UNIVERSITÉ DU QUÉBEC À MONTRÉAL

RAPPORT DE TRAVAIL: BATAILLE NAVALE

PRÉSENTÉ À L'UNIVERSITÉ DU QUÉBEC À MONTRÉAL
COMME EXIGENCE DU COURS INF5153
À M. CARL SIMARD

PAR
ALEXANDRE CROISETIÈRE(CROA11029807)
JÉRÉMIE GOUR(GOUJ18089208)
FABRICE MATHURIN(MATF25039105)

DÉPARTEMENT D'INFORMATIQUE
HIVER 2021

Diagramme de classes (voir l'annexe pour une plus grande résolution)

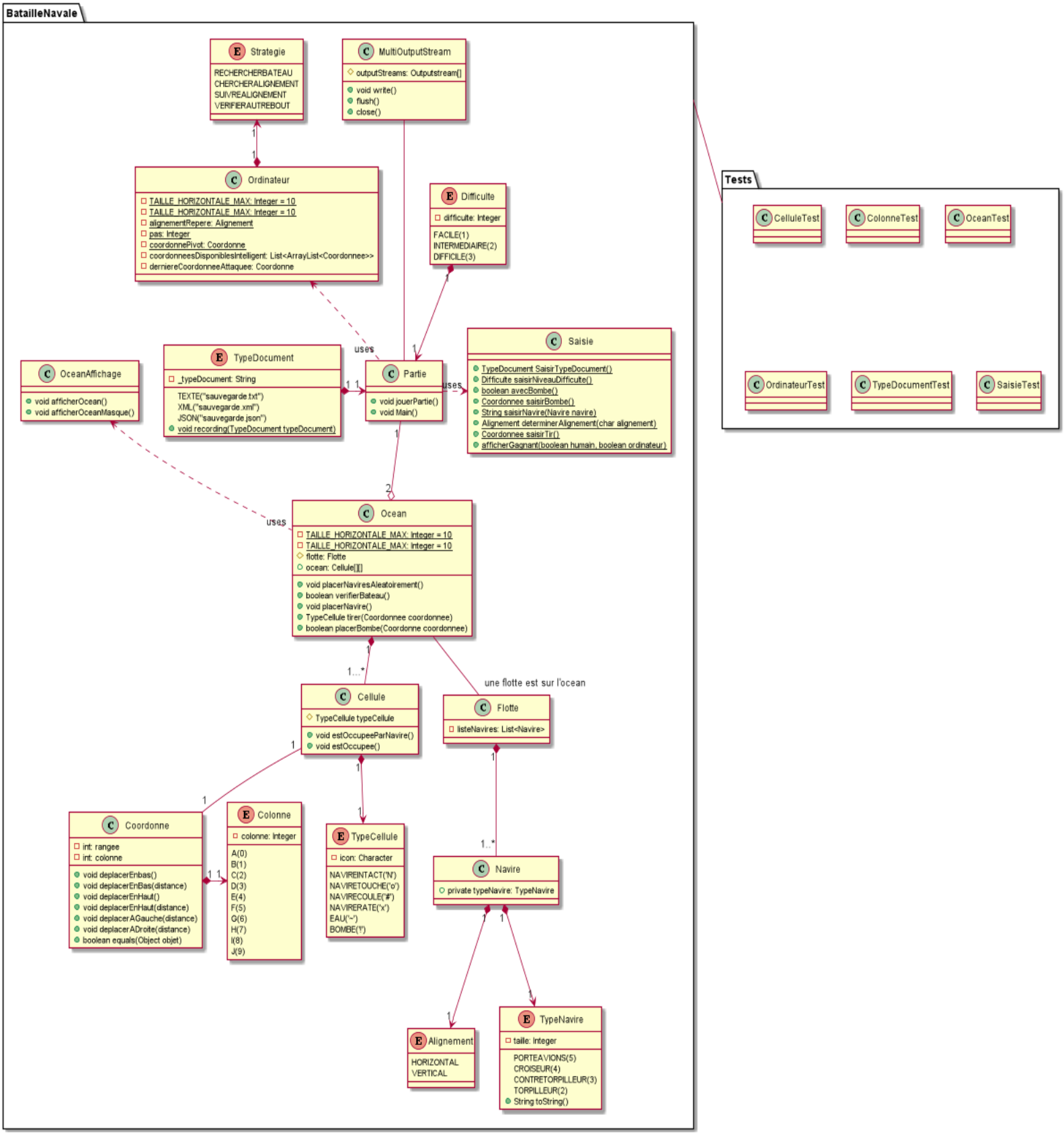

Diagramme de séquence (voir l'annexe pour une plus grande résolution)

