Documentação_TP_Jogo_RPG

Grupo 7
Breno
Gabriel
Leonardo
Renan

Quinta, 20 de Junho de 2019

Introdução

O seguinte relatório tem como objetivo descrever o projeto de um combate por turnos de um jogo de RPG utilizando-se da orientação a objetos em C++ para realizá-lo. A descrição do projeto consistia em criar o jogo utilizando dos pilares da orientação a objeto, ou seja, Encapsulamento, Herança e Poliformismo, além de outros temas ensinados durante a disciplina de Programação e Desenvolvimento de Software 2 (PDS2), como Exceções e o controle de versão, utilizando a ferramenta Github.

O jogo consiste em um grupo de heróis lutando contra um grupo de monstros, com cada um tendo sua vez de fazer uma ação como atacar ou usar uma habilidade. Cada vez que um personagem ou monstro é atingido seus pontos de vida diminuem. Vence a partida quem derrotar primeiro todos os montros do outro grupo.

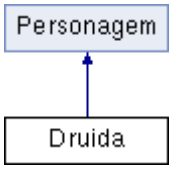
Índice dos Componentes

Lista de Classes

Classes

Referência da Classe Druida

`#include <Druida.h>`
Diagrama de hierarquia para Druida:



Membros Públicos

- **Druida** (std::string, int, int, int)
- **~Druida** ()
- int **ataque_basico** (**Personagem** *) override
- std::string **get_habilidade** (int) override
- std::string **usa_habilidade** (int, int, std::vector< **Personagem** * >, std::vector< **Personagem** * >) override
- std::string **habilidade_1** (int, std::vector< **Personagem** * >)
- std::string **habilidade_2** (int, std::vector< **Personagem** * >)
- std::string **habilidade_3** (int, std::vector< **Personagem** * >)

Outros membros herdados

Construtores e Destrutores

Druida::Druida (std::string *nome*, int *forca*, int *agilidade*, int *inteligencia*) : Personagem (nome, forca, agilidade, inteligencia)

```
 5 {
 6     _nome_classe = "Druida";
 7      habilidade 1 = "Cura - permite curar um aliado";
 8     _habilidade_2 = "Forma Animal - ataca um inimigo em forma de urso";
 9     _habilidade_3 = "Revitaliza - permite revitalizar EP/MP de um aliado";
10 }
```

Druida::~Druida ()

```
12 {
13
14 }
```

Funções membros

int Druida::ataque_basico (Personagem * *alvo*) [override], [virtual]

```
17 {
18     return alvo->recebe_ataque_fisico(_ataque);
19 }
```

std::string Druida::get_habilidade (int *habilidade_escolhida*) [override], [virtual]

```
22 {
23     switch (habilidade_escolhida)
24     {
25         case 1: return _habilidade_1; break;
26         case 2: return _habilidade_2; break;
```

```
   27         case 3: return _habilidade_3; break;
   28         default: return "Habilidade inválida"; break;
   29     }
   30 }
```

std::string Druida::habilidade_1 (int *segunda_escolha*, std::vector< Personagem * > *grupo_aliado*)

```
   70 {
   71     --segunda_escolha;
   72     // Testa se possui MP suficiente para usar habilidade
   73     if( mp >= CUSTO HABILIDADE 1 DRUIDA)
   74     {
   75         // Verifica se personagem está vivo
   76         if(grupo_aliado[segunda_escolha]->get_vivo())
   77         {
   78             int recupera_hp = grupo_aliado[segunda_escolha]->get_max_hp() *
FATOR_CURA_DRUIDA;
   79
   80             if(grupo_aliado[segunda_escolha]->get_hp() + recupera_hp >=
grupo_aliado[segunda_escolha]->get_max_hp())
   81             {
   82                 recupera_hp = (grupo_aliado[segunda_escolha]->get_max_hp() -
grupo_aliado[segunda_escolha]->get_hp());
   83             }
   84             grupo_aliado[segunda_escolha]->set_hp(recupera_hp +
grupo_aliado[segunda_escolha]->get_hp());
   85
   86             return _nome + " usou " + _habilidade_1 + " e conseguiu aumentar em
"
   87                 + std::to_string(recupera_hp) + " o HP de " +
grupo_aliado[segunda_escolha]->get_nome();
   88         }
   89         return grupo_aliado[segunda_escolha]->get_nome() + " está morto e não
pode ser curado";
   90     }
   91     return "Energia insuficiente para usar esta habilidade. " +  nome + "
desperdiçou sua vez.";
   92 }
```

std::string Druida::habilidade_2 (int *segunda_escolha*, std::vector< Personagem * > *grupo_inimigo*)

```
   95 {
   96     --segunda_escolha;
   97     //testa se inimigo esta vivo
   98     if(grupo_inimigo[segunda_escolha]->get_vivo() == true)
   99     {
  100
  101         //testa so o personagem tem mp suficientes
  102         if( mp >= (CUSTO HABILIDADE 2 DRUIDA))
  103         {
  104
  105
grupo inimigo[segunda escolha]->diminui hp(INCREMENTO ATAQUE DRUIDA +  ataque);
  106             _mp -= CUSTO_HABILIDADE_2_DRUIDA;
  107             std::string menssagem;
  108
  109             //confere se o inimigo foi morto pelo ataque
  110             if(grupo_inimigo[segunda_escolha]->get_hp() <= 0)
  111             {
  112                 grupo_inimigo[segunda_escolha]->set_vivo_morto(false);
  113                 menssagem = grupo_inimigo[segunda_escolha]->get_nome() + " foi
morto.";
  114             }
  115             return _nome + " se transformou em um urso e atacou " +
grupo_inimigo[segunda_escolha]->get_nome() + " causando " +
std::to string(INCREMENTO ATAQUE DRUIDA +  ataque) + " de dano. " + menssagem;
  116
  117         }
  118         else return "MP insuficiente para realizar este ataque. " + _nome + "
não conseguiu fazer nada.";
  119     }
  120     else return "Habilidade só pode ser usada em um inimigo que ainda esta vivo.
" + _nome + " não conseguiu fazer nada.";
  121 }
```

std::string Druida::habilidade_3 (int *segunda_escolha*, std::vector< Personagem * > *grupo_aliado*)

```
124 {
125     --segunda_escolha;
126     //testa de possui MP suficiente para usar esta
127     if(_mp >= CUSTO_HABILIDADE_3_DRUIDA)
128     {
129         if(grupo_aliado[segunda_escolha]->get_vivo())
130         {
131             int recupera_mp = grupo_aliado[segunda_escolha]->get_max_mp() *
FATOR_REVITALIZACAO_DRUIDA;
132
133             if(grupo_aliado[segunda_escolha]->get_mp() + recupera_mp >=
grupo_aliado[segunda_escolha]->get_max_mp())
134             {
135                 recupera_mp = (grupo_aliado[segunda_escolha]->get_max_mp() -
grupo_aliado[segunda_escolha]->get_mp());
136             }
137
138             grupo_aliado[segunda_escolha]->set_mp(recupera_mp +
grupo_aliado[segunda_escolha]->get_mp());
139
140             return _nome + " usou " + _habilidade_3 + " e conseguiu regenerar
"
141                 + std::to_string(recupera_mp) + " do MP/EP de " +
grupo_aliado[segunda_escolha]->get_nome();
142         }
143         return grupo_aliado[segunda_escolha]->get_nome() + " está morto e não
pode ser revitalizado";
144     }
145     return "MP insuficiente para usar esta habilidade. " +  nome + " desperdiçou
sua vez.";
146 }
```

std::string Druida::usa_habilidade (int *habilidade_escolhida*, int *segunda_escolha*, std::vector< Personagem * > *grupo_aliado*, std::vector< Personagem * > *grupo_inimigo*) `[override]`, `[virtual]`

```
33 {
34     std::string msg = "";
35     switch (habilidade_escolhida)
36     {
37         case 1:
38             if(segunda_escolha == 0)
39             {
40                 return "Escolher aliado";
41             }
42             msg = this->habilidade_1(segunda_escolha, grupo_aliado);
43             break;
44
45         case 2:
46             if(segunda_escolha == 0)
47             {
48                 return "Escolher inimigo";
49             }
50             msg = this->habilidade_2(segunda_escolha, grupo_inimigo);
51             break;
52
53         case 3:
54             if(segunda_escolha == 0)
55             {
56                 return "Escolher aliado";
57             }
58             msg = this->habilidade_3(segunda_escolha, grupo_aliado);
59             break;
60
61         default:
62             msg = "Habilidade inválida";
63             break;
64     }
65
66     return msg;
67 }
```

