Introduction à UML

L'acronyme UML vient de « *Unified Modeling Langage* » qui veut dire langage de modélisation unifié. Ce langage a été créé au début des années 1980 afin de normaliser les nombreux langages de modélisation existant antérieurement.

Il consiste à créer des graphiques à base de pictogrammes illustrant différents éléments conceptuels d'un système. Initialement développée par des chercheurs du domaine du génie logiciel et destiné aux éléments de conception orientée objet, cette méthode de conception est parfois utilisée par plusieurs branches du génie (électrique, mécanique, industriel, etc.).

UML est utilisé pour spécifier, visualiser, modifier et construire les documents nécessaires au bon développement d'un logiciel orienté objet. Il offre ainsi un standard de modélisation, pour représenter une architecture logicielle.

La dernière révision d'UML (datant de septembre 2013) propose 14 types de diagrammes permettant d'élaborer la conception d'un projet. Ces diagrammes sont dépendants hiérarchiquement et se complètent mutuellement de façon à permettre la modélisation d'un projet tout au long de son cycle de vie (que ce soit de sa conception, sa réalisation, son implémentation, son déploiement et même sa maintenance).

Diagrammes principaux

Il est important de comprendre que l'utilisation du langage UML n'est pas une méthode. Ainsi, son utilisation est laissé à l'appréciation de chaque concepteur. Il n'est pas fautif de l'utiliser partiellement et à sa convenance.

Il est donc fréquent de retrouver quelques aspects spécifiques du langage lors de la modélisation d'un projet. D'ailleurs, il est courant de retrouver en industrie seulement les diagrammes les plus utilisés et uniquement pour certains aspects d'un projet.

On considère généralement le diagramme de classe comme étant l'élément central d'UML. Néanmoins, certaines approches mettent l'emphase sur d'autres diagrammes comme les cas d'utilisation.

Malgré tout, les diagrammes suivants sont les plus fréquents :

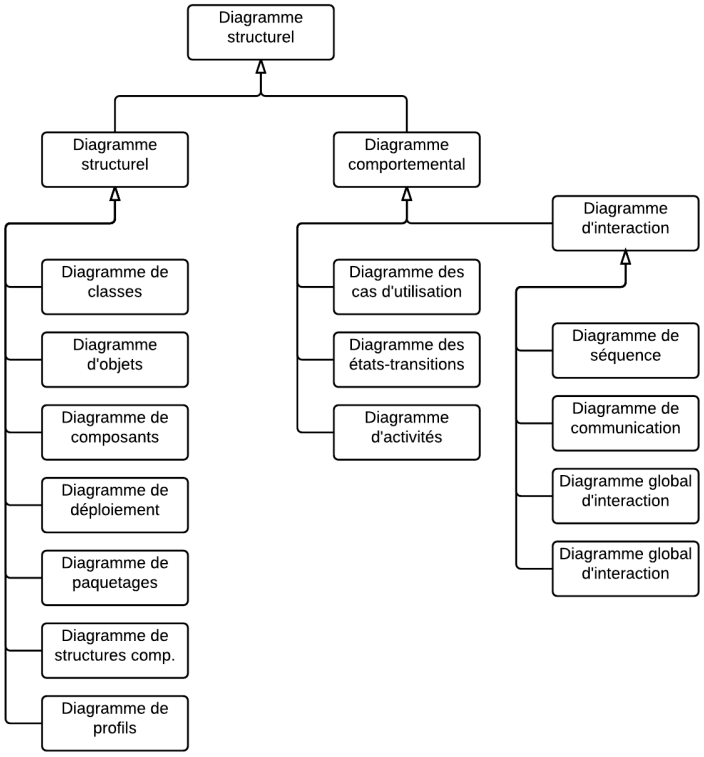
- diagramme de cas d'utilisation
- diagramme de classe
- diagramme des états-transitions
- diagramme de séquence

Ce document fait une courte présentation de ces 4 diagrammes. Ces présentations sont incomplètes et mettent l'emphase sur les notions fondamentales et plus généralement utilisées.

Les 14 diagrammes

- Les 6 diagrammes structurels (statiques)
    - Diagramme de classes
    - Diagramme d'objets
    - Diagramme de composants
    - Diagramme de déploiement
    - Diagramme des paquetages
    - Diagramme de structure composite
    - Diagramme de profils
- Les 3 diagrammes comportementaux
    - Diagramme des cas d'utilisation
    - Diagramme état transition
    - Diagramme d'activité
- Les 4 diagrammes d'interaction (dynamiques)
    - Diagramme de séquence
    - Diagramme de communication
    - Diagramme global d'interaction
    - Diagramme de temps

Le shéma suivant montre les liens existant entre les diagrammes.



