Recomendaciones para respetar SOLID

Violación del principio **SRP(1)**

Dentro de la clase ‘Empleado’ vemos que se tiene un método para actualizar el CSV con el método ‘actualizarCSV(String pathCSV, ArrayList<Empleado> empleadosActualizado)’, esto es una violación del principio de responsabilidad única ya que la clase sólo debería encargarse de las instancias de Empleado y sus métodos propios, más no de actualizar el CSV del sistema.

```
53      public static void actualizarCSV(String pathCSV, ArrayList<Empleado> empleadosActualizado){
54          try(BufferedWriter br = new BufferedWriter(new FileWriter(pathCSV, false))){
55              for (Empleado e : empleadosActualizado){
56                  br.write(e.getNombre()+","+e.getCedula()+","+e.getEmail()+","+e.getEstadoBoolean()+","+e.g
57                  br.newLine();
58                  br.write("");
59              }
60          }catch (IOException ioe){
61              ioe.printStackTrace();
62          }
63      }
```

Además, dentro de la clase contiene otro método llamado ‘cargarEmpleados’, que se encarga de agregar todos los datos del CSV al sistema, una responsabilidad que no debería estar en la clase de Servicio.

```
public static ArrayList<Empleado> cargarEmpleados(String pathEmpleados){
    ArrayList<Empleado> empleados = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(pathEmpleados))) {
        String line;
        while ((line = br.readLine())!= null){
            String[] parametros = line.split(",");
            Empleado empleado = new Empleado(parametros[0],parametros[1],parametros[2],Boolean.parseBoo
            empleados.add(empleado);
        }
    } catch (FileNotFoundException ex) {
        System.out.println("El archivo no existe.");
    } catch (IOException ex) {
        System.out.println("Error IOException:"+ex.getMessage());
    }
    return empleados;
}
```

Solución:

La clase Empleado sólo tiene que tener métodos relacionados a la entidad empleado, y el método de actualizar CSV debería tener su propia clase llamada CSV.

Para separar el cargar los datos de la clase Empleado se debe crear una clase o interfaz para que se encargue de cargar exclusivamente todos los datos al sistema. La clase o interfaz podría llamarse 'DataLoader' que contenga un método llamado 'cargarDatos()' que reciba un archivo cualquiera.

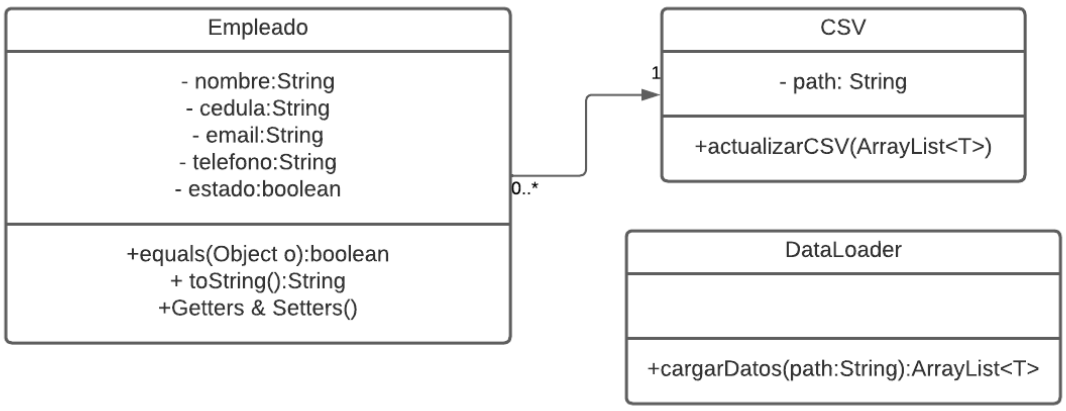
Código:

```
import java.util.ArrayList;


public class CSV<T>{
    private String path = "archivos/path.csv";
    public static <T> void actualizarCSV(ArrayList<T> listaActualizar){
        // Aca se agrega la logica para actualizar el .csv
    }
}

public class DataLoader {
    public static <T> ArrayList<T> cargarDatos(String path){
        ArrayList<T> datosCargados = new ArrayList<>();
        //Lógica para cargar los datos desde un path
        return datosCargados;
    }
}
```

UML:



