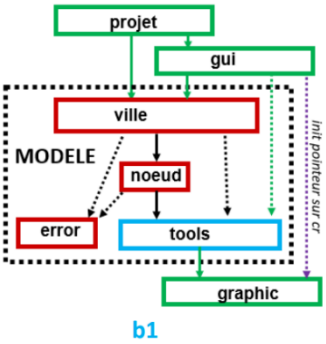
EL HAMID Ousmane : 312066

Laporte Lilian : 311043

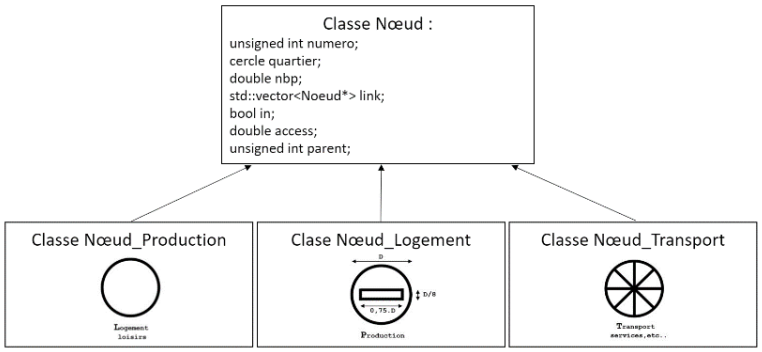
Rapport :

Dans le contexte du projet de ce semestre, il nous a été demandé de conceptualiser, implémenter et modéliser un archipel virtuel. Pour ainsi faire il était essentiel, surtout dans le cadre de programmation modulaire, d'esquisser l'ossature de notre programme pour le structurer en divers modules et leur assigner des tâches fixes. Un choix d'architecture nous était déjà proposé, il ne restait plus qu'a en choisir une. L'architecture b1 (affichée ci-contre), nous a paru être le choix plus évident et nous est venu naturellement puisque des ressources étaient déjà mises à notre disposition. Les tâches des différents modules ont été réparti comme suit :



b1

- Tools : niveau élémentaire s'occupant de la représentation du modèle (notions de cercle, point, vecteur, dessin des différents éléments élément de l'archipel) et des calculs liés au modèle (calcule de distance, d'intersections nœud/nœud, nœud/segment etc..).
- Graphic : est au cœur de la visualisation étant donné qu'il rassemble les fonctions de dessin de formes élémentaire utilisées par tools et des gestions de couleurs mais s'occupe également dans notre cas du changement de coordonnées et la gestion de distorsion. Il sert également de relais entre le modèle et l'interface graphique puisqu'il récupère le pointeur Cairo pour dessiner tout en assurant l'indépendance entre le modèle et le gui.
- Nœud : Ce module, comme son nom l'indique, permet de mettre en forme le concept de nœud, il s'agit d'un quartier auquel un nombre d'informations est associé (uid, capacité, position, lien avec d'autres quartiers). Il semblait alors tout à fait naturel de créer une classe pour structurer ces différentes données et y incorporer certaines méthodes pour assurer la détection d'erreur : en effet, le module nœud a directement accès aux informations sujettes à la présence d'erreur et à celles utilisés dans les fonctions de tools (intersections). Il s'agissait alors du meilleur candidat pour occuper cette fonctionnalité. D'autre part, 3 types de nœud/quartier nous ont été présentés : Logement/Transport/Production. Afin de réunir ces 3 types ayant chacun leurs spécificités, il semblait naturel de mettre en place une hiérarchie de classe comme (présentée ci-dessous), cette hiérarchie étant d'autant plus utile puisqu'une méthode enclenchant le dessin de chaque nœud est mise en place. Ainsi, en fonction du type du nœud, un dessin différent sera demandé auprès du module tools.



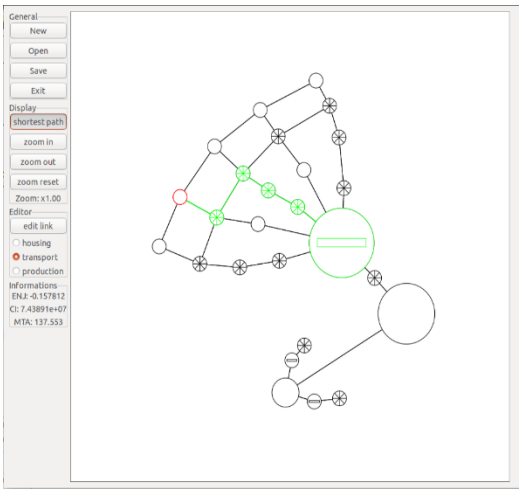
- Ville : Plus haut module du modèle, c'est là où est enfin représenté l'ensemble des Nœud et que les critères associés sont calculés. Ce module s'occupe également d'assurer deux fonctionnalités qui sont à la base de ce projet, la lecture et l'enregistrement des données (qui sont sauvegardés dans une instance statique de Ville). En effet, les données de la ville ne sont accessibles qu'au niveau de ce module, et c'est pour cela que le processus de lecture/enregistrement de fichiers y trouve parfaitement sa place. Ville assure également une fonction lançant le dessin des nœuds/liens. Toutes ces fonctionnalités et informations sont regroupés en attributs et méthodes au sein d'une classe Ville.

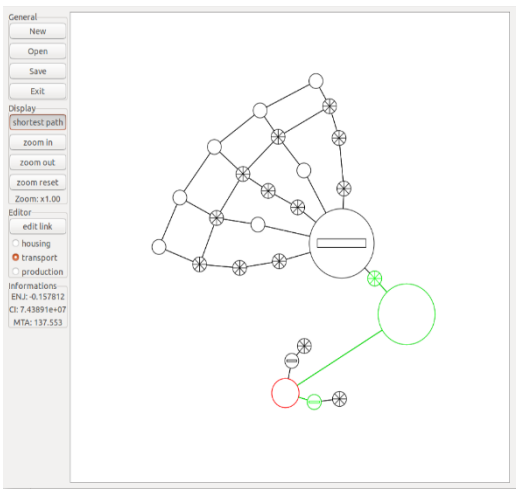
- Gui : Sans grande surprise, ce module s'occupe de l'interface graphique. Grâce à la librairie gtkmm Il met à disposition de l'utilisateur différentes fonctionnalités dont le contrôle est orchestré par une série de boutons différents. En outre, il permet aussi la mise en place d'un espace de dessin et la réalisation concrète du dessin de la ville (grâce à la fonction de dessin de Ville, appelant celle de Nœud, qui à son tour appelle des fonctions de Tools, qui a ultimement recours à graphic), mais également la création ex nihilo d'une nouvelle ville grâce à des interactions avec la souris de l'utilisateur.
- Projet : Module le plus haut du programme, il dessert une fonction main qui reçoit les (argc, argv) nécessaires à la lecture par ville et initialise la fenêtre de l'interface graphique.

Afin de représenter le graphe de la ville, la classe Ville dispose d'un vector city regroupant l'ensemble des nœuds de l'archipel, les liens entre ces nœuds sont quant à eux mis place dans la classe Nœud, également à l'aide d'un vector de pointeurs de type Nœud.

Un des multiples défis à réaliser lors de ce programme, et probablement le plus pointilleux, était la réalisation de l'algorithme de Dijkstra pour assurer la fonction "shortest_path" et le calcul MTA. Il fut élaboré comme suit :
Les valeurs ***in***, ***access*** et ***parent*** ont dû être ajoutés comme attribut à la classe Nœud et sont obtenus avec des getters et modifiés avec des setters durant l'algorithme. Le tableau **TA** est créé à chaque nouvel utilisation de l'algorithme et est trié à l'aide d'une fonction qui utilise le principe du tri par insertion vu au 1er semestre. Le tableau **TN** est le tableau qu'on utilise pour représenter tous nos nœuds dans la classe Ville,il s'agit d' un vector<Nœud*>. La fonction « find_min_access » comme indiquée dans le pseudocode trouve l'indice du nœud avec le plus faible temps d'accès et ***in*** à *true*. La fonction « compute_access » du pseudocode a été adaptée de sorte que le temps calculé était plus rapide si les 2 nœuds lié étaient des nœuds transports. Une fois l'indice retournée par l'algorithme, le temps d'accès du nœud en question est ajouté à un compteur qui est finalement normalisé par le nombre de nœuds Logement.
La partie d'initialisation pour trouver le nœud production est différente de celle pour trouver le nœud transport car dans cette dernière on initialise en plus les booléens ***in*** des nœuds Production à *false* ce qui simule le fait qu'on est déjà passé par ce nœud et donc qu'on ne peut pas le traverser. De plus, lors de l'analyse de tous les liens d'un certain nœud, nous avons ajoutés un booléen **nœud_production** qui indique si le nœud en question est de type Production. Si c'est le cas il ne faut pas analyser ses liens et passer l'attribut ***in*** du nœud à *false*. Lors de la recherche du nœud Production, le booléen **nœud_production** automatiquement à *false* car si on trouve un nœud production, on quitte l'algorithme avant la vérification du booléen. Ce booléen représente donc un système de sécurité pour être sûr qu'on ne puisse pas passer par un nœud Production.

Analyse de shortest path :





Le nœud Transport d'à côté est directement atteint. Le nœud Production est atteint en passant par les nœuds Transport. Même si cela représente un détour, le fait de passer par les nœuds Transport est plus rapide que de passer par un nœud Logement.	Le nœud Production d'à côté est directement atteint. Cependant le nœud Transport juste après n'est pas atteignable car il faut passer par le un nœud Production avant. Il faut donc aller jusqu'au nœud transport plus loin car c'est le seul atteignable.

Enfin, comme tout travail en groupe, il était important de se répartir les différentes tâches en fonction de l'attrait que ressentait chacun d'entre nous envers un module ou un autre afin d'optimiser notre travail. De ce fait, nous avons d'abord commencé par diviser le travail en deux parties : pendant que Ousmane s'occupait de la représentation du modèle et sa visualisation (modules tools, nœud, gui, graphic), pendant que Lilian s'occupait de la partie lecture/ouverture/sauvegarde de fichier, gestion d'erreur et calculs de critères. Tout naturellement, nous avons commencé par l'implémentation du module de plus bas niveau, tools (étant donné qu'il pouvait être testé indépendamment des autres modules en comparant ses fonctions d'intersection avec des dessins réalisés sur d'autres logiciels) parallèlement avec la lecture de fichier (qui elle aussi n'a pas besoin d'autre modules pour subsister), puis nous avons réalisé les autres modules un a un en fonction de leur hauteur au niveau des dépendances avant de réaliser la partie graphique. En prenant du recul, une de plus grosse erreur durant la réalisation de ce projet était d'attendre de réussir chaque tâche à faire et de compléter chaque fonction avant de les organiser et de les insérer dans les modules adéquats, ce qui constitue une perte de temps consiédarable.

Au cours de cette expérience, plusieurs bugs ont été détectés, mais celui qui a le plus retenu notre attention est sans doute celui lié à l'algorithme de Dijkstra, il n'affichait jamais la bonne valeur. Même après plusieurs debug, de nouveau bug apparaissait. Nous nous attendions à un tel problème pour cette partie car c'est la partie du code qui est la plus algorithmique et la plus complexe. Les erreurs provenaient d'un mauvais tri du tableau TA, d'une mauvaise initialisation de l'algorithme et du fait que nous n'arrivions pas à trouver un système fiable pour interdire le passage à travers un nœud Production. Nous avons donc dû chercher ces erreurs et comprendre l'algorithme à 100% pour les résoudre sans provoquer l'émergence de nouveaux bugs. Ainsi, de nombreux essais ont été réalisés pour corriger toutes ces erreurs.