Projet

Jeu sérieux : Placement de navires sur les quais d'un port de commerce

Rapport de projet

Groupe 2 → Gary Hébert
Jérémy Courel
Mounir El Mtakham
Alaaeddine Kheribeche

Clients → M. Eric Sanlaville
M. Stefan Balev

Sommaire

Introduction

Ce projet se déroule dans le cadre du Master MATIS à l'université du Havre, il a pour but de nous faire découvrir la gestion d'un projet de son commencement jusqu'à son aboutissement et ce en travaillant en équipe. Ce projet doit nous permettre d'appliquer tout ce qui a été vu lors de cette année concernant la gestion de projet ainsi que le travail collaboratif.

Le but de ce projet, pour nos clients (M. SANLAVILLE et M. BALEV) est la création d'un jeu sérieux sur le fonctionnement du port du Havre, plus principalement sur la gestion des arrivés au port et l'assignation de ces navires arrivés vers les quais, l'objectif du jeu sera donc de cumuler le moins de retard possible sur une partie.

Dans ce rapport nous allons vous présenter dans un premier temps le contexte de ce projet ainsi que son but, puis nous verrons comment nous avons géré ce projet, ensuite nous expliquerons l'aspect conception autour de celui-ci et nous terminerons par une présentation de l'application et de son interface.

1. Cahier des charges

1.1. Contexte

Un nombre important de navires arrivent quotidiennement sur les grands ports de commerce européens comme Le Havre, et y restent quelques heures ou quelques jours. Les quais sont de longueur fixe et sont spécialisés (conteneurs, vrac solide, voitures, passagers, hydrocarbures …). Ils peuvent donc être indisponibles (tout comme les remorqueurs, les pilotes ou les lamaneurs) ce qui induit une attente des navires en mer, et donc un coût important pour les armateurs. L'objectif pour la capitainerie qui gère les entrées dans le port, en plus de la sécurité, est de gérer au mieux l'ordre d'entrée des navires et leurs emplacements aux quais.

1.1.1. Intervenants sur le port

Voici les différents intervenants sur un port :

- Capitainerie → Assure la sécurité du port ;
- Armateurs → « Arme » le navire (mise à disposition d'un équipage, de matériel, de ravitaillement, ...) ;
- Gestionnaire des terminaux
- Pilote → Le pilote est transporté à bord des navires pour les diriger au sein du port ;
- Les remorqueurs → Bateaux relativement petits, très puissants et très manœuvrables, servant à guider, tirer, pousser les gros bateaux entrant et sortant des ports (ils sont au nombre de 9 pour le port du Havre) ;
- Les Lamaneurs → Chargé des opérations d'amarrage des navires ;
- Chargeurs → Organise le transport des marchandises .

1.1.2. Type de terminaux

Voici les différents terminaux présents sur le port du Havre (avec notamment les entreprises qui les utilises) :

- Charbon (EDF)
- Pétrolier (TOTAL)

- Roulier (Ro/Ro, import/export de voitures)
- Passager (Ferry, Paquebots)
- Conteneurs
- Éoliens (AREVA)

1.1.3. Informations sur les navires

Voici les informations concernant les différents navires qui transitent par le port du Havre :

- Longueur
- Type de navire
- Date d'arrivée
- Temps de chargement / déchargement
- Type de chargement

1.2. Besoin

Afin de mieux comprendre le processus exposé ci-dessus, on propose de construire un "jeu sérieux" dans lequel le joueur devra effectuer le meilleur choix par rapport à l'arrivée des navires et sera pénalisé suivant les retards occasionnés.
Le projet consiste donc à construire ce jeu sérieux, à partir des données qui seront fournies relativement au trafic, aux temps de séjour des navires, au nombre et à la taille des quais, etc.

1.2.1. Proposer une interface graphique élaborée

Il faudra tout d'abord proposer une interface graphique élaborée permettant au joueur d'évaluer au mieux la situation et son évolution, il faudra donc :

1. Afficher une alerte lorsqu'un navire occasionnera une pénalité supplémentaire ;
2. Afficher les quais disponibles lorsqu'un navire est sélectionné ;
3. Afficher les quais indisponibles lorsqu'un navire est sélectionné ;
4. Afficher le retard accumulé ;
5. Afficher le temps restant ;
6. Afficher les informations concernant le navire sélectionné ;
7. Afficher les informations sur le déroulement du jeu comme par exemple : l'arrivée d'un

navire, le retard d'un navire, le départ d'un navire, …

1.2.2. Donner au jeu un aspect fortement dynamique et ludique

Pour donner au jeu un aspect fortement dynamique et ludique on souhaite que celui-ci réponde à ces besoins :

1. Lorsque l'utilisateur effectue une action, une notification s'affiche aussitôt dans le cadre prévu à cet effet ;
2. Zoom possible en 2D sur les quais pour visualiser leurs état ;
3. Visualisation des quais en 3D pour que l'utilisateur ait une vue plus agréable de ceux-ci ;
4. Barre de progression sur les navires pour savoir ou il en sont dans leur chargement / déchargement ;
5. Une alerte (message en rouge dans la zone de « log ») prévient l'utilisateur qu'un ou plusieurs navire(s) vont être en retard ;
6. Visualisation des déplacements des navires sur la carte et du chemin emprunté ;
7. Sauvegarde des scores et possibilité de les consulter directement dans le jeu.

1.2.3. Gérer les parties

1. Les fichiers d'instances seront générés aléatoirement pour que l'utilisateur ne tombe pas plusieurs fois sur la même partie sauf s'il demande à la rejouer ;
2. Gestion du temps, à chaque tour de nouveaux navires arrivent, un tour équivaut à une ou plusieurs heure(s) / jour(s) ;
3. Gestion des points, calculé en grande partie en fonction du nombre de retard, il faudra donc minimiser ce nombre pour obtenir un bon score ;
4. Gestion du placement des navires sur les différents quais, éviter que des navires se chevauchent ;
5. Gestion des arrivées des navires ;
6. Gestion des priorités, par exemple le ferry qui est prioritaire sur les autres navires ;
7. Gestion de la difficulté des parties, le trafic sera plus ou moins important selon la difficulté choisie ;
8. Gestion des conflits sur la « route » empruntée par les navires.

2. Gestion du projet

Dans cette partie nous allons vous présenter les différents outils et moyens utilisés pour réaliser ce projet.

2.1. Réunions d'état d'avancement

Nous nous réunissions chaque semaine afin de montrer l'état d'avancement du projet et discuter des éventuels problèmes de diverses natures.

2.2. Liste de diffusion

Afin de garder contact entre les différents membres du groupe, nous avions mis en place une liste de diffusion, transmettant nos mails à tous les membres. Celle-ci nous permettait de simplifier cette tâche et de ne pas oublier un destinataire lors de l'envoi de mail. Cette liste de diffusion a été mise en place dès le début du projet pour travailler dans les meilleures conditions possible.

2.3. Git

Git est un gestionnaire de versions décentralisé, c'est à dire que chaque utilisateur possède un serveur git en local, à l'inverse de SVN (subversion) qui est centralisé, l'utilisateur modifie donc une version commune à tous les contributeurs du projet. C'est pourquoi nous avons choisi ce gestionnaire de version pour notre projet.

Il est composé de trois types de fichiers:

- snapshot : c'est l'état du fichier au moment de la sauvegarde par l'utilisateur ;
- local : c'est l'état des fichiers du projet après une commande « git commit » par l'utilisateur, mettant à jour la version locale du projet ;
- distant : c'est l'état des fichiers du projet commun à tous les utilisateurs. Pour la récupérer il est possible d'utiliser la commande « git pull » ou « git clone » pour copier le projet entier. La commande « git push » permet de la modifier par rapport à la version locale actuelle.

Nous avons utilisé l'IDE NetBeans, l'environnement de développement d'Oracle, notamment car celui-ci gère Git de base et permet donc de l'utiliser sans passer par une fenêtre shell.

Nous avons donc, dès le début du projet, mis en place un dépôt git sur le site GitHub, site stockant ce genre de dépôts.

2.4. Répartition des tâches

La répartition des tâches est primordiale dans un projet, les développeurs ont des spécialités et affinités et le non-respect de celles-ci peuvent mener à l'échec du projet.

Nous avions une équipe de deux développeur sur la partie IHM du jeu, la deuxième équipe devait quant à elle s'occuper de la partie persistance dans un premier temps puis rejoindre l'autre équipe sur la partie IHM.

