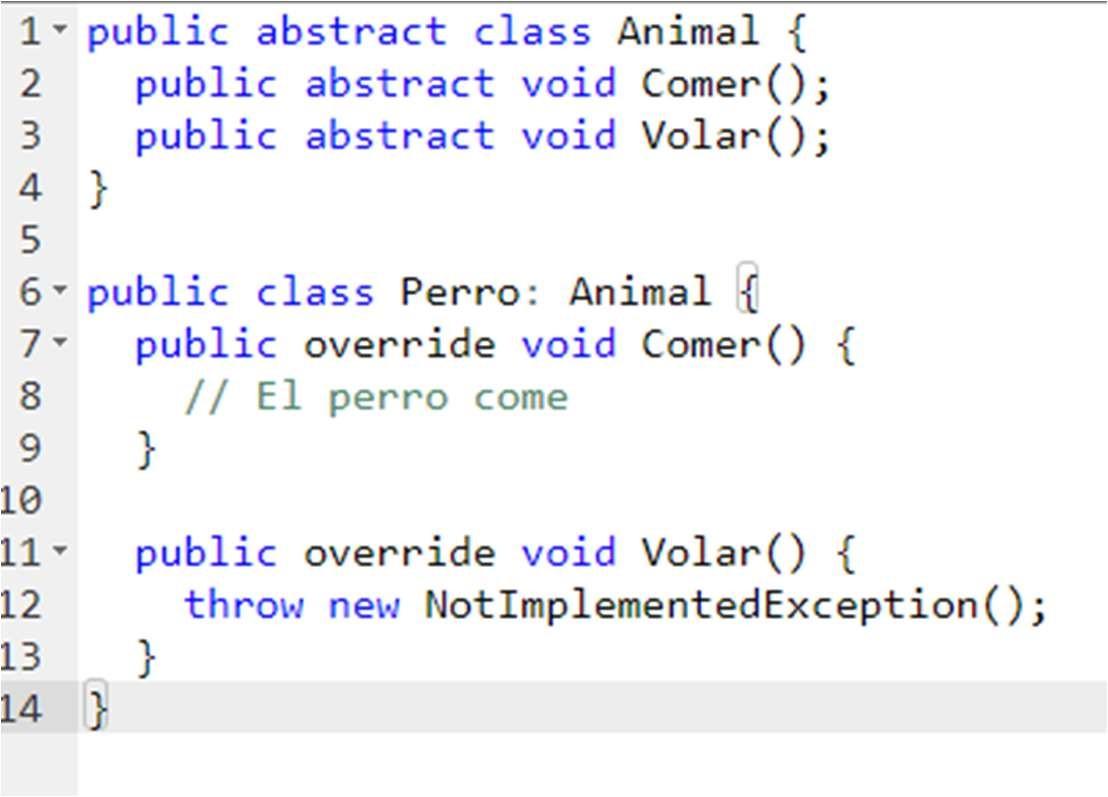
# EJERCICIO 1



En este caso, se está violando el **Principio de sustitución de Liskov,** debido a que Perro no puede sustituir a la clase padre Animal, al no tener implementado el método Volar().

Para solucionarlo, se podría remover la clase abstracta Animal y crear dos interfaces IComer e IVolar, en las que estén los métodos que corresponden a cada una, para que luego la clase Perro implemente la interfaz IComer y defina la lógica del método Comer().

public interface IComer {

*void* Comer();

}

public interface IVolar {

*void* Volar();

