



Projet de programmation

Partie Java



L3 Informatique
Semestre 5

2014

Ce dossier comporte l'organisation et les étapes de développement d'un projet d'Arènes en Java.

Il à été conçu par Dario OLIBET et Antoine de ROQUEMAUREL dans le cadre du module *Projet de programmation* de la L3 Informatique de l'université Toulouse III – Paul Sabatier.

Contenu de l'archive du projet

L'archive que vous avez reçus était organisée comme ceci :

rapport.pdf Le présent rapport que vous êtes en train de lire

v1Arene/ Contient le dossier avec la première version de l'application : notre propre arènes avec les règles que nous avons implémentées.

v1Arene/doc/ Contient la documentation de la première version du projet. Celle-ci à été générée à l'aide de *Doxygen*, elle est disponible en HTML ou en PDF (générée avec \LaTeX).

L'utilisation de *Doxygen* contrairement à *Javadoc* permet d'avoir une génération en PDF, mais également d'avoir l'apparition des diagrammes de classes ce qui facilite la compréhension du problème.

V2Commun/ Contient le dossier avec la version contenant notre stratégie pour l'arène commune distante. Les seuls fichiers que nous avons modifiés par rapport à la version de base sont les classes `Console`, `TestConsole` et `TestConsoleObjet`.

Le projet à été développé en Java6.

Table des matières

1	Organisation du travail d'équipe	4
1.1	Un outil de gestion de projet : Redmine	4
1.2	Un logiciel de versionnement : Git	5
2	Première version : local	6
2.1	Règles du jeu	6
2.1.1	Combattant	6
2.1.2	Caractéristiques	6
2.1.3	Combat	6
2.1.4	Équipements	7
2.2	Conception	7
3	Seconde Version : commun	9
3.1	Stratégie	9
3.2	Implémentation	9
4	Les difficultés rencontrés	10
A	Liste des tâches	11
B	Conventions d'écritures	11
C	Table des figures	15
D	Liste des tableaux	15

1

Organisation du travail d'équipe

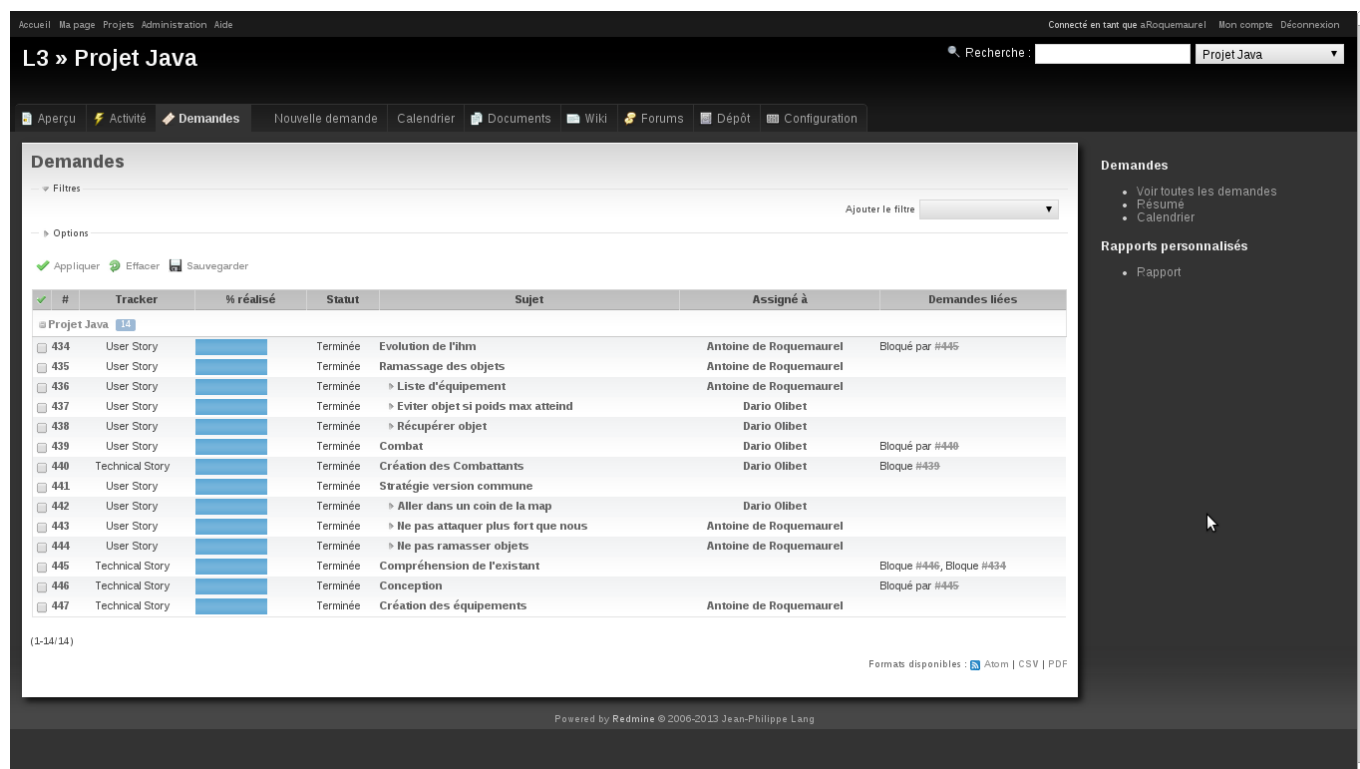
Pour ce projet, nous étions deux à travailler dessus, ainsi nous avons utilisé plusieurs techniques afin de se coordonner et de limiter les problèmes. Ceci n'est pas notre premier projet ensemble, notre travail en fut simplifié.