Un souci s'est vite posé. La mise à niveau en java de deux membres du groupe n'a pu se faire dû à diverses contraintes d'emploi du temps (entretiens, week-ends dans une autre région, travail demandé dans d'autres matières, …). La partie applicative a donc au final été réalisée qu'à deux.

Un document a été créé dans le but de noter les tâches accomplies, celles à faire et celle en cours de réalisation, pour cela nous avons utilisé un tableur sur le site https://docs.google.com car celui-ci permettait d'accéder au document de n'importe où et nous pouvions l'éditer tous en même temps, consulter les changements, cela permettait de ne pas avoir une copie (plus ou moins à jour) pour chaque membre du groupe.

Présentation	Création map	1				
	Affichage informations formes		1			
	Affichage des log		1			
	Affichage des navires arrivants / arrivés		1			
	Création des éléments du menu		1			
	Corriger le souci de l'eau sur la map	1				
	Représentation des navires sur la map	1				
	Tests algo pathfinding + coloration terminal à la sélection d'un navire	1				
	Fenêtre configuration partie + démarrer		1			
	affichage des scores		1			
	rotation bateaux en déplacement	1				
	écrire certains noms sur la map	0				
	couleurs daltonisme	0				
	Supprimer du JPanel le navire assigné à un quai					
Métier	Zoom map	1				
	Déplacement sur la map	1				
	Onclick sur la map	1				
	Déplacements des navires sur la map	1				
	Génération aléatoire instance		1			
	Extraction données map	1				
	Classe JSon lecture	1				
	Classe JSon écriture	0				
	Sauvegarder instance		0			
	Charger instance venant d'un fichier		0			
	Charger instance dans le jeu		1			
	sauvegarder scores		1			
	lire fichier scores		1			
	gestion arrivées de nouveaux navires	0				
	gestion de la place sur les quais	0				
	gestion départ navire après déchargement/chargement	0				
	reset map	1				
	bug retrouver le chemin de pathfinding	0				

Illustration 1: Capture d'écran d'une partie du document d'attribution des tâches

3. Conception

Dans cette partie nous allons vous présenter le coté conception du projet grâce à divers diagrammes UML (« Unified Modeling Language ») qui expliquerons comment nous avons pensé le projet au début de celui-ci ainsi que la manière dont nous allions le concevoir. Cette partie contiendra également un Modèle Conceptuel des Données (MCD) qui est un diagramme venant de la méthode de conception Merise. Tous ces diagrammes auront pour but de décrire les différentes utilisations du jeu produit, les interactions entre les différents matériaux dont celui-ci aura besoin et la façon dont il aura été développé.

3.1. Diagramme de cas d'utilisation

Voici le diagramme de cas d'utilisation concernant notre projet, dans celui-ci sont indiqués les cas

d'utilisations implantés (en bleu) et ceux qui ne le sont pas ou pas entièrement (en bleu avec un contour rouge), voici le diagramme :

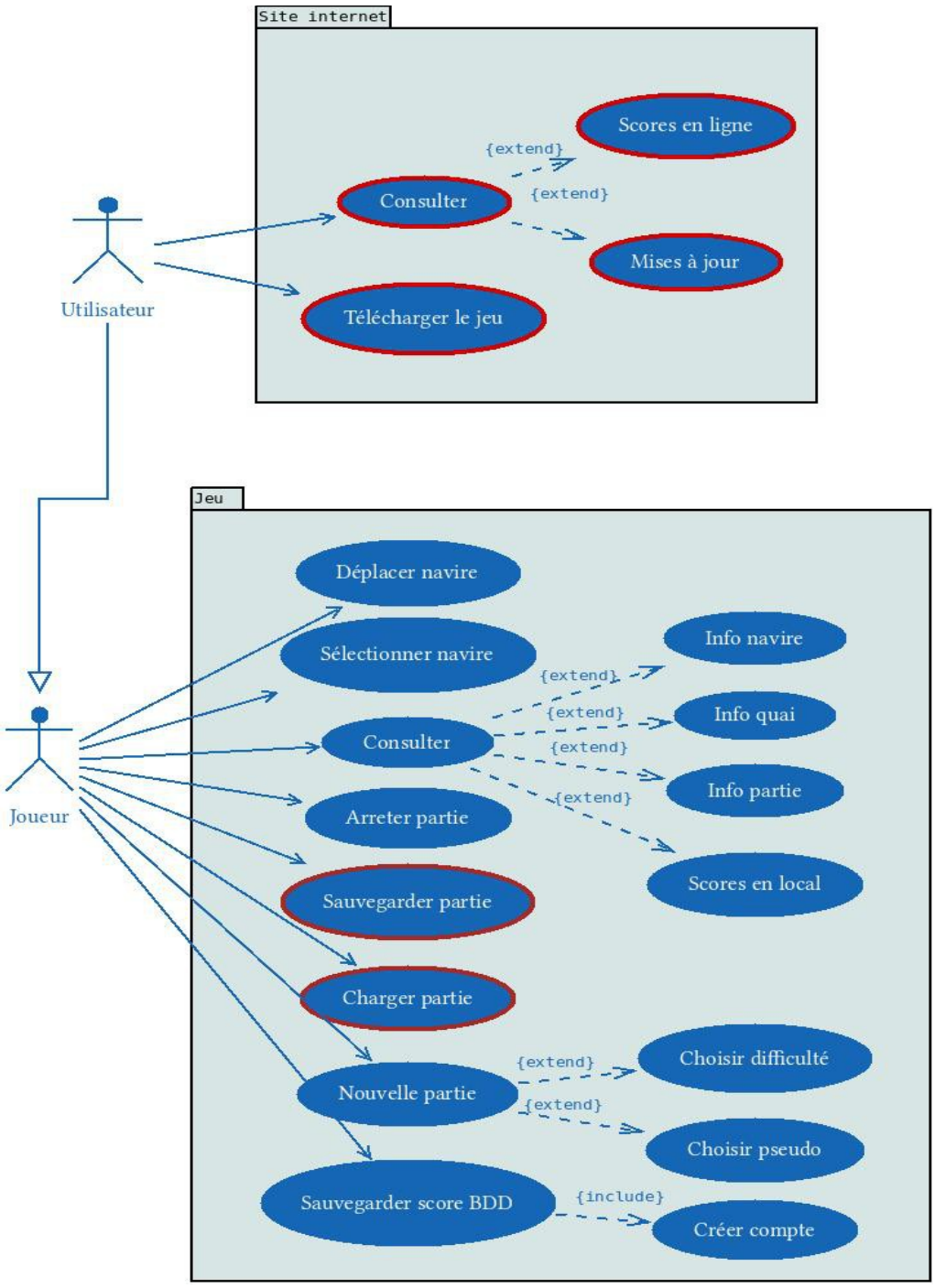


Illustration 2: Diagramme de cas d'utilisation

3.2. Diagramme de déploiement

Le diagramme de déploiement permet de voir les interactions entre les différents matériaux dont le jeu aura besoin. Dans ce diagramme, deux nœuds sont identifiés :

- PC (il contient 3 composants)
  - Base de données noSQL ;
  - Jeu sérieux ;
  - Navigateur.
- Serveur (il contient 1 composant)
  - Base de données SQL.

Ce diagramme indique qu'il y aura des interactions entre :

- La base de données noSQL et le jeu sérieux (pour le stockage en local des scores et des préférences de l'utilisateur) ;
- Le jeu sérieux et la base de données SQL (pour le stockage en ligne des scores et les comptes utilisateurs associés à ces scores) ;
- Le navigateur et la base de données SQL (pour la connexion de l'utilisateur et la consultation des scores en ligne).

Les interactions entre l'ordinateur de l'utilisateur et le serveur sur lequel sera stocké la base de données SQL se feront en TCP/IP.

