Société C++ UML

Introduction

Un société est composée d'individus. Les individus sont des hommes ou des femmes. Les hommes et les femmes peuvent donner naissance à d'autres individus. Au départ, seuls Adam, un homme, et Eve, une femme, existent dans la société. Ils donnent naissance à des individus qui eux mêmes donnent naissance à d'autres individus, etc. Un individu possède donc un père, une mère, des fils et des filles. Un individu peut malheureusement mourir. Les hommes et les femmes peuvent se marier ou se séparer.

Objectif et méthode

En annexe figure un programme C++ modélisant cette société. Le but de l'examen est de comprendre ce que fait ce programme, de corriger les erreurs qui y sont encore contenues et de l'étendre.

Question 1

Dans un souci de présentation, le programme situé en annexe a été modifié par rapport au vrai programme, celui qui a été écrit et compilé avec g++. Le vrai programme a été écrit de façon modulaire dans des fichiers d'en-tête .h et de définitions .cpp. Pour Individu, Homme, Femme,

a) donner les fichiers .h avec :
- les #include de .h non prédéfinis par le langage C ou C++,
- les instructions class Machin; où Machin est une classe définie dans un autre .h du logiciel.

NB : on n'écrira pas les déclarations de classes figurant déjà en annexe, ni les #ifndef #define #endif.

b) donner les fichiers .cpp avec les #include de .h non prédéfinis par le langage C ou C++.
NB : on n'écrira pas les définitions de fonctions membres.

Question 2

a) Dans le programme principal, les deux instructions suivantes :

```
Societe * s;
while (s->menu()!='Q');
```

sont-elles correctes ?

b) Si on avait mis :

```
Societe s;
while (s.menu() !='Q');
```

cela serait-il correct ?

c) En java, si on avait mis :

```
Societe * s;
while (s->menu()!='Q');
```

cela serait-il correct ?

d) En java, si on avait mis :

```
Societe s;
while (s.menu()!='Q');
```

cela serait-il correct ?

Choisir une solution C++ correcte que l'on suppose acquise pour la suite.

Question 3

a) Quand on le lance, le programme société actuel se plante lors de la construction de l'individu `adam`. Préciser le lieu exact du plantage et ajouter les deux instructions nécessaires à l'endroit adéquat.

b) A nouveau le programme société se plante lors de la construction de l'individu `eve`. Préciser le lieu exact du plantage et ajouter l'instruction nécessaire à l'endroit adéquat.

Question 4

On suppose que l'utilisateur tape **A** pour afficher tous les individus de la société. Donner la sortie du programme. Que se passe-t-il ? Modifier le programme et donner la nouvelle sortie du programme.

Question 5

a) Dessiner le *diagramme de généralisation* UML de ce programme.
b) Dessiner le *diagramme classe-association* du programme.
c) Quelles sont les méthodes redéfinies ?

Question 6

a) Ce programme permet-il de créer plusieurs individus ?
b) Quelle est la méthode qui correspond à une interaction homme-machine ?
c) Peut-on faire naître des individus de parents non mariés ? de parents de même sexe ? l'inceste est-il possible ?
d) dessiner un *diagramme de séquence* UML correspondant à l'interaction de l'utilisateur avec ce programme en supposant que **adam** et **eve** ont une fille **lrbb**, que **lrbb** et **adam** ont un fils **bhc** puis que l'utilisateur quitte le programme.

Question 7

On suppose que l'utilisateur et le programme ont suivi le diagramme de séquence de la question précédente. Dessiner le *diagramme d'instances* UML correspondant à l'état du programme juste avant que l'utilisateur quitte le programme.

Question 8

a) On suppose que l'utilisateur et le programme sont encore dans l'état du diagramme d'instances de la question précédente et que l'utilisateur tape **M** puis **lrbb** pour que lrbb meurt, puis **A** pour afficher tous les individus de la société. Donner la sortie du programme. Que se passe-t-il ?
b) Pour résoudre ce problème, on déclare les méthodes suivantes :

```
void cetEnfantEstMort(Individu *)
void mesEnfantsJeSuisMort(Liste *)
```

dans la classe `Individu`. Donner les définitions de ces deux méthodes. On pourra utiliser les appels aux méthodes `premier()`, `prochain()` et `enlever(Objet*)` de la classe `Liste`.
c) Ajouter les appels adéquats aux méthodes définies dans la méthode `mort()` afin de résoudre ce problème. Donner la nouvelle sortie du programme.