}

public class Perro implements *IComer* {

public *void* Comer() {

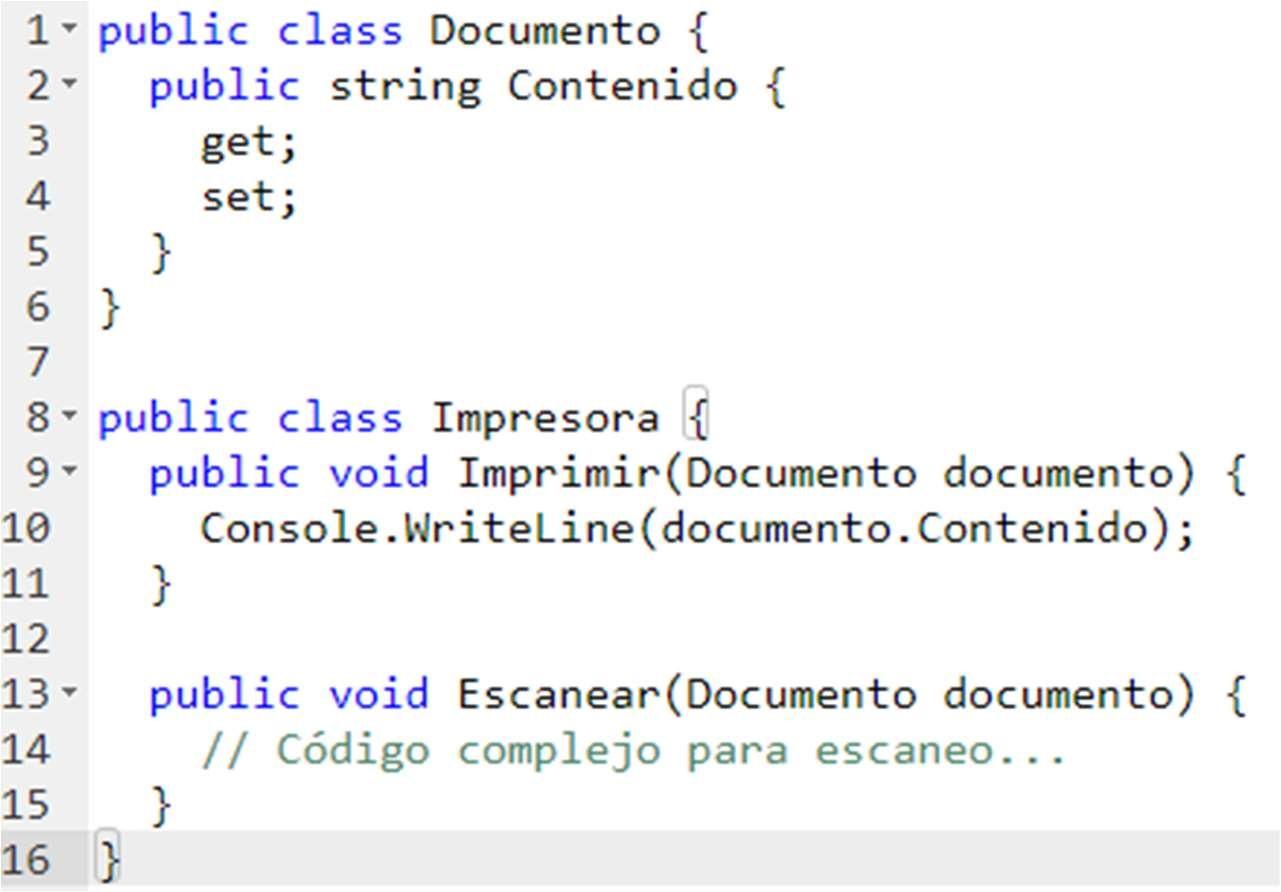
//La logica de comer para el perro en especifico

}

}

# 

# EJERCICIO 2



En este ejercicio, también se está violando **DIP** debido a que la clase Impresora está dependiendo de una clase concreta, que es Documento, en lugar de depender de abstracciones, que es lo que se estipula en el principio nombrado.

public class Documento: IImprimible, IEscaneable {}

public interface IImprimible {

*string* Contenido {

get;

set;

}

}

public interface IEscaneable {

*string* Contenido {

get;

set;