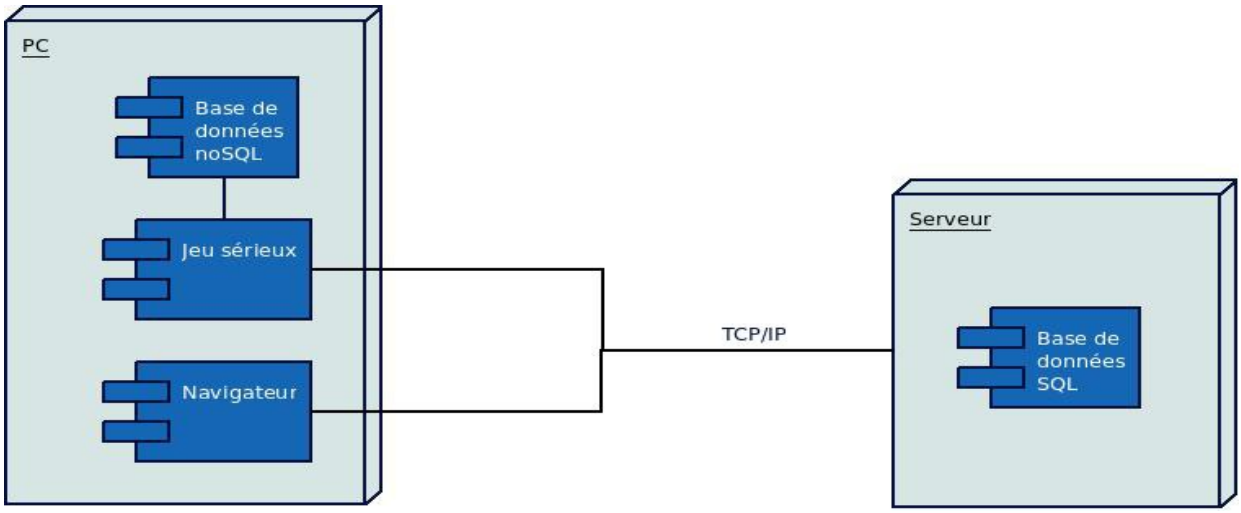


Illustration 3: Diagramme de déploiement

3.3. Architecture logicielle

Nous avons choisis une architecture séparé en quatre « package » qui sont :

- Accès au données

Ce « package » contiendra tous les accès aux fichiers (fichier de score local, fichier de paramètre du jeu) ainsi que les accès à la base de données.

- Métier

Ce « package » contiendra la gestion du jeu, ainsi que le traitement des informations récupérées dans des fichiers ou dans la base de données.

- Présentation

Ce « package » contiendra tous ce qui est interface utilisateur, tout ce que celui-ci visualisera sera créé dans ce « package ».

- Modèle

Ce « package » contiendra quant à lui tous les éléments qui nous permettront de manipuler nos données, tel que la classe Navire, la classe Quai, … Ce « package » décrira donc les données que nous manipulerons.

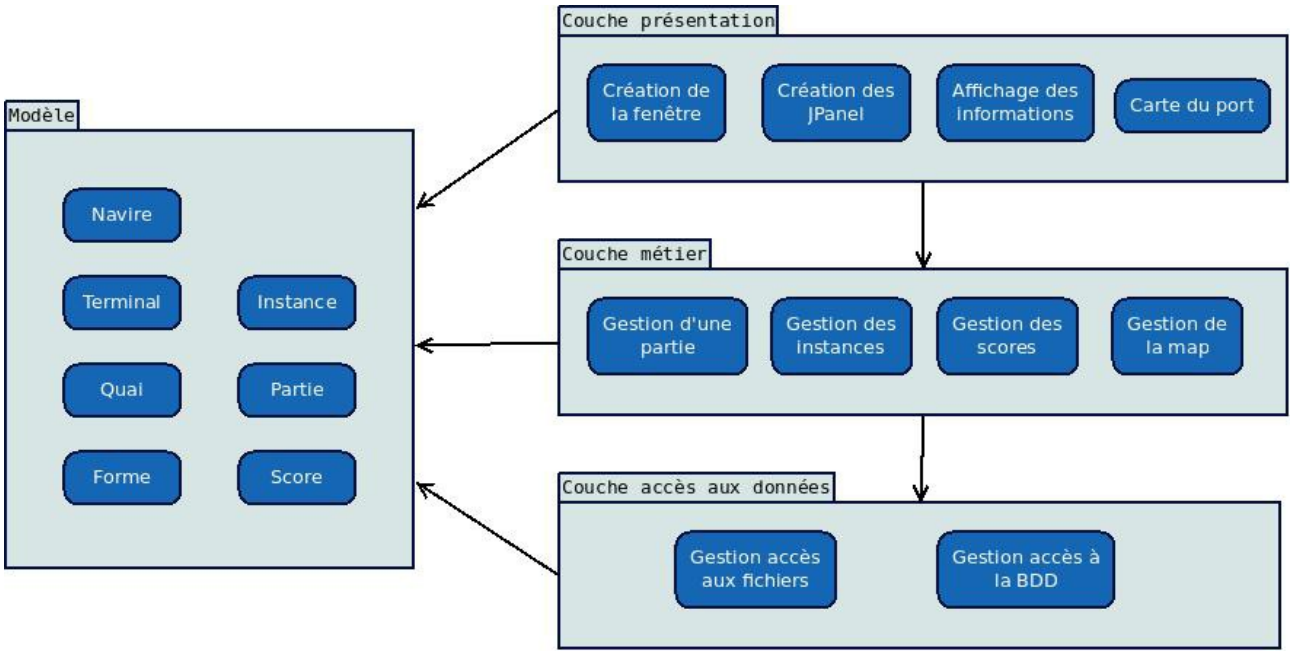


Illustration 4: Description de l'architecture logicielle utilisée

3.4. Diagramme de classes

Ce diagramme de classe contient toutes les classes créées à l'heure actuelle ainsi que les liaisons entre celle-ci, il manque juste les liaisons entre les classes des différents packages (qui n'ont pas été ajoutée dans un soucis de lisibilité), voici donc le diagramme de classes :

