# Projet : première partie

**Rendre cette partie dans son état d'avancement, quel qu'il soit, le 29 mars 2009 à minuit au plus tard.**

Mars 2009

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | 19 | 20 | 21 | 22 |
| 23 | 24 | 25 | 26 | 27 | 28 | |
| | | | | | | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 23 | 24 | 25 | 26 | 27 | 28 | 29 |

Le but du projet de programmation Java de cette année est l'implémentation d'un moteur de jeu d'aventures textuelles en Java. Un tel jeu fonctionne comme une boucle d'entrées/sorties où le moteur analyse les actions du joueur, écrites en un français largement appauvri, change l'état de l'univers virtuel en conséquence, et décrit textuellement les effets visibles au joueur.

Un exemple de partie pourrait ressembler au listing suivant, où les entrées du joueur sont précédées par un symbole « `>` » :

```
[Plage]
  Bienvenue dans l'aventure de démonstration. Vous êtes sur une
grande plage de sable blanc, bordée par la mer au sud et par une
forêt à l'est. Un phare abandonné la domine du côté est.

> Aller au phare.

La porte est fermée.

> Faire l'inventaire.

Votre inventaire actuel : un morceau de ficelle emmêlée.

> Aller à la forêt.

[Forêt]
  Une forêt de pins. Le sol est couvert d'aiguilles. Une vieille
pelle est posée contre un arbre. Un chemin serpente vers la plage
à l'ouest.

> Prendre la pelle.
```

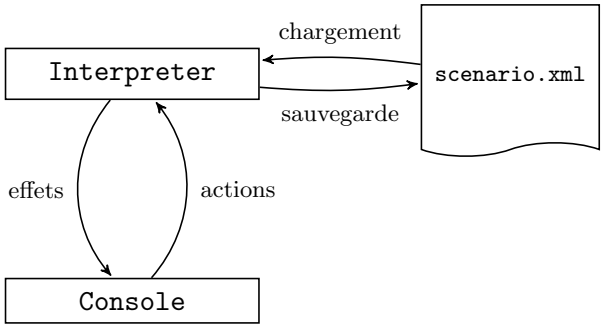
```
Vous prenez la pelle.

> Décrire.

[Forêt]
  Une forêt de pins. Le sol est couvert d'aiguilles. Un chemin
serpente vers la plage à l'ouest.

> Faire l'inventaire.

Votre inventaire actuel : un morceau de ficelle emmêlée, une pelle
rouillée.

> Aller à l'ouest.

[Plage]

> Aller à la mer.

L'eau est froide et tumultueuse. Vous vous dites que qu'apprendre à
nager aurait tout de même pu vous servir.

PARTIE PERDUE : Vous vous noyez.
```

**Déroulement** Ce projet est divisé en trois parties :

1. un interpréteur pour les scénarii de jeu,
2. la gestion des entrées/sorties, et
3. des améliorations libres (on vous fournira de nombreuses pistes dans ce sens).

# 1 Organisation de la première partie

L'architecture proposée pour ce projet suit le schéma suivant :



Un interpréteur pour un langage XML spécialisé (`Interpreter` sur le schéma) garde en mémoire l'état actuel de l'univers du jeu. Il peut charger et sauvegarder cet état courant sous la forme d'un fichier XML.

L'interpéteur interagit avec une console pour ses entrées/sorties avec le joueur. Pour cette première partie du projet, une première implémentation de cette interface utilisateur sera écrite (la `DebugConsole`), sachant que des modèles plus évolués seront écrits par la suite. Cette interaction se fait par des appels sur des actions par la console, qui donnent lieu à des messages de l'interpréteur affichées par la console.

D'autres éléments seront introduits dans ce schéma dans la deuxième partie du projet pour permettre une interaction plus « naturelle » avec l'interpréteur.

