Le génie pour l'industrie	**INF-111** Travail pratique #3 **Groupe :** 3 étudiants maximum (un seul rapport). **Remise** : 12 avril 2022. **Auteur :** Frédéric Simard (H2022)

1 - Introduction

1.1 - Contexte académique

Ce second devoir vous amène dans le monde développement de jeux vidéos et vise à pratiquer les notions suivantes:

- Programmation Orienté-Objet Appliquée
- Collections Java
- Développement Java Swing

Ce devoir fait suite au devoir 2 et continue avec le développement du jeu.

1.2 - Description du problème

Pour ce devoir, vous avez à ajouter des fonctionnalités au jeu développé lors du devoir 2. La mise en contexte se présente comme suit:

À la suite du travail réalisé lors de votre assignation précédente, un autre programmeur a continué le développement.

Il a notamment:

- remis de l'ordre dans le code (pour ceux pour qui ce n'était pas entièrement fonctionnel)
- implémenté un mécanisme de combat entre le héro est les créatures
- créer des objets d'équipements {armure, arme, casque, potion} pour aider le héros et les a dispersés dans le labyrinthe.

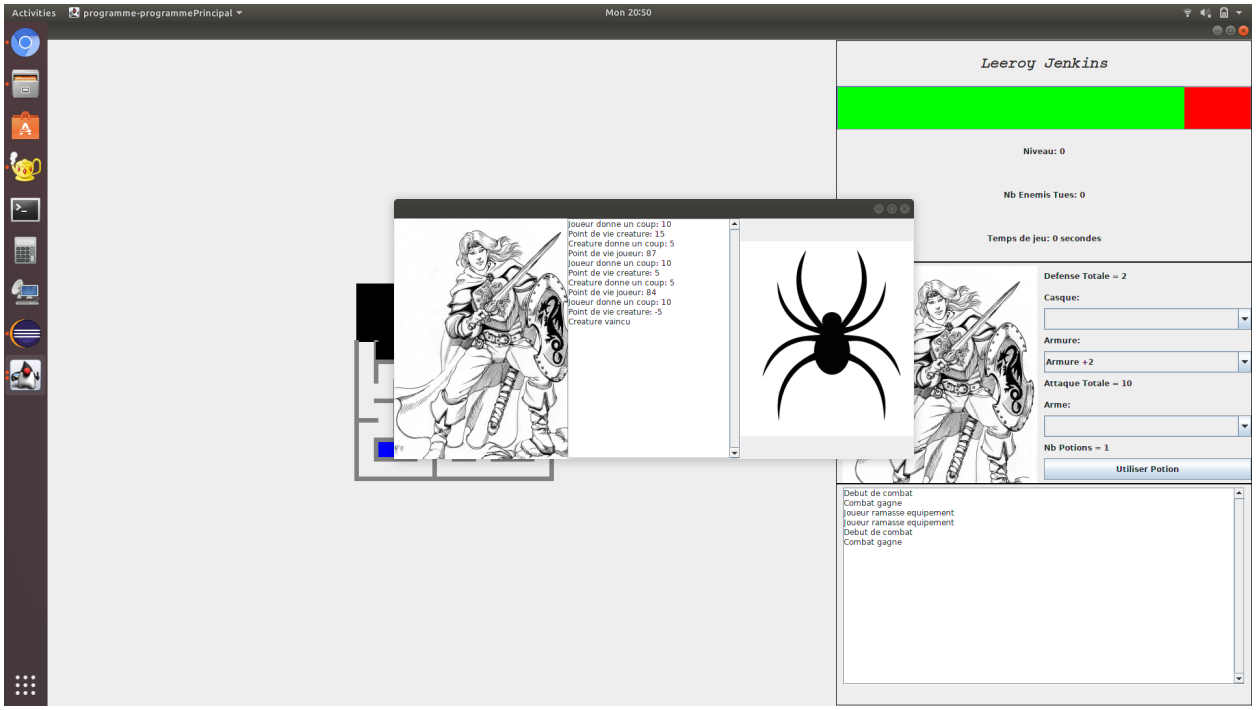
Il n'a par contre pas terminé le projet puisqu'il est maintenant parti en voyage. C'est à vous qu'incombe la tâche de compléter le jeu.

Voici ce qu'il reste à faire:

- Compléter le mécanisme de gestion des équipements, au niveau du joueur.

- Ajouter des éléments d'interface graphique pour informer le joueur sur l'état du héros et du jeu.

Lorsque vous aurez terminé, voici de quoi devrait avoir l'air votre jeu:



1.3 - Réalisation du travail

La première partie du devoir consiste à ajouter la gestion des équipements (Section 2). Il s'agit d'ajouter des fonctionnalités au programme actuel, en ajoutant du code dans des objets existants. Cette partie ne requiert qu'une bonne compréhension de l'héritage, du polymorphisme et de l'utilisation des collections.

Ensuite, le développement s'aligne vers l'ajout d'éléments graphiques, qui s'ajoutent au programme et qui interagissent avec les éléments existants, mais qui se fait principalement par l'ajout de fichiers (Section 3-4). Ces sections requièrent une bonne compréhension du développement à l'aide de Java Swing.

