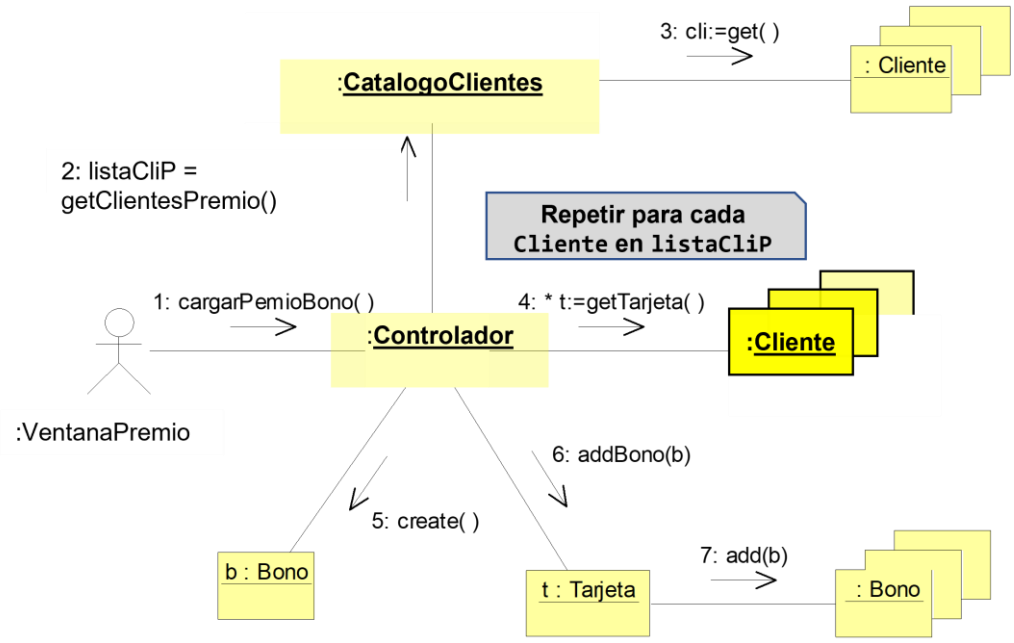
Boletín 2. Ejercicios patrones GRASP

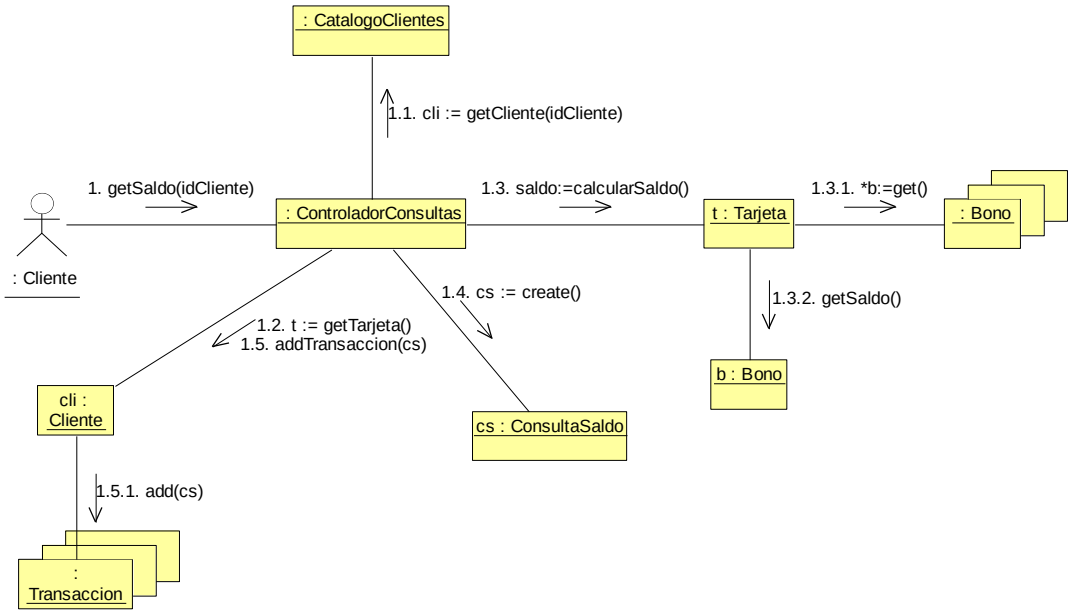
Los patrones GRASP son un conjunto de principios básicos de programación orientada a objetos que establecen cómo asignar responsabilidades a las clases y entre los que se encuentran los patrones **Experto**, **Creador** y **Controlador**. El patrón *Experto* establece la asignación de una responsabilidad a la clase que tenga la información necesaria para cumplimentarla; el patrón *Creador* establece que las instancias de cierta clase deben ser creadas en otra clase que las registre, almacene en alguna estructura de datos o tenga información de inicialización; y el patrón *Controlador* está relacionado con quién gestiona los eventos de entrada a un sistema, estableciendo que deben ser manejados por una clase que separe la capa de negocio de la capa de presentación.

