Práctico de Programación 4 – clase 11

Práctico 6, Ejercicio 4:

En todos los casos se implementará la clase A, ya que su implementación es la que varía de acuerdo a los diferentes diseños, desde a) hasta h). Se distinguen cinco grandes casos:

i) Diagrama a). La clase A mantiene en todo momento una referencia a una instancia de clase B.

```
class A {
      private:
            B *miB;
      public:
            A(B *);
            // Siempre que se construye un objeto de clase A
            // se debe proporcionar un objeto de clase B
            B *getMiB();
};
```

ii) Diagrama b). La clase A mantiene una referencia a una instancia de clase B, pero no necesariamente en todo momento.

```
class A {
      private:
            B *miB;
      public:
            A();
            // El constructor no requiere un objeto de clase B
            B *getMiB();
            void setMiB(B *);
            // En cualquier momento se permite setear
            // el objeto de clase B (o NULL para ningún objeto)
};
```

iii) Diagramas c), d) y e). La clase A mantiene una colección de referencias a instancias de clase B. Las diferencias entre estos tres casos se deben a los mínimos y máximos en la cantidad de elementos de la colección.

```
class A {
      private:
            ColeccionB *misB;
      public:
            A();
            // Se inicializa la colección misB vacía
            void agregarUnB(B *);
            // Se agrega un elemento a la colección misB
            void removerUnB(B *);
            // Se remueve un elemento de la colección misB
            // pero no se libera su memoria
};
```

iv) Diagrama f). Similar a i) pero en este caso la clase A es responsable de destruir la instancia de B.

```
class A {
      private:
            B *miB;
      public:
            A(B *);
            ~A(); // Hace delete miB
            B *getMiB();
};
```

v) Diagramas g) y h). Similar a iii) pero en este caso la clase A es responsable de destruir las instancias de B.

```
class A {
      private:
            ColeccionB *misB;
      public:
            A();
            ~A(); // Hace delete de cada elemento de misB
            void agregarUnB(B *);
            void removerUnB(B *);
};
```

Práctico 6, Ejercicio 9:

La solución se basa en el código presentado en la solución al Ejercicio 12 del Práctico 1 en las notas de la Clase 2 de práctico. Se introducen las siguientes modificaciones relevantes:

- Los empleados son identificados por su cédula de identidad.
- Los jornaleros tienen una colección de jornales (fecha y cantidad de horas) en lugar de la cantidad acumulada de horas trabajadas.
- En lugar de utilizar una implementación a medida para las colecciones, utilizamos la biblioteca STL (Standard Template Library) de C++. En letra negrita se marcan comentarios relevantes en relación a su uso.
- Se asume la existencia de un datatype DtJornal para almacenar los jornales, que tiene las operaciones getFecha y getHoras para obtener sus datos.

Clase Empleado

```
class Empleado {
      private:
            int ci;
            string nombre;
      public:
            Empleado(int, string);
            int getCI();
            string getNombre();
            virtual float darLiquidacion() = 0;
};
```

```
Empleado::Empleado(int unaCI, string unNombre) {
      ci = unaCI;
      nombre = unNombre;
}

int Empleado::getCI() {
      return ci;
}

string Empleado::getNombre() {
      return nombre;
}
```

Clase Comun

```
class Comun : public Empleado {
      private:
            float sueldoMensual;

      public:
            Comun(int, string, float);
            float getSueldoMensual();
            virtual float darLiquidacion();
};

Comun::Comun(int unaCI, string unNombre, float unSueldoMensual) :
             Empleado(unaCI, unNombre) {
      sueldoMensual = unSueldoMensual;
}

float Comun::getSueldoMensual() {
      return sueldoMensual;
}

float Comun::darLiquidacion() {
      return sueldoMensual;
}
```

