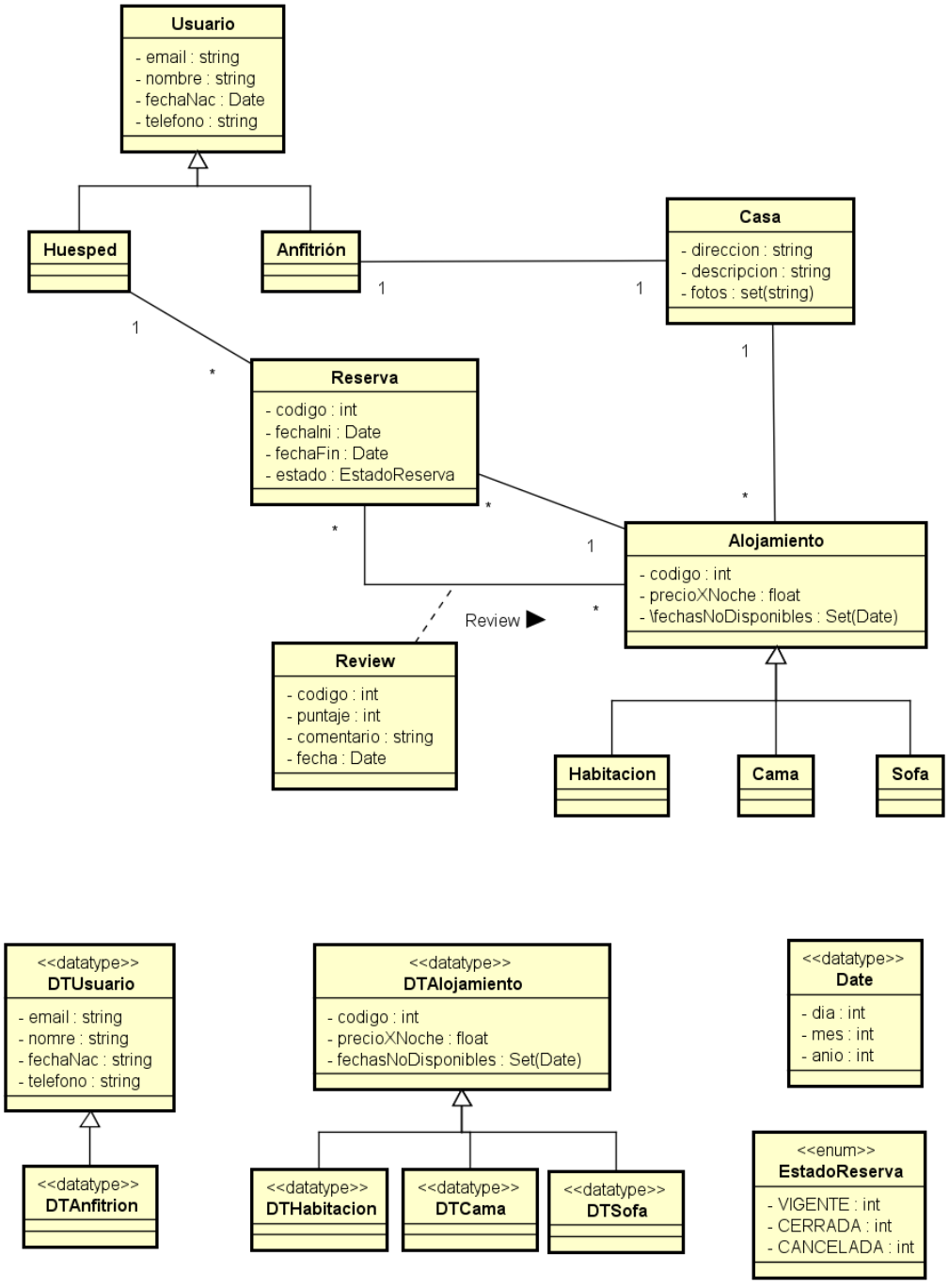
Programación 4

SOLUCIÓN FEBRERO 2018

Problema 1 (35 puntos)

a.



