Description brève

Le programme permet à une personne contrôler l'inventaire d'un héros avant de l'envoyer dans une bataille. Ainsi, il est possible de trier l'inventaire du héros, puis d'équiper des pièces d'armures et une arme, puis aller combattre un gobelin.

Description détaillée

Trier

Commandes : --type --numero –puissance –cle={cle} (défaut numero)

Cette commande permet de trier la liste des équipements du héros. Il est possible de spécifier s'il faut trier par puissance d'équipement, par type d'équipement, par numéro ou bien par une combinaison des trois.

Lister

Commandes : --status –inventaire (défaut status)
Cette commande permet de lister le statut du personnage avec « –status ». Voici un exemple d'affichage du status :

----------Informations générales---------
Héros: Olaf Odinkarsson
Vie max: 100
Attaque: 110 (Base: 10 - Arme: 100)
Défense: 110 (Base: 10 - Armure: 100)
----------Équipements utilisés----------
Tête : Casque legendaire (20 défense)
Torse : Plastron legendaire (20 défense)
Mains : Gants legendaires (20 défense)
Pantalons : Pantalons legendaires (20 défense)
Bottes : Bottes legendaires (20 défense)
Arme : Arc legendaire (100 attaque)

Avec « --inventaire » il est possible de lister l'ensemble des équipements que possède le personnage. Voici un exemple d'affichage de l'inventaire :

0 : Casque de cuir (Tete) - puissance: 1
1 : Plastron de cuir (Torse) - puissance: 1
2 : Gants de cuir (Mains) - puissance: 1
3 : Pantalons de cuir (Pantalons) - puissance: 1
4 : Bottes de cuir (Bottes) - puissance: 1
5 : Casque de fer (Tete) - puissance: 5
6 : Plastron de fer (Torse) - puissance: 5

Equiper

Commandes : --numero-{numero} –meilleur (defaut meilleur)

La commande « équiper » permet d'équiper un équipement, soit par son numéro, soit demander au programme de choisir automatiquement l'équipement le plus puissant de chaque catégorie avec l'option « --meilleur ». À noter que les types d'équipements sont (avec la statistique qu'elle donne entre parenthèse) :

1- Tete (défense)
2- Torse (défense)
3- Mains (défense)
4- Pantalons (défense)
5- Bottes (défense)
6- Arme (attaque)

Jouer

La commande jouer va lancer une simulation de combat entre le héros et un gobelin. Il sera possible de voir à chaque tour les dégâts infligés par chacun à l'un ou l'autre des combattants. À la fin du combat un message annonce si le héros a gagné ou perdu. Équiper des meilleures pièces d'équipement rendra le gobelin plus fort, pour que le challenge soit toujours relativement semblable. À noter que le combat est automatique. Le héros ou le gobelin attaqueront ou défendront de façon aléatoire jusqu'à ce qu'un des deux perde tous ces points de vie.

Help

Voici la sortie de la commande help :

