PROGRAMAÇÃO ORIENTADA A OBJECTOS

META 1 | RELATÓRIO | ENGENHARIA INFORMÁTICA 2017/2018

GABRIEL SILVA :: 21250322 | INÊS DINIS :: 21260791

INSTITUTO SUPERIOR DE ENGENHARIA DE COIMBRA, DEZEMBRO 2017

Índice

1. Descrição do projecto

Nesta actividade pretende-se a implementação de um programa em C++ para simulação de populações de formigas, com recurso à consola gráfica, sendo que estas comunidades existem dentro de um mundo fechado. O programa consiste na apresentação do mundo, que é configurável através da inserção de comandos ou ficheiro de texto. Durante a simulação, é possível acrescentar e visualizar elementos do mundo através de instruções do utilizador.

Nesta primeira meta é pretendida a elaboração das classes necessárias ao funcionamento do programa, bem como as suas responsabilidades e objectivos. São também pretendidos os mecanismos necessários à configuração do mundo, leitura, validação e processamento de comandos do utilizador e, ainda, uma primeira visualização dos elementos existentes.

Nos capítulos que se seguem são descritas as classes identificadas e utilizadas no programa, os requisitos cumpridos nesta primeira fase, as funções implementadas e a validação da aplicação.

2. Requisitos cumpridos

As funcionalidades implementadas nesta primeira meta compreendem a criação das classes de formigas, ninho e migalhas que existirão dentro do mundo. Nos capítulos seguintes descrevem-se estas funcionalidades em maior detalhe.

Componentes	Cumprida	Parcialmente Cumprida	Não Cumprida
Leitura e validação de comandos	x		
Leitura de comandos em ficheiro	x		
Criação do mundo, ninhos e formigas	x		
Visualização do mundo e conteúdo		x	
Formiga exploradora (move)	x		
Avanço de iteração	x		
Listagens (listamundo, listaninho, listaposicao)	x		

3. Conceitos/Classes

Da especificação identificam-se as classes Mundo, Ninho/Comunidade, Formiga e Migalha, sendo cada uma composta por diversos atributos como identificador, coordenadas, energia e avatar (o símbolo com que cada elemento poderá ser representado).

As classes e respectivas funcionalidades consideradas na primeira versão testada da aplicação consistem em:

Formigas : são o elemento mais básico.
Composição:
- Identificador (atribuído automaticamente à nascença)
- Energia
- Coordenadas
- Raio de movimento e de visão
- Caracter para a sua representação

Características:
- Permitem obter e definir/alterar os seus atributos
- São também constituídas por diversos tipos de formiga em que apenas o seu comportamento é variável
- Os seus objectos são criados, armazenados e destruídos na classe Ninho.

Ninho : onde as formigas nascem e em que contexto existem.
Composição:
- Contém um vector onde armazena as suas formigas
- Identificador (automaticamente atribuído aquando da sua criação)
- Energia
- Coordenadas
- Cor
- Caracter para a sua representação

Características:
- Permite adicionar formigas e obter a sua informação textual
- Os objectos desta classe são criados, armazenados e destruídos na classe Mundo

Mundo : onde os ninhos e respectivas formigas são criados e existem.
Composição:
- Contém um vector que armazena ninhos (que por sua vez contém o vector de formigas)
- Limite para as suas dimensões
- Mapa que indica posições ocupadas do mundo por elementos

Características:
- Permite adicionar um número igual de formigas a todos os ninhos
- Adicionar um novo ninho
- Obter a descrição textual dos ninhos existentes
- Avanço de iterações

Migalha
Composição:
- Identificador
- Coordenadas
- Energia

Características:
Não aplicável nesta versão.

3.1. Responsabilidades e objectivos das classes

Encapsulamento

Seguindo os princípios de encapsulamento em que cada classe é responsável pelas suas funcionalidades, permitindo que estas sejam modificadas sem alterar o funcionamento do programa, podemos verificar, por exemplo, que a classe Formiga é responsável por listar a informação de cada um dos seus objectos (formigas), bastando para isso que a classe Ninho apenas lhe solicite essa listagem.

As funções para adicionar formigas são responsabilidade do Ninho visto que este contém as formigas e que cada uma pertence obrigatoriamente a um ninho.