1.1 Un outil de gestion de projet : Redmine

Pour le projet, nous avons utilisé *Redmine*, une plateforme web de gestion de projet. Elle nous a permis de simplifier le travail, à ne rien oublier et à nous répartir le travail.

En effet, nous pouvons créer des tâches, signaler qu'elles sont en cours/terminées/en tests, leur donner des dates limites, les affecter à une personne etc... Ainsi lorsque l'un de nous commençait une tâche, il le signalait sur le *redmine*, ce qui permettait de tenir au courant son binôme de ses actions et de l'avancée du projet.

Nous nous sommes également servis du wiki que possède Redmine, ainsi nous avons pu y déposer des conventions d'écritures afin d'avoir un code qui soit cohérent, simple à relire et homogène. Les conventions sont disponibles en annexe A page 11.



The screenshot shows the Redmine web interface for a project named 'L3 » Projet Java'. The main section is titled 'Demandes' (Requests) and displays a table of tasks. The table has columns for '#', 'Tracker', '% réalisé', 'Statut', 'Sujet', 'Assigné à', and 'Demandes liées'. The tasks listed are numbered 434 through 447, with various trackers like 'User Story' and 'Technical Story'. The status of most tasks is 'Terminée' (Completed). The interface also includes a sidebar with 'Demandes' and 'Rapports personnalisés' (Custom Reports) sections.

#	Tracker	% réalisé	Statut	Sujet	Assigné à	Demandes liées
434	User Story		Terminée	Evolution de l'ihm	Antoine de Roquemaurel	Bloqué par #445
435	User Story		Terminée	Ramassage des objets	Antoine de Roquemaurel	
436	User Story		Terminée	↳ Liste d'équipement	Antoine de Roquemaurel	
437	User Story		Terminée	↳ Eviter objet si poids max atteint	Dario Olibet	
438	User Story		Terminée	↳ Récupérer objet	Dario Olibet	
439	User Story		Terminée	Combat	Dario Olibet	Bloqué par #440
440	Technical Story		Terminée	Création des Combattants	Dario Olibet	Bloque #439
441	User Story		Terminée	Stratégie version commune		
442	User Story		Terminée	↳ Aller dans un coin de la map	Dario Olibet	
443	User Story		Terminée	↳ Ne pas attaquer plus fort que nous	Antoine de Roquemaurel	
444	User Story		Terminée	↳ Ne pas ramasser objets	Antoine de Roquemaurel	
445	Technical Story		Terminée	Compréhension de l'existant		Bloque #446, Bloque #434
446	Technical Story		Terminée	Conception		Bloqué par #445
447	Technical Story		Terminée	Création des équipements	Antoine de Roquemaurel	

FIGURE 1.1 – Liste des tâches sous Redmine

1.2 Un logiciel de versionnement : Git

Afin de limiter les problèmes du travail collaboratif, nous avons utilisé un logiciel de versionnement Git. Il a deux intérêts, tout d'abord, nous pouvons travailler à deux en parallèle sur le projet sans se soucier de fusionner notre travail¹.

