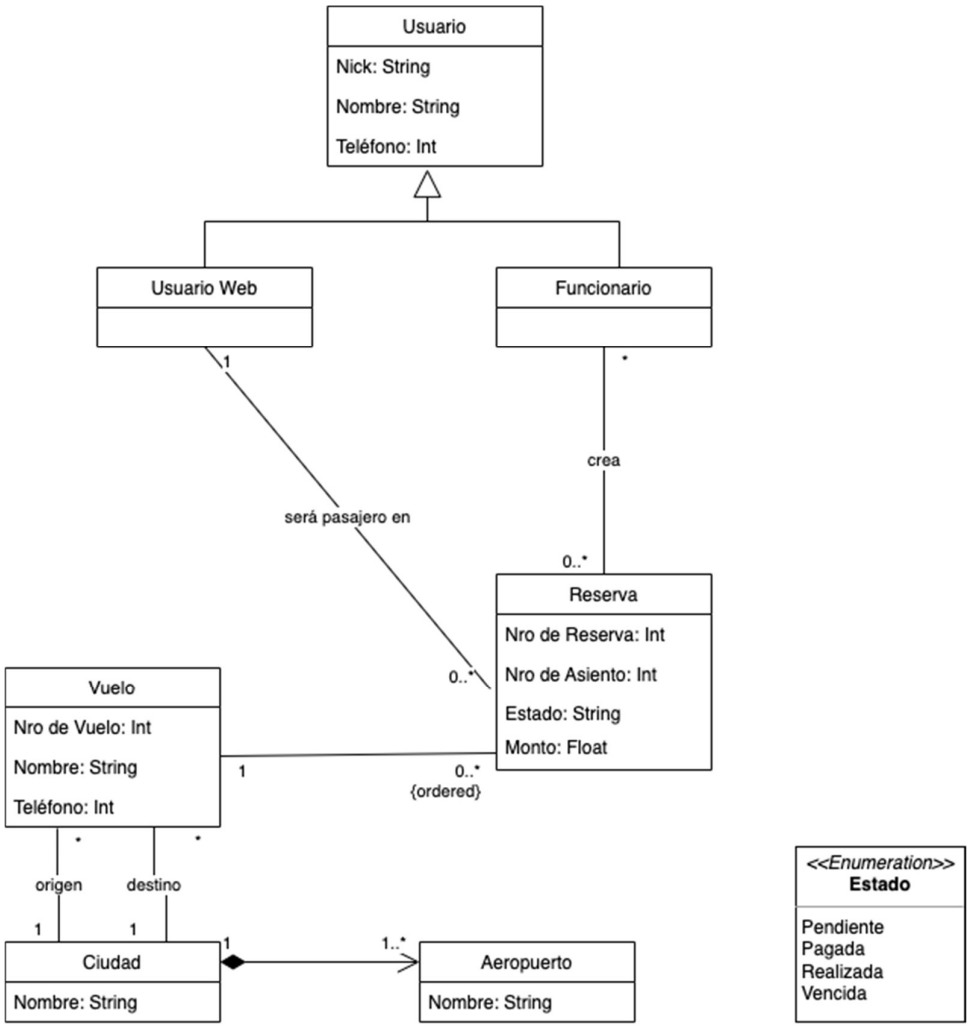
Programación 4

SOLUCIÓN EXAMEN DICIEMBRE 2021

Problema 1

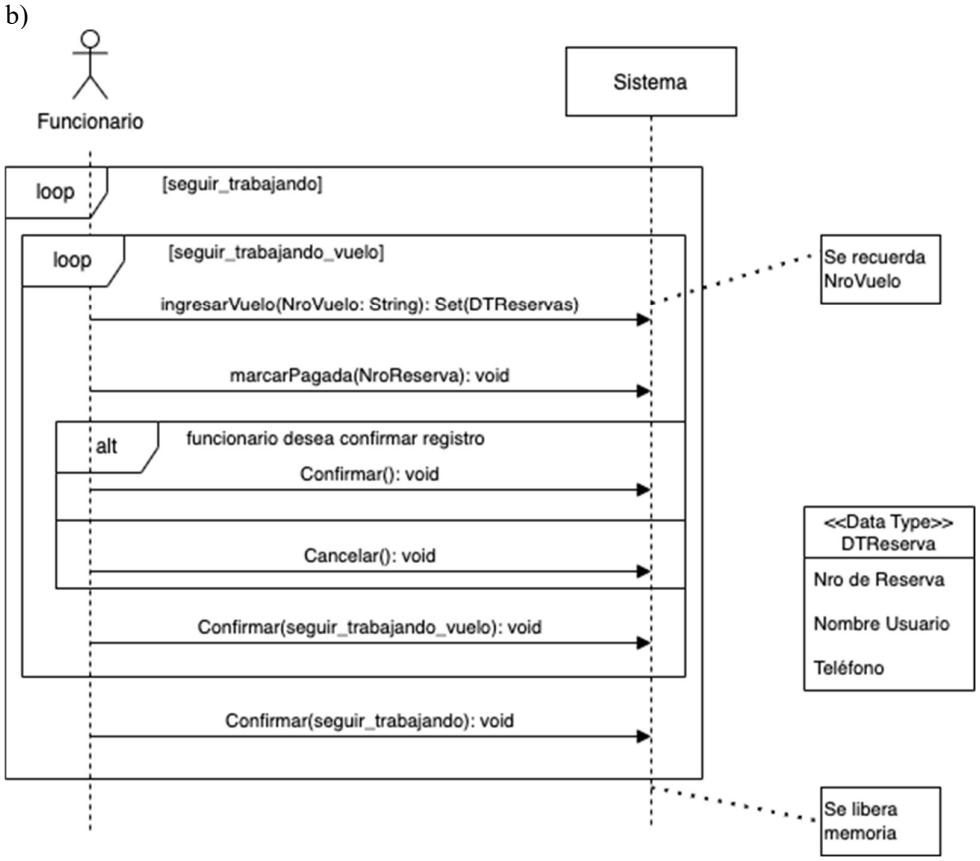
a)



Restricciones:

- Nick de Usuario debe ser único.
- Nro de Reserva debe ser único.
- Nro de Vuelo debe ser único.
- Nombre de ciudad debe ser único.
- Para cualquier vuelo, Ciudad Origen es diferente de Ciudad destino.

b)

