École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. (33)2-47-36-14-14
Fax (33)2-47-36-14-22
www.polytech.univ-tours.fr

Parcours des écoles d'Ingénieurs Polytech

Rapport de projet S4

Générateur de quêtes automatique pour la création d'un jeu

Auteur(s)

Thomas Couchoud
[thomas.couchoud@etu.univ-tours.fr]
Victor Coleau
[victor.coleau@etu.univ-tours.fr]

Encadrant(s)

Sébastien Aupetit
[aupetit@univ-tours.fr]
Richard Barruet
[richard.barruet@etu.univ-tours.fr]

Polytech Tours
Département DI

Version du 21 avril 2016

Table des matières

Introduction

L'objectif de ce projet est la création d'un logiciel permettant la génération automatique de quêtes pour des jeux vidéo. Ce projet a été fait en lien avec Mr Barruet, étudiant en 5ème année au Département Informatique de Polytech'Tours.

Suite à un premier entretien, nous avons défini le cahier des charges de notre programme. Tout d'abord, nous devions réaliser la base génératrice de quêtes, puis y ajouter une interface graphique simple. Le choix du langage étant libre, nous nous sommes orienté vers Java.

La principale motivation de ce projet, et plus généralement d'un générateur aléatoire, est de rallonger considérablement la durée de vie d'un jeu vidéo en implémentant un nombre quasi-infinie de quêtes.

Dans un premier temps, ce projet demande d'étudier l'univers des quêtes en général, ce qui fut principalement réalisé grâce à l'enquête effectuée à l'University of North Texas. Après cette étude, nous avons put nous lancer dans la réalisation de notre programme.

FIGURE 1 – Logo de notre application

Chapitre 1

L'univers des quêtes

1.1 Les MMORPG

1.1.1 Késako ?

Le terme MMORPG est l'acronyme de Massive Multiplayer Online Role-Playing Game (Jeu de Rôle en Ligne Massivement Multijoueur). Il s'agit d'un type de jeu vidéo uniquement en réseau où des centaines, voire milliers, de joueurs s'affrontent et coopèrent dans un riche univers sur serveur unique.

La majorité de ces jeux sont construit autour des thèmes classiques de fiction :

— médiéval-fantastique : Univers technologiquement moyennageux mais regorgeant de magie, de mythes, de légendes, de créatures fantastiques... (Seigneur des Anneaux)

— fantaisie : Même principe que la médiéval-fantaisie mais l'époque moyennageuse n'est pas recquise et la présence d'humains non-obligatoire. (World of Warcraft)

— science-fiction : Univers futuriste réaliste aux technologies avancées. Se déroule souvent dans un contexte inter-galactique aux espèces multiples. (Star Wars)

— etc...

FIGURE 1.1 – MMORPG - Eve Onlive

Au sein des MMORPG, on retrouve principalement trois mécaniques de jeu que sont :
— Le PvE (Player vs Environnement) : Le joueur est amené à collaborer avec d'autres afin d'affronter l'environnement peuplé de monstres contrôlés par le jeu.
— Le PvP (Player vs Player) : Escarmouches entre petits groupes de joueurs.
— Le RvR (Realm vs Realm) : Batailles à grandes échelles entre royaumes ou factions.

Le terme Jeu de Rôle vient du fait que chaque joueur incarne un unique personnage qu'il va devoir faire progresser à sa guise. Cette progression se traduit principalement par une montée de niveaux par accumulation de points d'expérience. Ceux-ci peuvent être acquis en éliminant des monstres ou en complétant des quêtes. Le gain de niveaux donne accès à de nouvelles compétences et équipements. Ces compétences ne s'obtiennent pas de façon linéaire mais se présentent souvent sous forme d'un arbre. Cela implique des choix qui spécialiseront et rendront toujours plus unique notre personnage.

De plus, chaque joueur fait partie d'un univers vivant dont il est un acteur. L'exemple majeur est le système de métier qui permet à chaque joueur de choisir un lot de compétences qui lui permettront de créer certains objets. Par exemple un forgeron utilisera des ressources acquises par un mineur pour créer de l'équipement.

1.1.2 Les quêtes

Comme dit précédemment, la résolution de quêtes (en solo ou à plusieurs) permet de recevoir de l'expérience et ainsi faire progresser le joueur. Celles-ci jouent un rôle central car c'est un moyen efficace de progresser rapidement. Une quête consiste majoritairement à rendre un service à un PNJ (Personnage non joué) qui nous récompensera.

Ces quêtes sont le plus souvent scénarisées et permettent ainsi de développer l'histoire et l'univers du jeu. Ainsi nous pouvons avoir à faire à plusieurs types de quêtes allant de la simple aide aux villageois (éliminer des loups qui menacent la population)au compte épique et héroïque pour sauver le monde.

Généralement, les quêtes en cours du joueur sont répertoriées dans un journal de quêtes comme dans la figure 1.2. Celui-ci fourni toutes les informations dont le joueur a besoin :
— Une description pour mettre en situation.
— Le nom du PNJ qui nous a confiée cette quête.
— La prochaine étape à réaliser.
— La difficulté.
— La récompense (parfois)

FIGURE 1.2 – Journal de quête

1.2 L'intérêt des quêtes aléatoires

Les quêtes scénarisées ont cependant un inconvénient majeur. Ayant une histoire à raconter, elles doivent être créées à la main par un développeur. C'est à dire qu'un humain doit définir une suite d'objectifs cohérents entre-eux permettant de raconter une histoire au fil des actions du joueur. Il arrivera donc un moment où cette histoire prendra fin. Cela induit que les quêtes de ce genre sont en nombre fini, les développeurs ne créant pas indéfiniment de nouveau contenu. Étant la principale motivation du joueur, leur fin provoquera une perte d'intérêt de celui-ci pour le jeu.

Afin d'éviter cela, il serait intéressant de proposer un moyen de créer des quêtes à l'infini. La méthode communément proposée est une génération aléatoire. Cependant, nous verrons par la suite qu'il y a de nombreuses limites à cette notion d'aléatoire.

1.3 Étude d'une enquête sur les quêtes de MMORPG

La première ressource à notre disposition fut l'étude "A Prototype Quest Generator Based on a Structural Analysis of Quests fromFour MMORPGs" réalisée par Jonathan Doranand et Ian Parberry de l'University of North Texas.

Cette étude porte sur l'analyse de plus de 3000 quêtes de quatre célèbres MMORPGs (Eve On-line, World of Warcraft, Everquest, Vanguard : Saga of Heroes). L'objectif était d'étudier la composition précise de ces quêtes afin d'en dégager une méthode de construction.

Premièrement, du point de vue du joueur, les quêtes ne sont que la succession d'actions mineures à réaliser dans un ordre précis. La première tache serait donc d'établir la liste de

ces actions élémentaires. Puis, d'attribuer un ou plusieurs supports pour chaque action. Par exemple, l'action "Aller quelque part" nécessite un complément de lieu : l'endroit où il faut se rendre. De même, l'action "Tuer" demande une victime, etc...

	Action	Pre-condition	Post-condition
1.	ε	None.	None.
2.	capture	Somebody is there.	They are your prisoner.
3.	damage	Somebody or something is there.	It is more damaged.
4.	defend	Somebody or something is there	Attempts to damage it have failed.
5.	escort	Somebody is there	They will now accompany you.
6.	exchange	Somebody is there, they and you have something.	You have theirs and they have yours.
7.	experiment	Something is there.	Perhaps you have learned what it is for.
8.	explore	None.	Wander around at random.
9.	gather	Something is there.	You have it.
10.	give	Somebody is there, you have something.	They have it, and you don't.
11.	goto	You know where to go and how to get there.	You are there.
12.	kill	Somebody is there.	They're dead.
13.	listen	Somebody is there.	You have some of their information.
14.	read	Something is there.	You have information from it.
15.	repair	Something is there.	It is less damaged.
16.	report	Somebody is there.	They have information that you have.
17.	spy	Somebody or something is there.	You have information about it.
18.	stealth	Somebody is there.	Sneak up on them.
19.	take	Somebody is there, they have something.	You have it and they don't.
20.	use	There is something there.	It has affected characters or environment.

Table 4: Atomic actions.

FIGURE 1.3 – Table des actions élémentaires

Deuxièmement, nous pouvons remarquer que dans la plupart des jeux étudiés, les mêmes schémas de quêtes reviennent assez souvent (appelés Stratégies). Il s'agira donc d'extraire ces archétypes de quêtes demandant la résolution d'objectifs plus ou moins similaires. On pourra de plus les rassembler en catégorie (appelée Motivation) afin par la suite de pouvoir modifier leur fréquence d'apparition.

Motivation	Description
Knowledge	Information known to a character
Comfort	Physical comfort
Reputation	How others perceive a character
Serenity	Peace of mind
Protection	Security against threats
Conquest	Desire to prevail over enemies
Wealth	Economic power
Ability	Character skills
Equipment	Usable assets

Table 1: NPC motivations.

Motivation	Quests	Percent
Knowledge	138	18.3%
Comfort	12	1.6%
Reputation	49	6.5%
Serenity	103	13.7%
Protection	137	18.2%
Conquest	152	20.2%
Wealth	15	2.0%
Ability	8	1.1%
Equipment	139	18.5%

Table 2: Distribution of observed NPC motivations.

FIGURE 1.4 – Motivations et leur répartition

Troisièmement, il faudra attribuer à chaque schéma précédemment trouvé un squelette initial d'actions élémentaires. Cette suite d'actions sera donc un base solide pour chaque stratégie puisque, comme dit pus haut, chacune d'entre elles représente un archétype de quêtes (même points de départ et d'arrivé).