Referência da Classe Experiencia

`#include <Experiencia.h>`

Membros Públicos

- **Experiencia** ()
- **~Experiencia** ()
- int **get_xp** ()
- int **get_level_atual** ()
- int **get_xp_prox_level** ()
- int **get_xp_necessaria** ()
- int **calcula_xp_monstro** ()
- void **adiciona_xp** (int xp)
- void **sobe_de_level** ()

Construtores e Destrutores

Experiencia::Experiencia ()

```
 4 {
 5     this->_level = 1;
 6     this->_xp_total = 0;
 7     this->_xp_para_prox_level = XP_PROX_LEVEL;
 8     this->_xp_necessaria = XP_PROX_LEVEL;
 9 }
```

Experiencia::~Experiencia ()

```
12 {
13
14 }
```

Funções membros

void Experiencia::adiciona_xp (int *xp*)

```
41 {
42     //entra no if caso o personagem ganhe xp suficiente para subir level
43     //TeSTE
44     int xp_aux = xp;
45     while(this->_xp_necessaria - xp_aux <= 0)
46     {
47         sobe_de_level();
48         xp_aux -= this->_xp_necessaria;
49         this->_xp_para_prox_level *=XP_MULTIPLICADOR;
50         this-> xp necessaria = this-> xp para prox level;
51         //this->_xp_para_prox_level = (this->_xp_para_prox_level) +
(this->_xp_para_prox_level)*(XP_MULTIPLICADOR);//atualiza xp necessaria para proximo
level
52     }
53         this->_xp_necessaria -= xp_aux;
54
55     /*if(this->_xp_total + xp >= XP_PROX_LEVEL)
56     {
57         sobe_de_level();
58         this->_xp_para_prox_level = (this->_xp_para_prox_level) +
(this->_xp_para_prox_level)*(XP_MULTIPLICADOR);//atualiza xp necessaria para proximo
level
59     }*/
60     this->_xp_total += xp;
61 }
```

int Experiencia::calcula_xp_monstro ()

int Experiencia::get_level_atual ()

```
22 {
23     return this-> level;
24 }
```

int Experiencia::get_xp ()

```
17 {
18     return this->_xp_total;
19 }
```

int Experiencia::get_xp_necessaria ()

```
32 {
33     return this->_xp_necessaria;
34 }
```

int Experiencia::get_xp_prox_level ()

```
27 {
28     return this-> xp para prox level;
29 }
```

void Experiencia::sobe_de_level ()

```
37 {
38     this->_level += 1;
39 }
```

Referência da Classe Feiticeiro

`#include <Feiticeiro.h>`

Diagrama de hierarquia para Feiticeiro:

Membros Públicos

- **Feiticeiro** (std::string, int, int, int)
- **~Feiticeiro** ()
- int **ataque_basico** (**Personagem** *) override
- std::string **get_habilidade** (int) override
- std::string **usa_habilidade** (int, int, std::vector< **Personagem** * >, std::vector< **Personagem** * >) override
- std::string **habilidade_1** (std::vector< **Personagem** * >)
- std::string **habilidade_2** (int, std::vector< **Personagem** * >)
- std::string **habilidade_3** (int, std::vector< **Personagem** * >)
- std::string **invoca_raio_paralisante** (**Personagem** *)

Outros membros herdados

Construtores e Destrutores

Feiticeiro::Feiticeiro (std::string *nome*, int *forca*, int *agilidade*, int *inteligencia*)

```
    3
:Personagem (nome, forca, agilidade, inteligencia)
    4 {
    5     _nome_classe = "Feiticeiro";
    6     _habilidade_1 = "Bola de Fogo";
    7     _habilidade_2 = "Drenar Energia";
    8     _habilidade_3 = "Raio paralizante";
    9 }
```

Feiticeiro::~Feiticeiro ()

```
   12 {
   13
   14 }
```

Funções membros

int Feiticeiro::ataque_basico (Personagem * *alvo*) [override], [virtual]

```
   17 {
   18     return alvo->recebe_ataque_fisico(_ataque);
   19 }
```

std::string Feiticeiro::get_habilidade (int *habilidade_escolhida*) [override], [virtual]

```
   30 {
   31     switch (habilidade_escolhida)
   32     {
   33         case 1: return _habilidade_1; break;
   34         case 2: return _habilidade_2; break;
   35         case 3: return _habilidade_3; break;
```

```
   36         default: return "Habilidade inválida"; break;
   37     }
   38 }
```

std::string Feiticeiro::habilidade_1 (std::vector< Personagem * > *grupo_inimigo*)

```
   75 {
   76     // Verifica se possui MP suficiente para habilidade
   77     if (_mp >= CUSTO_HABILIDADE_FT_1)
   78     {
   79         std::string msg = this->_nome + " conjurou uma bola de fogo sobre os
adversários\n";
   80         int dano = 0;
   81         _mp -= CUSTO_HABILIDADE_FT_1;
   82 
   83         for(Personagem* p : grupo_inimigo)
   84         {
   85             if(p->get_vivo())
   86             {
   87                 dano = p->recebe ataque magia(DANO BOLA DE FOGO);
   88                 msg += p->get_nome() + " sofreu " + std::to_string(dano) + " de
dano.\n";
   89 
   90                 if (!p->get vivo())
   91                 {
   92                     msg += p->get_nome() + " foi derrotado.\n";
   93                 }
   94             }
   95         }
   96         return msg;
   97     }
   98     return "Energia insuficiente para usar esta habilidade. " + _nome + "
desperdiçou sua vez.";
   99 }
```

std::string Feiticeiro::habilidade_2 (int *segunda_escolha*, std::vector< Personagem * > *grupo_inimigo*)

```
  102 {
  103     // Verifica se inimigo atacado está vivo
  104     if (grupo_inimigo[segunda_escolha-1]->get_vivo())
  105     {
  106         // Verifica se possui MP suficiente para usar a habilidade
  107         if(_mp >= CUSTO_HABILIDADE_FT_2)
  108         {
  109             _mp -= CUSTO_HABILIDADE_FT_2;
  110 
  111             int hp_inimigo = grupo_inimigo[segunda_escolha-1]->get_hp();
  112             int mp_inimigo = grupo_inimigo[segunda_escolha-1]->get_mp();
  113             int hp drena = hp inimigo * FATOR DRENAR;
  114             int mp_drena = mp_inimigo * FATOR_DRENAR;
  115 
  116             std::string msg = this->_nome + " usou drenar energia em " +
grupo inimigo[segunda escolha-1]->get nome() + ".\n";
  117             msg += "Foi absorvido " + std::to_string(hp_drena) + " de HP e " +
std::to_string(mp_drena) + " de MP\n";
  118 
  119             // Adiciona valores ao feiticeiro
  120             if ( hp + hp drena <  max hp)
  121             {
  122                 _hp += hp_drena;
  123             }
  124             else
  125             {
  126                 _hp = _max_mp;
  127             }
  128 
  129             if(_mp + mp_drena < _max_mp)
  130             {
  131                 _mp += mp_drena;
  132             }
  133             else
  134             {
  135                 _mp = _max_mp;
  136             }
  137 
```

```
  138             // Remove valores do inimigo
  139             grupo_inimigo[segunda_escolha-1]->set_hp(hp_inimigo - hp_drena);
  140             grupo_inimigo[segunda_escolha-1]->set_mp(mp_inimigo - mp_drena);
  141 
  142             return msg;
  143         }
  144         return "Energia insuficiente para usar esta habilidade. " + _nome + "
desperdiçou sua vez.";
  145     }
  146     return grupo_inimigo[segunda_escolha-1]->get_nome() + " já está morto! " +
_nome + " desperdiçou sua vez.";
  147 }
```

