Projet CPOO – 4INFO

Rapport de Conception

*« Jeu du Space Battle »*

Maloeuvre Nicolas  
Vannier Bastien

25/11/11

Table des matières

Introduction3

1 Analyse Conceptuelle4

1.1 Diagrammes Comportementaux4

1.1.1 Diagrammes de cas d'utilisation 4

1.1.2 Diagramme d'activité 7

*1.1.3 Diagrammes d'états-transitions*8

1.1.4 Diagrammes de séquence 12

1.2 Diagrammes Structurels18

1.2.1 Diagramme de classes 18

1.2.2 Diagramme de composants 18

2 Analyse des Choix22

2.1 Choix des patrons de conception22

2.2.1 Les patrons de conception de création 22

2.2.2 Les patrons de conception de structurel 23

2.2.3 Les patrons de conception de comportement 24

2.2.4 Le cas particulier du MVC 24

2.2 Choix des classes25  
2.2.1 Package « Modèle »26  
2.2.2 Package « Vue »26  
2.2.3 Package « Contrôleur »26

Conclusion27

Introduction

« Space Battle » est un jeu de société disposant d’un plateau de jeu représentant une partie du système solaire Alpha du Centaure composé de deux astéroïdes, quatre stations orbitales de quatre couleurs différentes situées aux quatre coins du plateau, quatre cases « artefact », quatre cases « station d'énergie ». La boîte de jeu contient également deux dés à six faces, douze pions (trois vaisseaux par joueur : un vaisseau-mère, un battlecruiser, une capsule de survie) et quatorze artefacts.

Chaque joueur est le commandant d'une flotte. Il choisit la couleur de sa station orbitale et de sa flotte. A quatre joueurs, chacun prend une flotte (vaisseau-mère, battlecruiser et capsule de survie). A trois joueurs, une flotte est laissée de côté. A deux joueurs, chacun prend deux flottes.

Chaque joueur part pour la course aux artefacts. Sur le plateau de jeu sont disposés quatre cases oranges, il s'agit des cases « artefact ». Le but de « Space Battle » réside dans le fait d'atteindre l'une de ces cases, prendre un artefact, l'emporter avec son vaisseau et le ramener à sa station orbitale. Pour être vainqueur, le joueur doit être le premier à réussir cette séquence trois fois. Nous pouvons également noter la présence de quatre cases grises, les cases « station d'énergie ». A partir de ces cases, un joueur recharge son vaisseau en énergie ce qui lui permet dans les tours suivants de viser et tirer sur l'un de ses adversaires depuis n'importe où sur la carte.

Le joueur commence à jouer avec son vaisseau-mère. Si ce dernier se fait détruire à la suite du tir d'un adversaire, le vaisseau-mère est remplacé par le battlecruiser. Enfin si le battlecruiser est détruit, le joueur doit rentrer à sa station orbitale avec sa capsule de survie. Une fois rentré à la station, il lui sera alors possible de repartir avec son vaisseau-mère.

Notre projet est de modéliser, concevoir et programmer, en C++, le jeu du « Space Battle » dont les règles principales ont été énoncées précédemment. Pour y arriver, la première étape consiste en la modélisation de notre future application. A l'intérieur de ce rapport de conception, nous allons vous présenter la modélisation que nous avons effectué grâce à divers diagrammes UML.

Analyse Conceptuelle

L’étude UML[[1]](#footnote-1) nous a permis de modéliser l’application par abstraction de certains détails inutiles, nous permettant d'en simplifier notre vision. Elle nous a également aidé à clarifier les besoins du jeu et les spécificités de l’application afin de la développer. Nous avons concentré notre réflexion sur différents diagrammes UML qui nous semblaient les plus adaptés à notre besoin. Dans un premier temps nous exposerons les diagrammes comportementaux que nous avons réalisé, puis, dans un second temps, les diagrammes structurels.

# Diagrammes Comportementaux

Cette partie regroupe les diagrammes de cas d'utilisation, d'activité, d'états-transitions et de séquences.

## 1.1.1 Diagrammes de cas d’utilisation

Le diagramme de cas d’utilisation permet la représentation de l’implication de l’utilisateur dans l’application et des interactions entre l’utilisateur et le système. On peut ainsi décrire ce que le futur système devra faire, sans spécifier de quelle manière il le fera. Dans notre cas nous avons réalisé trois diagrammes de cas d'utilisation distincts.

* Le premier diagramme (cf. Figure 1, page 5) représente le lancement du jeu et donc l’implication de l’utilisateur dans l’application pour initialiser une partie.
* Le second diagramme (cf. Figure 2, page 5) représente l'implication du joueur dans un tour de jeu. Ce diagramme se focalise uniquement sur les possibilités d’actions du joueur lors d’un tour.
* Enfin le dernier diagramme (cf. Figure 3, page 6) est une version plus poussée et avancée du diagramme précédent. A l’intérieur de celui-ci nous introduisons, en plus de l’implication de l’utilisateur, l’implication du système dans le jeu. Les différentes actions que l’utilisateur pourra effectuer sont directement liées aux principes du jeu et les relations d’inclusion et d’exclusion permettent d’identifier les enchaînements et les interactions avec le système.

