1.- Patrón Builder para crear objetos Sandwich de manera más flexible y modular. Agregar **Interfaz ISandwichBuilder** con una implementación **SandwichBuilder** con la que se pueden crear distintas versiones de un sandwich.

```
public class Sandwich{
    public string Bread { get; set; }
    public string Cheese { get; set; }
    public string Meat { get; set; }
    public string Vegetables { get; set; }
    public string Condiments { get; set; }

    public override string ToString(){
           return $"Sandwich with
           {Bread}bread,{Cheese}cheese,{Meat} meat,
           {Vegetables}vegetables,and{Condiments}condiments.";
    }
}

public interface ISandwichBuilder
{
    void SetBread(string bread);
    void SetCheese(string cheese);
    void SetMeat(string meat);
    void SetVegetables(string vegetables);
    void SetCondiments(string condiments);
    Sandwich GetSandwich();
}

public class SandwichBuilder : ISandwichBuilder
{
    private Sandwich sandwich;

    public SandwichBuilder(){
        sandwich = new Sandwich();
    }

    public void SetBread(string bread){
        sandwich.Bread = bread;
    }

    public void SetCheese(string cheese){
        sandwich.Cheese = cheese;
    }

    public void SetMeat(string meat){
        sandwich.Meat = meat;
    }
```

```
    public void SetVegetables(string vegetables){
        sandwich.Vegetables = vegetables;
    }

    public void SetCondiments(string condiments){
        sandwich.Condiments = condiments;
    }

    public Sandwich GetSandwich(){
        return sandwich;
    }
}

class Program
{
    static void Main(string[] args)
    {
        ISandwichBuilder sandwichBuilder = new SandwichBuilder();

        sandwichBuilder.SetBread("White");
        sandwichBuilder.SetCheese("Swiss");
        sandwichBuilder.SetMeat("Ham");
        sandwichBuilder.SetVegetables("Lettuce, Tomato");
        sandwichBuilder.SetCondiments("Mayo, Mustard");
        Sandwich hamSandwich = sandwichBuilder.GetSandwich();


        sandwichBuilder = new SandwichBuilder();
        sandwichBuilder.SetBread("Wheat");
        sandwichBuilder.SetCheese("Cheddar");
        sandwichBuilder.SetMeat("Turkey");
        sandwichBuilder.SetCondiments("Mayo");
        Sandwich turkeySandwich = sandwichBuilder.GetSandwich();

        Console.WriteLine(hamSandwich);
        Console.WriteLine(turkeySandwich);

    }
}
```

2. El patrón de diseño más adecuado sería el patrón Prototype ya que permite crear nuevos objetos Archer y Knight a partir de objetos prototipo existentes sin tener que copiar manualmente los atributos uno por uno.

```
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 abstract GameUnit Clone();

}

public class Archer : GameUnit
{
	public Archer()
	{
		LoadResources();
		Health = 100;
		Attack = 15;
		Defense = 5;
	}

	public override GameUnit Clone()
	{
		return new Archer(Health = Health, Attack = Attack, Defense = Defense);
	}
}
public class Knight : GameUnit
{
	public Knight()
	{
		LoadResources();
		Health = 200;
		Attack = 20;
		Defense = 10;
	}
	public override GameUnit Clone()
	{
		return new Archer(Health = Health, Attack = Attack, Defense = Defense);
	}
}
```

```
class Program {
        static void Main(string[] args)
        {
                Console.WriteLine("Creating original Archer...");
                Archer originalArcher = new Archer();
                Console.WriteLine("Copying Archers using Prototype...");
                Archer copiedArcher1 = originalArcher.Clone() as Archer;
                Archer copiedArcher2 = originalArcher.Clone() as Archer;

                Console.WriteLine("Creating original Knight...");
                Knight originalKnight = new Knight();
                Console.WriteLine("Copying Knights using Prototype...");
                Knight copiedKnight1 = originalKnight.Clone() as Knight;
                Knight copiedKnight2 = originalKnight.Clone() as Knight;
        }
}
```

3.- Factory Method, en la cual la clase MessagingApp posee un método que devuelve un IMessagingService adecuado para sí mismo.

```
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();
      }
}

public class FacebookApp : MessagingApp
{
      public override IMessagingService GetService()
      {
            return new FacebookService();
      }
}
```

4.- El patrón más adecuado es el patrón Prototype, debido a que hay varias partes de la ejecución del programa donde se clonan libros, lo cual es muy facilitado por este patrón.