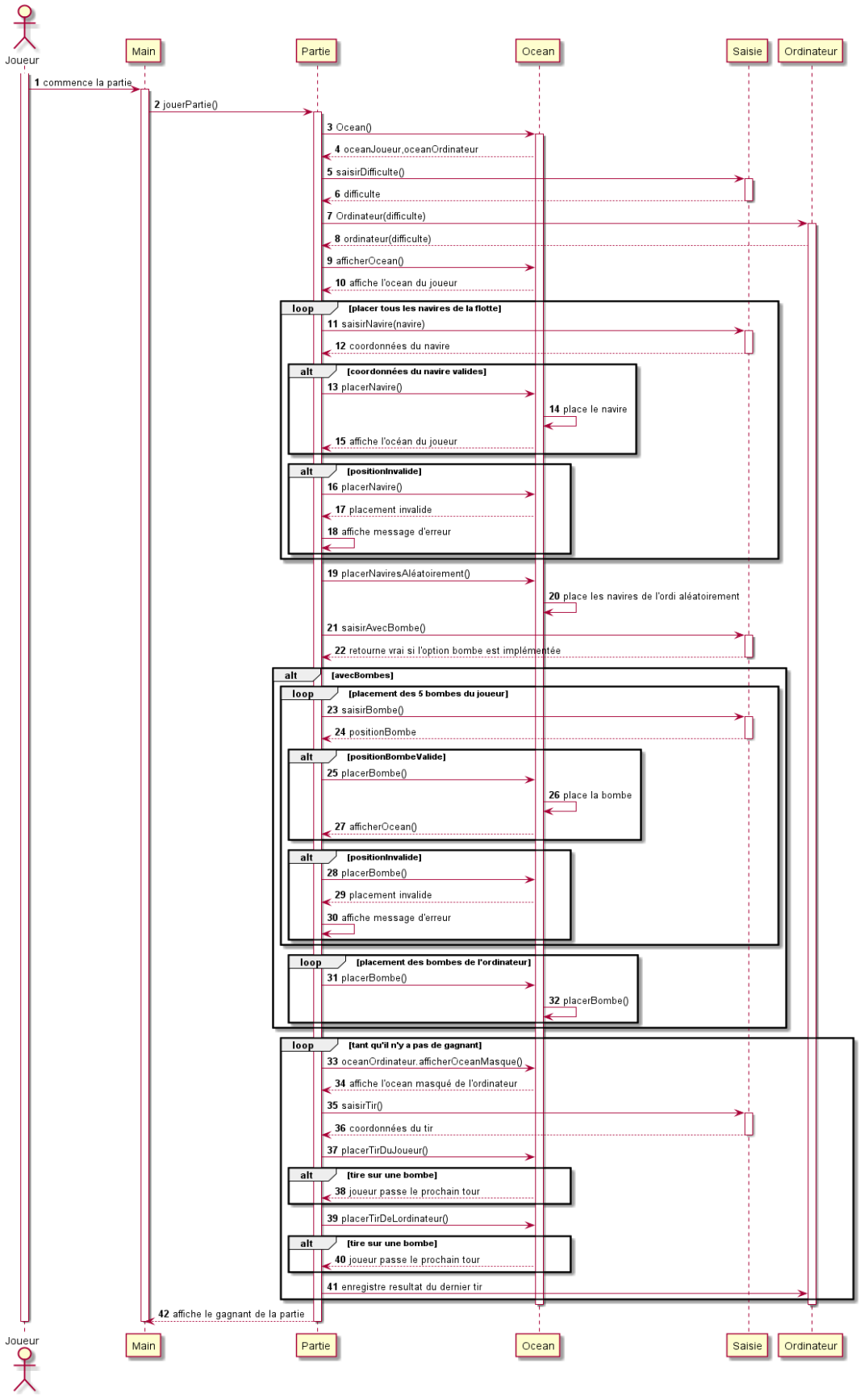

Diagramme de classes

Nous avons pris la décision de modéliser le jeu de bataille navale selon une conception simple. On remarque rapidement que beaucoup des fonctionnalités sont déléguées à la classe Océan, mais il sera justifié dans les sections suivantes pourquoi elles sont (ou non) appropriées. Plutôt que d'opter pour l'héritage, nous avons également choisi d'utiliser des types énumérés lorsque cela nous semblait utile.

