EJERCICIO 0
En el patrón Builder,
¿Qué principios SOLID se aplican? Justifiquen su respuesta.

SRP: Podes crear un builder para la responsabilidad de crear una clase, reduciendo asi las razones de cambio de la clase original por la creación y en caso de que el orden de creación sea de importancia esta responsabilidad tambien puede ser delegada a otra clase llamada Director.
OCP: Es abierto a la extensión ya que si el dia de mañana hay una nueva forma de crear una clase entonces se hace un nuevo Builder del que haga uso el Director para crear los objetos.
Liskov: Se ve aplicado el Liskov debido a que se pueden usar cualquier tipo de Builder que implemente la interfaz para poder pasarlos al Director y crear asi los objetos.

¿Qué principios se violan, si los hubiera? Justifiquen su respuesta.

Dependency Inversion Principle

El patrón Builder puede cumplir con este principio si se utilizan interfaces o abstracciones para los builders, permitiendo que las dependencias sean invertidas. Sin embargo, esto depende de cómo se implementa el patrón en concreto. Si el código cliente depende directamente de implementaciones concretas de builders, podría no cumplir completamente con este principio.

EJERCICIO 1:
APLICAMOS BUILDER

```
public Sandwich {
	bread: string
	cheese: string
	meat: string
	vegetables: string
	condiment: string

	// getters y setters…
}

public interface IBuilder {
	addBread(string bread)
	addCheese(string cheese)
	addMeat(string meat)
	addVegetables(string vegetable)
	addCondiments(string condiments)
}

public SandwichBuilder: IBuilder {
	result: Sandwich;

	reset () {
		this.result = new Sandwich();
	}
```

```
        addBread(string bread) {
                this.result.setBread(bread)
        }
        addCheese(string cheese) {
                this.result.setCheese(cheese)
        }
        addMeat(string meat) {
                this.result.setMeat(meat)
        }
        addVegetables(string vegetables) {
                this.result.setVegetables(vegetables)
        }
        addCondiments(string condiments) {
                this.result.setCondiments(condiments)
        }
        getProduct(): Sandwich{
                result = this.result
                this.reset()
                return result
        }
}

public class Director {
        builder: IBuilder

        Director(IBuilder builder) {
                this.builder = builder;
        }

        makeHamSanwdich() {
                this.builder.addBread(“White”)
                this.builder.addCheese(“Swiss”)
                this.builder.addMeat(“Ham”)
                this.builder.addVegetables(“Lettuce, Tomato”)
                this.builder.addCondiments(“Mayo, Mustard”)
        }

        makeHamSanwdich() {
                this.builder.addBread(“Wheat”)
                this.builder.addCheese(“Cheddar”)
                this.builder.addMeat(“Turkey”)
                this.builder.addVegetables(null)
                this.builder.addCondiments(“Mayo”)
        }
}
```

Ejercicio 2:

```
public abstract class GameUnit
{
        public int Health { get; set; }
        public int Attack { get; set; }
        public int Defense { get; set; }
        // Simula la carga de recursos costosos como modelos 3D, texturas, etc.
        public virtual void LoadResources()
        {
                Console.WriteLine("Loading resources...");
        }
}
public class Archer : GameUnit
{
        public Archer()
        {
                LoadResources();
                Health = 100;
                Attack = 15;
                Defense = 5;
        }
}
public class Knight : GameUnit
{
        public Knight()
        {
                LoadResources();
                Health = 200;
                Attack = 20;
                Defense = 10;
                }
          }
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Creating original Archer...");
      Archer originalArcher = new Archer();

      Console.WriteLine("Copying Archers manually...");
      Archer copiedArcher1 = new Archer
      {
         Health = originalArcher.Health,
         Attack = originalArcher.Attack,
         Defense = originalArcher.Defense
      };
      Archer copiedArcher2 = new Archer
      {
         Health = originalArcher.Health,
         Attack = originalArcher.Attack,
```

```
            Defense = originalArcher.Defense
        };

        Console.WriteLine("Creating original Knight...");
        Knight originalKnight = new Knight();

        Console.WriteLine("Copying Knights manually...");
        Knight copiedKnight1 = new Knight
        {
            Health = originalKnight.Health,
            Attack = originalKnight.Attack,
            Defense = originalKnight.Defense
        };
        Knight copiedKnight2 = new Knight
        {
            Health = originalKnight.Health,
            Attack = originalKnight.Attack,
            Defense = originalKnight.Defense
        };
    }
}
```

a) Prototipe:

```
abstract class GameUnit
{
        public GameUnit
        {
                Health { get; set; }
                Attack { get; set; }
                Defense { get; set; }

        }
        public clone()

}
public class Knight : GameUnit
{
         public Knight()
         {
                 LoadResources();
                 Health = 200;
                 Attack = 20;
                 Defense = 10;
         }
}
```

```
public class Archer : GameUnit
{
	public Archer()
	{
		LoadResources();
		Health = 100;
		Attack = 15;
		Defense = 5;
	}
}
class Program
{
	Console.WriteLine("Creating original Archer...");
	Archer originalArcher = new Archer();
	Archer copyArcher = originalArcher.clone()


}
```

Ejercicio 3:

a) APLICAMOS FACTORY METHOD

b)

```
public interface IMessagingService {
        void SendMessage(string message);
}

public class SMSService : IMessagingService {
        public void SendMessage(string message) {
                Console.WriteLine($"Sending SMS message: {message}"); // Lógica para enviar
        SMS...
        }
}

public class EmailService : IMessagingService {
        public void SendMessage(string message) {
                Console.WriteLine($"Sending Email: {message}"); // Lógica para enviar Email...
                }
}

public class FacebookService : IMessagingService {
        public void SendMessage(string message) {
                Console.WriteLine($"Sending Facebook Message: {message}"); // Lógica para
        enviar mensaje de Facebook...
                }
}

public abstract class MessagingApp {
        public abstract IMessagingService GetService();
        public MessagingApp(IMessagingService service) {}
}

public class SMSApp : MessagingApp {
        public override IMessagingService GetService() { return new SMSService(); }
}

public class EmailApp : MessagingApp {
        public override IMessagingService GetService() { return new EmailService(); }
}
```

Ejercicio 4

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
    public List<string> BorrowedStudents { get; set; }

    public Book()
    {
        // Simular la carga de recursos.
        Console.WriteLine("Acquiring a new book...");
        BorrowedStudents = new List<string>();
    }

    public void BorrowBook(string studentName)
    {
        BorrowedStudents.Add(studentName);
    }

    public void PrintBorrowedStudents()
    {
        Console.WriteLine($"Book: {Title}, Borrowed by: {string.Join(", ", BorrowedStudents)}");
    }
}

class Program
{
    static void Main(string[] args)
    {
            // Adquirir el libro original.
            Book originalBook = new Book
            {
                    Title = "Harry Potter",
                    Author = "J.K. Rowling"
            };

            originalBook.BorrowBook("Alice");
            // Adquirir una copia adicional del mismo libro manualmente.
            Book additionalCopy = new Book
            {
                    Title = originalBook.Title,
                    Author = originalBook.Author,
                    BorrowedStudents = new List<string>() // Inicializar la lista vacía.
            };
            // Prestar la copia adicional a otro estudiante.
            additionalCopy.BorrowBook("Bob");
            // Imprimir los estudiantes a los que se les prestó cada copia del libro.
```

```
        originalBook.PrintBorrowedStudents();
        additionalCopy.PrintBorrowedStudents();
        }
}
```

a) Prototipe
b)

```
abstract class Prototype
{
        public abstract Prototype clone(){
        }
}
public class Book : Prototype
{
   public string Title { get; set; }
   public string Author { get; set; }
   public List<string> BorrowedStudents { get; set; }

   public Book()
   {
      // Simular la carga de recursos.
      Console.WriteLine("Acquiring a new book...");
      BorrowedStudents = new List<string>();
   }

   public void BorrowBook(string studentName)
   {
      BorrowedStudents.Add(studentName);
   }

   public void PrintBorrowedStudents()
   {
      Console.WriteLine($"Book: {Title}, Borrowed by: {string.Join(", ", BorrowedStudents)}");
   }
}

class Program
{
   static void Main(string[] args)
   {
         // Adquirir el libro original.
        Book originalBook = new Book
        {
                Title = "Harry Potter",
                Author = "J.K. Rowling"
        };

        originalBook.BorrowBook("Alice");
```

```
            // Adquirir una copia adicional del mismo libro manualmente.
            Book additionalCopy = (Book)originalBook.Clone();
            // Prestar la copia adicional a otro estudiante.
            additionalCopy.BorrowBook("Bob");
            // Imprimir los estudiantes a los que se les prestó cada copia del libro.
            originalBook.PrintBorrowedStudents();
            additionalCopy.PrintBorrowedStudents();
            }
}
```

Ejercicio 5:

```
public class TravelPlan
{
   public TravelPlan(string flight, string hotel, string carRental,
                  string[] activities, string[] restaurantReservations)
   {
         // Constructor con muchos parámetros, algunos de los cuales pueden ser opcionales
(nulos o valores predeterminados).
         }

   // Propiedades y métodos...
}

// Ejemplo de uso:
 TravelPlan plan = new TravelPlan("Flight1", "Hotel1", null, new string[] {"Tour1", "Tour2"},
null, ...);
```

a) Builder

```
public interface IBuilder
{
         public reset()
         public setFlight()
         public setHotel()
         public setCarRental()
         public setActivitities()
         public setRestaurantReservations()
}

public class TravelPlanBuilder: IBuilder
{
         private travelPlan: TravelPlan
         public reset()
         {

         }
         public setFlight()
         {
                  //Instrucciones para agregar vuelo
         }
         …
}
public class TravelPlan
{

}
```

Ejercicio 6

```
class Program {
	static void Main() {
		// Crear un servicio
		SomeService service = new SomeService();
		// Realizar una tarea que requiere configuración
		service.PerformTask();
		 // Otro ejemplo: acceder a la configuración desde otra parte de la aplicación
		string apiEndpoint =
	ConfigurationManager.Instance.GetConfiguration("apiEndpoint");

		Console.WriteLine($"API Endpoint: {apiEndpoint}");
		}
	}
```

a)Singleton
b)

```
class Program {
	static void Main() {
		// Crear un servicio
		SomeService service = SomeService.getInstance();
		// Realizar una tarea que requiere configuración
		service.PerformTask();
		 // Otro ejemplo: acceder a la configuración desde otra parte de la aplicación
		string apiEndpoint =
	ConfigurationManager.getInstance().GetConfiguration("apiEndpoint");

		Console.WriteLine($"API Endpoint: {apiEndpoint}");
		}
	}
```