1. Fale sobre os princípios SOLID e forneça um exemplo prático de como cada princípio pode ser aplicado em uma aplicação .NET.

R.: Os Princípios de SOLID é dividido em 5 princípios que cada letra representa um princípio.

Por exemplo:

- a letra S representa Single Responsibility Principle (Princípio de Responsabilidade Única).
- a letra O representa Open Closed Principle (Princípio do aberto fechado)
- a letra L representa Liskov Substitution Principle (Princípio da Substituição de Liskov)
- a letra I representa Interface Segregation Principle (Princípio da Segregação de Interfaces)
- e a letra D que representa Dependency Inversion Principle (Princípio da Inversão de Dependência)

O Princípio de Responsabilidade Única é "uma classe deve ter apenas um motivo para mudar", este texto é tradução de uma frase de Robert C. Martin,

que representa, uma classe não pode conter muitas responsabilidades. Segue um exemplo de violação do princípio:

```
namespace SOLID.SRP.Violacao
{
    public class Cliente
    {
        public int ClienteId {get;set;}
        public string Nome {get;set;}
        public string Email {get;set;}
        public string CPF {get;set;}
        public DateTime DataCadastro {get;set;}

        public string AdicionarCliente()
        {
            if (!Email.Contains("@"))
                return "Cliente com e-mail inválido";
```

```
                if (CPF.Length != 11)
                        return "Cliente com CPF inválido";

                using(var cn = new SqlConnection())
                {
                        var cmd = new SqlCommand();

                        cn.ConnectionString = "MinhaConnectionString";
                        cmd.Connection = cn;
                        cmd.CommandType = CommandType.Text;
                        cmd.CommandText = "INSERT INTO CLIENTE (NOME,
EMAIL, CPF, DATACADASTRO) VALUES (@nome, @email, @cpf, @dataCad))";

                        cmd.Parameters.AddWithValue("nome", Nome);
                        cmd.Parameters.AddWithValue("email", Email);
                        cmd.Parameters.AddWithValue("cpf", CPF);
                        cmd.Parameters.AddWithValue("dataCad", DataCadastro);

                        cn.Open();
                        cmd.ExecuteNonQuery();
                }

                var mail = new MailMessage("empresa@empresa.com", Email);
                var client = new SmtpClient
                {
                        Port = 25,
                        DeliveryMethod = SmtpDeliveryMethod.Network,
                        UseDefaultCredentials = false,
                        Host = "smtp.google.com"
                };
```

```
                mail.Subject = "Bem-vindo.";
                mail.Body = "Parabéns! Você está cadastrado.";
                client.Send(mail);

                return "Cliente cadastrado com sucesso!";
            }

        }
}
```

Uma solução do princípio em NET, seria separar a classe cliente das suas responsabilidades: de conectar no banco de dados, de enviar e-mail, entre outras responsabilidades.

```
namespace SOLID.SRP.Solucao
{
    public class Cliente
    {
        public int ClienteId {get;set;}
        public string Nome {get;set;}
        public Email Email {get;set;}
        public CPF Cpf {get;set;}
        public DateTime DataCadastro {get;set;}

        public bool Validar()
        {
            return Email.Validar() && Cpf.Validar();
        }
    }
}
```

```
namespace SOLID.SRP.Solucao
{
	public class CPF
	{
		public string Numero{get;set;}

		public bool Validar()
		{
			return Numero.Length == 11;
		}
	}
}

namespace SOLID.SRP.Solucao
{
	public class Email
	{
		public string Endereco{get;set;}

		public bool Validar()
		{
			return Endereco.Contains("@");
		}
	}
}

namespace SOLID.SRP.Solucao
{
	public class ClienteRepository
	{
		public void AdicionarCliente(Cliente cliente)
```

```
                {
                        using(var cn = SqlConnection())
                        {
                                var cmd = new SqlCommand();


                                cn.ConnectionString = "MinhaConnectionString";
                                cmd.Connection = cn;
                                cmd.CommandType = CommandType.Text;
                                cmd.CommandText = "INSERT INTO CLIENTE (NOME,
EMAIL, CPF, DATACADASTRO) VALUES (@nome, @email, @cpf, @dataCad))";


                                cmd.Parameters.AddWithValue("nome", cliente.Nome);
                                cmd.Parameters.AddWithValue("email", cliente.Email);
                                cmd.Parameters.AddWithValue("cpf", cliente.Cpf);
                                cmd.Parameters.AddWithValue("dataCad",
cliente.DataCadastro);


                                cn.Open();
                                cmd.ExecuteNonQuery();
                        }
                }
        }
}


namespace SOLID.SRP.Solucao
{
        public class ClienteService
        {
                public string AdicionarCliente(Cliente cliente)
                {
                        if(!cliente.Validar())
                                return "Dados inválidos";
```

```
                var repo = new ClienteRepository();
                repo.AdicionarCliente(cliente);

                EmailServices.Enviar("empresa@empresa.com",
cliente.Email.Endereco, "Bem Vindo", "Parabéns está Cadastrado");

                return "Cliente cadastrado com sucesso";
            }
        }
}

namespace SOLID.SRP.Solucao
{
    public static class EmailServices
    {
        public static void Enviar(string de, string para, string assunto, string
mensagem)
        {
            var mail = new MailMessage(de, para);
            var client = new SmtpClient
            {
                Port = 25,
                DeliveryMethod = SmtpDeliveryMethod.Network,
                UseDefaultCredentials = false,
                Host = "smtp.google.com"
            };

            mail.Subject = assunto;
            mail.Body = mensagem;
            client.Send(mail);
        }
```

```
        }
}
```