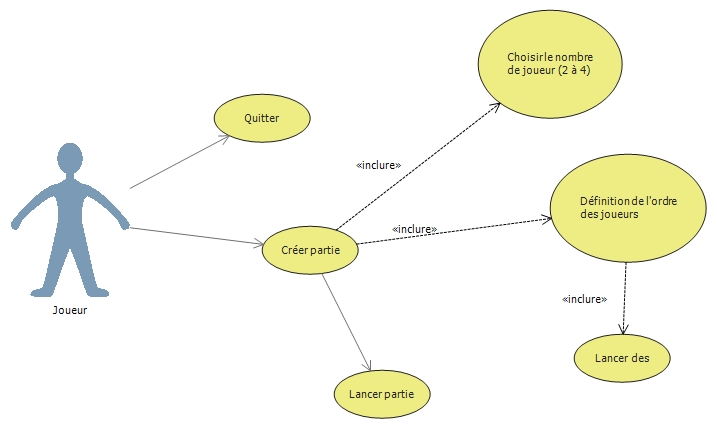


Figure 1. Diagramme cas d'utilisation : Initialisation d'une partie

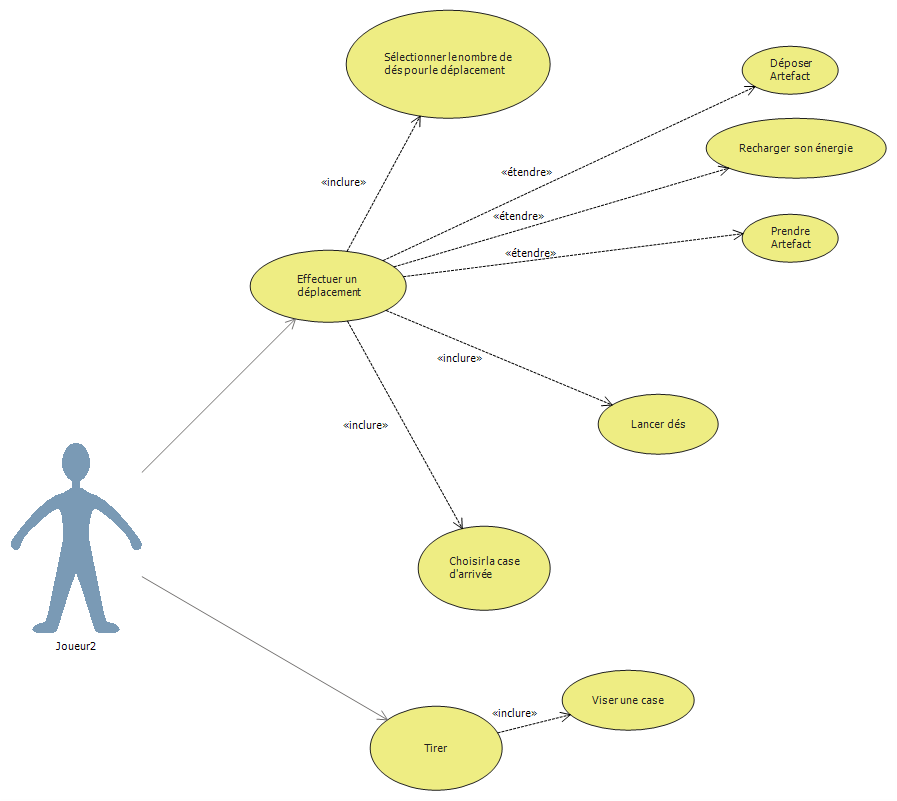


Figure 2. Diagramme cas d'utilisation : Déroulement d'un tour

## 

Figure 3. Diagramme cas d'utilisation : Déroulement d'un tour

## 1.1.2 Diagramme d’activité

Le diagramme d’activité (cf. Figure 5, page 7) permet la représentation du processus d’un tour. On peut ainsi voir toutes les possibilités et tous les cas possibles offerts aux joueurs lors d’un seul tour.

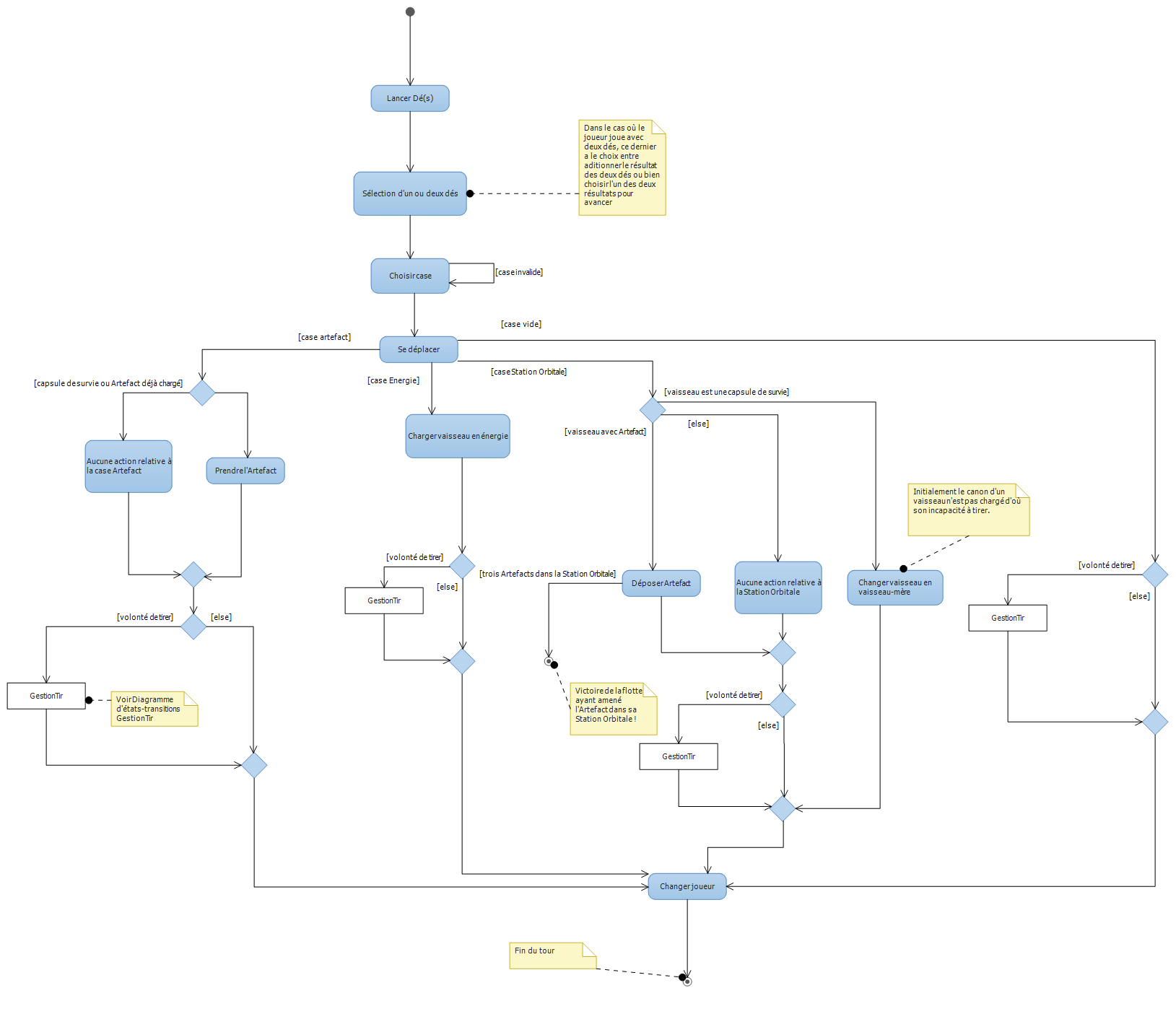


Figure 4. Diagramme d'activité : Déroulement d’un tour

## 1.1.3 Diagrammes d’états-transitions

Afin de clarifier le déroulement d’un tour effectué par un joueur, nous avons décomposé les différentes étapes et possibilités à travers différents diagrammes d’états-transitions dont voici un listing.

* Le diagramme représentant les transitions entre les différents états possibles du jeu (cf. Figure 5, page 8).