std::string Feiticeiro::habilidade_3 (int *segunda_escolha*, std::vector< Personagem * > *grupo_inimigo*)

```
  150 {
  151     // Verifica se inimigo atacado está vivo
  152     if (grupo_inimigo[segunda_escolha-1]->get_vivo())
  153     {
  154         // Verifica se possui MP suficiente para usar a habilidade
  155         if(_mp >= CUSTO_HABILIDADE_FT_2)
  156         {
  157             _mp -= CUSTO_HABILIDADE_FT_2;
  158 
  159             int random = rand() % 100 + 1;
  160             //Acertou
  161             if(random <= 50)
  162             {
  163                 return
invoca_raio_paralisante(grupo_inimigo[segunda_escolha-1]);
  164 
/*grupo inimigo[segunda escolha-1]->recebe ataque magia(DANO RAIO PARALIZANTE);
  165                 grupo_inimigo[segunda_escolha-1]->set_perdeu_vez(true);
  166                 return "Raio acertou em cheio, causando " +
std::to_string(DANO_RAIO_PARALIZANTE) + " de dano e impedindo "
  167                 + grupo inimigo[segunda escolha-1]->get nome() + "de agir no
próximo turno.";
  168                 */
  169             }
  170             else
  171             {
  172                 return "Raio foi conjurado mas não conseguiu acertar o alvo. "
+ _nome + " desperdiçou sua vez.";
  173             }
  174         }
  175         return "Energia insuficiente para usar esta habilidade. " + _nome + "
desperdiçou sua vez.";
  176     }
  177     return grupo_inimigo[segunda_escolha-1]->get_nome() + " já está morto! " +
_nome + " desperdiçou sua vez.";
  178 }
```

std::string Feiticeiro::invoca_raio_paralisante (Personagem * *alvo*)

```
   22 {
   23     alvo->recebe_ataque_magia(DANO_RAIO_PARALIZANTE);
   24     alvo->set_perdeu_vez(true);
   25     return "Raio acertou em cheio, causando " +
std::to_string(DANO_RAIO_PARALIZANTE) + " de dano e impedindo "
   26     + alvo->get_nome() + " de agir no próximo turno.";
   27 }
```

std::string Feiticeiro::usa_habilidade (int *habilidade_escolhida*, int *segunda_escolha*, std::vector< Personagem * > *grupo_aliado*, std::vector< Personagem * > *grupo_inimigo*) `[override]`, `[virtual]`

```
   41 {
   42     std::string msg = "";
   43     switch (habilidade_escolhida)
   44     {
   45         case 1:
   46             msg = this->habilidade_1(grupo_inimigo);
   47             break;
   48 
   49         case 2:
```

```
50              if(segunda_escolha == 0)
51              {
52                  return "Escolher inimigo";
53              }
54
55              msg = this->habilidade_2(segunda_escolha, grupo_inimigo);
56              break;
57
58          case 3:
59              if(segunda_escolha == 0)
60              {
61                  return "Escolher inimigo";
62              }
63
64              msg = this->habilidade_3(segunda_escolha,grupo_inimigo);
65              break;
66
67          default:
68              msg = "Habilidade inválida";
69              break;
70      }
71      return msg;
72  }
```

Referência da Classe Guerreiro

`#include <Guerreiro.h>`

Diagrama de hierarquia para Guerreiro:

Membros Públicos

- **Guerreiro** (std::string, int, int, int)
- **~Guerreiro** ()
- int **ataque_basico** (**Personagem** *) override
- std::string **get_habilidade** (int) override
- std::string **usa_habilidade** (int, int, std::vector< **Personagem** * >, std::vector< **Personagem** * >) override
- std::string **habilidade_1** ()
- std::string **habilidade_2** (int, std::vector< **Personagem** * >)
- std::string **habilidade_3** (int, std::vector< **Personagem** * >)

Outros membros herdados

Construtores e Destrutores

Guerreiro::Guerreiro (std::string *nome*, int *forca*, int *agilidade*, int *inteligencia*) : Personagem (nome, forca, agilidade, inteligencia)

```
 4 {
 5     _max_mp = 100;
 6     _nome_classe = "Guerreiro";
 7     _habilidade_1 = "Primeiros Socorros - recupera 25% do HP";
 8     _habilidade_2 = "Execução";
 9     _habilidade_3 = "Fúria de Batalha";
10 }
```

Guerreiro::~Guerreiro ()

```
12 {
13
14 }
```

Funções membros

int Guerreiro::ataque_basico (Personagem * *alvo*) [override], [virtual]

```
17 {
18     return alvo->recebe_ataque_fisico(_ataque);
19 }
```

std::string Guerreiro::get_habilidade (int *habilidade_escolhida*) [override], [virtual]

```
22 {
23     switch (habilidade_escolhida)
24     {
25         case 1: return _habilidade_1; break;
26         case 2: return _habilidade_2; break;
27         case 3: return _habilidade_3; break;
28         default: return "Habilidade inválida";
```

```
   29         break;
   30     }
   31 }
```

std::string Guerreiro::habilidade_1 ()

```
   72 {
   73     //testa se tem energia suficiente usar esta habilidade
   74     if(_mp >= CUSTO_HABILIDADE_1)
   75     {
   76             int recupera_hp = _max_hp * FATOR_CURA;
   77             if(_hp + recupera_hp >= _max_hp)
   78             {
   79                 recupera_hp = (_max_hp - _hp);
   80             }
   81 
   82             _hp += recupera_hp;
   83             return _nome + " usou " + _habilidade_1 + " e conseguiu recuperar
" + std::to_string(recupera_hp) + " de HP.";
   84     }
   85     return "Energia insuficiente para usar esta habilidade. " + _nome + "
desperdiçou sua vez.";
   86 }
```

std::string Guerreiro::habilidade_2 (int *segunda_escolha*, std::vector< Personagem * > *grupo_inimigo*)

```
   89 {
   90     //testa se inimigo esta vivo
   91     if(grupo_inimigo[segunda_escolha-1]->get_vivo() == true)
   92     {
   93         //testa se é possivel executar o inimigo
   94         if(grupo_inimigo[segunda_escolha-1]->get_hp() <
(grupo_inimigo[segunda_escolha-1]->get_max_hp() * PONTO_DE_EXECUCAO))
   95         {
   96             _mp = 0;
   97             grupo_inimigo[segunda_escolha-1]->set_vivo_morto(false);
   98             return  grupo_inimigo[segunda_escolha-1]->get_nome() + " foi
executado. " + _nome + " gastou toda sua energia ao fazer isso.";
   99         }
  100     return "Ainda não é possivel executar este inimgo.";
  101     }
  102     else return "Habilidade só pode ser usada em um inimigo que ainda esta vivo.
" + _nome + " não conseguiu fazer nada.";
  103 }
```

std::string Guerreiro::habilidade_3 (int *segunda_escolha*, std::vector< Personagem * > *grupo_inimigo*)

```
  106 {
  107     std::string menssagem = "";
  108 
  109     //testa se inimigo esta vivo
  110     if(grupo_inimigo[segunda_escolha-1]->get_vivo() == true)
  111     {
  112 
  113         //testa so o personagem tem energia e HP suficientes
  114         if(_hp > _ataque && _mp >= (CUSTO_HABILIDADE_3))
  115         {
  116             grupo_inimigo[segunda_escolha-1]->diminui_hp(2 *  ataque);
  117             _hp -= _ataque;
  118             _mp -= CUSTO_HABILIDADE_3;
  119             std::string menssagem;
  120 
  121             //confere se o inimigo foi morto pelo ataque
  122             if(grupo_inimigo[segunda_escolha-1]->get_hp() <= 0)
  123             {
  124                 grupo_inimigo[segunda_escolha-1]->set_vivo_morto(false);
  125                 menssagem = grupo_inimigo[segunda_escolha-1]->get_nome() + "
foi morto.";
  126             }
  127             return grupo_inimigo[segunda_escolha-1]->get_nome() + " sofreu " +
std::to_string(2 * _ataque) + " de dano. " + menssagem;
  128 
  129         }
```

```
  130         else return "HP insuficiente para realizar este ataque. " + _nome + "
não conseguiu fazer nada.";
  131     }
  132     else return "Habilidade só pode ser usada em um inimigo que ainda esta vivo.
" + _nome + " não conseguiu fazer nada.";
  133 }
```

std::string Guerreiro::usa_habilidade (int *habilidade_escolhida*, int *segunda_escolha*, std::vector< Personagem * > *grupo_aliado*, std::vector< Personagem * > *grupo_inimigo*) [override], [virtual]

```
   34 {
   35     std::string msg = "";
   36     switch (habilidade_escolhida)
   37     {
   38         case 1:
   39         {
   40             msg = this->habilidade_1();
   41             break;
   42         }
   43         case 2:
   44         {
   45             if (segunda_escolha == 0)
   46             {
   47                 return "Escolher inimigo";
   48             }
   49             msg = this->habilidade_2(segunda_escolha, grupo_inimigo);
   50             break;
   51         }
   52         case 3:
   53         {
   54             if (segunda_escolha == 0)
   55             {
   56                 return "Escolher inimigo";
   57             }
   58             msg = this->habilidade_3(segunda_escolha, grupo_inimigo);
   59             break;
   60         }
   61         default:
   62         {
   63             msg = "Habilidade inválida";
   64             break;
   65         }
   66     }
   67
   68     return msg;
   69 }
```