1.4 - Fichiers fournis

Une version complète de la solution du devoir 1, avec quelques fonctionnalités supplémentaires vous est fournie. Profitez-en pour comparer cette solution à votre version et prenez le temps de comprendre vos erreurs, s'il y a lieu.

Prenez note des changements ou ajouts suivants:

- un nouveau package a été défini: **equipements**
- une classe a été ajoutée à **modele::GestionnaireCombat**
- du code a été ajouté pour créer des équipements et les disperser dans le labyrinthe. Ce code a été ajouté dans plan de jeu et ressemble un peu au code définie pour les créatures. (voir: modele::PlanDeJeu::initEquipements, par exemple)
- Aussi, une classe de base a été ajoutée pour regrouper les fonctionnalitées communes aux personnages et équipements du labyrinthe, la classe **Donjon::AbstractObjet**.
- Un dossier contenant les images du héro et des créatures

Le programme fourni devrait compiler et s'exécuter sans changement. Lorsque le héros rencontre une créature, tout mouvement devrait devenir impossible tant que le combat n'est pas terminé. De plus, lorsque le héro passe sur un point jaune d'équipement, rien ne se produit. Le point jaune reste en place.

Ces deux éléments sont indicatifs des fonctionnalités manquantes que vous compléterez dans les prochaines sections.

Section 2 - Gestion des équipements (semaine 1)

Problème

L'interaction entre le joueur et les équipements et se fait comme suit:

- Quand le héros marche sur une pièce d'équipement, il le ramasse.
- La pièce d'équipement est stockée dans une liste qui appartient au héros.
- Plus tard, je joueur pourra équiper le héro avec les pièces d'équipement, mais cela sera géré par l'interface graphique (Section 4).

Stratégie d'implémentation

Tous les éléments permettant au héros de ramasser les équipements seront fait immédiatement et pourront être validés. Un certain nombre de services seront implémentés tout de suite, mais ils ne seront utilisés que quand l'interface graphique sera prête.

Implémentation

Ramasser les équipements

1) Le joueur doit pouvoir garder en mémoire les équipements ramassés. Pour cela, ajouter, à la classe joueur une collection (de votre choix) qui contiendra tous les équipements ramassés par le joueur.

2) Pour ramasser une pièce d'équipement, ajouter une méthode membre au joueur qui reçoit une référence à la pièce d'équipement à ramasser. Dans cette méthode on exécute:
    a) La mutatrice de l'équipement est appelée pour indiquer qu'il n'est plus au sol.
    b) La référence à l'équipement est ajoutée à la collection.

3) Ensuite il reste à mettre en place la détection de l'événement où le héros marche sur une pièce d'équipement. L'implémentation de ce processus est très similaire à celui de la détection d'un contact entre le héros et une créature, qui est déjà implémenté. Inspirez-vous du code déjà présent dans **PlanDeJeu::validerEtatJeu**, pour détecter quand le héros trouve de l'équipement.

Voilà, si vous tester votre programme, à chaque fois que le joueur marche sur un point jaune celui-ci disparaît. Si vous ajoutez des messages consoles supplémentaires, vous devriez être capable de valider que des équipements sont ajoutés à la collection membre du joueur.

Remise à zéro

Dans le cas où le joueur perd un combat contre une créature, une nouvelle partie est automatiquement lancée. Quand c'est le cas, il faut vider la liste d'équipement. Ajouter une instruction à cet effet dans la méthode **Joueur::remiseAZero**.

Utilisation des équipements

Il y a quatre types d'équipements: Casque, Armure, Arme et potions. Les trois premiers peuvent tous être équipés sur le héro pour lui donner un bonus de défense ou d'attaque. La potion sert à régénérer les points de vie.

Au héros, il faut ajouter 3 variables membres. Ces variables sont des références au casque équipé, à l'armure équipée et à l'arme équipée. Seule une pièce d'équipement par type peut être équipée en tout temps, même si plusieurs peuvent être présents dans la collection.

Il s'agit maintenant d'ajouter des services qui permettent d'interagir avec ces trois variables membres.

NOTE: les services suivants ne sont pas utilisés tout de suite, mais seront appelés lors du développement de la section 4.

1) **getEquipements**, cette méthode informatrice retourne une référence sur la collection d'équipement (comparer à **PlanDeJeu::getCreatures**)
2) **getCasqueEquipe**, cette méthode retourne une référence au casque équipé (null si aucun n'est équipé)
3) **getArmureEquipe**, cette méthode retourne une référence à l'armure équipé (null si aucune n'est équipé)

4) **getArmeEquipe**, cette méthode retourne une référence à l'armure équipé (null si aucune n'est équipé)
5) **equiper**, cette méthode permet d'équiper une pièce d'équipement reçu en paramètre (AbstractEquipement). Voici le processus:
    a) identification du type de la pièce d'équipement (instanceof)
    b) les seuls types d'équipements à identifier sont: Casque, Armure, Arme.
    c) change l'équipement équipé en question
    d) remet la variable membre *armure* à 0
    e) assigne à *armure* la somme des valeurs obtenues des équipements de défense {armure et casque}. Un équipement non équipé ne compte pas.
    f) remet la variable membre *bonusAttaque* à 0
    g) assigne à *bonusAttaque* la valeur de l'arme équipé, s'il y en a une
6) **utiliserPotion**, cette méthode trouve la première potion dans la collection d'équipement, l'enlève et remet les points de vie à point de vie max. Note, si cette méthode est appelée mais qu'il n'y a aucune potion elle n'a aucun effet.
7) **remiserAZero**, ajouter à cette méthode ce qu'il faut pour enlever l'équipement s'il y a une remise à zéro. Faites également un appel à **equiper(null)**, pour régénérer les calculs.

