Projet C++

Paul-Edouard Graves

Edouard Matheu

2020 - 2021

POLYTECH SORBONNE

Sommaire

Description de l'application

Dans le monde d'après, les casinos et les paris sportifs sont devenus obsolètes. Les hommes préfèrent désormais parier sur des combats entre animaux et plus particulièrement ceux avec des singes : c'est le début de *L'Arène des Singes.* Notre application est donc un jeu où chaque joueur peut parier sur un singe et voir suivant le combat qui a remporté sa mise ou non. Ce jeu peut se jouer à plusieurs (plusieurs participants) ou tous seul (contre une IA). Au début du jeu chaque joueur se voit attribuer une somme aléatoire entre 1 et 100, puis chacun doit parier durant son tour sur un singe avec une mise dont le bénéfice augmente en fonction de la quantité qu'il parie et suivant le résultat du combat. Un joueur a perdu quand sa somme de départ vaut 0. La partie s'arrête quand il reste en « vie » ou que tous les joueurs ont quitté la partie.

Utilisation des contraintes

Notre application se compose de *14 classes* avec une *hiérarchie sur 3 niveaux* (voir le diagramme UML).

Notre classe *Singe*, est une *classe abstraite* (classe mère) qui permettra de faire hériter les *attributs* et les *méthodes* aux différents singes. Cette classe possède donc *5 méthodes virtuelles pures* qui seront redéfinies dans les classes singes qui héritent de cette classe.

```
virtual void combat(Singe& adversaire, int i) = 0;
virtual void soin(Singe& a) = 0;
virtual void soin(int luckSinge) = 0;
virtual bool vie() = 0;
virtual void resetPA() = 0;
```

Nous avons également utilisé 2 conteneurs de la STL, à savoir un *vecteur* qui permet de stocker les différents joueurs de la partie `std :: vector <Joueur> tab;` (se situe dans la classe *Jeu*) et un *pair* qui permet d'associer le nom du joueur et le singe sur lequel le joueur a décidé de parier `std :: pair <std :: string, Singe*> player;`

Nous avons seulement qu'une seule surcharge d'opérateur car on n'a pas trouvé d'autres qui pouvaient être utiles à notre application. La surcharge de l'opérateur *operator<<* est définit plusieurs fois pour afficher différentes caractéristiques importantes pour le déroulement de la partie. On peut le retrouver dans les classes : *Singe* et *Joueur.*

```
friend std :: ostream& operator<<(std :: ostream& os, Singe& a)
{
    os << "Stats " << a.nom << std :: endl;
    os << "PDV : " << a.pdv << std :: endl;
    os << "PDC : " << a.pdc << std :: endl;
    os << "PDS : " << a.pds << std :: endl;
    os << "Armure : " << a.armure << std :: endl;
    os << "Point d'action : " << a.pa << std :: endl;
    os << "Type : " << a.sigle;
    os << std :: endl;

return os;
};
```

Nous avons néanmoins essayé d'utiliser l'opérateur *operator==*, pour comparer si un joueur avait perdu la partie mais cela n'a fonctionné.

Dans les tests, on peut retrouver les différents tests unitaires que nous avons réalisés à savoir si un combat entre singe se fait normalement, qu'un singe peut combattre/soigner qu'un singe qui est vivant, ne pas pouvoir miser plus que ce que l'on a et enfin voir si les gains obtenus en cas de victoire sont les bons

