Rapport :

Titre provisoire + lien page web

(A3P(AL) 2015/2016 Gα)

I.A) Auteur(s)

I.B) Thème (phrase-thème validée)

I.C) Résumé du scénario (complet)

I.D) Plan (complet, avec indication de la partie "réduit" si exercice 7.3.3)

I.E) Scénario détaillé (complet, avec indication de la partie "réduit" si exercice 7.3.3)

I.F) Détail des lieux, items, personnages

I.G) Situations gagnantes et perdantes

I.H) Éventuellement énigmes, mini-jeux, combats, etc.

I.I) Commentaires (ce qui manque, reste à faire, ...)

II. Réponses aux exercices **(à partir de l'exercice 7.5 inclus)**

III. Mode d'emploi (si nécessaire, instructions d'installation ou pour démarrer le jeu)

IV. Déclaration obligatoire anti-plagiat **(*)**

V, VI, etc... : tout ce que vous voulez en plus

(*) Cette déclaration est obligatoire :
- soit pour préciser toutes les parties de code que vous n'avez pas écrites vous-même et citez la source,
- soit pour indiquer que vous n'avez pas recopié la moindre ligne de code (sauf les fichiers zuul-*.jar qui sont fournis évidemment).

Violet Evergarden + lien page web
(A3P(AL) 2023/2024 G9)

I.A) Auteur

- Anthony Pradier

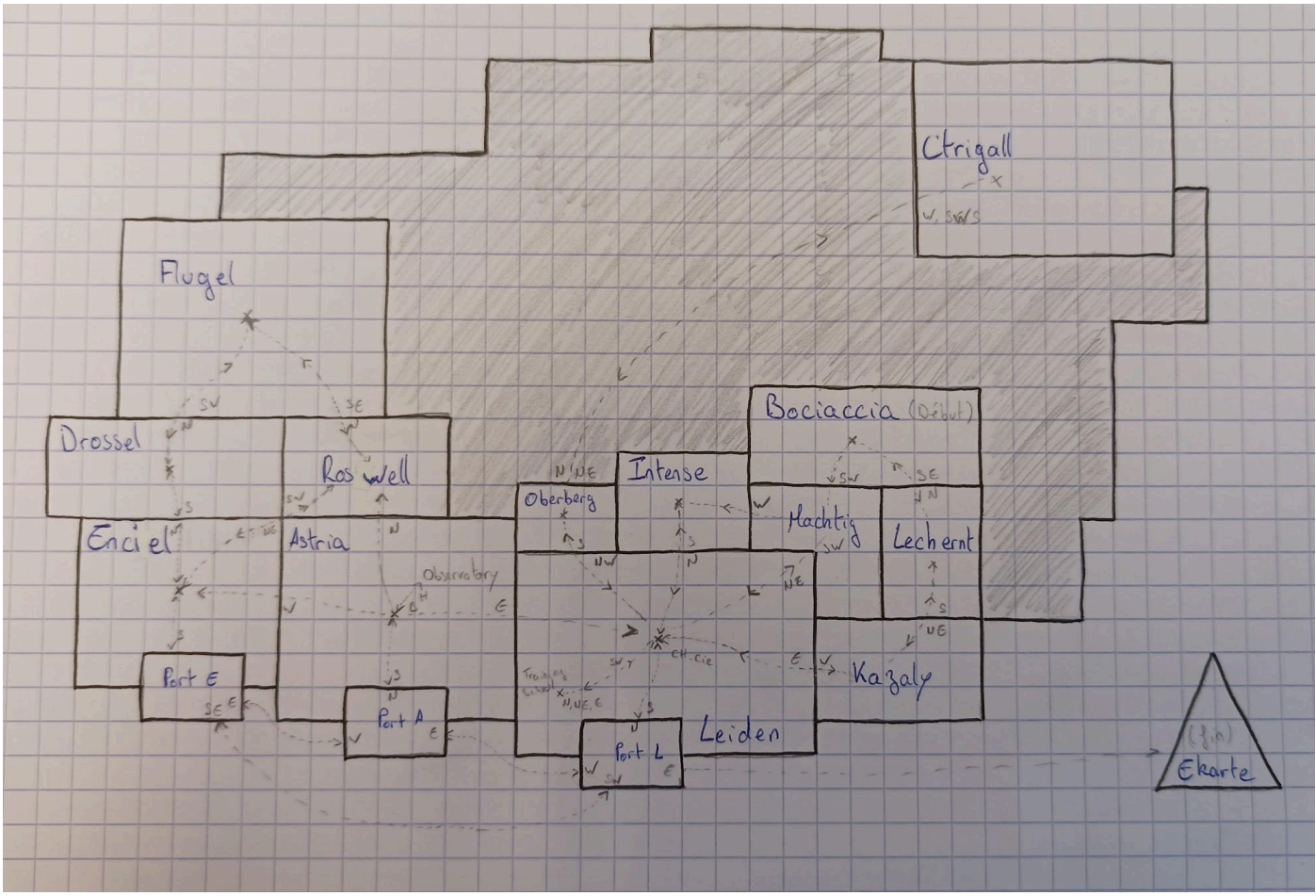
I.B) Thème

- Au Japon, Violet recherche un être cher et le sens d'une phrase

I.C) Résumé du scénario complet

- Violet, une adolescente illettrée, rejoint l'armée à 15 ans et perd son major, le major Gilbert, pendant la dernière bataille. Ne comprenant pas la signification des mots que lui a prononcés le major avant sa mort, Violet se met en quête de devenir une poupée de souvenirs automatique pour tenter de développer ses sentiments et sa compréhension des mots, et de chercher tant bien que mal le major Gilbert, déclaré mort au combat.

I.D) Plan complet



Que je transférerai sur la carte suivante en jeu :

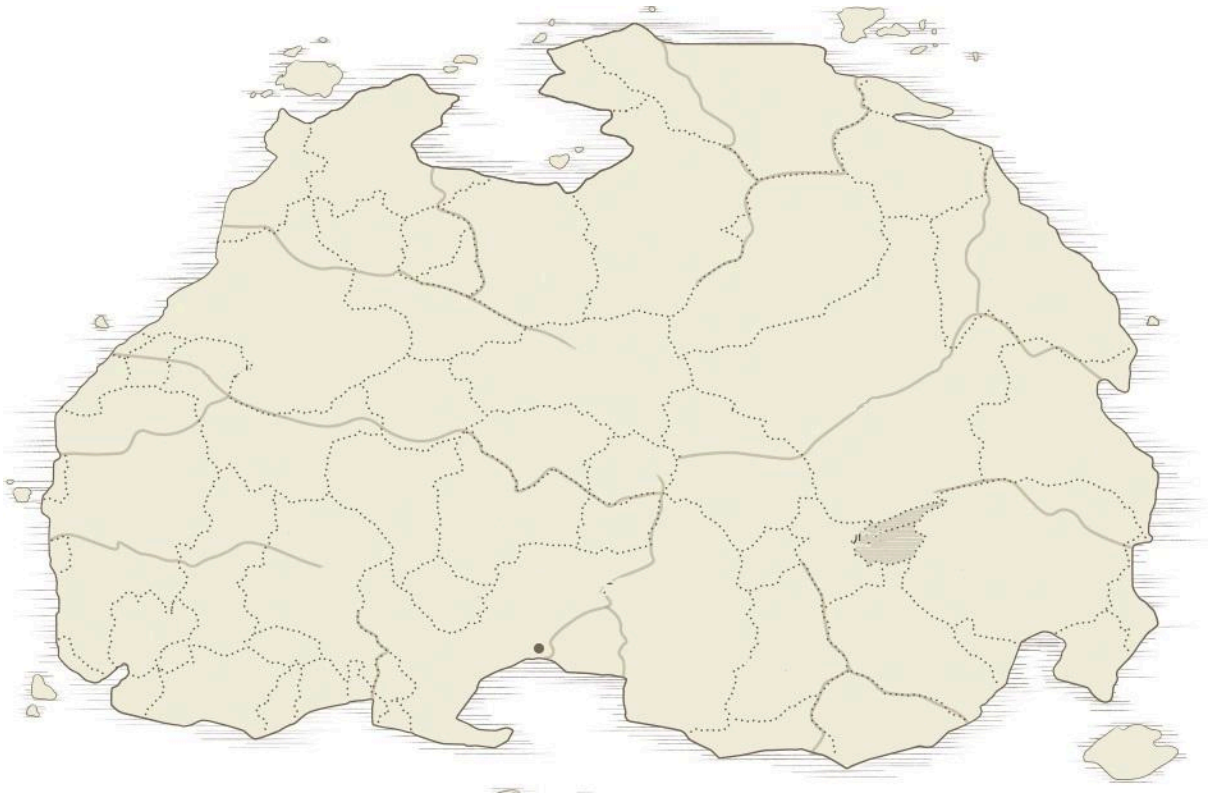


Image prise sur le site https://violet-evergarden.fandom.com/wiki/Category:Locations

I.E) Scénario détaillé

- Violet, une adolescente illettrée qui a grandi dans la guerre, rejoint l'armée japonaise à 15 ans. Elle reçoit un minimum d'éducation de la part de son major, le major Gilbert qui lui donne d'ailleurs son prénom, en apprenant à lire, écrire et parler. Du fait de leur statut hiérarchique, ils deviennent proches au point que le major développe des sentiments pour elle. Le jeu commence dans la ville de Bociaccia, pendant une bataille où Violet élimine les soldats ennemis sans cesse, avec aisance. Durant la dernière bataille de cette guerre à Intense, gagnée par l'armée japonaise, à la fin, le major Gilbert est touché par des balles et par une explosion de grenade. Incapable de bouger, Violet tente de le sortir de cette situation, alors qu'elle a elle même perdu ses deux bras à cause d'une grenade. Le major lui donne alors comme dernier ordre de vivre en lui prononçant ces mots "Vis… Sois libre. Du plus profond de mon coeur, je t'aime".