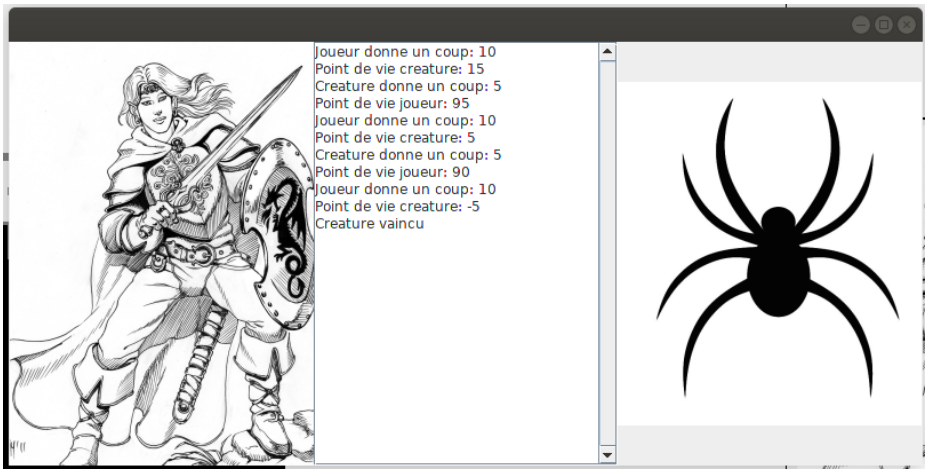
Section 3 - Ajout de la fenêtre de combat (semaine 2)

Problème

En ce moment, lors d'un combat entre le héros et une créature, rien ne se passe, du point de vue du joueur. On aimerait qu'une fenêtre style pop-up apparaisse et qu'elle indique au joueur l'état du combat.

Stratégie d'implémentation

La fenêtre pop-up est un JFrame qui est lancé au moment du combat. Le JFrame est lui même composé de trois panneaux, celui de gauche présente une image du héro, celui du centre une région de texte qui rapporte les messages et celui de gauche une image de la créature.



De plus, une fonctionnalité sera ajoutée pour bloquer l'exécution du jeu tant et aussi longtemps que la fenêtre n'aura pas été fermée.

Implémentation

1) Créer une nouvelle classe héritant de **JFrame** et implémentant l'interface **MonObserver**.
2) La classe contient les **variables membres** suivants:
    a) une référence au héro
    b) une référence à la créature combattu
    c) une référence au gestionnaire de combat
    d) une référence à un JTextArea
    e) une référence à un JScrollPane
    f) une référence à un JPanel (le panneau principale)
3) **Le constructeur** reçoit 3 paramètres, des références au: héro, créature et gestionnaire de combat. L'initialisation de la classe est ensuite divisé entre les sous programmes suivants (tous appelé, mais décrit plus bas):
    a) configuration du frame
    b) configuration de l'image du héro
    c) configuration de la boîte de message
    d) configuration de l'image de la créature

    Puis, le JFrame fait un appel à *requestFocus()* et *setVisible(true)*.
4) **configuration du frame**, la configuration du frame consiste à:
    a) initialiser la référence au JPanel, à l'aide d'un appel à getContentPane().
    b) définir la position de la fenêtre à 600,300
    c) définir la taille de la fenêtre à 800,400
    d) définir un GridLayout(0,3)
    e) ajouter un WindowListener, qui lors d'un événement de type windowClosing, fait un appel à *gestionCombat.combatTermine()*
5) **configuration image héro**:

    a) ajoute l'image au JPanel, à l'aide des lignes de codes suivantes (à vous de les faire fonctionner):

```
image = ImageIO.read(new File("images/hero.png"));
add(new JLabel(new ImageIcon(image)));
```

6) **configuration de la boîte de message**:
    a) crée une région de texte de 16 lignes et 20 colonnes
    b) la rendre non-éditable
    c) initialiser la référence membre prévu à cet effet
    d) crée un JScroll et lui attacher la région de texte (https://stackoverflow.com/questions/19212126/how-can-we-add-jscrollpane-on-jtextarea-in-java)
    e) scroll.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    f) initialiser la référence membre prévu à cet effet
    g) ajouter le JScroll au JPanel
7) **configuration image créature**:
    a) Choisir l'image en fonction du type de créature combattu.
    b) ajouter l'image au JPanel
8) **ajout de la fonctionnalité MonObserver**:
    a) La méthode avertir doit aller chercher les messages généré par le gestionnaire de combat et les afficher dans la région de texte.