Referência da Classe Jogo

`#include <Jogo.h>`

Membros Públicos

- **Jogo** ()
- **~Jogo** ()
- bool **carrega_arquivos** ()
- void **imprime_vetores** ()

Construtores e Destrutores

Jogo::Jogo ()

```
10 {
11 }
```

Jogo::~Jogo ()

```
13 {
14 }
```

Funções membros

bool Jogo::carrega_arquivos ()

```
17 {
18
19     //carrega o arquivo contendo os herois
20     std::ifstream arquivo("herois.csv");
21
22     if(!arquivo.is_open())
23     {
24         std::cout << "Erro: falha no carregamento de arquivo";
25     }
26
27     while(arquivo.good())
28     {
29         std::string nome,forca,agilidade,inteligencia;
30
31         getline(arquivo,nome,',');
32         getline(arquivo,forca,',');
33         getline(arquivo,agilidade,',');
34         getline(arquivo,inteligencia,'\n');
35
36         Personagem novo_personagem = Personagem(nome, std::stoi(forca),
std::stoi(agilidade), std::stoi(inteligencia));
37
38         _herois.push_back(novo_personagem);
39     }
40     arquivo.close();
41
42     //carrega o arquivo contendo os monstros
43     arquivo.open("monstros.csv");
44
45     if(!arquivo.is_open())
46     {
47         std::cout << "Erro: falha no carregamento de arquivo";
48     }
49
50     while(arquivo.good())
51     {
52         std::string nome,forca,agilidade,inteligencia;
53
54         getline(arquivo,nome,',');
```

```
   55         getline(arquivo,forca,',');
   56         getline(arquivo,agilidade,',');
   57         getline(arquivo,inteligencia,'\n');
   58 
   59         Monstro novo_personagem = Monstro(nome, std::stoi(forca),
std::stoi(agilidade), std::stoi(inteligencia));
   60 
   61         _monstros.push_back(novo_personagem);
   62     }
   63     arquivo.close();
   64 
   65         //Arquivos carregados com sucesso
   66         return true;
   67 };
```

void Jogo::imprime_vetores ()

```
   70 {
   71 
   72     //Imprime conteudo do vetor herois
   73     std::cout << "HEROIS:" << std::endl;
   74     for (unsigned int i = 0; i < _herois.size(); i++)
   75     {
   76          herois[i].imprime();
   77     }
   78 
   79     //Imprime conteudo do vetor de monstros
   80     std::cout << std::endl << "MONSTROS:" << std::endl;
   81     for (unsigned int i = 0; i < _monstros.size(); i++)
   82     {
   83         _monstros[i].imprime();
   84     }
   85 }
```

Referência da Classe Monstro

`#include <Monstro.h>`

Diagrama de hierarquia para Monstro:

Membros Públicos

- **Monstro** (std::string, int, int, int)
- **~Monstro** ()
- std::string **usa_habilidade** (int, int, std::vector< **Personagem** * >, std::vector< **Personagem** * >) override

Outros membros herdados

Construtores e Destrutores

Monstro::Monstro (std::string *nome*, int *forca*, int *agilidade*, int *inteligencia*) : Personagem (nome, forca, agilidade, inteligencia)

```
 4 {
 5     _nome_classe = "Monstro";
 6    // modifica_atributos_secundarios();
 7 }
```

Monstro::~Monstro ()

```
10 {
11
12 }
```

Funções membros

std::string Monstro::usa_habilidade (int *habilidade_escolhida*, int *segunda_escolha*, std::vector< Personagem * > *grupo_aliado*, std::vector< Personagem * > *grupo_inimigo*) `[override], [virtual]`

```
20 {
21     std::string msg = "";
22     switch (habilidade_escolhida)
23     {
24
25         case 1:
26             msg = "Usou " + _habilidade_1;
27             break;
28         case 2:
29             msg = "Usou " + _habilidade_2;
30             break;
31         case 3:
32             msg = "Usou " + _habilidade_3;
33             break;
34         default:
35             msg = "Habilidade inválida";
36             break;
37     }
38
39     return msg;
40 }
```

Referência da Classe Partida

`#include <Partida.h>`

Membros Públicos

- **Partida** (std::vector< **Personagem** * >, std::vector< **Personagem** * >, int)
- **~Partida** ()
- void **inicia** ()
- bool **terminou** ()
- std::vector< **Personagem** * > **determina_ordem** ()
- void **turno** (std::vector< **Personagem** * >)
- void **vez_do_personagem** (**Personagem** *)
- void **atacando** (**Personagem** *, std::vector< **Personagem** * > &)
- void **atacando** (**Personagem** *, std::vector< **Personagem** * > &, int inimigo_atacado)
- void **usando_habilidade** (**Personagem** *)
- void **refresh_tela** ()
- int **submenu_partida** (std::vector< std::string >)
- void **cor_jogador_atual** ()
- void **reseta_cor** ()
- void **vez_da_cpu** (**Personagem** *)

Construtores e Destrutores

Partida::Partida (std::vector< Personagem * > *grupo_a*, std::vector< Personagem * > *grupo_b*, int *modo_de_jogo*)

```
36 {
37     _grupo_blue = grupo_a;
38     _grupo_red = grupo_b;
39     _partida_terminou = false;
40     _modo_de_jogo = modo_de_jogo;
41     int _experiencia_monstros;
42 }
```

Partida::~Partida ()

```
44 {
45 };
```

Funções membros

void Partida::atacando (Personagem * *p*, std::vector< Personagem * > & *grupo_inimigo*)

```
183 {
184     // Mostra opções de quem atacar
185     std::vector <std::string> opcoes = {};
186 
187     for (unsigned int i = 0; i < grupo_inimigo.size(); i++)
188     {
189         opcoes.push_back(grupo_inimigo[i]->get_nome());
190     }
191 
192     int escolha = submenu_partida(opcoes);
193 
194     // Computa o ataque e imprime o resultado
195     while(!grupo_inimigo[escolha-1]->get_vivo())
196     {
197         std::cout << grupo inimigo[escolha-1]->get nome() << " já foi morto,
ataque outro!" << std::endl;
198         escolha = submenu_partida(opcoes);
```

```
  199     }
  200     int dano = p->ataque_basico(grupo_inimigo[escolha-1]);
  201     std::cout << p->get_nome() << " causou " << std::to_string(dano) << " de dano
em " << grupo_inimigo[escolha-1]->get_nome() << ". ";
  202
  203     // Informa caso o personagem tenha morrido
  204     if (!(grupo_inimigo[escolha-1]->get_vivo()))
  205     {
  206         std::cout << grupo_inimigo[escolha-1]->get_nome() << " foi morto em
combate.";
  207     }
  208
  209     std::cout << std::endl;
  210     return;
  211 }
```

void Partida::atacando (Personagem * *atacante*, std::vector< Personagem * > & *grupo_inimigo*, int *inimigo_atacado*)

```
  215 {
  216     int dano = atacante->ataque_basico(grupo_inimigo[inimigo_atacado]);
  217
  218     std::cout << atacante->get_nome() << " atacou e causou " <<
std::to_string(dano) << " de dano em " << grupo_inimigo[inimigo_atacado]->get_nome()
<< ". ";
  219
  220     //informa caso o personagem tenha morrido
  221     if(!(grupo_inimigo[inimigo_atacado]->get_vivo()))
  222     {
  223         std::cout << grupo_inimigo[inimigo_atacado]->get_nome() << " foi morto
em combate.";
  224     }
  225
  226     std::cout << std::endl;
  227     std::cout << "Aperte Enter para continuar..." << std::endl;
  228     std::getchar();
  229     refresh tela();
  230     return;
  231 }
```

void Partida::cor_jogador_atual ()

```
  460 {
  461     if(_grupo_que_estajogando == 'b')
  462     {
  463         std::cout << BOLDBLUE;
  464     }
  465     else std::cout << BOLDRED;
  466 }
```

