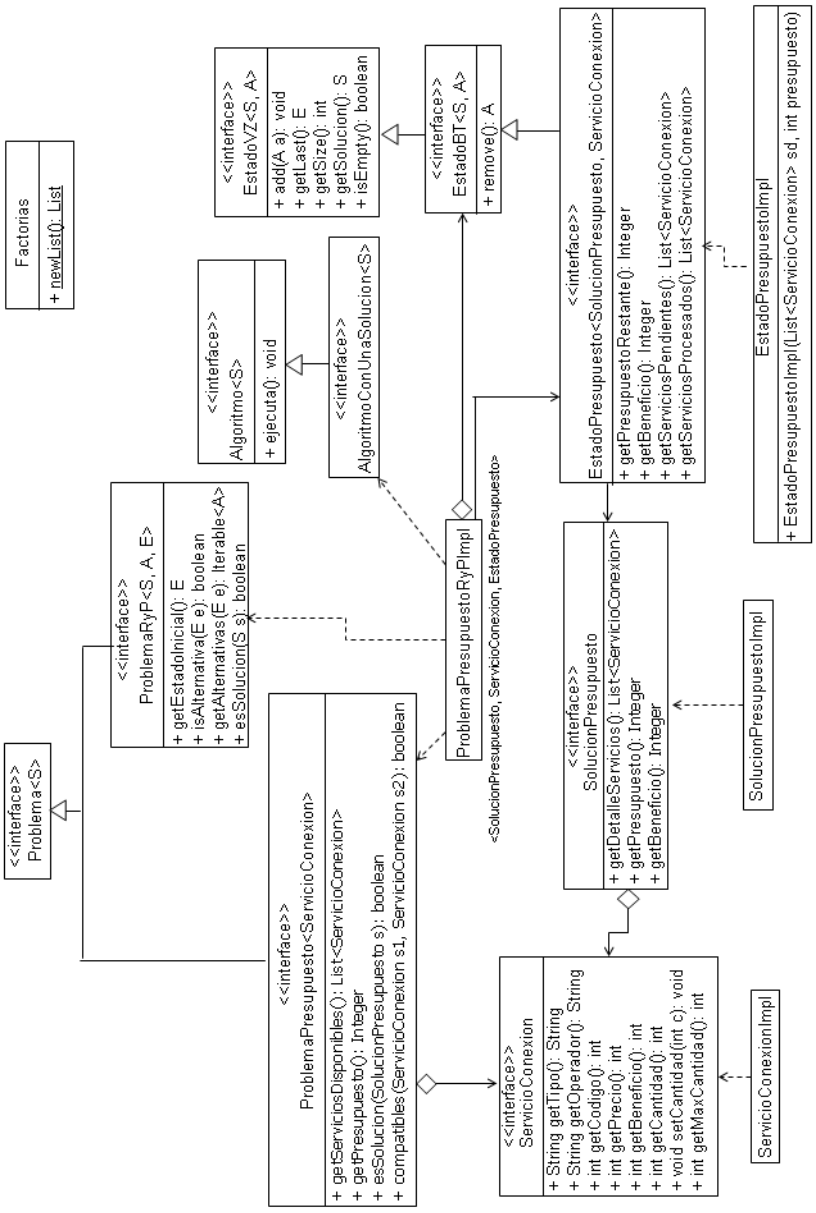
Ejercicio 1:

El departamento de informática recibe un presupuesto para ampliar las conexiones de voz y de datos de la empresa. El presupuesto se repartirá entre las distintas ofertas recibidas de los operadores de telefonía y datos. A parte de conocer el tipo de servicio (SMS, movil, Internet, etc), el precio y el operador (Orange, Yoigo, etc) se han estimado los beneficios que se obtendría si se contratara el servicio. Tenga en cuenta que un servicio se puede contratar más de una vez, por ejemplo se puede contratar más de una línea de móvil, siempre y cuando no se supere un cierto número de contratos a partir del cual la empresa no consigue más beneficios. También puede ser que el operador exija exclusividad o que si se contrata un servicio de fibra óptica no se pueda contratar otro de ADSL, por tanto no todos los servicios son compatibles. Para determinar que servicios contratar y obtener el mayor beneficio posible ajustado al presupuesto inicial se decide utilizar la técnica de ramifica y poda.

SE PIDE: con ayuda del siguiente diagrama UML implementar las funciones comentadas con // **TODO** en la clase `ProblemaPresupuestoRyPImpl`:



```
public class ProblemaPresupuestoRyPImpl implements
   AlgoritmoConUnaSolucion<SolucionPresupuesto>,
   ProblemaRyP<SolucionPresupuesto, ServicioConexion, EstadoPresupuesto>,
   ProblemaPresupuesto {

   private List<ServicioConexion> serviciosDisponibles;
   private Integer presupuesto;

   public EstadoPresupuesto getEstadoInicial() { // TODO }
   public Iterable<ServicioConexion> getAlternativas(EstadoPresupuesto e) { // TODO }
   public boolean isAlternativa(EstadoPresupuesto e) { // TODO }
   public boolean esSolucion(SolucionPresupuesto s) { // TODO }

   // Clase disponible para implementar el método getAlternativas()
   class IterableRyP implements Iterable<ServicioConexion>, Iterator<ServicioConexion> {
      int maxcount;
      ServicioConexion servicio;

      public IterableRyP(ServicioConexion servicio, int maxcount) {
           this.servicio = servicio;
           this.maxcount = maxcount;
      }

      public Iterator<ServicioConexion> iterator() { return this;  }
      public boolean hasNext() { return maxcount>=0; }
      public ServicioConexion next() {
          servicio.setCantidad(maxcount--);
          return servicio;
      }
      public void remove() { throw new UnsupportedOperationException(); }
   }

    // Devuelve verdadero si dos servicios son compatibles, falso en caso contrario.
    // NO implementar.
    public boolean compatibles(ServicioConexion s1, ServicioConexion s2) { ... }
}
```

Puntuación: 3,33 puntos **Tiempo estimado: 45 minutos**

SOLUCION

```
public class ProblemaPresupuestoRyPImpl implements
    AlgoritmoConUnaSolucion<SolucionPresupuesto>,
    ProblemaRyP<SolucionPresupuesto, ServicioConexion, EstadoPresupuesto>,
    ProblemaPresupuesto {

      List<ServicioConexion> serviciosDisponibles;
      Integer presupuesto;

      public EstadoPresupuesto getEstadoInicial() {
            return new EstadoPresupuestoImpl(serviciosDisponibles, presupuesto);
      }

      public Iterable<ServicioConexion> getAlternativas(EstadoPresupuesto e) {
            List<ServicioConexion> le = e.getServiciosPendientes();
            ServicioConexion o = le.get(le.size()-1);
            int mcount = Math.min(e.getPresupuestoRestante()/ o.getPrecio(),
                                  o.getMaxCantidad());
            for(ServicioConexion s: e.getServiciosProcesados()) {
                  if (s.getCantidad()>0 && !compatibles(o,s)) {
                        mcount = 0;
                        break;
                  }
            }
            return new IterableRyP(o,mcount);
      }

      public boolean isAlternativa(EstadoPresupuesto e) {
            return e.getServiciosPendientes().size()>0;
      }


      public boolean esSolucion(SolucionPresupuesto s) {
            return (s.getDetalleServicios().size() == getServiciosDisponibles().size()
                  && s.getPresupuesto()<=getPresupuesto());
      }

}
```