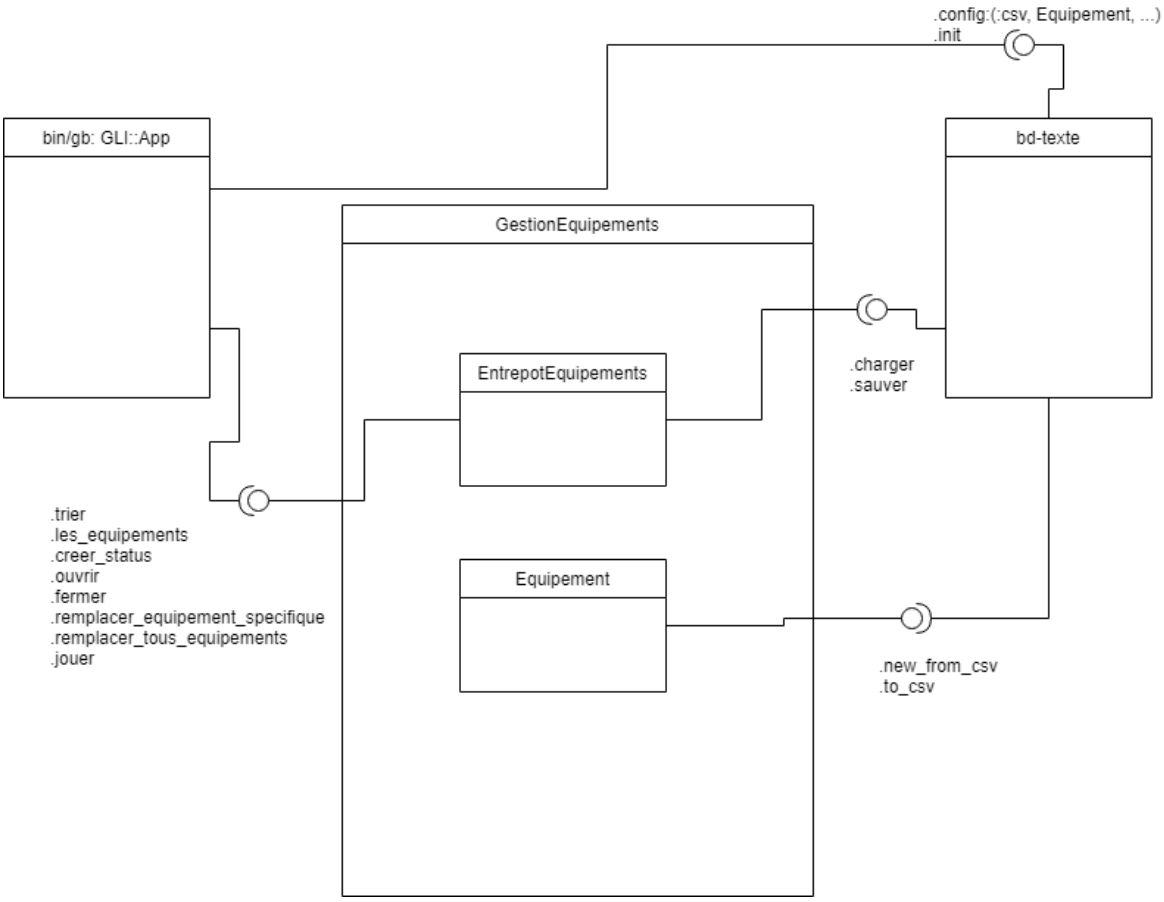
```
NAME
   gv - Application en ligne de commande pour aller battre des gobelins

SYNOPSIS
   gv [global options] command [command options] [arguments...]

VERSION
   0.0.1

GLOBAL OPTIONS
   --depot=arg          - Depot de donnees a utiliser pour les equipements (default: .joueur.txt)
   --help               - Show this message
   --[no-]stdin, --[no-] - Utilisation de stdin plutot que le depot
   --version            - Display the program version

COMMANDS
   equiper    - Equiper une pièce d'equipement
   help       - Shows a list of commands or help for one command
   init       - Cree une nouvelle base de donnees pour gerer des vins (dans './.joueur.txt' si --
                depot n'est pas specifie)
   jouer      - Permet de combattre avec son heros
   lister, ls - Affiche le status du heros ou de son inventaire
   trier      - Tri les enregistrements selon divers criteres
```

Description de l'environnement

J'ai travaillé de la maison en étant connecté sur java.labunix via Putty. Je faisais mes modifications au code sur ma machine locale (Windows 10) et ensuite j'envoyais mes fichiers via Filezilla pour après compiler et exécuter sur le serveur de l'université. Mon éditeur de texte était Atom. Il s'agit d'un éditeur de texte très puissant, qui identifie plusieurs langages de façon native, dont le Ruby. Il possède une intégration avec Git ce qui me permet de voir dans l'arborescence de mon projet les fichiers que j'ai modifiés. Il possède aussi de l'auto complétion de fonctions selon les attributs que j'ai créé ou les fonctions que j'ai créées. Cette dernière fonctionnalité ralentit énormément l'éditeur de texte par contre vers la fin

Description de l'architecture

Puisque je suis parti de l'architecture « GLI::App » de l'application de gestion des vins, la structure de mon projet est identique à celle-ci. J'ai surtout ajouté beaucoup de fonctions entre le « GLI :App » et « EntrepotEquipements » pour faire le lien entre l'interface et la couche de métier pour mes commandes, qui sont nouvelles et différentes de l'application desquelles elles s'inspirent.



Description des tests

Aucun test n'a été effectué pour mon application.

Conclusion

Parmi les choses que j'ai apprises avec le langage Ruby est d'adopter une pensée plus fonctionnelle dans le sens où le langage permet de faire beaucoup de chose en peu de lignes du moment que l'on sache s'y prendre comme il le faut. J'ai encore de la misère à saisir toutes les nuances du langage, notamment au niveau de la méta programmation. Il m'arrive souvent d'utiliser des structures de données peu adaptées à la situation, car elles n'existent pas ou sont très peu utilisées dans les langages orientés objets courants que nous apprenons.

Par exemple, utiliser la fonction « reduce » pour aller chercher la meilleure arme dans la liste de toutes les armes de cette application m'a pris beaucoup de temps. Je ne voyais pas autrement qu'avec une boucle conventionnelle comment faire pour aller chercher cette donnée.

Une autre chose que j'ai bien aimé c'est qu'il est possible d'aller chercher ou assigner dynamiquement des valeurs à des attributs d'objets via la fonction « send ». Dans mon code il y a un moment où j'essayais d'assigner une valeur à un équipement dont je ne connaissais pas encore le type. Grâce à cette fonction j'ai pu assigner la valeur sans avoir à faire un switch/case.

Pour conclure, la façon de penser change dramatiquement par rapport à des langages plus conventionnels, tels que Java ou C#. Cela fait beaucoup plus penser à un langage comme Haskell qui prends un temps énorme avant d'en saisir toutes ces subtilités.