  Après la guerre, Violet se réveille dans un lit d'hôpital à Enciel suite à 112 jours de coma et veut de suite chercher Gilbert. Son ancien lieutenant, Claudia Hodgins a fondé sa compagnie postale qu'il avait juré de créer quand la guerre serait terminée. Il propose alors à Violet de travailler pour lui en tant femme de ménage pour qu'elle puisse construire une nouvelle vie.

Découvrant que la compagnie de Claudia consistait à écrire et envoyer des lettres, elle lui demande de changer de poste, et de travailler en tant que Poupée de souvenirs automatique en disant “Je veux savoir ce que "je t'aime” veut dire”. Claudia accepte la demande, et confie la formation de Violet à Benedict. Ses premières tâches consistent à trier des lettres par villes et par rues, puis à les livrer. Elle s’entraîne ensuite à rédiger des lettres pour des clients qui se disent tous insatisfaits de son travail, en raison de son manque d'expérience, mais en acquiert petit à petit à force d’être aidée par ses collègues plus expérimentés.

Voyant que Violet s’accroche à ses ambitions, Claudia Hodgins l’inscrit dans une école spéciale à Leiden pour suivre une formation dans le but de devenir une Poupée de souvenirs automatique reconnue, grâce à une broche spéciale. Elle écrit pendant cette formation une lettre pour le frère d’une amie qu’elle s’est faite dans sa classe. Avec cette reconnaissance, les missions qu’elle mène sont très souvent un succès, et elle peut voyager avec beaucoup plus d’assurance pour les effectuer. Elle peut par exemple se rendre à l’observatoire d’Astria et y rencontrer des astronomes, à Ctrigall pour écrire au nom d’un soldat mort ou encore au Royaume de Drossel pour exprimer les sentiments de la princesse au Prince de Flugel. A partir de là, Violet est libre d’aller où elle veut pour accomplir ses missions, peut voyager à sa guise, mais ne doit pas perdre de temps car plus le temps avance, plus les chances de retrouver Gilbert sont faibles. Plusieurs missions, plus ou moins facultatives permettent à Violet de rencontrer de nouvelles personnes, et d’augmenter ses chances de retrouver Gilbert.

A Astria, Violet pourra prendre le téléphérique pour aller en haut d’une montagne jusqu’à l’observatoire d’Astria et ne pourra descendre uniquement par ce même téléphérique. Violet retourne à Intense, là où elle a perdu Gilbert et se convainc de sa mort. Elle dépose une lettre à la tombe qui lui a été inaugurée. Plus tard, elle rendra visite au frère de Gilbert qui lui racontera la vérité. Elle apprend alors l’existence de l’île d’Ekarte à l’est du Japon où vit Gilbert, à l’écart de tout problème.

FIN

I.F) Détails des lieux, items, personnages

- ➢ Lieux
  - Company(N: Intense, NE: Machtig, E: Kazaly, S: Port L, SW: School, NW: Oberberg, U: FirstFloor)
    - Compagnie Postière CH située dans la ville de Leiden, lieu principal/central. Compagnie de Poupées de souvenirs automatiques employées pour écrire des lettres pour des clients qui ne peuvent pas le faire : retranscrire sur papier

les émotions et les sentiments que le client veut faire passer. Lieux où Violet, Claudia, Bénédict, Iris, Cattleya, et d'autres employés travaillent.

- FirstFloor(NE: ClaudiaOffice, SE: CattleyaOffice, S: WorkRoom, D: Company)
  - Premier étage de la Compagnie, entourée par des bureaux et de la salle de travail, salles dans lesquels on peut y trouver des personnages
- ClaudiaOffice(W: FirstFloor)
  - Bureau de Claudia Hodgins situé au premier étage de la Compagnie. On le trouve très souvent là, et parfois aussi Cattleya.
- CattleyaOffice(W: FirstFloor)
  - Bureau de Cattleya situé au premier étage de la Compagnie. On la trouve régulièrement ici, parfois dans le bureau de Claudia, parfois en voyage.
- WorkPlace(N: FirstFloor)
  - Salle de travail située au premier étage de la Compagnie. On y trouve quelques employés à cet endroit.
- School(N: Company, NE: Company, E: Company)
  - École située à Leiden au sud-ouest de la compagnie où Violet doit se former pour passer un concours et devenir Poupée de souvenirs automatiques. Il y a plusieurs autres candidates, dont Luculia pour qui elle écrira une lettre à son frère. Elle y rencontre la professeur Rhodanthe qui est chargée de les former.
- Oberberg(N: Ctrigall, NE: Ctrigall, S: Company)
  - Ville au nord-ouest de Leiden dans laquelle on y trouve un hangar stockant des avions de petite taille. Violet en utilise un pour se rendre à Ctrigall pour chercher Gilbert.
- Intense(S: Company)
  - Ville de la bataille décisive pendant la guerre entre le Japon et les États-Unis. C'est ici que Gilbert perd un bras et son œil droit, et que Violet perd ses deux bras en voulant le sauver.
- Bociaccia(SE: Lechernt, SW: Machtig)
  - Ici débute l'histoire. Première bataille que mène Violet pour l'armée japonaise, où elle cause à elle seule un bain de sang
- Machtig(E: TownHall, SW: Company, W: Intense)
  - Ville contenant un marché que Violet et Gilbert font pour Thanksgiving. Il lui achète une broche qu'elle trouvait belle, de la même couleur que les yeux de son major.
- TownHall(N, W, NW: Machtig)
  - Hôtel de ville de Machtig, lieu dans lequel travaille Dietfried Bougainvillea.
- Lechernt(N: Bociaccia, S: Kazaly)

        - Là où vivent Clara et sa fille, dans une grande maison à la campagne. Violet doit écrire plusieurs dizaines de lettres à Clara, atteinte d'un cancer et bientôt mourante, qui a en tête d'en envoyer une pour chaque anniversaire d'Anne.
    - Kazaly(NE: Lechernt, W: Company)
        - Ville natale d'Iris. Violet assiste à l'anniversaire des 18 ans d'Iris.
    - Astria(N: Roswell, E: Company, S: Port A, W: Hospital, U: Observatory)
        - Capitale de l'astronomie. Possède un observatoire en haut d'une montagne, accessible via une grande ligne de télécabine. C'est une ville traversée par des chaînes de montagnes, ce qui explique l'impossibilité de l'accès à Astria depuis les villes alentour.
    - Observatory(D: Astria)
        - Observatoire situé à Astria dans lequel de nombreux chercheurs en astrologie et théologie y travaillent. On y observe une splendide comète une fois tous les 400 ans.
    - Hospital(N: Drossel, NE: Roswell, S: Port E, W: Opera)
        - Hôpital situé dans la ville d'Enciel à l'ouest d'Astria. C'est ici que Violet se réveillera après la guerre et 112 jours de coma.
    - Opera(E: Hospital)
        - Opéra situé à Encial dans lequel Irma Fliech fait ses prestations.
    - Drossel Castle(N: Flugel, S: Hospital)
        - Château situé au Royaume de Drossel. Violet y rencontre la princesse Charlotte Abelfreyja Drossel pour qui elle doit écrire une lettre de demande en mariage qu'elle enverra au prince Damian Baldur Flugel.
    - Flugel Castle(SE: Roswell, SW: Drossel)
        - Château situé dans le Royaume de Flugel, là où réside le prince Damian Baldur Flugel. Ce lieu n'a pas d'importance pour l'histoire.
    - Roswell(N: Flugel, SW: Hospital, NE: AidanHouse)
        - Gigantesque lac situé dans la ville de Roswell, où réside Oscar Webster.
    - AidanHouse(S, SW, W: Roswell)
        - Maison D'Aiden où vit sa famille.
    - Ctrigall(S: Oberberg, SW: Oberberg, W: Oberberg)
        - Région très froide et très enneigée où une guerre civile entre deux régions fait encore rage. Accessible uniquement en avion
    - Ekarte()
        - Île de fin, là où vit le major Gilbert depuis plusieurs années paisiblement
    - Port E(E: Port A, SE: Port L)
        - Port situé dans la ville d'Enciel
    - Port A(E: Port L, W: Port E)
        - Port situé dans la ville d'Astria

- Port L(E: Ekarte, SW: Port E, W: Port A)
    - Port situé dans la ville de Leiden