std::vector< Personagem * > Partida::determina_ordem ()

```
   71 {
   72     _experiencia_monstros = 0;
   73     // Unifica os dois grupos da partida em um vetor de apontadores para
personagens
   74     std::vector <Personagem*> ordem;
   75
   76     for(unsigned int i = 0; i <  grupo blue.size(); i++)
   77     {
   78         _grupo_blue[i]->set_grupo('b');
   79         ordem.push_back(_grupo_blue[i]);
   80
   81     }
   82
   83     for(unsigned int i = 0; i < _grupo_red.size(); i++)
   84     {
   85         _grupo_red[i]->set_grupo('r');
   86         ordem.push_back(_grupo_red[i]);
   87     }
   88
   89     // Calcula a experiência a ser gerada pelos monstros na batalha
   90     for(unsigned int i = 0; i< ordem.size(); i++)
   91     {
   92         int xp = ordem[i]->calcula_xp_monstro(ordem[i]);
   93         _experiencia_monstros += xp;
```

```
   94         std::cout <<" EXP ATUAL " << _experiencia_monstros << std::endl;
   95     }
   96
   97     // Ordena esse vetor por agilidade dos personagens
   98     std::sort(ordem.begin(), ordem.end(), compara_agilidade);
   99
  100     // Retorna o vetor ordenado
  101     std::cout << "EXPERIENCIA DOS MONSTROS = " << _experiencia_monstros <<
std::endl;
  102     return ordem;
  103 }
```

void Partida::inicia ()

```
   48 {
   49     refresh_tela();
   50     std::cout << "Nova partida iniciada." << std::endl;
   51     std::cout << "Aperte Enter para continuar..." << std::endl;
   52     std::getchar();
   53     refresh_tela();
   54
   55     // Cnicializa random seed para geração de números aleatórios
   56     srand(time(NULL));
   57
   58     // Cria um vetor de apontadores para personagens e ordena o mesmo por agilidade
   59     std::vector <Personagem*> ordem = determina_ordem();
   60
   61     // Chama um novo turno até a partida terminar
   62     while(!_partida_terminou)
   63     {
   64         //inicia um turno
   65         turno(ordem);
   66     }
   67     std::cout << "Partida encerrada." << std::endl;
   68 }
```

void Partida::refresh_tela ()

```
  379 {
  380     // Limpa a tela
  381     system("clear");
  382
  383     std::cout <<
"--------------------------------------------------------------------------------" <<
std::endl;
  384
  385     // Imprime na tela os jogadores da partida divididos em dois times
  386     for (unsigned int i = 0; i < _grupo_blue.size(); i++)
  387     {
  388         //std::cout << "|    ";
  389         std::cout << BOLDBLUE << _grupo_blue[i]->get_nome() << ", " <<
_grupo_blue[i]->get_nome_classe() << RESETCOLOR;
  390         int left = _grupo_blue[i]->get_nome().size() +
_grupo_blue[i]->get_nome_classe().size() + 5;
  391         if(_grupo_blue[i]->get_vivo())
  392         {
  393             left = 45 - left;
  394
  395             for (int i = 0; i < left; i++)
  396             {
  397                 std::cout << " ";
  398             }
  399             std::cout << "   HP: " << BOLDGREEN << std::setw(3) <<
_grupo_blue[i]->get_hp() << "/" << std::setw(3) << _grupo_blue[i]->get_max_hp();
  400             reseta_cor();
  401
  402             if(_grupo_blue[i]->get_nome_classe() == "Guerreiro")
  403             {
  404                 std::cout << " - EP: " << BOLDYELLOW << std::setw(3) <<
_grupo_blue[i]->get_mp() << "/" << std::setw(3) << _grupo_blue[i]->get_max_mp();
  405             }
  406
  407             else
  408             {
  409                 std::cout << " - MP: " << BOLDMAGENTA << std::setw(3) <<
_grupo_blue[i]->get_mp() << "/" << std::setw(3) << _grupo_blue[i]->get_max_mp();;
  410             }
```

```
411             reseta_cor();
412         }
413         else
414         {
415             left = 57 - left;
416             std::cout << " morreu.";
417             for(int i = 0; i < left; i++)
418             {
419                 std::cout << " ";
420             }
421         }
422         std::cout << std::endl;
423     }
424     std::cout << "                                VS
" << std::endl;
425
426     for (unsigned int i = 0; i < _grupo_red.size(); i++)
427     {
428        // std::cout << "|   ";
429         std::cout << BOLDRED << _grupo_red[i]->get_nome() << " " << RESETCOLOR;
430         int left = _grupo_red[i]->get_nome().size() + 3;
431
432         if(_grupo_red[i]->get_vivo())
433         {
434             left = 44 - left;
435             for(int i = 0; i < left; i++)
436             {
437                 std::cout << " ";
438             }
439
440             std::cout << "   HP: " << BOLDGREEN << std::setw(3) <<
 grupo red[i]->get hp()<< "/" << std::setw(3) <<  grupo red[i]->get max hp();
441             reseta_cor();
442             std::cout << " - MP: " << BOLDMAGENTA << std::setw(3) <<
_grupo_red[i]->get_mp() << "/" << std::setw(3) << _grupo_red[i]->get_max_mp();
443             reseta cor();
444         } else
445         {
446             left = 57 - left;
447             std::cout << "morreu.";
448
449             for(int i = 0; i < left; i++)
450             {
451                 std::cout << " ";
452             }
453         }
454         std::cout <<std::endl;
455     }
456     std::cout <<
"--------------------------------------------------------------------------" <<
std::endl << std::endl;
457 }
```

void Partida::reseta_cor ()

```
469 {
470     std::cout << RESETCOLOR;
471 }
```

int Partida::submenu_partida (std::vector< std::string > *opcoes*)

```
283 {
284     std::string buffer;
285     unsigned int escolha = -1;
286     char c;
287
288     // Imprime na tela as opções
289     for(unsigned int i = 0; i < opcoes.size(); i++)
290     {
291         std::cout << i+1 << ". " << opcoes[i] << std::endl;
292     }
293
294     // Recebe a entrada do jogador
295     std::getline (std::cin,buffer);
296     if(!buffer.empty())
297     {
298         c = buffer.at(0);
```

```
299         escolha = atoi(&c);
300     }
301 
302      // Se o jogador fez uma escolha valida retorna o resultado
303     if (escolha <= opcoes.size() && escolha > 0)
304     {
305         refresh_tela();
306         return escolha;
307     }
308 
309     // Em caso de escolha inválida, repete as opções
310     while(!(escolha <= opcoes.size() && escolha > 0))
311     {
312         refresh_tela();
313         std::cout << "Opção inválida, tente novamente:" << std::endl;
314         std::cin.clear();
315         for(unsigned int i = 0; i < opcoes.size(); i++)
316         {
317             std::cout << i+1 << ". " << opcoes[i] << std::endl;
318         }
319 
320         // Recebe a entrada do jogador.
321         std::getline (std::cin,buffer);
322         if (!buffer.empty())
323         {
324             c = buffer.at(0);
325             escolha = atoi(&c);
326         }
327     }
328     refresh_tela();
329     return escolha;
330 }
```

bool Partida::terminou ()

```
333 {
334     bool terminou_blue = true;
335     bool terminou_red = true;
336 
337     // Confere se tem alguem vivo no primeiro grupo
338     for(unsigned int i = 0; i <  _grupo_blue.size(); i++)
339     {
340         if(_grupo_blue[i]->get_vivo())
341         {
342             terminou_blue = false;
343         }
344     }
345 
346     // Confere se tem alguem vivo no segundo grupo
347     for(unsigned int i = 0; i < _grupo_red.size(); i++)
348     {
349         if(_grupo_red[i]->get_vivo())
350         {
351             terminou_red = false;
352         }
353     }
354 
355     // Informa se um time perdeu
356     if(terminou_blue)
357     {
358         std::cout << "O time azul foi derrotado!" << std::endl;
359     }
360     if(terminou_red)
361     {
362         std::cout << "O time vermelho foi derrotado!" << std::endl;
363 
364         for(unsigned int i = 0; i < _grupo_blue.size(); i++)
365         {
366             if(_grupo_blue[i]->get_vivo())
367             {
368 
_grupo_blue[i]->set_experiencia_adquirida(_experiencia_monstros);
369                 std::cout << _grupo_blue[i]->get_nome() << " recebeu " <<
 experiencia monstros << " de XP " << " e esta no level " <<  grupo blue[i]->get_level()
<< " Forca: "<< _grupo_blue[i]->get_forca() << " Agilidade: " <<
_grupo_blue[i]->get_agilidade() << " Inteligência: "<<
_grupo_blue[i]->get_inteligencia()<< std::endl;
```

```
370             }
371         }
372     }
373
374     // Se terminou para um dos dois grupos terminou a partida
375     return (terminou_red || terminou_blue);
376 }
```