}

}

public class Impresora {

public *void* DocumentoEscaneado(IEscaneable documento) {

// Código para escaneo

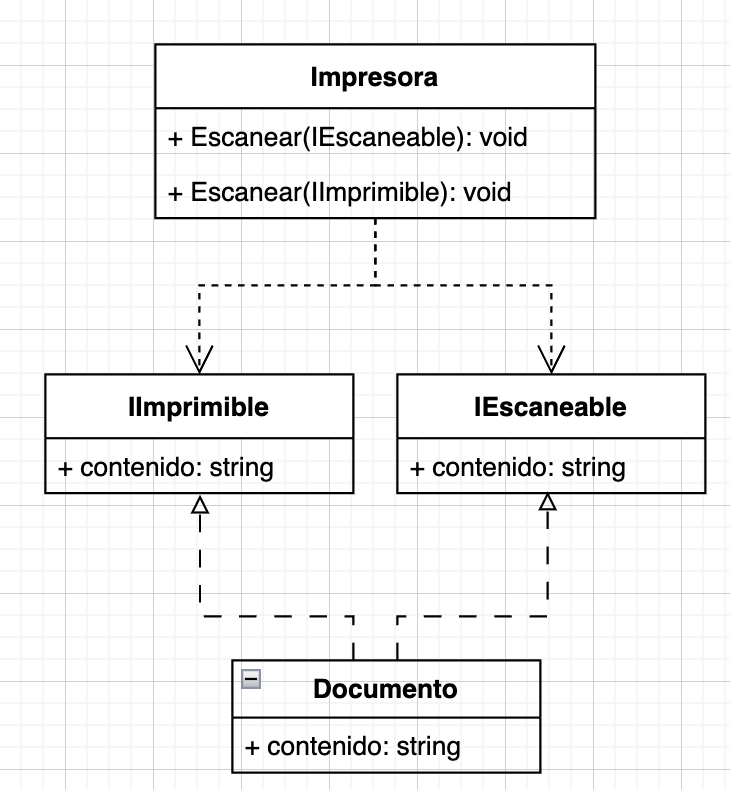
}

public *void* ImprimirDocumento(*IImprimible* *documento*) {

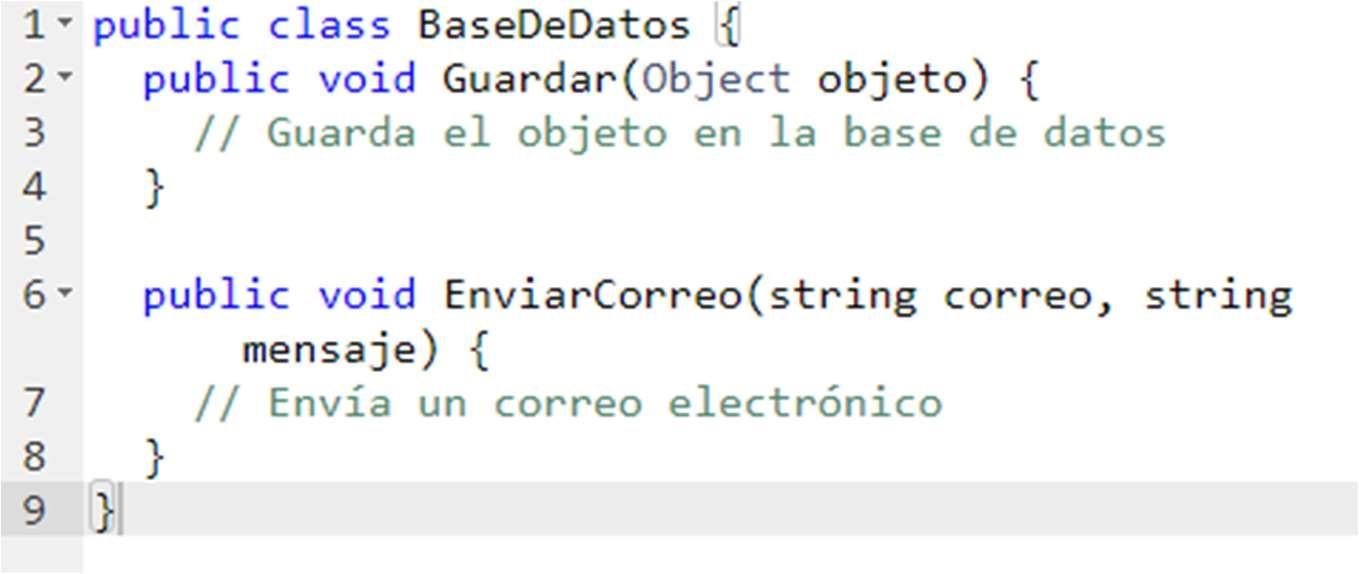
Console.WriteLine(documento.Contenido);

}

}



# EJERCICIO 3



En este caso, se está violando el principio **SRP**, debido a que la clase BaseDeDatos tiene más de una razón para cambiar, las cuales podrían ser un cambio en el método Guardar o EnviarCorreo.