➢ Personnages

- Violet Evergarden
    - Héroïne du jeu, adolescente de 15 ans au début du jeu et 20 ans à la fin de l'histoire. Capacité de combat élevée au début du jeu, qui décroît au fil de l'avancée. Capacité à comprendre les autres nulle au début, qui croît au fil de l'avancée. Travaille pour la compagnie CH.
- Gilbert Bougainvillea
    - Major de Violet pendant la guerre avec qui il a créé beaucoup de liens. Présumé mort, sans aucune preuve. Il s'est installé sur l'île d'Ekarte pour ne pas nuire à la vie de Violet
- Dietfried Bougainvillea
    - Frère de Gilbert Bougainvillea, Capitaine de la marine pendant la guerre. Il réside à Machtig, travaille dans l'Hôtel de ville de Machtig et a des informations sur son frère. Assez froid avec Violet.
- Claudia Hodgins
    - Président de la Compagnie Postière CH, ancien lieutenant de Violet pendant la guerre. On peut le trouver dans la majorité du temps à la Compagnie ou dans son bureau
- Cattleya Baudelaire
    - Sous-directrice de la Compagnie, femme de Claudia Hodgins. On la trouve souvent à la compagnie ou dans son bureau ou dans le bureau de Claudia, mais voyage régulièrement.
- Iris Canary
    - Diplômée de l'école de Poupée de souvenirs automatiques. Enfance passée à Kazaly, employée de la Compagnie CH. On la trouve la majorité du temps à la Compagnie, ou sinon à Kazaly.
- Benedict Blue
    - Premier à s'occuper de la formation de Violet à la Compagnie. Employé de la Compagnie, ont le trouve tout le temps ici, et parfois en déplacement pour poster des lettres.
- Leon Stéphanotis
    - Étudiant en astronomie à Astria. On le trouve généralement dans l'observatoire ou sinon à Astria.
- Rhodanthe
    - Diplômée de l'école de Poupée de souvenirs automatiques. Elle se charge de la formation des poupées dans cette école. Assez froide avec Violet. On la trouve uniquement à l'école de formation.

- Luculia Marlborough
    - Meilleure amie de Violet, participe également à la formation des poupées de souvenirs automatiques. Soeur de Spencer Marlborough qui sombre dans l'alcool à cause de la guerre dans laquelle ils ont perdu leurs parents. On la trouve majoritairement à Leiden et voyage quelques fois.
- Spencer Marlborough
    - Frère de Luculia Marlborough qui sombre dans l'alcool à cause de la guerre. Il reçoit une lettre de la part de sa sœur, écrite par Violet qui dit "Merci d'être en vie". On le trouve uniquement à Leiden.
- Ann Magnolia
    - Fille de Clara Magnolia qui vit à Lechernt avec sa mère dans une maison isolée à la campagne. On la trouve uniquement à Lechernt.
- Clara Magnolia
    - Mère d'Ann Magnolia, atteinte d'une maladie incurable qui la rend mourante. Elle fait appelle à Violet pour écrire des lettres pour chaque anniversaires d'Ann. Période très émouvante pour Violet qui doit contenir sa tristesse. On la trouve uniquement à Lechernt.
- Irma Fliech
    - Chanteuse d'opéra à Enciel. Elle demande à Violet de lui écrire une chanson, pour envoyer un message à une personne pendant sa prestation. On la trouve uniquement à Enciel à l'opéra.
- Oscar Webster
    - Habitant à Roswell près du grand lac. Tombe dans l'alcool depuis la mort de sa fille Olivia Webster. Il demande à Violet pour qu'elle lui écrive une lettre à déposer sur la tombe de sa fille. Il lui demande également de courir sur l'eau du lac, et la considère à la fin du service comme une déesse. Il trouve que Violet ressemble à sa fille, et lui offre comme cadeau l'ombrelle qu'elle avait l'habitude d'utiliser. On le trouve uniquement au lac de Roswell.
- Charlotte Abelfreyja Drossel
    - Princesse du Royaume de Drossel. Doit se marier avec le prince Damian Baldur Flugel. Violet doit écrire une lettre qui exprime ses sentiments au Prince. On la trouve uniquement au Royaume de Drossel.
- Aidan Field
    - Soldat de Ctrigall. Il rencontre violet alors qu'elle cherche le major Gilbert. Il est victime de coups de feu, il ne s'en sort pas vivant. Il demande cependant à Violet d'écrire une lettre d'adieu à sa famille qui réside à Roswell, au nord-est de la ville. On le trouve uniquement à Ctrigall.

➢ Items à définir

I.G) Situations gagnantes et perdantes

- Gagnantes :
    a. Violet retrouve Gilbert sur l'île d'Ekarte et arrive à le faire revenir à Leiden. Plus la jauge d'expérience de Violet est élevée, plus elle à de chances de faire revenir Gilbert.
    b. Easter Egg : Il y a un bubble tea caché dans le jeu, obtenable après avoir fait une certaine série de quêtes. Boire ce bubble tea fera gagner le jeu immédiatement.
- Perdantes :
    a. Violet retrouve Gilbert mais ne le convainc pas de revenir.
    b. Violet met trop de temps à retrouver Gilbert, et se résout à accepter sa mort.

I.H) Éventuellement énigmes, mini-jeux, combats, etc.

- À définir

I.I) Ce qu'il reste à faire

- Retranscrire le plan sur papier sur une version numérique
- définir les items
- définir les énigmes, mini jeux et combats

II) Exercices

★ 7.5 : Avez-vous compris pourquoi on vous demande de créer et utiliser cette procédure, et pourquoi la situation était "inacceptable" avant cette amélioration ?
Pensez aux futures modifications qui pourraient arriver, qui vous demanderaient d'afficher plus de choses pour chaque lieu ...
En tous cas, lorsqu'on a effectué cette amélioration, **il n'est plus question de revenir en arrière dans de futurs exercices** et de dupliquer à nouveau des instructions, a minima dans `printWelcome` et `goRoom`.
    ○ On crée cette procédure pour éviter la duplication de code, d'autant plus que les instructions sont lourdes et redondantes (répétition de `if`)

★ 7.6 : La version de `setExits` proposée dans le livre teste si chaque paramètre est `null` avant de l'affecter à l'attribut correspondant, mais ce n'est pas indispensable.
Comprenez-vous pourquoi ?
    ○ Les attributs de sorties de la classe Room sont déclarés, mais pas initialisés, et n'ont par définition aucun objet. Ils ont donc la valeur `null` et la gardent si les paramètres correspondant aux sorties sont `null`. En revanche, si un paramètre est non `null`, alors l'attribut correspondant changera et contiendra cette fois-ci un objet.

★ 7.7 : Comprenez-vous pourquoi il est logique de demander à `Room` de produire les informations sur ses sorties (et ne pas lui demander de les afficher), et pourquoi il est logique de demander à `Game` d'afficher ces informations (et ne pas lui demander de les produire) ?
    ○ Je dirai que c'est parce que étant donné que les attributs des `exits` sont en `private`, la classe `Game` n'est pas censée y avoir accès, et que seule la classe `Room` peut les renvoyer. Pour l'afficher, il est préférable de faire cette instruction dans la classe `Game` car on pourrait peut-être avoir besoin plus tard des sorties en chaîne de caractères sans forcément vouloir les afficher.

★ 7.8 **Implémentez dans votre projet** les changements décrits dans cette partie du livre. N'y aurait-il pas une **méthode inutile** désormais dans la classe `Room` ? Par quoi a-t-elle été remplacée ? Vous devez comprendre pourquoi son **nom** a été ainsi **modifié**.
    ○ La méthode `setExits` est devenue inutile. On l'a remplacé par la méthode `setExit`. Leur nom est différent car dans la première, on définissait toutes les

sorties tandis que dans la deuxième on n'en définit qu'une seule, d'où le "s" en moins.

★ 7.10 Fonctionnement de `getExitString`

- On crée une variable `String returnString` qui contient une chaîne de caractères `"Exits:"`, à laquelle on veut concaténer les sorties. On crée ensuite l'ensemble des clés de la `HashMap` propre à la salle actuelle, clés qui sont des `String`, d'où le `Set<String>`, sans oublier d'importer la classe `Set` avec `import java.util.Set`. On fait une boucle pour parcourir cet ensemble de clés, on concatène ces clés à `returnString` précédées d'un espace pour rendre la chaîne lisible. Puis on la `return`.