Clase Jornalero

```
class Jornalero : public Empleado {
      private:
            set <DtJornal> jornales; // Se utiliza un set
                                     // Los jornales no tienen identidad
            float valorHora;
      public:
            Jornalero(int, string, float);
            void agregarJornal(DtJornal);
            int calcularCantHoras();
            float getValorHora();
            virtual float darLiquidacion();
};
```

```
Jornalero::Jornalero(int unaCI, string unNombre, float unValorHora) :
                    Empleado(unaCI, unNombre) {
      valorHora = unValorHora;
      // el set de jornales se inicializa vacío automáticamente
      // no es necesario pedir memoria dinámica explícitamente
}

void Jornalero::agregarJornal(DtJornal dj) {
      jornales.insert(dj); // se agrega un elemento al set
}

int Jornalero::calcularCantHoras() {
      int result = 0;
      set<DtJornal>::iterator it; // iterador sobre el set de jornales
      for (it= jornales.begin(); it!=jornales.end(); ++it)
      // se itera desde la posición inicial hasta la final
            result = result + it->getHoras(); // *it es de tipo DtJornal
            // se accede al elemento actual
      reutrn result;
}

float Jornalero::getValorHora() {
      return valorHora;
}

float Jornalero::darLiquidacion() {
      return calcularCantHoras()*valorHora;
}
```

Programa principal

```
int main() {
      map<int, Empleado *> empleados;
      // La colección de empleados es un map (implementación en STL
      // de un diccionario)
      // En este caso el tipo base es Empleado * y la clave es int
      // No es necesario pedir memoria dinámica explícitamente
      int opcion;
      do {
            cout << "Ingresar opción (0-Salir,
                                    1-Agregar empleado,
                                    2-Eliminar empleado,
                                    3-Agregar jornal a un jornalero,
                                    4-Calcular liquidación)" << endl;
            cin >> opcion;
            switch (opcion) {
                  case 0:
                        break;
                  case 1:
                        int ci;
                        string nom;
                        int tipoEmp;
                        Empleado *emp;
                        cout << "Ingresar cedula" << endl;
                        cin >> ci;
```

```
            cout << "Ingresar nombre" << endl;
            cin >> nom;
            cout << "Ingrear tipo de empleado
                    (1-Comun, 2-Jornalero)" << endl;
            cin >> tipoEmp;
            if (tipoEmp == 1) {
                  float sueldo;
                  cout << "Ingresar sueldo" << endl;
                  cin >> sueldo;
                  emp = new Comun(ci, nom, sueldo);
            } else {
                  float vHora;
                  cout << "Ingresar valor hora" << endl;
                  cin >> vHora;
                  emp = new Jornalero(ci, nom, vHora);
            }
            empleados[ci] = emp;
            // Se agrega al map empleados un nuevo elemento emp
            // de tipo Empleado * identificado por ci
            break;
      case 2:
            int ci;
            Empleado *emp;
            cout << "Ingresar cedula" << endl;
            cin >> ci;
            emp = empleados[ci];
            // Se obtiene el empleado a partir de su
            // identificador
            empleados[ci] = NULL;
            // Se elimina de la colección
            delete emp;
            // Se libera la memoria ocupada por el empleado
            break;
      case 3:
            int ci;
            Jornalero *jorn;
            DtJornal dj;
            cout << "Ingresar cedula" << endl;
            cin >> ci;
            // cargar fecha y cantidad de horas en dj
            jorn = <dynamic_cast>(Jornalero *)empleados[ci];
            // Se obtiene el empleado desde el map
            // y se castea a Jornalero *(se asume que la cédula
            // ingresada corresponde a un jornalero)
            jorn->agregarJornal(dj);
            // Se agrega el jornal al jornalero
            break;
      case 4:
            float liqTotal = 0;
            map<int, Empleado *>::iterator it;
            // Iterador sobre el map de Empleado *
            for (it= empleados.begin(); it!=empleados.end();
                 ++it)
                  // Se recorre todo el map
                  liqTotal = liqTotal +
                              it->second->darLiquidacion();
                  // Para cada elemento de la iteración
```

```
                              // (par <int, Empleado *>) se obtiene el
                              // segundo elemento (el empleado)
                        cout << "Total liquidación: " << liqTotal << endl;
            } // end switch
      } while (opcion != 0); // end while
      return 0;
}
```

Observaciones:

- En general, el tipo `map` es el más adecuado para implementar diccionarios (elementos con clave).
- Para colecciones de elementos sin identificadores existen varias alternativas como ser `set`, `vector` o `list`, dependiendo de los requerimientos específicos, por ejemplo, si se requieren operaciones de conjunto o si interesa mantener la secuencia de los elementos.

