TP n°4 *MUD, conception objet en équipe*

Dans ce TP, vous allez vous familiariser avec un framework de MUD (Multi-User Dungeon) qui vous est fourni. Un MUD classique est un jeu d'aventure et d'exploration utilisant une interface purement textuelle. Le framework qui vous est proposé, que j'ai appelé `alamud`, permet de développer un jeu sans programmation (sauf si vous voulez ajouter de nouvelles fonctionnalités) : les objets, les lieux, et ce qui se produit quand on fait une action... tout peut être décrit simplement en YAML.

Exercice 1. Cloner le framework

Logger vous dans votre compte sur `https://gitlab.com/`. Puis visitez la page du projet `alamud` :

`https://gitlab.com/denys.duchier/alamud`

Cliquez sur `Fork` pour obtenir votre propre copie sur gitlab. Vous pouvez maintenant obtenir une copie locale en utilisant l'url HTTPS de votre copie :

```
git clone https://gitlab.com/VOTRE-ID-GITLAB/alamud.git
```

Exercice 2. Utiliser le mini-jeu

Le framework est fourni avec un mini-jeu dont le seul but est de servir d'exemple. Pour démarrer le jeu :

```
cd alamud
./mud.py
```

Maintenant ouvrez un navigateur (récent car il doit supporter les websockets) et dirigez le sur l'url `http://localhost:9999`. Cliquez sur `Register` pour vous créer un compte. Choisissez un identifiant, e.g. `thoralf` ; un mot de passe ; et optionellement une description, e.g. `le teigneux`, pour mieux caractériser votre personnage. Cliquez également sur le genre approprié (cette information est parfois utilisée pour des raisons grammaticales quand un texte descriptif est préparé par le moteur de jeu).

Si vous séchez et ne savez pas quelle commande utiliser, vous pouvez tricher : le répertoire `mud/games/iut` contient toutes les données du mini-jeu. En particulier, les règles correspondant aux commandes possibles se trouve dans le fichier `mud/games/iut/__init__.py` dans la fonction `make_rules`.

Vous pouvez aussi vous connecter à plusieurs sur le même serveur.

Exercice 3. Découvrir la description du mini-jeu

Dans le répertoire `mud/games/iut`, il y a 2 fichiers YAML pour décrire le mini-jeu :

`initial.yml`
: ce fichier décrit tous les objets, les lieux, etc. . . ainsi que ce qui doit se passer quand survient un évènement.

`static.yml`
: ce fichier donne des paramètres de configuration, ainsi que des reactions par défaut quand l'une d'elle manque dans `initial.yml`.

3.1 `initial.yml` : ce fichier contient une séquence de *documents yaml* ; chaque document commence par le séparateur `---` et décrit un objet du jeu.

Location : Voici la description d'un lieu :

```
---
id: parking-000
type: Location
contains:
  - lampe-000
events:
  info:
    actor: "Parking de l'IUT."
  look:
    actor: |
      Vous êtes sur le parking de l'IUT.  Au nord-est, se trouve
      l'entrée du département info.  Il y a un arbre près de vous.
      Les branches les plus basses sont à portée de main.
```

Chaque object doit avoir un `id` qui l'identifie de manière unique, et un `type` qui nomme la classe dont il est une instance. Un `Location` ou un `Container` peuvent contenir d'autre objets ; ceci est spécifié par la clé `contains` qui est une liste d'`id`.

Enfin, on peut attacher à un objet des règles indiquant ce qu'il faut faire en réaction à certains évènements le concernant. Ces règles sont données sur la clé `events`.

L'évènement `info` est automatiquement déclenché quand on arrive dans un lieu et provoque l'affichage d'une description brève de ce lieu. `events.info.actor` est donc ce qu'il faut afficher au joueur qui vient d'arriver dans ce lieu.

L'évènement `look` est déclenché quand on regarde (autour de soi) ou quand on regarde quelque chose en particulier. `events.info.actor` est ce qu'il faut afficher au joueur dans ce cas.

