M2107.PT COO/POO/GP : RING

Le sujet du projet proprement dit est donné dans la dernière partie de ce document. Le projet comporte 3 parties, toutes indispensables pour obtenir une note maximale :

- Modélisation basique en UML
- Développement en Java
- Gestion de projets

Gestion de projets

> Pour plus de détails, lisez attentivement les insctructions concernant la gestion de projet)

Planification

En cours de Gestion de Projets, vous avez appris comment maîtriser le processus de développement d'un projet. Il vous faudra employer certaines des connaissances acquises pour ce projet. En particulier, il vous sera demandé de :

- Définir les tâches à réaliser
- Evaluer leur difficulté et prévoir un coût à chacune (un nb d'heures de travail)
- Définir les dépendances entre tâches (quelles tâches doivent être terminées pour faire les autres)
- Déduire des dates de début et de fin prévues pour chacune des tâches

Ces informations seront consignées dans le tableau de suivi d'avancement et dans les fiches de tâches dont des modèles sont donnés en annexe.

Suivi de l'avancement

Le tableau de suivi d'avancement doit être tenu à jour pendant toute la durée du projet. On vous donne un modèle de document dont il faudra reprendre le contenu en fonction de votre propre projet. Il comprend :

- Le tableau d'avancement avec les tâches, leur planification ainsi que leur exécution
- Le tableau avec les tâches et la participation de chacun
- Des éléments calculés automatiquement correpondant à l'analyse de la valeur acquise

Pendant le déroulement du projet, vous consignerez les dates de début et de fin réelles de chacune des tâches, ainsi que leur coût réel, dans le premier onglet du tableau de suivi.

Le second onglet doit être rempli pour que les enseignants puissent juger de l'implication de chacun dans le projet.

Les derniers onglets ne doivent pas être modifiés : ils calculent automatiquement des indicateurs d'avancement selon la méthode d'anlayse de la valeur acquise, en fonction de ce qui est saisi dans le premier onglet. Ainsi, vous pourrez suivre la progression de votre projet et estimer à quel point vous êtes en avance ou en retard, du point de vue des délais ou de celui des coûts (temps de travail). Après le début du projet, il est possible de réviser la planification des tâches non encore effectuées.

Rendu

En plus de votre propre copie du tableau d'avancement et participation, on vous demande de remplir une documentation comportant :

- un texte d'une page ou deux pour analyser le déroulement du projet et sa réalisation en regard de ce qui avait été prévu
- les modèles au format bitmap
- des fiches détaillant les cas d'utilisation les plus complexes

En outre, on joindra une archive zip du code ainsi que du projet Modelio.

Modélisation avec UML

Diagrammes à produire

Avant de vous lancer dans le développement proprement dit, vous devez concevoir le logiciel en employant des techniques de modélisation. En particulier, il vous est demandé de :

- Etablir un diagramme de cas d'utilisation complet
- Etablir une première version du diagramme de classes (modèle du domaine) en n'intégrant que les classes relatives aux personnages, aux équimements et à la description du monde, et en ignorant la gestion des menus et de l'interaction avec les utilisateurs
- Etablir un diagramme de classes complet, intégrant tous les éléments du logiciel

Il est naturel de faire évoluer les diagrammes au fil du développement. Si au moment de l'implémentation en Java, vous vous rendez compte que votre conception est inadaptée, modifiez-là. Le temps de conception associé sera alors attribué aux tâches de développement correspondantes, et pas aux tâches correpondant aux premiers efforts de modélisation, en principe terminées très tôt dans le projet.

Modèle du domaine et classes de dialogue

Le modèle du domaine est le coeur du diagramme de classes. Il correspond aux classes dites *"métier"* et qui devront être présentes quelle que soit la forme que prendra le logiciel (application mobile, web, PC etc) et donc indépendamment de l'interface utilisateurs. Typiquement, dans une application de vente en ligne, on aura toujours des produits, des commandes etc, quelle que soit l'interface.

Toutefois, pour produire un système opérationnel, il est nécessaire d'ajouter des classes de dialogue qui seront chargées de gérer les entrées-sorties (dialogue avec les utilisateurs, enregistrements dans des fichiers). Ces classes interagissent avec les classes du modèle du domaine (classes métier) et permetent de séparer l'interface utilisateur en développant des classes de dialogue spécifiques. Dans le cas d'une application mobile on aura des boutons etc, dans une application web on aura des pages etc, et dans une application en mode texte dans un terminal on aura des menus etc.

La seconde version du diagramme de classes (domaine + classes de dialogue)

présente un diagramme de classes complet.

