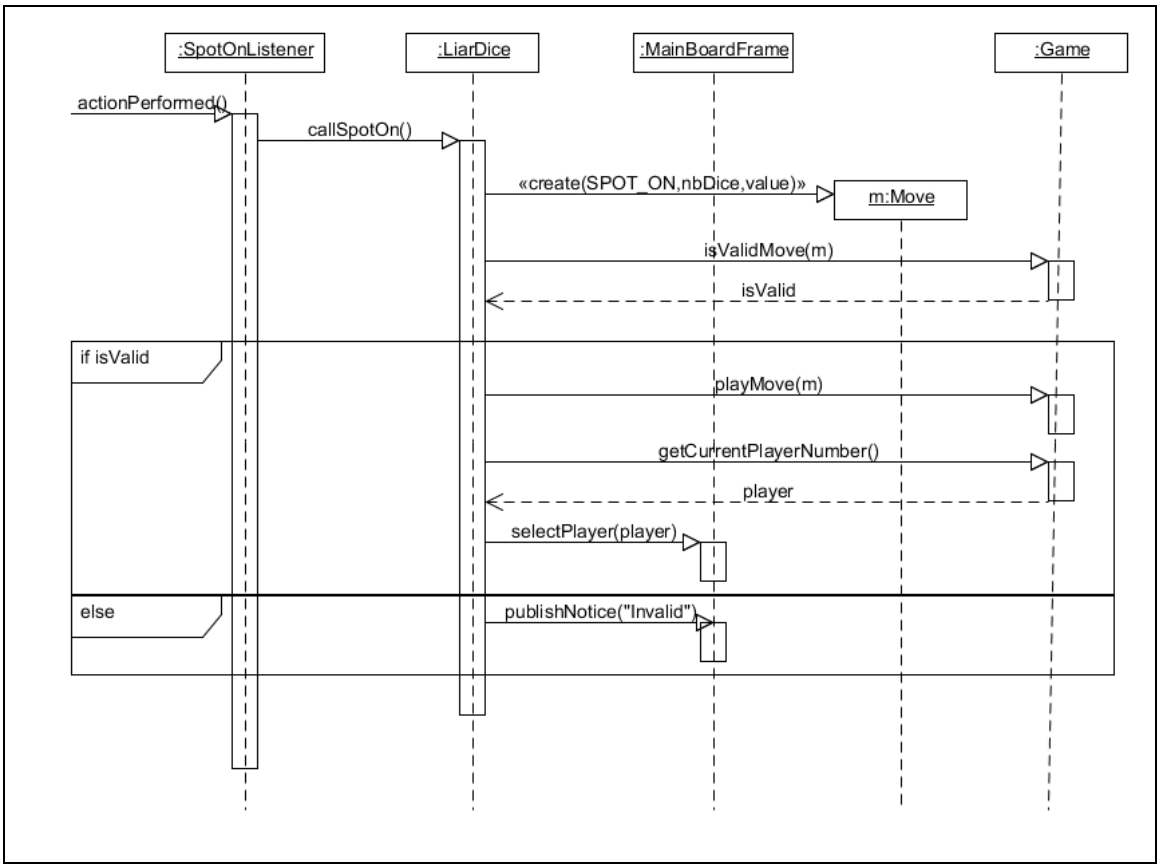

Figure-A VII-1 Diagramme de séquences du cas d'utilisation LBOT-UC-004

ANNEXE VIII

DIAGRAMME DE CLASSES DU MODULE AI – 2IÈME VERSION

Le diagramme qui suit emploie la notation UML pour représenter les différentes classes composant la deuxième version du module AI, ainsi que leurs propriétés et les relations entre elles. Cette deuxième version est issue de la deuxième étape du processus de conception.

