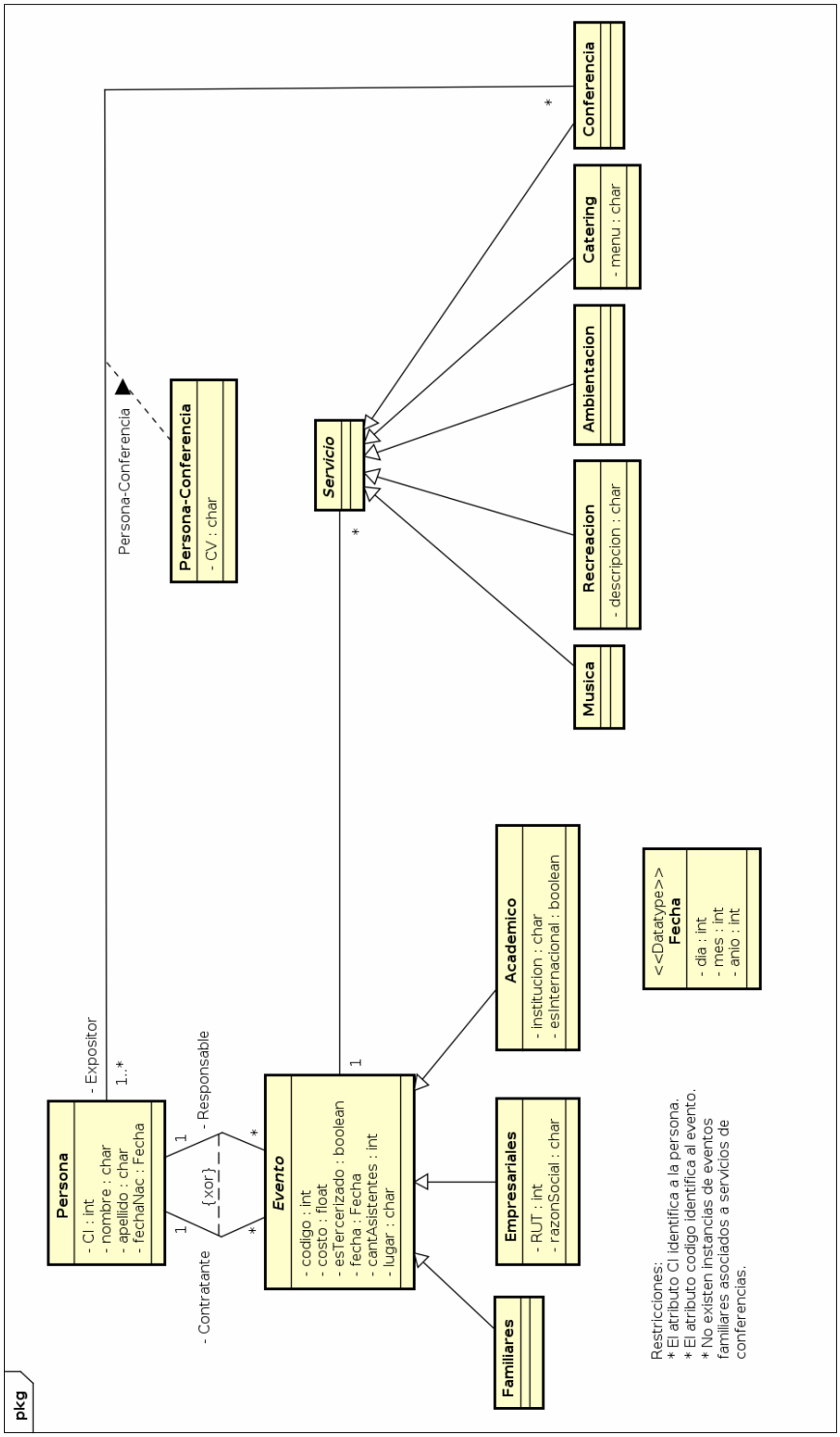
Programación 4

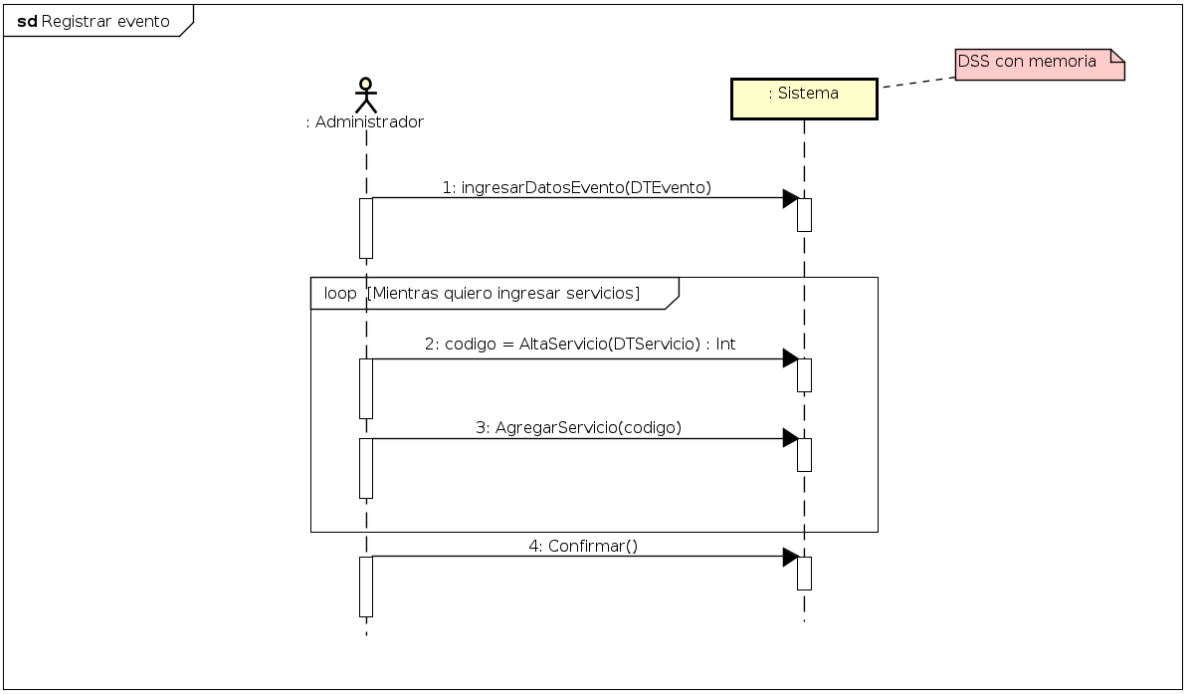
SOLUCION DICIEMBRE 2017

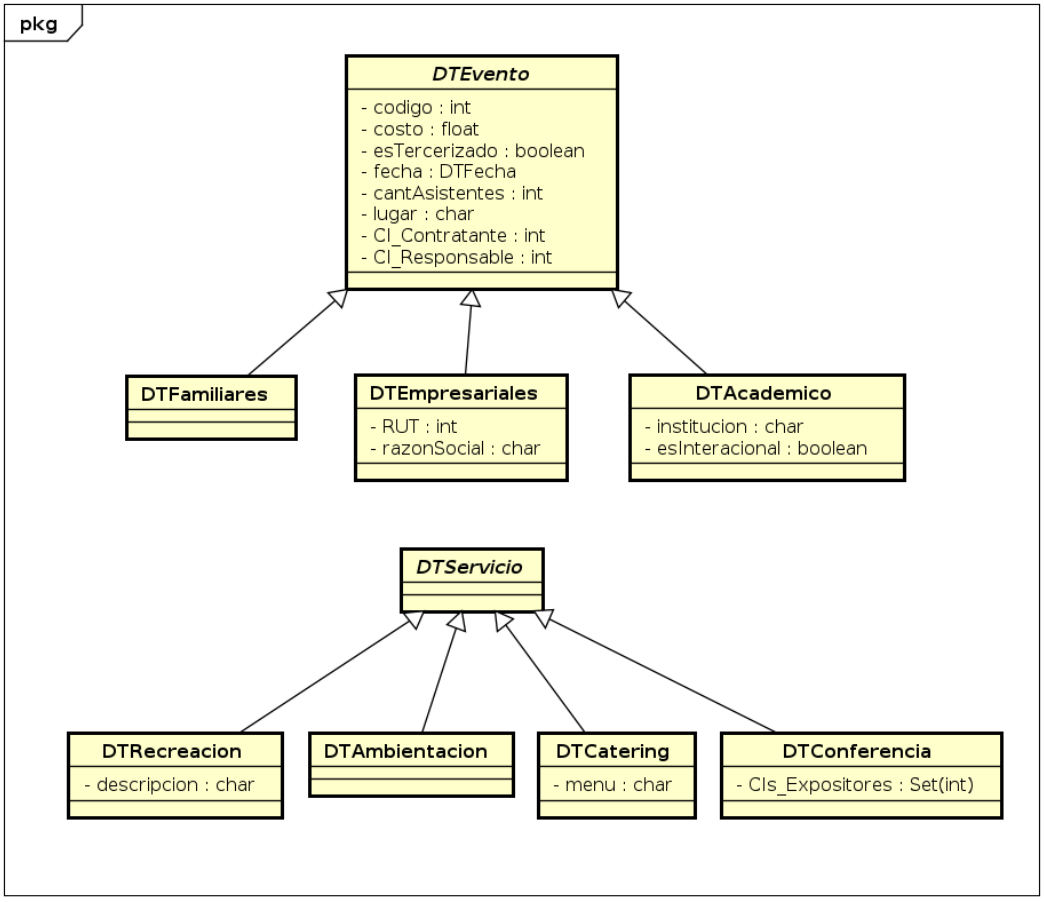
Problema 1 (35 puntos)

i)



ii)


sd Registrar evento
: Administrador
: Sistema
DSS con memoria
1: ingresarDatosEvento(DTEvento)
loop [Mientras quiero ingresar servicios]
2: codigo = AltaServicio(DTServicio) : Int
3: AgregarServicio(codigo)
4: Confirmar()


pkg
DTEvento
- codigo : int
- costo : float
- esTercerizado : boolean
- fecha : DTFecha
- cantAsistentes : int
- lugar : char
- CI_Contratante : int
- CI_Responsable : int
DTFamiliares
DTEmpresariales
- RUT : int
- razonSocial : char
DTAcademico
- institucion : char
- esInteracional : boolean
DTServicio
DTRecreacion
- descripcion : char
DTAmbientacion
DTCatering
- menu : char
DTConferencia
- CIs_Expositores : Set(int)

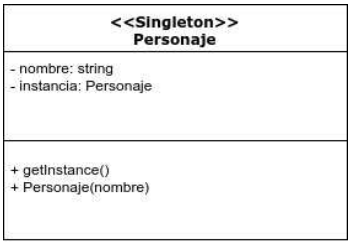
Problema 2 (35 puntos)

Parte I:

