taEVALUACIÓN PARCIAL – RESOLUCIÓN DE CASOS

FACULTAD:	**Tecnología Informática**		
CARRERA:	**Ingeniería en sistemas**		
ALUMNO/A:	**FRANCISCO JOSÉ ESCOBAR**		
SEDE:	**Buenos Aires**	LOCALIZACIÓN:	**Centro**
ASIGNATURA:	**Trabajo de diploma**		
CURSO:		TURNO:	**Noche**
PROFESOR:	**Battaglia, Nicolás**	FECHA:	**01/10/2021**
TIEMPO DE RESOLUCIÓN:	**7 días parte práctica**	EXAMEN PARCIAL NRO:	**1**
MODALIDAD DE RESOLUCIÓN:		Virtual Domiciliario	

Criterios de calificación: Para acreditar los saberes deberá obtener, al menos, el 60% de los aspectos conceptuales, además de, al menos, el 60% de los aspectos procedimentales. La calificación final se obtendrá luego de la defensa oral del trabajo presentado.

Criterios de resolución: Los alumnos recibirán la consigna del examen en la fecha de evaluación prevista por el cronograma de la asignatura. Deberán resolver y entregar el este examen en el plazo conforme al plan de clases de la asignatura.

Criterios de evaluación: Se evaluará la claridad en el planteamiento de los aspectos conceptuales y procedimentales. Desarrollo del diagrama de flujo. Desarrollo del código acorde al diagrama de flujos. La evaluación se hará a partir de la siguiente grilla:

Criterio	**Calificación**	**Observaciones**
Instancia oral		
Aspectos Conceptuales		
Pregunta 1		
Pregunta 2		
Pregunta 3		
Pregunta 4		
Pregunta 5		
Pregunta 6		
Aspectos procedimentales		
Identificación de patrones		**Composite -**
Desarrollo del diagrama de clase punto 1		
Desarrollo del código acorde al diagrama punto 1		
Conexión a BBDD		
Calificación final		

Forma de entrega del examen

Se deberá entregar un documento en PDF con la resolución de todos los aspectos conceptuales y la resolución del punto 1 de los aspectos procedimentales, junto con el código fuente para cumplir los puntos 2 y 3. La defensa oral de la parte práctica

consistirá en mostrar el funcionamiento de la solución y responder algunas preguntas que el docente considere necesarias sobre aspectos teóricos.

Los archivos entregados deberán tener el siguiente formado:

- Día turno, Apellido y Nombre, 1er parcial, TD1 3K 2020.PDF.

Aspectos conceptuales

1. Describa brevemente que caracteriza al software como productos.

   Rta: Del software como producto podemos destacar las siguientes características:
   - Es el producto del trabajo de los programadores, ingenieros en sistemas, arquitectos de software, diseñadores y analistas.
   - No es físico: se desarrolla o modifica haciendo uso del intelecto. No se manufactura en el sentido clásico.
   - Como producto, se puede afirmar que luego de su puesta en marcha se necesita que se lo mantenga.

2. Defina los conceptos de Autorización y Autenticación. Describa como planteo estos conceptos en su trabajo final.

   Rta: En el contexto de sistemas de información el concepto de "autorización" se refiere a lo que está permitido o no realizar sobre un sistema, está relacionado a los permisos que tenemos y que acciones podemos realizar sobre el sistema teniendo en cuenta estos permisos. En cambio, el concepto de "autenticación" se refiere a que el sistema se asegure que el usuario que está queriendo iniciar sesión en un sistema sea el que dice que es y no otro, ingresando sus credenciales correspondientes y el sistema validando esto internamente, siempre utilizando para más seguridad la encriptación de éstas.
   En mi trabajo final se ven reflejados en lo que es el manejo de sesiones, donde primero autentico el inicio de sesión del usuario cuando este ingresa sus credenciales, y luego autorizo al usuario mediante el uso de familias y patentes para acceder a ciertas acciones que estos permisos le otorguen.