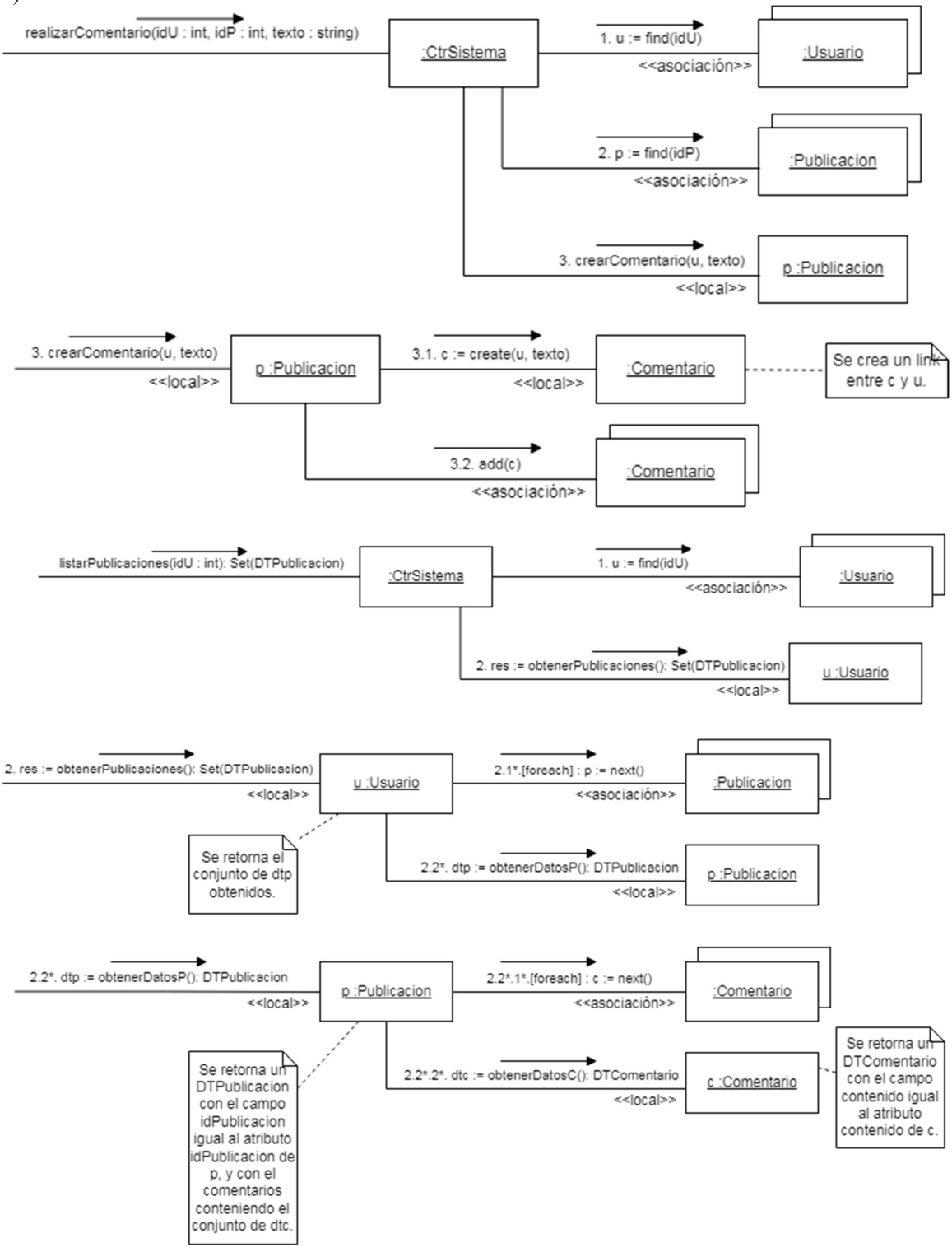

Funcionario
Sistema
loop
[seguir_trabajando]
loop
[seguir_trabajando_vuelo]
Se recuerda NroVuelo
ingresarVuelo(NroVuelo: String): Set(DTReservas)
marcarPagada(NroReserva): void
alt
funcionario desea confirmar registro
Confirmar(): void
Cancelar(): void
<<Data Type>>
DTReserva
Nro de Reserva
Nombre Usuario
Teléfono
Confirmar(seguir_trabajando_vuelo): void
Confirmar(seguir_trabajando): void
Se libera memoria