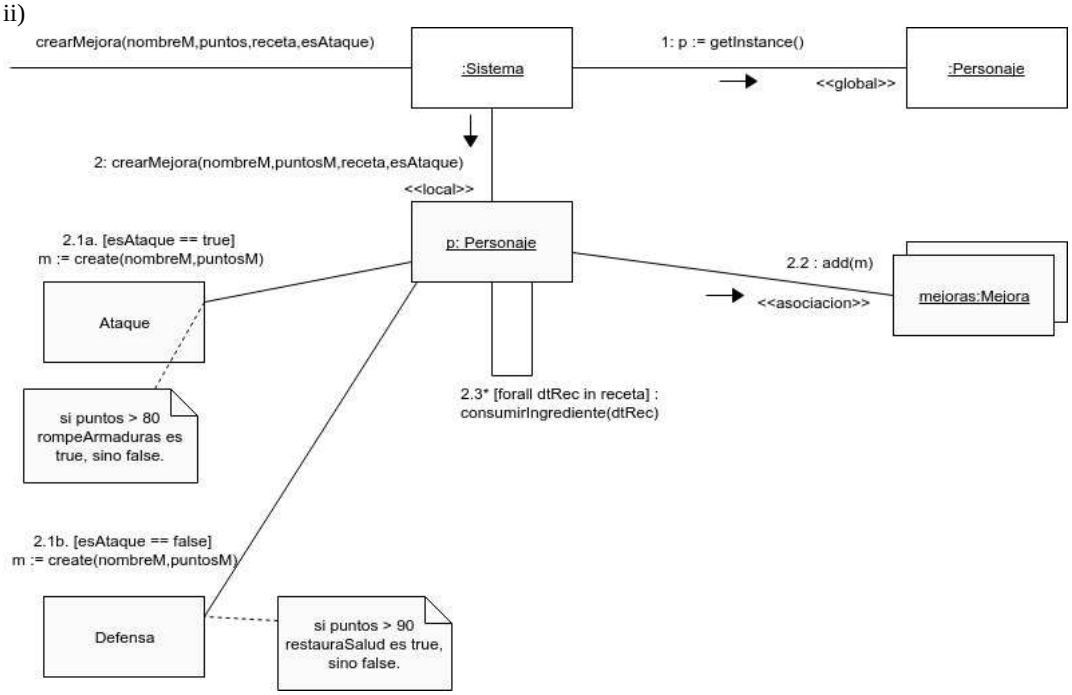
i)

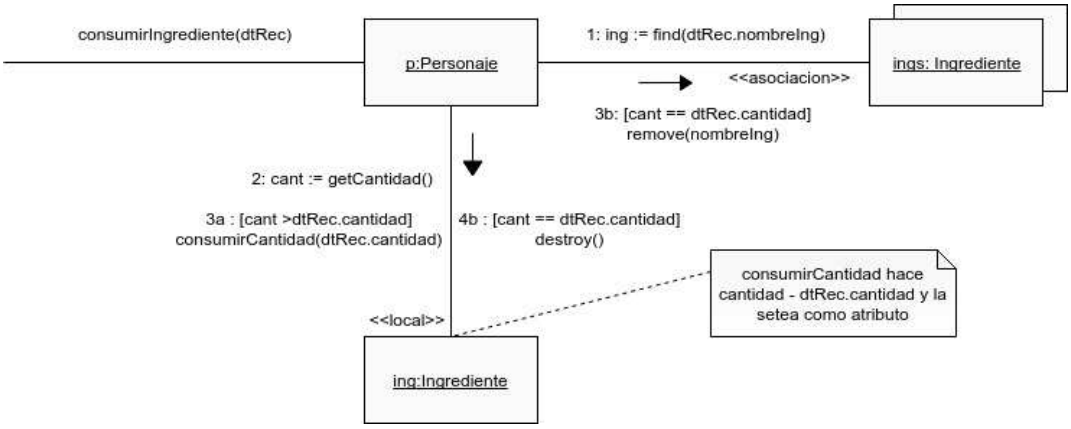
Se utiliza el patrón Singleton.

Problema tipo: Asegurar que una clase tenga una sola instancia y proveer un acceso global a ella.



ii)