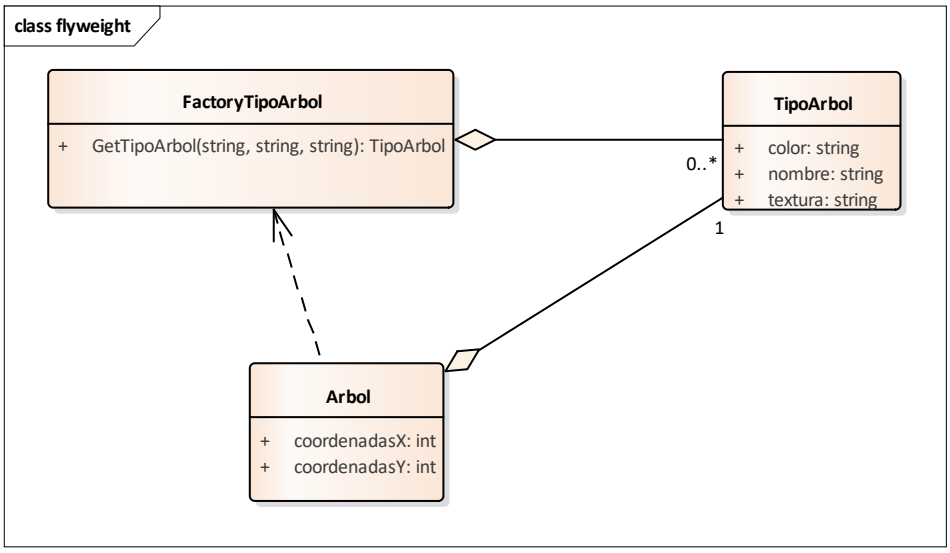
3. Enumere y describa brevemente los cuatro elementos iniciales de todo patrón.

   Rta: Los elementos iniciales de todo patrón son:
   a. **El nombre del patrón**: Nos indica el nombre del patrón, siempre representativo a lo que realiza, pudiendo intuir su función solo leyendo su nombre.
   b. **El problema que describe cuándo aplicarlo**: Nos indica el momento en el que es necesario implementar el patrón y en que problemas.
   c. **La solución que describe los elementos que constituyen el diseño**: Nos indica como el patrón resuelve el problema.
   d. **Las consecuencias o resultados que representan las ventajas e inconvenientes:** Nos indica las ventajas y desventajas de implementar el patrón, y que resultados trae su implementación.

4. Describa, diagrame y ejemplifique el patrón Flyweight (no utilice los ejemplos dados en clase).

Rta: El patrón Flyweight es un patrón estructural cuyo objetivo es compartir partes del estado de un objeto entre varios objetos para así ahorrar recursos computacionales (sobre todo en memoria) de manera importante.

Supongamos que tenemos un bosque (o sea, muchos arboles que se pueden repetir). Cada árbol será representado por su propio objeto, que tiene cierto estado (coordenadas, textura, etc.). Lo que ocurriría si no aplicamos este patrón es que muchos objetos arboles van a duplicar la misma información (nombre, textura, color). Por eso, aplicamos este patrón para almacenar estos valores dentro de objetos flyweight separados (la clase TipoArbol). Ahora, en vez de almacenar la misma información en miles de objetos Árbol, vamos a referenciar uno de los objetos flyweight con un grupo particular de valores.



5. Describa los principios SOLID. Ejemplifique.

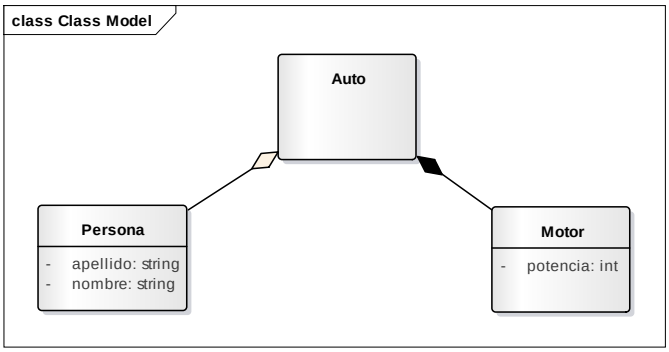
Rta: Los 5 principios SOLID de diseño de aplicaciones de software son:

- **Single Responsibility Principle:** Este principio establece que cada clase debe tener responsabilidad sobre una sola parte de la funcionalidad proporcionada por el software, y esa responsabilidad debe estar completamente encapsulada por la clase. Se centra en mantener alta la cohesión. Responde a la pregunta de qué hace y a quién conoce la clase.
  Un ejemplo puede ser un sistema de gestión de facturas, donde dentro de la clase “factura”, tengo también toda la información del cliente, por ejemplo, en vez de manejar dos clases distintas con responsabilidades bien distinguidas, una de “factura” y otra de “cliente” y una relación entre ambas.
- **Open/Closed Principle:** Este principio establece que las clases deben estar abiertas a ser extendidas y cerradas a ser modificadas, es decir, el comportamiento de una entidad debe poder ser alterado sin tener que modificar su propio código fuente. Se centra en mantener bajo el acoplamiento.

Un ejemplo puede ser una clase “Persona” donde se guardan los empleados de una organización y se quiere almacenar allí también los clientes, y se procede a modificar la clase dentro, agregando campos de clientes y cambiando las implementaciones internas que antes correspondían a los empleados y funcionaba correctamente. Este principio plantea el uso de la extensión por parte de la herencia, entonces en este caso se podría crear una clase “Cliente” que herede de la clase “Persona”, y en este caso no estaría modificando el código de la clase “Persona” y heredaría sus campos y métodos.