Restricciones:

UNICIDAD DE ATRIBUTOS

- No existen dos instancias de Usuario con igual valor de email

- No existen dos instancias de Alojamiento con igual valor de codigo

- No existen dos instancias de Reserva con igual valor de codigo

- No existen dos instancias de Review con igual valor de codigo

DOMINIO DE ATRIBUTOS

- fechaNac, fechaIni, fechaFin y fecha en Review, toman valores de Date mayores al 01/01/1900

- precioXNoche toma valores positivos

- puntaje toma valores en el rango [1..5]

ATRIBUTOS CALCULADOS

- fechasNoDisponibles se calcula como la suma de todas las fechas para las cuales el Alojamiento se encuentra "reservado". Para esto se suman todas las fechas

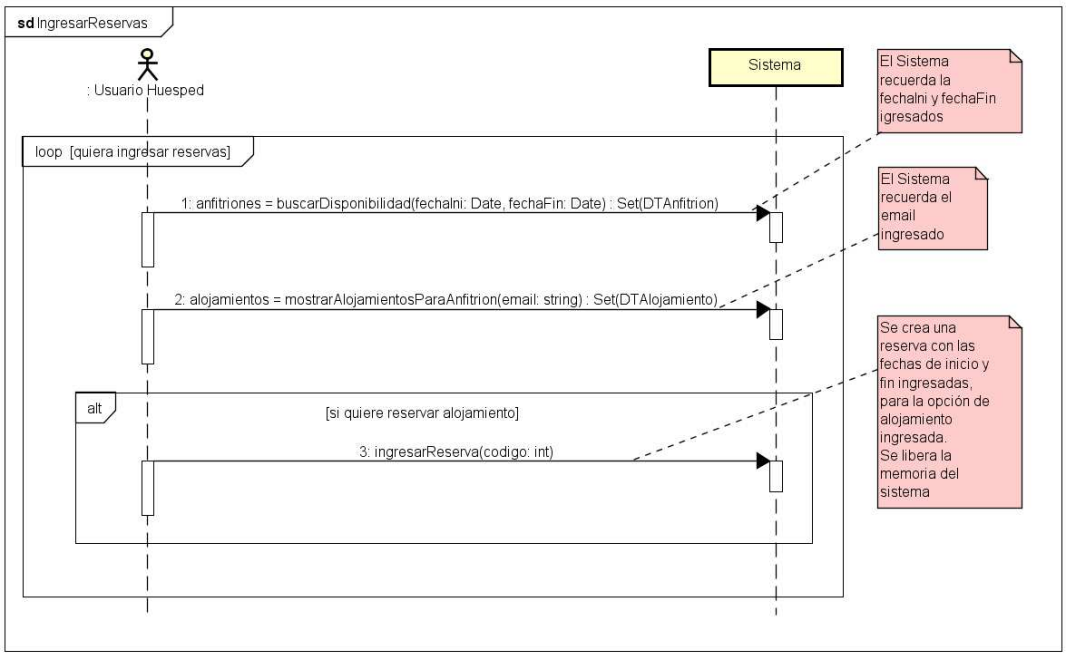
comprendidas entre los valores de fechaIni y fechaFin inclusive, de cada Reserva asociada al alojamiento.

REGLAS DE NEGOCIO

- Para una instancia de Reserva y Alojamiento, existe una instancia de Review que las une solamente si el estado de la Reserva es "CERRADA".

- El valor del atributo fecha en una Review es igual o posterior al valor del atributo fechaFin en una Reserva.

b.



Problema 2 (35 puntos)

Parte A:

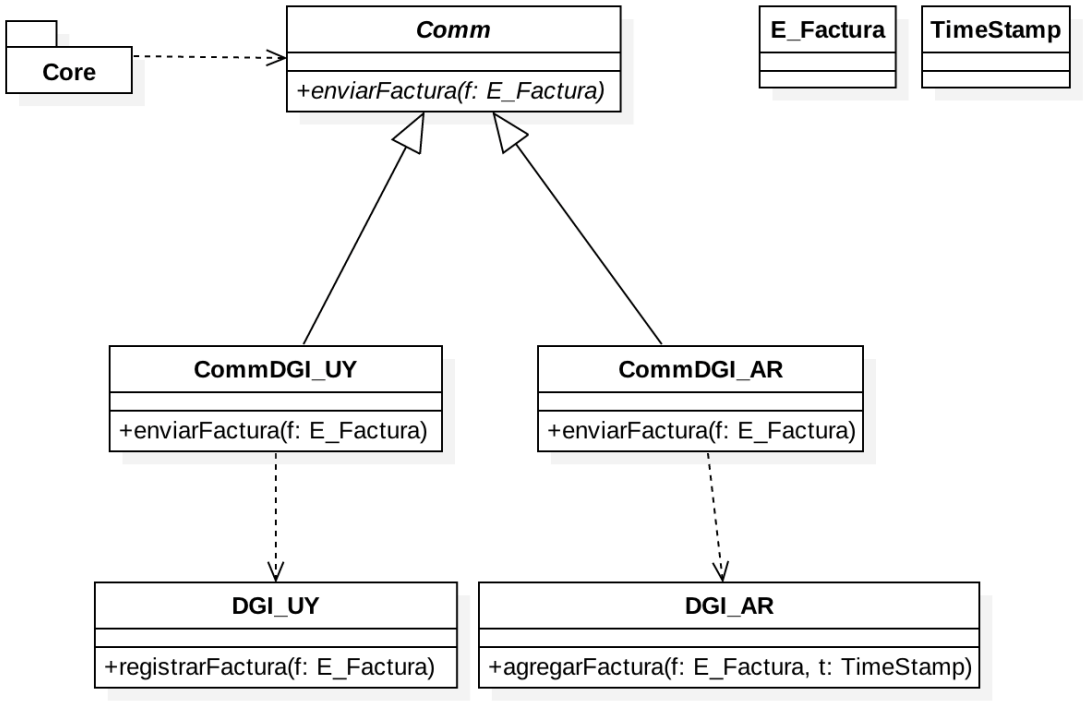
El GRASP Bajo Acoplamiento es un criterio de asignación de responsabilidades que se utiliza en la etapa de diseño de bajo nivel que sugiere que el acoplamiento general se mantenga bajo. El acoplamiento es una medida de que tanto una clase está relacionada, tiene conocimiento o depende de otras clases. Las desventajas de una clase altamente acoplada es que depende/conoce a muchas otras clases, por lo que se dificulta entenderla en forma individual, se dificulta reutilizarla (pues se debe "llevar" al resto de las clases también) y cambios en las clases acopladas pueden implicar cambios en la clase original.

Parte B:

a.

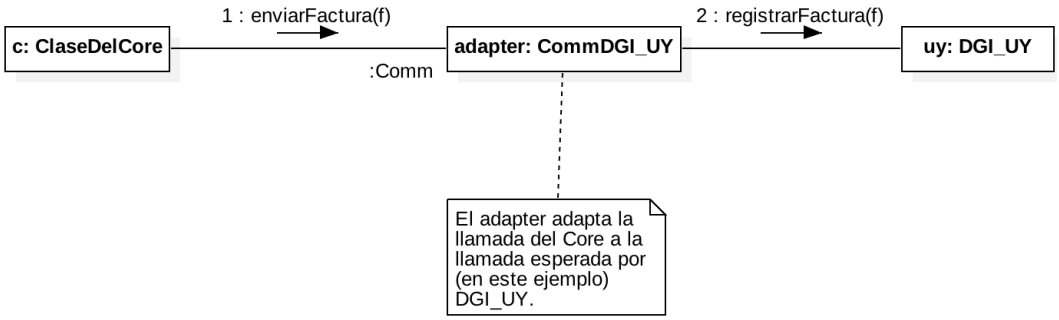
Se puede utilizar el patrón Adapter ya que su objetivo es adaptar interfaces/operaciones, convirtiendolas a la forma en que el código cliente (en este caso el Core) espera invocarlas. En este caso, de no adaptar la solución a los diferentes entes recaudadores, el Core no sería capaz de comunicarse con los diferentes entes, pues las operaciones de éstos son diferentes. Utilizando Adapter, todas las clases del Core continuarán utilizando la clase (o interfaz) Comm. El adapter sería CommDGI_UY (y CommDGI_AR) y el adaptee DGI_UY (y DGI_AR) mientras que el cliente es el Core. De esta forma se modifica el diseño de Comm pero no del Core, como se solicita en la letra.

b.



c.