Motivation	Strategy	Sequence of Actions
Knowledge	Deliver item for study	<get> <goto> give
	Spy	<spy>
	Interview NPC	<goto> listen <goto> report
	Use an item in the field	<get> <goto> use <goto> <give>
Comfort	Obtain luxuries	<get> <goto> <give>
	Kill pests	<goto> damage <goto> report
Reputation	Obtain rare items	<get> <goto> <give>
	Kill enemies	<goto> <kill> <goto> report
	Visit a dangerous place	<goto> <goto> report
Serenity	Revenge, Justice	<goto> damage
	Capture Criminal(1)	<get> <goto> use <goto> <give>
	Capture Criminal(2)	<get> <goto> use capture <goto> <give>
	Check on NPC(1)	<goto> listen <goto> report
	Check on NPC(2)	<goto> take <goto> give
	Recover lost/stolen item	<get> <goto> <give>
	Rescue captured NPC	<goto> damage escort <goto> report
Protection	Attack threatening entities	<goto> damage <goto> report
	Treat or repair (1)	<get> <goto> use
	Treat or repair (2)	<goto> repair
	Create Diversion	<get> <goto> use
	Create Diversion	<goto> damage
	Assemble fortification	<goto> repair
	Guard Entity	<goto> defend
Conquest	Attack enemy	<goto> damage
	Steal stuff	<goto> <steal> <goto> give
Wealth	Gather raw materials	<goto> <get>
	Steal valuables for resale	<goto> <steal>
	Make valuables for resale	repair
Ability	Assemble tool for new skill	repair use
	Obtain training materials	<get> use
	Use existing tools	use
	Practice combat	damage
	Practice skill	use
	Research a skill(1)	<get> use
	Research a skill(2)	<get> experiment
Equipment	Assemble	repair
	Deliver supplies	<get> <goto> <give>
	Steal supplies	<steal>
	Trade for supplies	<goto> exchange

Table 3: Strategies for each of the NPC motivations from Table 1 using actions from Table 4.

FIGURE 1.5 – Squelette de chaque stratégie

Grâce aux trois points précédents, nous pouvons à présent commencer à créer des quêtes. Cependant, leur nombre et variété seront limités au nombre de stratégies précédemment définies. Afin de remédier à cela, nous pouvons "casser" chaque action du squelette en une nouvelle suite d'actions précisant la manière dont celle-ci doit être réalisée. Par exemple, se rendre dans un lieu peut d'abord nécessiter d'apprendre où se situe ce lieu. Ainsi, l'action [Aller à X] deviendra [Apprendre où est X / Aller à X].

De même, acquérir un objet peut demander de le voler, il faudra donc d'abord trouver la personne possédant cet objet avant de le voler.

La quatrième étape sera donc de définir quelles actions élémentaires pourront se décomposer (toute décomposition n'étant pas pertinente) et en quelle nouvelle série d'actions. Dans la figure

suivante chaque action entre chevrons signifie que celle-ci peut se décomposer elle-même à son tour. En effet, en ce qui concerne les autres, il serait illogique de les scinder. Par exemple un GOTO peut se casser en [LEARN, GOTO]. Dans ce cas, au moment où nous devrons effectuer le second GOTO, nous aurons déjà appris où se trouve notre objectif. La seule possibilité logique est donc d'y aller, et non d'essayer de faire des actions supplémentaires pour y arriver.

	Rule	Explanation
1.	<subquest> ::= <goto>	Subquest could be just to go someplace.
2.	<subquest> ::= <goto> <QUEST> goto	Go perform a quest and return.
3.	<goto> ::= ε	You are already there.
4.	<goto> ::= explore	Just wander around and look.
5.	<goto> ::= <learn> goto	Find out where to go and go there.
6.	<learn> ::= ε	You already know it.
7.	<learn> ::= <goto> <subquest> listen	Go someplace, perform subquest, get info from NPC.
8.	<learn> ::= <goto> <get> read	Go someplace, get something, and read what is written on it.
9.	<learn> ::= <get> <subquest> give listen	Get something, perform subquest, give to NPC in return for info.
10.	<get> ::= ε	You already have it.
11.	<get> ::= <steal>	Steal it from somebody.
12.	<get> ::= <goto> gather	Go someplace and pick something up that's lying around there.
13.	<get> ::= <goto> <get> <goto> <subquest> exchange	Go someplace, get something, do a subquest for somebody, return and exchange.
14.	<steal> ::= <goto> stealth take	Go someplace, sneak up on somebody, and take something.
15.	<steal> ::= <goto> <kill> take	Go someplace, kill somebody and take something.
16.	<spy> ::= <goto> spy <goto> report	Go someplace, spy on somebody, return and report.
17.	<capture> ::= <get> <goto> capture	Get something, go someplace and use it to capture somebody.
18.	<kill> ::= <goto> kill	Go someplace and kill somebody.

Table 5: Action rules in BNF.

FIGURE 1.6 – Décomposition d'action élémentaires

De plus, certaines d'entre-elles pourront avoir plusieurs décompositions possibles afin là encore d'augmenter la diversité des quêtes générées.

Ici on voit déjà un premier problème, la décomposition infinie. En effet, les nouvelles actions crées lors d'une décomposition peuvent elles aussi être par la suite décomposées pouvant alors provoquer des quêtes infinies. Par exemple, dans le cas d'un GET se scindant en [GOTO, GET, GOTO, EXCHANGE], le second GET pourra réinvoquer la même décomposition, recréant encore un nouveau GET, lui-aussi susceptible de se casser. Nous aurions donc un empilement infini de sous-quêtes identiques. Ceci n'a clairement aucun intérêt, tant pour le joueur (quête infaisable) que pour nous (crash du programme). Afin d'éviter cela, il faudra donc intégrer au générateur de quêtes une limite.

Grâce aux points vus précédemment, nous sommes dorénavant capables de produire sur papier des quêtes de plusieurs niveaux. Celles-ci possèdent un squelette logique bien que vide (les actions ne portent encore sur aucun objet et ne sont qu'abstraites). Le schéma suivant représente ce que nous appelons un arbre de quête. On y voit le squelette initial de la stratégie (ligne la plus haute) puis les décompositions successives de certains éléments.

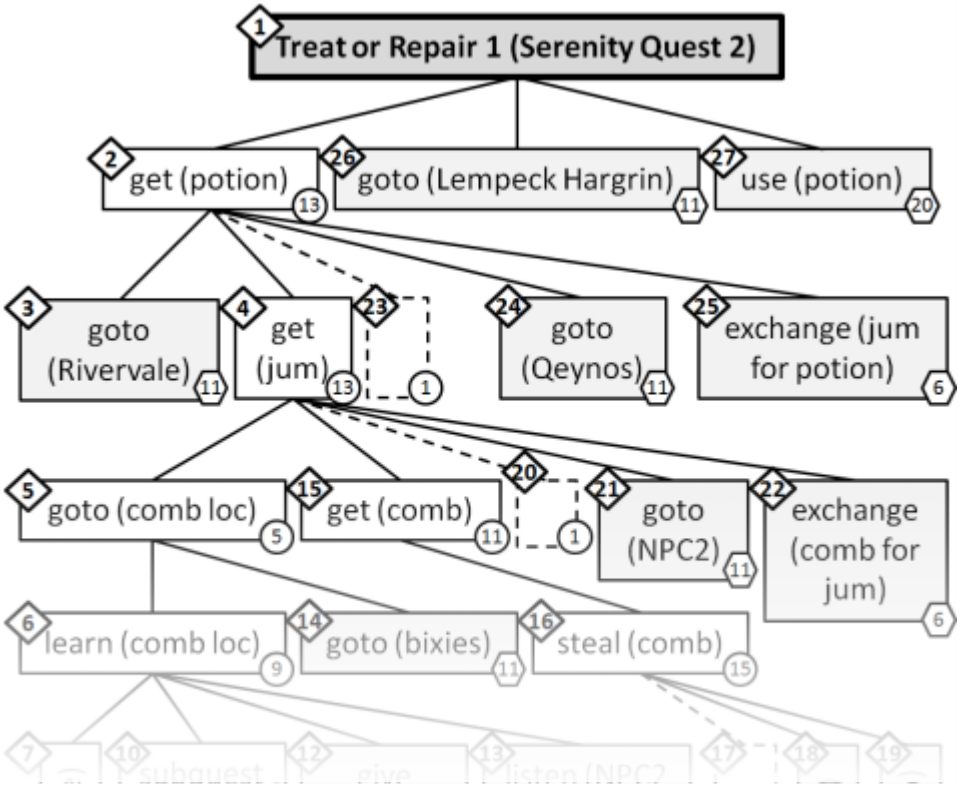

Figure 3: Analysis of "Cure for Lempeck Hargrin" from Everquest.

FIGURE 1.7 – Arbre de quête

Chapitre 2

Le programme

Après avoir étudié la structure générale des quêtes, nous nous sommes lancés dans la création de notre propre générateur de quêtes aléatoires. Suite à une petite discussion avec M. Barruet, nous avons défini une sorte de cahier des charges : le programme devra ainsi générer la quête et l'afficher à l'utilisateur de manière simple.

La partie qui va suivre va donc détailler la structure et le fonctionnement du programme. Celle-ci détaillera chaque bloc de fonctionnement du programme ainsi que l'acheminement qui a permis d'aboutir à sa forme actuelle. Ayant été plus rapides que prévu, certaines parties ont été ajoutées afin d'avoir un programme plus riche. Ainsi les parties qui vont suivre sont :

— La base du programme : La partie qui génère les différentes actions à réaliser.
— Les objectifs : La caractérisation d'une action.
— L'héritage : La logique de passation des objectifs entre quêtes.
— L'interface graphique.
— L'API.

Néanmoins, avant de nous lancer dans les explications, il est nécessaire que nous fixions deux mots de vocabulaire qui reviendront souvent. Ce que nous appelons une "Action" représente une action précise à réaliser. Par exemple "Aller échanger xxx contre yyy avec Mr zzz". Enfin, une "Quête" est tout simplement une liste ordonnée de différentes Actions.

2.1 La base du programme

Concentrons-nous sur l'une des parties majeures du programme. Celle-ci consiste en la génération des actions de base à réaliser. En d'autres termes, nous réalisons le squelette de notre quête, ce qui ressemble à cela :

```
SPY
GOTO
   EXPLORE
SPY
GOTO
   LEARN
      GOTO
      QUEST
         GOTO
         STEAL
            STEALTH
            TAKE
         GOTO
         GIVE
      LISTEN
   GOTO
```

2.1.1 Réalisation d'une maquette

Étant la partie où tout à débuté, nous ne nous sommes pas lancés directement dans la partie de codage. Tout commença par la réalisation d'une "maquette" sur papier de la structure de notre programme. Ainsi nous avons défini les objets de base dont nous aurons besoin :

- Une énumération ActionType définissant les actions de base réalisables (GOTO, USE, etc.).
- Une énumération Motivations définissant les différentes motivations des PNJ.
- Une énumération Strategies définissant les différentes stratégies des PNJ.
- Une interface ActionExecutor qui sera à implémenter dans une classe qui définira comment une quête doit être générée.
- Un objet Action définissant une action à réaliser.
- Un objet Quest définissant une quête (ou sous-quête), étant principalement une liste d'actions.