★ 7.11 `getLongDescription`

```java
/**
 * Renvoie une description longue de la salle, comprenant
 * la description, les sorties et plus tard d'autres informations
 * @return La description complète de la salle
 */
public String getLongDescription() {
    return "Je suis " + this.aDescription + ".\n" + this.getExitString();
}
```

★ 7.12

★ 7.13

★ 7.14

```java
/**
 * Permet au joueur d'afficher les informations completes de la salle actuelle
 * @param Une commande
 */
private void look(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        System.out.println("Je ne peux pas encore regarder quelque chose en particulier.");
    } else {
        System.out.println(this.aCurrentRoom.getLongDescription());
    }
} // look()
```

★ 7.15

```java
/**
 * Permet au joueur de récupérer de l'énergie, ou de consommer un consommable spécial
 * @param La commande
 */
private void eat(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        System.out.println("Je ne peux rien manger de spécial pour l'instant.");
    } else {
        System.out.println("Je n'ai plus faim, je vais enfin pouvoir reprendre mon travail.");
    }
} // eat()
```

★ 7.16

-
```java
/**
 * Affiche toutes les commandes utilisables
 */
public void showAll() {
    for(String vCommand : this.aValidCommands) {
        System.out.print(vCommand + ", ");
    }
    System.out.println();
}
```
-
```java
/**
 * Demande à la classe CommandWords d'afficher les commandes utilisables
 */
public void showCommands() {
    this.aValidCommands.showAll();
}
```
-
```java
/**
 * Affiche le message d'aide
 */
private void printHelp() {
    System.out.println("Les commandes utilisables sont :");
    aParser.showCommands();
} // printHelp()
```

★ 7.17

- Si on ajoute une nouvelle commande, on n'a plus besoin de changer la classe `Game` puisque la classe `Game` s'occupait à la base d'afficher les commandes. Il fallait ajouter les commandes dans les commandes valides de la classe `CommandWords`, puis les écrire à la main dans la classe `Game`. Tandis que maintenant, les méthodes `showAll` et `showCommands` permettent d'afficher toutes les commandes seulement à partir des commandes valides de `COmmandWords`, ce qui nous évite de modifier la classe `Game`.

★ 7.18

```java
/**
 * Retourne les Commandes utilisables sous forme d'une String
 * @return Les commandes utilisables dans une String
 */
public String getCommandList() {
    String vCommands = "";
    for(String vCommand : this.aValidCommands) {
        vCommands += " " + vCommand;
    }
    return vCommands;
}
```

```java
/**
 * Affiche les commandes utilisables renvoyees par getCommandList()
 */
public void showCommands() {
    System.out.println(this.aValidCommands.getCommandList());
}
```

★ 7.18.2

```java
/**
 * Renvoie les salles accessibles dans un StringBuilder
 * @return Les salles accessibles dans un StringBuilder
 */
public StringBuilder getExitString() {
    StringBuilder vExits = new StringBuilder("Sorties :");
    Set<String> vExitSet = this.aExits.keySet();
    for(String vE : vExitSet) {
        vExits.append(" " + vE);
    }
    return vExits;
} // getExitString()
```

★ 7.18.3

★ 7.18.4

- Violet Evergarden

★ 7.18.5

```java
// attribut constant et static dans lequel on stoque toutes les Room pour y avoir accès
public static final HashMap<String, Room> GAME_ROOM = new HashMap<String, Room>();
```

Initialisé hors du constructeur pour ne pas avoir de problème de compilation dû au mot final

```java
public class Room {
    private String aDescription; //decrit un lieu
    private String aName;
    private HashMap<String, Room> aExits;

    /**
     * Constructeur naturel
     * @param Décrit la salle
     */
    public Room(final String pDescription, final String pName) {
        this.aDescription = pDescription;
        this.aName = pName;
        this.aExits = new HashMap<String, Room>();
        Game.GAME_ROOM.put(this.aName, this);
    } // Room()
```

- J'ai donc dû créer un attribut `aName` dans la classe `Room` qui stocke le nom donné de la pièce

```java
Room vCompany = new Room("dans la Compagnie postière CH, à Leiden", "Company");
Room vFirstFloor = new Room("au premier étage de la Compagnie", "FirstFloor");
Room vClaudiaOffice = new Room("au bureau de Claudia", "ClaudiaOffice");
Room vCattleyaOffice = new Room("au bureau de Cattleya", "CattleyaOffice");
Room vWorkRoom = new Room("dans la salle de travail de la Compagnie", "WorkRoom");
```

qui est précisé ici en 2e paramètre du constructeur. Le constructeur Room permet de stocker la pièce en question grâce à une clé qui n'est autre que le nom de la pièce.

★ 7.18.6

- Plus besoin de la classe Scanner car cette classe lit dans le terminal, tandis que dans la nouvelle version, on lit à partir de la fenêtre de jeu.

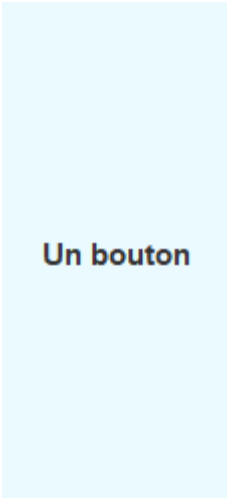
★ 7.18.7

- addActionListener() permet d'ajouter au composant en question un "écouteur d'événement" qui lui est propre. Par exemple pour un bouton, l'événement est le click, pour un champ de texte c'est "entrer", etc…
- actionPerformed() est le bloc de code exécuté lors de l'événement en question

★ 7.18.8

```
//Creation d'un bouton
JButton vBouton1 = new JButton("Un bouton");
vBouton1.setBackground(new Color(235, 250, 255));
```

- Ajout d'un bouton avec un arrière plan coloré qui a comme nom de commande



"un bouton"

```
// Pacement du bouton
vPanel.add(vBouton1, BorderLayout.EAST);
```

- Placement du bouton à l'est du panneau d'affichage

```
vBouton1.addMouseListener(new MouseListener() {
    public void mousePressed(final MouseEvent pE) {}

    public void mouseReleased(final MouseEvent pE) {}

    public void mouseEntered(final MouseEvent pE) {
        vBouton1.setBackground(new Color(230, 230, 255));
        t.println("Dans le bouton");

    }

    public void mouseExited(final MouseEvent pE) {
        vBouton1.setBackground(new Color(235, 250, 255));
        t.println("Hors le bouton");
    }

    public void mouseClicked(final MouseEvent pE) {
        if(pE.getButton() == 1) {
            t.println("Bouton click gauche");
        } else if(pE.getButton() == 3) {
            t.println("Boutton click droit");
        }
    }
});
```

- Ajout de quelques événements souris pour vérifier le bon fonctionnement

★ 7.19

- Le MVC est une architecture logicielle dans laquelle intervient une interface graphique.
- Le MVC peut avoir un lien avec le projet car on utilise une interface graphique
- Les MVC contiennent 3 types de modules, le modèle, la vue et le contrôleur. Le modèle est le module qui contient tout un tas d'informations, des structures de données, etc… La vue est le rendu graphique du modèle, c'est là que l'utilisateur peut visualiser les données. Le contrôleur est le module qui récupère des actions ou des données, et qui modifie ou l'affichage, ou les données du module.
- On a une classe qui peut afficher quelque chose sur la fenêtre, c'est UserInterface, ce serait la Vue dans notre projet. Nos contrôleur sont pour l'instant GameEngine, CommandWords, Command, Parser, … Mais nous n'avons pas de classes qui stockent des données, sauf Room qui stock les sorties des pièces, et dans mon cas, GameEngine qui stock toutes les salles du jeu. Il faudrait alors changer d'emplacement le lieu de stockage des rooms, et créer une ou plusieurs classes qui permettraient de stocker des données.

★ 7.20

```java
public class Item {
    private String aDescription;
    private int aWeight;
    private int aPrice;

    /**
     * Constructeur naturel
     * @param La description de l'objet
     * @param Le poids de l'objet
     * @param Le prix de l'objet
     */
    public Item(final String pDescription, final int pWeight, final int pPrice) {
        this.aDescription = pDescription;
        this.aWeight = pWeight;
        this.aPrice = pPrice;
    } // Item()
```

- Avec les 3 getters associés aux 3 attributs de la classe
- `private Item aItem;` On ajoute un attribut Item dans la classe Room. Il est par défaut null, et on ne l'initialise pas lors de l'appel du constructeur. On le fait grace à une méthode setItem()

```java
/**
 * Ajoute un item à la pièce
 */
public void setItem(final Item pItem) {
    this.aItem = pItem;
} // setItem()
```

- ```java
  Item vTypeMachine = new Item("Machine à écrire", 2000, 10000);
  Item vPaper = new Item("Papier vierge sur lequel on peut écrire", 1, 5);
  ```
  On crée des Items dans la même méthode où on crée les salles, puis on ajoute
  ```

l'Item dans une salle

```
vCompany.setItem(vTypeMachine);
vWorkRoom.setItem(vPaper);
```

★ 7.21

- Dans les Rooms, les informations à propos des Items devraient être produites à partir d'une méthode dans la classe Item renvoyant la description des Items. C'est donc la classe Item qui produit cette information dans une String, et c'est la classe GameEngine qui doit l'afficher, car c'est dans cette classe qu'on retrouve l'attribut aCurrentRoom, à partir de laquelle on veut afficher les informations des Items.

★ 7.21.1