Programación 4

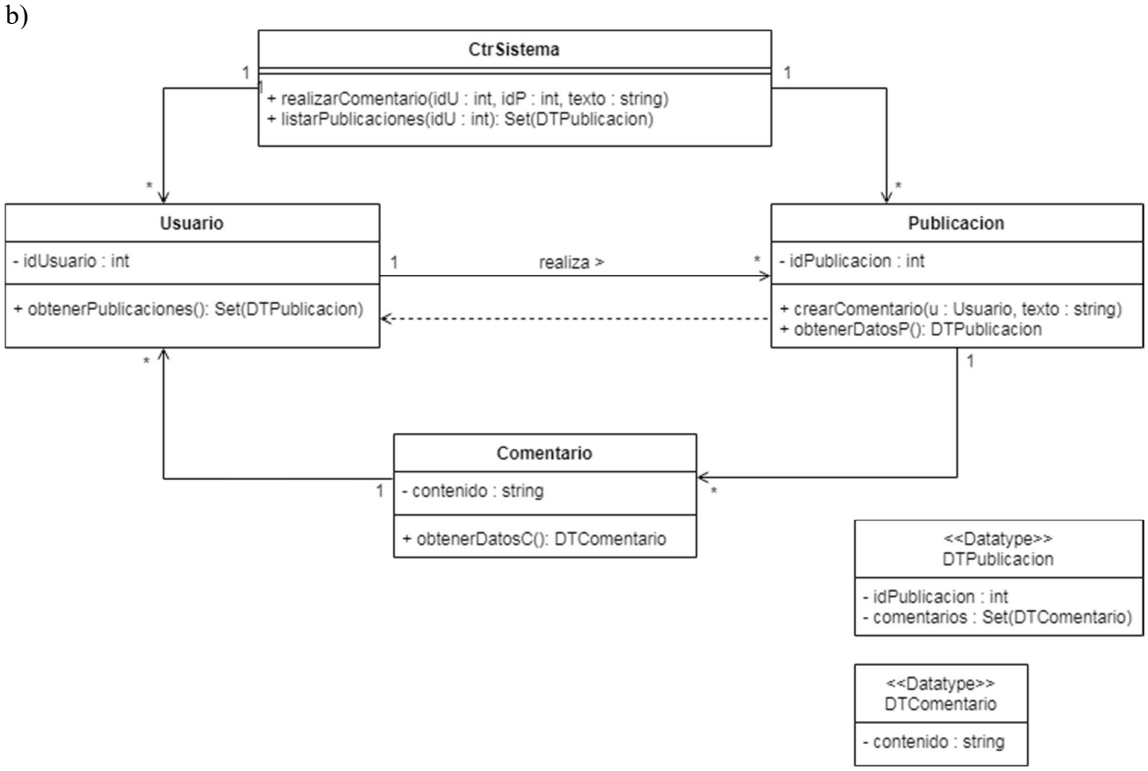
SOLUCIÓN EXAMEN DICIEMBRE 2021

Problema 2

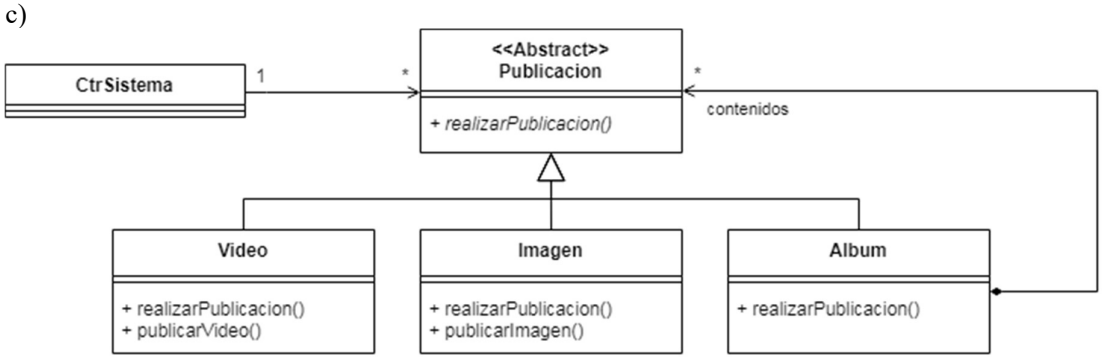
a)



b)



c)



Se aplica el patrón Composite, con los siguientes roles: Cliente es CtrSistema, Componente es Publicacion, Compuesto es Album, Hoja es Video e Imagen.

```
Album::realizarPublicacion() {
      forall p in contenidos
            p.realizarPublicacion();
}

Video::realizarPublicacion() {
      …
      publicarVideo();
      …
}

Imagen::realizarPublicacion() {
      …
      publicarImagen();
      …
}
```

Programación 4

EXAMEN DICIEMBRE 2021

Problema 3

a)

Nota: En las respuestas para esta parte se valoraba la comprensión del estudiante sobre el problema tipo y el patrón, y no únicamente las respuestas si/no. En este sentido era importante expresar un razonamiento a partir de la realidad y el problema resuelto por el patrón que llegara a la respuesta y no únicamente enunciar dicho problema y los roles. Las respuestas a continuación eran las esperadas, pero dependiendo del contexto y la justificación otras respuestas podían ser valoradas.

