Objectif(s) : *héritage, polymorphisme*

L'objet de ce sujet est de construire une application semblable à une Histoire Dont Vous Êtes le Héros (HDVELH). L'application se présente comme un jeu sur la ligne de commande. Un texte est présenté au joueur, qui décrit un *événement*. Le texte se termine par une question, à laquelle le joueur doit répondre. La réponse du joueur détermine la suite des événements.

Le scénario d'une HDVELH peut être vu comme un *graphe orienté connexe* $G = (V, E)$, où chaque sommet $v \in V$ du graphe est un *événement*, et chaque arête $e \in E$ est une *transition* entre événements successifs. Pour que l'histoire soit cohérente ce graphe doit être *connexe* (i.e. ne doit contenir aucun sommet isolé), avoir un unique événement de départ, et au moins un événement de fin. Notez que G peut contenir des *cycles*.

Pour réaliser cette application nous allons procéder de la façon suivante :

1. préparation de la structure de données de graphe orienté
2. création du type d'événements de base `Event`
3. création du type `Scenario`
4. création d'un scénario de test
5. dérivation de nouveaux types d'événements par héritage du type de base

Remarque : Afin que le jeu puisse être étendu et muni plus tard d'une interface graphique utilisateur (GUI=IHM) plus sophistiquée, on confie la gestion de l'interface graphique de l'application à un objet gestionnaire de type `GUIManager` dont le code est fourni sur le dépôt git. Nous ne nous intéressons ici qu'à la version dont l'interface est la ligne de commande. (dit autrement : au lieu d'utiliser les méthodes membres de `System.out` comme par ex. `println`, on utilise exactement les mêmes méthodes mais depuis un objet de type `GUIManager`).

Exercice 1. Scénario

Construire un scénario de HDVELH avec l'outil en ligne twine (https://twinery.org/). Pour pouvoir utiliser le programme de test fourni votre scénario doit contenir au moins le sous-graphe de la Figure 1.

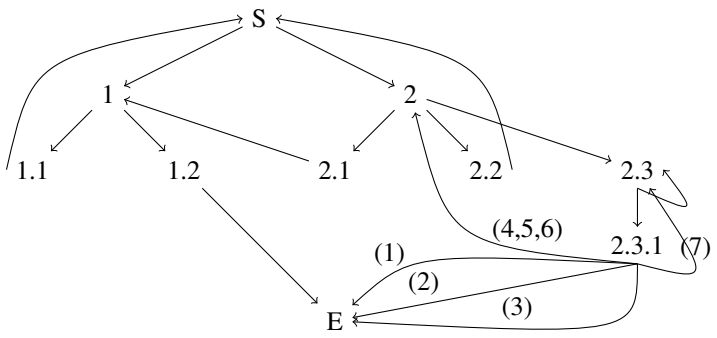

FIGURE 1 – Graphe scénario de base

Vous pourrez par exemple utiliser le début de scénario simpliste suivant (mais vous pouvez aussi créer le votre).

S. Vous marchez sur un chemin. Vous arrivez à une fourche.
Quel chemin choisissez-vous : (1) à gauche **[allez en 1.]** *ou (2) à droite* **[allez en 2.]** *?*

1. Après 1h de marche vous vous retrouvez sur un chemin de crête, bordé de part et d'autre par un précipice sans fond. Le passage se rétrécit. Bientôt face à vous se trouve une arche en pierre et une porte en bois, côte-à-côte. Le chemin derrière vous est en train de s'effondrer, vous ne pouvez pas reculer.
Par quelle voie continuez-vous : (1) l'arche **[allez en 1..1]***, ou (2) la porte en bois* **[allez en 1..2]** *?*

   1..1 Vous traversez une faille temporelle qui vous ramène au point de départ **[allez en S]**

   1..2 Derrière la porte se trouve un coffre.
   Vous l'ouvrez : **[allez en 1..2.1]**

2. Après avoir marché un moment le chemin vous mène au pied d'une montagne. Un torrent d'eau à flan de montagne alimente une rivière en contre-bas qui coule vers l'Est. Un raft est amarré à quelques pas.