Intégration

1) dans **GestionnaireCombat::executerCombat(...)**, ajouter l'appel au constructeur du JFrame
2) attacher le JFrame comme un observer du gestionnaireDeCombat
3) enlever la ligne: combatEnCours = false; à la fin de la méthode run() du gestionnaireDeCombat.

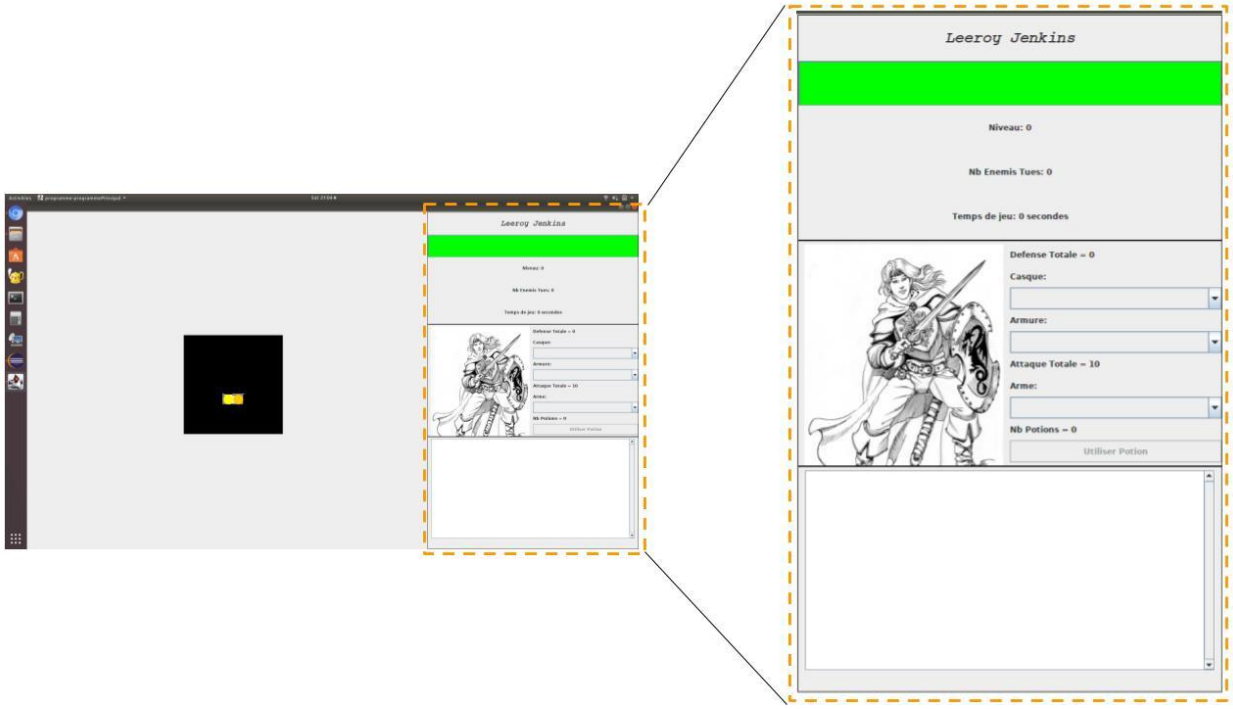
Validation

À ce moment-ci, si vous jouez une partie, vous devriez être en mesure d'observer les éléments suivants:

- À chaque fois que votre héros touche une créature et qu'un combat commence, votre fenêtre apparaît.
- Pendant le combat, des messages apparaissent.
- Tant que la fenêtre est présente, vous ne devriez pas pouvoir déplacer le héro dans le donjon, mais devriez pouvoir le faire dès que la fenêtre est fermée.
- Aussi, mais plus difficiles à observer, le mouvements des créatures devrait s'arrêter pendant un combat.

Section 4 - Ajout des panneaux d'informations (semaines 3 et 4)

Cette vise à ajouter des panneaux d'informations dans une bande à droite dans la console de jeu. Ces panneaux donnent de l'information sur le héros, son équipement et rapporte des messages sur l'état du jeu.