`EstadoM`: En este caso se justifica la aplicación del patrón *state* puesto que la operación `getMontoDiario()` de `Mesa` requería un comportamiento diferente según el estado. Dicho comportamiento es delegado en la operación `facturar()` de la clase `EstadoM` que dependiendo del estado concreto se comportaba distinto.

`EstadoP`: En este caso no se justifica la aplicación del patrón *state* puesto que no hay ningún comportamiento delegado al estado. En definitiva, la aplicación del patrón *state* planteada cumple la única función de modelar dónde se encuentra el pedido, algo que podría ser hecho con un enumerado o similar.

b)

Nota: En las respuestas de esta parte se espera no solo que la implementación brindada sea correcta, sino que siga el diagrama dado. La única operación de la clase `Mozo` presente es `cerrarYCalcularTot()` mientras que `Mesa` cuenta con dos operaciones en el diagrama: `getMontoDiario()` y `RetiroCliente()`.

```
DtFactura Mozo::cerrarYCalcularTot(){
      set<Mesa *>::iterator it;
      int acum = 0;    //Inicializaciones
      for(it = mesas.begin(); it!=mesas.end();++it){    //2.1
            acum += (*it)→getMontoDiario();             //2.2
            (*it)→ RetiroCliente();                     //2.3
      }
      return DtFactura(this→cedula, acum);
}

int Mesa::getMontoDiario(){
      int x;
      x = this→estado→facturar();      //2.2.1
      this→montoDiario += x;           //Nota
      return this→montoDiario;         //Nota
}

void Mesa::RetiroCliente(){
      EstadoM *temp = this→ estado;    //Manejo de memoria
      this→estado = this→estado→ siguienteEstado();    //2.3.1
      delete temp;                     //Manejo de memoria
}
```

c)

```
//EstadoM.h (El .cpp de EstadoM no es necesario)
class EstadoM{
      public:
            virtual int facturar() = 0;
            virtual void siguienteEstado() =0;
            virtual ~EstadoM();
}

//ConClientes.h
class ConClientes{
      public:
            int facturar();
            void siguienteEstado();
            ConClientes(int monto);
      private:
            int monto;
}

//ConClientes.cpp
int ConClientes::facturar(){
      return this→monto
}

void ConClientes::siguienteEstado(){
      Estado* res = new Vacia();
      return res;
}

//Controlador.h
class Controlador{
      private:
            map<String, Empleado*> empleados; //No es válido tener
            // dos colecciones, una de mozos y otra de deliverys
            static Controlador* instancia;
            Controlador();
      public:
            static Controlador* getInstancia();
            DtFactura calcularTotalMozo(String ciMozo);
            //En el DCD la CI figuraba  erróneamente como int
            bool agregarMesa(int IdMe,String idMo);
}

//Controlador.cpp

bool Controlador::agregarMesa(int idMe, String idMo){
      Empleado* e = empleados[idMo];
      Mozo* m =<dynamic_cast>(Mozo*)e;
      if(m == NULL)         //No hay un mozo con ci buscada
            return false;
      else
            return m→nuevaMesa(idMe);
}
```

```
bool Mozo::nuevaMesa(int idM){
      Mesa* m = mesas[i];
      if (m==NULL)
            m = new Mesa(idM);
      else
            return false;
      mesas[i] = m;
      return true;
}
```