Curso 2019-2019. Cuatrimestre Primavera. 21 de Junio de 2019
Programación Orientada a Objetos.
Examen Final

Imaginad que se desea crear un programa para una gestionar el día a día de la operativa de una compañía de seguros.

La fase inicial del análisis ha conducido al diagrama de clases incompleto (faltan métodos en las clases) que se muestra en la última página.

El programa debe gestionar **seguros** que los **clientes** contratan con la compañía. La compañía ofrece **seguros de coches** y **de hogar**. Los clientes pueden ser **empresas** o **personas**.

El cliente firma con la compañía una **póliza**, en la que se relacionan las **cláusulas** que detallan las coberturas del seguro. El sistema asocia a cada seguro la relación de **partes de incidencias** que el cliente notifique.

El cliente debe pagar una cuota anual por cada seguro cuyo valor se calcula aplicando a la cuota base anual el descuento o el recargo que el cliente obtenga en función de los partes de accidentes cargados a cuenta de ese seguro.

Se dispone de una clase Fecha ya diseñada y programada (NO debéis programarla) que ofrece tres métodos estáticos, que podréis utilizar cuando lo consideréis necesario:

```
public static String getFechaDeHoy() ;
```

Devuelve la fecha del día en que se invoca en un String (podéis suponer un formato dd-mm-aaaa, aunque no os será necesario este detalle para la resolución del examen).

```
public static int getAnyoDeFecha(String fecha) ;
```

Devuelve un entero representando el año de la fecha pasada como argumento.

```
public static int compareTo(String unaFecha, String otraFecha) ;
```

Compara las dos fechas pasadas como argumentos. Devuelve: un entero negativo si la fecha indicada por `unaFecha` es anterior a la fecha indicada por `otraFecha`; un cero si ambas fechas son idénticas; un entero positivo si la fecha indicada por `unaFecha` es posterior a la fecha indicada por `otraFecha`.

NOTA 1: Al resolver las preguntas del examen podéis suponer implementadas los métodos **getXXX() y setXXX()** que acceden a los atributos de las clases. **No es necesario que los implementéis**.

NOTA 2: Para la realización de los ejercicios **NO debéis suponer que los partes están ordenados** en modo alguno.

Ejercicio 1 **(1 punto).** A partir del diagrama UML mostrado (eso quiere decir que NO hay que escribir nada para las clases derivadas de Exception), escribid para todas las clases excepto para InterficieUsuario y Fecha, la parte de código de la definición que comienza con el "public class…." correspondiente a la declaración de los todos los atributos de las clases, incluyendo aquellos que aparecen al implementar las relaciones.

SOLUCIÓN:

```
class Controlador {
```

```
    private Map<String,Cliente> clientes ; // Se admite también una lista
    private List<Seguro> seguros ;
. . .
}

public class Cliente {

    protected String nif ;
    protected String direccion ;
    protected String telefono ;
    protected String clienteDesde ;
    protected List<Seguro> seguros ;
.  .  .
}

public abstract class Seguro {

    protected String fechaContrato ;
    protected double cuotaAnualBase ;
    protected double descuentoORecargo ;
    protected Cliente cliente ;
    protected Poliza poliza ;
    protected List<Parte> partes ;
. . .
}

public class Poliza {
    private String fechaDeFormalizacion ;
    private List<Clausula> clausulas ;
    private Seguro seguro ;
. . .
}

public class Clausula {
    private String texto ;
. . .
}

public class Parte {
    private String descripcion ;
    private String informe ;
    private double costeIndemnizacion ;
    private String fechaIncidencia ;
    private Seguro seguro ;
. . .
}
public class SeguroDeCoche extends Seguro{
    private String gamaVehiculo ;
. . .
}

public class SeguroDeHogar extends Seguro{
    private String tipoVivienda ;
. . .
}

public class Persona extends Cliente{
    private String nombre ;
    private String apellidos ;
. . .
}

public class Empresa extends Cliente{
    private String nombreComercial ;
. . .
}
```

Ejercicio 2 **(1,5 puntos)**. Implementad en la clase Seguro el método que se indica a continuación:

```
public String listaPartesEntreFechas(String fechaInicio, String
fechaFinal) ;
```

Este método genera un String que incluye detalles de los partes de incidencia asociados al seguro en cuestión, ocurridos entre la fecha indicada en el argumento `fechaInicio` y la fecha indicada por el argumento `fechaFinal` (ambas incluidas).