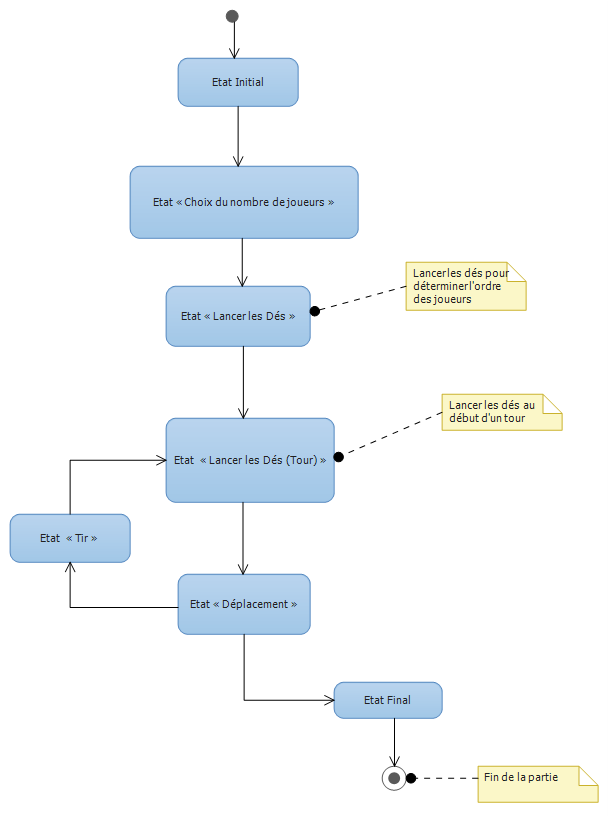


Figure 5. Diagramme d'états-transitions : Etats de jeu

* Le diagramme représentant l’état d’un vaisseau lors de son arrivée sur une case de type « station d'énergie » :

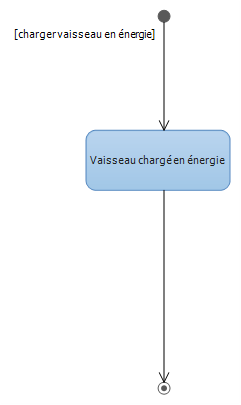
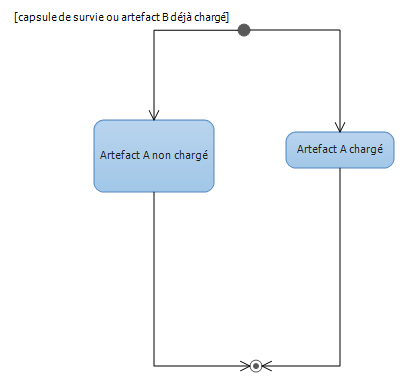


Figure 6. Diagramme d'états-transitions : Case « station énergie »

* Le diagramme représentant les états lors de l’arrivée d’un vaisseau sur une case de type « artefact » :