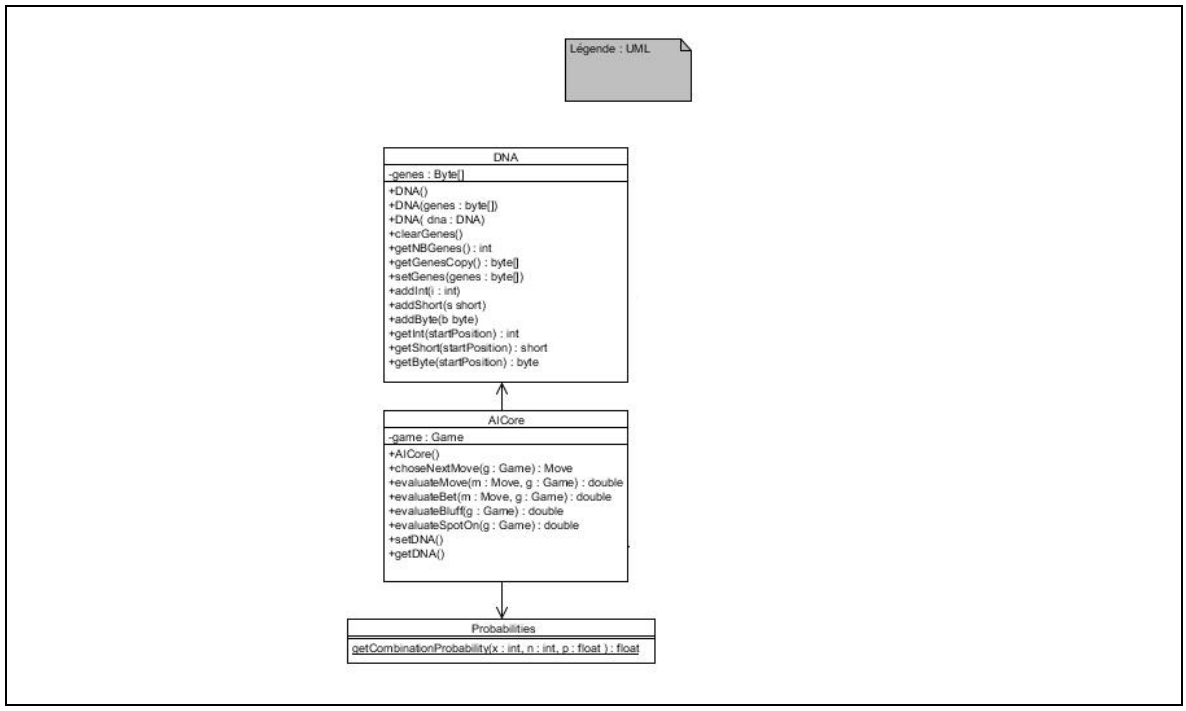

Figure-A VIII-1 Diagramme de classes du module AI – Deuxième version

ANNEXE IX

DIAGRAMME DE CLASSES DU MODULE PERSISTENCE

Le diagramme qui suit emploie la notation UML pour représenter les différentes classes composant le module Persistence, ainsi que leurs propriétés et les relations entre elles. Cette deuxième version est issue de la deuxième étape du processus de conception.

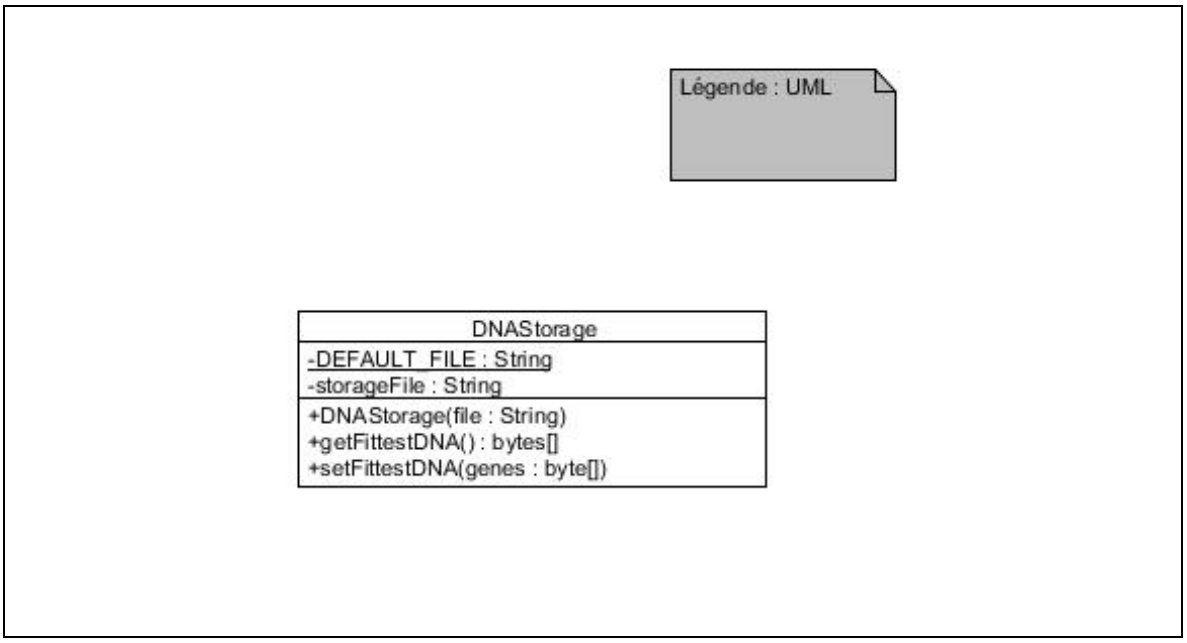

Figure-A VIX-1 Diagramme de classes du module Persistence

ANNEXE X

DIAGRAMME DE CLASSES DU MODULE LIARBOTTRAINER

Le diagramme qui suit emploie la notation UML pour représenter les différentes classes composant le module LiarBotTrainer, ainsi que leurs propriétés et les relations entre elles. Cette deuxième version est issue de la deuxième étape du processus de conception.