Classes com objectivo focado, coeso e sem dispersão

Classe Formiga: concentra-se apenas nos dados relativos às formigas, como accções, coordenadas e obtenção das suas informações.

Classe Ninho: responsabiliza-se apenas pela sua própria existência e por criar formigas chamando, para isso, a classe Formiga, responsável pelo "nascimento" das formigas.

Responsabilidades de interface e lógica

Nesta primeira versão da aplicação, a interacção com o utilizador não é da responsabilidade máxima de uma classe, pelo esta é efectuada através da escrita de comandos. Contudo, a classe Mundo estabelece a ponte entre o utilizador e a aplicação, pois é através dela que ninhos e formigas poderão ser criados, bem como as suas dimensões serem estabelecidas e o avanço de iterações, pelo que recebe e processa as suas instruções, encaminhando-as para as classes respectivas.

As classes Mundo, Ninho, Formiga e futuramente Migalha são as que ficam responsáveis pela lógica e funcionamento da aplicação, pois são os elementos que existirão dentro do mundo criado.

A classe Mundo representa a envolvente de toda a lógica, permitindo obter a informação de qualquer classe, sendo apenas necessário solicitar-lhe a informação pretendida.

3.2. Principais classes da aplicação

Classe: Formiga
Responsabilidades:
- Criar formigas
- Obter informação de uma formiga
- Configurar atributos de uma formiga
- Mover uma formiga

Colaborações: Ninho

Classe: Ninho
Responsabilidades:
- Criar ninhos
- Adicionar (criar) formigas ao ninho
- Obter informação de um ninho
- Obter informações das suas formigas
- Configurar atributos de um ninho

Colaborações: Formiga, Mundo

Classe: Mundo
Responsabilidades:
- Criar mundos
- Verificar posições ocupadas
- Adicionar (criar) ninhos
- Adicionar (criar) formigas ao ninho
- Obter informação de um ninho
- Obter informação das formigas de um ninho
- Obter informação dos ninhos por coordenadas

Colaborações: Formigas, Ninho, Migalhas

Classe: Migalha
Responsabilidades:
- Não aplicável nesta versão

Colaborações: Formigas

4. Código

Nos capítulos anteriores apresentaram-se os conceitos e classes implementadas nesta versão da aplicação. Apresenta-se de seguida uma versão detalhada dos elementos do código da aplicação.

4.1. Classes

4.1.1. Formiga

```
class Ant{
 static int counter;
 char Avatar;
 int ID;
 int energy;
 int PosX, PosY; //movimento coordenadas
 int Rvisao,Rmov; //raio de visão e de movimento
 int limite;
 public:
     Ant(int a=50, int b=0, int c=0, int d=0, int e=0, char f='*', int
g=0):energy(a),PosX(b), PosY(c),Rvisao(d),Rmov(e),Avatar(f),
limite(g){ ID = ++counter; };
     char getAvatar() const {return Avatar;}
     int getID() const{return ID;}
     int getEnergy() const{return energy;}
     int getPosX() const{return PosX;}
     int getPosY() const{return PosY;}
     int getRvisao() const{return Rvisao;}
     int getRmov() const{return Rmov;}
     string getInfo(){
      ostringstream os;
      os << "ID: " << getID() << " Energy: " << getEnergy() << " Pos:
      " << getPosX() << " " << getPosY() << " Raio Visao: " <<
      getRvisao() << " Raio Mov: " << getRmov();
      return os.str();
     }
     void setEnergy(int add){ energy += add; }
     void setPosX(int add){ PosX += add;}
     void setPosY(int add){ PosY +=add;}
     ~Ant(){};
};
```

A classe *Ant* (Formiga) é constituída por um ID que é atribuído automaticamente de cada vez que uma formiga é criada. Como referido anteriormente, a classe é responsável por todos os dados e acções das formigas, bem como da obtenção da sua informação textual.

