Objectif : Différencier et utiliser plusieurs paradigmes de programmation.

1 Problématique : Fortnite

Chaque combattant de Fortnite est extrêmement personnalisable. C'est une des caractéristiques qui fait le succès du jeu.
Il est ainsi possible de choisir les éléments : outfit, glider, contrail, back bling, emote, pickaxe, toy.

			
outfit	glider	emote	pickaxe

TABLE 1 – Exemples d'éléments

Il faut ajouter à cela les objets récupérés en cours de partie et gérer l'évolution de l'énergie et du bouclier du personnage.

> Quel concept de programmation mettre en place pour manipuler les combattants et les faire évoluer dans une partie ?

2 Un concept déjà utilisé : paradigme impératif

Retour historique

Un paradigme de programmation est un style de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation. La programmation impérative (du latin imperare : ordonner) est une séquence ordonnée d'instructions, fournies à l'ordinateur. premiers programmes = bas niveau = instructions aux plus proches de la machine (lis cette adresse mémoire...). Programmation impérative se calque sur le schéma de fonctionnement des ordinateurs (Von Neumann voir cours 1ere).
1951 : Grace Hopper 1° compilateur (A-0) = traduire programme en langage machine, puis langage Cobol en 1959 (programme doit être calqué sur langage humain plutôt que machine)
1954 : FORTRAN FORmula TRANslator ; reste encore très utilisé (même si remplacé par C, C++) car nombreuses bibliothèques, coût important pour les remplacer, performant.
1991 : Fortran 90 : modernisation, modules, récursivité
1963 : BASIC : dérivé du Fortran pour les débutants, l'apprentissage ; mais trop simple, maintenance difficile ; Djikstra écrit article assassin ; fait disparaître au profit du Pascal
1970 : Pascal ; écrit par Niklaus Wirth ; très rapide car travaillant principalement en mémoire vive ; Turbo Pascal dans les années 80.
1972 : C ; crée par Dennis Ritchie (B trop limité) pour dvp système UNIX ; avec Ken Thomson.

Christophe Viroulaud

2.1 Stocker les informations

Différentes structures de données abordées en Première permettent de contenir les informations du combattant.

Activité 1 :

1. Proposer un type (construit) de données et construire la structure permettant de stocker les éléments qui composent un combattant.
2. Ajouter une structure permettant de stocker et gérer les objets du sac à dos.
3. Ajouter enfin deux variables exprimant l'énergie et le bouclier. Ces variables seront initialisées à *100*.

Itération sur un dictionnaire. Pas de garantie d'ordre (selon version de Python <3.7).
https://docs.python.org/fr/3/tutorial/datastructures.html#looping-techniques

2.2 Modifier les informations

Lors d'une partie le joueur peut ramasser des objets pour les mettre dans son sac. Il peut également subir des dégâts lors d'une attaque.

Activité 2 :

1. Implémenter une fonction *subir_attaque* qui prendra un paramètre *degat* de type *integer* et qui modifiera les caractéristiques *shield* et *energy* du combattant.
2. Implémenter une fonction *ramasser_objet* qui ajoute un objet dans le sac s'il n'est pas plein (trois objets maximum). Cette fonction prendra un paramètre *objet* de type *string*.
3. Dans Fortnite chaque objet a également plusieurs propriétés. Construire un type construit de données qui stocke les caractéristiques des armes présentées.

ATTENTION à éviter les variables globales !

Limites du paradigme

Les fonctions définies influent sur le combattant, mais d'autres items du jeu évoluent également pendant la partie. Les armes par exemple ont un nombre de munitions variables, le parachute peut être ouvert ou fermé.
De plus, comment donner une arme à chaque combattant ? Le système se complexifie alors, d'où l'intérêt d'introduire un nouveau paradigme.

3 Une nouvelle approche : paradigme objet

Christophe Viroulaud

Retour historique

Dvpt informatique = pb de + en + complexe à résoudre (*abstraction*)
- impératif pas forcément suffisant ou pratique
- modélisation d'une application informatique sous la forme d'objets, ayant des propriétés et pouvant interagir entre eux. La modélisation orientée objet est proche de la réalité ce qui fait qu'il sera relativement facile de modéliser une application de cette façon.
- De plus, les personnes non-techniques pourront comprendre et éventuellement participer à cette modélisation.