Diagramme de séquences

Le diagramme de séquence représente la logique d'une partie de bataille navale. La logique du jeu se situe au niveau de la classe Partie qui interagit avec les classes Ordinateur, Saisie et Océan pour saisir les tirs à effectuer et pour modifier la grille de jeu.

Justification des choix de conception et positionnement par rapport à GRASP et SOLID

GRASP

Nous avons conçu le programme en tentant de respecter les principes de design étudiés. Bien que certains aient été mis en application, d'autres auraient dû être implémentés. Dans cette section, nous survolerons ainsi chacun des patrons GRASP en rehaussant leur utilisation dans le code.

Patron #1: Spécialiste de l'information

Dans ce programme, la classe Ocean est la classe spécialiste de la grille de jeu. Elle connaît la représentation des cellules (classe Cellule) et elle est composée par une matrice de celles-ci. C'est cette classe qui peut le mieux tirer l'information sur la disposition des différents navires et bombes sur l'océan. Puisque la classe Océan est spécialiste de ces informations, les données des cases du jeu peuvent ainsi restées encapsulées même si la logique du jeu se passe dans la classe Partie. On évite aussi d'avoir un gros BLOB en gardant l'ensemble de la représentation de la grille de jeu encapsulée dans la même classe. Il en va de même avec la classe Ordinateur qui s'occupe de calculer les coordonnées du prochain tir et de suivre un patron de stratégie selon le niveau de difficulté implémenté lors d'une partie.

Patron #2: Créateur

La responsabilité de créer les objets des classes Océan et Ordinateur est portée par la classe Partie. Notamment, le besoin d'utiliser les instances de la classe Océan est situé principalement dans la la classe Partie qui veut interagir avec la grille de jeu même avec une connaissance limitée de son implémentation. Ensuite, la responsabilité de créer différents navires est déléguée à la classe Flotte qui contient un ensemble de navires de tailles fixes. Cependant, c'est la classe Océan qui instancie une Flotte, car dans notre programme, une flotte appartient à un océan. C'est aussi la classe Océan qui s'occupe de placer les différents navires d'une flotte sur sa propre matrice. Cette matrice composée d'un tableau double de cellules est aussi initialisée par la classe Océan lors de son instanciation. Finalement, c'est la

classe Partie qui gère la création des objets de type Océan, puisqu'une partie de bataille navale est composée de deux grilles de jeu.

Patron #3: Faible couplage

Ce patron a été mis en application en vue de minimiser le couplage entre les classes. Par exemple, le programme a été conçu de façon à ce que les classes qui sont composées d'instances d'autres classes ne soient pas affectées par les changements posés sur celles-ci. En ce sens, le programme est plutôt modulaire. Notamment, un ajout d'un différent type de cellule (classe TypeCellule) ne briserait pas la conception du programme. Il en serait de même pour l'ajout, la modification ou le retrait d'un type de navire qui n'affecteraient pas la classe Océan qui est composée d'une flotte (ensemble de navires de la classe Navire).

L'utilisation d'une classe Partie pour la logique d'affaires du programme fait aussi en sorte que le niveau de dépendances entre les classes est minimisé, car la plupart des manipulations qui sont faites dans le jeu sont basées sur des coordonnées (données par la classe Coordonnée) et non pas directement sur les objets.

On répond ainsi à la question: est-ce que cet objet a vraiment besoin de connaître celui-ci? Dans le code, la réponse est souvent non. La classe Ordinateur ne connaît pas vraiment la grille de jeu,ne fait que déterminer les coordonnées du prochain tir à effectuer. La classe Océan ne connaît pas vraiment la représentation d'un objet de type Navire, elle ne prend que sa taille variable .

Finalement, on fait aussi souvent appel à des enumérations (TypeCellule, TypeDifficulte, TypeNavire) pour représenter le type des objets qui pourraient éventuellement être changés sans modifier le comportement du programme.