```
/**
 * Permet au joueur d'afficher les informations completes de la salle actuelle
 * Et en cas de second mot, d'afficher les informations sur l'Item correspondant si il existe
 * @param Une commande
 */
private void look(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        String vSWord = pCommand.getSecondWord().toLowerCase();
        boolean vIsItem = CommandWords.isItem(vSWord);
        if(vIsItem) {
            Item vItem = GAME_ITEM.get(vSWord);
            boolean vIsInCurrentRoom = this.aCurrentRoom.getItem().equals(vItem);
            if(vIsInCurrentRoom) {
                this.aGui.println(vItem.getLongDescription());
            } else {
                this.aGui.println("L'objet que je cherche n'est pas ici...");
            }
        } else {
            this.aGui.println("Cet objet n'existe pas...");
        }
    } else {
        this.aGui.println(this.aCurrentRoom.getLongDescription());
    }
} // look()
```

- Amélioration de la commande look qui affiche bien la description complète de l'objet passé en paramètre si il existe et si il est dans la Room actuelle, si il existe mais n'est pas dans la salle actuelle, affiche "L'objet n'est pas ici…", ou qu'il n'existe pas si c'est le cas. Il a fallu ajouter le getter getItem dans la classe Room

```java
/**
 * Verifie si la String en parametre fait parti des objets
 * @return si la String en parametre fait parti des objets
 */
public static boolean isItem(final String pString) {
    for(String pItemName : GameEngine.ITEM_NAMES) {
        if(pItemName.equals(pString)) {
            return true;
        }
    }
    return false;
} // isItem()
```

- Définition d'une méthode statique isItem() dans la classe CommandWords pour y avoir accès depuis la classe GameEngine

```java
// HashMap and Set for Items
public static final HashMap<String, Item> GAME_ITEM = new HashMap<String, Item>();
public static Set<String> ITEM_NAMES;
```

- Création d'une HashMap <String, Item> ainsi que de son keySet(). Les Items sont ajoutés lors de l'appel du constructeur Item(), et le Set est initialisé une fois tous les Items créés

```java
@Override
public boolean equals(Object vObj) {
    if(vObj == null) {
        return false;
    }
    if(vObj.getClass() != this.getClass()) {
        return false;
    }
    Item vItem = (Item)vObj;
    if(vItem.getName().equals(this.aName) && vItem.getDescription().equals(this.aDescription) && (vItem.getPrice() == this.aPrice)) {
        return true;
    }
    return false;
}
```

- Redéfinition de la méthode equals() qui est utilisée pour savoir si l'Item recherché est dans la pièce. Elle vérifie le nom, la description et le prix.

★ 7.22

- On créer dans un premier temps un attribut HashMap<String, Item> dans la classe Room qui contiendra les Items de chaque Room, et on l'initialise dans le constructeur
- On modifie quelques méthodes de la classe Room

```java
/**
 * Renvoie les Items de la Room
 * @return Les Items de la Room
 */
public HashMap<String, Item> getItems() {
    return this.aItems;
} // getItem()
```

```java
/**
 * Renvoie les objets presents dans la Room
 * @return les objets presents dans la Room
 */
public String getItemString() {
    if(this.aItems.size() == 0) {
        return "Pas d'objets ici";
    }
    String vItems = "Objets :";
    for(String vItemName : this.aItems.keySet()) {
        vItems += " " + vItemName;
    }
    return vItems;
} // getItemString()
```

Si il n'y a pas d'objet dans la Room, la taille de la HashMap vaut 0, d'où le if

```java
/**
 * Ajoute un item à la pièce
 */
public void setItem(final Item pItem) {
    this.aItems.put(pItem.getName(), pItem);
} // setItem()
```

-

```java
/**
 * Permet au joueur d'afficher les informations completes de la salle actuelle
 * Et en cas de second mot, d'afficher les informations sur l'Item correspondant si il existe
 * @param Une commande
 */
private void look(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        String vSWord = pCommand.getSecondWord().toLowerCase();
        boolean vIsItem = CommandWords.isItem(vSWord);
        if(vIsItem) {
            Item vItem = GAME_ITEM.get(vSWord);
            boolean vIsInCurrentRoom = vItem.isInRoom(this.aCurrentRoom);
            if(vIsInCurrentRoom) {
                this.aGui.println(vItem.getLongDescription());
            } else {
                this.aGui.println("L'objet que je cherche n'est pas ici...");
            }
        } else {
            this.aGui.println("Cet objet n'existe pas...");
        }
    } else {
        this.aGui.println(this.aCurrentRoom.getLongDescription());
    }
} // look()
```

Modification de la commande look

```java
/**
 * Renvoie si l'Item en parametre est dans la Room
 * @return Si l'Item en parametre est dans la Room
 */
public boolean isInRoom(final Room vRoom) {
    if(vRoom.getItems().size() == 0) {
        return false;
    }
    for(String vItemName : vRoom.getItems().keySet()) {
        if(this.aName.toLowerCase().equals(vItemName.toLowerCase())) {
            return true;
        }
    }
    return false;
} // isInRoom()
```

- Création d'une méthode isInRoom() dans la classe Item qui renvoie si l'Item est dans la Room précisée en paramètre

★ 7.22.1

- J'ai choisi d'utiliser une HashMap pour stocker les Items de chaque salle car ils sont beaucoup plus simples d'accès du fait que l'on cherche généralement un Item par son nom, on cherche donc une collection qui nous permet d'accéder à un Item par son nom et la HashMap est très appropriée pour cela

★ 7.23

- ```java
  private Room aCurrentRoom;
  private Room aPreviousRoom;
  ```
  On crée en même temps que aCurrentRoom la Room précédente aPreviousRoom, qui est null dans un premier temps
- ```java
  this.aPreviousRoom = this.aCurrentRoom;
  ```
  On lui affecte la valeur de la Room actuelle avant de changer de Room, si le changement est possible
- On ajoute dans interpretCommand la condition pour la commande back, ainsi que le nom de la commande dans la classe CommandWords

★ 7.24

- Si le joueur tape un second mot, le programme affiche que cela n'a pas de sens

★ 7.25

- Le problème rencontré lorsque l'on tape 2 fois d'affilé la commande back, on se retrouve à la même pièce qu'initialement car on stocke dans une variable la Room précédente qui devient la Room actuelle, et la Room dans laquelle on était est stockée dans une variable correspondant à la Room précédente. Taper autant de commande back fera tourner en boucle le joueur entre 2 Rooms.

```
/**
 * Commande back
 * @param La commande
 */
private void back(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        this.aGui.println("Ça n'a aucun sens...");
    } else {
        if(this.aPreviousRoom != null) {
            this.aGui.println("Je retourne en arrière");
            this.aCurrentRoom = this.aPreviousRoom;
            this.aPreviousRoom = null; // Un seul retour en arrière possible
            this.aGui.println(this.aCurrentRoom.getLongDescription());
        } else
            this.aGui.println("Je ne peux pas retourner en arrière...");
    }
} // back()
```

- Enfin, la commande back qui ne peut pas prendre de second mot, auquel cas le programme affiche un message. On change de salle et on change aPreviousRoom à null pour pouvoir faire un seul retour en arrière.

★ 7.26

- ```
  private Room aCurrentRoom;
  private Stack aPreviousRooms;
  ```
  On change le système de Room précédente, on la change pour une Stack, une pile. Lors de l'appel de createRooms() on initialise aPreviousRooms en tant que Stack vide : new Stack(), sans oublier d'importer la classe Stack
- ```
  this.aPreviousRooms.push(this.aCurrentRoom);
  ```
  Lorsque le changement de salle est possible, on ajoute en haut de la Stack la Room actuelle grâce à la méthode push(), avant de changer la valeur de celle-ci en celle de la prochaine Room.

```
/**
 * Commande back
 * @param La commande
 */
private void back(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        this.aGui.println("Ça n'a aucun sens...");
    } else {
        if(!this.aPreviousRooms.empty()) {
            Room vPreviousRoom = (Room)this.aPreviousRooms.pop();
            this.aGui.println("Je retourne en arrière");
            this.aCurrentRoom = vPreviousRoom;
            this.aGui.println(this.aCurrentRoom.getLongDescription());
        } else
            this.aGui.println("Je ne peux pas retourner en arrière...");
    }
} // back()
```

- On retire le dernier élément (l'élément en haut de la Stack) grâce à la méthode pop(), qui renvoie un Objet qu'il ne faut pas oublier de convertir pour éviter les erreurs de type. On regarde également avant de changer de salle, si la Stack des précédentes salle est vide, auquel cas on ne peut plus retourner en arrière

★ 7.28.1

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import java.io.FileNotFoundException;
```

- On réalise dans un premier temps les imports nécessaires de classes, pour lire le fichier texte

```
/**
 * Commande de test qui lit un fichier texte
 * @param pCommand La commande
 */
private void test(final Command pCommand) {
    if(!pCommand.hasSecondWord())
        this.aGui.println("Il me faut quelque chose à tester...");
    else {
        String vFileName = pCommand.getSecondWord();
        Scanner vSc;
        try {
            if(vFileName.endsWith(".txt")) {
                vSc = new Scanner(new File(vFileName));
            } else {
                vSc = new Scanner(new File(vFileName + ".txt"));
            }
            this.aGui.println("Fichier teste trouvé");
            while(vSc.hasNextLine()) {
                String vLine = vSc.nextLine();
                this.interpretCommand(vLine);
            }
            this.aGui.println("Il n'y a pas/plus de commandes");
        } catch(FileNotFoundException pE) {
            this.aGui.println(pE.toString());
        } // try catch
    } // if else
} // test()
```

- Après avoir bien-sûr ajouter la commande “test” dans la classe CommandWords ainsi que la condition adéquate dans la méthode interpretCommand(), on créer la commande de test comme cela. On regarde d’abord si un deuxième mot est présent dans la commande, si ce n’est pas le cas, alors il ne se passe rien de particulier. Dans le cas contraire, on récupère le second mot pour chercher si le fichier spécifié existe ou pas. 2 cas possibles, si le second mot se termine par l’extension “.txt”, alors on n’a pas besoin de le rajouter lors de la recherche du fichier, dans l’autre cas, il faut le rajouter après le nom du fichier. Si le fichier spécifié n’existe pas, une erreur survient mais ne met pas fin au programme grâce au try/catch. Si il existe, on regarde s' il existe une ligne dans ce fichier, et tant qu’il y en a une, on récupère cette ligne dans une String pour pouvoir en interpréter la commande avec la méthode interpretCommand(), jusqu’à ce qu’il n’y en n’ait plus.