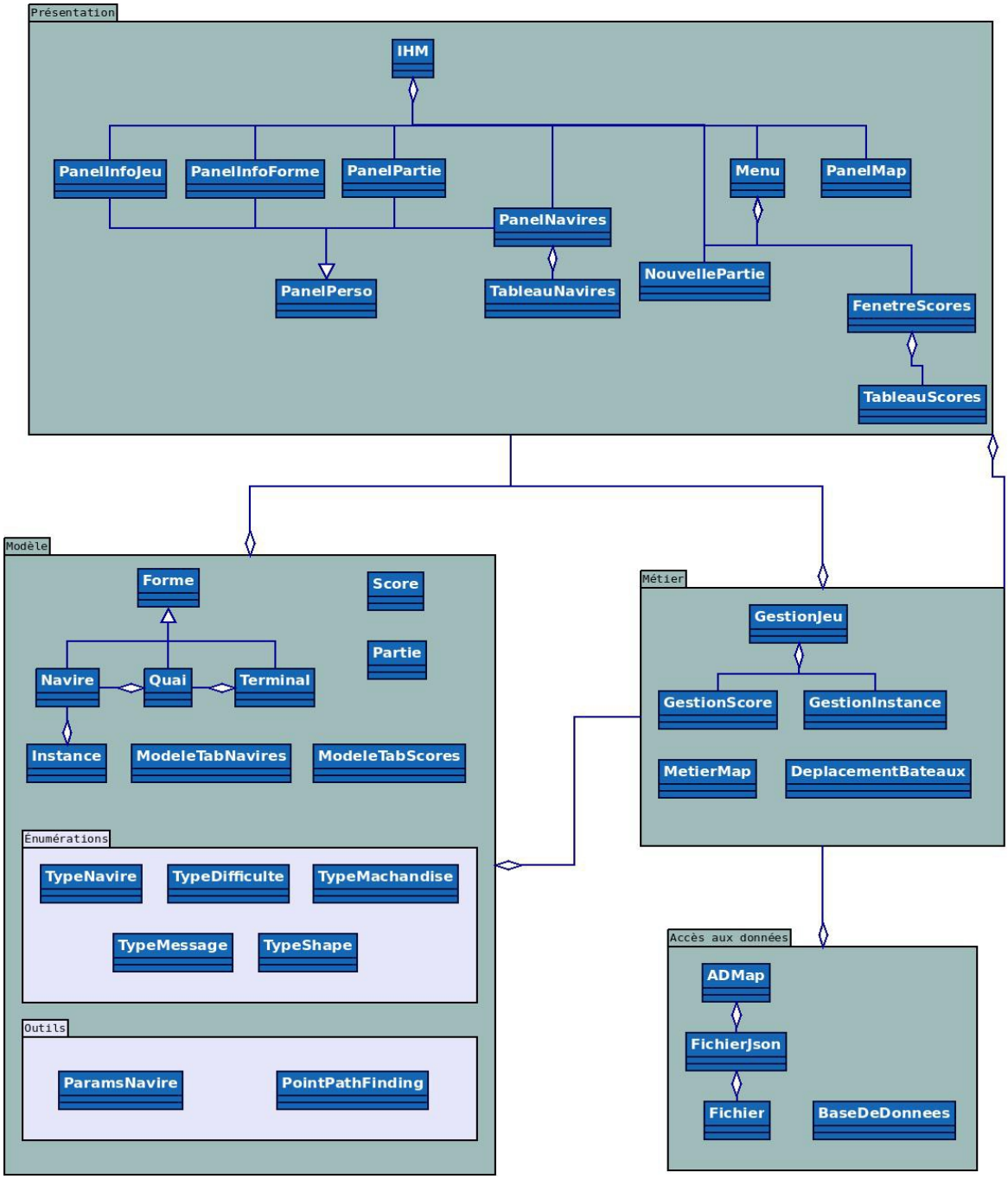


Illustration 5: Diagramme de classes

Les principales liaisons entre les packages « présentation » et « métier » permettent de mettre à jour l'interface utilisateur, comme actualiser les navires arrivés dans l'espace réservé à cet effet, actualiser le retard cumulé, … Ces actions se font de la couche métier vers la couche présentation, mais il y a aussi des actions qui s'effectue dans l'autre sens comme la gestion des actions utilisateur (commencer une nouvelle partie, consulter les scores, …).

Les classes de la couche accès aux données sont utilisées dans la couche métier, la couche accès aux données permet de bien séparer tout ce qui est dépendance aux fichiers et à la base de données donc les classes qui interagissent directement avec un fichier ou la base de données sont dans ce package et ce sont les classes du package métier qui viendront utiliser celles-ci.

Pour terminer il reste le package modèle, celui-ci représente les données que l'on sera amené à manipuler lors d'une partie, comme un navire, un quai, un score, … Ces données sont utilisées dans les couches métier et présentation.

3.5. Modèle conceptuel des données

Voici le MCD décrivant comment seront stockées nos données au sein de la base de données :

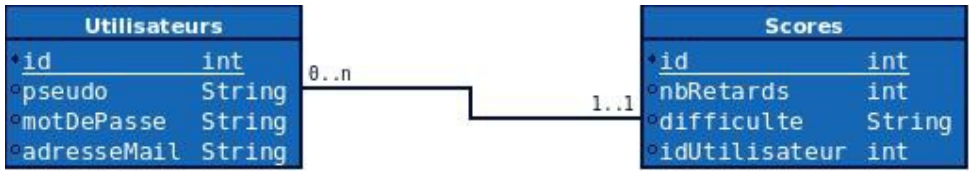


Illustration 6: Modèle conceptuel de données

Dans cette base de données on est censé avoir une table utilisateurs et une table scores, voici une description de celles-ci :

- Utilisateurs
  - Le champ « id » est la clé primaire, il ne peut être vide et est en « auto increment » ;
  - Le champ « pseudo » servira à stocker le pseudo du joueur ;
  - Le champ « motDePasse » stockera le mot de passe de l'utilisateur, celui-ci sera crypté en SHA1 ;
  - Le champ « adresseMail » stockera quant à lui l'adresse mail de l'utilisateur.
- Scores
  - Le champ « id » est la clé primaire, il ne peut être vide et est en « auto increment » ;

- Le champ « nbRetards » contiendra le nombre de retard cumulé lors de la partie, plus celui-ci sera petit, meilleur le score sera ;
- Le champ « difficulte » contiendra la chaîne de caractère décrivant la difficulté de la partie (FACILE, NORMAL et DIFFICILE) ;
- le champ « idUtilisateur » représentera l'identifiant (de la table Utilisateurs) de l'utilisateur ayant obtenu ce score.

4. Réalisation de l'interface du jeu

Dans cette partie nous allons détailler le fonctionnement du jeu ainsi que les choix effectués pour arriver à ce résultat.

4.1. OpenStreetMap

Pour représenter la carte du port du Havre, nous avons utilisé le site OpenStreetMap (http://www.openstreetmap.org) qui est un projet collaboratif ayant pour but de réaliser une cartographie du monde à partir de données libres et de données récoltées via la navigation satellite (GPS).

4.1.1. Mise à jour des données

Les données concernant le port du Havre n'étaient pas à jour sur le site, il a donc fallu les ajouter directement sur celui-ci afin d'éviter d'avoir à effectuer les corrections à la main directement dans le fichier OSM récupéré, voici donc les modifications apportées à la carte du port du Havre :

- Ajout des terminaux :
  - Roulier ;
  - de l'Océan ;
  - de France ;
  - Porte Océane ;
  - TMNSC.
- Ajout des quais :
  - de l'Atlantique ;

- des Amériques ;
- du Havre ;
- de Bougainville ;
- de L'Europe ;
- d'Osaka ;
- de l'Asie.

Ces contributions étaient actualisées directement après enregistrement, il nous suffisait donc d'exporter la nouvelle carte pour obtenir notre propre carte du port du Havre à jour.

4.1.2. Extraction des données brutes depuis OpenStreetMap

Le site openstreetmap.org permet d'extraire les données géographique d'une région en xml. Le document correspondant à notre zone de traitement pèse environ 21 mo.
Java permet, dans sa version standard, de parser un fichier XML. Lors de la lecture du fichier (la lecture se fait élément par élément), nous construisons des objets java (Node, Way) contenant leurs informations.
Ce document se présente sous la forme de 3 parties :

- Une première listant tous les points (nodes) utilisés sur la carte, ceux-ci sont identifiés par un identifiant et sont caractérisés par leurs coordonnées (latitude/longitude) ;
- Une seconde listant toutes les formes (way), celles-ci sont identifiées par un identifiant également et contiennent une liste de points (nodes). Elles peuvent aussi avoir un type de forme (building, amenity, highway...), un nom et diverses caractéristiques ;
- Enfin, une troisième est chargée de montrer les éventuelles relations entre différentes formes.

4.1.3. Bibliothèque de gestion du JSON

Java ne contient pas encore d'objets standard afin de traiter du JSON. Par contre, une bibliothèque actuellement en version beta deviendra standard dans la prochaine version de Java. Celle-ci est d'ailleurs déjà présente dans les bibliothèques disponibles de Maven.
Nous avons donc du apprendre à utiliser cette bibliothèque, ceci était relativement difficile étant donné le nombre extrêmement restreint de sources (car celle-ci est encore en beta, et les programmeurs utilisent des bibliothèques externes pour gérer le JSON).

Un fichier json est une liste d'objets composés d'un ensemble de clé-valeur. L'équivalent en programmation serait un array de hashmap.
L'ensemble de clé-valeur avec cette bibliothèque est un objet JsonObjectBuilder. On peut ajouter une clé et une valeur à cet objet simplement en utilisant sa méthode add(String clé, <T>valeur).
Ces objets représentent donc une forme (un bâtiment, par exemple) au niveau des données.
La liste des formes est un objet JsonArrayBuilder. Il suffit aussi d'utiliser sa méthode add(JsonObjectBuilder) afin de le remplir (l'ajout d'autres types est aussi possible).
Une fois toutes les formes ajoutées, il faut utiliser la méthode build() de cet objet. Cette méthode renvoie un objet JsonArray dont la méthode toString() renvoie les données au format JSON.
Il suffit donc ensuite de stocker cette chaîne dans un fichier.

4.1.4. Restructuration des données et conversion en JSON

Les données brutes en XML étant inutilement complexe dans leur représentation, nous avons décidé de la changer afin de la simplifier et ainsi améliorer la lisibilité du fichier.
Le fichier, au final, se présente comme une liste de formes. Celles-ci sont identifiées par un identifiant et sont caractérisées par un nom (optionnel), un type et une liste de « nodes » (liste de x et y, l'identifiant n'est pas utile).
Par exemple, un bâtiment simple est représenté par :

{"_id":69979243,
"_nom":"",
"_type":"NULL",
"_nodes":[{"x":0.104544, "y":49.4888273},
{"x":0.1045366,"y":49.4887379},
{"x":0.1045791,"y":49.4887365},
{"x":0.1045865,"y":49.4888258},
{"x":0.104544,"y":49.4888273}]}

Au final le fichier pèse environ 3.5 mo, c'est donc environ 6 fois plus léger que le fichier de départ. L'utilisation du JSON et la restructuration du fichier était donc totalement justifiée.