Tout d'abord, parlons rapidement de l'interface ActionExecutor. La partie qui nous intéresse est la méthode generateQuest(...). Cette dernière va être définie de plusieurs manières en fonction des différentes combinaisons d'actions possibles. Nous avons des implémentations de cette interface pour chaque stratégie (représentant le début d'une quête), ainsi que pour chaque action. Cependant, les actions peuvent avoir plusieurs implémentations disponibles. Par exemple un GET peut se casser en [STEAL] ou en [GOTO, GATHER].

Ensuite parlons plus en détail des énumérations. Concernant Motivations, il n'y a rien de plus à ajouter car celle-ci n'a aucune valeur particulière et ne sert qu'à associer une motivation à une stratégie. Concernant Strategies, chacun de ses éléments est associée à un ActionExecutor. Ainsi, lorsque nous voulons démarrer une quête, nous choisissons une stratégie et lui demandons de générer la liste d'actions lui correspondant. Enfin, parlons d'ActionType. Cette énumération est primordiale et intervient assez souvent. Nous verrons certaines de ses méthodes dans les parties qui vont suivre mais pour le moment l'essentiel est de savoir qu'elle contient une liste d'ActionExecutor pour chacune de ses valeurs. Comme dit plus haut, certaines actions peuvent se scinder de plusieurs manières, c'est ici que sont stockées les différentes possibilités.

Enfin, vous trouverez ci-dessous un schéma montrant la façon dont sont imbriqués nos objets Quest et Action afin d'éviter un maximum les confusions par la suite.

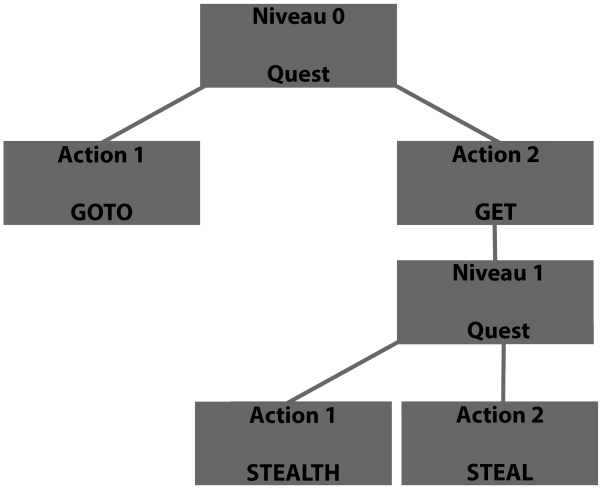

FIGURE 2.1 – Schéma d'une quête

2.1.2 Le codage primitif

Maintenant que nous avons la structure globale du programme, nous allons expliquer comment s'opère la génération du squelette de la quête. Voici un petit diagramme qui pourra vous aider à suivre les différentes étapes.

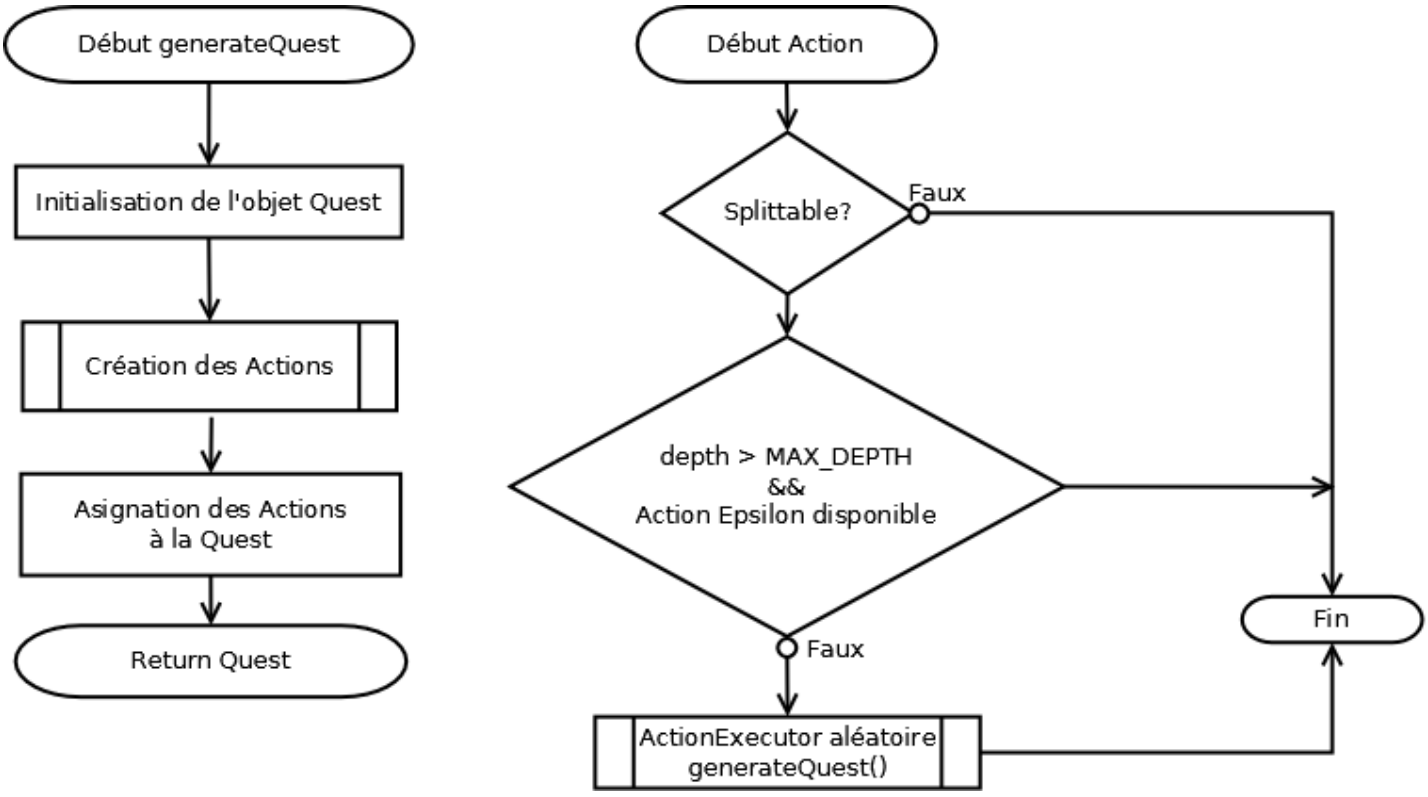

FIGURE 2.2 – Diagramme de la génération d'une quête

Tout va commencer par le choix d'une stratégie de départ. Cela peut être fait de trois manières :

— La stratégie est définie par l'utilisateur.
— L'utilisateur choisit une motivation puis la stratégie sera choisie aléatoirement parmi celles de la motivation.
— Choix aléatoire parmi toutes les stratégies disponibles.

Dans notre cas nous la choisissons de manière totalement aléatoire.

La stratégie ayant un ActionExecutor associé, nous allons pouvoir commencer à créer nos objets. C'est à partir de ce moment là qu'un processus qui va se répéter plusieurs fois démarre :

1. Appel de la fonction generateQuest(Action parent, int depth, Optional<> objectives) de l'ActionExecutor.

   Les paramètres se composent d'une Action parente (null si aucune). Si nous reprenons la figure 2.1, la Quest du niveau 0 n'aura pas de parent, tandis que la Quest du niveau 1 aura GET comme action parente. Ensuite nous avons la profondeur de la quête, sur le schéma cela correspond au niveau. Cela nous servira plus tard dans la génération. Enfin nous avons un Optional qui représente les objectifs de l'action. Nous détaillerons cela dans la section 2.2.

   Cette fonction réalise les actions suivantes :

   (a) Instanciation d'un objet Quest.
   (b) Instanciation des Actions. A ce moment là, l'action va prendre la décision ou non de se scinder en une autre Quest. Nous détaillerons cette partie juste après. L'essentiel est de savoir que cela revient à choisir un ActionExecutor et à démarrer le même processus que nous sommes actuellement en train de réaliser.
   (c) Ajout des actions à la Quest.
   (d) Retour de l'objet Quest.
2. A ce moment, nous disposons d'un objet Quest étant celui du niveau 0. Il ne nous reste plus qu'à le manipuler de la manière que nous souhaitons.

Voici maintenant le détail concernant le scindage des différentes Actions. Il est à noter que chaque action contient un attribut définissant si celle-ci est cassable ou non. En effet, parfois, même s'il existe des suites possibles à la quête, il serai illogique de le faire. Prenons, par exemple, le type GOTO. Dans le cas d'une sous-quête, il est probable que nous ayons quelque chose du type : "PNJ1 vous demande d'aller voir PNJ2, puis, revenir lui raconter ce qu'il s'est passé". Le GOTO peut se casser en [LEARN, GOTO]. Cependant, lors du retour, il serait illogique de casser le GOTO puisque nous savons déjà où est PNJ1. Ce qui suit n'est donc déclenché que si nous voulons tenter de scinder l'Action.

— Si notre type d'action contient des suites possibles, nous continuons. Par exemple, le type READ n'a pas de suites possibles. Il n'est donc pas nécessaire d'essayer de continuer.
— Si nous avons à disposition l'action epsilon et que la profondeur actuelle est supérieure à celle maximale, nous retournons l'action epsilon.
— Nous choisissons un ActionExecutor aléatoirement parmi ceux disponibles, puis appelons la méthode generateQuest.

Au final, cela nous permet d'obtenir une structure comme ci-dessous :

```
Go to {0}
Listen {0}
```

```
Go to {0}
   Learn where is {0}
   Go to {0}
      Learn where is {0}
         Get {0} from {1}
         Perfom subquest
            Go to {0}
               Explore {0}
         Give {0} to {1}
         Listen {0}
      Go to {0}
         Explore {0}
Report to {0}
```

2.2 Notion d'objectifs

La partie précédente nous a permit de créer le squelette de notre quête. Cependant nous avons besoin de définir plus précisément nos actions. Avoir un GOTO c'est bien, mais GOTO où ? Nous allons donc, dans cette partie, expliquer comment nous créons les objectifs de nos actions. Cependant, nous ne rentrerons que très peu dans le programme. Seule la structure sera expliquée plus en détails.