4.1.2. Ninho

```
class Nest{
  static int counter;
  char avatar;
  int ID;
  int energy, nova, transfere;
  vector <Ant*> ants;
  int PosX_n, PosY_n;
  WORD corNinho;
  public:
      Nest(int a, int b, int c, int d, int e, WORD cor, char f=
      'O'):energy(a),nova(b),transfere(c),PosX_n(d),PosY_n(e),corNinho
      (cor),avatar(f){ ID = ++counter;};
      int getID() const{return ID;}
      int getEnergy() const{return energy;}
      int getNova() const{return nova;}
      int getTransfere() const{return transfere;}
      int getPosX() const{return PosX_n;}
      int getPosY() const{return PosY_n;}
      char getAvatar() const{return avatar;}
      WORD getCorNinho() const{return corNinho;}
      string getAntsInfo() const{
        ostringstream os;
        for(auto it=ants.begin(); it < ants.end();it++){
          os << (*it)->getInfo() << endl;
        }
        return os.str();
      }
      string getInfoGeral() const{
        ostringstream os;
        os << "Ninho " << getID() << endl;
        os << "Posicao: " << getPosX() << " " << getPosY() << endl;
        os << "---Informacoes gerais---" << endl;
        os << "Energia: " << getEnergy()<< endl;
        os << "Nova: " << getNova() << endl;
        os << "Transferencia: " << getTransfere() << endl;
        os << "Numero de formigas: " << ants.size() << "\n" << endl;
        return os.str();
      }
      void addFormigas(int num,int x, int y, int limite);
      void andar();
       ~Nest(){
        auto i=ants.begin();
        for (; i<ants.end();i++){
                delete (*i);
        }
       };
     WORD getColor() const{return corNinho;}
};
```

A classe *Nest*/Ninho é responsável por criar ninhos e acrescentar-lhes formigas, que ficarão guardadas no vector de formigas. É responsável por obter a informação das suas formigas, bem como os dados e informação textual do ninho.

4.1.3. Mundo

```
class Mundo
{
    int limite;
    vector <Nest*> ninhos;
    int **mapa;

    public:
        Mundo(int limite){
            mapa = new int*[limite];
            for(int i=0; i < limite ; i++){
                mapa[i]= new int[limite];
            }
        };
        void setMapa(int x,int y){
            mapa[x][y]=1;
        }
        string getInfo() const;
        void newNinho(config_t inicial, int x, int y);
        void addFormigas(int num, int ID ,int x = -1,int y = -1);
        string getNinho(int ID) const;
        bool verificaPos(int x, int y);
        string getInfoCoord(int x, int y);
        void avancar(int num);
        string getInfoAntsNinho(int ID) const;
        ~Mundo(){
            for(int i=0;i<limite;i++){
                //delete[] mapa[i]; //não está a funcionar
            }
            delete[] mapa;
            auto i=ninhos.begin();
                for (; i<ninhos.end();i++){
                    delete (*i);
                }
        }
};
```

A classe Mundo contém o atributo que define as suas dimensões, o vector que guarda os ninhos que lhe são acrescentados e, ainda, uma matriz que apenas guarda as posições ocupadas de cada vez que um elemento é criado/removido. É possível adicionar novos ninhos e formigas, bem como obter a informação geral do mundo ou de ninhos em particular. É também a classe que estará responsável pelo avanço de iterações do mundo.

4.1.4. Migalha

```
class Migalha{
  int ID;
  int energy;
  int PosX, PosY;
  public:
      Migalha();
      ~Migalha();
};
```

Nesta versão da aplicação a classe Migalha ainda não se encontra implementada, contendo apenas os atributos necessários ao seu futuro funcionamento

4.2. Funções utilizadas

As funções utilizadas nesta versão estão divididas por diversos ficheiros de forma a não sobrecarregar excessivamente o código. As funções das classes também se encontram separadas em ficheiros, tendo sido apresentados os seus ficheiros *.h* anteriormente. De seguida, apenas se apresentam as restantes funções necessária ao funcionamento da aplicação, localizadas nos respectivos ficheiros *.cpp*.