4.1.5. Correction des données

Le traitement des données brutes donnait des anomalies de rendu, des éléments étaient absents et

certaines formes devaient être supprimées.
Par exemple, les coordonnées de certains bassins ont du être retravaillées à la main afin de refléter la réalité.
Certaines formes compliquaient certains traitement (notamment le déplacement des navires) et ont été supprimées.
A l'inverse, certaines formes ont du être créée, tel que l'océan qui n'était pas présent dans les données extraites d'OpenStreetMap.

4.2. Création des formes à l'aide du fichier JSON

4.2.1. Lecture du fichier

La lecture du fichier JSON est gérée par la classe FichierJson. Celle-ci, à partir d'un nom de fichier, renvoie un JsonArray.
La lecture du fichier se fait en 3 étapes :

- lire le contenu sous forme de String, comme on le ferait pour n'importe quel autre fichier ;
- parser cette chaine de caractères via un JsonReader ;
- récupérer le JsonArray.

4.2.2. Utilisation des données récupérées

Le JsonArray ainsi récupéré contient des Object. Dans notre cas, ils ne contiennent que des JsonObject, nous devons donc faire un cast de Object vers JsonObject.
Récupérer la valeur d'une clé se fait ensuite simplement en utilisant la méthode get(« clé »).toString().
Une fois toutes les valeurs récupérées nous n'avons plus qu'à construire les objets de type Forme (ou en héritant).

4.3. Création de la carte

4.3.1. Traitement des coordonnées

Les coordonnées des formes étant en latitude et longitude, nous devons les mettre à l'échelle de la fenêtre. Les coefficients sont calculés comme ceci :

- coefficient_x = largeur fenêtre / (valeur haute coordonnée x – valeur basse coordonnée x)
- coefficient_y = hauteur fenêtre / (valeur haute coordonnée y – valeur basse coordonnée y)

Les coordonnées seront donc à multiplier par ces coefficients.

Avant cela, il faudra par contre annuler les coordonnées, c'est à dire que si les coordonnées en x sont comprises entre 0.09 et 0.2050875, nous les soustrayons au minimum afin d'avoir un minimum de 0 et un maximum de 0.1150875, ceci afin de faciliter la mise à l'échelle des coordonnées.

Enfin, les coordonnées sur l'axe y en Java partent du haut et vont vers le bas. Il faut donc inverser les coordonnées sur cet axe, en faisant hauteur de la fenêtre – coordonnée y.

Ces nouvelles coordonnées sont stockées dans un objet Path2D. Cet objet fonctionne de la même manière qu'une balise Path de SVG en HTML.

La première coordonnée utilise la méthode moveTo(x, y) et les suivantes utilisent la méthode lineTo(x, y) de l'objet. Une fois terminé, si l'on veut relier le premier et le dernier point, nous pouvons utiliser la méthode closePath().

4.3.2. Traçage des formes

Cet objet Path2D peut ensuite être dessiné dans la fenêtre via la méthode fill ou draw de l'objet Graphics2D en paramètre de la méthode Paint de la fenêtre.

La méthode fill remplira la forme alors que la méthode draw ne dessinera que les contours de cette forme.

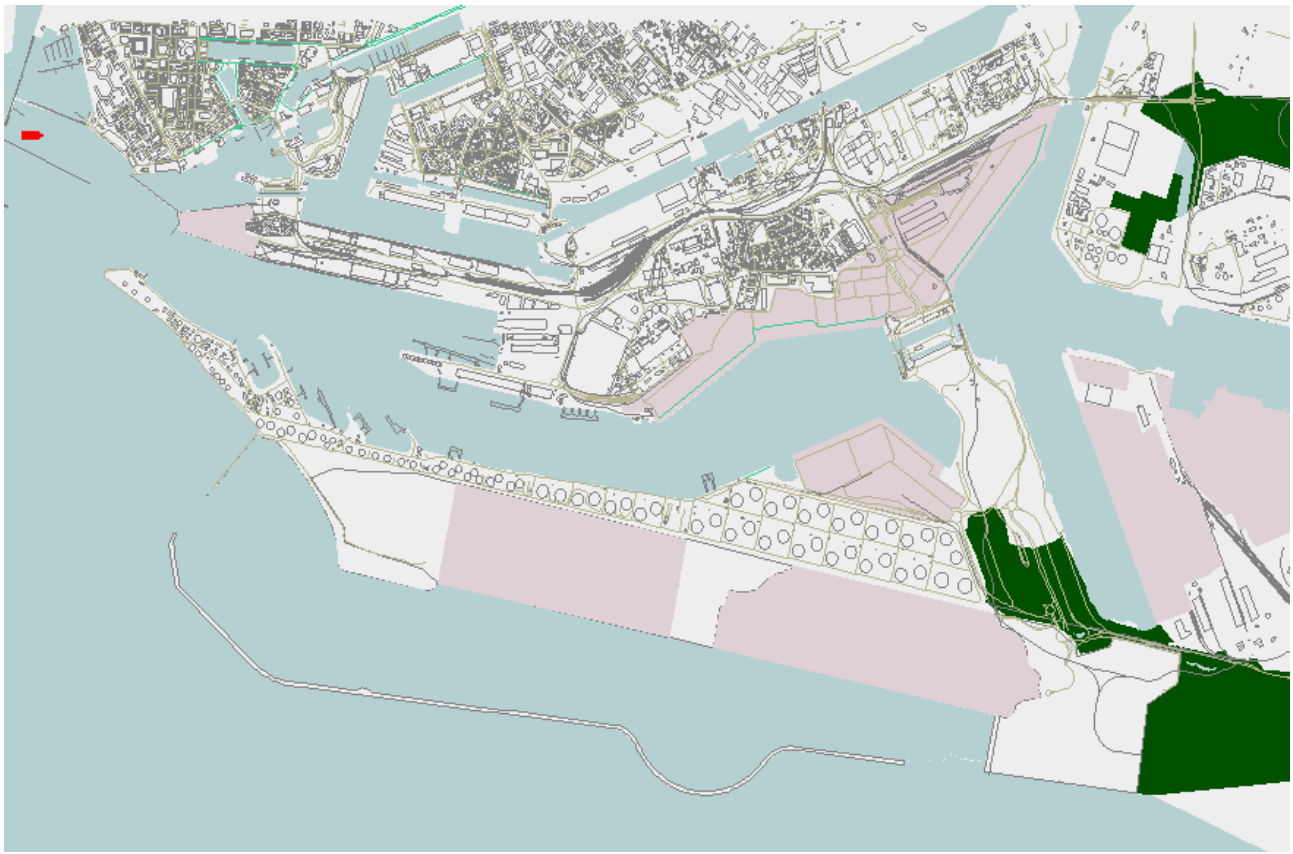


Illustration 7: Impression écran de la carte du port du Havre dessinée par notre application

4.4. Actions utilisateur sur la carte

4.4.1. Zoom

Le zoom dans l'application va de 1 (pas de zoom) à 10 et augmente et diminue par palier de 0.5.
L'intégration du zoom est relativement aisée, il suffit d'intégrer le coefficient dans le calcul des bornes limitant ce que l'on voit de ce que l'on ne voit pas dans les formes.
La principale difficulté est de zoomer sur la zone du curseur.

4.4.2. Déplacement

Le déplacement sur la carte se fait en maintenant le clic gauche appuyé.
Il faut tout d'abord déterminer dans quel sens la carte doit être déplacée. Pour cela nous calculons les valeurs suivantes :

➔ déplacement = coordonnée précédente du curseur – coordonnée actuelle du curseur

Pour chacun des axes.

Ce déplacement est à pondérer selon le niveau du zoom actuel et un coefficient arbitraire de déplacement.

Le calcul du déplacement de la carte donne donc :

➔ Point déplacement x = ancien point déplacement x + (coefficient de pondération * zoom * déplacement sur l'axe x)

➔ Point déplacement y = ancien point déplacement y + (coefficient de pondération * zoom * déplacement sur l'axe y)

Toutefois, afin de ne pas dépasser les limites de la carte, nous devons recalculer les bornes (haut, bas, gauche, droite).

Pour l'axe x, cela donne :

➔ calcul gauche = point déplacement x – largeur fenêtre * (coordonnée x curseur / largeur fenêtre)

➔ calcul droit = point déplacement x – largeur fenêtre * (1-(coordonnée x curseur / largeur fenêtre))

On ajoute donc ce calcul aux bornes gauche et droite, sauf si ceux-ci dépassent les limites des bornes gauche et droite en terme de latitude.