void Partida::turno (std::vector< Personagem * > *ordem*)

```
106 {
107     for(unsigned int i = 0; i < ordem.size(); i++)
108     {
109
110         // Confere se a partida já terminou antes da vez do personagem
111         _partida_terminou = terminou();
112         if(_partida_terminou)
113         {
114             return;
115         }
116
117         // Chama a função que permite o personagem agir no turno
118         if(ordem[i]->get_vivo())
119         {
120
121             //testa se esse personagem perdeu a vez
122             if(ordem[i]->get_perdeu_vez())
123             {
124                 std::cout << ordem[i]->get_nome() << " havia perdido a vez e não
pode fazer nada."<< std::endl;
125                 ordem[i]->set_perdeu_vez(false);
126                 std::cout << "Aperte Enter para continuar..." << std::endl;
127                 std::getchar();
128                 refresh_tela();
129             }
130             else
131             {
132                 // Caso controlado pela a CPU
133                 if(_modo_de_jogo == 2 && ordem[i]->get_grupo() == 'r')
134                 {
135                     vez_da_cpu(ordem[i]);
136                 }
137                 // Caso seja controlado por um jogador
138                 else
139                 {
140                     std::cout << "Vez de " << ordem[i]->get_nome() << ". O que
fazer?" << std::endl;
141                     vez_do_personagem(ordem[i]);
142                 }
143
144             }
145         }
146     }
147     return;
148 }
```

void Partida::usando_habilidade (Personagem * *p*)

```
234 {
235     std::vector <std::string> opcoes = {p->get_habilidade(1),
p->get_habilidade(2), p->get_habilidade(3)};
236     int escolha = submenu_partida(opcoes);
237
238     //tenta usar habilidade
239     std::string msg_habilidade = p->usa_habilidade(escolha, 0, _grupo_blue,
_grupo_red);
240
241     //caso a habilidade exiga escolher inimigo
242     if(msg_habilidade.compare("Escolher inimigo") == 0)
243     {
244         std::cout << "Em quem usar " <<  p->get_habilidade(escolha) << "?" <<
std::endl;
245
246         std::vector <std::string> opcoes2 = {};
247         for(unsigned int i = 0; i < _grupo_red.size(); i++)
248         {
249             opcoes2.push_back(_grupo_red[i]->get_nome());
```

```
  250         }
  251
  252         int escolha2 = submenu_partida(opcoes2);
  253
  254         msg_habilidade = p->usa_habilidade(escolha, escolha2, _grupo_blue,
_grupo_red);
  255     }
  256     //caso a habilidade exiga escolha um aliado
  257     else if(msg_habilidade.compare("Escolher aliado") == 0)
  258     {
  259         std::cout << "Em quem usar " <<  p->get_habilidade(escolha) << "?" <<
std::endl;
  260
  261         std::vector <std::string> opcoes2 = {};
  262
  263         for (unsigned int i = 0; i < _grupo_blue.size(); i++)
  264         {
  265             opcoes2.push_back(_grupo_blue[i]->get_nome());
  266         }
  267
  268         int escolha2 = submenu_partida(opcoes2);
  269
  270         msg_habilidade = p->usa_habilidade(escolha, escolha2, _grupo_blue,
_grupo_red);
  271     }
  272
  273     std::cout << msg_habilidade << std::endl;
  274     std::cout << std::endl;
  275     std::cout << "Aperte Enter para continuar..." << std::endl;
  276     std::getchar();
  277     refresh_tela();
  278
  279     return;
  280 }
```

void Partida::vez_da_cpu (Personagem * *p*)

```
  474 {
  475     bool inimigo_vivo = false;
  476     int inimigo_atacado;
  477
  478     //gra numero aleatório entre 1 e 100
  479     int random;
  480     random = rand() % 100 + 1;
  481
  482     // Determina ação da CPU
  483     // 10% de chance de passar a vez
  484     if (random < 10)
  485     {
  486         std::cout << "CPU controlando " << p->get_nome() << " ficou confusa e
passou a vez..." << std::endl;
  487         std::cout << "Aperte Enter para continuar..." << std::endl;
  488         std::getchar();
  489         refresh_tela();
  490     }
  491     // 20% de chance de usar habilidade especial
  492     else if(random > 80)
  493     {
  494         std::cout << "CPU controlando " << p->get_nome() << " usou habilidade."
<< std::endl;
  495         std::cout << "Aperte Enter para continuar..." << std::endl;
  496         std::getchar();
  497         refresh_tela();
  498     // 70% de chance de ataque normal
  499     }
  500     else
  501     {
  502         std::cout << "CPU controlando " << p->get_nome() << " decidiu atacar."
<< std::endl;
  503
  504         // Seleciona um inimigo vivo para atacar
  505         while(!inimigo_vivo)
  506         {
  507             // Gera número aleatório entre 1 e 100
  508             random = rand() % 100 + 1;
  509             // Determina qual inimigo atacar com chances iguais entre cada uma
das opções do grupo
```

```
  510             inimigo_atacado = std::floor(random*_grupo_blue.size() / 100);
  511             inimigo_vivo = _grupo_blue[inimigo_atacado]->get_vivo();
  512         }
  513 
  514         atacando(p, _grupo_blue, inimigo_atacado);
  515     }
  516 }
```

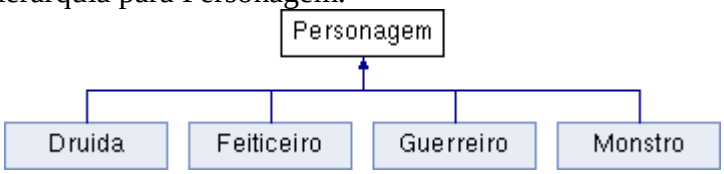
void Partida::vez_do_personagem (Personagem * *p*)

```
  151 {
  152     std::vector <std::string> opcoes = {"Atacar", "Usar habilidade", "Passar
vez"};
  153     int escolha = submenu_partida(opcoes);
  154 
  155     if(escolha == 1)
  156     {
  157         std::cout << p->get_nome() << " vai atacar. Quem atacar?" << std::endl;
  158 
  159         if(p->get_grupo()=='b')
  160         {
  161             atacando(p, _grupo_red);
  162         }
  163         else
  164         {
  165             atacando(p, _grupo_blue);
  166         }
  167     }
  168 
  169     if(escolha == 2)
  170     {
  171         std::cout << p->get_nome() << " vai usar uma habilidade especial." <<
std::endl;
  172         usando_habilidade(p);
  173     }
  174 
  175     if(escolha == 3)
  176     {
  177         std::cout << p->get_nome() << " passou a vez." << std::endl;
  178     }
  179     return;
  180 }
```

Referência da Classe Personagem

`#include <Personagem.h>`

Diagrama de hierarquia para Personagem:



Membros Públicos

- **Personagem** (std::string, int, int, int)
- **~Personagem** ()
- std::string **get_nome** ()
- virtual std::string **get_nome_classe** ()
- int **calcula_xp_monstro** (**Personagem** *)
- int **get_forca** ()
- int **get_agilidade** ()
- int **get_inteligencia** ()
- int **get_ataque** ()
- int **get_defesa** ()
- char **get_grupo** ()
- bool **get_vivo** ()
- bool **get_perdeu_vez** ()
- int **get_hp** ()
- int **get_max_hp** ()
- int **get_mp** ()
- int **get_max_mp** ()
- int **get_level** ()
- void **set_hp** (int)
- void **set_mp** (int)
- void **set_grupo** (char)
- void **set_perdeu_vez** (bool)
- void **diminui_hp** (int)
- void **set_vivo_morto** (bool)
- void **set_agilidade** (int)
- void **set_experiencia_adquirida** (int)
- void **set_atributos_lvl_up** ()
- void **set_atributos_secundarios_lvl_up** (int, int, int)
- void **imprime** ()
- virtual int **ataque_basico** (**Personagem** *)
- virtual std::string **get_habilidade** (int)
- virtual std::string **usa_habilidade** (int, int, std::vector< **Personagem** * >, std::vector < **Personagem** * >)
- int **recebe_ataque_fisico** (int)
- int **recebe_ataque_magia** (int)
- std::string **morreu** ()

Atributos Protegidos

- std::string **_nome**
- std::string **_nome_classe**
- **Experiencia _xp**
- int **_forca**
- int **_agilidade**