O Princípio de aberto fechado representa que uma classe deve estar aberta para extensão e fechada para modificação.

Segue uma violação desse princípio em NET:

```
namespace SOLID.OCP.Violacao
{
        public class DebitoConta
        {
                public void Debitar(decimal valor, string conta, TipoConta tipoConta)
                {
                        if(tipoConta == TipoConta.Corrente)
                        {
                                //Debito Conta Corrente
                        }

                        if(tipoConta == TipoConta.Poupanca)
                        {
                                // Valida Aniversário da Conta
                                // Debita Conta Poupanca
                        }
                }
        }
}
```

Segue a abaixo a solução:

```
namespace SOLID.OCP.Solucao
{
```

```
        public abstract class DebitoConta
        {
                public string NumeroTransacao {get;set;}
                public abstract string Debitar(decimal valor, string conta);

                public string FormatarTransacao()
                {
                        const string chars =
"ABCasDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
                        var random = new Random();
                        NumeroTransacao = new string(Enumerable.Repeat(chars, 15)
                                .Select(s => s[random.Next(s.Length)]).ToArray());

                        // Numero de transação formatado
                        return NumeroTransacao;
                }
        }
}

namespace SOLID.OCP.Solucao
{
        public class DebitoContaCorrente : DebitoConta
        {
                public override string Debitar(decimal valor, string conta)
                {
                        // Débito Conta Corrente
                        return FormatarTransacao();
                }
        }
}
```

```
namespace SOLID.OCP.Solucao
{
    public class DebitoContaPoupanca : DebitoConta
    {
        public override string Debitar(decimal valor, string conta)
        {
            // Valida Aniversário da Conta
            // Débito Conta Corrente
            return FormatarTransacao();
        }
    }
}
```

O Princípio de Substituição de Liskov, os subtipos devem ser substituíveis pelos seus tipos base.

Exemplo de uma violação do princípio em NET:

```
namespace SOLID.LSP.Violacao
{
    public class Retangulo
    {
        public virtual double Altura {get;set;}
        public virtual double Largura {get;set;}
        public double Area {get{return Altura * Largura;}}
    }
}

namespace SOLID.LSP.Violacao
{
    public class Quadrado : Retangulo
```

```
        {
            public override double Altura
            {
                set { base.Altura = base.Largura = value; }
            }

            public override double Largura
            {
                set { base.Altura = base.Largura = value; }
            }
        }
}

namespace SOLID.LSP.Violacao
{
    public class CalculoArea
    {
        private static void ObterAreaRetangulo(Retangulo ret)
        {
            Console.Clear();
            Console.WriteLine("Calculo da área do Retangulo");
            Console.WriteLine();
            Console.WriteLine(ret.Altura + " * " + ret.Largura);
            Console.WriteLine();
            Console.WriteLine(ret.Area);
            Console.ReadKey();
        }

        public static void Calcular()
        {
            var quad = new Quadrado()
```

```
                {
                        Altura = 10,
                        Largura = 5
                };

                ObterAreaRetangulo(quad);
        }
    }
}
```