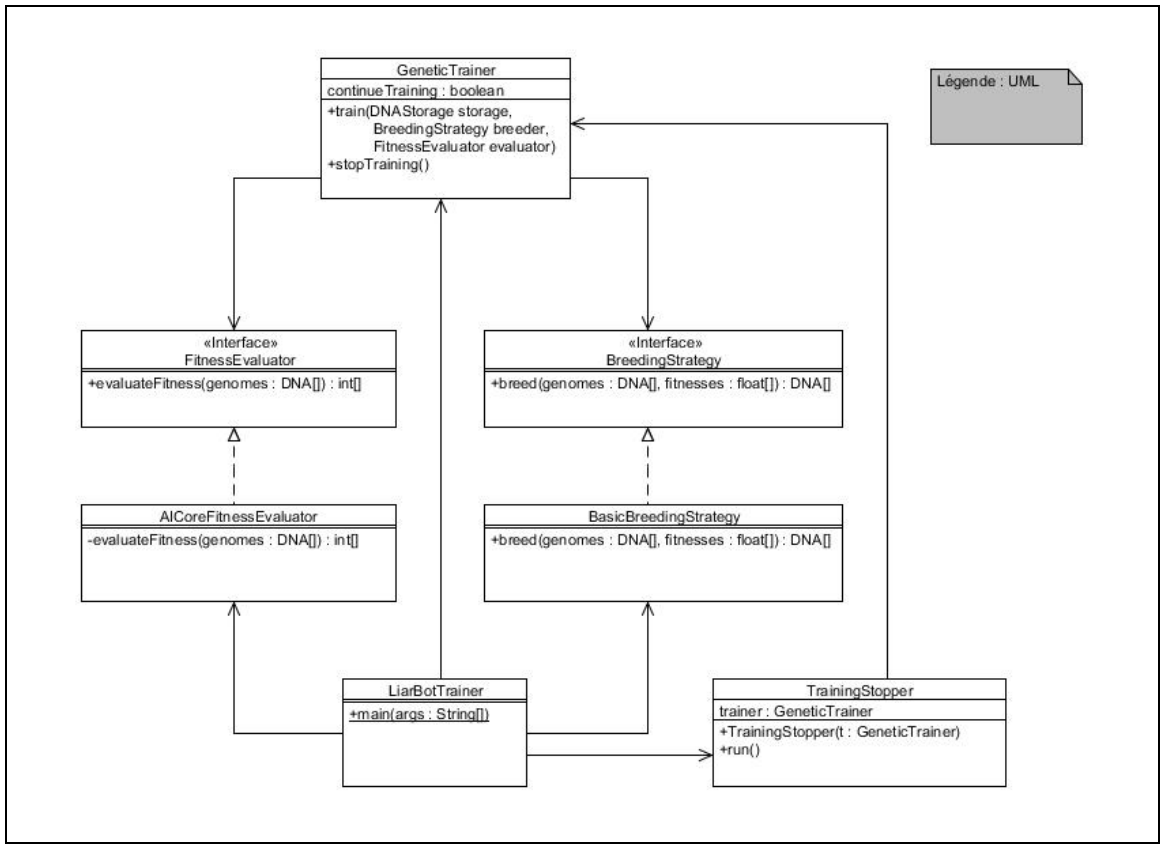

Figure-A X-1 Diagramme de classes du module LiarBotTrainer

ANNEXE XI

DIAGRAMME DE SÉQUENCES DE L'OPTIMISATION GÉNÉTIQUE

Le diagramme de séquences qui suit emploie la notation UML pour représenter les interactions entre les différentes classes du système Liar Bot qui permettent de réaliser l'optimisation par algorithmes génétiques.