2.2.1 Définition des types d'objectifs

La première chose dont nous avons eu besoin est un moyen d'identifier les objectifs. Dans plus de la moitié des cas, cela n'a pas trop d'importance puisque l'objectif est unique (GOTO, LISTEN, ...). Cependant, certains requièrent deux, voire trois objectifs (GET, EXCHANGE). Afin de pouvoir identifier quel paramètre joue quel rôle, nous avons créé une nouvelle énumération ObjectiveType.

Maintenant que nous disposons d'une manière de pouvoir identifier les objectifs, nous devons définir lesquels seront utilisés pour chaque ActionType. Afin de simplifier les choses, chaque ActionType disposant d'un seul objectif, aura comme ObjectiveType, OBJECTIVE. Pour ceux nécessitant plusieurs paramètres, nous avons rédigé un fichier texte répertoriant chaque ActionType avec leur différents ObjectiveType.

Le fichier en question est disponible ici. Il se présente sous cette forme :

```
ActionType:
   - ObjectiveType1 (type de valeur qu'est supposé prendre cet objectif)
   - ObjectiveType2 (^)
```

2.2.2 Création d'un fichier XML

Maintenant que nous sommes capables de reconnaître chaque objectif, il est temps de leur assigner des valeurs. Pour cela nous avons créé une liste des différentes valeurs possibles, cor-

respondant notamment aux différents objets, pnj et zones que pourrait avoir un jeu.

Nous avions commencé par répartir ces différentes valeurs dans plusieurs fichiers texte où chacun représentait une catégorie (zone, pnj, objet). Cela était satisfaisant pour compléter rapidement les objectifs. Cependant après plusieurs essais, nous nous sommes rendu compte que cela ne suffirait pas.

Il nous manquait une certaine hiérarchie qui nous permettrait de mieux catégoriser nos éléments. Par exemple, le fait qu'un pnj puisse être une bête ou un humain pose des problèmes pour nos objectifs. "Allez parler à Vache" n'est pas très convainquant. De plus les fichiers texte ne sont pas très appropriés à la modification de la structure des ces éléments. Rédiger un fichier texte par catégorie pourrait être envisageable mais très lourd. Se rajoute à cela, la difficulté de pouvoir définir aisément, les relations entre chaque catégories.

Afin de remédier à tout cela, nous nous sommes orientés vers une structure XML. Ainsi chaque élément sera défini de la manière suivante :

```
<element value="vache" />
```

Et chaque catégorie comme ceci :

```
<category value="humans">...</category>
```

Grâce à cette structure, nous pouvons répondre simplement à notre problème de hiérarchisation de nos éléments. Reprenons l'exemple des PNJ :

```
<category value="pnj">
   <category value="humans">
      <element value="human1" />
   </category>
   <category value="beasts">
      <element value="beast1" />
   </category>
</category>
```

Non seulement nous résolvons notre problème, mais, nous gagnons aussi en simplicité. Il est clairement plus facile de devoir gérer un fichier XML que d'avoir un fichier texte par catégorie. De plus, la lisibilité est meilleure de cette manière.

Après plusieurs essais, nous avons abouti à ce fichier XML qui est utilisé pour la génération des quêtes. Vous y retrouverez de nombreuses catégories ainsi crées, afin de différencier au maximum les éléments pour l'héritage entre Actions que nous verrons dans la section 2.3.

2.2.3 Interprétation du fichier XML

Notre fichier XML étant créé, il est nécessaire de l'importer dans notre programme. Pour cela, nous utilisons la librairie sax de java, afin de lire notre fichier et le transformer en objets. Ceux-ci sont au nombre de deux : XMLStringObjectiveElement et XMLStringObjectiveCategory.

XMLStringObjectiveElement représente un élément. Il contient son chemin dans le ficher XML et sa valeur.

XMLStringObjectiveCategory représente, lui, une catégorie. Il dispose d'une liste d'XMLStringObjectiveElement, ainsi qu'une liste d'XMLStringObjectiveCategory contenant les potentielles sous-catégories.

Ainsi, nous obtenons, en JAVA, une structure très similaire à notre fichier XML.

2.2.4 Utilisation de ces données

Il est maintenant temps d'utiliser nos valeurs pour les affecter à nos objectifs. Pour cela, rien de plus simple : il nous suffit de sélectionner un XMLStringObjectiveElement et de l'affecter à un objectif.

Lors de nos premiers essais, nous ne stockions que la valeur de l'élément dans l'objectif. Cependant, il a été préférable de stocker l'XMLStringObjectiveElement à la place, pour l'héritage des objectifs que nous verrons dans la section 2.3. Cela permet de pouvoir identifier la provenance de l'élément.

Vous trouverez ci-dessous un exemple de génération de quête. Celle-ci est très courte mais ce qui nous intéresse, ce sont les éléments d'objectif. Dans cette image nous avons marqués les valeurs d'objectif de manière différente :

```
[(chemin dans le fichier XML) valeur]
```

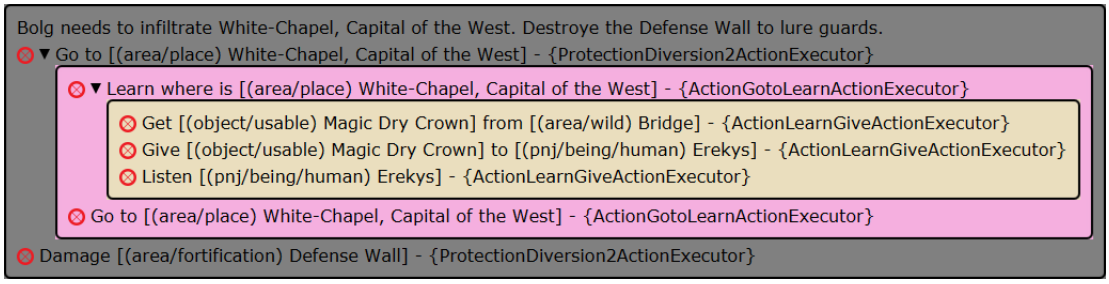

FIGURE 2.3 – Exemple de génération (avec héritage expliqué en 2.3)

Comme vous pouvez vous en douter, l'essentiel est de placer un élément qui a du sens dans chaque objectif. Avoir un nombre important de catégories joue donc un rôle majeur. Plus nous pouvons différencier nos objets, notamment grâce aux catégories, mieux sera la génération. Le choix de ces dernières, pour chaque élément, est définit dans les ActionExecutors. Pour cela nous utilisons les chemins :

— /pnj/human/ : nous renverra un élément aléatoire de la catégorie voulue.
— /pnj/human/* : nous renverra un élément pioché aléatoire dans la catégorie et toutes ses sous-catégories.

2.3 Héritage et Logique

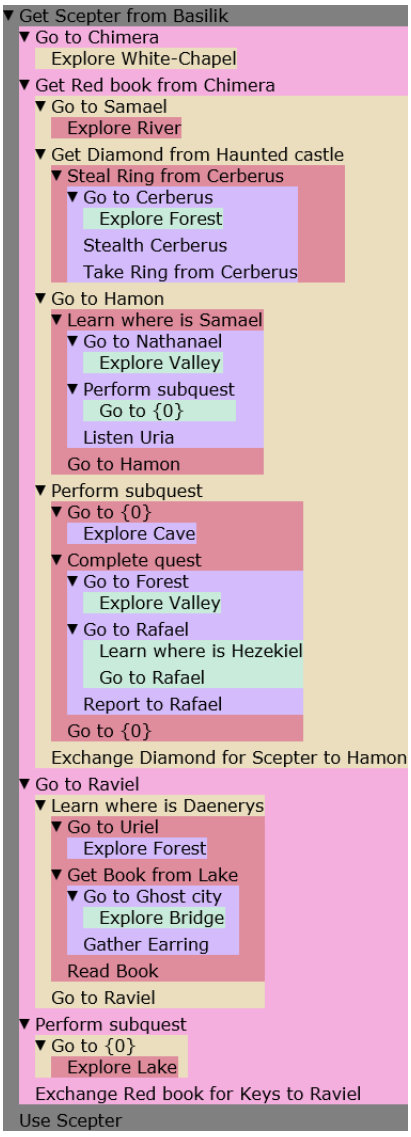

FIGURE 2.4 – Exemple de génération sans logique

Ci-contre, un exemple de génération avant l'implémentation de l'héritage. Chaque ligne (Action) a un sens par rapport à elle-même. Et de la même manière chaque bloc de couleur (Quest) est lui aussi cohérent. Prenons par exemple la première Quest de couleur marron/jaune, nous devons récupérer des diamants, aller voir Hamon, pour enfin lui échanger ce que nous avons collecté contre un sceptre. Cependant, le lien entre une Action est sa sous-quête est inexistant. Si nous reprenons notre exemple et tentons de voir d'où il provient, nous remarquons que "Obtenir un livre auprès de Chimera" s'est d'une manière transformé en "Obtenir (par échange) un sceptre auprès de Hamon". L'héritage a donc été implémenté afin de corriger ces problèmes.

La construction de cette logique a été faite pas à pas en réalisant de nombreuses générations. Lorsqu'un problème survenait, nous tentions de comprendre d'où provenait l'incohérence pour tenter de la corriger. De ce fait, la structure est beaucoup moins organisée et est majoritairement sous forme d'ajout de méthodes dans des classes existantes.

La première étape fut d'avoir un lien entre une Action et sa sous-quête. Dans la section 2.1.2, nous avions expliqué le fonctionnement de la fonction generateQuest. Ses paramètres comportait un Optional<>. Celui-ci est en réalité un Optional<HashMap<ObjectiveType, XMLStringObjectiveElement» et représente les objectifs de l'Action parent (si présente). Grâce à celui-ci, nous somme en mesure de corriger le problème évoqué en exemple au début de cette section. En effet, au lieu de remplir nos objectifs de manière aléatoire, nous nous aidons de ceux du parent.