Segue a abaixo a solução:

```
namespace SOLID.LSP.Solucao
{
    public abstract class Paralelogramo
    {
        protected Paralelogramo(int altura, int largura)
        {
            Altura = altura;
            Largura = largura;
        }

        public double Altura (get; private set;}
        public double Largura (get; private set;}
        public double Area (get { return Altura * Largura;}}
    }
}

namespace SOLID.LSP.Solucao
{
    public class Retangulo : Paralelogramo
```

```
        {
                public class Retangulo(int altura, int largura)
                        :base(altura,largura)
                {}
        }
}

namespace SOLID.LSP.Solucao
{
        public class Quadrado : Paralelogramo
        {
                public class Quadrado(int altura, int largura)
                        :base(altura,largura)
                {
                        if(largura != altura)
                                throw new ArgumentException("Os dois lados do quadrado
precisam ser iguais");
                }
        }
}

namespace SOLID.LSP.Solucao
{
        public class CalculoArea
        {
                private static void ObterAreaRetangulo(Paralelogramo ret)
                {
                        Console.Clear();
                        Console.WriteLine("Calculo da área do Retangulo");
                        Console.WriteLine();
                        Console.WriteLine(ret.Altura + " * " + ret.Largura);
```

```
            Console.WriteLine();
            Console.WriteLine(ret.Area);
            Console.ReadKey();
        }

        public static void Calcular()
        {
            var quadrado = new Quadrado(5,5);
            var retangulo = new Retangulo(10,5);

            ObterAreaRetangulo(quadrado);
            ObterAreaRetangulo(retangulo);
        }
    }
}
```

O Princípio da Segregação de Interfaces, diz que muitas interfaces específicas são melhores do que uma interface geral.

Exemplo de uma violação do princípio em NET:

```
namespace SOLID.ISP.Violacao
{
    public interface ICadastro
    {
        void ValidarDados();
        void SalvarBanco();
        void EnviarEmail();
    }
}

namespace SOLID.ISP.Violacao
```

```
{
    public class CadastroProduto : ICadastro
    {
        public void ValidarDados()
        {
            // Validar valor
        }

        public void SalvarBanco()
        {
            // Inserir dados na tabela de Produto
        }

        public void EnviarEmail()
        {
            // Produto não tem e-mail
            throw new NotImplementedException();
        }
    }
}

namespace SOLID.ISP.Violacao
{
    public class CadastroCliente : ICadastro
    {
        public void ValidarDados()
        {
            // Validar CPF, Email
        }

        public void SalvarBanco()
```

```
        {
            // Inserir dados na tabela de Cliente
        }

        public void EnviarEmail()
        {
            // Enviar e-mail para cliente
        }
    }
}
```

Segue a solução da violação, abaixo:

```
namespace SOLID.ISP.Solucao
{
    public interface ICadastro
    {
        void SalvarBanco();
    }
}

namespace SOLID.ISP.Solucao
{
    public interface ICadastroCliente : ICadastro
    {
        void ValidarDados();
        void EnviarEmail();
    }
}

namespace SOLID.ISP.Solucao
```

```
{
	public interface ICadastroProduto : ICadastro
	{
		void ValidarDados();
	}
}

namespace SOLID.ISP.Solucao
{
	public class CadastroCliente : ICadastroCliente
	{
		public void ValidarDados()
		{
			// Validar CPF, Email
		}

		public void SalvarBanco()
		{
			// Inserir dados na tabela de Cliente
		}

		public void EnviarEmail()
		{
			// Enviar e-mail para cliente
		}
	}
}

namespace SOLID.ISP.Solucao
{
	public class CadastroProduto : ICadastroProduto
```

```
        {
            public void ValidarDados()
            {
                // Validar valor
            }

            public void SalvarBanco()
            {
                // Inserir dados na tabela de Produto
            }
        }
}
```

O Princípio da Inversão de Dependências, diz que devemos depender de abstrações e não de classes concretas.