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

	public Book()
	{
		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)}");
	}

	public Book Clone()
	{
		return new Book
		{
			Title = this.Title,
			Author = this.Author,
			BorrowedStudents = new List<string>()
		};
	}
}

class Program
{
	static void Main(string[] args)
	{
		Book originalBook = new Book
		{
			Title = "Harry Potter",
			Author = "J.K. Rowling"
		};
		originalBook.BorrowBook("Alice");
```

```
            Book additionalCopy = originalBook.Clone();
            additionalCopy.BorrowBook("Bob");

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

Ejercicio 5

1 - El ejercicio muestra una clase cuyo constructor requiere especificar una gran cantidad de parametros, por lo que el patron mas adecuado a implementar seria el patron Builder.

2 -

```
public class TravelPlan
{
	public TravelPlan()
	{
		// Constructor modificado con 0 parametros.
		// Las propiedades previamente inicializadas
		// pasan a tener valores nulos o por defecto.
	}

	// Propiedades y métodos...

	// nuevos setters para las propiedades que se
	// inicializaban en el constructor
	public void setFlight(string flight)
	{
		…
	}

	// todos los otros setters
}

public interface TravelPlanBuilder
{
	public void reset();
	public void buildFlight();
	public void buildHotel();
	public void buildCarRental();
	public void buildActivities();
	public void buildRestaurantReservations();
	public void build…
}

public class TravelPlanABuilder
{
	private TravelPlan travelPlan;
```

```
      public TravelPlanABuilder()
      {
            this.reset()
      }

      public void reset()
      {
            this.travelPlan = new TravelPlan();
      }

      public TravelPlan getResult()
      {
            return this.travelPlan;
      }

      public void buildFlight()
      {
            this.travelPlan.setFlight(“Flight1”);
      }

      public void buildHotel()
      {
            this.travelPlan.setHotel(“Hotel1”);
      }

      // otros metodos build
      …
}

public class TravelPlanDirector
{
      private TravelPlanBuilder builder;

      public TravelPlanDirector(TravelPlanBuilder builder)
      {
            this.builder = builder;
      }

      public void changeBuilder(TravelPlanBuilder builder)
      {
```

```
            this.builder = builder;
      }

      public TravelPlan make()
      {
            thus.builder.reset()
            thus.builder.buildFlight();
            this.builder.build…
            return this.builder.getResult();
      }
}

// Ejemplo de uso:
builder = new TravelPlanABuilder();
director = new TravelPlanDirector(builder);
TravelPlan plan = director.make();
```

Ejercicio 6

1 - El código parece intentar disponibilizar una instancia de una clase con datos de configuración a partir de una propiedad de clase de ConfigurationManager. Esto sería mejor implementado utilizando el patrón Singleton.

2 -

```
class Program
{

	public class ConfigurationManager
	{
		private ConfigurationManager instance;

		public ConfigurationManager()
		{
		}

		public static ConfigurationManager getInstance()
		{
			if (this.instance == null)
			{
				this.instance = new ConfigurationManager();
			}
			return this.instance;
		}

		// propiedades y métodos de la clase
	}

	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
		configManager = ConfigurationManager.getInstance();
```

```
            string apiEndpoint = configManager.GetConfiguration(
                  "apiEndpoint");
            Console.WriteLine($"API Endpoint: {apiEndpoint}");
      }
}
```