Question 9

a) On suppose à nouveau que l'utilisateur et le programme sont dans l'état du diagramme d'instances de la question 7 et que l'utilisateur tape **M** pour que **adam** meurt, puis **A** pour afficher tous les individus de la société. Donner la sortie du programme. Que se passe-t-il ?
Pour résoudre ce problème, on déclare une méthode

```
void monConjointJeSuisMort()
```

dans la classe `Individu`. Donner sa définition dans la classe `Individu`.
b) Faut-il la redéfinir dans les sous-classes `Homme` et `Femme` ? Si oui , donner ces redéfinitions.
c) Ajouter l'appel adéquat à la méthode `monConjointJeSuisMort()` dans la méthode `mort()` afin de résoudre le problème.

Question 10

On souhaite donner la possibilité au logiciel de *cloner* un individu *original* en individus *copies* de cet original. Pour cela, la spécification d'interface est la suivante : on rajoute une ligne **`C. Clonage`** dans le menu. Si l'utilisateur tape **`C`**, le logiciel demande le nom de l'original à l'utilisateur et crée un nouvel individu de même sexe que l'original. On suppose que le nom du clone est celui de l'original suivi su suffixe `Clone` (si on clone `adam`, ce clone aura pour nom `adamClone`). On décide donc de créer deux méthodes : `clonage()` et `imprimerOriginalEtCopies()`, et deux attributs : `monOriginal` de type `Individu*` et `mesCopies` de type `Liste*` dans `Individu`.

a) écrire la méthode de classe `clonage()` en utilisant le constructeur existant et sans utiliser de constructeur par recopie.
b) écrire la methode `imprimerOriginalEtCopies` et modifier les méthodes `imprimer()`et `menu()`
c) modifier le constructeur et le destructeur de la classe `Individu`,
d) modifier la méthode `mort()` (pour cela on pourra écrire des méthodes `monOriginalJeSuismort(Individu*)` et `mesCopiesJeSuisMort(Liste*)` sur le modèle de la question 8).