D'autre part, tous les logs étant enregistrés, nous pouvons savoir qui à fait quoi et quel jour, cela permet de voir également l'avancée du projet.

Enfin, toutes les modification sont stockées sur le serveur, ainsi en cas de problème, il est très facile de revenir à la version précédente ou même de comparer deux versions afin de voir les changements et de comprendre rapidement pourquoi une fonctionnalité à régressé.

Nous avons aussi pu nous en servir afin d'effectuer des branches de tests.

1. À condition de ne pas travailler sur deux lignes de code identiques

2

Première version : local

2.1 Règles du jeu

Nous avons décidé d'implanter des règles basé sur d'une part les combattants et d'autre part les équipements. Afin de gagner une partie le combattant doit être le dernier en vie dans l'arène.

2.1.1 Combattant

Chaque personnage dispose de caractéristiques, ces caractéristiques doivent respecter une règle d'équilibre. Si cette règle n'est pas respectée, le joueur n'est pas accepté sur le serveur : $\frac{vie}{10} + Attaque + Défense + Vitesse = 10$.

2.1.2 Caractéristiques

Nombre de point de vie Chaque personnage dispose d'un certain nombre de pv¹.

Attaque/Force La force d'un personnage

Vitesse/esquive Chance pour un personnage d'éviter une attaque.

Nombre d'objets Nombre d'objets qu'un personnage peut porter.

Le nombre d'objets qu'un personnage peut porter est choisi arbitrairement et compris entre 1 et 5 et ne rentre pas dans la règle d'équilibre.

		Vie	Attaque	Défense	Vitesse	Nb objet	Description
Capitaine	Luffy	30	3	2	2	3	Équilibré
Barde	Brook	10	2	3	4	2	Peu résistant mais ayant une bonne défense et une bonne fuite
Spadassin	Zorro	20	4	1	3	1	Préfère un combat rapide tout en essayant d'éviter un maximum d'attaques
Cyborg	Franky	20	3	4	1	4	Défensif et fort, mais très lent
Mascotte	Chopper	20	1	3	4	5	Bonne défense et bonne fuite.

TABLE 2.1 – Les différentes classes de personnage créées

2.1.3 Combat

Lorsqu'un combat à lieux, deux règles sont mises en place.

1. Point de vie

La première est basée sur l'esquive : la vitesse multipliée par 10 est le pourcentage de chances d'esquiver une attaque, lorsqu'une attaque est esquivée, aucun dégât n'est subi.

La deuxième est basée sur le ratio entre l'attaque et la défense, si l'attaque de l'attaquant est supérieure à la défense du défenseur nous avons : $attaque - defense = dégâts$ qui sont infligés au défenseur. Dans le cas où son attaque est inférieure à la défense, un seul point de dégât est causé.

2.1.4 Équipements

Un équipement peut donner un ou plusieurs bonus d'attaque, défense et vitesse. Tout équipement à une durée de vie qui correspond aux nombre de tours qu'un personnage l'utilise, ce nombre de tours est choisis arbitrairement entre 1 et 5².

Nos équipements sont tous équilibrés. Ils donnent un bonus total de 2 et un malus total de 1, donc $bonusAttaque + bonusDefense + bonusVitesse = 1$. Chaque personnage peut récupérer n'importe quel équipement mais chacun d'eux à une préférence selon sa classe afin d'être le plus équilibré possible.

		Attaque	Défense	Vitesse	Durée	Description
Chapeau de paille	Luffy	1	-1	1	2	
Canne	Brook	2	-1	0	1	
Sabre	Zorro	2	0	-1	4	
Plastron	Franky	0	2	-1	5	
Bottes	Chopper	-1	0	2	3	

TABLE 2.2 – Les différents équipements créés

2.2 Conception

Afin de concevoir au mieux l'application, nous avons tout d'abord étudié le code source existant. Une fois que nous avons compris son fonctionnement, nous avons définis nos équipements et nos personnes en fonction d'un **Element**.