Se sugiere la implementación del método toString() en la clase Parte para que el código de dicho método sea más corto. El string resultante NO debe contener ninguna otra información del seguro asociado al parte.

SOLUCIÓN:

```
    public String listaPartesEntreFechas(String fechaInicio, String fechaFinal){
        StringBuilder result = new StringBuilder() ;
        Iterator<Parte> it = this.partes.iterator() ;
        while(it.hasNext()){
            Parte parte = it.next() ;
            String fechaDeParte = parte.getFechaIncidencia() ;
            int compConInicio = Fecha.compareTo(fechaDeParte, fechaInicio) ;
            int compConFinal = Fecha.compareTo(fechaDeParte, fechaFinal) ;
            if((compConInicio==0 || compConInicio>0) &&
                    (compConFinal==0 || compConFinal<0)){
                result.append(parte.toString()).append("\n\n") ;
            }
        }
        return result.toString() ;
    }
```

En la clase Parte:

```
    public String toString() {
        StringBuilder result = new StringBuilder();
        result.append("Descripcion: ").append(this.descripcion).append("; Informe: ");
        result.append(this.informe).append("\nCoste indemnización: ") ;
        result.append(this.costeIndemnizacion).append("; fecha incidencia : ") ;
        result.append(this.fechaIncidencia);
        return result.toString();
    }
```

Ejercicio 3 **(1,5 puntos)**. Implementad en la clase Seguro los siguiente métodos:

```
public int numPartesEnAnyo(int anyo) ;
```

Este método devuelve el número de incidencias notificadas para el seguro durante el año indicado por el argumento `anyo`.

SOLUCIÓN:

```
    public int numPartesEnAnyo(int anyo){
        int result = 0 ;
        Iterator<Parte> it = this.partes.iterator() ;
        while(it.hasNext()){
            Parte parte = it.next() ;
            String fechaDeParte = parte.getFechaIncidencia() ;
            int anyoIncidencia = Fecha.getAnyoDeFecha(fechaDeParte) ;
            if(anyo==anyoIncidencia){
                result++ ;
            }
        }
        return result ;
    }
```

```
public double costeIncidenciasEnAnyo(int anyo) ;
```

Este método devuelve la suma de los costes de todas las incidencias que se dieron para el seguro durante el año indicado por el argumento `anyo`.

SOLUCIÓN:

```
public double costeIncidenciasEnAnyo(int anyo){
      double result = 0 ;
    Iterator<Parte> it = this.partes.iterator() ;
    while(it.hasNext()){
        Parte parte = it.next() ;
        String fechaDeParte = parte.getFechaIncidencia() ;
        int anyoIncidencia = Fecha.getAnyoDeFecha(fechaDeParte) ;
        if(anyo==anyoIncidencia){
            result += parte.getCosteIndemnizacion() ;
        }
    }
    return result ;
}
```

Ejercicio 4 **(2 puntos)**. En la clase Seguro el atributo `descuentoORecargo` indica el porcentaje en % en el que el atributo `cuotaAnualBase` debe incrementarse para calcular la cuota anual correspondiente a un seguro que el cliente debe pagar. Un valor positivo indica un recargo; un valor negativo indica un descuento.
La clase Seguro incorpora el método que sigue:

```
public void actualizarDescuentoORecargo() ;
```

Este método calcula un nuevo valor para el atributo `descuentoORecargo` según las siguientes reglas.

- Si durante el año anterior al año en curso se han generado más de N partes de incidencias asociados a este seguro O la suma de los costes por indemnización de los partes generados durante el año anterior al año en curso superan los TOTIND Euros, se incrementa el valor del atributo `descuentoORecargo` en D1.
- Si durante el año anterior al año en curso no se ha generado ningún parte de incidencia, el atributo `descuentoORecargo` se reduce en D2.
- Si no se cumple ninguna de las condiciones anteriores, el valor de `descuentoORecargo` no se altera.
- En ningún caso el valor del atributo `descuentoORecargo` puede valer más de 25 y menos que -15.