public class BaseDeDatos {

public *void* Guardar(*Object* *objeto*) {

// Guardar objeto en la base de datos

}

}

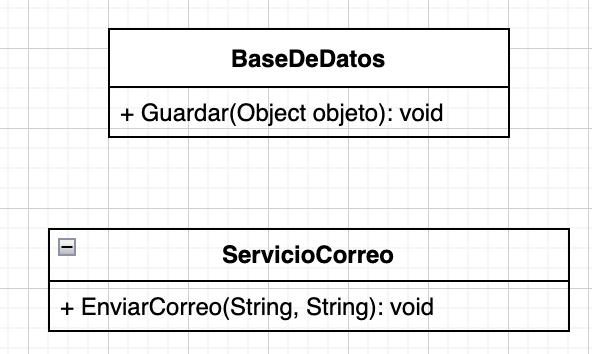
public class ServicioCorreo {

public *void* EnviarCorreo(*String* *correo*, *String* *mensaje*,) {

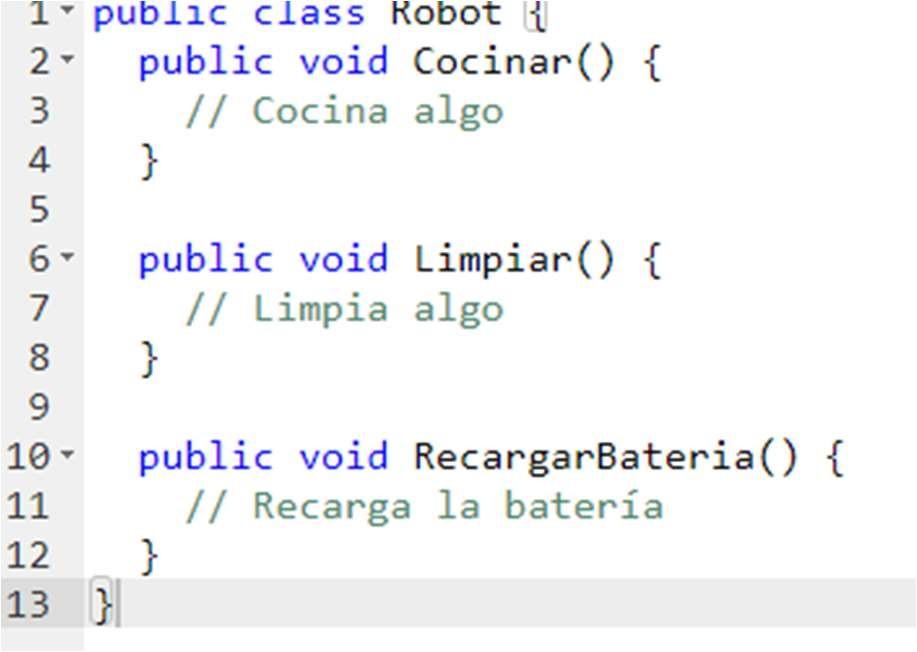
// Enviar correo

}

}



# EJERCICIO 4



En este caso, se está violando el principio **SRP**, debido a que la clase Robot tiene más de una razón para cambiar, las cuales podrían ser un cambio en los métodos Cocinar, Limpiar o RecargarBateria.