Violación del principio **SRP(2)**

Dentro de la clase ‘Servicio’ al igual que Empleados vemos que se tiene un método para actualizar el CSV con el método ‘actualizarCSV(String pathCSV, ArrayList<Servicio> serviciosActualizado)’, esto es una violación del principio de responsabilidad única ya que la clase sólo debería encargarse de las instancias de Servicio y sus métodos propios, más no de actualizar el CSV del sistema.

```
public static void actualizarCSV(String pathCSV, ArrayList<Servicio> serviciosActualizado){
    try(BufferedWriter br = new BufferedWriter(new FileWriter(pathCSV, false))){
        for (Servicio s : serviciosActualizado){
            br.write(s.getNombreServicio()+","+s.getDuracion()+","+s.getPrecio()+","+ s.getEstadoBoolea
            br.newLine();
            br.write("");
        }
    }catch (IOException ioe){
        ioe.printStackTrace();
    }
}
```

Además, dentro de la clase ‘Servicio’ contiene otro método llamado ‘cargarServicios’, que se encarga de agregar todos los datos del CSV al sistema, una responsabilidad que no debería estar en la clase de Servicio.

```
public static ArrayList<Servicio> cargarServicios(String ruta){
    ArrayList<Servicio> sv = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(ruta))) {
        String line;
        while ((line = br.readLine())!= null){
            String[] parametros = line.split(",");
            Servicio s = new Servicio(parametros[0], Double.parseDouble(parametros[1]), Double.parseDou
            sv.add(s);
        }
    } catch (FileNotFoundException ex) {
        System.out.println("El archivo no existe.");
    } catch (IOException ex) {
        System.out.println("Error IOException:"+ex.getMessage());
    }
    return sv;
}
```

Solución:

La clase Servicio sólo tiene que tener métodos relacionados a la entidad servicio, y el método de actualizar CSV debería tener su propia clase llamada CSV.

Para separar el cargar los datos de la clase Servicio se debe crear una clase o interfaz para que se encargue de cargar exclusivamente todos los datos al sistema. Se puede utilizar la clase DataLoader creada anteriormente.

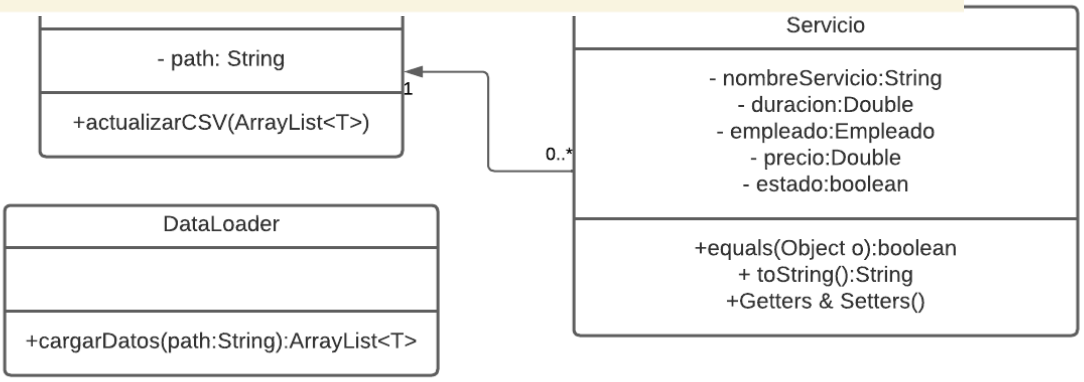
Código:

```java
import java.util.ArrayList;


public class CSV<T>{
    private String path = "archivos/path.csv";
    public static <T> void actualizarCSV(ArrayList<T> listaActualizar){
        // Aca se agrega la logica para actualizar el .csv
    }
}

public class DataLoader {
    public static <T> ArrayList<T> cargarDatos(String path){
        ArrayList<T> datosCargados = new ArrayList<>();
        //Lógica para cargar los datos desde un path
        return datosCargados;
    }
}
```

UML:



Violación del principio **SRP(3)**

Dentro de la clase ‘Atencion’ vemos que se tiene un método para actualizar el Archivo serializado con el método ‘actualizarSER(String pathSER, ArrayList<Atencion> atencionesActualizado)’, esto es una violación del principio de responsabilidad única ya que la clase sólo debería encargarse de las instancias de atenciones y sus métodos propios, más no de actualizar el Serializado del sistema.

```java
public static void actualizarSER(String pathSER, ArrayList<Atencion> atencionesActualizado){
    try(ObjectOutputStream bOS = new ObjectOutputStream(new FileOutputStream(pathSER, false))){
        bOS.writeObject(atencionesActualizado);
    }catch (IOException ioe){
        ioe.printStackTrace();
    }
}
```