Los valores N, TOTIND, D1 y D2 varían en función del tipo de seguro del que se trate, según la tabla mostrada a continuación:

Clase	N	TOTIND	D1	D2
SeguroDeCoche	2	500	4	1
SeguroDeHogar	3	700	5	2

La clase Seguro incorpora los métodos abstractos `getN()`, `getTOTIND()`, `getD1()` y `getD2()` que implementan sus subclases.

Implementad los métodos `getN()`, `getTOTIND()`, `getD1()` y `getD2()` de SeguroDeCoche y SeguroDeHogar para que devuelvan los valores de la tabla anterior.

Implementad en la clase Seguro el método `actualizarDescuentoORecargo()`.

Obviamente, para implementar este método podéis utilizar los métodos de apartados anteriores y cualesquiera que consideréis necesarios.

SOLUCIÓN:

```
    public void actualizarDescuentoORecargo(){
        int anyoAnterior = Fecha.getAnyoDeFecha(Fecha.getFechaDeHoy()) - 1;
        int numPartes = this.numPartesEnAnyo(anyoAnterior) ;
        double costeIndem = this.costeIncidenciasEnAnyo(anyoAnterior) ;
        //No hay partes, aumentar el descuento si este no es menor que el valor permitido
        if(numPartes==0){
            if(this.descuentoORecargo>=-15+this.getD2()){
                this.descuentoORecargo -= this.getD2() ;
                return ;
            }
            return ;
        }
        // Hay partes
        if(numPartes>this.getN() || costeIndem>this.getOTIND()){
            if(this.descuentoORecargo<=25-this.getD1()){
                this.descuentoORecargo += this.getD1() ;
                return ;
            }
        }
        return ;
    }
```

Métodos en SeguroDeCoche:

```
    public int getN() {
        return 2 ;
    }
    public double getOTIND() {
        return 500 ;
    }
    public int getD1() {
        return 4 ;
    }
    public int getD2() {
        return 1 ;
    }
```

Métodos en SeguroDeHogar:

```
    public int getN() {
        return 3 ;
    }
    public double getOTIND() {
        return 700 ;
    }
    public int getD1() {
        return 5 ;
    }
    public int getD2() {
        return 2 ;
    }
```

Ejercicio 5 **(2 puntos)**. Implementad en la clase Controlador el siguiente método:

`public Map<Integer,Double> mapaAnyoACoste(String nif) throws NoClienteException`

Este método lanza una excepción si no existe ningún cliente con el NIF igual al valor del argumento `nif`.

En caso de que exista tal cliente, el método crea y retorna un mapa. Cada clave de ese mapa es un año. El valor correspondiente a esa clave es el coste de las indemnizaciones pagadas al cliente por la compañía de seguros en ese año por todos los seguros tomados por el cliente.

Para implementar este método NO debéis utilizar el método del ejercicio 3. Se sugiere también que NO uséis el método `containsKey()`; usad en su lugar el método `get()`.

SOLUCIÓN:

```
    public Map<Integer,Double> mapaAnyoACoste(String nif) throws NoClienteException{
        Map<Integer,Double> result = new HashMap<>() ;
        Cliente cli = this.clientes.get(nif) ;
        if(cli==null){
```

```
            throw new NoClienteException("No existe ningún cliente con NIF " + nif) ;
        }
        List<Seguro> seguros = cli.getSeguros() ;
        // Iterar sobre todos los seguros del cliente
        for(Seguro seguro: seguros){
            List<Parte> partes = seguro.getPartes() ;
            // Iterar sobre todos los partes del seguro
            for(Parte parte: partes){
                int anyoParte = Fecha.getAnyoDeFecha(parte.getFechaIncidencia()) ;
                double coste = parte.getCosteIndemnizacion() ;
                Double costeAc = result.get(anyoParte) ;
                // Si no hay pareja, crearla y poner solo el coste
                if(costeAc==null){
                    result.put(anyoParte, coste) ;
                }else{
                    // Si ya había pareja, poner como valor la suma de los costes
                    result.put(anyoParte, coste+costeAc) ;
                }
            }
        }
        return result ;
    }
```