public interface *ICocinar* {

public *void* cocinar();

}

public interface *ILimpiar* {

public *void* limpiar();

}

public interface *IRecargarBateria* {

public *void* recargarBateria();

}

public class Robot: *ICocinar*, *ILimpiar*, *IRecargarBateria* {

public *void* cocinar() {

Console.WriteLine("Cocinando...");

}

public *void* limpiar() {

Console.WriteLine("Limpiando...");

}

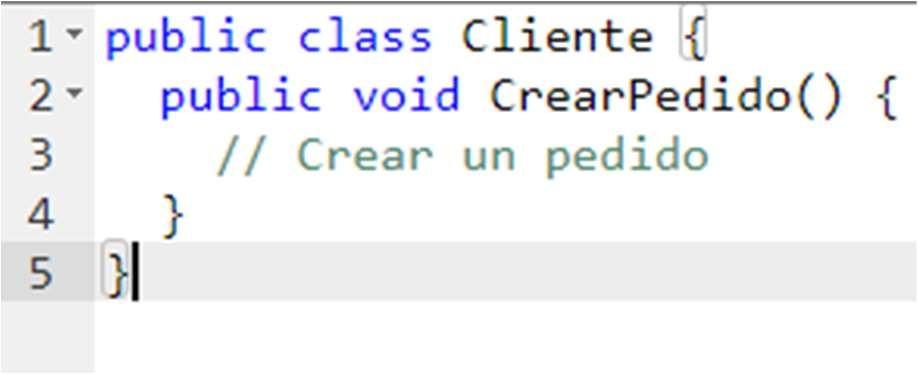
public *void* recargarBateria() {

Console.WriteLine("Recargando bateria...");

}

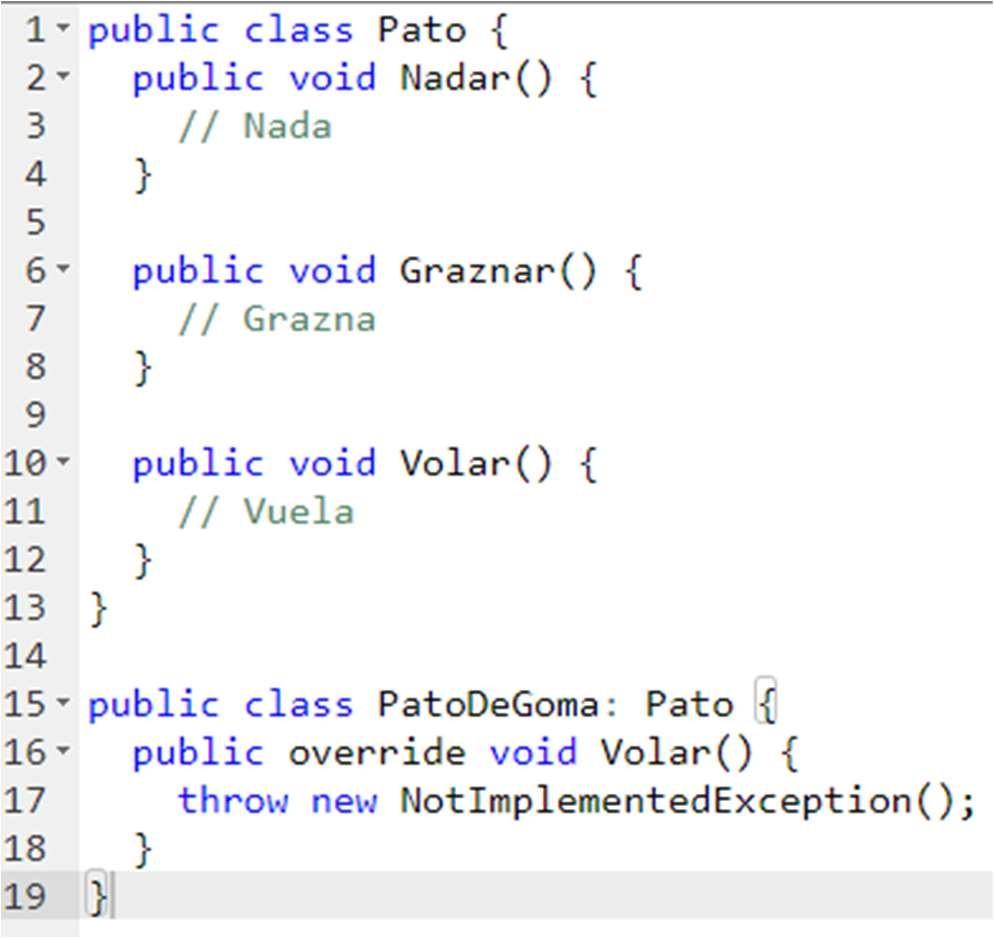
}

# EJERCICIO 5



Esta clase está excelente. Sin embargo, el método CrearPedido() no está alineado con los otros métodos que podría tener la clase Cliente, este tendrá más de una razón de cambio. Incumpliendo **SRP**.