Patron #4: Contrôleur

Pour contrôler le flux des messages sans coupler le modèle objet à l'extérieur, nous avons majoritairement utilisé la classe Partie. Elle est inhérente à la logique du jeu et coordonne les messages entre trois classes auxquelles elle est fortement liée: la classe Ordinateur qui détermine les prochains coups de l'ordinateur, la classe Saisie qui interagit avec l'utilisateur et la classe Océan qui est responsable des mouvements sur la grille de jeu. Elle relie l'information, mais délègue les responsabilités aux classes qui sont plus spécialistes du traitement de cette information.

Patron #5: Forte cohésion

Il n'était pas évident de concevoir le programme sans tout concentrer et en maintenant un faible couplage. Par exemple, la classe Océan porte beaucoup de responsabilités. Elle initialise la grille de jeu dont elle est composée et s'occupe de modifier les cellules (eau, navire, bombe, etc) par lesquelles elle est modélisée. Pour réduire le travail fait par cette classe afin qu'elle ne se concentre qu'à placer les objets sur son océan, nous avons entre autres utilisé une classe OceanAffichage qui s'occupe seulement d'afficher la grille de jeu. Par contre, la fonction placerNaviresAleatoirement() dans la classe Ocean aurait dû être déléguée à une autre classe qui s'occuperait de calculer les coordonnées des navires de l'ordinateur à placer. On aurait aussi pu utiliser une classe qui ne s'occupe seulement que de vérifier si un

tir ou si le positionnement d'un navire sur la grille de jeu est un emplacement valide au lieu d'attribuer également cette responsabilité à la classe Océan.

Cependant, le programme nous semble être conçu de façon à ce que chaque classe puisse être décrite en une seule phrase.

- Partie s'occupe de la logique de jeu.
- Océan gère la grille de jeu.
- Saisie gère les entrées de l'utilisateur au clavier.
- Ordinateur calcule les coordonnées des prochains tirs à effectuer.

Le faible couplage dans notre programme permet aussi une meilleure maintenance et évolutivité, dans le sens où si l'on voudrait ajouter de nouvelles fonctionnalités comme poser différents types d'objets tels que des navires ou des bombes sur la grille de jeu, il serait possible de le faire sans briser la logique d'affaires..

Patron #6: Polymorphisme

On avait initialement pensé utiliser le polymorphisme pour représenter les différentes instances des différents types de navires qui auraient pu être implémentés. Cependant, puisque les instances de navire ne sont pas, à proprement parler, de grand intérêt pour la logique du programme, nous avons choisi d'utiliser un type énuméré qui contient la taille de chacun des navires. On peut donc ainsi associer ce type à chaque instance de navire plutôt que de créer une multitude de classes inutiles et redondantes.

Cependant, il aurait pu être utile d'utiliser le polymorphisme pour la classe Ordinateur puisque l'intelligence de cette classe dépend du niveau de difficulté choisi. On aurait ainsi pu avoir une classe parent Ordinateur et des enfants OrdinateurFacile, OrdinateurIntermediaire et OrdinateurDifficile ayant des comportements différents. Cela aurait pu permettre d'éviter certains blocs de conditions et ainsi laisser à l'objet décider de son comportement.

Patron #7: Fabrication pure

La classe MultiOutputStream est basée sur ce patron. Elle permet d'enregistrer les différents types de document en dupliquant la sortie standard du programme. Elle ne fait pas vraiment partie de la logique d'affaires du programme ni du concept-clé qui est de jouer une partie de bataille navale. Cela permet ainsi de réduire la dépendance du programme à sa façon de gérer ses sorties en délégant cela à une classe plus de type fonctionnelle. La classe et les fonctions qui utilisent les services qui y sont offerts sont documentés pour faciliter la lisibilité du code.

Patron #8: Indirection

Le principe d'indirection n'est pas fortement respecté dans le programme. D'abord, nous n'avons pas utilisé de classe pour représenter un joueur. Cela ne nous semblait pas absolument nécessaire. Dans cette conception, la classe Partie demande à la classe Saisie de saisir le prochain tir ou la position du navire à placer sur la grille de jeu. Ensuite, la classe Partie transmet cette information à la classe Océan qui modifie la grille de jeu selon l'information transmise. Par ailleurs, nous aurions pu instancier plus

directement des objets de type Joueur et utiliser des interfaces pour implémenter les différentes fonctions qu'un joueur ou l'ordinateur peuvent avoir lors d'une partie.