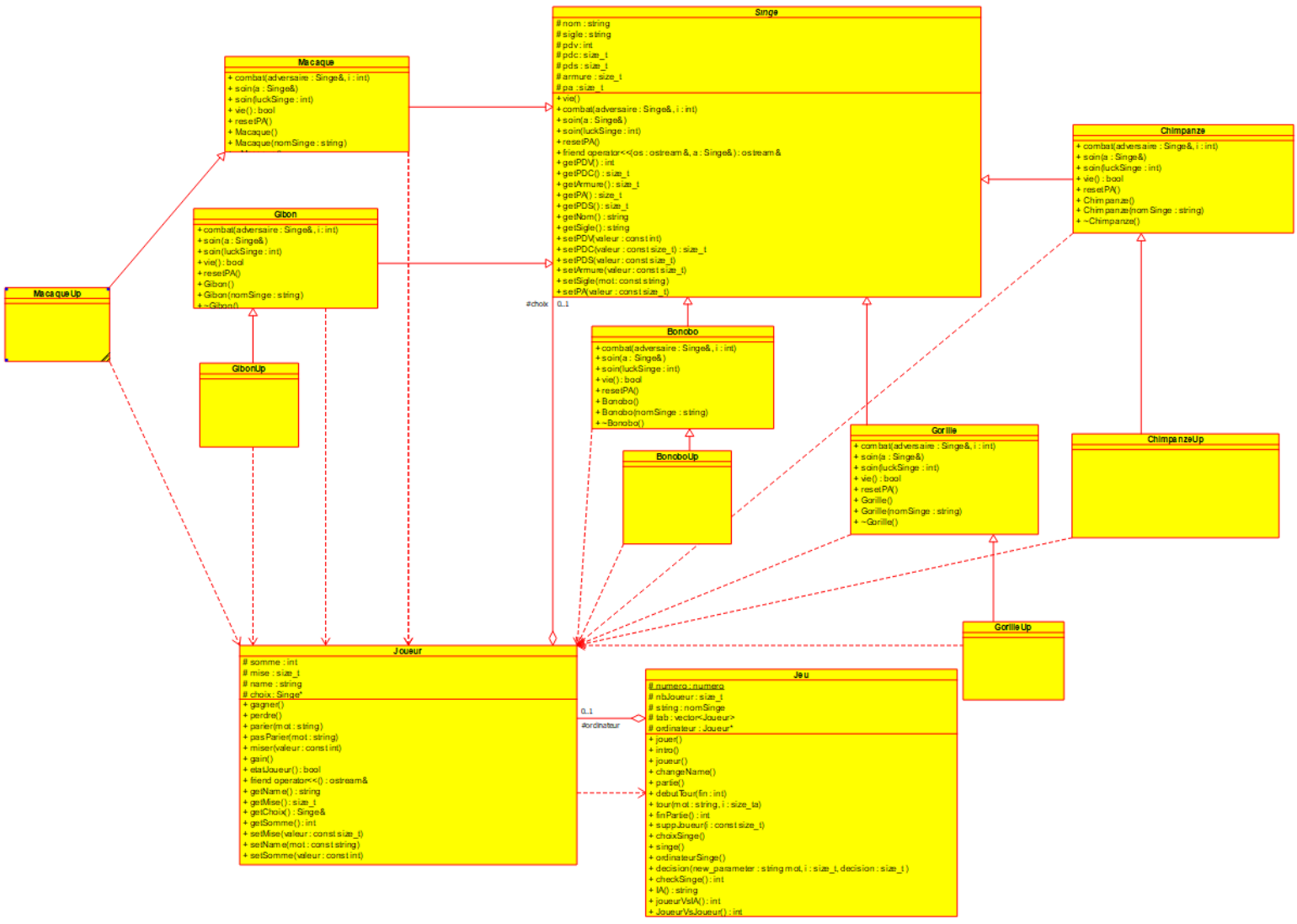
```
TEST_CASE("1:Tests de combats simples entre singes"){
  //Test de combat simple
  Gorille gorille;
  REQUIRE(gorille.getPDV() == 90);
  GorilleUp gorilleUp;
  REQUIRE(gorilleUp.getPDV() == 180);
  std::cout << "gorille combat gorilleUp "<< std::endl;
  gorille.combat(gorilleUp, 3);
  std::cout << "gorille combat gorilleUp "<< std::endl;
  gorilleUp.combat(gorille, 3);
  REQUIRE_FALSE(gorille.vie());
  REQUIRE(gorilleUp.vie());
  std::cout << "gorille a " << gorille.getPDV() << " pdv" << std::endl;
  std::cout << "gorilleUp a " << gorilleUp.getPDV() << " pdv" << std::endl;

  //Un singe ne peut soigner qu'un singe en vie
  ChimpanzeUp chimpanzeUp;
  chimpanzeUp.soin(gorille);
  chimpanzeUp.soin(gorilleUp);
  REQUIRE(gorille.getPDV() == 0);
  REQUIRE(gorilleUp.getPDV() == 140);
}

TEST_CASE("2:Tests sur des Joueurs"){
  //Alice ne peut pas miser plus que ce qu'elle a
  Joueur alice = Joueur("Alice");
  int somme_alice = alice.getSomme();
  alice.setMise(100);
  REQUIRE(alice.getSomme() == somme_alice);

  //Test de la formule de gain
  int gains_theoriques = somme_alice + (100*13);
  alice.gain();
  REQUIRE(alice.getSomme() == gains_theoriques);
}
```