Implementación utilizando colecciones genéricas:

Clase Empleado

```
class Empleado : public ICollectible {
                 // Para poder ser almacenado en una colección genérica
      private:
            int ci;
            string nombre;
      public:
            Empleado(int, string);
            int getCI();
            string getNombre();
            virtual float darLiquidacion() = 0;
};


Empleado::Empleado(int unaCI, string unNombre) {
      ci = unaCI;
      nombre = unNombre;
}

int Empleado::getCI() {
      return ci;
}

string Empleado::getNombre() {
      return nombre;
}
```

Clase Comun

```
class Comun : public Empleado {
      private:
            float sueldoMensual;
      public:
            Comun(int, string, float);
            float getSueldoMensual();
            virtual float darLiquidacion();
};

Comun::Comun(int unaCI, string unNombre, float unSueldoMensual) :
             Empleado(unaCI, unNombre) {
      sueldoMensual = unSueldoMensual;
}

float Comun::getSueldoMensual() {
      return sueldoMensual;
}

float Comun::darLiquidacion() {
      return sueldoMensual;
}
```

Clase Jornalero

```
class Jornalero : public Empleado {
      private:
            ICollection *jornales; // Se utiliza una colección
                                   // Debe ser un puntero a la interfaz
                                   // que luego se instancia con una
                                   // con una realización particular
            float valorHora;
      public:
            Jornalero(int, string, float);
            ~Jornalero(); // Debe tener un destructor para destruir
                          // la colección de jornales (y los jornales
                          // propiamente dichos)
            void agregarJornal(DtJornal *);
            // DtJornal debe realizar la interfaz ICollectible
            // y el parámetro debe ser un puntero o referencia
            int calcularCantHoras();
            float getValorHora();
            virtual float darLiquidacion();
};

Jornalero::Jornalero(int unaCI, string unNombre, float unValorHora) :
                     Empleado(unaCI, unNombre) {
      valorHora = unValorHora;
      jornales = new List;
      // Supongamos que tenemos una clase List que realiza la
      // interfaz ICollection
}
```

```
Jornalero::~Jornalero() {
      IIterator *it = jornales->getIterator();
      while (it->hasCurrent())
      {
            delete it->getCurrent();
            it->next();
      }
      delete it; // Se libera la memoria dinámica ocupada por el iterador
      delete jornales; // Se libera la memoria dinámica de la colección
}

// La implementación anterior asume que:
// - El iterador se mantiene consistente aunque se eliminen objetos cuya
//   gestión de memoria no es realizada por la colección.
// - El destructor de la clase List (el tipo de la colección jornales)
//   libera toda la memoria obtenida dinámicamente por operaciones de la
//   propia clase.

void Jornalero::agregarJornal(DtJornal *dj) {
      jornales->add(dj); // se agrega un elemento a la colección
}

int Jornalero::calcularCantHoras() {
      int result = 0;
      DtJornal *dj;
      IIterator *it = jornales->getIterator();
      while (it->hasCurrent()) {
            dj = <dynamic_cast>(DtJornal *)it->getCurrent();
            // Se debe castear el resultado al tipo concreto
            result = result + dj->getHoras();
            it->next();
      }
      reutrn result;
}

float Jornalero::getValorHora() {
      return valorHora;
}

float Jornalero::darLiquidacion() {
      return calcularCantHoras()*valorHora;
}
```