Mise en situation

Les présentations qui suivent présentent quelques exemples tirées de la mise en situation suivante : conception d'un jeu de type "*Tower Defense*" possédant les caractéristiques principales suivantes :

- jeu en 2 dimensions avec vue du dessus;
- surface de jeu à chemin ouvert (pas de chemin prédéterminé pour les ennemis);
- 3 types de tours différentes :
    - tir de missile à tête chercheuse;
    - tir de laser;
    - surface circulaire ralentissant les ennemis;
- 3 types d'ennemis :
    - rapides mais faibles;
    - relativement lent mais se déplaçant en groupe;
    - très lent mais très résistant;
- le joueur peut, pendant la partie, réaliser ces tâches principales :
    - interaction de la partie :
        - démarrer une nouvelle partie;
        - arrêt de la partie;
        - pause et continue;

        - déterminer la vitesse de jeu selon trois modes (1x, 2x et 3x);
    - interaction avec les tours :
        - ajouter des tours;
        - cliquer sur une tour et voir ces statistiques;
        - améliorer une tour ayant le focus;
        - vendre la tour ayant le focus (remboursement complet si la tour n'a jamais été utilisée).

Diagramme des cas d'utilisation

Le diagramme des cas d'utilisation ("*use case diagram*") est, dans sa forme la plus simple, une représentation simple et de haut niveau des interactions entre les usagers et le système.

Les utilisateurs se nomment "*acteurs*" et sont représenté par le pictogramme très simple d'un humain. On retrouve 4 familles principales d'acteur :

- acteurs principaux (usagers typiques du système comme un client, un étudiant, …);
- acteurs secondaires (opérateurs du système tel que des employés d'administration, de support et de maintenance sur le système, …);
- couche matériel (périphériques externes d'entrée telles que imprimante, capteurs, …);
- autres systèmes (serveur, …).

Constituants

On retrouve trois éléments principaux dans un diagramme de cas d'utilisation :

- les acteurs représentés par le pictogramme d'un humain (typiquement, le bonhomme allumette);
- les cas d'utilisation reprsentés par une ellipse dans laquelle on retrouve un titre décrivant le cas d'utilisation;
- les relations unissants les éléments entre eux selon ces 4 possibilités :
    - une *relation simple* fait le lien entre un acteur et un cas d'utilisation (représentée par un trait simple);
    - une *inclusion* est lorsqu'un cas en contient un autre (représentée par un trait pointillé muni du mot clé *inclure* ou "*include*")
    - une extension est lorsqu'un cas en contient un autre conditionnellement (représentée par un trait pointillé muni du mot clé *étendre* ou "*extend*");
    - une généralisation (ou une spécialisation) fait référence à un cas ayant une spécificité par rapport au cas général (représenté par un trait plein terminé par une flèche blanche orienté vers le cas général)