Portal : un `Portal` représente un passage entre différents lieux. Ce passage a des `Exit` dans chacun de ses lieux.

```
---
id: portal-parking-porche-000
type: Portal
exits:
  - id: parking-000-nord-est
    location: parking-000
    direction: nord-est
  - id: porche-000-sud-ouest
    location: porche-000
    direction: sud-ouest
```

Ceci représente un passage entre le parking de l'IUT et le porche du département info. L'exit sur le parking est située dans la direction nord-est : lorsqu'on est sur le parking, on peut donc atteindre le porche du département en se déplaçant vers le nord-est. L'exit sur le porche est naturellement située en direction sud-ouest.

Thing : un `Thing` représente une chose, un objet ordinaire, situé dans un lieu ou un conteneur. La clé `name` permet de lui donner un nom. Le nom est nécessaire pour pouvoir dire "`prendre le badge`".

```
---
id: badge-000
type: Thing
name: badge
props:
  - takable
  - key-for-porte-porche-000
events:
  info:
    actor: "Un badge de sécurité."
  look:
    actor: |
      C'est un badge de sécurité permettant d'ouvrir des
      portes sécurisées.
```

Tous les objets du jeu peuvent avoir des propriétés, et celles-ci peuvent être modifiée pendant le jeu. La clé `props` donne les propriétés que l'objet a initialement. La propriété `takable` indique que l'objet peut être pris. La propriété `key-for-porte-porche-000` indique que l'usage de cet objet permet d'ouvrir l'objet d'id `porte-porche-000`.

Parts : un objet peu également avoir des composants indiqués par la clé `parts`. Ceux-ci fournisse des objets nommable sur lesquels on peut opérer. Par exemple : un bouton qu'on peut pousser.

```
---
id: hall-000
type: Location
props:
  - dark
events:
  info:
    actor: "Le hall du département info."
  look:
    actor: |
      Vous êtes dans le hall du département info de l'IUT.
      Au sud-ouest, la porte d'entrée mène au porche.  Un
```

```
      escalier mène à l'étage.  Un tableau d'affichage est
      accroché au mur.
parts:
  - tableau-000
  - notice-000
```

Ici, il y a un tableau d'affichage qu'on peut donc observer ; et une notice accrochée au tableau, qu'on peut donc lire.

De plus, ce lieu a la propriété `dark`. Ceci veux dire que vous ne pouvez rien y faire car vous n'y voyez rien. Pour remédier à cela, il faut une source de lumière : par exemple une lampe torche allumée ; mais celle-ci peut être tenue par un autre joueur.

Effect : Ce qui se passe en réaction à un évènement peut être affiné très précisément en indiquant des effets grâce à la clé `effects`.

```
---
id: bouton-cabane-000
type: Thing
name:
  - bouton
props:
  - pushable
events:
  look:
    actor: |
      Un bouton étiqueté: <i>Taxi Service</i>.
  push:
    actor: |
      Vous appuyez sur le bouton, mais rien ne semble se
      produire.
    observer: |
      {{ actor.name }} appuie sur le bouton, mais rien ne
      semble se produire.
    effects:
      - type  : ChangePropEffect
        modifs: =portal-ordinateur-cabane-exit-2:activated
```

Ici, la propriété `pushable` indique que l'objet (un bouton) peut être poussé. L'évènement `push` est déclenché lorsqu'on dit "`pousser le bouton`". `events.push.actor` indique ce qu'il faut afficher au joueur en réaction à cet évènement.

Mais il peut y avoir d'autres personnes dans le même lieu que le joueur, et celles-ci voient ce qu'il fait. Il faut donc aussi afficher quelque chose pour les informer. `events.push.observer` sert à cela. Le texte correspondant (comme tous les textes) est un template. Il contient l'expression de template `{{ actor.name }}` qui sera remplacée par le nom du joueur.