1960 : SIMULA = plutôt introduction des concepts
1971 : Les travaux de ces deux informaticiens ont ensuite influencé ceux d'Alan Kay, informaticien américain à l'origine du langage Smalltalk dans les années 70. A l'époque, Alan Kay travaillait au laboratoire PARC de Xerox en Californie. C'est dans ce même laboratoire qu'est apparu (en 1973) le premier ordinateur utilisant un système d'exploitation avec interface graphique, la Station Xerox Alto qui a ensuite inspiré le Mac Intosh, puis le système d'exploitation Windows de Microsoft. Il faut remarquer que les travaux d'Alan Kay portaient simultanément sur la conception d'interface graphique et la programmation objet. Cette nouvelle manière de programmer était pour lui une manière élégante de représenter informatiquement les interfaces graphiques.

3.1 Décrire une classe

3.1.1 Définition

Une *classe* définit une nouvelle structure de données. Elle regroupe plusieurs composantes de natures différentes. Dans notre cas c'est le combattant. Chaque langage implémentant la programmation orienté objet défini une syntaxe propre.

```
1 class Combattant:
```

La classe peut être vue comme le squelette générique du combattant.

Les bonnes pratiques (PEP8 - Python Enhancement Proposals = propositions d'amélioration) conseillent de mettre une majuscule pour les noms de class.

3.1.2 Ajouter des attributs

La fonction *__init__* est appelée automatiquement quand nous créons un combattant. Chaque caractéristique est représentée par une variable propre à la classe : *un attribut*.

```
1 class Combattant:
2       """Crée un combattant avec ses caractéristiques propres"""
3       def __init__(self,tenue,parachute,trainee):
4             self.outfit=tenue
5             self.glider=parachute
6             self.contrail=trainee
```


Christophe Viroulaud

Nous commençons ici avec seulement 3 attributs.
Remarquez l'utilisation du mot dédiée *self* en Python. La syntaxe peut être différente dans d'autres langages (javascript = this, C++ = rien). Pourquoi *self* ? permet de faire référence explicite à l'instance de la classe en cours. Différencie variable locale des autres.
Enfin on aurait pu initialiser les attributs dans un second temps et pas directement dans le ___init___.

3.2 Créer un objet

3.2.1 Créer un combattant

La classe est la structure (le squelette) d'un combattant. Pour l'instant aucun joueur n'a été construit. On crée une *instance* de la classe *Combattant* et on la stocke dans la variable *joueur1*.

```
1 joueur1=Combattant("cuddle-team-leader","hot-rod","flames")
```

Activité 3 :

1. Compléter la classe *Combattant* en ajoutant toutes les caractéristiques.
2. Créer un nouvel attribut permettant de stocker le contenu du sac à dos.

3.2.2 Manipuler les attributs

Nous pouvons consulter les valeurs d'un attribut mais aussi les modifier.

```
1 >>> joueur1.outfit
2 cuddle-team-leader
3 >>> joueur1.glider="pterodactyl"
```

En Python nous avons pouvons accéder à tout le contenu de la classe. Ce n'est pas le cas dans tous les langages : la POO est souvent associée à la notion *d'encapsulation*. Le programmeur peut vouloir masquer le contenu d'implémentation. Par convention en Python les attributs privés sont précédés d'un _.

3.3 Manipuler les données

La manipulation de l'objet passe de préférence par une interface constituée de fonctions définies dans la classe : *les méthodes*.

```
1  class Combattant:
2      """Crée un combattant avec ses caractéristiques propres"""
3      def __init__(self,tenue,parachute,trainee):
4          self.outfit=tenue
5          self.glider=parachute
6          self.contrail=trainee
7
8      def get_outfit(self):
9          return self.outfit
10
11     def set_outfit(self,nouvelle):
12         self.outfit=nouvelle
```


Christophe Viroulaud

getters (accesseurs) et setters (mutateurs)

Activité 4 :

1. Ajouter les attributs *energy, shield* et *backpack*.
2. Adapter les fonctions *subir_attaque* et *ramasser_objet* pour en faire des méthodes de la classe *Combattant*.

Et Python dans tout ça ; quel paradigme ?

3.4 Le monde est objet

Activité 5 :

1. Créer un objet *Weapon* possédant les attributs présentés précédemment.
2. Ajouter une méthode *tirer* qui vide une balle du magasin. Cette méthode renverra le texte *Chargeur vide !* quand toutes les balles auront été tirées.
3. Implémenter une méthode *recharger* qui remplit le magasin.

Comment gérer le nombre de balles à recharger ? un attribut *cachée* qui enregistre le nombre de balles à la création de l'arme.

Christophe Viroulaud