- int **_inteligencia**
- int **_ataque**
- int **_defesa**
- int **_hp**
- int **_mp**
- int **_max_hp**
- int **_max_mp**
- bool **_vivo**
- bool **_perdeu_vez**
- char **_grupo**
- std::string **_habilidade_1**
- std::string **_habilidade_2**
- std::string **_habilidade_3**

Construtores e Destrutores

Personagem::Personagem (std::string *nome*, int *forca*, int *agilidade*, int *inteligencia*)

```
24 {
25      //atributos principais
26     _nome = nome;
27     _forca = forca;
28     _agilidade = agilidade;
29     _inteligencia = inteligencia;
30     Experiencia _xp;
31 
32     //calcula os atributos secundarios a partir dos principais
33     _ataque = 10 * forca;
34     _defesa = 2 * forca; //maximo de 60
35     if(_defesa > 60)
36     {
37          defesa = 60;
38     }
39     _mp = 10 * inteligencia;
40     _hp = 10 * forca + 5 * _agilidade + 5 * _inteligencia;
41      max hp =  hp;
42     _max_mp = _mp;
43 
44     //estado do personagem
45     _vivo = true;
46     _perdeu_vez = false;
47 
48     //habilidades
49     _habilidade_1 = "Habilidade 1";
50     _habilidade_2 = "Habilidade 2";
51     _habilidade_3 = "Habilidade 3";
52 }
```

Personagem::~Personagem ()

```
54 {
55 
56 }
```

Funções membros

int Personagem::ataque_basico (Personagem * *alvo*) `[virtual]`

```
210 {
211     return alvo->recebe_ataque_fisico(_ataque);
212 }
```

int Personagem::calcula_xp_monstro (Personagem * *monstro*)

```
187 {
188     if (monstro->get_nome_classe() == "Monstro")
```

```
189     {
190         int experiencia = monstro->get_agilidade() + monstro->get_forca() +
monstro->get inteligencia();
191         return experiencia;
192     }
193     return 0;
194 }
```

void Personagem::diminui_hp (int *redutor*)

```
 13 {
 14     _hp -= redutor;
 15 }
```

int Personagem::get_agilidade ()

```
 87 {
 88     return _agilidade;
 89 }
```

int Personagem::get_ataque ()

```
117 {
118     return _ataque;
119 }
```

int Personagem::get_defesa ()

```
122 {
123     return _defesa;
124 }
```

int Personagem::get_forca ()

```
 82 {
 83     return _forca;
 84 }
```

char Personagem::get_grupo ()

```
 59 {
 60     return _grupo;
 61 }
```

std::string Personagem::get_habilidade (int *habilidade_escolhida*) `[virtual]`

```
215 {
216     switch (habilidade_escolhida)
217     {
218         case 1: return _habilidade_1; break;
219         case 2: return  habilidade_2; break;
220         case 3: return _habilidade_3; break;
221         default: return "Habilidade inválida";
222         break;
223     }
224 }
```

int Personagem::get_hp ()

```
 97 {
 98     return _hp;
 99 }
```

int Personagem::get_inteligencia ()

```
 92 {
 93     return _inteligencia;
 94 }
```

int Personagem::get_level ()

```
 71 {
 72     return _xp.get_level_atual();
 73 }
```

int Personagem::get_max_hp ()

```
102 {
103     return _max_hp;
104 }
```

int Personagem::get_max_mp ()

```
107 {
108     return _max_mp;
109 }
```

int Personagem::get_mp ()

```
112 {
113     return  mp;
114 }
```

std::string Personagem::get_nome ()

```
 76 {
 77     return _nome;
 78 }
```

std::string Personagem::get_nome_classe () `[virtual]`

```
294 {
295     return  nome classe;
296 }
```

bool Personagem::get_perdeu_vez ()

```
132 {
133     return _perdeu_vez;
134 }
```

bool Personagem::get_vivo ()

```
127 {
128     return _vivo;
129 }
```

void Personagem::imprime ()

```
196 {
197     std::cout << _nome << " >>";
198     std::cout << " forca:" << _forca;
199     std::cout << " agilidade:" << _agilidade;
200     std::cout << " inteligencia:" << _inteligencia;
201     std::cout << std::endl;
202     std::cout << " ataque:" << _ataque;
203     std::cout << " defesa:" << _defesa;
204     std::cout << " mp:" << _mp;
205     std::cout << " hp:" << _hp;
206     std::cout << std::endl;
207 }
```

std::string Personagem::morreu ()

```
281 {
282     if(_vivo)
283     {
284         return " hp= " + std::to_string(_hp);
285     }
286     else
287     {
288         _vivo = false;
289         return " morreu";
290     }
291 }
```

int Personagem::recebe_ataque_fisico (int *ataque*)

```
247 {
248     //numero aleatório entre 0 e 100
249     int random;
250     random = rand() % 100 + 1;
251     //se menor que agilidade o ataque erra
```

```
252     if(random <= _agilidade)
253     {
254         std::cout <<  nome + " conseguiu se esquivar. ";
255         return 0;
256     }
257     else
258     {
259         //defesa reduz o ataque
260         int resultado = ataque * (100-_defesa)/100;
261         _hp -= resultado;
262         if(_hp <= 0)
263         {
264             _vivo = false;
265         }
266         return resultado;
267     }
268 }
```

int Personagem::recebe_ataque_magia (int *ataque*)

```
271 {
272     _hp -= ataque;
273     if (_hp <= 0)
274     {
275         _vivo = false;
276     }
277     return ataque;
278 }
```

void Personagem::set_agilidade (int *agilidade*)

```
 18 {
 19     _agilidade = agilidade;
 20 }
```

void Personagem::set_atributos_lvl_up ()

```
152 {
153     //_xp.sobe_de_level();
154     _forca = round(_forca * MULTIP_LEVEL_UP);
155     _agilidade =round (_agilidade * MULTIP_LEVEL_UP);
156     _inteligencia = round (_inteligencia * MULTIP_LEVEL_UP);
157     set_atributos_secundarios_lvl_up(_forca, _agilidade, _inteligencia);
158 }
```

void Personagem::set_atributos_secundarios_lvl_up (int *forca*, int *agilidade*, int *inteligencia*)

```
161 {
162     _ataque = 10 * forca;
163      defesa = 2 * forca; //maximo de 60
164     if (_defesa > 60)
165     {
166         _defesa = 60;
167     }
168      mp = 10 * inteligencia;
169     _hp = 10 * forca + 5 * _agilidade + 5 * _inteligencia;
170     _max_hp = _hp;
171     _max_mp = _mp;
172 }
```

void Personagem::set_experiencia_adquirida (int *exp*)

```
175 {
176     int level = _xp.get_level_atual();
177     _xp.adiciona_xp(exp);
178
179     while (level < _xp.get_level_atual())
180     {
181         set_atributos_lvl_up();
182         level++;
183     }
184 }
```

void Personagem::set_grupo (char *grupo*)

```
 65 {
```

```
 66     _grupo = grupo;
 67     return;
 68 }
```

void Personagem::set_hp (int *hp*)

```
147 {
148     _hp = hp;
149 }
```

void Personagem::set_mp (int *mp*)

```
142 {
143     _mp = mp;
144 }
```

void Personagem::set_perdeu_vez (bool *status*)

```
137 {
138     _perdeu_vez = status;
139 }
```

void Personagem::set_vivo_morto (bool *status*)

```
  8 {
  9      vivo = status;
 10 }
```

std::string Personagem::usa_habilidade (int *habilidade_escolhida*, int *segunda_escolha*, std::vector< Personagem * > *grupo_aliado*, std::vector< Personagem * > *grupo_inimigo*) `[virtual]`

```
227 {
228     switch (habilidade_escolhida)
229     {
230         case 1:
231             std::cout << "Usou " << _habilidade_1 << std::endl;
232             break;
233         case 2:
234             std::cout << "Usou " << _habilidade_2 << std::endl;
235             break;
236         case 3:
237             std::cout << "Usou " << _habilidade_3 << std::endl;
238             break;
239         default:
240             std::cout << "Habilidade inválida" << std::endl;
241             break;
242     }
243     return "";
244 }
```

Arquivos

Referência do Arquivo 20191_team_7/Druida.cpp

```
#include "Druida.h"
```

Referência do Arquivo 20191_team_7/Experiencia.cpp

```
#include "Experiencia.h"
```

Referência do Arquivo 20191_team_7/Feiticeiro.cpp

```
#include "Feiticeiro.h"
```