- **Liskov Substitution Principle:** Este patrón establece que una clase derivada puede ser reemplazada por cualquier otra que use su clase base sin alterar el correcto funcionamiento de un programa, es decir, por ejemplo, si una función espera como parámetro una clase base esta puede ser reemplazada por cualquier clase derivada. Si un subtipo hace algo que el cliente del supertipo no espera, es una violación al principio.
  Un ejemplo es una clase “Cliente” que conoce a la clase “Stack” heredando esta de la clase “List”. La clase “Cliente” va a poder acceder también al comportamiento que adopta la clase “Stack” por parte de la clase “List”, siendo esta inesperada para la clase “Cliente” considerándose así una violación al principio. Esto se resuelve con que el cliente solo conozca a la clase “Stack” y que la clase “Stack” oculte la implementación de la clase “List” para que la clase “Cliente” no lo vea.
- **Interface Segregation Principle:** Este patrón establece que los clientes no deben estar forzados a implementar interfaces que no usan. Guarda relación con la cohesión de las aplicaciones.
  Un ejemplo es una interfaz “IPájaro” que tiene como métodos definidos el de volar, y de esta interfaz heredan las clases “Loro”, “Águila” y “Pingüino”. La clase “Pingüino” va a estar obligada a implementar la interfaz, por ende, va a tener que implementar el método de volar cuando los pingüinos no pueden volar. Este caso se resolvería creando una interfaz nueva de “IPajarosNoVoladores” por ejemplo, donde hayan definidos otros métodos que la clase “Pingüino” si deba implementar, como nadar o comer.
- **Dependency Inversion Principle:** Este patrón establece que las clases de alto nivel no deberían depender de las clases de bajo nivel, ambas deben depender de las abstracciones. Al diseñar la interacción entre un módulo de alto nivel y uno de bajo nivel, la interacción debe considerarse como una interacción abstracta entre ellos.
  Un ejemplo es una clase “AlmacenadorDeDatos” que se relaciona con la clase “BaseDeDatosSQL”. En este momento tenemos una gran dependencia entre la clase “AlmacenadorDeDatos” con cómo se implementa para almacenar la información. Una solución para esto es que la clase “AlmacenadorDeDatos” se relacione con una interfaz “Datos”, y que de esta interfaz hereden otras clases concretas como puede ser “Archivo”, o mismo otro tipo de base de datos como “MongoDB”, eliminando la dependencia entre la clase “AlmacenadorDeDatos” y las clases concretas de bajo nivel.

6. Escriba (en esta hoja) el código necesario para implementar la clase Auto en base al siguiente diagrama




```
Public class Auto
{
        private Motor _motor;
        private List<Persona> _personas;

        Auto()
        {
                This._motor = new Motor();
                This._personas = new List<Persona>();
        }

}
```

Aspectos procedimentales

La empresa Saraza S.A. necesita desarrollar un sistema militar que permita, de forma sencilla, cambiar el tipo de arma que se utilizará para atacar un objetivo enemigo.

Las posibilidades, son:

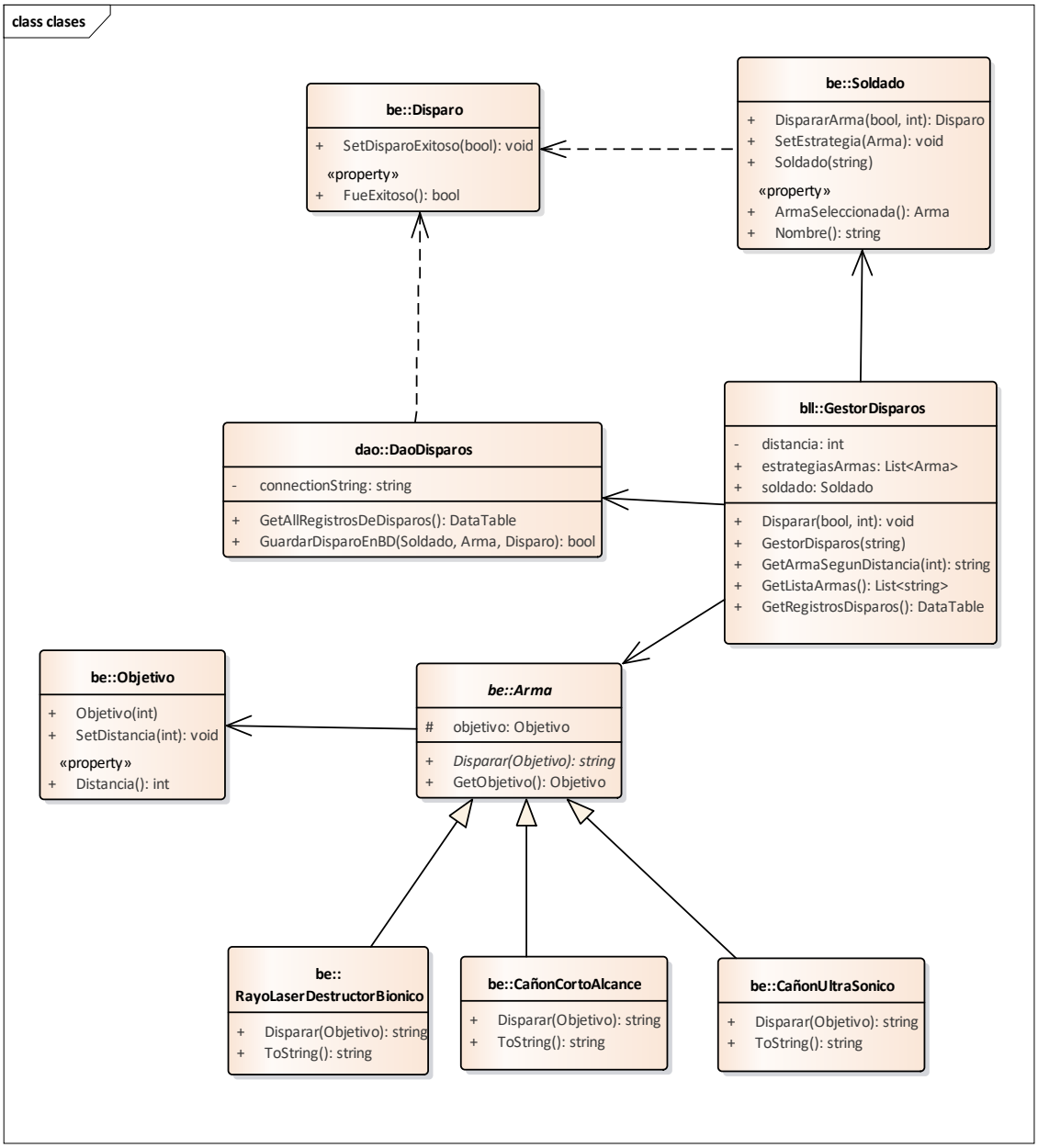
- Si el objetivo se encuentra a menos de 10k, se deberá utilizar un cañón de corto alcance
- Si el objetivo se encuentra entre 10 y 50k, el arma a utilizar será un cañón ultrasónico.
- Si el objetivo está a más de 50km y a menos de 200, se deberá usar el rayo láser destructor biónico.

El sistema cuenta con un radar que indica todos los objetivos y a que distancia se encuentran de la base, y deberá permitir cambiar dinámicamente el arma con solo presionar un botón.

Se pide diseñar y desarrollar un sistema para dar soporte a esta estructura, utilizando patrones de diseño para tal fin. Realizar diagrama de clases y código correspondiente.
Persistir en una base de datos cada uno de los disparos, distancia del objetivo, y si el disparo fue acertado o no.

Diagrama de clases aproximado, deberá cumplir lo propuesto con el patrón strategy.

IMPORTANTE: Utilizar una arquitectura de 3 o más capas. Los datos deberán estar normalizados.


class clases
be::Disparo
+ SetDisparoExitoso(bool): void
«property»
+ FueExitoso(): bool
be::Soldado
+ DispararArma(bool, int): Disparo
+ SetEstrategia(Arma): void
+ Soldado(string)
«property»
+ ArmaSeleccionada(): Arma
+ Nombre(): string
dao::DaoDisparos
- connectionString: string
+ GetAllRegistrosDeDisparos(): DataTable
+ GuardarDisparoEnBD(Soldado, Arma, Disparo): bool
bll::GestorDisparos
- distancia: int
+ estrategiasArmas: List<Arma>
+ soldado: Soldado
+ Disparar(bool, int): void
+ GestorDisparos(string)
+ GetArmaSegunDistancia(int): string
+ GetListaArmas(): List<string>
+ GetRegistrosDisparos(): DataTable
be::Objetivo
+ Objetivo(int)
+ SetDistancia(int): void
«property»
+ Distancia(): int
be::Arma
objetivo: Objetivo
+ Disparar(Objetivo): string
+ GetObjetivo(): Objetivo
be::
RayoLaserDestructorBionico
+ Disparar(Objetivo): string
+ ToString(): string
be::CañonCortoAlcance
+ Disparar(Objetivo): string
+ ToString(): string
be::CañonUltraSonico
+ Disparar(Objetivo): string
+ ToString(): string