Choisissez-vous (1) de suivre le chemin dans la montagne **[allez en 2.1]** *ou (2) de tenter votre chance sur les rapides de la rivière en empruntant le raft* **[allez en 2.2]***, ou (3) de passer sous le torrent d'eau* **[allez en 2.3]** *?*

1..2.1 E : bravo, vous avez trouvé la clé !

etc.

Continuez le scénario selon le graphe de la Figure 1.

Exercice 2. Diagramme de classes initial

Proposer un premier diagramme de classe qui modélise un scénario de HDVELH.

Exercice 3. Structure de graphe orienté

On définit un graphe orienté comme une structure chaînée de nœuds de type `NodeMultiple`, où chaque nœud est constitué des éléments suivants :

— une donnée (i.e. le contenu informatif du nœud), de type `Object`

— un ensemble d'arêtes sortantes représentées par autant de références vers des nœuds de type `NodeMultiple`.

Question 3.1. En vous inspirant de la structure de nœud utilisée pour une liste simplement chaînée, proposer une solution pour le type `NodeMultiple` de sorte que chaque objet de ce type puisse avoir un nombre `NODE_MAX_ARITY` maximum d'arêtes sortantes. Nous appelerons les nœuds accessibles par les arêtes sortantes les *nœuds fils* du nœud courant.

Question 3.2. Dans cette solution, comment sont modélisés les *arcs* du graphes ?

Question 3.3. Implémentez les méthodes membres de `NodeMultiple` (voir le patron fourni sur git), conformément à leurs javadoc.

Remarque : La méthode `toString()` est *non-récursive*, et ne retourne que le contenu informatif du nœud.

Exercice 4. Type d'événements de base `Event`

Un événement de base est une question de type QCM posée au joueur, à laquelle il est attendu que le joueur réponde par un nombre entier correspondant à la réponse choisie.

Un objet de type `Event` est vu comme un nœud de type `NodeMultiple`, muni d'un identifiant. Le contenu informatif (i.e. `data`) est une donnée textuelle qui correspond à l'exposé de la mise en situation du joueur.

Un événement est *exécutable* par un appel à sa méthode membre `run()`. Lors de son exécution, un événement de type `Event` se comporte de la façon suivante :

1. l'exposé de la situation est affiché par l'intermédiaire de l'IHM (i.e. GUI)
2. une réponse est attendue de la part du joueur (un message `PROMPT_ANSWER` est affiché pour demander cette réponse)
3. la réponse du joueur est interprétée (selon un protocole détaillé plus loin) de façon à calculer l'arête sortante qui conduit à l'événement suivant choisi
4. l'événement suivant est accédé.

Un événement est considéré comme *final* ssi il n'a aucun événement suivant, c'est-à-dire qu'il n'est associé à aucune arête sortante.

On considère également les contraintes suivantes :

— les réponses possibles sont présentées selon l'ordre naturel des entiers, **en commençant par 1**

— la réponse correspond à l'index du nœud suivant *majoré de 1* (le tableau des arêtes sortantes d'un nœud doit donc être *compacté à gauche*, c'est-à-dire qu'il ne doit contenir aucun trou et commencer à l'indice 0).

Question 4.4. D'après la description ci-dessus, proposer les définitions (éventuelles) de variables de classe et d'attributs publics et privés nécessaires.

Question 4.5. Écrire le constructeur de signature `public Event(GUIManager gui, String data)`, qui crée un objet dont l'IHM est `gui` et le contenu informatif est `data`.

Question 4.6. Écrire le constructeur par défaut de signature `public Event()`, qui crée un objet avec une IHM par défaut et un contenu informatif à `null`.

Question 4.7. Redéfinir la méthode `toString()` de sorte qu'elle produise une chaîne de la forme
`Event #[id] ([class]): [data].`

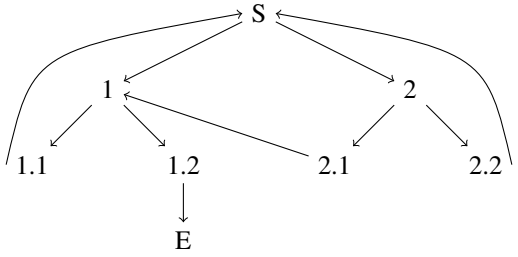

FIGURE 2 – Scénario 1

Question 4.8. Écrire les accesseurs (getters/setters) fourni sur le git.

