UT5 TA3 – Grupo 1

Ejercicio 0.

El principio SOLID que más se aplica es el SRP, ya que el fundamento del patrón Builder es la separación de una clase de su(s) proceso(s) de construcción. Este patrón es más redituable en términos de encapsulación cuando dichos procesos de construcción son complejos, cuentan con múltiples pasos intermedios o requieren diferir parte de la inicialización.

Además, se puede implementar una clase Director que es la que ofrece métodos públicos que corresponden a cada proceso de producción. Por lo tanto, esta clase también cumple con SRP pero, como contrapartida, no cumple con el OCP debido a que agregar un procedimiento de construcción siempre requiere modificar el Director. Además, los distintos Builder quedan fuertemente acoplados a los Productos.

Ejercicio 1.

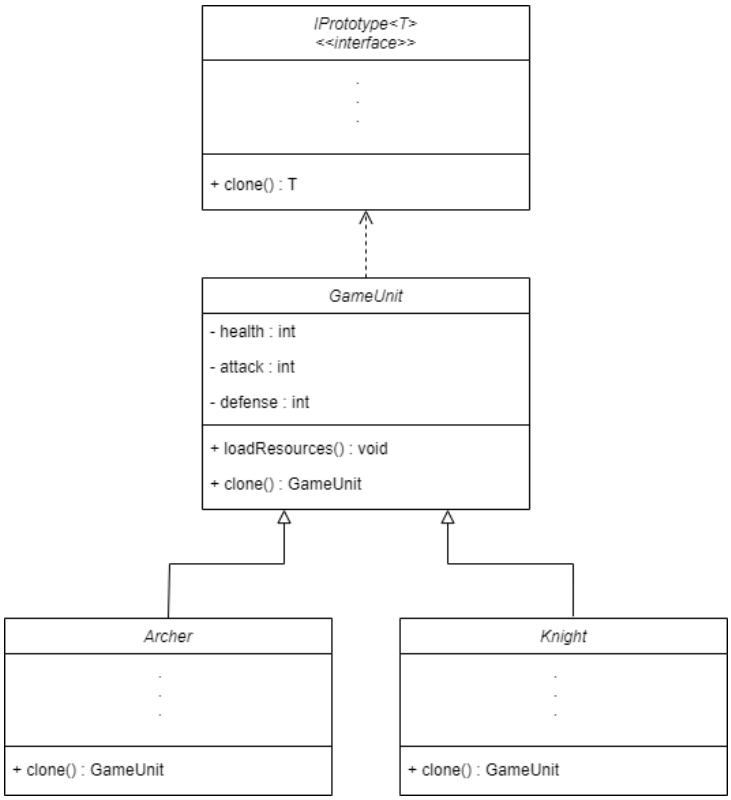
Patrón a aplicar: Builder.

```
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; }

}


abstract class SandwichBuilder{

    protected Sandwich sandwich;

    public Sandwich(string bread, string cheese, string meat,String vegetables, string condiments){
     Bread = bread;
     Cheese = cheese;
     Meat = meat;
     Vegetables = vegetables;
     Condiments = condiments;
     }

    public abstract Sandwich buildSandwich(string bread, string cheese, string meat,String vegetables, string condiments){
        this.sandwich(bread,cheese,meat,vegetables,condiments);
    }

    public Sandwich getSandwich() { return sandwich; }


}


public class HamSandwichBuilder extends SandwichBuilder{

    @Override
    public Sandwich buildSandwich(string bread, string cheese,String vegetables, string condiments){
        this.sandwich(bread,cheese,"Ham",vegetables,condiments);
    }

}


public class TurkeySandwichBuilder extends SandwichBuilder{

   @Override
    public Sandwich buildSandwich(string bread, string cheese, string condiments){
        this.sandwich(bread,cheese,"Turkey",null,condiments);
    }


}



public class Program{


    public static void main(String[] args){

        Sandwich turkey = new TurkeySandwichBuilder.buildSandwich("Wheat", "Cheddar", "Mayo")
        Sandwich ham = new hamSandwichBuilder.buildSandwich("White", "Swiss","Lettuce, Tomato", "Mayo, Mustard")

    }
}
```

Ejercicio 2.

Planteamos aplicar el patrón *Prototype* ya que, se desea crear distintos “guerreros” donde poseen las mismas características/atributos.

La solución planteada en base a la estructura que define el patrón *Prototype*, se crea una interfaz “*IPrototype*” del tipo T donde contiene el método denominado “*clone()*” del tipo T. La clase abstracta “*GameUnit*” implementa la interfaz mencionada anteriormente y las clases “*Archer*” y “*Knight*” heredan los atributos y métodos de la clase “*GameUnit*” para crear copias de nuevos arqueros y caballeros con las mismas características de arqueros y caballeros ya existentes mediante el método denominado “*clone()*” que debe ser implementado correctamente en las clases denominadas “*Archer*” y “*Kngiht*”.



