EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Apartado 1. (2,5 puntos)

Tienes que diseñar parte de una aplicación que se utilizará durante procesos electorales de ámbito geográfico extenso para gestionar de forma flexible y ágil tanto el recuento de los votos como la distribución de los resultados según vaya avanzando el escrutinio.

Un *punto de recuento* puede ser una *mesa electoral individual* o bien un *centro regional de recuento* que agrupe a otros puntos de recuento. Cualquiera de ellos se caracteriza por un nombre, un estado del recuento (que puede estar *no iniciado*, *activo* o *terminado*) y el número de votos escrutados hasta ese momento a favor de cada *entidad elegible* o susceptible de ser votada (por ejemplo, estas pueden ser partidos, candidatos a Presidente, opciones de un referéndum, etc). Las entidades elegibles quedan descritas mediante un texto breve (su nombre o descripción). Antes de iniciarse la votación los centros de recuento habrán sido configurados añadiendo a cada uno de ellos los puntos de recuento que agrupa.

El único recuento manual real se realiza a nivel de mesas electorales. En cada mesa, tras el recuento de cada urna, se le suman a cada entidad elegible los votos que ha recibido en dicha urna. La aplicación debe propagar automáticamente esta actualización de votos escrutados en cada mesa electoral al centro regional de recuento que incluya de manera directa a esa mesa, y análogamente de ese centro de recuento a otro, y así sucesivamente. Por ejemplo, en unas elecciones nacionales, se acumularán los votos desde las mesas a los municipios, de estos a las provincias, a las comunidades autónomas, etc.

En el diseño de esta parte de la aplicación se debe contemplar la existencia de otros módulos (programados aparte) que implementarán diversos tipos de agentes interesados en recibir puntualmente información de cada actualización de votos en los puntos de recuento de su interés. Esos agentes pueden corresponderse con medios de comunicación, oficinas del gobierno, interventores, robots en la Web, etc. Tu diseño debe incluir mecanismos para que sea fácil integrar un número arbitrario de estos actores. Cada agente debe poder indicar a uno o varios puntos de recuento de interés que le notifiquen de las actualizaciones en el recuento de votos de todas las entidades elegibles.

Se pide:

(a) Representar el diagrama de clases en UML para el fragmento de aplicación descrito arriba, sin incluir los métodos *getters* o *setters* ni los constructores, ni las clases relacionadas con una posible interfaz de usuario.
[1,8 puntos]

(b) Para cada uno de los siguientes requisitos del programa, describe el/los métodos necesarios (con su nombre, argumentos, tipo de resultado, y demás características) e indica las clases más apropiada para contenerlos:

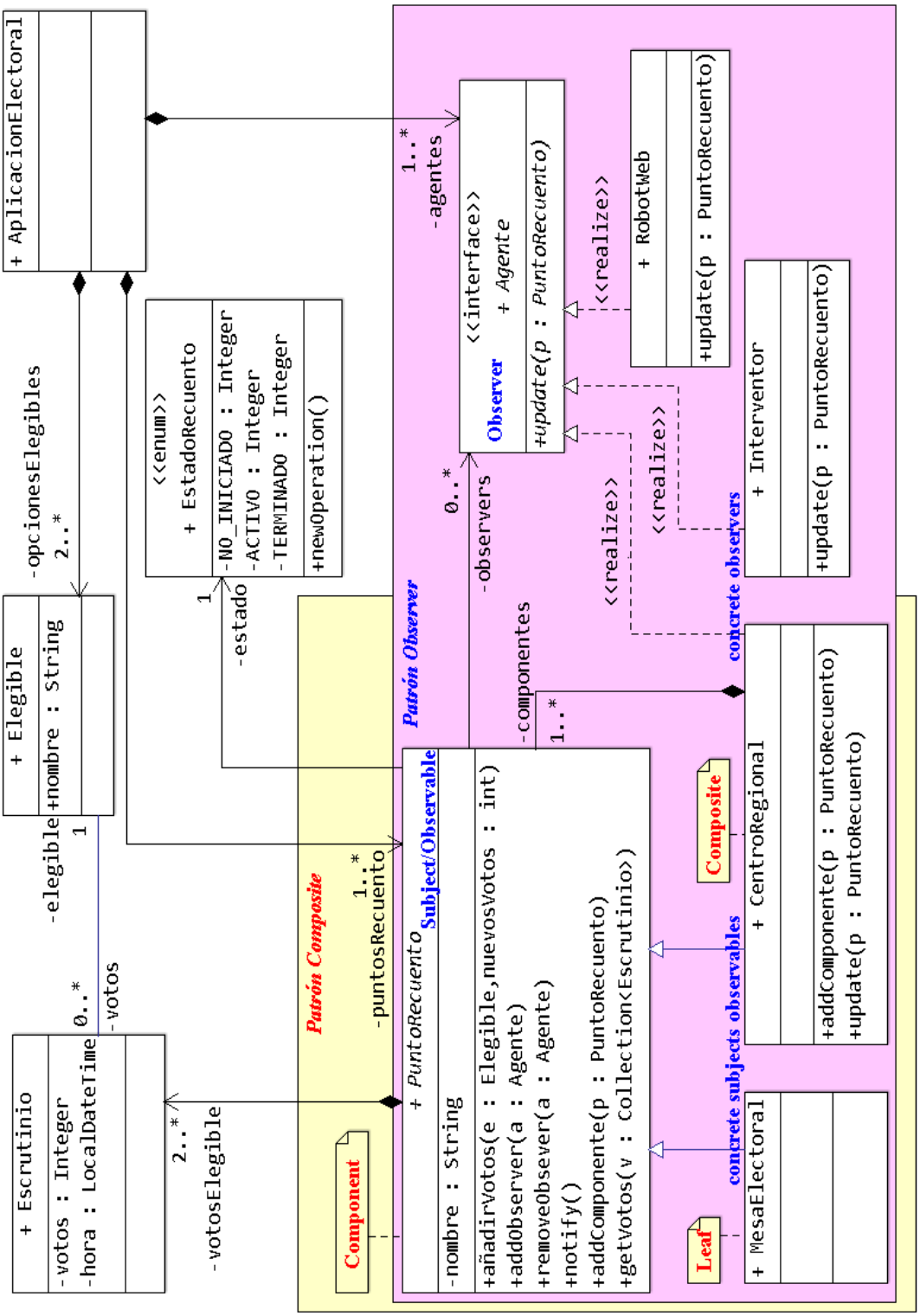
1. Añadir votos escrutados a favor de un candidato [0,2 puntos]
2. Configurar los agentes seguidores con sus puntos de recuento de interés, y comunicar a estos agentes que hay nuevos votos [0,2 puntos]

(c) ¿Qué patrones de diseños has utilizado y qué clases de tu diseño se corresponden con ellos? [0,3 puntos]

EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas



(b) métodos con parámetros y resultado detallados en diagrama de clases
(b.1) método añadirVotos en clase abstracta PuntoRecuento
(b.2) métodos addObserver, removeObserver, notify en PuntoRecuento y método update en interfaz Agente y sus implementaciones.
(c) patrones detallados en el diagramas, con el rol que cada clase de la aplicación tiene en cada patrón

EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Apartado 2. (2,5 puntos)

Se quiere construir una aplicación para la gestión de redes de ordenadores. Una red de ordenadores puede contener ordenadores, o bien otras redes. Las redes y ordenadores tienen un nombre y un identificador único numérico que el sistema asigna automáticamente en su creación. Los ordenadores tienen además un sistema operativo (Windows, MacOS o Ubuntu).

El programa debe emular mandar un mensaje desde un ordenador a otro ordenador, o desde un ordenador a una red (en cuyo caso, será recibido por todos los ordenadores que pertenezcan directa o indirectamente a la red destino). Se ha de lanzar un error si se trata de enviar un mensaje desde un ordenador hasta otro ordenador o red que no sea alcanzable (el origen y el destino no están contenidos en la misma red, directa o indirectamente).