Avant de donner plus de détails sur la spécification de l'interpréteur, voici à quoi ressemble le début du listing précédent avec la `DebugConsole` :

```
[beach]
  Bienvenue dans l'aventure de démonstration. Vous êtes sur 'un
beach' de sable blanc, bordée par 'le sea' au sud et par 'un forest'
à l'est. 'un lighthouse' abandonné la domine du côté est.

> go(lighthouse)

La porte est fermée.

> inventory()

Votre inventaire actuel : 'un string'.

> go(forest)

[forest]
  'un forest' de pins. Le sol est couvert d'aiguilles. 'un shovel' est
posée contre un arbre. Un chemin serpente vers 'le beach' à l'ouest.

> take(shovel)

Vous prenez 'le shovel'.
...
```

Les actions utilisateurs sont entrées à l'aide de leurs noms symboliques, et les noms des objets sont pour l'instant aussi leurs identifiants dans le fichier de scénario.

## 2 Le langage de description

Le langage de description de scénarii de jeu est un dialecte XML. On trouvera la DTD définissant la syntaxe du dialecte dans `src/resources/scenario.dtd`, et des exemples

dans `samples` dans l'archive de la première partie du projet.

Les trois principales sortes d'éléments XML dans le langage de description sont :

- Les *effets*, qui sont des opérations implémentées dans l'interpréteur (par exemple `print` pour afficher un message, `call` pour appeler une action, `win` pour finir le jeu sur une victoire).
- Les *instances* qui sont des objets résidant en mémoire dans l'interpréteur, et contiennent d'autres instances et des actions. Une instance est déclarée avec un identifiant unique, et peut *étendre* une autre instance, par exemple :

  ```
  <instance id="string" extends="object"/>
  ```

  Dans cet exemple, `string` pourra appeler les actions d'`object` et voir les instances dans sa portée. Des actions d'`object` peuvent être redéfinies dans `string`.
- Les *actions* qui sont définies dans le scénario dans certaines instances. Ce sont ces actions qui sont réalisées par le joueur dans l'instance courante.

Le langage a la particularité d'utiliser XPath pour naviguer entre les instances dans l'évaluation des effets : ainsi le test

```
<pre test="*[ancestor::*/@id = 'object']"/>
```

sert à déterminer s'il existe une instance (sélectionnée par `*`) dans la portée de l'instance courante dont un ancêtre est identifié par `object`, ce qui sert à implémenter une relation d'extension entre cette instance et l'instance `object`.

## 2.1 Syntaxe

La syntaxe est définie dans la DTD `scenario.dtd`. Celle-ci est chargée automatiquement par l'interpréteur pour valider la syntaxe du fichier scénario fourni si elle se trouve dans le `CLASSPATH`.

Un scénario est défini dans un élément `scenario` avec en premier chef une séquence d'effets initiaux (dans un élément `storyline`) et ensuite une séquence de bibliothèques (des éléments `library` – il y a justement une bibliothèque par défaut fournie dans le répertoire `src/resources/library.xml`) définissant des instances.

En bref, un fichier de scénario valide a la forme

```
<?xml version="1.0"?>
<!DOCTYPE scenario SYSTEM "scenario.dtd">
<scenario>
  <storyline start="universe">
    <!-- séquence d'effets initiaux -->
    <!-- ... -->
  </storyline>

  <!-- séquence de bibliothèques -->
  <library>
    <!-- séquence d'instances -->
    <!-- ... -->
```

```
  </library>

  <library>
    <!-- séquence d'instances -->
    <!-- ... -->
  </library>

  <!-- ... -->
</scenario>
```

À noter que l'élément `storyline` déclare l'instance active au début du scénario, qui **doit** apparaître dans une des bibliothèques (ici `universe`).

## 2.2 Exemples

Pour mieux comprendre le langage de description, voici quelques exemples d'utilisation.

### 2.2.1 Hello, world!

```
<?xml version="1.0"?>
<!DOCTYPE scenario SYSTEM "scenario.dtd">
<scenario>
  <storyline start="universe">
    <print>Hello, world!</print>
  </storyline>
  <library>
    <instance id="universe"/>
  </library>
</scenario>
```

Ce scénario se contente d'afficher « Hello, world! » sur la console. Il possède une seule instance, `universe`, parce qu'elle est obligatoire pour initialiser le scénario, mais ici elle est vide. Il n'y a aucune action proposée.

```
[universe]
  Hello, world!

>
```