Annexe

```
int main() {
  cout << "bonjour" << endl;
  Societe * s;
  Homme * adam = new Homme("adam");
  Femme * eve = new Femme("eve");
  adam->maFemme = eve;
  eve->monHomme = adam;
  while (s->menu()!='Q');
  cout << "au revoir" << endl;
}

class Societe { public:
  char menu();
};

char Societe::menu() {
  cout << "Societe :" << endl;
  cout << "\t N. Naissance" << endl;
  cout << "\t M. Mort" << endl;
  cout << "\t G. mariaGe" << endl;
  cout << "\t S. Separation" << endl;
  cout << "\t A. Affichage" << endl;
  cout << "\t Q. Quitter" << endl;
  char r; cin >> r;
  switch(r) {
  case 'N': Individu::naissance(); break;
  case 'M': Individu::mort(); break;
  case 'G': Individu::mariaGe(); break;
  case 'S': Individu::separation(); break;
  case 'A': Individu::imprimerTout(); break;
  }
  return r;
}

#define TAILLE_NOM 32

class Individu : public Objet { public:
  static Liste * mesInstances;
  char nom[TAILLE_NOM];
  Homme * monPere;
  Femme * maMere;
  Liste * mesFils;
  Liste * mesFilles;
  Individu(char *);
  virtual ~Individu();
  void imprimer();
  virtual void imprimerNom();
  virtual void imprimerConjoint();
  void imprimerEnfants();
  void imprimerParents();
  static void naissance();
  static void mort();
  static void mariage();
  static void separation();
  static void imprimerTout();
  static Individu * getInstance(char *);
};
```

```
Liste * Individu::mesInstances = NULL;


Individu::Individu(char * n) {
  strcpy(nom, n);
  monPere = NULL;
  maMere = NULL;
  mesFils = new Liste();
  mesFilles = new Liste();
  mesInstances->ajouter(this);
}

Individu::~Individu() {
  mesInstances->enlever(this);
  delete mesFilles;
  delete mesFils;
}

void Individu::imprimerNom() {
  cout << "individu " << nom << endl;
}

void Individu::imprimer() {
  imprimerNom();
  imprimerConjoint();
  imprimerParents();
  imprimerEnfants();
}

void Individu::imprimerConjoint() { ; }

void Individu::imprimerEnfants() {
  cout << "mes fils: "  << endl;
  mesFils->imprimerNom();
  cout << "mes filles: "  << endl;
  mesFilles->imprimerNom();
}

void Individu::imprimerParents() {
  cout << "mon pere: ";
  monPere->imprimerNom();
  cout << "ma mere: ";
  maMere->imprimerNom();
}

void Individu::imprimerTout()
{
  int n = 0;
  Individu * i = (Individu*) mesInstances->premier();
  if (i==NULL) return;
  cout << "-------- " << endl << "INDIVIDU " << ++n << endl;
  i->imprimer(); cout << "-------- " << endl;
  for (;;) {
    i = (Individu*) mesInstances->prochain();
    if (i==NULL) return;
    cout << "-------- " << endl << "INDIVIDU " << ++n << endl;
    i->imprimer(); cout << "-------- " << endl;
  }
}
```

```
void Individu::naissance()
{
  char sh[TAILLE_NOM];
  cout << "nom pere ? ";
  cin >> sh;
  Homme * h = Homme::getInstance(sh);
  if (!h) {
    cout << "pas d'homme portant ce nom." << endl ;
    return;
  }
  char sf[TAILLE_NOM];
  cout << "nom mere ? ";
  cin >> sf;
  Femme * f = Femme::getInstance(sf);
  if (!f) {
    cout << "pas de femme portant ce nom." << endl;
    return;
  }

  int sexe = random() % 2;
  char s [TAILLE_NOM];
  nomHasard(s);

  Individu * enfant;
  switch(sexe) {
  case 0:
    enfant = new Homme(s);
    h->mesFils->ajouter(enfant);
    f->mesFils->ajouter(enfant);
    break;
  case 1:
    enfant = new Femme(s);
    h->mesFilles->ajouter(enfant);
    f->mesFilles->ajouter(enfant);
    break;
  }
  enfant->monPere = h;
  enfant->maMere = f;
  enfant->imprimer();
}


void Individu::mort()
{
  char s[TAILLE_NOM];
  cout << "nom ? ";
  cin >> s;
  Individu * i = getInstance(s);
  if (i) {
    delete i;
    cout << "individu mort." << endl;
  }
  else cout << "pas d'individu portant ce nom." << endl;
}
```

```
void Individu::mariage()
{
  char sh[TAILLE_NOM];
  cout << "nom homme ? ";
  cin >> sh;
  Homme * h = Homme::getInstance(sh);
  if (!h) {
    cout << "pas d'homme portant ce nom." << endl;
    return;
  }
  char sf[TAILLE_NOM];
  cout << "nom femme ? ";
  cin >> sf;
  Femme * f = Femme::getInstance(sf);