★ 7.28.3

```
// COMMANDES DE TEST
/**
 * Regarde si la Room en parametre est la meme que la Room actuelle
 * @param pCommand la commande
 */
private void roomIs(final String pCommand) {
    Room vRoom = GAME_ROOM.get(pCommand.toLowerCase());
    if(this.aCurrentRoom.equals(vRoom))
        this.aGui.println("true");
    else
        this.aGui.println("false");
}

/**
 * Regarde si la Room en parametre est la meme que la Room actuelle
 * @param pCommand la commande
 */
private void roomHas(final String pCommand) {
    Item vItem = GAME_ITEM.get(pCommand.toLowerCase());
    if(vItem.isInRoom(this.aCurrentRoom))
        this.aGui.println("true");
    else
        this.aGui.println("false");
}
```

- création de méthodes de test, 2 pour l'instant qui n'apparaissent pas dans la classe CommandWords, mais uniquement dans la methode test, elles ne sont appelées que dedans pour ne pas surcharger interpretCommand() de méthodes inutiles au jeu. On redéfinit la méthode equals() dans la classe Room qui vérifie et le nom et la description, et pour les Item, une méthode déjà fonctionnelle a été présentée dans le rapport

★ 7.29

- Après avoir créé une classe Player, on définit des constantes pour regrouper les joueurs et les PNJ du jeu

```
// HashMap and Set for PLayers
public static final HashMap<String, Player> GAME_PLAYER = new HashMap<String, Player>();
public static Set<String> PLAYER_NAMES;
public static final HashMap<String, Player> GAME_PNJ = new HashMap<String, Player>();
public static Set<String> PNJ_NAMES;

// Main Character
private MC aMC;
```

- On crée donc 2 sous classes de Player, une qu'on appelle MC pour Main Character, et une autre PNJ, ce qui aura l'avantage de différencier facilement les actions et interactions que subissent tous les personnages en fonction de si c'est un PNJ ou pas.
- Les PNJ pouvant eux même faire des actions, on ne veut pas que les commandes qu'ils effectuent affichent du texte. On définit donc tout un tas de procédure dans GameEngine qui afficheront chacune un texte en particulier, comme celle-ci par exemple

```
public void cannotGoHere() {
    this.aGui.println("Je ne peux pas aller là...");
}
```

```
// Creation of Players
PNJ vClaudia = new PNJ("Claudia", vCompany, 50);
PNJ vGilbert = new PNJ("Gilbert", vCompany, 50);

aMC = new MC("MC", vBociaccia, 50, this);

// Creation of Players Set

this.PNJ_NAMES = this.GAME_PNJ.keySet();
this.PLAYER_NAMES = this.GAME_PLAYER.keySet();
```

- On initialise le MC et les PNJ dans la méthode createRooms() ainsi que les Set de Player et de PNJ

```
public class Player {
    private String aName;
    private Room aCurrentRoom;
    private String[] aInventory;
    private double aMaxWeight;
    private double aCurrentWeight;
    private Stack aPreviousRooms;

    public Player(final String pName, final Room pCurrentRoom, final double pMaxWeight) {
        this.aName = pName;
        this.aCurrentRoom = pCurrentRoom;
        this.aMaxWeight = pMaxWeight;
        this.aCurrentWeight = 0;
        this.aPreviousRooms = new Stack();
    }
```

- Et voici la classe Player, sans montrer tous les getters et les quelques setters de la classe

★ 7.30

- Modification dans la classe Player pour l'inventaire, on autorise un unique Item pour le moment

```
public class Player {
    private String aName;
    private Room aCurrentRoom;
    private Item aInventory;
    private double aMaxWeight;
    private double aCurrentWeight;
    private Stack aPreviousRooms;

    public Player(final String pName, final Room pCurrentRoom, final double pMaxWeight) {
        this.aName = pName;
        this.aCurrentRoom = pCurrentRoom;
        this.aMaxWeight = pMaxWeight;
        this.aCurrentWeight = 0;
        this.aPreviousRooms = new Stack();
    }
```

- Ajout dans la classe CommandWords le nom des commandes "tak", "drop" et "info" qui sera expliquée par la suite. Ajout ensuite dans la méthode interpretCommands() les possibilités pour "take", "drop" et "info"

```java
/**
 * Permet de récupérer un Item et de le mettre dans son inventaire, si celui-ci existe et si
 * il se trouve dans la même Room que le joueur
 */
public void take(final Command pCommand) {
    if(!pCommand.hasSecondWord()) {
        this.GE.noSens();
        return;
    }
    String vItemName = pCommand.getSecondWord();
    if(CommandWords.isItem(vItemName)) {
        Item vItem = GameEngine.GAME_ITEM.get(vItemName.toLowerCase());
        if(vItem.isInRoom(this.getCurrentRoom())) {
            if(this.getInventory() != null) {
                this.GE.fullInventory();
                return;
            }
            this.GE.itemInRoom(vItem.getName());
            this.setInventory(vItem);
            this.addWeight(vItem.getWeight());
            this.getCurrentRoom().removeItem(vItem);
        } else {
            this.GE.itemNotInRoom();
        }
    } else {
        this.GE.notItem();
    }
} // take()
```

- On regarde si la commande contient bien un second mot. Si c'est le cas, on vérifie l'existence de l'item, et si c'est le cas, on vérifie la présence de l'Item dans la Room actuelle du joueur. Si c'est encore le cas, on change l'inventaire du joueur, et on retire l'Item de la HashMap qui stock les Items de la Room

```java
/**
 * Enlève l'Item de l'inventaire pour le placer dans la HashMap
 * de la Room actuelle qui stock les Items
 */
public void drop(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        this.GE.noSens();
        return;
    }
    if(this.getInventory() == null) {
        this.GE.emptyInventory();
        return;
    }
    this.getCurrentRoom().setItem(this.getInventory());
    this.removeWeight(this.getInventory().getWeight());
    this.setInventory(null);
} // drop()
```

- La commande "drop" reprend le même principe mais à l'envers. On ne vérifie pas l'existence de l'Item pour l'instant cas l'inventaire est un unique Item.

```
public void playerInfo(final Player pPlayer) {
    String vInfo = "Nom : " + pPlayer.getName() + "\n";
    vInfo += "Lieu actuel : " + pPlayer.getCurrentRoom().getName() + "\n";
    vInfo += "Poids : " + pPlayer.getCurrentWeight() + "/" + pPlayer.getMaxWeight() + "\n";
    vInfo += "Inventaire : ";
    if(pPlayer.getInventory() != null) {
        vInfo += "\n";
        vInfo += "   - " + pPlayer.getInventory().getName();
    } else {
        vInfo += "sac vide \n";
    }
    this.aGui.println(vInfo);
} // playerInfo()
```

- On crée une commande d'info pour les Player qui affiche les infos du joueur si il n'y a pas de second mot, ou celles d'un pnj si il y a un second mot et que celui-ci existe

★ 7.31

```
public class Inventory {
    private HashMap<String, Item> aItems;
    private HashMap<Item, Integer> aStacked;
    public Inventory() {
        this.aItems = new HashMap<String, Item>();
        this.aStacked = new HashMap<Item, Integer>();
    }
```

- Pour gérer facilement le stockage des Items, on peut généraliser les inventaires des Player et des Room grâce à une classe Inventory, qui stocke de la même manière avec une HashMap<String, Item> les Items des Players et des Rooms. La différence est que j'ai rajouté un moyen de stacker des Items pour ne pas se retrouver avec plusieurs fois le même Item dans l'Inventory (cette fonctionnalité sera complétée ultérieurement)