# EJERCICIO 6



En este caso, se está violando el **Principio de sustitución de Liskov**, debido a que PatoDeGoma no puede sustituir a la clase padre Pato, al no tener implementado el método Volar().

Para solucionarlo, se podría remover la clase abstracta Pato y crear tres interfaces INadar, IGraznar e IVolar, en las que estén los métodos que corresponden a cada una, para que luego la clase PatoDeGoma implemente la interfaz INadar, IGraznar y defina la lógica de los métodos.

public interface INadar {

public *void* nadar();

}

public interface IGraznar {

public *void* graznar();

}

public interface IVolar {

public *void* volar();

}

public class PatoDeGoma implements *INadar*, *IGraznar*{

public *void* nadar() {

System.out.println("Nadando");

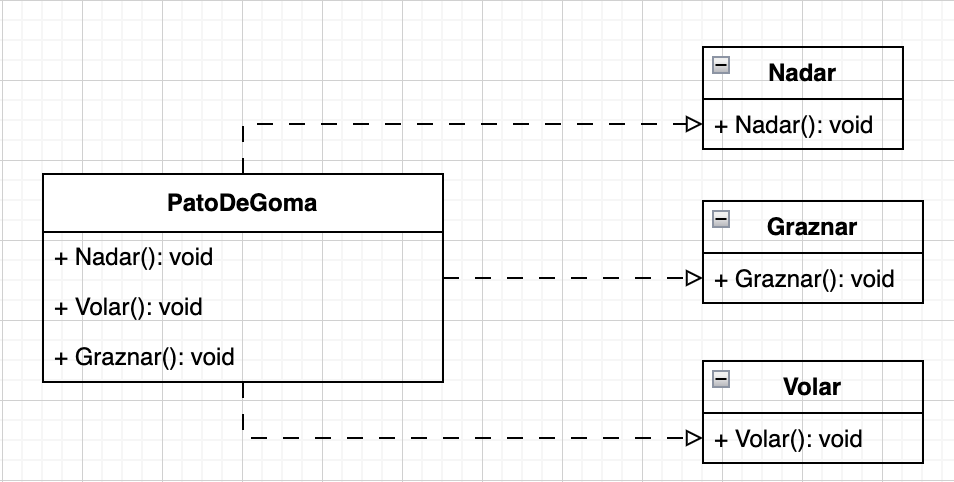
}

public *void* graznar() {

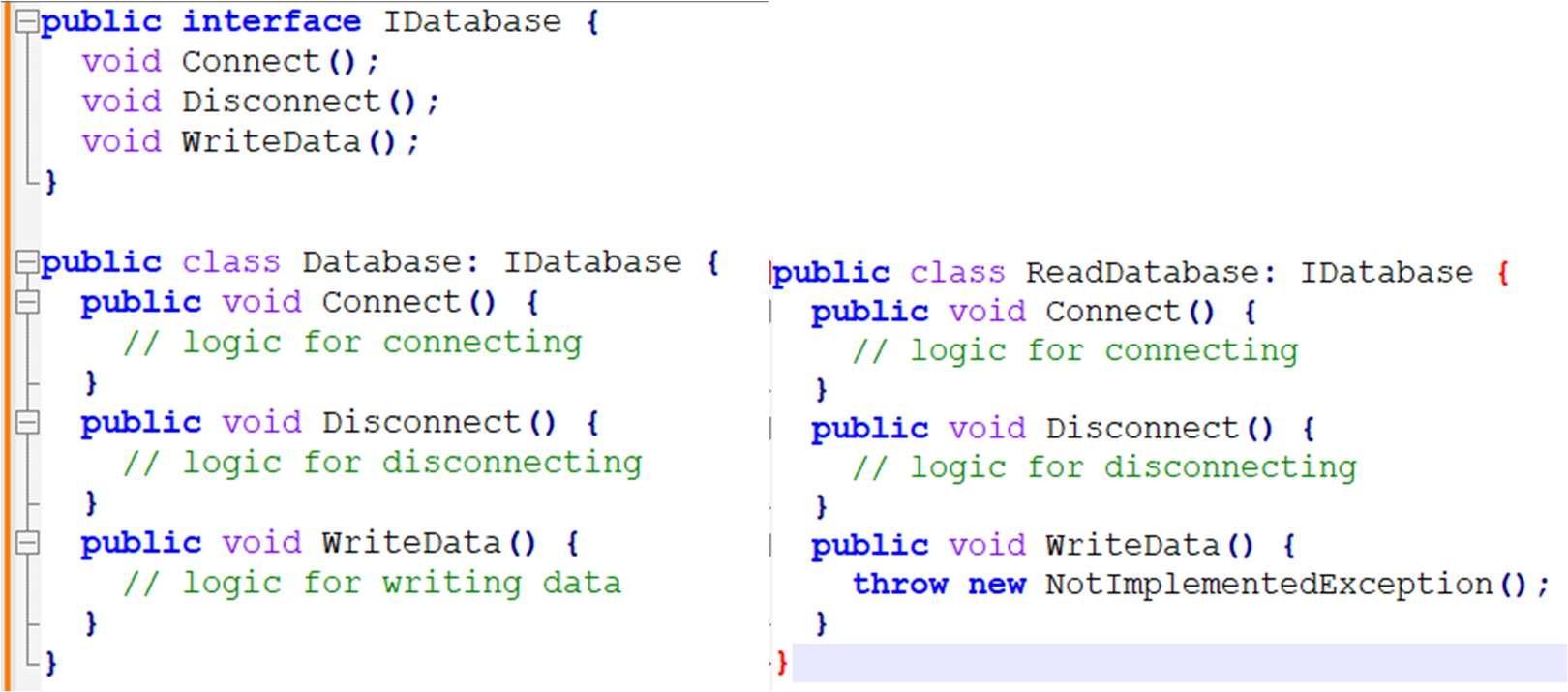
System.out.println("Graznando");