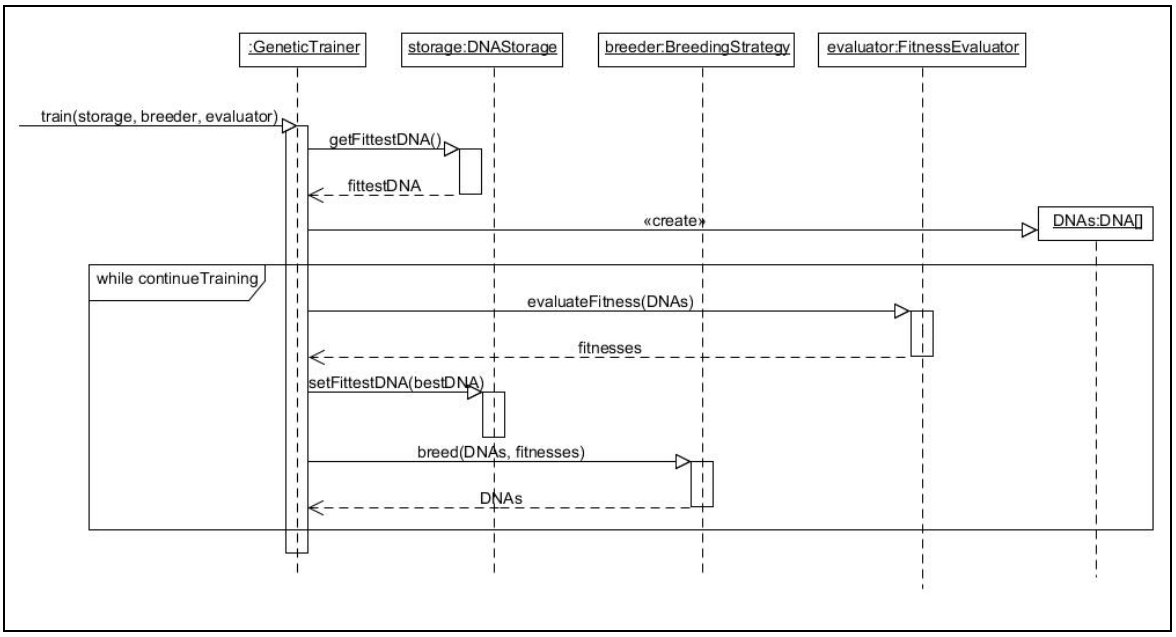

Figure-A XI-1 Diagramme de séquences de l'optimisation génétique

ANNEXE XII

RÈGLEMENTS DU JEU LIAR'S DICE

Le jeu Liar's Dice possède plusieurs variantes au sein de ses règlements. La variante présentée ici est celle qui est utilisée par le système Liar Bot et celle pour laquelle l'agent intelligent du système a été développé.

Initialement, tous les joueurs possèdent cinq dés. Une partie est composée de manches qui, elles-mêmes, sont composées de tours. Lorsqu'un manche commence, tous les joueurs brassent leurs dés puis les cachent de façon à pouvoir les voir, mais que les autres joueurs ne le puissent pas. Ensuite, le joueur désigné pour commencer la manche prend un pari sur un nombre de dés et une valeur de dé. À son tour, le joueur suivant choisit entre trois actions distinctes; déposer un pari plus élevé, contester le pari du joueur précédent ou encore confirmer le pari du joueur précédent. Déposer un pari plus élevé consiste à formuler un pari concernant un nombre de dés plus élevé ou encore le même nombre de dés, mais une valeur supérieure. Si le joueur dépose un pari supérieur, c'est maintenant au tour du troisième joueur et ainsi de suite jusqu'à ce qu'un joueur conteste ou confirme le pari du joueur ayant joué avant lui.

Lorsqu'un joueur conteste le pari du joueur précédent, les dés de tous les joueurs sont révélés. S'il n'y a pas, en jeu, au moins autant de dés de la valeur spécifiée dans le pari que le nombre de dés spécifié dans le pari, alors le pari était vrai. Dans ce cas, le joueur qui a faussement contesté le pari perd un de ses dés et une nouvelle manche commence; celui-ci en reçoit le premier tour. Dans le cas contraire, le joueur qui avait déposé le pari fautif perd un de ses dés et une nouvelle manche commence; celui-ci en reçoit le premier tour.

Lorsqu'un joueur confirme le pari du joueur précédent, les dés de tous les joueurs sont révélés. S'il y a exactement, en jeu, autant de dés de la valeur spécifiée dans le pari que le

nombre de dés spécifié dans le pari, alors le pari est confirmé. Dans ce cas, tous les joueurs, à l’exception de celui qui a confirmé le pari, perdent un dé et une nouvelle manche commence; le joueur qui avait déposé le pari qui a été confirmé en reçoit le premier tour. Dans le cas contraire, le joueur qui a faussement confirmé le pari perd un de ses dés et une nouvelle manche commence; celui-ci en reçoit le premier tour.

Les manches se succèdent ainsi jusqu’à qu’il ne reste plus qu’un seul joueur à qui il reste des dés. Ce joueur est couronné vainqueur.