Además, al igual que Empleado y Servicio se tiene un método que se encarga de cargar los datos de las atenciones al Sistema, algo que no debería ser responsabilidad de esta clase.

```
public static ArrayList<Atencion> cargarAtenciones(String path) {
    ArrayList<Atencion> atenciones = new ArrayList<>();
    //leer la lista de personas del archivo serializado
    try (ObjectInputStream oi = new ObjectInputStream(new FileInputStream(path))) {
        atenciones = (ArrayList<Atencion>) oi.readObject();
    } catch (FileNotFoundException ex) {
        System.out.println("Archivo no existe");
    } catch (IOException ex) {
        System.out.println("Error IO: "+ex.getMessage());
    } catch (ClassNotFoundException  ex) {
        System.out.println("Error class: "+ex.getMessage());
    }
    return atenciones;
}
```

Solución:

La clase Atencion sólo tiene que tener métodos relacionados a la entidad servicio, y el método de actualizar SER debería tener su propia clase llamada SER que contenga un método para serializar cualquier objeto que se le mande.

Para separar el cargar los datos de la clase Atencion se debe crear una clase o interfaz para que se encargue de cargar exclusivamente todos los datos al sistema. Se puede utilizar la clase DataLoader creada anteriormente.

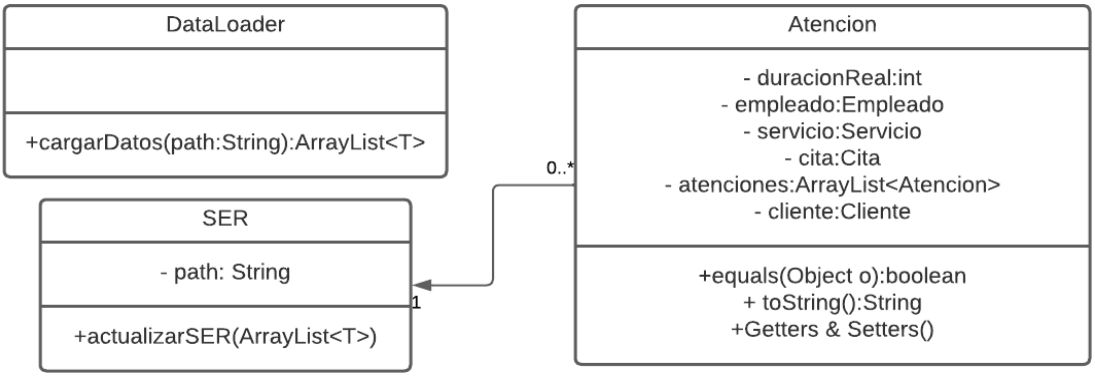
Código:

```
import java.util.ArrayList;

public class SER {
    private String path = "archivos/path.ser";
    public static <T> void actualizarSER(ArrayList<T> listaActualizar){
        // Aca se agrega la logica para actualizar el .ser
    }
}

public class DataLoader {
    public static <T> ArrayList<T> cargarDatos(String path){
        ArrayList<T> datosCargados = new ArrayList<>();
        //Lógica para cargar los datos desde un path
        return datosCargados;
    }
}
```

UML:



Violación del principio **OCP(1)**

En la clase del controlador del Juego llamado ‘JuegoController’ existe el método para escoger una imagen llamada ‘escogerImagen(int i)’ donde se pasa un número aleatorio entre el 1 y el 4 para seleccionar entre las imágenes.

Pero esto es una violación al principio abierto/cerrado debido a que si más adelante se quiere agregar otra imagen se tendrá que modificar este método ya creado en lugar de simplemente extenderlo.

```
public void escogerImagen(int i){
    if(i == 0){
        this.imagenElegida = this.pathGato;
    }
    if(i == 1){
        this.imagenElegida = this.pathPerro;
    }
    if(i == 2){
        this.imagenElegida = this.pathZorro;
    }
    if(i == 3){
        this.imagenElegida = this.pathPato;
    }
}
```

Solución:

Se podría crear una clase que contenga una lista con todos los paths de las imágenes del videojuego, todo esto implementado por una interfaz Provider, que contiene el método para proveer los paths, así cuando el programador quiera agregar otra imagen sólo deberá agregarla a la clase mediante otro

método llamado ‘agregarImagen()’. Esta misma clase también contenga un método ‘seleccionarImagenAleatorio()’ que devuelva un path de imagen de manera aleatoria.