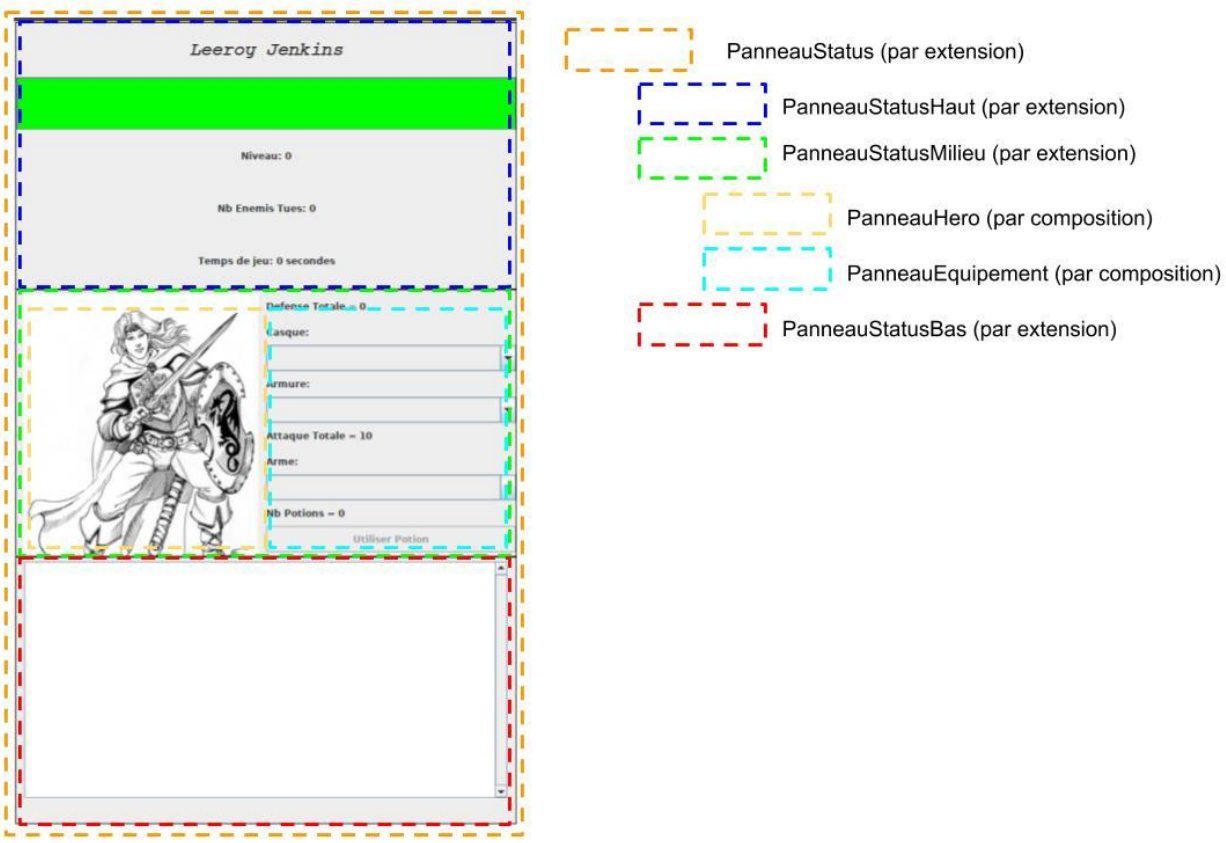


Stratégie d'implémentation

Plusieurs panneaux sont assemblés pour former la bande d'information. La figure qui suit montre les différents panneaux à définir et leur relation hiérarchique.

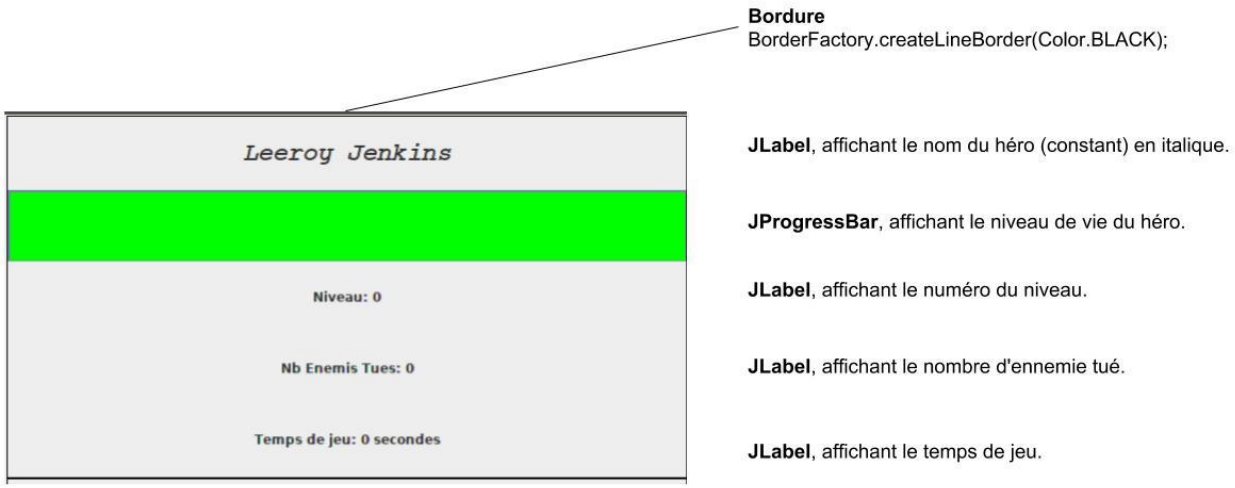
La première tâche consiste à créer chacun des panneaux et à s'assurer qu'ils tombent tous à la bonne place et ont tous la bonne taille.

Ensuite, il s'agit d'ajouter tous les éléments, puis d'y rattacher les fonctionnalités.



Implémentation des éléments visuels

1) Créer les panneaux selon le guide de la figure précédente. Voici quelques détails pour vous guider:
    a) Le PanneauStatus à une largeur égale à ⅓ de la largeur de la taille de l'écran, et une hauteur égale à la taille de l'écran. Il est instancié et membre du PanneauPrincipal. Le PanneauPrincipal a un Layout de type Border. À vous de trouver la bonne façon de placer le PanneauStatus à droite: https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html
    b) Chacun des PanneauxStatus{Haut, Milieu, Bas} est membre du PanneauStatus et doivent être placé en une colonne. Leur taille devrait s'ajuster automatiquement.
2) Pour le **PanneauStatusHaut**, voici la liste des éléments le composant:



Le nom du héro est en italique (*italic*) et est en caractère gras (**bold**) et est de taille 24. À vous de trouver comment changer le *font* d'un JLabel.

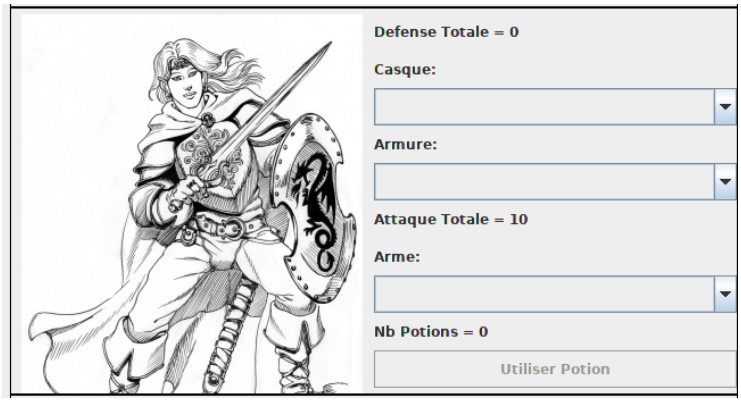
Vous devez définir une bordure, la ligne de code permettant de la définir vous est fournie, à vous de trouver comment la mettre en place.

Le JProgressBar à une couleur d'avant-plan (foreground) qui est vert et arrière plan (background) rouge.

Les autres éléments ne requiert aucune configuration particulière.

Prenez note que tous les JLabel sont centrés (setHorizontalAlignment)

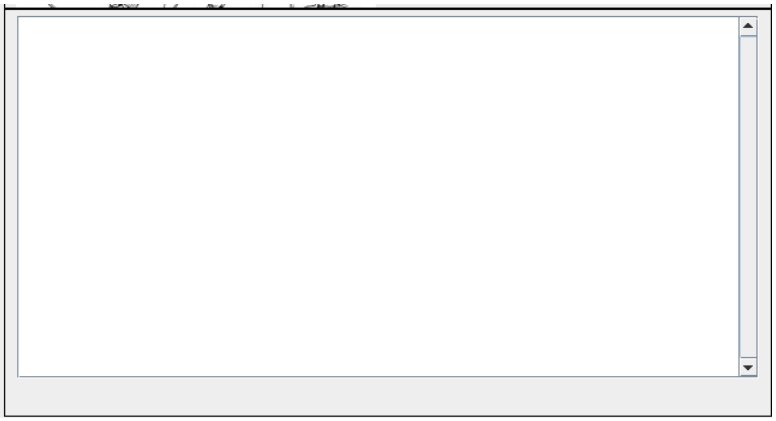
3) Pour le **PanneauStatusMilieu**, voici la liste des éléments le composant:



   a) Tout d'abord le panneau milieu à une bordure comme le panneau du haut. Il comporte aussi deux sous-panneaux ajoutés par composition: le panneau gauche et le panneau droit.
   b) Le panneau gauche contient une image du héro (voir Fenêtre de combat)

c) Le panneau droit comporte les éléments suivants:
    i) Éléments de défense
        (1) JLabel montrant la défense totale
        (2) JLabel indiquant Casque
        (3) JComboBox
        (4) JLabel indiquant Armure
        (5) JComboBox
    ii) Éléments d'attaque
        (1) JLabel montrant l'attaque totale
        (2) JLabel indiquant Arme
        (3) JComboBox (**NOTE**: les JComboBox utilisent un generic, pour indiquer le contenu {Casque, Armure, Arme})
    iii) Éléments d'attaque
        (1) JLabel indiquant le nombre de potions
        (2) JButton indiquant "Utiliser Potion". Le JButton est initialement inactif

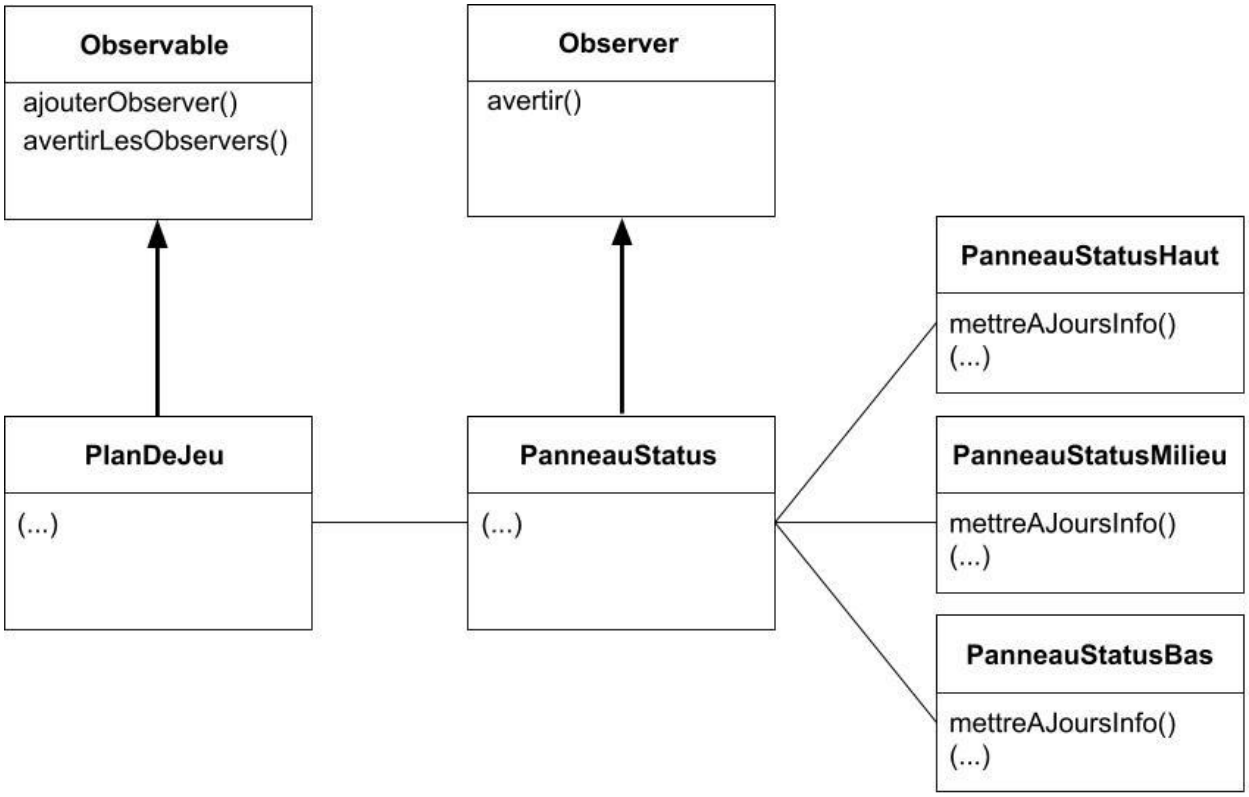
4) Pour le **PanneauStatusBas**, Un seul élément est présent il s'agit d'un JTextArea. Il est défini de la même façon que celui de votre fenêtre de combat, vous aurez à ajuster la taille pour qu'il s'insère bien dans votre panneau.



Communication des événements (modèle vers vue)

La communication entre le modèle (Plan de jeu) et la vue se fait selon le patron de conception Observer.

Un problème, par contre, est que les sous-PanneauxStatus{Haut, Milieu, Bas} doivent également être avertis. On pourrait rendre toutes ces classes Observer, mais on se retrouverait avec un programme où il y a trop de connections entre les classes, ce qui limite sa simplicité. À la place, vous limiterez l'implémentation au PanneauStatus, mais ce dernier s'occupera de relayer l'information aux sous-panneaux. Voici une image illustrant le schéma de communication.



1) PanneauStatus doit implémenter Observer et être attaché au plan de jeu.
   NOTE: pour obtenir une référence au PanneauPrincipal, appeler: PlanDeJeu.getInstance();, voir contrôleur clavier pour un exemple.
2) Ajouter une méthode appelée mettreAJoursInfo() dans chacun des sous-panneaux. Appeler ces méthodes dans la définition de PanneauStatus::avertir().

Mise à jour de l'information - PanneauStatusHaut

1) Les informations à mettre à jours pour le PanneauStatusHaut
   a) JProgressBar = joueur.getPointDeVie()*100/joueur.getPointDeVieMax()
   b) JLabel niveau = planDeJeu.getNiveau()
2) En ce qui concerne le nombre de créatures tuées et le temps joués en secondes leur implémentation vous est laissé en défi (2.5 points bonus pour chacun). Il vous faudra modifier le modèle pour y arriver. Bonne chance.

Mise à jour de l'information - PanneauStatusMilieu

1) Les informations à mettre à jours pour le PanneauStatusMilieu
   a) JLabel "attaque totale" = joueur.getForce()
   b) JLabel "defense totale" = joueur.getArmure()
2) Pour ce qui est des JComboBox est du nombre de potions, c'est un peu différent. Voici le processus par lequel ces éléments sont mis-à-jours

Vide les ComboBox (removeAllItems)
Crée un compteur de potions égal à zéro.

Pour chaque ComboBox (Casque, Armure, Arme), ajoute d'abord une référence à l'Équipement équipé, s'il y en a un.

Obtient ensuite une référence à la liste d'équipement: joueur.getEquipements()

Pour chaque élément de la liste
- l'ajouter au ComboBox auquel il appartient
- ajoute 1 au compteur s'il s'agit d'une potion

Affiche le nombre de potions
Si le nombre de potions est plus grand que 0, enable(true) le JButton, sinon enable(false).

Prenez note que les éléments équipés se retrouvent 2 fois dans la liste, c'est voulu, ça simplifie un peu l'implémentation.

Mise à jour de l'information - PanneauStatusBas

La PanneauStatusBas sera implémenté en fin de travail.