Figure7. Diagramme d'états-transitions : Case « artefact »



* Le diagramme représentant les états lors de l’arrivée d’un vaisseau sur une case de type « station orbitale » :

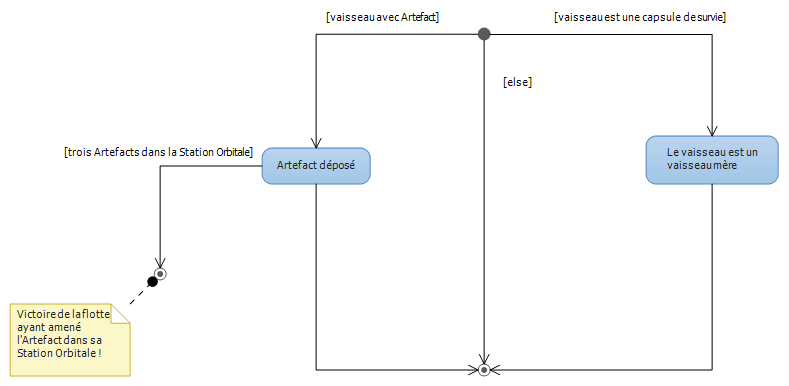


Figure 8. Diagramme d'états-transitions : Case « station orbitale »

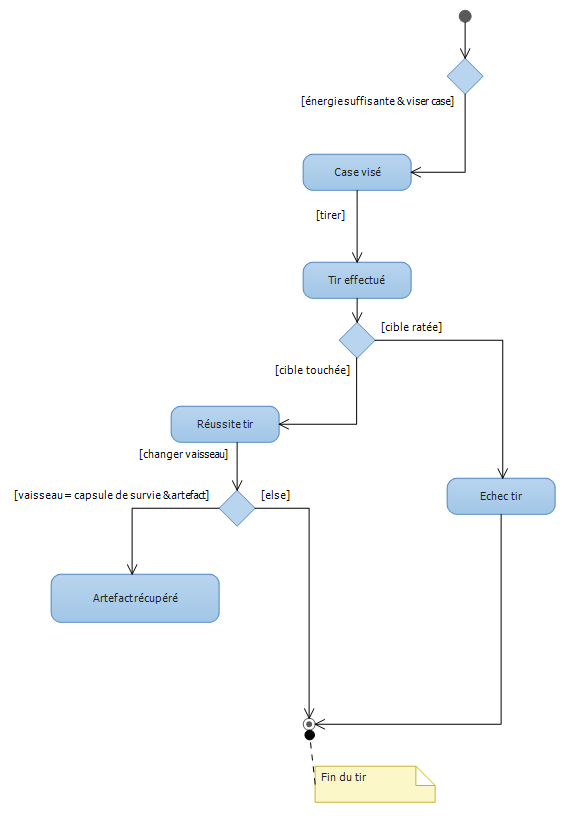
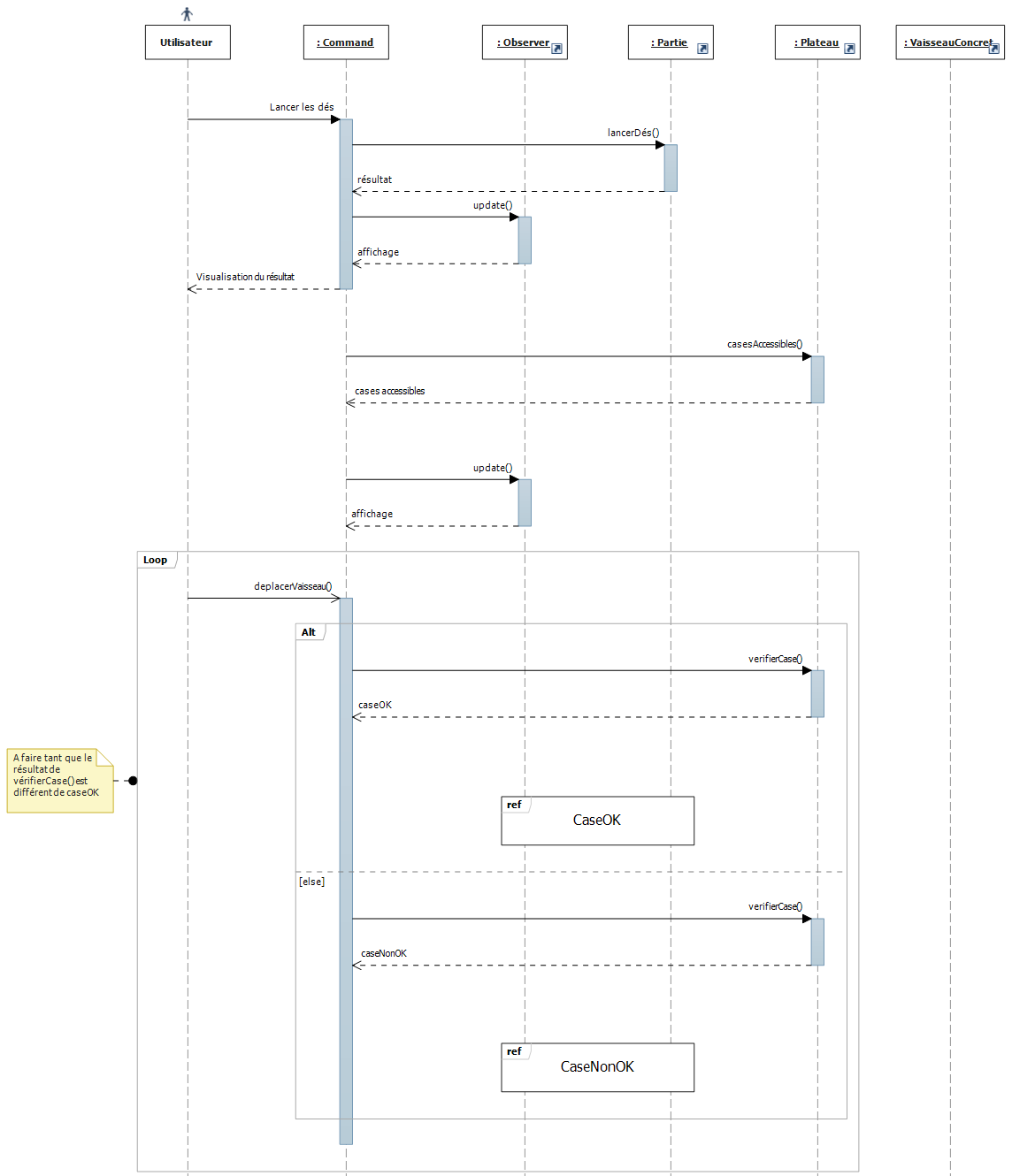
* Le diagramme représentant les transitions et états lors d’une phase de tir :

Figure 9. Diagramme d'états-transitions : Tir

## 1.1.4 Diagrammes de séquence

Nous avons réalisé divers diagrammes de séquence représentant certains scénarios disponibles à l’intérieur du jeu.

* Le premier diagramme de séquence représente le début du tour d’un joueur avec le lancer des dés, l’affichage des cases accessibles et la sélection de l’une de ces cases par le joueur. Le rectangle symbolisant l’interface « Command », représente le contrôleur du jeu. Ce dernier récupère les évènements provoqués par l’utilisateur pour effectuer les traitements associés. Le rectangle « Observer » caractérise la vue du jeu. Les autres rectangles représentent différentes classes du modèle. Les lignes verticales pointillées représentent les lignes de vie des objets.

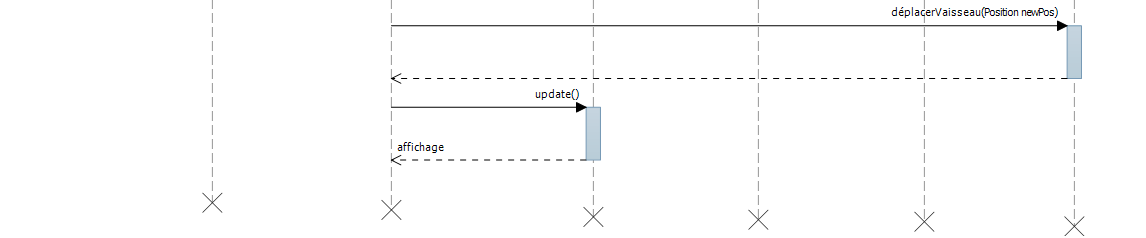
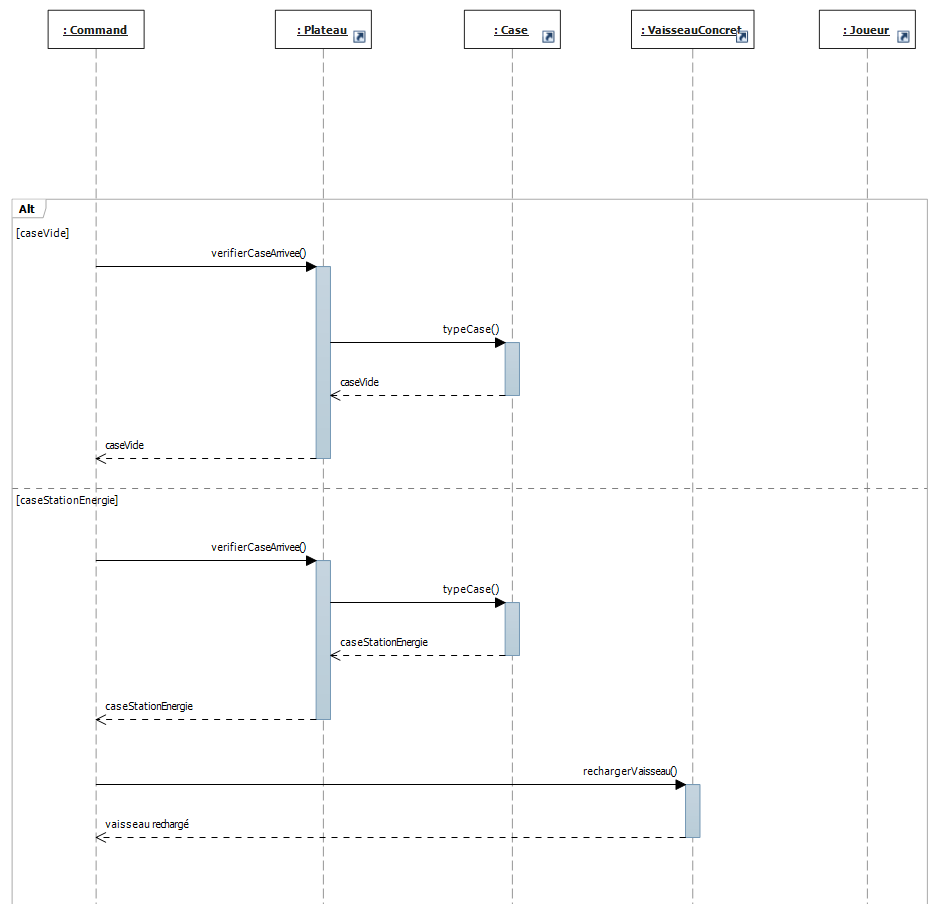
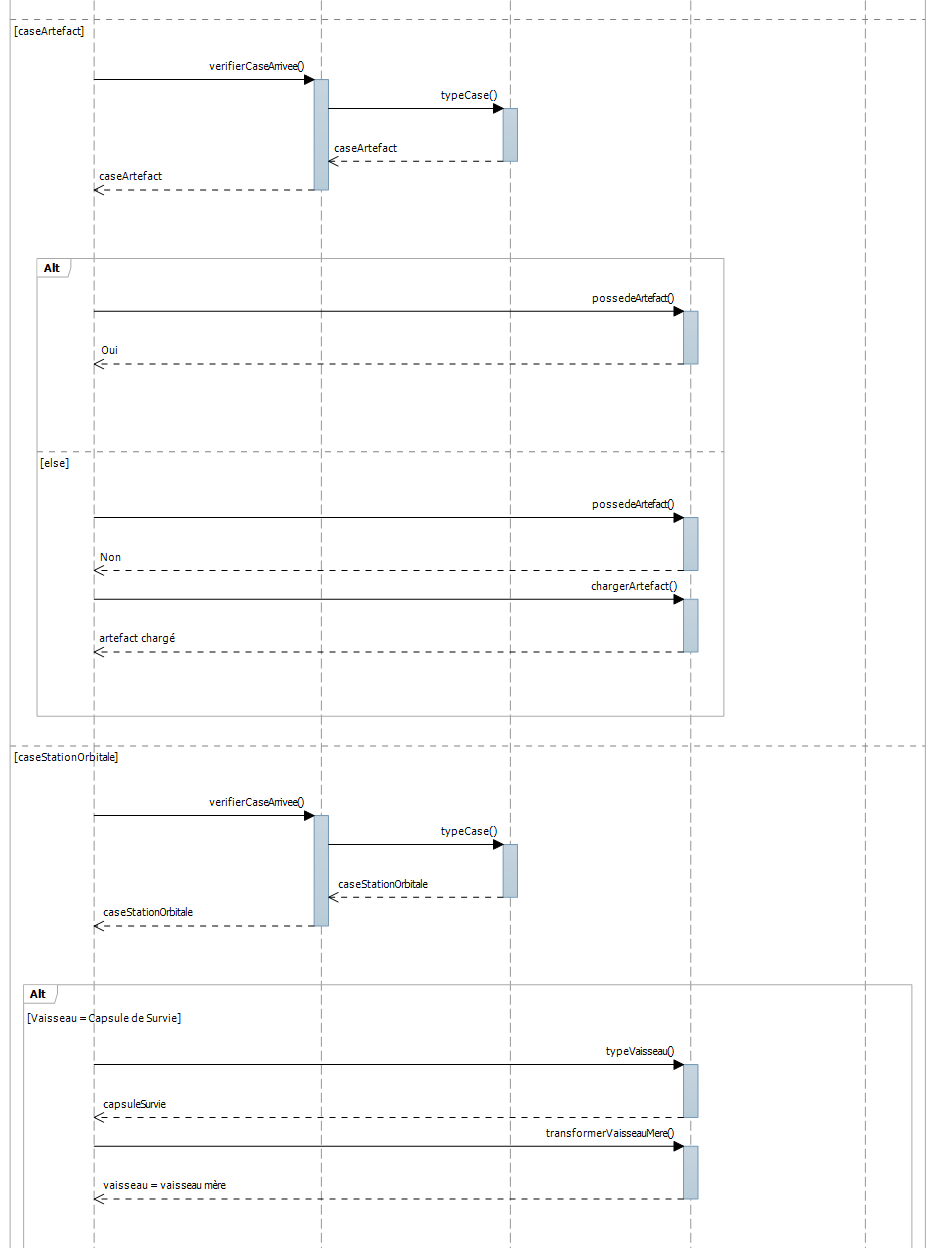


Figure 10. Diagramme de séquence : Début tour

* Le second diagramme de séquence représente les différents scénarios possibles à la fin du déplacement d’un vaisseau. En effet la séquence des événements va être différente selon le type de case sur laquelle le vaisseau arrive. Ce diagramme fait intervenir des composantes alternatives.





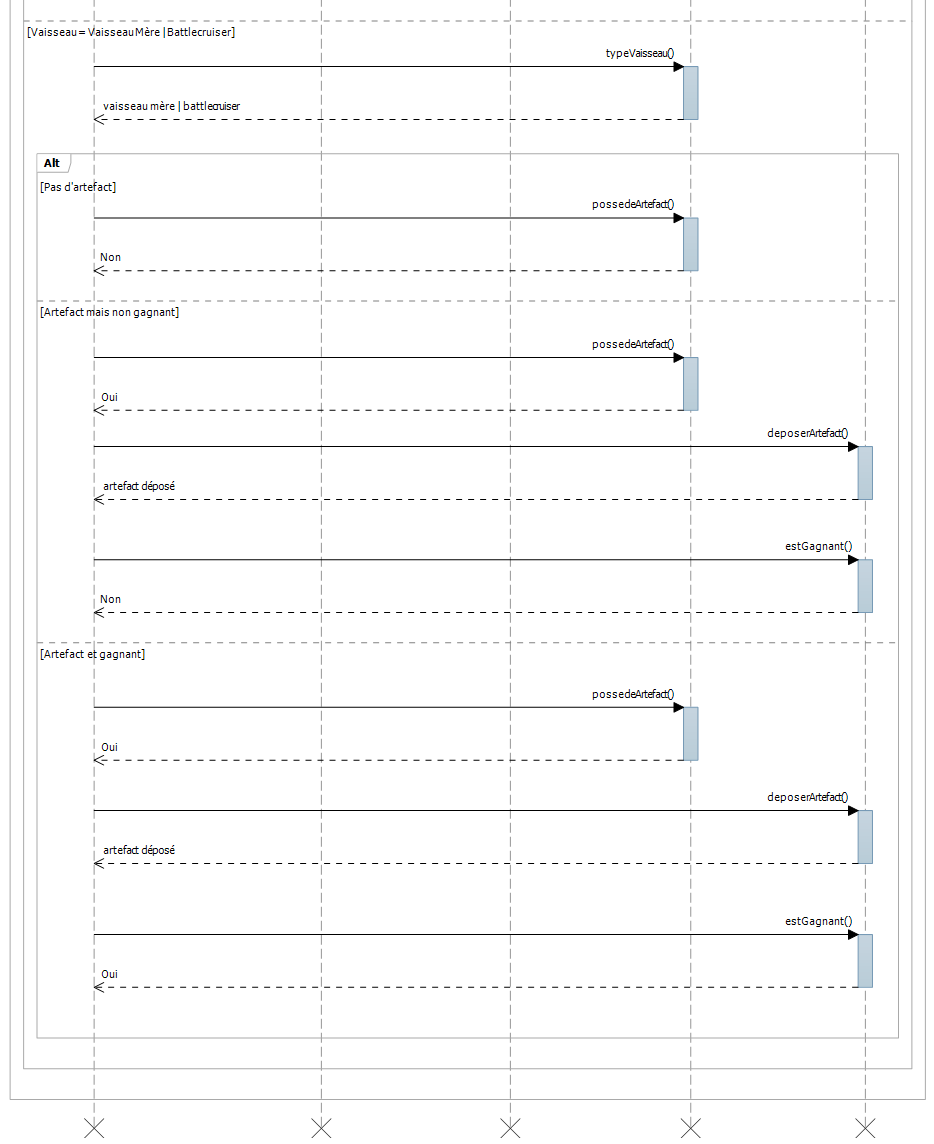


Figure 11. Diagramme de séquence : Fin déplacement

* Le troisième diagramme de séquence représente deux scénarios d’erreurs en se basant dans le cadre d’un tir. Le premier représente le cas d’un tir effectué alors que le nombre de charges d’énergie est insuffisant. Le second représente le cas d’un tir manquant sa cible.

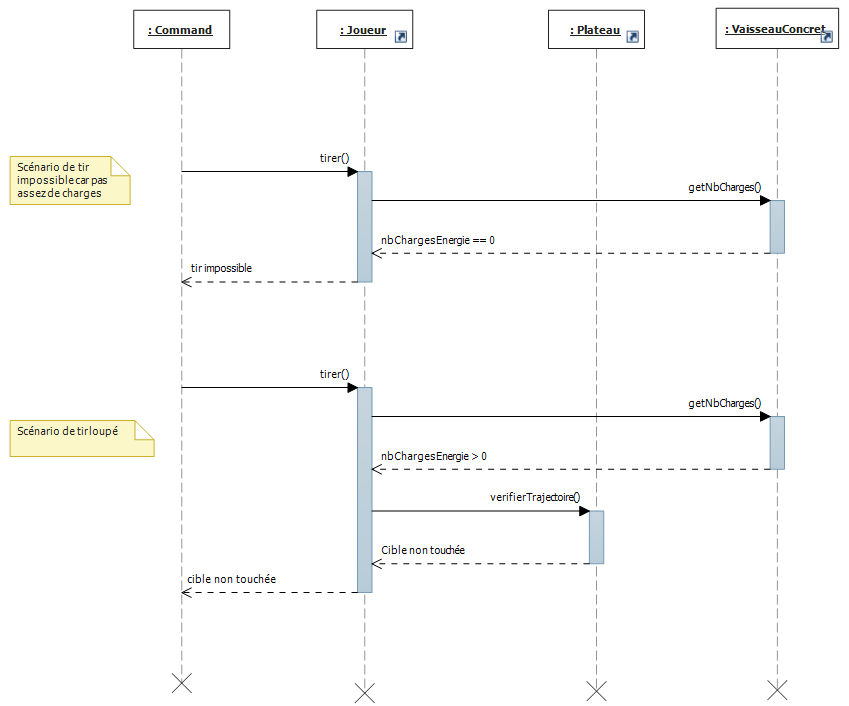


Figure 12. Diagramme de séquence : Tir

* Le quatrième diagramme représente la séquence des événements lors de la création d’une partie par l’utilisateur.

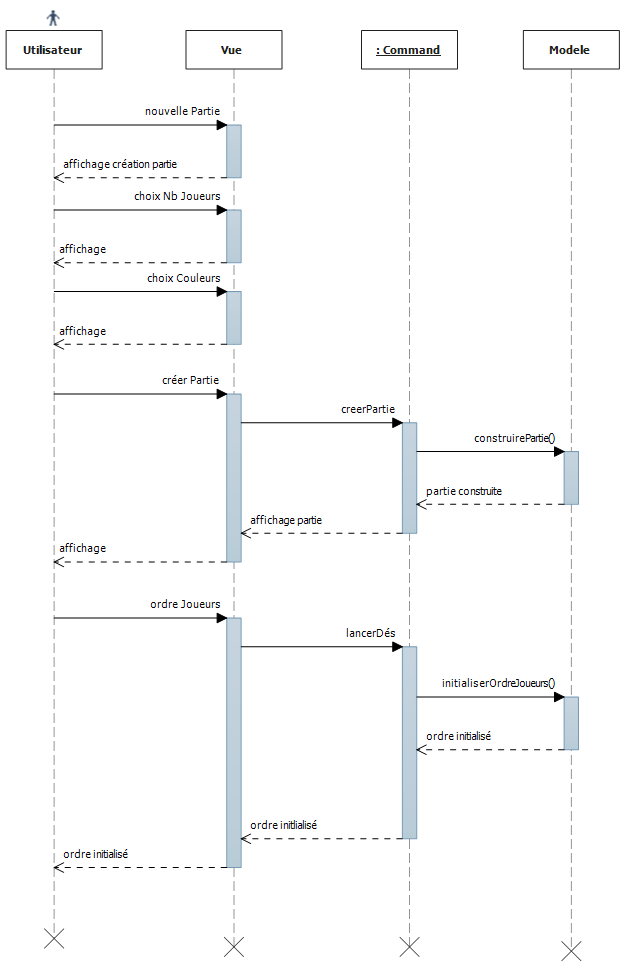


Figure 13. Diagramme de séquence : Création partie

# Diagrammes Structurels

## Diagrammes de classes

Nous avons réalisé trois diagrammes de classes basés sur le concept Modèle-Vue-Contrôleur (cf. Section « Choix des Classes », page 24 pour des explications annexes).

* Diagramme de classes « Modèle », cf Figure 14 page 19.
* Diagramme de classes « Vue », cf Figure 15 page 20.
* Diagramme de classes « Contrôleur », cf Figure 16 page 21 .

## 1.2.2 Diagramme de composants

Le diagramme de composants décrit l'organisation du système du point de vue des éléments logiciels comme les modules (paquetages, fichiers sources, bibliothèques, exécutables), des données (fichiers, bases de données) ou encore d'éléments de configuration (paramètres, scripts, fichiers de commandes). Ce diagramme permet de mettre en évidence les dépendances entre les composants (qui utilise quoi).

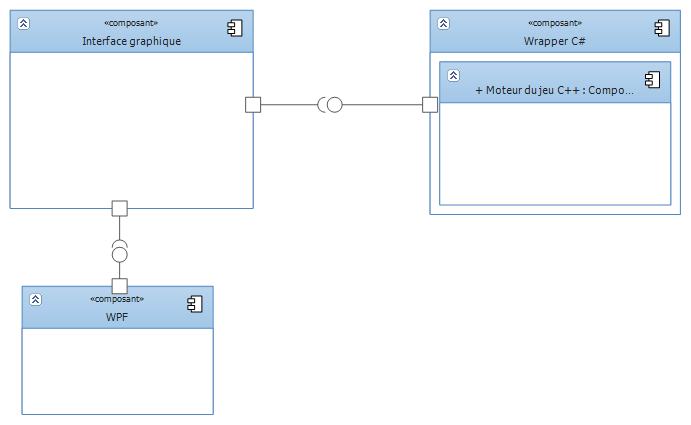


Figure 17. Diagramme de composants

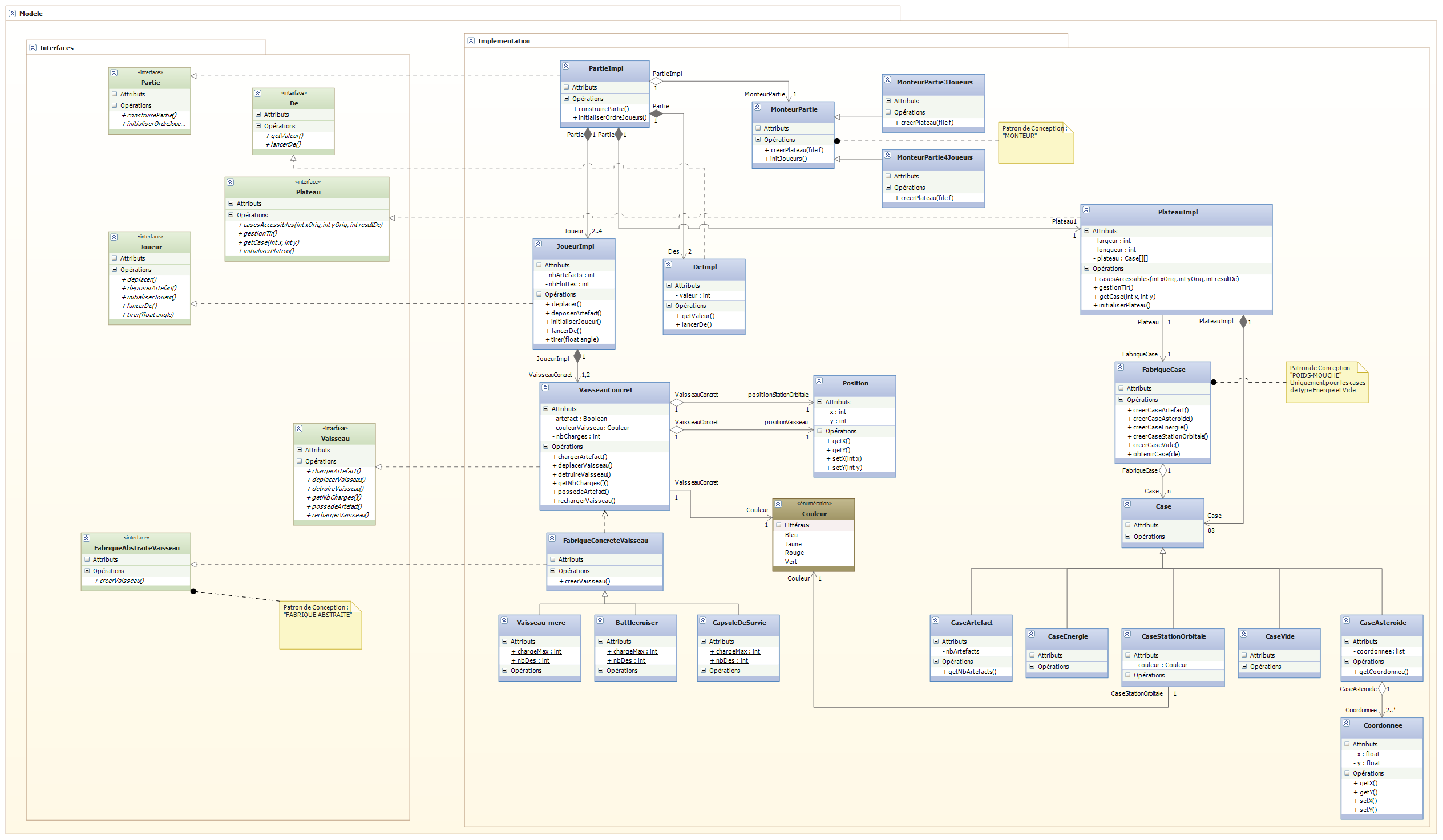


Figure 14. Diagramme de classes : Modèle

## 

Figure 15. Diagramme de classes : Vue

## 

Figure 16. Diagramme de classes : Contrôleur

Analyse des Choix

# 2.1 Choix des patrons de conception

Vous trouverez ici, les différents Design Patterns[[2]](#footnote-2) utilisés dans le cadre de notre projet. Ces derniers sont reconnus comme étant de bonnes pratiques de modélisation en réponse à des problèmes de conception logicielle connus et identifiés.

Ces derniers sont divisés en trois grandes parties distinctes, les patrons de conception de « création », de « structure » et ceux de « comportement ».

## 2.1.1 Les patrons de conception de création

Les patrons de conception de création sont utiles afin de résoudre des problèmes de conception liés à la création de nouveaux objets.

### Fabrique Abstraite / Abstract Factory

Le but de ce patron de conception est de fournir une interface pour la création des familles d’objets sans spécifier leurs classes concrètes. Ce dernier sera utilisé, à titre pédagogique, dans le cas des vaisseaux. En effet, dans le cadre initial de notre projet, ce patron de conception ne semble pas être nécessaire à mettre en place. Néanmoins, il est intéressant à mettre en place dans le cas d’implémentations futures qui pourraient être mises en place comme par exemple une implémentation des vaisseaux en 2D ou en 3D.

### Monteur / Builder

Ce Design Pattern est utilisé afin de séparer la création d’objets complexes de leur représentation. De plus, le même processus de création peut servir à créer différents objets complexes. Dans le cadre du jeu, ce dernier servira à créer et mettre en place une partie, c’est à dire instancier l’ensemble des composantes, objets nécessaires au déroulement d’une partie.

## 2.1.2 Les patrons de conception de structure

Cet ensemble de patrons de conception s’occupe de la structuration et de la relation des classes entre elles.

### Poids-Mouche / Flyweight

Ce patron de conception est utilisé dans le but de partager efficacement un grand nombre de petits objets. Ce dernier semble tout à fait approprier à être utilisé dans le cas des différentes cases du plateau. En effet, seules quelques instances de la classe *Case* seront instanciées (les différents types de cases) pour l'ensemble du plateau. Il sera judicieux de l’utiliser pour les cases du type *CaseEnergie* et *CaseVide*. En effet ces dernières sont identiques pour l’ensemble des cases du même type sur le plateau. Par contre les cases telles que *CaseArtefact* ou *CaseStationOrbitale* sont différentes pour chacune des cases de ce type. Par exemple, chaque *CaseArtefact* possède son propre nombre d’artefacts.