Como ves en el programa de más abajo, la llamada al método `enviarMensaje` debe causar la impresión por consola de una confirmación de la recepción del mensaje en cada ordenador que lo recibe. En la tercera llamada a `enviarMensaje`, el ordenador "Turing" no es alcanzable desde "Godel", ya que "Godel" está en la red "EPS", que a su vez está en la red "UAM". Ni "EPS", ni "UAM" contienen a "Turing" directa o indirectamente, y por eso se lanza una excepción.

Se pide:
a) Implementar las clases y excepciones necesarias para que el siguiente código produzca la salida de más abajo. Se valorará específicamente la calidad del diseño, el uso correcto de conceptos de orientación a objetos y la concisión del código [2,25p]
b) ¿Qué patrón(es) de diseño has utilizado? Explica los roles de las clases, interfaces y métodos en los patrones usados [0,25p]

```
public class MonitorRed {
  public static void main (String... args) {
    Red redUAM = new Red("UAM");                  // Red de primer nivel con nombre UAM
    Red redEPS = new Red("EPS", redUAM);   // subred EPS dentro de la red UAM
    Red redCiencias = new Red("Ciencias"); // Red de primer nivel con nombre Ciencias
    Ordenador PCeps1 = new Ordenador("Escher", redEPS, SistemaOperativo.MACOS); // Ordenador Escher en red EPS
    Ordenador PCeps2 = new Ordenador("Godel", redEPS, SistemaOperativo.WINDOWS);       // Ordenador Godel en
red EPS
    Ordenador PCuam = new Ordenador("Bach", redUAM, SistemaOperativo.UBUNTU);   // Ordenador Bach en red UAM
    Ordenador PCciencias = new Ordenador("Turing", redCiencias, SistemaOperativo.UBUNTU);  // Ordenador en
Ciencias

    try {
        PCeps1.enviarMensaje(redEPS, "Mensaje de prueba a EPS");          // De Escher a toda la EPS
        System.out.println("======================" );
        PCeps2.enviarMensaje(redUAM, "Otro mensaje a toda la UAM"); // De Godel a toda la EPS
        System.out.println("======================" );
        PCeps2.enviarMensaje(PCciencias, "Mensaje a Turing");      // Falla: Turing no es alcanzable
    } catch (NoAlcanzableExcepcion e) {
        System.out.println(e);
    }
  }
}
```

Salida esperada:

```
Equipo Escher(id:3)[MACOS] : recibido "Mensaje de prueba a EPS" para EPS(id:1)[red] desde Escher(id:3)[MACOS]
Equipo Godel(id:4)[WINDOWS] : recibido "Mensaje de prueba a EPS" para EPS(id:1)[red] desde Escher(id:3)[MACOS]
======================
Equipo Escher(id:3)[MACOS] : recibido "Otro mensaje a toda la UAM" para UAM(id:0)[red] desde
Godel(id:4)[WINDOWS]
Equipo Godel(id:4)[WINDOWS] : recibido "Otro mensaje a toda la UAM" para UAM(id:0)[red] desde
Godel(id:4)[WINDOWS]
Equipo Bach(id:5)[UBUNTU] : recibido "Otro mensaje a toda la UAM" para UAM(id:0)[red] desde
Godel(id:4)[WINDOWS]
======================
Error: el elemento Turing(id:6)[UBUNTU] no es alcanzable desde Godel(id:4)[WINDOWS]
```

Nota: En el constructor de la clase Red no es necesario comprobar ciclos en la estructura de la red (se asume que no existen). Tampoco es necesario añadir métodos para cambiar los ordenadores de una red a otra.

EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Una solución al ejercicio de continua:

```
enum SistemaOperativo { MACOS, UBUNTU, WINDOWS }

class NoAlcanzableExcepcion  extends Exception{
        private ElementoRed source, dest;
        public NoAlcanzableExcepcion(ElementoRed  s, ElementoRed t) { this.source = s; this.dest = t; }
        public String toString() { return "Error: el elemento "+this.dest+" no es alcanzable desde
"+this.source;}
}
abstract class ElementoRed {
        protected String nombre;
        protected Red red;

        public ElementoRed (String n, Red p) {
                this.nombre = n;
                this.red = p;
                if (p!=null) p.add(this);
        }
        public ElementoRed (String n) { this(n, null);  }
        public abstract void recibirMensaje(ElementoRed  eq, ElementoRed tar, String mensaje);
        public boolean contains(ElementoRed  el) { return this.equals(el); }
}
class Ordenador extends ElementoRed {
        private SistemaOperativo ssoo;

        public Ordenador (String name, Red p, SistemaOperativo so) {
                super(name, p);
                this.ssoo = so;
        }
        public void enviarMensaje(ElementoRed  tar, String mensaje) throws NoAlcanzableExcepcion  {
                if (this.red.puedeAlcanzar( tar)) tar.recibirMensaje( this, tar, mensaje);
                else throw new NoAlcanzableExcepcion( this, tar);
        }
        public void recibirMensaje(ElementoRed  eq, ElementoRed tar, String mensaje) {
                System.out.println("Equipo "+this+" : recibido \""+mensaje+"\" para "+tar+" desde "+eq);
        }
        public String toString() { return this.nombre+"["+this.ssoo+"]"; }
}
class Red extends ElementoRed {
        private List<ElementoRed>  elementos = new ArrayList<ElementoRed>();

        public Red (String name) { super(name); }
        public Red (String name, Red padre) { super(name, padre); }
        public void add(ElementoRed  equipo) { this.elementos.add(equipo); }
        public void recibirMensaje(ElementoRed  src, ElementoRed tar, String mensaje) {
                for (ElementoRed e: this.elementos)
                                e.recibirMensaje( src, tar, mensaje);
        }
        protected boolean puedeAlcanzar(ElementoRed  el) {
                if (this.contains(el)) return true;                     // es alcanzable desde esta red...
                if (this.red != null) return this.red.puedeAlcanzar(el);        // ... o desde el padre
                return false;
        }
        public boolean contains (ElementoRed el) {
                if (this.equals(el)) return true;       // true: el propio objeto
                for (ElementoRed er : this.elementos)   // buscar recursivamente hacia abajo
                        if (er.contains(el)) return true;
                return false;
        }
        public String toString() { return this.nombre+"[red]"; }
}
```

EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Una solución al ejercicio de NO continua:

```
enum SistemaOperativo { MACOS, UBUNTU, WINDOWS }
class NoAlcanzableExcepcion  extends Exception{
    private ElementoRed source, dest;
    public NoAlcanzableExcepcion(ElementoRed  s, ElementoRed t) { this.source = s; this.dest = t; }
    public String toString() { return "Error: el elemento "+this.dest+" no es alcanzable desde "+this.source;
}
}
abstract class ElementoRed {
    private static int numEls = 0;
    protected int id;
    protected String nombre;
    protected Red red;
    public ElementoRed (String n, Red p) {
            this.nombre = n;
            this.red = p;
            if (p!=null) p.add(this);
            this.id = ElementoRed.numEls++;
    }
    public ElementoRed (String n) { this(n, null);  }
    public abstract void recibirMensaje(ElementoRed  eq, ElementoRed tar, String mensaje);
    public boolean contains(ElementoRed  el) { return this.equals(el); }
    public String toString() { return this.nombre+"(id:"+this.id+")"; }
}
class Ordenador extends ElementoRed {
    private SistemaOperativo ssoo;
    public Ordenador (String name, Red p, SistemaOperativo so) {
            super(name, p);
            this.ssoo = so;
    }
    public void enviarMensaje(ElementoRed  tar, String mensaje) throws NoAlcanzableExcepcion  {
            if (this.red.puedeAlcanzar( tar)) tar.recibirMensaje( this, tar, mensaje);
            else throw new NoAlcanzableExcepcion( this, tar);
    }
    public void recibirMensaje(ElementoRed  eq, ElementoRed tar, String mensaje) {
            System.out.println("Equipo "+this+" : recibido \""+mensaje+"\" para "+tar+" desde "+eq);
    }
    public String toString() { return super.toString()+"["+this.ssoo+"]"; }
}
class Red extends ElementoRed {
        private List<ElementoRed>  elementos = new ArrayList<ElementoRed>();
        public Red (String name) { super(name); }
        public Red (String name, Red padre) { super(name, padre); }
        public void add(ElementoRed  equipo) { this.elementos.add(equipo); }
        public void recibirMensaje(ElementoRed  src, ElementoRed tar, String mensaje) {
                for (ElementoRed e: this.elementos)
                                e.recibirMensaje( src, tar, mensaje);
        }
        protected boolean puedeAlcanzar(ElementoRed  el) {
                if (this.contains(el)) return true;                     // es alcanzable desde esta red...
                if (this.red != null) return this.red.puedeAlcanzar(el);        // ... o desde el padre
                return false;
        }
        public boolean contains (ElementoRed el) {
                if (this.equals(el)) return true;               // true: el propio objeto
                for (ElementoRed er : this.elementos)   // buscar recursivamente hacia abajo
                        if (er.contains(el)) return true;
                return false;
        }
        public String toString() { return super.toString()+"[red]"; }
}
```

EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Apartado 3. (2,5 puntos)

Queremos tener una clase para gestionar la agrupación entre objetos *emparejables* que implementen la siguiente interfaz para definir su afinidad:

```
interface Emparejable  {
  boolean afinCon(Emparejable m); //debe ser reflexiva: x.afinCon(y) igual que y.afinCon(x)
}
```

La afinidad entre objetos de un mismo tipo T que implemente dicha interfaz se debe gestionar asociando cada objeto *emparejable* de tipo T con todos aquellos objetos conocidos de tipo T con los que él sea afín. La afinidad no siempre significa igualdad, proximidad o parecido directo, sino que puede significar lejanía o cualquier otra relación reflexiva. Por ejemplo, en una biblioteca, dos libros podrían definirse afines si son del mismo autor. Y en una dieta por ejemplo, dos platos se consideran afines si sus aportes calóricos son muy distintos (en el programa de abajo se consideran afines con al menos 200 kcal de diferencia).

El objeto que implementa la agrupación por afinidad se inicializa opcionalmente con unos objetos emparejables, y permite que después se le añadan y borren otros objetos emparejables, pero sólo aceptará objetos de un único tipo T para el que haya sido creado. Al añadir un objeto se le incluirá en todas la colecciones de objetos previamente almacenados que sean afines con él y quedará asociado con su propia colección de objetos afines con él. Al borrar un objeto no solo desaparecerá él mismo con su colección de afines, sino que también se le eliminará de todas las colecciones de afinidad en las que hubiese sido incluido.

Se pide: codificar las clases `AgrupacionAfines` y `Plato` para que este programa ejemplo produzca la salida dada abajo, teniendo en cuenta que el orden de la líneas respeta el orden en que se añaden los objetos y cada agrupación de afines entre corchetes [ ] está ordenada alfabéticamente por nombre:

```
public class Main {
  public static void main(String[] args) {
    AgrupacionAfines<Plato> a =
          new AgrupacionAfines<>(new Plato("Ensalada", 100), // nombre de plato y Kcal
                                 new Plato("Lentejas", 375),
                                new Plato("Paella", 200),
                                 new Plato("Ensalada", 999)); //ya existe, no añadir
//Nota: En la versión de continua se añaden los mismos elementos,
// pero con el método add, en lugar de con el constructor
    System.out.println( a.add(new Plato("Cocido", 450)).add(new Plato("Filete", 225)) );

    System.out.println( a.remove( new Plato("Ensalada", 0) ) ); // eliminar "Ensalada"
  }
}
```

Salida esperada, donde se han añadido saltos de línea a mano para clarificar el contenido:

```
{Ensalada:100=[Cocido:450, Lentejas:375],
Lentejas:375=[Ensalada:100],
Paella:200=[Cocido:450],
Cocido:450=[Ensalada:100, Filete:225, Paella:200],
Filete:225=[Cocido:450]}

{Lentejas:375=[],
Paella:200=[Cocido:450],
Cocido:450=[Filete:225, Paella:200],
Filete:225=[Cocido:450]}
```

Nota. La primera línea de la salida significa que `Cocido` y `Lentejas` son afines a `Ensalada` (más de 200 Kcal de diferencia), pero ambos no tienen por qué ser afines entre sí (y en este caso, no lo son).

EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

```
public class Plato implements Emparejable, Comparable<Plato>{
	private final String nombre;
	private final int calorias;
	public Plato(String nombre, int calorias){
		this.nombre=nombre;
		this.calorias=calorias;
	}
	public boolean afinCon(Emparejable m) {
		return m instanceof Plato &&
				Math.abs(((Plato)m).calorias-calorias) >=200;
	}
	public String toString(){
		return nombre+":"+calorias;
	}
	public int hashCode() {
		return nombre.hashCode();
	}
	public boolean equals(Object obj) {
		if (this == obj) return true;
		if (obj instanceof Plato){
		 Plato other = (Plato) obj;
		 return nombre.equals(other.nombre);
		}
		else return false;
	}
	public int compareTo(Plato o) {
		return nombre.compareTo(o.nombre);
	}
}
public class AgrupacionAfines<T extends Emparejable> {
	private final Map<T, Set<T>> grupos=new LinkedHashMap<>();
	public AgrupacionAfines(T... elementos){
		for (T t:elementos)
			add(t);
	}
	public AgrupacionAfines<T> add(T e) {
		if (!grupos.containsKey(e)){
			Set<T> afines= new TreeSet<>();
			for (Map.Entry<T, Set<T>> g:grupos.entrySet()){
				if (e.afinCon(g.getKey())){
					afines.add(g.getKey());
					g.getValue().add(e);
				}
			}
			grupos.put(e, afines);
		}
		return this;
	}
	public AgrupacionAfines<T> remove(T e) {
		if (grupos.containsKey(e)){
			Set<T> afines= grupos.remove(e);
			for (T a:afines){
				grupos.get(a).remove(e);
			}
		}
		return this;
	}

	public String toString() {
		return grupos.toString();
	}
}
```

EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Apartado 4. (2,5 puntos)

Se quiere construir una clase genérica `ConjuntoInfinito` para emular conjuntos ordenados, de tamaño potencialmente infinito. Este tipo de conjuntos no mantiene sus elementos en memoria (ya que sería imposible), sino que se configuran con una condición de pertenencia, que se utiliza para saber si un elemento pertenece al conjunto.

El constructor de `ConjuntoInfinito` recibe tres parámetros: la condición de pertenencia, y dos parámetros adicionales que permiten buscar miembros potenciales del conjunto. El segundo parámetro es el elemento inicial a partir del que se buscarán elementos, y el tercero es una expresión que permite obtener el siguiente elemento potencial del conjunto. Este siguiente elemento pertenecerá al conjunto sólo si cumple la condición de pertenencia.

Para facilitar el recorrido de los elementos de un conjunto infinito, se creará una clase `RecorreConjunto`. Esta clase debe tener métodos para obtener el elemento actual de la iteración (`elementoActual`) y avanzar al siguiente elemento (`avanza`). Adicionalmente debe ser posible convertir un `ConjuntoInfinito` a un conjunto ordenado estándar de Java (truncándolo hasta un límite máximo de elementos) y a un Stream limitado a cierta longitud. Además, se debe tener un método (`contiene`) para comprobar si un elemento pertenece al conjunto.