En los dos primeros ejercicios, se expresa cómo se implementa una determinada funcionalidad o responsabilidad mediante una interacción de objetos de diferentes clases, y se debe indicar si se viola alguno de los patrones GRASP mencionados arriba y modifique la colaboración de objetos para que no se viole ninguno. En algunos de ellos también se pide que se escriba el código antes y después de la corrección. Los últimos tres ejercicios son preguntas de exámenes de TDS de cursos anteriores.

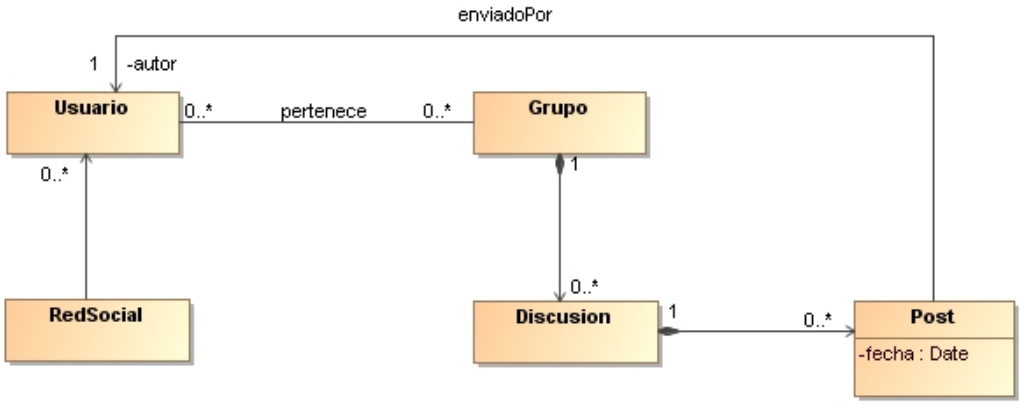
1. El siguiente diagrama UML muestra una posible interacción relacionada con un caso de uso "Regalo Bono 3G" de una aplicación para una operadora de telefonía móvil que ofrece tarjetas 3G de prepago para conexión a Internet. La funcionalidad de la interacción es la siguiente: cuando compras un bono para cierto número de horas de conexión (o de megabytes transferidos) existe la posibilidad de conseguir otro bono de regalo según un sorteo que se realiza a final de mes. Desde una ventana se lanza la carga del bono regalo a la tarjeta de los clientes premiados. Un objeto `Cliente` tiene una referencia a su objeto `Tarjeta` que puede tener asociados uno o más objetos `Bono`.



2. Dado el siguiente diagrama de comunicación UML que muestra una posible interacción para la consulta de saldo por parte de un cliente de una operadora de telefonía móvil que ofrece tarjetas 3G de prepago para conexión a Internet, **señala qué patrones GRASP son violados y modifica la interacción de modo que no se violen dichos patrones**. La funcionalidad de la interacción es la siguiente: un cliente realiza una petición de consulta de saldo y el sistema calcula el saldo como la suma del saldo de cada bono cargado en la tarjeta, además crea una transacción de tipo "consulta de saldo" que la registra en el cliente. Un objeto `Cliente` tiene una referencia a su objeto `Tarjeta` que puede tener asociados uno o más objetos `Bono`; cada cliente tiene una colección de objetos `Transacción` en la que se almacenan objetos transacción como `ConsultaSaldo`.



3. El diagrama de clases de abajo muestra una parte del modelo de dominio de una aplicación que gestiona la red social "Indignados del Mundo". Un usuario puede pertenecer a alguno de los grupos existentes. En cada grupo se inician discusiones a las que los usuarios pueden añadir posts. Cada post registra la fecha y el usuario que lo envío. La aplicación ofrece la funcionalidad de calcular el número de posts enviados por un usuario en cierto intervalo de tiempo a través del método `numPostsUsuarioEnPeriodo()` que es mostrado abajo y que está incluido en la clase `RedSocial` que actúa como controlador. Indica si se viola algún patrón GRASP. Justifica la respuesta utilizando un diagrama de secuencia.

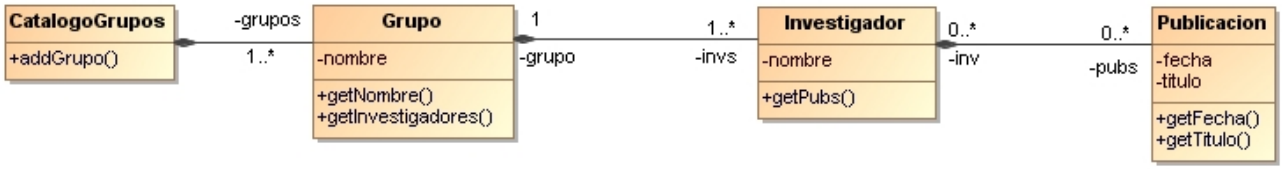


```
public class RedSocial {..

   public int numPostsUsuarioEnPeriodo(Usuario usuario, Date f1, Date f2) {
       int num = 0;
       Collection <Grupo> grupos = usuario.getGrupos();
       for (Grupo grupo : grupos){
              Collection <Discusion> discusiones = grupo.getDiscusiones();
              for (Discusion discusion : discusiones) {
                     Collection<Post> posts = discusion.getPosts();
                     for (Post post : posts) {
                            Date fecha = post.getFecha();
                            if (fecha.after(f1)&& fecha.before(f2) &&
                                                  (post.getAutor(). equals(usuario))
                                   num = num+1;
                     }
              }
       }
       return num;
   }
}
```