### Façade / Facade

Le patron de conception *Façade* fournit une interface commune à un ensemble d'interfaces pour gérer cet ensemble plus facilement. Il forme donc un point d'entrée simplifié dans une API[[3]](#footnote-3). La façade permet en effet de limiter les points d'entrée d'une API, ce qui a comme effet bénéfique pour l'ensemble des classes, s'il est complexe, d'avoir une chance d'être utilisé plus simplement.

Lorsque l'on développe un module, fonctionnel ou technique, les chances pour que l'interface de ce module soit importante sont grandes. En effet, si l'on considère ici l'interface comme étant l'ensemble des méthodes (et attributs) publiques de toutes les classes présentes, et que l'on part sur un ensemble de dix classes ayant chacune cinq méthodes publiques (petit module), on se retrouve avec 50 méthodes à découvrir. L'architecture n'est pas lisible au premier coup d'œil, il faut donc simplifier l'utilisation de l'API.

L'utilisation d'une façade, qui transforme l'ensemble des classes en une composante, facilite l'utilisation et donne du même coup une chance pour que les efforts du programmateur servent plusieurs fois en toute simplicité.

Dans le cadre de notre application, ce Design Pattern pourra être intéressant à être utilisé. Néanmoins, ce dernier n’apparaît pas en l’état actuel des choses sur les diagrammes de classes réalisés.

## 2.1.3 Les patrons de conception de comportement

Cette dernière classe de patrons de conception se charge de la maîtrise des interactions entre différents objets.

### Commande / Command

Ce patron de conception permet d’encapsuler une requête dans un objet pour permettre diverses opérations sur la requête. Il permet de séparer complètement le code initiateur de l’action, du code de l’action elle-même. Ce patron de conception est souvent utilisé dans les interfaces graphiques où, par exemple, un objet peut-être connecté à différentes Commandes de façons à ce que l’objet n’ait pas besoin de connaître les détails de l’action effectuée par la Commande.

### Observateur / Observer

Ce Design Pattern est utilisé pour envoyer un signal à des modules qui jouent le rôle d’observateur. En cas de notification, les observateurs effectuent alors l’action adéquate en fonction des informations qui parviennent depuis les modules qu’ils observent. Dans notre cas, ce dernier permet de synchroniser la vue du plateau avec le modèle de données. Il lie un objet source à des observateurs pour que, lorsque la source est modifiée, ses observateurs soient avertis.

## 2.1.4 Le cas particulier du MVC

Le MVC (Modèle – Vue – Contrôleur) est une architecture et méthode de conception qui organise l’intreface homme-machine (IHM) d’une application. Ce dernier divise l’IHM en un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (logique de contrôle, gestion des événements, synchronisation), chacun ayant un rôle précis dans l’interface.

### Le modèle

Le modèle représente les données de l'application. Il assure la gestion de ces dernières et garantit leur intégrité. Le modèle offre des méthodes pour mettre à jour ces données (insertion, suppression, changement de valeur). Il offre aussi des méthodes pour récupérer ces données. Les résultats renvoyés par le modèle sont dénués de toute présentation.

### La vue

La vue correspond à l'interface avec laquelle l'utilisateur interagit. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toutes les actions de l'utilisateur (clic de souris, sélection d'une entrée, boutons, etc). Ces différents événements sont envoyés au contrôleur. La vue n'effectue aucun traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur.

### Le contrôleur

### Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de l'utilisateur et enclenche les actions à effectuer. Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle, ce dernier avertit la vue que les données ont changé pour qu'elle se mette à jour. Certains événements de l'utilisateur ne concernent pas les données mais la vue. Dans ce cas, le contrôleur demande à la vue de se modifier. Le contrôleur n'effectue aucun traitement, ne modifie aucune donnée. Il analyse la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande.

# 2.2 Choix des classes

Les classes que nous avons choisies découlent directement des patrons de conception utilisés et des diagrammes de séquence et d'activité.

Nous pouvons noter la présence de trois packages principaux liés à notre architecture de type MVC. Nous trouverons donc les packages « Modèle », « Vue » et « Contrôleur » qui communiqueront ensemble.

## 2.2.1 Package « Modèle »

Les classes liées au package « Modèle » représente le modèle des données. Nous trouverons à l’intérieur de ce dernier les différentes interfaces avec leurs classes d’implémentation associées.

## 2.2.2 Package « Vue »

Les classes liées au package « Vue » représente les différentes vues nécessaires au fonctionnement de notre jeu. Nous pouvons y trouver des interfaces telles que *VuePlateau*, *VueDes*, *VueVaisseau* etc associées à leur classe d’implémentation. L’un des particularités de ce package est la classe *Observer* liée au patron de conception de même nom, permettant de notifier les différentes vue lors d’un changement.

## 2.2.3 Package « Contrôleur »

Les classes liées au package « Contrôleur » représente les différentes possibilités d’actions pour l’utilisateur. Nous pouvons noter la présence de la classe *Command* liée au patron de conception de même nom. De cette classe héritent divers autres classes telles que *DeplacerVaisseau* ou bien encore *Tirer*. De plus à l’intérieur de ce package, le package « Instruments » permet de définir les divers moyens d’interactions de l’utilisateur avec l’application. Dans notre cas, nous avons la possibilité d’utiliser la main, par l’intermédiaire de la souris, ainsi que le clavier.

Conclusion

La modélisation exposée dans ce rapport va nous permettre de passer à l’étape suivante de notre projet consistant au développement de notre jeu.

La génération automatique du code sera possible et se fera à l’aide de notre modélisation via Visual Studio. Elle permet de convertir un projet de modélisation en code source. Avec ses outils de modélisation, Visual Studio permet de synchroniser le code source et le modèle UML d’un projet. Il permet ainsi la génération automatique de code et la mise à jour du diagramme de classes suite à une modification du code source. Ceci permet au développeur d’avoir toujours accès à une vue du modèle de son projet.

Néanmoins ces fonctionnalités sont disponibles pour les langages .NET, C# et VB. Ceci n’est pas le cas pour le C++. Il nous sera tout de même possible d’utiliser cette fonctionnalité, sans le mécanisme de rétro-ingénierie, à l’aide de templates de génération de code source en utilisant la syntaxe T4 (Text Transformation Template Toolkit).

1. Unfied Modeling Language [↑](#footnote-ref-1)
2. patrons de conception en anglais [↑](#footnote-ref-2)
3. Application Programming Interface [↑](#footnote-ref-3)