}

}



# EJERCICIO 7



En el código dado, se está violando el principio **ISP** debido a que la clase ReadDatabase no define el método WriteData que se nombra en la interfaz que esta implementa, IDatabase.

Para solucionarlo, es mejor crear varias interfaces específicas en las que estén los métodos que se quieran definir al implementarlas: IConnect y IWriteData.

Luego, las clases implementarían solo las interfaces que le correspondan.

public interface IConnect {

public *void* connect();

public *void* disconnect();

}

public interface IWriteData {

public *void* writeData(*String* *data*);

}

public class DataBase implements *IConnect*, *IWriteData* {

@*Override*

public *void* connect() {

System.out.println("Connect");

}

@*Override*

public *void* disconnect() {

System.out.println("Disconnect");

}

@*Override*

public *void* writeData(*String* *data*) {

System.out.println("Write data");

}

}

public class ReadDataBase implements *IConnect* {

@*Override*

public *void* connect() {

System.out.println("Connect");

}

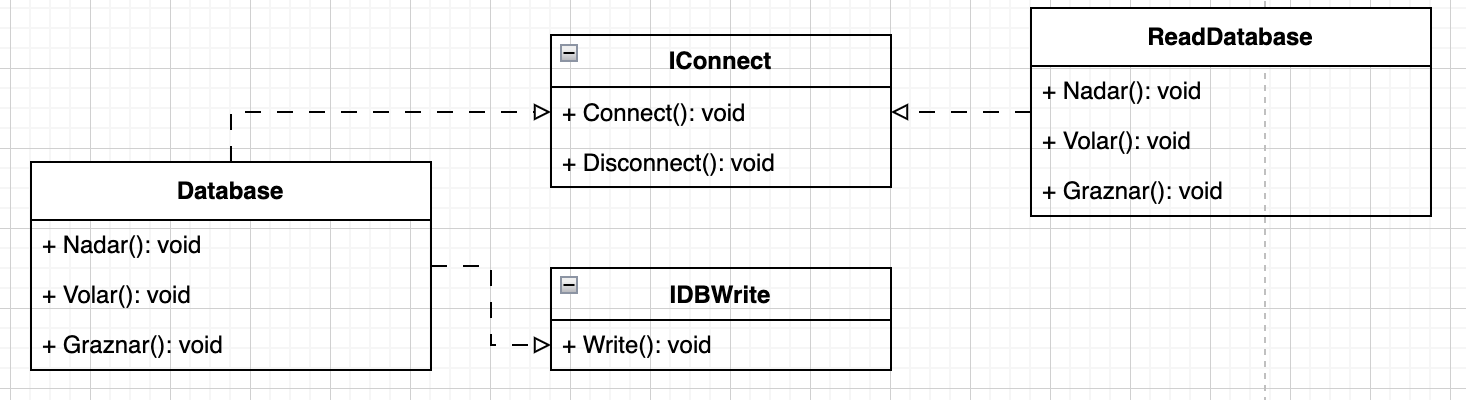
@*Override*

public *void* disconnect() {

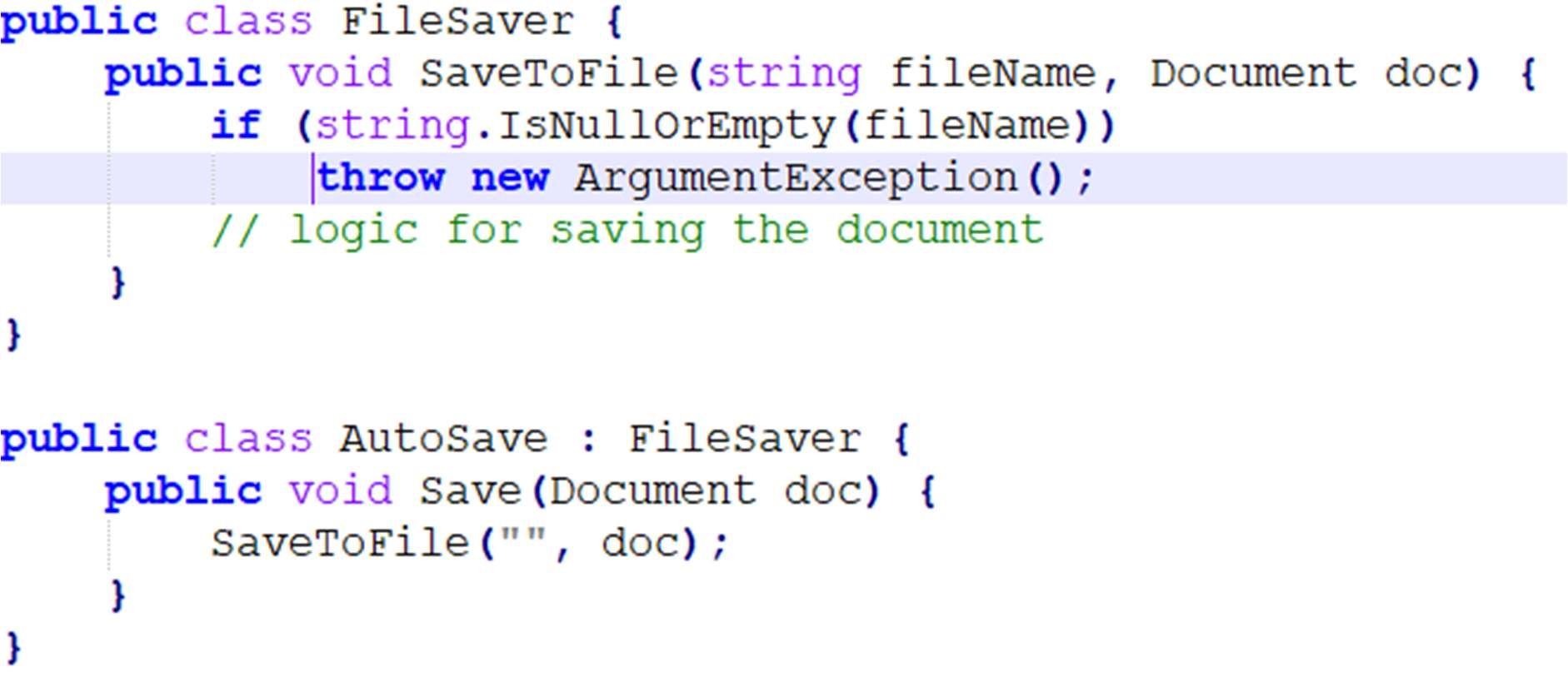
System.out.println("Disconnect");

}

}



# EJERCICIO 8



**SRP**: El principio establece que una clase debe tener una única razón para cambiar. En el código proporcionado, la clase FileSaver tiene la responsabilidad de guardar un documento en un archivo, pero también realiza la validación del nombre del archivo. Podría ser más adecuado mover la validación del nombre del archivo a otra clase o método separado, lo que permitiría que FileSaver se centre únicamente en la tarea de guardar el documento.

public class FileSaver {

private bool ValidateFileName(string filename) {

return !string.IsNullOrEmpty(filename);

}

public void SaveToFile(string filename, Document doc) {

if (!ValidateFileName(filename)) {

throw new ArgumentException("Invalid filename");