Exemple

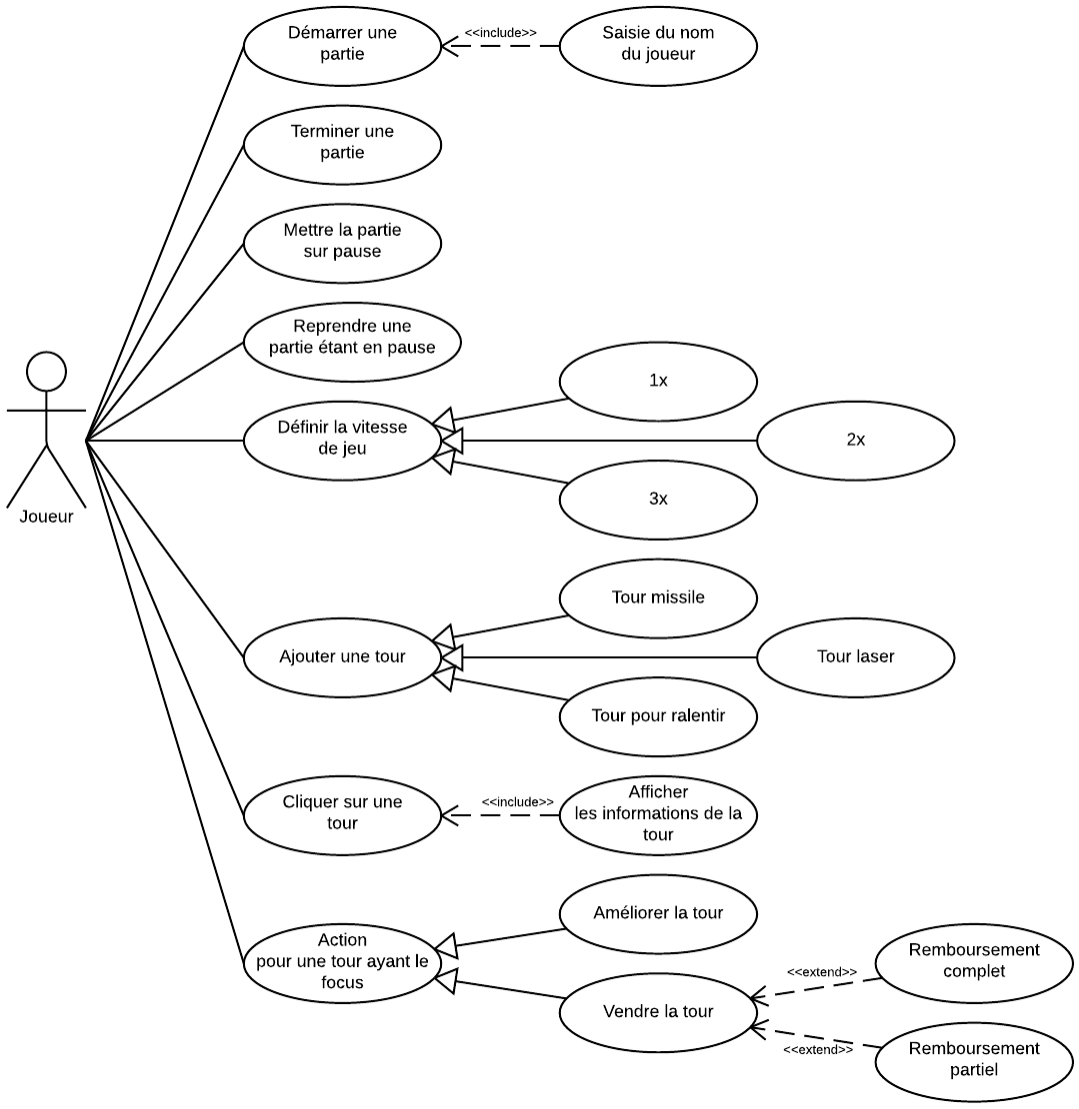


Diagramme de classes

Le diagramme de classes est un schéma utilisé pour présenter les classes et les interfaces des systèmes ainsi que les différentes relations entre celles-ci. Une classe, décrit les responsabilités, les attributs et les comportement d'un ensemble d'objets de même type. De cette façon, elles permettent de modéliser un programme pour ainsi découper une tâche complexe en plusieurs sous-tâches plus simples.

Plusieurs autres types de relations sont possibles entre les classes et chacune est représentée par un pictogramme spécifique. Pour cette présentation, nous n'aborderons que les relations suivantes : héritage, association, agrégation et composition.

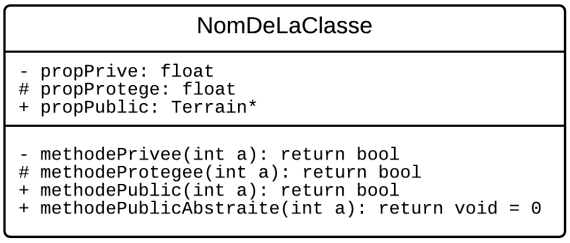
La classe

La classe est l'élément de base et est représentée par un pictogramme rectangulaire divisé en trois parties distinctes :

1. le nom de la classe;
2. la liste de tous les attributs;
3. la liste de toutes les méthodes.

Au moment de la conception, il est courant et pertinent d'ajouter :

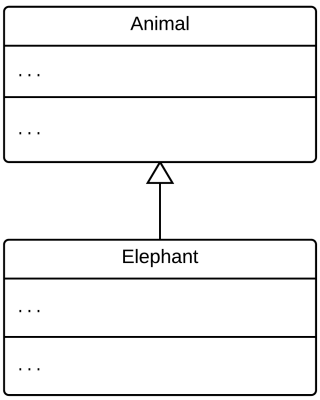
- le type de chaque attribut et la valeur par défaut s'il y en a;
- la signature détaillée de chaque méthode;
- le mode d'accès pour chaque attribut et méthode :
    - `-` : privé
    - `#` : protégé
    - `+` : public.