Clase Archer

```
public override GameUnit clone() {
        return new Archer {
                health = this.health,
                attack = this.attack,
                defense = this.defense
        };
}
```

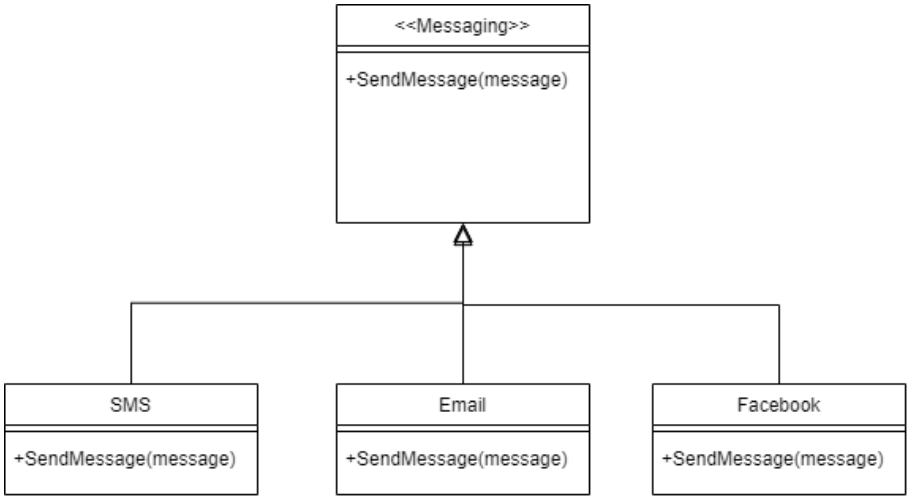
Clase Knight

```
public override GameUnit clone() {
        return new Knight {
                health = this.health,
                attack = this.attack,
                defense = this.defense
        };
}
```

Ejercicio 3.

1- Factory: Para este caso, se podría crear una interfaz que contenga el método sendMessage(). Luego, se crea una clase para cada tipo de servicio. En cada clase, el método contendrá la lógica necesaria para cada tipo existente. Ahora el método solo recibirá el “message”, porque no es más necesario pasarle el tipo, porque cuando se invoque el método será parte de una clase de este.

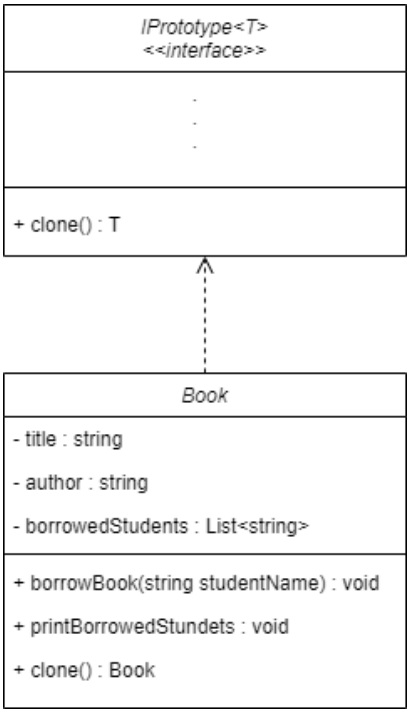
2- El día de mañana si queremos agregar un nuevo servicio de mensajería, sería tan fácil como crear una clase que implemente la interfaz “Messaging”.



Ejercicio 4.

Planteamos aplicar el patrón *Prototype* ya que, se desea clonar el objeto libro para posteriormente crear un nuevo libro con las mismas características del libro original a excepción de la lista que contiene los estudiantes que solicitaron el libro a clonar. Este patrón se caracteriza por permitir crear nuevos objetos a través de la clonación de un objeto ya existente permitiendo modificar los atributos que posee sin afectar al objeto original.

La solución planteada en base a la estructura que define el patrón *Prototype*, se crea una interfaz "*IPrototype*" del tipo T donde contiene el método denominado "*clone()*" del tipo T. La clase "*Book*" implementa la interfaz mencionada anteriormente donde al implementar el método "*clone()*" se crea el objeto "*Book*" con exactamente los mismos atributos que el objeto "Book" original a excepción de la lista que contiene los estudiantes que solicitaron el libro a clonar ya que, es un nuevo libro.

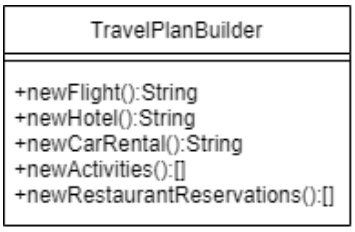



```
public Book clone() {
        return new Book {
                title = this.title,
                author = this.author,
                borrowedStudents = new List<string>()
        };
}
```

Ejercicio 5.

1- Para el siguiente caso, consideramos que lo mejor es utilizar el Builder.

2- Se puede armar el plan de viaje de manera personalizada, evitando ingresar campos en NULL.



Ejercicio 6.

1- A la configuración intentan acceder/utilizar desde diferentes partes de la aplicación, por lo tanto, al tener una única instancia de esta permite que se acceda al recurso de manera ordenada, de modo que no haya colisiones.

2- Se crea una clase “Configuración”, la cual es un Singleton.