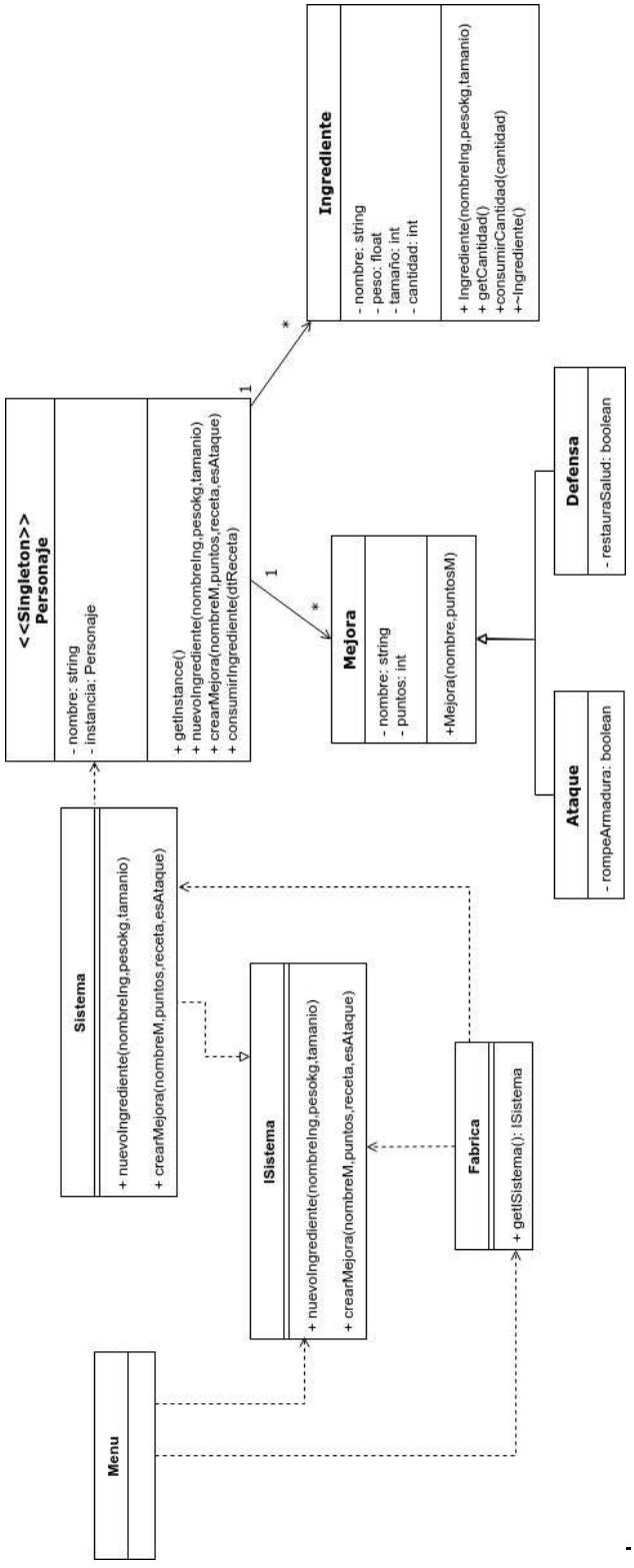


Parte II:

i)

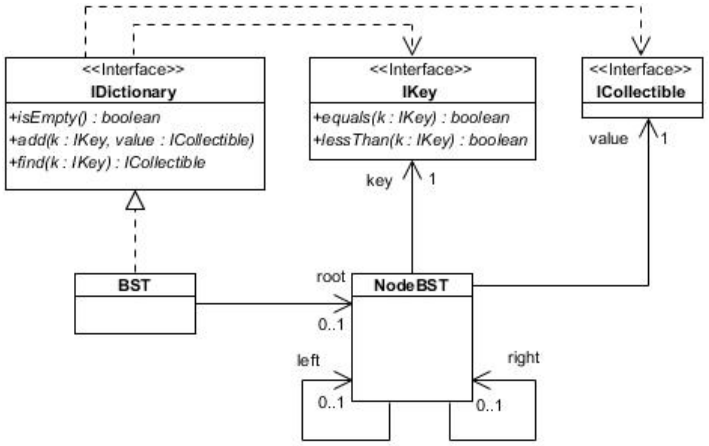
El mecanismo Factory: permite que ciertas clases obtengan información de otras sin acoplarse directamente. Es decir Menu conoce las operaciones de las clases del Sistema pero no conoce ninguna de las clases implementadas en él.

ii)


<<Singleton>>
Personaje
- nombre: string
- instancia: Personaje
+ getInstance()
+ nuevoIngrediente(nombreIng,pesokg,tamanio)
+ crearMejora(nombreM,puntos,receta,esAtaque)
+ consumirIngrediente(dtReceta)
Ingrediente
- nombre: string
- peso: float
- tamaño: int
- cantidad: int
+ Ingrediente(nombreIng,pesokg,tamanio)
+ getCantidad()
+consumirCantidad(cantidad)
+~Ingrediente()
Mejora
- nombre: string
- puntos: int
+Mejora(nombre,puntosM)
Defensa
- restauraSalud: boolean
Ataque
- rompeArmadura: boolean
Sistema
+ nuevoIngrediente(nombreIng,pesokg,tamanio)
+ crearMejora(nombreM,puntos,receta,esAtaque)
ISistema
+ nuevoIngrediente(nombreIng,pesokg,tamanio)
+ crearMejora(nombreM,puntos,receta,esAtaque)
Fabrica
+ getISistema(): ISistema
Menu
1
*
1
*