- Ainsi les Player dont le MC peut stocker autant d'Items qu'il le veut, tant que la charge maximale n'est pas atteinte, comme le montre le code suivant

```
/**
 * Permet de récupérer un Item et de le mettre dans son inventaire, si celui-ci existe, si
 * il se trouve dans la même Room que le joueur et si la charge maximale n'est pas dépassée
 */
public void take(final Command pCommand) {
    if(!pCommand.hasSecondWord()) {
        this.GE.noSens();
        return;
    }
    String vItemName = pCommand.getSecondWord().toLowerCase();
    if(CommandWords.isItem(vItemName)) {
        Item vItem = this.getCurrentRoom().getInventory().getItems().get(vItemName);
        if(vItem.isInRoom(this.getCurrentRoom())) {
            if(this.getCurrentWeight()  + vItem.getWeight() > this.getMaxWeight()) {
                this.GE.fullInventory();
                return;
            }
            this.GE.itemCollected(vItem.getName());
            this.getInventory().addItem(vItem);
            this.addWeight(vItem.getWeight());
            this.getCurrentRoom().getInventory().removeItem(vItem);
        } else {
            this.GE.itemNotInRoom();
        }
    } else {
        this.GE.itemNotExist();
    }
} // take()
```

- `private Inventory aInventory;` J'ai donc remplacé le simple Item en tant qu'inventaire par un objet de type Inventory pour less Player et les Rooms, ajouté le getter qui convient et modifié en conséquence quelques méthodes

```java
/**
 * Enlève l'Item de l'inventaire pour le placer dans la HashMap
 * de la Room actuelle qui stock les Items
 */
public void drop(final Command pCommand) {
    if(!pCommand.hasSecondWord()) {
        this.GE.noSens();
        return;
    }
    if(this.getInventory().empty()) {
        this.GE.emptyInventory();
        return;
    }
    String vItemName = pCommand.getSecondWord().toLowerCase();
    if(CommandWords.isItem(vItemName)) {
        Item vItem = this.getInventory().getItems().get(vItemName);
        if(vItem == null) {
            this.GE.itemNotInInventory();
            return;
        }
        this.getCurrentRoom().getInventory().addItem(vItem);
        this.removeWeight(vItem.getWeight());
        // Condition si l'objet est stackable, retirer une occurence
        this.getInventory().getItems().remove(vItemName);
        this.GE.itemDroped(vItem.getName());
    } else {
        this.GE.itemNotExist();
    }
} // drop()
```

- Et voici la commande drop

★ 7.31.1

- Détails de la classe Inventory ⇔ ItemList

```java
public class Inventory {
    private HashMap<String, Item> aItems;
    private HashMap<Item, Integer> aStacked;
    public Inventory() {
        this.aItems = new HashMap<String, Item>();
        this.aStacked = new HashMap<Item, Integer>();
    }
```

```java
/**
 * Renvoie la HashMap de l'inventaire
 * @return la HashMap de l'inventaire
 */
public HashMap<String, Item> getItems() {
    return this.aItems;
}

/**
 * Renvoie si l'inventaire est vide ou pas
 * @return si l'inventaire est vide ou pas
 */
public boolean empty() {
    return this.aItems.isEmpty();
}
```

```java
/**
 * Ajoute un Item dans l'inventaire. Si celui-ci est stackable
 * on crée un élément <Item, Integer> dans la HashMap aStacked si il n'existe pas encore
 * ou on lui ajoute une occurence si il existe déjà. Si le nombre d'occurence est à 0
 * l'Item n'est plus dans l'Inventaire.
 * @param pItem Item que l'on veut mettre dans l'Inventaire
 */
public void addItem(final Item pItem) {
    // if(!pItem.isStackable()) {
        this.aItems.put(pItem.getName().toLowerCase(), pItem);
        return;
    // }
}
```

```java
/**
 *
 */
public void removeItem(final Item pItem) {
    this.aItems.remove(pItem.getName().toLowerCase());
}
```

- Le système rendant possible le stacking d'Item sera complété ultérieurement

★ 7.32

-
```java
private double aMaxWeight;
private double aCurrentWeight;
```
Voici les deux attributs de la classe Player qui gèrent le poids porté et le poids max portable par le joueur

```
if(vItem.isInRoom(this.getCurrentRoom())) {
    if(this.getCurrentWeight()  + vItem.getWeight() > this.getMaxWeight()) {
        this.GE.fullInventory();
        return;
    }
```

- Grâce à cette condition dans la commande take, le joueur ne peut pas prendre d'objet si le poids de celui-ci en plus du poids actuel dépasse le poids max

★ 7.33

```
public String items() {
    String vInfo = "";
    if(!this.aInventory.empty()) {
        // à modifier quand les objets pourront être stackés
        vInfo += this.aInventory.getItems().size() + " objets (" + this.aCurrentWeight + "/" + this.aMaxWeight + ")";
        HashMap<String, Item> vItems = this.aInventory.getItems();
        for(String vIName : this.aInventory.getItems().keySet()) {
            vInfo += "\n   - " + vItems.get(vIName).getName() + " ~ " + vItems.get(vIName).getWeight() + " kg ~ " + vItems.get(vIName).getPrice() + " ¥";
        }
    } else {
        vInfo += "sac vide \n";
    }
    return vInfo;
} // items
```

★ 7.34

- Étant donné que l'on a plusieurs Items dans le jeu, on veut savoir si celui-ci est mangeable avant de pouvoir le manger. J'ai donc créé un attribut dans la classe Item ainsi que son getter associé

```
private boolean aStackable;
```

```
/**
 * Permet au joueur de consommer de la nourriture présente dans l'inventaire
 * et lui octroyer les effets associés
 * @param pCommand La commande
 */
public void eat(final Command pCommand) {
    if(!pCommand.hasSecondWord()) {
        this.GE.cannotEat();
        return;
    }
    String vIName = pCommand.getSecondWord().toLowerCase();
    ItemList vItemList = this.getItemList();
    Item vItem = vItemList.getItems().get(vIName);
    if(!CommandWords.isItem(vIName)) {
        this.GE.itemNotExist();
    } else if(vItem == null) {
        this.GE.itemNotInInventory();
    } else if(!vItem.isEatable()) {
        this.GE.itemNotEatable();
    } else {
        if(vIName.equals("cookie")) {
            this.setMaxWeight(this.getMaxWeight() * 2);
            this.GE.itemEaten(vItem.getName());
            vItemList.removeItem(vItem);
        } // else if...
        this.removeWeight(vItem.getWeight());
    } // else if
} // eat()
```

- Et voici comment se présente ma commande eat. On regarde s'il y a un second mot, si oui, je crée les variables qui me seront utiles puis je fais une série de tests pour savoir si l'objet existe et s' il est bien dans l'inventaire du joueur. Puis avec l'arrivée de prochains Items mangeables, je traiterai quel Item doit être mangé avec une série de else if

★ 7.42

```java
/**
 * Nombre d'action effectué
 */
public int aNbActions;

/**
 * Nombre max d'action faisable
 */
public int aMaxActions;
```

- On crée dans GameEngine deux nouveaux attributs, un qui sert à compter le nombre d'action effectué, et un autre qui détermine le nombre maximum d'action effectué. Or on ne veut pas rajouter 1 à aNbActions si la commande effectuée est erronée, par exemple si on écrit "go nooord". Il faut donc faire en sorte que les commandes/méthodes de Player renvoient un booléen pour savoir si l'action a bien été effectuée.

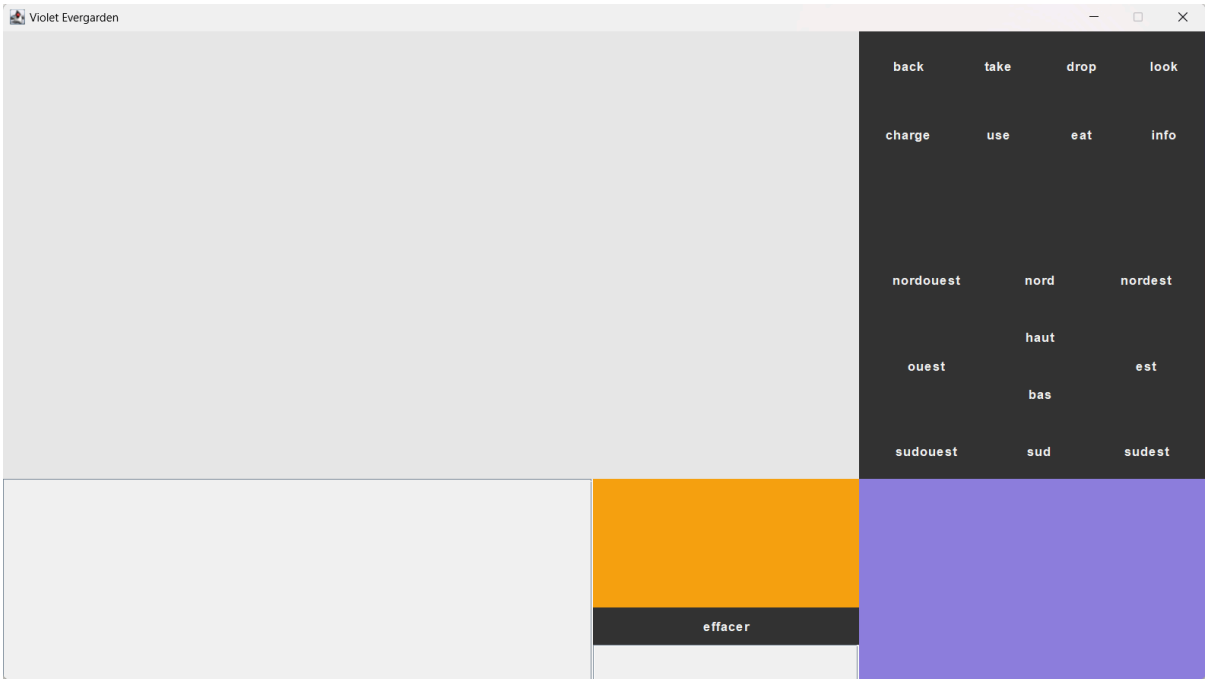
```java
boolean vAction = false;
if (vCommandWord.equals("help")) {
    vAction = this.aMC.printHelp();
} else if (vCommandWord.equals("go")) {
    vAction = this.aMC.goRoom(vCommand);
} else if(vCommandWord.equals("look")) {
    vAction = this.aMC.look(vCommand);
} else if(vCommandWord.equals("eat")) {
    vAction = this.aMC.eat(vCommand);
```