Dans ce cas, on calcule la différence entre le calcul et la limite puis on l'ajoute à la direction opposée, exemple pour la borne de gauche :

si calcul gauche < limite x

calcul droit += (limite x – calcul gauche)

calcul gauche = limite x

4.4.3. Clic

Le clic permet la sélection des formes dans l'application, notamment les quais, navires et terminaux.

La force des Path2D réside dans leur méthodes contains, intersects.

La première permet de savoir si cette path contient un point de coordonnées(x, y) ou une autre path, insersects permet de savoir si une path est en collision avec une autre path.

Il suffit donc, pour savoir sur quelle forme l'utilisateur a cliqué, de tester pour chaque forme :

➔ forme.getPath().contains(point)

Si ce test renvoie vrai, alors l'utilisateur a sélectionné cette forme.

Le traitement suivant dépend du type de cette forme :

- Navire : On l'enregistre comme étant sélectionné et on affiche ses informations dans le panel d'informations ;
- Quai : On affiche les informations du quai dans le panel d'informations. De plus, si un navire est déjà sélectionné, on teste la possibilité pour le quai de prendre en charge ce bateau puis on le fait se déplacer ;
- Terminal : On affiche les informations du terminal dans le panel d'informations. De plus, si un navire est déjà sélectionné, on teste la possibilité pour le quai de prendre en charge ce bateau puis on le fait se déplacer.

4.4.4. Affichage des coordonnées géographiques (latitude/longitude)

Lorsque l'utilisateur déplace le curseur sur la carte, l'application affiche les coordonnées géographiques réelles de cet endroit.

Le calcul est le suivant :

➔ axe x = point x curseur / coefficient x + borne gauche

➔ axe x = (hauteur fenêtre - point x curseur) / coefficient x + borne haute

4.5. Déplacement des navires sur la carte

4.5.1. Choix de l'algorithme de recherche de chemin

Deux algorithmes principaux existent dans la recherche de chemin dans un environnement en deux dimensions.

4.5.1.1. Dijkstra

Cet algorithme est un algorithme déterministe, il trouvera toujours la meilleure solution à condition qu'elle existe. Par contre, cet algorithme a une complexité très importante en terme de temps d'exécution, dû à son caractère polynomial.

Dijkstra calcule en fait le coût de chaque endroit depuis le point de départ, le parcours se fait en

largeur et l'algorithme parcourra tous les points de la carte, sans exception.

4.5.1.2. A*

Cet algorithme ne garantie pas que la solution trouvée soit la meilleure, mais il trouvera l'une des meilleures solutions. Le parcours du graphe se fait dans ce cas en profondeur et le premier chemin trouvé est retenu.

L'algorithme choisit à chaque itération le meilleur déplacement possible. Si il se trouve bloqué il change un choix de déplacement précédent.

Par exemple, si l'on veut aller de la position (0, 0) à (2, 2), l'algorithme détermine que le meilleur déplacement sera le déplacement diagonale bas droit, en second droit et bas, et le restant en troisième position. Si le déplacement bas droit n'est pas possible il tentera à droite ou en bas, etc.

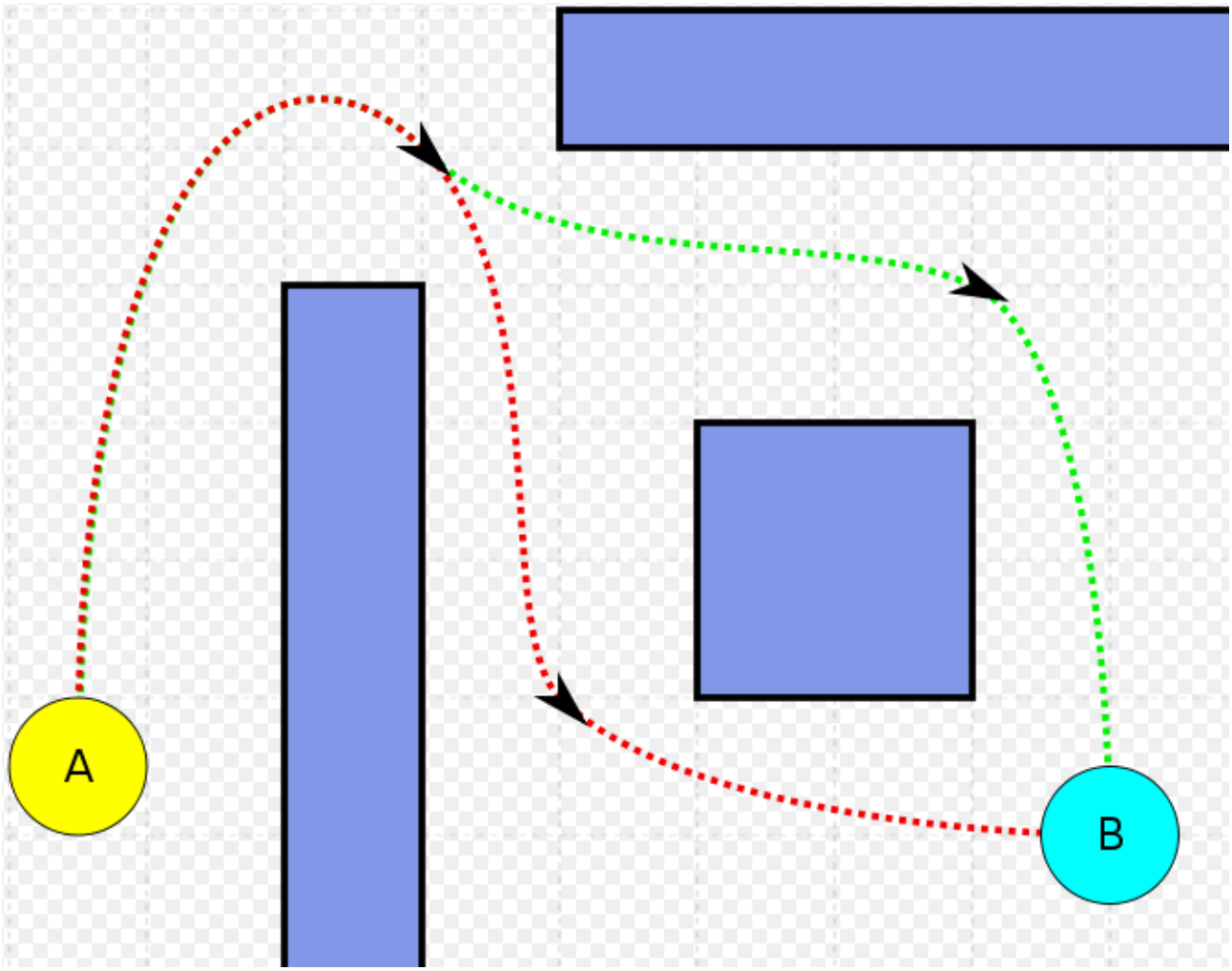


*Illustration 8: Illustration de l'algorithme A**

http://commons.wikimedia.org/wiki/File:Pathfinding_2D_Illustration.svg

4.5.2. Adaptation de l'algorithme au problème

4.5.2.1. Relation entre point géographique et sommet du graphe

Les coordonnées géographiques ne forment pas un graphe, on ne peut pas dire « je vais me déplacer de 1 vers la droite », il n'y a pas de pas dans les coordonnées géographiques. On doit donc choisir une unité arbitraire afin de se déplacer dans les latitudes/longitudes.

Cette unité doit être assez précise pour le calcul du chemin tout en ne l'étant pas trop, dans un souci de temps de calcul. Nous avons donc déterminé qu'un pas pourrait être un déplacement de 0.0008 en latitude ou longitude.

Enfin, pour en faire un graphe nous devons créer les sommets nous-même étant donné qu'ils n'existent pas.

Nous ne pouvons pas décemment créer tous les points avec un pas de 0.0008 alors que la plupart ne nous serviront pas, le temps de calcul perdu serait important.

Nous créons donc les sommets au fur et à mesure que l'on progresse dans le graphe, en vérifiant à chaque fois si ce sommet n'existe pas déjà.

4.5.2.2. Déterminer l'arrivée à destination

Ceci jusqu'à atteindre le point de destination. Atteindre le point de destination est de la même manière subjectif. On ne peut tomber sur le point précisément qu'avec beaucoup de chance. En effet, atteindre le point 0.145465465415416, 49.46548947956465 à partir du point 0.116574984654, 49.4354596765463 avec un pas de 0.0008 est sans doute impossible. Il faut donc déterminer une marge d'erreur acceptable (un zone très proche du point de destination qui sera considérée acceptable).