Finalement `events.push.effects` décrit une liste d'effets à executer. Ici la liste a un seul élement. `ChangePropEffect` est un effet qui permet de modifier des propriétés d'objets du jeu. la clé `modifs` indique les propriétés à modifier : sa valeur est soit une modification, soit une liste de modification. `=portal-ordinateur-cabane-exit-2:activated` indique que l'objet d'id `portal-ordinateur-cabane-exit-2` doit être attribué la propriété `activated`.

Si on voulait enlever la propriété, on écrirait `-=portal-ordinateur-cabane-exit-2:activated`, i.e. avec un signe “moins” au début.

Enter/Traverse/Leave : quand vous vous déplacez par un passage, les personnes sur votre lieu de départ vous voient partir, et celles sur votre lieu d’arrivée vous voient arriver. Un déplacement est donc normalement divisé en 3 évènements successifs : `enter-portal`, `traverse-portal`, `leave-portal`.

Voici un exemple, pour l’arbre dans lequel on peut grimper :

```
---
id: portal-parking-arbre-000
type: Portal
exits:
  - id: parking-000-haut
    location: parking-000
    direction: haut
    events:
      enter-portal:
        actor   : "Vous grimpez prestement dans l'arbre."
        observer: "{{ actor.name }} grimpe dans l'arbre."
      leave-portal:
        observer: |
          {{ actor.name }} descend de l'arbre et vous rejoint
          sur le parking de l'IUT.
  - id: arbre-000-bas
    location: arbre-000
    direction: bas
    events:
      enter-portal:
        actor   : "Vous descendez prudemment de l'arbre."
        observer: "{{ actor.name }} vous quitte et descend de l'arbre."
      leave-portal:
        observer: "{{ actor.name }} vous rejoint dans l'arbre."
```

FailedAction : l’évènement `failed-action` est généré quand il est impossible d’effectuer une action, soit par ce qu’on n’y voit rien car il fait trop sombre, soit parce qu’on n’arrive pas à trouver un des éléments nécessaire à l’action.

Voici l’entrée `events.failed-action` fournie par `static.yml`. Elle illustre une syntaxe **conditionelle** pour `events.failed-action.actor` :

```
failed-action:
  - props   : action:cannot-see
    actor   : "Dans cette obscurité, vous n'y voyez rien."
  - props   : action:go
    actor   : "Il n'y a pas d'issue dans cette direction."
  - props   : action:cannot-find-object
    actor   : "{{ action.object }}: introuvable!"
  - props   : action:cannot-find-object2
    actor   : "{{ action.object2 }}: introuvable!"
```

C’est une liste, ou chaque élément à une clé `props` indiquant les propriétés qui doivent être vérifiées pour cette alternative soit applicable.

La propriété `action:cannot-see` est vérifiée si l’`action` (donnée par la moteur de jeu dans le contexte d’évaluation) a la propriété `cannot-see`.

Exercice 4. Ajout d'une salle dans le département info

Mettez-vous en équipe (binôme ou trinôme, selon la configuration de la salle de TP). Ajoutez chacun les remotes nécessaires pour travailler avec vos co-équipiers. Puis en pair (triplet) programming, ajoutez chacun un lieu au mini-jeu.

Vous devrez donc également ajouter des passages pour accéder à ces lieux.

Exercice 5. Ajout d'objets

Vous allez probablement chacun avoir besoin d'une lampe, un flambeau, une bougie, ou que sais-je pour pouvoir voir dans les lieux sombres. Mettez-en dans votre jeu.

Exercice 6. Allumer un flambeau avec un briquet

Pour cet exercice, il faudrait un flambeau éteint et un briquet. Vous devez alors modéliser la possibilité d'allumer le flambeau avec le briquet.

Exercice 7. Ascenseur

Ajoutez au département info un ascenseur pour se déplacer entre le rez de chaussée et le 1er étage. La difficulté ici est de bien gérer les portes et les transitions de l'automate corespondant à l'ascenseur. Vous **devez** utiliser les outils UML pour vous aider à concevoir la modélisation de l'ascenseur.