Ejercicio 6 **(2 puntos)**. Implementad en la clase Controlador los siguientes métodos:

6.1 Implementad en la clase Controlador el siguiente método:

`private Parte creaParte(String linea, Seguro seguro) throws FormatoException`

Este método intentará crear un nuevo parte cuyos atributos toman valores a partir del contenido del argumento `línea`. Este método asociará el parte creado con el seguro cuya referencia se pasa en `seguro`.

El contenido de `línea` se ajusta al siguiente patrón formado por varios campos:

<Descripción>:<CosteIndemnizacion>:<FechaIncidencia>:<Informe>

En dicho patrón, lo encerrado entre el carácter '<' y el carácter '>' se usa para indicar que en el la línea aparecerán valores correspondientes a lo encerrado entre ambos caracteres. El carácter ":" actúa como separador de los valores.

Como los nombres indican, el primer campo contiene el valor de la descripción de un parte, el segundo la representación textual del coste de la indemnización de un parte, el tercero la fecha de la incidencia y el cuarto el informe del parte.

En el argumento `linea` el campo <CosteIndemnizacion> puede tener un contenido erróneo (una secuencia de caracteres que NO se corresponden con la representación textual de un número real: "150B", por ejemplo). En tal caso, el método NO crea el parte y lanza una excepción `FormatoException`. **Considerad que los otros 3 campos NO contienen ningún error de formato**.

NOTA 1: Considerad implementado el constructor de Parte cuya cabecera se indica a continuación:

`public Parte(String desc, double coste, String fecha, String info, Seguro s)`

NOTA 2: recordad que para generar un valor double a partir de un String se usa el **método estático** `double parseDouble(String str)` de la clase Double. Recordad también que este método lanza la excepción `NumberFormatException` si el argumento `str` no contiene una representación textual de un número real.

SOLUCIÓN:

```
    private Parte creaParte(String linea, Seguro seguro) throws FormatoException {
        String[] args = linea.split(":");
        double costeInd = 0.0;
        try {
            costeInd = Double.parseDouble(args[1]);
        } catch (NumberFormatException ex) {
            throw new FormatoException("Valor de coste de indemnización de "
                    + "Parte no representa a un double: " + args[1]);
        }
        return new Parte(args[0], costeInd, args[2], args[3], seguro);
    }
```

6.2 Implementad en la clase Controlador el siguiente método:

`public Seguro readSeguro(InputStream is) throws FormatoException;`

Este método lee de un stream (que puede ser el stream gestionado por un `FileInputStream`, por ejemplo) una secuencia de líneas que siguen el patrón que se muestra a continuación:

SEGURO<TIPO-DE-SEGURO>

<FechaDeContrato>:<cuotaAnualBase>:<descuentoORecargo>

<ValorAtributo>

PARTE

<Descripción>:<CosteIndemnizacion>:<FechaIncidencia>:<Informe>

PARTE

<Descripción>:<CosteIndemnizacion>:<FechaIncidencia>:<Informe>

En el patrón, lo encerrado entre el carácter '<' y el carácter '>' se usa para indicar que en el archivo aparecerán valores correspondientes a lo encerrado entre ambos caracteres. El carácter ":" actúa como separador de los valores.

Un archivo correspondiente a un seguro con DOS partes, acabaría con la línea que sigue a la línea que contiene el segundo "PARTE": NO habría líneas en blanco al final del archivo.

Si <TIPO-DE-SEGURO> es "DECOCHE" (es decir, si la primera línea es "SEGURODECOCHE"), el método debe crear y devolver un objeto SeguroDeCoche, y <ValorAtributo> contendrá el valor a asignar al atributo `gamaVehiculo`; si <TIPO-DE-SEGURO> es "DEHOGAR" (es decir, si la primera línea es "SEGURODEHOGAR"), el método debe crear y devolver un objeto SeguroDeHogar y <ValorAtributo> contendrá el valor a asignar al atributo `tipoVivienda`.