Se pide:

a) Implementar las clases e interfaces necesarias para que el siguiente código produzca la salida de más abajo. Se valorará específicamente la calidad del diseño, el uso correcto de conceptos de orientación a objetos y la concisión del código. [2p]

b) ¿Qué patrón(es) de diseño has utilizado? Explica los roles de las clases, interfaces y métodos en los patrones usados. [0,5p]

```
public class Conjuntos {
  public static void main(String ...args) {
    ConjuntoInfinito<Integer>  pares =           // Conjunto infinito de los números pares
         new ConjuntoInfinito<>(  x -> x % 2 == 0, // condición de pertenencia al conjunto
                                  0,              // elemento inicial (no necesariamente debe cumplir la
condición)
                                  n -> n + 1);    // Calcula el próximo elemento a considerar

    RecorreConjunto<Integer>  rec = pares.getRecorreConjunto();

    int num = 0;
    while (num++ < 20) {                                        // imprime los 20 primeros elementos
        System.out.print(rec.elementoActual()+" ");     // imprime el elemento actual
        rec.avanza();                                   // avanza al siguiente elemento
    }

    System.out.println("\nContiene el 2934: "+pares.contiene(2934));  // Lo contiene, ya que es par
    SortedSet<Integer>  conjuntoNormalOrdenado  = pares.toSet(5);    // Convertir a un conjunto normal
ordenado...
    System.out.println( conjuntoNormalOrdenado );                    // ... que se trunca a 5 elementos
    System.out.println( pares.toStream(5).reduce(0,  (x, y) -> x+y));  // obtiene un stream de longitud 5 y suma

  }
}
```

Salida esperada:

```
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
Contiene el 2934: true
[0, 2, 4, 6, 8]
20
```

EVALUACIÓN CONTINUA & NO CONTINUA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
public class ConjuntoInfinito<T extends Comparable<T>> { // ConcreteAggregate en patrón Iterator
        private Predicate<? super T> pertenencia;
        private T semilla;
        private Function<? super T, ? extends T> funcionSiguiente;

        public ConjuntoInfinito(Predicate<? super T> pertenencia,   T semilla,
                                Function<? super T, ? extends T> funcionSiguiente) {
                this.pertenencia = pertenencia;
                this.semilla = semilla;
                this.funcionSiguiente = funcionSiguiente;
        }
        // getters para que el Iterator acceda a componentes necesarios para poder iterar
        public Predicate<T> pertenencia() { return (Predicate<T>) pertenencia; }
        public T semilla() { return semilla; }
        public Function<T, T> funcionSiguiente() { return (Function<T, T>) funcionSiguiente; }

        public RecorreConjunto<T> getRecorreConjunto() { // método createIterador en patrón
            return new RecorreConjunto<>(this); // Factory Method que devuelve un iterador nuevo
        }
        public boolean contiene(T x) {// parte exclusiva en No Continua
            RecorreConjunto<T> it = this.getRecorreConjunto(); // usa su propio Iterator
            while (! it.elementoActual().equals(x))  it.avanza();
            return true;
        }
        public SortedSet<T> toSet(int max) {
            SortedSet<T>  resultado = new TreeSet<>();
            RecorreConjunto<T> it = this.getRecorreConjunto(); // usa su propio Iterator
            IntStream.rangeClosed(1,max).forEach(i -> { resultado.add(it.elementoActual());
                                                        it.avanza(); });
            return resultado;
        }
        public Stream<T> toStream(int i) { return this.toSet(i).stream(); } // reutiliza toSet()
}
public class RecorreConjunto<T extends Comparable<T>> {// clase ConcreteIterator en patrón Iterator
        private  T actual;
        private ConjuntoInfinito<T> conjunto; // conjunto sobre el que se itera

        public RecorreConjunto(ConjuntoInfinito<T> conjunto) {
                this.conjunto = conjunto;
                this.actual = this.conjunto.semilla();
                // la semilla puede no cumplir la pertenencia la conjunto, luego:
                if (! this.conjunto.pertenencia().test( this.actual))  avanza();
        }

        public T elementoActual() { return actual; } // currentItem() en patron
        public void avanza() {// this may never end! // next() en patrón
                do // bucle principal para "generar" candidatos a base de apply y test
                        this.actual = this.conjunto.funcionSiguiente().apply( this.actual );
                while ( ! this.conjunto.pertenencia().test( this.actual ) );
        }
}
```