Question 4.9. Écrire la méthode `public boolean isFinal()` qui retourne `true` ssi l'objet `this` est final selon la définition donnée plus haut.

Question 4.10. Écrire la méthode `public boolean isInRange(int index)` qui vérifie si l'`index` placé en paramètre est bien dans les bornes autorisées pour les nœuds fils de cet événement. On rappelle que l'ordre des nœuds suivants est supposé être l'ordre naturel, sans trou.

Question 4.11. Écrire la méthode `public int interpretAnswer()` qui choisit l'arête sortante vers le prochain événement en fonction de la réponse fournie par le joueur. Noter que «l'arête sortante vers le prochain événement » n'est pas le prochain événement lui-même, mais une valeur d'index dans la structure de données qui recueille les nœuds successeurs.

On prendra soin de traiter les cas d'erreurs : quels sont-ils ? On doit pouvoir recenser au moins 1 cas d'erreur fatale, et 3 cas d'erreurs non-fatales.

Question 4.12. Écrire la méthode `public Event run()` qui reproduit le comportement décrit en début d'exercice.

Exercice 5. Scénario de jeu `Scenario`

Question 5.13. Implémenter les constructeurs et accesseurs de la classe `Scenario`.

Question 5.14. implémenter la méthode `public String run()` qui exécute les événements du scénario, depuis l'événement de tête jusqu'à un événement final, selon le parcours choisi par le joueur.

Question 5.15. Tester le scénario 1 décrit par la figure 2, où S est le nœud de tête et E un nœud final. Vous pouvez utiliser une histoire qui corresponde au graphe décrit.

Exercice 6. Type d'événements à solution exacte `EventExactSolution`

Le type `EventExactSolution` est un type d'événements, qui dérive du type de base `Event`. Ce type permet de définir des événements où une réponse textuelle est attendue de la part du joueur. Par exemple, à la question «*Quelle est la valeur de π ?* », la réponse attendue doit être une sous-chaîne à gauche de la chaîne de caractères «3.14159» d'au moins 4 caractères.

Question 6.16. Proposer une définition de la classe `EventExactSolution` qui redéfinisse la méthode `interpretAnswer()` de la classe de base.

Question 6.17. Tester, en créant le scénario 2 qui modifie le scénario 1 précédent tel que l'illustre la figure 3, par l'ajout d'un événement à solution exacte.

Exercice 7. Type d'événements à solution aléatoire `EventRandomSolution`

Un événement à solution aléatoire est un événement qui ne pose aucune question au joueur. À la place, il détermine le chemin sortant à partir d'un jet de dé à n faces, et d'une partition de l'ensemble $D = [1 \dots n]$. Cette partition est un paramètre d'instance de l'événement. La partition se présente sous la forme d'un tableau des bornes maximales des parties. Par exemple, le tableau `[3,6,7]` représente la partition de D en 3 sous-ensembles $(1, 2, 3)$, $(4, 5, 6)$ et (7).

L'exécution d'un événement de ce type procède comme suit :