El método lanzará una excepción FormatoException si hay algún problema de formato en los valores a asignar a los atributos de SeguroDeCoche, SeguroDeHogar o Parte. Debéis suponer que:

1. La línea que contiene SEGURO<TIPO-DE-SEGURO> siempre está presente y NO tiene errores.
2. Las dos líneas que siguen a la línea SEGURO<TIPO-DE-SEGURO> siempre están presentes y NO tienen errores.
3. La línea PARTE PUEDE estar presente O PUEDE NO ESTARLO. Lo estará si hay partes de incidencias asociados al seguro y no lo estará cuando no los haya.
4. Si la línea que contiene PARTE está presente entonces la línea que le sigue estará presente; si la línea que contiene PARTE NO está presente, tampoco habrá línea que le siga. Cuando la línea de PARTE está presente, la línea que le sigue PUEDE CONTENER EL ERROR mencionado en la especificación del método `getParte()`.

NOTA 1: Utilizad el método `creaParte(…)` que se ha especificado antes, incluso si no lo habéis implementado**.**

NOTA 2: Considerad implementados (**NO TENÉIS QUE IMPLEMENTARLOS**) los siguientes constructores de SeguroDeCoche y SeguroDeHogar respectivamente:

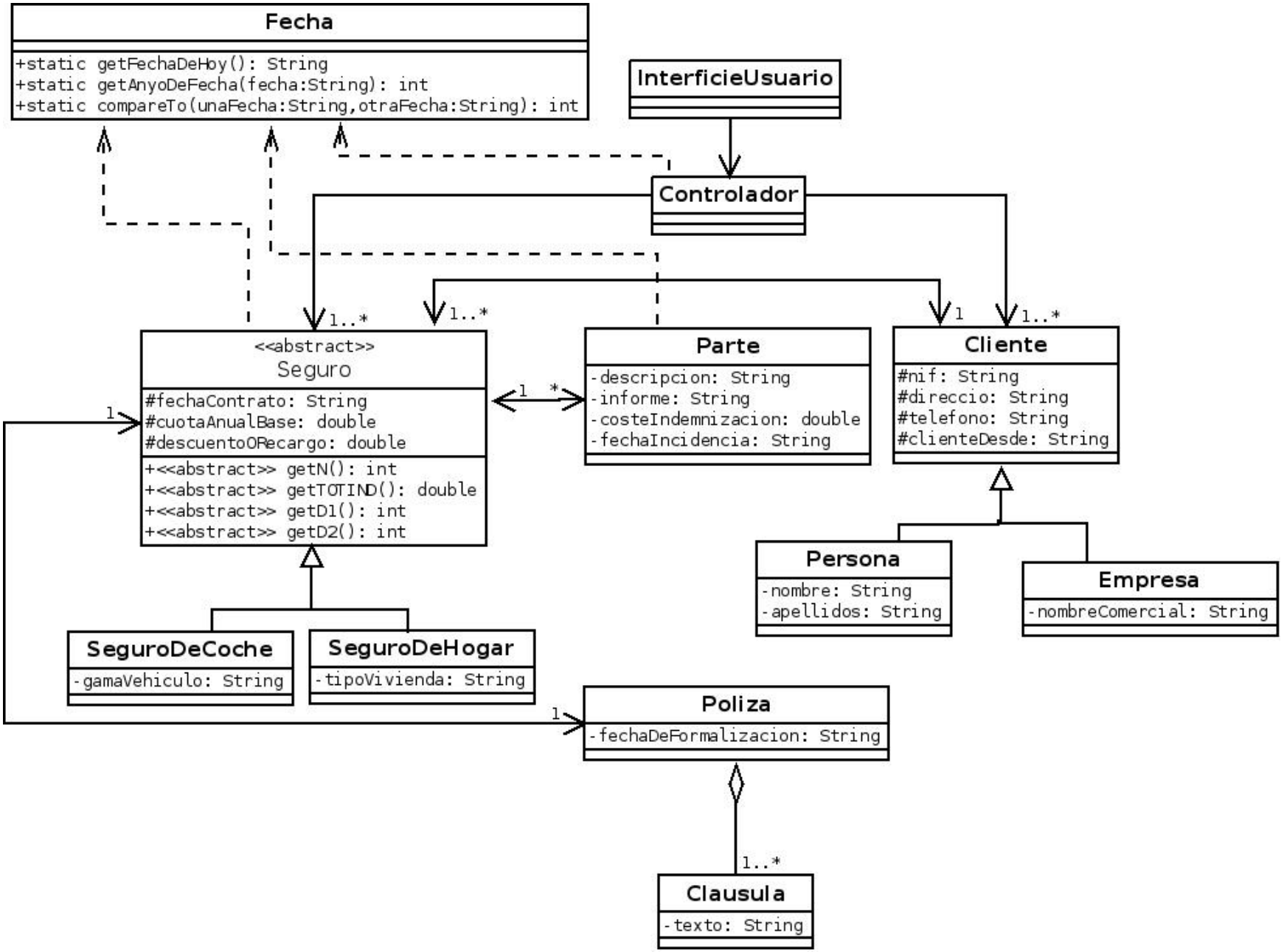
```
public SeguroDeCoche(String gamaVehiculo, String fechaContrato, double
cuotaAnualBase, double descuentoORecargo);


public SeguroDeHogar(String tipoVivienda, String fechaContrato, double
cuotaAnualBase, double descuentoORecargo);
```