Diagramme UML



Les flèches en trait plein représentent *l'héritage*

Les flèches en pointillées représentent les *#include*

Exécution du code

Le jeu se lance à partir du terminal car nous n'avons pas eu assez de temps pour finir l'interface graphique avec SFML. Celle-ci se trouve sur Github si vous souhaitez la voir

L'interface graphique devait se composer de 3 pages qui s'affichent successivement (page d'accueil, choix du nombre de joueur et les noms et la page de jeu)

Les bibliothèques que nous avons utilisées sont courantes et sont celles que nous avons utilisés en TP à l'exception de la bibliothèque de SFML que nous n'avions pas utilisé encore.

Nous avons un Makefile qui permet donc de compiler l'application avec le comme nom d'exécutable *prog*.

Pour les tests unitaires, il faut aller dans le dossier *tests* et compiler avec le deuxième *Makefile*

Remarque : Il faut éviter de taper n'importe quoi dans le terminal lors du choix du Singe, sinon ça arrête le programme

Implémentation

Les parties du code qui sont intéressantes :

- Combat aléatoire entre les singes ils ont 1 chance sur 3 (si 3 joueurs dans la partie) de pouvoir attaquer sans pour autant s'attaquer soit même

```
luckAttaquant = rand() % tab.size();
luckCible = rand() % tab.size();
luckSinge = rand() % 3 + 1;
tab[luckAttaquant].getChoix().combat(tab[luckCible].getChoix(), luckSinge); // on attaque le singe d'un joueur
tab[luckAttaquant].getChoix().soin(luckSinge); // permet de se soigner
tab[luckAttaquant].getChoix().resetPA(); // on remet les points d'action au max

urn checkSinge();
```

- L'IA choisit aléatoirement un singe pour parier et parie aléatoirement une somme

```
void Jeu :: ordinateurSinge()
{
    int valeur;
    int valeur2;
    srand((unsigned int)time(0));
    string word;
    word = IA();
    ordinateur -> parier(word); // L'IA choisit un singe aléatoirement

    valeur2 = ordinateur -> getSomme();
    valeur = rand() % valeur2;
    if(valeur == 0)
    {
        valeur += 1;
    }
    ordinateur -> setMise(valeur);
    ordinateur -> miser(ordinateur -> getMise()); // L'IA mise aléatoirement
}
```

- Gérer suivant si on a un seul joueur et donc jouer avec l'IA ou lancer une partie avec plusieurs joueurs distincts
- Pouvoir afficher au cours de la partie les différentes statistiques par joueur ou encore de l'IA quand on joue tous seul
- En cas d'erreur de frappe, donner à l'utilisateur la possibilité de pouvoir rentrer à nouveau son choix, mise