4.2.1. Main.cpp

```
int uniform01(int lower, int upper){
    static default_random_engine e;
    static uniform_int_distribution<int> u(lower,upper);
    return u(e);
}
int main(){
  string command, param;
  vector<string>comm_list;
  int arg;
  comm_list=load_commands("command_mundo.txt"); //inicializa comandos
  string c, l;
  config_t configs;

  do{
    Consola::clrscr();
    Consola::gotoxy(0,15);
  cout << "Insira comando.: para sair escreva 'sair'" << endl;
  getline(cin,command);
  if (command == "sair") {
        cout << "encerrando" << endl; break; }
  else if(check_command(command,comm_list)==false){
    cout << "Comando invalido" << endl;
    return 0;
  }

  if (command != "executa" && command != "inicio"){ // se for executa
ou inicio não vale a pena estar a pedir parâmetros
      cout << "Insira parametro.: " << endl;
      getline(cin,param);
      arg = atoi(param.c_str());
      cout << listaComandos(comm_list);
  }
  int num = whichCommand(comm_list, command, arg, configs);
  }while(1);
  return 0;
}
```

A função *uniform01* está de momento definida neste ficheiro, devendo ser corrigida numa próxima versão visto que apenas é necessária para o movimento aleatório das formigas.

A função main entra num ciclo que solicita um comando ao utilizador. Antes de verificar se o comando é válido, verifica se o utilizador escreveu "sair", evitando fazer verificações desnecessárias. Um método semelhante é utilizado caso o utilizador insira os comandos "executa" ou "inicio", avançando logo para a próxima instrução pois não precisam que seja inserido nenhum parâmetro adicional. A função whichCommand é responsável por reencaminhar para as funções respectivas, consoante o comando que o utilizador inseriu. É utilizada uma estrutura para guardar os parâmetros da configuração inicial.

4.2.2. Commands.cpp

4.2.2.1. Carregar comandos

```
vector<string> load_commands(string ficheiro){
  vector<string>comm_list;
  ifstream fs;
  string command;
  fs.open(ficheiro);
  while(getline(fs, command)){
    comm_list.push_back(command);
  }
  fs.close();
  return comm_list;
}
```

A função *load_commands()* recebe o nome de um ficheiro *.txt* onde está guardada a lista de comandos disponíveis. A função cria um vector para onde irá ler cada um dos comandos, devolvendo-o à função *main*.

4.2.2.2. Verificar se um comando é válido

```
bool check_command(const string& command,const vector<string>&list){
  int spaces,exists;
  if((spaces = space_count(command))!=0)
    return false;
  if ((exists = check_existence(command,list))==false)
    return false;
  return true;
}
```

Esta função recebe o comando inserido e o vector com a lista de comandos aceites. Antes de verificar se o comando existe, verifica se o comando foi inserido com algum espaço.

4.2.2.3. Verificar se um comando existe

```
bool check_existence(const string& command,const vector<string>&list){
  for(int i= 0;i<list.size();i++){
    if(command == list[i])
      return true;
  }
  return false;
}
```

Esta função é chamada pela *check_command()*, recebendo o vector com os comandos aceites e o comando introduzido pelo utilizador, verificando se este coincide com algum existente na lista.

4.2.2.4. Listar os comandos aceites

```
string listaComandos(const vector<string>&comm_list){
    ostringstream os;
    for(int i= 0;i<comm_list.size();i++)
    os << comm_list[i] << endl;
    return os.str();
}
```

Uma função simples que apenas percorre o vector com os comandos e apresenta-os na consola. Poderá ser útil para permitir ao utilizador visualizar os comandos que pode escrever caso não os saiba.

4.2.2.5. Executa a acção correspondente ao comando

```
int whichCommand(const vector<string>&comm_list, const string
&command, int arg, config_t &inicial){
    if (command == "defmundo")
      if(arg <= 10 && arg > 0)
        inicial.lim = arg;
      else{
        Consola::clrscr();
        Consola::gotoxy(0,15);
        cout << "Valor introduzido muito alto\nPrima uma tecla para
tentar de novo";
        Consola::getch();
      }
    else if (command == "defen")
        inicial.energiaNinho = arg;
    else if (command == "defpc")
        inicial.energiaLim = arg;
    else if (command == "defvt")
        inicial.energiaTransf = arg;
    else if (command == "defmi")
        if(arg <= 100 && arg >0)
          inicial.percentMigalh = arg;
        else
          cout << "Valor introduzido muito alto";
    else if (command == "defme")
        inicial.energiaMigalh = arg;
    else if (command == "defnm")
        inicial.maxMigalhInst = arg;
    else if (command == "executa")
        leExecuta(comm_list,inicial);
    else if (command == "inicio")
        if(inicial.lim != -1 && inicial.energiaLim != -1 &&
inicial.energiaTransf != -1 && inicial.energiaNinho != -1){
            segundosComandos(inicial);
        }
        else{
            Consola::clrscr();
            Consola::gotoxy(0,15);
            cout << "Ainda nao executou  todas as configuracoes
iniciais\nPrima uma tecla para tentar de novo";
            Consola::getch();
        }

    return 1;}
```

Esta função preenche os campos da estrutura recebida por referência ou chama as funções correspondentes aos comandos *executa* e *inicio*. Deverá ser garantido que a função *segundosComandos()* só é chamada caso todos os campos da estrutura estejam preenchidos, pelo que é feita essa verificação.

4.2.2.6. Conta número de espaços numa string

```
int space_count(const string& verify){
  int nspaces = 0;
  for (int i=0; i<verify.size();i++)
    if(verify[i] == ' ')
      nspaces++;

  return nspaces;
}
```

Esta função conta e devolve o número de espaços existentes numa string.

4.2.2.7. Leitura de comandos em ficheiro de texto

```
void leExecuta(const vector<string>&comm_list,config_t &inicial){
  ifstream fs ("executa.txt");
  string command;
  string arg;

  if(fs.is_open()){

      while(getline(fs,command)){
        getline(fs,arg);
        cout << "Li o comando " << command << " com o argumento " <<
arg << endl;
        if(check_command(command,comm_list)){
          if(command != "executa")
            whichCommand(comm_list, command,
atoi(arg.c_str()),inicial);
          else{
            cout << "Este comando nao existe";
            return;
          }
        }
      }
  }
}
```

A função abre o ficheiro de texto *executa.txt* onde deverão ser indicados os comandos e os argumentos em diferentes linhas. Chama a função de verificação e, caso não seja o comando *executa (*pois não queremos entrar em ciclo infinito a chamar a própria *executa()*) chama a função *whichCommand()* enviando-lhe as instruções necessárias para preencher o campo correcto da estrutura.

4.2.3. Simulacao.cpp

4.2.3.1. Solicitação de novos comandos e parâmetros

```
void segundosComandos(config_t inicial){
  string command, param, arg1,arg2,arg3;
  vector<string>comm_list;
    Mundo a(inicial.lim);
  comm_list=load_commands("command_simul.txt");
  cout << "Iniciando simulacão " << endl;
  do{
    Consola::clrscr();
    Consola::gotoxy(0,15);
      cout << "[SIMUL]Insira comando.: para sair escreva 'sair'" <<
endl;
      getline(cin,command);
      if (command == "sair"){
         cout << "[SIMUL] encerrando" << endl; exit(1);
      }
      else if(check_command(command,comm_list)==true){
        if (command == "ninho"){
            do{
                cout << "Linha: ";
                getline(cin, arg1);
                cout << "\nColuna: ";
                getline(cin, arg2);
            }while(atoi(arg1.c_str()) >= 10 || atoi(arg2.c_str()) >=
10 || atoi(arg1.c_str()) < 0 || atoi(arg2.c_str()) < 0);
            Consola::clrscr();
            a.newNinho(inicial, atoi(arg1.c_str()),
atoi(arg2.c_str()));
            Consola::gotoxy(0,15);
            cout << "Prima uma tecla para voltar as opcoes";
            Consola::getch();
        }
        else if (command == "criaf"){
            cout << "Numero de formigas: ";
            getline(cin, arg1);
            cout << "\nID do ninho:";
            getline(cin, arg2);
            Consola::clrscr();
            a.addFormigas(atoi(arg1.c_str()), atoi(arg2.c_str()));
            Consola::gotoxy(0,15);
            cout << "Prima uma tecla para voltar as opcoes";
            Consola::getch();
        }
        else if (command == "cria1"){
            cout << "ID do ninho:";
            getline(cin, arg1);
            do{
                cout << "\nx:";
                getline(cin, arg2);
                cout << "\ny:";
                getline(cin, arg3);
            }while(atoi(arg2.c_str()) >= 10 || atoi(arg3.c_str()) >=
10 || atoi(arg2.c_str()) < 0 || atoi(arg3.c_str()) < 0);
            Consola::clrscr();
```

```
a.addFormigas(1,atoi(arg1.c_str()),atoi(arg2.c_str()),atoi(arg3.c_str(
)));
            Consola::gotoxy(0,15);
            cout << "Prima uma tecla para voltar as opcoes";
            Consola::getch();
        }
        else if (command == "listamundo"){
            Consola::clrscr();
            Consola::gotoxy(0,0);
            cout << a.getInfo();
            cout << "Prima uma tecla para voltar as opcoes";
            Consola::getch();
        }
        else if (command == "listaninho"){
            cout << "ID do ninho:";
            getline(cin, arg1);
            Consola::clrscr();
            Consola::gotoxy(0,0);
            cout << a.getNinho(atoi(arg1.c_str()));
            cout << "Prima uma tecla para voltar as opcoes";
            Consola::getch();
        }
        else if (command == "listaposicao"){\
            cout << "X:";
            getline(cin, arg1);
            cout << "\nY:";
            getline(cin, arg2);

if(a.verificaPos(atoi(arg1.c_str()),atoi(arg2.c_str()))==false){
                Consola::clrscr();
                Consola::gotoxy(0,0);
                cout << a.getInfoCoord(atoi(arg1.c_str()),
atoi(arg2.c_str()));
            }
            else
                cout << "Posicao esta vazia";
            cout << "Prima uma tecla para voltar as opcoes";
            Consola::getch();
        }
        else if (command == "tempo"){
            cout << "Numero de iteracoes: ";
            getline(cin, arg1);
            a.avancar(atoi(arg1.c_str()));
          }
    }
  }while(1);
}
```