Rendu

Les diagrammes doivent être produits avec Modelio et rendus dans une archive *zip* ou *tar.gz*. Un document PDF rassemblant les différents diagrammes produits doit également être fourni. Les documentations textuelles doivent être rendues au format PDF.

Implantation en Java

Le code correspondant à la conception en UML doit être produit en Java. On n'attend qu'une interface en mode texte, avec un programme s'exécutant depuis un terminal. Des démonstrations devront être produites à chaque réunion d'avancement, et une démonstration globale devra être produite à la fin, durant la soutenance. La démonstration se fera à partir de l'archive qui aura été transmise aux enseignants pour le rendu final.

Sujet du projet

> Le sujet de ce projet reprend une production de Laurent Audibert.

L'objectif du projet consiste à proposer un modèle UML d'une petite application permettant de mettre en oeuvre des combats entres deux personnages, puis de proposer une implémentation en Java de cette application.

Description des combattants

Les combattants peuvent être de trois classes : guerrier, mage ou chasseur. Tous les combattants du jeu sont caractérisés par un *nom*, une *force* physique `FOR`, une *dextérité* `DEX`, une *intelligence* `INT`, une capacité de *concentration* `CON`, une *vitalité* `VIT` et une *expérience* `EXP`.

La vitalité d'un personnage est son nombre de points de vie. Ce nombre décroit au fur et à mesure qu'un personnage est blessé au combat. Le personnage meurt quand sa vitalité devient nulle. L'expérience d'un combattant augmente quand il gagne un combat et décroit quand il perd, avec une valeur plancher et une valeur plafond.

Les sections suivantes décrivent les aptitudes des trois types de combattants qui doivent être mises en oeuvre dans le jeu. Chaque personnage possède un certain nombre de capacités (armes, sortilèges…).

Caractéristiques communes

Dans tous les cas, les équilibres suivants doivent être respectés :

- `(FOR + DEX + INT + CON) <= (100 + EXP)`
- `1 <= EXP <= 20`

Au début d'un combat, la vitalité est initialisée de la manière suivante :

- `VIT = 200 - (FOR + DEX + INT + CON) + EXP x 3`

Guerriers

Un guerrier est un combattant qui brille plus par sa force physique que par son intelligence. La contrainte suivante est imposée à tous les guerriers :

- `FOR >= (DEX + 10) >= (INT + 10) >= CON`

Mages

Bien entendu, la force d'un magicien réside dans sa capacité à faire de la magie, au détriment de sa force physique. Les contraintes suivantes sont imposées à tous les magiciens :

- `INT >= max(FOR, DEX) + 15`
- `CON >= max(FOR, DEX) + 15`

Chasseurs

Un chasseur est un combattant assez complet capable aussi bien de manier l'épée que la magie, mais à chaque fois sans être aussi spécialiste qu'un mage ou un guerrier. Les contraintes suivantes sont imposées à tous les chasseurs :

- `FOR >= 20`
- `DEX >= 20`
- `INT >= 20`
- `CON >= 20`

Description des capacités

Les capacités permettent aux combattants de porter des coups, de parer des coups et de se soigner. Elles se subdivisent donc en trois catégories : *attaque*, *défense*, *soin*. Le nombre de capacités d'un combattant doit être inférieur à `EXP / 2`, mais ne peut descendre en dessous de 2.

Toutes les capacités fonctionnent selon le même principe :

1. Vérifier sa réussite : La **probabilité de réussite** `PBA` de la mise-en oeuvre d'une capacité est fonction de l'aptitude du combattant ainsi que des caractéristiques de la capacité. Une probabilité est comprise entre 0 et 1.
2. Mesurer l'efficacité : Si la mise en oeuvre d'une capacité a réussi, son **efficacité** `EFF` est fonction de l'aptitude du combattant ainsi que des caractéristiques de la capacité, sinon son efficacité est nulle bien entendu.

Dans les sous-sections suivantes, les formules de calcul pour l'efficacité `EFF` font apparaître un signe `~` qui doit être interprété comme *"est à peu près égal à, avec une part aléatoire"*. Ainsi, pour calculer une efficacité, on applique la formule de calcul mais le résultat varie en fonction d'un tirage aléatoire, entre plus et moins 25% de l'efficacité calculée.

Les capacités possèdent un certain nombre de caractéristiques. La somme de ces caractéristiques doit toujours être égale à 100 et aucune caractéristique ne peut être inférieure à 20. Le tableau suivant détaille toutes les catégories de capacités des combattants.