}

// Lógica para guardar el documento en un archivo

}

}

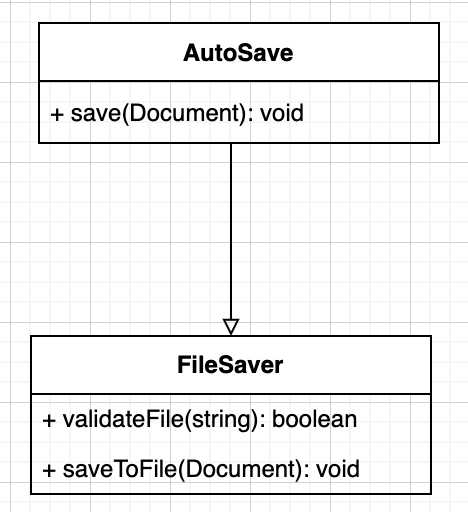
public class AutoSave: FileSaver {

public void Save(Document doc) {

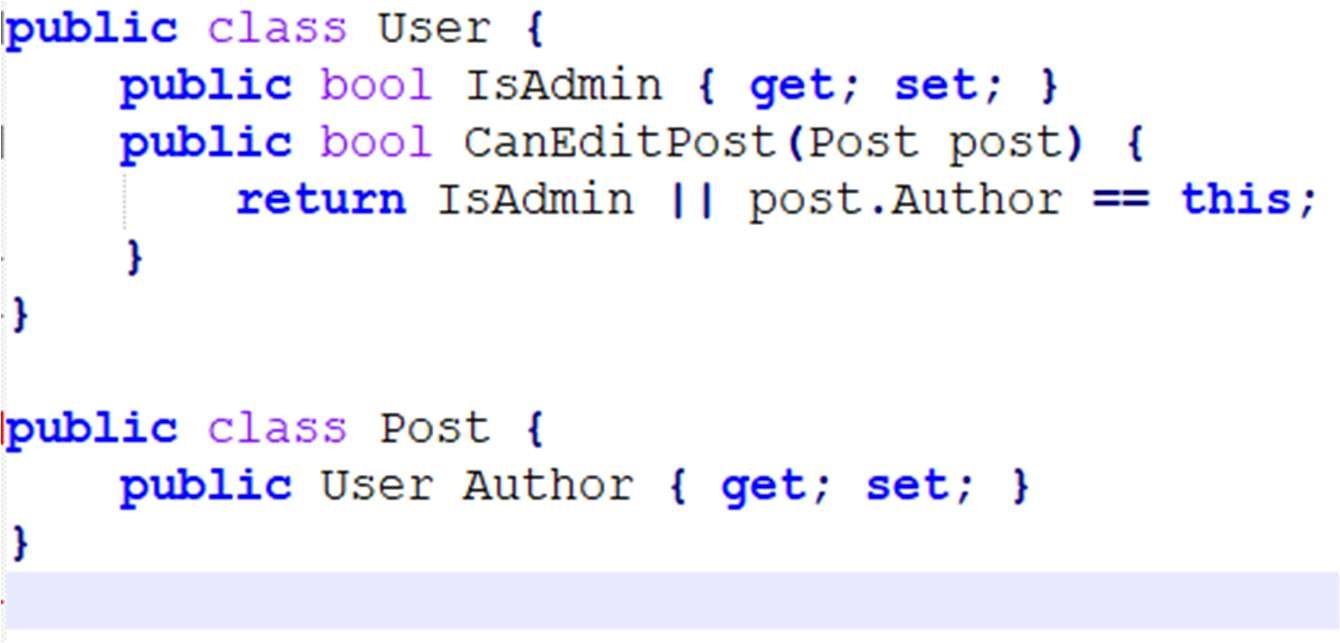
SaveToFile("", doc);

}

}



# EJERCICIO 9



El **Principio de Responsabilidad Única** establece que una clase debe tener una única razón para cambiar. En el caso de la clase User, se le ha asignado la responsabilidad de determinar si un usuario puede editar una publicación. Sin embargo, también tiene una propiedad IsAdmin que indica si el usuario es administrador.

public class User {

public bool IsAdmin { get; set; }

}

public class Post {

public User Author { get; set; }

}

public class PostEditor {

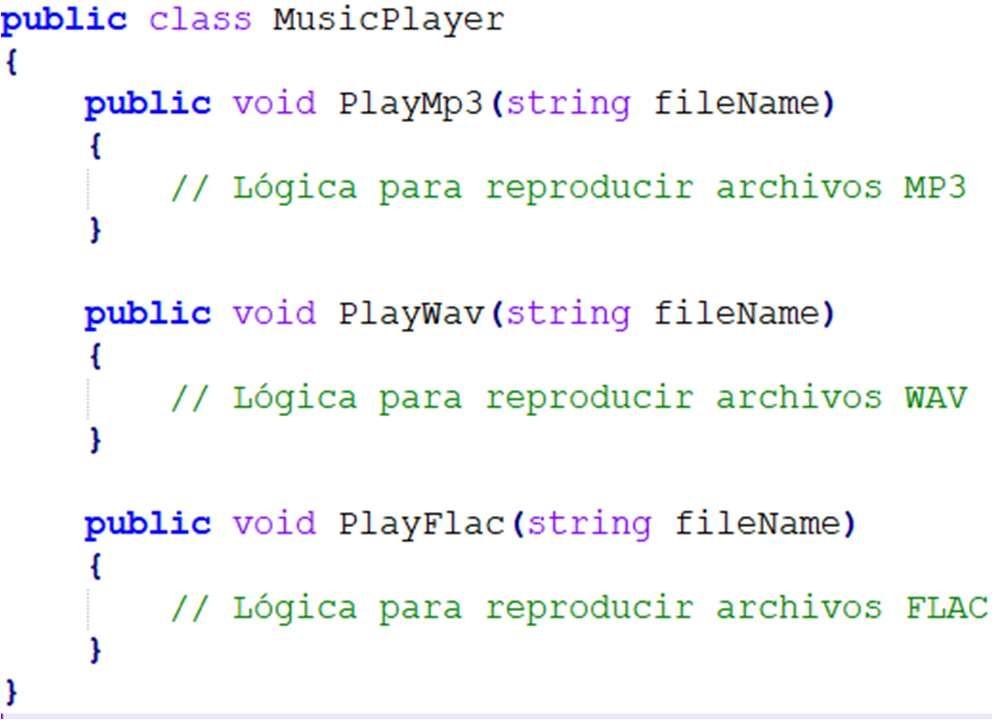
public bool CanEditPost(User user, Post post) {

return user.IsAdmin || post.Author == user;

}

}

# EJERCICIO 10



La clase Music Player incumple con **OCP** porque si se desea agregar un nuevo archivo para reproducir, se debe modificar la clase MusicPlayer agregando otro método especial para reproducir este archivo.

public interface IMusicPlayer {

void Play(string fileName);

}

public class Mp3Player implements IMusicPlayer {

public void Play(string fileName) {

// Lógica para reproducir archivos MP3

}

}

public class WavPlayer implements IMusicPlayer {

public void Play(string fileName) {

// Lógica para reproducir archivos WAV

}

}

public class FlacPlayer implements IMusicPlayer {

public void Play(string fileName) {

// Lógica para reproducir archivos FLAC

}

}

public class PlayMusic {

public void Play(IMusicPlayer musicPlayer) {

musicPlayer.Play();

}

}