4. (1 punto) El código Java mostrado abajo es parte de una aplicación de gestión de la investigación de una universidad. Se trata de una ventana que muestra una lista con todas las publicaciones cuya fecha está comprendida en un rango de fechas que son introducidas con `JDateChooser`. El diagrama de clases de abajo muestra las clases del dominio que intervienen en la operación que es lanzada al pulsar el botón "Ver Publicaciones" una vez que las fechas son introducidas. Existe un catálogo de grupos de investigación, cada grupo tiene un nombre y está formado por un conjunto de investigadores, cada investigador tiene un nombre y un conjunto de publicaciones, y cada publicación tiene los campos fecha (tipo `Date`) y título. Cada clase también incluye los métodos *getter* y *setter* para sus campos.

   Indica qué patrones GRASP se han violado y reescribe el código de forma que no se violen. Si se definen nuevos métodos se debe escribir su código. Indica también si se ha violado algún principio del diseño de aplicaciones.




```
public class VentanaPublicaciones {

   private JFrame frmPublicaciones;
   private Date date1;
   private Date date2;
   private JTextArea textArea;
     …
   public VentanaPublicaciones() {
            initialize();
   }
   private void initialize() {
     frmPublicaciones = new JFrame();
     frmPublicaciones.setTitle("Publicaciones");
     frmPublicaciones.setBounds(100, 100, 450, 300);
     …
     JButton btnVerPublicaciones = new JButton("Ver Publicaciones");
     btnVerPublicaciones.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                   ArrayList<Grupo> grupos = CatalogoGrupos.getInstancia().getGrupos();
                   date1 = dateChooser.getDate();
                   date2 = dateChooser_1.getDate();
                   for (Grupo grupo: grupos)
                          for (Investigador inv:grupo.getInvestigadores())
                                 for (Publicacion pub: inv.getPubs()){
                                        SimpleDateFormat formatter =
```

```
                                        new SimpleDateFormat("dd-MM-yyyy");
                                try {
                                   Date fechaPub = formatter.parse(pub.getFecha());
                                   if  (fechaPub.after(date1) && fechaPub.before(date2)){
                                        textArea.append(pub.getTitulo()+"\n");
                                            }
                                } catch (ParseException e1) {
                                        e1.printStackTrace(); }
                            }
                    }
            });
            panel.add(btnVerPublicaciones);
            JScrollPane scrollPane = new JScrollPane();
            panel_1.add(scrollPane);
            textArea = new JTextArea();
            scrollPane.setViewportView(textArea);
        }
    }
```

5. El siguiente código es parte de una aplicación bancaria que dispone de la funcionalidad de *bloquear todas las cuentas de un cliente* a petición de Hacienda. La clase `ViewGestionCuentas` implementa una ventana a través de la cual se pueden realizar diversas operaciones sobre las cuentas de un cliente. La clase `Cliente` tiene un atributo cuentas de tipo `ArrayList<Cuenta>` que registra todas las cuentas de ese cliente y el método `getCuentas()` retorna esa colección. La clase `Cuenta` tiene un atributo `bloqueada` de tipo booleano que determina si una cuenta está bloqueada y el método `setBloqueada()` que establece su valor. Al bloquear la cuenta se crea una instancia de la clase `Bloqueo` que es una subclase de la clase abstracta `Transaccion`. Esta clase tiene un atributo para registrar la fecha actual y sus subclases añaden atributos según el tipo de transacción. La clase `Cuenta` también tiene un atributo transacciones de tipo `ArrayList<Transaccion>` que registra todas las transacciones realizadas sobre una cuenta y el método `getTransacciones()` que retorna dicha colección. La clase `CatatogoClientes` es una clase que implementa un catálogo de clientes.

```
public class ViewGestionCuentas extends JFrame {
  private Cliente cliente;
  private JTextField campoDNI;
  …
  public ViewGestionCuentas() {
     …
     JPanel panel = new JPanel();
     JLabel rotuloDNI = new JLabel("DNI Cliente ");
     campoDNI = new JTextField();
     campoDNI.setText("0");
     panel.add(rotuloDNI);
     panel.add(campoDNI);
          …
     JButton botonBloquear = new JButton("Bloquear");
     botonBloquear.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
               Cliente cliente = CatalogoClientes.getInstancia().
                                    getCliente(campoDNI.getText());
               if (cliente != null){
                   ArrayList<Cuenta> cuentas = cliente.getCuentas();
                   for (Cuenta cuenta: cuentas) {
                         cuenta.setBloqueada(true);
                         ArrayList<Transaccion> transacciones =
                                                cuenta.getTransacciones();
                    transacciones.add(new Bloqueo(new Date()));
                   }
            } else JOptionPane.showMessageDialog(ViewGestionCuentas.this,
                                "No existe cliente con ese dni");
       }
     });
     …
  }
 …
}
```