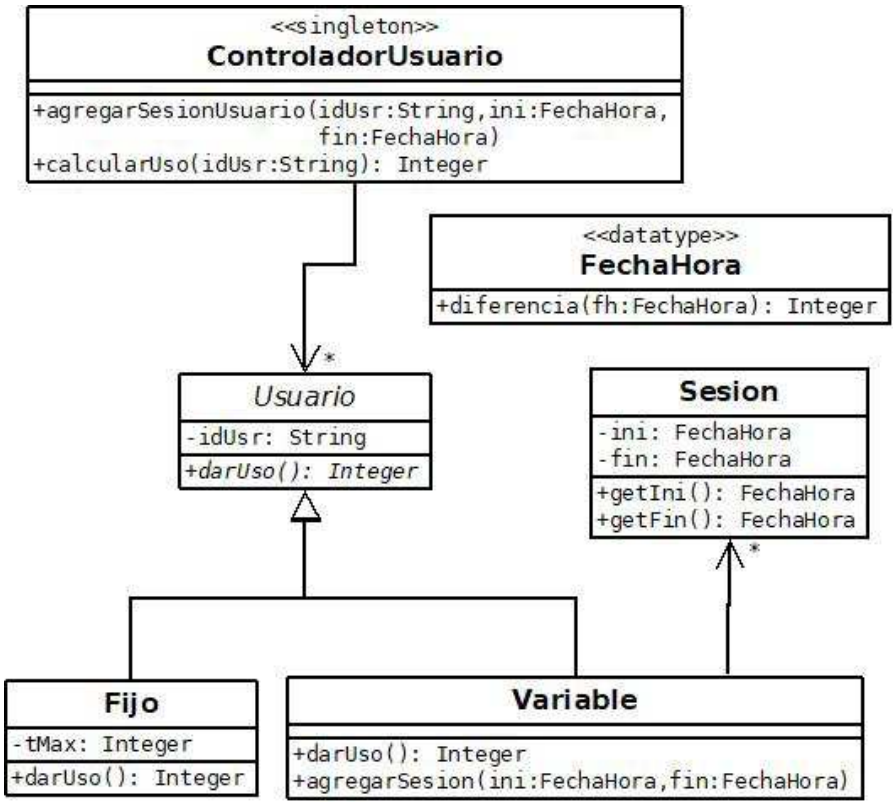
Ejemplo de comunicación entre una clase del Core y la clase DGI_UY:



Para el caso de DGI_AR el diagrama sería similar pero en vez de adaptar la llamada hacia "registrarFactura(f:E_Factura)" se adapta hacia "agregarFactura(f:E_Factura, t:TimeStamp)", es decir, se agrega el parámetro TimeStamp y se utiliza el nombre de operación "agregarFactura" en vez de "registrarFactura".

<u>Problema 3</u> (30 puntos)

a. Se agrega la operación polimórfica `darUso` en la jerarquía de clases de Usuario y `agregarSesion` en Variable.



b.

```
class ControladorUsuario {
private:
  static ControladorUsuario *instance;
  ControladorUsuario();
  map<string, Usuario *> usuarios;
public:
  int calcularUso(string);
  void agregarSesionUsuario(string, FechaHora, FechaHora);
  static ControladorUsuario *getInstance();
}

class Usuario {
private:
  string idUsr;
public:
  virtual int darUso()=0;
}

class Fijo : public Usuario {
private:
  int tMax;
public:
  int darUso();
}

class Variable : public Usuario {
private:
  set<Sesion *> sesiones;
public:
  int darUso();
  void agregarSesion(FechaHora, FechaHora);
}

class Sesion {
private:
  FechaHora ini, fin;
public:
  Sesion(FechaHora, FechaHora);
```

```
    FechaHora getIni();
    FechaHora getFin();
  }

  class FechaHora {
  private:
    int anio, mes, dia, hora, minutos;
  public:
    int diferencia(FechaHora);
  }
```

c.

```
Sesion::Sesion(FechaHora i, FechaHora f) {
  ini = i;
  fin = f;
}

FechaHora Sesion::getIni() {
  return ini;
}

FechaHora Sesion::getFin() {
  return fin;
}

void Variable::agregarSesion(FechaHora i, FechaHora f) {
  sesiones.insert(new Sesion(i,f));
}

void  ControladorUsuario::agregarSesionUsuario(string  idUsr,  FechaHora
i, FechaHora f) {
  Usuario *u = usuarios[idUsr];
  ((Variable *)u)->agregarSesion(i, f);
}

int Fijo::darUso() {
  return tMax;
}

int Variable::darUso() {
  set<Sesion *>::iterator it;
```

```
  int sum = 0;
  for (it=sesiones.begin; it!=sesiones.end(); ++it) {
    FechaHora i = it->getIni();
    FechaHora f = it->getFin();
    sum = sum + fin.diferencia(ini);
  }
  return sum;
}

int ControladorUsuario::calcularUso(string idUsr) {
  Usuario *u = usuarios[idUsr];
  return u->darUso();
}
```