NOTA 3: Tened presente que podéis utilizar tantas variables locales como creáis necesarias para depositar en cada una de ellas el contenido de una línea del stream.

SOLUCIÓN:

```
    public Seguro readSeguro(InputStream is) throws FormatoException {
        Scanner sc = new Scanner(is);
        String linea = null;
        String tipoSeguro = null;
        String valAttr = null;
        Seguro seguro = null;
        Parte parte = null;
        String[] args = null;
        // Procesar línea SEGURO<TIPO-DE-SEGURO>: siempre presente sin errores
        tipoSeguro = sc.nextLine();
        // Dos líneas siguientes a SEGURO<TIPO-DE-SEGURO>: presentes y sin errores
        linea = sc.nextLine();
        args = linea.split(":");
        valAttr = sc.nextLine();
        if (tipoSeguro.equalsIgnoreCase("SEGURODECOCHE")) {
            seguro = new SeguroDeCoche(valAttr, args[0], Double.parseDouble(args[1]),
                    Double.parseDouble(args[2]));
        } else {
            seguro = new SeguroDeHogar(valAttr, args[0], Double.parseDouble(args[1]),
                    Double.parseDouble(args[2]));
        }
        //SI HAY PARTES HABRÁ LÍNEAS
        while (sc.hasNextLine()) {
            // LEERÁ Línea de PARTE.
            linea = sc.nextLine();
            // Leerá linea que sigue a PARTE: siempre presente
            linea = sc.nextLine();
            seguro.addParte(this.creaParte(linea, seguro));
        }
        return seguro;
```

```
}
```


Fecha
+static getFechaDeHoy(): String
+static getAnyoDeFecha(fecha:String): int
+static compareTo(unaFecha:String,otraFecha:String): int
InterficieUsuario
Controlador
1..*
1..*
1
1..*
<<abstract>>
Seguro
#fechaContrato: String
#cuotaAnualBase: double
#descuentoORecargo: double
+<<abstract>> getN(): int
+<<abstract>> getTOTIND(): double
+<<abstract>> getD1(): int
+<<abstract>> getD2(): int
1
*
Parte
-descripcion: String
-informe: String
-costeIndemnizacion: double
-fechaIncidencia: String
Cliente
#nif: String
#direccio: String
#telefono: String
#clienteDesde: String
Persona
-nombre: String
-apellidos: String
Empresa
-nombreComercial: String
SeguroDeCoche
-gamaVehiculo: String
SeguroDeHogar
-tipoVivienda: String
1
Poliza
-fechaDeFormalizacion: String
1..*
Clausula
-texto: String