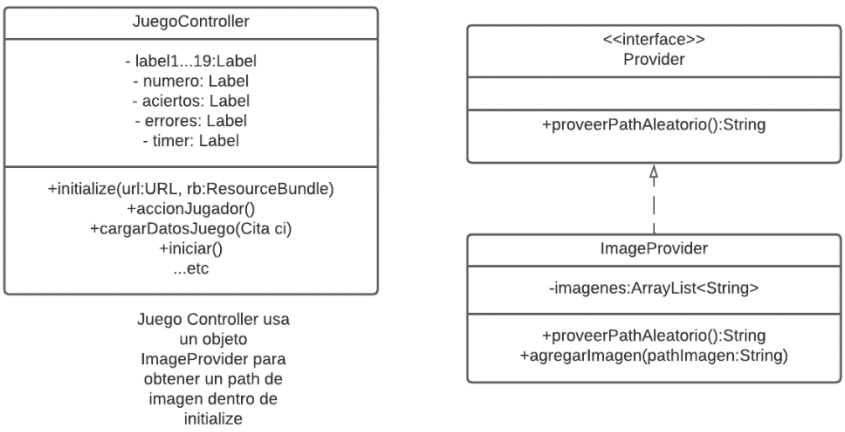
Código de la posible solución:

```
public interface Provider {
    public String proveerPathAleatorio();
}

public class ImageProvider implements Provider {
    private ArrayList<String> imagenes = new ArrayList<String>();

    @Override
    public String proveerPathAleatorio(){
        String pathImagenSeleccionado;
        Random rand = new Random();
        int indiceRamdon = rand.nextInt(imagenes.size()-1);
        pathImagenSeleccionado = imagenes.get(index: indiceRamdon);
        return pathImagenSeleccionado;
    }
    public void agregarImagen(String pathImagen){
        imagenes.add(e: pathImagen);
    }
}
```

UML:



Violación del principio **DIP(1)**

Dentro de la clase ‘Cliente’ se tiene un método llamado ‘cargarClientes()’ además de no cumplir con SRP, no está dependiendo de una abstracción si no de una implementación concreta via un archivo .txt.

```java
public static ArrayList<Cliente> cargarClientes(String pathClientes) {

    ArrayList<Cliente> clientes = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(pathClientes))) {
        String linea;
        while ((linea = br.readLine()) != null) {
            String[] parametros = linea.split(",");
            Cliente cliente = new Cliente(parametros[0], parametros[1], parametros[2], parametros[3], p
            clientes.add(cliente);
        }
    } catch (FileNotFoundException ex) {
        ex.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return clientes;
}
```

Solución:

Para solucionar este inconveniente, podemos tener una interfaz llamada DataLoader con el método cargarDatos y que tenga una clase que sea específicamente para cargarDatosTXT implementando cargarDatos.

Código:

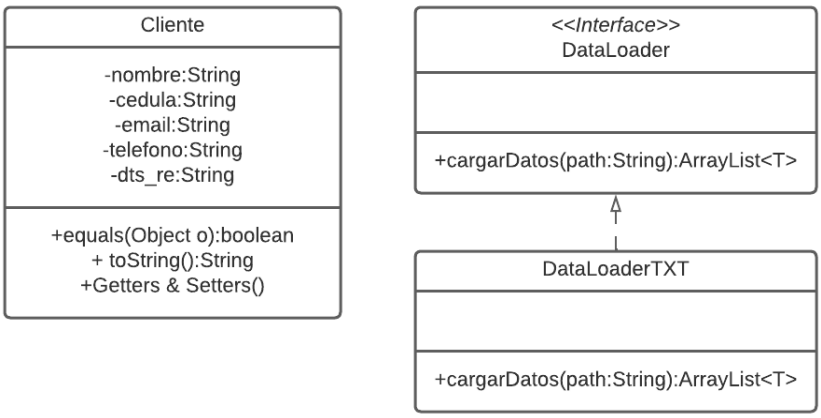
```java
import java.util.ArrayList;

public interface DataLoader<T> {
    public ArrayList<T> cargarDatos(String path);
}

public class CargarDatosTXT<T> implements DataLoader<T> {

    @Override
    public ArrayList<T> cargarDatos(String path){
        ArrayList<T> datosCargados = new ArrayList<>();
        //Lógica para cargar los datos desde un path via .txt
        // datosCargados = ...
        return datosCargados;
    }
}
```

UML:



Violación del principio **DIP(2)**

Dentro de la clase 'Atencion' se tiene un método llamado 'cargarAtenciones()' además de no cumplir con SRP, no está dependiendo de una abstracción si no de una implementación concreta vía un archivo serializado.