### 2.2.2 Fonction factorielle

```
<?xml version="1.0"?>
<!DOCTYPE scenario SYSTEM "scenario.dtd">
<scenario>
```

```
  <!-- The storyline. -->
  <storyline start="universe">
    <print>Computing factorial(10): </print>
    <call name="fact"><arg value="10"/></call>
  </storyline>

  <library>
    <instance id="universe">
      <action name="fact">
        <params><param name="n"/></params>
        <pre test="$n &gt; 0"/>
        <fail>1</fail>
        <call name="_fact">
          <arg value="$n"/>
          <arg value="1"/>
          <arg value="1"/>
        </call>
      </action>
      <action name="_fact">
        <params><param name="n"/><param name="c"/><param name="f"/></params>
        <pre test="$n &gt;= $c"/>
        <call name="_fact">
          <arg value="$n"/>
          <arg value="$c + 1"/>
          <arg value="$c * $f"/>
        </call>
      </action>
      <action name="_fact">
        <params><param name="n"/><param name="c"/><param name="f"/></params>
        <value query="$f"/>
      </action>
    </instance>
  </library>
</scenario>
```

L'instance `universe` de ce scénario définit plusieurs actions qui vont récursivement calculer $n!$. La première action `fact` affiche 1 (cas `fail`) si $n \leq 0$ (`>` encode le symbole `>` en XML) et appelle sinon une action auxiliaire `_fact` avec des arguments initiaux. C'est cette action auxiliaire qui réalise le calcul.

Elle a deux définitions qui sont essayées consécutivement : la première pour laquelle les préconditions (définies dans des éléments `pre`) sont satisfaites voit ses effets exécutés par l'interpréteur.

```
[universe]
```

```
  Computing factorial(10): 3628800

>
```

### 2.2.3 Scénario de démonstration

L'archive du projet contient aussi un scénario de démonstration dans `samples/demo.xml`, qui correspond aux listings présentés plus tôt. Ce scénario utilise la bibliothèque d'instance par défaut par le biais d'une entité XML :

```
<!DOCTYPE scenario SYSTEM "scenario.dtd"[
  <!ENTITY default SYSTEM "library.xml">
]>
<scenario>
  <!-- ... -->
  <!-- Load the default library. -->
  &default;
```

Ce code équivaut à copier intégralement le contenu du fichier `library.xml` à l'emplacement de l'entité `&default;`. Cette bibliothèque est trouvée automatiquement par l'interpréteur dans le `CLASSPATH`.

## 2.3 Sémantique

Il n'y a pas de sémantique formelle d'écrite pour ce petit langage, mais cette section donne des éléments de compréhension.

L'interpréteur maintient deux arborescences distinctes : l'arboresence des éléments XML, qui sert aussi à sauvegarder une partie à n'importe quel moment, et l'arborescence des relations d'extension entre instances.

### 2.3.1 Portée d'une instance

Une instance voit dans sa *portée* tous les éléments XML `instance` et `reference`, ainsi que toute la portée de l'instance qu'elle étend. Dans l'exemple suivant, `universe` a dans sa portée l'instance `inventory`. L'instance `cop` a dans sa portée immédiate l'instance `badge` et la référence `gun` vers une instance `gun` définie ailleurs. Mais `cop` voit aussi `inventory` dans sa portée.

```
<instance id="universe">
  <instance id="inventory"/>
  <action name="speak">
    <print><name/> ne peut pas parler.</print>
  </action>
</instance>
```

```
<instance id="cop" extends="universe">
  <action name="speak">
    <print>Je n'ai rien à dire.</print>
  </action>
  <instance id="badge" />
  <reference id="gun"  />
</instance>
```

Les actions visibles sont elles aussi celles dans la portée de l'instance. Cependant un mécanisme de redéfinition est à l'œuvre : dans l'exemple, l'appel à `speak` depuis l'instance `cop` a l'effet d'afficher « Je n'ai rien à dire » sur la console, ignorant ainsi l'action `speak` définie dans `universe`.

### 2.3.2 Références

Une référence ajoute une instance fille symbolique à une instance donnée. Elle est vue comme une instance fille normale du point de vue de l'instance mère, si ce n'est pour un mécanisme d'*alias* :

```
<instance id="beach" extends="a_beach">
  <reference id="sea" alias="south"/>
  <!-- ... -->
```

La portée de l'instance `beach` contient ici virtuellement deux pointeurs vers `sea` : une référence avec le nom `sea`, mais aussi un alias `south`.