Exemplo de uma violação do princípio em NET:

```
namespace SOLID.DIP.Violacao
{
    public class Cliente
    {
        public int ClienteId {get;set;}
        public string Nome {get;set;}
        public Email Email {get;set;}
        public CPF Cpf {get;set;}
        public DateTime DataCadastro {get;set;}

        public bool Validar()
        {
            return Email.Validar() && Cpf.Validar();
        }
```

```
        }
}

namespace SOLID.DIP.Violacao
{
        public class CPF
        {
                public string Numero{get;set;}

                public bool Validar()
                {
                        return Numero.Length == 11;
                }
        }
}

namespace SOLID.DIP.Violacao
{
        public class Email
        {
                public string Endereco{get;set;}

                public bool Validar()
                {
                        return Endereco.Contains("@");
                }
        }
}

namespace SOLID.DIP.Violacao
{
```

```
	public class ClienteRepository
	{
		public void AdicionarCliente(Cliente cliente)
		{
			using(var cn = SqlConnection())
			{
				var cmd = new SqlCommand();

				cn.ConnectionString = "MinhaConnectionString";
				cmd.Connection = cn;
				cmd.CommandType = CommandType.Text;
				cmd.CommandText = "INSERT INTO CLIENTE (NOME,
EMAIL, CPF, DATACADASTRO) VALUES (@nome, @email, @cpf, @dataCad))";

				cmd.Parameters.AddWithValue("nome", cliente.Nome);
				cmd.Parameters.AddWithValue("email", cliente.Email);
				cmd.Parameters.AddWithValue("cpf", cliente.Cpf);
				cmd.Parameters.AddWithValue("dataCad",
cliente.DataCadastro);

				cn.Open();
				cmd.ExecuteNonQuery();
			}
		}
	}
}

namespace SOLID.DIP.Violacao
{
	public class ClienteService
	{
		public string AdicionarCliente(Cliente cliente)
```

```
            {
                if(!cliente.Validar())
                    return "Dados inválidos";

                var repo = new ClienteRepository();
                repo.AdicionarCliente(cliente);

                EmailServices.Enviar("empresa@empresa.com",
cliente.Email.Endereco, "Bem Vindo", "Parabéns está Cadastrado");

                return "Cliente cadastrado com sucesso";
            }
        }
}

namespace SOLID.DIP.Violacao
{
    public static class EmailServices
    {
        public static void Enviar(string de, string para, string assunto, string
mensagem)
        {
            var mail = new MailMessage(de, para);
            var client = new SmtpClient
            {
                Port = 25,
                DeliveryMethod = SmtpDeliveryMethod.Network,
                UseDefaultCredentials = false,
                Host = "smtp.google.com"
            };

            mail.Subject = assunto;
```

```
                mail.Body = mensagem;
                client.Send(mail);
            }
        }
}
```

Segue abaixo a soluçao do Princípio em NET:

```
namespace SOLID.DIP.Solucao
{
    public class Cliente
    {
        public int ClienteId {get;set;}
        public string Nome {get;set;}
        public Email Email {get;set;}
        public CPF Cpf {get;set;}
        public DateTime DataCadastro {get;set;}

        public bool Validar()
        {
            return Email.Validar() && Cpf.Validar();
        }
    }
}

namespace SOLID.DIP.Solucao
{
    public class CPF
    {
        public string Numero{get;set;}
```

```
                public bool Validar()
                {
                        return Numero.Length == 11;
                }
        }
}

namespace SOLID.DIP.Solucao
{
        public class Email
        {
                public string Endereco{get;set;}

                public bool Validar()
                {
                        return Endereco.Contains("@");
                }
        }
}

namespace SOLID.DIP.Solucao
{
        public interface IClienteRepository
        {
                void AdicionarCliente(Cliente cliente);
        }
}

namespace SOLID.DIP.Solucao
{
        public class ClienteRepository : IClienteRepository
```

```
        {
            public void AdicionarCliente(Cliente cliente)
            {
                using(var cn = SqlConnection())
                {
                    var cmd = new SqlCommand();

                    cn.ConnectionString = "MinhaConnectionString";
                    cmd.Connection = cn;
                    cmd.CommandType = CommandType.Text;
                    cmd.CommandText = "INSERT INTO CLIENTE (NOME,
EMAIL, CPF, DATACADASTRO) VALUES (@nome, @email, @cpf, @dataCad))";

                    cmd.Parameters.AddWithValue("nome", cliente.Nome);
                    cmd.Parameters.AddWithValue("email", cliente.Email);
                    cmd.Parameters.AddWithValue("cpf", cliente.Cpf);
                    cmd.Parameters.AddWithValue("dataCad",
cliente.DataCadastro);

                    cn.Open();
                    cmd.ExecuteNonQuery();
                }
            }
        }
}

namespace SOLID.DIP.Solucao
{
        public interface IClienteService
        {
            void AdicionarCliente(Cliente cliente);
        }
```

```
}

namespace SOLID.DIP.Solucao
{
	public interface IEmailService
	{
		void Enviar(string de, string para, string assunto, string mensagem);
	}
}

namespace SOLID.DIP.Solucao
{
	public static class EmailService : IEmailService
	{
		public static void Enviar(string de, string para, string assunto, string
mensagem)
		{
			var mail = new MailMessage(de, para);
			var client = new SmtpClient
			{
				Port = 25,
				DeliveryMethod = SmtpDeliveryMethod.Network,
				UseDefaultCredentials = false,
				Host = "smtp.google.com"
			};

			mail.Subject = assunto;
			mail.Body = mensagem;
			client.Send(mail);
		}
	}
```

```
}

namespace SOLID.DIP.Solucao
{
	public class ClienteService : IClienteService
	{
		private readonly IClienteRepository _clienteRepository;
		private readonly IEmailService _emailService;

		public ClienteService(
			IClienteRepository clienteRepository,
			IEmailService emailService)
		{
			_clienteRepository = clienteRepository;
			_emailService = emailService;
		}

		public string AdicionarCliente(Cliente cliente)
		{
			if(!cliente.Validar())
				return "Dados inválidos";

			_clienteRepository.AdicionarCliente(cliente);

			_emailService.Enviar("empresa@empresa.com",
cliente.Email.Endereco, "Bem Vindo", "Parabéns está Cadastrado");

			return "Cliente cadastrado com sucesso";
		}
	}
}
```