Ci-dessous le diagramme de classe :

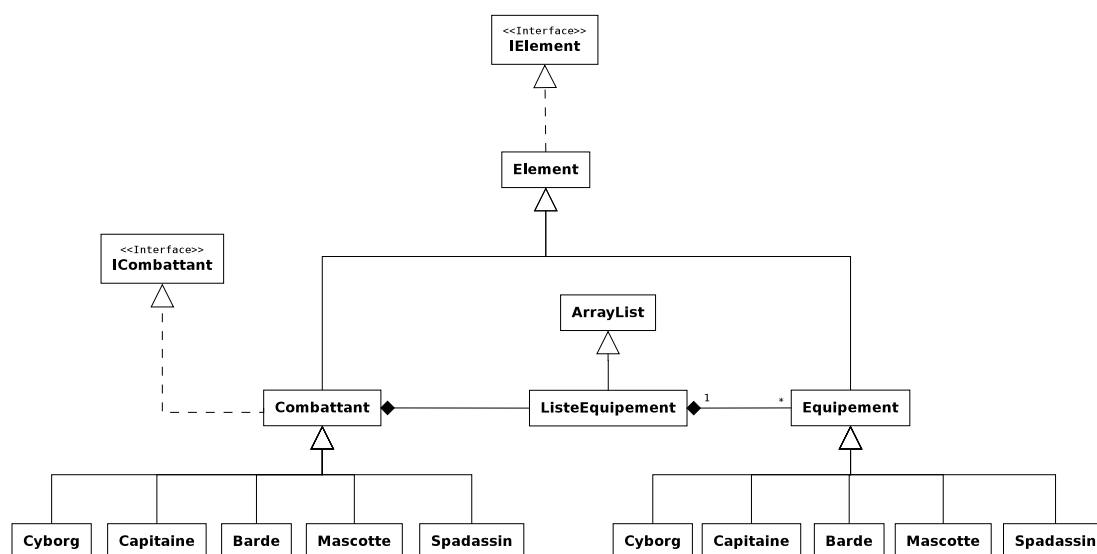


FIGURE 2.1 – Diagramme de classe des **Element**

2. Nous avons essayer d'avoir une certaine logique, en fonction du type de l'équipement. Par exemple un plastron aura une plus grande durée qu'une canne

Une fois ces classes créés, il fallait implémenter les règles du jeu, pour cela nous avons modifiés le serveur afin que les personnages puissent :

- Ramasser des objets, s’il en a la possibilité (vérification du poids) Classe **Arene**
- Implémentation des duels : Classe **DuelBasic**
- Vérification du respect de la règle d’équilibre : Classe **Arene**
- Ajout du port d’objets dans le calculs des statistiques : Classe **ListeEquipements**.

Nous avons également modifié un peu l’IHM afin de distinguer la différence entre un objet et un personnage : les objets sont représenté par un carré alors que les combattants par des ronds.

L’architecture de notre jeu est donc assez classique, mis-à-part la gestion de la possession des équipements. Nous avons choisis de créer une nouvelle classe **ListeEquipement** héritant de **ArrayList**, ce qui nous a permis de simplifier au maximum le reste de notre code. Une simple surcharge de la méthode **add** nous a permis d’ajouter un équipement uniquement si le nombre d’objets maximum n’est pas atteint. La souplesse des **ArrayList** nous a également permis de parcourir très facilement la liste d’équipement à l’aide des **Iterateur**.

3

Seconde Version : commun

3.1 Stratégie

Notre personnage, Franky a pour caractéristiques une attaque de 65 et une défense de 35. Il ne possède donc aucune esquive et aucune possibilité d'avoir un équipement.

Nous avons adapté notre objet ainsi que notre stratégie par rapport à ces caractéristiques. L'objet est une bombe, ayant un poids léger pour contrer les personnages ne prenant aucun équipement faisant un poids de 0. Ses bonus/-malus sont : 2 de force, 0 de défense, 0 d'esquive et un malus de 100 en vie. Ayant un bonus de 2 en force son poids était donc de $2 \times \frac{3}{4} = 1.5$ arrondi à 1. Nous nous débarrassons ainsi d'un possible ennemi, qui, après une certaine durée, pourrait avoir des statistiques pouvant nous battre.

La stratégie de notre combattant est, dans une première partie, de se diriger dans le coin le plus proche de l'arène et rester dans un coin de 30×30 pendant 2 minutes. Une fois que ce temps est écoulé notre personnage décide d'errer dans la totalité de l'arène en cherchant le combat, maintenant que les personnes se sont entretuées. Si il voit un adversaire ayant plus de vie que lui, il fuit en espérant que quelqu'un le rendra plus faible pour pouvoir gagner plus tard.