Cette marge d'erreur est tout juste inférieure au pas utilisé. Si un point est à moins de 0.0008 de distance d'un autre point, on considère que ces deux points sont les mêmes.

4.5.2.3. Sens de déplacement

Un navire peut se déplacer dans 8 axes différents :

- Haut ;
- Bas ;
- Gauche ;

- Droit ;
- Diagonale haut gauche ;
- Diagonale haut droit ;
- Diagonale bas droit ;
- Diagonale bas gauche.

4.5.3. Implémentation de l'algorithme

Nous avions besoin, pour implémenter l'algorithme, d'un objet contenant des coordonnées x et y de type double, contenant d'autres objets du même type, symbolisant ses voisins, de diverses autres informations et de la possibilité de comparer ces objets entre eux.
Cet objet est PointPathFinding, héritant de Point2D.Double et implémentant l'interface Comparable.
L'algorithme se déroule au final comme ceci :

1. On créé deux ArrayList, une chargée de stocker les points visités (pointVisites), l'autre les points en attente d'être traités (pile)
2. On ajoute le point de départ à la pile
3. Tant que la pile n'est pas vide :
   - ➔ On prend le premier point
   - ➔ Si ce point est le point de destination, on arrête
   - ➔ Sinon on prend les voisins de ce point et on met à jour les coûts de ces points
   - ➔ Si ces voisins ne sont pas déjà visités on les ajoute dans la pile et dans les points visités
4. Si on a trouvé la destination
5. On reconstitue le plus court chemin trouvé
6. On parcourt chaque point :
   - ➔ On change la position du bateau
   - ➔ On met à jour la carte
   - ➔ On attend 30 ms

4.6. Gestion des quais

4.6.1. Gestion de l'arrivée et du départ des bateaux

Lorsqu'un utilisateur attribue un bateau à un quai ce quai vérifie qu'il peut prendre en charge le bateau.

Celui-ci vérifie donc que :

- Le type de marchandise est pris en charge
- Une place pour ce navire est disponible

Si le quai peut le prendre en charge, le déplacement est lancé.

Lorsque la date de départ du navire est atteinte celui-ci se déplace hors de la carte et est supprimé.

4.7. IHM

Voici l'interface réalisée pour répondre au besoin de ce projet :

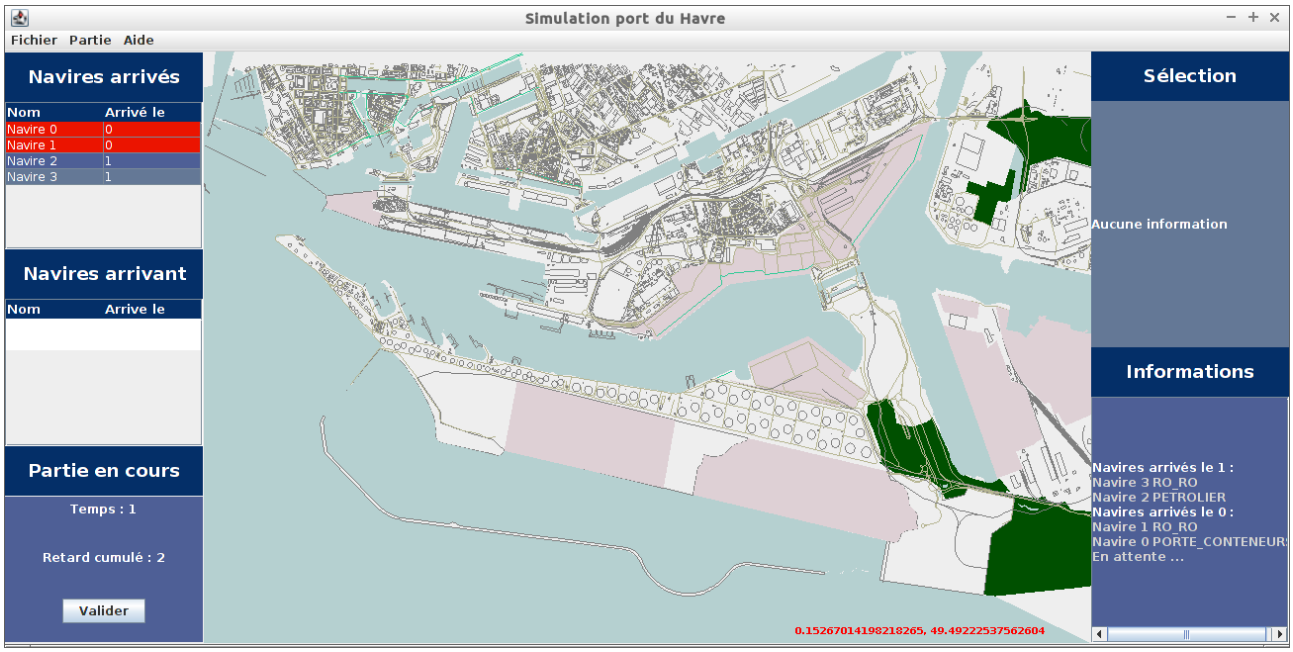


Illustration 9: Interface générale du jeu

Nous allons tout d'abord détailler les éléments qui composent celle-ci :

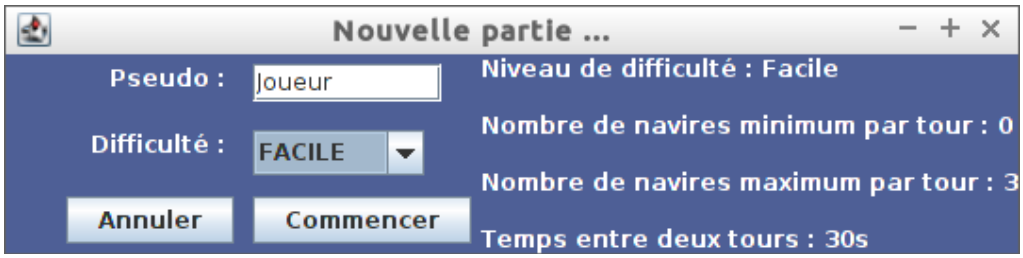
- Le menu via lequel l'utilisateur pourra sauvegarder la partie courante, charger une partie, éditer ses préférences, lancer une nouvelle partie, obtenir de l'aide, obtenir la résolution de la partie courante, afficher les scores, consulter la documentation du jeu, le tutoriel et le « A

propos » ;

- En haut à gauche nous avons un panel dans lequel sont affichés les navires déjà arrivés, si un navire est en retard, alors celui-ci sera affiché sur fond rouge (comme c'est le cas pour les deux premiers sur l'Illustration 9) ;
- Au milieu à gauche nous avons un panel dans lequel sont affichés les navires qui arriveront au tour suivant, lorsque l'on passe au tour suivant, les navires sont transférés dans le tableau du dessus ;
- En bas à gauche nous avons les informations concernant la partie en cours, le temps courant et le nombre de retard cumulé depuis le début de la partie ;
- En haut à droite nous avons un panel servant à afficher les informations sur la « forme » sélectionnée, par exemple un navire ou un terminal ;
- En bas à droite il y a un panel servant un peu de « log », en effet, celui-ci sert à afficher les informations de la partie courante, telle que l'arrivée des navires ;
- Pour terminer au centre nous avons la carte du port du Havre, celle sur laquelle le déroulement du jeu va se passer.

4.8. Nouvelle partie

La création d'une nouvelle partie se fait via le menu, dans la catégorie « Partie » ensuite « Nouvelle ... », suite au clic sur ce menu l'utilisateur voit apparaître cette fenêtre :



L'utilisateur peut alors choisir son pseudo pour la partie qu'il va lancer, le mode de difficulté (il pourra d'ailleurs visualiser ce qui change d'un mode à l'autre sur la partie droite de la fenêtre), ensuite il ne lui reste plus qu'à lancer la partie pour que la partie démarre.

4.8.1. Mode de difficulté

Pour le jeu nous avons choisi de créer trois mode de difficulté, pour le moment la seule différence entre ces trois mode est le nombre de navire minimum et maximum qui peuvent arriver au début d'un tour, chaque tour dure 30 secondes mais cela pourra être modifié après avoir effectué plus de tests sur le jeu. Voici les réglages pour les différents niveaux de difficulté (par tour) :

- Facile :
  - Nombre de navire minimum : 0 ;
  - Nombre de navire maximum : 3 ;
  - Durée : 30 secondes.
- Normal :
  - Nombre de navire minimum : 1 ;
  - Nombre de navire maximum : 5 :
  - Durée : 30 secondes.
- Difficile :
  - Nombre de navire minimum : 2 ;
  - Nombre de navire maximum : 7 :
  - Durée : 30 secondes.

Ces niveaux ne sont pas définitifs, ils devront certainement être réajusté en fonction du ressenti des joueurs.

4.8.2. Création d'une instance

Une instance représente la liste des navires qui arriveront lors d'une partie, une instance est générée aléatoirement en fonction du niveau de difficulté choisi, un nombre aléatoire est choisi entre le nombre minimum et le nombre maximum de navires (défini en fonction du niveau de difficulté de la partie courante). Avec le nombre ainsi obtenu, on génère le nombre de navire qui arriveront au même tour. Pour les paramètres des navires, nous avons deux choix :

- Tout aléatoire, c'est-à-dire que le type de navire, son type de marchandise, sa longueur, … seront générés aléatoirement ;
- À partir d'un fichier Json dans lequel sont recensés une centaine de navires existants (faute

de trouver une base exploitable, ce fichier a été réalisé à la main à partir du site http://www.marinetraffic.com).

Voici un exemple de navire stocké dans ce fichier :

{"nom":"MAURETANIA", "type":"PETROLIER", "longueur":"240", "largeur":"50"}

Une instance contient donc la liste de tous les navires qui arriveront lors d'une partie, elle est entièrement générée au démarrage d'une partie.

4.8.3. Lancement du jeu

Au lancement du jeu, les informations concernant l'IHM sont réinitialisées (si une partie était en cours avant le lancement de celle-ci alors on supprime le tout pour partir sur une base « propre »), ensuite une nouvelle instance est générée puis le jeu se lance, voici donc le déroulement logique de celui-ci (pour un tour) :

- Ajout des navires arrivés dans le Panel correspondant ;
- Suppression des navires arrivés du panel des navires arrivant ;
- Ajout des navires arrivant le tour prochain dans le panel correspondant ;
- Mise à jour du nombre de retards ;
- Actualisation de l'interface ;

Cette étape est répétée autant de fois qu'il y a de tour, à moins que l'utilisateur abandonne la partie en cours.

Une fois arrivé à la fin de la partie, la fenêtre des scores s'affiche avec le score courant surligné en rouge comme on peut le visionner sur cette image :

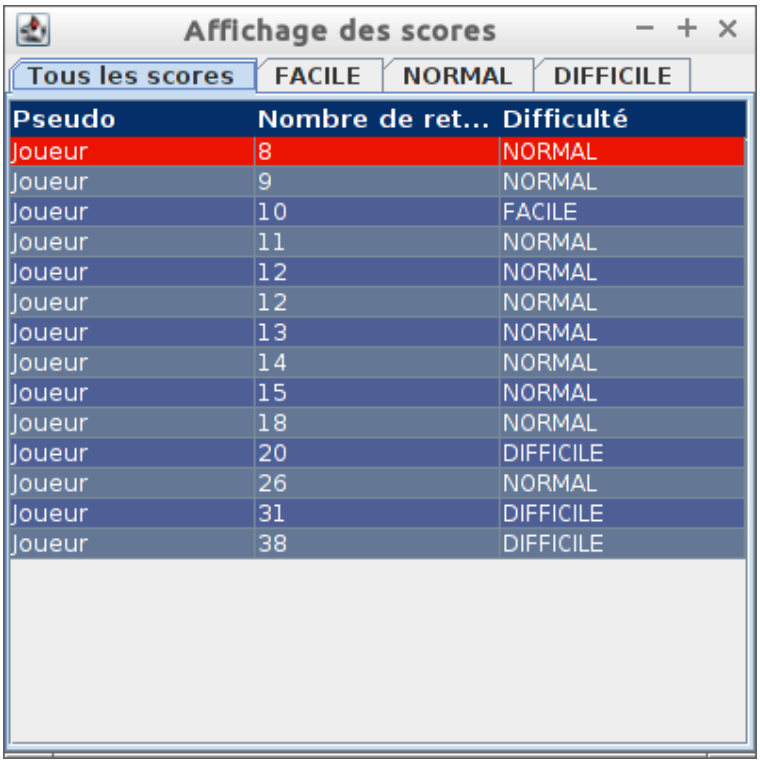


Illustration 10: Fenêtre des scores

Dans cette fenêtre l'utilisateur peut visualiser les scores globaux, ou alors juste ceux dans la difficulté de son choix, si cette fenêtre est appelé à la suite d'une partie alors le score du joueur sera surligné en rouge comme sur l'image, sinon cette fenêtre est accessible via le menu « Partie » puis « Scores » et alors l'affichage sera le même mais sans le score surligné en rouge.

4.9. Les scores

Les scores locaux sont enregistrés au format Json, dont voici le format pour un score déjà enregistré dans ce fichier :

{"Pseudo":"Joueur","Difficulté":"NORMAL","Nombre de retard":26,"Temps":1392291111990}

le champ « Temps » est en milliseconde, il est stocké uniquement pour différencier les scores entre eux dans le cas ou il y aurait deux fois la même ligne (même pseudo, même difficulté et même score). À chaque fois que l'utilisateur demande à voir les scores, une lecture de ce fichier sera effectuée pour afficher les scores dans le JTable correspondant.

Lorsqu'une partie se termine normalement, le score sera automatiquement enregistré, pour cela on génère un JsonObject qui sera inséré dans un JsonArray (obtenu par lecture du fichier des scores) pour ensuite enregistrer ce tableau dans le fichier correspondant.

4.10. Sauvegarde / Chargement d'une partie

Cette partie est en cours de réalisation, c'est pourquoi elle est incomplète.

Une fonction du menu permet de sauvegarder la partie courante, c'est à dire de prendre l'instance, l'état de la carte (Quais + Navires) et l'état de la partie, de les transformer en objet Json et ensuite de sauvegarder cela dans un fichier. Ce fichier pourra alors être lu lorsque l'utilisateur voudra charger cette partie, une fois celle-ci chargée, tous les paramètres seront remis dans l'état dans lequel ils étaient au moment de la sauvegarde. Même si une partie n'est pas censé durer longtemps, nous pensons qu'un utilisateur peut vouloir sauvegarder une partie pour pouvoir la reprendre plus tard. C'est pourquoi nous avons tenu à implémenter cette fonctionnalité, de même que celle-ci existe dans la plupart des jeux.

Conclusion

Ce projet nous a été profitable dans trois domaines : le domaine technique, la gestion au niveau humain d'un projet et la prise de conscience des difficultés que l'on peut rencontrer lors d'un travail en équipe.

Nous avons en effet, dans le but de travailler en équipe de manière optimale, utilisé sur un projet conséquent le gestionnaire de versions Git, ceci nous a fait prendre conscience de l'importance de ce type d'outil dans un projet en équipe.

Nous avons, en plus de cela, utilisé des technologies qui nous étaient encore inconnues ou méconnues telles que le traitement des données XML et JSON en java et utilisé un design pattern très intéressant : le layered architecture, séparant couches de présentation, métier, accès aux données et modèle.

Le projet, au final, est relativement massif, garder une qualité au niveau de l'organisation du code et au niveau du code en lui-même était un défis. Penser à la meilleure manière d'associer les classes, d'utiliser ou implémenter un algorithme, de traiter une fonctionnalité, nous avons aussi réfléchi au sujet de l'interface en cherchant à maximiser sa simplicité.

Le travail en groupe n'est pas toujours facile, nous ne choisissions en effet pas nos partenaires, à la manière du monde du travail, et avions un panel de compétences relativement hétéroclite, il a donc fallu gérer ces disparités au mieux pour aboutir à une version jouable du jeu. Enfin, nous avons pris conscience de l'importance du chef de projet dans le bon déroulement d'un projet.

5. Annexes

API fichier javax.json

https://java.net/projects/json-processing-spec/downloads/download/javax.json-api-1.0-fab-javadoc.zip

API en ligne

https://json-processing-spec.java.net/nonav/releases/1.0/pfd-draft/javadocs/index.html?overview-summary.html

Gestion dépendance de javax.json par Maven

http://mvnrepository.com/artifact/javax.json/javax.json-api