Ainda uma função implementada de forma provisória. A função solicita novos comandos e parâmetros ao utilizador, necessários para adicionar elementos ao mundo ou avançar iterações. Tal como anteriormente, caso seja introduzido o comando *sair*, não avança para as próximas instruções.

5. Validação e pontos em falta na aplicação

A aplicação foi testada a nível de inserção de comandos do utilizador e através da criação de objectos directamente na função *main* do programa. Já é possível criar um mundo, adicionando-lhe ninhos e formigas. A aplicação imprime na consola os elementos aquando da sua criação.

Relativamente a falhas e comportamentos anómalos, testou-se a inserção de parâmetros inválidos e o programa já tem a capacidade de os reconhecer, apresentando o respectivo erro.

Os erros/falhas detectados são maioritariamente devido a pontos em falta na aplicação, portanto a lista que se apresenta serve maioritariamente para orientação para o processo de desenvolvimento:

- Durante a simulação é possível utilizar o comando *criaf* para acrescentar diversas formigas. Contudo, se utilizar esse comando para criar apenas uma formiga o programa dá erro
- O programa verifica posições ocupadas para a adição de ninhos. Contudo, ainda é possível acrescentar diversas formigas na mesma posição
- Falta verificar se o ficheiro com os comandos principais existe. Caso contrário, o programa não deverá prosseguir
- O programa ainda não contém uma função para apresentar todos os elementos do mundo em simultâneo. Consegue apresentar os ninhos e as formigas, mas apenas no momento da criação e não durante a simulação
- Os destrutores das classes encontram-se funcionais. Contudo, o destrutor da classe Mundo, se tentar eliminar a matriz das posições ocupadas rebenta a aplicação
- Relativamente à interface com o utilizador, de momento é solicitado o comando e os parâmetros em separado. Será melhor permitir que o utilizador insira logo o comando e o parâmetro numa só linha.

6. Conclusões

Nesta primeira fase foram criadas as classes base necessárias para o funcionamento da aplicação, bem como as funções de leitura e validação de comandos, sendo possível criar um mundo, ninhos e formigas.

Os próximos passos serão a correcção da falha detectada com a utilização do comando *criaf* para apenas 1 formiga, bem como a verificação de posições ocupadas por formigas. O erro ao destruir a matriz pertencente à classe mundo também deverá ser corrigido.

Serão também criados os diferentes tipos de formigas e as suas variações de comportamento e, também a existência de migalhas que fornecem energia às formigas.

Por fim, deverá ser possível que a aplicação apresente na consola os elementos do mundo ao longo da simulação, utilizando para tal um comando ou, mais interessante ainda, apresentar em metade do ecrã o mundo e na outra metade a secção que solicita os comandos. Será também acrescentado um comando adicional, por exemplo "*help*", que permita ao utilizador que desconhece a aplicação visualizar uma listagem dos comandos que pode utilizar.