Referência do Arquivo 20191_team_7/Guerreiro.cpp

```
#include "Guerreiro.h"
```

Referência do Arquivo 20191_team_7/Jogo.cpp

```
#include "Jogo.h"
#include "Personagem.h"
#include "Monstro.h"
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
```

Referência do Arquivo 20191_team_7/main.cpp

```
#include <iostream>
#include "Personagem.h"
#include "Guerreiro.h"
#include "Feiticeiro.h"
#include "Druida.h"
#include "Jogo.h"
#include "Partida.h"
```

Funções

- int **main** ()

Funções

int main ()

```
13 {
14     Guerreiro a0 = Guerreiro("Jon Snow",10,10,10);
15     Druida a1 = Druida("Jojen",10,10,10);
16     Feiticeiro a2 = Feiticeiro("Bloodraven",10,10,10);
17     Guerreiro a3 = Guerreiro("Bronn",10,10,10);
18
19     Monstro b0 = Monstro("Ratox", 2, 2, 2);         // um inimigo fraco para morrer
rápido
20     Monstro b1 = Monstro("Ramsey", 10, 10, 10);     // um inimigo igual ao hérois
21     Monstro b2 = Monstro("Cthulhu", 25, 25, 25);    // um inimigo forte
22
23
24     //Inicia partida com os personagens escolhidos
25     std::vector <Personagem*> grupo_a = { &a0, &a1, &a2, &a3};
26     std::vector <Personagem*> grupo_b = { &b0, &b1, &b2 };
```

```
27
28      // grupo_a[3]->ataque_basico(grupo_b[2]);
29
30      Partida partida = Partida(grupo_a, grupo_b, 2);
31
32      partida.inicia();
33      return 0;
34  }
```

Referência do Arquivo 20191_team_7/Monstro.cpp

```
#include <Monstro.h>
```

Referência do Arquivo 20191_team_7/Partida.cpp

```
#include "Partida.h"
#include <algorithm>
#include <iomanip>
#include <cmath>
#include "time.h"
```

Definições e Macros

- #define **RESETCOLOR** "\033[0m"
- #define **BOLDGREEN** "\033[1m\033[32m"
- #define **BOLDBLUE** "\033[1m\033[34m"
- #define **BOLDMAGENTA** "\033[1m\033[35m"
- #define **BOLDRED** "\033[1m\033[31m"
- #define **BOLDYELLOW** "\033[1m\033[33m"
- #define **BLACK** "\033[30m"
- #define **RED** "\033[31m"
- #define **GREEN** "\033[32m"
- #define **YELLOW** "\033[33m"
- #define **BLUE** "\033[34m"
- #define **MAGENTA** "\033[35m"
- #define **CYAN** "\033[36m"
- #define **WHITE** "\033[37m"
- #define **BOLDBLACK** "\033[1m\033[30m"
- #define **BOLDYELLOW** "\033[1m\033[33m"
- #define **BOLDCYAN** "\033[1m\033[36m"
- #define **BOLDWHITE** "\033[1m\033[37m"

Funções

- bool **compara_agilidade** (**Personagem** *a, **Personagem** *b)

Definições e macros

#define BLACK "\033[30m"

#define BLUE "\033[34m"

#define BOLDBLACK "\033[1m\033[30m"

#define BOLDBLUE "\033[1m\033[34m"

#define BOLDCYAN "\033[1m\033[36m"

#define BOLDGREEN "\033[1m\033[32m"

#define BOLDMAGENTA "\033[1m\033[35m"

#define BOLDRED "\033[1m\033[31m"

#define BOLDWHITE "\033[1m\033[37m"

#define BOLDYELLOW "\033[1m\033[33m"

#define BOLDYELLOW "\033[1m\033[33m"

#define CYAN "\033[36m"

#define GREEN "\033[32m"

#define MAGENTA "\033[35m"

#define RED "\033[31m"

#define RESETCOLOR "\033[0m"

#define WHITE "\033[37m"

#define YELLOW "\033[33m"

Funções

bool compara_agilidade (Personagem * *a*, Personagem * *b*)

```
31 {
32     return(a->get_agilidade() > b->get_agilidade());
33 }
```

Referência do Arquivo 20191_team_7/Personagem.cpp

```
#include "Personagem.h"
#include <string>
#include <iostream>
```

Referência do Arquivo 20191_team_7/Druida.h

```
#include "Personagem.h"
#include <iostream>
```

Componentes

- class **Druida**

Definições e Macros

- #define **CUSTO_HABILIDADE_1_DRUIDA** 30
- #define **CUSTO_HABILIDADE_2_DRUIDA** 30
- #define **CUSTO_HABILIDADE_3_DRUIDA** 60
- #define **INCREMENTO_ATAQUE_DRUIDA** 60
- #define **FATOR_CURA_DRUIDA** 0.70
- #define **FATOR_REVITALIZACAO_DRUIDA** 0.50

Definições e macros

#define CUSTO_HABILIDADE_1_DRUIDA 30

#define CUSTO_HABILIDADE_2_DRUIDA 30

#define CUSTO_HABILIDADE_3_DRUIDA 60

#define FATOR_CURA_DRUIDA 0.70

#define FATOR_REVITALIZACAO_DRUIDA 0.50

#define INCREMENTO_ATAQUE_DRUIDA 60

Referência do Arquivo 20191_team_7/Experiencia.h

```
#include <math.h>
```

Componentes

- class **Experiencia**

Definições e Macros

- #define **MULTIP_LEVEL_UP** 1.30
- #define **XP_MULTIPLICADOR** 1.25
- #define **XP_PROX_LEVEL** 100

Definições e macros

#define MULTIP_LEVEL_UP 1.30

#define XP_MULTIPLICADOR 1.25

#define XP_PROX_LEVEL 100

Referência do Arquivo 20191_team_7/Feiticeiro.h

```
#include "Personagem.h"
#include <iostream>
```

Componentes

- class **Feiticeiro**

Definições e Macros

- #define **CUSTO_HABILIDADE_FT_1** 25
- #define **CUSTO_HABILIDADE_FT_2** 20
- #define **CUSTO_HABILIDADE_FT_3** 30
- #define **DANO_BOLA_DE_FOGO** 40
- #define **FATOR_DRENAR** 0.20
- #define **FATOR_F_CURA** 0.30
- #define **DANO_RAIO_PARALIZANTE** 20

Definições e macros

#define CUSTO_HABILIDADE_FT_1 25

#define CUSTO_HABILIDADE_FT_2 20

#define CUSTO_HABILIDADE_FT_3 30

#define DANO_BOLA_DE_FOGO 40

#define DANO_RAIO_PARALIZANTE 20

#define FATOR_DRENAR 0.20

#define FATOR_F_CURA 0.30

Referência do Arquivo 20191_team_7/Guerreiro.h

```
#include "Personagem.h"
#include <iostream>
```

Componentes

- class **Guerreiro**

Definições e Macros

- #define **FATOR_CURA** 0.25
- #define **PONTO_DE_EXECUCAO** 0.25
- #define **CUSTO_HABILIDADE_1** 15
- #define **CUSTO_HABILIDADE_3** 15

Definições e macros

#define CUSTO_HABILIDADE_1 15

#define CUSTO_HABILIDADE_3 15

#define FATOR_CURA 0.25

#define PONTO_DE_EXECUCAO 0.25

Referência do Arquivo 20191_team_7/Jogo.h

```
#include <vector>
#include <Personagem.h>
#include <Monstro.h>
```

Componentes

- class **Jogo**

Referência do Arquivo 20191_team_7/Monstro.h

```
#include <string>
#include "Personagem.h"
```

Componentes

- class **Monstro**

Referência do Arquivo 20191_team_7/Partida.h

```
#include <Personagem.h>
#include <vector>
#include <iostream>
#include <string>
```

Componentes

- class **Partida**

Referência do Arquivo 20191_team_7/Personagem.h

```
#include "Experiencia.h"
#include <string>
#include <vector>
```

Componentes

- class **Personagem**

Conclusão

O projeto foi um grande desafio, pois por mais simples que seja o jogo, é algo complexo implementar um software com tantas variações. Mas ao mesmo tempo o projeto foi muito facilidado pelo uso do Github e pela utilização do paradgima de orientação a objetos, pois depois de pronta, uma classe não precisava sofrer alterações, (pelo menos não deveria), caso fosse necessário mudar outras coisas no código. É claro que houveram problemas devido a classes que ficaram muito grandes, com métodos que poderiam estar em outra classes, mas ainda assim foi possível corrigir essas falhas só implementando outras classes de maneira diferente.