Esto viola el principio de inversión de dependencias debido a que pueda que algún día cambiemos la forma en la que guardamos la información de la empresa, y tendríamos que rescribir todo el código de 'cargarServicios' si se llega a refactorizar.

```
public static ArrayList<Atencion> cargarAtenciones(String path) {
    ArrayList<Atencion> atenciones = new ArrayList<>();
    //leer la lista de personas del archivo serializado
    try (ObjectInputStream oi = new ObjectInputStream(new FileInputStream(path))) {
        atenciones = (ArrayList<Atencion>) oi.readObject();
    } catch (FileNotFoundException ex) {
        System.out.println("Archivo no existe");
    } catch (IOException ex) {
        System.out.println("Error IO: "+ex.getMessage());
    } catch (ClassNotFoundException  ex) {
        System.out.println("Error class: "+ex.getMessage());
    }
    return atenciones;
}
```

Solución:

De igual manera que en la clase Cliente, podemos tener una interfaz llamada DataLoader con el método cargarDatos y que tenga una clase que sea específicamente para cargarDatosSER implementando cargarDatos pero sólo para archivos serializados.

Código:

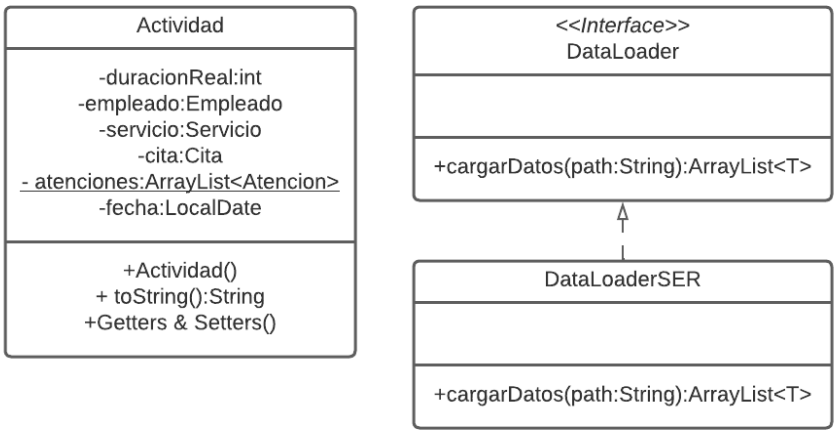
```java
import java.util.ArrayList;

public interface DataLoader<T> {
    public ArrayList<T> cargarDatos(String path);
}

import java.util.ArrayList;

public class CargarDatosSER<T> implements DataLoader<T>{
    @Override
    public ArrayList<T> cargarDatos(String path){
        ArrayList<T> datosCargados = new ArrayList<>();
        //Lógica para cargar los datos desde un path via .ser
        // datosCargados = ...
        return datosCargados;
    }
}
```

UML:



Violación del principio **DIP(3)**

Dentro de la clase 'Cita' tenemos un método para actualizar el archivo serializado con la lista de citas serializadas, pero esto viola además de SRP, el principio de inversión de dependencias debido a que dependemos de un archivo serializado para actualizar los datos del sistema.

```java
public static void actualizarSER(String pathSER, ArrayList<Cita> citasActualizado){
    try(ObjectOutputStream bOS = new ObjectOutputStream(new FileOutputStream(pathSER, false))){
        bOS.writeObject(citasActualizado);
    }catch (IOException ioe){
        ioe.printStackTrace();
    }
}
```

Solución:

Podemos hacer una interfaz llamada ‘DataUpdater’ que tenga como método update(ArrayList<T> datos), y podemos utilizar una clase especializada ‘DataUpdaterSER’ para guardar los datos vía archivo serializado.

Código:

```java
import java.util.ArrayList;

public interface DataUpdater<T> {
    public void update(ArrayList<T> datos);
}
```

```java
import java.util.ArrayList;

public class DataUpdaterSER implements DataUpdater{

    @Override
    public void update(ArrayList datos) {
        //Logica para guardar los datos en un archivo .ser
    }

}
```

UML: