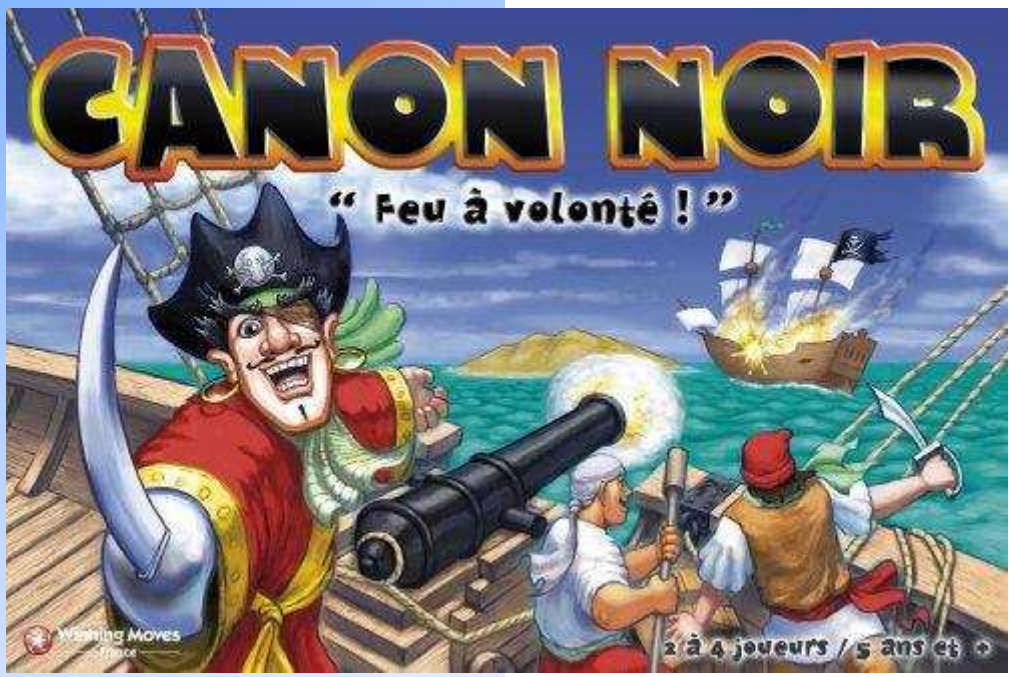
Desfeux Nicolas
Texier Aurélien

Projet « Canon Noir »



Rapport de conception

4ème année
Département Informatique
INSA Rennes

Table des Matières

I. Introduction

Le projet « Canon noir » est un projet visant à développer le jeu appelé « Canon noir ». Ce jeu de société est un jeu de plateau pouvant se jouer de 2 à 4 joueurs. Chaque joueur dirige un ou deux bateaux sur le plateau, et doit pour gagner ramener 3 trésors au port associé au bateau qu'il dirige.

Pour cela, il doit lancer des dés pour déplacer son bateau sur le plateau qui contient deux îles sur lesquelles il est impossible de naviguer. Ce plateau présente également des cases « Trésor » qui lui permettent de récolter un trésor, et des cases « Canon » qui lui permettent de tirer au canon sur les adversaires. Cette dernière possibilité permet de récupérer leur trésor et/ou de les handicaper dans leur progression, puisque si le bateau est touché par le tir, l'assaillant récupère le trésor transporté, et l'attaqué voit son bateau devenir moins puissant (il existe trois types de bateaux, à savoir les caravelles, les frégates et les radeaux).

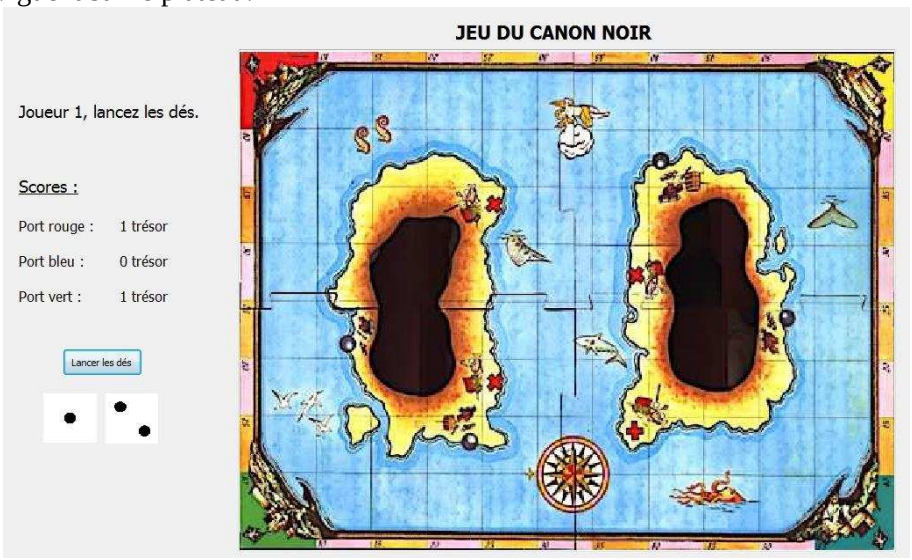
La première phase de ce projet consiste donc à modéliser ce jeu par l'intermédiaire de différents diagrammes permettant de traduire les différentes fonctionnalités du jeu, les interactions avec les joueurs, les différents cas d'utilisation.

II. Spécifications générales

Le jeu sera développé de façon à ce que les interactions avec les joueurs se fassent exclusivement par l'intermédiaire de la souris, plus particulièrement du clic gauche. On suppose alors que les joueurs, devant l'écran de l'ordinateur, suivent les instructions du jeu (lancer les dés, choisir case pour déplacement, choisir le nombre de joueurs, …) et s'échangent la souris pour jouer chacun leu tour.

Voici un aspect général de l'interface graphique du jeu au moment de son lancement.

Voici maintenant un aspect général de la même interface pouvant exister durant le jeu, lorsque les bateaux naviguent sur le plateau.



III. Diagramme de cas d'utilisation

Afin de mieux nous familiariser avec le jeu « Canon Noir », nous avons étudié les règles en détail. L'étude de ces règles nous a permis de réaliser en premier lieu un diagramme de cas d'utilisation, que nous allons maintenant vous présenter.
Seul le joueur, par l'intermédiaire de la souris, est un élément extérieur au système. De ce fait, il est le seul à être en interaction avec celui-ci. On peut voir via notre diagramme ci-dessous les différents types d'actions qu'il est amené à effectuer lors du déroulement du jeu.

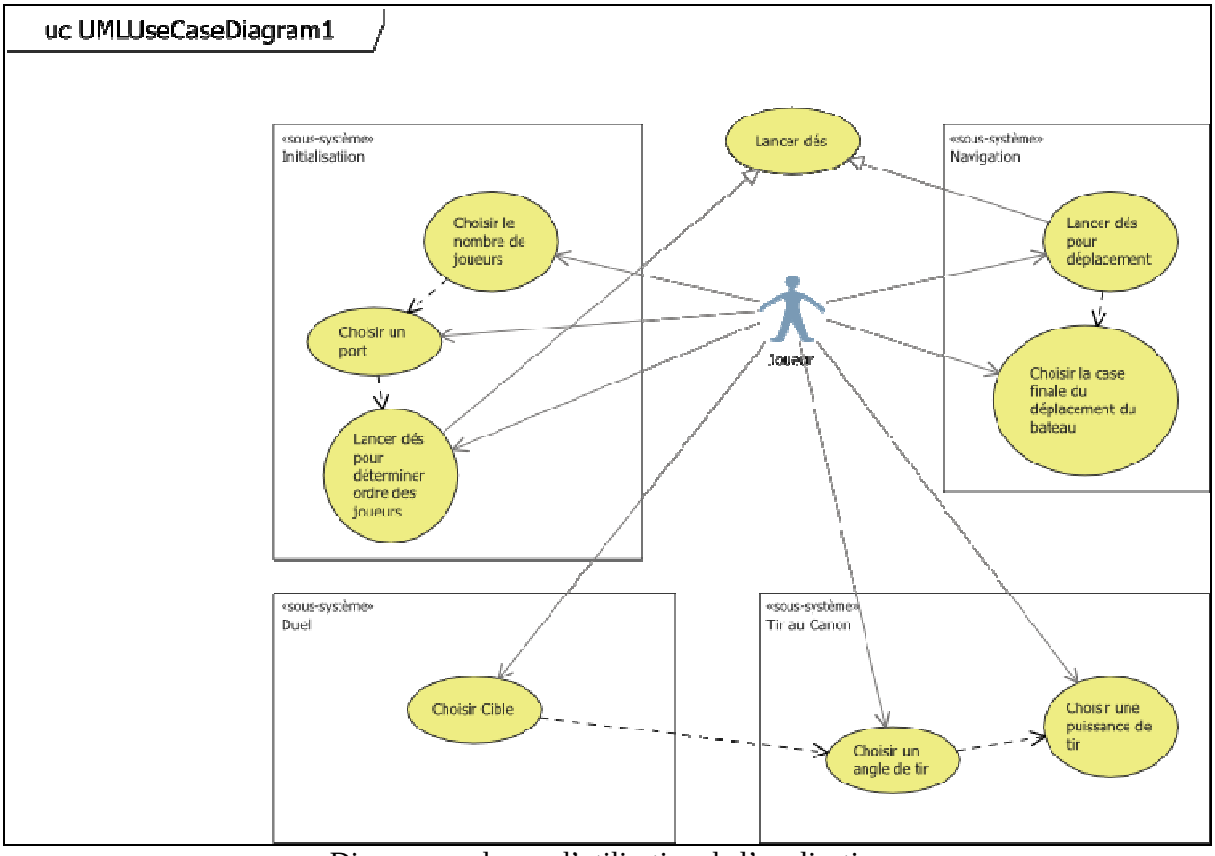
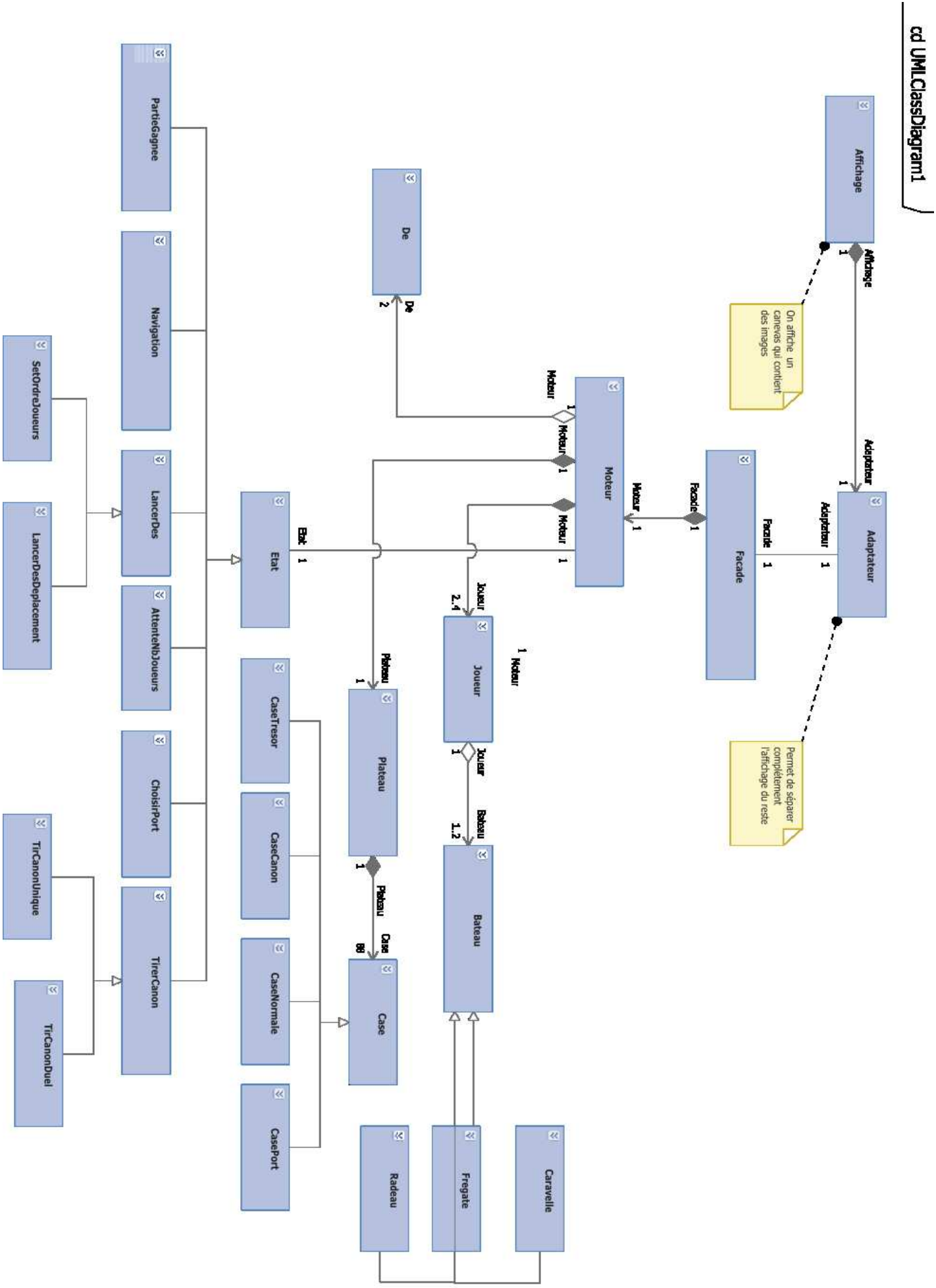


Diagramme de cas d'utilisation de l'application

IV. Diagramme de Classes

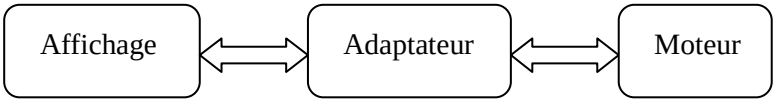
Le diagramme de classe est un outil permettant de décrire de manière succincte la totalité des classes qui seront utilisées durant le développement du projet. Il vise à exposer et clarifier les objets qui interviendront pendant le fonctionnement du jeu « Canon noir » (avec leurs attributs et leurs méthodes), ainsi que leurs relations entre eux. Le but est de concevoir un diagramme de classes qui permette de développer une application robuste et flexible, et qui présente une forte cohésion et un faible couplage. C'est pour cela que l'utilisation des Design Patterns est fortement conseillée, car il a été prouvé que ce sont des modèles de conception qui permettent d'obtenir ces propriétés.

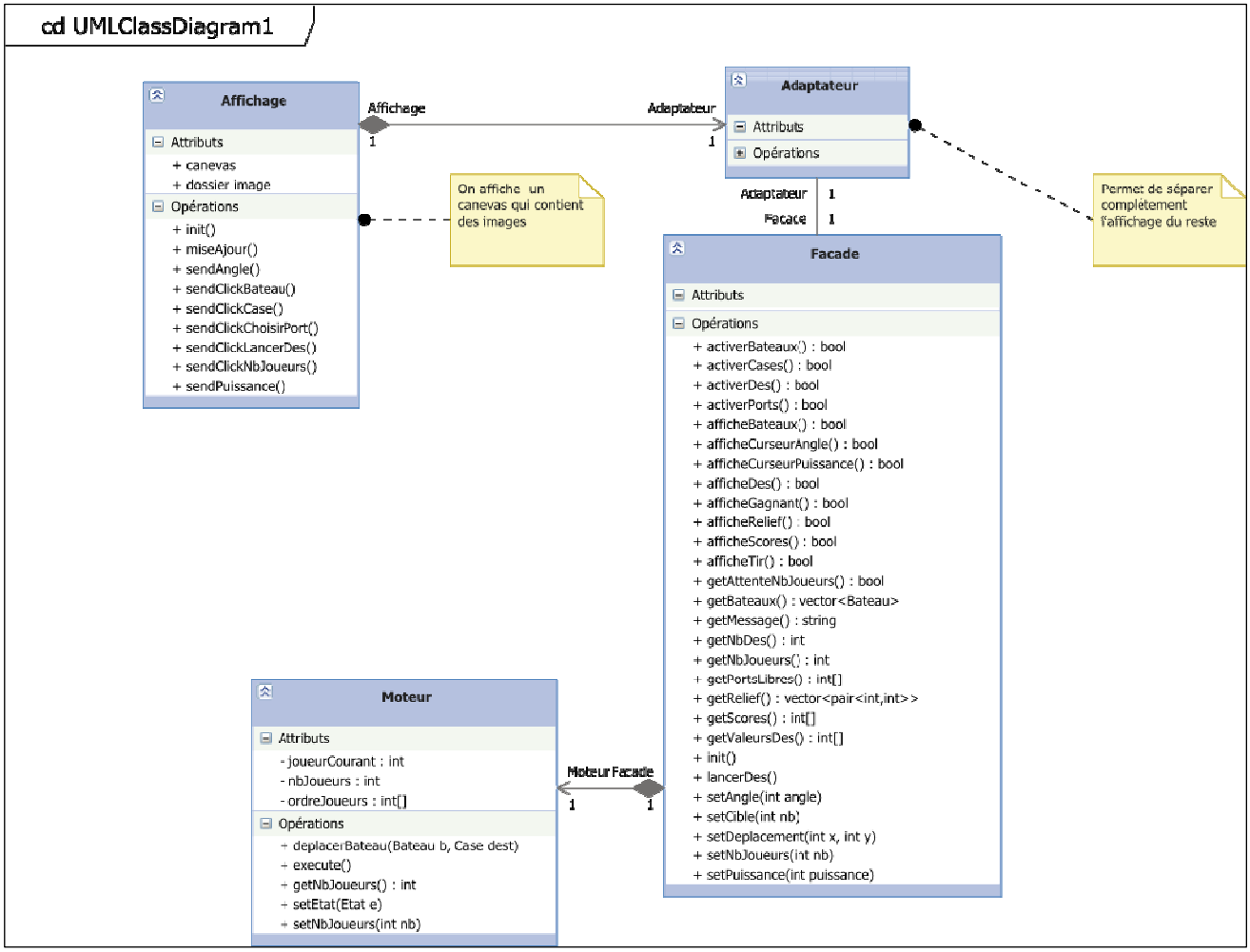
Sur la page suivante notre diagramme de classe complet. Nous détaillerons ensuite certaines parties que nous trouvons pertinentes car elles mettent en œuvre des design patterns reconnus.


cd UMLClassDiagram1
Affichage
On affiche un canevas qui contient des images
Adaptateur
Permet de séparer complétement l'affichage du reste
Facade
De
Moteur
Etat
Joueur
Plateau
Bateau
Case
CaseTresor
CaseCanon
CaseNormale
CasePort
Radeau
Fregate
Caravelle
PartieGagnee
Navigation
SetOrdreJoueurs
LancerDes
LancerDesDeplacement
AttenteNbJoueurs
ChoisirPort
TirCanonUnique
TirerCanon
TirCanonDuel

Model View Controller

Le jeu développé aura une structure générale qui sépare l'interface graphique, avec toutes ses actions d'affichage, du moteur qui lui a le rôle de recevoir les données et les changements, et de recalculer toutes les conséquences de ces changements. L'interface n'a plus alors qu'à fournir les informations au moteur, et à récupérer ce qu'il résulte pour l'affichage. Pour cela, un Design Pattern appelé « Model View Controller » permet de bien séparer ces deux parties. Le voici ci-dessous :

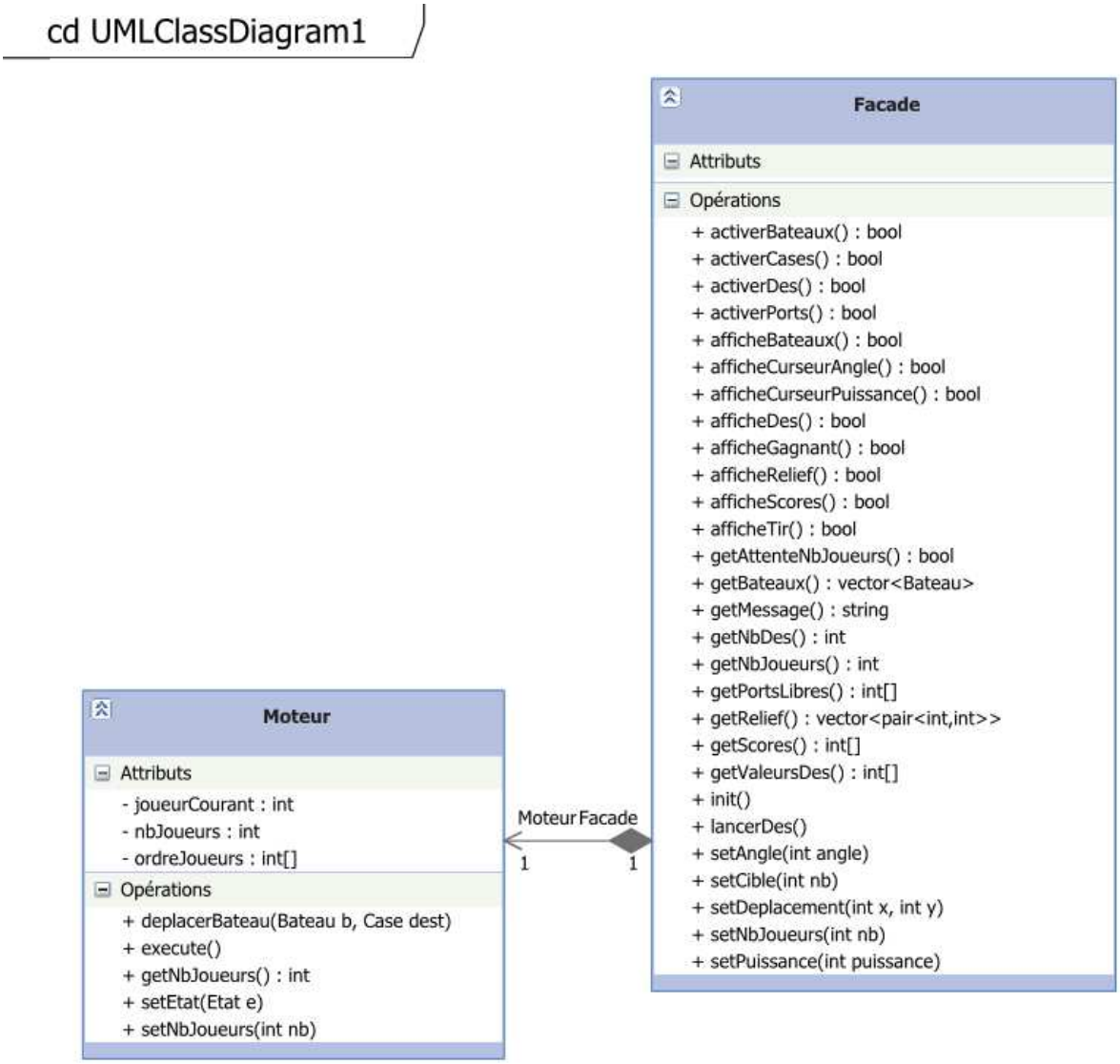




Il présente un fort avantage, le fait que le moteur et l'affichage ne soient pas liés. De ce fait, si nous étions plus tard amenés à modifier complètement l'affichage (par exemple faire un affichage en trois dimensions), ce serait beaucoup plus facile puisqu'il suffirait de refaire une classe « Affichage » et de ré-implémenter les méthodes nécessaires. De même, réciproquement, on pourrait aisément se servir de l'affichage déjà fait du jeu pour ré-implémenter un moteur différent (pour un jeu différent ou ayant des règles différentes).

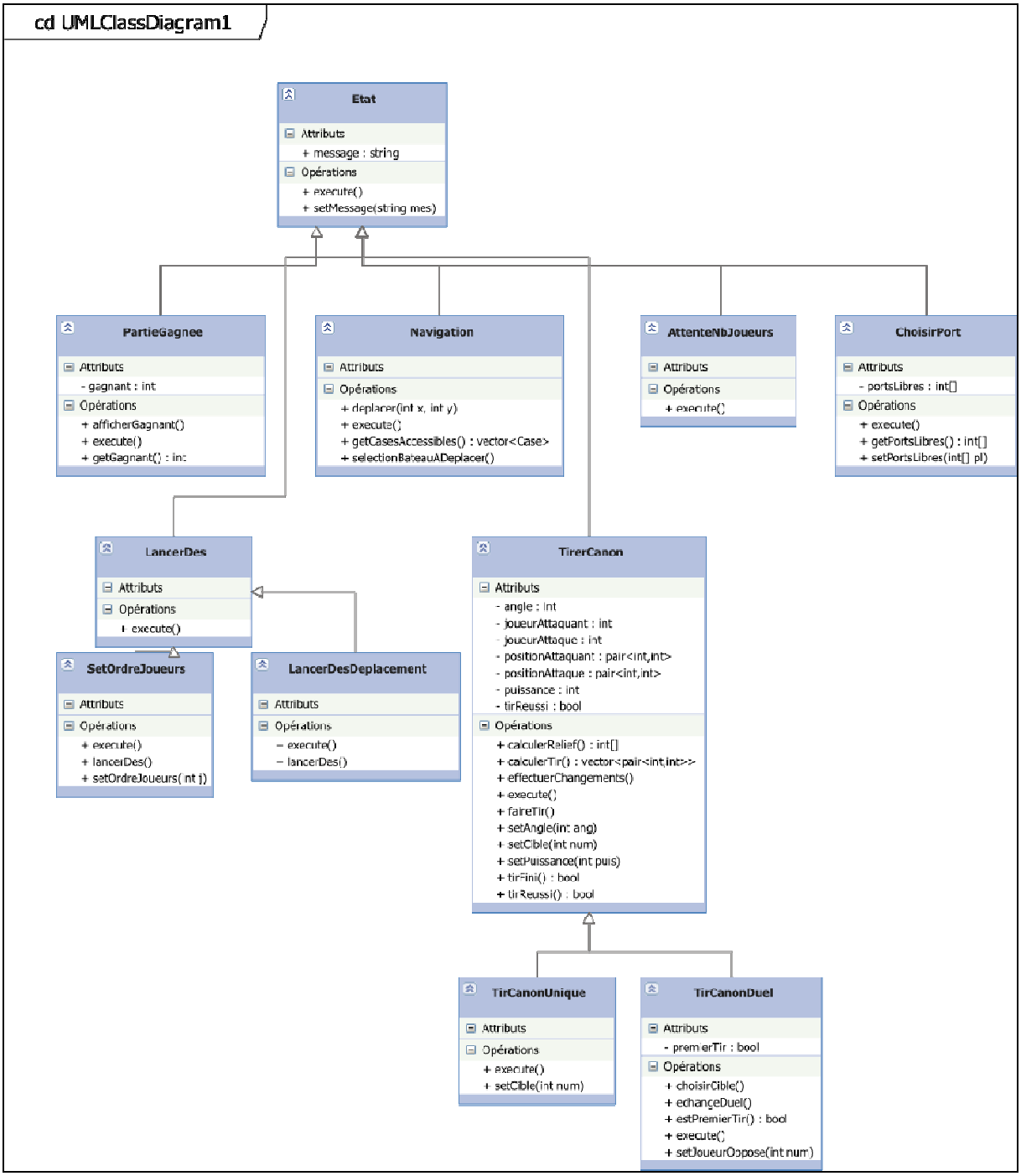
Façade

L'utilisation du Design Pattern « Façade » est utilisé ici pour réunir toutes les méthodes dont a besoin l'affichage. Cela permet de bien séparer le moteur et d'utiliser la classe Façade pour le faire fonctionner. Voici dans notre cas son utilisation :


cd UMLClassDiagram1
Facade
Attributs
Opérations
+ activerBateaux() : bool
+ activerCases() : bool
+ activerDes() : bool
+ activerPorts() : bool
+ afficheBateaux() : bool
+ afficheCurseurAngle() : bool
+ afficheCurseurPuissance() : bool
+ afficheDes() : bool
+ afficheGagnant() : bool
+ afficheRelief() : bool
+ afficheScores() : bool
+ afficheTir() : bool
+ getAttenteNbJoueurs() : bool
+ getBateaux() : vector<Bateau>
+ getMessage() : string
+ getNbDes() : int
+ getNbJoueurs() : int
+ getPortsLibres() : int[]
+ getRelief() : vector<pair<int,int>>
+ getScores() : int[]
+ getValeursDes() : int[]
+ init()
+ lancerDes()
+ setAngle(int angle)
+ setCible(int nb)
+ setDeplacement(int x, int y)
+ setNbJoueurs(int nb)
+ setPuissance(int puissance)
Moteur
Attributs
- joueurCourant : int
- nbJoueurs : int
- ordreJoueurs : int[]
Opérations
+ deplacerBateau(Bateau b, Case dest)
+ execute()
+ getNbJoueurs() : int
+ setEtat(Etat e)
+ setNbJoueurs(int nb)
Moteur Facade
1
1

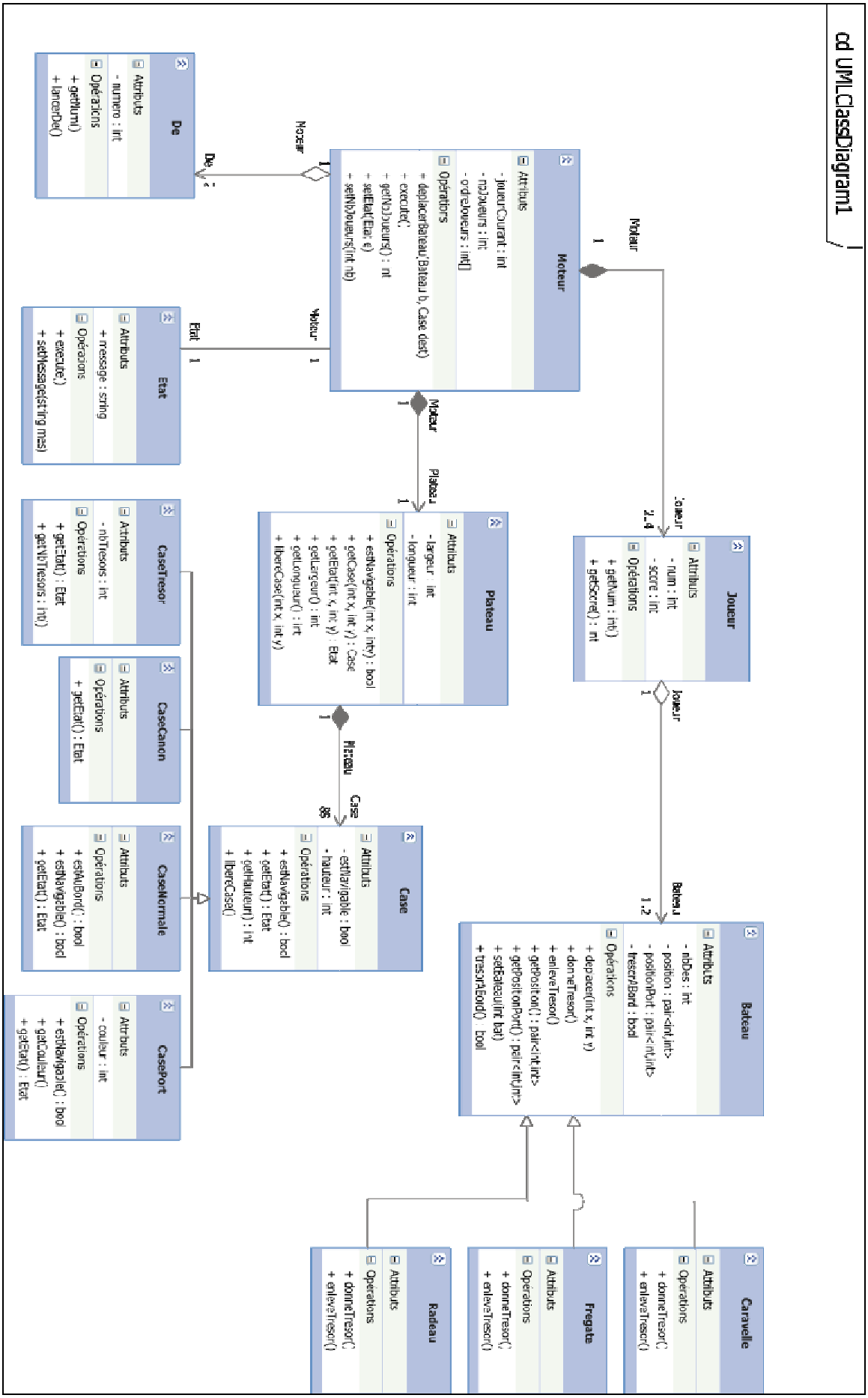
Etat

Le Design Pattern Etat est utilisé ici pour modéliser les différents états que peut prendre la partie, à savoir l'attende du nombre de joueurs, le choix du port pour les joueurs, le lancer des dés, la navigation pour déplacer les bateaux, le tir au canon et la fin de la partie où un joueur a gagné. Le moteur, au cours du jeu, se trouve donc forcément dans un de ces états, et cela lui permet de modifier son comportement quand son état interne change. Voici donc dans notre cas l'utilisation de ce Design Pattern :



Enfin, nous présentons ici un zoom des autres classes agrégées par le moteur, qui sont les classes De, Plateau et Joueur.

La classe Plateau est notamment constituée de 8x11 Cases qui peuvent être soit des CaseCanon, soit des CaseTresor, soit des CasePort, soit des CaseNormal (qui ne présentent pas de propriétés particulières). L'héritage était donc bien approprié à cette configuration.

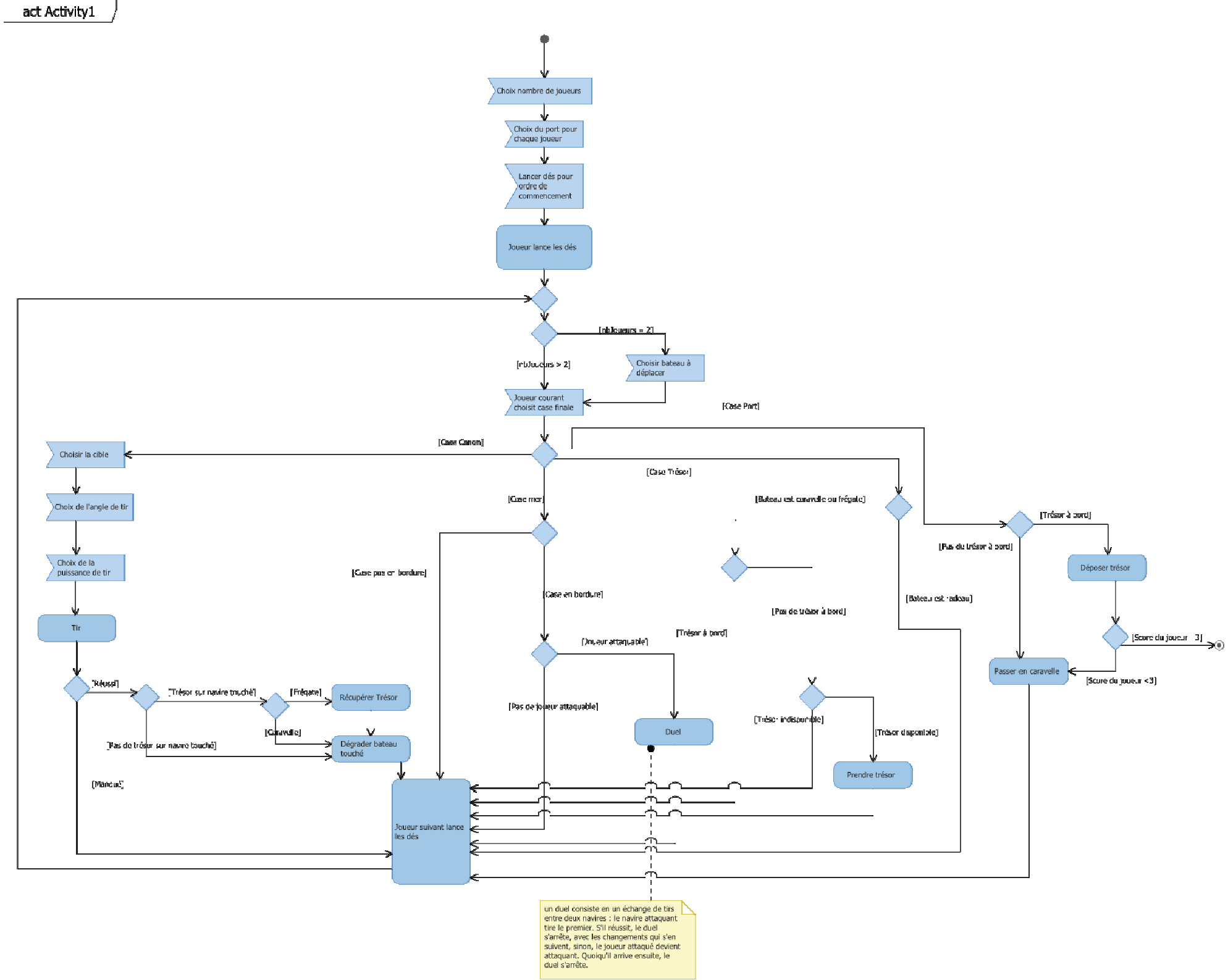

cd UMLClassDiagram1
Moteur
Attributs
- joueurCourant : int
- nbJoueurs : int
- ordreJoueurs : int[]
Opérations
+ deplacerBateau(Bateau b, Case dest)
+ execute()
+ getNbJoueurs() : nt
+ setEtat(Etat e)
+ setNbJoueurs(int nb)
De
Attributs
- numero : int
Opérations
+ getNum()
+ lancerDe()
Etat
Attributs
+ message : string
Opérations
+ execute()
+ setMessage(string mes)
Joueur
Attributs
- num : int
- score : int
Opérations
+ getNum : int()
+ getScore() : nt
Plateau
Attributs
- largeur : int
- longueur : int
Opérations
+ estNavigable(int x, int y) : bool
+ getCase(int x, int y) : Case
+ getEtat(int x, int y) : Etat
+ getLargeur() : int
+ getLongueur() : int
+ libereCase(int x, int y)
Case
Attributs
- estNavigable : bool
- hauteur : int
Opérations
+ estNavigable() : bool
+ getEtat() : Etat
+ getHauteur() : int
+ libereCase()
CaseTresor
Attributs
- nbTresors : int
Opérations
+ getEtat() : Etat
+ getNbTresors : int()
CaseCanon
Attributs
Opérations
+ getEtat() : Etat
CaseNormale
Attributs
Opérations
+ estAuBord() : bool
+ estNavigable() : bool
+ getEtat() : Etat
CasePort
Attributs
- couleur : int
Opérations
+ estNavigable() : bool
+ getCouleur()
+ getEtat() : Etat
Bateau
Attributs
- nbDes : int
- position : pair<int,int>
- positionPort : pair<nt,int>
- tresorABord : bool
Opérations
+ deplacer(int x, int y)
+ donneTresor()
+ enleveTresor()
+ getPosition() : pair<int,int>
+ getPositionPort() : pair<int,int>
+ setBateau(int ba)
+ tresorABord() : bool
Caravelle
Attributs
Opérations
+ donneTresor()
+ enleveTresor()
Fregate
Attributs
Opérations
+ donneTresor()
+ enleveTresor()
Radeau
Attributs
Opérations
+ donneTresor()
+ enleveTresor()
Moteur
1
Moteur
1
De
1..2
Moteur
1
Moteur
1
Etat
1
Moteur
1
Joueur
2..4
Joueur
1
Bateau
1..2
Plateau
1
Plateau
1
Case
65

La

classe Joueur agrège la classe Bateau, puisque chaque joueur possède un ou deux bateaux qu'il gère selon sa volonté. Les objets Bateau stockent donc leur position sur le plateau, ainsi que leur port associé. Ces objets peuvent être des Caravelle, Fregate ou Radeau, et donc ils possèdent des propriétés différentes. Là aussi, l'héritage était donc approprié.

V. Diagramme d'activités

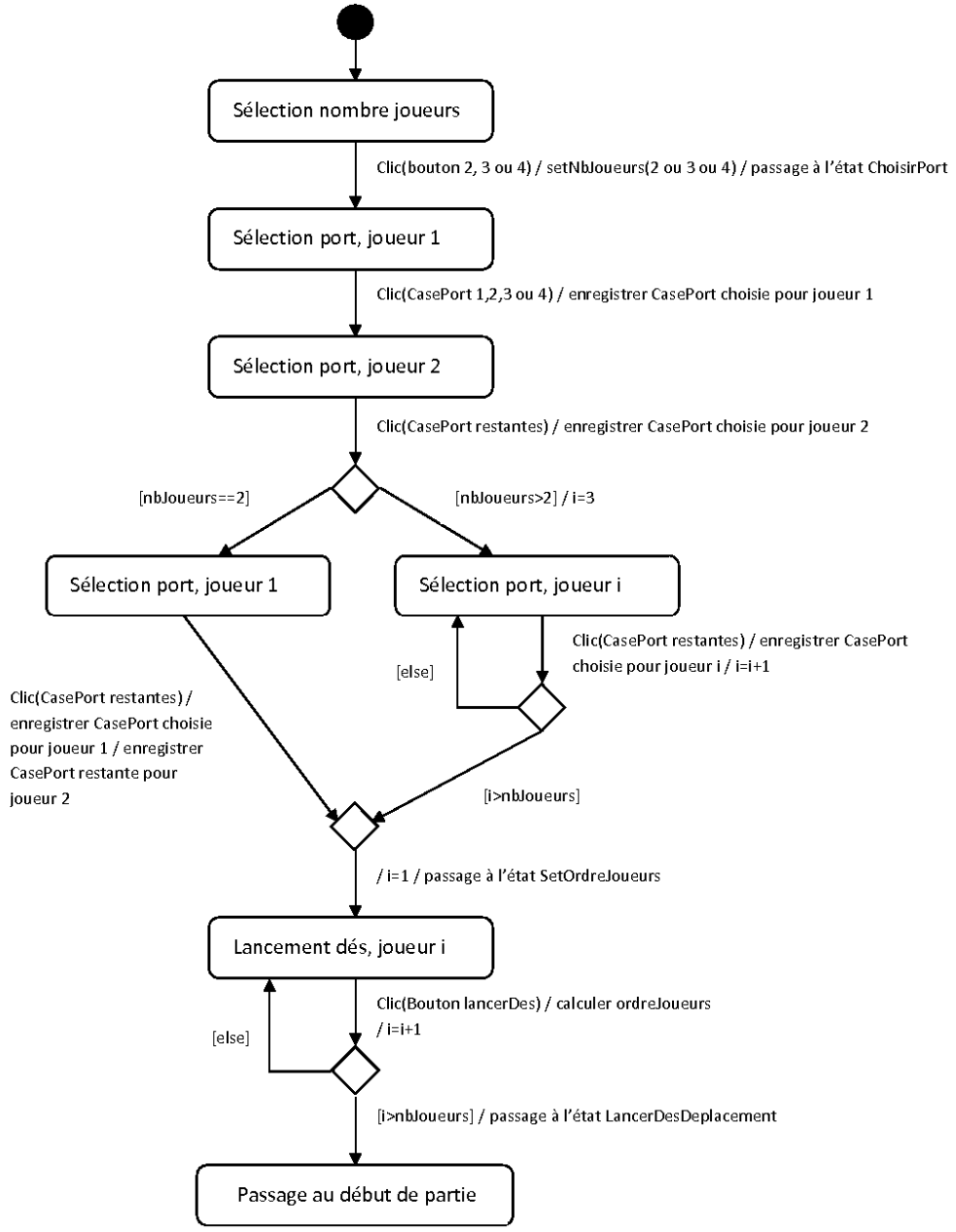
Voici un diagramme d’activités qui illustre le fonctionnement général de l’application. Il décrit pour chaque étape les différents cas qui peuvent se présenter selon la case où le bateau se situe, selon les types de bateaux recevant un tir au canon…