3.2 Implémentation

Afin d'implémenter cette stratégie nous avons utilisé la méthode `run`. Pour la première version, nous n'avions pas effectué de stratégie et n'avions pas cherché le moyen de contrôler le déplacement de notre personnage, ainsi une nouvelle étude du fonctionnement de `Console` nous a permis de trouver les solutions à nos problèmes.

Nous avons donc créé un attribut permettant de compter le nombre d'actions écoulés, c'est-à-dire le nombre de fois ou `run` est lancé. C'est comme ceci que nous avons pus dès le premier lancement du `run` choisir un endroit ou aller dans la map, et n'y rester qu'un certain temps.

4

Les difficultés rencontrés

Notre difficulté principale fut la compréhension du système de **Remote**. En effet une **Arene** ne possédait qu'un entier comme guise de référence, la question étant comment pouvoir récupérer un **Element**, ou même mieux un **Combattant** à partir de cet entier. C'est en cherchant dans le projet que nous avons trouvé la solution : La récupération de la **Remote**, puis le cast vers une **Console**, récupération d'un **Element** avec la fonction `getElement` pour finalement caster de nouveau si le besoin s'en faisant sentir.

Cependant, la javadoc du système étant assez complète, nous avons pu comprendre le fonctionnement de l'architecture et ainsi effectuer nos propres modifications.

A

Liste des tâches

Projet Java - Demandes

#	Tracker	% réalisé	Statut	Sujet	Assigné à	Demandes liées
Projet Java (14)						
434	User Story	100	Terminée	Evolution de l'ihm	Antoine de Roquemaurel	Bloqué par #445
435	User Story	100	Terminée	Ramassage des objets	Antoine de Roquemaurel	
436	User Story	100	Terminée	Liste d'équipement	Antoine de Roquemaurel	
437	User Story	100	Terminée	Eviter objet si poids max atteint	Dario Olibet	
438	User Story	100	Terminée	Récupérer objet	Dario Olibet	
439	User Story	100	Terminée	Combat	Dario Olibet	Bloqué par #440
440	Technical Story	100	Terminée	Création des Combattants	Dario Olibet	Bloque #439
441	User Story	100	Terminée	Stratégie version commune		
442	User Story	100	Terminée	Aller dans un coin de la map	Dario Olibet	
443	User Story	100	Terminée	Ne pas attaquer plus fort que nous	Antoine de Roquemaurel	
444	User Story	100	Terminée	Ne pas ramasser objets	Antoine de Roquemaurel	
445	Technical Story	100	Terminée	Compréhension de l'existant		Bloque #446, Bloque #434
446	Technical Story	100	Terminée	Conception		Bloqué par #445
447	Technical Story	100	Terminée	Création des équipements	Antoine de Roquemaurel	

FIGURE A.1 – Liste des tâches Redmine

B

Conventions d'écritures

Conventions d'écriture en Java

Voici les conventions écritures que nous avons fixé, il faudra les respecter pour que nous ayons un code propre et homogène, de plus elles ont été fixées pour que ce soit le plus simple pour nous (lecture rapide, propreté etc...)

Le nommage

Les attributs

Les attributs doivent **toujours** commencer avec une minuscule, pour séparer les mots, on les sépare avec une majuscule. Privilégiés les noms de variables claires et explicite, quitte à ce qu'il soit un peu long, on a l'auto complétion que diable! Donc les variables d'une lettre, à bannir! (à part *peut être* i dans le cas d'un for, s'il n'est pas réutilisé après le for

Les attributs seront préfixé par `_` afin de pouvoir les reconnaître facilement, il n'est donc pas indispensable d'utiliser `_`.

Exemples de noms d'attribut

```
int _monSuperAttribut;  
boolean _vousAvezPerdu;
```

Les variables en paramètre de méthodes

Les variables paramètre sont les variables qui ne sont que dans une méthode (et donc, elles sont détruites à la fin de la méthode). Ces variables doivent respecter la règle précédente à la différences qu'elle doivent toute commencer par un `p` (pour paramètre)

Exemples de noms de paramètre

```
int pMonSuperParametre;  
boolean pVousAvezPerdu;  
String pCacamou;
```

Les noms de constantes

Les constantes doivent être tout en majuscule, les différents mots de la constante sont séparés par des underscore (`_`). Même remarque que pour les attributs, choisissez des noms de constante clair, compréhensible par tous, pas seulement par ceux qui sont dans votre tête!