Patron #9: Protégé des variations

Les objets du système semblent être bien protégés des modifications. Comme mentionné précédemment, nous avons utilisé des énumérations pour encapsuler certains types d'objets qui pourraient éventuellement être ouverts à la modification. Cependant, nous n'avons pas utilisé d'interface pour cela, car ces objets n'offrent pas de services.

SOLID

Responsabilité Unique:

Nous avons découpé notre code pour nous assurer que chaque objet porte une seule responsabilité. Par exemple, la classe Ordinateur est responsable de proposer les coordonnées pour l'ordinateur, Saisie traite les entrées du joueur et Ocean modifie les cellules de la grille de jeu en plaçant les navires ou en effectuant les tirs. Par contre, nous supposons que la classe Ocean s'occupe de trop de fonctionnalités. C'est l'océan qui est responsable de modifier les cellules et de placer les bateaux, mais nous croyons qu'il n'aurait pas dû être lui-même responsable de déterminer où placer les bateaux à travers la fonction placerNaviresAleatoirement(). Cette méthode pourrait être transférée à la classe Ordinateur ou être implémentée à l'aide d'une interface. De plus, c'est aussi la classe Océan qui vérifie si un bateau peut être placé sur sa grille de jeu. Par souci d'avoir une classe Dieu, nous aurions pu concevoir une classe pour faire ces types de vérification.

Principe Ouvert/Fermé:

Nous avons bâti notre application de sorte à ce qu'elle soit ouverte à l'extension et fermée à la modification. Notamment, certains paramètres du jeu sont figés avec des constantes en *final static*. Par exemple, si on voulait modifier la taille de la grille, il n'y aurait qu'à changer les valeurs des constantes de taille dans la classe Ocean ou changer les valeurs possibles de l'énumération Difficulté pour implémenter d'autres niveaux. On relève certaines applications de ce principe dans le code, notamment avec la classe Ordinateur. Celle-ci agit différemment selon la difficulté choisie, mais l'ajout d'une nouvelle difficulté ne brise pas la logique du programme, car c'est seulement la stratégie qu'elle emploie pour sélectionner les coordonnées du prochain tir à effectuer qui varie. D'une autre part, la stratégie peut varier aussi. Puisque le principe s'appuie majoritairement sur les notions de polymorphisme, on retient qu'on aurait pu utiliser des interfaces pour réaliser les différentes stratégies que l'ordinateur emploie.

Substitution de Liskov:

Puisque le programme est conçu sans sous-classe en favorisant l'utilisation de types énumérés, il n'y a aucun scénario où une sous-classe altère des propriétés souhaitables d'une classe parent.

Ségrégation des interfaces:

Bien qu'il aurait été souhaitable, nous n'avons pas utilisé d'interfaces dans le programme. Cependant, on remarque que si on avait opté pour du polymorphisme afin de gérer les alternatives structurelles entre les difficultés de l'ordinateur, les interfaces auraient permis l'implémentation de fonctions différentes. Comme l'ordinateur auquel l'ordinateur ferait appel seulement lorsque nécessaire. On peut dire que le principe est respecté puisque le programme n'utilise pas d'interfaces, mais certaines classes (Ordinateur, Océan, Partie) dépendent de fonctions qui ne sont pas nécessairement utilisées et qui devraient être implémentées autrement.

Injection des Dépendances:

Puisqu'on n'utilise pas de polymorphisme (on en faisait d'abord utilisation pour la représentation des navires), la seule implémentation du principe est au niveau des déclarations de nouvelles ArrayLists en List.

Choix des patrons

Stratégie:

Le patron Stratégie est respecté dans le programme en favorisant la composition à l'opposition de l'héritage. Ce patron est principalement utilisé dans la classe Ordinateur en variant selon le niveau de difficulté choisi.

Singleton:

Dans le cadre de notre projet, il n'est pas nécessaire d'assurer qu'une classe ait une seule instance. De multiples instances de nos classes sont utilisées à travers une exécution du programme. Par exemple, il y a deux instances de la classe Océan, soit l'océan du joueur et l'océan de l'ordinateur.