Détaillons un peu plus deux lignes de l'exemple : "Get Red book from Chimera" et "Exchange Diamond for Scepter to Hamnon". La première ligne est fixée, c'est notre parent. Cependant pour l'échange nous connaissons déjà la majorité des valeurs. Nous allons générer un nouvel objet pour l'échange mais le reste sera récupéré du parent. Notre seconde ligne deviendra donc "Exchange {Diamonds} for [Red book] to [Chimera]". Nous avons placé les objectifs du parent entre crochets et ceux générés à cette étape entre accolades.

Si nous regardons à l'intérieur de l'ActionExecutor scindant un GET en [GOTO, GET, GOTO, SUBQUEST, EXCHANGE], nous remarquons lors de l'initialisation de l'Action d'échange :

```
new Action(..., buildObjective(objectives, new ObjectiveHelper(OBJ_GET, OBJ_GET),
    ...);
```

Deux parties méritent une explication. La première est le nouvel objet ObjectiveHelper. Plusieurs constructeurs sont disponibles mais nous parlerons simplement du plus complet, les autres étant simplement des raccourcis lorsque que nous voulons des valeurs par défaut. Trois

paramètres sont nécessaires : un ObjectiveType source, un de destination ainsi qu'un XML-StringObjectiveElement. L'intérêt principal de cet objet est son unique méthode getValue qui prend en paramètre un Optional<...> représentant nos objectifs. Celle-ci va se charger de nous renvoyer une valeur d'objectif, correspondant à nos critères :

- Si l'ObjectiveType source est présent dans la liste d'objectifs de notre parent (donc dans l'Optional<...>), nous renverrons la valeur lui étant associée.
- Si le cas précédent n'est pas rempli, nous renverrons la valeur par défaut.

Cet objet parait très simpliste et ne fait que très peu de chose. Cependant il rend la lecture du code beaucoup plus visible et simplifie la rédaction.

Deuxièmement, la méthode buildObjective qui est présente dans l'interface ActionExecutor est définie dans ce fichier grâce au mot-clé default apparu avec JAVA 8. La méthode prend deux paramètres, un Optional<...> d'objectifs (le même qui est passé dans notre méthode generateQuest) ainsi qu'une liste d'ObjectiveHelper. Son but est de construire notre nouvel Optional<...> d'objectifs qui sera utilisé par une Action fille. Le principe est simple, pour chaque ObjectiveHelper, nous associons la valeur que retourne sa méthode getValue avec l'ObjectiveType destination que nous avons renseigné.

Nous allons maintenant illustrer l'aisance de lecture que peut apporter ces deux méthodes dans la création de nos actions. Voici la ligne exacte qui crée l'action d'échange de notre exemple :

```
Action actionExchange = new Action(quest, this.getClass(), depth,
    ActionType.EXCHANGE, buildObjective(objectives, new ObjectiveHelper(OBJ_GET,
    OBJ_GET), new ObjectiveHelper(OBJ_GIVE, objectiveObject), new
    ObjectiveHelper(PNJ, pnjExchange)), false);
```

Certes, de premier abord, cela parait tout de même assez massif, mais vu le travail réalisé cela reste raisonnable :

- quest : Représente la quête parent.
- this.getClass() : Est la classe de l'ActionExecutor ayant crée cette action et est utilisé simplement à des fins de debug. Vous avez d'ailleurs pu apercevoir cette valeur dans la figure 2.3.
- depth : Définissant la profondeur de l'Action dans notre arbre.
- ActionType.EXCHANGE : Renseignant le type de l'action.
- buildObjective(...) : Créé la liste des objectifs :
  - new OobjectiveHelper(OBJ_GET, OBJ_GET) Va définir l'OBJ_GET du parent comme étant l'OBJ_GET de notre action.
  - new ObjectiveHelper(OBJ_GIVE, objectiveObject) Définira une variable locale objectiveObject comme étant l'OBJ_GIVE de notre échange.
  - new ObejctiveHelper(PNJ, pnjExchange) Agira de la même manière que le précédent.
- false : Indique que l'action n'est pas cassable.

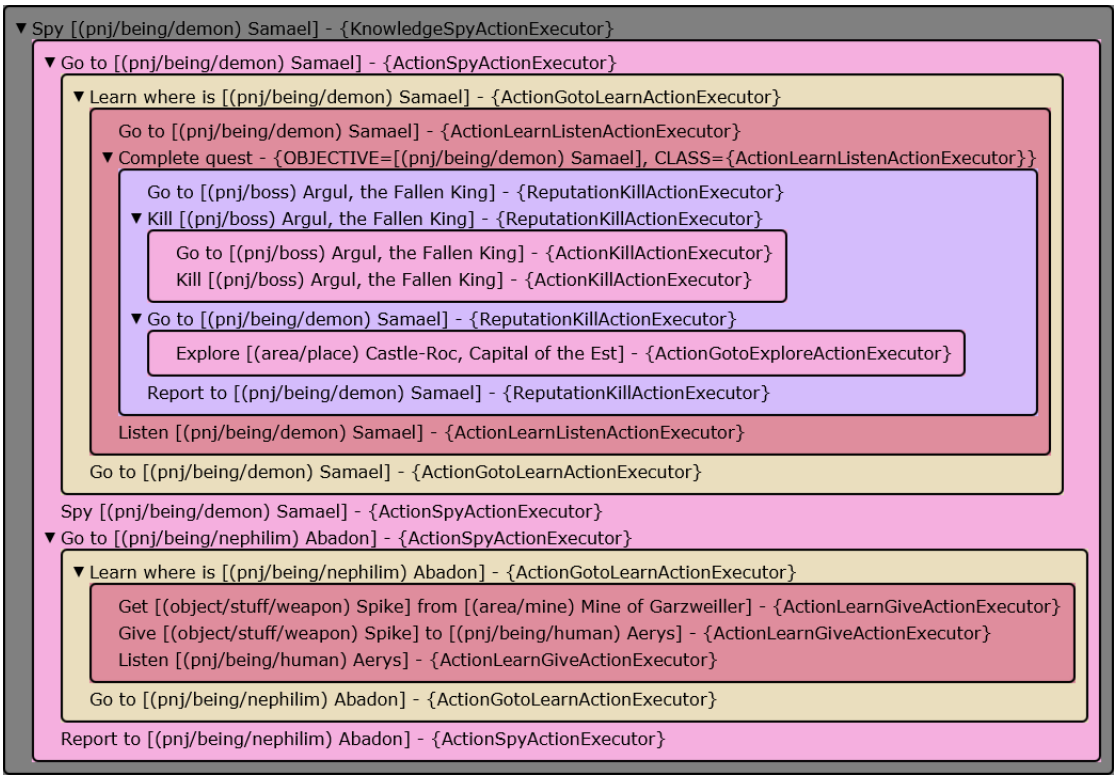

FIGURE 2.5 – Exemple de génération avec héritage

La figure précédente nous montre une génération qui a été réalisée avec cette notion d'héritage. Certes, nous remarquons que notre problème majeur a été résolu, mais un bon nombre d'incohérences plus mineures sont toujours présentes.La plus part d'entre elles se traitent en faisant des essais et leur résolution consiste principalement à changer la manière dont une action est créée. Cela implique d'autres changements dans le code :

— Changement de la structure de certaines décompositions. Dans ce cas, nous avons jugé que ce qui était présent dans notre PDF de référence n'avait aucun sens avec le reste de la quête. Cela se traduit soit par une modification de l'attribut de cassage d'une action soit par la modification de la séquence des types d'action.

— Suppression de certaines décompositions. Prenons l'exemple d'un KILL, celui-ci pouvait se casser en [GOTO, KILL]. Cependant, dans chaque ActionExecutor manant à un KILL, un GOTO était déjà présent. Du fait de l'inutilité de cette décomposition, nous l'avons supprimée.

— Un petit ajout qui a été fait assez tard dans le développement, nous a aussi permit d'étendre notre logique. Il s'agit d'avoir la possibilité d'obtenir le chemin XML d'un élément. Il est assez difficile d'expliquer les différents usages que nous avons fait de celui-ci. Essayons plutôt de l'entrevoir par un exemple. Un EXPLORE n'aura pas la même valeur si l'objectif est un PNJ humain ou un objet. En effet, si l'on cherche un PNJ, on considère qu'il peut se trouver n'importe où, cependant un objet perdu sera lui considéré comme étant seulement dans une zone sauvage. La zone à explorer n'est donc pas la même suivant la provenance de notre objectif.

— Apparition d'une fonction isActionAllowed dans l'interface ActionExecutor. Son implémentation par défaut nous renvoi true. Cependant il nous est possible de la redéfinir

au besoin. Nous nous en servons pour filtrer certains découpages selon les objectifs. Par exemple un GOTO peut se scinder en [EXPLORE]. Nous autorisons cela que si notre objectif est un PNJ humain, une bête ou bien une zone précise. Cela n'aurait aucun intérêt d'explorer une zone pour trouver une forêt par exemple.

— De la même manière, nous avons voulu filtrer quels ActionExecutors pouvaient être utilisés lors de la génération de SUBQUEST. En effet, ce type d'action génère une nouvelle quête à l'intérieur de celle actuelle. Cependant, cela implique qu'un PNJ nous a demandé de réaliser celle-ci contre un service. Par exemple, on doit apprendre où est PNJ1 en demandant à PNJ2. Mais ce dernier demande à ce qu'on lui fasse sa quête pour qu'il nous aide. Il serait donc stupide qu'il nous demande d'aller nous entrainer au combat. Nous avons donc décidé d'implémenter un système (simple boolean défini à l'avance), qui définira si une stratégie est éligible pour la génération de notre sous-quête.