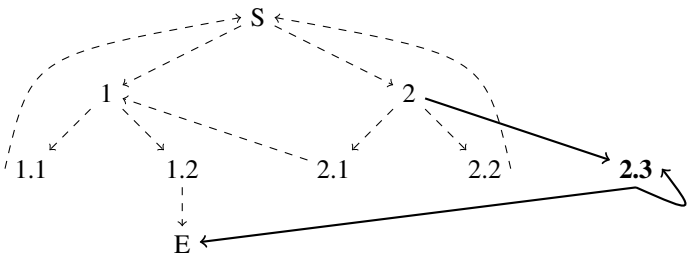

FIGURE 3 – Scénario 2

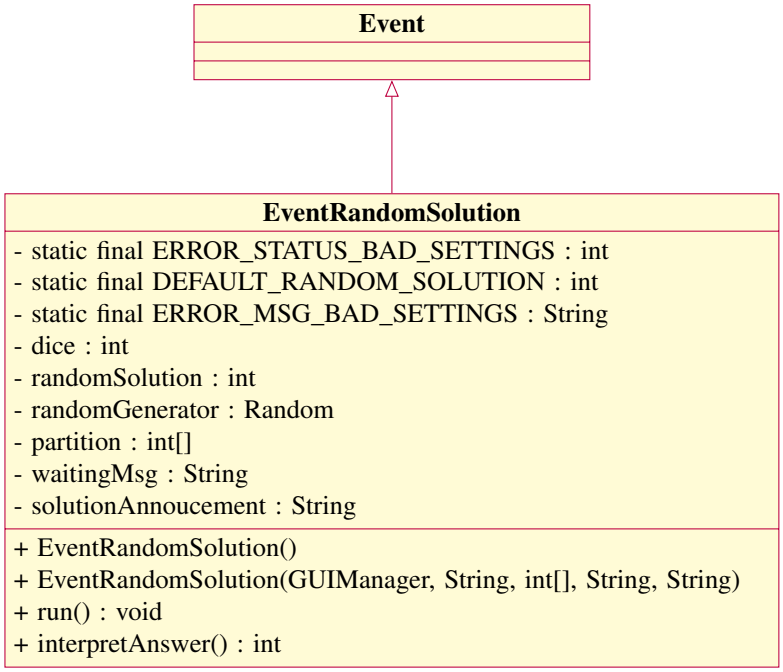

FIGURE 4 – Classe `EventRandomSolution`

1. le contenu informatif est exposé au joueur
2. un message d'attente est affiché qui annonce le jet du dé
3. un jet de dé a lieu, qui est interprété selon les règles énoncées ci-dessus
4. un message annonce le résultat du jet
5. le joueur est dirigé vers l'événement suivant qui a été déterminé.

On fournit en figure 4 l'extrait de diagramme de classe pour `EventRandomSolution` (on rappelle que '+' dénote un membre *public*, et '-' dénote un membre *privé*).

Question 7.18. Comment le nombre de faces du dé peut-il être établi sans qu'il soit un paramètre d'entrée du constructeur ? Pourquoi est-ce mieux ?

Question 7.19. Implémenter la classe `EventRandomSolution`. Penser à gérer les cas d'erreurs.

Question 7.20. Tester avec le scénario 3, décrit en figure 5.

Question 7.21. (subsidiaire 1, pour les plus avancés) Écrire les méthodes `public String NodeMultiple.toStringRecurs()` et `public String DirectedGraph.toStringRecurs()` qui produisent une représentation textuelle du graphe (depuis le nœud courant ou depuis la tête du graphe). Pour simplifier, on pourra par exemple adopter une représentation de la forme suivante (mais ce n'est pas la seule possible) :

```
S (head)
- 1
```

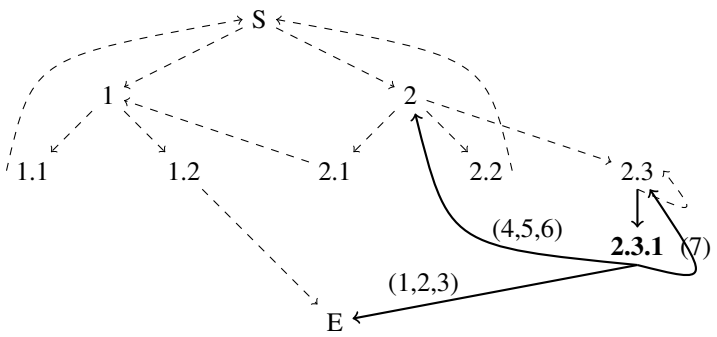

FIGURE 5 – Scénario 3

```
-- 1.1
-- 1.2
--- E (end)
- 2
-- 2.1
-- 2.2
```

Question 7.22. (subsidiaire 2) Proposer de nouveaux types d'événements. Que souhaiterions-nous pouvoir faire, pour faciliter la création de nouveaux événements avec des comportements différenciés ?