Problema 3 (30 puntos)

i)



ii)

```
// Interfaces
class IDictionary {
public:
   virtual bool isEmpty() = 0;
   virtual void add(IKey* k, ICollectible* value) = 0;
   virtual ICollectible* find(IKey* k) = 0;
   virtual ~IDictionary() {}
}

class IKey {
public:
   bool equals(IKey* k) = 0;
   bool lessThan(IKey* k) = 0;
   virtual ~IDictionary() {}
}

class ICollectible {
public:
   virtual ~IDictionary() {}
}

// Header BST
class BST : public IDictionary {
private:
   NodeBST* root;
public:
   BST();
   bool isEmpty();
   void add(IKey* k, ICollectible* value);
   ICollectible* find(Ikey* k);
   ~BST();
}

// Header NodeBST
class NodeBST {
private:
   IKey* key;
```

```
  ICollectible* value;
  NodeBST* left;
  NodeBST* right;
public:
  NodeBST(IKey* k, ICollectible* v, NodeBST* l = NULL, NodeBST* r = NULL);
  ~NodeBST();
  void add(IKey* k, ICollectible* v);
  ICollectible* find(Ikey* k);
}

// CPP BST
BST::BST() {
  this->root = NULL;
}

bool BST::isEmpty() {
  if (this->root == NULL) {
       return true;
  } else {
       return false;
  }
}

void BST::add(IKey* k, ICollectible* value) {
  if (k == NULL) {
       throw std::invalid_argument("Clave incorrecta");
  }
  if (this->root == NULL) { // BST vacio
       this->root = new NodeBST(k, value);
  } else {
       this->root->add(k, value);
  }
}

ICollectible* BST::find(Ikey* k) {
  if (k == NULL) {
       return NULL;
  }
  if (this->root = NULL) { // arbol vacio
       return NULL;
  } else {
       return this->root->find(k);
  }
}

BST::~BST() {
  if (this->root != NULL) {
       delete this->root;
  }
}

// CPP NodeBST
NodeBST::NodeBST(IKey* k, ICollectible* v, NodeBST* l = NULL, NodeBST* r = NULL) {
  this->key = k;
  this->value = v;
  this->left = l;
  this->right = r;
}
```

```
void NodeBST::add(IKey* k, ICollectible* v) {
  if (k->equals(this->key)) {
       throw std::invalid_argument("Clave existente");
  }
  if (k->lessThan(this->key)) { // subarbol izquierdo
       if (this->left == NULL) { // subarbol izq. vacio
              this->left = new NodeBST(k, v);
       } else { // agrego en subarbol izq.
              this->left->add(k, v);
       }
  } else { // subarbol derecho
       if (this->right == NULL) { // subarbol derecho vacio
              this->right = new NodeBST(k, v);
       } else { // agrego en subarbol derecho
              this->right->add(k, v);
       }
  }
}

ICollectible* NodeBST::find(Ikey* k) {
  if (k->equals(this->key)) {
       return this->value;
  }
  if (k->lessThan(this->key)) { // subarbol izquierdo
       if (this->left == NULL) { // subarbol izq. vacio
              return NULL;
       } else { // agrego en subarbol izq.
              this->left->find(k);
       }
  } else { // subarbol derecho
       if (this->right == NULL) { // subarbol derecho vacio
              return NULL;
       } else { // agrego en subarbol derecho
              this->right->find(k);
       }
  }
}

NodeBST::~NodeBST() {
  delete key;
  if (this->left != NULL) {
       delete this->left;
  }
  if (this->right != NULL) {
       delete this->rigth;
  }
}
```