TP n°$4b$ *Guide d'extension du MUD*

Exercice 1. Le flambeau et le briquet

Pour étendre le mud avec un flambeau qu'on peu allumer avec un briquet, il faut ajouter le support pour une nouvelle action *"allumer X avec Y"*, et aussi donner au flambeau et au briquet des propriétés qu'on utilisera dans la modélisation.

Propriétés : Je propose que le flambeau ait des propriétés exprimant les capacités suivantes :

- prenable
- inflammable

et que le briquet ait (au moins) les suivantes :

- prenable
- igniteur

Ajout de l'action "`allumer X avec Y`" :

`mud/games/iut/__init__.py`
ajouter une règle pour ce nouveau pattern ; cette règle utilise la nouvelle action `LightWithAction` (pas encore écrite).

`mud/actions/light.py`
définir une classe `LightWithAction` qui hérite de `Action3` (voir `mud/actions/action.py`) :

- Action1 : juste un acteur
- Action2 : acteur+objet
- Action3 : acteur+objet+objet2

Une action n'est que le résultat de l'analyse syntaxique de la commande utilisateur. Le vrai boulot est effectué par un évènement qui est la représentation sémantique qui lui correspond. Chaque action est associée à un `EVENT` qui est la classe de l'évènement à construire. Ici il faudra utiliser le nouvel évènement `LightWithEvent` (pas encore écrit).

Les paramètres de l'action (**`object`** et `object2`) sont juste des mots. il faut réaliser une résolution sémantique contextuelle pour déterminer les objets auxquels ils font référence. Les attributs (de classe) `RESOLVE_OBJECT` et `RESOLVE_OBJECT2` doivent être affectés des noms de méthodes à utiliser pour faire cette résolution.

Ces noms de méthodes sont de la forme `resolve_for_XXX`. Les méthodes correspondantes se trouvent dans la classe `Player` (voir en bas du fichier `mud/models/player.py`). Une méthode `resolve_for_XXX` appelle une autre méthode `find_for_XXX`. Les méthodes `find_for_XXX` ont chacune une docstring qui explique leur stratégie de résolution.

`mud/actions/__init__.py`
Les modules qui veulent utiliser une action importent `mud.actions` dont le code est dans `mud/actions/__init__.py`. Pour rendre une nouvelle action disponible, il faut donc qu'elle soit importée (comme variable) dans ce fichier.

`mud/events/light.py`
définir une classe `LightWithEvent` qui hérite de `Event3` (voir `mud/events/event.py`). La méthode `perform` spécialise l'exécution de l'évènement. Elle invoque :

```
self.inform("light-with")
```

Cette méthode cherche (sur l'objet), sous la clé `events`, un template pour `light-with.actor` et s'en sert pour créer la contribution narrative à envoyer au joueur.

Puis elle cherche un template pour `light-with.observer` et s'en sert pour créer, pour chaque observateur, une contribution narrative qu'elle lui envoit.

`mud/events/__init__.py`
ici aussi il faut importer `LightWithEvent` comme variable pour le rendre disponible au travers du module `mud.events`.

Exercice 2. Modélisation de l'ascenseur.

Modéliser l'ascenseur était plus complexe que je ne l'avais anticipé. Avec une petite extension du MUD qui simplifie la spécification déclarative, j'ai créé une solution `ascenseur-initial.yml` que j'ai mise à disposition sous pub.

Elle contribue 3 nouveaux lieux au département (mais il faudrait aussi modifier initial.yml pour faire apparaître les nouvelles issues dans les lieux précédemment existants) :

- un coin de l'ascenseur dans le hall
- un coin de l'ascenseur à l'étage
- la cage de l'ascenseur

il y a un bouton pour appeler l'ascenseur en bas, un autre en haut, et un bouton dans l'ascenseur pour changer d'étage.

Le portal de l'ascenseur à 3 exits :

- une issue au rez de chaussée

- une issue à l'étage
- une issue dans la cage de l'ascenseur

selon que l'ascenseur est en haut ou en bas, l'issue dans la cage de l'ascenseur est associée soit à l'issue à l'étage, soit à celle au rez de chaussée.

Lorsqu'on tente d'entrer dans une issue, le moteur de jeu calcule un objet `Traversal` avec un attribut `exit1` (pour l'issue d'entrée) et un autre `exit2` (pour l'issue de sortie). Lorsqu'il n'y a que 2 exits pour un portal, l'objet `Traversal` peut être déterminé automatiquement. Ici il y en a 3, donc il faut un peu d'aide de la modélisation.

Le moteur de jeu demande à `exit1` si, dans le yaml, il a une entrée pour la clé `traversal`. Si `exit1` n'en a pas, il transmet la requète à son portal.

De plus, le yaml peut contribuer des *effets* supplémentaires à l'exécution normale d'un évènement. Ces effets sont donnés par une liste sous la clé `effects`. Ceci est particulièrement utile pour les effets associé à l'action de pousser un bouton : en effet, selon le bouton, on veut pouvoir exécuter des effets très différents.

Les effets disponibles sont définis dans le répertoire `mud/effects`.