act Activity1
Choix nombre de joueurs
Choix du port pour chaque joueur
Lancer dés pour ordre de commencement
Joueur lance les dés
[nbJoueurs = 2]
[nbJoueurs > 2]
Choisir bateau à déplacer
Joueur courant choisit case finale
[Case Port]
[Case Canon]
Choisir la cible
Choix de l'angle de tir
Choix de la puissance de tir
Tir
[Case Trésor]
[Case mer]
[Bateau est caravelle ou frégate]
[Trésor à bord]
[Pas de trésor à bord]
Déposer trésor
[Score du joueur = 3]
[Score du joueur <3]
Passer en caravelle
[Case pas en bordure]
[Case en bordure]
[Bateau est rafiau]
[Pas de trésor à bord]
[Trésor à bord]
[Joueur attaquable]
[Réussi]
[Trésor sur navire touché]
[Frégate]
Récupérer Trésor
[Caravelle]
[Pas de trésor sur navire touché]
Dégrader bateau touché
[Pas de joueur attaquable]
Duel
[Trésor indisponible]
[Trésor disponible]
Prendre trésor
[Manqué]
Joueur suivant lance les dés
un duel consiste en un échange de tirs entre deux navires : le navire attaquant tire le premier. S'il réussit, le duel s'arrête, avec les changements qui s'en suivent, sinon, le joueur attaqué devient attaquant. Quoiqu'il arrive ensuite, le duel s'arrête.

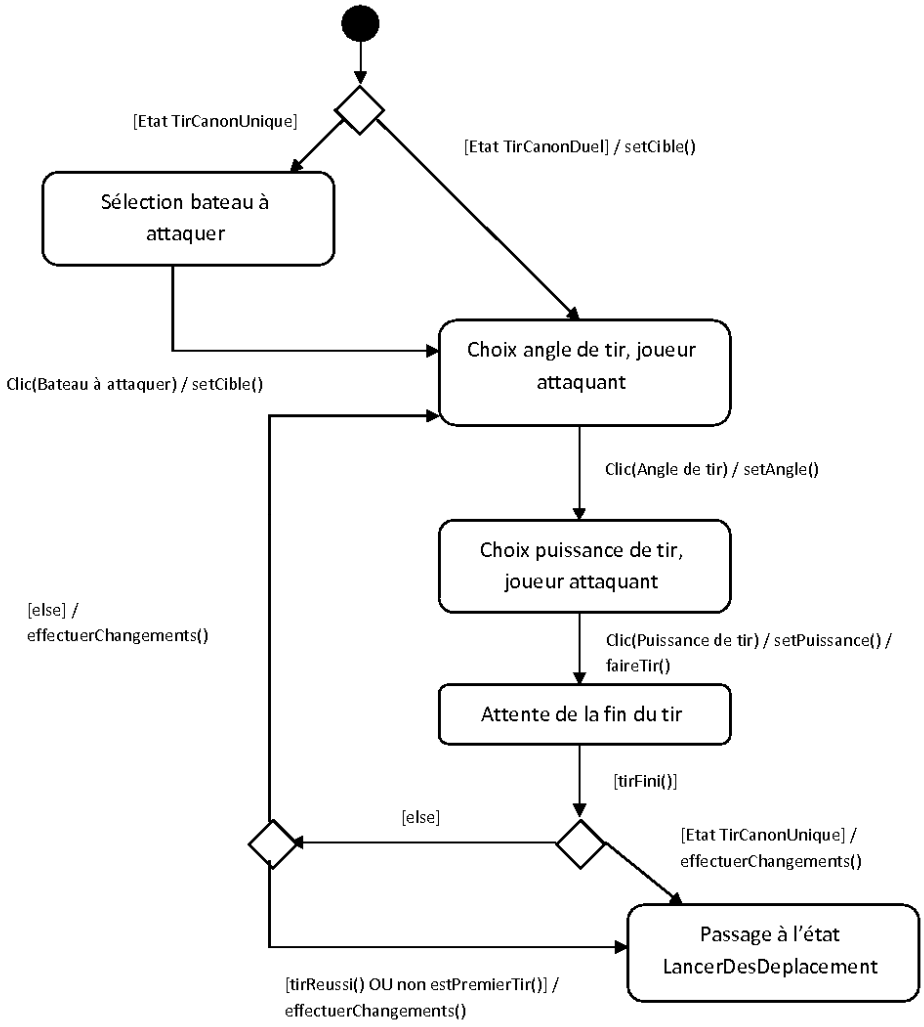
VI. Diagramme d'états-transitions

Les diagrammes d'états-transitions sont utilisés pour décrire les classes complexes.

Ici, en premier lieu pour notre cas, nous avons choisi d'effectuer des diagrammes d'états-transitions pour les classes qui interviennent durant la phase d'initialisation. Ce diagramme décrit donc toutes les étapes et tous les cas qui se présentent lors du démarrage du jeu, jusqu'au commencement de la partie à proprement dite. Les classes décrites sont donc « AttenteNbJoueurs », « ChoisirPort » et « SetOrdreJoueurs ». Le voici ci-dessous :



En second lieu, nous avons édité un diagramme d'états-transitions pour décrire le fonctionnement de la classe TirerCanon, qui est parente des classes TirCanonUnique et TirCanonDuel. Le but est de décrire précisément les étapes qui se succèdent lors du tir au canon, et de montrer les différences qu'il réside entre un tir unique et un tir en duel. Le voici ci-dessous :



VII. Diagramme d'interactions

Nous avons réalisé plusieurs diagrammes de séquences afin de décrire au mieux le fonctionnement de notre application.