Le mécanisme de redéfinition n'est pas utilisé pour les références : toutes les références de la portée pointant vers la même instance sont sélectionnées, mais aussi toutes celles utilisant le même alias !

### 2.3.3 Actions

On a déjà mentionné le mécanisme de sélection des actions avec l'exemple du calcul d'une factorielle. Si plusieurs actions avec le même nom sont dans la portée *immédiate* d'une instance, alors elles sont essayées en séquence jusqu'à ce qu'une voie ses préconditions remplies. Si aucune ne réussit, alors le message d'erreur de la dernière est utilisé. En revanche les actions de même nom dans la portée étendue sont ignorées.

```
<action name="drop">
  <params><param name="o"/></params>
  <pre test="$o[ancestor::*/@id = 'object']"/>
  <pre test="inventory[* = $o]"/>
  <fail>Il n'y a pas <name query="$o" det="de"/> dans votre inventaire.</fail>
  <post test="* = $o"/>
  <post test="not(inventory[* = $o])"/>
  <print>Vous abandonnez <name query="$o"/>.</print>
```

```
    <move src="inventory/*[. = $o]"/>
  </action>
```

Chaque action, outre une séquence d'effets à exécuter, déclare optionnellement

- un ensemble de paramètres `param` dans un élément `params` (dont l'ordre sera utilisé lors des appels),
- des préconditions `pre`, dont la conjonction doit être vérifiée pour que l'action ait lieu ; ce sont des requêtes XPath qui sont évaluées par l'interpréteur dans le contexte courant,
- un effet d'affichage exécuté en cas d'échec des préconditions dans `fail` (c'est identique à un effet `print`),
- des postrelations `post` dont la conjonction qui devrait être vérifiée à l'issue des effets de l'action – sans quoi une exception devrait être levée.

#### 2.3.4 Effets

La sémantique des différents effets devrait être relativement explicite (la DTD est commentée en ce sens). Noter cependant que ces effets usent et abusent d'expressions XPath pour sélectionner des instances et références, et que leur sémantique travaille systématiquement sur des ensembles.

Par exemple,

```
<add dest="//*[ancestor::*/@id = 'a_house']"
     query="demultiplicator/*[ancestor::*/@id = 'a_bread']"/>
```

va ajouter à chaque instance d'une maison une référence vers chaque variété de pain disponible dans le démultiplicateur.

## 3 Considérations pratiques

L'archive fournit une configuration de `ant` et `ivy` qui devrait être facile d'emploi. Pour le début du développement, il suffit d'appeler

```
ant
```

pour lancer le programme `fr.ens_cachan.dptinfo.projet.Main` (qui est à écrire). Le choix du fichier de scénario d'entrée est `samples/hello.xml`, mais peut être changé dans `build.properties` ou en appelant `ant` avec l'option `-Dinput=/path/to/scenario.xml`.

### 3.1 Code fourni

Trois classes du paquet `fr.ens_cachan.dptinfo.projet.interpreter` vous sont au moins partiellement fournies, pour un développement utilisant XOM pour la navigation dans un arbre XML, et jaxen pour l'évaluation d'expressions XPath :

1. la classe `Interpreter`, où le code chargé de rechercher la DTD et la bibliothèque par défaut est fourni, et qui prévoit l'analyse du fichier XML pour XOM,

2. la classe `CompiledNodeFactory`, qui implémente le mécanisme prévu par XOM pour spécialiser des nœuds XML : les classes héritent de la classe `nu.xom.Element` et ajoutent leurs propres comportements. La fabrique fournie permet de créer des éléments en utilisant les classes spécialisées (à faire) adéquates,
3. la classe `InstanceNavigator` permet de redéfinir certains comportements de navigation des documents XML pour jaxen. Ici il y a essentiellement les signatures des méthodes qu'il faut redéfinir pour pouvoir utiliser jaxen comme évaluateur XPath sur notre double arborescence de nœuds XML et de relations d'extension entre instances.

La javadoc de ces fichiers peut vous inspirer sur comment nous avons implémenté ces classes...

Les choix de XOM et jaxen, et les classes fournies partiellement sont là pour vous aider à démarrer, mais vous êtes bien sûr libres de partir sur d'autres bases, d'implémenter votre propre DOM et votre propre évaluateur XPath, ou d'en choisir d'autres.