Exemple de noms de constante

```
final int MA_SUPER_CONSTANTE;  
final boolean VOUS_AVEZ_PERDU;  
final String CONST;
```

Les noms de méthodes

Les méthodes doivent commencer par une minuscule, et séparer les différents mots par une majuscule (Camel case). Les fonctions ne retournant rien (procédures) doivent toujours être à l'infinifit.

À l'opposé les fonctions retournant quelques choses doivent être au participe passé.

Il faut décomposer au maximum, n'hésitez pas à faire une méthode private si besoin est, c'est toujours plus clair d'avoir une fonction, dictant explicitement ce qu'elle fait par son nom que 10 lignes de code bizarroïdes avec 2-3 lignes de commentaire! Et donc, les noms de fonctions sont essentiels!!

Exemple de noms de fonction

```
void afficher(String pTexteAAffiche) {  
    System.out.println(pTexteAAffiche);  
}  
boolean inferieur(int entier1, int entier2) {  
    return (entier1 < entier2);  
}
```

Les noms de classes ou d'interface

Les noms de classe ou d'interface doivent tous commencer par une majuscule, les différents mots sont séparés par une majuscule, choisissez des noms de classes claires!

(oui, je me répète, mais c'est ce qui fait toute la compréhension facile, ou non, d'un programme les noms de variables, classe, paramètre, méthodes etc...)

Exemple de noms de classe

```
class MaSuperClasse { }
class UneAutreClasse extends MaSuperClasse { }
interface MaSuperInterface { }
```

Les noms de packages

Les noms de package ont la même convention que les noms d'attributs: ils commencent par une minuscule, et les différents mots sont séparés par une majuscule(Camel case), cela permettra de différencier les packages des classes.

Exemple de noms de package

```
package monPackage;
package monSuperPackage.unAutrePackage;
import monSuperPackage.unAutrePackage.MaSuperClasse;
```

L'indentation

La règle est simple, on ouvre une accolade, la ligne suivante sera décalé vers la droite(une tab = 4 caractères), on ferme une accolade, on décale l'accolade vers la gauche et tout ce qui suis.

Egalement, si une ligne est trop longue, on va à la ligne, et décalons d'une ligne vers la droite, une fois l'instruction finie, on redécale vers la gauche.

Dans le cas d'un switch, le break doit s'aligner avec le case 42: tout ce qui est entre case et break sera indenté.

Merci de mettre un espace avant chaque accolades, oui je sais je suis psychorigides, mais c'est moche sinon.

Exemple d'indentation

```
class MaSuperClasse {
    int monSuperAttribut;

    void afficherHelloWorld() {
        System.out.println(
            "Hello World");

        switch(yatta) {
            case 42:
                // ^^
                break;
            case 1337:
                // ...
                break;
            default:
                //
        }
    }
}
```

Les accolades

Les accolades ouvrante sont positionnés à la fin de la ligne demandant une accolade (switch, if, class, else, elseif, ...)

Les accolades fermantes sont positionnés une ligne après la dernière instruction. (avec une désindentation)

Les else et elseif se mettent sur la même ligne que l'accolades fermante.

Exemple d'indentation

```
int maSuperMethode() {  
    if(true) {  
        // bla bla  
    } else if(false) {  
        // bla bla  
    } else {  
        //instruction  
    }  
  
    switch(var) {  
        case 0:  
            // ^^  
            break;  
        case 1:  
            // yatta  
            break;  
        default:  
            // :-)  
    }  
}
```

Position des mots clefs

Le mot clef abstract doit toujours être avant le mot clef class ou avant le type de retour de la méthode.

Exemple de méthodes abstraites

```
abstract class Cacamou {  
    abstract int maMethode();  
}
```

C

Table des figures

1.1	Liste des tâches sous Redmine	4
2.1	Diagramme de classe des Element	7
A.1	Liste des tâches Redmine	11

D

Liste des tableaux

2.1	Les différentes classes de personnage créées	6
2.2	Les différents équipements créés	7