Arme

> Chaque type d'arme (épée, hallebarde etc) peut avoir des caractéristiques différentes. On peut également parer un sortilège avec une arme mais dans ce cas, l'efficacité de l'arme est divisée par 3.

- Catégorie : attaque, défense
- Caractéristiques : impact `IMP` et maniabilité `MAN`
- Probabilité de réussite
- pour une attaque : `PBA = DEX x MAN / 10000`
- pour une défense : `PBA = DEX x MAN / 5000`
- Efficacité

- pour une attaque : `EFF ~ FOR x IMP / 100`
- pour une parade : `EFF ~ FOR x IMP / 50`

Sort d'attaque

> Chaque type de sortilège (boule de feu, éclair etc) peut avoir des caractéristiques différentes. Un sort d'attaque ne peut pas être utilisé en défense.

- Catégorie : attaque
- Caractéristiques : puissance `PUI` et facilité `FAC`
- Probabilité de réussite : `PBA = INT x FAC / 10000`
- Efficacité : `CON x PUI / 100`

Bouclier

> Chaque type de bouclier (écu, pavoi etc) peut avoir des caractéristiques différentes. On peut également parer un sortilège avec un bouclier mais dans ce cas, l'efficacité du bouclier est divisée par 2.

- Catégorie : parade, attaque
- Caractéristiques : protection `PRO` et maniabilité `MAN`
- Probabilité de réussite
- pour une défense : `PBA = DEX x MAN / 10000`
- pour une attaque : `PBA = DEX x MAN / 5000`
- Efficacité
- pour une défense : `EFF = FOR x PRO / 100`
- pour une attaque : `EFF = FOR x PRO / 50`

Sort de défense

> Chaque type de sortilège (boule de feu, éclair etc) peut avoir des caractéristiques différentes. Un sort d'attaque ne peut pas être utilisé en défense.

- Catégorie : défense
- Caractéristiques : puissance `PUI` et facilité `FAC`
- Probabilité de réussite : `PBA = INT x FAC / 10000`
- Efficacité : `EFF = CON x PUI / 100`

Remède

> Chacun peut transporter de petits remontants dans sa besace, et les employer pour regagner de la vitalité afin de poursuivre le combat.

- Catégorie : défense
- Caractéristiques : puissance `PUI` et facilité `FAC`
- Probabilité de réussite : `PBA = DEX x FAC / 10000`
- Efficacité : `EFF = FOR x PUI / 100` (la force est aussi un indicateur de la constiyution du personnage, lui permettant de récupérer plus ou moins vite)

Sort de soin

> Les mages et les chasseurs peuvent employer la magie pour se soigner.

- Catégorie : défense
- Caractéristiques : puissance `PUI` et facilité `FAC`
- Probabilité de réussite : `PBA = INT x FAC / 10000`
- Efficacité : `EFF = CON x PUI / 100`

Création d'un nouveau combattant

Pour créer un nouveau combattant, il faut :

1. Choisir un type de personnage (guerrier, chasseur ou magicien) et son nom
2. Choisir la valeur de ses aptitudes (force, dextérité, intelligence, concentration) en respectant les contraintes détaillées plus haut
3. Lui affecter une expérience de 1
4. Créer 2 capacités en respectant les contraintes exposées plus haut, puis les lui affecter

Déroulement d'un duel

> Un duel se déroule par tours de jeu. Un tour de jeu ne concerne qu'un seul combattant.

Initialisation du duel

1. Calculer et affecter la vitalité à chacun des participants selon la formule présentée plus haut

2. Déterminer qui commence : c'est le plus expérimenté, ou le gagnant d'un tirage au sort en cas de niveau d'expérience identique.

Tour de jeu

Les joueurs jouent chacun leur tour.
A chaque tour de jeu, un joueur peut décider :

- soit d'attaquer
- soit de défendre
- soit de se soigner
- soit de capituler et dans ce cas, il perd la partie mais son personnage ne meurt pas.

Résolution d'une attaque

Si un joueur *J1* décide d'attaquer :

1. On détermine si l'attaque de *J1* est réussie en procédant à un tirage aléatoire en regard de la probabilité de réussite calculée (voir plus haut)
2. On affiche si l'attaque est réussie ou non ; l'efficacité de l'attaque `EFF_A` est calculée mais pas affichée immédiatement
3. On passe au tour de jeu du joueur *J2* qui pourra au choix :

- défendre et dans ce cas, la défense est résolue conformément à la section suivante
- choisir d'encaisser l'attaque et dans ce cas sa vitalité `VIT` est réduite de l'efficacité `EFF_A` de l'attaque du joueur *J1*. A son tour, il pourra attaquer normalement ou se soigner.

Résolution d'une défense

> Cette section est un peu complexe. Le principe de cette résolution technique est que plus un combattant attend pour décider d'une défense, moins elle a de chances d'être couronnée de succès.

Dans le cas à un joueur *J1* aurait réussi une attaque au tour précédent, le joueur *J2* peut décider de défendre. Dans ce cas, *J2* peut au choix :

- Décider de défendre immédiatement et alors

1. On détermine si la défense de *J2* est réussie en procédant à un tirage aléatoire en regard de la probabilité de réussite `PBA` calculée (voir plus haut)
    - si sa défense est couronnée de succès alors on en calcule l'efficacité `EFF_D` .
    - si la défense échoue alors on considère que `EFF_D = 0`
2. La vitalité `VIT` de *J2* est réduite de `(EFF_A - EFF_D)` (une différence négative ne pouvant pas faire regagner de vitalité)
3. On passe au tour du joueur *J1*

- Evaluer la puissance de l'attaque avant de prendre une décision. Dans ce cas :

1. On affiche la puissance de l'attaque `EFF_A` du joueur *J1* au tour précédent
2. Le joueur *J2* décide s'il veut finalement défendre ou non :
    - S'il souhaite défendre, il peut le faire comme s'il l'avait décidé immédiatement (voir plus haut) mais la probabilité de réussite `PBA` de la défense est réduite d'un quart, ainsi que son efficacité
    - S'il préfère encaisser l'attaque alors
    1. Sa vitalité `VIT` est réduite de l'efficacité `EFF_A` de l'attaque du joueur *J1*
    2. Il peut finalement procéder à une attaque à ce tour, mais sa probabilité ainsi que son efficacité seront réduites d'un quart.

Résolution d'un soin

Au début d'un tour de jeu, un joueur peut décider de se soigner. Dans ce cas :

1. On détermine si le soin est réussi en procédant à un tirage aléatoire en regard de la probabilité de réussite calculée (voir plus haut)
2. L'efficacité `EFF` du soin est calculée
3. La vitalité `VIT` du combattant est augmentée de `EFF`

Premier tour de jeu

1. Le combattant qui commence met en oeuvre une capacité de son choix (généralement une attaque)

- la réussite est déterminée par tirage aléatoire en fonction de sa probabilité calculée
- en cas de réussite, l'efficacité est évaluée puis appliquée dans le cas d'un soin, et affichée dans les autres cas

1. L'efficacité d'une attaque réussie est mémorisée pour impacter l'adversaire lors de son tour de jeu

Fin de partie

Une partie se termine quand l'un des deux adversaires décède ou capitule. Un combattant décédé l'est définitivement et ne pourra plus combattre.

Le vaincu perd un point d'expérience (sauf s'il n'en avait déjà plus qu'un) et éventuellement un point dans une aptitude tirée au sort de manière à respecter les contraintes du tableau de la figure 1. Le perdant peut également être amené à perdre une capacité de son choix pour que le nombre de ses capacités reste inférieur au nombre de ses points d'expérience divisé par deux.

Les vainqueur gagne un point d'expérience (sauf s'il n'en avait déjà vingt) et éventuellement un point dans une aptitude de son choix tout en respectant les contraintes définies plus haut. Sauf dans le cas d'une capitulation, il peut également copier une capacité de son adversaire et l'ajouter aux siennes, ou remplacer l'une des siennes s'il en possédait déjà le maximum autorisé.

Fonctionnalités de l'application

L'application doit permettre de créer de nouveaux personnages. L'application doit également permettre de gérer des duels entre deux adversaires qui sont chargés depuis un fichier. La sauvegarde des combattants est automatiquement gérée par l'application. Ces fonctionnalités correspondent au coeur du logiciel.

Il est en outre souhaitable que l'application permette de jouer à deux avec deux ordinateurs distincts (l'application étant exécutée sur les deux machines) en utilisant un fichier d'échange (dont l'emplacement est paramétrable) pour synchroniser le déroulement d'un duel.

Vous pouvez imaginer des extensions, mais concentrez-vous avant tout sur le coeur du projet.

Comment vous y prendre ?

> Ce qui suit sont des conseils. Vous pouvez refuser de les suivre mais vous pourriez alors éprouver des difficultés certaines.

Conseils généraux

D'une manière générale, il est plus prudent d'avoir toujours une version opérationnelle du programme. Ainsi, on ne développe pas *classe par classe* mais *fonctionnalité par fonctionnalité*, en démarrant par les fonctionnalités les plus importantes.

Typiquement, pour ce projet, la question de la séparation entre un serveur et un client (avec les communications afférentes) n'est pas la plus mince affaire. Nous nous conseillerions donc de d'abord développer un programme simple et *mono-utilisateur* permettant de gérer simplement les PNJ et un seul PJ dans un labyrinthe. Ce programme permettra d'implémenter et de valider tout ce qui a trait aux règles du jeu et une bonne partie des menus en charge de l'iterface homme-machine.

Parallèlement à ce développemnt, on peut réfléchir sur la meilleure manière de s'y prendre pour gérer la communication entre des clients et un serveur.

A ce point, on dispose d'un programme *mono-utilisteur* mais opérationnel et complet, ainsi que d'une analyse du fonctionnement client-serveur attendu. Si on est en retard sur le planning, on aura toujours ça à présenter et c'est heureux : mieux vaut un programme fonctionnel mais incomplet qu'un programme trop ambitieux mais non fonctionnel. En tout état de cause, le développement de la version client-serveur devient plus aisée à partir de ce point.

Donc :

- Démarrer tôt
- Développer une fonctionnalité après l'autre, en démarrant par les plus centrales : ne pas faire feu de tout bois
- Ne pas multiplier les types de monstres et objets au début : se contenter d'un petit nombre et se concentrer sur le moteur du jeu
- Ne commencer aucune extension possible tant que le coeur du jeu n'est pas achevé et pleinement fonctionnel
- Tester régulièrement et méthodiquement
- Pour travailler à deux, utiliser des systèmes de versioning (GIT ou SVN). A minima, employer des dossiers partagés (Dropbox ou autre)
- Ne pas attendre d'aévoir tout modélisé en UML pour commencer le

développement en Java : le codage doit s'appuyer sur les modèles, mais il peut également guider la conception

Exemple de progression attendue pour le développement

Etude préliminaire

- Diagramme de cas d'utilisation complet
- Diagramme de classes métier

Planification et premières implémentations

- Définir une tâche pour chaque cas d'utilisation
- Ordonner les tâches par ordre d'importance et en fonction des contraintes de dépendance (les cas concernant la version *mono-utilisateur*) seront prioritaires
- Assigner un développeur à chaque tâche
- Implémenter en Java le diagramme de classes métiers (avec Modelio, c'est juste un clic)

Développement, développement, développement

- Etablir un planning détaillé (dates de début et de fin prévues pour chaque tâche)
- Développer les cas d'utilisation jugés les plus prioritaires

Suivi de projet

- On suit le planning de développement et on consigne la progression dans les tableaux de suivi.
- Si on prend du retard, on retravaille la panification pour avoir toujours un plan réaliste.
- En cas de retard important, on peut recentrer ses ambitions, revoir le diagramme de cas d'utilisaion à la baisse, et donc supprimer les tâches correspondantes. Ces élagages doivent être faits en partant de l'expression des besoins.

Quel que soit le point auquel vous serez arrivés, il faut que le programme soit utilisable. Le programme le plus ambitieux, s'il n'est pas fonctionnel, sera jugé nul.

Erreurs à ne pas commettre

Trop s'éloigner des règles

Les règles sont adaptables mais dans une certaine mesure seulement, et sans affecter la nature du jeu. En revanche, la coloration culturelle du jeu peut changer (pokemons on ce que vous voudrez au lieu de mages/guerriers/chasseurs etc) et des détails de règles peuvent changer. Il n'est d'ailleurs pas certain que les règles de combat proposées soient parfaitement équilibrées, et un travail sur les différents seuils et paramètres peut améliorer le *game design*.

Etre trop ambitieux

Ce qui compte, c'est de produire un logiciel fonctionnel. Il est par exemple possible d'employer des librairies graphiques pour améliorer le rendu du jeu, mais c'est bien moins important que le fonctionnement effectif.

Concrètement, un projet trop ambitieux mais fonctionnant mal serait beaucoup moins bien évalué qu'un projet mois ambitieux mais opérationnel. Il est donc recommandé de d'abord focaliser ses efforts sur le moteur du jeu avec une interface simple, avant de se lancer dans des développements plus ambitieux.

En outre, si quelques points de "bonus" seront attribués à des fonctionnalités ou éléments supplémentaires, il ne faut pas s'attendre qu'en terme de "rentabilité" heure/point, les heures passées à des développements supplémentaires soient au même niveau.

Pierre Gérard (2016) M2107