  if (!f) {
    cout << "pas de femme portant ce nom." << endl;
    return;
  }
  h->maFemme = f;
  f->monHomme = h;
  cout << "homme et femme maries." << endl;
}
```

```
void Individu::separation()
{
  char sh[TAILLE_NOM];
  cout << "nom homme ? ";
  cin >> sh;
  Homme * h = Homme::getInstance(sh);
  if (!h) {
    cout << "pas d'homme portant ce nom." << endl;
    return;
  }
  char sf[TAILLE_NOM];
  cout << "nom femme ? ";
  cin >> sf;
  Femme * f = Femme::getInstance(sf);
  if (!f) {
    cout << "pas de femme portant ce nom." << endl;
    return;
  }
  f->monHomme = NULL;
  h->maFemme = NULL;
  cout << "homme et femme separes." << endl;
}
```

```
Individu * Individu::getInstance(char * n)
{
  Individu * i = (Individu*) mesInstances->premier();
  if (i==NULL) return NULL;
  if (strcmp(i->nom, n)==0) return i;
  for (;;) {
    i = (Individu*) mesInstances->prochain();
    if (i==NULL) return NULL;
    if (strcmp(i->nom, n)==0) return i;
  }
}
```

```
class Homme : public Individu {

public:

  static Liste * mesInstances;
  Femme * maFemme;

  Homme(char *);
  ~Homme();

  void imprimerNom();
  void imprimerConjoint();
  static Homme * getInstance(char *);

};


Liste * Homme::mesInstances = NULL;


Homme::Homme(char * n) : Individu(n) {
  mesInstances->ajouter(this);
  maFemme = NULL;
}


Homme::~Homme() {
  mesInstances->enlever(this);
}


void Homme::imprimerNom() {
  cout << "homme " << nom << endl;
}


void Homme::imprimerConjoint() {
  cout << "mon conjoint: ";
  if (maFemme) cout << maFemme->nom << endl;
  else cout << "null" << endl;
}


Homme * Homme::getInstance(char * n) {
  Homme * i = (Homme*) mesInstances->premier();
  if (i==NULL) return NULL;
  if (strcmp(i->nom, n)==0) return i;
  for (;;) {
    i = (Homme*) mesInstances->prochain();
    if (i==NULL) return NULL;
    if (strcmp(i->nom, n)==0) return i;
  }
}
```

```
class Femme : public Individu {

public:

  static Liste * mesInstances;
  Homme * monHomme;

  Femme(char *);
  ~Femme();

  void imprimerNom();
  void imprimerConjoint();
  static Femme * getInstance(char *);

};


Liste * Femme::mesInstances = NULL;


Femme::Femme(char * n) : Individu(n) {
  mesInstances->ajouter(this);
  monHomme = NULL;
}


Femme::~Femme() {
  mesInstances->enlever(this);
}


void Femme::imprimerNom() {
  cout << "femme " << nom << endl ;
}


void Femme::imprimerConjoint() {
  cout << "mon conjoint: ";
  if (monHomme) cout << monHomme->nom << endl;
  else cout << "null" << endl;
}


Femme * Femme::getInstance(char * n)
{
  Femme * i = (Femme*) mesInstances->premier();
  if (i==NULL) return NULL;
  if (strcmp(i->nom, n)==0) return i;
  for (;;) {
    i = (Femme*) mesInstances->prochain();
    if (i==NULL) return NULL;
    if (strcmp(i->nom, n)==0) return i;
  }
}
```

```
class Objet {
public:
  Objet();
 ~Objet();
  virtual void imprimer();
  virtual void imprimerNom();
};
Objet::Objet() {}
Objet::~Objet() {}
void Objet::imprimer() { ; }
void Objet::imprimerNom() { ; }



class Liste {
  // méthodes inutiles pour comprendre le problème non représentees
  void imprimer();
  void imprimerNom();
  void ajouter(Objet *); // ajoute l'objet dans la liste.
  int enlever(Objet *);  // enlève l'objet de la liste.
  Objet * premier();     // retourne le premier objet la liste.
  Objet * prochain();    // retourne le prochain objet la liste.
};

// les definitions de la classe Liste sont ici.

void Liste::imprimer()
{
  Objet * ptr = (Objet *) premier();
  if (ptr) ptr->imprimer();
  else return;
  for ( ;; ) {
    ptr = (Objet *) prochain();
    if (ptr) ptr->imprimer();
    else break;
  }
}

void Liste::imprimerNom()
{
  Objet * ptr = (Objet *) premier();
  if (ptr) ptr->imprimerNom();
  else return;
  for ( ;; ) {
    ptr = (Objet *) prochain();
    if (ptr) ptr->imprimerNom();
    else break;
  }
}

void nomHasard(char * s)
{
  int l = 3 + (random() % 3);
  char c[8];
  for (int i=0; i<l; i++) c[i] = 'a' + (random() % 26);
  c[l] = '\0';
  strcpy(s, c);
}
```