Prototype:

Nous aurions pu utiliser le patron de conception Prototype lors de création des grilles des jeux (classe Océan). On pourrait utiliser l'océan du joueur comme exemple d'instanciation pour l'océan de l'ordinateur, et même instancier chaque cellule en copiant la première, mais ce n'est pas vraiment nécessaire, car la création des océans n'est pas coûteuse et la copie peut s'avérer compliquée.

Template méthode:

On n'a pas utilisé template méthode, par contre, nous aurions pu l'utiliser puisque chaque difficulté utilise des étapes différentes et similaires bien définies afin de proposer les cases qu'il voudrait attaquer. Ainsi, on aurait pu permettre par héritage de redéfinir certaines étapes de chacunes de ses difficultés. En fait, l'ordinateur aurait pu être une classe abstraite et on aurait pu créer les sous-classes facile, intermédiaire et difficile. En faisant de la sorte, nous aurions pu modifier leur algorithme selon la difficulté tout en gardant une structure générale.

Observer:

Nous n'avons pas utilisé ce patron, mais il aurait pu être utile pour envoyer des informations à l'ordinateur. Présentement, c'est la classe Partie qui informe à l'ordinateur quel type de cellule a été atteint à travers la méthode publique recevoirResultat(cellule.Type) de l'ordinateur. En implémentant ce patron, nous aurions pu créer une méthode qui aurait comme responsabilité d'aviser l'ordinateur lorsqu'il y a un changement dans la grille.

State:

Nous utilisons ce patron dans la classe Ordinateur à travers des enums. Par exemple, quand l'ordinateur, au niveau difficile, touche un navire pour la première fois, il va passer en mode chercherAlignement. Ainsi, il va tenter de repérer si le bateau est placé horizontalement ou verticalement en attaquant toutes ses cellules voisines. Une fois que la deuxième cellule du bateau est attaquée, l'ordinateur connaît l'alignement du navire qu'il tente de couler et passe à l'état suivreAlignement afin de tenter de couler le bateau au complet. À l'extérieur, la classe Partie ne fait que demander la méthode prochaineAttaque() à l'objet ordinateur afin de recevoir la prochaine coordonnée que l'ordinateur voudrait attaquer. Mais à l'intérieur de l'objet, il va suivre un différent algorithme dépendamment de son état Strategie.

KISS:

Somme toute, nous avons choisi de *keep it simple, stupid*. La grosseur du programme a développé n'étant pas si grande, nous avons pour un design simple et facile à comprendre.

Identification des limites

On peut rapidement relever que la logique de jeu instaurée dans la classe Partie aurait pu être plus découpée. La fonction jouerPartie() s'occupe d'exécuter tout le déroulement d'une partie et on y trouve plusieurs boucles et structures de conditions qui font en sorte que la procédure du programme est figée. Par ailleurs, il serait facile de faire évoluer le programme, par exemple, en ajoutant un nouveau type de cellule (occupé par un nouveau type d'objet, comme par exemple un kraken qui se déplacerait à chaque tour et ferait automatiquement couler un bateau lorsqu'il en rencontre un). Il en va de même pour l'ajout de nouveaux types de navires à une flotte. On pourrait également facilement agrandir la grille de jeu, mais

difficilement donner de nouveaux types d'actions aux joueurs. Bref, c'est surtout la classe Partie et l'implémentation des fonctions de jeu qui est figée.

Gestion de projet

Nous avions d'abord envisagé de faire la conception en équipe, mais puisque nous avions certains différents d'horaire, Jérémie a conçu et développé le programme seul. Il a fait plusieurs itérations de modèles de classe pour trouver une implémentation qui lui semblait modulaire. Cependant, beaucoup de modifications ont dû être apportées pour tenter de minimiser le couplage entre les objets. Ensuite, Fabrice s'est occupé de créer la classe Ordinateur afin de gérer l'intelligence de l'ordinateur dépendamment du niveau de difficulté sélectionné. Finalement, Alexandre s'est occupé de toute l'infrastructure de tests et des activités de refactoring avec Jérémie. Il aurait certainement été préférable de coordonner l'ensemble des activités avec la majorité des membres de l'équipe, mais nous avons commencé à réaliser le programme légèrement trop tard pour ce que nous aurions dû faire.