Validation

À ce moment-ci, vous devriez pouvoir observer les éléments suivants, en situation de jeu:
- La quantité de points de vie du joueur qui change lors des combats.
- La mise à jour des ComboBox, quand le héro ramasse de l'équipement au sol

Communication des éléments de contrôle (vue vers modèle)

Quelques éléments de contrôle sont présents dans le PanneauStatus. Tout d'abord, il y a le JButton qui permet de prendre une potion, puis les JComboBox.

JButton, utiliser potion

Le JButton, doit simplement appeler Joueur::utiliserPotion(), une méthode que vous avez définie précédemment.

Pour l'implémentation, suivez la méthode par déclaration de Classe implicite, dans le gestionnaire d'événement.

JComboBox

Pour ce qui est des JComboBox, voici un extrait de la solution:

```
public void itemStateChanged(ItemEvent event) {
	if (event.getStateChange() == ItemEvent.SELECTED) {
		Object item = event.getItem();
		joueur.equiper((AbstractEquipement) item);
	}
}
```

Il s'agit de fournir une définition pour la gestion d'événement de type itemStateChanged. Cet événement correspond à l'événement découlant d'une action sur la liste déroulante (ComboBox). La définition fournie effectue les opérations suivantes:

- Obtient la référence à l'élément sélectionné
- Fait suivre cet élément à la méthode **Joueur::equiper()**

Validation

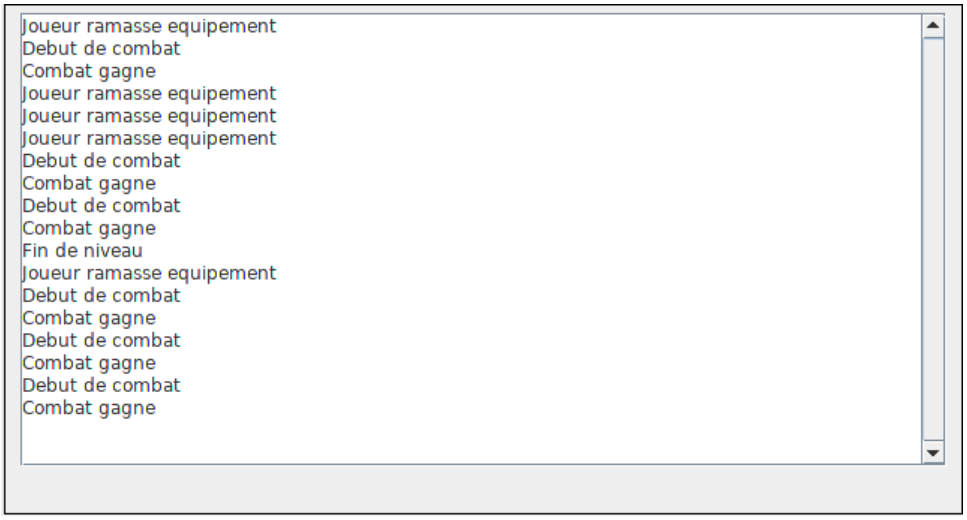
Voici comment valider les deux éléments:

- Après un combat, dans lequel le joueur a perdu des points de vie, utiliser une potion, pour retrouver les points de vie perdu. (Il faut avoir ramassé une potion au sol)
- Quand vous ramassez une pièce d'équipement au sol (autre qu'une potion), cet élément doit être visible dans le ComboBox et les points doivent se mettre à jour automatiquement (auto-équipé).
- Quand vous ramassez une pièce d'équipement au sol (autre qu'une potion), alors qu'un autre équipement est déjà équipé, il doit s'ajouter à la suite dans la ComboBox et doit être visible lors d'un click sur la liste déroulante.

PanneauStatusBas

Pour ce dernier panneau, un défi vous est lancé. Voici une description du comportement et une description sommaire de la stratégie d'implémentation. C'est à vous de trouver comment réaliser cet élément.

Le PanneauStatusBas agit un peu à la manière d'une console. C'est à dire qu'il offre un service static équivalent à println() et permet d'afficher des messages au Joueur. Voici l'état de la boîte après une courte séance de jeu.



La méthode permettant d'ajouter un message doit donc être accessible de partout dans le programme.

La stratégie d'implémentation consiste à utiliser une collection java pour contenir les messages. À chaque appel de la méthode pour ajouter un message, le message est ajouté à la collection. Lors d'une mise à jour, tous les messages sont inscrits dans le JTextArea.

C'est à vous de choisir les messages que vous voulez générer dans votre programme.

Bonne chance!

Instructions de remise

La remise se fera par moodle. Vous devez vous enregistrer comme une équipe, à l'aide de l'interface disponible.

Vous avez jusqu'à la date/heure limite pour soumettre votre travail, tous les travaux remis plus de quelques secondes après l'heure de la remise seront pénalisés.

Votre remise ne doit contenir que le code source, soit les fichiers .java et le dossier src/. Tout autre fichier présent sera considéré comme un manquement aux instructions.

Tous les membres de l'équipe doivent être identifiés, au minimum, dans les commentaires d'en-tête du programme principal.