2. O que são Delegates em C# e como o tipo genérico Func pode ser utilizado. Forneça um exemplo de código onde um Func é utilizado para encapsular uma função anônima que calcula a soma de dois números.

R.: Delegate é uma referência de um método. Usando delegate é possível encapsular a referência a um método dentro de um objeto de delegação.

Exemplo de código:

```
namespace Delegate_Func
{
        public void Program()
        {

                Func<int,int,int> soma = (x,y) => x + y;

                Console.WriteLine(soma(2,4)); // 6
        }
}
```

3. Explique a diferença entre as classes Task e Thread no .NET. Quando usar uma sobre a outra? Forneça um exemplo prático de uso de Task.

R.: Task são como uma promessa, um exemplo de Task<T> promete devolver um T, e isso não necessariamente precisa ser agora, pode devolver mais tarde. A Task trabalha no modelo de assíncrono.

Já a Thread são linha de execução, é uma forma de dividir um processo em duas ou mais tarefas, que pode ser executadas em paralelos.

```
namespace Exemplo_Task
{
        public static Task Program(string[] args)
```

```
        {
            Console.WriteLine("Tecle algo para iniciar...\n");
            Console.ReadKey();

            await Aguardar(5);

            Console.WriteLine("Já passou 5 segundos...\n");
            Console.WriteLine("fim");
            Console.ReadLine();
        }

        public static Task Aguardar(int tempo)
        {
            Console.WriteLine("Iniciando espera...");
            await Task.Delay(TimeSpan.FromSeconds(tempo));
            Console.WriteLine("Fim da espera...");
        }
}
```

4. O que é Dependency Injection? Explique como o .NET Core/6 implementa o padrão de injeção de dependência e forneça um exemplo de código.

R.: É o princípio da inversão de dependência (DIP) do SOLID. Este princípio prega que uma classe deve depender de abstrações e não depender de implementações.

No .NET Core 6 a injeção de dependência é funcionalidade nativa e suportada pelo framework. Ela é implementada por meio da interface IServiceCollection, que permite registrar e configurar os serviços e dependências do aplicativo.

```
namespace Exemplo_DependencyInjection
{
        public static Program()
        {
```

```
            var builder = WebApplication.CreateBuilder(args);

            builder.Services.AddSingleton<ICadastroCliente, CadastroCliente>();
            builder.Services.AddSingleton<ICadastroProduto, CadastroProduto>();
            builder.Services.AddSingleton<IClienteRepository, ClienteRepository>();
            builder.Services.AddSingleton<IClienteService, ClienteService>();
            builder.Services.AddSingleton<IEmailService, EmailService>();
        }
}
```

5. Descreva o funcionamento do Entity Framework e explique as diferenças entre Code First, Database First e Model First. Qual a abordagem que você prefere e por quê?

R.: O Entity Framework permite que desenvolvedores trabalhem com dados na forma de propriedade e objetos específicos de domínio, sem se preocupar com as tabelas e colunas do banco de dados.

A diferença de Code First é que o Entity Framework gera o banco de dados através do código que foi gerado da aplicação, de acordo com os atributos e as propriedades das classes.

Já no Model First o Entity Framework gera o banco de dados através de modelos usando o Entetity Framework Designer. O modelo é armazenado em um arquivo EDMX e através dele é gerado esquema de banco de dados.

E o DataBase First permite fazer a engenharia reversa de um modelo de banco de dados já existente. O modelo também é armazenado em um arquivo EDMX, e pode ser exibido e editado no Entity Framework Designer.

Eu prefiro a funcionalidade Code First, porque eu posso iniciar um banco de dados através do código da minha aplicação, sem precisa de um banco de dados existente.