L'héritage

C'est le principe de division par généralisation et spécialisation. Ce type de relation implique toujours une classe parent et une classe enfant pour laquelle, la classe enfant hérite des attributs et méthodes (publics et protégés) de la classe parent.

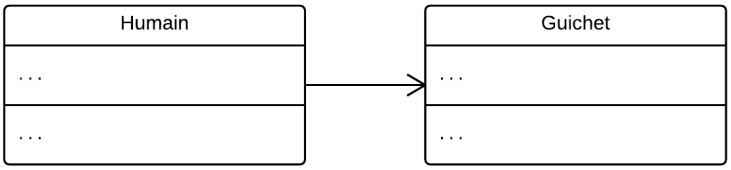
Une flèche blanche pointant vers le parent est le pictogramme utilisé pour représenter l'héritage. De cette façon, le pictogramme se lit ainsi, la classe enfant *X* hérite de la classe parent *Y*.



L’association

Ce type de relation implique l’usage d’une classe par une autre. Par exemple, une première classe peut modifier le contenu d’une deuxième classe simplement en passant cette dernière comme paramètre dans une de ses méthodes. Un autre cas, une méthode de la première classe retourne une nouvelle instance d’une deuxième classe.

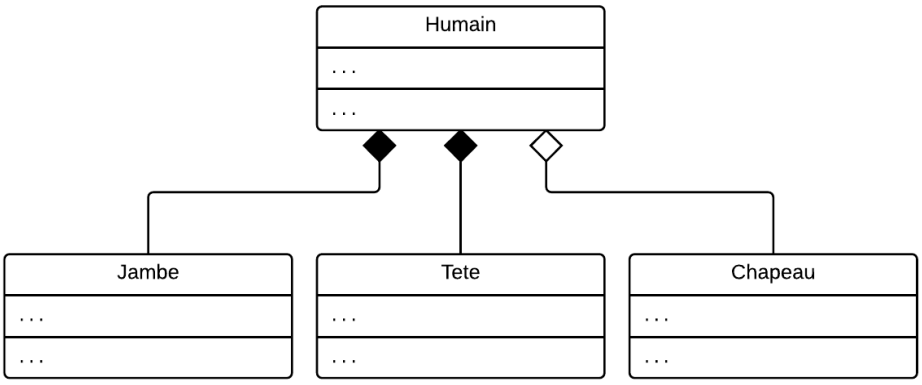
Dans tous ces cas, il y a association entre ces deux classes. Mais ni l’une ni l’autre ne va obligatoirement utiliser ou nécessiter l’autre pour exister. Le pictogramme utilisé est un trait simple.



L’agrégation et la composition

C’est une relation de subordination entre deux classes. On dit qu’une première classe regroupe d’autres classes ou que l’instance de la première classe utilise une instance d’une deuxième classe. On peut spécialiser la notion d’agrégation lorsque le cycle de vie de la deuxième classe dépend de la première classe. On parle alors de composition.

Le pictogramme utilisé est une ligne terminée par un losange blanc (pour l’agrégation) ou noir (pour la composition).