ANNEXE XIII

FONCTIONNEMENT DES ALGORITHMES GÉNÉTIQUES

La technique d'optimisation par algorithmes génétiques vise à résoudre un problème donné en émulant le phénomène qu'est l'évolution. Chez les êtres vivants, l'évolution repose sur trois grands principes :

- Sélection : les individus les plus adaptés à leur environnement on plus de chances de pouvoir propager leurs gènes. Les autres ont tendance à s'éteindre avant de pouvoir se reproduire.
- Croisement : reproduction de deux individus qui donne lieu à un ou plusieurs nouveaux individus portant une partie des gènes de chaque parent.
- Mutation : altération de l'information génétique (gènes) d'un individu

Donc, le processus d'évolution naturelle peut être résumé ainsi :

1) On retrouve au départ une certaine quantité d'individus (population initiale)
2) La sélection est opérée sur la population et ne conserver que les individus les plus adaptés à leur environnement.
3) De nouveaux individus sont générés à partir d'individus existants, par croisements et mutations.
4) Les nouveaux individus sont ajoutés à la population et le processus recommence.

Les algorithmes génétiques visent à reproduire ce processus pour identifier la meilleure solution possible à un problème donné. Dans le contexte des algorithmes génétiques, chaque individu est une tentative de solution au problème à optimiser. Les gènes d'un individu sont un tableau de bits ou d'octets qui décrit les caractéristiques de la solution proposée par cet individu. Ces caractéristiques peuvent être une suite d'étapes, un ensemble de valeurs pour des constantes utilisées par la solution, etc.

Le processus utilisé par les algorithmes génétiques est le suivant :

1) Création d'une population initiale d'individus (incluant leurs gènes)
2) Évaluation de l'aptitude (« fitness ») de chaque individu de la population face au problème à résoudre.
3) Création d'une nouvelle population à partir des opérations de sélection, croisement et mutation. Pour ce faire, tant que la taille de la nouvelle population est inférieure à la taille de la population actuelle, les étapes suivantes sont répétées :
    a) Deux individus sont choisis parmi la population actuelle. Plus l'aptitude d'un individu est élevée, plus il a de chances d'être choisi (sélection).
    b) Deux nouveaux individus sont créés, chacun héritant d'une portion des gènes de chaque parent (croisement).
    c) La valeur de certains bits de chaque nouvel individu est inversée pour diversifier le matériel génétique de la nouvelle population (mutation).
    d) Les deux nouveaux individus sont insérés dans la nouvelle population.
4) Retour à l'étape 2 avec la population nouvellement créée.

ANNEXE XIV

RÉSULTATS DES TESTS DE L'AGENT INTELLIGENT

Tel que décrit dans le plan de tests du projet Liar Bot, l'évaluation du niveau de compétence de l'agent intelligent du système Liar Bot est réalisée en faisant jouer des utilisateurs de divers niveau de compétence à jouer contre lui. Dans le cadre des tests réalisés, six utilisateurs se sont mesurés à l'agent : deux débutants, deux joueurs intermédiaires et deux experts (ces estimations du niveau des utilisateurs sont réalisées par les utilisateurs eux-mêmes).

Chaque utilisateur a joué une partie, contre 4 instances de l'agent intelligent. Le nombre de dés restants à chaque joueur après la fin de chaque manche a été comptabilisé. Finalement, des moyennes ont été réalisées entre les parties impliquant des utilisateurs de même niveau. La figure XIV-1 offre un graphique qui compare la moyenne du nombre de dés restants à l'agent intelligent et aux joueurs humains dans le cadre des parties impliquant des utilisateurs de débutants. Les figure XIV-2 et XIV-3 présentent respectivement les données relatives aux parties impliquant les utilisateurs de niveau intermédiaire et avancé.

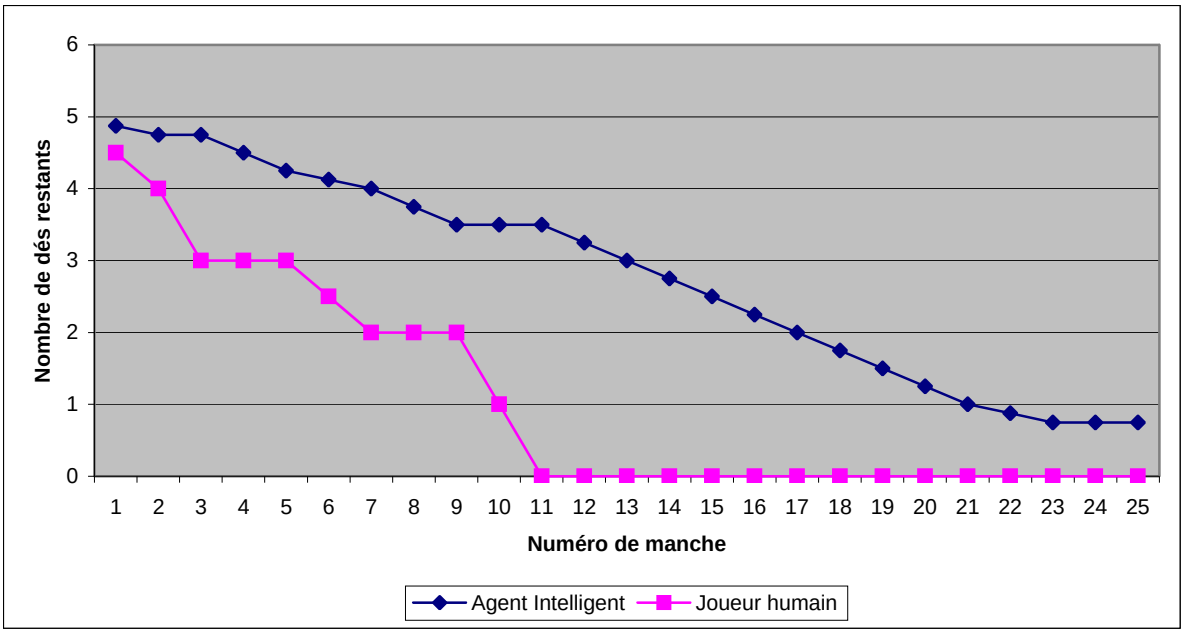

Figure-A XIV-1 Moyenne des dés restants à la fin de chaque manche dans les parties opposant l'agent intelligent à des joueurs débutants

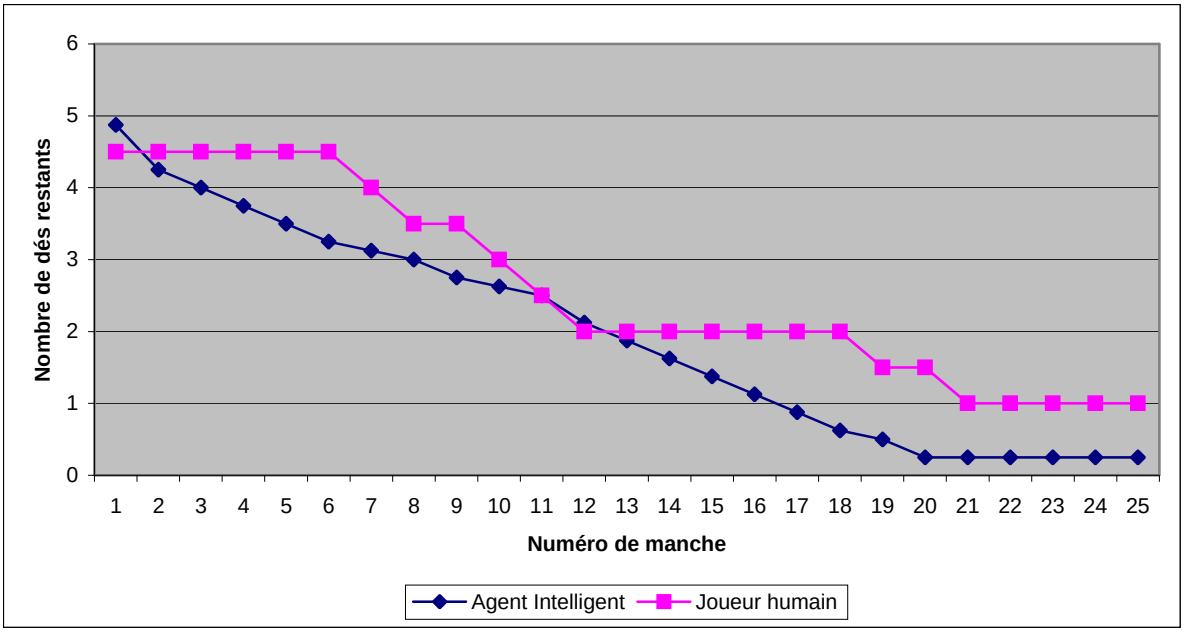

Figure-A XIV-2 Moyenne des dés restants à la fin de chaque manche dans les parties opposant l'agent intelligent à des joueurs intermédiaires

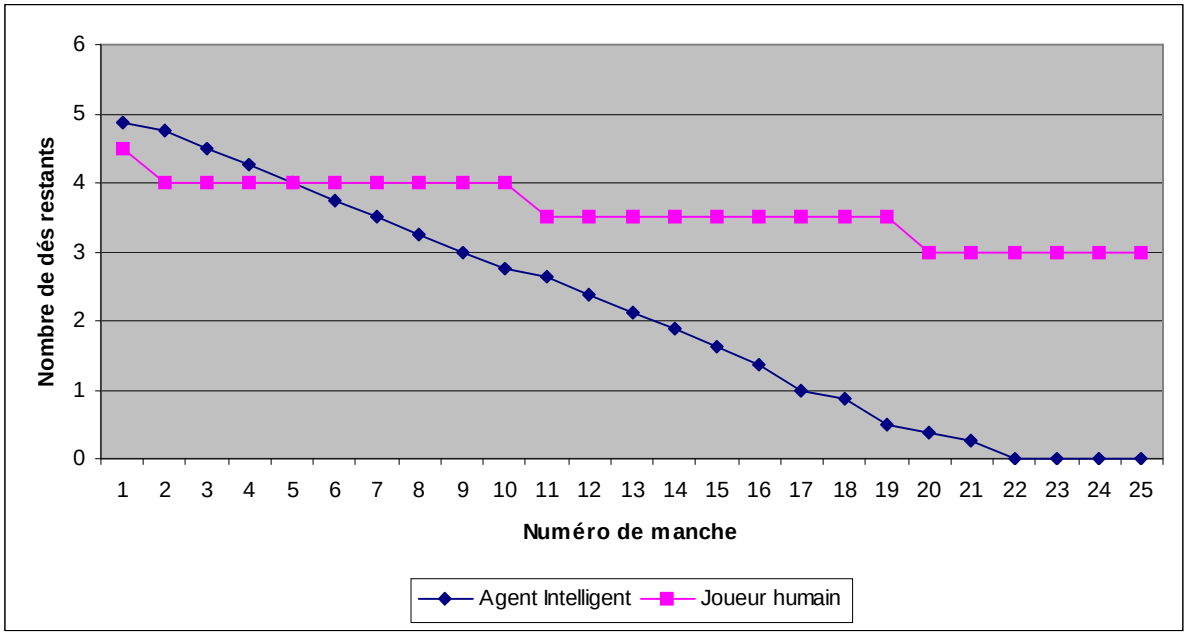

Figure-A XIV-3 Moyenne des dés restants à la fin de chaque manche dans les parties opposant l'agent intelligent à des joueurs experts

APPENDICE A

Les deux figures suivantes représentent des captures d'écran du prototype d'interface utilisateur réalisé dans le cadre de ce projet. La figure A-1 illustre la fenêtre permettant la création d'une nouvelle partie et la figure A-2 donne un exemple d'une partie en cours.

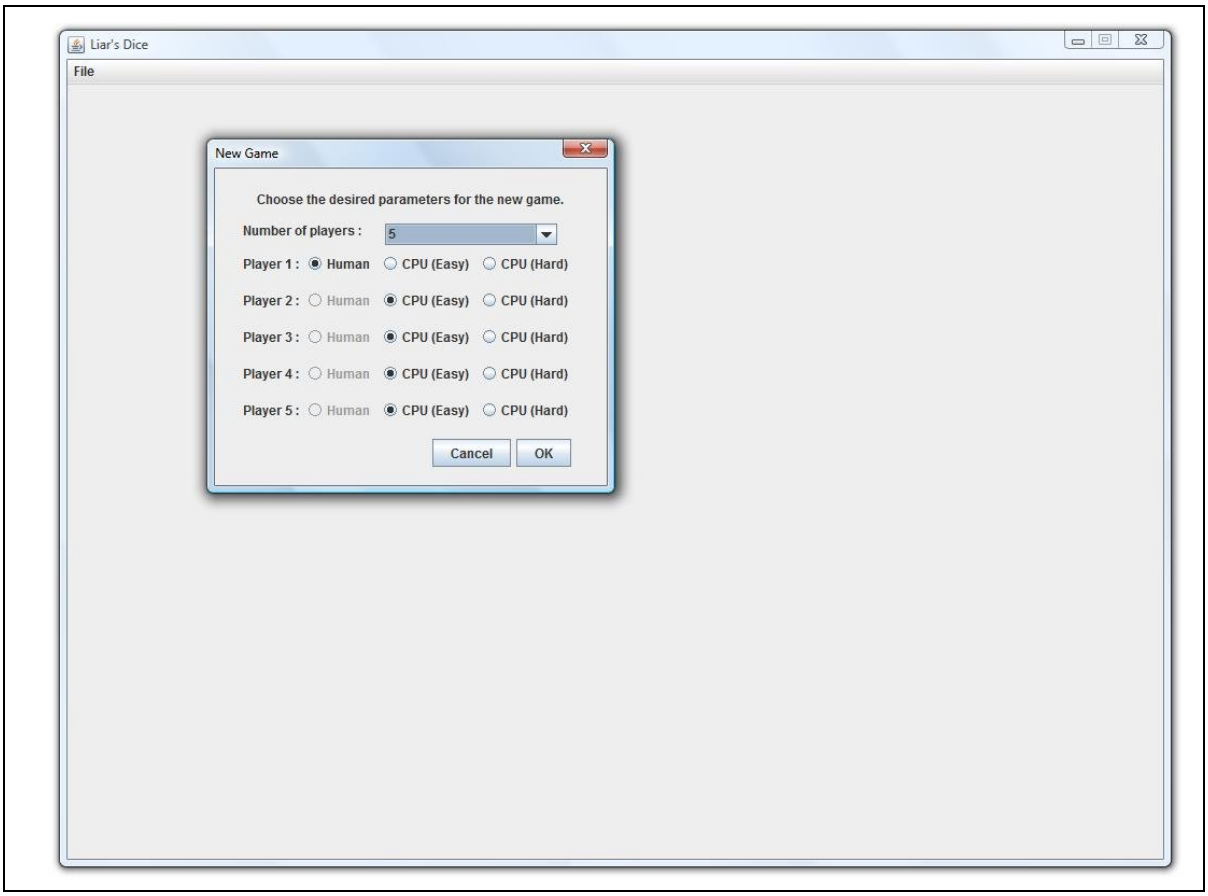

Figure A-1 Interface utilisateur – Création d'une nouvelle partie

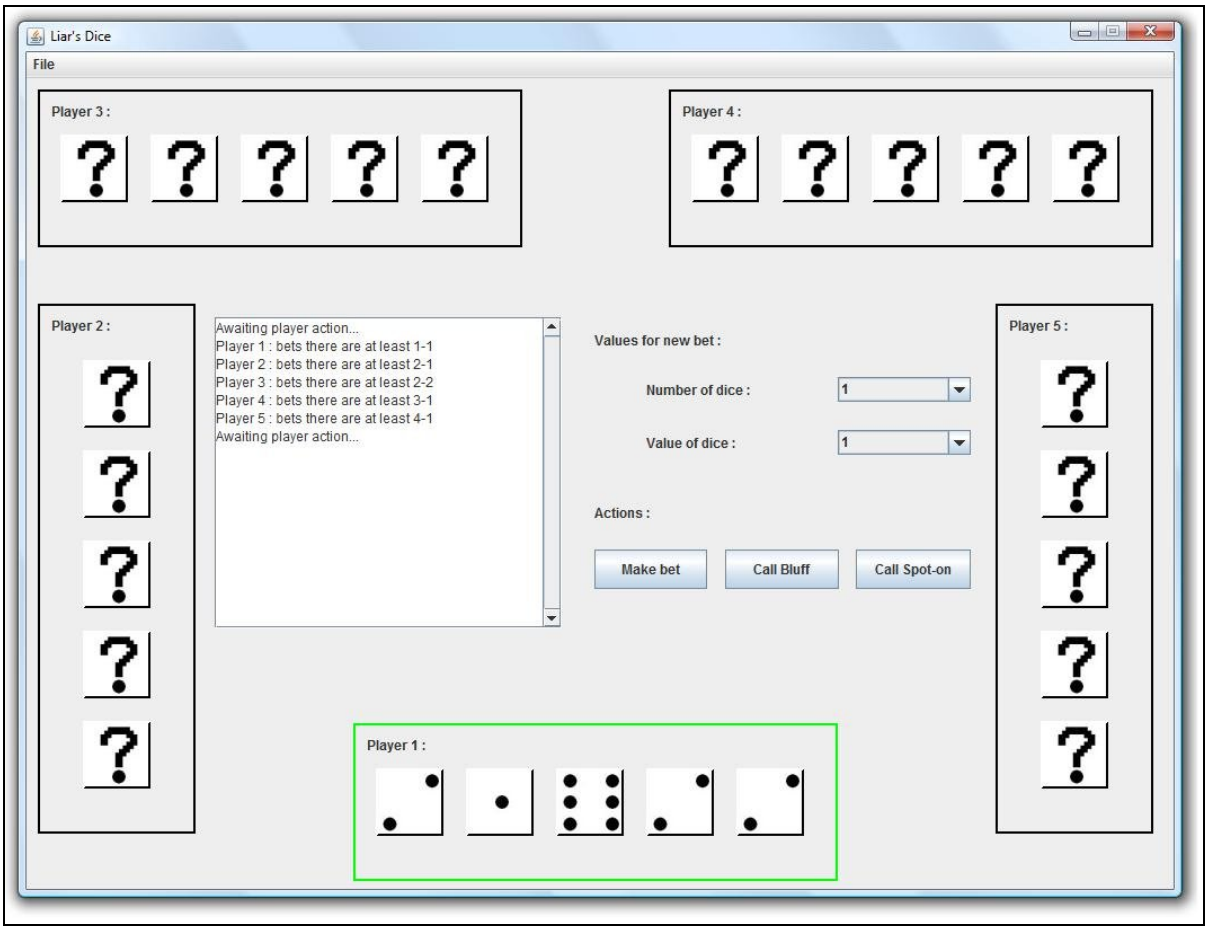

Figure A-2 Interface utilisateur – Partie en cours