- On doit donc stocker cette valeur dans la méthode interpretCommands(), dans une variable vAction par exemple.

```java
if(vAction) {
    this.pnjActions();
}
if(this.aNbActions > this.aMaxActions) {
    this.aGui.println("Je n'ai plus d'espoir, j'arrête mes recherches...");
    this.endGame();
}
```

- À la fin des else if, on regarde si l'action a bien été effectuée, et si oui on appelle une méthode qui fait faire une action aux PNJ, qui elle même ajoute 1 à aNbActions. Et à la fin, on vérifie si ne nombre d'action effectuées dépasse le nombre d'actions maximum, auquel cas le jeu s'arrête

★ 7.42.2

- 

  - Voilà mon IHM actuelle, que je n'ai pas encore implémenté dans le projet pour ne pas avoir à tout modifier tout de suite. C'est une IHM que je fais dans un projet externe que j'implémenterai dans le projet Zuul ultérieurement
  - Il y a donc un JPanel principal avec un GridBagLayout dans lequel j'ai mis un JPanel pour l'Image, pour les boutons, pour une image en guise de carte, pour un JTextArea, un JTextField, pour un bouton et pour une autre image certainement
  - La carte serait donc dans le JPanel en violet, et l'autre image dans le JPanel orange.
  - Dans le JPanel des boutons qui a un Borderlayout, j'ai mis au sud un JPanel avec un GridBagLayout pour y mettre tous les boutons directionnels, et au nord, un JPanel avec un GridBagLayout pour y mettre toutes les autres commandes utiles aux jeu (j'exclu les commandes quit, test, etc…)

★ 7.43

- Plusieurs trap door étant déjà implémentées, on va s'intéresser uniquement à la commande back

```java
public boolean back(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        this.GE.noSens();
        return false;
    }
    if(!this.getPreviousRooms().empty()) {
        Room vPreviousRoom = (Room)this.getPreviousRooms().pop();
        if(this.getCurrentRoom().isExit(vPreviousRoom)) {
            this.GE.goBack();
            this.setCurrentRoom(vPreviousRoom);
            this.GE.longDescription(this.getCurrentRoom());
        } else {
            this.GE.cannotGoBack();
        }
        return true;
    }
    this.GE.cannotGoBack();
    return false;
} // back()
```

- Dans un premier cas, si la commande a un second mot, cela n'a pas de sens, il ne se passe rien et l'action n'est pas comptée comme effectuée. Dans un second cas, si la pile qui stocke les précédentes Room est vide, c'est à dire si on revient à la Room de départ via la commande back, on ne peut plus revenir en arrière et l'action n'est pas comptée comme effectuée. Dans les autres cas, l'action est comptée comme effectuée. On regarde si la Room en haut de la pile fait partie des sorties de pa Room actuelle, et si c'est le cas, back est possible, et impossible sinon.

★ 7.44

```java
public class Beamer extends Item {
    /**
     * Attribut qui stocke la Room dans laquelle le Player se téléportera à
     * l'utilisation du Beamer
     */
    private Room aChargedRoom;
```

- Pour le Beamer, on crée donc une sous class d'Item avec un attribut en plus, qui stocke la Room qu'on aura chargé avec la commande charge

```java
/**
 * Charge le Beamer avec la Room en paramètre
 * @param pRoom La Room à charger
 */
public void chargeBeamer(final Room pRoom) {
    this.aChargedRoom = pRoom;
} // chargeBeamer()

/**
 * Renvoie la Room chargée
 * @return La Room chargée
 */
public Room getChargedRoom() {
    return this.aChargedRoom;
} // getChargedItem()

/**
 * Renvoie si le Beamer est chargé
 * @return Si le Beamer est chargé
 */
public boolean isCharged() {
    return this.aChargedRoom != null;
} // isCharged()
```

- Voici les 3 méthodes qui constituent les Beamer, une qui permet de stocker la Room, une qui permet de la récupérer, et une autre pour savoir si elle est chargée

```java
public boolean charge(final Command pCommand) {
    if(pCommand.hasSecondWord()) {
        this.GE.noSens();
        return false;
    }
    Beamer vBeamer = (Beamer)this.getItemList().getItems().get("beamer");
    if(vBeamer != null) {
        vBeamer.chargeBeamer(this.getCurrentRoom());
        this.GE.beamerCharged();
        return true;
    }
    this.GE.beamerNotInInventory();
    return true;
}
```

- Pour charger le beamer, on regarde si la commande possède un second mot ou pas. Si non, on récupère le beamer depuis la ItemList du MC grâce à la méthode get(), sans oublier de le convertir en Beamer car les Objets stockés sont des Items. Si l'Objet renvoyé (le Beamer renvoyé) vaut null, c'est qu'il n'est pas dans l'ItemList du MC, et s'il ne vaut pas null, il existe bien dans l'ItemList du MC, et la charge peut bien s'effectuer.

★ 7.46

- ```java
  public class TransporterRoom extends Room {
  ```

- On crée une classe TransporterRoom qui hérite de Room, sans nouvel attribut particulier
  ```

```java
/**
 * Renvoie une Room aléatoire parmis celles possibles
 * @return Une Room aléatoire
 */
public Room getRandomExit() {
    return this.findRandomRoom();
}

/**
 * Renvoie une Room aléatoire parmis celles possibles
 * @return Une Room aléatoire
 */
public Room findRandomRoom() {
    Object[] vArrayRooms = GameEngine.GAME_ROOM.values().toArray();
    Random vRdm = new Random();
    Room vRoom;
    do {
        vRoom = (Room)vArrayRooms[vRdm.nextInt(vArrayRooms.length)];
    }
    while(vRoom.getName().equals("Ekarte") || vRoom.getName().equals("Observatory") || vRoom.getName().equals("Ctrigall"));
    return vRoom;
}
```

- On définit les deux méthodes spécifiques aux TransporterRoom. La première qui renvoie la Room, et la deuxième qui détermine quelle est la prochaine Room aléatoire. A première vue, la méthode getRandomExit() n'est pas nécessaire étant donné qu'elle ne fait qu'appeler la méthode findRandomRoom(), mais pour des soucis de meilleur compréhension, la première s'avère utile.
- Dans dindRandomRoom(), on récupère toutes les Room du jeu stockées dans GameEngine en tant que constante, puis on convertit cette HashMap en tableau d'Object. On crée un nouvel objet Random, sans oublier l'import, puis on déclare vRoom la Room à renvoyer, null dans un premier temps. Ensuite, on écrit une boucle do while, qui affecte un Objet à vRoom aléatoirement depuis le tableau d'Object, il faut donc convertir cet Object en Room. Le nombre aléatoire généré est un nombre compris entre 0 inclus et la taille du tableau exclu. Le while permet d'éviter que vRoom prenne certaines valeurs. Ici j'ai utilisé des "ou" ("||") car le nombre de Room à éviter n'est que de 3, mais si ce nombre était plus grand, j'aurai créé une HashMap qui regroupe toutes les Room qui peuvent être choisies aléatoirement, et j'aurai créé mon tableau d'Object à partir de cette HashMap.

III) Mode d'emploi

IV) Déclaration anti plagiat

- ★ Carte du jeu reprise du site https://violet-evergarden.fandom.com/wiki/Category:Locations à laquelle j'ai effacé les noms, et à laquelle je rajouterai les miens.
- ★ Inventaire
  - Pour effectuer n'importe quelle quête, doit avoir au moins une feuille vierge dans l'inventaire. Les feuilles vierges peuvent uniquement se trouver à la Compagnie
- ★ A FAIRE
  - Images du jeu
  - Items du jeu
  - commandes dans les fichiers texte court.txt et long.txt
  - les commandes/méthodes de test à déplacer dans la classe Player ?
  - commande eat à compléter plus tard lorsqu'il y aura des consommables
  - méthode pnjActions() dans GameEngine pour lire les fichiers texte des PNJ et effectuer leur action suivante => créer un interpretCommand() pour les PNJ ?

- Ajouter les pnj + fichier text pour pnj