2.4 Interface graphique

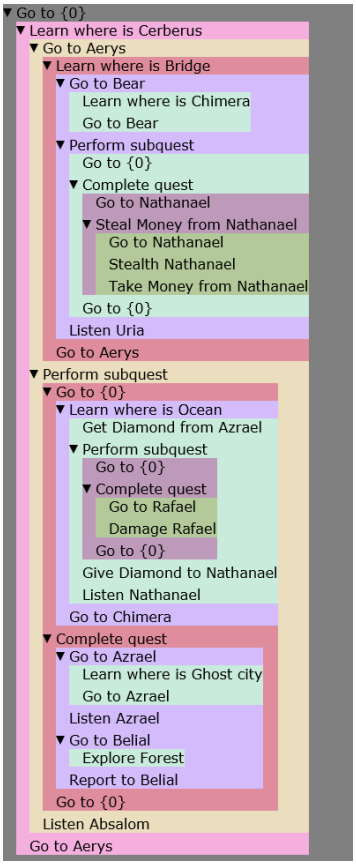

FIGURE 2.6 – Interface graphique à ses débuts

La réalisation de ce projet n'impliquait pas seulement la génération de la quête, mais aussi un moyen de visualiser celle-ci. La figure ci-contre vous montre une ancienne version de l'interface. Le but est de pouvoir représenter simplement nos actions d'une manière que chaque joueur puisse comprendre. Nous nous appuyons sur deux éléments, des boites représentant les quêtes et des lignes pour les actions.

Ainsi, chaque bloc de couleur représente une quête. Chaque sous-quête est indentée par rapport à la précédente, ce qui nous donne l'impression de profondeur. La couleur était au début présente uniquement pour nous en tant que développeur afin de nous repérer plus simplement entre les blocs. Cependant nous avons décidé de la laisser car nous l'avons jugée très utile pour les utilisateurs finaux.

Vous pouvez remarquer sur la gauche de chaque action disposant d'une sous-quête une petite flèche pointant vers le bas. En cliquant dessus nous avons la possibilité de "fermer" un bloc. Encore une fois cela a pour but de pouvoir simplifier la lecture. En effet, lorsque nous avons des blocs très grands, une fois que nous avons effectué les actions, nous pouvons nous contenter de la phrase de description générale. Par exemple "Aller chercher le livre rouge" est suffisant, pas besoin de savoir "Aller chercher des diamants, aller voir Hamon, lui échanger les diamants contre le livre rouge" si nous l'avons déjà fait.

Du coté un peu plus technique, nous avons réalisé notre interface avec la libraire JavaFX. Deux nouveaux composants ont été créés afin de simplifier le code : ActionNode et QuestNode.

Comme l'on peut s'en douter, ActionNode sera l'affichage d'une Action (donc une ligne). Celle-ci est formée d'une VBox (Pane où les éléments sont rangés verticalement) contenant deux parties :

— Une HBox (comme VBox mais horizontalement) comportant deux éléments, une image représentant la flèche de fermeture/ouverture du bloc (ajoutée uniquement si l'action contient une sous-quête) ainsi que la description de l'action.
— Un QuestNode pour la sous-quête (uniquement si présente).

L'image de la flèche capture les clics de souris et modifie l'affichage du bloc de sous-quête grâce à la méthode setManaged(boolean). Celle-ci est similaire à setVisible(boolean) mais comporte un avantage supplémentaire. L'élément n'est plus pris en compte dans le calcul des dimensions. En effet, si l'on change juste la visibilité, nous verrions juste un gros trou dans notre bloc. En revanche, setManaged va agir comme si l'on avait retiré le bloc et tout ce qui était en dessous de celui-ci va prendre sa place.

D'avantage d'images de l'interface seront présentes dans le chapitre 3 afin d'éviter de surcharger le PDF. La dernière version de l'interface vous sera montrée, ce qui a plus d'intérêt.

Chapitre 3

Ajouts supplémentaires au projet

A partir de ce point, tout ce qui va suivre sont des ajouts qui n'étaient pas prévus dans notre cahier des charges. Cependant, ayant eu plus de temps que prévu, et quelques idées supplémentaires, nous avons pris la décision d'implémenter de nouvelles fonctionnalités.

3.1 Affichage plus développé

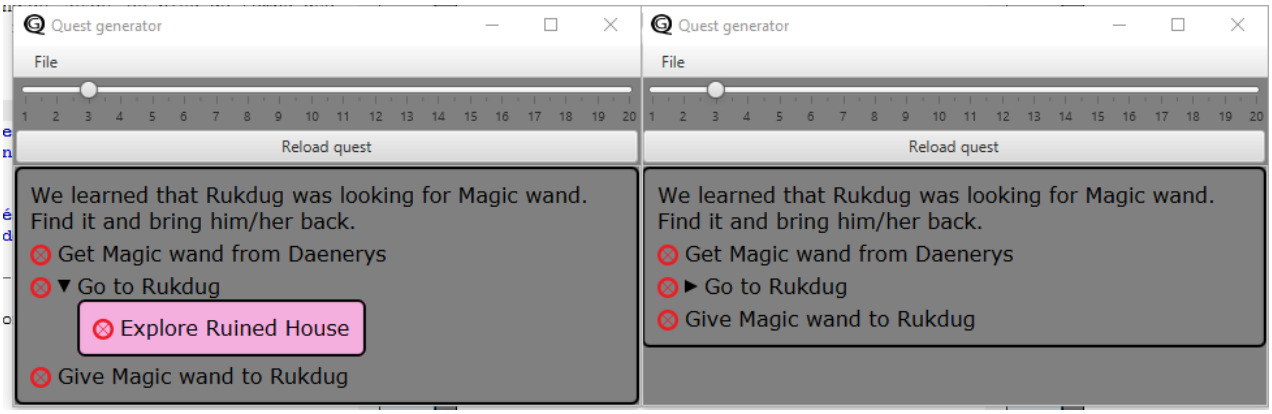

FIGURE 3.1 – Dernière version de l'interface (bloc ouvert à gauche, fermé à droite)

L'affichage de nos quêtes est dorénavant un peu plus développé. Cela consiste principalement en la modification de la présentation de nos anciens éléments. Cependant, on constate aussi l'apparition de nouveaux éléments :

- Correction de bugs mineurs, comme des espaces entre les éléments qui étaient mal gérés lorsque la flèche de fermeture n'était pas présente.
- Adaptation des blocs à la fenêtre. Lors de la version précédente, lorsque le texte dépassait de la fenêtre (horizontalement), celui-ci était ignoré et nous devions étirer la fenêtre pour pouvoir voir la suite du texte. Dorénavant, le texte se réparti sur plusieurs lignes afin d'éviter ce problème.
- Ajout d'un slider permettant de choisir la profondeur maximum des futures générations.
- Ajout d'un bouton permettant de générer une nouvelle quête.
- Ajout d'une barre de menu regroupant différentes options que nous allons détailler dans les sections suivantes comprenant l'exportation des quêtes, le changement de modes d'affichage et l'ouverture de la fenêtre d'évènements.
- Ajout de raccourcis clavier pour chacune de ces actions.
- Ajout de phrases de description pour le début de chaque quête ayant pour but d'immerger un peu plus le joueur en donnant un sens à ce qu'il va faire.

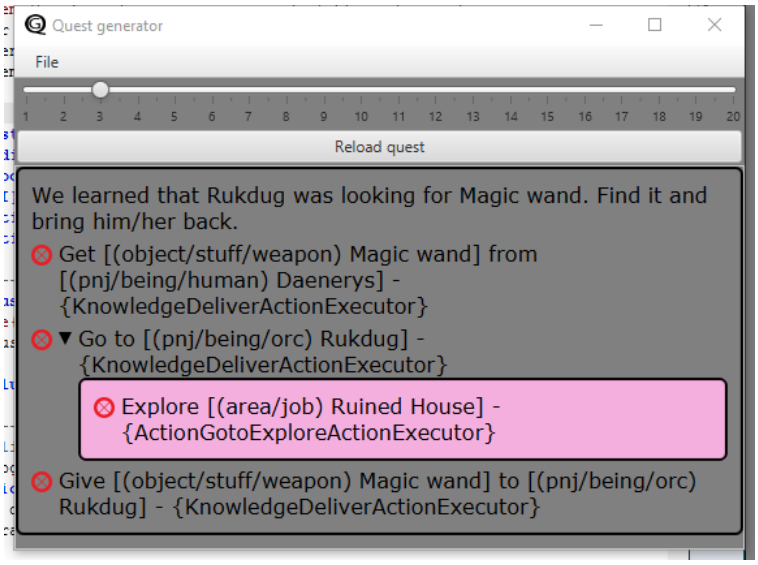

FIGURE 3.2 – Mode debug et illustration de la répartition du texte sur plusieurs lignes

L'un des deux modes de vue possible est le mode debug. Celui-ci n'est destiné qu'aux personnes voulant travailler avec le générateur de quêtes. Il nous permet de connaitre chaque valeur d'objectif plus en détail et d'obtenir la classe ayant générée le bloc dans lequel se trouve notre action.

3.2 Gestion de l'avancement des quêtes

Le second mode de vue est lié à un nouveau concept que nous avons intégré. Avoir une quête générée est déjà une bonne chose, mais il pourrait être intéressant de savoir quelles quêtes nous avons réalisées et lesquelles sont à faire. Nous avions tout d'abord pensé à faire une coche à coté de chaque action que le joueur pourrait valider lui-même. Cependant, nous avons jugé plus intéressant de développer un système qui gèrera lui-même tout l'avancement de la quête. Ayant eu au même moment l'idée de développer une API (section 3.5), réaliser cette notion avait un très gros avantage.

Nous ne rentrerons encore une fois pas des les détails mais allons expliquer le fonctionnement global. Chaque Action et Quest dispose d'un attribut boolean done définissant si celle-ci à été complétée ou non. Le principe de l'API sera de modifier cette valeur. En ce qui concerne l'interface, cette valeur est représentée par le rond (rouge dans notre cas) au début de chaque ligne. Un rond rouge signifie que l'action n'a pas été réalisée et un rond vert que l'action est terminée.

Concernant les actions avec sous-quête, le principe de fonctionnement est différent. En effet l'attribut done n'a aucun sens car l'action dépend de la sous-quête qu'elle contient. Notre action sera considérée comme étant réalisée lorsque tous les éléments enfants auront été complétés. Dans la figure 3.3 le "Go to Aerys" sera donc complété que lorsque tous les éléments du bloc rose auront été fait.

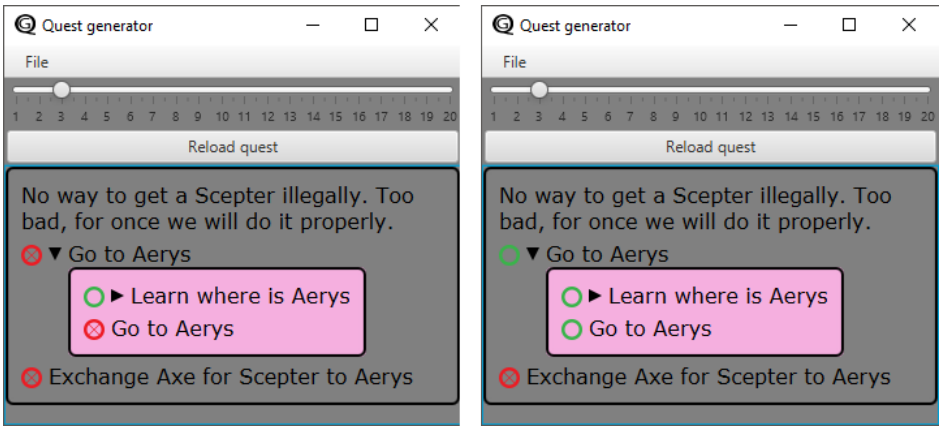

FIGURE 3.3 – Progression des actions

En jouant un peu avec l'interface et ces nouvelles valeurs nous avons mis un nouveau mode de vue qui est le mode de présentation. Celui va nous permettre de voir la quête au fur et à mesure que nous la réalisons. Cela permet une immersion plus réaliste dans la quête.

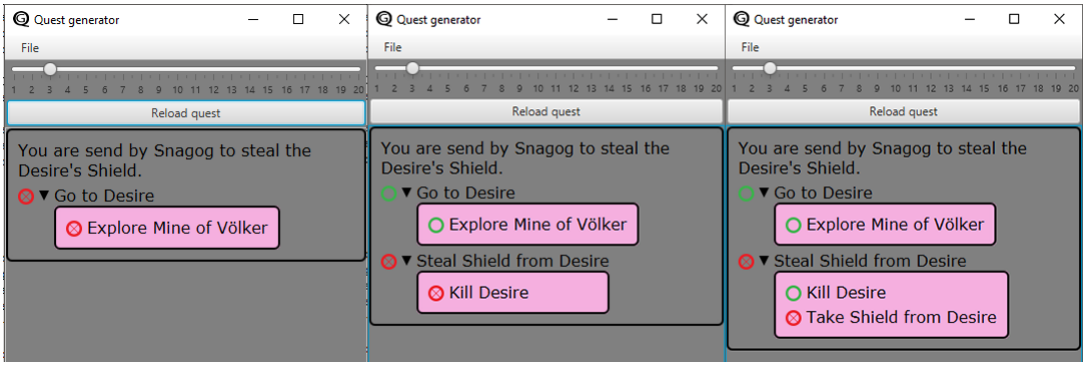

FIGURE 3.4 – Mode de présentation

3.3 Modification de constantes de manière externe

Beaucoup de parties du programme reposent sur des constantes que nous avons définies nous-même. Cependant, il est possible qu'une personne utilisant ce programme veuille se servir de ses propres valeurs, imposant alors de modifier le code source et de recompiler. La solution étant assez lourde, nous avons décidé d'autoriser la lecture de fichiers externes. Nous procédons de la manière suivante : si un fichier portant le même nom que celui de notre code est disponible à l'emplacement de l'exécution du programme, nous l'utilisons, sinon nous prenons notre fichier par défaut.

De cette manière nous pouvons modifier :

— strings.xml Étant le ficher XML contenant toutes les valeurs d'objectif. Une structure doit être respectée pour assurer un bon fonctionnement du programme.

— MotivationsPorbabilities.properties En réalité, lors du choix aléatoire d'une motivation, celles-ci sont pondérées. Il est possible de modifier les valeurs de pondération dans ce fichier.
— QuestSentences.properties Vous retrouverez ici toutes les phrases mettant en situation les quêtes.
— ActionSentences.properties De la même manière, ce fichier contient toutes les descriptions d'actions.

3.4 Exportation des générations

En plus de la génération des quêtes, nous avons ajouté différents moyens d'exporter les résultats. L'un d'entre eux, que vous avez déjà pu voir, est de prendre une capture d'écran de l'interface. Ce procédé est pratique pour présenter ce résultat simplement. Cependant, il n'est pas du tout réutilisable par un autre programme. Afin de remédier à cela, nous avons ajouté d'autres types d'exportations.

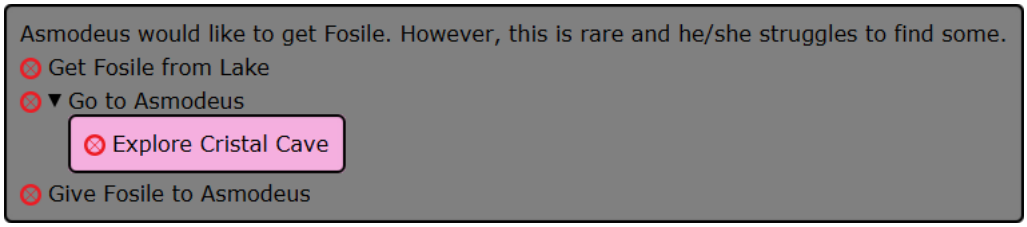

FIGURE 3.5 – Exportation PNG

Raw

```
[Asmodeus would like to get Fosile. However, this is rare and he/she struggles to
    find some.]
Get Fosile from Lake
Go to Asmodeus
   Explore Cristal Cave
Give Fosile to Asmodeus
```

Ce type d'exportation est celui qui se rapproche le plus de notre fichier image. En effet, nous avons ce qu'affiche l'interface mais au format texte. Chaque phrase de description se trouve entre crochets, et les actions sont écrites directement. La profondeur est représentée par l'indentation. Ce format d'exportation peut permettre d'enregistrer simplement les quêtes et de pouvoir les lire soit en ouvrant le ficher soit avec un programme externe.

Actions

```
GET
GOTO
```

```
   EXPLORE
GIVE
```

Le format Actions n'offre que très peu d'informations. Grâce à celui-ci, nous récupérons le squelette de notre quête. Cela peut être intéressant si un programme veut implémenter sa propre logique des objectifs mais n'a pour nous aucune utilité.

XML

```
<?xml version="1.0" ?>
<quest description="Asmodeus would like to get Fosile. However, this is rare and
    he/she struggles to find some.">
    <action type="GET">
        <objectives>
            <objective type="OBJ_GET" value="Fosile" path="object/rare"></objective>
            <objective type="LOC_OBJECTIVE" value="Lake" path="area/wild"></objective>
            <objective type="CLASS" value="ReputationObtainActionExecutor"
                path="class"></objective>
        </objectives>
    </action>
    <action type="GOTO">
        <objectives>
            <objective type="OBJECTIVE" value="Asmodeus"
                path="pnj/being/angel/fallen"></objective>
            <objective type="CLASS" value="ReputationObtainActionExecutor"
                path="class"></objective>
        </objectives>
        <quest>
            <action type="EXPLORE">
                <objectives>
                    <objective type="OBJECTIVE" value="Cristal Cave"
                        path="area/dangerous"></objective>
                    <objective type="CLASS" value="ActionGotoExploreActionExecutor"
                        path="class"></objective>
                </objectives>
            </action>
        </quest>
    </action>
    <action type="GIVE">
        <objectives>
            <objective type="OBJ_GIVE" value="Fosile" path="object/rare"></objective>
            <objective type="LOC_OBJECTIVE" value="Asmodeus"
                path="pnj/being/angel/fallen"></objective>
            <objective type="CLASS" value="ReputationObtainActionExecutor"
                path="class"></objective>
        </objectives>
    </action>
</quest>
```

Le format XML apparait tout de suite comme un format très imposant. Nous n'avons pas une vue très simple de la chose et aucune phrase d'action n'est présente. Cependant, il est probablement le fichier le plus intéressant de tous. En effet, celui-ci contient tous les éléments nécessaires pour reconstituer la quête. Nous avons les objets Quest avec leur liste d'Action ainsi que les objectifs de ces derniers.

Grâce à ce format, un programme peut aisément charger la quête sans avoir à la générer lui même. Cela constitue donc un point majeur pour l'intégration de notre réalisation dans un projet plus général.

3.5 API

L'utilisation de notre projet dans un jeu était quelque chose qui nous intéressait fortement. En effet, cela donnait un peu plus de sens à ce que nous faisions. Avoir un format de fichier lisible par un programme externe était déjà un moyen fonctionnel de le faire. Cependant ayant eu encore du temps pour pousser plus loin notre réalisation, nous nous somme penché sur un moyen de le rendre utilisable comme une librairie.

Notre programme étant déjà très avancé, génération des quêtes mais aussi gestion de l'avancement de celle-ci, nous avons jugé utile de rendre possible à un jeu, de pouvoir se "connecter" à ce processus. Ainsi, l'utilisateur aurait juste à dire "Génère moi une quête", "Le joueur à tué xxx" à notre librairie et celle-ci répondra en conséquences.

3.5.1 Gestion des événements présents dans le jeu

La première partie est donc de pouvoir comprendre les messages que nous envoi un jeu. Celui-ci peut nous transmettre une quantité énorme d'informations. Cependant nous ne voulons pas traiter ces données nous-même. En effet, chaque jeu dispose d'un moyen différent de les utiliser. Nous nous somme rabattu sur une implémentation simple. Chaque Quest implémentera l'interface GameListener et permet au jeu de signaler les évènements qui nous intéressent.

Ces derniers correspondent en réalité à nos ActionType. Par exemple, dès qu'un échange sera effectué, le jeu pourra le signaler à la quête et cette dernière se chargera de vérifier si celui-ci est correct. Si tel est le cas, nous validons l'action en cours.

Chaque méthode de l'interface est différente car consiste en la passation des différents objectifs de l'action en paramètres. Par exemple, exchangeEvent aura trois paramètres (objet donné, objet reçu, avec qui est fait l'échange), tandis que gotoEvent en a un seul (où nous sommes allés). Chaque méthode retourne un boolean représentant la complétion ou non d'une action.

De plus, nous avons ajouté une méthode getActionToDo() dans la classe Quest. Celle-ci a pour but de nous renvoyer l'Action que nous devons actuellement réaliser pour avancer dans notre quête. Cela peut être utile si un jeu désire afficher uniquement l'action en cours sans avoir à montrer tout l'arbre de quête.

3.5.2 Notification de progression des quêtes

Pouvoir envoyer des informations de ce qu'il se passe dans le jeu à notre système de quête est un point fort. Cependant il est aussi utile que le jeu puisse recevoir des informations sur sa quête. Pour cela, l'application utilisant notre libraire pourra implémenter l'interface QuestListener. Il ne restera plus qu'à renseigner cette classe auprès de notre quête grâce à la méthode addQuestListener(QuestListener). Une fois que cela est fait, la classe pourra recevoir deux informations :

— actionDone(Action) Informe lorsque qu'une action est réalisée.
— questDone(Quest) Nous indique quand une quête est terminée, c'est à dire que tout ses enfants sont complétés.

Conclusion

Ce projet nous a permis de découvrir plus en détail l'univers des quêtes. Etant nous même des joueurs de jeux vidéos, nous avions déjà quelques connaissances sur le sujet. Cependant l'aspect technique de la génération de quêtes nous était totalement inconnu. Le fait d'avoir eu l'opportunité de travailler sur un générateur aléatoire de quêtes fut une occasion pour nous de découvrir une des mécaniques de jeux vidéos.

Tout notre travail s'est basé sur une enquête réalisée à l'University of North Texas étudiant la structure des quêtes au sein des MMORPG. A partir de là, nous avons tenté de réaliser un programme générant une quête aléatoire. L'enjeu ici était de pouvoir en générer un nombre infini contrairement à certains jeux qui n'en ont qu'un nombre fini.

A la fin de nos heures de projet, nous considérons que le programme réalisé répond aux critères fixés au début du projet. Celui-ci est capable de générer une quête qui a un sens. De plus, comme convenu, une interface est disponible afin de visualiser celle-ci.

Ayant avancé rapidement, nous avons aussi eu l'occasion d'implémenter des fonctionnalités supplémentaires. Cela passe notamment par une gestion de la progression des quêtes et par l'ajout d'une API. Cette dernière permet de concrétiser notre projet puisqu'elle le rend utilisable par n'importe quel jeu.

Nous avons trouvé que la partie la plus difficile à mettre en place fut la logique des objectifs. En effet, toute la génération du squelette fut rapide. Il aura fallu de nombreuses générations et beaucoup de réflexion afin d'obtenir des quêtes logiques.

Cependant, tout cela est loin d'être parfait. La logique peut encore être poussée plus loin, mais faute de temps nous l'avons laissée à un stade qui reste tout de même correcte. De plus, la gestion de la profondeur n'est que très peu visible et n'influe peut être pas de la manière optimale. Enfin nous avons pensé à un point qui pourrait être amélioré. L'API permet certes à un jeu de se connecter sur notre système de quêtes mais le problème est que nous gérons nos objectifs avec notre propre objet. Cela impose que le jeu utilise ce même objet, ce qui peut poser des problèmes. De plus la structure du fichier XML est plus ou moins définie par notre programme et les catégories de bases ne sont pas modifiables.

Ce projet fut une très bonne expérience pour nous, permettant de mettre en œuvre nos connaissances en JAVA.

Annexe A

Liens utiles

Voici une petite liste d'url intéressantes au sujet de ce projet :

— Site de Polytech'Tours : www.polytech.univ-tours.fr
— Repository GitHub du projet : https://github.com/MrCraftCod/Quest-Generator
— Wiki GitHub : https://github.com/MrCraftCod/Quest-Generator/wiki
— Méthodes avec le mot clef default avec JAVA 8 : https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html
— JavaFX : http://docs.oracle.com/javafx/
— Enquête sur les quêtes dans les MMORPG = https://larc.unt.edu/techreports/LARC-2011-02.pdf

Annexe B

Fiche de suivi de projet PeiP

Attention, cette fiche doit être complète à la fin du projet. Les comptes-rendu des 2 premières séances sont ici donnés à titre d'exemple.

Séance n° 1 du 13/01/2016	— Découverte du sujet. — Prise de contact avec les encadrant. — Lecture de l'article envoyé par M. Barruet. — Début d'un plan de programme sur papier. — Début d'une réalisation sous JAVA.

Séance n° 2 du 20/01/2016	— Avancement dans le programme :
	  — Ajout des string pour les objectifs.
	  — Ajout d'un début de documentation.
	  — Prise en compte des actions non cassables.
	  — Arrêt de la quête quand possible et pondération de l'action ϵ.
	  — Utilisation de ThreadLocalRandom.
	— Ajout d'un fichier xml pour les valeurs d'objectifs.

Séance n° 3 du 27/01/2016	— Avancement dans le programme :
	  — Ajout et lecture des strings sous forme XML.
	  — Ajout d'une fenêtre de base.
	— Ajout du fichier décrivant les types d'objectifs utilisés.
	— Ajout d'un fichier décrivant la passation des objectifs entre les différentes actions.

Séance n° 4 du 03/02/2016	— Avancement dans le programme :
	  — Continuation de la passation des objectifs.
	  — Ajout d'une classe permettant de retrouver le chemin d'une valeur d'objectif.
	  — Ajout d'une fonction permettant de savoir si l'élément est dans le chemin indiqué.
	  — Ajout d'un objectif permettant d'identifier la classe qui a créée une action.
	  — Réinitialisation de la profondeur à chaque sous-quête. Cela évite d'avoir des sous-quêtes avec trop peu d'actions à faire.
	  — Ajout d'une fonctionnalité expérimentale pour donner une description aux quêtes. Sera sûrement utilisée pour le début des quêtes afin de donner un peu plus de crédibilité à celle-ci.
	  — Ajout d'informations supplémentaires dans la fenêtre affichant la quête si l'on est en mode debug.

Séance n° 5 du 10/02/2016	— Avancement dans le programme :
	  — Avancée dans la logique des objectifs :
	  — Possibilité d'exclure certains ActionExecutors en fonction des objectifs parents.
	  — Modification de la logique précédente en fonction des observations faites sur des générations, ajout de nouvelles catégories d'éléments, d'une logique plus poussée en utilisant des exclusions en fonctions des objectifs.
	  — L'observation des générations nous a amené à considérer l'action USE comme prenant en compte deux paramètres : utiliser quoi et sur qui/où ?
	  — Une nouvelle génération nous a amené à vouloir améliorer la génération décomposition des actions. En effet celles-ci n'avaient parfois aucun sens avec le reste.
	  — Suppression de la subquest (GOTO) ayant aucun sens et suppression de la sous-quête dans le GetSubquest. En effet devoir faire une subquest avant de retourner voir un PNJ n'est pas logique étant donné que nous avons déjà l'objet que nous devons ramener.
	  — Ajout d'une logique permettant de déterminer quelles Stratégies sont disponibles pour les sous-quêtes.
	  — Logique de passation des objectifs aux sous-quêtes.
	  — Le kill n'a aucun intérêt à être décomposé en GOTO-KILL.
	  — Les décompositions du STEAL ne contiennent plus de GOTO car toutes les quêtes (sauf une) contenant un STEAL possèdent déjà un GOTO juste avant. Pour la quête restante celui-ci à été directement ajouté avant le STEAL.

Séance n° 6 du 24/02/2016	— Avancement dans le programme :
	  — Ajout de phrases de description pour chaque quête.
	  — Définit quelles stratégies peuvent êtres utilisées en tant que sous-quêtes.
	— Avancement dans le rapport.

Séance n° 7 du 02/03/2016	— Avancement dans le programme : Correction d'espaces en trop entre les lignes dans l'interface si une description de quête est vide. — Avancement dans le rapport.

Séance n° 8 du 09/03/2016	— Avancement dans le programme :
	  — Ajout d'une différenciation des actions réalisées ou non, ce qui permet d'avoir une interface suivant la progression de la quête.
	  — Ajout d'une API.
	  — Ajout de la possibilité de lire des fichiers externes pour les phrases et objectifs.
	  — Ajout de différentes options d'exportation : XML TXT PNG ACTIONS.
	— Création d'un wiki simple.
	— Avancement dans le rapport.

Séance n° 9 du 16/03/2016	— Avancement dans le programme :
	  — Correction de problèmes dans la passation des objectifs.
	  — Phrases de description en anglais.
	— Ajout d'exemple de code dans le wiki.
	— Avancement dans le rapport.

Séance n° 10 du 23/03/2016	— Avancement dans le programme :
	  — Correction des couleurs dépassant des bords.
	  — Ajout des probabilités de tirage pour les motivations (cf. PDF).
	  — Possibilité de changer les probabilités grâce à un fichier.
	  — Ajout d'animations.
	— Mise à jour du wiki : Ajout de la page concernant les probabilités des motivations.
	— Avancement dans le rapport.

Séance n° 11 du 30/03/2016	— Avancement dans le rapport.

Séance n° 12 du 20/04/2016	— Finalisation du rapport. — Réflexion sur la soutenance.

Générateur de quêtes automatique pour la création d'un jeu

Rapport de projet S4

Résumé : Ce projet consiste à créer un générateur de quêtes aléatoires. En effet, les jeux de type MMORPG s'appuient énormément sur celles-ci. Le fait de pouvoir les générer aléatoirement permet de prolonger la durée de vie de ceux-ci. Afin d'arriver à nos fins, nous nous sommes appuyés sur une enquête réalisée à l'University of North Texas.

Mots clé : Générateur, Quêtes, Aléatoire, RPG, MMORPG

Abstract : This project consists in the creation of a random quest generator. MMORPGs are games that heavily rely on them. The fact of being able to generate them randomly allow us to extend the lifetime of the game. In order to create that software, we based our development on a study done at the University of North Texas.

Keywords : Generator, Quests, Random, RPG, MMORPG

Auteur(s)

Thomas Couchoud
[thomas.couchoud@etu.univ-tours.fr]
Victor Coleau
[victor.coleau@etu.univ-tours.fr]

Encadrant(s)

Sébastien Aupetit
[aupetit@univ-tours.fr]
Richard Barruet
[richard.barruet@etu.univ-tours.fr]

Polytech Tours
Département DI

Ce document a été formaté selon le format EPUProjetPeiP.cls (N. Monmarché)

École Polytechnique de l'Université de Tours
64 Avenue Jean Portalis, 37200 Tours, France
http://www.polytech.univ-tours.fr