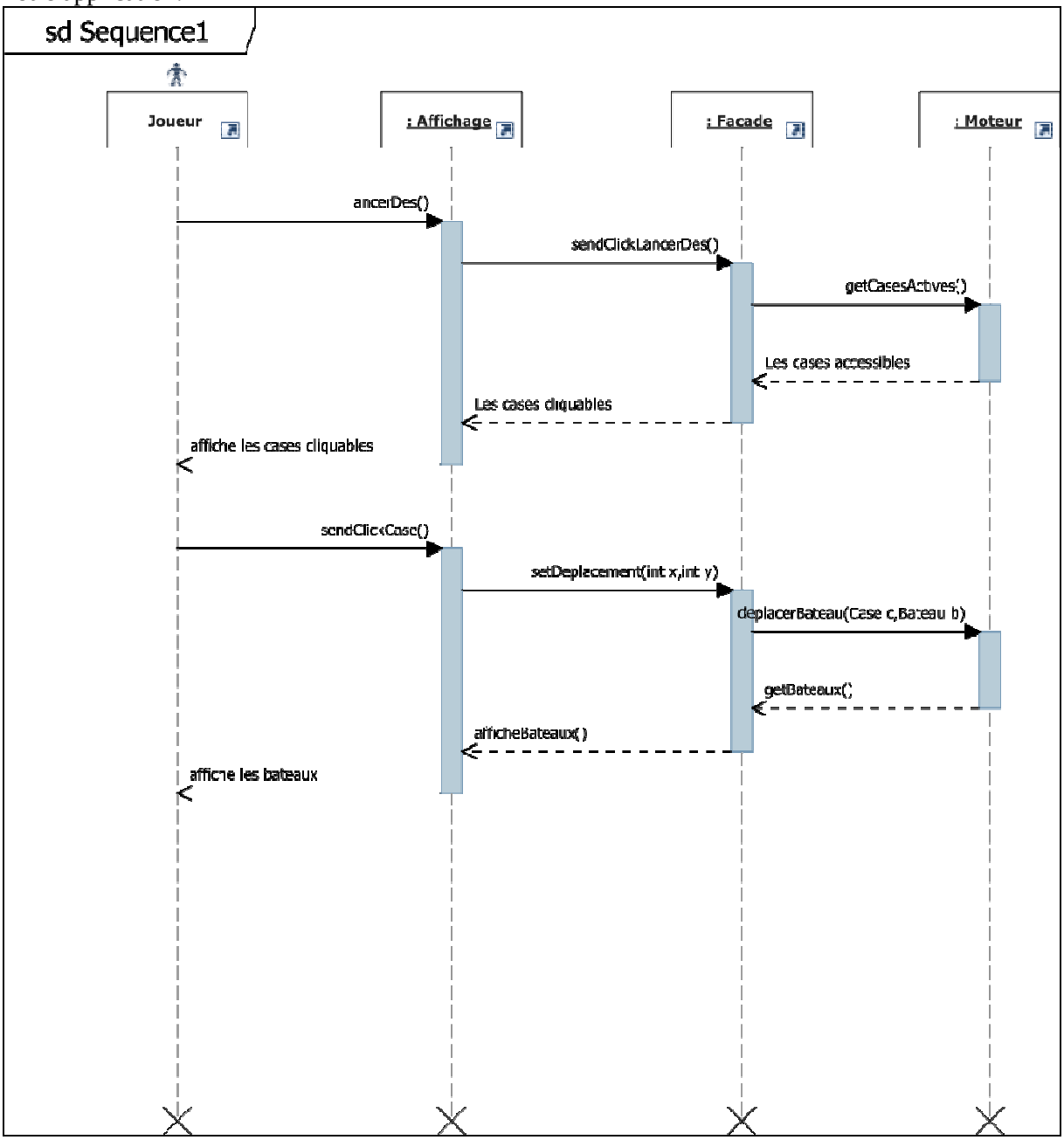


Diagramme de séquences : Déplacement d'un bateau d'un joueur

Ce diagramme met en évidence le déplacement d'un bateau pour le joueur courant, et les actions qu'il a à réaliser pour effectuer ce déplacement. On y voit aussi les interactions entre le moteur et la façade.

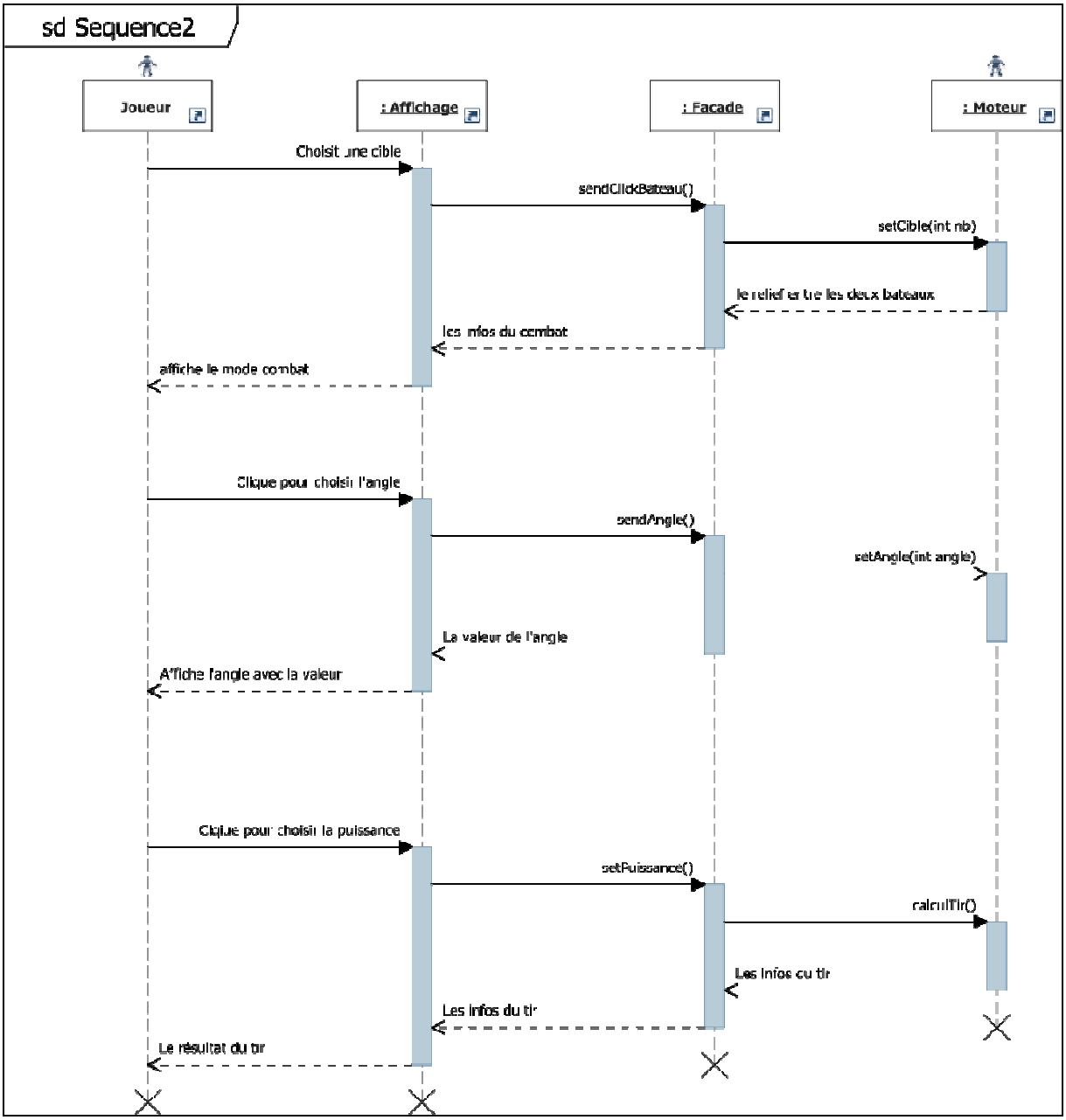


Diagramme de séquences : Tir au Canon unique

Ce diagramme montre le fonctionnement du jeu dans le cas où un joueur arrive sur une case Canon Noir. Il y est décrit l'ensemble des actions qu'il doit faire pour effectuer un tir, et ce qu'il se passe en interne de l'application lorsque ces actions sont effectuées.

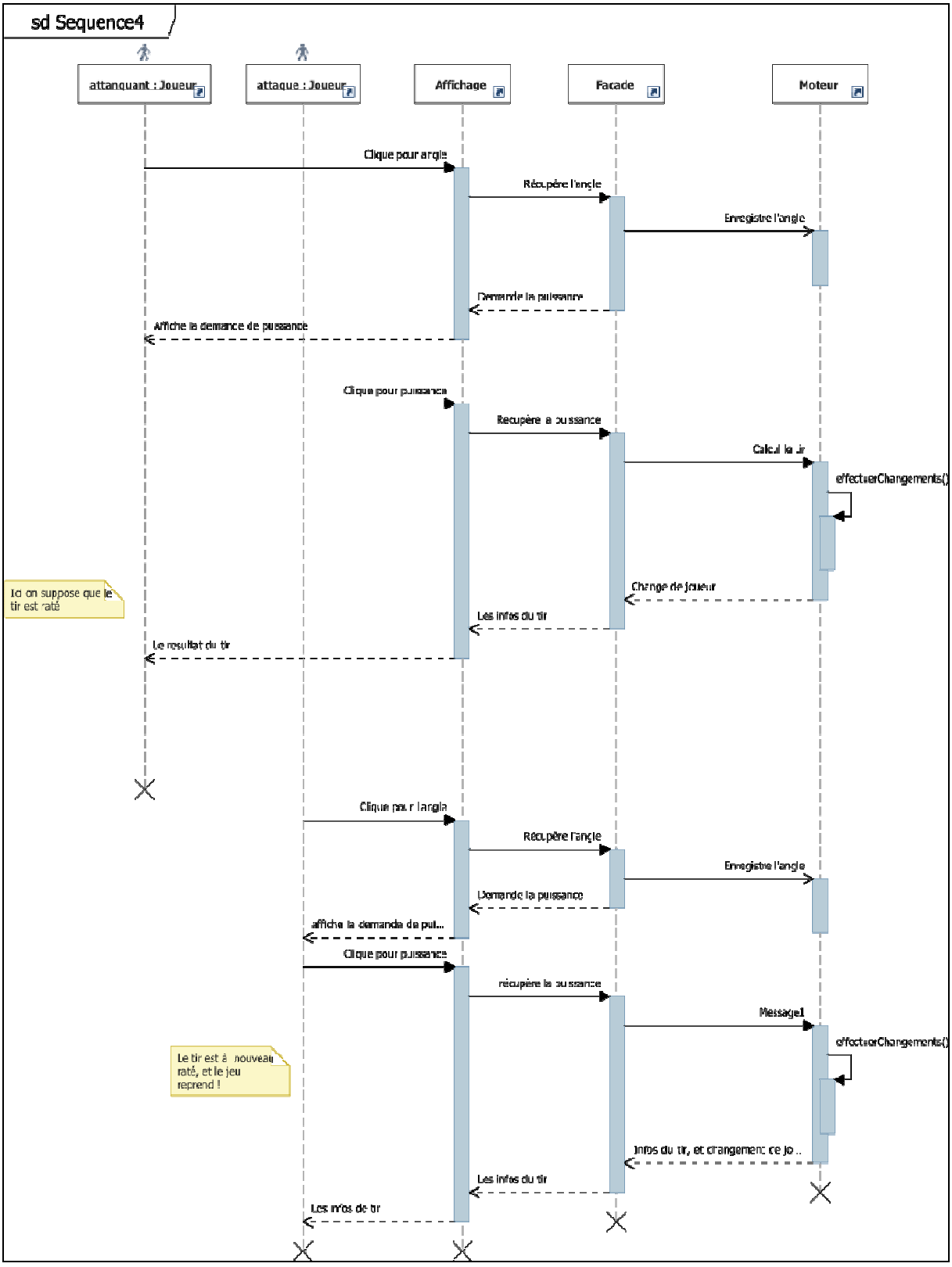


Diagramme de séquence : Duel

Ce diagramme présente le fonctionnement de l'application dans le cas d'un Duel où chacun des joueurs rate son tir. On y voit toutes les informations qui lui sont demandées, et les résultats que cela induit.

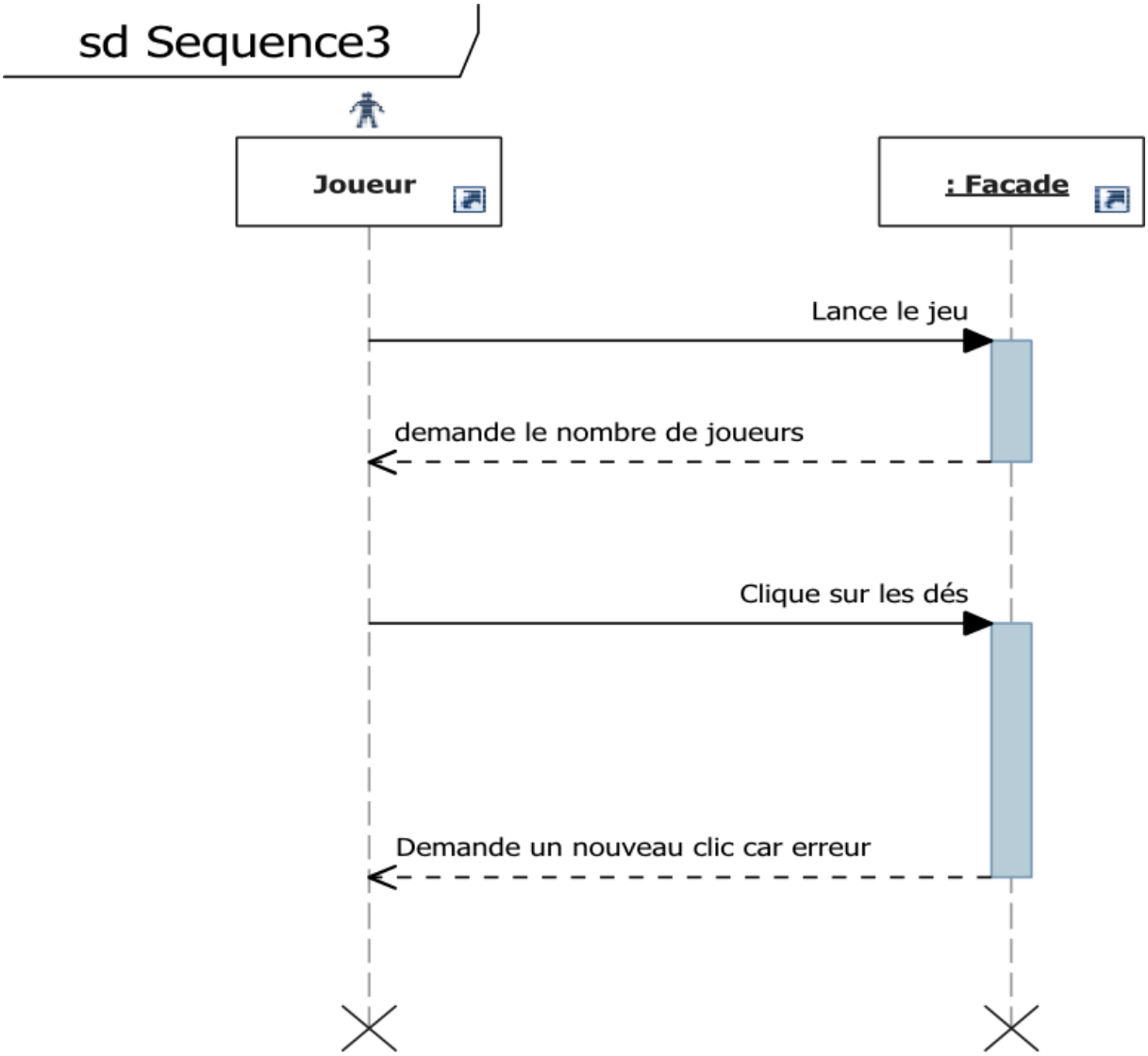
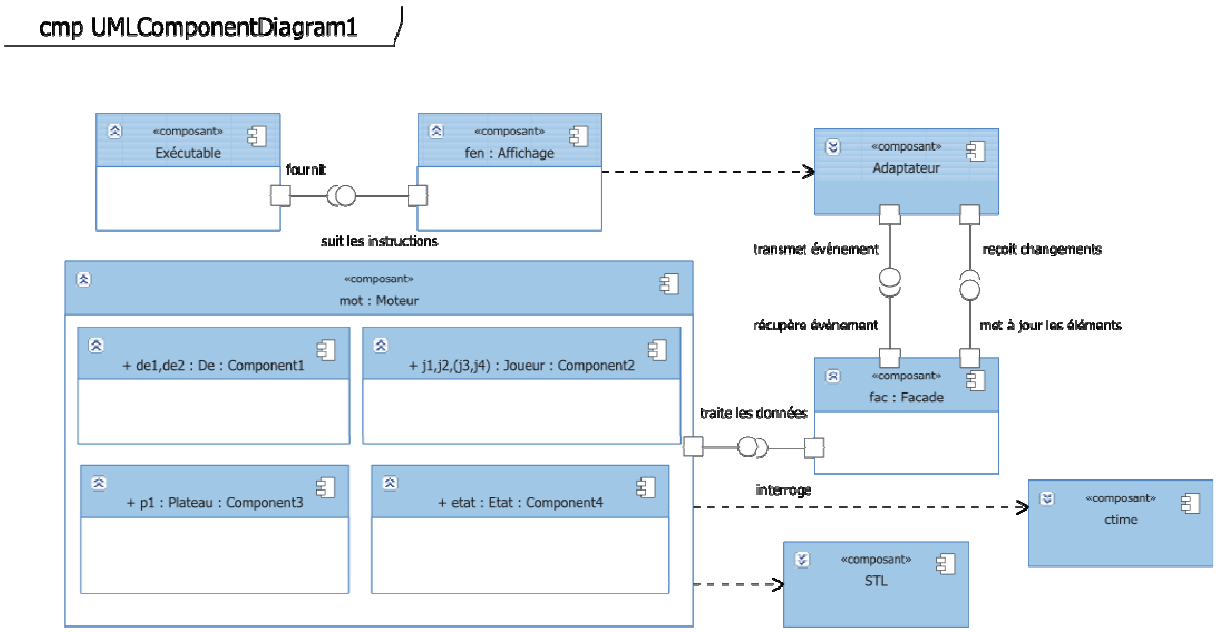


Diagramme de séquences : erreur utilisateur

Ce diagramme présente le comportement de l'application dans le cas où l'utilisateur effectue une erreur de manipulation (un clic qui n'est pas dans les cases qui sont proposées par exemple).

VIII. Diagrammes de composants



IX. Conclusion

Tout au long de ce rapport, nous avons essayé, notamment grâce aux différents diagrammes, de décrire le plus précisément possible la future application « Canon noir ». Tous ces diagrammes nous ont permis à la fois de cibler les besoins, de fixer les idées et de cerner la conception du jeu.

La prochaine étape consiste à générer le code issu de ces diagrammes, afin de pouvoir entamer le développement à proprement dit du jeu en question.