Programa principal

```
int main() {
      IDictionary *empleados = new Hash;
      // La colección de empleados es una realización de la interfaz
      // IDictionary, en este caso se asume la disponibilidad de una clase
      // denominada Hash
      int opcion;
      do {
            cout << "Ingresar opción (0-Salir,
                                    1-Agregar empleado,
                                    2-Eliminar empleado,
```

```
                              3-Agregar jornal a un jornalero,
                              4-Calcular liquidación)" << endl;
    cin >> opcion;
    switch (opcion) {
          case 0:
                break;
          case 1:
                int ci;
                string nom;
                int tipoEmp;
                Empleado *emp;
                cout << "Ingresar cedula" << endl;
                cin >> ci;
                cout << "Ingresar nombre" << endl;
                cin >> nom;
                cout << "Ingrear tipo de empleado
                        (1-Comun, 2-Jornalero)" << endl;
                cin >> tipoEmp;
                if (tipoEmp == 1) {
                      float sueldo;
                      cout << "Ingresar sueldo" << endl;
                      cin >> sueldo;
                      emp = new Comun(ci, nom, sueldo);
                } else {
                      float vHora;
                      cout << "Ingresar valor hora" << endl;
                      cin >> vHora;
                      emp = new Jornalero(ci, nom, vHora);
                }
                empleados->add(new KeyInt(ci), emp);
                // Se agrega a la colección de empleados un nuevo
                // empleado y su clave, en este caso perteneciente
                // a la clase KeyInt, que realiza la interface IKey
                // (claves que son números enteros)
                break;
          case 2:
                int ci;
                Empleado *emp;
                cout << "Ingresar cedula" << endl;
                cin >> ci;
                KeyInt ki(ci); // Clave con la cédula a buscar
                emp = <dynamic_cast>(Empleado *)empleados
                                                    ->find(&ki);
                // Se obtiene el empleado a partir de su
                // identificador, es necesario castear
                empleados->remove(&ki);
                // Se elimina de la colección
                delete emp;
                // Se libera la memoria ocupada por el empleado
                break;
          case 3:
                int ci;
                Jornalero *jorn;
                DtJornal dj;
                cout << "Ingresar cedula" << endl;
                cin >> ci;
                // cargar fecha y cantidad de horas en dj
```

```
                        KeyInt ki(ci); // Clave con la cédula a buscar
                        jorn = <dynamic_cast>(Jornalero *)empleados
                                                        ->find(&ki);
                        // Se castea a Jornalero * (se asume que la cédula
                        // ingresada corresponde a un jornalero)
                        jorn->agregarJornal(new DtJornal(dj));
                        // Se agrega una copia del jornal porque la
                        // colección de jornales está implementada
                        // utilizando memoria dinámica
                        break;
                  case 4:
                        float liqTotal = 0;
                        Empleado *emp;
                        IIterator *it = empleados->getElemIterator();
                        // Se itera sobre los elementos del diccionario
                        while (it->hasCurrent) {
                              emp = <dynamic_cast>(Empleado *)
                                                      it->getCurrent();
                              liqTotal = liqTotal +
                                          emp->darLiquidacion();
                              it->next();
                        }
                        cout << "Total liquidación: " << liqTotal << endl;
            } // end switch
      } while (opcion != 0); // end while
      return 0;
}
```

Observación:

En general, si una clase A mantiene como miembro a una colección de objetos de clase B, implementada como una colección genérica (por ejemplo, realizaciones de ICollection o IDictionary), el destructor de A debe liberar explícitamente la memoria de la colección B teniendo en cuenta que (los siguientes dos puntos valen tanto para colecciones genéricas como para colecciones paramétricas de STL):

- Si las instancias de B dependen exclusivamente de A, entonces además debe liberarse la memoria dinámica ocupada por cada elemento de clase B.
- Si las instancias de B NO dependen exclusivamente de A, entonces NO debe liberarse la memoria dinámica ocupada por cada elemento de clase B.