Exemple

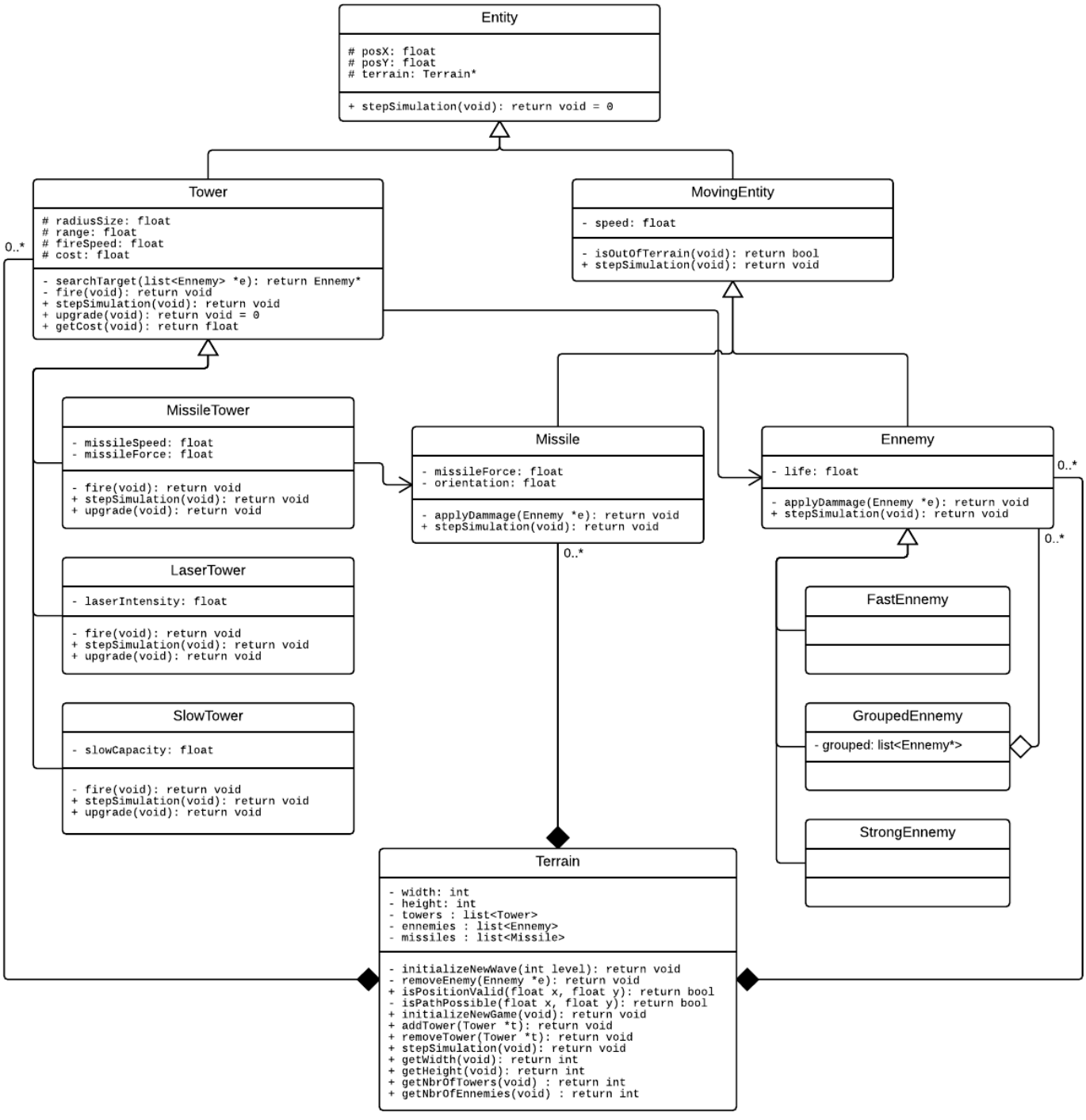

Entity
posX: float
posY: float
terrain: Terrain*
+ stepSimulation(void): return void = 0
Tower
radiusSize: float
range: float
fireSpeed: float
cost: float
- searchTarget(list<Ennemy> *e): return Ennemy*
- fire(void): return void
+ stepSimulation(void): return void
+ upgrade(void): return void = 0
+ getCost(void): return float
0..*
MovingEntity
- speed: float
- isOutOfTerrain(void): return bool
+ stepSimulation(void): return void
MissileTower
- missileSpeed: float
- missileForce: float
- fire(void): return void
+ stepSimulation(void): return void
+ upgrade(void): return void
LaserTower
- laserIntensity: float
- fire(void): return void
+ stepSimulation(void): return void
+ upgrade(void): return void
SlowTower
- slowCapacity: float
- fire(void): return void
+ stepSimulation(void): return void
+ upgrade(void): return void
Missile
- missileForce: float
- orientation: float
- applyDammage(Ennemy *e): return void
+ stepSimulation(void): return void
0..*
Ennemy
- life: float
- applyDammage(Ennemy *e): return void
+ stepSimulation(void): return void
0..*
0..*
FastEnnemy
GroupedEnnemy
- grouped: list<Ennemy*>
StrongEnnemy
Terrain
- width: int
- height: int
- towers : list<Tower>
- ennemies : list<Ennemy>
- missiles : list<Missile>
- initializeNewWave(int level): return void
- removeEnemy(Ennemy *e): return void
+ isPositionValid(float x, float y): return bool
- isPathPossible(float x, float y): return bool
+ initializeNewGame(void): return void
+ addTower(Tower *t): return void
+ removeTower(Tower *t): return void
+ stepSimulation(void): return void
+ getWidth(void): return int
+ getHeight(void): return int
+ getNbrOfTowers(void) : return int
+ getNbrOfEnnemies(void) : return int

Diagramme des états-transitions

à venir...

Diagramme de séquence

à venir...